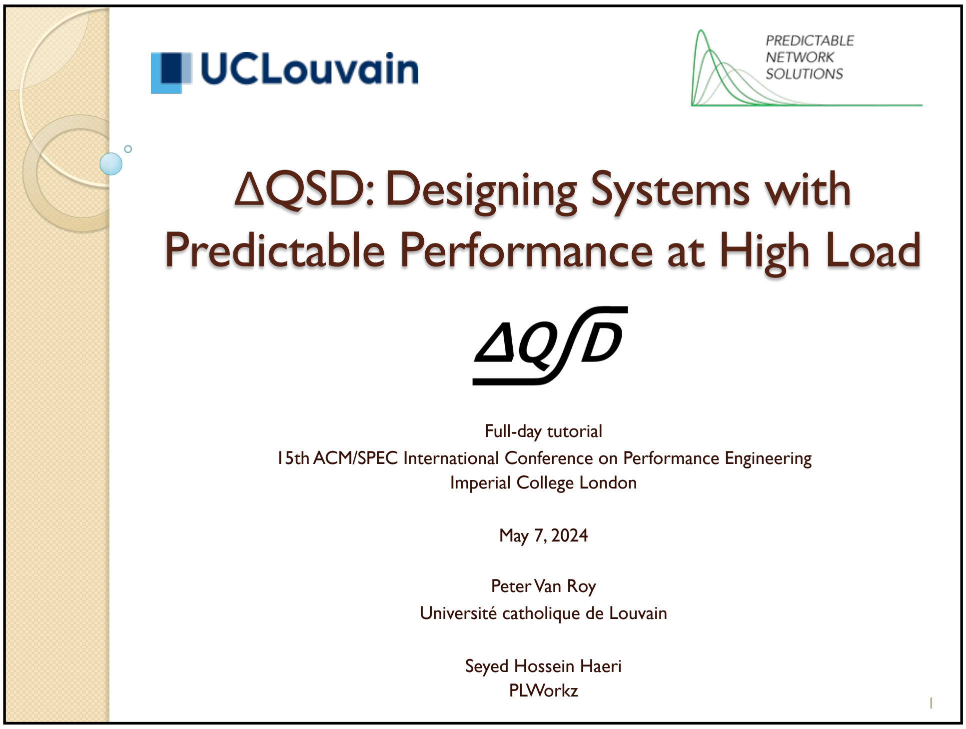


ΔQSD: Designing Systems with Predictable Performance at High Load

Full-day tutorial
15th ACM/SPEC International Conference on Performance Engineering
Imperial College London

