UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSITÉ CATHOLIQUE DE LOUVAIN

Software-Defined Systems for Network-Aware Service Composition and Workflow Placement

Pradeeban Kathiravelu

Supervisor: Doctor Luís Manuel Antunes Veiga
Co-Supervisor: Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

2019

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSITÉ CATHOLIQUE DE LOUVAIN

Software-Defined Systems for Network-Aware Service Composition and Workflow Placement

Pradeeban Kathiravelu

Supervisor: Doctor Luís Manuel Antunes Veiga
Co-Supervisor: Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Doctor Joaquim Armando Pires Jorge, Instituto Superior Técnico, Universidade de Lisboa
Members of the Commitee:
Doctor Elhadj Benkhelifa, School of Computing and Digital Technologies, Staffordshire University, UK
Doctor John Javid Taheri, Faculty of Health, Science and Technology, Karlstad University, Sweden
Doctor Luís Manuel Antunes Veiga, Instituto Superior Técnico, Universidade de Lisboa
Doctor Fernando Henrique Côrte-Real Mira da Silva, Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions

European Commission
Fundação para a Ciência e a Tecnologia
INESC-ID

2019

Abstract

Composing complex workflows efficiently from diverse services on the Internet requires communication and coordination across heterogeneous execution environments, ranging from data centers and clouds to the edge managed by different infrastructure providers. Through complete virtualization of network and its services, network softwarization provides efficient management of network architecture. This dissertation exploits the flexibility and management benefits of the network softwarization to solve the problems of service composition and workflow placement at Internet scale. We present two main contributions: first, a set of extensions to network softwarization to simplify and enhance application development and deployment, and second, a scalable architecture to compose service chains in wide area networks. Finally, we evaluate these contributions in the context of big data applications. We thus intend to mitigate the challenges concerning resource management and interoperability of heterogeneous infrastructures, to efficiently compose and schedule various service workflows at Internet scale, while sharing the network and the computing resources among several users.

Network Softwarization revolutionizes the network landscape in various stages, from building, incrementally deploying, and maintaining the environment. Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) are two core tenets of network softwarization. SDN offers a logically centralized control plane by abstracting away the control of the network devices in the data plane. NFV virtualizes dedicated hardware middleboxes and deploys them on top of servers and data centers as network functions. Despite its growing application, network softwarization has not been fully exploited for effectively composing service workflows of multiple users sharing third-party network infrastructures and services. To this end, we propose our contributions to extend network softwarization for network-aware service composition and workflow placement in heterogeneous infrastructures.

First, we separate network from infrastructure by exploiting network softwarization to move out of data centers toward the edge seamlessly, and from simulations to actual deployments, with little or no additional development effort. We extend SDN in cloud and data center environments to unify various phases of development, by uniformly managing the executions of the network applications from an extended SDN controller, regardless of the execution environment and phase. We thus deploy the workloads seamlessly across the phases, from simulations and emulations to physical deployment environments. We further extend this work to support multiple Service Level Agreements (SLAs) across diverse network flows in data centers, by selectively enforcing redundancy on the network flows. Thus, we aim for Quality of Service (QoS) and efficient resource provisioning, while adhering to user policies. Finally, we design a cloud-assisted overlay network, as a latency-aware virtual connectivity provider. Consequently, we propose cost-efficient data transfers and workflow executions at Internet scale.

Second, we propose a scalable architecture to compose service chains in wide area networks efficiently. We exploit SDN and Message-Oriented Middleware (MOM) for a logically centralized composition and execution of service workflows. We thus propose a Software-Defined Service Composition (SDSC) framework for web service compositions, Network Service Chains (NSCs), and a network-aware execution of data services. We further present Software-Defined Systems (SDS) consisting of virtual network allocation strategies for multi-tenant service executions in large-scale networks comprised of multiple domains.

Finally, we investigate how our proposed SDS can operate efficiently for real-world application scenarios of heterogeneous infrastructures. While traditionally web services are built following standards and best practices such as Web Services Description Language (WSDL), network services and data services offered by different service providers often fall short in providing common Application Programming Interfaces (APIs), thus resulting in vendor lock-in. We look into facilitating interoperability across service implementations and deployments, to enable seamless workflow executions and service migrations. We propose big data applications and smart environments such as Cyber-Physical Systems (CPS) and the Internet of Things (IoT) as our two application scenarios. We thus build CPS and big data applications as composable service chains, offering them an interoperable execution.

Our research contributions highlight that network softwarization can be used to build and deploy network applications with minimal repetitive effort, from initial design and development stages to production. Evaluations on the proposed SDS demonstrate performance and economic benefits to service composition and workflow placement at various scales, from data centers to the Internet. By managing and leveraging redundancy in the network flows and network paths, our SDS prototypes ensure that SLAs are met in the critical network flows of multi-tenant systems. Furthermore, our SDS framework reduces Internet latency by up to 30%, yet in an economic approach. Finally, we elaborate the broader applicability of our proposed SDS by extending it to CPS and big data applications.

Keywords — Network Softwarization, Software-Defined Networking (SDN), Software-Defined Systems (SDS), Network Functions Virtualization (NFV), Service-Oriented Architecture (SOA).

Resumo

Compor e escalonar fluxos de trabalho na escala da Internet requer a comunicação e coordenação entre vários serviços em ambientes de execução heterogéneos - de centros de dados e nuvens computacionais aos ambientes de borda operados por vários provedores de infraestrutura. A softwarização em redes permite a gestão mais eficiente do sistema, afinando o seu controlo e melhorando a capacidade de reutilização dos serviços de rede, por meio de uma virtualização completa da rede e seus serviços. Esta dissertação explora os benefícios de flexibilidade e gerenciamento do softwarização em redes para resolver os problemas de composição de serviço e posicionamento de fluxo de trabalho na escala da Internet. Nós apresentamos duas contribuições principais: primeiro, um conjunto de extensões para softwarização em redes para simplificar e aprimorar o desenvolvimento e a implantação de aplicativos e, segundo, uma arquitetura escalonável para compor cadeias de serviço em redes de longa distância. Por fim, avaliamos essas contribuições no contexto de aplicativos de big data. Assim, nós pretendemos atenuar os desafios relacionados ao gerenciamento de recursos e à interoperabilidade de infraestruturas heterogêneas para compor e agendar com eficiência vários fluxos de trabalho de serviços em escala de Internet ao mesmo tempo em que compartilhamos a rede e os recursos de computação entre vários usuários.

A softwarização em redes revoluciona o cenário da rede em vários aspectos, incluindo sua criação, implantação incremental e manutenção do ambiente. Redes definidas por software (SDN) e virtualização de funções de rede (NFV) são dois dos princípios centrais da softwarização em redes. SDN oferece um plano de controlo logicamente centralizado, abstraindo o controle dos dispositivos de rede no plano de dados. NFV virtualiza middleboxes dedicados e os implementa em servidores e centros de dados como funções de rede. Apesar de sua crescente aplicação, a softwarização em redes não foi explorada para a composição do serviço e a execução do fluxo de trabalho de vários usuários que consomem infraestruturas e serviços de rede de terceiros. Para este fim, nós propomos três contribuições principais, ampliando e aproveitando a softwarização em redes para fluxos de trabalho de composição de serviços em infraestruturas heterogêneas.

Primeiro, nós estendemos SDN em ambientes de nuvem e centros de dados para unificar várias fases de desenvolvimento e implantar as cargas de trabalho de forma transparente, de simulações e emulações, a ambientes físicos de implantação. Além disso, nós estendemos este trabalho para oferecer suporte a vários acordos de nível de serviço (SLAs) em diversos fluxos de rede nos centros de dados, aplicando seletivamente a redundância nos fluxos de rede. Assim, nós visamos a Qualidade de Serviço (QoS) e o provisionamento eficiente de recursos, ao mesmo tempo em que aderimos às políticas dos utilizadores. Por fim, nós projetamos uma rede de sobreposta assistida por nuvem, como um provedor de conectividade virtual com ciente da latência. Consequentemente, nós propomos transferências de dados económicas na escala da

Internet, separando a rede da infraestrutura.

Em segundo lugar, nós propomos uma arquitetura escalonável para compor cadeias de serviço em rede de longa distância com eficiência. Nós estendemos SDN e Middleware Orientado a Mensagens (MOM) para composição e execução de fluxo de trabalho de serviços logicamente centralizados. Assim, nós propomos uma estrutura Composição de Serviços Definidos por Software (SDSC) para composições de serviço web, cadeia de serviços de rede (NSCs) e uma execução de serviços de dados que cientes da rede. Além disso, nós apresentamos Sistemas Definidos por Software (SDS), que consistem em estratégias de alocação de rede virtual para execuções de serviços multilocatários em redes de grande escala, compostas de vários domínios.

Em terceiro lugar, nós investigamos como a nossa arquitetura SDS pode operar com eficiência para cenários de aplicações reais de infraestruturas heterogêneas. Embora tradicionalmente os serviços na web sejam criados seguindo padrões e práticas recomendadas, como WSDL (Web Services Description Language), serviços de rede e serviços de dados oferecidos por diferentes provedores de serviços geralmente ficam aquém do fornecimento de interoperável interfaces de programação de aplicativos (APIs), geralmente resultando em bloqueio do fornecedor. Nós procuramos facilitar a interoperabilidade entre implementações e implantações de serviços para permitir migrações contínuas. Nós propomos arcabouços para grandes volumes de dados e ambientes inteligentes, como sistemas ciber-físicos (CPS) e Internet das coisas (IoT), como nossos dois cenários de aplicativos. Assim, nós construímos CPS e aplicativos de big data como cadeias de serviços compostos, oferecendo-lhes uma execução interoperável.

Nossas contribuições à pesquisa destacam que a softwarização em redes pode ser usada para construir e implantar aplicativos de rede com esforço repetitivo mínimo, desde os estágios iniciais de projeto e desenvolvimento até a produção. As avaliações sobre a SDS proposta demonstram benefícios econômicos e de desempenho para a composição do serviço e o posicionamento do fluxo de trabalho em várias escalas, dos data centers à Internet. Ao gerenciar e aproveitar a redundância nos fluxos de rede e caminhos de rede, nossos protótipos SDS garantem que os SLAs sejam atendidos nos fluxos de rede críticos dos sistemas de multilocatário. Além disso, nossa estrutura de SDS reduz a latência da Internet em até 30%, ainda que em uma abordagem econômica. Por fim, nós elaboramos a aplicabilidade mais ampla do nosso SDS proposto estendendo-o ao CPS e aos aplicativos de big data.

Palavras Chave — Softwarização em Redes, Redes Definida por Software (SDN), Sistemas Definidos por Software (SDS), Virtualização de Funções de Rede (NFV), Arquitetura Orientada a Serviços (SOA).

Résumé

La composition efficace de flux de travail complexes à partir de divers services sur Internet nécessite une communication et une coordination dans des environnements d'exécution hétérogènes, allant des centres de données et des clouds à la périphérie, gérés par différents fournisseurs d'infrastructure. Grâce à la virtualisation complète du réseau et de ses services, la logiciellisation du réseau permet une gestion efficace de l'architecture du réseau. Cette thèse exploite les avantages de flexibilité et de gestion de la logiciellisation du réseau pour résoudre les problèmes de composition de services et de placement de flux de travail à l'échelle Internet. Nous présentons deux contributions principales: d'une part, un ensemble d'extensions de la logiciellisation du réseau pour simplifier et améliorer le développement et le déploiement d'applications, et, d'autre part, une architecture évolutive permettant de composer des chaînes de services dans des réseaux étendus. Enfin, nous évaluons ces contributions dans le contexte des applications mégadonnées. Nous mitigeons ainsi les problèmes de gestion des ressources et d'interopérabilité d'infrastructures hétérogènes afin de composer et de planifier efficacement divers flux de travail de service à l'échelle Internet, tout en partageant le réseau et les ressources informatiques entre plusieurs utilisateurs.

La technologie de logiciellisation du réseau révolutionne le paysage réseau à différentes étapes, de la création au déploiement progressif, en passant par la maintenance de l'environnement. Le réseau défini par logiciel (SDN) et la virtualisation des fonctions de réseau (NFV) sont deux principes fondamentaux de la gestion de réseau. SDN offre un plan de contrôle centralisé de manière logique en soustrayant le contrôle des périphériques réseau dans le plan de données. NFV virtualise le matériel intergiciel dédié et le déploie en tant que fonctions réseau sur des serveurs et des centres de données. En dépit de son adoption, la logiciellisation du réseau n'a pas été pleinement exploitée pour la composition efficace des flux de travail de service de plusieurs utilisateurs partageant des infrastructures et des services réseau tiers. À cette fin, nous proposons nos contributions pour étendre le logiciellisation du réseau pour la composition de services réseau et le placement de flux de travail dans des infrastructures hétérogènes.

Premièrement, nous séparons le réseau de l'infrastructure en exploitant la logiciellisation du réseau pour passer des centres de données vers le bord de manière transparente, et des simulations aux déploiements réels, avec peu ou pas d'effort de développement supplémentaire. Nous étendons SDN dans les environnements de cloud et de centres de données pour unifier différentes phases de développement, en gérant de manière uniforme les exécutions des applications réseau à partir d'un contrôleur SDN étendu, quels que soient l'environnement et la phase d'exécution. Nous déployons ainsi les charges de travail de manière transparente au cours des phases, des simulations et émulations aux environnements de déploiement physiques. Nous étendons encore ce travail pour prendre en charge plusieurs accords de niveau de service (SLA) sur différents flux

de réseau dans des centres de données, en appliquant de manière sélective la redondance sur les flux de réseau. Ainsi, nous visons une qualité de service (QoS) et un provisionnement efficace des ressources, tout en respectant les politiques de l'utilisateur. Enfin, nous concevons un réseau de superposition assisté par le cloud, en tant que fournisseur de connectivité virtuelle prenant en compte le temps de latence. En conséquence, nous proposons des transferts de données et des exécutions de flux de travail rentables à l'échelle Internet.

Deuxièmement, nous proposons une architecture évolutive pour composer efficacement les chaînes de services dans les réseaux étendus. Nous exploitons SDN et intergiciel à messages (MOM) pour une composition et une exécution logiquement centralisées des flux de travail de service. Nous proposons donc un cadre de composition de service défini par logiciel (SDSC) pour les compositions de services Web, des chaînes de services réseau (NSC) et une exécution des services de données adaptée au réseau. Nous présentons en outre des systèmes définis par logiciel (SDS) constitués de stratégies d'allocation de réseau virtuel pour les exécutions de services à locataires multiples dans des réseaux à grande échelle comprenant plusieurs domaines.

Enfin, nous étudions comment notre SDS proposée peut fonctionner efficacement pour des scénarios d'applications réelles d'infrastructures hétérogènes. Alors que les services Web sont généralement construits conformément aux normes et aux meilleures pratiques telles que le langage WSDL (Web Services Description Language), les services réseau et les services de données proposés par différents fournisseurs de services ne permettent pas toujours de fournir des interfaces de programmation d'application (API) communes, ce qui a pour effet de verrouiller les fournisseurs. Nous cherchons à faciliter l'interopérabilité entre les implémentations et les déploiements de services, afin de permettre des exécutions et des migrations de services transparentes. Nous proposons des applications mégadonnées et des environnements intelligents tels que système cyber-physique (CPS) et l'Internet des objets (IoT) comme nos deux scénarios d'application. Nous construisons ainsi des applications CPS et mégadonnées sous forme de chaînes de services composables, en leur offrant une exécution interopérable.

Nos contributions de recherche soulignent que le logiciellisation du réseau peut être utilisée pour créer et déployer des applications réseau avec un minimum d'efforts répétitifs, des étapes de conception et de développement initiales à la production. Les évaluations des SDS proposés démontrent les performances et les avantages économiques liés au placement de services et de flux de travail à différentes échelles, des centres de données à Internet. En gérant et en exploitant la redondance des flux et des chemins réseau, nos prototypes SDS garantissent que les accords de niveau de service sont établis dans les flux réseau critiques des systèmes multi-locataires. De plus, notre cadre SDS réduit la latence d'Internet jusqu'à 30%, tout en restant économique. Enfin, nous développons l'applicabilité plus large de la SDD en l'étendant aux applications CPS et mégadonnées.

Mots Clés — Logiciellisation du réseau, Réseau défini par logiciel (SDN), Systèmes définis par logiciel (SDS), Virtualisation des fonctions réseau (NFV), Architecture orientée services (SOA).

List of Publications

Thesis Contributions

The work and results presented in this dissertation are partially described in the following publications.

Journals:

(J1) **Kathiravelu, P.**, Van Roy, P., & Veiga, L. *Composing Network Service Chains at the Edge: A Resilient and Adaptive Software-Defined Approach.* In Transactions on Emerging Telecommunications Technologies (ETT). **(JCR IF: 1.535, Q2)**. pp. 1 – 22. Aug. 2018. Wiley. `https://doi.org/10.1002/ett.3489`

(J2) **Kathiravelu, P.**, Van Roy, P., & Veiga, L. SD-CPS*: Software-Defined Cyber-Physical Systems. Taming the Challenges of CPS with Workflows at the Edge.* In Cluster Computing – The Journal of Networks Software Tools and Applications. **(JCR IF: 2.040, Q2)**. pp. 1 – 17. Nov. 2018. Springer. `https://link.springer.com/article/10.1007%2Fs10586-018-2874-8`

(J3) **Kathiravelu, P.**, Sharma, A., Galhardas, H., Van Roy, P., & Veiga, L. *On-Demand Big Data Integration: A Hybrid ETL Approach for Reproducible Scientific Research.* In Distributed and Parallel Databases (DAPD). **(JCR IF: 1.179, Q2)**. pp. 273 – 295. May 2019. Springer. `https://doi.org/10.1007/s10619-018-7248-y`

(J4) **Kathiravelu, P.**, Van Roy, P., & Veiga, L. *Interoperable and Network-Aware Service Workflows for Big Data Executions at Internet Scale.* In Concurrency and Computation: Practice and Experience (CCPE). **(JCR IF: 1.114, Q2)**. pp. 1 – 18. Feb. 2019. Wiley. `https://doi.org/10.1002/cpe.5212`.

Book Chapters:

(B1) **Kathiravelu, P.** & Veiga, L. *SDN helps other Vs in Big Data.* Chapter of Big Data and Software Defined Networks. pp. 253 – 273. Mar. 2018. IET. ISBN: 978-1-78561-304-3. `https://www.theiet.org/resources/books/computing/bigdata.cfm`

(B2) Cardellini, V., Grbac, T.G., Kassler, A., **Kathiravelu, P.**, Lo Presti, F., Marotta, A., Nardelli, M. & Veiga, L. *Integrating SDN and NFV with QoS-aware service composition.* Chapter of Autonomous Control for a Reliable Internet of Services: Methods, Models, Approaches, Techniques, Algorithms, and Tools. pp. 212 – 240. May 2018. Springer. `https://link.springer.com/chapter/10.1007/978-3-319-90415-3_9`

Conferences:

(C1) **Kathiravelu, P.**, Chiesa, M., Marcos, P., Canini, M., Veiga, L. *Moving Bits with a Fleet of Shared Virtual Routers.* In Networking 2018. **(CORE Rank A)**. pp. 370 – 378. May 2018 (Acceptance Rate: 24%). IFIP.

(C2) **Kathiravelu, P.**, Grbac, T.G., & Veiga, L. *Building Blocks of Mayan: Componentizing the eScience Workflows Through Software-Defined Service Composition.* In 23rd International Conference on Web Services (ICWS 2016) **(Awarded a travel grant)**. **(CORE Rank A)**. pp. 372 – 379. June 2016. IEEE. `https://doi.org/10.1109/ICWS.2016.55`

(C3) **Kathiravelu, P.** & Veiga, L. *Software-Defined Simulations for Continuous Development of Cloud and Data Center Networks.* In 24th International Conference on Cooperative Information Systems (CoopIS 2016). **(CORE Rank A)**. On the Move to Meaningful Internet Systems: OTM 2016 Conferences, pp. 3 – 23. Oct. 2016. Springer. `https://dx.doi.org/10.1007/978-3-319-48472-3_1`

(C4) **Kathiravelu, P.**, Galhardas, H., & Veiga, L. *∂u∂u Multi-Tenanted Framework: Distributed Near Duplicate Detection for Big Data.* In 23rd International Conference on Cooperative Information Systems (CoopIS 2015). **(CORE Rank A)**. On the Move to Meaningful Internet Systems: OTM 2015 Conferences, pp. 237 – 256. Oct. 2015. Springer. `https://doi.org/10.1007/978-3-319-26148-5_14`

(C5) **Kathiravelu, P.**, Van Roy, P., & Veiga, L. *Software-Defined Data Services: Interoperable and Network-Aware Big Data Executions.* In The Fifth International Conference on Software Defined Systems (SDS 2018). **Best Paper Award**. pp. 145 – 152. Apr. 2018. IEEE. `https://doi.org/10.1109/SDS.2018.8370436`

(C6) **Kathiravelu, P.** & Veiga, L. *SD-CPS: Taming the Challenges of Cyber-Physical Systems with a Software-Defined Approach.* In The Fourth International Conference on Software Defined Systems (SDS 2017). pp. 6 – 13. May 2017. IEEE. `https://doi.org/10.1109/SDS.2017.7939133`

Symposia and Workshops:

(W1) **Kathiravelu, P.** & Veiga, L. *CHIEF: Controller Farm for Clouds of Software-Defined Community Networks.* In 3rd International Symposium on Software Defined Systems (SDS 2016). pp. 1 – 6. Apr. 2016. IEEE. `https://doi.org/10.1109/IC2EW.2016.8`

(W2) **Kathiravelu, P.**, Chen, Y., Sharma, A., Galhardas, H., Van Roy, P., & Veiga, L. *On-Demand Service-Based Big Data Integration: Optimized for Research Collaboration.* In Third International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2017), co-located with 43rd International Conference on Very Large Data Bases (VLDB 2017). pp. 9 – 28. Sep. 2017. LNCS. `https://link.springer.com/chapter/10.1007/978-3-319-67186-4_2`

(W3) **Kathiravelu, P.** & Veiga, L. *Selective Redundancy in Network-as-a-Service: Differentiated QoS in Multi-tenant Clouds.* In On the Move to Meaningful Internet Systems: OTM 2016 Workshops. On the Move to Meaningful Internet Systems. pp. 87 – 97. Oct. 2016. Springer, Cham. https://link.springer.com/chapter/10.1007/978-3-319-55961-2_9

(W4) **Kathiravelu, P.** & Sharma, A. *A Dynamic Data Warehousing Platform for Creating and Accessing Biomedical Data Lakes.* In Second International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2016), co-located with 42nd International Conference on Very Large Data Bases (VLDB 2016). pp. 101 – 120. Sep. 2016. LNCS. https://link.springer.com/chapter/10.1007/978-3-319-57741-8_7

(W5) **Kathiravelu, P.** *Software-Defined Networking-Based Enhancements to Data Quality and QoS in Multi-Tenanted Data Center Clouds.* In International Conference on Cloud Engineering (IC2E 2016) Doctoral Symposium **(Awarded a travel grant)**. pp. 201 – 203. Apr. 2016. IEEE. https://doi.org/10.1109/IC2EW.2016.19

(W6) **Kathiravelu, P.**, Sharifi, L., & Veiga, L. *Cassowary: Middleware Platform for Context-Aware Smart Buildings with Software-Defined Sensor Networks.* In 2nd Workshop on Middleware for Context-Aware Applications in the IoT (M4IOT 2015), co-located with ACM/USENIX/IFIP Middleware 2015. pp. 1 – 6. Dec. 2015. ACM. https://doi.org/10.1145/2836127.2836132

(W7) **Kathiravelu, P.** & Veiga, L. *An Expressive Simulator for Dynamic Network Flows.* In 2nd IEEE International Workshop on Software Defined Systems (SDS 2015) in conjunction with the IEEE International Conference on Cloud Engineering (IC2E 2015). pp. 311 – 316. Mar. 2015. IEEE. https://doi.org/10.1109/IC2E.2015.43

Short Papers:

(S1) **Kathiravelu, P.** & Veiga, L. *SDN Middlebox Architecture for Resilient Transfers.* In 15th International Symposium on Integrated Network Management (IM 2017). **(CORE Rank A)**. pp. 560 – 563. May 2017. IFIP/IEEE. https://doi.org/10.23919/INM.2017.7987329

(S2) **Kathiravelu, P.** & Veiga, L. *SENDIM for Incremental Development of Cloud Networks: Simulation, Emulation & Deployment Integration Middleware.* In International Conference on Cloud Engineering (IC2E 2016). pp. 143 – 146. Apr. 2016. IEEE. https://doi.org/10.1109/IC2E.2016.22

Other Contributions

Books:

The following books partly present the network softwarization frameworks exploited in this dissertation from a software engineering point of view with sample codes.

(A1) **Kathiravelu, P.** & Sarker, M.O.F. *Python Network Programming Cookbook, Second Edition*. ISBN: 978-1-78646-399-9. Aug. 2017. Packt. `https://www.packtpub.com/networking-and-servers/python-network-programming-cookbook-second-edition`.

(A2) Ratan, A., Chou, E., **Kathiravelu, P.**, & Sarker, M.O.F. *Python Network Programming: Conquer all your networking challenges with the powerful Python language*. ISBN: 978-1788835466. Jan. 2019. Packt. `https://www.amazon.com/Python-Network-Programming-networking-challenges/dp/1788835468`.

The following publications present the research collaborations partly resulted from this dissertation.

Conferences:

(C7) Marcos, P., Chiesa, M., Muller, L., **Kathiravelu, P.**, Dietzel, C., Canini, M., & Barcellos, M. *Dynam-IX: a Dynamic Interconnection eXchange*. In The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2018). **(CORE Rank A)**. Dec. 2018. ACM (Acceptance Rate: 17.2%).

(C8) Caixinha, D., **Kathiravelu, P.** & Veiga, L. *ViTeNA: An SDN-Based Virtual Network Embedding Algorithm for Multi-Tenant Data Centers*. In 15th International Symposium on Network Computing and Applications (NCA 2016). **(CORE Rank A)**. pp. 140 – 147. Oct. 2016. IEEE. `https://doi.org/10.1109/NCA.2016.7778608`

Posters and Extended Abstracts:

(P1) Marcos, P., Chiesa, M., Muller, L., **Kathiravelu, P.**, Dietzel, C., Canini, M., & Barcellos, M. *Dynam-IX: A Dynamic Interconnection eXchange*. In SIGCOMM 2018 **(CORE Rank A*)**. Aug. 2018. ACM.

(P2) **Kathiravelu, P.** & Sharma, A. *SPREAD – System for Sharing and Publishing Research Data*. In Society for Imaging Informatics in Medicine Annual Meeting (SIIM 2016). June 2016. `https://c.ymcdn.com/sites/siim.org/resource/resmgr/siim2016abstracts/Research_Kathiravelu.pdf`

(P3) **Kathiravelu, P.** & Sharma, A. *Near Duplicate Detection for Medical Data Warehouse Construction*. In AMIA 2016 Joint Summits on Translational Science. Mar. 2016.

(P4) **Kathiravelu, P.**, Kazerouni, A., & Sharma, A. *Data Café – A Platform For Creating Biomedical Data Lakes*. In AMIA 2016 Joint Summits on Translational Science. Mar. 2016.

National Conferences and Workshops:

(N1) **Kathiravelu, P.**, Grbac, T.G, & Veiga, L. *A FIRM Approach to Software-Defined Service Composition*. In MIPRO 2016 - 39th International Convention on Telecommunications & Information (CTI). pp. 634 – 639. May 2016. `https://doi.org/10.1109/MIPRO.2016.7522206`

(N2) **Kathiravelu, P.** & Sharma, A. *MEDIator: A Data Sharing Synchronization Platform for Heterogeneous Medical Image Archives*. In Workshop on Connected Health at Big Data Era (BigCHat'15), co-located with 21st SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2015). Aug. 2015. ACM. `https://doi.org/10.13140/RG.2.1.3709.4248`

Acknowledgments

My Ph.D. life was an intercontinental one, spanning the globe. I express my gratitude to several individuals who contributed to the timely completion of my doctoral dissertation. I attribute the success of my research work to the expertise and leadership of my Ph.D. supervisors. I like to share my sincere gratitude with my supervisor, Prof. Luís Manuel Antunes Veiga (IST), for his continuous support and guidance throughout my MSc and Ph.D. His enthusiasm and optimism have been a significant motivation for me during my six years at IST. I like to thank my co-supervisor, Prof. Peter Van Roy (UCLouvain) for his guidance. His vision and blue hat thinking were fundamental in strengthening my thesis and pointing me in the right direction.

I would like to thank Prof. Ashish Sharma (Emory) for being my mentor and sharing his immense knowledge of distributed systems and their applications for biomedical big data research. It has always been a pleasure to work with him at Emory and learn several aspects of research from his expertise. I am thankful to Prof. Helena Galhardas (IST) – Every time I had a chance to work with her, she always gave her full support and ensured that she shared her knowledge promptly. I thank Prof. Marco Canini (KAUST) for sharing his wisdom in network softwarization and hosting me at KAUST. I like to thank Prof. Tihana Galinac Grbac (URijeka) for hosting me in Rijeka graciously and offering me exclusive access to an SDN research lab throughout my stay. Our fruitful discussions helped me shape my early Ph.D. research.

I extend my thanks to Prof. Marco Chiesa (KTH), Ed Warnicke (Cisco), Prof. Olivier Bonaventure (UCLouvain), and Prof. Etienne Riviere (UCLouvain) for their support. I also recall the continuous guidance I received as an undergraduate from Vishaka Nanayakkara (Uo-Moratuwa), who encouraged and motivated me to pursue higher studies.

I thank the Software-Defined Systems (SDS) community and the Services Society for their feedback and support on my research. I also appreciate the industry for their help - specifically, RIPE NCC for offering me an Atlas Probe, Amazon Web Services (AWS) and Voxility for their insights, and Packt for publishing my books on network programming. An Erasmus Mundus experience will not be possible without the ground support of the international students' office of the participating universities. I would like to thank NMCI at IST – specifically Ana Barbosa, for being there always for the students. I also thank Paula Barrancos (INESC-ID) and ICTEAM administrative staff: Vanessa Maons (UCLouvain) and Sophie Renard (UCLouvain) for their continuous assistance.

Thanks to European Master in Distributed Computing (EMDC) and Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) double-degree programs, I met many wonderful people in Lisboa and throughout Europe. I am thankful for all those vivid memories. I also made friends with several colleagues during my stay at IST, UCLouvain, Emory, KAUST,

and URijeka, with whom I shared many productive discussions and exciting moments. I thank every one of them, especially, Dr. Sergio Esteves (IST), Xiao Chen (Dell EMC), Daniel Porto (IST), Miguel Coimbra (IST), Prof. Leila Sharifi (Urmia), Pedro Marcos (UFRGS), Dr. Mennan Selimi (Cambridge), Dr. Denis Weerasiri (Amazon), Saminda Wijeratne (Georgia Tech), and Dr. Richard Gil Martinez (Elastic).

I thank my family for their encouragement and emotional support: my mother, Selvathie Kathiravelu, for her kindness and motivation, and the memories of my father, Kanapathipillai Kathiravelu. I finally thank my wife Gu Juejing for her love, care, and sacrifices. Her enthusiasm and courage made this journey a beautiful one.

* * *

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030. This work was also partially funded by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 and PTDC/EEI-SCR/6945/2014, the LightKone H2020 project under Grant Agreement 732505 from the European Commission, the ENDEAVOUR H2020 project under Grant Agreement 644960 from the European Commission, and the COST action 1304 Autonomous Control for a Reliable Internet of Services (ACROSS). This work was also supported by the funds from KAUST, Emory University, Google Summer of Code (GSoC) 2014 - 2016, and the Linux Foundation's OpenDaylight Project.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

Acronym	Description	Page
5GEx	5^{th} Generation Exchange	30
AAA	Authentication, Authorization, and Accounting	20
ABA	Affinity-based Approach	28
AD-SAL	API-Driven Service Abstraction Layer	20
AMI	Amazon Machine Image	61
AMQP	Advanced Message Queuing Protocol	26
API	Application Programming Interface	5
AWS	Amazon Web Services	23
BGP	Border Gateway Protocol	62
BOS	Building Operating System	29
CapEx	Capital Expenditures	4
CDM	Copy Data Management	31
CDN	Content Delivery Network	3
CG	Column Generation	28
CLI	Command Line Interface	61
CPE	Customer Premises Equipment	28
CPS	Cyber-Physical System	2
CRUD	Create, Retrieve, Update, and Delete	151
DaaS	Data-as-a-Service	30
DC	Data Center	2
DDoS	Distributed Denial of Service	63
DICOM	Digital Imaging and Communications in Medicine	142
DO	Domain Orchestrator	30
DOM	Document Object Model	46
DSL	Domain Specific Language	42
ebXML	Electronic Business XML	32
EC2	Elastic Compute Cloud	57

Acronym	Description	Page
ECMP	Equal-Cost Multi-Path	81
ESB	Enterprise Service Bus	32
ETL	Extract, Transform, and Load	16
FHIR	Fast Healthcare Interoperability Resources	32
ForCES	Forwarding and Control Element Separation	18
FTTH	Fiber to the home	72
GbE	Gigabit Ethernet	55
GCP	Google Cloud Platform	23
HDFS	Hadoop Distributed File System	100
HL7	Health Level Seven International	32
I2V	Infrastructure-to-Vehicle	135
IDS	Intrusion Detection System	80
IIoT	Industrial Internet of Things	30
ILP	Integer Linear Programming	28
IMDG	In-Memory Data Grid	40
IoT	Internet of Things	3
ISP	Internet Service Provider	9
IXP	Internet eXchange Point	34
JDBC	Java Database Connectivity	152
LSO	Lifecycle Service Orchestration	28
MANET	Mobile Ad-hoc Network	29
MAS	Multi-Agent Systems	29
MD-SAL	Model-Driven Service Abstraction Layer	20
MdO	Multi-Domain Orchestrator	30
MEC	Mobile and Edge Computing	27
MILP	Mixed Integer Linear Programming	28
MINA	Multinetwork INformation Architecture	30
MIP	Mixed Integer Programming	33
MOM	Message-Oriented Middleware	7
MPI	Message Passing Interface	21
MPLS	Multiprotocol Label Switching	9
MPTCP	Multipath Transmission Control Protocol	25
MQTT	Message Queuing Telemetry Transport	27
NaaS	Network-as-a-Service	14

Acronym	Description	Page
NAT	Network Address Translation	83
NETCONF	Network Configuration Protocol	45
NFV	Network Functions Virtualization	3
NGSON	Next Generation Service Overlay Network	27
NSC	Network Service Chain	15
NSO	Network Service Orchestration	28
OGSA-DAI	Open Grid Services Architecture - Data Access and Integration	33
OLIA	Opportunistic Linked Increases Algorithm	25
ONOS	Open Network Operating System	19
OO	Overarching Orchestrator	30
OpEx	Operational Expenditures	4
PDQ	Preemptive Distributed Quick	24
PE	Provider Edge	28
PoP	Point of Presence	55
QoE	Quality of Experience	10
QoS	Quality of Service	3
REST	Representational State Transfer	5
ROA	Resource-Oriented Architecture	26
RPC	Remote Procedure Call	20
RTT	Round-Trip Time	70
S3	Simple Storage Service	59
SaaS	Software-as-a-Service	23
SAL	Service Abstraction Layer	20
SD-CPS	Software-Defined Cyber-Physical Systems	15
SD-WAN	Software-Defined Wide Area Network	4
SDB	Software-Defined Building	29
SDCD	Software-Defined Cloud Deployment	14
SDDC	Software-Defined Data Center	4
SDDS	Software-Defined Data Services	16
SDE	Software-Defined Environment	29
SDIA	Software-Defined Internet Architecture	22
SDIIoT	Software Defined Industrial Internet of Things	30
SDIoT	Software Defined Internet of Things	30
SDN	Software-Defined Networking	3
SDS	Software-Defined System	2

Acronym	Description	Page
SDSC	Software-Defined Service Composition	15
SEED	Standard for the Exchange of Earthquake Data	142
SFC	Service Function Chaining	27
SLA	Service Level Agreement	6
SLO	Service Level Objective	6
SOA	Service-Oriented Architecture	5
SSD	Solid-State Drive	19
STOMP	Simple / Streaming Text Oriented Message Protocol	27
TCIA	The Cancer Imaging Archive	103
TCP	Transmission Control Protocol	25
TOSCA	OASIS Topology and Orchestration Specification for Cloud Applications	124
UDDI	Universal Description, Discovery, and Integration	26
URI	Uniform Resource Identifier	95
V2I	Vehicle-to-Infrastructure	135
V2V	Vehicle-to-Vehicle	135
VANET	Vehicular Ad-hoc Network	29
VCE	Vienna Cloud Environment	33
VM	Virtual Machine	8
VNE	Virtual Network Embedding	28
VNF	Virtual Network Function	3
VNFaaS	VNF-as-a-Service	28
WADL	Web Application Description Language	5
WAN	Wide Area Network	3
WSDL	Web Services Description Language	5
XML	Extensible Markup Language	42
XMPP	Extensible Messaging and Presence Protocol	27
YAML	YAML Ain't Markup Language	100
YANG	Yet Another Next Generation	20

I Thesis Overview

1 Introduction

Composing user workflows seamlessly from heterogeneous third-party services is a challenging problem due to the volume of service providers and the diversity of services [182]. Research and enterprises have proposed *Network softwarization* [4, 346] and *Software-Defined Systems (SDS)* [92, 85, 258], to enhance the interoperability and flexibility of large-scale networks and systems. Network softwarization enables complete virtualization of the network, to facilitate dynamic formation, efficient configuration, incremental deployment, and seamless migration of network architectures through software constructs [224]. SDS refers to a wide range of frameworks that adopt or extend network softwarization for heterogeneous systems, where an execution environment is separated into i) a data plane consisting of devices and ii) a centralized control plane that centrally manages actions and policies on the data plane devices in a unified manner [164]. Regardless of these promising developments, existing SDS and network softwarization frameworks do not adequately support service composition and workflow placement in multi-domain networks consisting of multiple tenants. This dissertation proposes SDS frameworks for network-aware service composition and workflow placement at Internet scale. We first extend network softwarization to facilitate seamless deployment and migration of services and scaling them across diverse network environments. We then present an SDS architecture and optimization algorithms for efficient service composition and workflow placement across heterogeneous execution environments in wide area networks. Finally, we propose network-aware execution of big data applications and Cyber-Physical Systems (CPS), exploiting our SDS frameworks and contributions to network softwarization.

1.1 Context

Service composition and workflow placement at Internet scale should extend and exploit the network management capabilities for efficient resource sharing across several user workflows that consume the services. Services are getting pervasive on the Internet, with several third-party network, cloud, and service providers offering resources to the end users. A sophisticated user workflow often invokes several services from multiple providers [3]. However, execution environments managed by third-party providers lack interoperability among them, thus preventing the users from exploiting resources and services from various providers to compose their workflows. While network softwarization has made promising improvements to network management [127], it limits its focus mainly to data center (DC) networks. Currently, the potential for extending network softwarization for service composition and workflow placement at Internet scale remains mostly unexplored. To reap the benefits of network softwarization for Internet services, we need a "bridge" between the network management capabilities of network softwarization, and the

service composition and workflow scheduling handled at the application level by the service providers.

The prevalent demand for high data rate and low latency of the Internet applications has driven more infrastructure and service providers to distribute their resources closer to the end users [52]. Latency-sensitive Internet applications [305] such as high-frequency trading [189], online gaming [78], remote surgery [26], eScience workflows [319], and the Internet of Things (IoT) [95, 365, 334] have a high demand for a quick response. These Internet applications are reaching geographically diverse locations, far from the tier-1 cities that typically host cloud data centers. With the need for low latency [7], these Internet applications perform better when they are deployed and served from the edge [305], compared to cloud regions that are typically farther to the users than the edge providers. The demand for a locality-aware execution is met with an increasing number of edge providers to serve the large and geographically-distributed user base [335]. Subsequently, cloud providers are also opening up more regions [366] to offer better Quality of Service (QoS) to the geographically distributed users.

Increasing volume and variety of the providers, yet with lack of interoperability among their interfaces [102], makes composing service workflows abiding by the user policies a hard problem. The growth of service and infrastructure providers increases the potentials for efficient service composition and workflow placement. Despite the growing number of service providers, users still cannot seamlessly choose services from multiple providers to compose their workflows, due to the incompatibility between the service providers. The ever-increasing volume and variety of services in the edge as well as the IoT devices, often consist of little to no interoperable interfaces. Coupled with these challenges of heterogeneity and interoperability of the service providers, the diverse policies and demands of the users that consume these services make an optimal resource allocation across these platforms for service workflows a complex research challenge.

Software-Defined Networking (SDN) [231] and Network Functions Virtualization (NFV) [148] are two key enablers of network softwarization. Through its unified view and control of the data plane devices, SDN facilitates programmability and management capabilities to the network. SDN separates the control of the data plane devices into a logically unified network controller. Thus, it facilitates a global awareness of the network data plane devices. SDN enables efficient control of the network, typically within a cloud or a data center, but also extended to Wide Area Network (WAN) scenarios such as Content Delivery Networks (CDNs) [351]. On the other hand, NFV virtualizes various network services and deploys them on servers as Virtual Network Functions (VNFs) [36] instead of having them as individual hardware middleboxes [341]. Software middleboxes are cheaper to acquire and easier to manage from a global controller, compared to hardware middleboxes. These traits have indeed facilitated the adoption of network softwarization by several service and network providers [122].

We need a network softwarization architecture, extending SDN for Internet services, to manage the involved multiple heterogeneous infrastructures and users efficiently. While SDN offers network-awareness through the unified view of its controller, it typically limits its scope to a data center. Consequently, several challenges in separating a large-scale network environment consisting of multiple domains from its infrastructure remain unaddressed.

Software-Defined Systems (SDS): SDS are a set of network softwarization frameworks and

approaches inspired by the centralized logical control offered by the SDN. SDS intends to bring the programmability and control of SDN to heterogeneous systems, built atop various network environments. We posit that a network softwarization architecture should be developed as an SDS for network-aware service composition and workflow placement to extend and expand the scope of SDN for heterogeneous service workflows.

A complete SDS architecture should be built to efficiently share resources from the diverse providers for the service composition and execution of several user workflows. As a complete softwarization of the network systems is beyond the scope of classic SDN, recently, more and more SDS have been built, including Software-Defined Storage [323], Software-Defined Data Center (SDDC) [13], Software-Defined Radio [178], and Software-Defined Wide Area Networks (SD-WAN) [238]. These SDS approaches build and manage storage, data centers, and wide area networks via a software control plane that has control over the entire system. Some SDS extend and leverage SDN as their core, while others merely follow a software-defined approach inspired by SDN. Regardless of these promising developments, existing SDS frameworks still focus on a single provider, by virtue of having a unified global view of the system. Multi-domain wide area networks, such as inter-cloud [143] and edge environments, require collaboration and coordination among several providers, each managing their network domain – potentially with an SDN controller. Therefore, additional research and implementation are necessary to make an SDS approach for service compositions at Internet scale, considering the diversity of the services that compose the user workflows.

1.2 Challenges of network softwarization

We posit that a network softwarization framework, with a focus on the end user devices and client applications, can offer the users more control over the network and their applications in a multi-tenant environment. We observe that to fully reap the benefits of the pervasive edge nodes and network softwarization, we should bring the control of the resource allocation and executions back to the users, despite sharing the resources from multiple providers across several geo-distributed tenants.

Network softwarization has yielded positive outcomes concerning the performance and management of the network architectures while minimizing the capital and operational expenditures (CapEx and OpEx) of the enterprises [56]. Two major technological factors drive the preference for network softwarization: i) performance and flexibility achieved by separating the network infrastructure from the network service execution [168], and ii) the ability to control the network flows based on the user preferences from the application plane, for a high QoS [318]. SDN controllers are, in practice, software applications developed in high-level languages, such as Java or Python. Therefore they can be extended and invoked from the application layer. They, on the other hand, control and manage the network data plane devices. Thus, the controllers are capable of providing cross-layer optimizations to the network systems, by receiving status updates from the network plane, while adhering to the policies specified from the application plane. NFV replaces expensive proprietary hardware middleboxes with VNFs that are cheaper

to acquire and maintain [36]. Thus, network softwarization has achieved popularity due to its technological and economic advantages.

Multitenancy: The users have limited control in resource allocation for their workflows composed of several third-party services. The cloud environments consist of users from several organizations. We call these sets of users the *tenants* of the systems. Multitenancy [124], the ability to support several tenants with shared resources, is a core pillar of cloud computing. It advocates sharing of the underlying infrastructure and platform among several third-party organizations, i.e., the tenants of the environment. A tenant typically consists of a set of users controlled by a single administrator account of the organization. Each tenant receives its own 'slice' of the cloud resources without sacrificing the privacy and isolation of data and execution belonging to each tenant, as illustrated by Figure 1.1. However, multitenancy comes with the cost of limited control and flexibility to the end users – the resources are entirely managed and provisioned by the provider, often oblivious of the sophisticated end user policies. Each cloud environment is maintained by its provider, as a network domain independent and often incompatible with other cloud environments. The current cloud and services ecosystems have essentially lead to vendor lock-in, by hindering seamless service migrations across the providers. Subsequently, the users have limited capabilities to consider all the available service instances in multi-domain environments consisting of several cloud and edge providers.

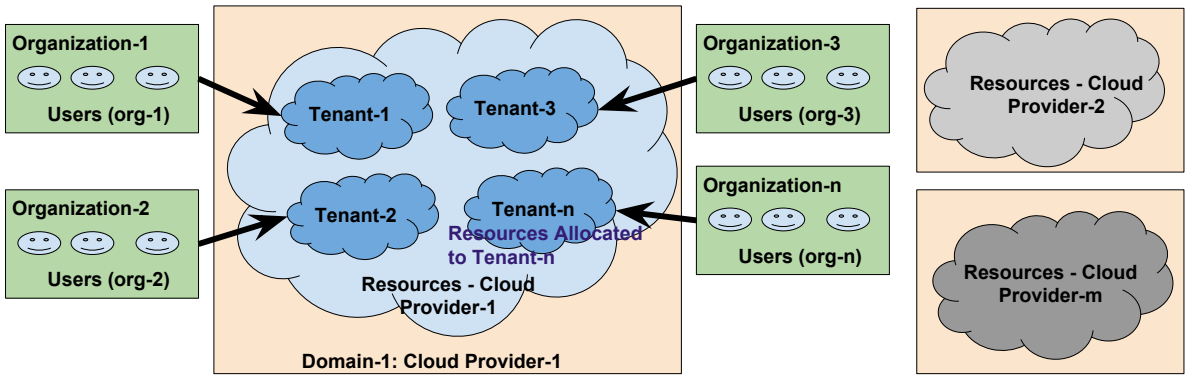


Figure 1.1: Multitenancy and the Tenant Users of a Cloud Environment

SOA Standards: Service-Oriented Architecture (SOA) is a design paradigm that distributes the applications as a client-server architecture, following commonly accepted standards and protocols. The service provider hosts the web services server, consisting of one or more web applications or web services that accept concurrent queries from several (often, geo-distributed) users with clients for the respective web services and applications [248]. Cloud providers have widely adopted SOA due to its support for a distributed user base [209]. Web services are developed following standards and descriptive languages such as the Web Application Description Language (WADL) [145] and Web services description language (WSDL) [76] as well as protocols such as the Representational State Transfer (REST) [270] and SOAP (formerly, Simple Object Access Protocol) [57]. SOA description languages such as WSDL and WADL support defining the functionality of the web services in a unified manner [76]. These protocols mandate standardized Application Programming Interfaces (APIs) to access and update the underlying

data.

Service Composition: Despite the standardization efforts on services fostered by the web service protocols, service composition across multiple providers remains an open challenge. Workflows are composed by chaining the service outcomes, by using the output of a service invocation as the input for another service [11]. In practice, there have been limited coordination across the cloud and network providers. Therefore inter-cloud service workflow execution, where a tenant workflow is composed of service executions at various cloud provider environments, remains a complex undertaking.

Resource Allocation: Sharing of computing and network resources across several tenants makes cloud resource allocation a hard problem to solve. Recently, networks are adopting the concept of multitenancy and network softwarization to share the bandwidth across the tenants [304]. Even though recent advancements on network softwarization indeed attempt to enhance the interoperability and management of the network architectures [217], they limit their focus to the providers, rather than the tenants. Network tenants have their distinct policies as well as preferences for their applications' network flows. Some applications have a higher priority than the others, having more demanding Service Level Objectives (SLOs). Moreover, system-wide policies should be respected while also supporting the policies of the several tenant users, abiding by the Service Level Agreements (SLAs). These constraints make efficiently sharing network resources among several tenants a challenging problem, more so as the tenants have their preferences and policies for their cloud applications and services.

1.3 Thesis Aim and Objectives

This dissertation exploits network softwarization to solve the problems of service composition and workflow placement at Internet scale. Numerous providers offer service implementations with different QoS guarantees and SLAs. Services are inclusive of several variants such as web services, network services, and data services. Service description standards and protocols focus on interoperability across service interfaces, to provision resources and scheduling workflows spanning various service providers. Nevertheless, in practice, standardization of the interfaces remains mostly limited. On the other hand, the cloud and edge environments consist of several tenants and multiple users belonging to each tenant organization, each with their own set of policies, priorities, and SLOs [180]. Consequently, current workflow placement approaches are limited in terms of feasibility, scalability, and optimality in efficiently provisioning resources for user workflows spanning various infrastructure and service providers across the Internet. The diversity of services and their users make an optimal service composition and workflow placement, for a tenant user consuming several third-party services, a complex research problem [190].

We identify several challenges that should be addressed to be able to build SDS for network-aware service composition and workflow placement. We then formulate our research questions that arise from those challenges. The thesis objective is to find solutions for the identified research challenges in service composition and workflow placement, by extending and leveraging network softwarization. Thus, we propose a set of contributions to address the identified research

problem and solve the associated research questions.

1.3.1 Problem Formulation

First, we should extend the scope of network softwarization beyond its current focus that is typically limited to network deployments rather than an end-to-end development and deployment of network environments. In practice, building a network environment consists of several steps, starting with modeling the algorithms and architectures and finally deploying them in production. Network emulators such as Mininet [196] leverage SDN to efficiently emulate a network architecture that can seamlessly be migrated into a physical deployment environment. Nevertheless, the scope of such emulators are limited to data center networks, and it is often infeasible to emulate cloud and edge environments due to resource constraints. In contrast, network and cloud simulators are capable of modeling complex systems, but they are typically incompatible with physical deployments. Hence, seamless migration of a network from simulation to physical deployment is practically impossible. Therefore, currently, network deployments incur repeated efforts between early development (such as models, simulations, emulations, and prototypes) and production, with limited incremental development and deployment capabilities. Consequently, current network softwarization landscape, despite its benefits, has a limited scope – typically restricted to production, and not covering the early development efforts.

Second, we need to complement classic SDN with more widespread light-weight SOA such as Message-Oriented Middleware (MOM) [87] protocols, to achieve interoperability and coordination across service executions at Internet scale. While SDN improves the configurability of the networks, heterogeneous devices and executions on the Internet do not typically support SDN protocols. Furthermore, as a network protocol, the scope of SDN is limited to control of network data plane devices, usually belonging to a single domain such as a data center. Existing approaches that exploit application layer protocols to manage the networks together with SDN [276] are still limited in scalability and interoperability, with little support to diverse application scenarios such as IoT and big data workflows. SOA offers scalable distributed executions across wide area multi-domain networks, supporting a vast range of devices and services. We should extend SDN with SOA to facilitate efficient management of heterogeneous services, rather than just the network infrastructure.

Third, we should identify the potential and feasibilities for coordination and management of an entire wide area network of multi-domain environments such as the edge and multi-clouds. A global view of the whole network achieved by SDN is advantageous in managing and controlling the environments that are operated by a single entity, such as a data center or a cloud provider. While extending SDN for wide area networks has previously been proposed in frameworks such as SD-WAN [238], they limit their focus primarily to network services offered by a single provider. On the other hand, edge and inter-cloud environments consist of several network providers offering diverse services to multiple tenants. Consequently, a global view and control of the entire network are currently infeasible in such multi-domain environments, due to technological as well as administrative and management challenges in making coordination and collaboration across multiple providers. We need to devise an SDS that offers management and coordination

capabilities for multi-domain environments without sacrificing the independence of each domain.

Fourth, we need to identify whether network softwarization can be leveraged to offer an economical alternative to existing network connectivity providers, as an overlay network that provides seamless network service execution and data transfer, independent of the underlying network infrastructure. The economic aspects drive the move towards network softwarization at Internet scale. We can create overlay networks that span the globe on top of cloud Virtual Machines (VMs), by leveraging the cloud providers that have a global presence. Such an overlay network supported by cloud VMs as its infrastructure is known as a Cloud-Assisted Network [131]. However, network softwarization approaches such as cloud-assisted networks have not been studied adequately for their potential benefits concerning monetary cost and performance. We must ensure that the proposed SDS offers better performance and bandwidth for service workflows while incurring same or lower costs.

Thus, in this work, we extend and apply the concept of network softwarization to manage and leverage the services to compose scientific and enterprise workflows and place them in a network-aware manner, in heterogeneous infrastructures. We aim to leverage network softwarization to efficiently control heterogeneous environments, ranging from data centers, clouds, and edge, and manage the development lifecycle of these systems, spanning simulation to deployment phases. Consequently, we propose efficient SDS frameworks to address the identified research challenges, in achieving these goals.

1.4 Research Questions

We first identify whether an SDS approach will be feasible and effective in offering network-aware service composition and workflow placement. We then look into the specifics on how to efficiently build such an approach, and how does it perform compared to the existing approaches. Our research questions, listed below, are driven by several factors, to support the service providers and tenants, concerning interoperability, modeling, performance, monetary cost, and enhanced control and management capabilities. Among the identified research questions, Q3 defines the core of the thesis. Q1 and Q2 facilitate us to reach Q3. Q4 and Q5 are the follow-ups of Q3.

Q1: Can we seamlessly scale and migrate network applications through network softwarization at various stages of development, from simulation to physical deployment?

Typically, deployment management frameworks [125] deploy and migrate actual algorithms across various physical deployments. However, during the early development phase, network architectures and algorithms are developed with different realizations such as simulations and emulations, before the actual physical deployment. We seek to address the challenge of network workload migrations in two-dimensions, concerning scale as well as the development phase. First, our approach should scale the network algorithms and workflows seamlessly along with the infrastructure, from enterprise and cloud data centers to the edge environments. Such a scaling ensures that networks can be deployed across various environments, with minimal challenges of

scale. Second, it should support a migration with little manual and repeated development efforts, from simulations to deployments. Such a migration enables faster end-to-end development and deployment process. Network emulators such as Mininet indeed support a seamless migration between the emulations and physical deployments, by leveraging their native integration with the SDN controllers [96]. However, current network simulators still lack integration with the SDN controllers. Hence, currently it is impossible to manage a simulated network through a centralized controller efficiently, or realistically model the controller algorithms and SDN architectures without having the resources for a one-to-one emulation. We assess whether it is feasible to separate the infrastructure from the execution through network softwarization, and consequently achieve seamless deployments and migrations across the two-dimensions.

Q2: Can such network softwarization offer economic and performance benefits to the end users, through its separation of the network from its infrastructure?

Even if we establish the feasibility for network softwarization approaches for several network architectures, their adoption largely depends on the benefits that they can offer to the end users. To this end, we look into approaches such as cloud-assisted networks, and when they can be economically and technologically sustainable. Hence, we seek to answer two primary questions in a use case scenario of leveraging a cloud-assisted network as an alternative connectivity provider. First, when would such an approach be cheaper than existing alternative connectivity providers, including transit providers, Internet Service Providers (ISPs), and Multiprotocol Label Switching (MPLS) [93] network providers? Second, we need to look into the technological aspects: would a software-defined approach be able to provide higher performance or additional features such as enhanced control of the network application executions to the user? It is necessary to study the performance benefits of a cloud-assisted network compared to the alternatives for such an approach to be technically and economically viable and beneficial. Furthermore, we should exploit opportunities that may arise from a separation of the network from the infrastructure. For instance, the viability of a network provider built over multiple cloud offerings and that does not own any fixed or dedicated resources, and the potential to easily manipulate the network flows based on the user policies exploiting the flexibility of network softwarization.

Q3: Can we efficiently chain services from several edge and cloud providers for tenant workflow compositions, by federating SDN deployments of the providers using SOA?

A complex workflow typically depends on several services and therefore requires chaining of the service outcomes. The flexibility of NFV enables the end users to compose service workflows optimally by leveraging several third-party edge VNFs. However, in practice, seamless workflow executions across multi-domain environments require communication and collaboration between the execution environments of multiple providers. While SDN manages a single domain such as cloud and data center networks, we need to federate the SDN deployments across these environments to enable tenant workflows spanning multiple domains. On the other hand, tenants need to schedule their services in a network-aware manner, abiding by their policies, for the high overall performance of their service workflows. Thus, the tenants should have increased direct control of their workflow executions with the ability to exploit multiple providers for their service instances and choose the best fit for their service compositions. We seek to federate various SDN

environments with SOA to enable a seamless execution across wide area multi-domain networks and efficiently schedule the service workflows from the decentralized user devices, ranging from servers to smart mobile devices.

Q4: Can we enhance the interoperability of diverse and distributed real-world applications, by following SOA extended with network softwarization?

Various execution scenarios, including big data applications and smart environments such as the CPS and the IoT, typically have limited standardization and interoperability across different implementations. Big data workflows and smart devices consist of diverse storage media and APIs, managing heterogeneous data formats. Nevertheless, their execution is confined to their platforms as their interfaces offer limited interoperability beyond their own framework. The incompatible interfaces prevent users from choosing multiple providers freely for their workload executions and migrate seamlessly across the providers. In contrast, web services are developed for a distributed and interoperable execution. Therefore, an increasing number of big data workflows are composed of *data services* (also known as *big services* or *big data services*) [357], web service implementations that access and process big data in a standardized and decentralized approach. Similarly, compositionality demands in CPS [251] can be met with an SOA approach, by adopting a client-server web services architecture and distributing the user workload as service executions. Given this state of affairs, we aim to understand how to leverage SOA and network softwarization to provide interoperability among the executions of diverse distributed applications and heterogeneous services.

Q5: Can we improve the performance, modularity, and reusability of big data applications, by leveraging network softwarization and SOA?

Big data applications should consider network proximity to minimize communication overhead during the distributed execution of their data-intensive workflows. Standard web service executions are agnostic to the network characteristics beyond data centers. However, the need for a network-aware execution is more prominent in data services than the traditional web services due to the communication overhead caused by moving large volumes of data across the data storage and execution nodes. Therefore, building big data applications with network-agnostic data service workflows is inefficient. We seek to find whether we can build complex big data applications as service workflows by exploiting SOA while leveraging network softwarization for latency-awareness and Quality of Experience (QoE) in the workflow execution. We first aim at enhancing the modularity and reusability of the big data executions by exposing them as data service workflows. Ultimately, we aim at increasing the performance of the big data executions in multi-domain wide area networks such as the Internet, by building an SDS framework for data service execution.

1.5 Thesis Contributions

This dissertation proposes a unified SDS framework that extends network softwarization to address several research challenges of multi-domain networks that hinder a network-aware service composition and tenant workflow placement at Internet scale. Our unified SDS framework

consists of several interdependent components, each focusing on one or two of the identified research questions.

1.5.1 Unified SDS Framework

Our proposed unified SDS framework aims to mitigate the challenges in composing and placing diverse service workflows across heterogeneous multi-domain environments consisting of networks managed by multiple providers, by extending and leveraging network softwarization to wide area networks shared by several tenants. Network softwarization promises significant enhancements on how service workflows are composed and placed, from the perspectives of both networking and application layers. We see this dissertation as the first exploration and extension of network softwarization at Internet scale with various tenants and several independent network, infrastructure, cloud, and service providers.

As the core enabler and prerequisite of our framework, we first must ensure interoperability across the diverse execution environments as well as the variety of tenant workloads, for seamless scheduling and migration of service workflows. The services composing the workflows should be interoperable, to execute service workflows spanning multi-domain networks managed by multiple providers, with minimal repetitive development and manual deployment efforts from the network application developers. Therefore, we propose a cloud network development and deployment approach that unifies the network application execution - across not only the deployment environments but also the development stages. We see this as the first step in enabling interoperability from the tenant application itself. By facilitating service execution migrations across early development stages such as simulations and emulations, we highlight the feasibility of execution migration across the environments that typically lack interoperability.

The current cloud ecosystem consists of mostly disconnected networks with little interaction and coordination across the infrastructure and service providers at the cloud and the edge. The ability to compose a workflow spanning multiple providers is limited not only by the technological challenges but also by the business vision and the enterprise policies of the providers. Our SDS framework separates the network from the infrastructure, to counter the dependence of the network service workflows on the providers. It facilitates an independent third-party cloud user to consume resources from multiple cloud providers seamlessly. Consequently, it lets the cloud user be an alternative connectivity provider that offers an overlay network to the end users. This approach enhances the interoperability of the cloud infrastructure further, while also providing performance enhancements to the tenant cloud applications with the inter-cloud architecture.

Moreover, the current services ecosystem gives limited flexibility to the tenants who share the network for their services placement and execution. This limited flexibility and control given to the tenants prevent them from leveraging multiple service providers at the cloud and the edge for a single workflow, even when the tenants possess the ability to develop and deploy network applications seamlessly across various execution environments. Even the current network softwarization approaches such as classic SDN give a limited capability for the tenant applications to pass their preferences and policies to the network level. This state of affairs limits the tenants from efficiently ensuring SLOs at the network level, more so when the network is

managed by a third-party and shared by several other tenants. Our SDS framework extends network softwarization to enable efficient use of middleboxes to tag the network flows from the applications with user-specified information such as the SLOs, policies, and priority levels for their network flows. It thus provides better flexibility and control for the tenant workflows, and offering differentiated QoS and service level guarantees based on the tenant demands.

Our SDS framework further enriches the services ecosystem with its core contributions, while having the aforementioned interoperability enhancements in place as a prerequisite and foundation for network-aware service composition and workflow placement across the multi-domain networks. As our core contribution, we then extend SDN beyond data centers, with MOM. By using messages for inter-domain coordination, rather than static dedicated network links, the extended SDN architecture ensures not to introduce hierarchies among the networks, or compromise their security. Using a MOM approach facilitates multi-domain control via dedicated network connections as well as the public Internet with protected access, thus further aiming to enhance the performance, scalability, and extensibility of the multi-domain service workflow compositions and migrations. Our framework thus enables composing workflows on services deployed on multiple network domains and migrates the execution based on the load on the service instances.

While our enhancements enable composing network service chains, consuming the services deployed on multiple edge and cloud providers, we must compose these chains adhering to the user policies and SLOs. Remarkably, service chains have additional constraints compared to the stand-alone service instance selection and execution. Our SDS framework consists of optimization algorithms to ensure that the service instances are chosen while adhering to both user policies and resource availabilities at the nodes. It uses both the network-level statistics facilitated by the SDN architecture extended with MOM, as well as the service-level statistics from the web services engines. It further exploits graph-based algorithms and formulates MILP to solve the network service chain placement as an optimization problem for multi-domain inter-cloud and edge environments.

We finally look into the practical use cases of our contributions. Our proposed enhancements to the services ecosystem with network softwarization provide a first-step towards interoperable network-aware service composition workflows at the Internet scale. We demonstrate CPS and big data workflows as the two use cases of our proposed SDS framework. In the case of CPS, we expose the CPS as composable microservice chains and thus enabling the use of our SDS framework to execute them efficiently at the edge. For big data workflows, first, we propose an approach that allows service-based access and processing across heterogeneous big data environments. Big data consists of a large volume of data and a variety of data types from diverse data sources. Our proposed unified data service approach to big data enables exposing big data workflows as composable data service chains. Consequently, our SDS framework facilitates big data workflows spanning multiple providers, the same way it supports the scheduling and execution of multi-domain workflows of web services, network services, and microservices of CPS.

Our research contributions aim at addressing the identified challenges from three fronts - namely, networking, services, and big data. Our unified SDS framework exploits network softwarization, SOA, and data services, respectively, to tackle the challenges in these fronts. We

thus propose a set of SDS components, as part of our proposed unified SDS framework. Each of the SDS components individually addresses a particular limitation on these three fronts to enable service compositions from third-party service providers for user workflows adhering to the tenant policies and SLOs, while not owning the infrastructure and sharing the resources with several other tenants of the service or infrastructure provider. Our SDS components function individually on their own to address their specific research problem, and also as part of the unified framework towards the global goal of the unified SDS framework. Our contributions collectively aim to optimize user workflow placement in a scalable and network-aware manner in various scales.

Our contributions fit under three primary parts – network softwarization, SOA, and data services. Each of the parts builds upon the previous ones. First, we research the challenges and opportunities of network softwarization in data centers and clouds, regarding performance and QoS. We thus build frameworks that exploit the benefits of network softwarization for technological and economic enhancements of the end user application execution. Second, as the core of the thesis, we propose our SDS framework for network-aware service compositions and workflow placements. We exploit our findings from the network softwarization research and extend them with SOA. Third, we present interoperable big data services as an extended use case of our research contributions on SOA. We finally extend our research findings on network softwarization for network-aware execution of big data workflows with SOA.

1.5.2 Individual Contributions

This dissertation develops and presents several individual research contributions as part of the proposed unified SDS framework for service composition and workflow placement in multi-domain networks. Each of the research questions identified in Section 1.4 is addressed through one or more contributions, whereas each contribution focuses on one or two research questions. Each contribution is composed of research work presented in our journal papers (J), book chapters (B), conference papers (C), symposium and workshop papers (W), and short papers (S). A complete list of publications is presented at the beginning of this document, identifying each contribution by a respective index [J1 – J4, B1 – B2, C1 –C6, W1 – W7, and S1 – S2]. Figure 1.2 presents the major thesis contributions against the respective research questions that they attempt to address along with the scale of their execution environment. It further illustrates the relationships among the individual contributions in addressing the research questions.

Our contributions solve our identified research questions in heterogeneous execution environments of an increasing scale: i) intra-domain (inside data centers and cloud data centers), ii) multi-domain (across multiple cloud providers and multiple data centers), iii) edge, and iv) the Internet. *SENDIM* aims at solving Q1 in cloud data centers. *SMART* solves Q2 in data centers. *NetUber* solves Q1 and Q2 inside a single cloud provider as well as across multiple cloud environments. *Mayan* and *Évora* solve Q3 in multi-domain and edge environments respectively. *SD-CPS* solves Q3 and Q4 in edge environments. *Óbidos* solves Q4 and Q5 inside and between data centers. *Mayan-DS* solves Q4 and Q5 at Internet scale.

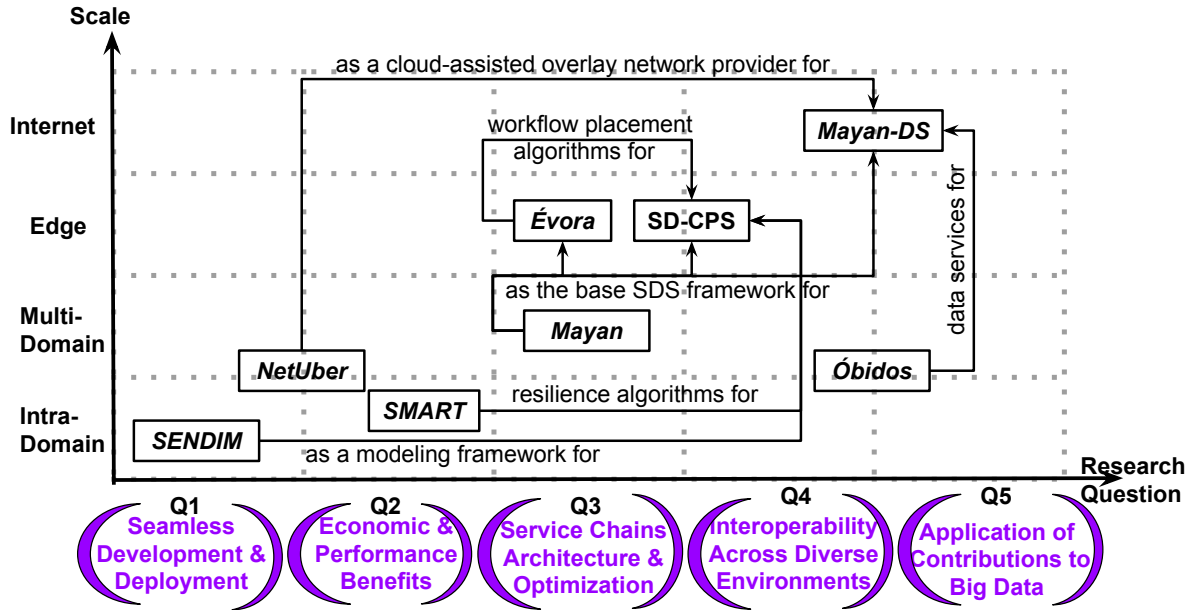


Figure 1.2: Thesis Contributions

Our latter contributions leverage our previous contributions as their core. *Mayan* functions as the core SDS framework for our latter contributions, *Évora*, *SD-CPS*, and *Mayan-DS*. Furthermore, *SD-CPS* exploits *SENDIM* as its modeling framework, *SMART* for its resilience, and *Évora* for its workflow placement. *Mayan-DS* exploits *Óbidos* as its data services and *NetUber* as its cloud-assisted overlay network provider. Below, we summarize the thesis contributions, identified by the research questions that they aim to answer along with their target execution environments.

