UCLouvain

PREDICTABLE NETWORK SOLUTIONS

INPUT | OUTPUT

The ΔQSD Paradigm for System Development

June 22, 2022
HiPEAC Tutorial

Peter Van Roy
Université catholique de Louvain

Neil Davies, Peter Thompson
Predictable Network Solutions Ltd.

Seyed Hossein Haeri
PLWorkz

1

1

ΔQSD

- ΔQSD is an industrial-strength paradigm for system design that can predict performance and feasibility early on in the design process
  - Developed over 30 years by a small group of people around Predictable Network Solutions Ltd.
  - Widely used and validated in large industrial projects, with large cumulative savings in project costs
- ΔQSD properties
  - Compositional approach that considers performance and failure as first-class citizens
  - Captures uncertainty throughout the design process

2

2

Goals of the tutorial

- Understand the basic principles of the ΔQSD paradigm for system design
- Understand the two main concepts of ΔQSD, quality attenuation (ΔQ) and outcome diagram
- Understand the main principles of system design with ΔQSD using refinement and performance computation
- Understand how to determine infeasibility
- Give the concepts so you can start using ΔQSD in your own designs

3

3

Organization of the tutorial

- Table of contents
  - Case studies: Small cells, iPhone launch, Cardano Shelley
  - Quality attenuation (ΔQ)
  - Outcome diagrams
  - Shelley block diffusion algorithm
  - Some typical ΔQs
  - Extras (work in progress): systems with iterative queries, slacks and hazards, shared resources, blame assignment (diagnosis), limitations
  - Conclusions
- Caveat
  - I am not the inventor of ΔQSD. I am a computer scientist with long experience in distributed systems and programming languages. I created this tutorial as part of learning ΔQSD, because I consider ΔQSD to be a interesting and innovative approach that should be more widely known.
  - This tutorial is still a work in progress and is incomplete: please send me your errata and constructive comments

4

4

Two main concepts

- Quality attenuation (ΔQ)
  - A ΔQ is a cumulative distribution function that defines both the delay and failure probability between a start event and an end event
  - Because the ΔQ combines delay and failure in a single quantity, it makes it easy to examine trade-offs between them
- Outcome diagram
  - An outcome is any well-defined system behaviour with observable start and end events; each outcome has a ΔQ
  - An outcome diagram is a causal directed graph that defines the relationships between all system outcomes; it allows computing ΔQ for the whole system
  - The outcome diagram can be used from the beginning to the end of the design process. It can express partially defined systems and it can be refined from an initial unknown design up to the final, constructed system.

5

5

1. Case Studies

6

6

Case studies

- As motivation for ΔQSD we present three case studies
  - Small cells
  - iPhone launch
  - Cardano Shelley
- These are industrial case studies done by PNSol that have limited documentation and are partially covered by NDA
- In these scenarios, the ΔQSD paradigm is used in two ways
  - Diagnostic use: debugging of existing systems with problems
  - Design use: designing systems from the start
- But we encourage the use of ΔQSD for design!
  - This is one of the motivations of this tutorial: to disseminate the ΔQSD paradigm so it can be used during the design process
  - Prevention is much better than cure!

7

7

Small cells case study

8

8

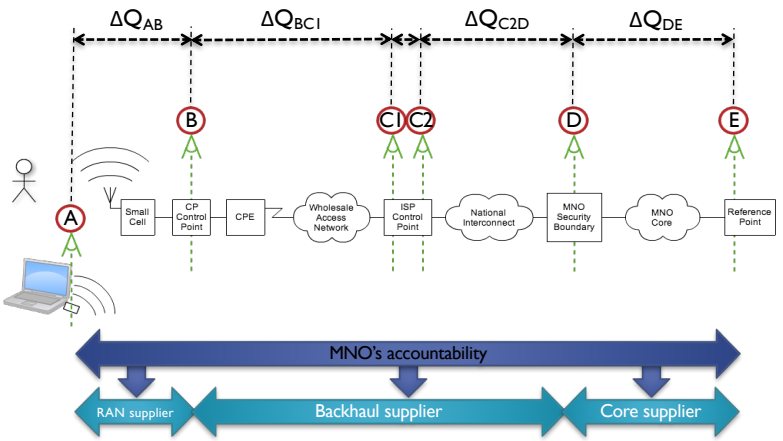
Small cells case study

- A major MNO (Mobile Network Operator), who shall remain unnamed, deployed small cells
  - Small cell: low-powered cellular radio access nodes with range 10m-3km
  - Backhaul using consumer DSL broadband
- The system worked but did not scale
  - Voice quality had major problems, cells were failing
  - What part of the system is the cause and who is to blame?
- PNSol was brought in to investigate
  - Determined outcome diagram for complete system
  - Measured ΔQ across system to pinpoint the problem
  - Focus on problematic behavior shown by ΔQ
  - ΔQSD led to successful diagnosis and cure proposal

9

9

Who is to blame for my system crashing?



MNO (erroneously) believed that: (1) its contracts would deliver the service & contain the hazards; and (2) there were no residual hazards.

10

10

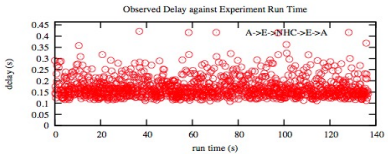
How PNSol gathered the evidence

- **Establish end to end measurement**
  - From synthetic traffic generator… (**A**)
    - includes an observer
  - …to reference point (**E**)
    - reflects traffic, acts as a protocol peer, and includes an observer
  - Add internal observers to get spatial discernment (**B**, **C**, **D**)
- **Analyse measurements to obtain ΔQ distributions**
  - Outcome diagram **A** → **B** → **C1** → **C2** → **D** → **E**
  - Measure quality attenuation ΔQ for outcomes
  - Identify issues and anomalies for further investigation
- **Each observation point *doubles* the spatial fidelity**
  - Example: even with just **A** and **E** there is definitive knowledge as to whether the effect is occurring upstream or downstream.

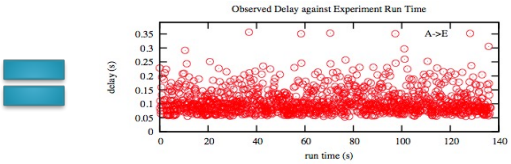
11

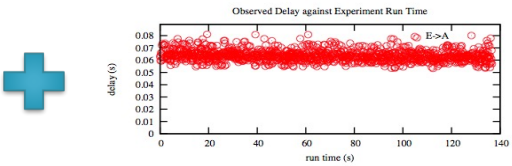
11

Which direction has issues?



Decompose round trip time variability





Mean upstream variability is 5 times greater than downstream; tail variation is x10

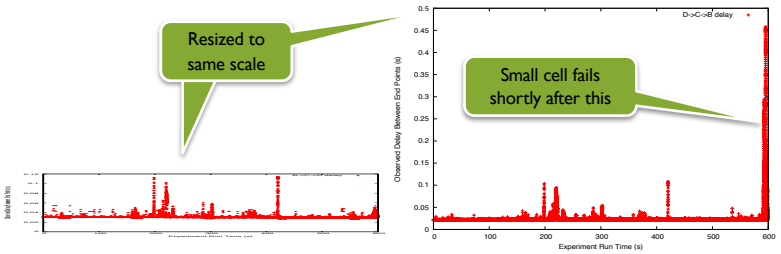
Problem in upstream? No, actually not!

12

12

Who is to blame for the system failing?

Examine sub-paths to isolate the issue



Upstream Downstream

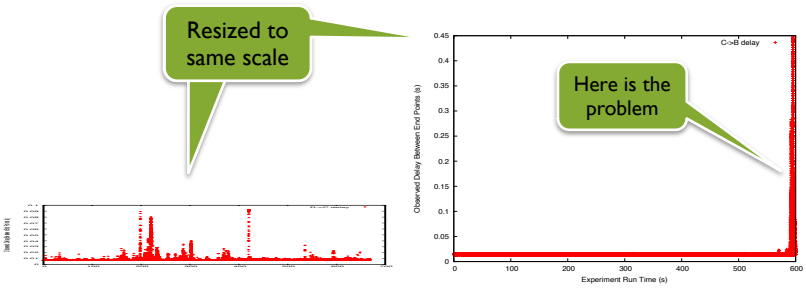
13

13

Where is the issue?

Use spatial resolution to isolate the problem

National Interconnect Wholesale Access Core

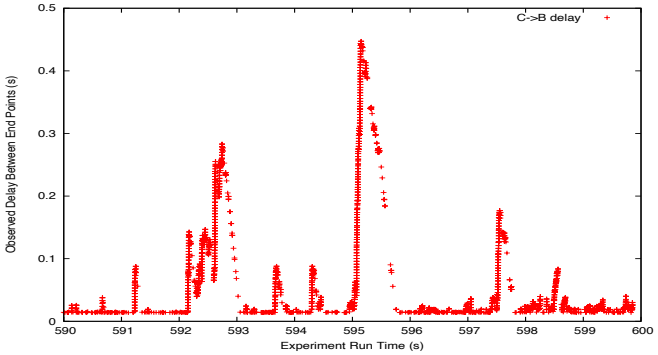


14

14

Zoom in on the issue

Expand temporal resolution to examine the problem



Typical queue overload pattern:
get into 'trouble' very quickly, get out of it far more slowly
Temporary overloads have long-lasting effects!

