Loquat: A Framework for Large-Scale Actor Communication on Edge Networks

Christopher S. Meiklejohn, Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: {christopher.meiklejohn, peter.vanroy}@uclouvain.be

***Abstract*—We provide a lightweight decentralized publish-subscribe framework for supporting large-scale actor communication on edge networks. Our framework, called *Loquat*, does not depend on any reliable central nodes (e.g., data centers), provides reliability in the face of massive node failures and network partitioning, and provides scalability as the number of nodes increases. We consider that high reliability, i.e., that send operations reach close to 100% of live destination nodes, is a critical property for communication frameworks on edge networks. But reliability is difficult to achieve in a scalable way on edge networks because of the network's dynamicity, i.e., frequent node failures and partitioning. For example, both Internet of Things networks and mobile phone networks consist of devices that are often offline. To achieve reliability, our framework is based on two hybrid gossip algorithms, namely HyParView and Plumtree. Hybrid gossip algorithms combine gossip with other distributed algorithms to achieve both efficiency and high resilience. Our current implementation is written in Erlang and has demonstrated scalability up to 1024 nodes in Amazon's cloud computing environment.**

I. Introduction

Edge networks are growing in importance, as exemplified by the rapid growth of the number of connected devices in the Internet of Things (IoT) (50 billion by 2020 according to [13]). But the majority of today's Internet applications do the bulk of their computing at the logical center of the network, i.e., in data centers or other large nodes. Our goal is to provide a communication framework as a prerequisite for doing computations directly at the edge. A major problem that such a framework faces is the high dynamicity of an edge network, i.e., frequent node failures and network partitioning. The framework introduced in this paper is designed specifically to provide reliable FIFO communication at large scales on highly dynamic networks.

Actor-based systems define computations as a set of actors, each of which does a single sequential computation, and that communicate through asynchronous message passing. In this paper we single out three systems, SD Erlang [8], Akka Cluster [1], and Orleans [7], [5], as representing the state of the art of actor-based systems. All three of these systems are decentralized, i.e., they have no inherent dependency on the logical center of the network. However, as the number of nodes in the system increases, all three of the above-mentioned systems break down. Three abilities needed for distributed computing are membership (knowing the set of all nodes in the system), failure detection (knowing which nodes have failed), and name-based routing (the name of a destination actor is sufficient to route the message). All three systems break down in one or more of these abilities when the number of nodes increases in a network subject to massive failures.

Loquat provides a decentralized publish/subscribe framework that maintains all three abilities as its scale increases. Actors do not communicate directly, but indirectly through *topics*: they send to a topic and receive from a topic. Communication through a topic is implemented by a broadcast tree that is repaired by a gossip algorithm. The broadcast tree provides efficiency and the gossip algorithm provides resilience in the face of failure. The combination of an efficient tree-based algorithm and a resilient gossip algorithm is an example of *hybrid gossip*. It is well-known that gossip algorithms can be extremely resilient. However, they are often inefficient. Hybrid gossip algorithms combine the resilience of gossip algorithms with the efficiency of other distributed algorithms. Loquat is built on top of two hybrid gossip algorithms, namely HyParView for membership and failure detection [19] and Plumtree for broadcast [18]. In this way, Loquat is able to provide all three abilities as the number of nodes in the system increases.

A. Contributions

This paper has three main contributions:

- *Survey of large-scale actor systems*: We survey state-of-the-art actor systems and analyze how well they support building large-scale distributed systems on dynamic networks.
- *Loquat publish/subscribe protocol*: We define the Loquat protocol for lightweight publish/subscribe that is decentralized, scalable, reliable, and works well on dynamic networks. In particular, because Loquat is built on top of hybrid gossip algorithms, it supports dynamic networks with massive failures and frequent offline operation such as commonly occur in edge networks.
- *Loquat publish/subscribe programming model*: We define the Loquat programming model, which generalizes the Erlang programming model. Whereas in Erlang, actors communicate directly with each other, in Loquat, actors communicate through *topics*. This decouples the actors in similar fashion to coordination languages. The programming model defines an API that smoothly extends Erlang's actor communication API.