(I) Network Softwarization

Q1 (cloud data centers): *SENDIM* [C3, W7, S2], a Software-Defined Cloud Deployment (SDCD) framework that extends network softwarization to separate tenant workloads in the clouds and data centers from their deployment environment. *SENDIM* offers an integrated and more encompassing modeling approach consisting of seamless simulations, emulations, and physical deployments to network application development. Instead of limiting its focus to physical deployments, *SENDIM* exploits network softwarization to bridge the gap between these various stages of the development procedure. It further provides unified deployment and migration capabilities across these various realizations of the modeled algorithms and architectures.

Q1 & Q2 (clouds): *NetUber* [C1], a virtual connectivity provider without any fixed infrastructure. First, we study whether a cloud-assisted network built on several cloud instances can be a viable solution to realize an on-demand virtual Network-as-a-Service (NaaS) provider. We then propose *NetUber* as a dynamic connectivity provider that can be built entirely on cloud spot instances (i.e., cheap but volatile cloud VMs offered by cloud providers that can be shut down by the cloud provider with short notice when the resources are scarce for

the cloud provider) by third-party users. *NetUber* builds a dynamic overlay over the cloud resources and offers it to the end users as an economical or high-performance alternative to ISPs and enterprise connectivity providers.

Q2 (data centers): *SMART* [W3, S1], cross-layer optimizations that leverage SDN and software middleboxes to offer various levels of QoS to multi-tenant network flows. *SMART* tags the network flows with contextual information such as tenant policies from the user applications and reads them from the controller. *SMART* then exploits the tags to provide QoS at the network level abiding by the respective user preferences. *SMART* further imposes redundancy selectively to ensure resilience for critical tenant network flows. It thus supports user policies efficiently while sharing the network among multiple tenants.

(II) Service Oriented Architecture

Q3 (inter-domain): *Mayan* [C2, W1, B2], a Software-Defined Service Composition (SDSC) framework that extends SDN with MOM for network-aware service compositions in multi-domain wide area networks. *Mayan* aims to make complex workflows as compositions of web services decoupled across multi-domain network environments, with its SOA implementation. At the same time, it uses an extended SDN architecture to make the service compositions and workflow placements network-aware. *Mayan* further enables communication between the service endpoints on the Internet via various communication channels facilitated by the interoperable web service APIs. It expresses network applications as composable service workflows and seamlessly places and migrates them across the networks. Thus, *Mayan* brings the configurability and programmability of SDN to service compositions beyond the data center scale.

Q3 (edge): *Évora* [J1] algorithms that enable users to compose Network Service Chains (NSCs) by chaining several third-party edge VNFs efficiently. *Évora* aims at bringing the control of the NSCs back to the end users, despite sharing the infrastructure with several other tenant users. It composes NSCs by choosing the VNFs adhering to the user policies, in edge environments consisting of multiple network domains. It ensures efficient stateful executions by scheduling the subsequent service workflow invocations from the same user to the same VNF instances. In the case of service unavailability, *Évora* also ensures seamless migration of the workflow execution to the next best service composition of VNF instances, together with the current state and outcome of the workflow execution. Thus, extending the *Mayan* architecture for NSCs, *Évora* proposes optimal VNF allocation algorithms for the NSC execution at the edge.

Q3 & Q4 (edge): Software-Defined Cyber-Physical Systems (SD-CPS) [J2, C6, W6] that coordinate each CPS execution step, performed by a microservice [246], through an extended controller deployment. *SD-CPS* extends the *Mayan* architecture for smart execution environments such as the IoT and CPS. It exploits *Évora* for agile and resilient CPS workflow placement and execution at the edge. Each light-weight microservice of *SD-CPS* performs a simple single action, optionally holding its own data store to cache the outcomes of the microservice execution. By creating, placing, deploying, migrating, and managing the CPS computing processes as service workflows at the edge, *SD-CPS* orchestrates the entire

CPS lifecycle effectively and efficiently. Thus *SD-CPS* addresses the general challenges of CPS, concerning modeling, development, performance, management, communication and coordination, scalability, and fault-tolerance, through its SDS architecture.

(III) Data Services

Q4 & Q5 (inter-data center): *Óbidos* [J3, C4, W2, W4], a hybrid ETL (Extract, Transform and Load) approach for scientific data integration that incorporates human-in-the-loop for selective data integration driven by the user queries and sharing of integrated data between users. *Óbidos* enables user-driven data services in a multi-tenant big data execution environment. *Óbidos* outperforms both the eager ETL [330] and lazy ETL [183] approaches, for scientific research data integration and sharing, through its selective loading of data and metadata, while storing the integrated data in a scalable integrated data repository.

Q4 & Q5 (Internet): *Mayan-DS* [J4, C5, B1], a Software-Defined Data Services (SDDS) framework that optimally schedules the data services at Internet scale, considering the locality of data and execution as well as the several alternative network paths. *Mayan-DS* extends *Mayan* for an adaptive network-aware execution of data services and big data applications such as *Óbidos*. It provides interoperability for big data executions by representing them as composable data service workflows. It further exploits SDN for a latency-aware execution of data service workflows, at various scales – from data centers to the Internet.

Based on how each of our contributions fits towards achieving the overall thesis vision, we group them into three categories – as the core, prerequisites, and the applications of the contributions. First, *Mayan* and *Évora* constitute the core of the thesis. Second, *SENDIM*, *NetUber*, *SMART*, and *Óbidos* provide us with the early work that facilitates us to build our core contributions on top of them. Third, *SD-CPS* and *Mayan-DS* apply our contributions to real-world scenarios such as CPS and big data applications. As such, *SD-CPS* and *Mayan-DS* leave us with avenues for future work in research and implementation. Among the early works, *Óbidos* offers a sample SOA implementation to big data with real-world data, thus allowing evaluations on big data required for our work on SDDS and our future deployment of *Mayan-DS*. The rest of our early work provides the foundation of networking and SDN for the core of the thesis.

1.6 Thesis Roadmap

This document is organized as the following parts, as depicted by Figure 1.3. We discuss the background and related work in Chapter 2. Part II presents our network softwarization research to design QoS-aware platforms for building and deploying multi-tenant workloads from data centers to cloud environments. Part III presents our SOA research to compose and execute diverse multi-tenant workloads as service composition workflows. Consisting of our core contributions, Part III exploits our contributions from Part II on network softwarization to service composition and workflow placement in multi-domain wide area networks and their applications such as CPS. Part IV elaborates our data services research to compose and deploy network-aware

big data workflows at Internet scale. It serves as an extension and a use case study of Parts II and III, with a focus on big data.

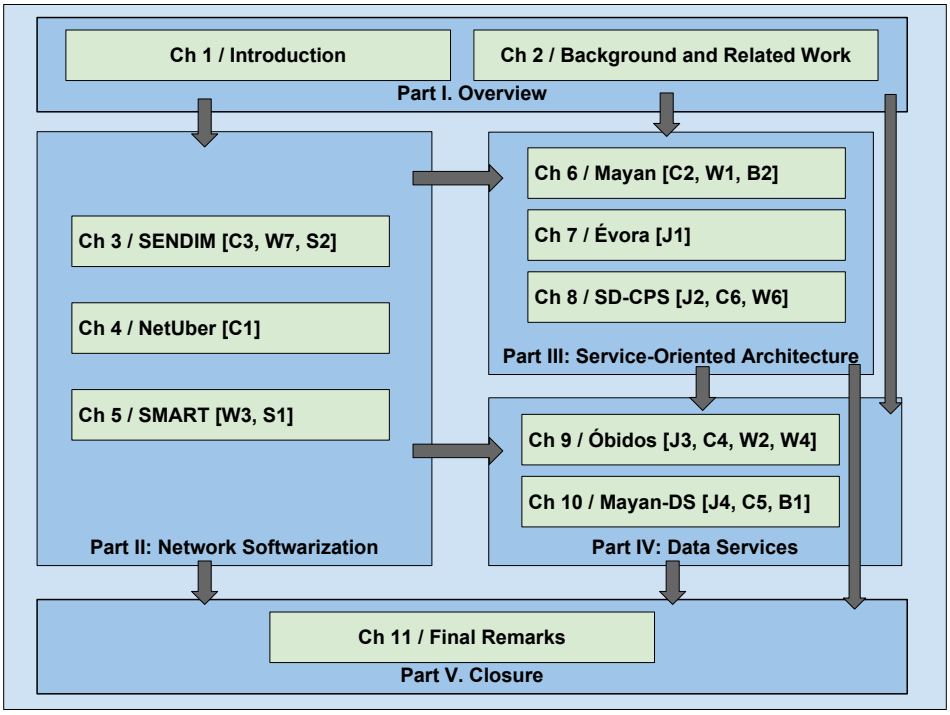


Figure 1.3: Thesis Overview

In Part II, Chapter 3 and Chapter 4 present our research for seamless and flexible deployments of network applications by decoupling the networks from the underlying infrastructure. Chapter 3 presents *SENDIM* [C3, W7, S2], an SDS for modeling and placing workflows at data centers. Chapter 4 proposes *NetUber* [C1], an economical connectivity provider built as a cloud overlay. Chapter 5 describes *SMART* [W3, S1], a software middlebox architecture that leverages network softwarization to tag and prioritize network flows in a data center environment, by selectively enforcing redundancy.

In Part III, Chapter 6 presents *Mayan* [C2, W1, B2], an SDSC framework to compose and execute service composition workflows efficiently in wide area networks by leveraging SDN and web services engines. Chapter 7 presents *Évora* [J1] algorithms to choose the best-fit VNFs to compose NSCs at the edge, abiding by the user policies. Chapter 8 presents *SD-CPS* [J2, C6, W6], an SDS framework that combines and exploits the previous contributions of the thesis for CPS.

In Part IV, Chapter 9 describes *Óbidos* [J3, C4, W2, W4], a multi-tenant big data integration framework for on-demand human-in-the-loop data integration. Chapter 10 presents *Mayan-DS* [J4, C5, B1], network-aware workflow scheduling at Internet scale for data services. Finally, in Part V, Chapter 11 concludes this document with an overall summary of our research findings and future work.

2 Background and Related Work

As the scale and complexity of the systems are growing larger and larger, programmability of the networks is researched intensively [66]. SDN increases the reusability of the network algorithms, by providing a logically centralized control plane separated from the data plane that forwards the data [231]. In this chapter, we will look into the background and related work, on network softwarization and service chaining in the data center, cloud, and edge environments, and SDS for IoT, CPS, and big data.

2.1 Network Softwarization

First, we look into the related work on network softwarization, specifically on network modeling and deployment, decoupling networking from the infrastructure, and network flow scheduling in data centers with the recent advancements on network softwarization.

2.1.1 Software-Defined Networking (SDN) and Software-Defined Systems (SDS)

SDN research aims at increasing network management capabilities in heterogeneous environments, ranging from network emulations to data centers and clouds [46]. SDN can be leveraged to handle network flow load balancing [149, 348] and policy enforcement [280] effectively. While previous works highlight the network management capabilities of SDN, further research is necessary to centrally orchestrate the computing, networking, and storage resources for heterogeneous network environments.

Enterprise SDN controllers have APIs that support interaction between their components and external entities. These APIs are categorized into northbound, southbound, eastbound, and westbound, based on who/what they typically communicate with. The southbound API lets the controller communicate with the SDN switches, by implementing the SDN protocols such as OpenFlow [232] and the Forwarding and Control Element Separation (ForCES) [108]. The northbound API is the user-facing API that enables the users and their network services and applications to interact with the controller. It usually implements the REST and MOM protocols. The eastbound and westbound APIs are the APIs for administration (i.e., the configuration of the controller and the network) and inter-domain control (controller-controller communications) traffic. The controller APIs facilitate communication and coordination inside and outside an SDN network. These APIs support the control of the data plane devices efficiently, while also enabling extensions to the controller for additional functionalities.

Several distributed and pervasive systems exploit OSGi (formerly known as the Open Services Gateway initiative) [259] for their modular and extensible architecture [286]. OSGi frameworks support componentization of platforms and architectures, where a user can develop and deploy new features and extensions as interactive OSGi bundles into an existing OSGi framework dynamically and incrementally. An OSGi framework integrates seamlessly with project management and dependency management platforms such as Apache Maven [242]. OpenDaylight [233] and the Open Network Operating System (ONOS) [47], two popular SDN controllers managed by the Linux Foundation, leverage Apache Karaf [250] as their core OSGi framework. The modular architecture facilitated by OSGi enables efficient configuration and extensions of the controllers. Thus, they achieve interoperability and extensibility, providing network orchestration and service management.

Distributed controller architectures such as FlowVisor [304] and Hyperflow [325] offer scalability to the controller to manage larger networks without causing overhead to the network control plane. FlowVisor expands the OpenFlow controller to be able to run parallel independent experiments. Hyperflow proposes a distributed control plane for OpenFlow. These distributed controller architectures provide a centralized logical view while reducing the potential bottlenecks caused by a physically centralized architecture [204].

As classic SDN is limited in scale and scope, several SDS have been built, by extending SDN. SDS data plane consists of multiple (often, dumb) devices or components, which can be viewed and controlled by a logically centralized control plane. Software-Defined Data Center (SDDC) extends virtualization, SDN, Software-Defined Storage [323], and middleboxes to have programmable data centers with a better QoS. Software-Defined Flash proposes a hybrid Software-Defined storage consisting of software storage as well as a solid-state drive (SSD) as a large-scale Internet storage system [260]. Extending SDN beyond data centers, SD-WAN [163] offers a centralized control to manage and orchestrate a wide area network, a much larger scale of networks compared to the traditional data center networks. SDS provides the management capabilities of the conventional centralized systems, while not compromising the scalability and performance of the distributed systems, offering the best of both worlds.

The current SD-WAN offerings focus on a given enterprise, rather than the end users consuming several enterprise offerings. SD-WAN transfers data between geographically distributed data centers using MPLS, Long-Term Evolution (LTE), or broadband. Leveraging multi-wavelength networks, SD-WAN achieves higher throughput and lower latency [163]. Approaches such as MPLS and SD-WAN enable volatile data to be handled well within the SLAs [30] and time limits in a WAN. There are several enterprise offerings of SDDCs such as Big Switch [303], Microsoft [239], and Plexxi [229]. Nuage [253] offers SDDC as well as SD-WAN. However, primarily all these efforts focus on the enterprises, and there is a limited control to the end users in achieving SLAs and QoS in the network level, based on their own policies. We posit that with the raising offerings and the complex applications of the end users, SDS frameworks should give more prominence to the end users, to offer their executions better QoS and flexibility in wide area networks, while not sacrificing scalability.

OpenDaylight

The modular architecture and industrial support of OpenDaylight have enabled several SDS to be built leveraging OpenDaylight. OpenDaylight consists of a Service Abstraction Layer (SAL) mechanism to interact and integrate with the other entities (i.e., devices, applications, and other controllers) in the network environment. It has a static API-Driven Service Abstraction Layer (AD-SAL) with dedicated REST API for each plugin and a dynamic Model-Driven Service Abstraction Layer (MD-SAL) with a common REST API. MD-SAL, the current default mechanism of OpenDaylight for its SAL, offers a seamless deployment and installation of additional features, further enabling efficient use of OpenDaylight's modular architecture. The southbound API that connects the controller to the devices is modular, with implementations to standard protocols such as OpenFlow as well as proprietary interfaces. The northbound API that connects the controller to the applications is intent-based, thus enabling to extend and expose the controller capabilities to the applications and architectures deployed on top of the controller. MD-SAL represents both the southbound devices as well as the northbound applications as models, i.e., objects developed in YANG (Yet Another Next Generation) [50] data modeling language.

Internally, OpenDaylight handles the models in an approach compatible with MOM. A SAL model is either a producer or a consumer, based on their role in the interaction, regardless of whether the data comes from the devices or the applications. This unified representation of devices and applications enables integrated processing of them within the SAL. MD-SAL exposes OpenDaylight's internal data, composed of the data tree, Remote Procedure Calls (RPCs) [314], and notifications through the controller APIs. A producer writes data to the data tree and implements relevant APIs, whereas a consumer reads from the data tree leveraging the API. The producer generates notifications for any of its events in the controller. The controller receives relevant notifications based on its interests. The consumer then issues an RPC to retrieve the data from the provider. MD-SAL APIs are asynchronous, unlike AD-SAL that offers both synchronous and asynchronous APIs. However, MD-SAL APIs return a *Future* object, which can be blocked until the execution completion, to emulate the synchronous behavior. OpenDaylight's flexible policy management enables efficient network service execution in multi-tenant environments. OpenDaylight automatically and securely discovers the devices and controllers through its Authentication, Authorization, and Accounting (AAA) framework. It further enables chaining of services and protocols, thus allowing us to implement VNFs and NSC orchestration capabilities on top of the controller, with less development effort compared to the other SDN controllers.

2.1.2 Network Modeling

Currently, network modeling frameworks such as simulators and emulators have limited compatibility across each other. Compatibility between the emulators and controllers facilitate smooth migrations between emulated environments and physical environments [96]. Nevertheless, network modeling framework APIs are incompatible among each other, due to lack of standardization across the network modeling frameworks. Furthermore, unlike the emulators,

network simulators lack native integration with the controllers. Therefore, additional effort is necessary to unify and enable seamless execution across the diverse modeling frameworks.

Network emulators can emulate cloud networks that can be managed by an SDN controller, with minimal configuration efforts. Mininet emulations closely resemble physical networks, by emulating the Open vSwitch [275] virtual switch developed for hardware virtualization environments. Servers emulated by Mininet can also execute or invoke the processes and applications installed in the host machine, and hence can be configured to provide emulation of an entire cloud system inside a single computer [196]. Maxinet distributes the Mininet execution to a computer cluster for a single SDN emulation [350]. Therefore, it succeeds in mitigating the challenges of scalability and resource scarcity in a Mininet emulation.

Simulators are typically used when the considered system is too complicated to emulate within the given time with the available resources, or in earlier stages of development when a simulation is adequate. SimGrid simulates the networks in network flow level, where simulators such as GTNetS simulate at packet level [332]. NS-2 is a widely used network simulation tool [70], that was later extended as NS-3. Leveraging the Message Passing Interface (MPI), NS-3 provides a distributed simulation of networks [70], which can also be introduced into a live network, offering emulation capabilities to some extent [118]. Frameworks with the ability to both simulate and emulate the cloud networks, such as NS-3 and EmuSim [62], fall short in simulating controller algorithms without re-writing them. Hence, they do not offer a seamless migration to physical network deployments from simulations or emulations.

Current SDN controllers lack the simulation capabilities or an efficient simulator integration unlike their seamless integration with emulators. SDN controllers are not optimized for quick modeling or prototyping without involving an external modeling framework such as an emulator. Even the controllers with an extensible modular architecture, such as OpenDaylight and ONOS, do not natively support cloud simulation. The other SDN controllers such as Beacon [115], Maestro [61], POX [186], and Floodlight [123], also do not offer simulations or modeling capabilities. To provide seamless modeling capabilities, we posit that the controller southbound APIs should be extended to connect to simulators, by carefully simulating the OpenFlow switches.

A network simulator should be efficient at the same time should easily integrate or interact with the SDN controllers, to truly replace emulators at the early stages of SDN design. By mimicking the behavior of the network emulators, while not emulating the network, a simulated network can also be managed by an SDN controller. CloudSimSDN [310] extends CloudSim [63] for the simulation of software-defined cloud and data center networks. Nevertheless, CloudSimSDN or the other existing cloud or network simulators do not simulate the SDN southbound to control the simulated network via the controller. Therefore, they do not resolve the limitations in migrating simulations to emulations during the continuous SDN developments.

An approach to support automatic migration across various realizations such as simulations and emulations requires an extension beyond the current configuration management tools. Configuration management tools such as Chef [320], CFEngine [59], and Puppet [216] automate and manage the configurations of the servers and VMs in cloud-scale deployments [313]. However, these tools are entirely oblivious to the code that they deploy as they target the system administrators and cover mostly the deployment stage. Therefore, it is still impossible for them

to identify the early development phases and configure and deploy the applications in a unified manner.

Existing modeling frameworks do not adequately separate the application logic such as load balancing, application scheduling, and policy control in SDN, from the descriptions of the system that is simulated or emulated such as the properties of the data center and parameters of the network. Hence, the potential reuse of the application logic across simulation and emulation environments is prevented. Integrating simulation, emulation, and deployments among heterogeneous systems has been proposed and researched to mitigate the administration overhead, in systems such as sensor networks [133]. However, an integrated framework to enable incremental development and deployment of SDN applications, as well as migration between different physical deployments, is still lacking.

Further research is necessary to build a network modeling framework that can integrate and leverage SDN controllers for its operation, while still being able to operate as a compact stand-alone network simulator. Modeling frameworks should be optimized to function in a more unified manner, as much as possible, to reduce repeated development efforts and learning curves. Nevertheless, we observe that modifying or extending the APIs of the current network simulators for interoperability and integration with an SDN controller indeed requires considerable effort. Therefore, we propose that a modeling framework compatible with the SDN paradigm, and that can simulate SDN networks with seamless integration with the controllers, needs to be developed. We posit that such a unified approach should manage the entire development process, from visualization, simulation, emulation, to physical deployments, to increase the productivity by minimizing the installation hell imposed due to the manual migration across multiple platforms.

2.1.3 Decoupling Networking from the Infrastructure

The desire for flexibility in interconnections by network operators has led to an increasing interest in decoupling the network from its underlying infrastructure. The massive amount of content shared on the Internet, coupled with the bandwidth requirements to provide high QoE for latency-sensitive applications such as online gaming and video conferencing, has resulted in ever-increasing bandwidth demand. Consequently, cloud-assisted networks have been recently proposed, to increase performance, flexibility, latency-awareness, and reliability of wide area networks [150]. Cloud-assisted networks leverage cloud resources to compose large-scale overlay networks. This approach is appealing because cloud platforms generally guarantee high levels of availability through various levels of SLAs [41].

The research proposes virtual connectivity providers that do not control the infrastructure [121, 360, 60]. Software-Defined Internet Architecture (SDIA) [284] decouples the architecture of the Internet from the infrastructure, by modifying the way interdomain tasks operate, through SDN and MPLS. Jingling [132] separates the network functions outside the network towards external *Feature Providers*. Cabo decouples ISPs into infrastructure providers and service providers, with concurrent networks that run multiple virtual routers atop each physical router, thus virtualizing links between any two virtual nodes [121]. Slicing the home networks can enable various service providers to reduce the costs and overhead associated with deploy-

ment and management, by sharing a common infrastructure [360]. A trusted third party such as Google Fi [137] can function as a virtual ISP by exploiting the resources of multiple ISPs [370]. CloudDirect [79] offers auxiliary services such as backup and disaster recovery atop cloud offerings. These research efforts have shown promising outcomes by decoupling networking from the infrastructure.

There have been industrial efforts, aiming at a fast direct interconnection between two endpoints, without relying on traditional connectivity providers in the region. PacketDirect [262] is an SDN-based platform that reduces the time to set up interconnections. MPLS providers such as iTel [172] connect multi-location decentralized offices with a private layer-2 network, a unified connection to the whole organization. These providers differ from the traditional MPLS networks that merely provide connectivity between two endpoints, thus still requiring Ethernet connections for each office. Furthermore, Megaport [235] and Console Connect [84] provide scalable point-to-point connectivity to cloud and network providers. Nevertheless, these enterprises focus on offering end-to-end connectivity for enterprises, rather than end users.

Several new service providers are starting to offer cloud-based networks as an alternative to traditional connectivity providers such as ISPs. Primary cloud providers such as Amazon have built their own global backbone network [287]. Therefore, they do not rely on the transit providers for their data transfer. These cloud networks are well provisioned and maintained, which makes them better than the Internet paths regarding loss rate and jitter [151]. Based on these observations, companies such as Teridion [321] and Cloudflare [80] offer cloud-assisted networks for Software-as-a-Service (SaaS) providers as a premium service for a higher price compared to using standard Internet-based connectivity. Voxility [338] leverages its vast distributed infrastructure to provide network services and end-to-end interconnection, cheaper than the transit providers with more flexible agreement options for short-term and small-scale interconnections.

Spot instances (also known as *preemptible instances* or *low-priority VMs*), which are cheaper but volatile cloud instances, have been exploited in the research for economical cloud architectures. Cloud providers such as Amazon Web Services (AWS) and Google Cloud Platform (GCP) have been auctioning underutilized computing resources in their marketplace as spot instances for a much lower price, compared to their on-demand instances, while offering the same resources and capabilities as their on-demand counterparts. Cloud platforms typically consist of a *spot market*, where the cloud provider sells their spot instances. The spot markets generally have idle and affordable resources in multiple regions. Bidding in the spot markets of multiple regions can minimize the costs of CPU-intensive workloads, increasing the availability of the Internet services [155]. Cloud brokerage services have been built on spot instances with scheduling and reservation mechanisms, to minimize computing costs for jobs with a strict deadline, up to 57% [359]. Cost efficiency and performance of in-memory caches have been improved in the cloud, by deploying in spot instances while exploiting burstable instances for a backup [344]. But the scope of those research work is limited to computing. Consequently, there has been limited research efforts on utilizing cloud spot instances and a cloud-assisted network for economical network connectivity, data transfer, and network resources.

Various approaches have been proposed, to reap the economic benefits, while addressing

the technical challenges inherent to the volatile nature of spot instances. Spot instances can be suddenly interrupted with a notification period of up to two minutes. Applications that can tolerate the volatile nature of the spot instances can use them as an economical alternative to the on-demand ones. A third party or a broker leveraging resources from multiple cloud providers, and reselling them in a vertical market, has been found to be beneficial for both the broker as well as the cloud providers [219]. Dynamic bidding policies are developed to support deadline-constrained jobs in spot instances [363]. Temporal multiplexing of burstable instances (to have a constant higher availability of CPU cycles) and spatial multiplexing of spot instances (to have reliable connectivity with redundancy in the path) can be performed inside a single AWS region with minimal overhead [128]. However, no comprehensive study has been conducted to realistically determine the feasibility of a virtual ISP that leverages the resources of spot instances, as a regular cloud user.

2.1.4 Network Flow Scheduling

Network flows must be scheduled considering several parameters such as policies and QoE. Several algorithms and approaches have been proposed for an efficient network flow scheduling, by offering deadline-awareness and congestion control. Nevertheless, the existing network flow scheduling approaches limit their focus mostly to a single provider such as a data center or a cloud network. They should be extended to consider SLAs and QoS guarantees in multi-tenant environments such as the cloud and multi-domain environments such as the edge, consisting of several service providers and users.

There have been research efforts on network protocols to enhance the fairness, congestion control, and QoS of the critical network flows. D^3 is a congestion control protocol that provides a deadline-aware alternative to TCP for data centers [354]. Preemptive Distributed Quick (PDQ) [162] is designed to complete flows quickly and meet the deadlines in a fair manner by following a few pre-defined procedures, enhancing the flow completion time offered by TCP. ProgNET [315] leverages the Web Services Agreement specification (WS-Agreement) [23] and SDN for SLA-aware cloud networks. Dynamically rerouting the network flows to optimize the bandwidth consumption has been proposed in the previous work [9]. QJump [142] is a Linux Traffic Control module that allows critical latency-sensitive applications to jump the queues in the presence of packets of lower priority levels, focusing on a shorter flow completion time. pFabric [15] finds that the increase in the flow completion time of short flows is due to the waiting for long flows to complete. It focuses on optimizing the flow completion time for latency-sensitive short flows, practically ranging up to a few 10s of milliseconds. FastPass leverages a centralized arbiter to find the ideal time to determine when the packets should be transmitted and through which path [272].

Several network protocols exploit flowlets to propose new network flow scheduling approaches or enhancements to existing base protocols. Flowlets (also known as *subflows*) are defined as bursts or chunks of packets of a flow, which are separated in time from each other by an interval or a gap [181]. Partitioning of flows into flowlets enables routing the flowlets efficiently in multiple alternative routes. Conga offers congestion-aware load balancing for data

center networks through *flowlet switching* [14]. Multipath TCP (MPTCP) extends the Transmission Control Protocol (TCP) to use the available multiple paths between the origin and destination nodes to send a network flow [40]. MPTCP uses subflows in transferring data between the nodes using the multiple paths in a network and recomposes the original flow at the destination from the subflows. While MPTCP increases the bandwidth utilization of the network and its efficiency, MPTCP can be unfair towards the TCP clients in the network [187]. Opportunistic Linked Increases Algorithm (OLIA) is proposed as an enhancement to the fairness of MPTCP, making it fair and pareto-optimal [187]. These approaches enhance the resilience and fairness in network flow scheduling.

Redundancy is often exploited at higher levels across the network stack for reliable networks. It is argued that redundancy can be used more pervasively, to reduce latency in critical network flows and applications [339]. Advancement in network traffic monitoring [44], SDN [247], and middleboxes [341] makes it possible to leverage redundancy of flows at a lower network level. We posit that redundancy should be exploited in both network packets as well as the network routes to ensure that the flows with higher priority are scheduled on time.

2.2 Service Composition Workflows in Wide Area Networks

Next, we look into the related works on service compositions in the cloud and edge environments. We will look into the research on SDN and SDS for service compositions, and then look into how smart environments such as CPS and IoT leverage SDS to control and manage their execution.

2.2.1 Service-Oriented Architecture (SOA)

eScience workflows are often computation-intensive and require either a large pool of resources or a long time to complete their execution. Due to the distributed nature of the sensors, complex enterprise services such as weather forecast and disaster prediction frameworks [207] and extraterrestrial activity monitoring systems [21] are traditionally deployed on clusters of servers across the globe. Several computing nodes, typically executing web services or web applications, are invoked to compose such a workflow on the web. Many systems consume simplistic public services and APIs and extend the service invocation outcomes to produce a complex service composition. Mission critical workflows of eScience consist of redundancy in links and alternative implementations and deployments of web services and web applications, to handle failures, congestion, and overload in their computing nodes.

Web services are developed and deployed in middleware systems collectively known as web services engines such as Axis2 [271] and CXF [34]. Web services engines are capable of converting WSDL into high-level programming languages such as Java and C. This feature supports quickly developing web services following a standard API and format. Furthermore, the web services engines contain service health statistics including information on how many requests a service deployment executed, and how many are on the fly. Service composition workflows typically

exploit this information for the timely execution of each of the services that compose them. Thus, web services engines enable developing and deploying service compositions with minimal custom code developments and system admin efforts [139].

Web services registry is another middleware in the services ecosystem, which stores and manages the service endpoints in a centralized manner. Web services registry provides management and governance capabilities to web services, by maintaining a list of service endpoints and descriptions. It enables listing the Internet services efficiently for the users to discover and consume them. System administrators usually can retrieve the list of multiple web service deployments from a web services registry, and monitor the health of the service compositions, through the web services engines that host the services. Many specifications have defined and standardized the web services registry. Universal Description, Discovery, and Integration (UDDI) [86] offers a standardized directory structure as a registry of web services description. Distributed and effective web services registries are built to minimize the load on the registry, to avoid registry being a single point of failure. Ad-UDDI is a distributed web services registry, with an active monitoring mechanism [109]. Web services registries are a crucial aspect of service composition, by allowing the services to be discovered to compose the service workflows.

Recently, more and more services of the service composition workflows are built with frameworks other than the web services engines. While SOA has been at the forefront of the service composition [266], it is not uncommon to develop business processes and service compositions through other architectural paradigms such as Resource-Oriented Architecture (ROA) [358]. Solutions based on parallel and distributed frameworks such as MapReduce [97] and Dryad [171] can be more efficient and scalable at each service level, in replacing traditional service compositions. Top-k automatic service composition [98] exploits MapReduce [97] to compose parallel and efficient service compositions for large-scale service sets. Current web services engines and registries should be extended to consider this growing services ecosystem effectively.

2.2.2 SDS for Service Compositions

Network-aware service execution can improve the performance of the service composition workflows in a wide area network, in addition to interoperability offered by the web service standards [190]. The web services engines operate at the application layer and are typically oblivious to the network status. We propose that integrating SDN into the services ecosystem will enable network-aware service compositions, by offering a better perspective on the underlying network of the service instances to the web service middleware. Heterogeneous environments such as IPv6-enabled pervasive devices [317] and networks of wireless sensors and actuators use MOM protocol implementations for their communication and collaboration [167, 82]. Therefore, extending SDN with MOM increases the applicability and scalability of SDN beyond a data center. Thus, we posit that an SDS built by extending SDN for inter-domain networks with MOM will help to manage the service composition workflows in wide area networks efficiently.

MOM protocols and brokers continue to be a crucial aspect of the SOA ecosystem. MOM protocols are proposed as a communication channel for the enterprise middleware architectures [220]. Advanced Message Queuing Protocol (AMQP) [336], Message Queuing Telemetry

Transport (MQTT) [35], Simple / Streaming Text Oriented Message Protocol (STOMP) [317], OpenWire, and Extensible Messaging and Presence Protocol (XMPP) [292] are the common MOM protocols, implemented by the MOM brokers such as Apache ActiveMQ [309] and Apache QPid [27]. The MOM brokers allow messaging publishers to publish messages while letting the subscribers subscribe and listen to their interested subset of information published by the publishers.

Network softwarization facilitates context-awareness and traffic engineering capabilities in service compositions while enhancing service management and deployment [264]. The Next Generation Service Overlay Network (NGSON) specification offers context-aware service compositions by leveraging network softwarization [176]. Research efforts focus on efficient resource utilization as well as enabling pervasive services [208] motivated by the standardization effort of NGSON. Previous work has proposed data-aware and network-aware workflow scheduling in data centers and clouds [227]. However, they mostly narrow their focus to a given domain managed by a single vendor. Palantir [362] leverages SDN to optimize MapReduce performance with network proximity data, further highlighting the performance benefits of network softwarization in diverse application scenarios.

2.2.3 Network Service Chaining (NSC)

NSC (also known as SFC, Service Function Chaining) is a chain of network operations or middlebox actions created in a programmable and configurable manner [146], enabled by SDN [296] and NFV [369]. We can consider NSCs as a special case of service compositions, where each service in the composition is a VNF. While VNF APIs need to be compatible and interoperable with each other [288], the entire service workflow should be carried out in a network-aware manner for a high-performant NSC. The VNFs should be placed strategically to minimize communication overheads and the number of hops between the service instances during the execution of an NSC workflow. Each VNF in an NSC has its specific requirements, constraints, and objectives. Due to the various number of network services each NSC is composed of, challenges inherent to the VNF placement multiply in NSC scenarios [42]. Traditionally, NSCs have had a few services serially chained. However, the need for service migrations and mobility in the Mobile and Edge Computing (MEC) environments has given rise to network services as VNFs and microservices. This rising number of VNF offerings has further increased the scale and complexity in composing the NSCs [42].

Composing and scheduling NSC abiding by the user goals at multi-tenant edge and cloud environments is a challenging problem. The system and user goals can be complementary or even contradictory to one another. For example, one user's requirement for latency-aware execution may lead to a higher energy cost for the network, whereas the system might have a preference for low energy consumption. Ideally, a user should be able to construct her NSCs using VNFs from various providers to satisfy her own goals and policies. The research proposes VNF placement in distributed edge data centers considering various such heuristics [194]. However, seamless migrations and interoperability among the edge VNF providers significantly lack in the current enterprise services landscape. The previous research [36] mostly looks at the NSC scheduling as

an optimization problem from the perspective of a VNF provider and often limits its focus to a single provider environment. However, NSC at the edge has more dimensions than data centers and clouds, as edge environments consist of several nodes of different scales and numerous services, from multiple domains. Therefore, there are many open research challenges remain unaddressed, in supporting user preferences and policies efficiently for a user who consumes VNF resources from several third-party service providers to compose her NSC.

Network softwarization has enabled a fine-grained automated control of network services. It brings down the time to develop an NSC from the scale of hours or a few days down to the range of seconds or minutes [29]. Furthermore, leveraging network softwarization, system administrators may configure the networks to adhere to a specific set of policies, such as energy and carbon efficiency [226]. Light-weight container-based frameworks have been proposed for service provisioning with minimal service migration time [119].

Several solutions have been proposed for context-aware execution of applications at the edge, with focus on bandwidth, latency, and jitter [257]. NetFATE [214] offers a VNF-as-a-Service (VNFaaS), virtualizing network services of both Customer Premises Equipment (CPE) devices and Provider Edge (PE) nodes. Though NetFATE can be extended to support NSCs, it focuses on a network provider offering the VNFs, rather than facilitating user-defined NSC over third-party edge VNFs. The existing research limits its focus to the service providers. An NSC user has more constraints that are currently beyond her limit. Workflow scheduling approaches should be extended to consider the user policies and constraints in composing and placing the NSCs.

NSC orchestration frameworks focus on end-to-end management of the NSC lifecycle, consisting of several VNFs. Initiatives such as Network Service Orchestration (NSO) [58] and Lifecycle Service Orchestration (LSO) [234] aim at addressing the shortcoming in resource orchestration for workflows in the network environments. They unify the network services through standardization, to improve the interoperability of the services. ESCAPE is a scalable architecture for NSC orchestration, which leverages Mininet, ClickOS [228] virtualized software middlebox platform, NetConf configuration protocol [113], and POX, to support VNF implementation, traffic steering, and Virtual Network Embedding (VNE) [291]. OpenDaylight extends its controller to offer SFC as an incubation project. UNIFY programmatic framework performs NSCs in a data center environment based on OpenStack [300] and OpenDaylight through joint virtualization and control [111]. It presents an NSC control plane to integrate various VNF execution environments, supporting dynamic VNF creation and orchestration in cloud networks [311].

NSCs have been optimally provisioned across the selected VNF instances using Integer Linear Programming (ILP) [166] and Mixed Integer Linear Programming (MILP) models [327]. Using a Column Generation (CG) decomposition [100] along with an ILP, previous research minimizes the bandwidth consumption in the data center networks of NSCs [166]. This research has been extended concerning geographically distributed cloud environments to have minimal inter-cloud traffic and response time. An Affinity-based Approach (ABA) minimizes the total delays and resource cost in NSCs, solving the complex resource allocation problem faster than ILP in the distributed cloud environment [49]. Nevertheless, resource allocation for NSC continues to be a hard problem due to the ever-increasing service instances as well as the users in

the edge and inter-cloud environments.

2.2.4 SDS for CPS and IoT

CPS and IoT are two broad categories of systems that often implement the Multi-Agent Systems (MAS) paradigm [64]. While they both consist of autonomous agents making smart decisions, CPS focuses more on the integration and interaction of physical and cyber elements. On the other hand, IoT focuses on the connectivity of several pervasive entities. Therefore, while IoT offers a vision of the Internet, things, and semantics for smart devices, CPS looks at networking and interoperability of cyber and physical entities, incorporating human-in-the-loop. Large-scale industrial systems have been adopting CPS to replace their traditional stand-alone physical systems.

A CPS comprises numerous sensors or sensor-based devices that collect different types of data from various access points and autonomous systems with frequent communication among them [200]. Smart homes [245], Smart grids [184], smart cities [144, 261], Mobile Ad-hoc Networks (MANETs) [222], and Vehicular Ad-hoc Networks (VANETs) [361] are a few systems that are currently built as CPS. The complexity of CPS increases due to both the volume and variety of its components. The edge offers a compromise on bandwidth usage and resource availability between on-device or on-premise computation and computation in the cloud. Thus, several CPS, including smart campus [213] and connected cars [223], utilize the computing resources in the edge nodes as surrogates for the workload execution from their resource-constrained devices [269].

The state-of-the-art indicates the potential to leverage network softwarization to manage CPS environments more effectively. SDN has been proposed to improve the resilience of multi-networks in CPS [282], secure the CPS networks through SDN-assisted emulations [25], enhance the resilience of CPS [106], and manage the critical CPS communications [244]. These research efforts highlight the potential of SDN and how its global network awareness and controllability can be extended to model and manage CPS. However, currently, there is insufficient research on network softwarization that aims at mitigating the challenges of CPS regarding the management and control of its resource-constrained heterogeneous devices.

Several SDS frameworks have been built to manage physical environments such as smart buildings. Software-Defined Environment (SDE) [104, 205] consists of an SDN controller and physical and virtual SDN switches as its core, where the control of computing, network, and storage is built atop a virtualized network. By leveraging NFV [43], middlebox actions typically handled by hardware middleboxes such as firewalls and load balancers are replaced by relevant software components in an SDE. Software-Defined Building (SDB) [94] envisions a Building Operating System (BOS) that functions as a sandbox environment for heterogeneous device firmware to run as applications atop it. The BOS spans multiple buildings in a campus, rather than confining itself to a single building. The scale of SDB and SDE can be increased through collaboration and coordination of the controllers of the buildings or environments, to cater for CPS. However, the increased dynamics, diversity, and mobility of CPS compared to the environments controlled by SDB and SDE hinder adopting them for CPS.

An orchestrator hierarchy, consisting of an Overarching Orchestrator (OO) and a set of Domain Orchestrators (DOs), has been proposed, to control multiple domains and services. Each DO controls a specific technological domain by coordinating with the respective controller. The OO sits atop multiple domains, orchestrating the DOs while ensuring end-to-end service orchestration in SDN and cloud environments [51]. The 5G Exchange (5GEx) builds upon SDN and NFV for a Multi-domain Orchestrator (MdO) that spans multiple domains, technologies, and operators, for an automated and high-performance network service provisioning [301]. *SDCPS* [126] is an architecture with a hierarchy of multi-domain SDN controllers for CPS, similar to MdO. Both the *SDCPS* and MdO approaches require a hierarchy of SDN controllers, which is not common in practice due to the existence of several service providers with business policies that prevent them from sharing the network flow statistics and their control through a centralized global controller owned by a single third-party entity.

Albatross [203] is a membership service that addresses the uncertainty of distributed systems. Albatross tends to be ten times more efficient than the previous membership services by exploiting the standard interface offered by SDN to monitor and configure network elements. It addresses common network failures, while avoiding interfering with working processes and machines, and maintaining a quick response time and high overall availability. The challenges such as split-brain scenarios and violations in availability and consistency that are addressed by Albatross are relevant for CPS too. However, CPS has its peculiar challenges uncommon in a typical distributed system, starting from building the CPS to operating the CPS, due to its diverse nature in implementation and devices. Moreover, even though Albatross aims to address the uncertainty and difficulties in ensuring reachability and stability of distributed systems, it narrows itself to data centers equipped with SDN, leaving wide area networks as future work.

Software Defined Internet of Things (SDIoT) [174] is an SDS for IoT devices that handles the security [91], storage [92], and network aspects in a software-defined approach. SDIoT proposes an IoT controller that operates as an orchestrator between the Data-as-a-Service (DaaS) layer consisting of end user applications, and the physical layer consisting of the database pool and sensor networks. Software-Defined Industrial Internet of Things (SDIIoT) [342] extends SDIoT to Industrial Internet of Things (IIoT) [175], to offer more flexible and scalable networks. Various research and enterprise use cases are proposed and built, including SDIoT for smart urban sensing [211], and end-to-end service network orchestration [333]. Multinetwork INformation Architecture (MINA) self-observing and adaptive middleware [283] has been extended with a layered SDN controller to implement a controller architecture for IoT [281]. While sharing similarities with IoT, CPS addresses a broader set of problems with more focus on ground issues on interoperability and interactions between cyber and physical dimensions. Hence, an SDS approach for CPS needs to address more demands and constraints inherent to CPS.

2.3 SDS for Big Data

Finally, we look into the data services and big data workflows that lead us towards SDS for big data applications.

2.3.1 Software-Defined Data Services (SDDS)

SDN has been extended for networks beyond data centers, and to manage big data workflows. The research findings on leveraging SDN for service compositions [264] are not optimized for big data workflows, as they focus more on the standard web services. However, enterprises such as IBM [89], NEXION Networks [249], Catalogic [72], SAP [294] and Oracle [255] provide network-aware data services as part of their middleware frameworks, by bringing a software-defined approach to data services. These SDS frameworks for data services are collectively known as SDDS. SDDS is typically a data service execution model to schedule the big data workflows in a centralized and unified manner while abstracting the execution of data services away from the data storage. SDDS relies on the ability to express big data applications as composable data service workflows and to control them with an extended SDS architecture. Thus, SDDS brings the benefits of SDN to data services execution.

The SDN controller is capable of managing network flows, including flows with a large volume of data (known as *elephant flows* [88]). The controller in an enterprise data center is often a physically distributed cluster of high-performance servers, albeit being logically centralized [103]. Furthermore, despite the volume, scale, and variety of the data flow, the controller is communicated only for the control flows. Data flows of the data services are transmitted entirely at the data plane consisting of the switches and servers, as in any regular network flows, without frequently invoking the controller. Therefore, the controller manages network flows of big data efficiently [153] without causing a bottleneck or becoming a single point of failure due to resource scarcity.

Several enterprise SDDS offerings focus on a single data service or a set of data services. Portworx open source SDDS framework follows a container-based approach for dev-ops [277], by exploiting Kubernetes [48], Mesos [159], and Docker [237] swarms. RedHat SDDS offers storage orchestration enabled by full lifecycle management consisting of provisioning, installing, configuring, tuning, and monitoring the data [328]. PrimaryIO APA is an SDDS storage solution with a higher performance achieved through virtualization and intelligent caching [141, 278]. HPE SDDS aims to offer scalable, secure, and highly available virtualized storage solution [302]. Commvault SDDS mitigates vendor lock-in through its service-based data access and Software-Defined Storage [236]. It further offers backup, archival, and recovery, with performance, scalability, and agility [83]. Thus, Commvault aims at offering performance regardless of the physical location of the data in the data center. Other enterprise offerings include SDDS-based big data storage solutions such as PureStorage [279], and Software-Defined Copy Data Management (CDM) frameworks such as IBM Spectrum [169] and Catalogic ECX [72] that centrally orchestrate how data is copied and shared among the users of various data sources.

Despite the recent surge in the enterprise offerings, current SDDS frameworks have several limitations for generalized large-scale big data executions. First, they have a limited focus. They focus only on one or just a few specific data services rather than providing a generic data services framework. Many SDDS offerings limit their attention to storage, or a Software-Defined Storage [92, 69], rather than focusing on service execution. Second, they focus entirely on a single-domain network such as their cloud or data centers, rather than SDDS executions

at Internet scale. Third, they also lack interoperability between execution frameworks or across multi-domain wide area networks, due to their closed vendor-specific implementation. The executions are limited to the data centers and are confined to a single big data application due to the interoperability constraints. Fourth, they do not leverage the existing research efforts on SDS and SD-WAN to offer network-aware service workflows efficiently. These limitations highlight the need for a comprehensive network-aware and interoperable SDDS framework at Internet scale.

2.3.2 Interoperability in Data Services

Interoperability is a significant challenge concerning data access and integration across heterogeneous data sources. Extracting and transforming data from web sources must consider various data storage and access interfaces. Data storage formats and access interfaces have been standardized across multiple research fields, to facilitate seamless access to heterogeneous data sources. For example, Health Level Seven International (HL7) Fast Healthcare Interoperability Resources (FHIR) [161] is a standard for consistent data exchange between healthcare applications. Enterprise Service Bus (ESB) [73] are middleware frameworks that facilitate communication between various enterprise APIs and SOA platforms, mitigating their incompatibility. There have been proposals to exploit ESB and SOA for big data systems [256]. However, a majority of data sources fail to adhere to these standards, resulting in a significant lacking of interoperability between the data sources [179]. Consequently, data integration across various scientific web data sources is challenging, and typically not effective and efficient without human involvement.

Data services offer the best of both worlds from web services and big data frameworks to the execution of big data workflows in wide area networks. While web services are developed following standards and descriptive languages such as WSDL as well as protocols such as REST and SOAP, big data execution frameworks lack such practices and protocols. Data services offer service-based APIs to big data storage and execution frameworks and provide a standardized, interoperable execution model across various frameworks. A data service can be a simple web service that processes big data or a composition of several data services each performing simple data executions such as data access, integration, transformation, and updates. Their interoperable APIs facilitate composable data service chains, where the output of one or more data services is fed as the input for one or more data services. This potential for service compositions supports extensible and reusable big data workflows, bringing the advantages of SOA to big data.

Current web services registries lack network-awareness at Internet scale and are not optimized for data services. Data services and web services are developed and deployed independently at several locations that are neither directly connected nor centralized. Finding the service deployments and chaining their execution outcomes for a large-scale workflow is impossible without prior knowledge of their existence and access mechanisms. Web services registries [326] such as UDDI and Electronic Business XML (ebXML) [105] are developed to identify and store the web services engines and service instances to enable easy discovery of the service endpoints.

However, their focus is limited to service discovery. Therefore, we cannot entirely rely on them to find the service instances to compose data service workflows in a latency-aware manner.

Data service frameworks enable efficient data access, integration, and sharing of heterogeneous big data. OGSA-DAI (Open Grid Services Architecture - Data Access and Integration) [24] facilitates federation and management of various data sources through its web service interface. The Vienna Cloud Environment (VCE) [53] offers service-based data integration of clinical trials and consolidates data from distributed sources. VCE offers data services to query individual data sources and to provide an integrated schema atop the individual datasets. EUDAT [197] is a platform to store, share, and access multidisciplinary scientific research data. EUDAT hosts a service-based data access feature B2FIND [352], and a sharing feature B2SHARE [28]. These frameworks significantly enhance the interoperability of big data storage and executions through their service-based approach. Given the potential of SDN and its extension to control multi-domain wide area networks [276], we posit that big data workflows can execute in a decentralized and distributed environment spanning data centers, by extending and leveraging SDN for managing the decentralized data services.

2.3.3 Network-Aware Big Data Workflows

Although several research efforts provide network-aware service compositions in wide area networks [190], their application to big data workflows is significantly lacking. Vivaldi [90] leverages network coordinates in estimating latency between two endpoints. While Vivaldi and similar approaches have been used to predict latency in service composition workflows at Internet scale, their use in a multi-domain network consisting of several independent networks is mostly limited. Models based on game theory have been proposed to understand dynamic service placements in geo-distributed cloud environments comprised of multiple service providers [368]. Similarly, significant research efforts have been made to identify an optimal service and data placement in wide area networks that are shared by several users, considering the dynamic availability of the resources [254]. Previous work has proposed storage efficiency through efficient data placement, to minimize data migration [356]. NetMIP considers network properties and resource consumptions in the cloud network nodes for service compositions, rather than merely addressing the QoS of the services as a static property [349]. It presents optimization problems to maximize the QoS utility value while minimizing network resource consumption. NetMIP poses the optimization problems as Mixed Integer Programming (MIP) problems that can be resolved by commercial and research MIP solvers such as Gurobi. Nevertheless, NetMIP and similar research projects limit their focus to typical computation-intensive service composition workflows and do not cater for data-intensive workflows in multi-domain networks.

Researchers propose solutions such as middlebox-based aggregation, efficient computation of aggregation, and allowing queries to reroute the data flows without impacting the existing queries, to support network-aware big data processing [289]. Fine-tuning the data placement in the distributed data stores inside and beyond data centers have attracted a considerable research interest [265]. DIANA offers network-aware scheduling for data-intensive workloads in grid environments [230]. DIANA is more efficient than pulling data to the execution node in

computation-intensive workloads or pushing the execution to the data in data-intensive workloads, in a network-agnostic manner. However, unlike the grid and cloud environments, inter-cloud and edge environments are managed by multiple providers. Big data workflow execution across services spanning such multi-vendor multi-domain environments need to consider additional parameters such as the possible interconnection between the network domains and the high-latency network links (typically, the public Internet routes) that connect the domains.

Recent developments in SDN and SD-WAN [173] have opened up the potential for network-aware service compositions in heterogeneous network environments. While geographical proximity is one deciding factor in picking the service composition for the workflows, the existence of dedicated connectivity between two geographically distributed servers also plays a decisive role in finding the best instances. For example, cloud providers such as AWS offer Cloud Direct Connect services between the server of a user and the data centers of a cloud region, typically the nearest to the user server. Similarly, MPLS [93] providers connect the enterprise users to their remote servers with high bandwidth, without having to go through the public Internet, which is typically slow and thus hinders the performance of the organizational workflows between the distributed locations. These options have increased the potential for latency-aware big data workflows at inter-cloud and edge environments.

Direct connect services that offer the Internet fast paths [193] are getting more mainstream for data-intensive workflows. Cloud providers such as AWS leverage their backbone to route their entire data traffic, without relying on external Internet paths or other connectivity providers, in most of the cloud regions. While such networks are readily available for each provider, currently they are not shared across the providers or third-party users. Even though ISPs and Internet eXchange Points (IXPs) [5] are more equipped with the potential to alter the network paths across the autonomous systems and differentiate the network flows, social aspects of the Internet and net neutrality regulations prevent them from implementing these research avenues globally, without hindering a large segment of the existing end users. A more user-centric approach in composing network-aware big data workflows would solve these challenges in offering a differentiated QoS [295] and SLA guarantees, without having to alter the Internet ecosystem significantly.

2.4 Discussion

The management capabilities offered by SDN can be extended to the cloud and edge environments, as demonstrated by SDS frameworks. An SDN controller is a software developed in a high-level language that is capable of controlling the underlying network. Thus, it opens up opportunities for cross-layer optimizations and more fine-grain control of the underlying network from the user applications. Recently, SDN has been extended for wide area networks and used for use cases other than network management and traffic engineering, such as latency-aware execution of big data applications. Therefore, we posit that workflow scheduling at Internet scale can be optimized in terms of performance, management, and user policy awareness by extending and exploiting SDN.

Edge data centers offer the benefits of proximity typical for the on-premise deployments, while still providing the separation of infrastructure from the applications of the user as in a cloud platform [305]. On-premise deployment of VNFs offers bandwidth-efficient execution for interactive I/O bound applications. Nevertheless, configuring all the VNFs in-house is not feasible for everyone due to the deployment and maintenance costs as well as the resource limitations. For example, thin IoT clients do not have the resources to host their VNFs in them due to resource scarcity. Therefore, several edge providers [2, 340, 110] have started to offer numerous network services (such as captcha verification [10] and firewalls [1]) in addition to the infrastructure, close to the end users. This increasing reach of edge environments has created a demand for better control of multi-domain environments.

Constructing workflows in a multi-domain environment faces several challenges, including aspects of data management and secured interoperable access across the service instances. Service composition workflows require chaining of services spanning multiple data centers and nodes of various scale, offered by different providers from numerous network domains, with the need to abide by the user policies. Each service in a workflow has its specific requirements, constraints, and objectives. The volume and variety of services, coupled with these various constraints, make service composition and workflow placement at Internet scale a multidimensional optimization problem that has not been studied well yet.

SDN should be extended with MOM and web services registries for context-aware service compositions at Internet scale, to consider multiple network and service providers. While SDN offers a global view of the network in a data center environment, such a unified view of a wide area network may not even be feasible to achieve for a single central controller due to the organizational policies and challenges in scalability. An inter-domain controller deployment with multiple controllers is necessary to cater for this scale and segregation of the network, without assuming the presence of a central authoritative entity in the multi-domain wide area network. Thus, we posit that SDN controllers of different domains should have a limited level of access to the controllers from other domains/vendors to support communication and coordination across various service providers. MOM supports such protected access to internal data belonging to the data plane devices controlled by one another, based on a subscription-based configuration rather than a static topology. SDN controllers are necessarily software applications that handle the events and notifications from the data plane devices. Therefore, they can be extended to communicate through MOM messages between one another. Furthermore, we should extend and exploit web services registries together with the controller deployment for service discovery and service composition in a QoS-aware manner, to reap the performance and flexibility benefits of the several service instances.

Currently, big data frameworks have much larger throughput than the SDN controllers to cater to the volume of the data that they process. Research and enterprises have proposed extending big data frameworks with SDN approaches. To enable managing big data with SDN, we should identify the differences and challenges among the big data workflows and the SDN control flows. Real-time big data streaming engines boost millions of messages/s. For example, Apache Gearpump [130] reports throughput of 18 million messages/s with an 8 ms latency on a 4-node cluster. However, such scale and concurrency of data is typical for data plane, and not

for the SDN controller as the network control plane handles only for the first packet of the first flow. Moreover, end-to-end control of the network service workflows is a more involved process than big data streaming, thus impeding the controller throughput. However, we foresee that a more involved SDN-based management framework for big data flows will require enhanced controller architectures to handle the demands of big data.

In this document, we propose SDS frameworks for service composition and workflow placement at various scales of networks, by extending SDN to address the challenges that are not yet addressed in the state-of-the-art. Specifically, we research and build SDS for i) an integrated modeling and deployment of cloud networks, ii) decoupling the network connectivity from their underlying infrastructure, iii) exploiting SDN and middleboxes for resilience in critical network flows, iv) architecture and algorithms for composing service chains in wide area networks abiding by the user policies, v) enhancing the interoperability of diverse CPS applications by expressing them as service workflows, supported by network softwarization, and vi) big data executions with network-awareness and minimal data movements, via an SDDS approach. We first propose our network softwarization approach in a data center environment, then extend our SDS to the cloud and edge environments, and finally present its use cases in CPS and big data executions at Internet scale. The following chapters elaborate in detail our contributions on these domains.

II Network Softwarization

3 Incremental Development of Cloud Networks

Cloud networks and architectures are tested over simulation, emulation, and physical deployment environments, as the required accuracy and precision differ across various stages of development. Many network algorithms are tested for their functionality and efficiency in a simulation environment, before deploying them on the actual physical execution environment. While a simulator can effectively estimate some parameters of the evaluated network architecture, an emulator or a physical test environment will reveal more accurate insights. On the other hand, network emulators often create several separate processes on their host machine to represent the emulated system comprised of network switches and hosts more precisely, resulting in high resource consumption, thus hindering the emulation of enterprise networks in research test environments with limited resources. These factors highlight the necessity for both simulations and emulations to coexist for continuous development of network applications.

Current network simulators do not support seamlessly migrating a workload from simulation to emulation or vice versa, thus resulting in repetitive developments of simulations and emulations followed by manual deployments. Network emulations are compatible with cloud and data center networks, thanks to their native integration with SDN controllers. Thus, the emulators provide a more realistic resemblance to the physical network deployments. As Mininet emulates the virtual switches and flows, the emulated network can be dynamically monitored and managed through an SDN controller connected with the emulator. However, existing network simulators lack such integration with the SDN controllers, thus preventing a seamless migration across the simulators and emulators. Furthermore, in software development and testing process, automation and management tools focus on either deployment aspects or early development stages. The configuration management tools automate the deployment across different infrastructures, reducing the manual efforts of DevOps [313]. However, they focus only on product deployment aspects and do not manage the early design and development configuration efforts. On the other hand, the modeling frameworks that focus on initial prototyping lack automation capabilities. Therefore, currently, repeated development and manual deployment efforts are inevitable across simulations and emulations.

Incompatible interfaces of the modeling frameworks limit the use of modeling frameworks despite their availability. Modeling frameworks such as simulators and emulators are developed independently, typically by different communities. They often consist of incompatible APIs and different development language and syntax. Therefore, porting simulations to emulations or physical environments of varying system properties is a tedious undertaking. Such migration also requires extensive code changes to the user algorithm or even a rewrite of the majority of the code, with a manual redeployment over the environments. Since learning and using the simulators and emulators require a considerable investment of time, developers are often forced

to choose one over other, rather than choosing the best option for a given development phase.

Research efforts are looking into minimizing the development efforts needed in modeling and migrating the simulations and emulations. EmuSim is an integrated cloud simulation and emulation environment [62], offering evaluation at different granularity and accuracy. However, unlike Mininet, EmuSim emulations cannot seamlessly be ported into enterprise cloud networks, and cannot be used to evaluate the production-ready algorithms in a physical deployment. We propose that a simulator that abstracts away the architecture and policies from the network topology and implementations by following the paradigm of network softwarization would resolve this shortcoming in the modeling of cloud networks.

This chapter presents *SENDIM*[1], a **S**imulation, **E**mulation, a**N**d **D**eployment **I**ntegration **M**iddleware for cloud networks. *SENDIM* manages the development and deployment of algorithms and architectures the entire length from visualization, simulation, emulation, to physical deployments. *SENDIM* coordinates heterogeneous cloud network systems and manages their deployments, by extending the principles of SDN. *SENDIM* offers a unified modeling and management process for building, analyzing, and migrating efficient cloud network architectures with different topologies, design dimensions, and deployment environments. *SENDIM* abstracts configurations and logic away from the application to provide seamless migration across multiple environments and different realizations, such as simulations and emulations. *SENDIM* provides a **Software-Defined Cloud Deployment (SDCD)**, by automating deployments and migrations across different platforms and infrastructures. We foresee a shorter time to deliver as *SENDIM* minimizes the development overhead caused by incompatible APIs, programming languages, and granularity among the simulators and emulators, as well as the administrative overheads caused by the deployment migration.

SENDIM consists of an efficient simulator for continuous development of cloud and data center networks. In addition to providing a separation of concerns to the simulations, *SENDIM* leverages the logically unified storage space in the control plane to store the status of the ongoing simulations dynamically. The state of the simulations can be leveraged to pause and resume long-running simulation workflows or execute a large complex cloud network simulation consisting of various smaller simulation tasks. As the simulated network system is separated from the network application logic, the simulated application logic can be seamlessly scaled or migrated. *SENDIM* is efficient in a continuous simulation of cloud networks through its incremental development and deployment. It offers smooth scaling with problem size and the host environment. It further supports a seamless migration of simulation to emulation through its integration with the SDN controller. Evaluations on the *SENDIM* prototype highlight its scalability with network size.

This chapter is composed of the contents of the publications: [C3, W7, S2].

[1]Sendim is a northeastern Portuguese town close to the Spanish border, where the rare Mirandese language is widely spoken.

3.1 *SENDIM*: Software-Defined Cloud Deployments

An SDCD framework requires a unified control plane, where simulations and emulations, as well as migrations among them, are configured by control plane software, rather than by an entirely manual process. In this section, we will look into *SENDIM* and how it provides SDCD.

SENDIM **Simulation Sandbox** is a simulator designed focusing on the SDN integration and interoperability with the emulators. Network emulators possess compatible and interoperable executions with the physical deployments. Therefore, the major challenge in an end-to-end development and deployment process relies on bringing the network simulations closer to the emulations. The current simulators do not possess the capability to integrate with the SDN controllers in an approach compatible with emulators and physical networks. We built the simulation sandbox from scratch for *SENDIM* as its core module, rather than extending or integrating an existing simulator, due to the limitations of interoperability with the network emulators in the current network simulators.

The simulation sandbox splits the conventional simulation logic into: i) The cloud or data center network system, simulated in the *SENDIM* simulation sandbox; and ii) The application plane deployed directly on top of the controller environment northbound, which invariably executes in simulation, emulation, or a physical deployment. Various network scenarios such as load balancing, network resource allocation and sharing, and bandwidth throttling, can hence be implemented and deployed into the controller as extensions while the execution network environment is simulated in the simulation sandbox. The separated and unified application logic in the controller instances allows reuse of code across simulation, emulation, and physical environments while facilitating the changes in the scale and simulated system without changing the underlying logic that has been simulated.

Emulators are unable to model and execute larger networks due to resource limitations. On the other hand, SDN support and integration with the controllers is typically limited in the existing simulation environments. The *SENDIM* simulation sandbox brings the best of both worlds into the cloud network simulations, including the SDN awareness and seamless integration of emulators, as well as the minimal resource demands of the simulators.

The simulation sandbox executes the application logic on the simulated system, offering measures to represent the system properties such as the processing power of the servers and the bandwidth of the network. While emulated networks perform an actual execution of the algorithms over virtual switches and hosts, the *SENDIM* simulation sandbox simulates the execution by representing the networked system using relevant Java objects, rather than actually executing the application across switches and hosts. Hence, *SENDIM* can simulate and visualize a network of hundred thousands of nodes from the controller in a similar manner to the emulated networks.

The state of the simulation is stored and distributed in an In-Memory Data Grid (IMDG) [129], as the simulation progresses in the simulation sandbox. In a complex simulation workflow with multiple discrete steps, when the simulation halts, the simulation sandbox stores the state of the simulation in a distributed map in the IMDG cluster. Hence, when the

simulation resumes, it reads from the map and starts from where the simulation previously stopped. Storing of the simulation state in an IMDG cluster also offers breakpoints to recover a simulation when a failure occurs amid the execution. As the simulation workload increases, more data grid nodes are spawned adaptively, increasing the number of IMDG instances in the simulation cluster. *SENDIM* also leverages the information of state, when available, to determine the necessity to scale itself in the IMDG cluster to accommodate more complex simulations. For example, a simulation that is in its final stages may not require additional instances, and hence, no new *SENDIM* instances will be spawned in the cluster. Thus, *SENDIM* aims to reuse of simulation elements in building up a larger and more complex system, while adaptively scaling to accommodate complex simulations.

Figure 3.1 illustrates the *SENDIM* workflow, indicating both the physical and simulated entities in a single diagram. The *SENDIM* Descriptors define the data plane consisting of either i) simulated virtual nodes and links or ii) physical, virtual, or emulated data plane devices. The application plane contains the algorithms to be executed or simulated in the network environment as functional execution blocks. The algorithms can be streamed to each other to represent a series of procedures or services as a workflow such as an NSC.

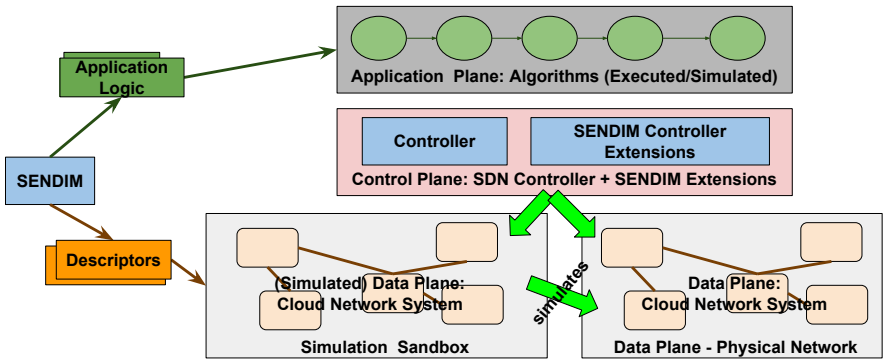


Figure 3.1: Separation of the Application Logic From the Execution Environment

The control plane consists of the SDN controller as well as the *SENDIM* controller extensions. The controller extensions ensure that the applications can be executed on the simulated data plane similar to a physical or an emulated data plane. The simulated data plane assumes a passive role, while the control plane ensures that the simulations in the application plane are executed on the data plane. The connectivity between the controller and the simulator mimics the communication and integration of an emulator with an SDN controller.

Anatomy of a *SENDIM* Application: *SENDIM* separates a simulation into the core application logic and the system architecture to be simulated, to facilitate the continuous development of cloud networks, including the simulation phase. Following a component-based software engineering approach [158], *SENDIM* lets the user define the application logic (such as the load balancing or resource allocation algorithms) as Java bundles and deploy them into the SDN controller.