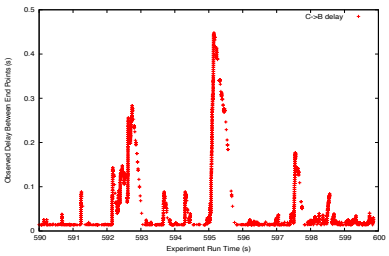
⇒ Later in the tutorial we will study queues to understand this

15

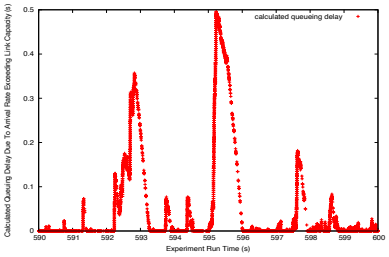
15

Actual + predicted measures

Use predictability of ΔQ to check the conclusion



Measured delay
in access network



Calculated delay
(from mathematical model)
due to arrival pattern of traffic
exiting MNO security gateway

16

Technical diagnosis

- **A queue is forming in the wholesale access network**
  - This is because the arrival rate from the MNO security boundary exceeds the sync rate (service capacity) of the xDSL line
  - The queue exhibits temporary overloading, which degrades overall behaviour for long time periods
  - This is in breach of the wholesaler's technical terms & conditions
- This queue delays ***all*** traffic, including small cell control traffic
  - Small cells are known to fail if their control loops exceed a given round trip time. The figures here are 5x that limit.
- System reset is just the extreme failure case
  - Delays of that magnitude adversely effect voice quality as well
  - Causes small cells to "breathe" inappropriately
  - **Dramatically weakens deployment business case**

17

Systemic diagnosis and cure

- Why is the system crashing?
  - There is an **unmanaged hazard** that sits with the MNO
- Root cause is that the subsystems don't compose
  - The pre-requisites for use of one element are not met by other elements of the system
    - This is a common structural problem, not unique to this MNO or technology
  - They believed that they only had to match bandwidths
    - They should be matching ΔQ (Quality Transport Agreements)
- **Recommendations to the MNO:**
  - **Note on corporate risk register: records the risks and opportunities that may affect the delivery of the Corporate Plan**
  - **Technical training to improve contractual processes & hazard management**

18

iPhone launch case study

19

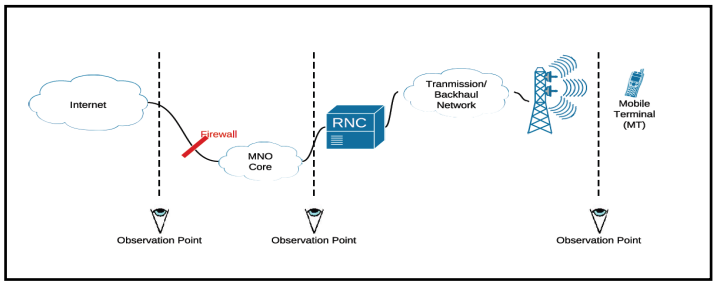
iPhone launch case study

- The iPhone was initially supported in UK by one MNO
- A second MNO prepared to enter this market
  - Before the launch, the performance was known to be bad for the second MNO, and the first MNO had gleefully prepared a major ad campaign focusing on this fact
    - Both MNOs are large UK operators who will remain unnamed
  - Using ΔQSD, PNSol managed to diagnose and correct the problem just before the launch
    - Thus saving the bacon of the second MNO
  - Result was a 100% improvement in http download KPI, which placed the second MNO in first place
    - To the great embarrassment of the first MNO

20

Diagnosis approach and solution

- For data collection, observation points were placed at the RNC (Radio Network Controller) and around the network edges

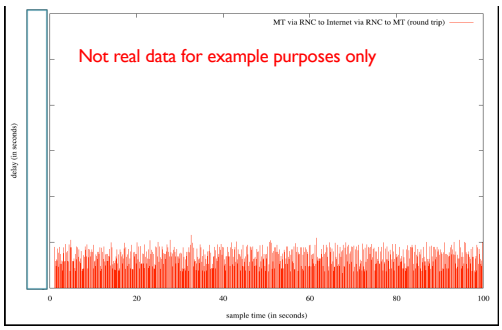


- The ΔQSD paradigm was used for the diagnosis
  - Determine outcome diagram for end to end delivery of packets and measuring ΔQ for intermediate points
  - Isolate cause and effect to pinpoint the problem, finding where loss and delay are introduced in an unexpected pattern
  - Ultimately, to find solutions

21

Packet delivery behaviour

The RTT (Round-Trip Time) during the first 100 seconds



Here we observed a RTT delay introduced for each packet in a sample low-rate stream over the entire path during the first 100 seconds of the data collection

This sample did not show any unexpected behaviour in the network in terms of loss and delay during this period;

However …

22

Packet delivery behaviour

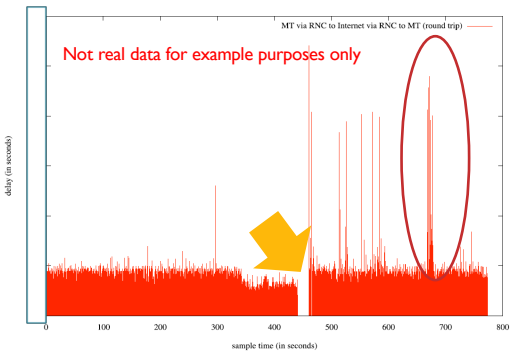
The RTT for the full duration of the data collection

With the full sample time at almost 800 seconds we observed unexpected behaviour;

- Service break occurred
- Excessive delays of up to 1s

This directly correlates to a bad experience being delivered to end users

- And delivering quality is about making bad experiences rare



The next step is to divide the paths (MT, RNC, Internet) into sections and deal with the issues in a focused way…

23

Packet delivery behaviour

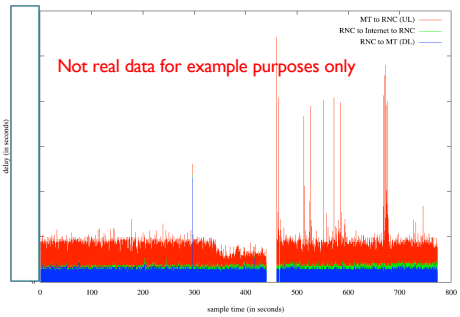
Combined observations split by element

Improvements typically are focused on getting the best from the down link (DL) RNC to MT….

But as can be seen from the **BLUE** on the chart (RNC to MT DL) we only observed a single outlier during the total sample time

For the full round trip across the core to the internet and back shown in **GREEN** we again observed no real issues

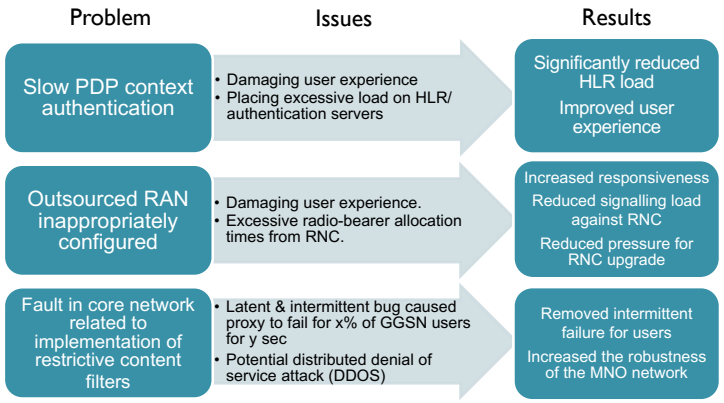
The MNO suspected the RNC DL was the major trouble spot. As can be seen with the **RED** (MT to RNC UL) we found it was really on the UL: this is where the service break occurred.



Observing the end-to-end behaviour of packet flows enables the true cause of issues to be identified and corrected

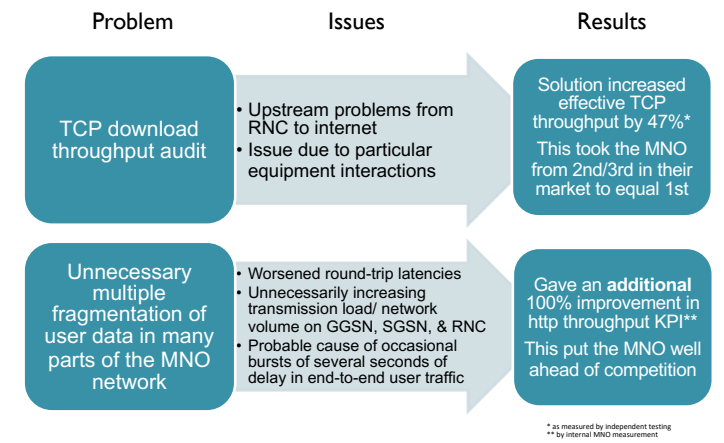
24

iPhone launch findings using ΔQSD (1 of 2)



25

iPhone launch findings using ΔQSD (2 of 2)



26

Cardano Shelley case study

27

Cardano Shelley case study

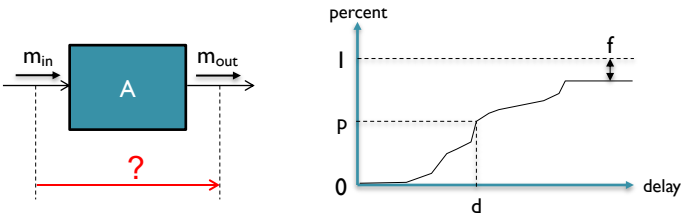
- The previous case studies used ΔQSD for diagnosis
  - PNSol was brought in to diagnose problems in running systems
- Cardano Shelley used ΔQSD in the design process
  - Design is the preferred way to use ΔQSD
- Cardano Shelley is part of the Cardano blockchain, supporting the Ada cryptocurrency
  - An important part of Cardano is block diffusion, to allow whichever node is authorized to create a block to add it to the most recently created block
  - The initial implementation, Jormungandr, did not achieve sufficient performance
  - A further implementation, Shelley, was done using ΔQSD to guide the design from early on, and achieved adequate performance in a decentralised environment
  - We will give the Shelley example later on in the tutorial, as soon as we have introduced the necessary concepts

28

2. Quality Attenuation (ΔQ)

29

Quality attenuation (ΔQ)



- Message m_{in} enters component A and m_{out} exits
- How do we characterize the message traveling through A?
  - The delay between entry and exit: delay value (a number)
  - The message might be dropped: chance of failure (a percentage)
  - The delay is not always the same for all messages: jitter
- We combine all this into a single quantity ΔQ
  - p percent of messages have delay ≤ d and f percent of messages fail
  - Delay and failure are not considered separately
  - This helps to examine trade-offs delay/failure in the same design

30

Combining delay and failure

- Delay and failure are combined in one quantity ΔQ
  - Two parts of system design that are usually separate are considered together
  - This allows to easily examine trade-offs between delay and failure in the design
- Performance and fault tolerance should not be separate
  - They are two sides of the same coin
  - For example, failure can be reduced by increasing delay, which is all part of one ΔQ
    - By changing the maximum delay threshold: increasing delay tolerance will reduce the percentage of messages that are considered failed
    - By retrying: failure can be made arbitrarily small by increasing delay
    - Both of these techniques are captured by the ΔQ quantity