II. Survey of Large-Scale Actor Systems

We present a survey of existing actor systems and analyze their support for distributed programming at large scales.

A. SD Erlang

SD Erlang (Scalable Distributed Erlang) conservatively extends the distribution support in Erlang [10], [23]. Chechina *et al.* [8] identify two main problems with the scalability of SD Erlang: (1) quadratic space growth in management of the global process registry, a global naming service for associating names with process identifiers, through transitive connection sharing and full replication, and (2) explicit placement, or how to determine where actors should be spawned as cluster size grows. The authors propose two solutions to solve the identified problems:

- *Reducing transitive connection sharing:* By subdividing nodes into smaller groups and only supporting full connectivity within each group and not across groups, nodes limit the number of nodes that they have to connect to, perform failure detection on, and replicate the global process registry of. In this model, each node can become a member of multiple groups and can explicitly request a connection with another node in the system, without transitive connection sharing.
- *Semi-explicit process placement:* When spawning a new process, per-node attributes can be used to filter the list of available nodes to choose from for hosting that process. This allows application developers to target nodes by available memory, CPU size, or other user-defined attributes.

While these changes enable SD Erlang to break through the 100 node scalability bottleneck previously identified by the authors [15], these solutions still assume that explicit process naming through the global registry is desirable, from an application developer point of view. Additionally, a node that participates in too many process groups also will fall into the same trap of replicating too much information. Both techniques are unscalable.

B. Akka Cluster

Akka Cluster provides a clustering facility for the Akka actor system [1], which is heavily inspired by the Erlang actor model, however implemented on top of the JVM in Scala. Cluster membership in Akka is performed through a gossip-based membership service [11], similar to the gossip-based membership service implemented by the Riak Core distributed systems framework [17] for Distributed Erlang.

Akka has demonstrated scalability up to 2400 nodes, deployed slowly over the course of 4 hours to ensure membership convergence. Akka additionally provides a routing facility based on node-level attributes for routing to actors or nodes that can perform a particular task: these roles are statically assigned at deployment time.

The nodes in Akka Cluster establish themselves in a ring and failure detection is performed by deterministically selecting f nodes adjacent on the ring to transmit heartbeat messages [16] to every interval. The f parameter is user-specified, so the cluster does not attempt to maintain an average time to detect failures, and the user must configure f appropriately for ideal coverage of the cluster. This structure works well for low failure rates but is not designed for resilience at high failure rates.

C. Orleans

Microsoft's Orleans [5] is another actor system, but through the use of an abstraction called "virtual actors", removes the need for explicit placement, explicit binding, and the manual setup and teardown of actors. Instead, Orleans automatically launches what are referred to as *grain activations* which are actors that are registered into a one-hop distributed hash table (DHT). When a message is sent to one of these actors, they are created if they do not exist, and they are shut down when they are no longer needed.

Virtual actors can be transparently migrated, as the one-hop DHT provides a directory service for all of the currently running activations of a grain. Virtual actors can come as stateless workers, where multiple activations can exist, or as single activation workers, where only a single activation is allowed. Each Orleans silo, or server where grain activations are executing, maintains a single TCP connection to other nodes in the cluster, and a similar model is used to Akka to perform failure detection: membership information is hashed into a virtual ring and peers periodically heartbeat a fixed amount of their neighbors on the ring. Orleans provides no FIFO ordering on messages, with at-least-once messaging through timeout/retries, and requires a persistent backing store for storing membership information about who is currently participating in the cluster.

The membership information requires that the entire DHT be available to service requests. Therefore, this information is heavily cached on each of the silos. Compared to Distributed Erlang, the one-hop DHT is partitioned across the node of the system with membership stored in a central persistent store whereas Distributed Erlang fully replicates this information to every node in the cluster. Orleans has been run at 200 nodes with low failure rates.