SENDIM develops each component of the execution as a separate function, to support chaining and reusing the simple simulations into complex ones. It thus represents a complex

simulation as a function of functions, consisting of simulation blocks **s** and their respective list of input parameters **p**. Equation 3.1 illustrates a simulation **S** consisting of the simulation function **s3** which takes as its input the outputs of the several simulations, including **s1** and **s2**. **s1** and **s2** respectively have the inputs of **p1** and **p2**. **s1** and **s2** are independent and can run in parallel, while **s3** must wait for the completion of their execution as it depends on their execution outcome.

$$\mathbf{S} =< \mathbf{s3}, (< \mathbf{s1}, \mathbf{p1} >, < \mathbf{s2}, \mathbf{p2} >, ..) > \tag{3.1}$$

Figure 3.2 shows the anatomy of a *SENDIM* application. A network application is composed of the application logic as well as the **Descriptors** in *SENDIM*. *SENDIM* represents and models the cloud network systems through its Domain Specific Language (DSL) defined in XML (Extensible Markup Language). Cloud systems and deployments are expressed in *descriptors*, the XML configuration files created abiding by the *SENDIM* DSL. The descriptors define the syntax for expressing the application parameters and deployment details that can be separated from the applications. The relevant parsers parse the descriptors in the *SENDIM* middleware. Similar to the physical or emulated SDN networks, data is transferred across the simulated data plane entities, with control flows between the simulated data plane and the SDN controller.

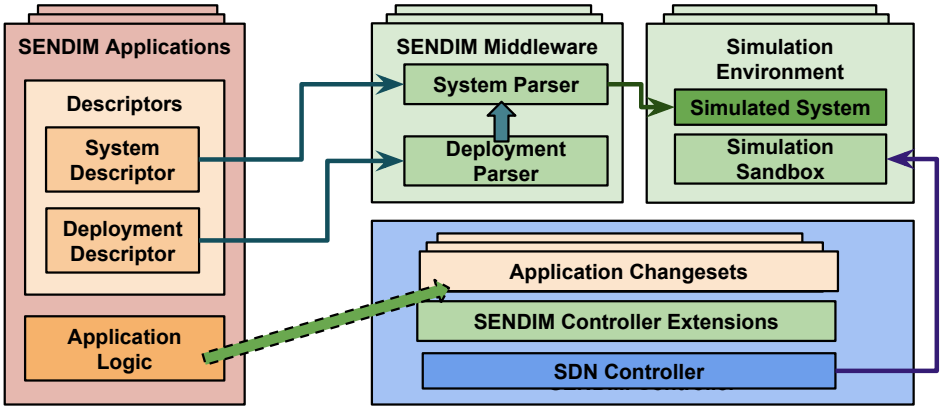


Figure 3.2: *SENDIM* Middleware and Applications

The Descriptors consist of a **Deployment Descriptor** and a **System Descriptor**. The deployment descriptor defines the system the application should be deployed on, including the physical or emulation environments as well as the controller endpoint. The system descriptor defines the system to be simulated (such that the application logic can be executed on top of it) in the simulation sandbox in case of simulations. *SENDIM* provides expressiveness to simulation developments from the initial design, by letting the users compose systems with different topologies and nodes and flows in the network, by defining them in the system descriptors. The system descriptors define the system's static properties and dynamic properties. Static properties define the nodes and links composing the network, as well as the properties of these nodes and links. Dynamic properties are the properties that change frequently, such as the flows on the network and their properties. A basic definition of a switch is shown below with the neighbors defined.

```
<node id="s1" type="switch">
```

```
        <link id="s2"></link>
        <link id="s5"></link>
</node>
```

The granularity of the network flows can be set to flow, packet, or an intermediate level where the flows are composed of several subflows and time intervals between them. A basic flow definition is shown below, along with various policies for the flow.

```
<flow id="F1">
    <origin>h2</origin>
    <destination>h5</destination>
    <intent type='decrement'><energy priority='2'></energy></intent>
    <intent type='increment'><throughput priority='1'></energy></intent>
</flow>
```

SENDIM deploys the incremental developments of the application logic into the SDN controller as **Application Changesets**. The application changesets are supported by the modular architecture of the controllers, natively supporting an incremental development. Frequent code changes are minimized as only changing the configuration files is sufficient to change the deployment or the dynamic properties of the system, and code changes are necessary only when the application logic is changed.

3.2 *SENDIM* Algorithms

SENDIM consists of two major operations: the initialization of an application, and the incremental updates.

System Initialization: Each application has an Initiator workflow which deploys the initial version of the application into the controller and optionally simulate or emulate the system in the specified environment. It is also possible to deploy and execute the application into multiple environments at once using *SENDIM* deployers. The entire application is deployed into the controller for the initial or first deployment of the algorithm. Subsequent iterative deployments only need to deploy the changesets. Algorithm 1 presents the pseudocode of the application initializer workflow. The background color in the algorithms presented in this document represents the nature of the execution. Red represents conditions, blue represents data manipulations, and green represents the other computing executions.

The algorithm accepts as input, the descriptors, the controller endpoint, and application bundle references. Initially, the execution realization (i.e., a simulation, an emulation, or a physical deployment) and the endpoint of the deployment are identified by parsing the deployment descriptor (line 2). Then the application bundles are deployed into the controller instance (line 3). If the deployment environment is identified as a physical network, the network application is executed in the deployment environment (line 4 - 7). While the network flows are executed as they are in physical deployments, emulations and simulations first need to construct the virtual networked systems before executing the network flows on them. If the execution is a simulation, the simulation sandbox is built from the system descriptors (line 9). The application is then executed in the network environment simulated in the simulation sandbox (line 10 - 12). In

Algorithm 1 *SENDIM* Application Initialization

```
1: procedure DEPLOY(deplDescriptor, sysDescriptor, controller, appBundles)
2:     realization ← parse(deplDescriptor)
3:     deploy(controller, appBundles)
4:     if (realization.env.isPhysical()) then
5:         while (TRUE) do
6:             execute(realization.env)
7:         end while
8:     else if (realization.env.isSimulation()) then
9:         simulationSandbox ← construct(realization.env, sysDescriptor)
10:        while (TRUE) do
11:            execute(simulationSandbox)
12:        end while
13:    else if (realization.env.isEmulation()) then
14:        emulator ← realization.env
15:        descriptor ← convert(emulator, sysDescriptor)
16:        construct(emulator, descriptor)
17:        while (TRUE) do
18:            execute(emulator)
19:        end while
20:    end if
21:    for all (appID ∈ controller.getAllExecutionIDs()) do
                                ▷ Clear all the distributed objects of the executions in the controller
22:        clearDistributedObjects(appID)
23:    end for
24: end procedure
```

the case of emulations (line 13), first, the emulator endpoint is identified from the deployment descriptors (line 14). In the case of the emulations, the system descriptors are converted to have the script for the respective emulator (line 15), whereas the simulations execute natively inside *SENDIM*, based on the system descriptor. Once the emulation environment is configured based on the descriptors (line 16), the application is executed in the specified emulator environment such as Mininet (line 17 - 19). Finally, when the controller terminates, all the executions that are either started at the time of initialization or later should be stopped. Therefore, all the relevant distributed objects are cleared when the execution terminates (line 21 - 23).

Incremental Development: *SENDIM* creates the revisions or updates to the applications as changesets and updated bundles, which can be deployed in the application hosted in the respective environment. As development progresses, the changesets are deployed in the controller environment and application is thus tested incrementally. Algorithm 2 presents the subsequent deployments of an application that has already been initialized or deployed in the controller.

A unique identifier *appID* is created and stored for each of the application that is part of a workflow. *appBundles.getOne().getID()* retrieves the ID *appID* (line 2) of a bundle from the appBundles. The appID is unique and the same for all the bundles that are part of a single application workflow. Thus, appID identifies the workflow from the bundles, by getting the ID

Algorithm 2 Iterative and Incremental Development

```
1: procedure UPGRADE(deplDescriptor, sysDescriptor, controller, appBundles)
2:     appID ← appBundles.getOne().getID()
3:     stopExecution(appID)
4:     clearDistributedObjects(appID)
5:     getAndUpdateState(appID)
6:     changeSets ← constructChangeSets(appBundles)      ▷ Deploy the Δ, i.e., only the incremental changes
7:     deploy(deplDescriptor, sysDescriptor, controller, changeSets)
8: end procedure
```

of any random bundle from the given application workflow. The execution is generally halted when the application is redeployed. If the application is the only execution on the system, and the controller does not manage any other systems simultaneously, the controller can be simply restarted during this. However, in an environment with multiple simulations or executions and multiple networking systems, upgrades occur in a tenant-aware manner. Here, only the upgraded application is paused using *stopExecution()* (line 3) and resumed by clearing only the distributed objects and state associated with the application using *clearDistributedObjects()* (line 4). *getAndUpdateState()* in line 5, gets and updates the state of the execution block to enable state-aware executions. Changesets of the application since its last deployment are computed and built, using *constructChangeSets()* (line 6). Finally, the application changesets are deployed into the environment consists of the controller and the physical or virtual network, using *deploy()* in line 7. *SENDIM* thus provides a hot deployment [170] with minimal downtime.

3.3 Implementation

We developed *SENDIM* as an OSGi bundle to seamlessly integrate with the OSGi-based frameworks such as OpenDaylight controller, and dynamically deploy the algorithms built on top of *SENDIM* into OSGi-based controllers as bundles without changing their code. We leveraged OpenDaylight Beryllium as the default controller due to the modular architecture of OpenDaylight. *SENDIM* uses the OpenDaylight interfaces and programming constructs to connect with OpenDaylight as a simulation component that can utilize the control functionality of OpenDaylight. *SENDIM* uses Oracle Java 1.8.0 as the development language, Apache Karaf 3.0.3 as the OSGi runtime, Mininet 2.2.1 as the default emulator, and Apache Maven 3.1.1 to build the application logic. *SENDIM* simulation sandbox integrates and interacts with the OpenDaylight controller through the OpenDaylight southbound interfaces to programmatically simulate Open vSwitch OF13 switches. We implemented OpenDaylight MD-SAL extensions to deploy the applications as OSGi bundles and execute them over the simulated network. We defined the higher-level northbound abstraction of *SENDIM* applications that are deployed in or integrated to OpenDaylight through YANG [50] data modeling language for the Network Configuration Protocol (NETCONF) [298]. *SENDIM* leverages an IMDG cluster of Infinispan 7.2.0 [225] for a state-aware distributed adaptive scaling of its simulations.

Figure 3.3 illustrates the *SENDIM* architecture and a higher level deployment view, along

with the cross-platform deployment migrations. The *SENDIM* interfaces include northbound and southbound APIs. The northbound APIs communicate with the applications and algorithms deployed on top of *SENDIM*, whereas the southbound APIs interact with the SDN controller southbound protocols such as OpenFlow. *SENDIM* connects with the emulators and controllers respectively through the Converters and the controller extensions. The **Emulator Converters** automatically translate the descriptors comprised of the topologies and flows defined in *SENDIM* DSL into the relevant emulator scripts if the user indicates an emulator as the execution environment for her applications. For example, the *SENDIM* Mininet Converter converts the *SENDIM* DSL into Python scripts for Mininet emulations, and the Mininet instance identified by the deployment descriptor emulates the network and the algorithms.

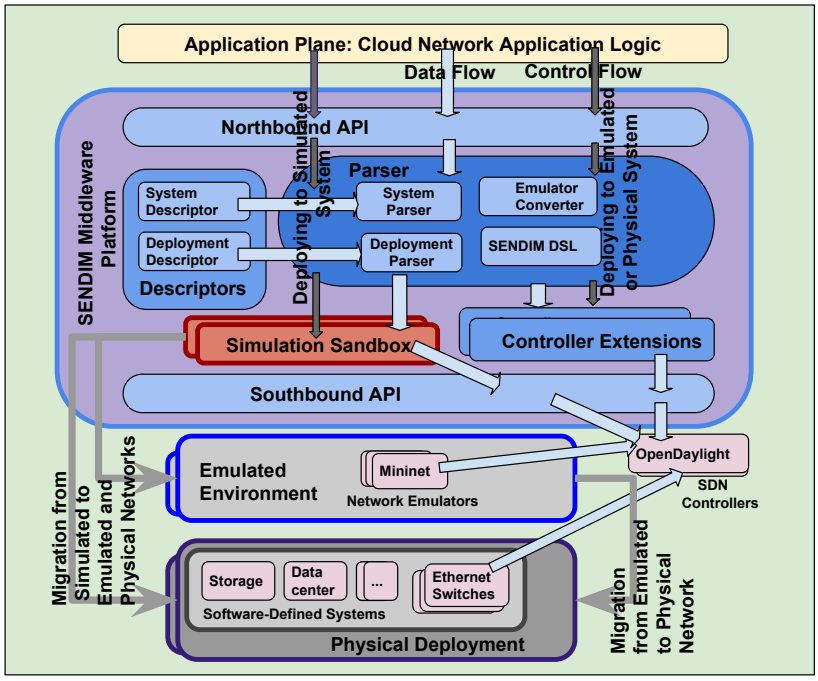


Figure 3.3: *SENDIM* Architecture and Deployments

SENDIM controller extensions enable interaction between the controller and the simulation sandbox, and thus facilitate the simulation of the SDN data plane. The simulation sandbox simulates the system and executes the algorithm on the simulated system when the *deployment descriptor* indicates a simulation environment as the deployment platform. The simulation sandbox can operate as a stand-alone simulator or an OpenDaylight module deployed as an OSGi bundle into the Apache Karaf container of OpenDaylight. An SDN controller can manage the simulated SDN system via its southbound API. The simulation sandbox is kept light-weight with an easily extensible API to model networks that are too complex or resource-heavy for an emulator. *SENDIM* consists of Document Object Model (DOM) [345] parsers which parse system and deployment information from the respective descriptor, provided as XML files.

SENDIM consists of an easy upgrade and revert feature for itself as well as the algorithms deployed on top of it. OSGi enables versioning of the plugins. Therefore, when a user deploys

updated bundles as *changesets* into the core controller runtime of *SENDIM*, *SENDIM* will upgrade itself as well as the deployed user algorithms to the new version upon the restart of the framework, without removing or altering the existing bundles. A remote file copying approach deploys the changesets into the remote hosts securely. *SENDIM* uses shell scripts with remote calls to invoke or restart the remote execution and to invoke the automation tools and packet installers. Upgrades based on deployed changesets enable the deployed system to be quickly reverted to a previous version, by removing the malfunctioning changesets or later versions of the bundles from the repository. This checkpointing feature [191] enables a quick revert to the previously known stable working state, should critical test cases fail.

3.4 Evaluation

We evaluated *SENDIM* for its modeling and migration capabilities using a cluster configured with up to 6 servers, each with Intel® Core^{TM} i7-4700MQ CPU @ 2.40GHz × 8 processor, 8 GB memory, and Ubuntu 14.04 LTS 64 bit operating system. We benchmarked *SENDIM* against CloudSim for its simulation time as independent simulations as well as incremental updates, while ensuring its accuracy in simulating simple network tasks such as routing algorithms, cloud resource allocation, and network load balancing. We then evaluated the effectiveness of *SENDIM* in migrating the executions between simulations to emulations and vice versa. We repeated each experiment 6 times and considered the average time.

3.4.1 Simulations with *SENDIM*

First, we benchmarked the scalability of *SENDIM* with the size of the simulated problem and the number of servers that host the simulator for a distributed simulation. We simulated random routing of network flows in a data center of small-world datacenter topology [306] with a varying number of nodes (i.e., switches and hosts). We maintained the links across the data center for each of the node to be proportional to the number of nodes in the data center. The modeled networks consist of a varying average degree, the average of the number of links per each node, up to 100. We first benchmarked the simulation time of *SENDIM* in a single node, against CloudSim [63], as CloudSim has been extended as *CloudSimSDN* to simulate SDN [310]. We then deployed *SENDIM* as a clustered simulation environment over 5 different servers and the controller in another server to benchmark the distributed execution. We compared the distributed execution time of *SENDIM* simulations against $Cloud^2Sim$ [185], the distributed execution for CloudSim, deployed on 5 servers.

We denote the simulations by an integer index, $n \in \mathbb{Z}^+$. The executions in CloudSim represent the time consumed by CloudSim as a single entity. On the other hand, the *SENDIM* simulation involves executing i) the controller with the *SENDIM* controller extensions and, ii)

the *SENDIM* simulation sandbox.

$$
\begin{aligned}
T(n) &\implies \{\text{The time taken by CloudSim to execute n.} \\
T(n_c) &\implies \begin{cases}\text{The maximum time taken by the relevant controller instance to execute the controller workflows to simulate the system algorithms identical to that performed by CloudSim in execution } n.\end{cases} \\
T(n_s) &\implies \begin{cases}\text{The maximum time taken by the simulation sandbox instance to simulate the data plane of system } n.\end{cases}
\end{aligned}
\tag{3.2}
$$

Since the controller and the simulation sandbox execute in parallel, the time taken for *SENDIM* to complete any simulation execution is defined as T'(n) = max($T(n_c)$, $T(n_s)$). T'(n) depends on the component (either the simulation sandbox or the controller) that takes longer to complete the simulation task. In case of distributed $Cloud^2Sim$ executions, T(n) also considers the maximum time taken by the instances, hence considering the last instance to complete the execution, which is often the master of the simulation cluster. We benchmarked CloudSim and *SENDIM* simulation sandbox in a single as well as multiple instances of the simulator by comparing T(n) and T'(n).

Figure 3.4 indicates the time taken by *SENDIM* and CloudSim to complete the simulation, highlighting the performance and horizontal scalability of *SENDIM* for large simulations. While CloudSim outperformed *SENDIM* for smaller scale simulations, *SENDIM* outperformed CloudSim for networks with more than 1000 nodes. The exact number of nodes where *SENDIM* starts to outperform CloudSim depends on the nature of each simulation. However, we observed that *SENDIM* simulated larger simulations better when its performance benefits become more pronounced compared to its initial overhead. While both *SENDIM* and CloudSim scaled out reasonably well, the performance degraded with distributed execution in $CloudSim/Cloud^2Sim$ for smaller simulations. We attribute this performance loss to the increase in communication and coordination overheads. *SENDIM* distributed execution did not have such overheads, as it distributes the workload adaptively, leveraging the controller's network awareness. Additional instances were involved in the *SENDIM* simulation only when the load was high. Therefore, *SENDIM* avoided performance losses caused by premature scale-outs. Even for larger simulations, *SENDIM* showed higher vertical (scaling up for larger workloads) and horizontal (scaling out to multiple execution servers) scalability, compared to CloudSim.

3.4.2 Incremental Updates and State-Aware Executions

We then evaluated the efficiency of *SENDIM* in its support for incremental development. We benchmarked *SENDIM* simulation sandbox against CloudSim in a series of simulations composed of incremental updates. We simulated a slightly modified ping-all function across the networks consisting of random and uniform links among the switches. The simulation itself is smaller in scale since we have already established the efficiency of simulating larger simulations with *SENDIM* in Section 3.4.1.

Table 3.1 describes the stages of our execution. Executions 1_c and 1_s represent the initialization of the controller environment and *SENDIM* simulation sandbox respectively, along with

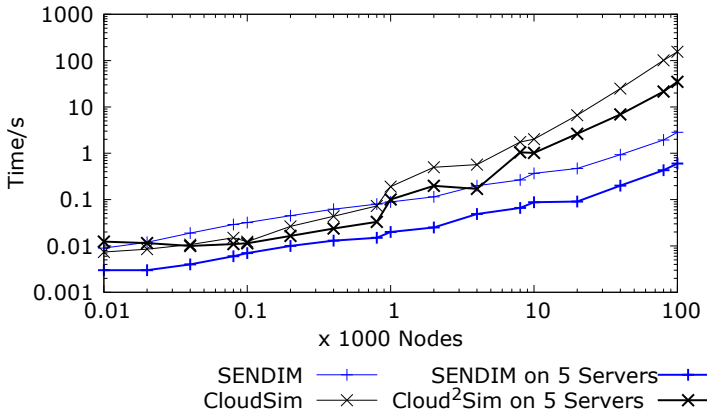


Figure 3.4: Simulating a random routing across a data center network

Table 3.1: Steps of the Executed Simulation: T(n) vs. $T(n_s)$ & $T(n_c)$

CloudSim vs. *SENDIM* (Sandbox & Controller)	Description of the Execution
1 vs. 1_s & 1_c	Initialization of the environment
2 vs. 2_s & 2_c	Initial application deployment in the environment
3 vs. 3_s & 3_c	Updated algorithm on the same simulated system
4 vs. 4_s & 4_c	Updated algorithm on the same simulated system
5 vs. 5_s & 5_c	Reverting the algorithm to a previous state
6 vs. 6_s & 6_c	Changing the simulated system with the same algorithm

the first simulation, identical to the CloudSim simulation 1. Execution 2_c indicates the initial deployment of a subsequent simulation logic in the controller, whereas execution 2_s indicates the respective deployment of the simulated system into the simulation sandbox. Execution 3_c and 4_c represent updates to the application logic deployed in the controller, with no change in the system simulated in the simulation sandbox. Execution 5_c indicates a revert to the execution 2 by removing the changesets that were deployed in executions 3_c and 4_c. During executions 3, 4, and 5, the simulated system is not changed while the changesets are dynamically deployed into or removed from the controller. Therefore, there was no time measured for 3_s, 4_s, and 5_s as the system was already simulated into the simulation sandbox even before the algorithm starts its execution in the controller. Execution 6_s indicates the change in the simulated environment, while not changing the simulation logic in the controller. As the controller did not have to execute the algorithm again in this case, 6_c does not consume additional time. We observed the time consumed by CloudSim and the *SENDIM* counterparts to perform the execution steps. Figure 3.5 indicates a breakdown of the execution times.

SENDIM executions are not uniform, as *SENDIM* stores the states from the previous simulations and leverages that information in the subsequent simulations. CloudSim does not store the states of previous simulations as a continuous simulation workflow. Therefore, all six CloudSim executions are similar, discrete, and independent from each other, resulting in a constant simulation time across all the six simulation executions considered. Execution 1_c, which

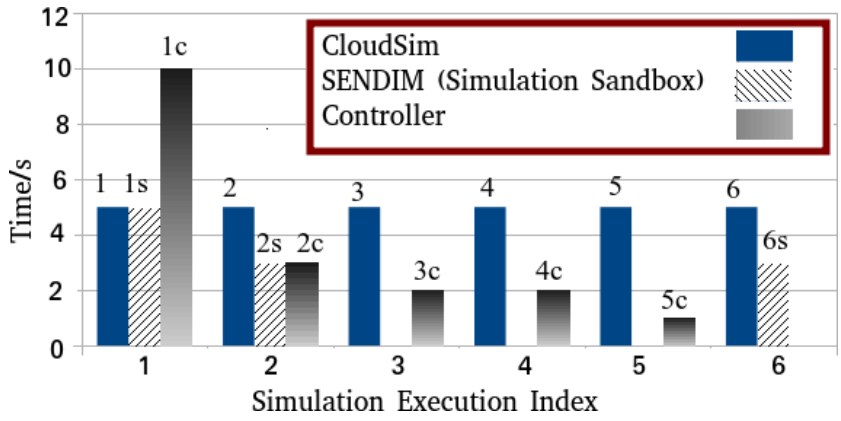


Figure 3.5: Time taken for Simulation Executions

represents the initialization of the controller environment takes more time than the CloudSim simulation execution 1, as the initialization time of the control plane solely depends on the SDN controller. We observe that the time consumed by the *SENDIM* simulation sandbox itself is equal to CloudSim as during execution 1_s, though it communicates and coordinates with the controller, and performs a complete simulation of the network system. The time taken to complete the simulation was around 50% of the execution time of simulation 2 in CloudSim. Subsequent executions 3_c and 4_c consume lower time in *SENDIM*. On the other hand, unaware of this contextual information, executions 3 and 4 consume as much as the time of execution 1 or 2 in CloudSim, giving a competitive advantage to *SENDIM*. Since execution 5_c is just a mere revert, it consumed even lesser time. Nevertheless, execution 5 is again a fresh simulation of the entire network for CloudSim, as it does not offer breakpoint and revert features. Execution 6_s still takes less time than CloudSim. Its simulation time is equal to 2_s, whereas 6_c does not consume additional time as there was no change in the controller algorithms in this step.

Our observations highlight that overall, *SENDIM* consumes up to less than 50% of the time compared to CloudSim for continuous modeling of cloud networks. In our chosen case of continuous modeling of cloud networks with incremental development and reverts, *SENDIM* easily outperforms CloudSim due to its optimized design for modeling incremental and test-driven cloud network developments and parallel execution of the simulation environment and the controller that executes the network algorithm on the simulated system. We note that the exact performance enhancement would indeed depend on the nature of the simulation.

We found that the overhead in the development time of *SENDIM* simulations is well justified by the time saved in the continuous updates in the simulated systems. The overhead caused by distributing the simulations to the controller and the simulation sandbox can be considered negligible, except for the initial start-up time. Furthermore, the state-aware simulations enabled linear scalability for *SENDIM* simulation sandbox against the size of the computer cluster and the problem scale.

3.4.3 Seamless Migrations Across Development and Deployment Dimensions

Finally, we evaluated the efficiency of *SENDIM* in performing migrations between simulations and emulations, and vice versa.

Simulation to Emulation Migrations: We evaluated the performance of the simulation to emulation migrations by manually invoking the *SENDIM* Mininet converter. We modeled networks with a varying number of nodes consisting of a range of average degree for the switches, with a random routing of network flows across the nodes. We expressed the flows and the network properties as a system descriptor, which is the core of the simulation scripts written following the *SENDIM* DSL. We then converted the simulation scripts into Mininet emulation scripts using the Mininet converter. We observed the total execution time to migrate an application from simulation to emulation, by converting the system descriptor scripts, initiating Mininet with the network consisting of nodes, links, their properties, and network flows as specified in the descriptor, and start the Mininet emulation of the network routing execution. Figure 3.6 illustrates the total execution time for the migrations, against a varying number of nodes and average degree.

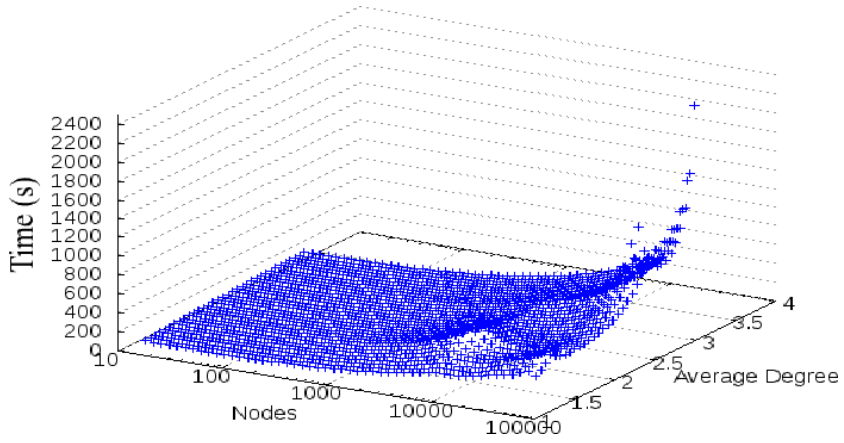


Figure 3.6: Migrating a Simulation to Emulation

The migration time increases with the number of nodes and the average degree of the modeled network. Time taken for the migration increases with the number of nodes, as the converter has to read, parse, and convert the system descriptor into Mininet scripts. There was an increase in time with the rise in the average node degree, which is more visible in larger networks with more nodes. The number of links and the alternative routing paths increase with the increasing average degree. Therefore, the Mininet converter has to convert more lines of *SENDIM* simulation script into the Mininet script, with the increasing average degree which is more prominent as the number of nodes also increased. *SENDIM* was able to handle 10,000 nodes within a few seconds. It successfully migrated larger complex networks of 100,000 nodes, albeit with an above-linear increase in the time taken. This increase was due to the memory and processing required for parsing the descriptor files as large as 26 MB, and writing Mininet scripts that are up to 18 MB in size for 100,000 nodes with up to 100 links per node.

Emulation to Simulation Migrations: Then, we assessed the ability of *SENDIM* in automatically migrating the emulations to simulations when the resource requirements for the modeled problem space exceeds the available resources. We configured *SENDIM* to execute a random routing algorithm first as a Mininet emulation that later transforms itself into a *SENDIM* simulation as the resources become scarce with the increasing problem size. We benchmarked this configuration with the execution time with pure Mininet emulations and a complete simulation with *SENDIM* simulation sandbox without such automatic migrations.

Figure 3.7 shows the time taken for Mininet to emulate the network, the *SENDIM* simulation sandbox to simulate it, and *SENDIM* configured to resort to simulations when the emulations take more than a minute. The simulation sandbox constructed the system with nodes and links within a few seconds even for large systems with up to 100,000 nodes. On the other hand, Mininet consumed up to 300 seconds even for a network of 1800 nodes and links. Hence, *SENDIM* simulation sandbox showed a 100-fold performance increase when compared to Mininet in visualizing the network. When increasing the network size further up to 1900, Mininet failed to create the links, though it successfully emulated the nodes. For networks beyond the size of 8000, Mininet fails with an OSError when attempting to construct the nodes. When emulating a network of 1800 nodes with Mininet, Network Manager consumed as high as 99.6% of CPU during the links construction phase. Furthermore, we observed that the Open vSwitch daemon (ovs-vswitchd) consumed up to 100% of the CPU during the final stages of the emulation. However, simulation of the same network with *SENDIM* consumed only up to 6% of CPU.

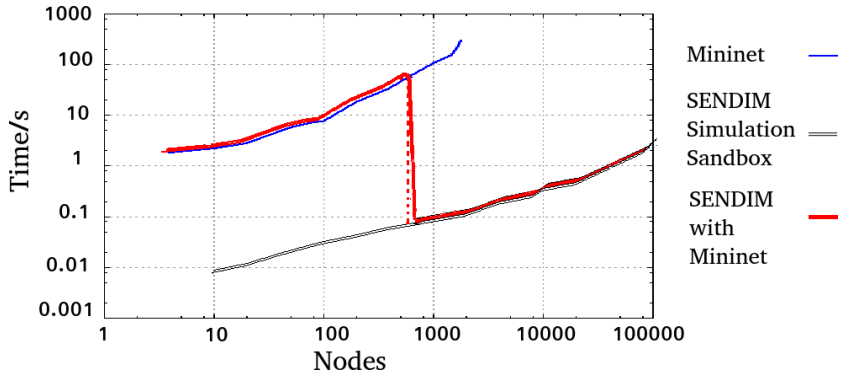


Figure 3.7: Network Construction with Mininet and *SENDIM* Simulation Sandbox

The results highlighted how *SENDIM* could offer the best of both worlds by invoking the emulator at small scales and then resort to its simulation sandbox to simulate larger networks when the resources are scarce to emulate. By transitioning seamlessly across both realizations, *SENDIM* offers the accuracy for finer problem spaces by emulating when resources are sufficient and simulating for larger networks with limited resources. The observed accuracy in simulating the network with the *SENDIM* simulation sandbox highlights the potential of using it at the early stages of the development and a seamless migration from simulations to emulations by either increasing the computing resources at the latter stages of development or by reducing scale of the simulated system (i.e. by reducing the problem size). The seamless migration with minimal efforts is made possible since the code to be executed in the controller remains the same,

and the *SENDIM* system descriptors can be easily ported to Mininet scripts with no manual code conversion or rewrite.

We believe the incremental development and deployment supported by *SENDIM* is a first step in bringing interoperability to simulations and emulations, which remain primarily incompatible across frameworks. SDN already facilitates bringing the emulations close to the physical developments by making the emulations deployment-ready in the actual execution environments. Leveraging an SDCD approach to make all the simulations and emulations interoperable is a substantial implementation challenge. However, the evaluations on *SENDIM* highlight that by developing and exploiting an SDN-based middleware framework for cloud network simulations, such migrations can become a reality. While *SENDIM* enables seamless migrations between Mininet emulations and its simulations, such migrations are limited to Mininet. Migrating *SENDIM* DSL into other emulator scripts can be achieved by implementing the relevant Emulator Converter in *SENDIM*. However, it is a significant engineering task to enable such migrations across existing several emulators and simulators, and that is beyond the scope of *SENDIM*.

Execution and Deployment Migrations Across the Environments: Configuration management systems offer automation for managing on-site and public cloud deployments with varying configurations and orchestrate multiple products hosted on them. Figure 3.8 highlights the potential for extending such configuration management beyond the current scope of such systems, by comparing the seamless deployment capabilities of *SENDIM* with the popular configuration management systems, Puppet and Chef. Transitions across both dimensions indicate, the migrations across different deployment environments for the same development dimension (in horizontal arrows), and migrations across different development dimensions in the same deployment environment (in vertical arrows). The vertical sets showed on transitions across both dimensions indicate the functionality offered by *SENDIM*, Chef, and Puppet, respectively, highlighting when *SENDIM* outperforms the others.

Configuration management systems function on migrating application executions across different deployment environments of the same development dimension, providing a varying degree of support for each development dimension. Emulations and physical deployments can effectively be handled and managed by the configuration management systems to migrate across different deployment environments. However, they are not efficient in managing simulations. Migration across different development dimensions is beyond the scope of the current configuration management systems. *SENDIM* offers complete support for migration across both dimensions – migrating across different deployment environments for the same development dimension, or migrating across different development dimensions over the same deployment environment. *SENDIM* manages an entire matrix of development and deployment dimensions, with extension points to deployment systems such as Docker. Hence, it provides seamless migration across more environments.

Puppet and Chef do not leverage the programmability offered by SDS. They also do not cover the management across the development dimension, unlike *SENDIM*. Both *SENDIM* and Puppet provide an easy-to-use declarative language as DSL, where Chef uses a Ruby-based DSL. *SENDIM* is the first step towards interoperability across various dimensions. As such, we

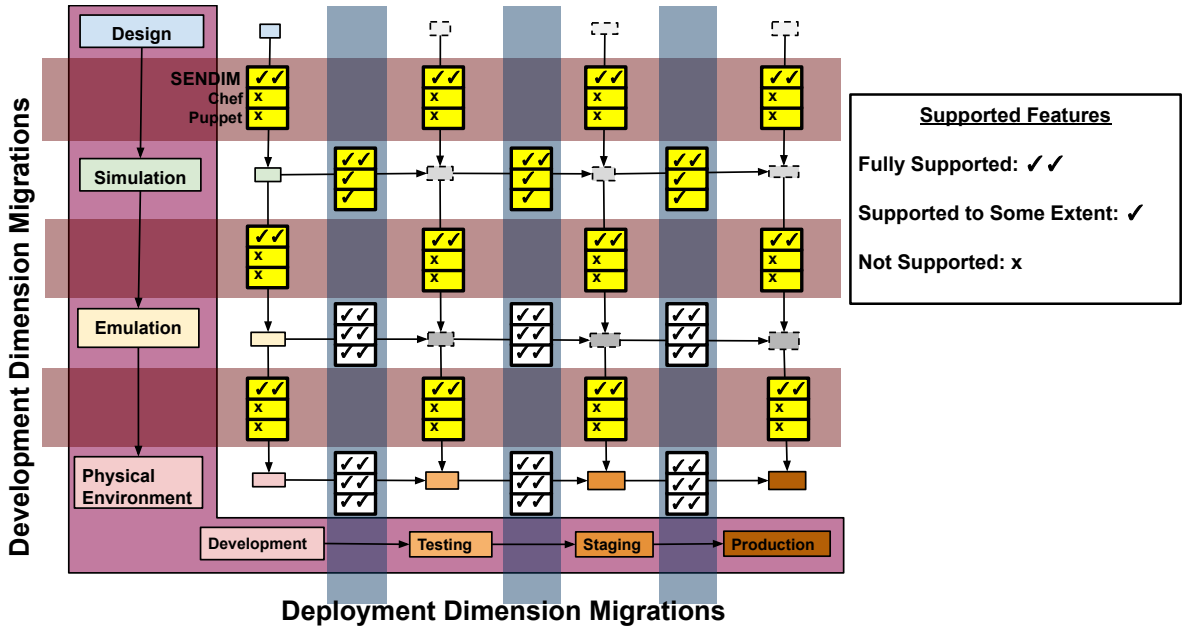


Figure 3.8: Comparative Qualitative Assessment with Configuration Management Systems

don't see it as a complete alternative to configuration management and automation systems such as Chef and Puppet. However, SDCD highlights the potential for extending the configuration management systems for complete automation throughout the development and deployment phases. We also foresee such approaches enhancing the migration capabilities in inter-cloud environments.

3.5 Conclusion

An SDCD approach provides quantifiable improvements concerning interoperability, scalability, and performance in cloud networks, by bringing the unified management of SDN controllers to the early stages of cloud network development. SDCD enables product deployments and updates to be coordinated and managed from a central location efficiently with less manual efforts. We built *SENDIM* as an SDCD framework to facilitate incremental modeling of cloud networks, state-aware simulations, and simulations of complex workflows such as NSC. *SENDIM* enables seamless deployment and migration through loose coupling of the deployed application from their execution environments and development realizations. The reduced repeated coding and increased management capabilities justify the additional effort of separating the application logic from the implementation. Preliminary assessments elaborated the seamless deployment and enhanced migration capabilities of *SENDIM* as well as the efficiency of the *SENDIM* simulation sandbox in operating as either a stand-alone network simulator or a distributed simulator integrated with an SDN controller.

4 Cloud-Assisted Networks as a Connectivity Provider

Network connectivity providers have been proposing novel approaches for cheaper and low-latency end-to-end connectivity. The mobile and edge computing paradigms have resulted in an increasing demand for dynamic short-term connectivity. Network connectivity providers such as transit providers currently offer bandwidth with minimum commitment, as low as 10 Mbps [81]. However, contracts shorter than one month are still uncommon, thus preventing the users from promptly interconnecting without committing to a long-term service agreement. To rectify this limitation, companies such as Epsilon [114] and PacketFabric [263] are working towards making bandwidth a tradeable utility, steering up dynamic interconnections among their users. However, these companies limit their focus primarily to the enterprise use and are not aligned to provide cheaper and high-performant connectivity to the end users. This state of affairs highlights the need for affordable alternative connectivity providers on a global scale, with short-term commitments. Such alternatives should consider two major factors, namely, technological performance as well as economic benefits.

First, the Internet applications operate with real-time constraints on latency and a need for stable bandwidth. Therefore, the proximity of these service deployments to an end user plays a significant role in the user's QoE. Cloud platforms are opening up more regions, availability zones, and Points of Presence (PoP) [31], to better serve the web applications with heavy bandwidth demands. Distributed cloud data centers [188] distribute or replicate the computation and data across multiple geographical regions near to the users. These distributed and decentralized architectures are getting more mainstream, as a compromise between the centralized cloud and independent on-premise deployments. Thus, an alternative connectivity provider should consider this changing network ecosystem of end users and service providers.

Second, the Internet consists of a large pricing disparity, including a different rate of price decline over the years, between the core Internet regions and the remote regions. The steady decline of IP transit prices in the past two decades has helped fuel the growth of traffic demands in the Internet ecosystem. Despite the decreasing unit price, bandwidth costs remain significant due to the ever-increasing scale and reach of the Internet. Furthermore, the oligopoly of a few connectivity providers and a substantial dependence on expensive long-haul links to the US or EU for international connectivity, have caused a higher price for IP transit in the remote Internet regions [55]. For instance, prices for 10 Gbps Ethernet (GbE) bandwidth remain up to 20 times more expensive in São Paulo and Sydney, compared to EU and USA. This disparity is even more pronounced in remote regions such as Central Asia and Sub-Saharan Africa. For example, as of 2014, while the transit cost per Mbps per month was 0.94$ in the US [252], it was 15$ in Kazakhstan and 347$ in Uzbekistan [218]. Even though the average IP transit prices at major Internet hubs have fallen by an annual 61% during the past two decades, this decline

has been much less pronounced elsewhere [252]. This variation in pricing plays a significant role in determining the success of an alternative connectivity provider in any region.

Cloud-assisted networks offload network functionality to cloud platforms, ultimately leading to more flexible and highly composable NSCs. As a network softwarization approach, cloud-assisted networks separate the network from the cloud infrastructure, offering network connectivity as an overlay provider. A cloud-assisted network leverages the cloud VMs to host the virtual routers and the underlying cloud network infrastructure for its data transfer. Despite the prevalence of cloud-assisted networks, their use as a connectivity provider for end user data transfers and the economic viability of such usage are not well studied. We argue that the feasibility of cloud-assisted network solutions deserves a broader study, considering the flexibility of network softwarization as well as the performance and monetary benefits it brings into network connectivity.

In this chapter, we study the technological and economic viability of a cloud-assisted network and propose *NetUber* as an efficient architecture to realize an alternative connectivity provider built atop a cloud-assisted network. *NetUber* consists of a broker that i) purchases spot VMs from the cloud providers, and ii) creates an overlay network over the multiple spot VMs to function as a large-scale inter-region connectivity provider to the end users who would buy connectivity directly from *NetUber*. Thus, *NetUber* operates as an on-demand virtual connectivity provider running atop several spot VMs configured as virtual routers. By leveraging the memory and CPU of the acquired spot instances, we further envision a deployment of auxiliary network services such as compression-as-a-service [147] and encryption-as-a-service [152], offering an optional compressed or encrypted data transfer between the regions.

Our primary contributions are: i) an economic model to exploit spot markets for direct secured connectivity between pairs of endpoints, and ii) an inter-cloud approach that leverages spot VMs in building a reliable virtual connectivity provider. When compared to traditional flat-price connectivity providers, our extensive evaluation shows that i) *NetUber* best suits the needs of small dynamic monthly transfers up to at least 50 TB and ii) *NetUber* cuts Internet latencies up to a factor of 30%. We see our contributions as a first step towards a more systematic understanding of the next-generation overlay interconnection networks driven by network softwarization efforts.

This chapter is composed of the contents of the publication: [C1].

4.1 Cloud-Assisted Networks: A Market Analysis

In this section, we look at the economic and technological perspectives towards building *NetUber* as a cloud-assisted network overlay and present our observations on cloud pricing trends that helped us in making the design decisions.

4.1.1 Cloud Instances

Large-scale dynamic cloud overlays require many cloud instances and a billing model that charges overlay operators only for their actual usage of cloud resources. The cost of acquiring and maintaining several on-demand instances remains a concern for overlay operators, despite the decreasing cloud instance pricing [39] and improved pricing models such as the per-second billing[1] offered by the cloud providers. We performed a brief analysis on the quoted prices for monthly contracts from ISPs and on-demand cloud instances to find whether it is feasible to build a cloud-assisted overlay network with the same SLOs as the ISPs using on-demand instances. We observe that it is more expensive to maintain such an overlay with on-demand instances compared to the ISP pricing. We should bring the cost of cloud instances and cloud data transfer down to ensure that an alternative connectivity provider built atop several cloud instances remain economically profitable while offering the same SLAs as the classic connectivity providers such as ISPs.

Spot instances: *spot instances* are spare computing resources, identical to their on-demand counterparts, which are offered by the large cloud providers at a much lower price but can suddenly be interrupted by the cloud provider with a short notification. Cloud providers advertise significant cost savings with their spot instances while providing the same performance guarantees offered by the on-demand instances. Amazon estimates that using its EC2 (Elastic Compute Cloud) spot instances can save up to 90% compared to its on-demand pricing [17]. Similarly, GCP preemptible instances [136] provide a flat rate of 80% of savings, and Microsoft Azure low-priority instances [299] are expected to offer the same, compared to their respective on-demand offerings.

Reserved instances: Cheaper cloud instances with a time commitment such as the *reserved instances* are unsuitable for dynamic overlays, as they limit the elasticity of the overlay. Reserved instances refer to the cloud instances that are leased for an extended period such as 1 - 3 years, for a discounted price. Reserved instances often offer savings similar to that of spot instances – for example, 82% savings in Azure. However, due to the dynamic nature of the network demand, the mandatory long-term commitment makes the reserved instances unfit for the cloud-assisted network use cases, unlike the spot instances. EC2 spot instances are also available with predefined duration from one to six hours, 30 - 50% cheaper than the on-demand instances, making them less volatile and more reliable [17]. However, such instances force the overlay provider to commit to those instances for at least an hour, to reap the benefit of their less volatility with the increased price compared to the regular spot instances. Therefore, such instances are unfavorable due to the unpredictability of the end user demand for more bandwidth from the overlay provider for their data transfers.

Regions and Availability zones: Cloud providers have a global presence with several *regions* and *availability zones* to maintain proximity to the user and offer fault tolerance to their cloud infrastructure. Cloud data centers are present in several geographical locations that are known as cloud regions. Each region has multiple availability zones. An availability zone consists of

[1] AWS, GCP, and Azure started their move to per-second billing from their per-hour and per-minute billings, starting from October 2017 onwards.

data centers that are physically isolated from the other availability zones in the region, yet connected through low-latency links internal to the cloud region network. Availability zones provide resilience and fault-tolerance to the cloud regions.

Spot instance price fluctuations: Price fluctuations of EC2 spot market are inevitable and are more vigorous for a few instance types in some availability zones at specific time frames. Even identical spot instances of different availability zones, inside the same region, often have different prices at times. In contrast, GCP preemptible instances have a fixed pricing scheme unlike the dynamic pricing of EC2 spot instances. We monitored the availability, performance, and price fluctuations of EC2 spot instances over a time frame of three months (April - June 2017). Throughout our experiments, Linux r4.8xlarge spot instances remained the cheapest, yet memory-optimized EC2 instances with 10 GbE network interface. Each of these R4 instances consists of 32 virtual CPUs and 244 GB of memory. They offer 10 Gbps bandwidth inside an AWS placement group (a logical grouping of EC2 instances inside an availability zone, configured by the cloud user to function as a cluster). Network transfers outside a placement group are limited to 5 Gbps [20].

Figure 4.1 depicts the price fluctuations among the Linux r4.8xlarge instances of the availability zones of Frankfurt and Sydney regions during the three months. We observed up to 89% of savings with r4.8xlarge spot instances in Sydney, Frankfurt, and North Virginia regions. The spot price for Frankfurt remained relatively low and stable across all the availability zones. However, the spot price even exceeded the on-demand price in Sydney for availability zone 2b at times, while the other two availability zones remained cheaper for the spot instances. Instances of the zones eu-central-1c and ap-southeast-2c had a relatively steady and cheap price.

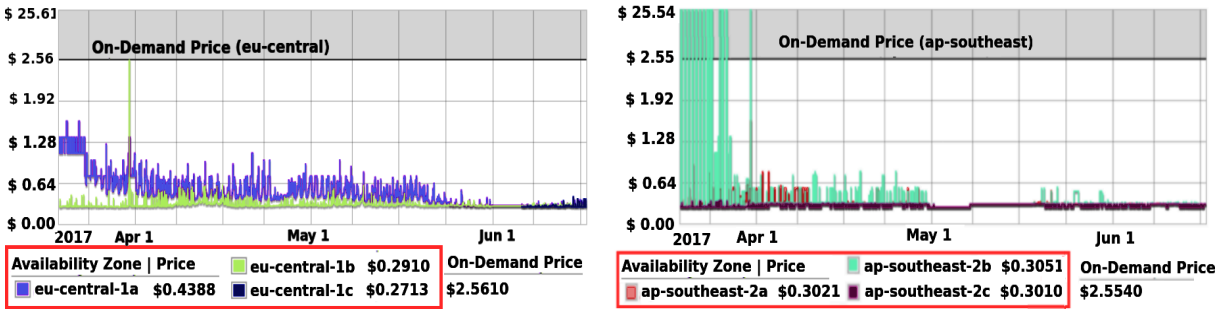


Figure 4.1: Linux r4.8xlarge Spot Instance Price in Frankfurt and Sydney, April - June 2017

We observe that by leveraging multiple availability zones in each region, a stable overlay could be operated using the cheapest spot instances over time, without opting for on-demand instances or instances with a predefined duration. While it is straightforward to opt for instances from the availability zones that have remained cheaper recently with a stable price, price increases and fluctuations in the future are unpredictable. Hence, while some spot instances belonging to a particular availability zone are being terminated, spot instances may be spawned in the other availability zones of the same region. Approaches such as *EC2 Spot Fleet* [19] enable the cloud users to spawn and manage multiple spot instances in a unified manner adhering to user specifications in target capacity and cost threshold. These approaches mitigate the challenges of scale and complexity in managing a large number of spot instances

at once. This dynamic nature of the cloud-assisted network poses the questions and challenges including, how the current active instances in each region is maintained effectively from a central location such as an Amazon S3 (Simple Storage Service) bucket and how the inter-region traffic should be re-adjusted to route to the current active instances.

4.1.2 Cloud Data Transfer

Inter-region cloud data transfer prices remain relatively high as there is no "spot data transfer" that provides cheaper data transfer with a volatile bandwidth. Around 20% and 25% of price reductions have been reported in 2010 [37] and 2014 [38] respectively, for data transfer out from the EC2 instances. Despite the price reductions, the data transfer prices still remain significant for large volumes of inter-region data transfer.

While it is typical for small enterprises and home users to connect to the cloud servers through the public Internet, a dedicated connection or co-locating with a cloud PoP is recommended for large-scale data transfers to provide adequate throughput and cost efficiency. Depending on a third-party connectivity provider such as ISPs to connect to the cloud server limits the scale of data transfer. For example, typically ISPs offer up to 1 TB per month for home users abiding by the data rate promised in the SLA. Using private direct connections, cloud data transfers avoid the bottleneck caused by the Internet-based connectivity between the user data centers and cloud servers. The virtual network overlay users must have an existing connection to the cloud provider or set it up directly with the cloud provider or its partners. The cloud user pays the installation cost and the monthly pay-per-use operational costs, to the cloud provider. The Direct Connects are not more expensive than the alternative direct/end-to-end connect options with the same throughput.

Cloud providers have varying pricing for their data transfers based on the origin and destination of the data transfer. Currently, cloud providers such as AWS and GCP do not charge for incoming data transfers, regardless of the origin – from the other regions as well as the Internet. They charge for outgoing data transfers, which differs based on the destination: the Internet, a server connected by the cloud Direct Connect, another region, another availability zone in the same region, or a cloud instance of the same availability zones. Data transfers outside the cloud network and long distance data transfers cost more than the short distance data transfers. For example, data transfers between regions cost more compared to the transfers between the availability zones of the same region. Typically, cloud providers do not charge for outgoing data transfers to another cloud instance of the same availability zone. AWS charges the inter-region data transfers independent of the destination region, with a few exceptions for nearest regions such as cheaper transfer between the US East regions – North Virginia and Ohio. On the other hand, GCP clusters the regions into four groups (worldwide, Asia, China, and Australia) and charge based on the group of destination region.

Cloud region price disparity: Figure 4.2 illustrates the variation of pricing among the AWS regions to transfer a unit of data (1 TB), for transfers up to 10 TB. Larger transfers become cheaper per unit, with the price reaching almost half when the total volume reaches 500 TB. For example, the unit cost decreased from 92.16$/TB to 51.20$/TB for transfers out from regions 1

- 8 (Canada and EU-based regions, and US-based regions except for GovCloud) to the Internet. The cloud data transfer options such as Direct Connect are cheaper than sending data from the cloud to an end user server directly through the Internet. The discrepancy in pricing among the regions is visible (which is comparable to the IP transit price disparity); regions 1 - 9 (US, Canada, and EU) remain much cheaper than the others.

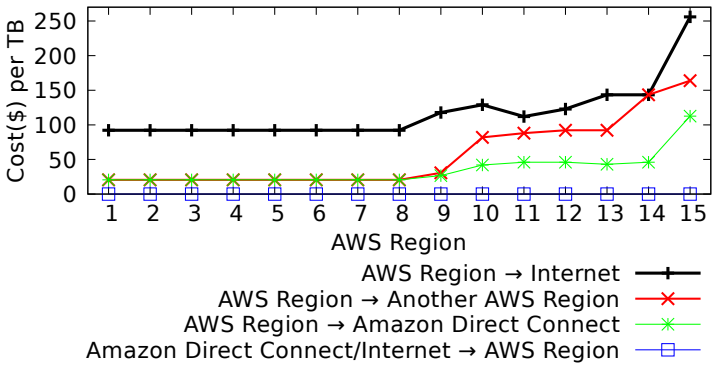


Figure 4.2: Data Transfer Cost for AWS

We observe that currently, the data transfer pricing models from the cloud providers are not supportive of a more comprehensive adoption of an Internet scale cloud-assisted network and making such overlay network a reality requires additional research effort. First, there is no differentiated pricing model based on the current local time or demand for bandwidth. Second, cloud providers charge for data transfers by the volume of data transferred, rather than by the data rate, unlike transit providers or ISPs. Regardless of the throughput, the cloud user pays the same amount based on the volume of data transferred. Therefore, there is no incentive for the cloud users to opt for a slower data rate even when their application is delay-tolerant. Furthermore, not considering the cloud overlay network scenario, cloud providers discourage long-distance inter-region data transfer to counter communication delays that are typically caused by a poor SaaS design. Based on our observations and subsequent analysis on the cloud data transfer offerings and the availability of cheaper spot VMs, we deduce that the data transfer will contribute with the most significant share to the expenses for the Internet scale overlays deployed over multiple regions. We posit that a cloud-assisted network should take additional measures such as data compression and minimizing data transfer path lengths to operate as a cost-efficient alternative connectivity provider.

4.2 Towards *NetUber* Deployments

In this section, we will look into the deployment architecture of three primary use cases of *NetUber*: i) a cheaper point-to-point connectivity between two regions for data transfers, ii) a premium connectivity between multi-cloud regions (i.e., cloud regions from two different providers) [274] for end users for faster data transfer and better SaaS performance, and iii) a connectivity provider with additional network services.

4.2.1 Economical Point-to-Point Connectivity

NetUber leverages spot instances to offer short-term or small-scale direct access between two geographically separated endpoints, as an economical alternative to enterprise MPLS networks. *NetUber* has no dedicated servers. It consists of several 'brokers,' simple cloud clients (i.e., such as AWS Command Line Interface (CLI) applications that request and maintain cloud instances. *NetUber* brokers, as well as the other *NetUber* instances that function as virtual routers, are hosted on a set of spot instances per region. *NetUber* leverages the cloud monitors such as AWS alarms to ensure that each region has at least one broker instance that is active and not scheduled for termination. For a stable overlay, *NetUber* needs some instances that function as virtual routers in each region, based on the bandwidth demand and the number of active instances at any given time. Since cloud providers bill their instances per second, at any moment, the broker purchases and maintains instances from the availability zone that has the cheapest among the available high-performance memory-optimized cloud instances in a region. It terminates the expensive instances at the earliest.

Each virtual router can dynamically connect to the virtual routers of certain spot instances of another region through the overlay, based on the users' connectivity or data transfer requests. *NetUber* uses simple network routing policies in its overlay to transfer data to the VMs of other regions that function as virtual routers. It is possible to use an Amazon Machine Image (AMI) to instantiate an enterprise cloud router in each spot instance of *NetUber*. Although such enterprise cloud routers efficiently route the data between the endpoints in the cloud network, these software routers are expensive to acquire and maintain over time. *NetUber* avoids the use of such virtual routers, to remain economically feasible. The routing policies of *NetUber* spot instances are primitive, due to the limited knowledge and control available to the cloud users regarding the underlying cloud backbone network. With the awareness of the cloud provider network, we can improve the inter-region routing efficiency of *NetUber*.

Consider a scenario where an end user chooses to send data from her server s_o to the destination server s_d. These servers are in the cloud provider regions r_o and r_d respectively and are connected to the cloud provider through a dedicated connection such as Amazon Direct Connect. Figure 4.3 illustrates a representation of a sample *NetUber* deployment that offers a direct connection between s_o and s_d. *NetUber* is composed of many spot VMs in multiple regions, connected through the overlay network of virtual routers.

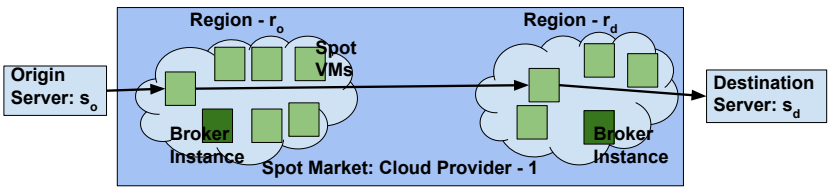


Figure 4.3: *NetUber* Deployment with a Single Cloud Provider

NetUber aims to maximize the effective data rate from its cloud instances, with minimal underutilization. *NetUber* shares the instances across users for multiple data transfers – at

the same time, or at different time intervals, based on the data rate required for the transfer. The broker instances monitor the resource utilization of the current VMs purchased in the spot market. A spot fleet defines how a group of spot instances should be spawned. Therefore, spot fleets are capable of spawning and managing multiple spot instances at once. The *NetUber* brokers alter the spot fleet policies to bid for more instances, when the existing VMs are not sufficient (measured with a margin, to avoid performance degradation) to address the demand for connectivity and when the price for the spot instances are profitable to *NetUber*. Hence, the broker instances acquire spot instances based on the current spot instance pricing, the bandwidth demand from the end users, and the spot fleet policies. The technical challenges include, i) initializing a newly spawned instance to operate as a virtual router in a short time, and ii) ensuring that the instances can be connected and identified through an overlay, other than their physical address, as spot instances remain volatile. The broker retrieves the list of spot instances in each region through the AWS CLI and EC2 APIs.

4.2.2 Higher Performance Point-to-Point Interconnection

NetUber aims to increase flexibility and control in choosing the shortest Internet paths. On the intra-domain traffic, an ISP can seek the shortest path as it controls the network. Since the Border Gateway Protocol (BGP) [329] decisions are mainly policy-oriented, Internet-based connections that typically rely on BGP may not result in the selection of the best or the shortest path. With the cloud instances, *NetUber* can choose to intelligently route the traffic towards the VMs in the specific regions, minimizing the number of hops and path length.

Currently, a few cities and geographical regions host the cloud regions for multiple providers. For example, North Virginia, Mumbai, London, São Paulo, Tokyo, and Singapore host both AWS [18] and GCP [135] regions. Ohio, North Carolina, Seoul, Canada central, and Ireland are AWS regions but not GCP regions; Iowa, Belgium, South Carolina, and Taiwan are GCP regions that are not AWS regions. Figure 4.4 elaborates this scenario with the cloud provider p having presence in regions r_o and r_i, and the provider $p+1$ with presence in regions r_i and r_d. None of the providers are present in both r_o and r_d, while both providers are present in r_i.

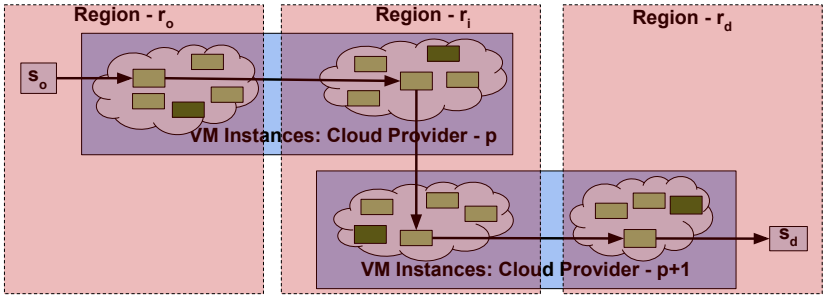


Figure 4.4: Deployment Across Multiple Cloud Providers

NetUber functions as a mediator between the two cloud providers to enable data transfer from $r_o \rightarrow r_d$ by interconnecting between the cloud providers in r_i. The inter-cloud interconnection at r_i incurs minimal latency thanks to the proximity of the multi-cloud servers, which may

even share the same co-location facilities or potentially be interconnected via a direct connection between the servers of the cloud providers. For example, AWS Direct Connect can offer a direct interconnection between a pair of AWS and GCP instances in r_i. Even the Internet-based connectivity between the cloud providers in the same region imposes minimal overhead due to the proximity of the cloud servers. Therefore, *NetUber* provides a higher performance point-to-point connectivity for the end users for data transfers to a geographically remote region, instead of connecting directly through an ISP.

NetUber needs to consider operational differences between the cloud providers for stable execution. For example, currently, an EC2 instance will be terminated by AWS when the current spot price exceeds the bid, with a 2-minute notice. GCP provides a 30-second notification, and Azure has proposed to offer the same. An EC2 spot instance is terminated either by the user or by AWS when the current spot price exceeds the user bid price or when the spot resource pool in an availability zone is overutilized. GCP terminates every spot instance 24 hours after it was started, as well as when there are inadequate resources in the spot markets to support the spot instances. *NetUber* avoids shutting down instances on its own for the sake of stability, except for terminating the additional instances after sustaining a significant spike in bandwidth demand.

A SaaS provider can use *NetUber*, instead of having geo-replicated deployments in multiple cloud regions which can be technically more challenging and more expensive. Besides, when state laws or organizational policies prohibit replicating data outside a given region, *NetUber* can be leveraged for data access with minimal data movement, yet with minimal redundancy. Furthermore, *NetUber* supports low-latency SaaS execution in more regions beyond those supported by any single cloud provider. For example, SaaS applications hosted in the region r_d can be accessed by an end user in r_o through *NetUber* more reliably than through the public Internet. Thus, a SaaS provider can exploit *NetUber* to create a point of presence in multiple regions while hosting the application in just a single region. Hence, *NetUber* can be a potential cost-efficient and high-performant alternative to geo-replicated solutions.

4.2.3 A Provider of Network Services

Hosting VNFs and SaaS on top of an overlay such as *NetUber* is straightforward as these applications directly consume the cloud resources. VNFs such as packet scrubbers, transcoder, firewalls, load balancers, and proxies, can be hosted in the spot VMs of *NetUber* as SaaS to perform middlebox actions to alter the data flow transferred atop the overlay. For example, forwarded data can be encrypted or compressed at an instance before the inter-region transfer, if prompted by the user, as additional services. Encryption enhances the privacy of the data transferred, while compression allows an economic transfer, with minimal latency as data can be compressed in-memory in the spot instances. We can host caching services in *NetUber* instances to optimize or limit WAN traffic. *NetUber* can also be used for content distribution or mitigation of Distributed Denial of Service (DDoS) attacks on the end user networks.

The choice of specific cloud instances depends on the offerings and cost model of each cloud provider. As of now, AWS Direct Connect offers maximum flat connectivity of 10 Gbps. AWS

recommends its memory-optimized R4 instances for in-network computations and stable network bandwidth. There are memory-optimized instances with either 10 GbE (i.e., cr1.8xlarge, r3.8xlarge, r4.8xlarge, r5.12xlarge, r5d.12xlarge, x1.16xlarge, and x1e.16xlarge) or 25 GbE instances (i.e., r4.16xlarge, r5.24xlarge, r5d.24xlarge, x1.32xlarge, and x1e.32xlarge). The 10 GbE instances can utilize their port speed completely with the maximum data rate of 10 Gbps offered by the AWS Direct Connect. On the other hand, currently, it is impossible to entirely utilize the ingress port of 25 GbE instances with a single AWS Direct Connect, as the current port speed is limited to 10 Gbps for the Direct Connect. *NetUber* finds the cheapest among the spot instances for a unit end-to-end (from an origin user server to a remote destination server) data transfer, as configured in the broker policies. It avoids selecting cheaper instances that offer slower end-to-end connectivity and consequently cost more to achieve the same data rate.

The memory-optimized 10 GbE and 25 GbE spot instances are ideal for computation-intensive network functions, as opposed to smaller unstable spot instances, due to their stable network with a promised data rate. As we observed over a period of 3 months, *NetUber* continued to leverage R4 instances of 10 GbE, primarily r4.8xlarge instances, among the given complete list of 10 GbE and 25 GbE memory-optimized instances. These optimized r4.8xlarge instances have sufficient memory (244 GB memory each) and CPU resources. Thus, with a relatively stable memory, computing, and networking resources across the regions, *NetUber* can be used as a framework for third-party network services on a cloud platform. The routing policies of *NetUber* spot instances are primitive, due to the limited knowledge and control available to the cloud users regarding the underlying cloud backbone network. With the awareness of the cloud provider network, we can improve the inter-region routing efficiency of *NetUber*.

4.3 Economic Models for Cloud-Assisted Connectivity

Cloud providers list their VM prices at an hourly rate though they charge per second. *NetUber* follows the same pricing scheme since it acquires the cloud instances that are the core of its infrastructure on a per-second basis. Since the connectivity providers list their charges per-month, we assume one month as the total time in our models, for a fair comparison.

Data Transfer Cost: Equation 4.1 presents $\lambda_{o,d}$, the end-to-end unit data transfer cost from s_o to s_d using a single cloud provider. Currently, cloud providers do not charge for incoming data from the Internet or another region. The *NetUber* end users incur a cost, D_o and D_d, to connect their servers s_o and s_d to the cloud provider. For example, AWS Direct Connect charges the end user per used port-hours, at an hourly rate. λ_{r_o,r_d} and λ_{r_d,s_d} refer to the unit data transfer cost between the cloud regions (r_o and r_d) and between the r_d and s_d.

$$\lambda_{o,d} = D_o + \lambda_{r_o,r_d} + \lambda_{r_d,s_d} + D_d \tag{4.1}$$

Equation 4.2 extends Equation 4.1 to compute the cost incurred in a multi-cloud scenario involving two cloud providers, as depicted by Figure 4.4. $\forall p, p+1 \in P \subset \mathbb{Z}^+$, $\lambda^{(p)}$ denotes the unit data transfer cost by the cloud provider p. The instance of the *cloud provider p+1* in r_i is just an external server connected through the public Internet (ISP-based) or a dedicated direct

connectivity as far as the *cloud provider p* is concerned, whereas it functions as the origin cloud server from the perspective of the *cloud provider p+1*. Thus, the cost associated with the *cloud provider p* is denoted by $\lambda^{(p)}_{r_o,r_i} + \lambda^{(p)}_{r_i,s_i}$, whereas the cost associated with the *cloud provider p+1* is denoted by $\lambda^{(p+1)}_{r_i,r_d} + \lambda^{(p+1)}_{r_d,s_d}$. $\forall p, p+1 \in P$, $D_i^{(p|p+1)}$ denotes the cost of interconnection between the *NetUber* spot instances of cloud providers p and $p+1$ at the region i. $D_o^{(p)}$ and $D_d^{(p+1)}$ are typically paid directly by the end user to the cloud provider, as such direct connects are set up directly by the end user. However, $D_i^{(p|p+1)}$ is paid by *NetUber* to one of the two providers who manages the direct connection from their instances to the other provider's instances at i. We leave addressing the challenges associated with creating and maintaining such direct connects between the spot instances of multiple cloud providers as a future work.

$$\lambda^{(p,p+1)}_{o,d} = D_o^{(p)} + \lambda^{(p)}_{r_o,r_i} + \lambda^{(p)}_{r_i,s_i} + D_i^{(p|p+1)} + \lambda^{(p+1)}_{r_i,r_d} + \lambda^{(p+1)}_{r_d,s_d} + D_d^{(p+1)} \tag{4.2}$$

Equation 4.3 generalizes Equation 4.2 for the scenario of *NetUber* exploiting multiple cloud providers for a single data transfer. When there is no overlap between the regions of the origin and destination cloud providers, $p, p+n \in P$, *NetUber* require data transfer through several intermediate cloud providers. However, we observe that currently the number of cloud providers and their regions are limited and are mostly overlapping across the providers and as such, this scenario remains mostly futuristic. Here, r_{oz}, r_{dz}, s_{dz} depict the origin region, destination region, and destination external server, from the perspective of $z \in P$.

$$\forall z \in [p, p+n] \subset P : \lambda^{(p,p+n)}_{o,d} = D_o^{(p)} + \sum_{z=p}^{p+n} \left(\lambda^{(z)}_{r_{oz},r_{dz}} + \lambda^{(z)}_{r_{dz},s_{dz}}\right) + \sum_{z=p}^{p+n-1} \left(D_i^{(z|z+1)}\right) + D_d^{(p+n)} \tag{4.3}$$

Total Cost: We formulate the total cost C from all the providers for *NetUber* data transfer, in Equation 4.4. C consists of the cost associated with acquiring the spot VMs and the cost of data transfer. $c_{p,r,v,t}$ defines the cost for a spot instance from the cloud provider from a region at a given time interval between t_0 and t_f. Since the spot price continues to fluctuate, the cost to acquire the required number of spot VMs ($v \in V$) in each of the regions ($r \in R$) is calculated as a time integral over their execution time, and summed for all the instances from each region of all the cloud providers. The data transfer cost is billed by the cloud provider per the volume of data transferred. Therefore, it is calculated by a time integral of data rate b_t through a cloud path to its completion. Since $\lambda_{o,d}$ denotes the unit data transfer cost involving all the cloud paths, we calculate the total data transfer cost from the first *NetUber* instance that receives the user traffic, $\forall v \in V_{r_o}$, for each region of all the cloud providers.

$$C = \sum_{p \in P} \sum_{r \in R} \left[\sum_{v \in V} \int_{t_0}^{t_f} c_{p,r,v,t} dt + \sum_{v \in V_{r_o}} \int_{t_0}^{t_f} (\lambda_{o,d} b_t) dt \right] \tag{4.4}$$

Effective Data Rate: We observed that the data rate of the inter-region data transfers b_t is proportional to the network interface (β) of the instance. β is 10 Gbps or 25 Gbps in the 10 GbE and 25 GbE instances used by *NetUber*, respectively. However, it is impossible to reach the full network interface capacity in the inter-region data transfer. Cloud data transfers between

regions have a degradation from the promised network interface. We define the degradation in the data rate of inter-region data transfer as a ratio of the network interface of the pair of VM instances, $\chi_t \in (0, 1)$. The actual data rate $b_t = \beta \times (1 - \chi_t)$.