May 7, 2024

Peter Van Roy
Université catholique de Louvain

Seyed Hossein Haeri
PLWorkz

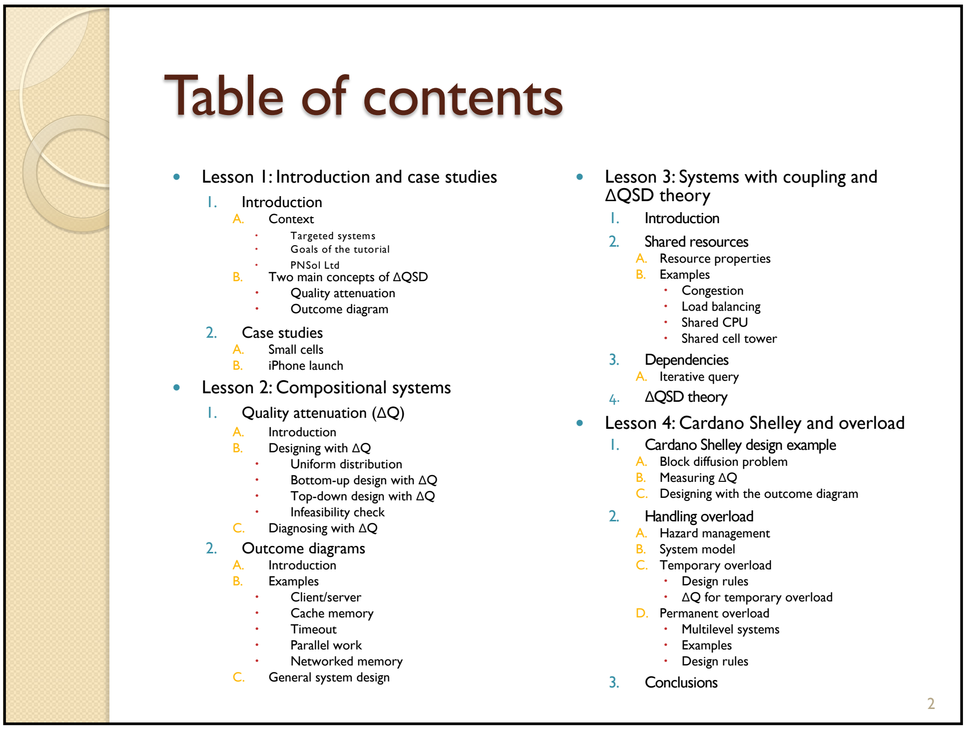
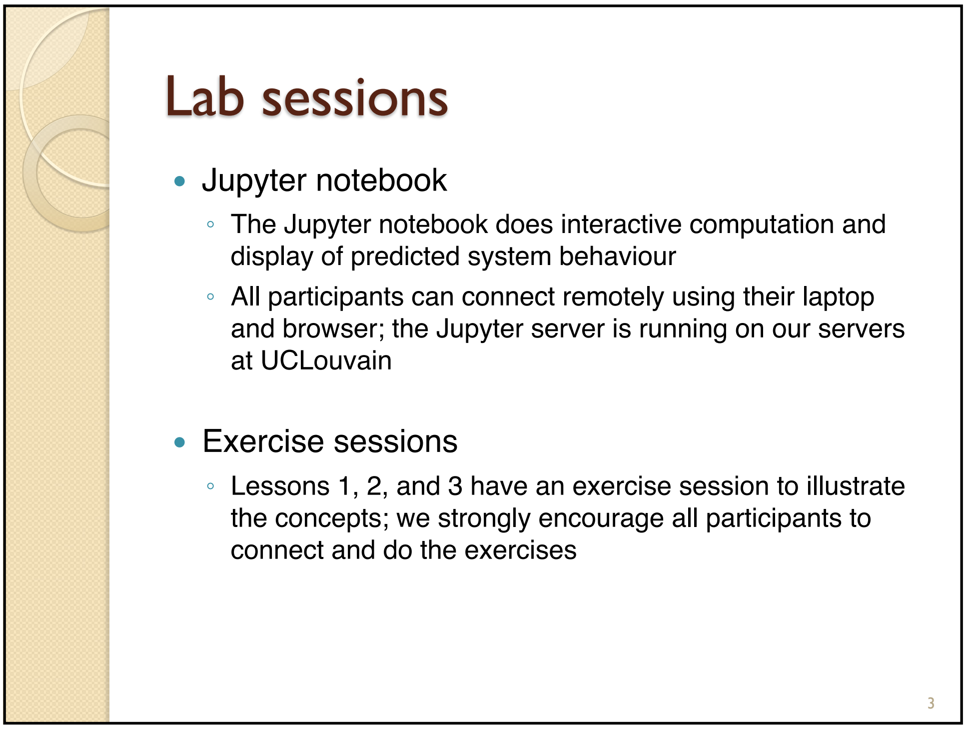
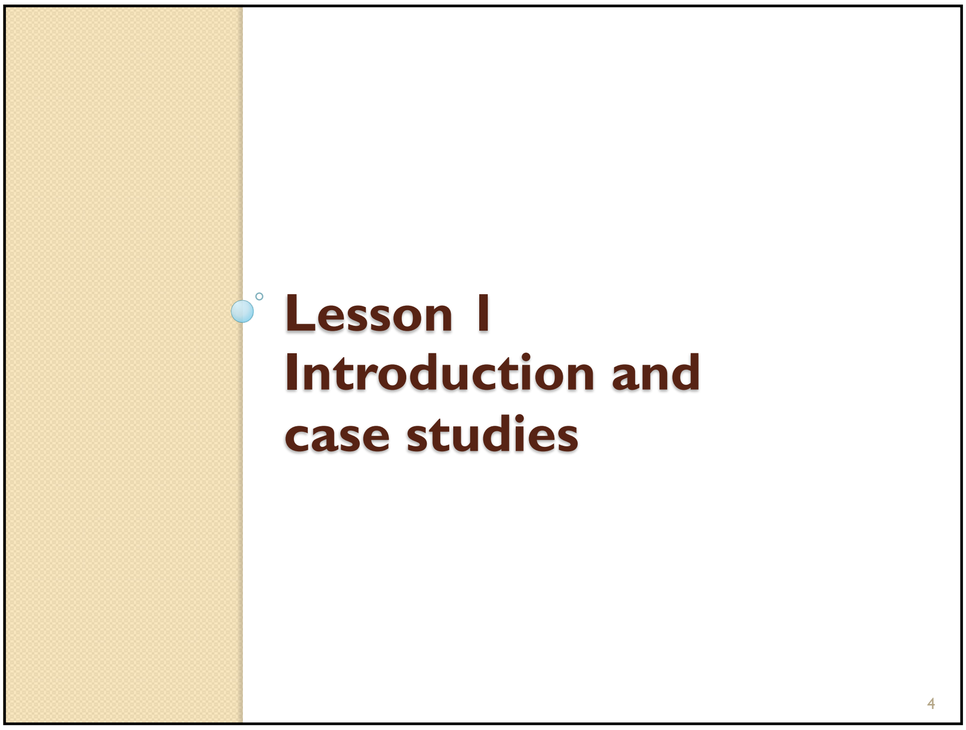