III. Hybrid Gossip

Hybrid gossip is a recent development in gossip algorithms where a gossip protocol is combined with another distributed algorithm, in order to achieve the good qualities of both. Typically, a gossip protocol will be highly resilient but less efficient, and the other distributed algorithm will be very efficient but less resilient than the gossip protocol. The hybrid algorithm is as resilient as the first and as efficient as the second. Loquat takes advantage of two hybrid gossip protocols, namely HyParView and Plumtree. To understand Loquat, it is important to understand the principles of these two algorithms.

A. HyParView

HyParView [19] is a hybrid gossip algorithm that provides a resilient membership protocol by using partial views to provide global system connectivity in a scalable way. Using partial views ensures scalability; however since each node only sees part of the system, it is possible that failures of other nodes break connectivity or greatly increase routing length. To overcome these problems, HyParView uses two different partial views that are maintained with different strategies.

The challenge is to ensure that the combination of all partial views at all nodes form a single connected component. To achieve this, HyParView maintains at each node both an active view and a passive view. There is a single TCP connection from each node to each member of its active view. This connection is used for both data dissemination and failure detection. No connections are maintained for the members of the passive view.[1]

The active and passive views are managed using different strategies. The active view contains $fanout + 1$ members and is used to flood information through the network and perform failure detection. The active view is managed using a reactive strategy: when a node in the active view is suspected to have failed, it is replaced by a node in the passive view. The passive view has a maximum of $\log(n)$ members (where n is the number of nodes in the system) and is used to maintain a backup set of members used to replace failing members in the active view. The passive view is managed using a proactive strategy: periodically each node performs a shuffle operation between its passive view and the passive view of a neighbor in the active view.

B. *Plumtree*

Plumtree [18] is a hybrid gossip algorithm that provides reliable broadcast by combining a deterministic tree-based broadcast protocol with a gossip protocol. The tree-based protocol constructs and uses a spanning tree to achieve efficient broadcast. However, it is not resilient to node failures. The gossip protocol is able to repair the tree when node failures occur. Thus the Plumtree protocol combines the efficiency of spanning trees with the resilience of gossip.

The Plumtree protocol is implemented as two phases. Given a message with a unique identifier, first push the identifier and payload to the nodes at the leaves of the broadcast tree. This is known as the *eager push* phase. Second, push only the message identifier to a random sampling of other nodes known by a standard gossip-based peer service.[2] This is known as the *lazy push* phase. If a node does not receive a message it has learned about through the lazy push phase within a designated timeout period, then it requests the message from a randomly selected peer.

Plumtree starts with a random sampling of nodes selected from the peer service. As the execution continues, nodes are removed from this set as duplicate messages are received. This prunes the set and computes the spanning tree that will be used for the eager phase of message broadcast.

[1]In practice, connections are maintained to a random subset of the passive view to speed up repair of the active view.

[2]In our case, the Loquat membership protocol, based on HyParView, presented in Section IV.

IV. Loquat

Loquat is a communication framework that generalizes Erlang's actor framework: actors communicate through topics instead of directly. An actor sends a message to a topic, and an actor can receive messages from a topic by subscribing to that topic. By using HyParView and Plumtree, Loquat's communication reliability stays close to 100% even for massive node failures in the network. Loquat's base protocol provides reliable broadcast with weak ordering. By *weak ordering* we mean that the order of message delivery is close to the send order, and all differences can be explained due to the varying latencies of message routing. To achieve the node-to-node FIFO ordering of Erlang's model, we extend the base protocol with per-node monotonic counters to ensure FIFO delivery.

We explain separately the Loquat protocol, its programming model, and the current status of our implementation.