Data compression at the source can significantly reduce the costs, given that cloud providers do not charge for the incoming data. Many cloud compression tools, general purpose or optimized for specific file formats, make lossless compressions feasible at the time of the cloud transmission [355]. By compressing the data before the inter-region transfer, we can significantly increase throughput or the actual data transferred per unit time. We define a compression ratio, γ_t as the percentage of size reduction from compression without incurring data loss. γ_t and χ_t vary with time, unpredictable to *NetUber*. Equation 4.5 illustrates the effective data rate b.

$$b = \frac{b_t}{1 - \gamma_t} = \frac{\beta \times (1 - \chi_t)}{(1 - \gamma_t)}; \chi_t, \gamma_t \in (0, 1) \tag{4.5}$$

***NetUber* Cost Model:** *NetUber* proposes to charge its end users based on their requested bandwidth (b), the length of the bandwidth usage (τ), and a direct unit (per time unit, per unit data rate) cost ($\Lambda_{o,d}$) to acquire the instances and data transfer from the cloud provider. $c_{\bar{p},r_i}$ defines the cost of acquiring intermediate instances (from the cloud region i) from any provider z for the cloud transfers involving multiple providers. This cost will be 0 if *NetUber* overlay is built with just one cloud provider ($|\bar{P}| = 1$), as there will be no inter-cloud data transfers that require an intermediate cloud instance as a mediator across the cloud providers. β defines the network interface of the instance. To find the instance cost per unit data rate, we divide the cost of instances by the capacity of their network interface. Thus, Equation 4.6 defines the total cost per unit data rate.

$$\Lambda_{o,d} = \beta^{-1} \times \left(c_{p_o,r_o} + c_{p_d,r_d} + \sum_{|\bar{P}| \geq 2} c_{\bar{p},r_i} \right) + \lambda_{o,d} \tag{4.6}$$

NetUber defines the charge (typically billed per second and charged monthly) for its end user $u \in U$ as a cost function f, as illustrated by Equation 4.7. The total cost for *NetUber* consists of the cost of acquiring spot instances and the data transfer costs. The cost function is defined dynamically, such that the total income of *NetUber* from its end users in the specified time frame for their connectivity demands remains higher than its total cost. This cost scheme ensures that the *NetUber* meets its monthly profit margin of ϵ.

$$\exists \epsilon > 0 : \sum_{u \in U} f(\tau, b, \Lambda_{o,d}) - C > \epsilon \tag{4.7}$$

Both technological and economic aspects have shaped *NetUber* design decisions and how *NetUber* purchases the spot instances. Various approaches have been proposed, to reap the economic benefits, while addressing the technical challenges inherent to the volatile nature of spot instances. *NetUber* exploits the differentiated pricing of various availability zones for a relatively stable overlay. While *NetUber* bids in multiple regions to acquire VMs in geograph-

ically distributed locations to host the virtual routers, it cannot use migrations between VMs in different regions for cost efficiency, as it will, in turn, increase the bandwidth consumptions and the number of hops. We see *NetUber* as the first complete research to propose a practical approach that separates the network from the infrastructure to function as a virtual connectivity provider. *NetUber* lets a third-party cloud user create an inter-cloud architecture to offer connectivity to end users, ensuring the economic and technical feasibilities.

4.4 Evaluation

In this section, we aim to answer two questions: i) when is *NetUber* more cost-efficient than connectivity providers?, and ii) how does the performance of *NetUber* compare to using ISP-based public Internet paths? We deployed a prototype of *NetUber* on AWS to evaluate its long-term performance and stability, and to answer the identified questions. We configured the policies to choose the memory-optimized cloud instances that are the cheapest per second and data rate. We executed our experiments continuously for a period of 3 months (April - June 2017) to find the monthly costs, mean performance indicators (such as throughput and latency), or the variations (such as jitter), appropriately.

Prototype development and deployment: We performed an initial assessment on several pairs of origin and destination regions to choose and maintain the necessary spot instances for a stable *NetUber* overlay. *NetUber* policies chose r4.8xlarge spot instances (each with 10 GbE network interface) as the primary instances across all the AWS regions that we evaluated throughout the 3 months. A single TCP connection between the instances of any two regions reached around 50 Mbps. With parallel connections, *NetUber* achieved 1.2 Gbps of maximum stable inter-region bandwidth between the pair of 10 GbE instances functioning as virtual routers. We confirmed that the obtained maximum bandwidth was independent of the origin and the destination regions[2]. Therefore, we deployed the *NetUber* prototype on at least 9 pairs of r4.8xlarge spot instances to achieve 10 Gbps bandwidth between two regions. We evaluated the other instance types such as r3.8xlarge and confirmed that no instances offered a lower degradation χ_t than the r4.8xlarge instances, hence justifying the choice of r4.8xlarge.

Infeasibility of using smaller or on-demand instances: We considered smaller spot instances and on-demand instances as potential alternatives to the R4 spot instances that *NetUber* primarily uses. The smaller instances such as C3 have a moderate network. We found two issues with these moderate network instances: i) we need to acquire a lot of them, which is more complicated to maintain due to the need for a substantial number of parallel connections, and ii) they are very unstable. We noted that with the cost of acquiring the 10 GbE instances, we could have around 2 - 4 Gbps, yet unpredictable, inter-region bandwidth with numerous moderate-network spot instances. The R4 instances r4.8xlarge and r4.16xlarge have 10 GbE

[2]We repeated the same experiments after 16 months from our original evaluation reported in this section, with the same instance types of 10 GbE. As of 2018 October, with up to 128 parallel connections, we achieved up to 4.8 Gbps between remote cloud regions and 441 Mbps between our servers and remote cloud regions. We believe that the cloud paths will continue to improve with time, thus making *NetUber* more competitive.

and 25 GbE interfaces, respectively. However, unlike these 10 Gigabit and 25 Gigabit memory-optimized instances, the other instances (low, moderate, high, up to 5 Gigabit, and up to 10 Gigabit network instances) offered no promised guarantees for the bandwidth.

We were able to obtain the R4 spot instances promptly while having to wait for up to one hour to acquire a large number of moderate network instances required to meet our bandwidth demand. The r4.8xlarge instances of *NetUber* stayed alive throughout the experiments that lasted up to 3 months, while smaller instances shut down intermittently. Thus, it was possible to have a stable overlay over the 10 GbE and 25 GbE memory-optimized spot instances, whereas it was not feasible over the smaller ones. Our evaluations confirmed that there was no difference in the quality of paths (concerning throughput, latency, loss rate, and jitter) between the on-demand and spot instances of the same instance type. Thus, we observed that using smaller on-demand instances provide worse data rate or a much higher cost than using the 10 GbE spot instances. Therefore, we could not favor the on-demand instances for the stability that they supposedly offer, compared to the spot instances. We indeed note that if the spot price reaches the on-demand price or more, *NetUber* can choose to spawn on-demand instances for its overlay network. However, we estimate that such a situation would be economically disadvantageous to *NetUber* as it would incur higher monetary costs in acquiring and maintaining the cloud instances, even though it is technically a trivial task to achieve.

4.4.1 Economical Alternative to Connectivity Providers

To evaluate the cost efficiency of *NetUber* for its end users, we compare the data transfer cost of *NetUber* from Frankfurt to Sydney, against the offerings of 2 connectivity providers in the EU/US regions, identified as CP-1 and CP-2 in Figure 4.5. Due to cost restrictions, our extensive study on the actual economic cost only covers this pair of EC2 regions. However, based on the past approximate spot instance pricing details we gathered, we believe that our findings are applicable for any pair of regions. We also consider a compressed data transfer with *NetUber* in the evaluations, accounting for a potential compression-as-a-service deployment in the spot instances. We report the cost of the instances as the average price during the period.

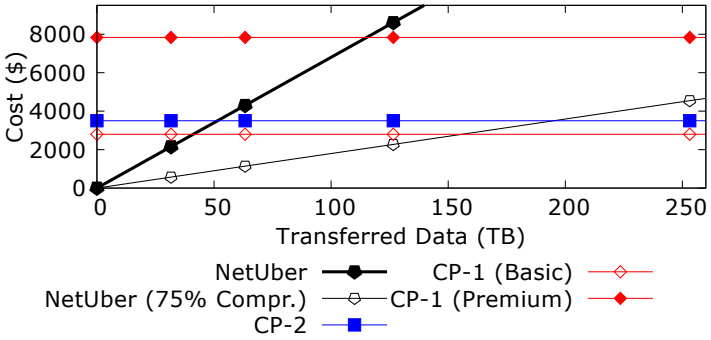


Figure 4.5: Monthly Fee for 10 GbE Flat Connectivity

We obtained price quotes of the connectivity providers CP-1 and CP-2 via private email queries. CP-1 is an infrastructure provider with an extensive, geographically-distributed infras-

tructure that offers connectivity as an alternative to transit providers. It provides two options: a basic scheme to connect to regular networks choosing the cheapest paths, and a more expensive scheme to connect to premium networks and large IXPs for better throughput and shortest routes. CP-2 is a transit provider. As these quotations are not public (as typically transit providers do not publicly list the prices), we must refrain from disclosing the providers. *NetUber* cost includes the costs of acquiring spot instances, data transfer costs, and the AWS Direct Connect cost for a continuous data transfer of the given volume.

Figure 4.5 depicts the minimum price of *NetUber* as the cost charged by EC2 for the spot instances and the data transfer, considering regular ($\gamma_t = 0$; $\chi_t = 0.88$) and compressed ($\gamma_t = 0.75$; $\chi_t = 0.88$) data transfers. Up to 75% of lossless compression has been reported in compressing real-time streaming data [147]. We demonstrate a similar (0.75) or higher γ_t with in-memory compression at r_o. With these values, b = b_t / (1 - 0.75) = $\beta \cdot (0.12) / (0.25) = 0.48 \cdot \beta = 4.8$ Gbps, from Equation 4.5. Here, $b_t = 0.12$ is derived from the observation that 1.2 Gbps was received from the 10 GbE R4 instances. The inter-region data transfer achieves only $0.12 \cdot \beta$ (or $0.48 \cdot \beta$ with compression). AWS Direct Connect offers a stable port speed, which can be as high as a constant value of 10 Gbps. Thus, AWS Direct Connect reaches the data rate of β, since $\beta = 10$ Gbps for our desired r4.8xlarge instances.

Up to 3164.0625 TB/month (10 Gbps = $10/8 \cdot 3600 \cdot 24 \cdot 30$ GB/month) of data can be transferred between a pair of instances with 10 GbE interface. For volumes of data transfers up to at least 50 TB, *NetUber* always offered a competitive price (compared to the benchmarked connectivity providers) and remained globally profitable. When 75% compression is assumed, *NetUber* was still cheaper up to 200 TB. By leveraging the availability of abundant memory in the memory-optimized instances (such as the R4 instances that *NetUber* primarily uses), we can employ enhancements profitable to *NetUber* or execute additional VNFs as value-added services on the data traffic.

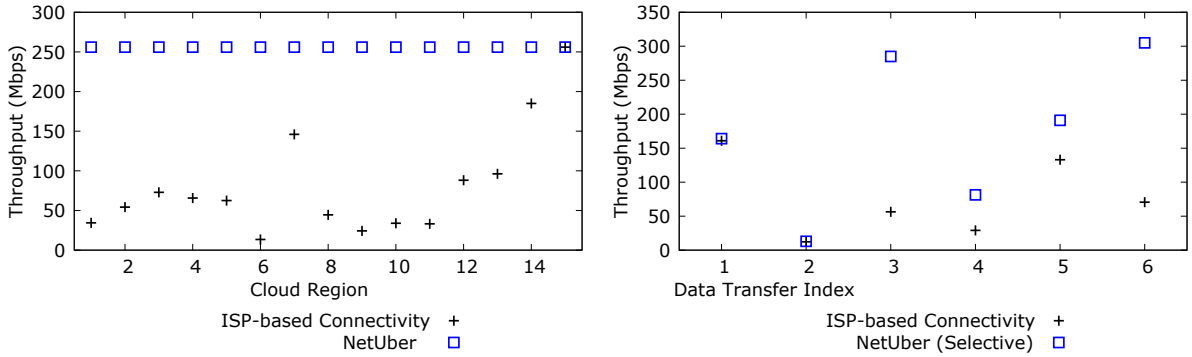
4.4.2 Higher Performance Point-to-Point Interconnection

NetUber is not always cheaper. But can it perform better when it is equally or more expensive than using the standard Internet-based connectivity? We benchmark the throughput, latency, and jitter of *NetUber* in various cases, against exclusively using the Internet-paths.

Throughput: We measured the throughput of *NetUber* against that of using the public Internet access provided by the ISPs, in transferring data between two distant servers s_o and s_d. s_o is a server close to the cloud region r_o. It is connected to the Internet via an ISP. s_d is a cloud server in the region r_d. We first measured the throughput of data flows between s_o and s_d directly via the public Internet ($T(s_o, s_d)_I$). Then we measured of throughput of *NetUber*, ($T'(s_o, s_d)$), via the *NetUber* cloud overlay between r_o and r_d. We connected s_o to r_o via the public Internet (thus with a throughput of $T(s_o, r_o)_I$). Throughput of *NetUber* is limited by $max(T(s_o, r_o)_I, T(r_o, r_d)_C)$. Here, $T(r_o, r_d)_C)$ refers to the data rate of *NetUber* overlay between the cloud regions.

Figure 4.6a illustrates the achieved throughput in sending data from our server in Atlanta to cloud servers in various regions, first directly via the ISP-based connectivity, and then via

NetUber. In the case of *NetUber*, we first send the data to a specific cloud region (typically the one that is proximate to the origin server) such that we achieve the highest data rate from our origin server to the cloud overlay. Then, *NetUber* forwards the data via the cloud overlay to the cloud server in the destination region. We observed North Virginia to offer the highest data rate in our experiment, as it provided 256 Mbps when connected to our server via the ISP. We note the geographical proximity of our server to North Virginia region as a potential influencing factor in providing the high throughput. Since the cloud overlay offered 1.2 Gbps, the path between the origin server and the origin cloud region remains the bottleneck in the *NetUber* data transfer.



(a) Atlanta to Various Cloud Regions

(b) Selectively Using *NetUber* Cloud Overlay

Figure 4.6: Throughput of *NetUber* with ISP-based cloud connect

We observe that connecting the cloud servers from far regions via the ISP offered lower throughput. ISP and *NetUber* provide the same throughput when $r_o = r_d$, i.e., the destination server is in the closest region to the origin server (as can be seen by the cloud region 15 in Figure 4.6a which refers to North Virginia). By routing the data traffic via the high-performing path consisting of Atlanta $\rightarrow$ North Virginia, we exploit *NetUber* to offer a higher end-to-end data rate compared to ISP. Furthermore, *NetUber* exploits the cloud path as well as the faster connectivity to the nearest cloud region to provide a uniform data rate to the cloud servers, regardless of their regions.

We then configured the *NetUber* deployment to send data between two geo-distributed endpoints, selectively leveraging the cloud overlay for part of the route. Figure 4.6b illustrates that such selective use of *NetUber* led to a better or equal throughput to using ISP for the data transfer entirely. The data transfer exploited the overlay only when routing through the overlay provided a better throughput, while resorting to entirely using the ISP-based public Internet paths when no cloud region was proximate to the origin or destination to offer better performance. Therefore, we note that one can even use *NetUber* selectively, such that only the transfers with a cloud region en route (such as the presence of a cloud region close to the origin or the destination) would go through the *NetUber* overlay to achieve better latency and throughput whereas the other data transfers resort to their default connectivity provider.

Latency: We benchmarked *NetUber* along with ISPs for faster Internet routes, against using just an ISP. We compare the Round-Trip Time (RTT) latency (i.e., ping time) between two

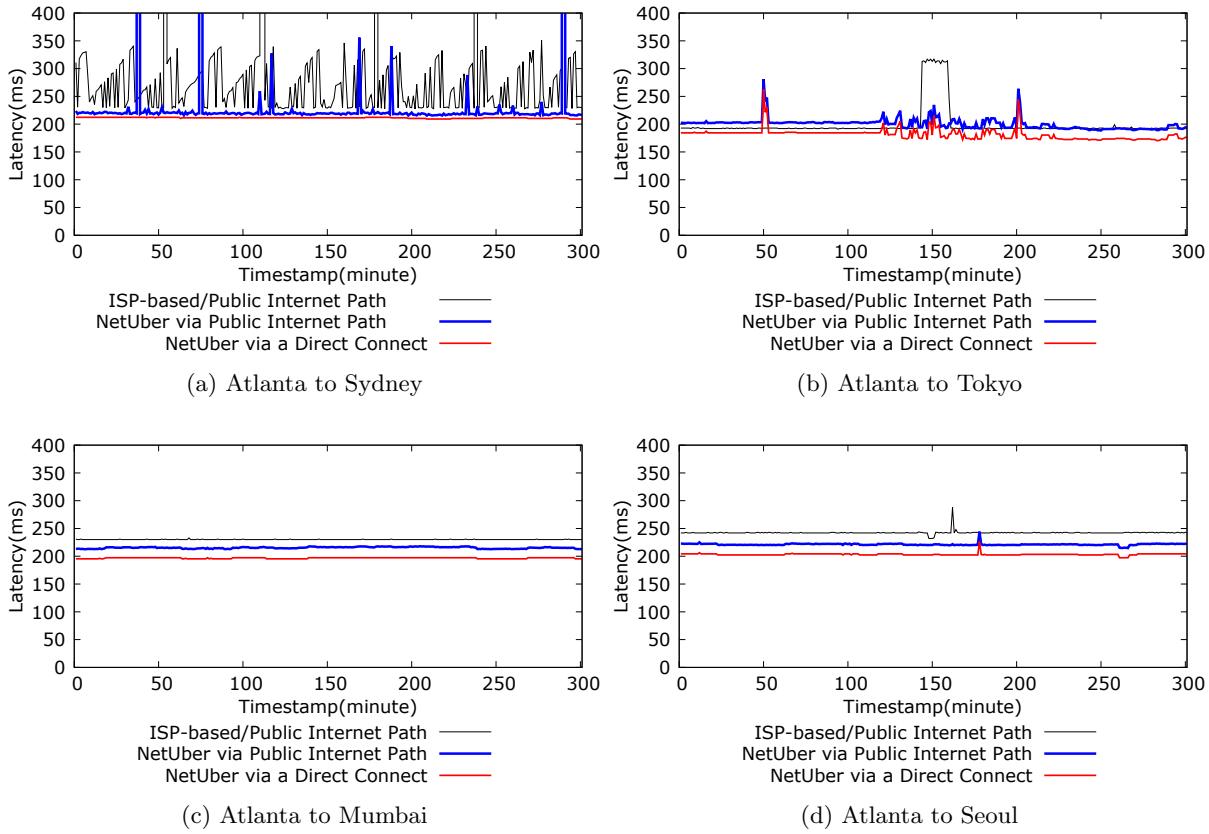
Table 4.1: Ping Times (ms): Regular Internet vs. *NetUber*

Origin → Destination	**Direct**	***NetUber (via)*** Improvement
Vladivostok, RUS → São Paulo, BRA	362.72	307.08 *(Tokyo)* 15.34%
Hobart, Tasmania, AUS → Mumbai, IND	347.22	248.41 *(Sydney)* 28.46%
Seoul, KOR → São Paulo, BRA	321.72	299.31 *(Seoul)* 6.97%
Tashkent, UZB → Singapore, SGP	351.61	258.57 *(Mumbai)* 26.46%
Nairobi, KEN → Tokyo, JPN	403.87	386.37 *(Mumbai)* 4.33%
Frankfurt, DEU → Tokyo, JPN	296.87	237.34 *(Frankfurt)* 20.05%
Thuwal, SAU → Tokyo, JPN	346.01	324.30 *(Frankfurt)* 6.27%
Prague, CZE → São Paulo, BRA	224.90	221.40 *(Frankfurt)* 1.56%
Nuuk, GRL → Sydney, AUS	415.02	352.46 *(Canada)* 15.07%
Fairbanks, AK, USA → Mumbai, IND	441.57	435.64 *(Canada)* 1.34%
São Paulo, BRA → Paris, FRA	239.45	210.72 *(São Paulo)* 12.00%
Tacuarembó, URY → Montreal, CAN	203.42	186.01 *(São Paulo)* 8.56%

endpoints that connect through *NetUber* against the latency using Internet-based connectivity of ISPs. For the geographically distributed servers, we used RIPE ATLAS Probes [33] as well as our physical servers, all connected to the Internet via an ISP. We sent pings between the server endpoints, first through the ISP, and then via *NetUber* by entering the overlay through the nearest AWS region. We repeated the evaluations ten times and listed the average ping times (in milliseconds) in Table 4.1, along with the AWS region that the ping is routed through for *NetUber*, as well as the improvement when using *NetUber*. In all the cases, we observe that going through *NetUber* overlay offered better latency (up to 30% improvement) than directly connecting through the ISP, as long as a cloud region exists relatively near to the origin server, en route to the destination.

Jitter: Finally, we benchmarked the jitter of *NetUber* against that of the ISP-based internet connectivity by observing the variations in latency. We modeled data transfer from Atlanta to Sydney, Tokyo, Mumbai, and Seoul, via the cloud region of North Virginia. North Virginia was chosen based on our previous observation of the highest data rate. We considered two scenarios of *NetUber* in connecting our server in Atlanta to the cloud servers. First, via the ISP-based connectivity, and then via a dedicated link (such as the AWS Direct Connect) that we modeled. Figure 4.7 shows the latency variations as observed during intervals of 5 hours (measured over different periods in a day), for the data flow between the 4 pairs of origin and destination.

We observe that *NetUber* offers minimal latency and jitter across all the destination cloud regions. On the other hand, when using the ISP-based public Internet paths entirely, the jitter significantly relies on the time of the experiment as well as the destination. The cloud paths were more stable than the Internet paths, with minimal latency and jitter. However, we observe a higher jitter and latency (even though still lower than using the public Internet paths entirely) when we configured *NetUber* with ISP rather than a dedicated link between the origin server and the cloud overlay. This observation indicates that the connection between the origin to the nearest cloud server via the ISP contributes more to the jitter when using *NetUber* without a direct connect. Even when *NetUber* demonstrated a high jitter due to the variation in cloud



(a) Atlanta to Sydney

(b) Atlanta to Tokyo

(c) Atlanta to Mumbai

(d) Atlanta to Seoul

Figure 4.7: Latency (RTT) variations of *NetUber* and the ISP-based Internet paths

overlay network as shown by the data transfer between North Virginia and Tokyo (Figure 4.7b), using public ISP to connect the origin and destination led to an even larger jitter.

Loss rate: Our pings indicated that the cloud paths of *NetUber* did not contribute to packet loss, and showed a positive impact on minimizing the packet losses. We observed a loss rate of 1.33% when data was transferred from Atlanta to Sydney via the public Internet paths, and 1% when *NetUber* was used in conjunction with the ISP to connect the origin server to the cloud region. However, *NetUber* with direct connect incurred 0% loss. All the other regions had a 0% loss rate in all 3 cases.

The results indicate that even without dedicated connections to the cloud provider, an ISP user can resort to *NetUber* for several reasons. First, *NetUber* still offers better latency, jitter, loss rate, and throughput for data transfer. Furthermore, it can offer better access to SaaS hosted in a far cloud region, compared to directly connecting through the user's ISP. However, *NetUber* demonstrates an even better performance when a dedicated connection such as Amazon Direct Connect connects the servers to the overlay. Similarly, *NetUber* can also be used in conjunction with Fiber to the home (FTTH) [307], and community-based initiatives [32] for faster Internet routes.

4.4.3 Qualitative Assessment

We note that the results presented depend on the regions and the cloud provider. Although we observed a stable network with the R4 spot instances over three months, we note that this depends on the market and demand of the spot instances, with significant dependence on how the spot markets will evolve in the future, concerning pricing and policies from the cloud provider. Similarly, the evaluations on performance such as latency, throughput, jitter, and loss rate depend on how the Internet paths behave as well as the performance of the cloud backbone network, over time. Our extensive evaluations are indeed limited to three cloud regions due to the limited cloud credits (i.e., the monetary cost to spend on the cloud resources) that were available to us. However, based on the historical data of cloud pricing, we note that the results can be generalized for any pair of cloud regions.

Virtual connectivity providers that do not control the infrastructure have been proposed, with an approach similar to that of *NetUber* [121, 360, 60]. *NetUber* focuses on leveraging the cheap spot instances, and thus offers an economical approach to deploy on a large scale. Software-Defined Internet Architecture (SDIA) [284] and *NetUber* share the goal of connecting endpoints on the Internet regardless of the underlying infrastructure. However, *NetUber* focuses on network virtualization and does not alter how the underlying physical network works. Consequently, *NetUber* can be deployed on existing cloud providers without any modifications to the cloud networks. While Jingling [132] delegates network functions to third parties, *NetUber* virtualizes the entire network with virtual routers running atop spot VMs, by a third party broker. Both *NetUber* and Jingling do not have control over the exact physical location of the system. Thus, specifying policies of the end users and identification of the cloud instances should be done through the service layer instead of the physical address. Moreover, cloud resources of *NetUber* can be leveraged for more than just connectivity, including network services such as caching, content distribution to multiple local subscribers, and data analytics over wide-area networks [337].

It will be a test of time to note how long a cloud-assisted network such as *NetUber* continue to be a competitive connectivity provider, technically and economically. We note the potential for the cloud provider themselves to operate such a cloud-assisted network overlay, thus limiting the market potential for a third-party overlay network provider such as *NetUber*. The inter-cloud architecture remains a more convincing use case for a third-party connectivity provider, which we leave as future work for deployment and evaluation across two or more cloud providers. An inter-cloud architecture would require cloud direct connects between the VMs of the cloud providers in multiple overlapping regions of the cloud providers. Consequently, this would need *NetUber* to purchase the cloud direct connect from one cloud provider from each pair of cloud providers that interconnect with each other in each region. We also leave Internet measurements across all the countries and the impact of *NetUber* on those regions as future work. Cloud direct connects to such remote regions would be challenging and impossible without a cross-country collaboration that spans the Internet. However, that would make a complete case for the challenges and opportunities for cloud-assisted networks as a connectivity provider.

4.5 Conclusion

Connectivity providers limit their agreements regarding minimum duration and scale, preventing end users with short-term (in the scales of minutes, as opposed to months), or low bandwidth requirements. We built *NetUber* as a cloud-assisted overlay that offers end-to-end short-term connectivity in smaller-scales to address this shortcoming. *NetUber* runs atop spot instances purchased from cloud providers for a low price. In this chapter, we built a case on why a virtual connectivity provider without any fixed resources may not just be technologically feasible, but also be economically sustainable. We presented case studies with *NetUber* as i) an economical alternative to connectivity providers for data transfers up to 50 TB, ii) a higher performance alternative to ISPs for inter-region data transfer, and iii) a provider for network services. We observe the enhancements concerning throughput, latency, jitter, and loss rate in comparison to ISPs and cheaper data transfer for small decentralized enterprises.

Our analysis of the spot instance prices indicates that the cost for a cloud-assisted overlay network depends on several factors, including geographical locations of the endpoints and current demand for the cloud instances in the region. The exact instance types to choose for a stable network and competitive pricing heavily depends on the time. As time passes by, the cloud providers may change their instance types, pricing schemes, or the policies regarding spot instances. While these changes may positively or negatively affect the feasibility to operate *NetUber* profitably, we highlight that currently, it is a possibility considering both monetary cost and performance. Furthermore, we observe that after 16 months since our original analysis presented in this chapter, the cloud inter-region bandwidth has improved, thus making *NetUber* even more competitive. We believe that the network softwarization achieved in the global-scale through the use of spot instances can provide innovations in the network industry, and may even attract cloud providers to offer their connectivity solutions given that they currently have a large unified global network presence.

5 SDN Middlebox Architecture for Resilient Transfers

Cloud providers share their infrastructure among various tenants, yet often with a separate virtual execution environment for each of their tenants. The network flows belonging to the tenant executions can be of different priorities based on the nature of their respective application. Cloud providers should ideally pass the tenant policies to the network control plane to ensure that the network flow scheduling approaches align with the application layer policies of each tenant for enhanced control of their network flows. Specifically, cloud data centers should schedule the critical tenant workflows on time adhering to the tenant policies, despite congestion in the network nodes or links. However, traditionally, networks are managed at each layer independently, from the physical layer to the application layer. Cross-layer optimization across the cloud and data center environments are essential to guarantee SLAs and QoS to the various tenant processes at the network level.

Networking research on SDN and middleboxes has shown the potential to propagate application-level policies to the cloud network. Software and hardware middleboxes offer additional capabilities to the network rather than merely forwarding or routing the packets of the network flows [71]. FlowTags [120] software middlebox provides the ability to tag the flows from the application layer. It enables seamless integration of other middleboxes into an SDN network, by ensuring that the tags in the network flow packets can remain unaltered despite the presence of various middleboxes in the network. Thus, we posit that software middleboxes such as FlowTags can be used in parallel with the SDN controller to translate the tenant policies and requirements into network flow rules, persist them in the network flow across the data plane devices en route as tags, and interpret them for the routing decisions. However, leveraging the collective potential of SDN and middleboxes for complete SLA-aware multi-tenant networks, incurring minimal overhead, remains an open research challenge.

This chapter proposes *SMART*, an **S**DN **M**iddlebox **A**rchitecture for **R**esilient **T**ransfer of critical network flows in multi-tenant cloud networks. *SMART* focuses on exploiting the global knowledge and control of the data plane devices readily available to the SDN controller to cater to the requirements of the tenant applications. *SMART* aims to offer a resilient transfer for critical flows, based on tenant preferences received at the network layer from the processes or applications executing on the servers. By employing a selective redundancy in a controlled manner, *SMART* guarantees timely end-to-end delivery and QoS for critical tenant application flows.

SMART middlebox architecture lets the tenant applications tag their critical flows (i.e., flows with a higher priority) with policies and SLOs. Intermediate nodes, which are typically switches in the flow path, read and interpret the tags as tenant network policies consisting

of thresholds such as maximum permissible time for flow completion and user-defined QoS parameters for the other properties at the origin node. *SMART* further defines soft-thresholds at a fraction of the respective threshold. The controller receives a notification when a critical network flow violates a soft-threshold, typically caused by congestion in the network flow path. Subsequently, *SMART* diverts the packets from a subflow of the flow in an alternative route to the destination, or clones and routes the subflow in an alternative route along with the original flow. Thus, *SMART* mitigates the impact of network congestions on critical flows, and thus reduces the SLA violations.

SMART opens a research question on how network-level enhancements such as diverting or cloning the flows can be beneficial for the application requirements and attempts to answer it by benchmarking against the traditional network routing in the cloud and data centers that are agnostic to the tenant application policies. Our evaluations highlight that the overhead imposed by *SMART* concerning redundancy and processing of network flows are negligible as *SMART* only manipulates the network flows of higher priority, a relatively small fraction in a data center network. Moreover, only the first packet of the first offending flow is forwarded to the SDN controller to alter the routing tables, per the OpenFlow protocol. We thus observe that *SMART* ensures SLA in the critical network flows through its selective redundancy and the ability to pass the application-level policies to the network-level through SDN and middleboxes, with minimal overhead.

This chapter is composed of the contents of the publications: [W3, S1].

5.1 *SMART* Approaches for Critical Network Flows

SMART identifies congestion through a perceived delay in flow completion time. It picks an intermediate node in the flow path as the **breakpoint node** to execute the proposed approaches. Further, in the network flow, *SMART* identifies a particular packet as the **breakpoint packet**. Finally, it creates a subflow starting from the breakpoint packet until the end of the flow.

5.1.1 *SMART* Alternative Approaches

SMART consists of three alternative approaches.

1. Diverting Approach: The diverting approach routes the subflows of a few selected critical flows in an alternative path to the destination when it expects an SLA violation due to a congested node or link in the original route. It diverts the subflow in single or multiple alternative paths excluding the original path. Choosing several routes in the diverting approach will be useful for higher priority critical flows when no specific alternative route can be considered the best alternative. As the preferred alternative routes may be longer or suboptimal than the original route, and as there is a need to reconstruct the flow, there is a potential time overhead or delay. The diverting approach will not incur duplicate packets if it diverts the subflow in just one alternative direction. However, diverting to multiple alternative paths will have duplicate packets similar to the following two approaches.

Table 5.1: Time and Bandwidth Overhead

Approach with n number of diverts/clones of critical flow packets	Duplicate Packets as a function of n	Potential Time Overhead
$Divert(n)$	$(n-1)$ * (0, 100]%	Possible
$Clone(n)$	n * (0, 100)%	No/Negligible
$Replicate(n)$	n * 100%	No/Negligible

2. Cloning Approach: *SMART* employs the cloning approach for critical flows with a higher priority, by cloning the flows selectively, rather than merely diverting them. The cloning approach duplicates the subflow following the breakpoint and routes it in single or multiple alternative routes. It leaves the original flow to continue unmodified in its route while cloning and routing subflows in an alternative route towards the destination. The updated rule in the breakpoint node ensures sending the packets in the original route as well as an alternative route. As the original flow is left to continue in its original route unmodified, cloning approach does not have a time overhead, yet there is bandwidth overhead.

3. Replicating Approach: In the cloning and diverting approaches, the controller clones or diverts the packets that follow the breakpoint packet, by changing the routing rules for the packets of the critical flows in the breakpoint node. The replicating approach, on the other hand, replicates the entire violating flow from the origin to the destination in single or multiple alternative routes. Cloning and replicating are further enhancements to the diverting approach, as the original route could end up being the better choice if the congested links or nodes recover during the transmission. Similar to the cloning approach, the replicating approach also does not have a time overhead. Replicating entire flows imposes, however, 100% of duplicate packets until the routing is complete. We can consider the replicating approach as a special case of the cloning approach, which clones the entire flow instead of a subflow starting from a breakpoint. We can also configure the replicating approach to drop the original flow, as in the diverting approach. However, that may introduce a time overhead while reducing the duplicate packets.

Table 5.1 summarizes the potential overhead for the critical flows in completion time and bandwidth usage. We measure the time overhead as a delay that may happen in the flow completion compared to the unmodified flow. Here n refers to the number of times the packets are cloned, diverted, or replicated. We can adaptively use the three approaches in conjunction, rather than defining them statically to be mutually exclusive. An adaptive/clone approach would invoke the cloning procedure for the subflow when a flow meets a soft-threshold, and then replicate all the subsequent flows of the same path and route them towards an alternative path(s) until the network path recovers from the congestion. This adaptive approach mitigates the overhead and technological challenges of branching and merging the flows, while also reducing the inherent overhead of using the replicating approach as the sole enhancement.

5.1.2 Clone Destination

When the network encounters congestion in certain paths used by the critical flows, *SMART* decides the exact destination to recompose the subflows based on the characteristics of the

congestion. *SMART* recomposes the flows at either the destination of the original flow or the next node following the identified congested network path. In a large data center with a few nodes identified to be contributing to the congestion, the cloned or diverted subflow can be routed towards the node that immediately follows the congested link or node, to avoid routing in a sub-optimal path when the congestion affects just one or a few of the nodes in the original route. The decision to recompose the flow at the earliest possible node also minimizes redundant packets by early recomposing of the original flow. If there are no such nodes or links identified to be contributing to the congestion, or if the network topology does not support an efficient data flow to the next node, *SMART* routes the cloned or diverted subflow towards the original destination in an alternative route.

Figure 5.1 illustrates a network with data flow between two nodes such as servers or smart devices, with multiple potential paths connecting them. In a data center network, these nodes are hosts or servers, while the intermediate nodes are traditionally switches that connect the underlying network. However, due to the heterogeneous nature of CPS and MEC environments, origin or destination can be smart mobile devices/terminals or virtual execution spaces in the controller, while intermediate or destination nodes can be surrogate nodes such as edge nodes or switches in a data center. With the dynamic traffic of network flows, a few services or network nodes and links may become congested. *SMART* identifies the congested, malfunctioning, or malicious nodes and links (highlighted as unhealthy in Figure 5.1) through its controller, by monitoring the responsiveness of the nodes.

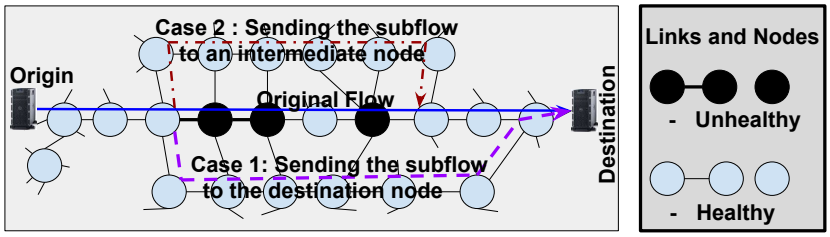


Figure 5.1: Subflows and Alternative Execution Paths

For mission-critical CPS such as ICU medical monitoring systems, the critical services should execute correctly and timely. In such workflows, when an intermediate node fails or becomes slow, the controller can enforce redundancy selectively in the data flows to ensure correctness and end-to-end delivery. *SMART* alters the paths of critical flows towards a healthy alternative dynamically when the current path fails. It creates subflows by diverting or cloning parts of the flows and sends them towards the *clone destination*. It then reconstructs the flow from the subflows at the clone destination. The clone destination can be as same as the original destination of the flow, or an intermediate node in the flow path. In case 1 identified in Figure 5.1, the clone destination is the same as the original destination of the flow. While this scenario may lead to a higher level of redundancy due to the presence of duplicate or replicated subflows, it avoids the need to manipulate the network flows at the intermediate nodes. However, case 2 has a clone destination that differs from the original. Here *SMART* sends the cloned subflow towards an intermediate node (on the original path connecting the origin and destination), in an alternative path, and recomposes the flow afterward. The case 2 minimizes unnecessary redundancy when

it is possible to recompose the flow at an intermediate node. When such a recomposition of flows (i.e., making flows reconverge) is impossible or inefficient at an intermediate node due to the technical difficulties, or due to the nature of the congestion or network failure itself, *SMART* recomposes the flow eventually, when it reaches the destination node, as in the case 1.

5.1.3 *SMART* Architecture

Figure 5.2 depicts the cross-layer deployment of *SMART*. In the application layer, the ecosystem consists of tenant applications and application processes. Applications can be of various priorities - some of them more critical and time-sensitive than the others. *SMART* forwards the information on priorities to the underlying network as policies, along with the control flow, by extending and leveraging the SDN controller. The deployment consists of switches, servers hosting the tenant applications, and the middleboxes in the network layer. In the network layer, data flows across the tenant processes are present as packets and flows.

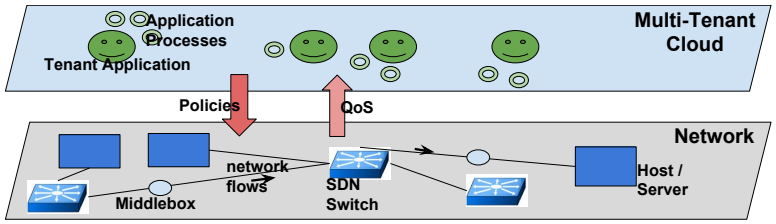


Figure 5.2: Application and Network Views of a Cloud Deployment

SMART exploits the monitoring capabilities offered by SDN, while extending the complementary features provided by middleboxes to include the information on SLA parameters, in the form of tags attached to the packets of critical flows. *SMART* adopts the FlowTags software middlebox to tag the flows with minimal overhead, as no other existing SDN-based approach enables per-flow custom policy enforcement in a network with the presence of software and hardware middleboxes. The FlowTagger deployed on the nodes tag the packets of the critical flows at the origin node, whereas the FlowTagger at the nodes en route destination reads the tags.

The *SMART* architecture includes the soft-thresholds at a fraction of the thresholds (i.e., hard-thresholds) as SLA parameters. Upon meeting a soft-threshold, based on the policy and the length and the priority level of the flow, either the flow is replicated and rerouted from its origin to the destination in an alternative route, or it is cloned or diverted from a breakpoint node. Critical flows can be all the flows of a given user, all the flows originating at a given node, or a set of flows following a policy defined by the tenant at the application layer. With a little redundancy, *SMART* attempts to meet the deadlines of the critical flows, by mitigating the potential SLA violation caused by congested nodes in the initial route.

Figure 5.3 depicts the higher level architecture of *SMART*. We extended OpenDaylight as the base SDN controller. We adapted FlowTags as a software middlebox inside each node of the data plane as well as a FlowTags controller. The *SMART* middlebox, consisting of the

FlowTagger, resides inside the nodes that are the origins of the flows. FlowTagger reads and writes tags to the packets.

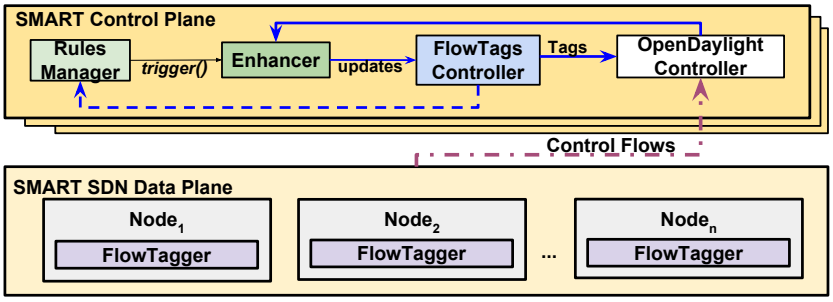


Figure 5.3: *SMART* Architecture

FlowTagger and ***Rules Manager*** are *SMART* components that are developed as FlowTags-capable software middleboxes. While in a typical FlowTags deployment an existing middlebox such as an Intrusion Detection System (IDS) is extended to read and write the tags, these *SMART* components perform no network function other than handling the tags and communicating with the FlowTags controller by invoking its API to generate or consume the tags in a unified manner, as presented by the FlowTags architecture [120]. *SMART* tags also include the current timestamp to track the time consumed in routing so far which can be used to estimate other optional information such as estimated monetary cost and energy consumption.

The FlowTags controller parses the tags from the packets forwarded to the controller. Policies, thresholds, and business rules are read and stored into the *SMART* controller from the configuration files, as defined in the network by system administrators or the tenants. Rules Manager gets and parses the rules from the tags, and triggers the ***SMART* Enhancer** according to the defined policies. *SMART* Enhancer consists of the enhancement algorithms that would wrap the base routing algorithms to enhance them. We designed the FlowTags controller, *SMART* Enhancer, and the rules manager as extensions to the OpenDaylight SDN controller.

Along with the other rules set by the SDN controller, the custom user-defined tags in the packets read from FlowTagger are interpreted as policies, which the packets should respect. Upon a violation of the policies in any of the nodes, the first packet of the violating flow is sent to the *SMART* controller and triggers it. The controller sets a breakpoint in the flow on the packet that triggered the controller and the location node of the packet when the violation occurred. The breakpoint node or packet can also be chosen algorithmically from the application layer.

5.2 *SMART* Algorithms

The *SMART* Enhancer consists of algorithms to mark the breakpoint and enforce its enhancements on the critical network flows. Algorithm 3 describes the *SMARTRoute*, the procedure of the *SMART* Enhancer. For the ease of expression, the rest of this section assumes

the cloning approach to be the default, while inherently referring to both cloning and diverting approaches.

Algorithm 3 *SMART* Enhancer Route

```
1: procedure SMARTRoute(flow, origin, dest)
2:     repeat
3:         BaseRoutingAlgorithm(flow, origin, dest)
4:         if (flow.policies.isThresholdMet()) then
5:             cloneOrigin ← markBreakPoint(flow, origin, dest)
6:             cloneDest ← findCloneDest(flow, flow.status)
7:             clonedFlow ← cloneFlow(flow, cloneOrigin, cloneDest)
8:             flow.status.update(cloneDest, cloneOrigin)
9:         end if
10:    until (flow.allReceived(cloneDest) ∨ flow.allReceived(dest))
11:    mergeFlows(flow, clonedFlow)
12: end procedure
```

The BaseRoutingAlgorithm (line 3) refers to any underlying routing algorithm such as Dijkstra's shortest path algorithm or Equal-Cost Multi-Path (ECMP) algorithm, which is to be enhanced by *SMART*. *SMARTRoute* routes the flows from the origin to the destination entirely using the BaseRoutingAlgorithm unless a critical flow meets a soft-threshold. The thresholds can be system-wide policies, such as minimal throughput and latency, in the network system and individual flow level. A skyline approach [54] is assumed in the presence of conflicting tenant-specific, flow-specific, or system-wide policies, to find a compromise considering all the requirements. The line 4 checks for the violation of soft-thresholds by the critical network flows to invoke the *SMART* enhancements on the flow.

The *markBreakPoint()* (line 5) chooses the clone origin consisting of a node and a packet as the breakpoint node and packet respectively. The *findCloneDest()* (line 6) decides the clone destination based on the flow and its status consisting of information potentially related to the policy violation. The cloneFlow() (line 7) clones or diverts the subflow, starting from the breakpoint packet to the rest of the flow with the breakpoint node as the origin. The *flow.status.update()* (line 8) updates the controller with the current status of the flow as the middlebox architecture reads the tags. The flow status consists of the information crucial for the reconstruction of the original flow at the flow destination, such as the sequence number and the original parent flow.

Flow Reconstruction: The flow reconstruction phase waits until all the packets necessary to recompose the original flow arrive at the clone destination or the final destination (line 9). Once the minimum packets (i.e., a copy of each packet composing the original flow) required to reconstruct the flow reach the clone destination, the mergeFlows() (line 10) reconstructs the original flow. If the clone destination is different from the original destination, the recomposed flow continues in its original route towards the destination. *SMART* leverages the sequence numbers and the status indicating the parent flow from the flow packets in reconstructing the flow. Once *SMART* has reconstructed the original flow at the clone destination, it drops the duplicate packets on the fly.

The cloning approach minimizes the extent of the necessity to reconstruct the flow. If all

the packets from the original flow arrive before the packets from the clone, *SMART* drops the cloned packets en route. For the diverting approach, and the cloning approach if the packets of the cloned flows arrived earlier, *SMART* will reconstruct the flow by merging the packets from the diverted or cloned subflow to the packets of the original flow that have already reached the clone destination.

The following critical flows of the same route may be replicated and rerouted, or diverted at the origin, in an alternative route. Thus, while a fraction of the initial short flows may still violate SLAs due to the time overhead imposed by the cloning and recomposing of the flows, following flows will be able to avoid the offending path altogether. When flows created by *SMART* replicate the entire flows, the clone destination considers only the first of the flows to arrive. As the replicating approach resends the whole flow from the origin to the destination in one or more alternative routes, the necessity for recomposing and flow manipulation is avoided, albeit with more redundancy.

***SMART* Breakpoint:** Breakpoint is a pointer to the node and the flow where *SMART* clones the subflow. The controller chooses the breakpoint dynamically and writes rules on the breakpoint nodes to divert or clone the upcoming packets of the critical flows. Information on breakpoints is not stored statically in the flows or the controller beyond the time frame of subflow construction. Algorithm 4 elaborates the procedure of choosing the breakpoint node and packet.

Algorithm 4 Marking the Breakpoint

```
1: procedure markBreakPoint(flow, origin, dest, policies, links)
2:     for all (link ∈ flow.route) do
3:         if (policies.isThresholdMet()) then
4:             breakPoint.node ← current.node
5:             breakPoint.packet ← current.packet
6:             Return breakPoint
7:         end if
8:     end for
9:     breakPoint ← flow.estimate(policies.breakPolicy)
10:    Return breakPoint
11: end procedure
```

First, the algorithm checks for each link in the flow route (line 2), whether a soft-threshold is met (line 2 - 3). If a specific node or a link is estimated to be responsible for the policy violation, the algorithm marks the node as the breakpoint node (line 4) and the current packet as the breakpoint packet (line 5). Then the algorithm returns the breakpoint (line 6). If no specific malfunctioning link or node identified, the delay is due to either i) network congestion across multiple nodes and links or ii) the flows being much larger than the typical flows in the data center and hence taking longer to complete the routing. In these cases, the breakpoints depend on policies. Therefore, if the algorithm did not identify any breakpoint in a specific link, it performs an estimation for a breakpoint from its knowledge of the network (line 9), and finally returns the breakpoint (line 10).

5.3 Implementation

We prototyped *SMART* by extending the OpenDaylight SDN controller with a FlowTags controller deployment to make the priority tags readable by the SDN controller. Originally a POX extension, we redesigned the FlowTags controller as an OSGi bundle to deploy in the Apache Karaf container of OpenDaylight, due to the extensible modular architecture of OpenDaylight. We built a simple software middlebox to tag the flows and then to read and interpret the tags from the control plane, rather than developing a complete reimplementation of FlowTags for OpenDaylight. We developed the controller extensions as independent OSGi bundles and deployed them alongside the controller core bundles. We emulated the data plane consisting of the nodes and middleboxes with Mininet through Python scripts.

Figure 5.4 depicts the *SMART* deployment, separated into a i) control plane consisting of the FlowTags-capable SDN controller and *SMART* components, and a ii) data plane consisting of the nodes - servers/hosts and OpenFlow-capable switches. The switches construct the network by connecting each other as well as the servers where the flows originate. We deployed FlowTags controller, Rules Manager, and the Enhancer in the Karaf container as OpenDaylight bundles. We developed the *SMART* control plane modules using Oracle Java 1.8. We implemented network flow routing algorithms commonly used in data center and cloud networks, such as the shortest path algorithm. *SMART* applies its algorithmic improvements on top of these base algorithms.

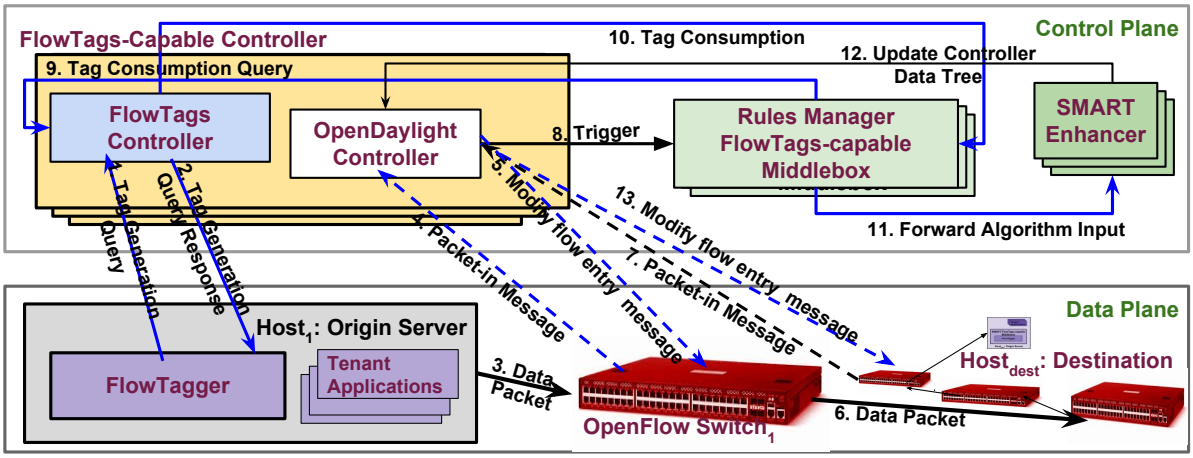


Figure 5.4: *SMART* Deployment and Execution

FlowTagger is a generator/writer of the tags, similar to the FlowTags-capable middleboxes for Network Address Translation (NAT). We deployed the FlowTagger in each of the hosts. It tags the packets of a selected subset of flows, defined as the critical flows according to the tenant applications, leaving the other flows unmodified. Thus *SMART* initializes the entire *SMART* enhancement workflow only on the critical flows as identified from the application layer. Following the packet processing walkthrough for tag generation, FlowTagger initially sends the tag generation query to the FlowTags controller and receives the tag generation query response. FlowTagger modifies the packet headers accordingly, and the data packets continue in the original

flow path through the switches. Switches in the flow path communicate with the OpenDaylight controller through the OpenFlow API. The switches send a packet-in message to the controller, and in turn, the switches receive the modify flow entry message from the controller. The data flows through the intermediate nodes/switches. Later, at a policy violation, as identified from the tagged packets of the critical flows, an OpenFlow message invokes the controller, further triggering the Rules Manager in the control plane.

Rules Manager is a consumer of tags, similar to the FlowTags-capable firewalls that read and interpret the tags. As SDN sends only the first packets of the violating flows to the controller, the control plane becomes the ideal location to retrieve the tags from the controller and read them by the Rules Manager, rather than having the Rules Manager as an additional middlebox in the data plane. Rules Manager sends the tag consumption query and receives the tag consumption response from the FlowTags API. It further forwards the contextual information of the packets of the flow in question to the *SMART* Enhancer, which is responsible for computing the routing decisions and propagating them to the SDN controller. Thus, the *SMART* Enhancer updates the OpenDaylight's data tree, the data structure storing distributed objects inside the controller. Accordingly, the controller updates the flow tables based on the Enhancer output to divert, clone, or replicate the relevant subflows or flows.

5.4 Evaluation

In this section, we aim to assess the efficiency of *SMART* in offering SLA-awareness, abiding by tenant-specified thresholds. We focus on flow completion time as the threshold.

Evaluation Environment: We emulated a data center network with around 1000 nodes, on a cluster with 6 identical nodes (Intel® Core™ i7-2600K CPU @ 3.40GHz processor and 12 GB memory), with Mininet. We used leaf-spine topology [14] as it offers multiple potential alternative paths between the pairs of nodes. Since leaf-spine topology of typical data centers has precisely two-hops, we extended it with longer path lengths. We emulated network flows across the network topology and congestion across specific links, identifying a few as critical while having the rest as regular flows. We then assess how *SMART* can omit the underperforming paths for the critical flows, to avoid SLA violations that would happen in the base routing approach.

Flow Completion Time: *SMART* deploys its approaches when it can incur a positive enhancement in flow completion time for the critical network flows as illustrated by Equation 5.1.

$$\exists \epsilon > 0 : \Delta T = T^o - \{T_{det} + T_{update} + T_d + T'\} > \epsilon. \tag{5.1}$$

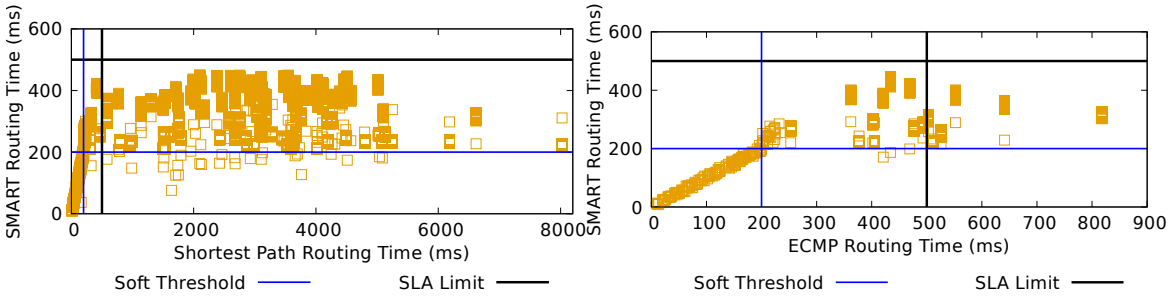
Here, ΔT - Enhancement in flow completion time for a critical network flow.
T^o - Estimated flow completion time, if the flow continues in the original congested path.
T_{det} - Time taken to detect and report soft-threshold violation to the controller.
T_{update} - Time taken to update the flow table rules in switches.
T_d - Time overhead at destination to recompose the flow.
T' - Minimum flow completion time with the cloned/diverted subflow in an alternative path.

The distributed extended OpenDaylight controller deployment of *SMART* was quick to detect and update the policy violations. Subflows still respect the ordering of packets with sequence numbers and flow IDs interpreted by *SMART* at the clone destination. Hence, reconstruction of the original flow at the destination is straightforward, dropping the duplicate packets. We estimate T_{det}, T_{update}, and T_d to be on the scale of milliseconds, which can be safely ignored from Equation 5.1 for long-running flows.

We configured *SMART* to have an adaptive clone approach, cloning of subflows followed by replicating the entire subsequent flows. For flows that take less than 1 second to complete routing, subflows of the violating flow, as well as the following critical flows in the same route, are cloned at the breakpoint, if they enhance the flow completion time of critical flows by a positive value upon meeting a soft-threshold. Otherwise, instead of cloning the violating flow, the *SMART* prototype replicates all the following critical flows at the origin and route the replicated flows towards the destination in an alternative path. Thus, this adaptive approach minimizes redundancy and overheads in recomposing the flows, while ensuring that the following critical flows will abide by the SLA, even if the first few flows may have violated the SLA. Replicating the critical flows at the origin also avoids the potential overhead *SMART* may impose by cloning in breakpoint for shorter mission-critical flows.

We benchmarked the flow completion time of the base routing approaches against the *SMART* enhancements over those base approaches for the critical flows. Figure 5.5a plots the flow completion time with shortest-path against the *SMART* enhancement. Each hollow square in the plot indicates a network flow, while the filled ones indicate 10 flows. We observe that there was no SLA violation with *SMART* enhancements for the critical tenant flows, while several of them violated the SLA with the base shortest-path algorithm in the presence of congestion.

We then repeated the experiment with ECMP as the base routing algorithm in place of shortest-path. Figure 5.5b shows the time taken to route the critical tenant flows with and without *SMART* enhancements. *SMART* improved the performance by cloning the critical flows in an alternative route readily available in ECMP, and replicated the following critical flows of the same congested path in the new route. We observed that in both cases, *SMART* ensured not to exceed the SLA limits for the critical flows, even when the base approach violates the SLA limit indicated as a hard-threshold in the tenant network policies.



(a) Shortest-Path (b) ECMP

Figure 5.5: Adaptive Clone/Replicate: *SMART* Enhancements vs. Base Algorithm

SMART avoided the SLA violations by up to 95%, in the modeled network environment.

Majority of the flows that initially violate SLAs abide by the SLA with *SMART* enhancements. Performance of the controller and switches in detecting the violations, and updating the rules, contributes to the potential SLA violations. However, by carefully considering a soft-threshold in addition to the hard-threshold, *SMART* avoided causing SLA violations to a critical flow that does not violate the SLAs in the base approach without the *SMART* enhancements. Abiding by a soft-threshold indicates that the existing route is adequate to meet SLA and no congestion observed along the path of the critical flow. Therefore, unless a critical flow meets its specified soft-threshold, *SMART* did not invoke its enhancements.

How well *SMART* fares in avoiding SLA violations for critical flows, of course, rely on several factors, including the percentage of the flows that are marked as priority flows and the definition of the SLA (in this case, how short the routing time is). We observe that there is a dynamic, minimum value in routing time that can be defined as the SLA limit. This value depends on the nature of the network and how fast the network flows can be cloned and recomposed. We evaluated *SMART* with the SLA limit of 500 ms. If we reduce this time slightly, *SMART* still functions reasonably well, as it clones the subflows during the fraction of the SLA limit, defined as the soft-threshold. However, if we reduce the SLA limit further, the overheads in network flow manipulations will become prominent, thus making *SMART* inefficient. Furthermore, significantly reduced soft-thresholds will also lead to a higher level of redundancy in the subflows, and consequently flooding the network and adversely affecting the flow scheduling for regular flows as well as the critical flows.

Assessment of Overheads: *SMART* exhibits an adaptive behavior to the nature of the congestion, finding the right time to clone or divert. The data plane devices perform the routing with minimal intervention from the controller unless a critical flow meets its soft-threshold. We observed that *SMART* cloned only around 16.7% of the packets of the critical flows in the modeled data center networks. The overall redundancy is further smaller, depending on the fraction of the flows that are considered critical. Thus, we observe that the *SMART* subflows do not contribute to the congestion themselves. We estimate the overhead to be lower than 100 ms in switches when *SMART* manipulates the breakpoints and updates the flow tables, with minimal overhead in the bandwidth. As we have integrated the *SMART* enhancement algorithms and FlowTagger with the SDN architecture, we observe no significant overhead by the deployment of *SMART*.

Qualitative Assessment: Several previous works, including MPTCP and the work that was built upon MPTCP, constitute a significant motivation behind the architecture of *SMART*. *SMART* borrows its subflow handling mechanisms from the MPTCP design. Among the other related works, QJump [142] and *SMART* focus on SLA for the higher priority flows through bypassing the traditional network routing. However, *SMART* adaptively leverages redundancy in addition to 'jumping the queues.' Conga [14] aims for a congestion-aware load balancing via a flowlet switching. Conga flowlets are similar to *SMART* subflows. However, Conga does not support differentiated SLA or QoS guarantees. The research works that leverage MPTCP or flowlets do not use redundant subflows for a reliable transfer of network flows. They also do not prioritize the flows based on user preferences to satisfy SLAs of critical flows. In addition to addressing these research gaps, *SMART* proposes an extended SDN and middlebox based

architecture for resending or cloning the subflows, if a flow has not reached the destination within the stipulated time. Hence, *SMART* aims at enhancing the SLA-awareness for the critical flows, through its extended SDN architecture that supports passing extended contextual information to the network plane from the tenant applications.

5.5 Conclusion

Enterprise clouds and data centers limit their optimizations to typically at individual layers, rather than aiming for cross-layer optimizations. Multi-tenant clouds consist of applications that are of different priority levels. The priority levels are either mandated by the SLAs or are indicated by the tenants for their application processes. Research on SDN and middleboxes aim to improve the data centers in various network aspects such as congestion control and delivery guarantees. Despite these promising developments, enterprise multi-tenant network scheduling approaches fail to adequately reflect the complex tenant requirements received from their applications at the network level.

We developed *SMART* as an SDN and middlebox architecture for multi-tenant clouds, by diverting or cloning subflows of critical flows for timely delivery in a network with congested links. *SMART* imposes a selective redundancy on the critical network flows of the tenant applications as a mean to improve their QoS. *SMART* propagates tenant policies to the network layer to equip the tenants with more control over the shared virtual network. Preliminary evaluations highlighted the efficiency of *SMART* in offering SLA-awareness to data center networks and applications, through its cross-layer enhancements.

SDN implementations offer limited opportunities to pass the user preferences from the applications to the network, due to the limited extensibility and scope of SDN protocols such as OpenFlow. On the other hand, a FlowTags middlebox implementation enables more descriptive and extensible data input from the applications hosted on the servers for cross-layer enhancements. *SMART* depends on both SDN and FlowTags. Therefore, its applicability in the real world depends on the prevalence of SDN data centers compliant with FlowTags. Consequently, we note that currently, *SMART* is more research-oriented, rather than an approach that can be deployed on the enterprise data centers with minimal effort. We see *SMART* as a first step in leveraging redundancy more efficiently together with the capabilities to dynamically change the network routing and pass the application level user policies to the network through an SDN architecture extended with a software middlebox.

III

Service-Oriented Architecture

6 Software-Defined Service-Compositions

Scheduling service composition workflows across multi-domain networks abiding by the tenant policies is a challenging task. Service compositions enable complex eScience workflows and enterprise business processes, by chaining the outputs of several services. The volume and distribution of data and the services that access and process the data continue to increase. The service composition workflows execute on diverse computational nodes, ranging from servers to lightweight smart devices, which are geographically distributed at Internet scale. Service providers face the challenge of finding the optimal deployment node for their services among these various geo-distributed nodes. On the other hand, the end users who consume these services have their intents and policies such as maximizing the throughput and uptime of their service workflows, while minimizing the monetary cost and end-to-end latency. The service providers specify and ensure that the requirements for efficient and uninterrupted execution of their services are met. However, an optimal service composition should meet the demands of the tenants that consume the workflows, in addition to the individual requirements of each service instance as well as the overall policies specified by the service providers.

The increasing demand for access to data and computing resources makes geo-distributed service composition workflows more prevalent. Enterprise and eScience workflows are typically composed of several web services and microservices. Workflows of mission-critical applications consist of redundancy in links as well as alternative implementations, often developed and maintained by multiple providers. Service providers deploy each of their service implementations across several nodes as alternative service instances, to ensure performance, scalability, fault-tolerance, congestion control, and load balancing. Distributed cloud computing [112] and volunteer computing [22] are two examples that permit multi-tenant computation-intensive complex workflows to execute in parallel, leveraging distributed resources. Nevertheless, the existing workflow scheduling approaches do not cater for the scheduling of tenant workflows abiding by their policies across multi-domain networks.

A service execution should be able to migrate between possible service deployments when a service instance fails to respond to the incoming service requests within a specified time interval. An inter-domain migration is necessary when the controller fails to resolve such service execution failures caused by issues such as network congestion, inside the same network domain due to the limited availability of services or other resources in the domain. While web services are implemented using several approaches, languages, and frameworks, they still offer interoperable SOA and RESTful APIs. These APIs unify the message passing between the services and enable seamless migration among the potential service instances in a best-effort and best-fit strategy.

In this chapter, we propose Software-Defined Service Composition (SDSC), an SDN-based

service composition approach for efficient service composition and workflow placement. First, by separating the execution from the data plane of the overall system, SDSC facilitates integration and interoperability of more diverse implementations and adaptations of the services. Second, we extend SDN with MOM to support network-aware scheduling of service composition workflows in multi-domain wide area networks. Finally, by deploying the service composition workflows over an extended SDN architecture, SDSC gives the controller an overview of the service composition environment such as the service instances and the servers that host the services. The SDSC distributed controller architecture thus possesses an increased control over the underlying network, while supporting the dynamic scheduling of the service composition workflows from various traditional web services engines and the distributed execution frameworks.

SDSC leverages both the service statistics of the web services engines and the network-awareness of the SDN controller to find the best-fit service instances to schedule each tenant workflow execution, among various service implementations and deployments. Each network domain is aware of the service requests served by its service instances through the extended SDN controller architecture. The controller maps the service requests to the underlying network and provisions the network resources accordingly to the service instances. The controller monitors congestion and failure of the underlying computing nodes and links, and dynamically remove or demote the malfunctioning nodes among the alternatives. MOM protocols facilitate the communication and coordination among the SDN controllers of different domains. Leveraging the MOM architecture of SDSC, each SDN controller propagates the changes in the network of its domain to the relevant SDN controllers of other domains, based on their subscriptions. Thus, SDSC fine-tunes the services placement, supports dynamic resource allocation for each service request, and facilitates intra- and inter-domain service execution migration.

We build *Mayan*,[1] an SDSC framework that exploits the existing web services engines and distributed execution frameworks such as MapReduce. *Mayan* i) facilitates an adaptive execution of scientific workflows and ii) enables a very large-scale reliable service composition by finding and consuming the current best-fit among the multiple service instances in a wide area network. While a distributed SDN controller architecture can facilitate a global network-awareness across multiple network domains, such an architecture is mostly infeasible as enterprises are in practice reluctant to share the network status and statistics with another domain or a third-party entity. Through its SDN controller architecture federated with MOM, *Mayan* enables collaboration and coordination without depending on such a static network hierarchy. Our evaluations highlight that *Mayan* can enhance the scalability of the service compositions in a context-aware manner.

This chapter is composed of the contents of the publications: [C2, W1, B2].