31

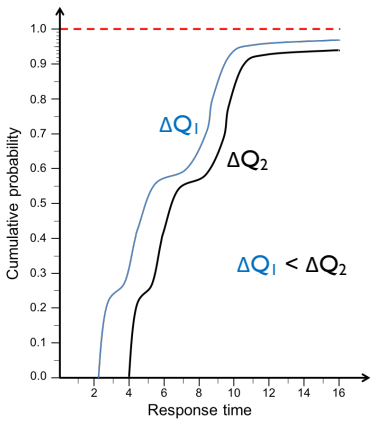
31

Partial order of ΔQ comparison

If we compare the CDFs of two ΔQs, then one is *less than* the other if its CDF is everywhere to the left and above the other

- Mathematically, this relation between two ΔQs is a *partial order*
- If the ΔQs intersect then they are not ordered

This provides a criterion for 'good enough' performance



32

Example designs with ΔQ

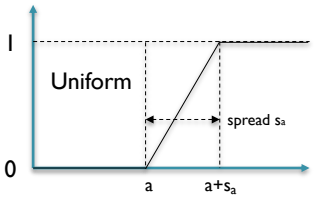
33

Example designs with ΔQ

- We can use ΔQ to help design a system
- Let's start with a simple system that is just a connection of two components
  - We will show both a top-down and a bottom-up design
    - In both cases, we determine the behavior of a new component
  - We will determine when the top-down design is infeasible: when there is no possible way to build it (because a component must have negative delay and/or negative loss!)
- We will use a simple ΔQ in these examples, namely a Uniform distribution
  - This is a reasonable approximation for components, but of course many other ΔQs occur in practice!
  - We will "add" and "subtract" ΔQs in the examples, note that technically this is convolution and deconvolution

34

Uniform distribution



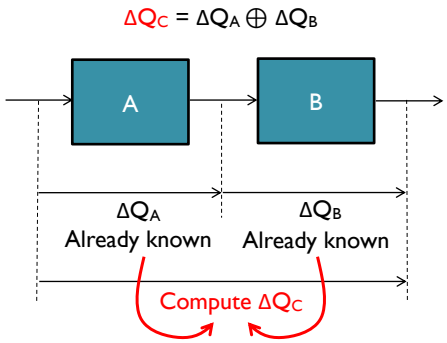
- A Uniform distribution approximates a component with buffer and server
  - a is the minimum time in the component
  - s_a is the spread of times in the component
  - $a+s_a$ is the maximum time in the component

- For our two examples, we use a Uniform distribution for ΔQ
  - It is one of the simplest distributions and it is useful in practice: many components have approximately a uniform distribution
  - Uniform distributions are good for "back-of-the-envelope" ΔQ computations; an automated tool can of course compute with the full, detailed ΔQ
- In this short tutorial, we will do back-of-the-envelope computations
  - It is easy to extend this and do the full computations

35

35

Bottom-up design with ΔQ

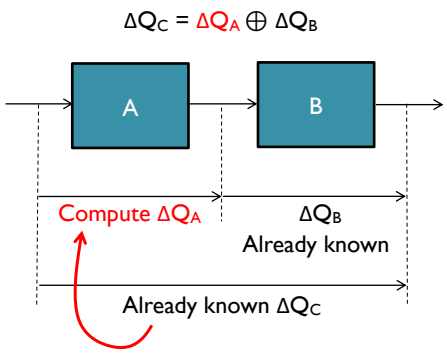


- We know component A has ΔQ_A and component B has ΔQ_B
  - What is ΔQ_C?
- We assume Uniform distributions for A and B and "add" them to get C:
  - The three components are: (a,s_a) , (b,s_b) , (c,s_c)
  - Then the result is: $c = a + b + m$ with $m=\min(s_a,s_b)/2$ and $s_c = \max(s_a,s_b)$
  - Overall delay c is a bit worse than the sum of the two delays
  - Overall spread s_c is determined by the worst component

36

36

Top-down design with ΔQ



- There is a global overall requirement of ΔQ_C and component B is known to have ΔQ_B
  - What ΔQ_A is needed for A?
- We assume Uniform distributions and "subtract" them:
  - $a = c - b - m$
    - A must be a little bit faster than just c–b!
    - $m=\min(s_a,s_b)/2$: if B has large spread then A is forced to have small spread
  - $s_a \le s_c$
    - A's worst possible spread is s_c

37

37

Check for infeasibility

- Let us compute when the top-down design becomes infeasible for Uniform distributions
- Component A is feasible means that $a>0$
  - Negative delay is impossible!
- $a>0$ implies $b < c - \min(s_a,s_b)/2$ (back-of-the-envelope)
  - This equation has B on both sides, namely b and s_b
  - It is mainly a constraint on b, but s_b can also play a role
  - Since $m=\min(s_a,s_b)/2$ then if s_b is large then it forces us to design A with small s_a which may or may not be possible
- If $b \ge c - m$ then the design is infeasible
  - For a given C, if B is not good enough then it is impossible to find an A with positive delay
  - For more complicated ΔQs, the infeasibility condition can be complicated but it is easily computed in software!

38

38

"Subtracting" Uniform distributions

- When doing top-down design, we do the opposite of addition
  - Mathematically, we are doing deconvolution which is much harder for hand computation than convolution
  - However, for specific distributions like Uniform it is easy
  - It is also not a problem for a tool, because even though it needs much more computation, the user does not notice
- Top-down design introduces a new subtlety: "goodness" changes direction
  - Bottom-up (addition): we compute the known behavior of a component, so decreasing s_a means it is performing better
  - Top-down (subtraction): we compute a requirement on a new component, so decreasing s_a makes it harder to satisfy

39

39

3. Outcome Diagrams

40

40

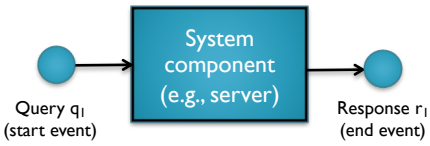
Outcome diagrams

- Now let's combine components (defined by ΔQ) into full systems (defined by outcome diagrams)
- Outcome diagrams define systems by looking at their behaviours from the outside
- They are purely observational
  - They are very different from UML diagrams
  - They say nothing about system state
- They are extremely useful
  - Many different kinds of component can be brought together, software, humans, mechanics
  - They allows estimating performance and feasibility early on in the design process

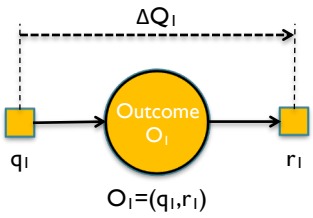
41

41

Single outcome



Component performing an outcome (in the running system)



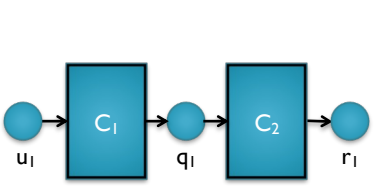
Causal connection between q_1 and r_1 (just an abstract relationship)

- An outcome O_1 is a specific system behaviour, which is defined by its start event q_1 and end event r_1
  - We don't care how the system is built, we simply observe it
  - Left figure shows the query and response messages entering and exiting a component
  - Right figure shows just the causal connection between the two events: query causes response, with quality attenuation ΔQ_1

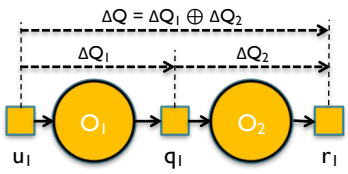
42

42

Outcome diagram



Block diagram



Outcome diagram

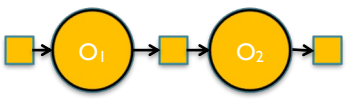
- We have a user click u_1 causing a query q_1 to be sent causing a response r_1 to be received
- An outcome diagram is a graph showing the causal connections between all the outcomes that we are interested in
  - We don't actually care (yet) how the system is constructed, we are only interested in the behaviour
  - Total ΔQ is the convolution of the individual ΔQ_1 and ΔQ_2 (all delays and failures are "added")

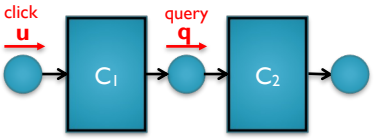
43

43

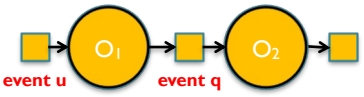
How outcome diagrams work

The outcome diagram shows the events and outcomes that we are interested in and how they are related





User click u enters C_1 and query q exits C_1 — Behaviour we are interested in (outcome)



Outcome O_1 occurs when events u and q both occur

- An outcome O_1 occurs when event u and event q both occur
  - Square boxes show where events may occur (locations in the system)
  - Circles show which outcomes can occur (behaviours we are interested in)
- New instances of O_1 can occur later when new instances of u and q occur
  - Many user clicks and queries can happen when the system is running
  - If new events u' and q' occur then a new outcome O_1' occurs