Table of contents

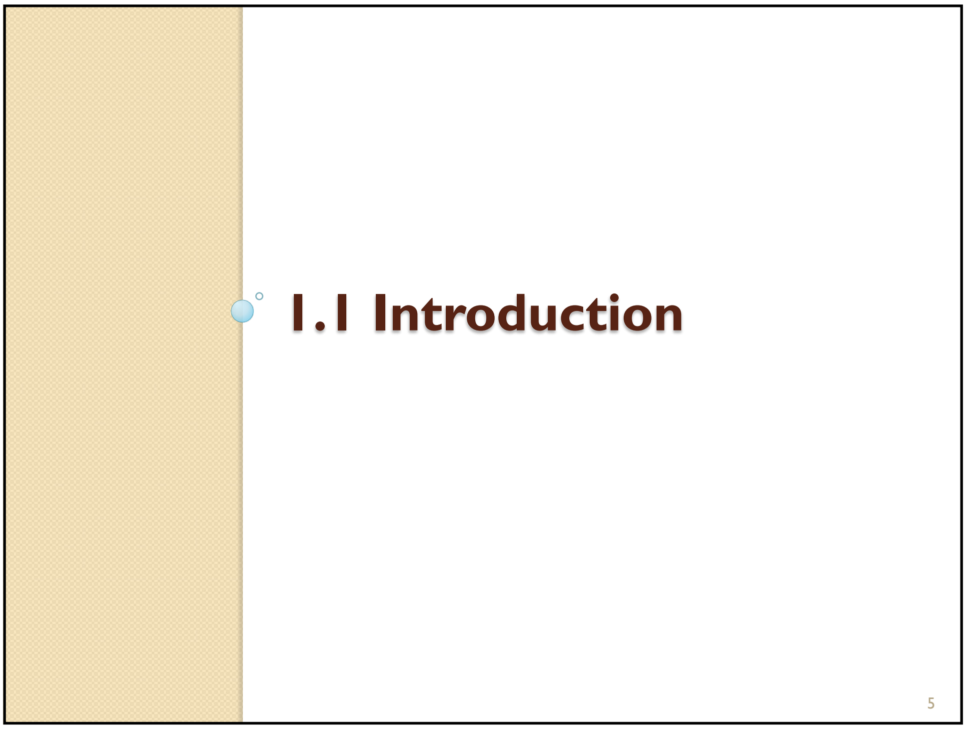


Lab sessions

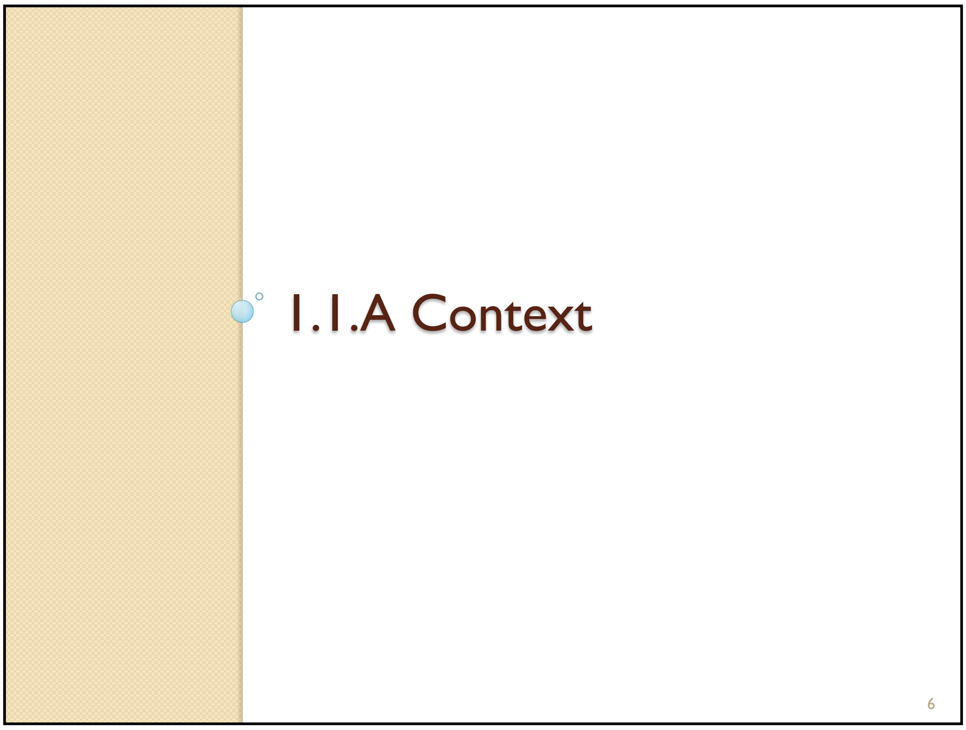
- Jupyter notebook
  - The Jupyter notebook does interactive computation and display of predicted system behaviour
  - All participants can connect remotely using their laptop and browser; the Jupyter server is running on our servers at UCLouvain
- Exercise sessions
  - Lessons 1, 2, and 3 have an exercise session to illustrate the concepts; we strongly encourage all participants to connect and do the exercises