6.1 SDSC Model for Service Composition Workflows

Service providers deploy instances of a web service or a network function, enabling the service users to consume them. A workflow exploiting multiple services can be composed by choosing

[1] *Mayan* is a mythical architect, commonly known as Mamuni Mayan in Tamil literature.

an instance of each relevant service from the available service instances. Service instances can be identical implementations of a service deployed in different server locations or endpoints, or entirely different implementations of a service definition offering the same functionality yet following different programming languages and paradigms.

$$\forall n \in \mathbb{Z}^+, \forall \alpha \in \{A, B, \ldots, N\} : s^n_\alpha \implies \text{Implementation } \alpha \text{ of service } s^n. \tag{6.1}$$

Service compositions chain the output of a service as the input of another. Equation 6.2 represents a sample service composition S where the outputs of both s^1 and s^2 invocations are provided as the input to s^3. Here, s^1 and s^2 can be executed in parallel, while s^3 will have to wait until both s^1 and s^2 return their respective output, as it depends on those service responses as its input.

$$S =< s^3, (< s^1, Input^1 >, < s^2, Input^2 >) > \tag{6.2}$$

Figure 6.1 illustrates a potential to deploy a workflow across several environments (A, B, .., Z) as a service composition. $s^{(2,3)}_Z$ represents a service that is functionally equal to the service composition of $s^3_A \circ s^2_A$, the output of s^2_A as an input to s^3_A. Hence, it is not an alternative to s^2_A or s^3_A. Not all the services have an implementation in considered environments, as indicated by the lack of C for s^2. When the service instance s^3_A is either congested or crashed, the service execution is migrated to s^3_B, the next best-fit implementation of s^3 for the workflow S_A.

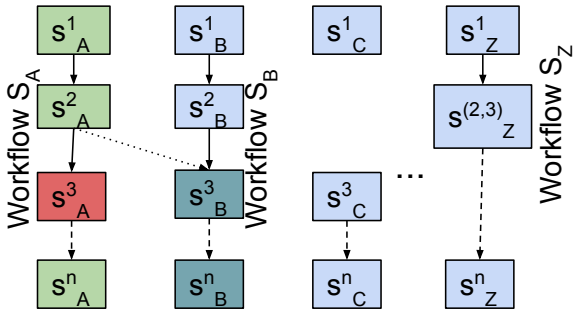


Figure 6.1: A sample representation of multiple alternative workflow executions.

Furthermore, each implementation of a service can have multiple deployments, distributed across the globe, either as replicated deployments or independent deployments by different service providers, as illustrated by Equation 6.3.

$$\forall m \in \mathbb{Z}^+ : s^n_{\alpha m} \implies m^{th} \text{ deployment of } s^n_\alpha. \tag{6.3}$$

A service composition workflow S can be represented as a composite function of its services, which are a subset of services available to compose a workflow. We consider N different service implementations and a varying number m_α of deployments for each implementation of s^x. Equation 6.4 derives the minimum number of execution alternatives for s^x.

$$\begin{aligned} &\forall x, y, z \leq n \in \mathbb{Z}^+ : S = s^x \circ s^y \circ \ldots \circ s^z. \\ \kappa_x \implies &\text{The minimum number of execution alternatives, } \forall s^x \in S. \\ &\therefore \forall s^x \in S : \kappa_x = \sum_{\alpha=A}^{N} m_\alpha. \end{aligned} \tag{6.4}$$

η_S represents the number of alternative execution paths for each service composition S. The service that has the minimum alternatives limits the minimum number of potentials for any given service composition workflow, as shown by Equation 6.5.

$$\eta_S \geq \min_{x \leq n} \kappa_x \geq 1. \tag{6.5}$$

The maximum number of alternatives is limited by a product of alternatives for each service, taking into account the alternatives due to various service combinations in the service composition as illustrated by Equation 6.6.

$$\eta_S \leq \prod_{x=1}^{n} \kappa_x. \tag{6.6}$$

Equation 6.7 thus summarizes our observations from Equation 6.5 and Equation 6.6.

$$\min_{x \leq n} \kappa_x \leq \eta_S \leq \prod_{x=1}^{n} \kappa_x. \tag{6.7}$$

It is necessary to choose the best-fit service deployments among the service composition alternatives for a given workflow, abiding by the user policies. For example, consider the scenario of parental control for web browsing. The web traffic may go through a firewall and virus scanner in a regular mode but go through the firewall and parental control before reaching the virus scanner before arriving at the children's browser sessions. Subsequently, a service composition workflow can be represented by multiple alternative forms as shown by Figure 6.2. The flow is divided and sent in two different paths: one going through s_1 and s_2 and the other just s_1, before both merging to s_3 as in the above parental control scenario. The flow is then directed to s_4, where based on the policies, a certain matching subset of data reaches the user directly while the remaining goes through s_5. This service composition can be defined as $(s_1 \circ s_2 + s_1) \circ s_3 \circ (s_4 \circ s_5 + s_4)$, or it can be reduced to $s_1 \circ (s_2 + 1) \circ s_3 \circ s_4 \circ (s_5 + 1)$.

These multiple representations of the service composition workflow increase the potential execution alternatives while also supporting parallel execution with redundancy in the paths. SDSC aims at context-aware service compositions by exploiting the service and network statistics from the web services engines and the SDN controller.

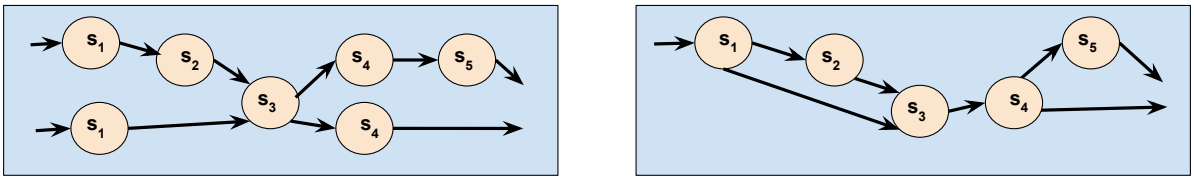


Figure 6.2: Parallel Execution Alternatives of a Service Composition Workflow

6.2 Solution Architecture

Mayan enables communication between the services and the network control plane through MOM messages. *Mayan* configures the web services engines to identify the status changes, such as an interruption in service availability and congestion in a workflow execution path, and pass them to the controllers that have subscribed to the topic. In this section, we will look into the *Mayan* controller deployment, its layered architecture, and its core algorithms.

6.2.1 *Mayan* Controller Farm

In its core, *Mayan* consists of a **Controller Farm**, a federated SDN deployment in a wide area network that enables communication and collaboration between multiple network domains via MOM. A single centralized global controller deployment (including the distributed controller architectures or hierarchies that are still managed by a single entity, thus effectively an administratively centralized one) is infeasible at Internet scale due to administrative and political hurdles that prevent managing data plane devices belonging to several providers and domains by a single entity. Therefore, *Mayan* proposes a federated controller deployment that comprises numerous controllers from different domains or organizations in a wide area network. Each controller has protected access to the controllers of the other domains comprising the federated controller deployment, facilitated through a MOM publish-subscribe [116] model.

The Controller Farm is a loosely connected federated deployment of controllers, without a static hierarchy or topology. Each controller in the Controller Farm communicates with other controllers in a wide area network flexibly through messages. The Controller Farm manages the communication and coordination across all the entities from the multiple domains. It controls the inter-domain workflows through subscriptions and messages between the controllers of each domain. In addition to the proposed federated controller deployment to manage the inter-domain workflows, each domain typically consists of a cluster of controllers instead of a stand-alone one, to prevent the controller from being the single point of failure or a bottleneck in each domain. In case of a stand-alone deployment, the controller is typically hosted on a dedicated server to avoid overloading the server with other computing workflows and applications. However, it is still possible to host the controller in a server that is also shared by the web services engines or a distributed execution framework, especially when either the number of servers or the web service execution workload is minimal.

By leveraging SDN extended with MOM, the Controller Farm aims to provide a seamless

scaling with the problem size. A controller instance connects to the Controller Farm by subscribing to the relevant topics from the MOM broker. Hence, a new domain can enter the multi-domain environment managed by *Mayan* with its controller extended with MOM. The controllers communicate with each other to collaborate in a protected and regulated manner, by passing messages, leveraging the public Internet or a dedicated/private direct network link between the domains. The MOM supports inter-domain communication and sharing the health statistics of the nodes to the relevant user. Local network topology data that is relevant to the other controllers, including information on the data tree, event notifications, and RPCs are shared with other controllers, based on their subscriptions. Thus *Mayan* disseminates crucial information on network topology and service health statistics of each domain with other interested/relevant domains, giving control to each domain on what to publish (and to whom, by deploying relevant authentication mechanisms offered by the MOM broker, to limit the audience) and which topics to subscribe.

The Controller Farm coordinates multi-domain networks through its MOM approach, as inter-domain controller hierarchies impose additional challenges concerning privacy and enterprise policies. A fixed or static hierarchy of inter-domain controllers would require a central controller deployment in one or a selected set of domains, thus forcing the controllers of other domains to accept a single controller as the core or primary authority (thus creating a controller hierarchy or levels of control and trust). Opening up controllers for a higher level controller from outside domain could also make the network topologies of each domain or infrastructure provider open for compromises from outside, making the crucial information on network topologies vulnerable to curious entities or onlookers from outside the domain or organization. Thus, a centralized approach would limit the applicability of the solution and is not feasible in a multi-domain edge and inter-cloud environments with multiple third-party data center and service providers. Therefore, *Mayan* proposes and leverages a Controller Farm instead of an inter-domain hierarchy of controllers.

6.2.2 Context-Aware Service Compositions with *Mayan*

Figure 6.3 depicts the deployment architecture of *Mayan*. The $Controller_A$ controls the $Domain_A$, whereas the $Controller_B$ controls the $Domain_B$. The s^1_A and s^1_B service instances satisfy the web service requests to the s^1 in the $Domain_A$ and $Domain_B$, respectively. An entire workflow execution sequence is carried out in a single original composition (typically from the same domain) unless any given threshold (such as the load on any of the service instance) is met. When a threshold is met, an event is triggered, and the controller is notified. If there is no service instance to satisfy the request in the current domain, *Mayan* sends the request for service provisioning to an alternative deployment from another service domain, based on the domain's subscription via its Event Listener. Upon successful acceptance of the service migration request, a service instance from the other domain continues to execute the remaining of service executions.

In addition to the services engines, *Mayan* also consists of a **Web Services Registry** in each domain, to list and describe the service endpoints belonging to the domain. *Mayan* maintains

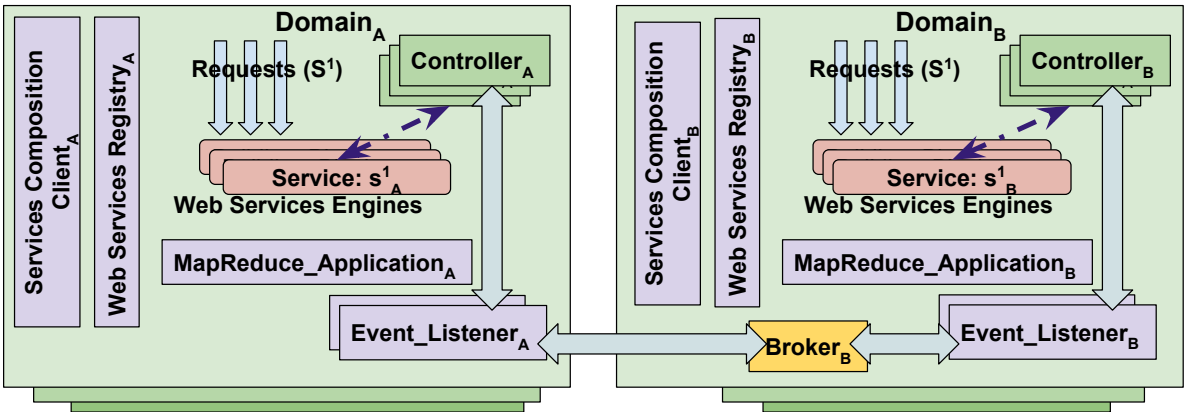


Figure 6.3: Inter-Domain Service Compositions with *Mayan* Controller Farm

constructs and components to ensure that the list of available services is up to date in the web services registry, to minimize inter-domain service compositions and workflow migrations. Once the latter segment of the workflow completes execution in the receiving domain, the result is sent back to the workflow consumer. A MapReduce application monitors and analyses the frequency of inter-domain service invocations and updates the controller accordingly to minimize communication overloads due to frequent data transfer between different domains. The web services registry holds a sorted list of services endpoints and service descriptions, with the input from the controller. Designed as a simple Java program, the *Mayan* registry is generalized to accommodate MapReduce services and the IMDG distributed applications, in addition to the regular web services such as Axis2 or CXF. When the services are developed with MapReduce or other distributed execution frameworks, *Mayan* registry holds the endpoints of the master node that receives the user requests.

The **Service Composition Client** in each domain enables the tenants to consume services from the domain and compose service workflows from the services hosted in the web services engines and the distributed execution frameworks used as alternatives to the traditional web services engines. *Mayan* respects user preferences when choosing the service instances for the service composition. If the user does not indicate a preference for specific service instances, *Mayan* identifies the service instances based on the tenant policies dynamically. First, service implementations are handled at the web services registry, whereas service deployments for the same implementation (i.e., server nodes deploying identical service instances) are handled at SDN level. Time taken to complete an individual service invocation is measured at the web services engine, and the time taken to complete the service composition is measured at the *Mayan* service composition client. The nodes hosting the web service deployments which consume more time to complete the service requests are moved downwards in the routing table to avoid further network flows from being routed to those nodes. Thus, SDSC partially delegates the procedure to find the best-fit service instance to the SDN controller.

For the initial requests to a web service in a service composition workflow, the service composition client retrieves the endpoint URIs (Uniform Resource Identifiers) of the potential service instances, leveraging the controller and the web services registry. The service composition client

thus identifies a service instance for each of the service composing the workflow, and executes the service composition workflow by chaining the responses from the service instances. The respective services return the results to the service composition client. If any service invocation depends on the results of a previous service invocation, the service will block and wait until it receives the results from those invocations. Services that do not have such dependencies execute in parallel. *Mayan* contains stickiness such that once a service instance serves a request, the same instance continues to serve further requests from the tenant workflow by default. Thus, *Mayan* avoids migration of state for a stateful invocation of continuous service executions.

The *Mayan* web services registry can be modified through a configuration file at startup time or can be edited later manually or by *Mayan* dynamically. Preference order of the services is changed based on the load on the web services as observed by the web services engine. Service descriptions in the web services registry define the service names, details on different installations, and multiple deployment endpoints for each of the service instances. The execution order, whether the service execution can be distributed, should the service wait until the previous execution to complete, are a few common properties included for the service composition.

6.2.3 Initializing the *Mayan* Framework

Algorithm 5 presents the initialization procedure of the *Mayan* framework in each domain. First, as shown in line 2, *initController()* initializes the SDN controller with bootstrapping information consisting of the endpoints of the MOM brokers. The MOM brokers can be local brokers managed by the domain of the controller, or remote brokers managed by other domains that the controller has subscribed to. The bootstrapping information enables the controller to communicate with the brokers for the inter-domain communications regarding the wide area network status as well as the migrations including the data storage and service executions. Then the flag *controller.initialized* is set. If the domain manages any broker instances, *broker.initialize()* initializes those instances next if they have not been initialized already by a prior initialization procedure (line 4 - 5). The boolean outcome of the initialize() is set as the value for the flag *broker.initialized*.

Once the controller and the brokers are initialized, *init(server)* (lines 7 - 9) initializes each server (precisely a subset of servers, the servers in a dedicated cluster in the data center, rather than all the servers of the data center, since *Mayan* does not necessarily span the entire data center) and its respective persistent storage (such as SQL and NoSQL databases) in the data center. Then *initImdgCluster()* initializes all the IMDG clusters (lines 10 - 12). Each IMDG cluster spans the entire execution nodes as a virtual in-memory cluster. An instance that initializes a cluster becomes the master instance of the cluster. Since we can have several IMDG clusters for each IMDG (such as Hazelcast [177] and Infinispan [225]), the procedure initializes at least one cluster per IMDG. Once the IMDG clusters are initialized, *initServiceEndpoints()* (line 13) initializes the service endpoints, to receive the client requests for the services and APIs from the clients. Then, *startIMDG()* (line 16) starts the IMDGs in each of the servers. Once an IMDG instance is launched, *joinImdgCluster(id, server)* joins the instance to the respective IMDG cluster (line 17). Finally, *initServices()* (line 19) initializes the service instances in each

Algorithm 5 Initialize the *Mayan* Framework

```
1: procedure InitMayan()
2:     controller ← initController(brokers)          ▷ Initialize the controller with the bootstrap information
3:     controller.initialized ← TRUE
4:     if (∃ broker ∈ controller.brokers; (broker.isLocal() ∧ ¬broker.initialized)) then
                                                      ▷ Presence of uninitialized local brokers
5:         broker.initialized ← broker.initialize()   ▷ initialize() returns TRUE, if successful
6:     end if
7:     for all (server ∈ cluster) do
8:         init(server)
9:     end for
10:    for all (imdg ∈ IMDGs) do
11:        imdgIDs ← initImdgCluster(imdg)
12:    end for
13:    initServiceEndpoints()
14:    for all (server ∈ cluster) do
15:        for all (id ∈ imdgIDs) do
16:            startIMDG(id, server)
17:            joinImdgCluster(id, server)
18:        end for
19:        initServices(server)
20:    end for
21:    while (controller.initialized) do              ▷ Periodic update to the SDSC environment of the domain
22:        if (broker.getEvent(t) ≠ ∅) then           ▷ A non-null event received at current time t
23:            ℰ(t) ← broker.getEvent(t)              ▷ Set the value of ℰ(t) from the event
24:            if (ℰ(t) = SIGINT) then                ▷ Interrupt Signal Received
25:                controller.initialized ← ¬controller.initialized   ▷ Negate the controller.initialized flag
26:            else
27:                update(∇ℰ(t))
                   ▷ An event composed of updates to the controller, as well as the broker and service instances
28:            end if
29:        end if
30:    end while
31: end procedure
```

server and starts receiving the service requests.

Once *Mayan* is initialized for the domain (typically a data center or a cluster), the *Mayan* controller continues to monitor for events from the broker (line 21). If a non-null event is received at time t (line 22), it is stored asynchronously in an event queue of $\mathcal{E}$, with the timestamp t (line 23). If the event is an interrupt signal *SIGINT* (line 24), the flag *isControllerInitialized* is inverted (line 25). Otherwise, the update function is invoked to update the controller, broker, and service instances appropriately via *update(*$\nabla\mathcal{E}(t)$*)* (line 27). Equation 6.8 defines the change propagated through the event $\mathcal{E}(t)$ as $\nabla\mathcal{E}(t)$. $\nabla\mathcal{E}(t)$ is a combination of changes in controller (c), local broker instances ($b \in B_c$) as well as the service instances ($s \in S_c$) managed by the controller c.

$$\nabla\mathcal{E}(t) = \frac{\partial\mathcal{E}(t)}{\partial c} + \frac{\partial\mathcal{E}(t)}{\partial b}|\forall b \in B_c + \frac{\partial\mathcal{E}(t)}{\partial s}|\forall s \in S_c \tag{6.8}$$

Thus, the update procedure keeps the environment updated in each domain until the controller is terminated. As the controller termination itself is defined as an event, the event of a controller (and consequently, the domain controlled by the controller) leaving the network is immediately propagated to the other controllers as a MOM event, before the controller terminates.

6.2.4 Scheduling Service Composition Workflows

Algorithm 6 presents the overall scheduling of a service s that composes a workflow on a cluster (or a data center) following the SDSC approach. First, an empty set named *potentialServers* is created to track the potential servers for the given service scheduling (line 2). Then, while the controller is in the initialized state (as illustrated in the Algorithm 5), the scheduling process continues to wait for the service invocations (line 3). Network congestion or resource scarcity in the current execution node can interrupt the execution, in the form of control flows. As illustrated by Equation 6.8, *update(s)* will trigger an update to the scheduled service executions, if $\frac{\partial \mathcal{E}(t)}{\partial s} \neq 0$. If the set *potentialServers* does not contain the best-fit server (as an ordered set, the first entry indicates the best-fit, and therefore, the algorithm simply confirms that the set is not empty) or if an update is triggered for the given service (line 4), the *potentialServers* set is initialized with the servers that host the relevant service instances in the current domain (line 5).

Algorithm 6 Context-Aware Scheduling of a Web Service

```
1: procedure SCHEDULE(CLUSTER, S)
2:     potentialServers ← ∅
3:     while (controller.initialized) do
4:         if ((potentialServers.getBest() = ∅) ∨ (update(s) ≠ ∅)) then        ▷ ∂E(t)/∂s ≠ 0
5:             potentialServers ← cluster.getServer(s)                        ▷ minimize(A(n, D))
6:             if (potentialServers = ∅) then
7:                 broker.update(s, status)        ▷ Service status as a MOM message to the associated brokers
8:                 serviceEndpoint ← broker.get(s).endpoint
                                                   ▷ Receive potential service instance to migrate the workflow
9:                 s.migrate(serviceEndpoint)
10:                return
11:            end if
12:            s.schedule(potentialServers.getBest()))
                                                   ▷ Choose the first in the ordered set of best servers in the cluster
13:        end if
14:    end while
15: end procedure
```

If the set remains empty after initialization, the algorithm identifies that the servers in the current domain did not meet the resource requirements for the service execution (line 6). Then the controller updates the brokers that it has subscribed to, with this status for the service execution (line 7). Then, the controller receives a potential service endpoint for the service invocation via the broker, from the other domains (line 8). Once an alternative service endpoint is identified, the service invocation is migrated to the newly identified service instance, and the

execution of the service in the current domain terminates (line 9). Finally, the service invocations are scheduled to the chosen server(s) using *s.schedule()* (line 12), until its completion. The chosen service instances continue to receive the client requests from the service composition client for the specific tenant workflow until another interrupt occurs. The invocations to the following services of the workflow are migrated accordingly to the new domain, if the domain also hosts those service instances and if such a migration is favorable for performance.

6.2.5 Layered Architecture of *Mayan*

Figure 6.4 illustrates the layered architecture of *Mayan* with a network view and a service composition view. The network view consists of the hosts deployed on top of the network. Controllers are connected to the switches through OpenFlow protocol. Service composition client receives the user requests and queries for the service deployment. It redirects the service calls to relevant service deployments, after consulting the web services registry. At the network level, the controller has the autonomy to decide one of the available alternative deployments for the same service implementation. The service composition view looks into the same hosts through a higher level of abstraction. OpenDaylight, the default SDN Controller of *Mayan*, enables communication between the two views, as it is known from, and aware of, both views - through its southbound OpenFlow API to network view, and through its northbound user-facing APIs to service composition view. Master and secondary instances aim at providing scalability and fault-tolerance to the service composition client.

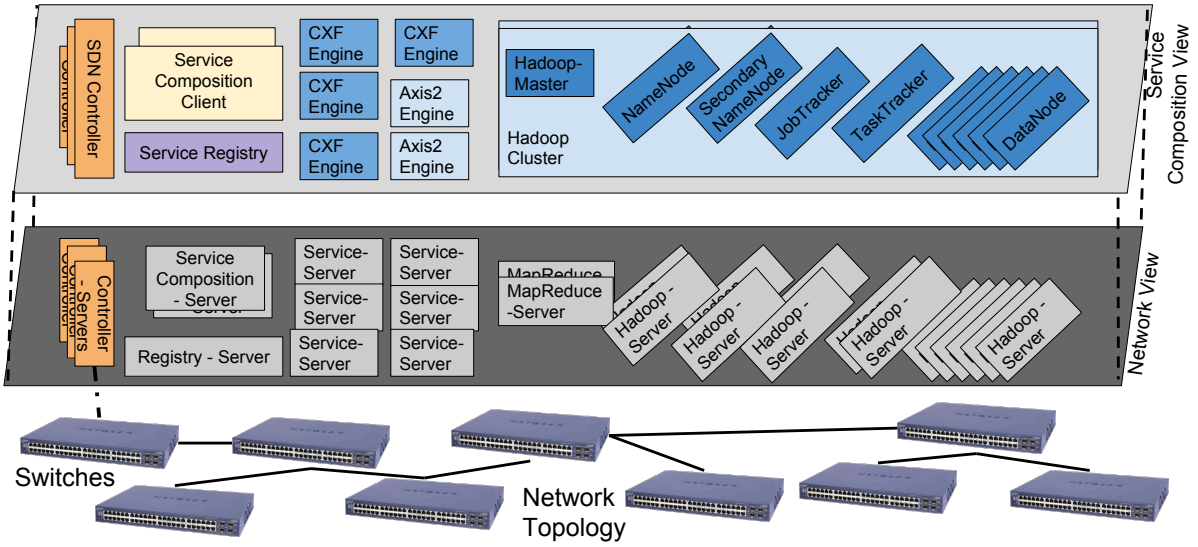


Figure 6.4: Three-Dimensional View of *Mayan*: Hosts, Network Topology, and Services

Service composition client invokes the web services hosted in the relevant web services engines. Moreover, it also connects to the IMDG clusters and Hadoop-Master instances which mimic the web services engines in producing the access point for those distributed execution clusters. The service composition client gets the service health information from the web services engines and Hadoop master instance of each of the Hadoop clusters and updates the SDN

controller on the routing table. It uses the web services registry to get the list of services and service descriptions including the service endpoints which were earlier removed from the routing table by *Mayan* due to their failure to respond or congestion. Thus, the servers hosting those removed services are added back to the flow tables after a specific time of blacklisting the congested or malfunctioning instances. In addition to the Hadoop-Master instance, each Hadoop cluster consists of a NameNode, JobTracker, TaskTracker, Secondary NameNode, and multiple DataNodes. The various DataNodes contribute to the Hadoop Distributed File System (HDFS). Service composition client finally returns the outcome of the web service invocations to the user.

6.3 Implementation

We prototyped *Mayan* to examine the feasibility and performance of an SDSC framework. In addition to Apache Axis2 1.6.3 and Apache CXF 3.1.3 web services engines, we exploited Apache Hadoop 2.7.1 and Hazelcast 3.6 to implement our services. We used OpenDaylight Beryllium as the core SDN controller. We custom developed a services registry to accommodate descriptions of services deployed in the web services engines, as well as MapReduce frameworks such as Hadoop in the service composition workflow to replace a traditional web services engine. *Mayan* depends on the availability of multiple implementations of services offering the same functionality, and numerous deployment instances of same implementations which can be chosen by the extended SDN controller, by partially delegating the service discovery to the network level. The flow tables list the hosts consisting of service deployments that are high on priority, for the service composition workflows. *Mayan* demotes the identified highly congested service deployments in the services list.

Mayan leverages OpenDaylight's data tree as the internal data store of its control plane. *Mayan* inter-domain controllers communicate and collaborate through AMQP messages brokered by ActiveMQ broker, without actually sharing a single global view of the network. The controller persists the messages in a queue for a higher level of parallel messages, to handle the scale efficiently. When the controller is overloaded, or the memory of the server that runs the controller is overutilized due to a higher number of messages, the broker persists the messages to an instance of KahaDB file-based data store in the local filesystem. The stored messages can later be retrieved when necessary. *Mayan* uses a YAML (YAML Ain't Markup Language) [45] style configuration for service descriptions in the registry. Given below is a sample service listing in the registry, with minimal description. This listing refers to data cleaning in an integrated data repository construction workflow.

```
services:
    service: dataconsolidation:
        type: composition
        entry_point: 192.168.0.164
        description: consolidates data from various data sources
        services:
            data_consolidate:
                order: 1
            detect_duplicates:
                order: 2
```

```
            serialized: false
        write:
            order: 3
            serialized: true
service: dupl_instance_count
    impl: axis2
        axa: 192.168.0.104;
        ...
    impl: cxf
        type: jaxws_preliminary_ver
            cxa: 192.168.0.130
        type: jaxws_ver_2
            update: true
            cxb: 192.168.0.131
        type: jaxrs
            cxc: 192.168.0.132
    impl: mapreduce_hadoop
        mra: 192.168.0.133
        ...
    impl: hazelcast
        hza: 192.168.0.158
        ...
```

dataconsolidation is a service composition consisting of simple service implementations, data_consolidate, detect_duplicates, and write. Here, data_consolidate retrieves data from multiple data sources. detect_duplicates finds duplicates out of the pairs of data elements from the invocation of data_consolidate. Finally, write service invocation outputs the data to the relevant integrated data repository. dupl_instance_count is a simple service with various implementations and deployments in Axis2, CXF, Hadoop MapReduce, and Hazelcast. The alternative endpoints represent multiple physical or virtual deployments of the same service code. Moreover, different implementations exist, even using the same services engine as depicted for CXF, where three implementations exist - one is a REST/JAX-RS based implementation whereas the other two are SOA/JAX-WS based implementations. We can add further extended service-specific parameters, as shown by the property "update" for CXF implementation of jaxws_preliminary_ver.

6.4 Evaluation

We evaluated *Mayan* for its controller performance and scalability of the execution workflows in multi-domain networks. We used a computer cluster with Intel® Core™ i7-4700MQ CPU @ 2.40GHz × 8 processor, 8 GB memory, and Ubuntu 14.04 LTS 64 bit operating system for the evaluation. Due to the limited accessibility to servers in wide area networks, we emulated SDN networks with Mininet and OpenDaylight extended with *Mayan* components. We engaged a varying number of servers in a cluster to benchmark the performance and speedup of the workflow execution with *Mayan* against the conventional network-agnostic service compositions.

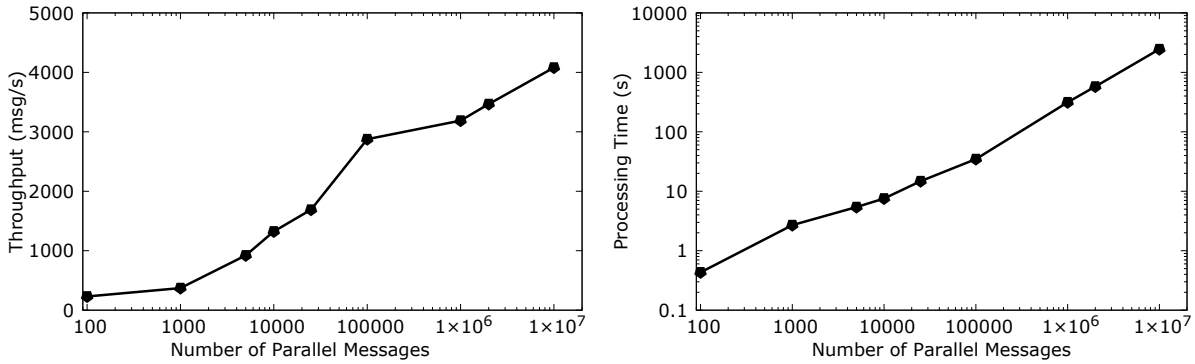
6.4.1 *Mayan* Controller Performance

We evaluated the controller efficiency in composing the service workflows with inter-domain messages. To quantitatively assess the controller performance, we limited our evaluation to a

stand-alone controller deployment that receives messages from the controllers of other domains through a MOM architecture and returns service migration notifications.

Prototype deployment: We emulated a simple CPS workflow with several sensors, actuators, and processing services connected to the *Mayan* controller deployment. We modeled the sensors as services that produce data and the actuators as those receive the input from the sensors and compute and perform actions based on the other service processing inputs. The workflow is executed in a single domain, until the services are overloaded with the requests, thus leading to a notification to the controller. Each message received by the controller triggers an execution migration between the service instances across the domains managed by multiple controllers. We thus built our workflow as a service composition with several service executions.

Controller Throughput: We evaluated the *Mayan* controller performance with a varying number of concurrent and parallel messages, concerning throughput, number of messages processed concurrently, and success rate. We measured the end-to-end throughput of the controller as the number of messages entirely transformed by the controller, arriving from the publisher domain and forwarded towards a relevant receiver domain. Figure 6.5 shows the throughput and the total processing time of a stand-alone controller deployment.



(a) Throughput of the controller

(b) Total processing time

Figure 6.5: *Mayan* stand-alone controller performance in processing messages in parallel

We observed the *Mayan* controller performance in handling messages to ensure that *Mayan* can manage multi-domain workflow traffic with minimal overhead. Figure 6.5a illustrates the throughput or the typical message processing rate of a single *Mayan* controller against that of the number of parallel messages that the controller processes. By effectively handling the parallel messages, the *Mayan* controller processes 5000 messages/s in a concurrency of 10 million messages. Figure 6.5b depicts the total time taken to process the complete set of messages at a single instance of the *Mayan* controller, against a varying number of messages. During a timeframe of 40 minutes, it processed a sum of 10 million messages. The controller linearly scaled concerning processing time with the number of parallel messages.

Success Rate: We observed how many messages can be processed and delivered per second with a high success ratio with a single controller deployment of *Mayan* in each domain. Figure 6.6 illustrates the success rate of message processing with an increasing number of parallel messages. The success rate remained high around 100%, always above 99.5% regardless of the number of

parallel messages the controller handles. For up to 10,000 parallel messages, the success rate remained at 100%, and only slightly reduced down to 99.5% for a more substantial number of messages: up to 10 million parallel messages. Even the reported 0.5% of failures indicates just the failed first attempt; *Mayan* resends the failed message, based on the configurations (whether to retry or ignore in case of a message processing or delivery failure). The evaluations highlight the resilience of *Mayan* regardless of the increased number of messages, except for a massive amount of messages such as 100 million parallel messages, which makes the server that hosts the controller run out of memory, due to a large number of messages in the in-memory queue.

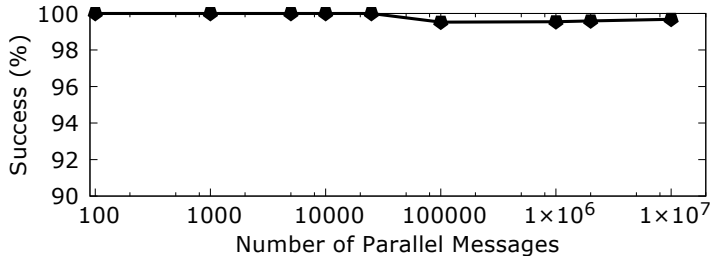


Figure 6.6: Success rate of the controller vs. number of messages processed in parallel

Mayan scales horizontally to cover more domains with a federated controller deployment. We observed the *Mayan* throughput in a clustered controller deployment in an AWS cloud deployment in a single placement group. We note that the throughput or the number of messages processed in a given time indeed increased in such a clustered deployment in each domain. Therefore, we can improve the throughput by deploying *Mayan* in a server with more resources, or through a distributed and clustered deployment in each service domain.

6.4.2 Speedup of Service Compositions with *Mayan*

To benchmark the speedup of the *Mayan* SDSC approach, we modeled a workflow performing distributed data cleaning and consolidation as i) a base distributed web service composition as well as ii) a network-aware SDSC execution. We used around 100 datasets of sizes between 1 - 3 GB from the Cancer Imaging Archive (TCIA) [77], a medical image repository. We further synthesized more data with near-duplicates from the original datasets. Figure 6.7 depicts the speedup of the distributed execution with a various number of distributed instances.

Both the base distributed execution as well as the *Mayan* execution scaled near-linearly for up to 100 geo-distributed execution nodes for the service workflow scheduling. We observed that initially, both the base and the SDSC approaches showed speedup with reduced time for workflow execution and migrations with an increasing number of service instances. We noted that the speedup with the base execution reached the global maximum much faster than the *Mayan* execution. The base execution started to poorly perform when we distributed the services to several instances beyond what is essential for the fastest execution time. *Mayan* minimized the cross-domain workflows while efficiently exploiting the services deployed close to each other. By leveraging the global knowledge of the network and data distribution readily available to

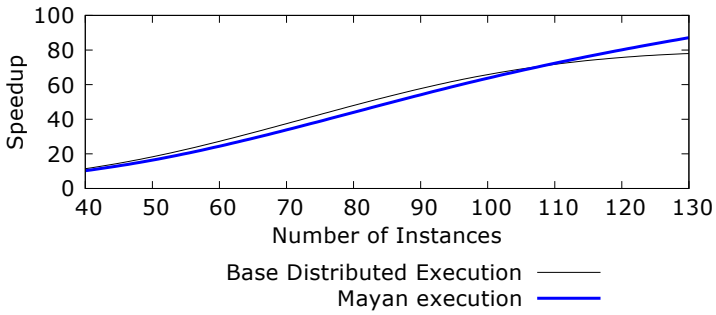


Figure 6.7: Speedup of distributed data cleaning and consolidation workflow

the controller, *Mayan* devised a context-aware distributed data cleaning workflow, with related objects, stored closer, increasing the locality.

The difference in scalability and speedup between the base execution and the *Mayan* execution indeed depends on the nature of the service composition, workload, and the considered multi-domain network environment. Furthermore, with even more nodes introduced to the execution environment, *Mayan* too ceases to provide speed up with additional nodes. However, we note that with its network-aware scaling and service workflow scheduling, *Mayan* offers better scalability, compared to the base network-agnostic distributed execution. Our observations highlight the performance benefits of the global network awareness in the SDSC approach achieved through the federated controller deployment with SDN and MOM. Furthermore, our evaluations illustrate the efficiency of the *Mayan* controller in offering the throughput and success rate necessary for the inter-domain control.

6.5 Conclusion

Complex eScience workflows aggregate and chain several geo-distributed web services and microservices for their execution. Large-scale workflow executions are often intensive in computation or communication, where delays and failures are not tolerated. Various alternative implementations and deployments exist for these services and applications, attempting to increase their fault-tolerance. However, seamlessly deploying and migrating service executions across these alternative instances is a challenging task due to the lack of coordination among the service providers of several domains.

SDSC is an integration of SDN into service composition to facilitate context-aware execution of service workflows. It minimizes latency and communication overhead, by enabling coordination across service providers of multiple domains. We built *Mayan* as an SDSC framework for an adaptive execution of scientific workflows through a federated deployment of SDN controller clusters in a wide area network. *Mayan* thus enables communication across inter-domain controller clusters, without sharing a global network view with any single entity.

Mayan finds the best-fit among the alternatives of available service execution options, considering various constraints of network and service level resource availability and requirements,

while respecting the locality of the service requests. *Mayan* utilizes the local service load information available for the web services engine. It achieves network level heuristics beyond data center scale, using protocols such as MOM in conjunction with SDN. Preliminary evaluations on the *Mayan* prototype showed how an SDSC approach could be leveraged as a reusable, scalable, and resilient distributed execution framework for the scientific workflows on a global scale.

7 Network Service Chain Orchestration at the Edge

As network services are often latency-sensitive, NSC should be carried out maintaining proximity among the services that compose the NSC as well as the user that consumes the NSC. Due to the inherent bandwidth cost associated with the distance between the network services and the end user, more and more third-party VNF providers choose to deploy their services at the edge, rather than a cloud. Finding the best-fit VNFs that offer a high QoE while abiding by the user policies remains a significant challenge in the multi-tenant edge environments. Mobile environments face further hurdles due to their resource-constrained nature and the requirement for live migration of data and execution. The need for locality and an optimal match between the user policies and the VNF offerings has created several research challenges in the context of service placement at the edge. A typical NSC consumes various network services, where one's output becomes the input of another. Despite their increasing number, current edge providers give limited control to the users to compose their NSCs by seamlessly choosing the VNF instances based on the user policies. By increasing the number of deployment and execution alternatives, the rise of the distributed and decentralized edge nodes has increased the complexity of finding the optimal VNF instances at the edge for an NSC.

Motivation: Given the above premises, we aim at addressing the following research questions in this chapter:

(RQ_1) Can we simplify and generalize the VNF allocation for seamless execution of the user NSCs at the edge, in the presence of several third-party edge VNF providers and numerous users?

(RQ_2) Can we bring the control of the NSC back to the user from the service providers, despite using third-party edge VNFs?

(RQ_3) Can we optimally provision the user NSCs across multi-provider edge VNFs, adhering to the user-defined policies?

(RQ_4) Can we scale SDN to compose NSCs at the edge in a resilient and agile manner?

Contributions: The goal of this chapter is to answer the identified research questions. The main contributions of this chapter are:

1. A scalable and optimal framework for NSC placements at the edge, consisting of an SDN architecture extended with MOM [87] for global awareness of the VNF nodes. (RQ_3 and RQ_4)

2. Generalizing the challenge of resource allocation, service discovery, and workflow migration on the edge platforms for an adaptive execution of NSC as MILP problems. (RQ_1)

3. A graph-based approach to solving the MILP models from the perspective of a remote user of third-party edge VNF providers. (RQ_1 and RQ_3)

4. Adaptive algorithms, executing decentralized at the devices of the VNF users, independent from the executions of the other devices, to optimally provision the user NSCs across the edge VNFs for a locality-aware and policy-aware execution. (RQ_2 and RQ_3)

We implemented *Évora*[1] (**E**dge **V**NF **O**rchestration with **R**esilience and **A**gility), to construct and deploy NSCs at the edge. *Évora* considers various system constraints such as bandwidth efficiency and economic aspects, as well as user-defined policies such as SLOs of required throughput and uptime. *Évora* exploits its SDN architecture extended with MOM to discover the global VNFs in a wide area network. It chooses the edge VNFs locally for the execution of the NSC requests of the users, by wholly and exclusively searching the nodes identified via the MOM messages. Thus, *Évora* follows a greedy approach of "think globally, act locally", and solves the NSC allocation efficiently at each edge device with dynamic programming.

Évora ensures optimality in the NSC placement at the edge from the user perspective, without sacrificing the performance of the user NSC and the VNF providers. *Évora* brings control of the NSC back to the user, from the cloud and service providers, as it used to be in the days before the invent of multi-tenant cloud platforms. Thus, *Évora* transforms the user into the central entity of the NSC ecosystem, from being a passive participant. Furthermore, it supports the seamless inclusion of more network domains by adding their respective controller to the *Évora* ecosystem by subscribing through the MOM messages.

We deployed and performed an experimental evaluation of *Évora* for NSC execution at the edge. In particular: (i) we evaluated the correctness and optimality of *Évora* in its NSC allocations adhering to the user-defined policies, and (ii) we assessed the complexity and scalability of the *Évora* graph-based approach in solving the optimal VNF allocation, posed as the MILP problems. The results obtained indicate that *Évora* ensures that the user-defined policies are met in allocating the user NSCs across the edge VNFs. Furthermore, we observed that the *Évora* extended SDN architecture enables scaling out the edge environment.

This chapter is composed of the contents of the publication: [J1].

7.1 Edge VNF Orchestration with Resilience and Agility

The *Évora* ecosystem consists of an **Event Manager** and an **Orchestrator**, two lightweight modules, executing independently in each user device. These modules let the users manage their NSCs themselves, albeit using third-party edge VNFs. The Event Manager finds

[1] Évora is a Portuguese city with a rich history of five millennia, located at a junction of several important routes of the Roman era.

Table 7.1: Notation of the *Évora* Representation

G	The acyclic multigraph of edge nodes
H	The hypergraph of edge VNFs
$\mathcal{V}$	The set of compute nodes hosting the VNFs
$\mathcal{L}$	The set of links that connect the compute nodes
$\mathcal{N}$	The set of compute nodes considered by an *Évora* deployment
S	The set of VNFs considered by an *Évora* deployment
S_i	The set of VNFs in a node n_i
S_j	The set of VNFs composing an NSC ψ_j
s_*	A VNF instance; $s_* \in S$
X	The set of VNFs, annotated by the node deployment
I	The set of intra-node links between the VNFs of each node
E	The set of links that connect the edge VNFs
R	The set of resources provided by the edge VNF provider

VNFs from the edge nodes, based on event notifications. The orchestrator optimally allocates the NSCs among the VNFs identified by the Event Manager, by formulating MILP models. Section 7.1.1 describes how the orchestrator internally represents the edge nodes and VNFs as graphs in the user devices. Section 7.1.2 formulates the MILP models to provision the user NSCs across the VNFs optimally. Section 7.2 elaborates the orchestrator algorithms to solve the MILP problems.

7.1.1 NSC at the Edge: Graph Representation

Évora models the MEC environment consisting of various interconnected computing nodes[2] as an undirected acyclic multigraph $G = (\mathcal{V}, \mathcal{L})$. $\mathcal{V}$ represents the set of nodes hosting the VNFs, and $\mathcal{L}$ represents the links that connect them. $\forall$i,j,k$\in \mathbb{Z}^+$, each node $n_i \in \mathcal{N} \subset \mathcal{V}$ consists of a set of services $S_i \subseteq S$. A user u_j can compose her NSC ψ_j as a composite function from a set of services, as shown by Equation 7.1. Complete details on the notations are listed in Table 7.1.

$$\forall s_* \in S_j \subseteq S : \psi_j = s_a \circ s_b \circ \ldots \circ s_x \tag{7.1}$$

Since each node offers one or more VNFs, *Évora* denotes the service deployments in a hypergraph $H = (X, E)$ as an overlay on the node graph $G = (\mathcal{V}, \mathcal{L})$. *Évora* minimizes the NSC resource allocation problem into how to place ψ_j in the hypergraph H optimally. Each vertex of H can be either i) a single vertex (interchangeable with a vertex of G) or ii) a complete graph of multiple service deployments within a vertex of G. The former refers to a node that offers only a single network service, while the latter refers to one that offers multiple network services. $E = \mathcal{L} \cup I$, where I represents the set of intra-node links between the services in each node.

[2]To avoid overloading the words 'node' and 'edge', we refer to the graph properties with 'vertex' and 'link'. In our terminology, 'edge' is strictly restricted to the edge environment, and 'node' is strictly restricted to the compute nodes (i.e., servers and smart devices) at the edge.

Equation 7.2 represents X as services annotated with their node deployments. As intra-node service composition is more bandwidth-efficient than inter-node executions, the edges of the hypergraph are differentiated in weight accordingly to represent the minimal latency between the E hosted in a given $\mathcal{V}$.

$$\forall x_i \in X : x_i = n_j.s_k$$
$$\text{where } s_k \implies \text{Any of the services deployed in the node } n_j \in \mathcal{V}. \tag{7.2}$$

Edge data centers often have (either logically or physically) separate links for control flows instead of sharing the same links with the data flows, as data flows are heavy while control flows are light-weight and more critical. The control flows of OpenFlow and SD-WAN span across the edge nodes as well as with the mobile/edge user device. As an illustrative example, Figure 7.1 demonstrates the multigraph G of an edge environment with a user, representing data flows with dark solid lines and control flows with dotted lines. *Évora* interprets the edge environment of VNF deployments as a hypergraph with the service instances inside a node as vertices, and nodes themselves as supernodes, where a single hyperedge connects all the services inside a data center. A hypergraph representation of this environment will lead to several intra-node links, $I = \{n_1.\{s_1, s_4\}\}, n_3.\{s_1, s_3\}, n_4.\{s_2, s_4\}, n_6.\{s_2, s_3, s_4\}, n_7.\{s_1, s_3\}, n_{12}.\{s_3, s_4\}, n_{16}.\{s_3, s_4\}\}$. The sample NSC is defined as $s_5 \circ s_4 \circ s_3$.

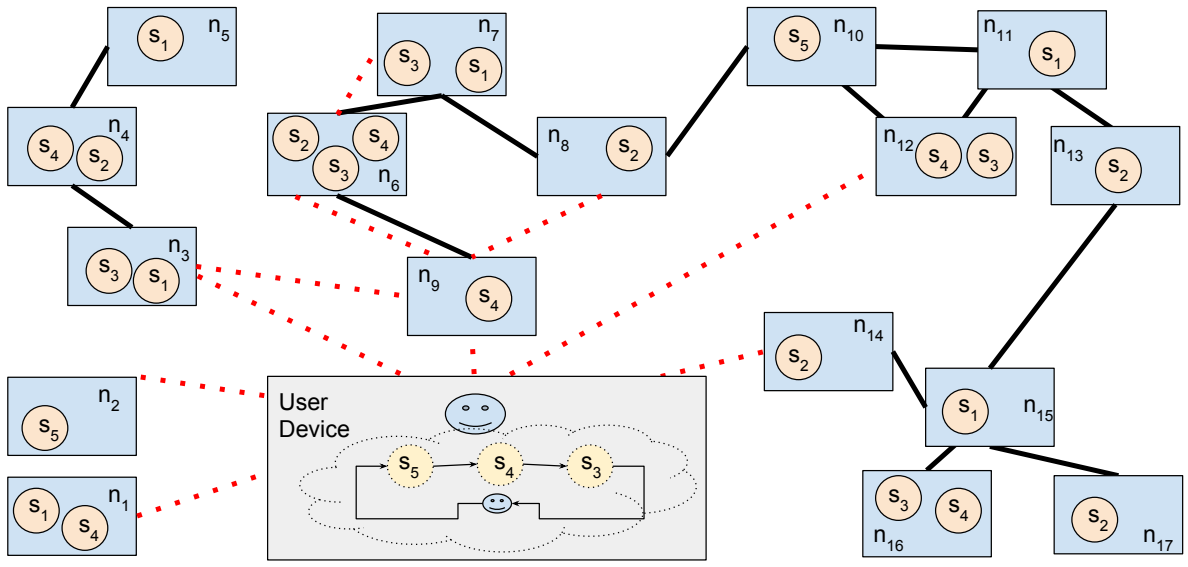


Figure 7.1: A User-Defined NSC Among the Edge Nodes

The orchestrator instance in the user device identifies the nodes required for the execution of each service in the NSC that the user has composed. The user device connects to the nearest nodes, through either a dedicated connection or the public Internet. The orchestrator considers an undirected acyclic graph with only the data links that are responsible for the VNF traffic in its service allocation, eliminating the control links from the model. The user defines her NSC

and seeks the best-fit for the execution based on her NSC policies. To minimize latency in an oversimplified approach based on the geographical distance between the edge nodes, the first and last VNFs need to be proximate to the user, whereas the services in the NSC themselves need to be close to each other. In the ideal case, all the services will be close to each other and the user. The proximity aims to minimize the latency caused by too many hops and the geographical distance between the user and services.

7.1.2 NSC at the Edge: MILP Models

Edge VNF User Perspective: The *Évora* orchestrator in the user device chooses the VNFs among the available alternatives accordingly, to satisfy the user's policies for the NSC execution. The user policies may consist of several parameters, such as throughput, monthly cost, and latency. For example, the user needs to maximize the throughput (T) of her NSCs while minimizing the total monthly cost (C) and latency (L). We define MILP models from the individual user perspective to support the user policies in a scalable manner. The algorithms executing from each VNF user device compute and solve these models for the user's own NSCs.

Equation 7.3 defines the latency L_s as the contribution to the overall latency of the NSC from a service s composing the NSC. τ_s represents latency of the service s. ϱ_s represents the latency between the ingress port and connecting link from the previous service, or the time taken for a unit data to arrive at the current service s from s_{-1} ($s_{-1}, s \in \psi$). Thus, $\tau_s + \varrho_s$ represents the time interval between the departure of a unit data from the VNF s_{-1} and the time VNF s outputs the data through its egress port. t_0^s and t_f^s represent the start and end time of the execution of any VNF s composing the NSC ψ.

$$L_s = \tau_s + \varrho_s = t_f^s - t_f^{s_{-1}} \tag{7.3}$$

Since the VNF provider can charge a dynamic unit price for the VNF offerings based on the current demand and resource availability, we define the cost incurred by a VNF offering by a time integral of the service unit cost over the VNF execution period. c_s denotes the unit cost to acquire a VNF s. Equation 7.4 describes the execution cost of s as part of a user NSC.

$$C_s = \int_{t_0^s}^{t_f^s} c_s dt \tag{7.4}$$

The user should aim at reducing her total operational cost of all the services in the NSC, rather than performing a local minimization for each of her VNFs. As a global minimization problem, we find and store the effective amount composed of C_s, L_s, and T_s of each VNF of NSC as C, L, and T for each NSC, as depicted by Equation 7.5. Here, the latency and cost of the NSC are a sum of the individual values of them for each of the VNF in the NSC. The VNF that has the least of the throughput among those composing the NSC limits the overall NSC throughput.

$$C = \sum_{s \in \psi} C_s; L = \sum_{s \in \psi} L_s; T = \min_{s \in \psi} T_s \tag{7.5}$$

In practice, a VNF provider does not change the unit cost of a service within a brief time. Therefore, for a reasonably short NSC execution time, we can assume the unit cost of a VNF to be constant throughout. Thus, we can simplify the total cost calculation as illustrated by Equation 7.6.

$$\frac{dc_s}{dt} = 0 \implies C_s = (t_f^s - t_0^s) \times c_s \tag{7.6}$$

Maximum total volume of data passed through a VNF can be expanded as *duration* × *throughput*, $(t_f^s - t_0^s) \times T_s$. Finally, Equation 7.7 models the edge NSC allocation as a MILP problem.

$$\underset{\alpha,\beta,\gamma \in \mathbb{N}}{\text{minimize}} (\alpha C + \beta L + \gamma T^{-1}) \tag{7.7}$$

The policy gives weight to the parameters, relative to each other as in a ratio. It can also include further constraints and SLOs such as "$T \geq 50$" to indicate the user requirement to have a minimum end-to-end throughput of 50 Mbps for the NSC. T^{-1}, the inverse of throughput, denotes how much time it takes to process a unit data (measured in Mb) end-to-end through the NSC. For example, a service we hosted on Amazon EC2 offered 50 Mbps or 0.02 s/Mb. α, β, and γ are integers defined by the user as the policy for the NSC in the form of "$C\ \alpha; L\ \beta; T\ \gamma$". The penalty value $F = \alpha C + \beta L + \gamma T^{-1}$ can be generalized to include more configuration options as in Equation 7.8. The minimization problem can also be extended to include additional properties (such as uptime, QoS guarantees, and carbon efficiency) and constraints.

$$\underset{\alpha,\beta \in \mathbb{N}, \gamma \in \mathbb{Z}, a,b \in \mathbb{R}_{>0}, \gamma \times c \in \mathbb{R}_{\leq 0}}{\text{minimize}} (\alpha C^a + \beta L^b + \gamma T^c) \tag{7.8}$$

Edge VNF Provider Perspective: The VNF provider needs to be economically sustainable. Therefore, she should maximize the number of VNF deployments η_s for each VNF $s \in S_i$, to increase the income, while confirming that the VNFs do not overutilize the total available resources $r \in R$, at any given time t. Equation 7.9 expresses these constraints while maximizing the overall platform profit over time. $\upsilon_{s,r}$ refers to the consumption of resource r by a single VNF deployment of s. Δc_s refers to the profit of each VNF, as a difference between the income c_s and expense estimated for the particular VNF deployment ι_s.

$$\begin{aligned} \Delta c_s = c_s - \iota_s; \underset{\eta_s \in \mathbb{N}, \epsilon \in \mathbb{R}_{>0}}{\text{maximize}} \left(\int_{t > \epsilon} (\eta_s \times \Delta c_s) dt \right). \\ \textbf{s.t.} \forall r \in R, r \geq \sum_{s \in S_i} \eta_s(t) \times \upsilon_{s,r}(t). \end{aligned} \tag{7.9}$$

A service provider should spawn a VNF instance in a node only when it is profitable for her (i.e., more income from the end users compared to the cost of running an additional VNF and consequently, more nodes). The provider may even terminate an existing instance without affecting the active NSC workflows to ensure that the overall VNF execution is profitable. Thus, when the VNF provider incurs an economic loss, the change in the number of any VNF instance will be a non-positive value as Equation 7.10 illustrates.

$$\forall s \in S_i : \Delta c_s(t) \leq 0 \implies \frac{d\eta_s}{dt} \leq 0 \tag{7.10}$$

Although we formulate MILP models for both the user and the edge VNF providers, we limit our focus to the edge user perspective of NSCs and solve the MILP optimization problem illustrated by the Equation 7.8. Since *Évora* executes from the user perspective, it considers each edge node as an indivisible atomic unit. Nevertheless, an edge data center often consists of several servers with their own constraints. Managing the VNFs and allocating resources for VNFs inside each edge node (regardless of its nature or scale) is analogous to NSCs in data centers, a well-studied subject [36].

7.2 *Évora* Algorithms

The *Évora* orchestrator executes algorithms that efficiently resolve the NP-hard problem of hypergraph matching [267] for the NSC allocation using dynamic programming. It initializes the global variables that represent the node and service graphs. For each user NSC, the orchestrator identifies the multiple execution paths and chooses the best fit for the current NSC based on its policy (which of the parameters to be minimized or maximized and their respective thresholds). It thus solves the MILP models efficiently for each user, for each of their NSCs along with their policies. The execution is performed only once per a user NSC defined with a given policy, unless there is a change in the user demand or the status of the node or the VNF instance. The chosen execution path is stored for subsequent executions of the NSC with the same policies. Thus *Évora* incurs no overhead in finding the optimal VNFs in subsequent accesses.

7.2.1 *Évora* Global Environment

Algorithm 7 defines the process of orchestrating the user environment by initializing and maintaining the environment consisting of graphs and other global variables. The *orchestrateEnvironment* procedure initializes the global variables (defined in lines 2 - 5). The events received as notifications from the edge nodes, based on the event subscriptions of the user, trigger updates to these variables.

The *orchestrateEnvironment* procedure takes as input a map of user-defined NSCs and their respective policies including SLOs (such as latency, throughput, and loss rate) and other objectives such as minimal cost. *initOrchestrator()* (invoked in line 8) initializes the *Évora* orchestrator that manages the VNF requests to chain them as NSC. *initEventManager()* (invoked

Algorithm 7 Orchestrating the Environment

```
1: global variables
2:     edgeNodesMap                                  ▷ <property of edge node, relevant value>
3:     servicesMultiMap                              ▷ <node, deployed services>
4:     matchingSubgraphsMultiMap                     ▷ <nsc, matching paths>
5:     nscMap                                        ▷ <<nsc, policy>, executionPath>
6: end global variables
7: procedure ORCHESTRATEENVIRONMENT(initNscMap<nsc, policy>)
                                   ▷ The policy includes the user policies for the MILP models
8:     initOrchestrator()
9:     initEventManager()
10:    for all (broker ∈ brokers) do
11:        subscribe(broker)
12:        initNodes ← getEvent(broker, INITIALIZE_EVENT)
13:        for all (node ∈ initNodes) do
14:            edgeNodesMap ← edgeNodesMap ∪ {(node.getKey(), node.getValue().properties),
                       (node.getKey()+ 'NEXT', node.getValue().links)}
                           ▷ Adding the node properties and linking the next nodes to the current node.
15:            servicesMultiMap ← servicesMultiMap ∪ {(node.getKey(), node.getValue().services)}
16:        end for
17:    end for
18:    nscMap ← < initNscMap, ∅ >
19:    repeat
20:        if (eventReceived) then
21:            updateUserEnvironment(event)
22:        end if
23:    until (aborted)
24:    edgeNodesMap ← ∅
25:    servicesMultiMap ← ∅
26:    nscMap ← ∅
27:    matchingSubgraphsMultiMap ← ∅
28: end procedure
```

in line 9) initializes the Event Manager. The Event Manager subscribes to the notifications as defined by the orchestrator, by providing the endpoint of the brokers co-located with a controller in edge nodes (line 11).

Upon initialization, the broker retrieves the details of edge nodes from the broker queue with the topic *INITIALIZE_EVENT* (line 12). *INITIALIZE_EVENT* is a system-defined global static constant, used as the topic for the data necessary to bootstrap the system in the broker. For each of the node received from the broker queue, its properties are added to the *edgeNodesMap* with the key of the node identifier. Furthermore, the *next nodes* are added to the *edgeNodesMap* with the key {*DATACENTER_ID*}_+*'NEXT'*, to represent the node graph (line 14), by denoting the properties of the links. The services of each node are added to the *servicesMultiMap* to represent the services hypergraph (line 15). Each service can be directly retrieved from the multimap by providing the node. The *initNscMap* is added as the keys of the nscMap global variable, as a <*nsc, policy*> tuple, with the value of each entry set to ∅, an empty or

null object (line 18).

The Event Manager listens to the relevant notifications published by the edge nodes. An event will be triggered for the availability of new nodes, unavailability or status update of an existing node, availability of new services in a current node, and unavailability or status update of an existing service in a node. The event is triggered either by the action of the edge nodes or by the user herself such as moving her location from where she executes the NSC from or updating her NSC definitions or their policies. When an event is received from one of the brokers in the edge nodes, the user environment and the global variables (*edgeNodesMap, servicesMultiMap, matchingSubgraphsMultiMap, and nscMap)* are appropriately updated by invoking the *updateUserEnvironment* procedure (lines 20 - 22). This update ensures that the NSC executions are dynamically updated to reflect the current status of the edge nodes. When the orchestrator is terminated, the global variables are reset to ∅ (lines 24 - 27). As the *orchestrateEnvironment* procedure manages the user environment, the user has the updated information on the available service deployments at the edge.

7.2.2 NSC Execution Paths at the Edge

Algorithm 8 is an implementation of a graph matching problem, optimally designed for adaptive NSC allocation at the edge. We built the algorithm based on our observation that NSCs have fewer vertices (typically, up to 10 VNFs), whereas the edge nodes are much larger in numbers (several thousand in MEC environments). This algorithm confirms the interconnection between the service deployments for a complete NSC execution, rather than merely confirming the existence of a VNF in a node from *servicesMultiMap.* The algorithm depicts the process of finding the potential execution paths for an NSC, which can be later used to determine the exact NSC execution flow for any given policy. It takes *nsc* defined by the user programmatically or through a configuration file, as well as a pointer to a graph traversal function **iTraverse()** as the input. Since the nearest nodes of the hypergraph represent the services in the same node, the algorithm by default chooses breadth-first traversal as the implementation of iTraverse(). Any user-defined traversal such as random walk can replace the traversal implementation. With the choice of the traversal algorithm, the constructed execution paths are already sorted to fit the search criteria.

partialNSCs is a list of arrays (line 2) that traces the potential NSC execution paths in the node graph, as the algorithm constructs them into partial chains of the NSC. Each array in the list consists of maximum entries equal to the number of services in the NSC in the form of a tuple <*service, currentNode*>. These arrays grow with the number of service deployments found from the edge nodes, as a matching subset of the complete NSC. Similarly, the *partialNSCs* list itself grows as more potential execution alternatives are constructed. Thus, it provides chains of pointers to the matching node for each of the services in the NSC. The multiple arrays in the *partialNSCs* list indicate the possible multiple execution paths, stored against their service. The algorithm initializes *partialNSCs* list with ∅ (line 5). Then it repeatedly traverses the graph until it completes visiting each of the nodes. iTraverse() traverses the *edgeNodesMap*, one node at a time, returning the current node at each move (line 7). Then the algorithm finds whether

Algorithm 8 Finding NSCs at the Edge

```
1: local variables
2:     partialNSCs                                          ▷ List{< currentNode, Service > []}
3: end local variables
4: procedure INITNSC(nsc, iTraverse())
5:     partialNSCs ← ∅                                      ▷ Initialize the list with an empty set, ∅
6:     repeat
7:         currentNode ← iTraverse(edgeNodesMap)            ▷ Traverse the map, one node at a time
8:         for all ((service ∈ currentNode) ∧ (service ∈ nsc)) do
                                                            ▷ Services composing nsc found in the currentNode
9:             for all ( (pNSC ∈ partialNSCs) ∧ (service ∉ pNSC) ) do
                                                            ▷ Service found not to be in partialNSC elements
10:                if (( (edgeNodesMap.get(currentNode.getKey()+NEXT)) ∧ pNSC) ≠ ∅ ) then
                                                            ▷ A link exists between an element in pNSC and currentNode
11:                    pNSC ← pNSC ∪ {(service, currentNode)}
12:                end if
13:            end for
14:            partialNSCs ← partialNSCs ∪ {newPartialNSC(service, currentNode)}
15:            for all (pc ∈ partialNSCs) do
16:                if (pc.length = nsc.length) then
                                                            ▷ Array at full capacity. pc has been constructed as the NSC
17:                    pc.persistPenaltyValues(pc)          ▷ Add metadata to pc object
18:                    matchingSubgraphsMultiMap ← matchingSubgraphsMultiMap ∪ {(nsc, pc)}
19:                    partialNSCs ← partialNSCs \ {pc}
20:                end if
21:            end for
22:        end for
23:    until (allNodesTraversed)
24: end procedure
```

the current node has one or more services that are part of the NSC (line 8), as it can readily retrieve the services available in each node from the *servicesMultiMap*.

For each array representing the constructed partial NSC *(pNSC)* in the *partialNSCs* list, the algorithm checks whether the service is not already present (line 9). Then it confirms that there are overlaps between a *pNSC* and the 'NEXT' pointers of the current node stored in the *edgeNodesMap* to denote the links (line 10). An overlap indicates the existence of one or more links, showing the possibility to add the current node into the NSC. Thus, the algorithm puts the <*service, currentNode*> tuple into the *pNSC* array to add the next entry in the chain (line 11). For each node with a service matching one from NSC, a new object is created with the tuple <*service, currentNode*> as the first VNF, and added to the *partialNSCs* list (line 14). This approach gives space to construct new potential NSCs with the currently traversed node as the first entry. Equation 7.11 depicts the construction of *partialNSCs* list in the form of NSCs. The subsets of the longest partial NSC are also stored in the list, to identify and store all the alternative execution paths effectively.

$$\forall y, z \in \mathbb{N}, \forall s_y \in S_j, \forall n_z \in V, \{l, m, o, \ldots, t, p, q, r, \ldots, k\} \subset \mathbb{N}, serviceIDs = \{p, q, r, \ldots, k\} :$$
$$partialNSCs = \{\ [\varnothing], [< s_p, n_l >], [< s_p, n_l >, < s_q, n_m >], [< s_p, n_l >, < s_q, n_m >, < s_r, n_o >],$$
$$\ldots, [< s_p, n_l >, < s_q, n_m >, < s_r, n_o >, \ldots, < s_k, n_t >]\ \} \tag{7.11}$$

Performance Optimizations: *Évora* stores the intermediate stages of the matches together with their computed penalty values, to speed up the performance of finding the complete NSCs. The algorithm checks the *partialNSCs* list during each traversal for complete NSCs (line 15). Each element in the *partialNSCs* list is an array towards building the NSC. Therefore, when any of the arrays consists of services equal to the number of services in the NSC, the algorithm considers the array to be complete (line 16). We expect a better performance by comparing the array lengths with the number of services composing NSC, instead of comparing the services themselves. *Évora* computes the penalty values consisting of parameters such as T^{-1} and C for each chosen NSC, by retrieving the unit values (such as c_s and τ_s) from the edge VNF provider APIs for each of the VNFs. Since its goal is to satisfy the optimization of the MILP models, it stores the computed penalty values consisting of parameters as metadata for the NSC, using *persistPenaltyValues()* (line 17). Storing of penalty values as metadata against each NSC increases the performance of finding the best-fit NSCs for different user-defined penalty ratios (such as α, β, and γ) at each execution.

Évora identifies and stores multiple matches of execution paths for each *nsc*, to traverse later and find the best fit for any user-defined policy for a given NSC. Finally, *pc* is added to the *matchingSubgraphsMultiMap* against the key, *nsc* (line 18). The caching of multiple options ensures a quick reaction to the dynamic changes of the edge nodes, such as node downtime, service interruption, resource overutilization in any given service deployment, and network congestion. *pc* is then removed from the *partialNSCs* list since the NSC is now complete and therefore no further service needs to be added to it (line 19). The algorithm thus finds all the potential execution paths until it finishes traversing the nodes detected through the MOM messages (line 23). Consequently, the *initNSC* procedure traverses the node graph, to find all the complete matching NSC execution paths and stores them accordingly in the service graphs. By a complete search with all the nodes in the graph, and then choosing the NSC with the minimal penalty value, *Évora* ensures that the optimal NSC is chosen.

7.2.3 Resilient and Adaptive Scheduling of the NSCs

Algorithm 9 presents the orchestrator procedure of scheduling the user-defined NSC. It takes as input the *nsc*), the relevant policies that need to be satisfied and optimized through the MILP models (*policy*), and a pointer to the iTraverse() function interface as the input parameters. First, the *scheduleNSC* procedure initializes *matchingSubgraphsMultiMap* and *nscGraph* for the relevant NSC execution, if they were not initialized previously. *initNSC()* procedure (elaborated in Algorithm 8) is called to initialize the *matchingSubgraphsMultiMap* (line 5). Each NSC is

defined along with the policies that it needs to abide by, such as minimal cost or minimal latency. The algorithm chooses the best fit among the matching execution paths from the multimap based on the user-defined policy, to resolve the MILP models (line 7). Since the penalty values for each NSC is stored as metadata against each NSC entry in Algorithm 8, we can solve the minimization/maximization problem by a simple integer substitution (of α, β, γ, $\ldots$).