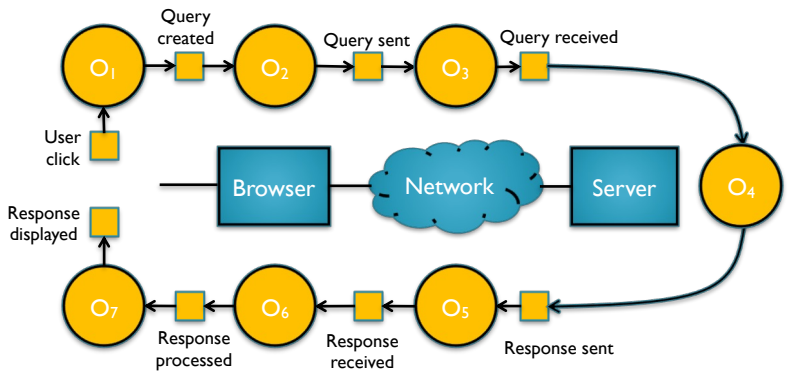
44

44

Client/server example

45

Generic RPC outcome diagram



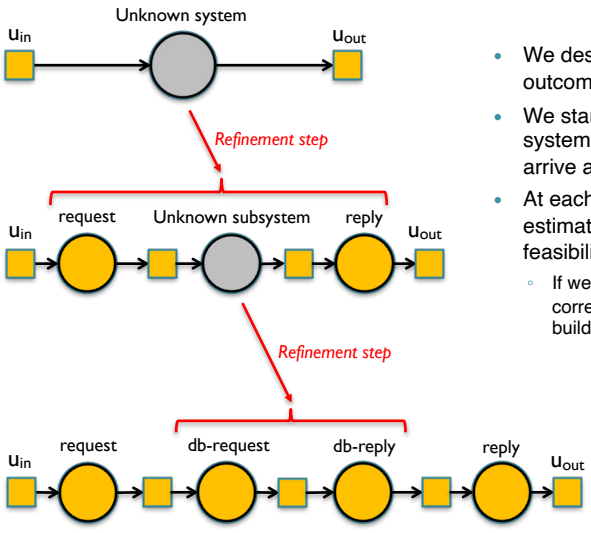
- This is a simple client/server shown as a simple outcome diagram
- Each square is an event and each circle is an outcome
- Each outcome has its own ΔQ
- Total ΔQ from user click to response displayed is addition of all ΔQs

46

General system design

47

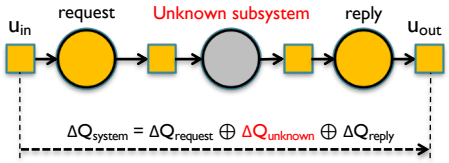
General system design



- We design the system's outcome diagram step by step
- We start from an unknown system and refine it until we arrive at the actual system
- At each step, we can compute estimated performance and feasibility
  - If we make a mistake, we can correct it before actually building the system

48

Example top-down design

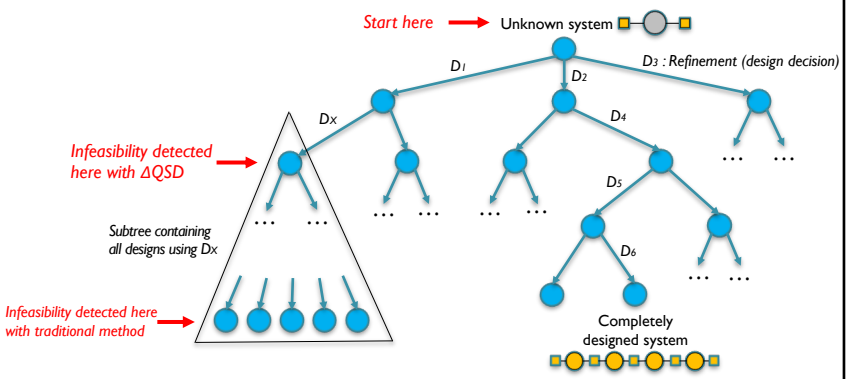


- We use a top-down design approach
  - We assume that ΔQ_{system} , $\Delta Q_{request}$, ΔQ_{reply} are all known: ΔQ_{system} is the system requirement, and $\Delta Q_{request}$ and ΔQ_{reply} have already been determined
  - We compute required $\Delta Q_{unknown}$ for the unknown subsystem to be designed
- If $\Delta Q_{unknown}$ is infeasible, then go back and change $\Delta Q_{request}$ and ΔQ_{reply}
  - If there is no way to solve the problem by changing $\Delta Q_{request}$ and ΔQ_{reply} then we need to go back even further and change the overall requirement ΔQ_{system} or change the outcome diagram (i.e., the system design)
- We navigate by going up and down the refinements until reaching a satisfactory design or until showing that no design is possible
- This gives a design tree…

49

49

Exploring the design space



- The design space is a tree of partially defined systems
  - The designer navigates the tree starting with an unknown system, making design decisions, until arriving at a completely designed system that satisfies the requirements
- The ΔQSD paradigm allows to compute infeasibility early on, even for partially defined systems

50

50

Syntax and semantics of outcome diagrams

- The ΔQSD paradigm makes it all precise
- Outcome diagrams have four primitive operators
  - Sequential composition (convolution)
  - Probabilistic choice (weighted sum)
  - Last-to-finish (all-to-finish) (arithmetic maximum)
  - First-to-finish (arithmetic minimum)
- They are defined as a formal language
  - Outcome diagrams are represented formally by outcome expressions with a semantics (blue phrases), usable by a software tool to do design steps and ΔQ computations
  - We do not define the formalisation in this talk, but it is an important part of our future work on ΔQSD

51

51

Connection to logic programming

- There is a precise correspondence between this design process and the execution of a logic program
  - Initial system requirements = query (initial formula)
  - Partially specified system = logical formula
  - Completely specified system = solution
  - Design decision = axiom choice
  - Infeasibility = unsatisfiability (failure)
- If a choice leads to failure, then the system backtracks
  - "If a design decision leads to infeasibility, then we remove it"
- Logical semantics
  - Proof theory: Designing a system is deduction in a proof system in which each design decision is an additional logical constraint
  - Model: Each partially specified system corresponds to a set of concrete systems that are coherent with that specification; each design decision restricts the set; infeasibility means an empty set

➡ A software tool for ΔQSD is a logic programming system

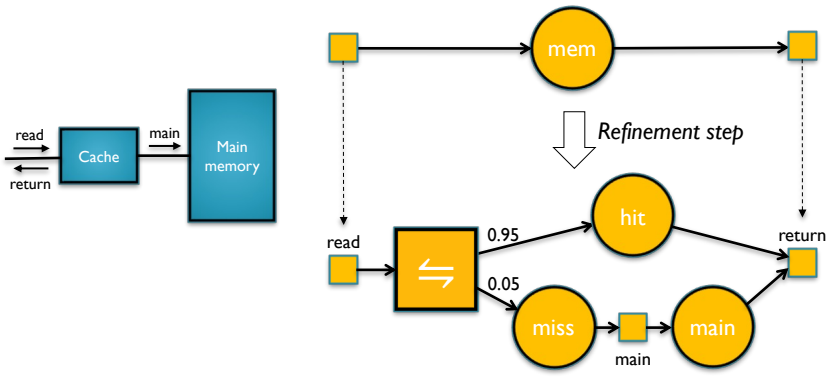
52

52

Cache memory example

53

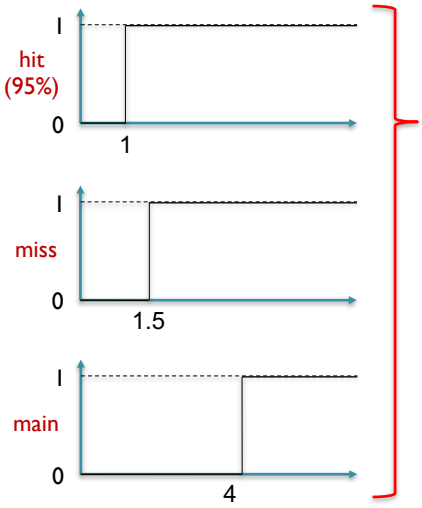
Cache memory example

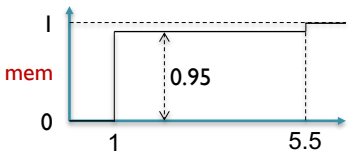


- A cache memory is modeled using probabilistic choice
- $\Delta Q_{mem} = h \cdot \Delta Q_{hit} + m \cdot (\Delta Q_{miss} \oplus \Delta Q_{main})$
- We can see the cache as one component or refine it

54

Cache quality attenuation





Combining the three ΔQs gives the cache memory's overall ΔQ_{mem}

55

55

4. Shelley Block Diffusion Algorithm

56

56

Context of block diffusion

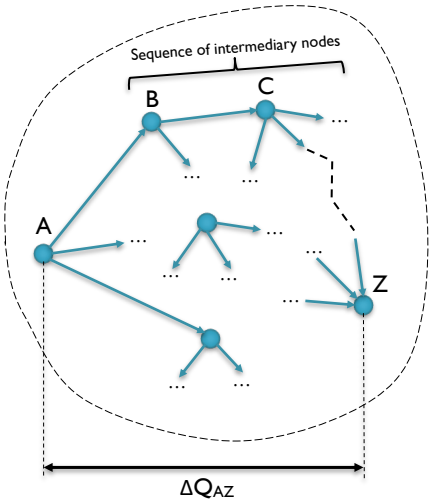
- Blockchain management in Cardano
  - We will use ΔQSD to solve a design problem in the Cardano cryptocurrency, which is implemented using a blockchain
  - A blockchain is a distributed ledger comprising a set of data blocks that are cryptographic witnesses to correctness of preceding blocks
  - A distributed consensus algorithm is used to agree on the correct sequence of blocks; Cardano uses the Ouroboros Praos consensus
  - Ouroboros Praos randomly selects a node to produce a new block during a specific time interval, weighted by distribution of stake
- Shelley block diffusion algorithm
  - The block-producing node is randomly chosen and needs a copy of the most recent block
  - Therefore this block must be copied to *all* potentially block-producing nodes in real time, which is called block diffusion
  - We will design a block diffusion algorithm using ΔQSD to ensure that the algorithm satisfies stringent timeliness constraints

57

57

Block diffusion problem statement

Node graph of Cardano blockchain



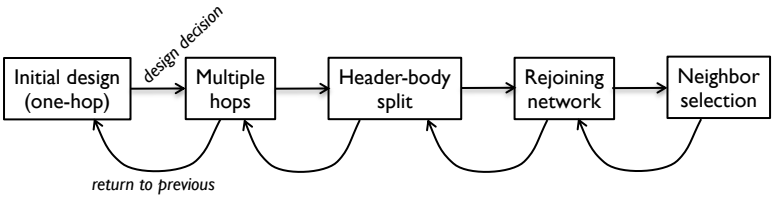
- Problem:
  - Determine ΔQ_{AZ} for randomly chosen nodes A and Z, as function of design
  - Determine design so that ΔQ_{AZ} satisfies performance constraints
  - ΔQ_{XY} is known (measured) 

- Design parameters:
  - Frequency of block production
  - Node connection graph
  - Block size
  - Block forwarding protocol
  - Block processing time