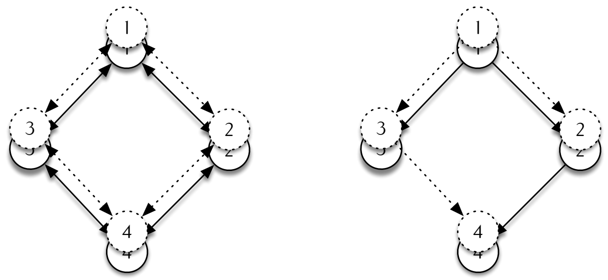

Figure 1: Superposition of topic-specific overlay graphs (left) and trees (right), rooted at node 1, that are constructed on top of the metadata tree. Overlay graphs are seeded using the membership information in the metadata tree and optimized by the broadcast tree protocol.

A. *Protocol Design*

The Loquat protocol combines three ideas: efficient membership, efficient broadcast, and multiple spanning trees.

1) Efficient membership: Efficient membership provides guaranteed connectivity with high probability despite partial knowledge at each node and frequent node failures. This protocol is based on HyParView [19], which ensures reliable communication even in the presence of massive network failures. HyParView is designed with two partial views at each node: a small active view of open connections and a larger passive view of known nodes but without open connections. The passive view contains candidates to replace failed nodes in the active view. Evaluation of HyParView shows that this offers high resilience to massive failures: compared to competing protocols Scamp and Cyclon, HyParView maintains higher reliability (where *reliability* is defined as the fraction of nodes actually reached by a send operation to a topic compared to the nodes that are alive and subscribed to the topic): close to 100% even for failure of up to 95% of nodes [25], [14].

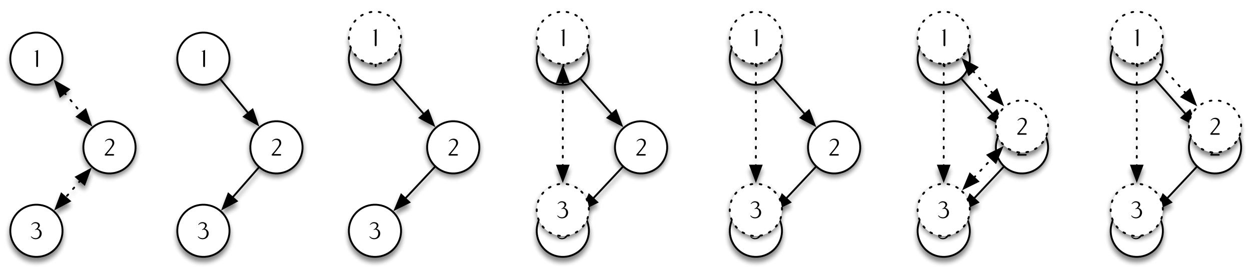

Figure 2: Construction of a tree for a particular topic, starting with metadata tree construction (left) to optimization of a topic-specific broadcast tree (right).

2) Efficient broadcast: Efficient broadcast, the second idea, leverages a spanning tree to reduce redundant communication between the participating nodes. When the network changes because of dynamicity, parts of the spanning tree may become defunct. Plumtree uses gossip to repair these parts, while keeping the spanning tree where the network is stable. This approach combines both efficiency and resilience: efficiency is generally very high and it will decrease locally and temporarily only for parts of the network that are unstable. We use a known extension to the Plumtree protocol to ensure reliable delivery, which is not guaranteed by the original protocol. In Plumtree, messages may be permanently lost if node failures exceed 70%. We therefore leverage a periodic anti-entropy process that will select nodes for pairwise reconciliation of dropped messages [26].