Algorithm 9 Scheduling an NSC at the Edge

```
1: procedure SCHEDULENSC(nsc, policy, iTraverse())
2:     nscGraph ← nscMap.get(<nsc, policy>)
3:     if (nscGraph = ∅) then                 ▷ One-time initialization per <nsc, policy>
4:         if (matchingSubgraphsMultiMap.get(nsc) = ∅) then
5:             initNSC(nsc, iTraverse())
6:         end if
7:         nscGraph ← matchingSubgraphsMultiMap.getBest(nsc, policy)
                                   ▷ Resolve the MILP models based on the policy.
8:         nscMap ← nscMap ∪{(<nsc, policy>, nscGraph)}
                       ▷ Replace the ∅ value set in Algorithm 7 for the entry.
9:     end if
10:    nextNodes ← nscGraph.getFirst()        ▷ For parallel execution with multiple first VNFs
11:    for all (node ∈ nextNodes) do
12:        node.schedule(nscGraph)
13:    end for
14:    broker.sendEvent(nsc, policy)
15: end procedure
```

nscGraph defines the chosen execution path or the chain of VNFs executing the NSC workflow. *nscGraph* is stored against the <*nsc, policy*> tuple in the *nscMap* (line 8). The first nodes to initialize the NSC workflow are identified from *nscGraph* (line 10). The VNF execution is scheduled at the first nodes in the execution path (lines 11 - 13). As the NSC may consecutively invoke two or more parallel service deployments, the execution is generalized for distributed and parallel execution of NSC. An event is sent to the respective broker using *sendEvent()* (line 14), with information on the NSC and policy.

7.3 Implementation

We prototyped *Évora* by extending OpenDaylight with VNF orchestration capabilities. We used Oracle Java 1.8.0 as the programming language and ActiveMQ as the MOM broker. We emulated user devices consisting of an Event Manager that connects with the brokers through MOM messages and an Orchestrator that orchestrates the user NSC execution.

Figure 7.2 presents an illustrative deployment of the edge nodes and the user device. It also demonstrates how the orchestrator of the user device visualizes the hypergraph of the edge node services in a *graph of complete graphs*. The edge nodes are detected through the MOM messages, by subscribing to the relevant notifications. The user chooses to find only the relevant VNF nodes at the edge. Thus, she can effectively control the size of the graphs based on her interests in various VNFs. The size of the *Évora* graph representation in each user device can

be anywhere from a few KBs to several MBs. *Évora* represents each node by a complete graph. Each link to the node that connects it to adjacent nodes or the Internet is depicted by multiple links to each of the service in the nodes. The connected graphs representing each node consists of vertices that denote the services connected by stronger links than the inter-node links of the hypergraph as illustrated.

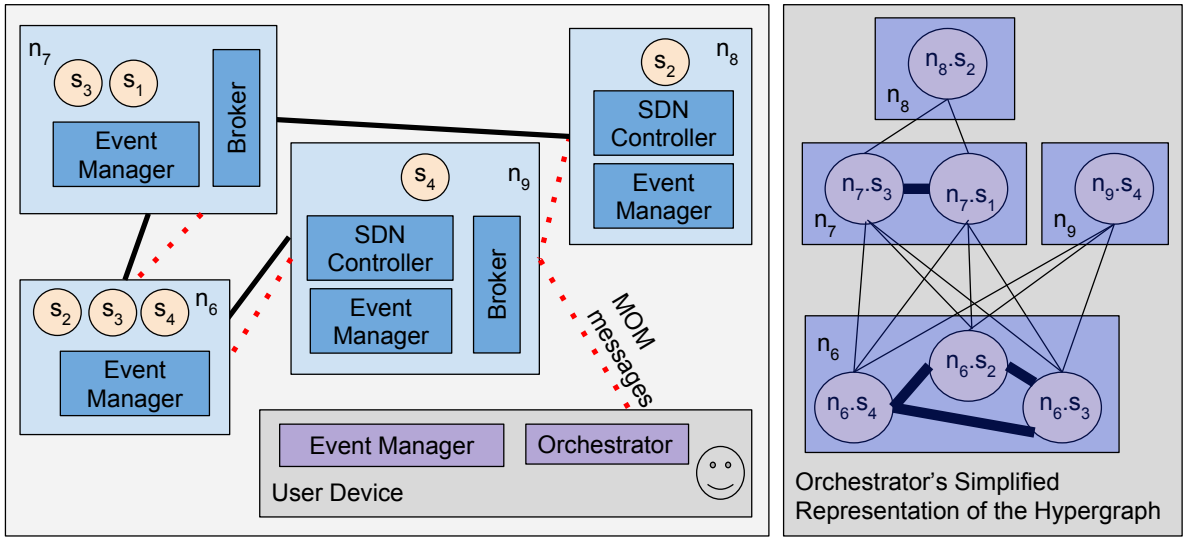


Figure 7.2: An *Évora* deployment: Edge Nodes and the User Device

Some nodes have a broker, while every edge node consists of an Event Manager. Event Manager is a thread that functions as an event publisher and event listener, to publish messages to the subscribers through the broker, and subscribe to notifications for relevant events. The user device is thus subscribed to the broker instances to know the status of the edge VNF deployments. We developed the interaction between the Event Manager and the broker through AMQP messages. The controllers manage the node networks, with the ability to dynamically program the network. MOM messages do not necessarily require a static link, as they can go through over the Internet or an overlay network, and can be formed dynamically without a predefined hierarchy.

There should be a secured connection between the nodes for the data transfer between the service executions in an NSC. Edge nodes of a single provider in a neighborhood are typically connected to form a private network to minimize latency, reduce expenditures and security risks caused by utilizing the public Internet for local inter-node traffic, and maximize throughput. Unlike a stand-alone VNF execution, NSC requires a workflow of multiple services. A VNF provider typically guarantees a protected connection for live migration of data between her VNF offerings. *Évora* ensures completeness of a user NSC execution through the availability of a connected graph of nodes consisting of all the services in the NSC, despite leveraging service instances from different providers to compose the NSC.

In a typical edge user scenario (such as an edge data center used for network services), the user is subscribed to offerings from particular edge nodes and will have dedicated connections

established between the edge nodes and the user on-premise deployments and servers to avoid latency. In a mobile environment or an execution environment composed of many light-weight edge nodes, the user needs to develop links dynamically through the Event Manager. As with the cloud platforms and web services, edge nodes should provide standardized APIs for the end users to interact with their service offerings, and support inter-node migration of data and execution. Unless a single edge VNF provider offers all the related network services, it is fair to expect that they provide relevant secured APIs for NSCs, to maintain a competitive business model.

7.4 Evaluation

We assess *Évora* algorithms with microbenchmarks to evaluate its performance and ability to abide by the user policies.

7.4.1 Problem Size and Scalability of *Évora*

Évora represents the edge services hypergraph as a graph of complete graphs, as elaborated in Section 7.1. The number of links in the node graph depends on the average degree of each edge node, in its connections with other edge nodes, and the total number of edge nodes considered in a given deployment of *Évora*. First, we evaluate the growth of the service graph representation against the increasing number of services in each node and the average number of links between the nodes.

Figure 7.3 illustrates the growth of the problem space of constructing NSCs at the edge. Just with 10 VNFs per edge node, and nine links in the node graph, *Évora* service graph reaches 1000 links. We observe that the number of links in the service graph representation grows linearly with the number of links between the edge nodes, and exponentially with the average number of services per edge node. The growth of the alternative parts composing the NSC illustrates the scale of the resource allocation problem.

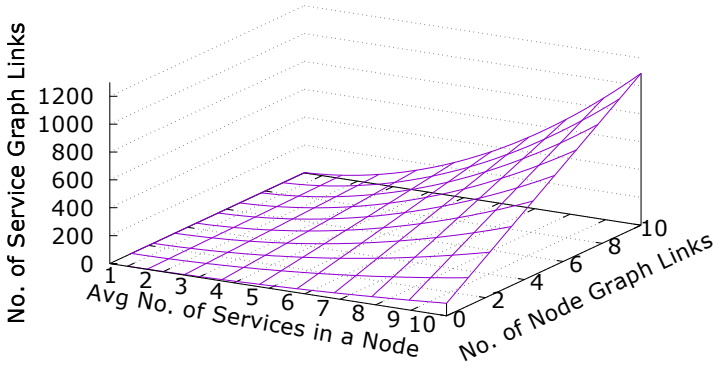


Figure 7.3: Representation of the service graph from the node graph

Table 7.2: Performance and Scalability of *Évora* Orchestrator Algorithms

Algorithm	**Frequency**	$\Theta(f(n))$
1: Construct the graphs	Once per user device	$\Theta(n^2)$
1: Updates to the graphs	Upon an update message from a broker	$\Theta(1)$
2: Finding potential VNFs for an NSC	Once per user NSC definition	$\Theta(n^2)$
3: Choosing the best-fit VNFs	Once per $< nsc, policy >$ pair	$\Theta(n)$
3: Policy-aware NSC scheduling	Once for each NSC execution	$\Theta(1)$

We further profiled the performance of *Évora* with microbenchmarks to evaluate its performance with scale. Table 7.2 lists how the orchestrator in the user device performs in various stages of its execution. The most time-consuming tasks are constructing the graphs initially and finding the potential VNFs for an NSC via graph matching for the first time. Constructing the edge and VNF graphs are performed at the initialization of the user device by the Algorithm 7. The time it takes is bound by the number of edge nodes and the number of brokers. The number of brokers can be at most as high as the number of edge nodes, in the worst case scenario for the performance of *Évora* orchestrator. In the best case, the number of brokers is much smaller in number compared to the number of edge nodes. Therefore the *Évora* orchestrator takes $\Theta(n^2)$ time to initialize the graphs, where n represents the number of edge nodes that the user device identifies. The data is stored in indexed map structures. Therefore, the subsequent update messages received from the brokers take $\Theta(1)$ time to keep the graphs updated, regardless of the number of edge nodes and VNFs present. Therefore, with time, the overhead caused by the edge notifications in practice remains constant. In the worst case, if most of the edge nodes send update notifications at once, the update time will be $\Theta(\log(n))$.

Once the graph is constructed, the orchestrator needs to find the potential VNFs for a given NSC, by executing Algorithm 8. The MILP models on maximizing or minimizing specific parameters are not considered in the initial graph matching to find and store all the matching pairs of VNFs. Hence, this step is performed regardless of the user policies. The number of VNFs in a user NSC is in practice less than 10. It does not grow into as large as the number of vertices (in the scale of thousands or even more, considering the future potential for pervasive edge nodes) and links in the graph representation of the edge environment. Therefore, *Évora* graph matching does not grow into an optimal graph matching problem that typically takes exponential time. *Évora* graph matching is performed as an execution of graph search for the times of the number of VNFs present in the user NSC. The intermediate results are stored in a variable ($partialNSCs$). The *Évora* graph matching takes quadratic time by leveraging a dynamic programming approach with the intermediate results. The time consumed depends on the number of VNFs in the graph. Furthermore, in practice, this graph matching is often executed just once, along with the initialization of the graphs. The subsequent updates are handled within a logarithmic time scale, similar to the graph initialization.

Finally, solving the MILP models for each $< nsc, policy >$ pair takes in all the cases $\Theta(n)$ time. Once the VNFs are chosen for a user NSC abiding by the policies, the subsequent overhead to the NSC execution from *Évora* orchestration is of a negligible constant time. Thus, by using its efficient map data structures to store the graphs, *Évora* minimizes the overheads on more

frequent executions, while making the MILP optimization time linear. Furthermore, by storing the previously computed results such as all the matching VNFs, *Évora* handles the updates more efficiently, from constant to logarithmic times in the average and worst case scenarios. While the initialization process itself takes a few seconds (depending on the computational power and the number of edge nodes) in the presence of thousands of edge nodes, this is a one-time initialization process.

We observed that *Évora* optimally and efficiently allocates the user NSCs across the edge VNFs, despite the presence of 1000 edge nodes, along with a million service graph links. We measured and estimated the time overhead from *Évora* for the first execution of an NSC to be less than 0.3%, taking a few seconds when the NSC executions (such as browsing the Internet through a protected NSC with firewalls, parental control, and load balancers) are in the scales of minutes to hours. Furthermore, except for the first flow of a user NSC, all the subsequent flows do not incur an overhead from the *Évora* algorithms.

7.4.2 Efficient VNF Allocation at the Edge

We evaluated the accuracy of *Évora* (to assess its effectiveness in the sense of how accurately its decisions reflect the user's intent) in VNF allocation at the edge, based on user-defined penalty functions consisting of one to three attributes: from latency (ms), throughput (Mbps), and monthly cost ($).

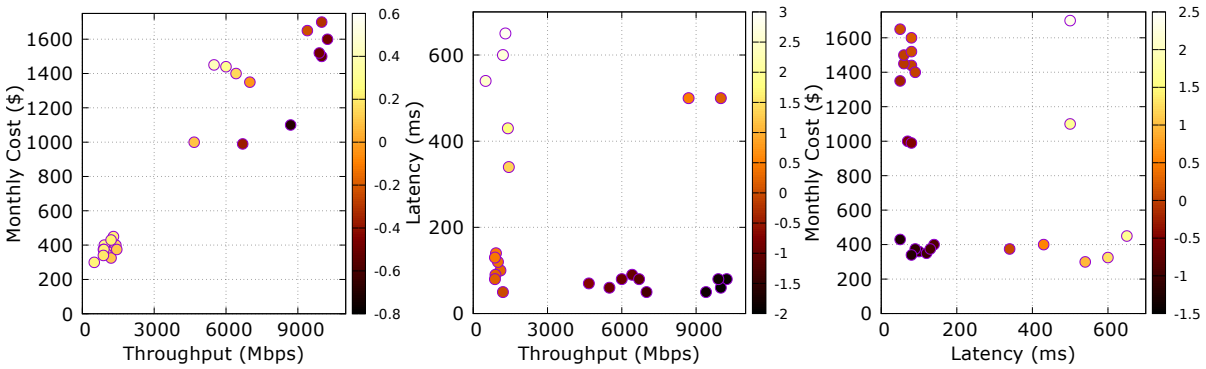
When handling two or more attributes, the penalty function gives weight (through user-defined policies) to the normalized values of each of the metrics (e.g., throughput and monthly cost) across each one's value range (e.g., from 0 to 12000 Mbps; from 0$ to 1800$, respectively). *Évora* normalizes the attribute values using feature scaling as illustrated by Equation 7.12. The $\max(X)$ and $\min(X)$ indicate the potential maximum and minimum values for any edge node for the attribute X.

$$\forall x \in X : \hat{x} = \frac{x - \min(X)}{\max(X) - \min(X)} \tag{7.12}$$

In the following figures, the more accurate (and thus better performant) *Évora* is regarding the user's intent, the lower the penalty function is, and the darker the circles are (so, lower penalty and darker circles are better). These darker circles should thus occupy regions of the graphs where the (possibly combined) metrics of interest to the user exhibit better values.

Two Variable Attributes in the User Policies: Figure 7.4 depicts the performance of *Évora*, calculating a composite penalty value from two variable attributes (each with an equal weight of 1) and minimizing the penalty value for a resource allocation adhering to the user-defined policy. The Figures show darker circles where the penalty function is lower (and therefore indicating closeness to what it captures as the user's intent).

In Figure 7.4a we can see that most darker circles are closer to the highest values of throughput (one aspect of the user's policy) but in the top-right corner (higher cost). Nonetheless, we



(a) Max. Throughput and Min. Cost (b) Max. Throughput and Min. Latency (c) Min. Latency and Min. Cost

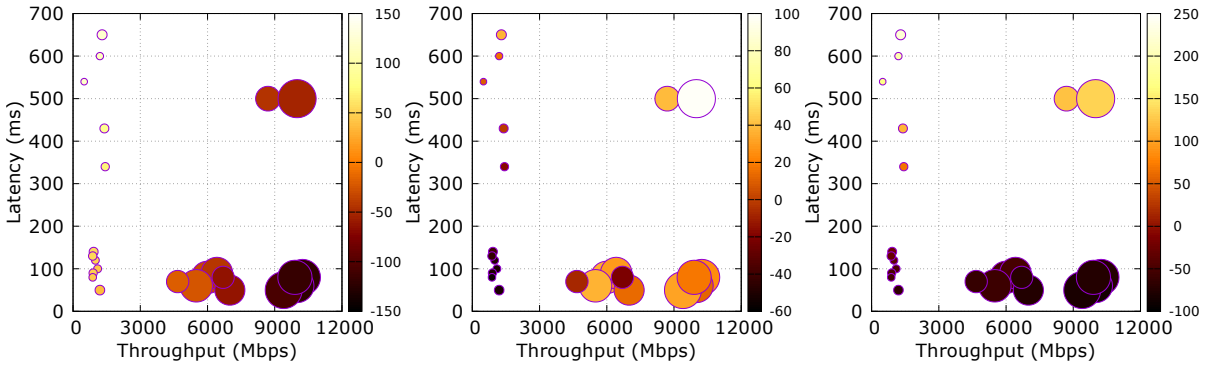
Figure 7.4: *Évora* policies with two attributes of equal weight.

note that there are also darker circles (two) on areas with lower cost ($1000, i.e., with significant savings from $1600) than other circles that offer lower bandwidth with a higher cost (i.e., lighter circles on the $1400 line). This observation indicates that our penalty function can award some of the lowest values (and therefore identify those as of higher interest to the user) to the combination of 9000 Mbps and around $1000 cost. This seems the best combination for the user indeed (remember she gives equal weight to monetary cost and throughput).[3] Figure 7.4b shows that the darker cycles are more present in the lower-right area of the chart. Thus, the penalty function can identify how to maximize throughput (well over 9000 Mbps) while minimizing latency (with equal weights) with great effectiveness (once again, any of values with the lowest distance to the lower-right corner would be optimal). Similarly, in Figure 7.4c, the darker circles are predominantly in the lower-left region, showing that the penalty function can capture solutions that minimize cost (lower than $400), with very reduced latency (as good as with solutions costing $1000 and downwards).

Three Variable Attributes in the User Policies: We then evaluated the VNF allocation considering all three attributes: latency, throughput, and monthly cost. Figure 7.5 illustrates the performance of *Évora* giving prominence to one of the three attributes while considering the others as a secondary preference, in a three-dimensional space. The attribute with the highest prominence is given the weight of 10, while the other two are given the weight 3. The monthly cost of the NSC is depicted by the radius of the circles, while the throughput and latency are illustrated by the x and y-axes respectively. Similarly, Figure 7.6 demonstrates the performance of *Évora* giving prominence to two or all of the three attributes equally.

In Figure 7.5a we can see that most darker circles are closer to the highest values of throughput (higher than 9000 Mbps). We also note that with equal throughput, the darkest circles are at the bottom-right, indicating lower latency in addition to the high throughput. As the cheaper services offered lower throughput, they are not chosen by *Évora*. Figure 7.5b shows that the

[3]It is an extra step, not hard under these circumstances, to identify this more adequate value (even though, for completeness, this could entail, e.g., exploring the Pareto frontier of the optimal solutions, that we do not consider here and leave for future work).



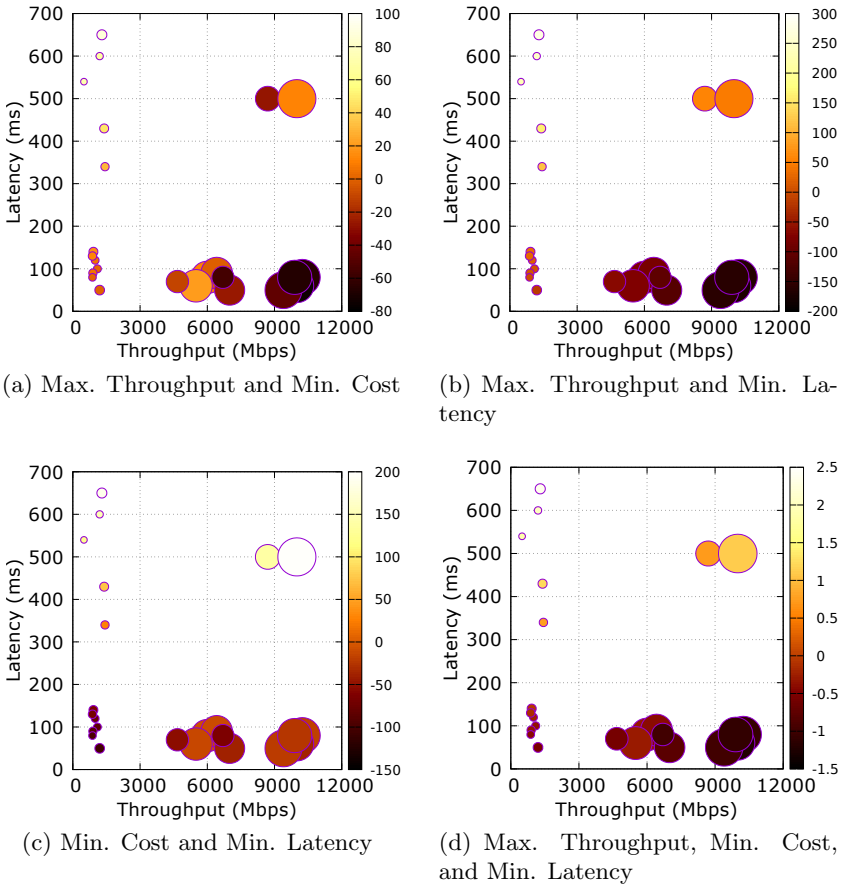
(a) Max. Throughput (b) Min. Cost (c) Min. Latency

Figure 7.5: *Évora* policies considering three attributes with prominence to one of the three attributes. The radius of the circles represents the cost.

dark cycles are those with the smallest radius, indicating how the cost is minimized. Moreover, still, the darkest circles are found at the bottom of the plot, indicating the minimal latency. Since higher throughput services typically entailed a higher cost, it was a trade-off that lower throughput services were chosen in favor of a lower cost. Similarly, in Figure 7.5c, the darker circles are predominantly in the bottom region, showing the penalty function can capture solutions that minimize latency (lower than 200 ms). Among those, the circles with the smallest radius (cheapest services) and those in the bottom-right (higher throughput) remain the darkest, indicating the efficacy of *Évora* in adhering to the user policies.

Figure 7.6a, Figure 7.6b, and Figure 7.6c give prominence to two of the three attributes equally (10), while still considering the third attribute (with a weight of 3). Figure 7.6d gives equal prominence (weight of 1) to all three attributes. Figure 7.6a displays the darkest circles at the highest values of throughput (higher than 9000 Mbps), while also having darker circles with smaller radius indicating the cheaper services (as found in a cheaper service offering around 6000 Mbps). Since latency was not given prominence, it can be noted that a smaller circle (lower cost) with high throughput (9000 Mbps) still has a dark complexion though it had a high latency (500 ms). Figure 7.6b shows the darkest circles in the bottom-right corner, showing high throughput (higher than 9000 Mbps) and lower latency (less than 100 ms), albeit with higher prices as the cost was not given prominence. Figure 7.6c contains the circles with the smallest radius on the bottom as the darkest, indicating the success in identifying choices of minimal cost and minimal latency (less than 150 ms). However, as a trade-off, here the services with lower throughput were chosen. Figure 7.6d attempts to give equal weight to all the properties. In doing so, it manages to maximize the throughput (typically above 9000 Mbps) and minimize the latency (typically below 100 ms). It also has dark circles with low radius, indicating the preference for cheaper services.

Our evaluations highlight how *Évora* supports user-driven policies effectively in selecting the NSCs, considering one to three attributes with multiple user-specified policies (including various attributes, normalized with weights). *Évora* provides a trade-off between the various attributes, with their normalized values and a user-specified weight for each of those attributes.



(a) Max. Throughput and Min. Cost

(b) Max. Throughput and Min. Latency

(c) Min. Cost and Min. Latency

(d) Max. Throughput, Min. Cost, and Min. Latency

Figure 7.6: *Évora* policies considering three attributes with prominence to two or all of the three attributes. The radius of the circles represents the cost.

The evaluations also demonstrate its capability of supporting policies with contradicting and conflicting outcomes - such as maximize throughput and minimize cost, when the high throughput service chains incur a higher cost. *Évora* can handle complex user policies with more than three attributes defining the penalty value that needs to be minimized, by considering more attributes as n-dimensional environments.

Évora looks at the service chain placement from the user perspective, whereas the previous works [36] make the VNF provider as their focus. LSO [234] encompasses the network softwarization for end-to-end management, orchestration, and control of network services. Standardization efforts, such as LSO and TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) [210], aim for complete end-to-end workflow orchestration. The *Évora* approach aligns with these efforts on service orchestration. It expands the existing research on network softwarization and LSO into edge computing environments, which are more dynamic than the traditional cloud and data centers. A hybrid network with dedicated physical/hardware network functions along with VNFs can offer elasticity to the network functions and support bursts in demand. Efficient placement of such a hybrid deployment can improve the throughput of NSCs, as shown through ILP models in previous research [243]. We can extend and adopt *Évora* for

these hybrid networks at the edge and enhance their efficiency.

7.5 Conclusion

VNFs are placed at the edge to support the overwhelming demand for latency-aware service execution. However, resource allocation at the edge for user NSCs is an NP-hard problem as various user policies must be satisfied to assure QoE while choosing the VNF instances to schedule the NSC execution. We researched the optimal service instance allocation in a workflow as a minimization problem, by assigning weights to various parameters (such as latency, throughput, and cost) defined by the user. We generalized the challenge of resource allocation, service discovery, and workflow migration on the edge platforms for an adaptive execution of service workflows as MILP problems. We solved the MILP problems with a graph-based approach, to optimally compose NSCs by leveraging the third-party edge VNFs from multiple providers.

We designed *Évora*, extending SDN and MOM for an adaptive execution of user NSCs, consuming several third-party VNFs. *Évora*, with its software-defined approach, schedules the VNF executions in each edge node that offers the particular services, ensuring that the user policies are met in composing her NSC. *Évora* thus helps to compose NSCs that are policy and latency-aware, in a scalable user-centric manner, in multi-domain edge environments. We evaluated the *Évora* models and algorithms for NSCs with various user-defined policies. Our evaluations highlight how *Évora* provides resilience and agility to the NSCs in the MEC environments through its orchestration of multiple execution paths spanning the edge nodes.

8 Software-Defined Cyber-Physical Systems

CPS faces several challenges in design and performance, due to the scale and variety in its devices and execution environments [199]. Deploying the workloads at the edge can offer a high-performant execution for the latency-sensitive CPS applications [268]. However, the current cloud and edge environments often do not favor a seamless deployment and smooth frequent migrations of diverse CPS workloads between the execution environments. Therefore, the number of edge nodes that can contribute their resources to the execution of a CPS workload is significantly limited. We posit that executing the CPS workloads as web service workflows can offer unified deployment and execution, as web services are developed following standards. We propose an SDS framework to efficiently manage CPS workflow scheduling at the edge with capabilities of frequent communication and workload migration.

Challenges: We first identify a set of core challenges faced in executing and managing CPS:

(Ch_1) Unpredictability of the execution environments [198]

(Ch_2) Orchestrating the communication and coordination within the CPS [273].

(Ch_3) Security, distributed fault-tolerance, and recovery upon system and network failures [68].

(Ch_4) Decision making in the large-scale geo-distributed execution environments [308].

(Ch_5) Modeling and designing the complex CPS environments [99].

(Ch_6) Management of the intelligent agents [64].

Motivation: Given the above premises, we aim at addressing the following research questions:

(RQ_1) Can SDN or a more encompassing network softwarization approach inspired by SDN help mitigate the identified challenges of CPS that hinder its wide-scale adoption?

(RQ_2) Can we seamlessly scale such an approach beyond data centers, to a wide area network, for modeling and executing CPS?

(RQ_3) Can we leverage the edge resources for a distributed execution of CPS workloads through a unified control, without additional overheads?

(RQ_4) Can we generalize the CPS execution as service workflows at the edge, to support interoperable execution across wide area networks such as edge and cloud-assisted networks?

Contributions: The goal of this chapter is to answer the identified research questions to mitigate the major challenges faced by CPS. The main contributions of this chapter are:

1. A generic framework for building and executing CPS through service workflows at the edge, to support interoperable execution of the workloads in multi-domain networks (RQ_3, RQ_4 and Ch_2).

2. Software-Defined Cyber-Physical Systems (*SD-CPS*), a scalable and distributed SDS, to coordinate the heterogeneous CPS devices by extending SDN with MOM protocols [87] for wide area networks (RQ_1, RQ_2, and Ch_4).

3. Incorporating user policies and resource requirements for efficient CPS resource allocation and migration. Thus, offering efficient control of the network resources for the user applications, despite the unpredictability of the CPS networks that are shared across several users (Ch_1).

4. A reusable execution model for the CPS workflows to modeling, incremental development, and seamless execution in a sandbox and production environments. Thus, managing the execution of CPS in the physical and cyberspaces effectively with minimal duplicate effort (Ch_5 and Ch_6).

5. Resilient and agile CPS execution by offloading the CPS workload as service compositions at the edge, by exploiting a logically centralized control of the edge resources (Ch_3).

SD-CPS mitigates the complexity of the computation and resource scarcity by decoupling and decomposing the execution of CPS into interoperable workflows of microservices and offloading the workflows to edge environments. As small autonomous and loosely-coupled services, an SOA with microservices [246] enables extensible modularized architecture for *SD-CPS* workflows, compared to typical web service composition workflows. Discovering the resource availability and deploying the workflow as service invocations at the nodes need to be performed effectively to reap the benefits of the edge. We design a controller deployment as the core of *SD-CPS* to achieve the overall coordination to manage the "cyber" of the CPS and orchestrate the CPS elements. *SD-CPS* executes CPS in a programmable and predictable manner, inherently addressing many of the operational challenges of CPS with its software-defined approach. The quantitative evaluations on a *SD-CPS* prototype highlight its efficiency in resource allocation through edge workflows and success rate in CPS execution modeling.

This chapter is composed of the contents of the publication: [J2, C6, W6].

8.1 MANETs and VANETs: A Case for *SD-CPS*

Complex computations at the mechanical or physical devices of CPS often require resources beyond what is available physically on the device, such as a smart vehicle or a mobile terminal. Hence, workloads heavy in computing or memory are delegated to the *cloud-based cyberspace*,

instead of executing them (often only as firmware) in the smart terminals of the CPS. In this section, we will look into a few common characteristics of CPS that motivate the case for an SDS framework for CPS, by discussing MANETs and VANETs as potential cases for *SD-CPS*.

MANETs are comprised of mobile devices, that can function independently as autonomous systems as well as sensors, while also communicating among themselves. A typical example of a MANET device, a smart mobile phone, can sense its environment, including i) background noise level through its audio sensor (microphone), ii) light/vision through its camera, and iii) motion through its motion/shock detectors. In addition to the sensing capabilities, a smart mobile device also has computing and memory resources (that can be leveraged to detect, analyze, and monitor user activity such as standing, walking, running, and cycling), though in a limited capacity compared to the traditional computation devices.

VANETs include autonomous automotive systems such as networks composed of self-driving vehicles [12] or smart vehicles [331]. A self-driving vehicle depends heavily on the contextual information sensed by itself as well as those shared by other smart vehicles, due to the absence of an experienced human driver. Smart vehicles collaborate and coordinate with one another, to share information such as current traffic, and dynamically decide their traveling path by analyzing the accumulated dynamic data, in real-time. While VANETs can be considered a subset of MANETs, VANETs have a higher degree of dynamism and speed than the other MANETs such as mobile terminals. VANETs are expanding their scope beyond the road traffic, also to consider the air traffic monitoring and aircraft control [293]. These latest advancements have widened the research challenges associated with the CPS.

Research identifies networking and message exchanging problems caused by the dynamism in VANETs [361]. An automobile such as a connected car [134] or an aircraft can pass through various control stations, while on the move. Therefore its proximity to sensors (including satellites for the air traffic control) or computing nodes (for connected cars) in the VANET differs with time. Thus, the current updated location of a smart device or terminal needs to be considered when some data is routed towards it. VANET CPS needs to route various messages from adjacent sensors or other vehicles towards a moving vehicle considering its current geographic location and ensure that an ongoing flow is served following the previous route to avoid data loss or inconsistency.

Exploiting SDN for CPS require further research beyond the typical traffic engineering use cases of SDN due to the scale and variety of CPS. SDN has been adopted for VANETs to coordinate the data and workload scheduling efficiently in a centralized manner [212]. Previous research proposes a cloud-assisted execution for context-aware vehicles [343]. A functional *SD-CPS* framework requires extending the SDN research on complex traffic engineering problems [8] catering to the complexity of CPS.

8.2 Solution Architecture

SD-CPS consists of multiple *tenants*, the users who configure and deploy their CPS workload as workflows composed of web services, abiding by their respective policies. *SD-CPS* virtually

associates each workflow with its tenant at the edge nodes despite sharing the execution space with other tenants. Each tenant defines SLOs for her CPS workflows, scheduling them based on tenant-defined policies. Consider a traffic analysis workflow w belonging to a VANET CPS. It is composed of various services, including i) data sensing at each of the vehicular sensor devices, ii) data matching from numerous adjacent vehicles for data correction to minimize noises, iii) data integration at an edge node, iv) analytics based on current and past data, v) traffic prediction on each route, and vi) receiving or sending personalized information from the edge cloud back to the vehicle. These services would typically be running iteratively or periodically, thus continuously carrying out monitoring, assessment, decision, and actuation activities. Equation 8.1 illustrates a generalized form of such a CPS workflow, composed of various services.

$$\forall x \in \mathbb{Z}^+ : w = s^1 \circ s^2 \circ \ldots \circ s^x. \tag{8.1}$$

Equation 8.2 represents w_i, a distributed execution of w by the same set of services, but with multiple instances in parallel. Here, a instances of s^1 run and send their outcomes to b instances of s^2. Eventually, the service workflow completes with the invocation of n instances of s^x with the previous service composition invocation ($s^1 \circ s^2 \circ ..$) outcome as its input.

$$\forall x, a, b, n \in \mathbb{Z}^+ : w_i = a \cdot s^1 \circ b \cdot s^2 \circ \ldots \circ n \cdot s^x. \tag{8.2}$$

The CPS data flow goes through various intermediate nodes, from the sensors that collect data as input devices, to the actuators that perform actions based on the contextual data. These links between the nodes form the execution paths of *SD-CPS* workflows. Decomposition of the workload into smaller deployable services (e.g., containerized microservices) at the edge helps increase the perceived path redundancy of the workflow. *SD-CPS* exploits an extended controller deployment, decentralized at Internet scale, to manage the metadata that governs the alternative execution paths for each workflow of a CPS execution. The workflows can have different replication levels for each service, based on the user policies indicating the importance of each of them. *SD-CPS* aims for increased resilience, load balancing, and congestion control among the underlying network paths with the path redundancy and the global awareness of the controller on their existence.

8.2.1 *SD-CPS* Coordination

In the core of the *SD-CPS* ecosystem is the **Controller Farm** proposed in Chapter 6. The Controller Farm coordinates the CPS workflows at the edge efficiently, as the component with the highest processing power. The CPS workload needs to be defined in a format that supports migration and interoperability between multiple edge nodes, due to the small-scale and heterogeneity of these nodes. Therefore, *SD-CPS* represents the workloads as a decomposable service chain, to achieve a seamless microservice workflow across the execution environments. The workload is offloaded to the edge as web service invocations if the resource capacity is limited in the CPS firmware. With a unified view, *SD-CPS* chooses the workflows to be executed in its cyberspace or at the edge, based on the resource availabilities. Thus, *SD-CPS* develops a

seamless approach to modeling a CPS in cyberspace and executing it in the physical environment, consisting of the mechanical devices (physical space) as well as the edge nodes as surrogates or extended cyberspace.

SD-CPS executes the same code of the CPS physical devices in a Modeling Sandbox first, before executing them in the production CPS environment, thus reducing the unpredictability of the CPS executions. CPS applications are often modeled as software simulations before being built as device firmware or deployed into the physical devices of CPS. *SD-CPS* performs a simulation through placeholders that can be controlled by its controller, in the same way the controller coordinates the modeled physical systems. The *SD-CPS* controller consists of the **Modeling Sandbox**, an extension to the Simulation Sandbox presented in Chapter 3. The Modeling Sandbox functions as a controlled space to execute the CPS models. Figure 8.1 represents how the systems are modeled in the sandbox environment of the *SD-CPS* controller. The counterparts of the CPS physical space are modeled in the cyberspace as *Virtual Intelligent Agents*. The interactions among the CPS counterparts in the physical space are mapped between the Virtual Intelligent Agents to reflect the communications in the cyberspace and to model the workflows better.

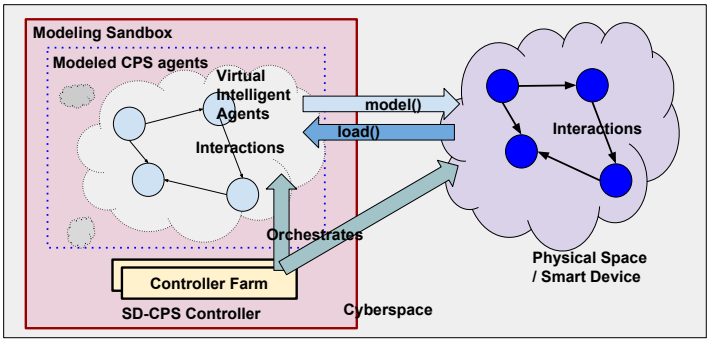


Figure 8.1: CPS Design and Development with *SD-CPS* Approach

The Controller Farm orchestrates both the physical systems and their simulated counterparts in the cyberspace. By simulating the API that connects the physical space into the cyberspace, *SD-CPS* creates a one-to-one mapping between the simulated Virtual Intelligent Agents and interdependent components of the physical system. The interactions are modeled and closely monitored in the Modeling Sandbox before the decisions are loaded into the physical space. Thus, the Modeling Sandbox seamlessly models the executions in the cyberspace as simulations and emulations and then loads the changes to the physical execution environment. The simulation functions as a virtual proxy for the designed system that it simulates - including its actors, such as sensors, actuators, and other physical and mechanical components. *SD-CPS* thus builds once and executes the same code in the controller's Modeling Sandbox or the physical space, reusing the same single development effort.

The *SD-CPS* modeling approach minimizes the code duplication by executing the real code from the controller, instead of having a simulation or model running custom code, thus independent of the actual execution. The *SD-CPS* controller, devised as an extension to OpenDaylight,

Table 8.1: Notation of the *SD-CPS* Representation

$\mathcal{N}$	The set of nodes (including the cyberspace and the edge nodes)
$\mathcal{L}$	The set of links that connect the CPS to $\forall n \in \mathcal{N}$
R	The set of computing resources (CPU, memory, ...) in a node
$P_{\mathcal{N}}$	The set of static properties of a node (availability, up time, ...)
$P_{\mathcal{L}}$	The set of static properties of a link (multitenancy, QoS guarantees, ...)
S_n	The set of services concurrently deployed in a node n
S_l	The set of services concurrently sharing a link l
X	The set of variables defining the utility value, Δ
l_n	A link towards the node n from the current CPS workload
i_p	Value of a static property p
r_n	Maximum resource capacity of a node
b_{l_n}	Maximum bandwidth allocated to a link l_n
t_{l_n}	Latency of a link l_n
r_s	Maximum resource consumption by a service s
b_s	Maximum bandwidth allocated to a service s

is developed in Java, a high-level language. Therefore, it enables the deployment of custom applications as controller plugins to alter or reprogram the behavior of CPS. It simulates the execution environment and the CPS workload through Java objects inside the SDN controller. The physical system loads the decisions from the cyberspace. The multi-tenant execution space of *SD-CPS* supports parallel modeling of multiple CPS. *SD-CPS* further avoids repeated computation efforts by caching the previously completed service outcomes.

8.2.2 Resource Allocation

SD-CPS deploys the CPS workflows as services across the edge, aiming to satisfy the CPS policies while maximizing the overall resource utilization of the edge nodes. *SD-CPS* leverages the node utilization and health statistical data retrieved from the edge nodes as messages, to estimate the utilization of edge nodes. The health check of the edge nodes ensures that the resource requirements of the services are met, and the available nodes are sufficiently utilized with minimal idling nodes that are connected to the edge.

SD-CPS thus identifies, with the deployment of a service $\overline{s}$ of a CPS workflow $\overline{w}$, how much resources a node n or a link l_n that connects to the node will have in excess. δ_{r_n} and $\delta_{b_{l_n}}$ define the underutilization of a resource r of a node n (with $\overline{s}$ in n), and the underutilization of bandwidth of a link l_n (with $\overline{s}$ in l_n) respectively in Equation 8.3. $\left(\sum_{s \in S_n} r_s\right)$ represents the total resource consumption by the existing service workloads in the node n. Similarly, $\left(\sum_{s \in S_l} b_s\right)$ denotes the consumption of the link resources (by default, bandwidth) by the existing service

workloads sharing the link l. Complete details on the notations are listed in Table 8.1.

$$\delta_{r_n} = r_n - \left(\left(\sum_{s \in S_n} r_s \right) + r_{\overline{s}} \right); \delta_{b_{ln}} = b_{l_n} - \left(\left(\sum_{s \in S_l} b_s \right) + b_{\overline{s}} \right) \tag{8.3}$$

SD-CPS gives equal weight to all the variables considered, including δ_{r_n}, $\delta_{b_{ln}}$, t_{l_n}, and i_p, using feature scaling, as illustrated by Equation 7.12. Equation 8.4 defines a compound utility value Δ_n for each n. Tenants define a coefficient for each of the resources, for their workflows to input the relative importance of each of node or link properties. The coefficients $c_r, c_b, c_t, and\ c_p$ are respectively defined for i) each resource (such as memory and CPU) availability in the nodes, ii) the bandwidth availability in the link towards the node, iii) latency to reach the edge from the CPS workload (i.e., latency between the CPS workfload or the physical device and the execution/surrogate node), and iv) other static properties. *SD-CPS* chooses the nodes with the maximum utility value as the execution space for the services of CPS workflow.

$$\forall n \in \mathcal{N}, \forall l_n \in \mathcal{L}, \{c_r, c_b\} \subset \mathbb{Z}^+, \{c_t, c_p\} \subset \mathbb{N}, P = P_{\mathcal{N}} \cup P_{\mathcal{L}} :$$
$$\Delta_n = \begin{cases} \sum\limits_{r \in R} (c_r \cdot \hat{\delta}_{r_n}) + c_b \cdot \hat{\delta}_{b_{ln}} + \left(\frac{c_t}{\hat{t}_{l_n}}\right) + \sum\limits_{p \in P} (c_p \cdot \hat{i}_p), & \{\forall r \in R : \delta_{r_n}, \delta_{b_{ln}}\} \subset \mathbb{R}_{\geq 0} \\ -\infty, & \{\forall r \in R : \delta_{r_n}, \delta_{b_{ln}}\} \not\subset \mathbb{R}_{\geq 0} \end{cases} \tag{8.4}$$

The inherent properties (such as the QoS guarantees and uptime) of the nodes and the links $P_{\mathcal{N}}$ and $P_{\mathcal{L}}$ are static and do not frequently change (except in the case of failures, which edge providers are supposed to minimize, abiding by the SLAs [316]). However, the node properties such as current available memory and CPU are dynamic and change with time, based on the other service executions. Similarly, the bandwidth of a link is shared with various flows and is limited by a maximum capacity of b_{l_n}. Multiple tenant workflows share b_{l_n}, and each tenant workflow is throttled in bandwidth allocation for fair use of the bandwidth by various tenants. Thus, utilizing a particular link l_n for $\overline{s}$ depends on the existing virtual tenant network allocations.

SD-CPS aims to deploy the services in the nodes that satisfy the minimum resource requirements (identified by a positive real value for each δ), that is also connected by a link to the CPS physical space with the necessary bandwidth. The utility value for the nodes that do not satisfy any of the minimum resource requirements for a workflow is set to negative infinity ($-\infty$) to avoid choosing these nodes for the CPS workflow deployment. When multiple suitable edge nodes (the nodes represented with a utility value other than $-\infty$) are available, *SD-CPS* chooses the one with the maximum utility value. Thus, *SD-CPS* identifies and uses the edge nodes that offer the best QoE to the user, abiding by the user-defined policies.

SD-CPS incorporates its resource allocation approach as an extension to the SDN controller. The controller, based on its contextual information on the nodes and the links, ensures that the resource availabilities in the node and the link that connects the node can support the resource requirements of the service workload. The controller then deploys the services of the workflow, fulfilling their resource requirements while aiming to satisfy the tenant policy in allocating the resources to the workflows. The execution node location for each type of service can be cached in

the CPS devices, thus invoking controller only once before the start of the workflow. Subsequent services of the same nature can execute on the already chosen node, without reverting to the controller for the resource allocation, unless controller notifies of failure, congestion, or a change in the status of the current execution nodes.

8.3 *SD-CPS* Controller

SD-CPS extends OpenDaylight Beryllium with MOM as its core SDN controller. We used Oracle Java 9.0.4 as the programming language for the CPS environments, and ActiveMQ 5.15.2 as the message broker. We implemented the resource allocation algorithms as OpenDaylight extensions. Figure 8.2 depicts the solution architecture of the *SD-CPS* controller. We built the Software-Defined Sensor Networks on top of the Controller Farm, representing each mobile terminal or a smart device as a sensor or an actuator. Tenant-Aware Virtual Network Allocation of *SD-CPS* allocates the bandwidth among the *SD-CPS* tenants who share the execution space. *SD-CPS* controller exposes its functionalities to its users through its APIs following the nomenclature of the SDN controller APIs.

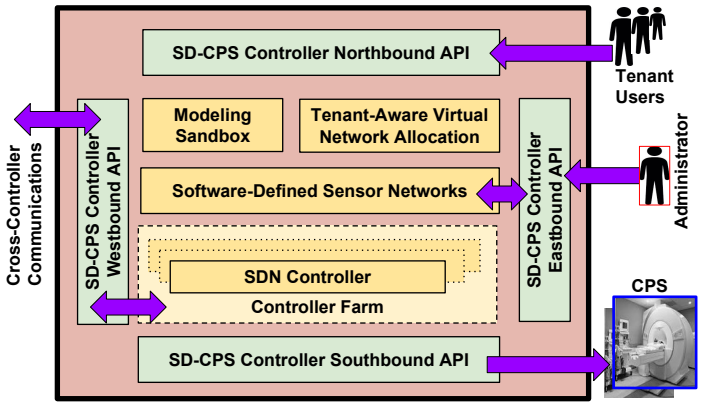


Figure 8.2: *SD-CPS* Controller Architecture

The *SD-CPS* controller APIs consist of the implementation of different integration protocols and connection points for the extended distributed controller deployment for CPS. The Westbound API enables inter-control communication among the controllers in *SD-CPS*, as well as inter-domain communications across multiple *SD-CPS* controller deployments. The *SD-CPS* system administrators leverage the Eastbound API to configure and manage the controller deployment. The Northbound API consists of the common SDN northbound protocols including REST and MOM protocols for the tenant processes to interact with the controller. The Southbound API includes implementations of SDN southbound protocols and MOM protocols for communication with the physical devices.

The *SD-CPS* controller communicates with the network data plane through implementations of SDN southbound protocols such as OpenFlow and interact with the devices that are not SDN-native through various southbound protocols inspired by OpenFlow. The incorporation of

multiple protocols ensures that while *SD-CPS* has SDN at its core, it is not limited to software-defined networks with SDN switches that are still far from widespread in CPS settings (outside data centers). Based on the defined policies, rules, and the tenant inputs from northbound, the controller determines and propagates the relevant actions back to the data plane consisting of the SDN switches and physical devices through the southbound. Thus, the southbound API handles the communication, coordination, and integration of the network data plane consisting of the CPS devices with the control plane.

Figure 8.3 shows the Software-Defined Sensor Networks enabled by *SD-CPS* as its communication medium. Messaging4Transport is an OpenDaylight bundle that we developed to expose the OpenDaylight data tree as messages in a messaging protocol. *SD-CPS* controller stores the dynamic context data sensed by the sensors of the various appliances into the data tree of OpenDaylight MD-SAL. Each device subscribes to the relevant topics while publishing the relevant information detected by their sensor. As the SDN controller is often a super-computer or a cluster of high-end servers, *SD-CPS* ensures efficient communication between its counterparts by leveraging its Software-Defined Sensor Networks approach and storing the dynamic data inside the data tree of the controller.

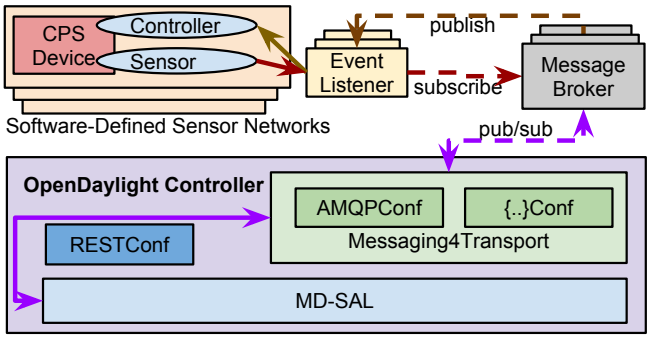


Figure 8.3: Network Layer - Higher Level View

While *SD-CPS* uses AMQP as its default messaging protocol, its design is extensible to use the implementations of the other MOM protocols. ZeroMQ [160] messaging library would offer better scalability and distributed execution due to its lack of brokers and message queues, compared to the MOM implementations of the messaging protocols. ZeroMQ aims at functioning in a truly distributed manner, allowing communications between the nodes rather than having a broker as a coordinating entity between the nodes. However, it also would introduce a management overhead where each CPS nodes must address the other nodes, rather than a commonly known (typically, static) broker. Therefore, we found ZeroMQ to be less desirable for a dynamic environment such as *SD-CPS*, considering the scalability of the network and management overhead as a trade-off. Furthermore, a MOM broker and the SDN controller co-exist in the *SD-CPS* ecosystem in a logically centralized manner, rather than creating a physically centralized bottleneck.

8.4 Evaluation

We deployed the *SD-CPS* controller on a server of AMD A10-8700P Radeon R6, 10 Compute Cores 4C+6G × 4 processor, 8 GB of memory, and 1 TB hard disk. We assessed the performance of *SD-CPS* in a scenario of CPS execution at the edge as service workflows, and its efficiency in composing and managing CPS, through simulations and microbenchmarks.

8.4.1 CPS Execution Modeling

We designed an execution environment with several nodes, and then deployed a VANET CPS prototype with Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I) and Infrastructure-to-Vehicle (I2V) communications [101]. We modeled a traffic data monitoring workflow with *SD-CPS*, as a composition of several parallel instances of a few services.

Execution Nodes at the Edge: We first modeled the execution environment with four categories of nodes: N1: 10,000 Embedded mobile systems in the connected cars, N2: 1000 edge cloud nodes, N3: 100 servers, and N4: 10 larger servers. The resource specifications of each category specify average values. Each node belonging to a category varies slightly from the other nodes of the category within a range of the resource capabilities of the category. Due to its pervasiveness and proximity to the vehicles, N1 has the least latency to most of the devices, while N4 has the highest of the CPU and memory with often poor latency from the vehicles compared to N1. Figure 8.4 depicts how the resource availability of each node is feature scaled (as illustrated by Equation 7.12 in Section 8.2) by the *SD-CPS* controller.

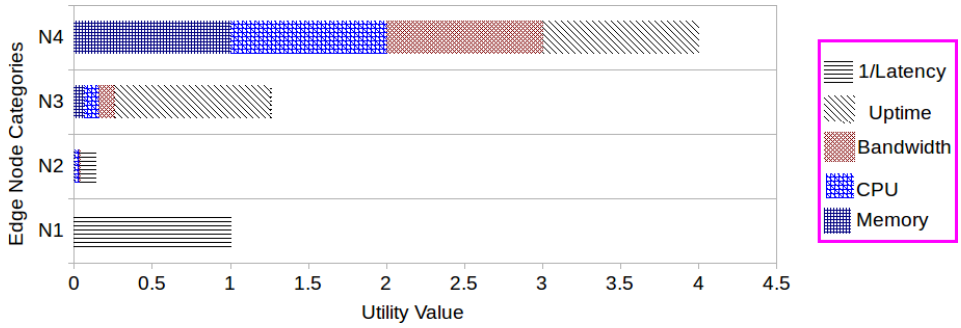


Figure 8.4: Properties of the nodes (normalized)

CPS Workflows: We then modeled a vehicular monitoring workflow W, composing of various services. Service S1 fetches and transforms personalized sensor information to the grid. Due to quick data transfer needs, S1 has a necessity for minimal latency. Throughput, CPU, and uptime are not crucial for its execution. Service S2 performs traffic data cleaning and integration at an aggregator node. It needs a minimal latency and also high memory for quick in-memory computations. S2 aims to output integrated data while avoiding outliers that may indicate malicious, dirty, or bogus data collected from S1. Service S3 performs data analysis for traffic congestion control with contextual data, with moderate latency and memory requirements. Service S4 conducts data crunching with historical data for more data science computations,

including traffic prediction, thus requiring high bandwidth, CPU, and memory (with relatively minimal demands concerning latency). Service S5 personalizes alerts and notifications to the vehicles. Characteristics and requirements of S5 are similar to those of S1.

W is modeled as $W = 1000S1 \circ 100S2 \circ 10S3 \circ 2S4 \circ 1000S5$, with parallel execution of multiple service instances of S1, S2, S3, S4, and S5 composing the workflow. 1000 of S1 service instances send their output to 100 of S2 service instances, which further have their outcome forwarded to 10 of S3 instances, which in turn have their findings sent to 2 of S4 instances. Finally, the relevant results of S4 are sent to the 1000 instances of S5 for the final processing.

Figure 8.5 illustrates the resource requirements of the services in the workflow. A workload is modeled by a set of tuples, <*resource_requirement*, *resource_utility_function_weight*>, in defining the utility value Δ. We set all the coefficients for the resources to 1 in workflow W, for ease of representation. However, *SD-CPS* supports different values for each coefficient. For example, workflows belonging to an ambulance or law enforcement authorities could be given higher priority. Such coefficients give tenants more control in choosing their execution nodes when multiple alternative nodes offer possible deployments to the services.

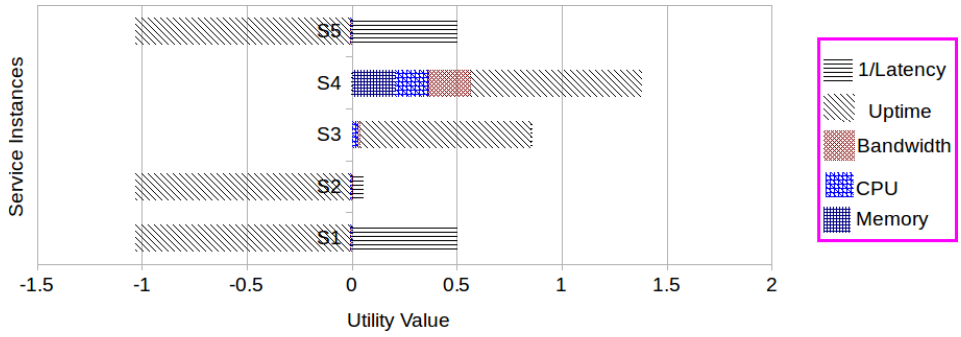


Figure 8.5: Resource requirements (normalized) of the services

The negative normalized values for certain resources indicate that even the minimum available resources in a node is sufficient for the service. For example, S1, S2, and S5 have negative normalized utility value for uptime, indicating that all the node types can serve them, concerning uptime. On the other hand, the highly positive value of uptime for S3 and S4 show their demand for high uptime. Similarly, since S1 and S5 require minimal latency, they consist of a high $\frac{1}{latency}$.

8.4.2 Resource Allocation Efficiency

We finally evaluated the resource allocation efficiency of *SD-CPS* with simulations and microbenchmarks, to confirm that the *SD-CPS* workflow-based approach minimizes idling nodes while ensuring fair resource utilization across all the node types. We deployed 1 million instances of workflow W across several edge nodes over the course of 1 hour. Figure 8.6 illustrates the percentage of services hosted in each node types over a parallel deployment of the workflows. Around 98% of the services were observed to be deployed in the node with the highest utility

value. The deviation for the remaining 2% of services accounts for the full utilization of specific type of nodes at times, thus aiming to maximize the utilization of all the node types.

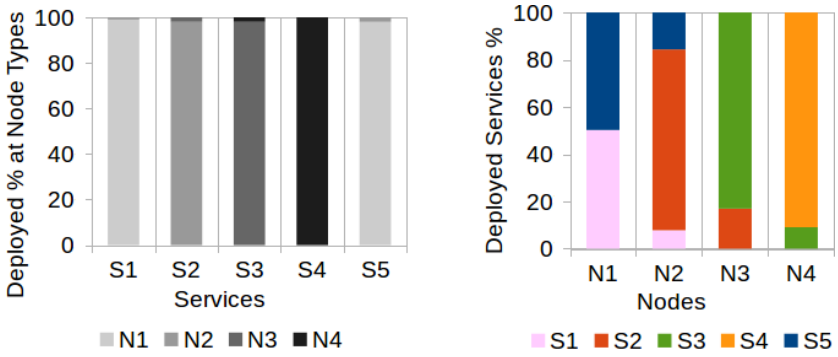
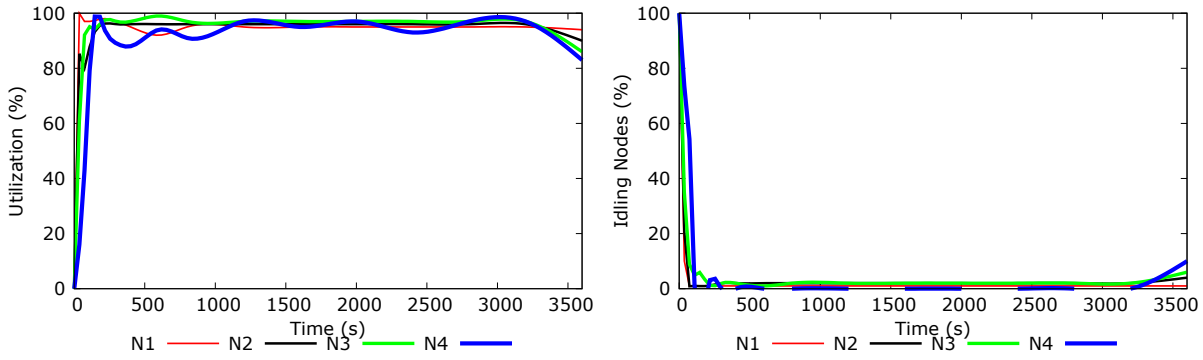


Figure 8.6: Service deployment over the nodes

We assessed the average resource utilization of the nodes and the percentage of idling nodes over the timeframe. Figure 8.7a illustrates the average resource utilization (%) across each node category. The resource utilization was always around 90%, with more variations in resource allocations to N4 nodes. There are just 10 of the N4 nodes, each with abundant resources, yet with high latency. The high latency prevents N4 from executing the more frequently occurring services (S1 and S5), potentially contributing to the relatively frequent drops in resource allocation of N4. The nodes of the other categories remained stable in their resource allocation.



(a) Average resource utilization

(b) Idling nodes

Figure 8.7: Parallel execution of 1 million workflows with *SD-CPS*

Figure 8.7b demonstrates the percentage of idling nodes during the execution of the workflows. Except during the start and the end of the workflows (fixed to have a timeframe of 1 hour), there was near-zero percentage of idling nodes. The results highlight the efficiency of the *SD-CPS* resource allocation, with minimal idling nodes and high resource utilization when there are sufficient requests to utilize the abundant resources.

The preliminary evaluations highlight that by distributing the workload as microservice workflows, *SD-CPS* ensures fair and efficient resource allocation across the edge nodes. We note that the more frequent latency-sensitive microservices that compose the CPS workflows

favor the edge nodes compared to a centralized server for their execution. On the other hand, computation-heavy executions require a larger node, such as a server, even if that incurs a higher latency. Such a fine-grain service placement is facilitated by composing CPS workload as microservice chains, rather than a monolith application, or even a big service in the CPS cyber-space. While resource allocation approaches across heterogeneous execution environments have been presented in the previous work, we note that *SD-CPS* is the first to propose CPS workflow placement at the edge in a scalable, managed, and network-aware manner. Furthermore, we observe that in the presence of such CPS service workflows at the edge, the edge nodes are highly utilized with minimal idling nodes. Although the exact resource utilization and the percentage of idling nodes would depend on the nature of the CPS workflows and the availability of the edge nodes, we note that the simulated CPS is representative of a typical CPS scenario and can be generalized to other CPS environments. By exploiting both SDN and SOA, *SD-CPS* offers better resource allocation, orchestration, and modeling capabilities compared to the traditional CPS. We thus note that by leveraging the abundant proximate smart devices and edge nodes, *SD-CPS* can facilitate a more widespread CPS adoption at the edge.

While SDIoT [174] approaches inspired the *SD-CPS* vision, *SD-CPS* is built from scratch for CPS. Therefore, it differs in scope and implementation to those of SDIoT, though they share similar motivation and can benefit from each other. SDIIoT [342] adopts SDN for IIoT. Although IIoT is related to CPS as CPS is a core enabler of Industry 4.0 [202], SDIIoT limits its focus to safety, reliability, and standardization aspects. Unlike the other approaches that aim to bring SDN to CPS such as *SDCPS* [126], *SD-CPS* does not require a hierarchy of SDN controllers. Thus it natively caters to multi-domain edge environments with multiple providers, with minimal administrative burden. *SD-CPS* caters to the resource provisioning, execution, and migration challenges of CPS, by offering a unified workflow-based approach at the edge, regardless of the specifics of domain, realization, or deployment of a CPS.

8.5 Conclusion

Wide-spread adoption of CPS is hindered by several challenges in terms of building, operating, and maintaining the CPS. The *SD-CPS* framework aims to mitigate the common challenges faced by CPS, through a standard software-defined approach for the design and operation of CPS. As an extended SDN architecture, *SD-CPS* orchestrates and manages the CPS workflows with a unified control. *SD-CPS* leverages the edge resources in offloading the CPS workload as service workflows. We observe that the execution of CPS as a workflow of microservices can enable resource allocation and migration with higher performance, supporting user policies effectively. Although currently the reach of edge nodes and stability of the networks in mobile CPS environments such as MANETs are limited, they continue to improve with innovation. We anticipate more and more edge nodes, connected by a stable network, will support the CPS execution environments, including the mobile CPS. We see *SD-CPS* as a futuristic approach for executing CPS at the edge as composable microservice workflows. We believe that with the SDN adoption in wide area networks and the increasing reach of edge nodes, approaches such as *SD-CPS* will lead the next generation CPS.

IV Data Services

9 On-Demand Big Data Integration

Reproducible research requires data integrated and shared on-demand across multiple organizations in a bandwidth-efficient manner. Data used in a scientific research study often needs to be shared among researchers for collaboration and reproducibility purposes. However, this process is not efficient. First, sharing data by replicating its contents creates an excessive overhead on bandwidth, storage, and data maintenance. Therefore, data must be shared with minimal data replication. Second, the current data integration approaches are not optimized for repeated scientific experiments beyond organizational boundaries, and hence lead to repeated and manual efforts. In this chapter, we propose an efficient data service approach for on-demand data integration, with the potential for efficient data sharing with human-in-the-loop and minimal data replication.

Big data integration is crucial for numerous application domains, such as reproducible science [285], medical research [201], and transport planning [165], to enable data analysis and information retrieval. Scientific research often requires access to big data from various data sources, often geographically distributed [157]. Scientific data is typically heterogeneous, including binary and textual data, and stored in structured, semi-structured, or unstructured formats. Furthermore, data sources usually support distinct data access interfaces, ranging from database SQL queries to service-based APIs [156]. Effectively and efficiently integrating large volumes of diverse data is challenging. To discover compelling scientific insights from data, it is often required to extract, transform, and load it into an *integrated data repository* (e.g., a data warehouse [74]). This process is typically called ETL [330]. An ETL process makes data accessible through a uniform schema, by constructing an integrated data repository. Thus it supports fast and efficient querying of the scientific research data.

Repetition of the ETL process to obtain the same integrated data must be avoided, even when the collaboration extends beyond the organizational boundaries. Researchers interested in the data resulting from an integration process may belong to one or many organizations. A typical use case among the medical research scientists is to virtually integrate datasets from heterogeneous distributed data sources and share the results among the collaborators. The sharing of the integrated data is often a manual procedure, oblivious to the ETL process. Such data sharing is inefficient and may lead to the existence and maintenance of duplicate data. A distributed ETL process to support sharing of the integrated data, with minimal repeated data integration and loading and minimal bandwidth overhead, is still lacking.

Motivation: Given the above premises, we aim at addressing the following research questions in this chapter:

(RQ_1) Can we increase the speed of the bootstrapping process in ETL by selectively accessing,

integrating, and loading metadata?

(RQ_2) Can we achieve faster execution time for repetitive scientific research queries by storing the previously integrated and loaded data in an integrated data repository?

(RQ_3) Can we incorporate the human knowledge into an ETL framework to selectively and incrementally integrate and load only the relevant subsets of metadata or data, from web data sources?

(RQ_4) Can the relevant subsets of data and metadata loaded by a research scientist for a specific experiment be shared efficiently for reproducibility purposes, minimizing data replication across multiple organizations and avoiding the repetition of the ETL process?

Contributions: The goal of this chapter is to answer the identified research questions, focusing on medical research as motivating real-life domain. The main contributions of this chapter are:

1. A novel hybrid ETL approach for accessing and integrating data and metadata from heterogeneous data sources, and loading the resulting data into a scalable integrated data repository. (RQ_1 and RQ_2)

2. The incorporation of human knowledge into a hybrid ETL process to selectively integrate and load subsets of data and metadata on-demand. (RQ_3)

3. A data sharing mechanism that enables to virtually share the relevant datasets efficiently through "pointers" to data, instead of repeatedly loading and replicating the actual data and metadata. (RQ_4)

We implemented *Óbidos*[1], an on-demand big data integration platform for scientific research. In this chapter, we elaborate in detail, how *Óbidos* supports a distributed hybrid ETL approach enhanced with human-in-the-loop for efficient data sharing. We deployed and performed an experimental evaluation of *Óbidos* for medical research data. In particular: (i) we compared *Óbidos* data loading and query execution times with eager [330] and lazy [183] ETL, and (ii) we evaluated the efficiency of *Óbidos* regarding the amount of data replication and bandwidth required in data sharing. The results obtained indicate that *Óbidos* performs better than or equal to both eager and lazy ETL approaches. We further observed that *Óbidos* data sharing feature avoids data replication and repeated ETL efforts. Thus, we present *Óbidos* data integration and data sharing as classic big data use cases where a data service approach can enhance the interoperability of diverse and distributed real-world big data applications.

This chapter is composed of the contents of the publication: [J3, C4, W2, W4].

[1] *Óbidos* is a medieval fortified town that has been patronized by various Portuguese queens. It is known for its sweet wine, served in a chocolate cup.

9.1 Motivation

Research has proposed several enhancements to data integration such as virtual data integration [195] and human-in-the-loop data integration [206]. Similarly, proposals such as distributed data sharing [367] aim at improving the efficiency of data sharing. LigDB [240] provides a query-based integration without storing any data, and efficiently handles unstructured data with no schema. However, these approaches do not focus on big data integration for scientific research that has its limitations and constraints as well as requirements such as reproducibility.

Traditionally, ETL has been an eager process, loading the entire content of the data sources into an integrated data repository as a first step. However, *eager ETL* is often unsuitable for handling scientific data. First, the bootstrapping process of eager data integration and loading takes too long. This time waste is unnecessary for scientific research [67] that often requires only a subset of data. Second, entirely integrating and loading the contents of data sources can be challenging due to the substantial resource demands with high loading time and bandwidth. Furthermore, eager ETL also demands large storage due to the typical amount of data to integrate. Third, scientific data sources are often accessible only to authorized people. Loading the entire contents of data sources into an integrated data repository may enable to bypass the data authorization permissions established for data sources. Users would then be able to access data from the integrated data repository, thus increasing the probability of data access violation.