58

58

Block diffusion design using ΔQSD



- First step: preparation
  - Define an initial design and its outcome diagram
  - Measure ΔQ between two nodes
- Second step: design process
  - We make design decisions and refine the outcome diagram to take each decision into account
  - Each refinement defines a new outcome diagram and computes its ΔQ
    - At each step, we decide whether to keep the design or whether to go back to a previous design and make another design decision
  - Details given in "Mind Your Outcomes", Computers 2022, 11, 45
    - https://www.mdpi.com/2073-431X/11/3/45

59

59

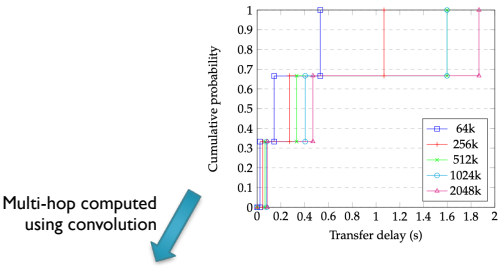
First step: measuring ΔQ

- First step is to measure ΔQ between two nodes across the Internet
  - This requires some preliminary work
- Four main factors
  - Block size: 64KB to 2048KB (5 steps)
  - Network speed: measured TCP speeds
  - Geographical distance (for single packet):
    - Short (same data centre), medium (same continent), long (different continents)
  - Network congestion: initially ignored
- Single-hop ΔQs are approximately step functions
  - Multi-hop ΔQs computed from single-hop (sequential composition operator, i.e., convolution)
  - Random path ΔQs computed from multi-hop (probabilistic choice operator, i.e., weighted sum)

60

60

Measured ΔQ for fixed paths

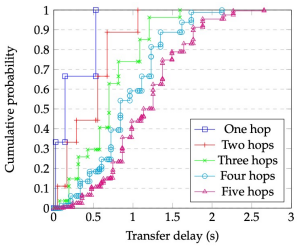


One-hop ΔQ per block size

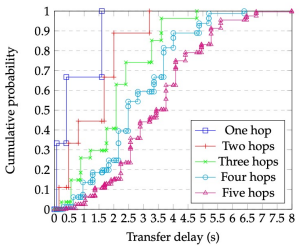
Weighted sum of short, medium, and long hops, computed using probabilistic choice operator

Multi-hop computed using convolution

Multi-hop computed using convolution



Multi-hop ΔQ for 64KB

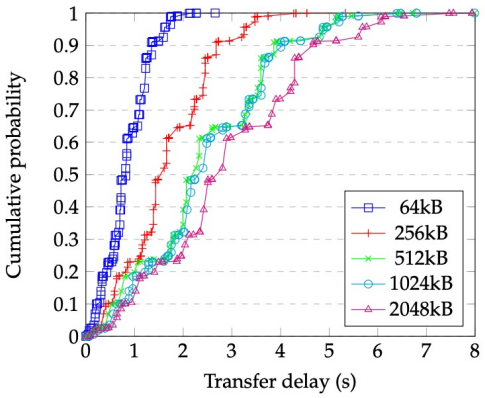


Multi-hop ΔQ for 1024KB

61

61

Measured ΔQ for varying paths

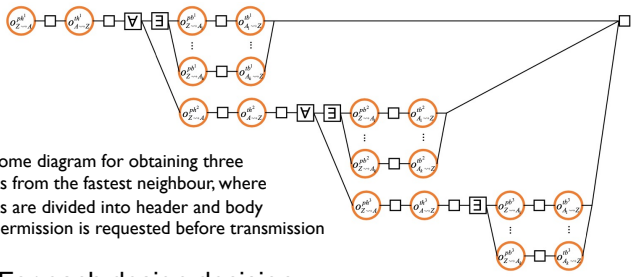


- ΔQ computed for varying path lengths
  - Percentage of paths of given length in a random graph of 2500 nodes of degree 10
  - Computed using probabilistic choice operator

62

62

Second step: design process



Outcome diagram for obtaining three blocks from the fastest neighbour, where blocks are divided into header and body and permission is requested before transmission

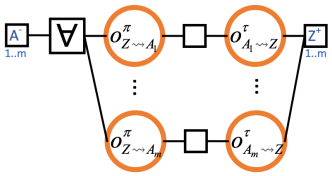
- For each design decision
  - Determine a new outcome diagram
  - Evaluate the effectiveness using the outcome diagram
- This leads step by step to a final outcome diagram, which corresponds to the complete distributed system
  - Let us explain one of the steps, namely obtaining several blocks from the fastest neighbour
  - The other steps are explained in the Computers paper

63

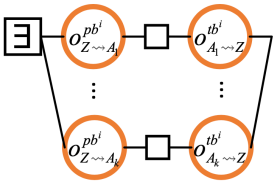
63

Obtaining three blocks (1)

All-to-finish operator



First-to-finish operator

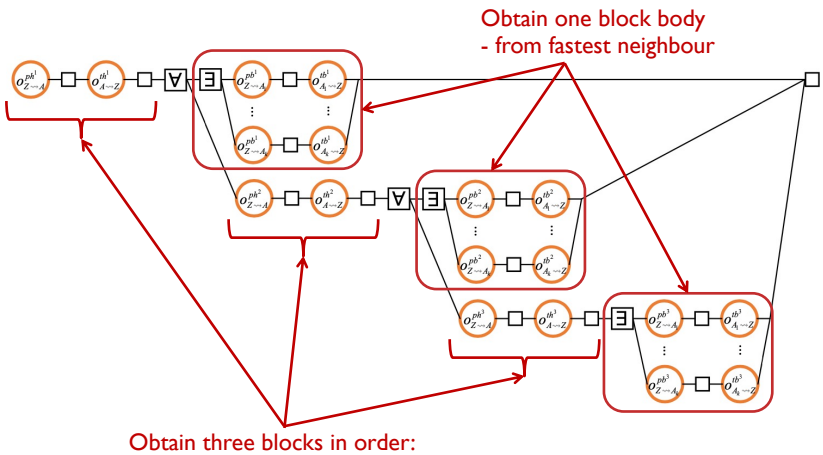


- We first explain the two operators that are needed
- Obtaining one block from each neighbour uses the all-to-finish operator (∀)
- Obtaining fastest block from one neighbour uses first-to-finish operator (∃)

64

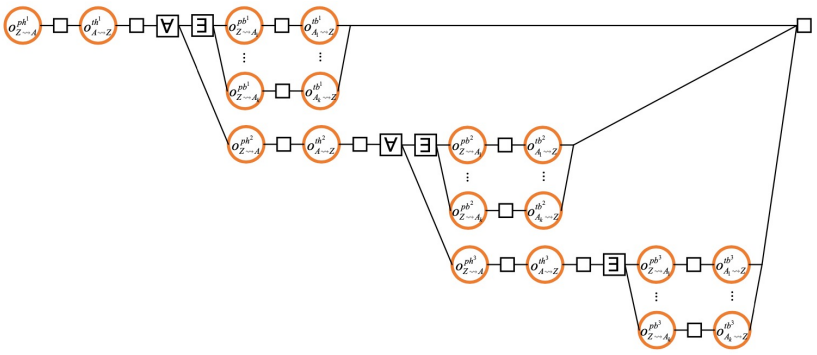
64

Obtaining three blocks (2)



65

Obtaining three blocks (3)



- The resulting outcome diagram correctly models the causality and performance of the block transfer; ΔQ is easily computed
- The outcome diagram is complex but it can be simplified by introducing abstractions
- A software tool would have no problem with it, of course

66

66

5. Some typical ΔQs

67

67

Some typical distributions

68

68

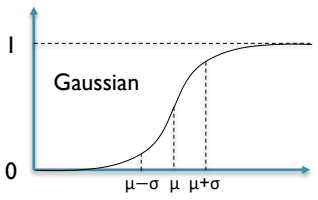
Some typical distributions

- A tool can compute arbitrarily complex ΔQs
  - There is no limitation on the complexity of the ΔQ
- But it's still important to know some typical ΔQs
  - A good engineer always knows when something is possible or impossible with back-of-the-envelope calculations
- We give theory and intuition for some common distributions
  - What to expect for typical components and networks
  - Gaussian, Uniform, exponential distributions

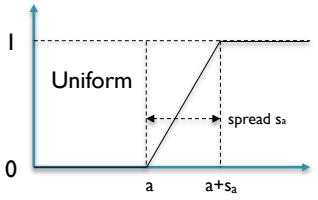
69

69

Two important distributions



- A Gaussian distribution approximates the sum of many independent random quantities
  - μ is the mean
  - σ is the standard deviation
- Gaussian is good for computing aggregates, but not for components
  - Gaussians have infinite tails!



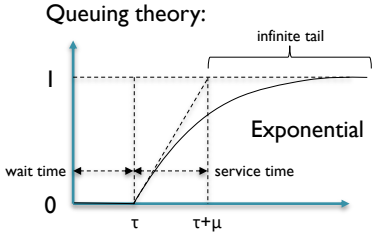
- A Uniform distribution approximates a component with buffer and server
  - a is the minimum time in the component
  - s_a is the spread of times in a component
  - $a+s_a$ is the maximum time in the component
- Uniform is good for single components, so it is used often for system design

70

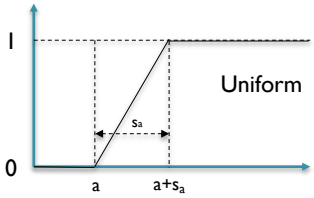
70

ΔQ of a typical component

Queuing theory:



Typical component:

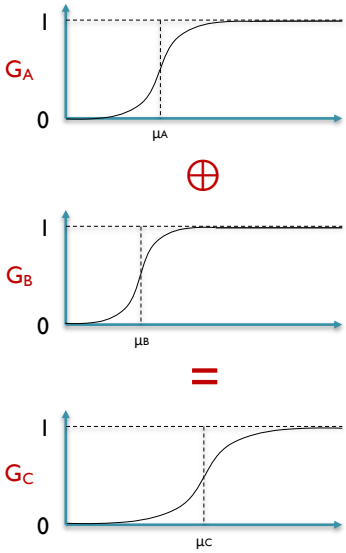


- We model a component using queuing theory: a buffer (wait time τ) followed by a server (service time μ)
  - Service time μ is modeled as an exponential distribution
  - Wait time τ is a distribution that depends on offered load
- Real components are often approximated
  - Service time has a max (no infinite tail)
  - Wait time is approximately constant
- Real components with a Uniform distribution
  - Minimum delay a = τ
  - Spread s_a = μ