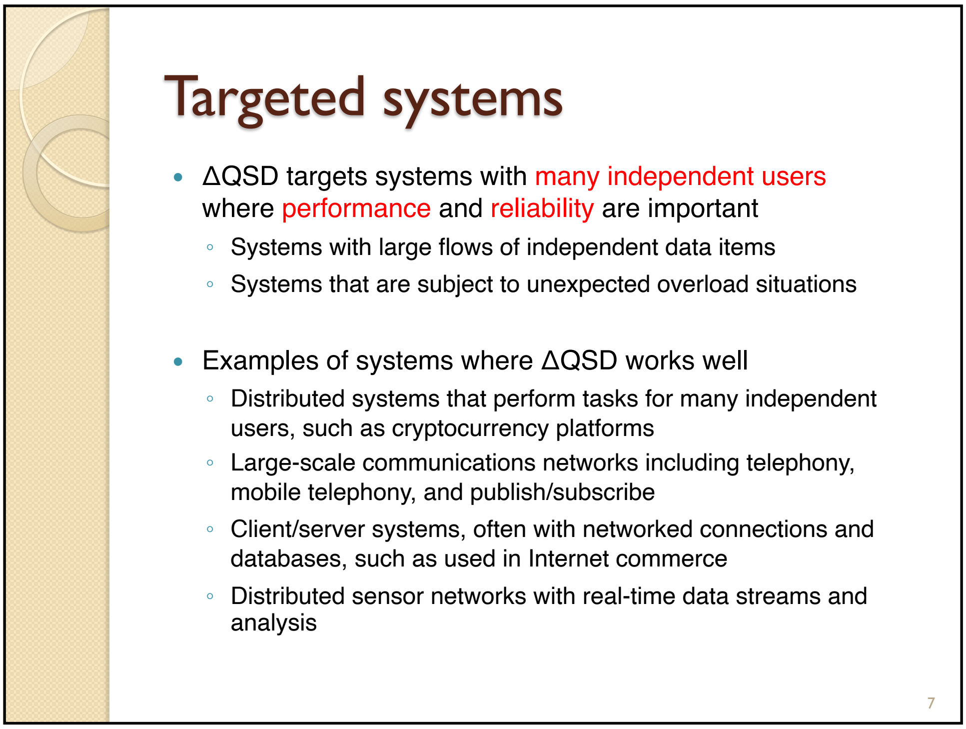
Lesson 1
Introduction and case studies



1.1 Introduction

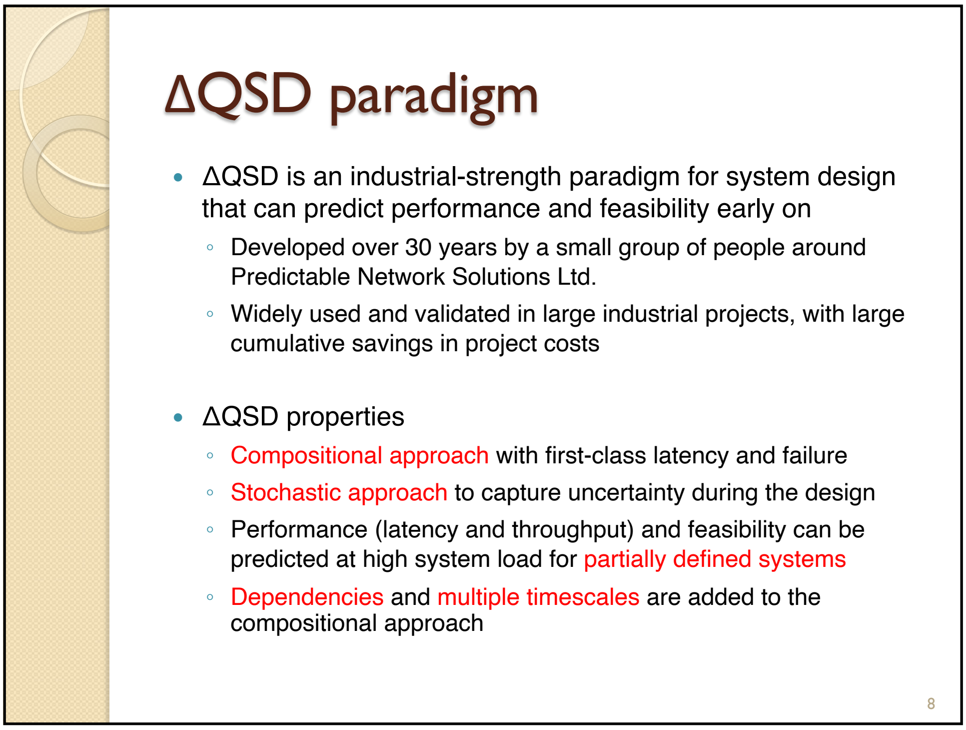


1.1.A Context



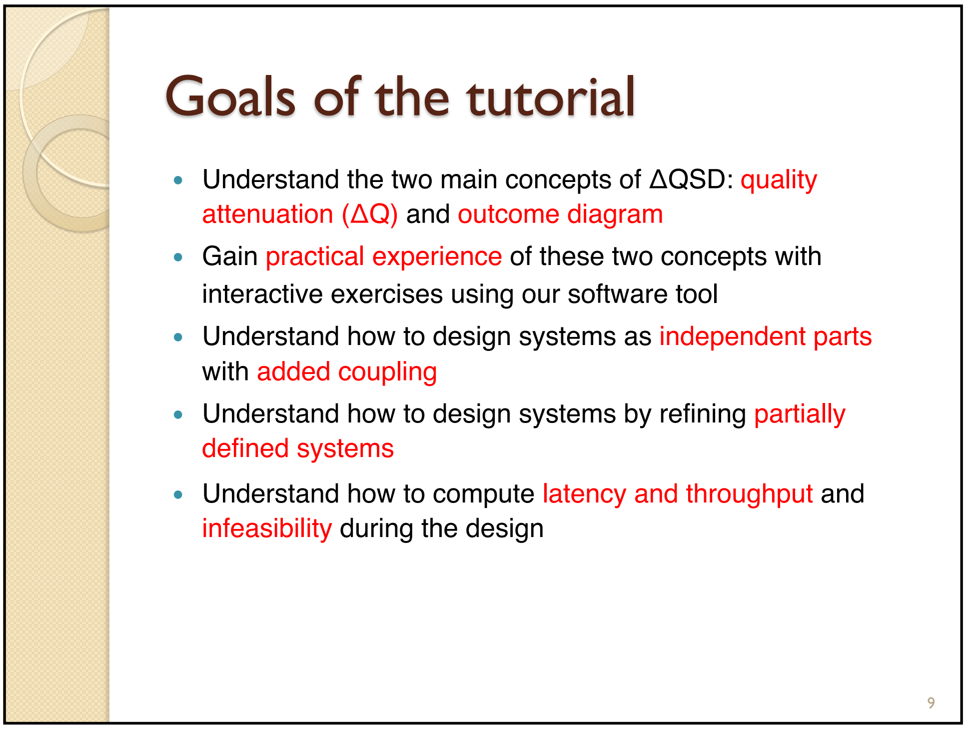
Targeted systems

- ΔQSD targets systems with many independent users where performance and reliability are important
    - Systems with large flows of independent data items
    - Systems that are subject to unexpected overload situations
- Examples of systems where ΔQSD works well
    - Distributed systems that perform tasks for many independent users, such as cryptocurrency platforms
    - Large-scale communications networks including telephony, mobile telephony, and publish/subscribe
    - Client/server systems, often with networked connections and databases, such as used in Internet commerce
    - Distributed sensor networks with real-time data streams and analysis