3) Multiple spanning trees: To support multiple topics, we construct multiple overlapping spanning trees, namely one spanning tree per topic. There is one spanning tree that covers all nodes, which we call the metadata tree. The metadata tree is used for topic management, for example to create new topics. Figure 1 shows an example of a metadata tree with superimposed topic-specific spanning trees (left) and their optimized versions (right) after being optimized by the broadcast protocol. Figure 2 shows the construction of a topic-specific tree from the gossip network: (1) an overlay network is computed, (2) the metadata tree is constructed, (3) node 1 creates an initial graph for topic X, (4) node 3 joins the graph, (5) the graph is optimized into a broadcast tree, (6) node 2 joins the graph, and (7) the graph is optimized into a broadcast tree and the redundant edge from node 3 to node 2 is removed.

In the general case when each topic is *dense*, i.e., covers a significant fraction of the system's nodes, one spanning tree per topic is not scalable [4]. However, if the number of nodes subscribed to each topic follows a power law similar to the popularity of Web pages (where *popularity* of a page is defined as the fraction of pages with k in-links, which on the Web is observed to be proportional to $1/k^2$), then the total size of all spanning trees for the whole system is proportional to the number of nodes, which ensures scalability. We are confident that some such law will hold, since it holds for almost all real-world network structures [12]. For the occasional network that has different properties, there exist several optimizations to improve scalability (see Section V).

B. Programming Model

Loquat is an upward-compatible extension of the Erlang actor framework. Loquat runs on a set of compute nodes, each of which hosts a set of actor processes. The usual Erlang primitives for creating and using process IDs remain valid and are not affected by Loquat. We provide a set of new primitives for Loquat to handle creation and deletion of topics and sending and receiving from topics. Loquat introduces topic identifiers, which are new atoms[3] in Erlang that identify topics, and topic epochs, which are nonnegative integers. The management of topic identifiers and epochs is not done by Loquat but is left to higher layers of the system.

- Create a new topic and its identifier:
  `spawn_topic(TopicId, Epoch)`
  Topic creation is an explicit operation, just like Erlang's process creation. If the topic already exists, this is equivalent to subscribing to a topic. To facilitate resource management of the topic's spanning tree, each topic is given an epoch number.
- Remove a topic from the system:
  `kill_topic(TopicId, Epoch)`
  This operation will force all subscribed actors to eventually unsubscribe from the topic. This is important for resource management since a topic with zero subscribers has a zero size spanning tree. In case of concurrent spawn and kill operations, the highest epoch number will win.

The use of epoch numbers is the mechanism that allows a system built on top of Loquat to manage the resource usage of a topic.

- Send a message to a topic:
  `TopicId ! Message`

[3]An atom in Erlang is an immutable symbolic value.

As in Erlang, sends are asynchronous and FIFO-ordered between sending and receiving actors. Sending a message to a topic with zero subscribers is a null operation.

- Subscribe to a topic and receive a message from the topic:
  ```
  receive(TopicId)
     [same as Erlang receive]
  end
  ```
  Subscribing to a topic is an idempotent operation. Each subscriber receives all messages sent to the topic.
- Unsubscribe from a topic:
  ```
  unsubscribe(TopicId)
  ```
  The receiving actor unsubscribes from a topic, which means it no longer receives messages from the topic from that point onwards. If zero actors on a given node are subscribed to the topic then the node is removed from the topic's spanning tree.

C. Implementation Status

Our work on Loquat is part of a larger project on general-purpose edge computing that started in the SyncFree EU project [3] and is continuing in the LightKone EU project [2]. In previous work, we have designed and implemented Lasp, a programming language for large-scale synchronization-free programming [21], [6], on top of Selective Hearing, a run-time system that uses Plumtree for communication [22], [20].

Our current Loquat prototype, written in Erlang, provides the communication layer for Lasp's runtime system. It facilitates internode communication, on a single topic, using a single spanning tree, and has demonstrated scalability up to 1024 nodes in Amazon's cloud computing environment while preserving FIFO ordering of messages and reliable delivery in the face of high node churn. Our plans are to extract the current implementation from the Lasp runtime system and build a standalone Loquat prototype that supports the full programming model with support for multiple topics. This will allow us to evaluate Loquat as a full replacement for large-scale actor programming with Erlang, eschewing the traditional approach using Distributed Erlang. At that point, we will perform quantitative evaluations at scale.