71

71

Sum of Gaussian distributions

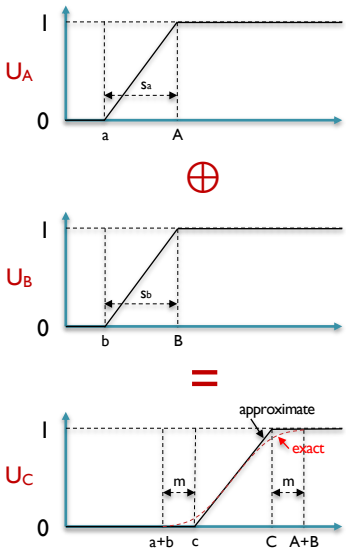


- Formulas: (exact)
  - $G_A = (\mu_A, \sigma_A)$
    $G_B = (\mu_B, \sigma_B)$
    $G_C = G_A \oplus G_B = (\mu_C, \sigma_C)$
  - $\mu_C = \mu_A + \mu_B$
    $\sigma_C^2 = \sigma_A^2 + \sigma_B^2$
  - $\sigma_C = \sqrt{\sigma_A^2 + \sigma_B^2}$
- In other words:
  - Means are added
  - Squares of standard deviations are added
- Intuitions:
  - Standard deviation increases more slowly than mean, because we're adding two independent variables
  - Mathematically, we are doing a convolution

72

72

Sum of Uniform distributions



- Formulas: (approximate)
  - $U_A = (a,s_a)$
    $U_B = (b,s_b)$
    $U_C = U_A \oplus U_B = (c,s_c)$
  - $s_c = \max(s_a , s_b)$
  - $c = a + b + m$
    where $m=\min(s_a , s_b)/2$
- In other words:
  - Starting times are added, plus a little more
  - Spread is the maximum of the spreads
- Intuitions:
  - Spread causes the delay to be a bit worse than just a simple sum
  - If there are several spreads, the biggest one will dominate

73

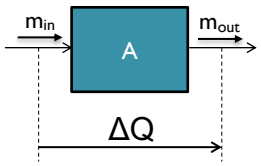
73

ΔQ for a typical component (from queuing theory)

74

74

A component as a queue



- Let's get some more intuition on how a component works
  - To get this intuition, we model the component as a queue
- A typical component has four parameters of interest
  - Offered load a: arrival rate / service rate of messages
  - Buffer size k: number of messages stored inside
  - Failure rate f: percentage of messages dropped
  - Delay d: time delay between input and output message

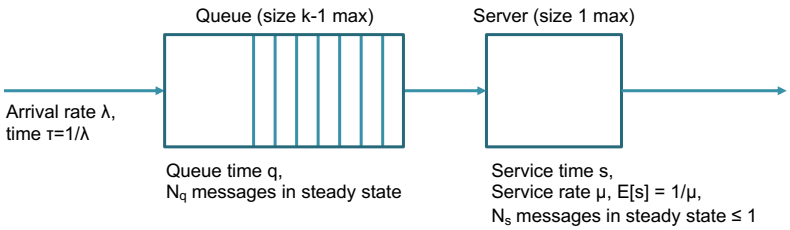
ΔQ

- These four parameters are all related
  - We use queuing theory to study these relationships

75

75

M/M/1/K queue



- We model a component as an M/M/1/K queue
  - M: arriving messages have Exponential distribution with rate λ
  - M: service time has Exponential distribution with rate μ
  - 1: one message can be served at a time
  - k: total buffer size is k (buffer size = queue size k-1 + server size 1)
- Offered load a = λ/μ (arrival rate / service rate)
- The two knobs we control are offered load a and buffer size k
  - When the component's buffer is full, new arrivals are dropped (failure)
  - ΔQ, i.e., failure rate and average delay, is function of a and k

76

76

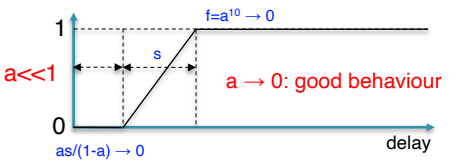
Effect of offered load a

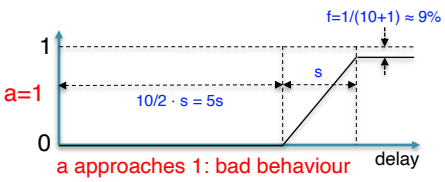
- The offered load is the most important parameter
  - a<1: the component has enough power to service all messages
  - a>1: the component is overloaded and performs very badly
- Low load (a<0.8)
  - Failure tends to 0, delay tends to 1 (as k increases)
  - An underloaded component behaves very well
- High load (a≥0.8)
  - When a gets close to 1 (around 0.8) things quickly get worse!
  - When a>>1, failure rate tends to (a-1)/a, up to 100% for high load!
  - Delay increases very quickly when a approaches 1
    - When a=1, delay is already k/2, half of buffer size, which can be huge
- Quick switchover somewhere between a=0.5 and a=1
  - As the load increases beyond 0.5, the system quickly gets very bad
  - The exact threshold depends on what you consider bad!
  - Even a temporary overload causes a big, long-lasting degradation
    - *This is the cause of the problem in the small cells case study*

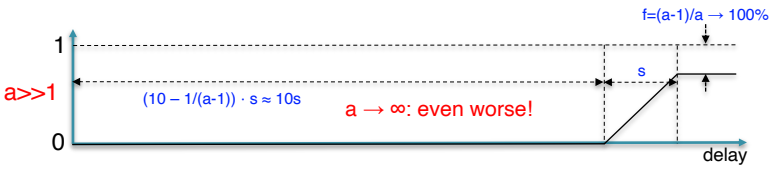
77

77

ΔQ as function of load a







- Let's visualize ΔQ as function of offered load a
- To make it understandable, we approximate the ΔQ as a Uniform distribution and we give asymptotic behaviors for three cases, a<<1, a=1, a>>1
  - We assume constant service time s and buffer size k=10
  - We simplify the complicated formulas of a M/M/1/K queue

78

78

Effect of buffer size k

- The buffer size k is the total number of messages that can be stored in a component
  - Manufacturers like to brag about buffer size. It might seem like a no-brainer that bigger is better, but this is wrong!
- We look separately at low load and high load
- Low load (a<0.8)
  - Bigger buffer decreases failures and increases delay
    - At low load, we can adjust k to trade off failure and delay
  - As k→∞ the failure rate f→0 and delay→1/(1-a) *(close to 1)*
    - Bigger buffers are good at low load
- High load (a>0.8)
  - Buffer size has less effect, since component is already bad!
  - Failure rate and delay are both high
  - Bigger buffer greatly increases delay (around k/2 for big a)
    - Bigger buffers are bad at high load
    - NICs that can store 1000 packets are especially bad when overloaded
    - With temporary overload, buffer will fill quickly, and then empty slowly
    - If you want good behaviour, don't ever overload not even temporarily!

79

79

ΔQ for a typical network

80

80

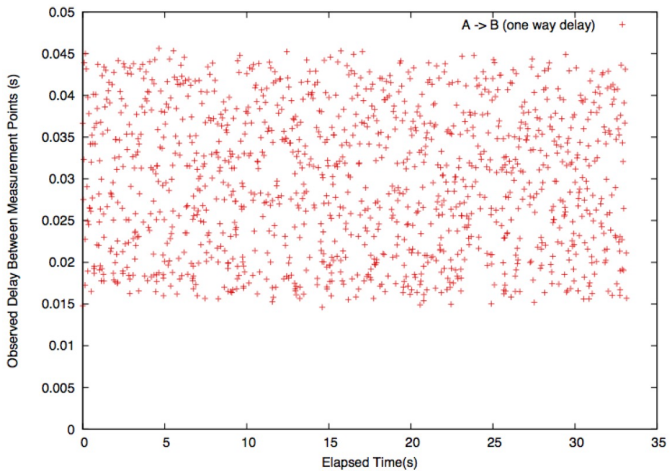
ΔQ for network packets

- For networks delivering packets the pure uniform distribution is not good enough
- Experience shows that the ΔQ has three parameters, G, S, V
  - $\Delta Q = \Delta Q_{IG} \oplus \Delta Q_{IS} \oplus \Delta Q_{IV}$
- Again, we can add and subtract ΔQs
  - Because of the simple structure, the equations are simple