Lazy ETL aims at mitigating the limitations of eager ETL, by integrating and loading the data only when necessary. It avoids loading the entire contents of data sources into an integrated data repository as the initial step, by eagerly integrating and loading only the metadata. A data source is composed of several data entries. For binary data such as the medical images stored in DICOM (Digital Imaging and Communications in Medicine) [241] and seismological data stored in SEED (The Standard for the Exchange of Earthquake Data) [6] standard formats, there is typically a piece of textual metadata (containing identifying information) attached to each data entry in the file header. Metadata is often sufficient for the initial scientific research demands. The metadata can be leveraged in the early stages of the research, while image processing can be performed at a later phase, only for images selected as relevant (from the metadata). A lazy ETL [183] framework for seismological research demonstrates how metadata can be efficiently used for study-specific queries without actually constructing an entire data warehouse beforehand, by using files in SEED format. Integrating and loading the metadata, in these cases, is faster than loading the entire data entry, due to the substantially smaller size of the metadata. Therefore, lazy ETL usually bootstraps faster than eager ETL.

Lazy ETL has its shortcomings in reproducible scientific research, especially when the metadata is significantly large or when the experiments consist of repetitive queries. First, persisting the previously processed data entries into the integrated data repository would make recurring scientific research experiments faster. While eager ETL loads the data entirely into an integrated data repository, current lazy ETL approaches are not able to persistently store data required for previous queries. Therefore, recurring scientific queries execute slower under lazy ETL than under eager ETL. The gain obtained by faster data integration and loading in lazy ETL is lost

when executing recurring queries because they cannot use stored results from previous queries. Second, scientific research often requires integrating large amounts of heterogeneous data from several web data sources [107]. Consequently, even an eager metadata-only ETL process (as prescribed by lazy ETL) can be challenging in scientific research. Third, metadata of some data sources tend to be as large as or larger than the data entries themselves. For example, Scality RING petascale object storage [297] consists of metadata up to 10 times larger than the data entries, supporting content-based searches through its metadata (designed for indexing). A typical lazy ETL process may fail to outperform an eager ETL process in bootstrapping in the presence of such data sources, due to the large size of metadata.

Currently, in some domains, ETL is performed on-demand by a user [192]. The user is involved in the ETL process by incrementally integrating and loading subsets of data or metadata that are relevant to a given research question. The user is often aware of the details about data source access and data location. So, the researcher may be able to directly access the required data without accessing and querying the corresponding metadata. This expert knowledge should be incorporated into the ETL process. This type of user involvement is called *human-in-the-loop ETL*. It often consists of two parts. First, the user manually searches and downloads the datasets from the web data sources. Then, she integrates and stores the result in an integrated data repository. By narrowing down the search space to a smaller subset of relevant data sources, human-in-the-loop ETL shortens the data integration and loading time. However, existing ETL frameworks do not support the automatic incorporation of human in the process. Therefore, currently, human-in-the-loop ETL process remains a cumbersome manual and repetitive task.

Reproducible science requires a hybrid ETL that enables a selective loading of metadata and data and storing the integrated data in an integrated data repository for later access. Since the number of web data sources, as well as the amount of data and metadata, tend to increase, the storage requirements for the integrated data repository must be adaptable. In particular, scalable storage is essential to accommodate data and metadata selectively accessed and incrementally integrated and loaded by the researcher. However, the current ETL approaches do not support such a selective ETL process into a scalable integrated data repository. We propose a service-based on-demand data integration as well as persisting and sharing integrated data across multiple research organizations, to address these identified shortcomings in data integration for reproducible scientific research.

9.2 *Óbidos*: An On-Demand Big Data Integration Platform

The *Óbidos* platform is instantiated for each organization. Users from the organization can access, integrate, and load data into the integrated data repository of the corresponding *Óbidos* instance. Furthermore, they can share datasets stored in the integrated data repository with other users from the same or different organizations. Section 9.2.1 presents the *Óbidos* hybrid ETL approach and the underlying architecture. Section 9.2.2 explains how *Óbidos* incorporates human knowledge in the ETL process to selectively and incrementally integrate and load subsets of data and metadata. Section 9.2.3 details *Óbidos* efficient data sharing mechanism beyond organization boundaries to minimize data replication and repeated ETL efforts.

9.2.1 Hybrid ETL Process

***Óbidos* Architecture:** Figure 9.1 depicts the architecture of an *Óbidos* instance. From bottom to top, *Óbidos* consists of i) a scalable **Integrated Data Repository**, ii) a **Data Management Layer** with constructs for fast data integration and loading, and iii) a **Query Rewriter** with constructs for efficient and unified access to the data in the integrated data repository and the data sources.

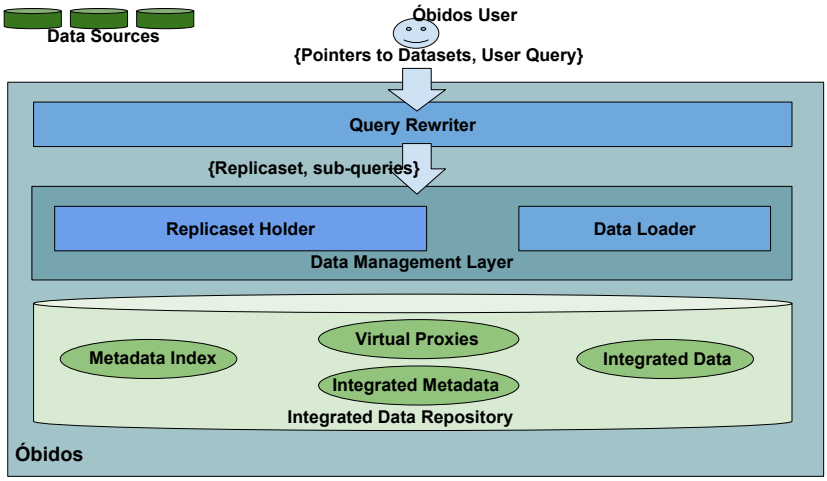


Figure 9.1: *Óbidos* Architecture

The **Integrated Data Repository** incrementally stores the data and metadata integrated and loaded by users. It consists of i) structured and unstructured data (including binary data) as **integrated data** and ii) the corresponding metadata as **integrated metadata**. Furthermore, the metadata stored in the integrated data repository needs to be indexed for efficient query execution over the binary data. We call this index of the integrated data repository, the **Metadata Index**. The Metadata Index functions as an internal index that is built over the integrated data and metadata in the *Óbidos* instance. *Óbidos* further stores the incomplete metadata entries, the metadata that is being loaded, as **virtual proxies**. The virtual proxies are stored as *future* or a placeholder for the complete metadata in the integrated data repository. The complete metadata will replace the virtual proxies once the entire metadata is loaded.

The **Data Management Layer** consists of data structures to manage the data in the integrated data repository and components to access, integrate, and load from the data sources. It stores its data structures in memory in a cluster of machines, aiming to offer fast access to the integrated data while not compromising fault-tolerance. A *virtual replica* is a pointer to a dataset from a distinct data source. A *replicaset* is an *Óbidos* data structure that is composed of several virtual replicas. Thus, each replicaset points to the distributed and diverse datasets relevant to a scientific research study. Furthermore, the replicasets are identified by timestamps. Therefore, the integrated data repository can be periodically updated with the changes or updates to the datasets in the data sources pointed by the replicasets.

The **Replicaset Holder** is the core module of the Data Management Layer. It identifies

each replicaset by a globally unique identifier known as replicasetID. The *Replicaset Holder* stores the replicasets in memory in a data structure that maps each item of integrated and loaded metadata into the corresponding replicasets. Thus, it indicates which of the datasets have already been loaded into the integrated data repository, either as integrated metadata and data or as virtual proxies. Moreover, it enables sharing the replicasets among users freely to make the datasets relevant to the scientific research available to other participants. Therefore, it serves as a component that prevents repetitive attempts to access, integrate, and load the same datasets. The **Data Loader** selectively loads metadata and data from the data sources. The location and the access mechanisms to the data sources are provided by the user and are stored in memory by the Data Loader.

Finally, the **Query Rewriter** enables uniform access to data sources as well as to the integrated data repository. It accepts as input a user query and pointers to the relevant datasets. Then, it converts the pointers to the datasets into replicasets. It also translates user queries into sub-queries that access either the data sources or the integrated data repository. If the data required to answer the user query is not present in the integrated data repository, it invokes the Data Loader to integrate and load the datasets to answer the user query as well as the virtual proxies corresponding to the replicaset.

***Óbidos* Incremental Data Integration and Loading:** *Óbidos* accesses data and metadata from the data sources, and incrementally integrates and loads the results of the user queries into an integrated data repository. The integrated data repository persists previous query answers as well as the data and metadata integrated and loaded for answering previous queries. Therefore, queries can be regarded as virtual datasets that can be re-accessed or shared (akin to the materialized view in traditional RDBMS).

Óbidos enables to incrementally integrate and load metadata to mitigate the challenges in loading the metadata entirely or eagerly. When incrementally loading the metadata, *Óbidos* replaces the counterparts of metadata that has not been loaded yet with a virtual proxy. The use of virtual proxies minimizes the volume of metadata integrated and loaded. *Óbidos* stores the virtual proxies in the integrated data repository in addition to the integrated data and the corresponding metadata. If only a fraction of metadata is relevant for a search query, it is sufficient to load only that fraction. Therefore, *Óbidos* selectively loads metadata as virtual proxies. The virtual proxies are later replaced by the complete metadata as the metadata is accessed and integrated. Thus, virtual proxies refer to the metadata of a dataset larger than that is integrated and loaded to the integrated data repository.

Often a virtual replica may be present in the Replicaset Holder, without having the exact data for the user query. This usually means, previously at least one different user query has been executed on the same virtual replicas. Therefore, while the virtual proxies of the replicaset are present, the exact data for the user query may not be present in the integrated data repository. With time, as more and more data are selectively integrated and loaded, the integrated data repository will contain the necessary data for the subsequent scientific research queries.

9.2.2 Human-in-the-Loop ETL Process

Óbidos supports a human-in-the-loop ETL process. By 'human-in-the-loop,' here we mean to incorporate the human knowledge that corresponds to the user-defined replicasets and queries to selectively access and integrate data from the data sources and incrementally loading the integrated data repository. A user identifies certain datasets as relevant to her scientific research, and these datasets are the ones against which the user query will be executed. She defines a replicaset by including pointers to these datasets as virtual replicas. The replicaset and a specific user query determine the data to be integrated and loaded by each selective data integration and loading process. This avoids the need to look for the desired data across data sources exhaustively.

Óbidos selective load process is initiated every time a user issues a query. First, *Óbidos* iteratively checks for the existence of the data necessary to answer the query in the instance. It queries the Replicaset Holder for each of the virtual replicas and then executes the user query on the integrated data repository. If the data is not available in the instance, it is integrated and loaded from the data sources. The results of the user queries are persistently stored into the integrated data repository. Furthermore, rather than just querying and loading only the answers of the user query, *Óbidos* selectively loads the metadata pointed by the replicaset. This ensures that the integrated data repository can be incrementally loaded with data, rather than merely storing discrete, incoherent, or independent sets of data. Figure 9.2 shows an *Óbidos* user defining a replicaset along with a user query to be executed on multiple data sources. The replicaset narrows down the search space from the entire data sources to specific datasets to answer the user query. She ensures with the knowledge of the data sources, the data required to answer her user query is part of the datasets pointed by her replicaset.

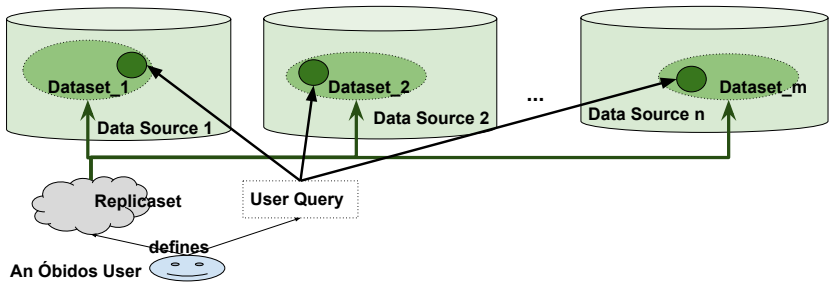


Figure 9.2: Narrowing down the search space with user-defined replicasets

The data integrated and loaded into the integrated data repository of an *Óbidos* instance should be available to be accessed later for scientific research. For example, when a user receives a replicaset from another user from the same or another organization, she may access her organization's instance to check for already loaded data. Algorithm 10 illustrates how a user initiates the selective and incremental data integration and loading process of *Óbidos*.

The algorithm starts by initializing a temporary variable *toLoad*, as a set, with the copy of the replicaset (line 2). *toLoad* tracks the virtual replicas belonging to the replicaset that have not yet been loaded from the data sources. Then, the algorithm proceeds to check the existence

Algorithm 10 *Óbidos* Human-in-the-Loop Incremental ETL

```
1: procedure SELECTIVELOAD(replicaset, userQuery)
2:     toLoad ← replicaset
3:     for all (virtualReplica ∈ replicaset) do
4:         wasLoadedBefore ← replicasetHolder.get(virtualReplica)
5:         if ¬(wasLoadedBefore) then
6:             loadData(virtualReplica, userQuery)
7:             replicasetHolder ← replicasetHolder ∪ {virtualReplica}
8:             toLoad ← toload \ {virtualReplica}
9:         end if
10:    end for
11:    if ((toLoad ≠ ∅) ∧(integratedDataRepository.query(userQuery) = ∅)) then
12:        for all (virtualReplica ∈ toLoad) do
13:            loadData(virtualReplica, userQuery)
14:        end for
15:    end if
16: end procedure
```

of the data pointed by each virtual replica in the instance (line 3). First, it queries the Replicaset Holder to check whether datasets pointed by the virtual replica have already been loaded by a previous query (line 4). If no dataset has yet been loaded for the virtual replica (line 5), the data relevant for the virtual replica and the user query is loaded from the data sources incrementally, invoking the *loadData* procedure (line 6). The Replicaset Holder matches the replicasets to the respective data and metadata integrated and loaded in the integrated data repository, by the selective load process. Therefore, in line 7, the virtual replica is added to the Replicaset Holder. Now since the dataset pointed by the virtual replica has already been loaded, the virtual replica is removed from *toLoad* (line 8).

The first loop (lines 3 - 10) checks whether the data, metadata, or virtual proxies relevant for one or more of the virtual replicas exist in the integrated data repository. It loads the data only when neither corresponding data and metadata nor virtual proxies are found for a given virtual replica. Therefore, a non-empty set of *toLoad* at the end of the loop indicates that at least a few virtual replicas were not loaded during this iteration. In that case, the algorithm proceeds to check whether the data and metadata necessary to answer the current user query are completely available in the integrated data repository (line 11). The user query will return a null value if the complete metadata and data necessary to answer the query are not present in the integrated data repository. Consequently, the *loadData* procedure is executed for all the virtual replicas in the *toLoad* set (lines 12 - 14).

The loadData Procedure: The *loadData* procedure is the core of the *Óbidos* human-in-the-loop incremental ETL approach. It accepts a replicaset and a user query as input arguments. First, the data sources are accessed, and the datasets identified by the replicasets are selectively loaded as virtual proxies, without loading the entire metadata. Then, the user query is executed against the data sources. The relevant metadata representing the results of a user query is integrated and loaded to the integrated data repository. If the user query also indicates access to the binary data, the respective binary data (usually a subset of data corresponding to the

metadata already loaded by the query) is also loaded to the integrated data repository. The *loadData* procedure selectively loads the metadata corresponding to the replicaset as virtual proxies. If previously a different user query was issued with the same virtual replica, the virtual proxies corresponding to the virtual replica would be present while the exact data and metadata to answer the current user query would be absent in the integrated data repository.

9.2.3 Data Sharing Process

An *Óbidos* instance is deployed in each organization. Each *Óbidos* instance is used by: i) users from the organization and ii) users from other organizations and external users who have limited access to the *Óbidos* instance. Users can share the datasets among them by sharing the replicasets or their respective replicasetIDs. Therefore, there is no need to replicate the actual data of the data sources nor the integrated data repository of an *Óbidos* instance.

Datasets can be shared by as a replicaset or the respective replicasetID. Replicasets are small in size. However, they grow with the number of data sources and diversity of data. ReplicasetID is smaller in size compared to the replicaset and is of a fixed size. Therefore, they are shared by default. A user outside the organization can access the data already loaded in an *Óbidos* instance using the replicasetID. Moreover, users can share the replicasets with other organizations, without letting them access the data in their integrated data repository. The receiver organization can then integrate and load the datasets pointed by the replicaset, into its own *Óbidos* instance. Figure 9.3 illustrates the process of data sharing between users *User_s1* and *User_r1* from two different organizations (called sender and receiver). The sender organization and the receiver organization can also represent the same organization if both users belong to the same organization.

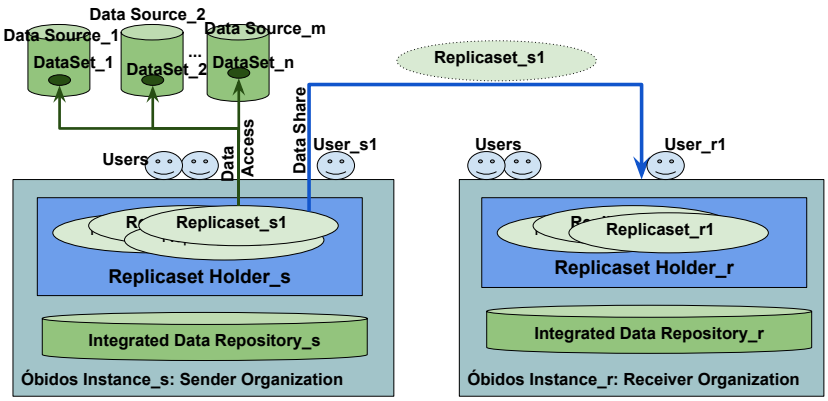


Figure 9.3: Data Sharing with *Óbidos*

Algorithm 11 describes the data sharing procedure executed by the *Óbidos* instance of the receiver organization. It takes as input: a replicaset (or its replicasetID) received from another user, the identification of users that created/sent and received the replicaset, and an optional object known as *accessSender* (line 1). A null value for the *accessSender* object indicates that the shared datasets should be accessed from the data sources. A non-null value indicates that

the datasets need to be accessed directly from the sender instance. The *accessSender* object consists of relevant access mechanisms such as the access key to the integrated data repository of the sender instance.

Algorithm 11 Data Sharing via a Replicaset

```
1: procedure SHAREREPLICASET(replicaset, sender, receiver, accessSender)
2:     if (replicaset.isURI())
3:         replicaset ← sender.get(replicaset) then
4:     end if
5:     if (accessSender ≠ ∅)
6:         sender.access(replicaset) then
7:     else
8:         receiver.selectiveLoad(replicaset, ∅)
9:     end if
10: end procedure
```

If a replicasetID is received, the replicaset is retrieved from the sender instance first (lines 2 - 4). Since the replicaset was initially created by a user of the sender organization, the datasets or the virtual proxies pointed by the replicaset would be present in the sender organization. Therefore, if the *accessSender* is set to a non-null value (line 5), the datasets pointed by the replicaset are accessed directly from the sender instance, by the receiver organization (line 6). Otherwise, the *shareReplicaset* procedure selectively loads the datasets pointed by the replicaset into the receiver instance, from the data sources (line 8). As there is no user query defined in a shared replicaset, the *selectiveLoad* procedure is invoked with a null value in place of the user query.

9.3 Implementation

We built *Óbidos* with several data services, including data cleaning, loading, and sharing. *Óbidos* consists of data structures, APIs, and software components that enable chaining of these data services for its execution.

9.3.1 Data Structures

The Replicaset Holder stores the replicasets in a minimal tree-like data structure, to offer efficient search and indexing capabilities. Figure 9.4 illustrates the data structures of the Replicaset Holder and the data representation of *Óbidos*.

The Replicaset Holder consists of a few instances of a *multi-map* data structure, storing a list of values against a given key. As each user composes several replicasets, the **userMap** stores a list of replicasets against the identification of the users (userID) that created them. Each value stored in the userMap represents a replicaset of a user and further points to a **replicasetMap**. The **replicasetMap** includes the virtual replicas belonging to each replicaset, and whether the replicasets have already been integrated and loaded to the integrated data repository. A **replicasetID** is a globally unique random value generated by appending a random string generated

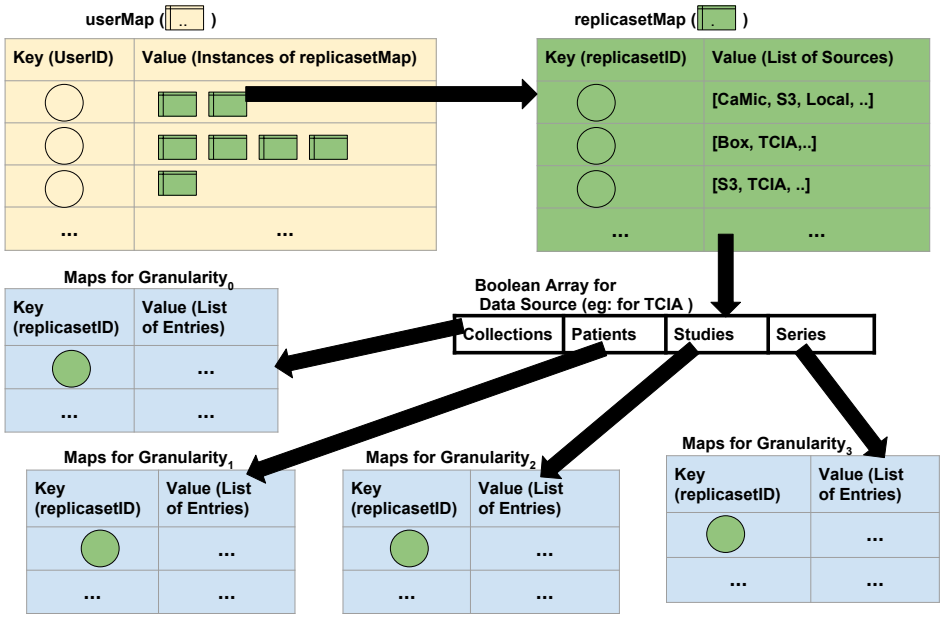


Figure 9.4: Data Structures of the Replicaset Holder

via Java's random UUID generator (using *UUID.randomUUID().getLeastSignificantBits()*) to the URL of the *Óbidos* deployment. The *replicasetMap* employs the replicasetID as its key and the list of data source names contributing data to a replicaset as the value. Thus, each *replicasetMap* stores the relevant data sources for each of the replicaset.

Replicasets include pointers to datasets from various data sources as virtual replicas. Multiple maps internally represent each data source belonging to a replicaset. Figure 9.4 represents an illustrative use case for hierarchical data storage. It considers cancer images of DICOM format stored in data repositories such as TCIA, S3 buckets, directories in Box.com, and a local folder/file hierarchy. Metadata includes identifying information on the data, including how it fits in the overall data storage or larger granularity. Therefore, the Replicaset Holder exploits the metadata to efficiently index and store the data.

The Replicaset Holder uses a boolean array A_b of length n to represent the replicasets of hierarchical data storage formats in a bit-map like structure, where n refers to the number of granularity levels. Each element $A_b[i]$ of the array represents the existence of a non-null entry in the map of i^{th} granularity. Thus, the boolean flags in A_b indicate the presence (or lack thereof) of the dataset in a particular granularity of a replicaset. If an entire level of granularity is included in the replicaset by the user, the relevant flag is set to true. The Replicaset Holder further consists of n maps, each representing one of the granularity levels in the data source. For the DICOM images, $n = 4$. Thus, 4 maps represent its 4 granularity levels - collections, patients, studies, and series, with an array of length 4 pointing to each map. Each of these maps stores replicasetIDs as its keys and lists of the entries in the specific granularity (shown as $Granularity_0$, $Granularity_1$, ...) as its values. The hierarchical data representation enables incremental loading and virtual proxies through its indexed data structure.

9.3.2 Service-based APIs

The *Óbidos* APIs are designed as CRUD (Create, Retrieve, Update, and Delete) RESTful services on replicasets. *Óbidos* offers a *data sharing* API to share scientific research datasets, by sharing the replicasets. Replicasets can also be shared outside *Óbidos*, through other communication media such as email. The data sharing method is typically one-to-one, meaning that a user shares data with another user in the same or different organization. However, it can also be listed for the public to be freely accessed.

The user accesses, queries, integrates, and loads the relevant data from the data sources by invoking the ***create*** *replicaset* procedure. This procedure creates a replicaset and initiates the selective data integration and loading process. When ***retrieve*** *replicaset* is invoked, the data corresponding to the given replicaset is retrieved from the integrated data repository. Furthermore, *Óbidos* checks for updates from the data sources pointed by the replicaset, if the data corresponding to the replicaset has already been integrated and loaded. Metadata of the replicaset is compared against that of the data sources for any corruption or local changes. The user deletes existing replicasets by invoking the ***delete*** *replicaset*. When a replicaset is deleted, the Replicaset Holder is updated immediately to avoid loading updates to the deleted replicasets. The user updates an existing replicaset to increase, decrease, or alter its scope, by invoking the ***update*** *replicaset*. Thus, the update process may, in turn, invoke parts of create and delete processes, as new data may be loaded while existing parts of data may be removed.

The Replicaset Holder associates each dataset to a user, through its data structures such as the userMap. While each user has her own virtually isolated space in memory, the integrated data repository consists of a data storage shared among all the users of the organization. Hence, before deleting a data entry from the integrated data repository, the data should be confirmed to be an 'orphan' with no replicasets referring to them from any of the users. Deleting data from the integrated data repository is initiated by a background system task, rather than the users. When the storage is abundantly available in a cluster, *Óbidos* advocates keeping orphan data in the integrated data repository rather than immediately initiating the cleanup process, and repeating it too frequently.

9.3.3 *Óbidos* Software Components

We separate the *Óbidos* architecture and interfaces from its implementation, avoiding tight coupling of the framework to ensure its reusability. The *Óbidos* architecture consisting of its data structures and interfaces is generic and can be exploited for the integration of data from data sources other than the medical research data. We utilize several open source frameworks as primary dependencies in our *Óbidos* prototype. *Óbidos* uses HDFS as the core of the integrated data repository, due to its scalability and support for storing unstructured and semi-structured, binary and textual data. *Óbidos* executes on a cluster of Infinispan [225] IMDG. Consequently, the Data Management Layer stores its data structures in an Infinispan cluster. The metadata of the binary data in HDFS is stored in tables hosted in Apache Hive [324] metastore based on HDFS. The Hive tables consisting of the metadata are indexed with the Metadata Index for users to query and locate the data from the integrated data repository efficiently.

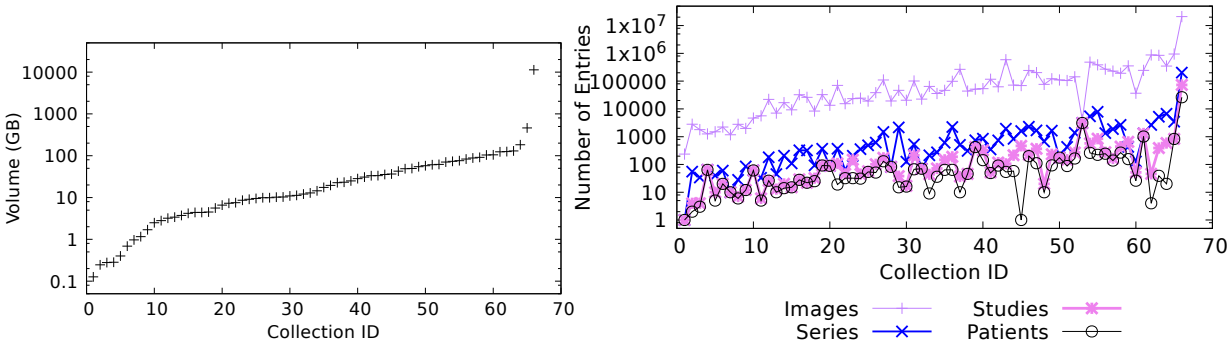
Óbidos supports SQL queries on unstructured data stored in HDFS, through the Metadata Index stored in Hive. Apache Drill [154] enables SQL queries on structured, semi-structured, and unstructured data. Therefore, the Query Rewriter unifies and accesses the storages seamlessly by leveraging Apache Drill. This approach allows efficient queries to the data, partially or wholly loaded into the integrated data repository. API Umbrella is deployed as the default API gateway to manage and provide users with access to the APIs. *Óbidos* incorporates authorization to its shared data from the integrated data repository through the use of API keys, leveraging the API gateway. A user can only access the data shared with her, and only with the API key that belongs to her. Thus, *Óbidos* provides access-controlled, unified, and scalable access to the data in the integrated data repository and the data sources.

Oracle Java 1.8 is used as the programming language in developing *Óbidos*. Apache Velocity 1.7 [138] is leveraged to generate the application templates of the *Óbidos* web interface. Hadoop 2.7.2 stores the integrated data along with its corresponding metadata and virtual proxies, whereas the Metadata Index is stored in Hive 1.2.0. Hive-jdbc package writes the Metadata Index into the Hive metastore through its JDBC (Java Database Connectivity) bindings to Hive. SparkJava 2.5 [312] compact Java-based web framework is leveraged to expose the *Óbidos* APIs as RESTful services. We deployed *Óbidos* as a web application on Embedded Tomcat 7.0.34. Infinispan 8.2.2 is used as the IMDG where its distributed streams support distributed execution of the hybrid ETL processes across the *Óbidos* clustered deployment. The data structures of the Data Management Layer are represented by instances of the Infinispan Cache class, which is a Java implementation of distributed HashMap. Drill 1.7.0 is exploited for the SQL queries on the integrated data repository. It uses the JDBC API provided by the drill-jdbc module to connect with the Query Rewriter programmatically.

9.4 Evaluation

We benchmarked *Óbidos* against the implementations of eager ETL and lazy ETL, using microbenchmarks derived from medical research queries on cancer imaging and clinical data. We deployed an *Óbidos* prototype to integrate medical data from various heterogeneous data sources including TCIA [77], DICOM imaging data hosted in Amazon S3 buckets, medical images accessed through caMicroscope [65], clinical and imaging data stored in local data sources including relational and NoSQL databases, and file system with files and directories along with CSV files as metadata. Our evaluations primarily used DICOM images stored as collections of various volume as shown in Figure 9.5, sorted according to their total volume. The data consists of large-scale binary images (in the scale of a few thousand GB, up to 10 000 GB) along with a smaller scale textual metadata (in the range of MBs).

The evaluated collections consist of diverse entries. Figure 9.5a shows the total volume of the collections. Figure 9.5b illustrates the number of patients, studies, series, and images in each of the collection. Each collection consists of multiple patients; each patient has one or more studies; each study has one or more series; each series has numerous images. We defined replicasets at these different levels of granularity. The varying pattern of Figure 9.5b, when compared against



(a) Total Volume of the Collections

(b) Various Entries in the Collections

Figure 9.5: Evaluated DICOM Imaging Collections (Sorted by Total Volume)

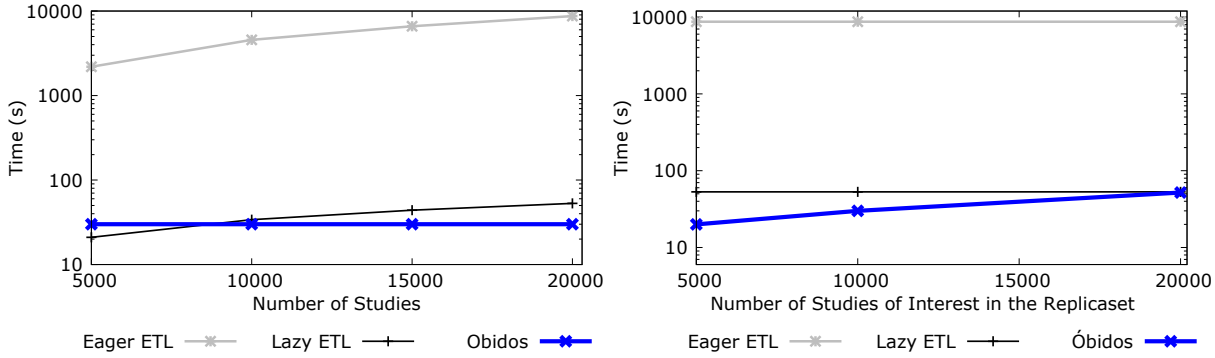
that of Figure 9.5a, shows that the total volume of a collection does not necessarily reflect the number of entries in it.

9.4.1 Performance of Integrating and Loading Data

We benchmarked *Óbidos* for its data integration and loading time against that of lazy ETL and eager ETL approaches, for a varying number of studies. Since the scientific research data sources span the globe, the network bandwidth will impact the performance of loading data directly from them. To avoid this influence, first, we replicated the data sources such as TCIA to data sources hosted on the local servers.

Figure 9.6a shows the data integration and loading time from different total volumes of data sources for the same replicasets of the user. Since lazy ETL and eager ETL approaches query the entire data sources, the increasing volume of data in the data sources leads to a larger time to integrate and load them. Eager ETL always took more time as it has to integrate and load the entire metadata and data. Since lazy ETL loads only the metadata eagerly, it loads faster than eager ETL. *Óbidos* selectively loads the metadata of only the data corresponding to the replicaset. The loading time remained constant, regardless of the growth of the increasing total volume of data in the data sources, as the replicaset and the user query remained the same. Therefore, the human-in-the-loop contributed positively to the integration and loading performance of *Óbidos* by narrowing down the search space from the data sources.

Óbidos consumed more time for the data integration and loading compared to the lazy ETL for smaller volumes of data. For small volumes, eagerly loading the entire metadata can be faster than the selective loading by *Óbidos*, as *Óbidos* executes the query on the data source and loads the virtual proxies, creating and updating the constructs such as the Metadata Index and the Replicaset Holder. However, as the total volume of data grows, the data loaded by *Óbidos* remained the lowest compared to both eager ETL and lazy ETL, thus resulting in faster data integration and loading. For repetitive user queries, both eager ETL and *Óbidos* outperformed the lazy ETL due to the availability of the integrated data repository in both eager ETL and *Óbidos*, and the storing of query answers in *Óbidos*.



(a) Change in total volume of data sources (Same user query and same replicaset)

(b) Varying number of studies of interest in the replicaset (same user query and constant total data volume)

Figure 9.6: Data load time

Figure 9.6b depicts the data integration and loading time for the same experiment, but with constant total volume of data sources while increasing the number of studies of interest in the replicaset. Since the total volume remained constant, the lazy ETL and eager ETL had the same data integration and loading time, as they are oblivious to the change in the number of studies of interest. However, the performance of *Óbidos* depends heavily on how the replicasets are defined. Therefore, with the growth of the replicaset, the loading time of *Óbidos* increased. Eventually, the data integration and loading time of *Óbidos* converged with the time taken by the lazy ETL approach, as the replicaset was defined to cover all the studies in the data sources (thus, making it same as eagerly loading the metadata).

Figure 9.7 shows the data integration and loading time from remote sources. The datasets were integrated and loaded directly from the remote data sources (such as TCIA and S3 buckets) through their web service APIs, to evaluate the effects of data downloading and bandwidth consumption associated with it. We changed the total volume of data in the data sources by adding more data to the data sources while keeping the replicaset unchanged. Eager ETL performed poor as binary data had to be downloaded over the network. Lazy ETL too performed slowly for large volumes as it must eagerly load the metadata (which itself grows with scale) over the network. As with the case of Figure 9.6a, Figure 9.7 too illustrates a fixed time for *Óbidos* data integration and loading. As the data was integrated and loaded over the Internet from the data sources, the time taken grew linearly for eager ETL and lazy ETL. However, lazy ETL consumed much lower time compared to the eager ETL. As only the datasets corresponding to the replicaset are accessed, integrated, and loaded, *Óbidos* uses bandwidth conservatively, loading no irrelevant data or metadata.

9.4.2 Performance of Querying the Integrated Data Repository

We then benchmarked *Óbidos* for its efficiency in querying the data and integrated data repository against the eager ETL. Query completion time depends on the number of entries in the queried data rather than the size of the entire integrated data repository. Hence, we

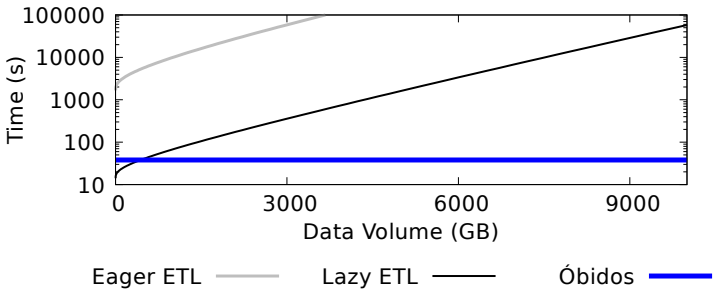


Figure 9.7: Load time from the remote data sources

measured the amount of data by the number of studies that were queried. Figure 9.8 depicts the query completion time of *Óbidos* and eager ETL. *Óbidos* showed a speedup compared to the eager ETL due to its efficient indexing of the binary data in the integrated data repository with Metadata Index and the efficiency of the Data Management Layer in managing the storage and execution. The unstructured data in HDFS was efficiently queried as in a relational database through the distributed query execution of Drill with its SQL support for NoSQL data sources.

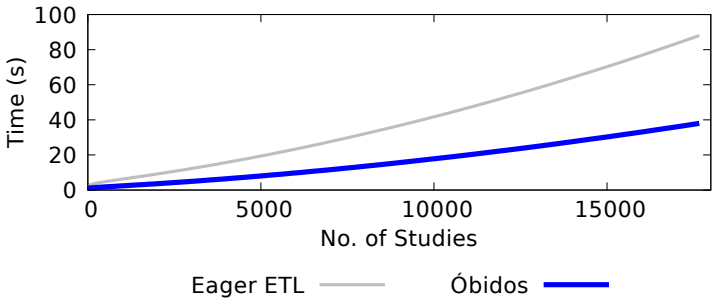
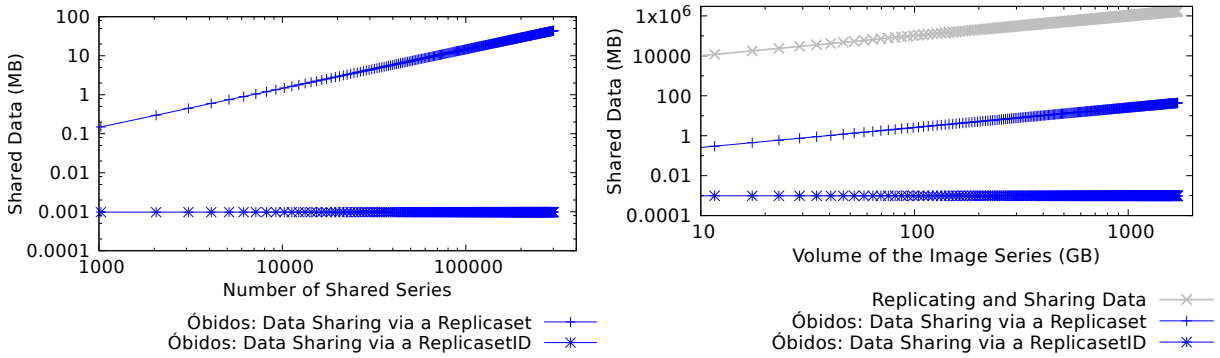


Figure 9.8: Query completion time for the integrated data repository

Óbidos brings the best of both worlds from eager ETL and lazy ETL approaches for scientific research data integration and loading. Typically, lazy ETL approaches do not consist of an integrated data repository. Therefore, we avoid comparing the query performance on the *Óbidos* integrated data repository against the lazy ETL. Eager ETL could outperform *Óbidos* for queries that access data not yet loaded in *Óbidos*, as eager ETL would have constructed an entire data warehouse beforehand. However, with the domain knowledge of the medical data researcher, the relevant datasets are loaded timely, and only those. The time required to construct a complete data warehouse would preclude any benefits of eager loading from being prominent. If data is also not loaded beforehand in eager ETL, it will consume much longer to construct the entire data warehouse before actually starting the processing of the user query. Moreover, loading everything beforehand may be irrelevant, impractical, or even impossible for scientific research studies due to the scale and distribution of the data sources.

9.4.3 Sharing Efficiency of Medical Research Data

Figure 9.9 benchmarks the bandwidth efficiency of the *Óbidos* data sharing approaches against the typical binary data transfers. Various image series of an average uniform size are shared between users inside an *Óbidos* instance and across multiple instances. *Óbidos* facilitates data sharing by sharing either the replicasetID or the replicaset itself. First, we used a varying number of series to compare the *Óbidos* data sharing approaches, as illustrated by Figure 9.9a.



(a) With changing number of shared series

(b) With changing volume of shared images

Figure 9.9: Volume of data shared in *Óbidos* use cases vs. in regular binary data sharing

Since sharing data by its actual content does not directly depend on the number of entries (unlike the *Óbidos* approaches), we then measured the volume of the image series in the case of benchmarking against sharing of data by replicating and sharing the actual content. Figure 9.9b highlights that negligible volume of data was shared in both cases of *Óbidos* as opposed to sharing the actual data. Sharing via the replicasetID outperformed sharing via replicaset in all the cases, as replicasetIDs are relatively smaller and of constant size. Even sharing the replicaset itself was more bandwidth efficient than actually replicating and sharing the data.

Óbidos is inspired by the data integration research and aims to bring the control of data integration to the end user. The use of a unified schema in (VCE) [53] to virtually integrate data is similar to the *Óbidos* approach. However, *Óbidos* offers a complete hybrid ETL approach and supports sharing data with minimal data replication. Various large-scale scientific data repositories such as EUDAT [197] share the motivation of *Óbidos* concerning data integration. Loading the entire data from all the sources is irrelevant for the consumers of EUDAT data, as in *Óbidos*. Hence, choosing and loading certain sets of data is supported by these service-based data access platforms.

Óbidos attempts to address several shortcomings in the current big data integration and sharing approaches with its hybrid ETL designed for reproducible scientific research. First, exploiting the scalable architecture offered by Hadoop and the other big data platforms to create an index to the unstructured integrated data. Second, managing the data in-memory for quicker data manipulations. Third, sharing the results and datasets efficiently with peers. The existing big data integration approaches [221] do not adequately address these fronts, whereas the *Óbidos* approach focuses on them. The incremental integration and loading approach enables *Óbidos* to

load complex metadata faster than the current lazy ETL approaches. The hierarchical structure and metadata format of binary image formats are not limited to DICOM. It is shared by various scientific data formats such as SEED [6]. Thus, we note that while we prototype *Óbidos* for medical research, the approach is also applicable to multiple research and application domains.

9.5 Conclusion

Óbidos is an on-demand data integration system with human-in-the-loop for scientific research. Inter-disciplinary researches require access and integration of datasets spanning multiple data sources on the Internet. In this chapter, we presented a hybrid ETL process driven by users, which selectively integrates and loads the data and metadata from heterogeneous data sources into a scalable integrated data repository. We built *Óbidos* as an implementation of the hybrid ETL for medical research data. *Óbidos* leverages the respective APIs offered by the data sources to integrate and load the data while providing its RESTful APIs for accessing its integrated data repository. We envisioned that various organizations with an *Óbidos* instance would be able to coordinate to construct and share the integrated datasets internally and between one another through its standard interoperable service APIs with minimal data replication. We believe that such approaches will continue to enrich the potential for chaining big data executions as data service workflows that can be dynamically composed and reused by multiple tenants.

10 Interoperable and Network-Aware Big Data Workflows

Interoperability and network-awareness are crucial for distributed big data applications to reduce bandwidth cost as well as communication and coordination overheads. Big data frameworks consist of diverse storage media and processing of a large volume of heterogeneous data from several scientific and enterprise domains [75]. Big data processing requires resources beyond what can be offered by a single server or even a data center, due to the volume, variety, and geo-distribution of the data. Therefore, big data is processed in a distributed and parallel manner [290], either on top of in-memory frameworks such as IMDG [129], on the disk in data stores such as relational databases and NoSQL data sources, or on hybrid architectures consisting of persistent data on disk as well as cached data and computations in-memory [364]. Such an execution leads to a significant amount of data processed and transferred across multiple computing nodes and therefore is usually heavy in its bandwidth demand [140]. Distributed execution frameworks such as IMDGs aim to minimize the performance degradation associated with communication and coordination overheads, by reducing, if not avoiding, unnecessary data movements between the execution and storage nodes [353].

Enterprise big data executions are typically confined to their platforms, and cannot be shared with other frameworks, due to the independent development of the big data platforms and incompatibility across their executions and interfaces [364]. The big data frameworks are designed to work in an environment such as a cluster, data center, or a cloud, that offer consistent network connectivity and topology. However, big data workflows can indeed extend beyond the boundaries of a data center, and continue to do so, more and more in recent times [347]. On the other hand, multiple network providers manage the wide area networks such as the inter-cloud and edge networks. Thus, centralized and unified control is challenging concerning both technologies as well as administration and policies. This state of affairs indeed hinders interoperability and the potential of the big data execution frameworks to efficiently schedule the workload and share the resources beyond a data center or a cloud. Due to these factors, distributing or chaining the execution of big data applications among several servers spanning multi-domain data centers and edge nodes in a wide area network is a significant undertaking.

While research efforts such as volunteer computing [22], namely at the edge, have leveraged resource sharing and workload dissemination across independent and decentralized execution nodes, their use is limited to specific applications. Initially proposed for CPU-bound workloads, volunteer computing and cycle sharing have recently been extended for data-intensive applications [16]. However, volunteer computing approaches significantly lack the potential to distribute data-intensive workloads in a network-aware manner, as the central coordinators in such approaches are developed with minimal control over the execution nodes for less intrusion [215] and easy integration with computing resources of independent participants [322].

Motivation: Given the above premises, we aim at addressing the following research questions in this chapter:

(RQ_1) Can we extend and adopt SDN and web services paradigms as a generic software-defined approach for interoperable and network-aware big data executions?

(RQ_2) Can such a software-defined approach enhance the performance, management, and scheduling of the data service workflows at various scales from data centers to the Internet?

(RQ_3) Can network domains advertise and share their dedicated network links among each other for dynamic data service compositions, through enterprise communication protocols such as MOM [87] protocols?

Contributions: The goal of this chapter is to answer the identified research questions. The main contributions of this chapter are:

1. Software-Defined Data Services (SDDS), a big data execution model, to generalize the big data applications, including storage and processing, as data services, and execute them across multi-domain wide area networks. (RQ_1)
2. A network-aware execution of big data workflows, leveraging the Internet paths as well as the direct links provided by various enterprises, such as the cloud direct connects. (RQ_3)
3. *Mayan-DS*, an SDDS framework that aims to solve the challenges that hinder efficient and interoperable big data executions inside and beyond data center networks. (RQ_2)

In this chapter, we elaborate the design and deployment of *Mayan-DS* from data centers to multi-domain wide area networks. First, *Mayan-DS* defines the big data workflows as data services, adopting the interoperability offered by the web service definitions across heterogeneous big data environments. It enables the execution to be agnostic to the storage media, format, and location, by defining each step of the execution as interoperable data services. Second, it exploits SDN for the performance, scalability, and bandwidth efficiency of the data services execution in the wide area network. By deploying a federated multi-domain controller over a wide area network, the SDDS controller receives a global overview of the data service instances comprising the workflows. By leveraging both the service statistics of the web services engines and the network status from the SDN controller, the *Mayan-DS* controller architecture schedules the data services composing the big data workflows in an interoperable and network-aware manner. Our evaluations on a *Mayan-DS* prototype with microbenchmarks highlight that *Mayan-DS* enhances the scalability and performance of big data executions in data centers and wide area networks.

This chapter is composed of the contents of the publication: [J4, C5, B1].

10.1 An SDDS Model at Internet Scale

Efficient placement of data and service is essential for distributed big data frameworks. A network-aware data service workflow should distribute the data objects $\{i, j\}$ belonging to data sets of interest D and execution across the servers, minimizing $l_{i,j}$ (the distance between the related data objects) as well as $l_{n,j}$ (the distance between the execution server n and the relevant data object j). These distances can be determined by a utility function, giving weights to the network properties such as bandwidth, throughput, and latency, rather than a mere physical distance between the service endpoints. However, in the simplistic form, we can define the distances to represent end-to-end latency that can be measured by a continuous ping. While geographical distance plays a role in latency, a direct dedicated link between two geographically distributed servers can offer minimal latency compared to two servers connected via the public Internet despite the geographical distance.

An SDDS framework should minimize the total area $A(n, D)$, referring to the total distance between the execution server and the related data objects from D, as depicted by Equation 10.1. The spread of the data set D is represented by $|D|$, a subset of among all the servers of the SDDS framework (covering an area of ξ).

$$A(n, D) = \underset{\forall i,j \in D}{\text{minimize}} \left(\int\limits_{|D| \subseteq \xi} \left(l_{i,j} + l_{n,j} \right) d\xi \right) \tag{10.1}$$

Equation 10.2 illustrates a simplified notion of the network distance. *Mayan-DS* gives a distance value of 0 to placements inside the same server, and positive values of α, β, and γ to placements in different servers of the same rack, servers of different racks, and servers of different data centers respectively. Cloud providers own several servers spanning data centers. Some servers are in the same data center (D_o). Some are in the same availability zone, i.e., data centers (D_o and D_d) connected by high-bandwidth low-latency network links. Each network region has multiple availability zones (a_o and a_d). Network traffic between regions (r_o and r_d) incur a large latency that depends on the number of hops between the regions and the nature of the backbone network that connects the regions. Moreover, if the different regions are connected by a low-latency dedicated link (distance value of $\gamma(r_o, r_d)_D$), they perform better than being connected via the public Internet (distance value of $\gamma(r_o, r_d)_I$).

$$\left(\gamma : \gamma(r_o, r_d)_I > \gamma(r_o, r_d)_D > \gamma(a_o, a_d) > \gamma(D_o, D_d) \right) > \beta > \alpha > 0. \tag{10.2}$$

Mayan-DS seeks optimal service placement at Internet scale. We define an Internet-based path $P_{(o,d)}$ between the services s_o and s_d, by $s_o \cdot s_d$. *Mayan-DS* iteratively expands a path into sub-paths consisting of direct dedicated paths and Internet paths. It denotes a low-latency dedicated link between any service s_i and another Internet service $s_{\bar{i}}$ by $\overrightarrow{s_i \cdot s_{\bar{i}}}$. The path $P_{(o,d)}$ can be replaced with an alternative path belonging to a set of paths $P'_{(o,d)}$ defined by Equation 10.3.

$$P'_{(o,d)} = s_o \cdot s_{\overline{o}} + \overrightarrow{s_{\overline{o}} \cdot s_{\overline{d}}} + s_{\overline{d}} \cdot s_d \tag{10.3}$$

Equation 10.4 denotes that if the server $s_{\overline{i}}$ is the same as the server of s_i, the value represented by the path $s_i \cdot s_{\overline{i}}$ is reduced to 0.

$$\forall s_i \in P'_{(o,d)} : s_i \equiv s_{\overline{i}} \implies s_i \cdot s_{\overline{i}} = \vec{0} \tag{10.4}$$

Thus, we observe $P'_{(o,d)}$ as a generic form of an execution path of the data service workflow, concerning two service endpoints, as illustrated by Equation 10.5.

$$P_{(o,d)} \subseteq P'_{(o,d)} \tag{10.5}$$

Consequently, the execution path of a larger service composition workflow, consisting of more than two services, can be expanded as shown by Equation 10.6.

$$W = s_o \circ s_{i_1} \circ \ldots \circ s_{i_n} \circ s_d \implies P'_W = P'_{(o,i_1)} + P'_{(i_1,i_2)} + \ldots + P'_{(i_n,d)} \tag{10.6}$$

Mayan-DS consists of an initialization procedure and a data scheduling procedure. *Mayan-DS* initialization procedure identifies the potential workflow execution paths as a set $P'_{(o,d)}$ as denoted by Equation 10.5. The *Mayan-DS* data service scheduling procedure leverages the best-performing option among the potential paths identified by the Equation 10.6.

10.2 Solution Architecture

The *Mayan-DS* framework consists of a federated deployment of SDN controllers that spans multiple network domains. Typically, a domain represents a data center or a network managed by a single provider. The federated deployment denotes a set of servers and topology of SDN switches that are controlled by an SDN controller in each domain. The controllers communicate with the controllers of the other domains through MOM messages, to propagate status updates on the network and bandwidth health statistics. The messages are limited to the ones relevant for each domain, based on the subscriptions of each controller. Thus, each controller achieves a limited access to the network topologies beyond its domain.

Figure 10.1 illustrates a sample deployment of *Mayan-DS* in a data center along with inter-domain communication between the controllers. Users invoke the data services individually, or as part of a larger workflow. The data services are hosted on the **Distributed Execution Frameworks** to support dynamic execution of the service requests on the instance with the best available resources. The controller is hosted on a server, while the other servers in the domain host the distributed execution frameworks such as an IMDG. These IMDG instances form a **Virtual Execution Cluster**, an execution environment at the application level. Distributed

big data executions are usually stateful. Therefore, once a given instance serves a client request, it continues to receive and serve all the subsequent requests.

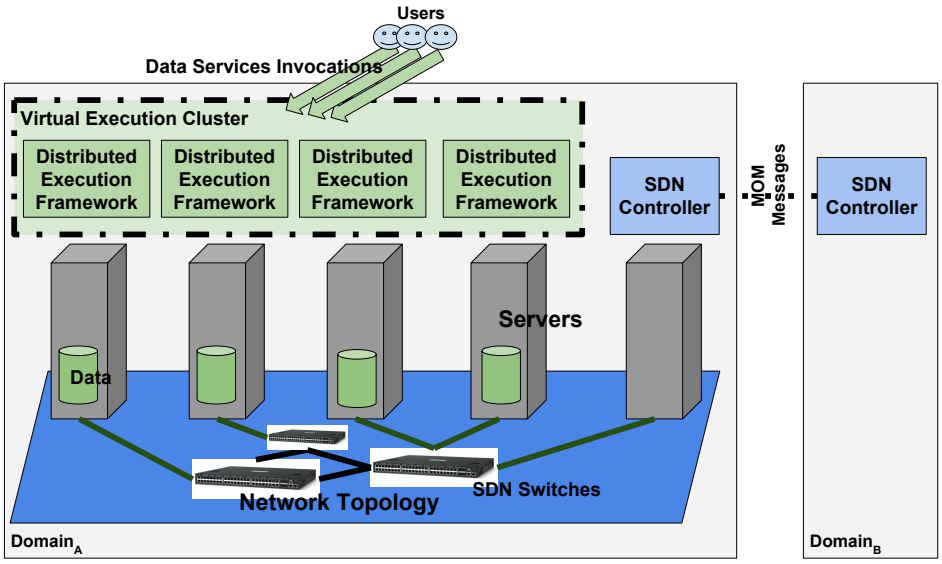


Figure 10.1: A Sample *Mayan-DS* Deployment

Mayan-DS leverages the global network knowledge of the SDN controller to find the best service instance among various potential service deployments for the service workflow execution. Each controller instance monitors and stores several metrics, including, the service execution completion time, the number of service failures, and end-to-end latency. The controllers compute and propagate the inter-domain network properties including the nature of the link between two endpoints in different domains such as bandwidth and latency and propagate them via MOM messages between controllers across the network domains. These messages include the update on the availability of a server or a data service initialized in the server, with the information received via the SDN protocols of the controller as well as the web services engines. *Mayan-DS* enables its tenants to share their direct connects with other users, by deploying a service that functions as an application level router [117]. By sharing such low-latency links across the tenants, *Mayan-DS* collectively minimizes the latency of the tenant workflows when scheduling the workflows based on the Internet paths are slower. Thus, *Mayan-DS* aims to give more options and control to the tenants on choosing their workflow execution path.

10.3 Prototype Implementation

We prototyped *Mayan-DS* to assess the feasibility of an SDDS framework. We created the data service APIs on top of Hazelcast 3.9.2 and Infinispan 9.1.5 IMDGs, using Apache Axis2 1.7.0 and Apache CXF 3.2.1 web services engines. The underlying persistent storage was composed of MySQL server and MongoDB. OpenDaylight Beryllium was leveraged as the core SDN controller, extended with AMQP for inter-domain control flows. We deployed the *Mayan-DS* on a cluster with Infinispan IMDG initialized on them, using ActiveMQ 5.15.3 as the default MOM broker.

When a service composition workflow execution requires execution of services spanning several data centers, *Mayan* aims to find the service instances in a network-aware manner, finding the best execution paths from the set of potential execution paths identified in Equation 10.5.

We developed *Mayan-DS* data services to represent various data actions, such as data storing, data deduplication, data aggregation, data analysis, and data manipulation. Some of these data services can be composite, with a series of data services chained to each other as a service composition. The data storage in *Mayan-DS* is performed at the initial stage, with later continuous updates. *Mayan-DS* defines all of its executions, including the data storage task, as data services. *Mayan-DS* leverages the constructs offered by the underlying distributed execution framework to identify the servers to host data, to minimize the communication overhead of migrating data back and forth inside the data center or cluster, as illustrated by Equation 10.1.

Figure 10.2 illustrates the layered architecture of the *Mayan-DS* framework. The data plane consists of switches and servers. The storage plane includes various SQL and NoSQL databases. The control plane comprises deployment of the OpenDaylight controller and IMDG clusters. The OpenDaylight controller and the IMDG clusters control the network data plane devices and data placement accordingly. An AMQP implementation supports the inter-domain workflows, by offering them as subscriptions between the controllers. Execution plane consists of the service APIs of web services engines and API gateways such as Kong, Tyk, and API Umbrella. The API gateways host the API endpoints of the multi-domain deployments to serve the user requests seamlessly.

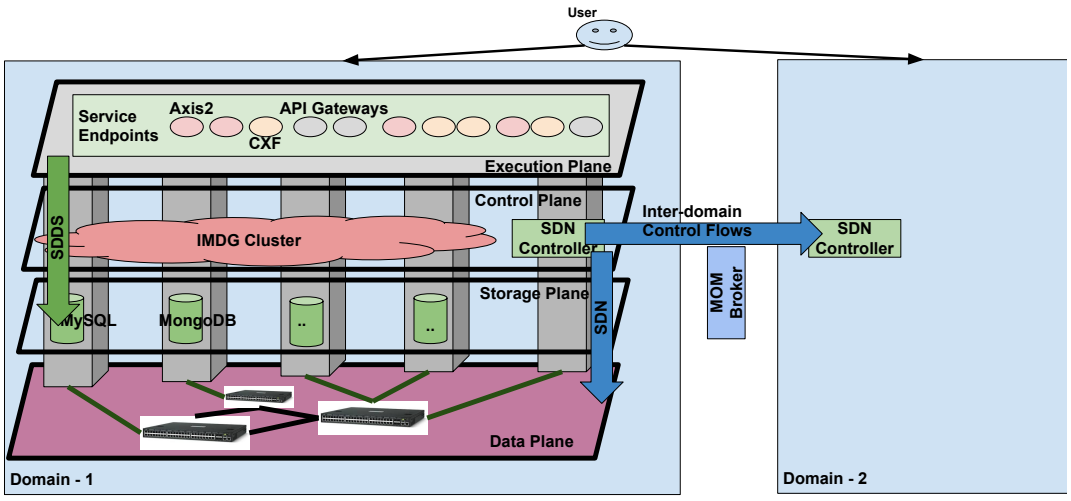


Figure 10.2: A Three-Dimensional View of the *Mayan-DS* Implementation

The separation of control from the data and storage planes enables efficient cross-control communications. The SDN controller facilitates cross-layer communication between the execution-data planes whereas the IMDGs enable execution-storage plane coordination. On the other hand, supporting inter-domain communications through MOM messages allows a dynamic multi-domain network, without requiring a static topology. Thus, *Mayan-DS* offers a platform for scalable and interoperable data service workflows.

Table 10.1: The Simulated *Mayan-DS* Deployment Environment (with modeled latency in ms)

Origin node	**Nodes connected via a direct link**
Svalbard	{Moscow (30.94), Oslo (25.33), ...}
Vladivostok	{Tokyo (10.62), ...}
Hobart	{Sydney (20.55), ...}
Cape Town	{Windhoek (19.29), ...}
Mangilao	{San Francisco (44.36), ...}
Manila	{Seoul (31.82), ...}
⋮	{⋮}

10.4 Evaluation

In this section, we seek to identify whether an SDDS approach can significantly enhance the big data workflow performance in multi-domain wide area networks at Internet scale. We evaluated *Mayan-DS* locally for its applicability for big data workflow scheduling. We then performed preliminary assessments on the feasibility of the *Mayan-DS* approach for an Internet scale execution. Due to the limited resources in evaluating *Mayan-DS* workflows in a geo-distributed manner, we resorted to simpler microbenchmarks. We leave the complete deployment and comprehensive evaluation of *Mayan-DS* in multi-domain wide area networks as future work.

We first assessed *Mayan-DS* with on-demand user-driven big data integration workflows that consume *Óbidos* service APIs, locally, for its feasibility. The big data we used in our assessments consist of a large volume of binary and textual data. It also consists of variety in data formats as well as data sources. The microbenchmarks on our *Mayan-DS* prototype confirmed that data service executions such as *Óbidos* could be extended to function as an SDDS workflow, with the architecture of *Mayan-DS*.

We then modeled a geo-distributed network with RIPE ATLAS Probes [33] and our physical servers, to evaluate the performance enhancements of a network-aware data service workflow execution at Internet scale. The network consists of nodes forming a connected graph spanning the globe. The nodes are connected to the Internet. We realistically modeled direct connects between selected pairs of nodes that are geographically close to each other, typically one in a remote region and another in a nearby city. Table 10.1 elaborates a part of the direct connects of our modeled network, together with the latency between the pairs of nodes. The direct links are bidirectional, although we list them as origin and destination for the ease of reference. We modeled the direct dedicated connects between the pairs of nodes with a 10 Gbps bandwidth, the maximum bandwidth offered by the AWS Direct Connect to connect the user servers to the cloud servers directly. We modeled the RTT of *Mayan-DS* with realistic values. We used a constant fiber path adjustment of 10% and metro fiber and a local loop length of 100 km. We considered the speed of light in fiber as 200 km/ms. We considered the equipment latency of 1 ms in our evaluation nodes. We thus estimated the latency between a pair of directly connected Internet nodes.

We evaluated *Mayan-DS* for network-aware big data workflows, by comparing the RTT

Table 10.2: Ping Times (ms) between two nodes: Regular Internet vs. *Mayan-DS*

No.	**Origin (o) → Dest. (d)**	**ISP (o → d)**	***Mayan-DS***	***Mayan-DS* Path**
1	Svalbard → São Paulo	339.164	246.436	o ⇒ Oslo → d
2	Vladivostok → São Paulo	373.712	282.451	o ⇒ Tokyo → d
3	Atlanta → Hobart	255	255	o → d
4	Hobart → São Paulo	413.869	340.617	o ⇒ Sydney → d
5	Atlanta → Svalbard	164	164	o → d
6	Atlanta → Vladivostok	287	194.68	o → Tokyo ⇒ d
7	Cape Town → Colombo	345.889	342.877	o ⇒ Windhoek → d
8	Atlanta → Windhoek	301	299.29	o → Cape Town ⇒ d
9	Mangilao → São Paulo	968.149	252.111	o ⇒ San Francisco → d
10	Singapore → Seoul	200.581	75.907	o → Manila ⇒ d
⋮	⋮ → ⋮	⋮	⋮	⋮

between two endpoints that connect through *Mayan-DS* against that of using the Internet-based connectivity of ISPs. We sent pings between the endpoints, first entirely utilizing the public Internet paths and then through *Mayan-DS*, where *Mayan-DS* chooses the best-path considering the available direct connections and the Internet paths, in composing the end-to-end connection between the origin and the destination nodes. Our considered big data applications for *Mayan-DS* are such as *Óbidos* that have a high volume and variety of data. Therefore, we note that our evaluations on *Mayan-DS* at Internet scale are indeed preliminary and are used to measure only the latency. As future work, we propose to measure more parameters such as throughput and jitter, in the context of big data applications. We observe that servers that accept incoming big data flows are limited in number, compared to those who accept simple network measurements. Therefore, we note that such evaluation would be limited by the capability to send big data across the globe, rather than research and technical challenges.

Table 10.2 lists a part of the ping times between two endpoints via the Internet-based routes as well as via *Mayan-DS*. The arrow → indicates a public Internet path (i.e., ISP-based connectivity) between two nodes, whereas ⇒ indicates a direct link. Therefore, ($o \Rightarrow i \rightarrow d$) indicates that the origin node o and the destination node d are connected via an intermediate node i, where there is a direct connect between o and i whereas the rest of the path (i.e., i to d) is a public Internet path. *Mayan-DS* exploits the available direct links that connect either the origin or the destination to an intermediate node.

We modeled a data service execution and deployed its instances across 40 pairs of geo-distributed nodes. Figure 10.3 depicts the data service invocation time via *Mayan-DS* and public Internet paths, assuming the delay caused by the service execution itself to be negligible. Each point denotes the invocation of a data service deployed in d from the origin o. o can either be a user, or a previous service whose output is sent to the service in d to compose the big data workflow.

The *Mayan-DS* controller exploits its awareness of direct links between pairs of Internet nodes and shares them for multi-tenant workflow execution through the nodes. We observe that *Mayan-DS* was able to minimize latency by exploiting the existing direct connects between a

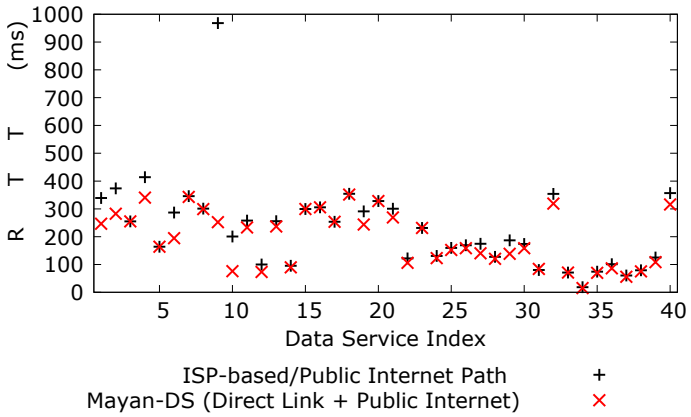


Figure 10.3: Ping times of *Mayan-DS* against the Public Internet-based Connectivity

pair of nodes as a part of its workflow execution path. *Mayan-DS* was able to achieve up to 33% reduction in latency with just a little fraction of the path routed through a low-latency direct link. When the direct link covers a substantial portion of the route, apparently, the latency reduced significantly up to 75% or more. We further observe that the availability of such direct links also minimized the outliers (i.e., the pairs of endpoints that consume an abnormally large amount of time for the data transfer), by reducing the dependency on long-haul public Internet-based links from the remote regions to the nearby regional Internet hubs. A big data workflow consists of several such data service invocations chained together. Therefore, the latency improvements of *Mayan-DS* in big data workflows will be even more prominent.

Mayan-DS exploits its network-awareness to schedule the workflows more adaptively, rather than the Internet paths that do not offer such flexibility to the end users. For example, our observations indicate that sometimes smaller towns have faster connectivity than a bigger metropolitan neighbor. While this might be due to the local congestions or other external factors, it is also difficult to predict which of the two cities will benefit more by leveraging the direct links between two servers in the cities. Therefore, by exploiting existing direct connects in a context-aware manner, *Mayan-DS* enhances the big data workflow execution performance.

10.4.1 Discussion

Exploiting services to offer unified access interfaces to big data has been proposed in previous research. While the studies on resource sharing, volunteer computing [22], and SOA [248] researched their peak a decade ago, widespread adoption of SDN and research on expanding SDN for wide area networks are recent. Therefore, despite their promising outcomes regarding interoperability, the research efforts on data services have not been extended to the Internet scale in a network-aware manner. *Mayan-DS* exploits the recent advancements in network softwarization to enable interoperable and network-aware data service workflows in wide area networks. While previous work has proposed data-aware and network-aware workflow scheduling in data centers and clouds [227], they mostly narrow their focus to a given domain managed by a

single vendor. *Mayan-DS* aims at the efficient scheduling of big data workflows in multi-domain wide area networks consisting of several federated clouds and data centers.

A computation-intensive workflow is typically smaller in duration and volume. Therefore, a quick decision is essential for minimal overhead, causing an additional constraint on service instance selection in multi-domain wide area networks. On the other hand, data service workflows have substantial demand for data volume and data rate and are typically long-running. Furthermore, overheads caused by the control flows in big data workflows are negligible, as these workflows are composed of elephant flows of data services. Therefore, it is feasible to have a more involved and time-consuming service instance selection procedure in the controller, in favor of optimal data and execution placement, for big data workflows.