Extensions to HyParView for high churn: The extremely high resilience of HyParView is essential for the correct operation of Loquat. The HyParView protocol defined in [19] was originally implemented and evaluated on the PeerSim simulator. Our real (non-simulated) environment has much higher churn than this simulated environment, in part because we deploy our system very quickly to reduce cost. We have made three changes to the original HyParView protocol in order to take high churn into account:

- *Isolation prevention*: When bootstrapping the cluster quickly, a node can trivially be isolated if there are too many joins in succession. This happens because a node accepting a join will drop a random node from its active view to make space for the joining node. This causes isolation if churn is very high, which is not handled by the original protocol. We remedy this situation by adding a mechanism to enforce a minimum allowed size for the active view.
- *Epochs to ensure FIFO*: The original protocol assumes FIFO delivery of messages between any two nodes in the cluster and uses TCP to guarantee this. However, at high churn a node may be connected and disconnected from the same node in rapid succession, which gives multiple TCP connections, and messages may arrive out of order. We remedy this situation by implementing a counter-based epoch (similar to Apache Cassandra) that ensures FIFO is guaranteed across connections.
- *Reactive shuffle*: The original protocol periodically does a shuffle between the passive view of a node and the passive view of one of the neighbors in its active view. At high churn this proactive shuffle is not sufficient: we extended the protocol to do a reactive shuffle between passive views whenever the node updates the active view.

We also made a fourth change to improve scalability:

- *Active view management*: The original protocol manages the active view using a reactive strategy, when it suspects that a node in the active view has failed. To keep the active view as full as possible, we supplement this with a proactive strategy that periodically attempts to promote nodes from the passive view to the active view.

With these changes, our implementation of HyParView maintains all the good properties of [19] and in addition is able to handle extremely high churn. In future work we will make a quantitative evaluation of these changes.

V. Related Work

We compare Loquat to other approaches to achieve scalable publish/subscribe. The main problem tackled in the other approaches is how to achieve scalability in the presence of a large number of dense topics. For Loquat on the other hand, we focus on reliability in the presence of massive node failures. Compared to the systems mentioned below, Loquat's use of HyParView gives it an increased reliability.

The publish-subscribe model for distributed computing was originally introduced through the virtual synchrony concept. It initially tried to deliver reliable atomic broadcast, which is difficult to provide on large-scale edge networks with high churn and network partitioning. In subsequent developments this guarantee was weakened to causal order.

More recently, approaches for maintaining scalable publish-subscribe systems have investigated two opposing approaches: either a gossip-based approach which is very resilient but leads to the delivery of messages to subscribers about topics that are not relevant (parasite messages) or maintaining a spanning tree per topic, which does not exhibit the problem of parasite messages, but relies on maintaining state for each topic's tree and is prone to problems under network partitioning.

Neither approach is scalable as the number of dense topics increases, and several proposals have been introduced to support numerous dense topics. The daMulticast protocol [4] improves scalability by modifying the gossip approach to organize topics in hierarchical fashion. The approach of [24] improves scalability by organizing subscribers according to their QoS requirements for delay and bandwidth. The SpiderCast protocol [9] combines partial views with a coverage-optimizing heuristic to construct and maintain an overlay network where each topic (specifically, the induced subgraph for the topic) remains, with high probability, connected under both churn and dynamic membership to solve both the parasite message and connection maintenance problems. The hierarchical and QoS-aware approaches are both complementary to Loquat and could be combined with it. This is still unclear for SpiderCast, because Loquat relies on reliable broadcast.