81

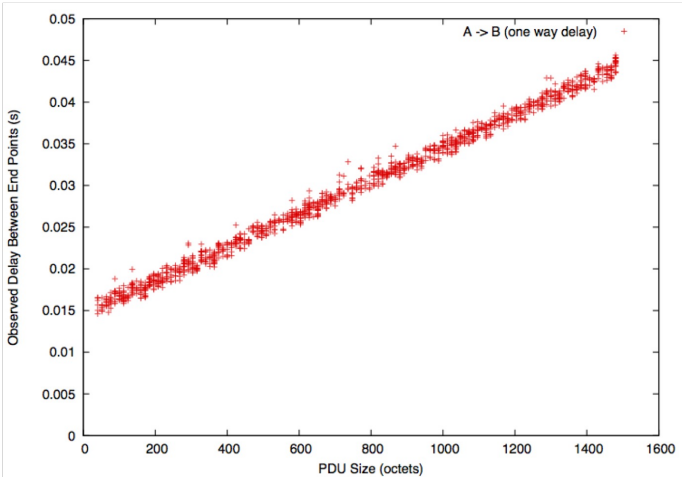
81

Raw two-point measurements



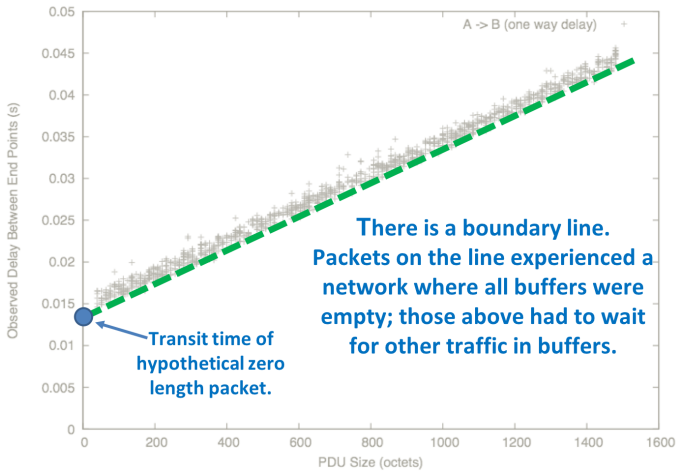
82

Measurements sorted by packet size



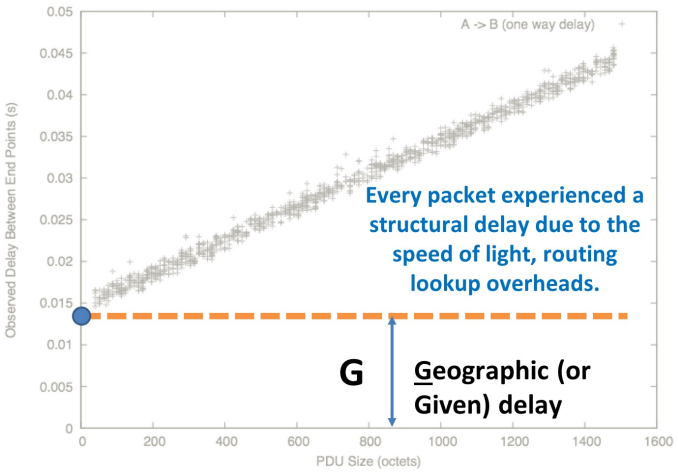
83

Minimum delays for each size



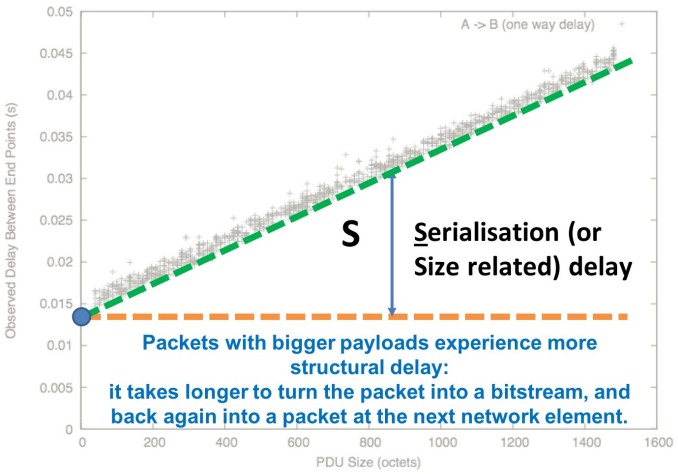
84

Extrapolate to zero size packet: G



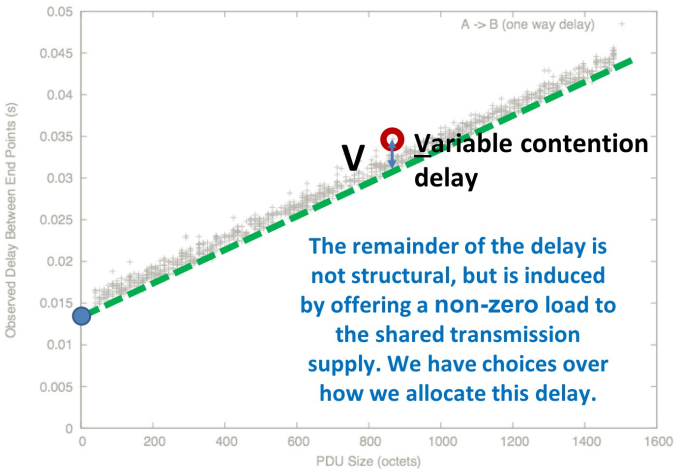
85

Extract S



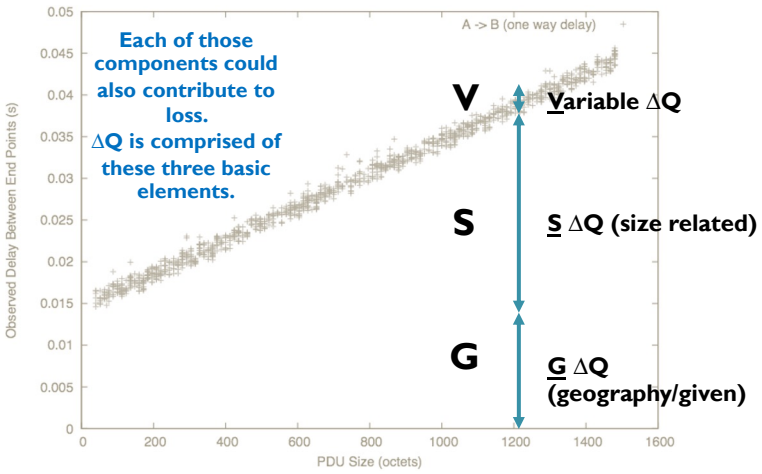
86

V is what remains



87

G, S,V from measured ΔQ



88

6. Extras (work in progress)

89

Extras (work in progress)

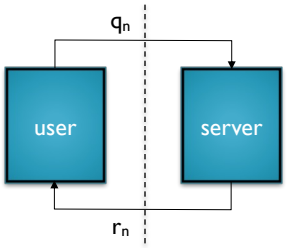
- This part of the tutorial is still work in progress
  - Some aspects of ΔQSD are still being systematized
  - Eventually, it will all be concrete enough so that all of it can be directly used by system designers
- Topics
  - Systems with iterative queries
  - Slacks and hazards
  - Shared resources
  - Blame assignment (diagnosis)
  - Limitations of ΔQSD

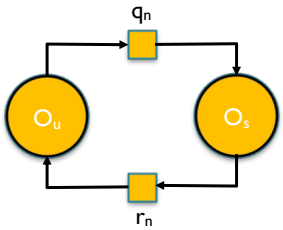
90

Systems with iterative queries

91

Systems with iterative queries

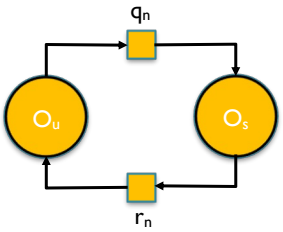




- Consider an iterative process where user sends query q_n to server which sends response r_n back to user, which sends query q_{n+1} and so forth
  - This is a common structure: it models many human-computer interactions on the Web, it models software doing iterative queries to a database, and many other repetitive processes
- How do we compute the ΔQ for this system?
  - There are two kinds of outcomes: $O_{s,n}=(q_n,r_n)$ and $O_{u,n}=(r_n,q_{n+1})$
  - The causal sequence is unbounded: $O_{s,0} < O_{u,0} < O_{s,1} < O_{u,1} < \ldots$

92

ΔQ for iterative queries



- Two equations must be solved simultaneously
  - The server cdf $\Delta Q_s(a)$ is function of load a (as we saw before)
  - Because of iterative execution, load a is function of total delay $\Delta Q_s + \Delta Q_u$
- Load a is expected rate of queries:

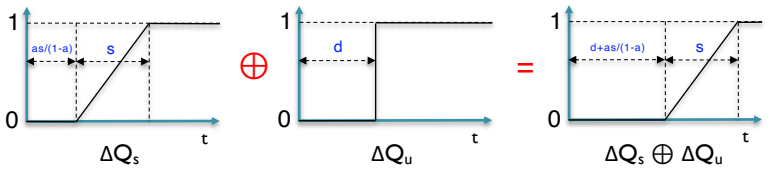
$$a = \int_0^\infty 1/t \; P(t,a)\, dt$$

- $P(t,a) = d(\Delta Q_s + \Delta Q_u)/dt$ is the pdf which is function of t & a
- Each value of load a gives another pdf P(t,a)
- Computing this integral gives an equation to solve for load a

93

93

Solving the equations

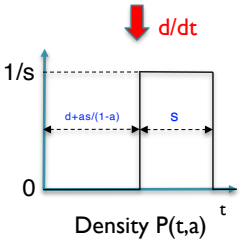


- Working out the integral gives:

$$a = 1/s \ln\left(1 + \frac{1}{d/s + a/(1-a)}\right)$$

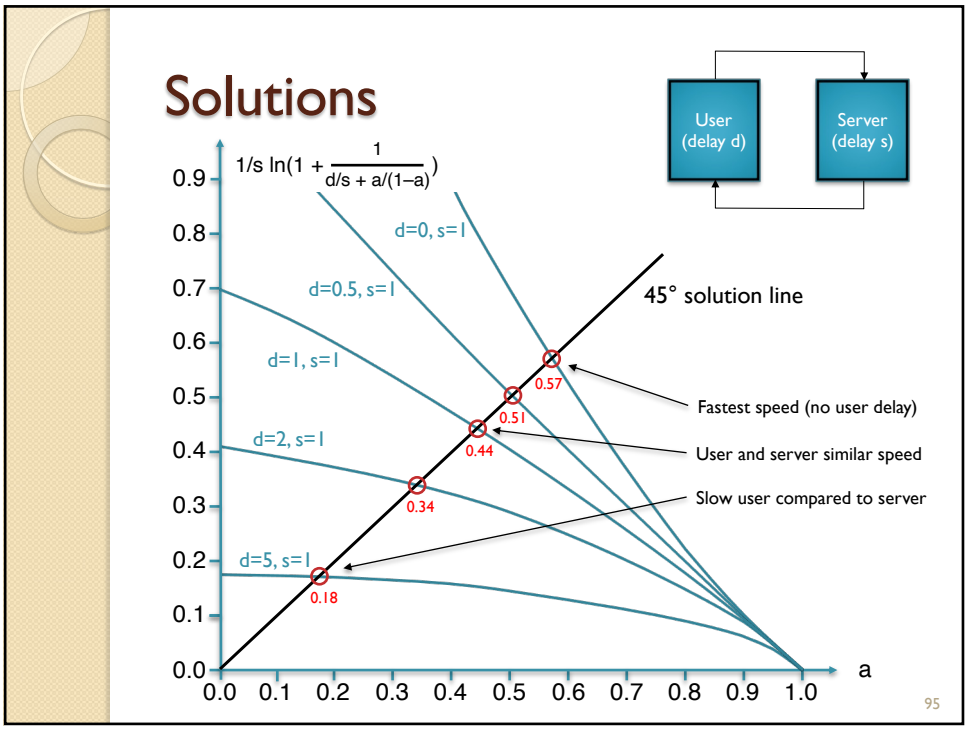
(assuming Uniform distribution)