The current enterprise SDDS offerings have several limitations. First, they limit their focus to certain data services rather than providing a generic data services framework. Second, they are limited to a single provider, such as a cloud or a data center. Third, they lack interoperability with other execution environments due to their vendor-specific implementation. These limitations highlight the need for a complete network-aware and interoperable SDDS framework. *Mayan-DS* is the first to propose a generic SDDS framework, with a fine-grain control inside data centers, as well as coordination between domains, for data service workflows. The *Mayan-DS* prototype is limited to specific frameworks (such as Infinispan and MongoDB). In practice, researching and implementing such an interoperable approach as a global framework is a challenging task. We looked into the way of addressing the research challenges through our proposed *Mayan-DS* architecture and its service-based approach.

Mayan-DS addresses the challenge of storing related data closer to each other through its network-awareness achieved by SDN extended with a MOM deployment. However, storing of data inside a cluster, ensuring that related objects stay closer, is specific to the storage plane. This task is achieved with the support of the human user in the data sources, for example, with careful indexing, to group the related objects. IMDGs and database frameworks already offer such efficient storage for quick query and access capabilities. *Mayan-DS* proposes to exploit such existing constructs in ensuring minimal data movements inside a data center or a cluster. It utilizes inter-domain control flows in identifying the best path for workflow execution.

SDDS aims to exploit the data awareness offered by the big data applications together with the network awareness of an extended SDN hierarchy in the wide area network. However, the realization of such systems for heterogeneous data sources is a complex problem due to several implementation challenges. Deploying an in-memory cluster on top of a persistent storage plane is a relatively trivial engineering task. However, porting existing big data workflows to use an SDDS approach involves writing the respective services to execute on top of the IMDG cluster. An increasingly complex workflow would require more and more data services to be developed and chained. The practicality of disseminating the service instances themselves across the nodes requires automation, as expecting to configure layers of platforms and software on top of the servers is questionable. One potential solution is to distribute the platform stack itself container instances, supported through frameworks such as Docker [237]. However, a detailed discussion of such possible implementation alternatives is future work.

Expanding the scope of the data service execution can be seamless once such a deployment

is established across several servers, thanks to the recent advancement of network softwarization research. In this paper, we proposed a federated SDN deployment extended with MOM as an SOA to enable network-aware big data workflows at Internet scale. However, the practicality and success of these approaches heavily depend on the adoption of the proposed framework by several infrastructure providers (including data center providers as well as independent nodes as in the case of volunteer computing). We limited our evaluations to our globally distributed servers, AWS cloud instances, and RIPE Atlas Probes. However, a latency-aware execution requires several millions of nodes in each region, to offer redundancy in execution paths with multiple alternative routes between two endpoints. We introduced the concept of SDDS at Internet scale with the fair assumptions on community adaptation of the SDDS framework. However, the realization of a complete interoperable SDDS framework and overcoming its operational challenges are left as future work.

10.5 Conclusion

SDDS extends network softwarization and SOA to bring the benefits of network-awareness and interoperability to big data applications. Data services aim at offering interoperability in wide area networks, by exploiting the standardization of web services for big data access and processing. The proliferation of data services has caused a management challenge in placing and composing data service workflows. As an SDDS framework, *Mayan-DS* extends data services with the management and resource allocation capabilities of SDN. It ensures minimal data migrations in big data applications, by keeping the related data and the execution close. *Mayan-DS* scales and distributes the data services, exploiting the global awareness of the network topology and network flow statistics of an SDN controller. It minimizes the overhead caused by frequent inter-node communications through an adaptive and context-aware execution supported by a federated deployment of SDN controllers in a wide area network. Thus, *Mayan-DS* reduces the bandwidth overhead common in distributed big data processing. Our evaluations on the *Mayan-DS* prototype indicate how an SDDS approach could be leveraged as a reusable, scalable, and resilient distributed execution framework for the big data workflows on a global scale.

V Closure

11 Final Remarks

We presented our research on network-aware SDS approaches for service composition and workflow placement across heterogeneous infrastructures in this document. Efficient resource allocation and coordination of executions in multi-tenant environments such as cloud platforms is a complex task. Executing a tenant workflow across several services hosted on multi-domain environments such as inter-clouds and the edge is even more challenging due to the limitations in composing service workflows spanning infrastructures and platforms of multiple service providers. Service providers should provide compatible APIs for seamless live migration of execution across the service instances. In this dissertation, we looked into leveraging SDN and SOA for the execution of service composition workflows of the tenant users across heterogeneous cloud and edge environments. We devised SDS architectures by exploiting network softwarization for the placement and execution of service composition workflows abiding by the tenant policies. We built SDS frameworks to efficiently manage tenant service composition workflows in heterogeneous environments, from the design to the deployment of service workflows. We thus proposed three primary contributions in this work.

First, we looked into the technical and economic feasibilities and benefits of network softwarization across heterogeneous infrastructures, from data centers to cloud networks, in various development phases from modeling to the deployment of networks. We proposed architectures to separate the network infrastructure from the execution to enable seamless migration of the workload, regardless of the execution environment and the development phase. We designed an SDN-based platform to unify the modeling and deployment of service workflows on the cloud networks. We adopted existing middlebox research, namely FlowTags, to tag the network flows with tenant policies and flow priority information. Thus, we enabled cross-layer optimizations in the network architecture, despite the presence of several network middleboxes such as load balancers and proxies in the data center network. We dynamically diverted network subflows to avert network congestion by leveraging an SDN middlebox architecture. Furthermore, by selectively enforcing redundancy on the critical tenant network flows, we enhanced SLAs in data center networks. We then introduced a virtual connectivity provider at Internet scale via cloud-assisted networks. We assessed the feasibility to exploit cloud spot VMs to build an overlay network that can function as a latency-aware connectivity provider. Specifically, we demonstrated how a third-party that does not own the infrastructure can still provide such a network by leasing cloud resources from multiple cloud providers. Thus, we bring network softwarization to various development stages and deployment environments.

Second, we extended our network softwarization approaches with SOA to enable network-aware service composition and workflow placement in multi-domain wide area networks. We presented optimal algorithms for QoS-aware service compositions at the edge with various web

services, network services, and data services, by constructing an extended SDN architecture with MOM. We identified and resolved, in addition to the technical limitations, challenges concerning enterprise policies to support an inter-domain service workflow execution and efficient resource management. Our resilient and adaptive SDSC approach brings the control of the service compositions back to the user, despite using several third-party service providers. Furthermore, we highlighted the potential of exposing diverse scenarios such as CPS, IoT, and big data applications as composable network-aware service workflows with the SDSC approach. Remarkably, smart environments can leverage the context-aware execution of the network flows, exploiting the programmability of network softwarization and the interoperability of a MOM-based publish-subscribe approach. We designed *SD-CPS*, a prototype SDS framework for CPS, and accessed it as a real-world use case of network-aware service workflows at the edge. Thus, we proposed a resilient and adaptive execution of service composition workflows across various service instances from diverse third-party service providers.

Third, we leveraged our research findings on network softwarization and service composition to execute big data applications in an interoperable and network-aware manner at Internet scale. While standards and protocols are in place in the web services domain, in practice, it is often not the case with enterprise big data applications. A framework to support service composition and workflow placement in wide area networks spanning multiple domains should ensure that the service implementations offer interoperable and compatible APIs or the framework should provide mediation across various application interfaces. While middleware frameworks such as ESB provide such mediation features, their scope is often limited to web services. We proposed interoperable big data executions by leveraging human-in-the-loop as well as network softwarization. We presented SDDS, extending the data services with the SDN paradigm. First, SDDS models the big data executions as composable data service workflows. Then, it manages and schedules the service executions in an interoperable manner, by logically separating the service workflow executions from the underlying architecture consisting of network and storage. We thus presented the design of an SDDS framework for network-aware big data executions at Internet scale.

11.1 Future Work

We proposed SDS algorithms and architectures for network-aware service composition and workflow placement. We built simulations and prototypes to evaluate the performance and efficiency of our approach to compose and deploy service workflows inside and between data centers. We see our work as the first step in an adaptive workflow execution in heterogeneous environments at Internet scale. Thus, we foresee the following list as future work extending our current contributions, while addressing the remaining open research challenges.

Multiple Providers and Regions for the Cloud-Assisted Networks: The viability of using cloud-assisted networks as an alternative connectivity provider depends on the need for connectivity services that are more dynamic than the traditional ones, and the potential for such a network to be beneficial in both economic and technological aspects. We established that the resilient architectures built atop spot instances could bring cloud expenditures down

enough to make cloud-assisted overlays profitable. With these observations, we proposed cloud-assisted networks as an economical and high-performance alternative to mainstream connectivity providers. Our evaluations were however mostly limited to particular pairs of AWS regions, despite the applicability of our approach to various cloud regions and providers. Our limitations were entirely due to the economic constraints in acquiring several cloud VMs and maintaining them over periods of months to measure the stability and performance of the overlay. We envision an Internet scale economic analysis and deployment of *NetUber* on top of multiple cloud infrastructures, as future work, to identify more interesting insights. Furthermore, the feasibility of the cloud-assisted networks and choices of our spot VMs largely depend on the current pricing model of the cloud providers. An adaptive and intelligent approach to managing the cloud VMs over time for a sustained execution of the cloud-assisted overlay network remains future work.

Deployment of SDSC approaches at Internet Scale: Deployment of an SDSC framework across multi-domain networks is a challenging task due to the requirement of coordination across several service and infrastructure providers. The challenges stem from the fact that the adoption of a new paradigm to facilitate inter-domain service compositions needs to consider organizational policies concerning interoperability and open access, in addition to the technical challenges. With a decentralized, federated controller deployment, *Mayan* seamlessly scaled out to cover a wide area network with several web service deployments, for context-aware workflows. We leave physical deployment and an extensive evaluation of a clustered and federated deployment of *Mayan* on such a global-scale as future work, due to the limitation in acquiring and maintaining multiple servers across several geographical regions. We posit that a large enterprise service provider with a global network presence and partnerships can initiate such a deployment with a relatively minimal effort.

Adaptive NSCs on Hybrid and Physical Middlebox Networks: Existence of legacy hardware middleboxes makes composing adaptive NSCs on top of hybrid and physical networks more challenging than a network with software middleboxes. Hybrid networks consist of both physical hosts and hardware middleboxes, as well as virtual hosts and VNFs. Enterprises contain several physical and hybrid networks in addition to the virtual networks. By spawning VNFs when there is a spike in demand or load for the existing instances of hardware middleboxes, we posit that a hybrid network can offer both the performance of the hardware network functions as well as the scalability and configurability of the VNFs. We propose to extend SDSC with backward-compatibility for legacy hardware middleboxes that are agnostic to network softwarization as future work. While adopting an agile network-aware service composition approach in a hybrid network composed of SDN switches and VNFs as well as legacy switches and hardware middleboxes is relatively straightforward due to the existence of virtual network components, the feasibility of such approaches in the physical systems with a complete absence of network softwarization remains an open question.

A Complete Orchestration Framework for Heterogeneous Systems: A comprehensive orchestration platform should generalize the deployments, executions, and their migrations by implementing compatible APIs or integrators for various SDN controllers and systems. Extension points should be developed to enable seamless migration across several deployment infras-

tructures and controllers. An encompassing and generic approach to interoperability is hard to achieve in heterogeneous multi-domain environments due to the diversity in the workflows in addition to the infrastructures. We propose as future work, to adopt and incorporate the *SD-CPS* approach with implementations of various networking and integration protocols for multiple use case scenarios. Furthermore, the *SD-CPS* deployments should be tested against baseline implementations of various CPS for their efficiency in addressing the identified challenges of CPS. We see our research work such as *SD-CPS*, *Mayan-DS*, and *SENDIM* as first steps towards the executions of various heterogeneous applications, such as CPS and big data applications, as interoperable network-aware service composition workflows.

A Distributed Network-Aware Data Integration Framework: Leveraging the SDDS approach, we can integrate big data in a bandwidth-efficient manner in a wide area network. We identified biomedical research data integration as a specific use case for network-aware data services where locality-awareness is mandatory due to its demand for high throughput and low latency. Complete automation of data integration workflows is often inefficient, as identifying the relevant data sets across widespread public and private scientific research data sources requires expert knowledge. Therefore, a human-in-the-loop approach improves the interoperability and efficiency of data services. However, machine learning approaches can learn to mimic human expertise over several iterations of data integration. While we limited the prototype evaluation of *Óbidos* to biomedical research data, we note that it can consume and integrate data from various scientific research data repositories such as EUDAT. We propose as future work, to leverage the network proximity among the data sources and the *Óbidos* instances for efficient data integration and sharing, by extending *Óbidos* as a data service of *Mayan-DS*. Thus, we foresee distributed virtual data warehouses - data selectively replicated and shared across various research organizations. Consequently, we propose to implement SDDS as a complete interoperable and network-aware execution platform for various real-world big data applications.

Scaling SDN for Big Data Executions: The SDN research should be on par with the rising dynamics of big data, with further innovation on network protocols and SDS implementations to match the scale and complexity of future data. Controllers are logically centralized, yet physically distributed entities. However, as the volume, variety, and velocity of big data increase further, the current centralized model offered by SDN and OpenFlow may eventually become a bottleneck. We should leverage and extend SDN for WAN, CDNs, and the Internet for an efficient big data storage and processing on the global scale. The open challenge hence lies in how current research efforts scale from local networks to autonomous systems of the Internet. We see our research efforts such as *Mayan* and *SMART* as first steps towards scaling SDN beyond data centers and supporting several tenants with network flows of different priority levels more efficiently. Further research is necessary to design and implement architectures to support big data at Internet scale through the SDN-based approaches that currently exist at a data center level.

Bibliography

[1] 365DataCenters. 365 - Managed Services, 2018. Available at https://www.365datacenters.com/services/managed-services/.

[2] 365DataCenters. 365 - U.S. Data Center Locations, 2018. Available at http://www.365datacenters.com/data-centers/.

[3] S. Abrishami, M. Naghibzadeh, and D. H. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

[4] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck. Network slicing & softwarization: A survey on principles, enabling technologies & solutions. *IEEE Communications Surveys & Tutorials*, 2018.

[5] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a large European IXP. *ACM SIGCOMM Computer Communication Review*, 42(4):163–174, 2012.

[6] T. Ahern, R. Casey, D. Barnes, R. Benson, and T. Knight. SEED Standard for the Exchange of Earthquake Data Reference Manual Format Version 2.4. *Incorporated Research Institutions for Seismology (IRIS), Seattle*, 2007.

[7] E. Ahmed and M. H. Rehmani. Mobile edge computing: opportunities, solutions, and challenges. *Future Generation Computer Systems*, 70:59–63, 2017.

[8] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou. A roadmap for traffic engineering in SDN-OpenFlow networks. *Computer Networks*, 71:1–30, 2014.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, volume 10, pages 19–19. USENIX, 2010.

[10] A. T. Al-Hammouri, Z. Al-Ali, and B. Al-Duwairi. ReCAP: A distributed CAPTCHA service at the edge of the network to handle server overload. *Transactions on Emerging Telecommunications Technologies*, 29(4):e3187, 2018.

[11] N. Alameh. Chaining geographic information web services. *IEEE Internet Computing*, 7(5):22–29, 2003.

[12] K. M. A. Alheeti, A. Gruebler, K. D. McDonald-Maier, and A. Fernando. Prediction of DoS attacks in external communication for self-driving vehicles using a fuzzy petri net

model. In *Consumer Electronics (ICCE), International Conference on*, pages 502–503. IEEE, 2016.

[13] G. Ali, J. Hu, and B. Khasnabish. Software-Defined Data Center. *ZTE Communications*, 4:002, 2013.

[14] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, pages 503–514. ACM, 2014.

[15] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[16] S. Alonso-Monsalve, F. García-Carballeira, and A. Calderón. A new volunteer computing model for data-intensive applications. *Concurrency and Computation: Practice and Experience*, 29(24):e4198, 2017.

[17] Amazon. Amazon EC2 Spot Instances, 2017. Available at https://aws.amazon.com/ec2/spot/pricing/.

[18] Amazon. AWS Regions and Endpoints, 2017. Available at http://docs.aws.amazon.com/general/latest/gr/rande.html.

[19] Amazon. How Spot Fleet Works, 2017. Available at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html.

[20] Amazon. Placement Groups, 2017. Available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[21] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[22] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid (CCGRID). Sixth International Symposium on*, volume 1, pages 73–80. IEEE, 2006.

[23] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement). In *Open grid forum*, volume 128:1, page 216, 2007.

[24] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N. P. Chue Hong, B. Collins, N. Hardman, A. C. Hume, A. Knox, M. Jackson, et al. The design and implementation of Grid database services in OGSA-DAI. *Concurrency and Computation: Practice and Experience*, 17(2-4):357–376, 2005.

[25] D. Antonioli and N. O. Tippenhauer. MiniCPS: A toolkit for security research on CPS Networks. In *Proceedings of the First Workshop on Cyber-Physical Systems-Security and/or PrivaCy*, pages 91–100. ACM, 2015.

[26] M. Anvari, T. Broderick, H. Stein, T. Chapman, M. Ghodoussi, D. W. Birch, C. Mckinley, P. Trudeau, S. Dutta, and C. H. Goldsmith. The impact of latency on surgical precision and task completion during robotic-assisted remote telepresence surgery. *Computer Aided Surgery*, 10(2):93–99, 2005.

[27] Apache Qpid. Open Source AMQP Messaging, 2013. Available at http://qpid.apache.org.

[28] S. B. Ardestani, C. J. Håkansson, E. Laure, I. Livenson, P. Stranák, E. Dima, D. Blommesteijn, and M. van de Sanden. B2SHARE: An open escience data sharing platform. In *e-Science, 11th International Conference on*, pages 448–453. IEEE, 2015.

[29] G.-P. Association et al. Contractual Arrangement: Setting up a Public-Private Partnership in the Area of Advance 5G Network Infrastructure for the Future Internet between the European Union and the 5G Infrastructure Association, 2013.

[30] D. O. Awduche and B. Jabbari. Internet traffic engineering using multi-protocol label switching (MPLS). *Computer Networks*, 40(1):111–129, 2002.

[31] AWS. AWS Global Infrastructure, 2017. Available at https://aws.amazon.com/about-aws/global-infrastructure/.

[32] B4RN. Broadband for the Rural North, 2018. Available at https://b4rn.org.uk/about-us/our-network/.

[33] V. Bajpai, S. J. Eravuchira, and J. Schönwälder. Lessons learned from using the ripe atlas platform for measurement research. *ACM SIGCOMM Computer Communication Review*, 45(3):35–42, 2015.

[34] N. Balani and R. Hathi. *Apache CXF web service development: Develop and deploy SOAP and RESTful web services*. Packt Publishing Ltd, 2009.

[35] A. Banks and R. Gupta. MQTT Version 3.1.1. *OASIS standard*, 29, 2014.

[36] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. On orchestrating virtual network functions. In *Network and Service Management (CNSM), 11th International Conference on*, pages 50–56. IEEE, 2015.

[37] J. Barr. AWS Outbound Data Transfer Prices Reduced By $0.02/GB, 2010. Available at https://aws.amazon.com/blogs/aws/aws-data-transfer-prices-reduced/.

[38] J. Barr. AWS Data Transfer Price Reduction, 2014. Available at https://aws.amazon.com/blogs/aws/aws-data-transfer-price-reduction/.

[39] J. Barr. AWS Blog. Category: Price Reduction, 2017. Available at https://aws.amazon.com/blogs/aws/category/price-reduction/.

[40] S. Barré, C. Paasch, and O. Bonaventure. Multipath TCP: from theory to practice. In *International Conference on Research in Networking*, pages 444–457. Springer, 2011.

[41] S. A. Baset. Cloud SLAs: present and future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[42] J. M. Batalla, G. Mastorakis, C. X. Mavromoustakis, C. Dobre, N. Chilamkurti, and S. Schaeckeler. Network Services Chaining in the 5G Vision. *IEEE Communications Magazine*, 55(11):112–113, 2017.

[43] J. Batalle, J. Ferrer Riera, E. Escalona, and J. A. Garcia-Espin. On the implementation of NFV over an OpenFlow infrastructure: Routing Function Virtualization. In *SDN for Future Networks and Services (SDN4FNS)*, pages 1–6. IEEE, 2013.

[44] B. Bauer. Network traffic monitoring, Sept. 6 2002. US Patent App. 10/236,402.

[45] O. Ben-Kiki, C. Evans, and B. Ingerson. YAML Ain't Markup Language (YAML) version 1.1. *yaml.org, Tech.Rep*, page 23, 2005.

[46] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd Symposium on Cloud Computing (SoCC)*, page 8. ACM, 2011.

[47] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking (HotSDN)*, pages 1–6. ACM, 2014.

[48] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. In *Cloud Computing*, volume 3, pages 81–84. IEEE, 2014.

[49] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan. Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications*, 102:1–16, 2017.

[50] M. Bjorklund. YANG-A data modeling language for the Network Configuration Protocol (NETCONF), 2010.

[51] R. Bonafiglia, G. Castellano, I. Cerrato, and F. Risso. End-to-end service orchestration across sdn and cloud computing domains. In *Network Softwarization (NetSoft), Conference on*, pages 1–6. IEEE, 2017.

[52] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[53] C. Borckholder, A. Heinzel, Y. Kaniovskyi, S. Benkner, A. Lukas, and B. Mayer. A Generic, Service-based Data Integration Framework Applied to Linking Drugs & Clinical Trials. *Procedia Computer Science*, 23:24–35, 2013.

[54] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Data Engineering, Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[55] B. Boudreau. Global Bandwidth & IP Pricing Trends, 2017. Available at http://www2.telegeography.com/hubfs/2017/presentations/telegeography-ptc17-pricing.pdf.

[56] C. Bouras, P. Ntarzanos, and A. Papazois. Cost modeling for SDN/NFV based mobile 5G networks. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2016 8th International Congress on*, pages 56–61. IEEE, 2016.

[57] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1, 2000.

[58] R. Boyd. Network Service Orchestration enabled by Tail-f, 2015. Available at https://blogs.cisco.com/cin/tail-f.

[59] M. Burgess and R. Ralston. Distributed Resource Administration Using CFEngine. *Software: practice and experience*, 27(9):1083–1101, 1997.

[60] C. X. Cai, F. Le, X. Sun, G. G. Xie, H. Jamjoom, and R. H. Campbell. CRONets: Cloud-Routed Overlay Networks. In *Distributed Computing Systems (ICDCS), 36th International Conference on*, pages 67–77. IEEE, 2016.

[61] Z. Cai, A. L. Cox, and T. Ng. Maestro: A System for Scalable OpenFlow Control. Technical report, TSEN Maestro-Technical Report TR10-08, Rice University, 2010.

[62] R. N. Calheiros, M. A. Netto, C. A. De Rose, and R. Buyya. EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, 43(5):595–612, 2013.

[63] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[64] D. Calvaresi, M. Marinoni, A. Sturm, M. Schumacher, and G. Buttazzo. The challenge of real-time multi-agent systems for enabling IoT and CPS. In *Proceedings of the International Conference on Web Intelligence*, pages 356–364. ACM, 2017.

[65] caMicroscope. caMicroscope, 2018. Available at http://camicroscope.org.

[66] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2):7–23, 1999.

[67] C. Ö. Çaparlar and A. Dönmez. What is Scientific Research and How Can it be Done? *Turkish journal of anaesthesiology and reanimation*, 44(4):212, 2016.

[68] A. Cardenas, S. Amin, B. Sinopoli, A. Giani, A. Perrig, and S. Sastry. Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, page 5, 2009.

[69] M. Carlson, A. Yoder, L. Schoeb, D. Deel, C. Pratt, C. Lionetti, and D. Voigt. Software Defined Storage. *Storage Networking Industry Assoc. working draft*, pages 20–24, 2014.

[70] G. Carneiro. NS-3: Network Simulator 3. In *UTM Lab Meeting April*, volume 20, 2010.

[71] B. Carpenter and S. Brim. *Middleboxes: Taxonomy and Issues*. RFC Editor, 2002.

[72] Catalogic. ECX, 2018. Available at https://catalogicsoftware.com/products/ecx/.

[73] D. Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[74] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD record*, 26(1):65–74, 1997.

[75] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, 275:314–347, 2014.

[76] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[77] K. Clark, B. Vendt, K. Smith, J. Freymann, J. Kirby, P. Koppel, S. Moore, S. Phillips, D. Maffitt, M. Pringle, et al. The Cancer Imaging Archive (TCIA): maintaining and operating a public information repository. *Journal of digital imaging*, 26(6):1045–1057, 2013.

[78] M. Claypool and K. Claypool. Latency can kill: precision and deadline in online games. In *Proceedings of the first annual SIGMM conference on Multimedia systems*, pages 215–222. ACM, 2010.

[79] CloudDirect. Move to Cloud ID - quickly, easily and securely, 2017. Available at https://www.clouddirect.net/.

[80] Cloudflare. Cloudflare Argo, 2017. Available at https://www.cloudflare.com/argo/.

[81] Cogent. Cogent IP Transit, 2017. Available at http://www.cogentco.com/en/products-and-services/ip-transit.

[82] M. Collina, G. E. Corazza, and A. Vanelli-Coralli. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC)*, pages 36–41. IEEE, 2012.

[83] Commvault. Commvault Introduces New Innovations for the Commvault Data Platform in Software Defined Data Services, Orchestration and User Interface, 2016. Available at https://www.commvault.com/news/2016/october/commvault-introduces-new-innovations-for-the-commvault-data-platform-in-software-defined-data-services-orchestration-and-user-interface.

[84] Console. Console - The Cloud Connection Company, 2017. Available at https://www.consoleconnect.com/.

[85] M. Cummings and S. Heath. Mode switching and software download for software defined radio: the SDR Forum approach. *IEEE Communications Magazine*, 37(8):104–106, 1999.

[86] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[87] E. Curry. Message-oriented middleware. In *Middleware for communications*, pages 1–28. John Wiley & Sons, 2004.

[88] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, pages 1629–1637. IEEE, 2011.

[89] Cyrus. May 9th: IBM announces version 2.2.6 of IBM Spectrum Copy Data Management, 2017. Available at https://spectrumcdmsite.wordpress.com/2017/05/12/may-9th-ibm-announces-version-2-2-6-of-ibm-spectrum-copy-data-management/.

[90] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM Computer Communication Review*, volume 34:4, pages 15–26. ACM, 2004.

[91] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos. SDSecurity: A software defined security experimental framework. In *Communication Workshop (ICCW), International Conference on*, pages 1871–1876. IEEE, 2015.

[92] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos. SDStorage: A Software Defined Storage Experimental Framework. In *Cloud Engineering (IC2E), International Conference on*, pages 341–346. IEEE, 2015.

[93] B. S. Davie and Y. Rekhter. *MPLS: technology and applications*. Morgan Kaufmann Publishers Inc., 2000.

[94] S. Dawson-Haggerty, J. Ortiz, J. Trager, D. Culler, and R. H. Katz. Energy Savings and the "Software-Defined" Building. *IEEE Design & Test of Computers*, 29(4):56–57, 2012.

[95] M. De Brito, S. Hoque, R. Steinke, A. Willner, and T. Magedanz. Application of the Fog computing paradigm to Smart Factories and cyber-physical systems. *Transactions on Emerging Telecommunications Technologies*, 29(4):e3184, 2018.

[96] R. L. S. De Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. R. Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), Colombian Conference on*, pages 1–6. IEEE, 2014.

[97] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[98] S. Deng, L. Huang, W. Tan, and Z. Wu. Top-Automatic Service Composition: A Parallel Method for Large-Scale Service Sets. *Automation Science and Engineering, IEEE Transactions on*, 11(3):891–905, 2014.

[99] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[100] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column generation*, volume 5. Springer Science & Business Media, 2006.

[101] K. C. Dey, A. Rayamajhi, M. Chowdhury, P. Bhavsar, and J. Martin. Vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication in a heterogeneous wireless network–Performance evaluation. *Transportation Research Part C: Emerging Technologies*, 68:168–184, 2016.

[102] T. Dillon, C. Wu, and E. Chang. Cloud computing: issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010.

[103] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN controller. *ACM SIGCOMM computer communication review*, 43(4):7–12, 2013.

[104] C. Dixon, D. Olshefski, V. Jain, C. DeCusatis, W. Felter, J. Carter, M. Banikazemi, V. Mann, J. M. Tracey, and R. Recio. Software Defined Networking to Support the Software Defined Environment. *IBM Journal of Research and Development*, 58(2/3):3:1–3:14, 2014.

[105] A. Dogac, Y. Tambag, P. Pembecioglu, S. Pektas, G. Laleci, G. Kurt, S. Toprak, and Y. Kabak. An ebXML infrastructure implementation through UDDI registries and RosettaNet PIPs. In *Proceedings of the SIGMOD international conference on Management of data*, pages 512–523. ACM, 2002.

[106] X. Dong, H. Lin, R. Tan, R. K. Iyer, and Z. Kalbarczyk. Software-defined networking for smart grid resilience: Opportunities and challenges. In *Proceedings of the 1st Workshop on Cyber-Physical System Security*, pages 61–68. ACM, 2015.

[107] X. L. Dong and D. Srivastava. Big data integration. In *Data Engineering (ICDE), 29th International Conference on*, pages 1245–1248. IEEE, 2013.

[108] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and control element separation (ForCES) protocol specification, 2010.

[109] Z. Du, J. Huai, and Y. Liu. Ad-UDDI: An active and distributed service registry. In *Technologies for E-Services*, pages 58–71. Springer, 2006.

[110] EdgeConneX. Edge Data Center Locations, 2018. Available at http://www.edgeconnex.com/edge-data-center-locations/.

[111] J. Elek, D. Jocha, and R. Szabo. Network Function Chaining in DCs: The Unified Recurring Control Approach. In *Software Defined Networks (EWSDN), Fourth European Workshop on*, pages 13–18. IEEE, 2015.

[112] P. T. Endo, A. V. de Almeida Palhares, N. N. Pereira, G. E. Goncalves, D. Sadok, J. Kelner, B. Melander, and J.-E. Mangs. Resource allocation for distributed cloud: concepts and research challenges. *IEEE network*, 25(4), 2011.

[113] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. *NETCONF configuration protocol*. RFC Editor, 2006.

[114] Epsilon. Epsilon Telecommunications Limited – Connectivity Made Simple, 2017. Available at http://www.epsilontel.com.

[115] D. Erickson. The beacon openflow controller. In *Proceedings of the second SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*, pages 13–18. ACM, 2013.

[116] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[117] A. Faisal and S. Kasetty. Application level router for routing heterogeneous input to the most appropriate application, 2011. US Patent 7,895,346.

[118] K. Fall, K. Varadhan, et al. The ns Manual (formerly ns Notes and Documentation). *The VINT project*, 47:19–231, 2005.

[119] I. Farris, T. Taleb, H. Flinck, and A. Iera. Providing ultra-short latency to user-centric 5G applications at the mobile network edge. *Transactions on Emerging Telecommunications Technologies*, 2017.

[120] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *NSDI*. USENIX, 2014.

[121] N. Feamster, L. Gao, and J. Rexford. How to lease the Internet in your spare time. *ACM SIGCOMM Computer Communication Review*, 37(1):61–64, 2007.

[122] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *Queue*, 11(12):20, 2013.

[123] M. Fernandez. Evaluating OpenFlow controller paradigms. In *ICN, The Twelfth International Conference on Networks*, pages 151–157, 2013.

[124] J. Fiaidhi, I. Bojanova, J. Zhang, and L.-J. Zhang. Enforcing multitenancy for cloud computing environments. *IT professional*, 1(1):16–18, 2012.

[125] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *SIGPLAN Notices*, volume 47:6, pages 263–274. ACM, 2012.

[126] N. M. Freris et al. A Software Defined architecture for Cyberphysical Systems. In *Software Defined Systems (SDS), Fourth International Conference on*, pages 54–60. IEEE, 2017.

[127] A. Galis, S. Clayman, L. Mamatas, J. R. Loyola, A. Manzalini, S. Kuklinski, J. Serrat, and T. Zahariadis. Softwarization of future networks and services-programmable enabled networks as next generation software defined networks. In *SDN for Future Networks and Services (SDN4FNS)*, pages 1–7. IEEE, 2013.

[128] A. Gandhi and J. Chan. Analyzing the Network for AWS Distributed Cloud Computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):12–15, 2015.

[129] N. Gaur, K. S. Bhogal, C. D. Johnson, T. E. Kaplinger, and D. C. Berg. System and method of optimization of in-memory data grid placement, Aug. 2 2016. US Patent 9,405,589.

[130] Gearpump. Apache Gearpump, 2018. Available at http://gearpump.apache.org/overview.html.

[131] H. H. Gharakheili and V. Sivaraman. Cloud Assisted Home Networks. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 31–36. ACM, 2017.

[132] G. Gibb, H. Zeng, and N. McKeown. Outsourcing network functionality. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 73–78. ACM, 2012.

[133] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213. ACM, 2004.

[134] D. J. Glancy. Autonomous and automated and connected cars-oh my: first generation autonomous cars in the legal ecosystem. *Minn. JL Sci. & Tech.*, 16:619, 2015.

[135] Google. Google Cloud Platform - Cloud Locations, 2017. Available at https://cloud.google.com/about/locations/.

[136] Google. Preemptible VM Instances, 2017. Available at https://cloud.google.com/compute/docs/instances/preemptible.

[137] Google. Project Fi, 2017. Available at https://fi.google.com/about/.

[138] J. D. Gradecki and J. Cole. *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[139] S. Graham, G. Daniels, D. Davis, Y. Nakamura, S. Simeonov, P. Brittenham, P. Fremantle, D. Koenig, and C. Zentner. *Building Web services with Java: making sense of XML, SOAP, WSDL, and UDDI*. SAMS publishing, 2004.

[140] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[141] J. Groff. VMware Certified Software-Defined Data Services Offering Enables Enhanced Performance Gains for Tier-1 Virtualized Applications, 2016. Available at http://www.primaryio.com/vmware-certified-software-defined-data-services-offering-enables-enhanced-performance-gains-for-tier-1-virtualized-applications/.

[142] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*. USENIX, 2015.

[143] N. Grozev and R. Buyya. Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014.

[144] L. Gurgen, O. Gunalp, Y. Benazzouz, and M. Gallissot. Self-aware cyber-physical systems and applications in smart buildings and cities. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1149–1154. EDA Consortium, 2013.

[145] M. J. Hadley. Web application description language (WADL), 2006.

[146] P. Halpernj. RFC7665, Service Function Chaining (SFC) architecture, 2015.

[147] L. Hamilton. Compression as a Service is Now Available in the Cloud! Boostedge.Net for ISPs, Telcos and Enterprises UbFast.com for end-users! , 2013. Available at http://www.businesswire.com/news/home/20130207006266/en/Compression-Service-Cloud!-Boostedge.Net-ISPs-Telcos-Enterprises.

[148] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, 2015.

[149] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-serve: Load-balancing web traffic using openflow. *ACM SIGCOMM Demo*, 4(5):6, 2009.

[150] O. Haq and F. R. Dogar. Leveraging the power of cloud for reliable wide area communication. In *Proceedings of the 14th Workshop on Hot Topics in Networks (HotNets)*, page 19. ACM, 2015.

[151] O. Haq, M. Raja, and F. R. Dogar. Measuring and Improving the Reliability of Wide-Area Cloud Paths. In *Proceedings of the 26th International Conference on World Wide Web*, pages 253–262. International World Wide Web Conferences Steering Committee, 2017.

[152] HashiCorp Suite. Vault Project - Encryption as a Service, 2018. Available at https://www.vaultproject.io/guides/encryption/index.html.

[153] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[154] M. Hausenblas and J. Nadeau. Apache Drill: Interactive Ad-hoc Analysis at Scale. *Big Data*, 1(2):100–104, 2013.

[155] X. He, P. Shenoy, R. Sitaraman, and D. Irwin. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 207–218. ACM, 2015.

[156] P. Heinzlreiter, J. R. Perkins, O. T. Tirado, T. J. M. Karlsson, J. A. Ranea, A. Mitterecker, M. Blanca, and O. Trelles. A Cloud-based GWAS Analysis Pipeline for Clinical Researchers. In *CLOSER*, pages 387–394, 2014.

[157] T. Hey and A. E. Trefethen. Cyberinfrastructure for e-Science. *Science*, 308(5723):817–821, 2005.

[158] A. Heydarnoori. Deploying Component–Based Applications: Tools and Techniques. In *Software Engineering Research, Management and Applications*, pages 29–42. Springer, 2008.

[159] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22. USENIX, 2011.

[160] P. Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[161] HL7. FHIR, 2018. Available at https://www.hl7.org/fhir/.

[162] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[163] A. C. Houle, L.-P. Boulianne, and L. Dupras. SD-WAN: A Technology for the Efficient Use of Bandwidth in Multi-Wavelength Networks. In *Optical Fiber communication/National Fiber Optic Engineers Conference (OFC/NFOEC)*, pages 1–10. IEEE, 2008.

[164] P. Hu. A system architecture for software-defined industrial Internet of Things. In *2015 IEEE International Conference on Ubiquitous Wireless Broadband (ICUWB)*, pages 1–5. IEEE, 2015.

[165] Z. Huang. *Data Integration For Urban Transport Planning*. Citeseer, 2003.

[166] N. Huin, B. Jaumard, and F. Giroire. Optimization of network service chain provisioning. In *IEEE International Conference on Communications 2017*, 2017.

[167] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. In *Communication systems software and middleware and workshops, COMSWARE. 3rd international conference on*, pages 791–798. IEEE, 2008.

[168] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[169] IBM. IBM Spectrum Copy Data Management, 2018. Available at https://www.ibm.com/us-en/marketplace/spectrum-copy-data-management.

[170] F. Irmert, M. Meyerhöfer, and M. Weiten. Towards Runtime Adaptation in a SOA Environment. *RAM-SE*, 7:17–26, 2007.

[171] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS operating systems review*, 41:3:59–72, 2007.

[172] iTel. iTel MPLS (IP VPN) High Performance Connectivity, 2017. Available at https://itel.com/mpls/.

[173] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[174] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, A. Rindos, et al. SDIoT: a software defined based internet of things framework. *Journal of Ambient Intelligence and Humanized Computing*, 6(4):453–461, 2015.

[175] S. Jeschke, C. Brecher, T. Meisen, D. Özdemir, and T. Eschert. Industrial internet of things and cyber manufacturing systems. In *Industrial Internet of Things*, pages 3–19. Springer, 2017.

[176] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research Directions in Network Service Chaining. In *SDN for Future Networks and Services*, SDN4FNS, pages 1–7, Nov 2013.

[177] M. Johns. *Getting Started with Hazelcast*. Packt Publishing Ltd, 2013.

[178] F. K. Jondral. Software-Defined Radio: Basics and Evolution to Cognitive Radio. *EURASIP journal on wireless communications and networking*, 2005(3):275–283, 2005.

[179] A. Kadadi, R. Agrawal, C. Nyamful, and R. Atiq. Challenges of data integration and interoperability in big data. In *Big Data, International Conference on*, pages 38–40. IEEE, 2014.

[180] B. R. Kandukuri, V. R. Paturi, and A. Rakshit. Cloud security issues. In *Services Computing, SCC. International Conference on*, pages 517–520. IEEE, 2009.

[181] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, 2007.

[182] G. Kapitsaki, D. Kateros, I. Foukarakis, G. Prezerakos, D. Kaklamani, and I. Venieris. Service Composition: State of the art and future challenges. In *2007 16th IST Mobile and Wireless Communications Summit*, pages 1–5. IEEE, 2007.

[183] Y. Kargín, M. Ivanova, Y. Zhang, S. Manegold, and M. Kersten. Lazy ETL in action: ETL technology dates scientific data. *Proceedings of the VLDB Endowment*, 6(12):1286–1289, 2013.

[184] S. Karnouskos. Cyber-physical systems in the smartgrid. In *Industrial Informatics (INDIN), 9th International Conference on*, pages 20–23. IEEE, 2011.

[185] P. Kathiravelu and L. Veiga. An Adaptive Distributed Simulator for Cloud and MapReduce Algorithms and Architectures. In *Utility and Cloud Computing (UCC), 7th International Conference on*, pages 79–88. IEEE, 2014.

[186] S. Kaur, J. Singh, and N. S. Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, volume 138, 2014.

[187] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2012.

[188] A. Khosravi, S. K. Garg, and R. Buyya. Energy and carbon-efficient placement of virtual machines in distributed cloud data centers. In *Euro-Par 2013 Parallel Processing*, pages 317–328. Springer, 2013.

[189] A. Kirilenko, A. S. Kyle, M. Samadi, and T. Tuzun. The flash crash: The impact of high frequency trading on an electronic market. *Available at SSRN*, 1686004, 2011.

[190] A. Klein, F. Ishikawa, and S. Honiden. Towards network-aware service composition in the cloud. In *Proceedings of the 21st international conference on World Wide Web*, pages 959–968. ACM, 2012.

[191] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, 1:23–31, 1987.

[192] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu. Towards reliable interactive data cleaning: A user survey and recommendations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 9. ACM, 2016.

[193] V. P. Kumar, T. Lakshman, and D. Stiliadis. Beyond best effort: router architectures for the differentiated services of tomorrow's Internet. *IEEE Communications magazine*, 36(5):152–164, 1998.

[194] S. Lange, A. Grigorjew, T. Zinner, P. Tran-Gia, and M. Jarschel. A Multi-objective Heuristic for the Optimization of Virtual Network Function Chain Placement. In *Teletraffic Congress (ITC), 29th International*, volume 1, pages 152–160. IEEE, 2017.

[195] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *European Semantic Web Conference*, pages 493–507. Springer, 2008.

[196] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[197] D. Lecarpentier, P. Wittenburg, W. Elbers, A. Michelini, R. Kanso, P. Coveney, and R. Baxter. EUDAT: A new cross-disciplinary data infrastructure for science. *International Journal of Digital Curation*, 8(1):279–287, 2013.

[198] E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72*, 2007.

[199] E. A. Lee. Cyber Physical Systems: Design Challenges. In *11th International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[200] E. A. Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869, 2015.

[201] G. Lee, S. Doyle, J. Monaco, A. Madabhushi, M. D. Feldman, S. R. Master, and J. E. Tomaszewski. A knowledge representation framework for integration, classification of multi-scale imaging and non-imaging data: Preliminary results in predicting prostate cancer recurrence by fusing mass spectrometry and histology. In *International Symposium on Biomedical Imaging: From Nano to Macro*, pages 77–80. IEEE, 2009.

[202] J. Lee, B. Bagheri, and H.-A. Kao. A Cyber-Physical Systems Architecture for Industry 4.0-based Manufacturing Systems. *Manufacturing Letters*, 3:18–23, 2015.

[203] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, page 9. ACM, 2015.

[204] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[205] C.-S. Li, B. Brech, S. Crowder, D. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, et al. Software defined environments: An introduction. *IBM Journal of Research and Development*, 58(2/3):1:1–1:11, 2014.

[206] G. Li. Human-in-the-loop data integration. *Proceedings of the VLDB Endowment*, 10(12):2006–2017, 2017.

[207] X. Li, B. Plale, N. Vijayakumar, R. Ramachandran, S. Graves, and H. Conover. Real-time storm detection and weather forecast activation through data mining and events processing. *Earth Science Informatics*, 1(2):49–57, 2008.

[208] J. Liao, J. Wang, B. Wu, and W. Wu. Toward a multiplane framework of NGSON: A required guideline to achieve pervasive services and efficient resource utilization. *Communications Magazine, IEEE*, 50(1):90–97, 2012.

[209] D. S. Linthicum. *Cloud computing and SOA convergence in your enterprise: a step-by-step guide*. Pearson Education, 2009.

[210] P. Lipton. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. Technical report, OASIS, 2017. Available at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[211] J. Liu, Y. Li, M. Chen, W. Dong, and D. Jin. Software-Defined Internet of Things for Smart Urban Sensing. *IEEE Communications Magazine*, 53(9):55–63, 2015.

[212] K. Liu, J. K. Ng, V. C. Lee, S. H. Son, and I. Stojmenovic. Cooperative data scheduling in hybrid vehicular ad hoc networks: VANET as a Software Defined Network. *IEEE/ACM transactions on networking*, 24(3):1759–1773, 2016.

[213] Y. Liu, G. Shou, Y. Hu, Z. Guo, H. Li, and H. S. Seah. Towards a smart campus: Innovative applications with WiCloud platform based on mobile edge computing. In *Computer Science and Education (ICCSE), 12th International Conference on*, pages 133–138. IEEE, 2017.

[214] A. Lombardo, A. Manzalini, G. Schembra, G. Faraci, C. Rametta, and V. Riccobene. An open framework to enable NetFATE (network functions at the edge). In *Network Softwarization (NetSoft), 1st Conference on*, pages 1–6. IEEE, 2015.

[215] D. Lombraña González, A. Harutyunyan, B. Segal, I. Zacharov, E. McIntosh, P. Jones, M. Giovannozzi, L. Rivkin, M. Marquina, P. Skands, et al. LHC@ HOME: A volunteer computing system for massive numerical simulations of beam dynamics and high energy physics events. In *Conf. Proc.*, volume 1205201:IPAC-2012-MOPPD061, pages 505–507, 2012.

[216] J. Loope. *Managing Infrastructure with Puppet: Configuration Management at Scale*. O'Reilly Media, Inc., 2011.

[217] V. Lopez, O. G. de Dios, L. Contreras, J. Foster, H. Silva, L. Blair, J. Marsella, T. Szyrkowiec, A. Autenrieth, C. Liou, et al. Demonstration of SDN orchestration in optical multi-vendor scenarios. In *Optical Fiber Communications Conference and Exhibition (OFC), 2015*, pages 1–3. IEEE, 2015.

[218] P. Lovelock. Unleashing the Potential of the Internet in Central Asia, South Asia, the Caucasus and Beyond. *ADB Consultant's Report*, pages 27–28, 2015.

[219] A. Ludwig and S. Schmid. Distributed Cloud Market: Who Benefits from Specification Flexibilities? *ACM SIGMETRICS Performance Evaluation Review*, 43(3):38–41, 2015.

[220] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni. Impact of mobility on Message Oriented Middleware (MOM) protocols for collaboration in transportation. In *Computer Supported Cooperative Work in Design (CSCWD), 19th International Conference on*, pages 115–120. IEEE, 2015.

[221] D.-M. Lyu, Y. Tian, Y. Wang, D.-Y. Tong, W.-W. Yin, and J.-S. Li. Design and implementation of clinical data integration and management system based on Hadoop platform. In *Information Technology in Medicine and Education (ITME), 2015 7th International Conference on*, pages 76–79. IEEE, 2015.

[222] J. Macker. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations, 1999.

[223] A. Mahmood, C. Casetti, C.-F. Chiasserini, P. Giaccone, and J. Harri. Mobility-aware edge caching for connected cars. In *Wireless On-demand Network Systems and Services (WONS), 12th Annual Conference on*, pages 1–8. IEEE, 2016.

[224] A. Manzalini and R. Saracco. Software Networks at the Edge: a shift of paradigm. In *SDN for Future Networks and Services (SDN4FNS)*, pages 1–6. IEEE, 2013.

[225] F. Marchioni. *Infinispan data grid platform*. Packt Pub., 2012.

[226] A. Marotta, E. Zola, F. D'Andreagiovanni, and A. Kassler. A fast robust optimization-based heuristic for the deployment of green virtual network functions. *Journal of Network and Computer Applications*, 95:42–53, 2017.

[227] F. Marozzo, F. Rodrigo Duro, J. Garcia Blas, J. Carretero, D. Talia, and P. Trunfio. A data-aware scheduling strategy for workflow execution in clouds. *Concurrency and Computation: Practice and Experience*, 29(24):e4229, 2017.

[228] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proceedings of the 11th Conference on Networked Systems Design and Implementation (NSDI)*, pages 459–473. USENIX, 2014.

[229] M. Mathews. PlexxiPulse – Software-Defined Data Center 50, 2017. Available at http://www.plexxi.com/2017/03/plexxipulse-2017-software-defined-data-center-50/.

[230] R. McClatchey, A. Anjum, H. Stockinger, A. Ali, I. Willers, and M. Thomas. Data intensive and network aware (DIANA) grid scheduling. *Journal of Grid computing*, 5(1):43–64, 2007.

[231] N. McKeown. Software-Defined Networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.

[232] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[233] J. Medved, R. Varga, A. Tkacik, and K. Gray. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *15th International Symposium on*, pages 1–6. IEEE, 2014.

[234] MEF. Lifecycle Service Orchestration — Third Network, 2017. Available at https://www.mef.net/third-network/lifecycle-service-orchestration.

[235] Megaport. Megaport, 2017. Available at http://megaport.com/.

[236] C. Mellor. Hammer hopes to nail software-defined future for Commvault, 2016. Available at https://www.theregister.co.uk/2016/10/26/commvault_set_fair_for_sustained_turnaround/.

[237] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[238] O. Michel and E. Keller. SDN in wide-area networks: A survey. In *Software Defined Systems (SDS), Fourth International Conference on*, pages 37–42. IEEE, 2017.

[239] Microsoft. Software-Defined Datacenter (SDDC) - Windows Server 2016, 2017. Available at https://www.microsoft.com/en-us/cloud-platform/software-defined-datacenter.

[240] E. Milchevski and S. Michel. ligDB-Online Query Processing Without (almost) any Storage. In *EDBT*, pages 683–688, 2015.

[241] P. Mildenberger, M. Eichelberg, and E. Martin. Introduction to the DICOM standard. *European radiology*, 12(4):920–927, 2002.

[242] F. P. Miller, A. F. Vandome, and J. McBrewster. Apache Maven, 2010.

[243] H. Moens and F. De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 418–423. IEEE, 2014.

[244] E. Molina and E. Jacob. Software-defined networking in cyber-physical systems: A survey. *Computers & Electrical Engineering*, 66:407–419, 2018.

[245] S. Munir and J. A. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), International Conference on*, pages 127–138. IEEE, 2014.

[246] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[247] T. D. Nadeau and K. Gray. *SDN: Software Defined Networks*. O'Reilly Media, Inc., 2013.

[248] E. Newcomer and G. Lomow. *Understanding SOA with Web services*. Addison-Wesley, 2005.

[249] Nexion. Nexion Networks, 2018. Available at https://www.nexionnetworks.com/cloud-solutions/.

[250] A. Nierbeck, J. Goodyear, J. Edstrom, and H. Kesler. *Apache Karaf Cookbook*. Packt Publishing Ltd, 2014.

[251] T. Nolte. Compositionality and CPS from a Platform Perspective. In *17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 57–60. IEEE, 2011.

[252] B. W. Norton. Internet Transit Prices - Historical and Projected, 2014. Available at http://drpeering.net/white-papers/Internet-Transit-Pricing-Historical-And-Projected.php.

[253] Nuage. Nuage Networks, 2017. Available at http://www.nuagenetworks.net/products/.

[254] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service Placement in a Shared Wide-Area Platform. In *USENIX Annual Technical Conference, General Track*, pages 273–288, 2006.

[255] Oracle. REST Data Services, 2018. Available at http://www.oracle.com/technetwork/developer-tools/rest-data-services/overview/index.html.

[256] C. Orłowski, E. Szczerbicki, and J. Grabowski. Enterprise service bus architecture for the big data systems, 2014.

[257] G. Orsini, D. Bade, and W. Lamersdorf. CloudAware: Empowering context-aware self-adaptation for mobile applications. *Transactions on Emerging Telecommunications Technologies*, 29(4):e3210, 2018.

[258] G. R. Osborn. Hardware resource identifier for software-defined communications system, May 23 2006. US Patent 7,050,807.

[259] OSGi Alliance. *OSGi service platform, release 3*. IOS Press, Inc., 2003.

[260] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.

[261] J. Pacheco, C. Tunc, and S. Hariri. Design and evaluation of resilient infrastructures systems for smart cities. In *Smart Cities Conference (ISC2), International*, pages 1–6. IEEE, 2016.

[262] PacketFabric. PacketDirect, 2017. Available at https://www.packetfabric.com/packetdirect/.

[263] PacketFabric. PacketFabric, 2017. Available at https://www.packetfabric.com/.

[264] F. Paganelli, M. Ulema, and B. Martini. Context-aware service composition and delivery in NGSONs over SDN. *Communications Magazine, IEEE*, 52(8):97–105, 2014.

[265] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AutoPlacer: Scalable Self-Tuning Data Placement in Distributed Key-Value Stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(4):19, 2015.

[266] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, WISE. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[267] O. Parekh. Iterative packing for demand and hypergraph matching. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 349–361. Springer, 2011.

[268] M. Pasha and K. U. R. Khan. Architecture and Channel Aware Task Offloading in Opportunistic Vehicular Edge Networks. *IJSSST*, 18(4):15.1–15.7, 2015.

[269] P. Patel, M. I. Ali, and A. Sheth. On Using the Intelligent Edge for IoT Analytics. *IEEE Intelligent Systems*, 32(5):64–69, 2017.

[270] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. big'web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.

[271] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. Axis2, middleware for next generation web services. In *Web Services (ICWS). International Conference on*, pages 833–840. IEEE, 2006.

[272] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *Proceedings of the conference on SIGCOMM*, pages 307–318. ACM, 2014.

[273] M. Persson and A. Håkansson. A Communication Protocol for different communication technologies in Cyber-Physical Systems. *Procedia Computer Science*, 60:1697–1706, 2015.

[274] D. Petcu. Multi-Cloud: expectations and current approaches. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 1–6. ACM, 2013.

[275] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending Networking into the Virtualization Layer. In *HotNets*, 2009.

[276] K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain sdn controllers. In *Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE, 2014.

[277] Portworx. Portworx, 2018. Available at https://portworx.com/.

[278] PrimaryIO. PrimaryIO: Application Performance Accelerator (APA) 2.5, 2018. Available at http://www.primaryio.com/.

[279] PureStorage. The data platform for the cloud era, 2018. Available at https://www.purestorage.com/uk/products.html.

[280] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM Computer Communication Review*, volume 43:4, pages 27–38. ACM, 2013.

[281] Z. Qin, G. Denker, C. Giannelli, P. Bellavista, and N. Venkatasubramanian. A software defined networking architecture for the internet-of-things. In *Network operations and management symposium (NOMS)*, pages 1–9. IEEE, 2014.

[282] Z. Qin, N. Do, G. Denker, and N. Venkatasubramanian. Software-defined cyber-physical multinetworks. In *Computing, Networking and Communications (ICNC), International Conference on*, pages 322–326. IEEE, 2014.

[283] Z. Qin, L. Iannario, C. Giannelli, P. Bellavista, G. Denker, and N. Venkatasubramanian. Mina: A reflective middleware for managing dynamic multinetwork environments. In *Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE, 2014.

[284] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proceedings of the 11th Workshop on Hot Topics in Networks (HotNets)*, pages 43–48. ACM, 2012.

[285] O. J. Reichman, M. B. Jones, and M. P. Schildhauer. Challenges and opportunities of open data in ecology. *Science*, 331(6018):703–705, 2011.

[286] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20. Springer-Verlag New York, Inc., 2007.

[287] D. Richman. Amazon Web Services' secret weapon: Its custom-made hardware and network, 2017. Available at https://www.geekwire.com/2017/amazon-web-services-secret-weapon-custom-made-hardware-network/.

[288] C. Rossenhövel, C. Price, and I. Váncsa. The Interoperability Challenge in Telecom and NFV Environments. Technical report, The Linux Foundation, 2017. Available at http://superuser.openstack.org/articles/interoperability-telecom-nfv-tutorial/.

[289] L. Rupprecht. Network-aware big data processing. In *Ph.D. Thesis*. Imperial College London, 2017.

[290] S. Sagiroglu and D. Sinanc. Big data: A review. In *Collaboration Technologies and Systems (CTS), International Conference on*, pages 42–47. IEEE, 2013.

[291] S. Sahhaf, W. Tavernier, J. Czentye, B. Sonkoly, P. Sköldström, D. Jocha, and J. Garay. Scalable architecture for service function chain orchestration. In *Software Defined Networks (EWSDN), Fourth European Workshop on*, pages 19–24. IEEE, 2015.

[292] P. Saint-Andre. Extensible messaging and presence protocol (XMPP): Core, 2011.

[293] K. Sampigethaya and R. Poovendran. Cyber-physical system framework for future aircraft and air traffic control. In *Aerospace Conference*, pages 1–9. IEEE, 2012.

[294] SAP. SAP Data Services, 2018. Available at https://www.sap.com/products/data-services.html.

[295] C. V. Saradhi, M. Gurusarny, and L. Zhou. Differentiated qos for survivable wdm optical networks. *IEEE Communications Magazine*, 42(5):S8–14, 2004.

[296] B. Sayadi, M. Gramaglia, V. Friderikos, D. von Hugo, P. Arnold, M.-L. Alberi-Morel, M. A. Puente, V. Sciancalepore, I. Digon, and M. R. Crippa. SDN for 5G Mobile Networks: NORMA perspective. In *International Conference on Cognitive Radio Oriented Wireless Networks*, pages 741–753. Springer, 2016.

[297] Scality. Scality is Storage for Digital Business, 2018. Available at https://www.scality.com/.

[298] J. Schönwälder, M. Björklund, and P. Shafer. Network configuration management using NETCONF and YANG. *IEEE Communications Magazine*, 48(9):166–173, 2010.

[299] M. Scurrell. Batch computing at a fraction of the price, 2017. Available at https://azure.microsoft.com/en-us/blog/announcing-public-preview-of-azure-batch-low-priority-vms/.

[300] O. Sefraoui, M. Aissaoui, and M. Eleuldj. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[301] A. Sgambelluri, F. Tusa, M. Gharbaoui, E. Maini, L. Toka, J. Perez, F. Paolucci, B. Martini, W. Poe, J. M. Hernandes, et al. Orchestration of network services across multiple operators: The 5G exchange prototype. In *Networks and Communications (EuCNC), 2017 European Conference on*, pages 1–5. IEEE, 2017.

[302] H. Sheikh. HPE Hyper Converged, 2016. Available at https://tdhpe.techdata.eu/Documents/SWEDEN/Server

[303] R. Sherwood. The Promise of the Software Defined Data Center: Abstraction, Hyper-convergence, and Dramatically Increased Business Agility , 2016. Available at http://www.bigswitch.com/webinar/the-promise-of-the-software-defined-data-center-abstraction-hyper-convergence-and.

[304] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[305] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[306] J.-Y. Shin, B. Wong, and E. G. Sirer. Small-world datacenters. In *Proceedings of the 2nd Symposium on Cloud Computing*, page 2. ACM, 2011.

[307] H. Shinohara. Broadband access in japan: Rapidly growing ftth market. *IEEE Communications Magazine*, 43(9):72–78, 2005.

[308] K. Siozios, D. Soudris, and E. Kosmatopoulos. *Cyber-Physical Systems: Decision Making Mechanisms and Applications*. River Publishers, 2017.

[309] B. Snyder, D. Bosnanac, and R. Davies. *ActiveMQ in action*, volume 47. Manning Greenwich Conn., 2011.

[310] J. Son, A. V. Dastjerdi, R. N. Calheiros, X. Ji, Y. Yoon, and R. Buyya. CloudSimSDN: Modeling and simulation of software-defined cloud data centers. In *Cluster, Cloud and Grid Computing (CCGrid), 15th International Symposium on*, pages 475–484. IEEE, 2015.

[311] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso. Multi-domain service orchestration over networks and clouds: A unified approach. *ACM SIGCOMM Computer Communication Review*, 45(4):377–378, 2015.

[312] Spark. Spark Framework: An Expressive Web Framework for Kotlin and Java, 2018. Available at http://sparkjava.com/.

[313] D. Spinellis. Don't Install Software by Hand. *Software, IEEE*, 29(4):86–87, 2012.

[314] R. Srinivasan. RPC: Remote procedure call protocol specification version 2, 1995.

[315] A. Stanik, M. Koerner, and L. Lymberopoulos. SLA-driven Federated Cloud Networking: Quality of Service for Cloud-based Software Defined Networks. *Procedia Computer Science*, 34:655–660, 2014.

[316] V. Stantchev and C. Schröpfer. Negotiating and enforcing qos and slas in grid and cloud computing. In *International Conference on Grid and Pervasive Computing*, pages 25–35. Springer, 2009.

[317] T. Szydło, P. Suder, and J. Bibro. Message-oriented communication for IPv6-enabled pervasive devices. *Computer Science*, 14(4):667–667, 2013.

[318] T. Taleb, B. Mada, M.-I. Corici, A. Nakao, and H. Flinck. PERMIT: Network slicing for personalized 5G mobile telecommunications. *IEEE Communications Magazine*, 55(5):88–93, 2017.

[319] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: scientific workflows for grids*. Springer Publishing Company, Incorporated, 2014.

[320] M. Taylor and S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, Inc., 2014.

[321] Teridion. Teridion, 2017. Available at https://www.teridion.com/.

[322] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[323] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth Symposium on Operating Systems Principles (SOSP)*, pages 182–196. ACM, 2013.

[324] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[325] A. Tootoonchian and Y. Ganjali. HyperFlow: A distributed control plane for OpenFlow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX, 2010.

[326] M. Treiber and S. Dustdar. Active web service registries. *IEEE Internet Computing*, 11(5), 2007.

[327] R. Trivisonno, I. Vaishnavi, R. Guerzoni, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani. Virtual links mapping in future sdn-enabled networks. In *SDN for Future Networks and Services (SDN4FNS)*, pages 1–5. IEEE, 2013.

[328] R. Turk. Red Hat Stroage: Why Software-Defined Storage Matters, 2016. Available at https://www.redhat.com/en/about/videos/why-software-defined-storage-matters.

[329] I. Van Beijnum. *BGP: Building reliable networks with the Border Gateway Protocol*. O'Reilly Media, Inc., 2002.

[330] P. Vassiliadis. A survey of Extract–transform–Load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[331] A. M. Vegni, M. Biagi, and R. Cusani. Smart vehicles, technologies and main applications in vehicular ad hoc networks. In *Vehicular Technologies-Deployment and Applications*. InTech, 2013.

[332] P. Velho and A. Legrand. Accuracy study and improvement of network simulation in the SimGrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[333] R. Vilalta, A. Mayoral, D. Pubill, R. Casellas, R. Martínez, J. Serra, C. Verikoukis, and R. Muñoz. End-to-End SDN orchestration of IoT services using an SDN/NFV-enabled edge node. In *Optical Fiber Communication Conference*, pages W2A–42. Optical Society of America, 2016.

[334] M. Villari, M. Fazio, S. Dustdar, O. Rana, L. Chen, and R. Ranjan. Software Defined Membrane: Policy-Driven Edge and Internet of Things Security. *IEEE Cloud Computing*, 4(4):92–99, 2017.

[335] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[336] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6), 2006.

[337] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: Wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450. USENIX Association, 2016.

[338] Voxility. The secure infrastructure for your amazing Cloud Service, 2019. Available at https://www.voxility.com/.

[339] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the ninth conference on Emerging networking experiments and technologies*, pages 283–294. ACM, 2013.

[340] vXchnge. vXchnge Markets, 2018. Available at http://www.vxchnge.com/markets/.

[341] M. Walfish, J. Stribling, M. N. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, volume 4, pages 15–15, 2004.

[342] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos. Software-defined industrial Internet of Things in the context of Industry 4.0. *IEEE Sensors Journal*, 16(20):7373–7380, 2016.

[343] J. Wan, D. Zhang, S. Zhao, L. Yang, and J. Lloret. Context-aware vehicular cyber-physical systems with cloud support: architecture, challenges, and solutions. *IEEE Communications Magazine*, 52(8):106–113, 2014.

[344] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang. Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 620–634. ACM, 2017.

[345] F. Wang, J. Li, and H. Homayounfar. A space efficient XML DOM parser. *Data & Knowledge Engineering*, 60(1):185–207, 2007.

[346] F.-Y. Wang, L. Yang, X. Cheng, S. Han, and J. Yang. Network softwarization and parallel networks: beyond software-defined networks. *IEEE network*, 30(4):60–65, 2016.

[347] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739–750, 2013.

[348] R. Wang, D. Butnariu, J. Rexford, et al. OpenFlow-Based Server Load Balancing Gone Wild. *Hot-ICE*, 11:12–17, 2011.

[349] S. Wang, A. Zhou, F. Yang, and R. N. Chang. Towards network-aware service composition in the cloud. *IEEE Transactions on Cloud Computing*, 1:1–14, 2016.

[350] P. Wette, M. Draxler, and A. Schwabe. MaxiNet: Distributed Emulation of Software-Defined Networks. In *Networking Conference, IFIP*, pages 1–9. IEEE, 2014.

[351] M. Wichtlhuber, R. Reinecke, and D. Hausheer. An SDN-based CDN/ISP collaboration architecture for managing high-volume flows. *IEEE Transactions on Network and Service Management*, 12(1):48–60, 2015.

[352] H. Widmann and H. Thiemann. EUDAT B2FIND: A Cross-Discipline Metadata Service and Discovery Portal. In *EGU General Assembly Conference Abstracts*, volume 18, page 8562, 2016.

[353] J. W. Williams, K. S. Aggour, J. Interrante, J. McHugh, and E. Pool. Bridging high velocity and high volume industrial big data through distributed in-memory storage & analytics. In *Big Data, International Conference on*, pages 932–941. IEEE, 2014.

[354] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[355] Y. Xing, G. Li, Z. Wang, B. Feng, Z. Song, and C. Wu. GTZ: a fast compression and cloud transmission tool optimized for FASTQ files. *BMC bioinformatics*, 18(16):549, 2017.

[356] R. Xiong, Y. Du, J. Jin, and J. Luo. HaDaap: A hotness-aware data placement strategy for improving storage efficiency in heterogeneous Hadoop clusters. *Concurrency and Computation: Practice and Experience*, 30(20):e4830, 2018.

[357] X. Xu, Q. Z. Sheng, L.-J. Zhang, Y. Fan, and S. Dustdar. From big data to big service. *Computer*, 48(7):80–83, 2015.

[358] X. Xu, L. Zhu, Y. Liu, and M. Staples. Resource-oriented architecture for business processes. In *Software Engineering Conference, APSEC. 15th Asia-Pacific*, pages 395–402. IEEE, 2008.

[359] M. Yao, P. Zhang, Y. Li, J. Hu, C. Lin, and X. Y. Li. Cutting your cloud computing cost for deadline-constrained batch jobs. In *Web Services (ICWS), International Conference on*, pages 337–344. IEEE, 2014.

[360] Y. Yiakoumis, K.-K. Yap, S. Katti, G. Parulkar, and N. McKeown. Slicing home networks. In *Proceedings of the 2nd SIGCOMM workshop on Home networks*, pages 1–6. ACM, 2011.

[361] S. Yousefi, M. S. Mousavi, and M. Fathy. Vehicular ad hoc networks (VANETs): challenges and perspectives. In *ITS Telecommunications Proceedings, 2006 6th International Conference on*, pages 761–766. IEEE, 2006.

[362] Z. Yu, M. Li, X. Yang, and X. Li. Palantir: Reseizing network proximity in large-scale distributed computing frameworks using SDN. In *Cloud Computing (CLOUD), 7th International Conference on*, pages 440–447. IEEE, 2014.

[363] M. Zafer, Y. Song, and K.-W. Lee. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *Cloud Computing (CLOUD), 5th International Conference on*, pages 75–82. IEEE, 2012.

[364] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th conference on Networked Systems Design and Implementation (NSDI)*, pages 2–2. USENIX, 2012.

[365] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. A. Lee, and J. Kubiatowicz. The Cloud is Not Enough: Saving IoT from the Cloud. In *HotCloud*, 2015.

[366] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[367] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong. Firework: Big data sharing and processing in collaborative edge environment. In *Fourth Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 20–25. IEEE, 2016.

[368] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. Dynamic service placement in geographically distributed clouds. *IEEE Journal on Selected Areas in Communications*, 31(12):762–772, 2013.

[369] Y. Zhang. *Network Function Virtualization: Concepts and Applicability in 5G Networks*. John Wiley & Sons, 2017.

[370] L. Zheng, C. Joe-Wong, J. Chen, C. G. Brinton, C. W. Tan, and M. Chiang. Economic viability of a virtual ISP. In *INFOCOM*. IEEE, 2017.