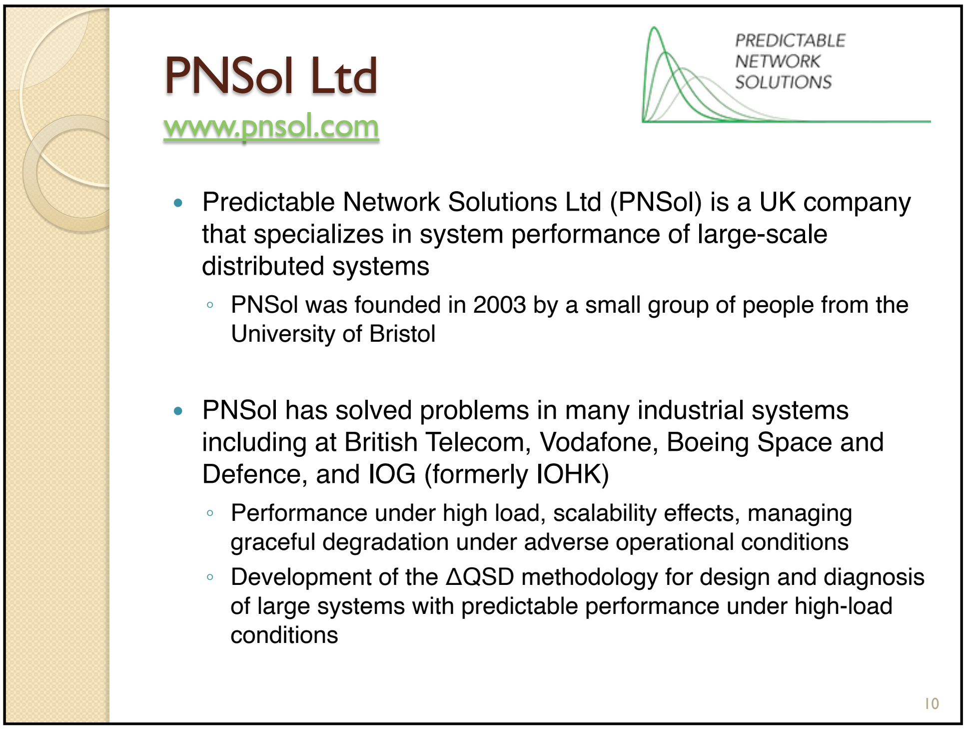
ΔQSD paradigm

- ΔQSD is an industrial-strength paradigm for system design that can predict performance and feasibility early on
    - 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 with first-class latency and failure
    - Stochastic approach to capture uncertainty during the design
    - Performance (latency and throughput) and feasibility can be predicted at high system load for partially defined systems
    - Dependencies and multiple timescales are added to the compositional approach



Goals of the tutorial

- Understand the two main concepts of ΔQSD: quality attenuation (ΔQ) and outcome diagram
- Gain practical experience of these two concepts with interactive exercises using our software tool
- Understand how to design systems as independent parts with added coupling
- Understand how to design systems by refining partially defined systems
- Understand how to compute latency and throughput and infeasibility during the design