- Let's look at the solutions
  - Ratio user/server time d/s is important
  - Solutions give good intuition but to be precise you need more computation



94

94



95

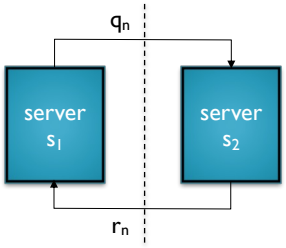
How to measure load

- There are two ways of measuring offered load
  - Arrival rate: number of events per second (as function of time)
  - Interarrival time: interarrival time between events (as function of time)
- What is the right way to compute average load?
- Usually we are interested in the arrival rate
  - Rate is a measure for how much work is being done
    - Work done = rate × duration
  - Rate can be computed using arithmetic average
    - Rate a_1 for duration d followed by rate a_2 for duration d gives average rate $(a_1+a_2)/2$ for duration 2d

96

96

Back-to-back servers



- A similar system is the connection of two servers back-to-back
- This is also a common situation, e.g., two collaborating human teams that communicate with one another
- If $s_1 \neq s_2$ then we can show that almost all waiting messages will queue up at the slow server (smallest s_i)
  - The slow server sets the pace
  - This happens even if the difference between s_1 and s_2 is only a few percent
  - Making the fast server even faster has no effect on performance

97

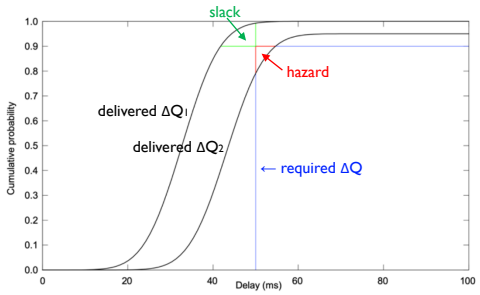
97

Slacks and hazards

98

98

Slacks and hazards

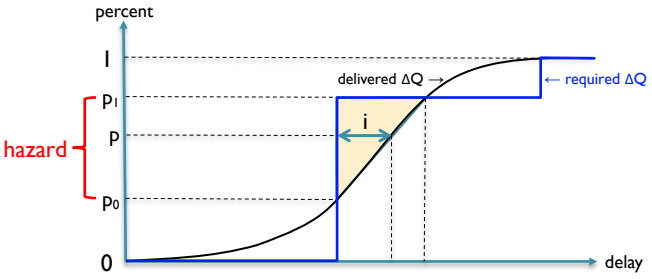


- We can compare a delivered ΔQ to a required ΔQ
  - ΔQ_1 satisfies the requirement; the green part shows the 'slack'
  - ΔQ_2 does not satisfy the requirement; the red part shows the risk or 'hazard' of this violation
- When creating a design, keep slack and hazard in mind
  - Slack gives an extra degree of freedom for the designer, whereas hazard is a potential problem that may need further attention

99

99

Computing hazard from ΔQ



- Risk = impact times probability of occurrence
  - Hazard = probability of occurrence = $p_1 - p_0$
  - Impact = cost (i.e., delay) when it does occur = i(p)
- Because ΔQ is a probability distribution, this is an integral
- $r = \int i \, dp$
  - Total risk is area of orange triangular part
  - Unit of risk is seconds: weighted expected delay

100

100

Order of hazards

	Order	Subject of concern
Compositional	0: Causality	Causal behavior is the only requirement. If ΔQ is best possible, can the system deliver its successful top-level outcomes, i.e., can the system ever work if causality is respected?
Compositional	1: Capacity	Markovian (independent) and linear (superposition) behaviour. Will the delivered ΔQ be within requirements at expected loads, i.e., constant average load within capacity constraints?
Dependent	2: Schedulability	Expected variability in behaviour which can be managed by proper scheduling. Can the QTAs be maintained during reasonable operational stress, i.e., expected load variability?
Dependent	3: Behaviour	Is the system sensitive to internal correlation effects, i.e., interactions between subsystems due to internal effects? For example, all devices doing http lookup at midnight.
Dependent	4: Stress	Is the system sensitive to external correlation effects, i.e., extreme behaviour of the users? For example, all users placing a call when a natural catastrophe occurs.

- We define a hierarchy of performance hazards
- ΔQ computation techniques depend on the order of hazards
  - Orders 0 and 1 assume independence; orders 2, 3, 4 introduce sharing

101

101

Design for overload

- The system must be designed to deal with overload (hazard levels 3 and 4 if long-lasting)
  - Ideally the load never approaches 1
    - As we saw before, when a>0.8 things get bad very quickly
  - But it will happen
    - It is usually too expensive to greatly overdimension the system
    - So overallocation must be combined with other techniques
- Solution
  - Overload must be dealt with at all timescales of interest, using different techniques at different timescales
    - Each level requires its own technique
    - Either mitigate at current level or propagate to next level
    - ΔQSD is used to do appropriate overload management
  - Software must be as idempotent as possible and non-idempotent parts should be isolated

102

102

Overload at different timescales

- Baseline system
  - When overloaded, the system may behave badly but it must never break ("weather the storm")
  - If the load fluctuation is one-shot, this may be sufficient ("ballistic")
- Levels w.r.t. individual tasks
  - Drop nonessential traffic; stop admitting new tasks; kick out tasks already in progress
- Levels w.r.t. system operation (timescale up to days)
  - Depending on timescale: reconfiguration, admission control, cold standbys, data center elasticity, software rejuventation, put human in the loop
- Levels w.r.t. system design (timescale from days to years)
  - One month: add new equipment
  - One year: system redesign, build new data center
  - Longer than one year: fire, forest, flood, nuclear accident, Carrington event, asteroid impact, supervolcano eruption

103

103

Shared resources

104

104

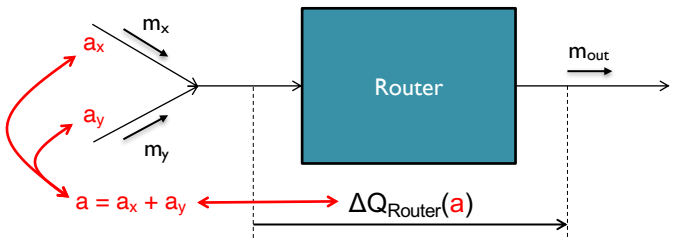
Shared resources

- Computing ΔQ is simple if all components are independent
  - This is the default, compositional approach we have seen so far
- But real systems have shared resources
  - We add sharing between components that share resources
  - Sharing is modelled by additional variables and their equations
  - Computing ΔQ is still possible by adding the equations to the solver
- Kinds of resources
  - Ephemeral resource: used at a particular time instant, e.g., CPU power and network bandwidth, can often be delayed (buffering)
  - Level resource: used over a time interval, e.g., memory
- Other issues (out of scope for today's tutorial)
  - Scheduling (hazard level 2): handle conflicting resource demands
  - Other axes: e.g., threshold resources with large penalty if overused such as working memory which leads to thrashing if overused

105

105

Example 1: congestion

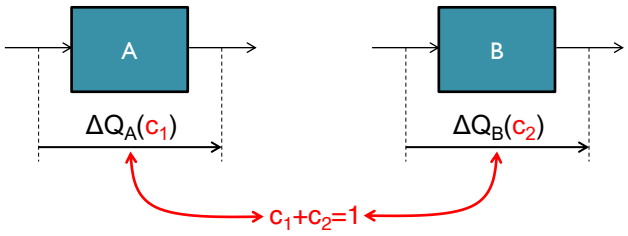


- Assume two message streams entering the same component (e.g., a router)
  - Total load is the sum of the two incoming loads: $a = a_x + a_y$
  - Sharing is modelled as the sum of loads
- Congestion, i.e., buffer overflow and message drop, is computed from ΔQ_{Router} using the queue model we saw before
  - Router will behave well if $a_x + a_y < 0.8$
  - Message delay and message failure are computed using the queue

106

106

Example 2: shared CPU



- Assume two components are implemented on the same processor core
  - Each component uses fraction c_i of the processing power with the constraint $c_1+c_2=1$
  - ΔQ of each component is function of its processor utilisation
- This gives extra arguments c_1 and c_2 to the ΔQs and an equation (constraint) linking them

107

107

Blame assignment (diagnosis)

108

108

Blame assignment (diagnosis)

- Diagnosis resembles top-down design
  - The design is finished and known to be inadequate, and we want to determine which suboutcome is the reason for the inadequacy
  - Assume we know the full outcome diagram with many suboutcomes
  - There may not be one unique answer, there may be between multiple problematic suboutcomes
  - It is important to ask “what if” questions
  - This is an exploratory technique that uses search and is computationally intensive
- This is still very much work in progress

109

109

Limitations

110

110

Limitations

- ΔQSD is a design approach that allows to predict performance and feasibility for partially specified systems
  - The default system model is fully compositional, with independent components
    - Quantitative behaviour of individual components must be known in advance
  - Dependencies are added where they affect the system
    - Forgetting to add some dependencies will reduce prediction accuracy
- ΔQSD is most applicable to systems that execute many independent instances of the same action
  - For systems that execute long sequences of dependent actions, the predictions will be less accurate
- Achieving ΔQSD's full power requires significant computation
  - It can be used for back-of-the-envelope design but with loss of accuracy
  - It is most suitable as foundation for a design tool

111

111

7. Conclusions

112

112

Conclusions and future steps

- This short presentation only scratches the surface of ΔQSD; there is much more
  - Practical measurement and computation of ΔQ
  - Practical design and diagnosis techniques
  - Case studies and typical ΔQs
  - Computation of shared resources (e.g., congestion)
- PNSol has detailed slide decks and documentation
  - Theory and practice of ΔQSD
  - Experience reports for large industrial projects
- We have an ongoing project to formalize ΔQSD and build tools
  - We are looking for Ph.D. students to help us
  - Publication "Mind Your Outcomes", Computers 2022, 11, 45 https://www.mdpi.com/2073-431X/11/3/45

113

113