VI. Conclusion

This paper presents the design of Loquat, a lightweight publish/subscribe framework for large-scale actor communication. In Loquat, actor processes communicate indirectly through topics instead of directly through actor names. Loquat is built using hybrid gossip algorithms HyParView and Plumtree and is designed specifically to have extremely high resilience, as part of its requirement to run on edge networks. Loquat is part of a long-term project to build a platform for general-purpose edge computation using synchronization-free programming techniques, which is mainly being done in the EU projects SyncFree [3] and LightKone [2]. Both HyParView, which implements a membership protocol, and Plumtree, which implements reliable broadcast, use hybrid gossip techniques to achieve extremely high resilience. HyParView is able to reach close to 100% of remaining nodes even in the face of failure of up to 95% of nodes.

We have built a prototype implementation of Loquat that demonstrates scalability up to 1024 nodes in Amazon's cloud computing environment. We present a programming model (including API) for this implementation and we discuss the problems we had to overcome to achieve scalability. Specifically, we had to modify the HyParView protocol to support high churn. In future work we will perform a quantitative evaluation of Loquat's resilience at scale.

Acknowledgements

We would like to thank Vitor Enes Duarte and Junghun Yoo for their help with the initial implementation of our prototype and Manuel Bravo for his comments on the paper. The research leading to these results has been partially funded by the SyncFree Project [3] in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n^{o} 609551. Christopher S. Meiklejohn is partially supported by the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030.

References

[1] Akka Cluster. http://akka.io. Accessed: 2016-11-09.

[2] LightKone: Lightweight computation for networks at the edge. European H2020 project to start in January 2017.

[3] SyncFree: Large-scale computation without synchronisation. https://syncfree.lip6.fr. European FP7 project 2013–2016.

[4] S. Baehni, P. T. Eugster, and R. Guerraoui. Data-aware multicast. In *International Conference on Dependable Systems and Networks*, 2004.

[5] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical report, March 2014.

[6] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: Distributed Deterministic Dataflow Programming for Erlang. In *Proceedings of the 13th ACM SIGPLAN Erlang Workshop*. ACM, 2014.

[7] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[8] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding. The Design of Scalable Distributed Erlang. In *Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK*, 2012.

[9] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. SpiderCast: A Scalable Interest-Aware Overlay for Topic-Based Pub/Sub Communication. In *International Conference on Distributed Event-Based Systems (DEBS '07)*. ACM, 2007.

[10] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proceedings of the 2005 ACM SIGPLAN Erlang Workshop*, pages 78–87. ACM, 2005.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[12] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010.

[13] D. Evans. The Internet of Things: How the next evolution of the Internet is changing everything. *Cisco IBSG White Paper*, Apr. 2011.

[14] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, 2003.

[15] A. Ghaffari. Investigating the scalability limits of distributed Erlang. In *Proceedings of the 13th ACM SIGPLAN Erlang Workshop*, pages 43–49. ACM, 2014.

[16] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 66–78. IEEE, 2004.

[17] R. Klophaus. Riak Core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[18] J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 301–310. IEEE, 2007.

[19] J. Leitao, J. Pereira, and L. Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429. IEEE, 2007.

[20] C. Meiklejohn, S. H. Haeri, and P. Van Roy. Declarative, Sliding Window Aggregations for Computations at the Edge. In *First International Workshop on Edge Computing (EdgeCom 2016)*, Jan. 2016.

[21] C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195. ACM, 2015.

[22] C. Meiklejohn and P. Van Roy. Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In *Workshop on Planetary-Scale Distributed Systems (W-PSDS 2015)*, 2015.

[23] H. Svensson and L.-A. Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the 2007 ACM SIGPLAN Erlang Workshop*, pages 43–54, 2007.

[24] M. A. Tariq, G. G. Koch, B. Koldehofe, I. Khan, and K. Rothermel. Dynamic publish/subscribe to meet subscriber-defined delay and bandwidth constraints. pages 458–470, 2010.

[25] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[26] J. West. Controlled Epidemics: Riak's New Gossip Protocol and Metadata Store. https://www.youtube.com/watch?v=s4cCUTPU8GI. Accessed: 2016-02-06.