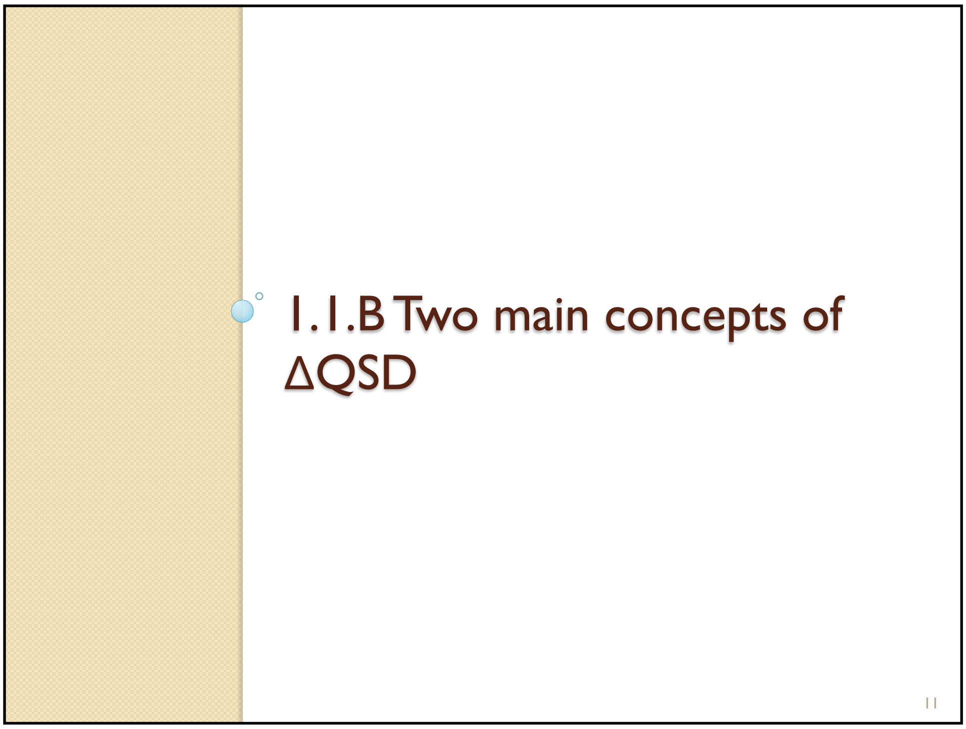


PNSol Ltd

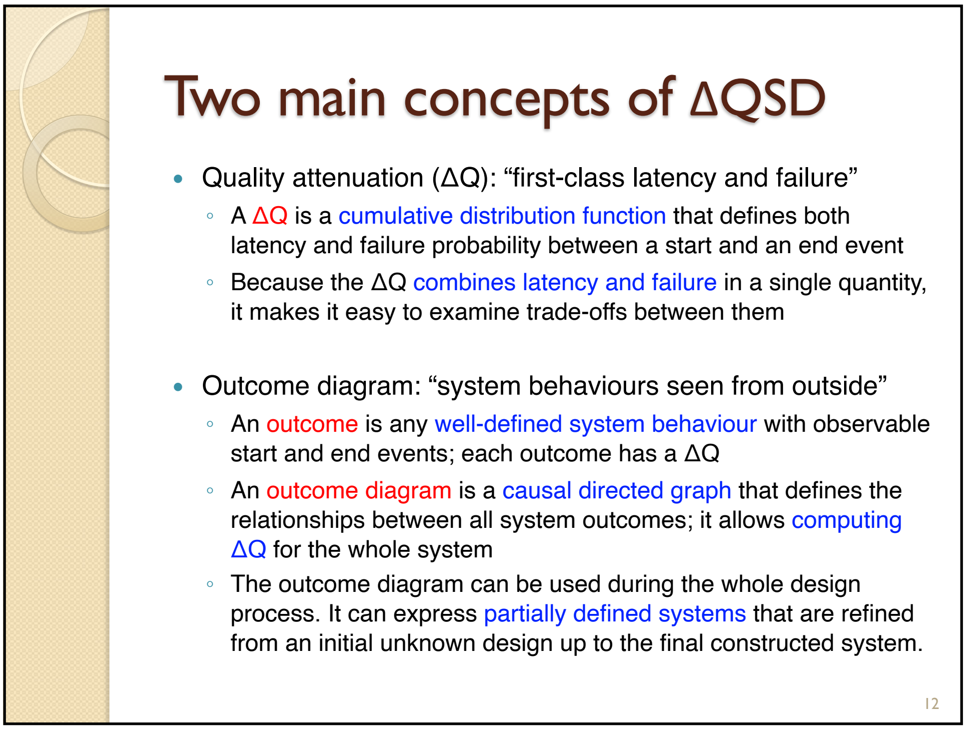
www.pnsol.com

PREDICTABLE
NETWORK
SOLUTIONS

- Predictable Network Solutions Ltd (PNSol) is a UK company that specializes in system performance of large-scale distributed systems
  - PNSol was founded in 2003 by a small group of people from the University of Bristol
- PNSol has solved problems in many industrial systems including at British Telecom, Vodafone, Boeing Space and Defence, and IOG (formerly IOHK)
  - Performance under high load, scalability effects, managing graceful degradation under adverse operational conditions
  - Development of the ΔQSD methodology for design and diagnosis of large systems with predictable performance under high-load conditions

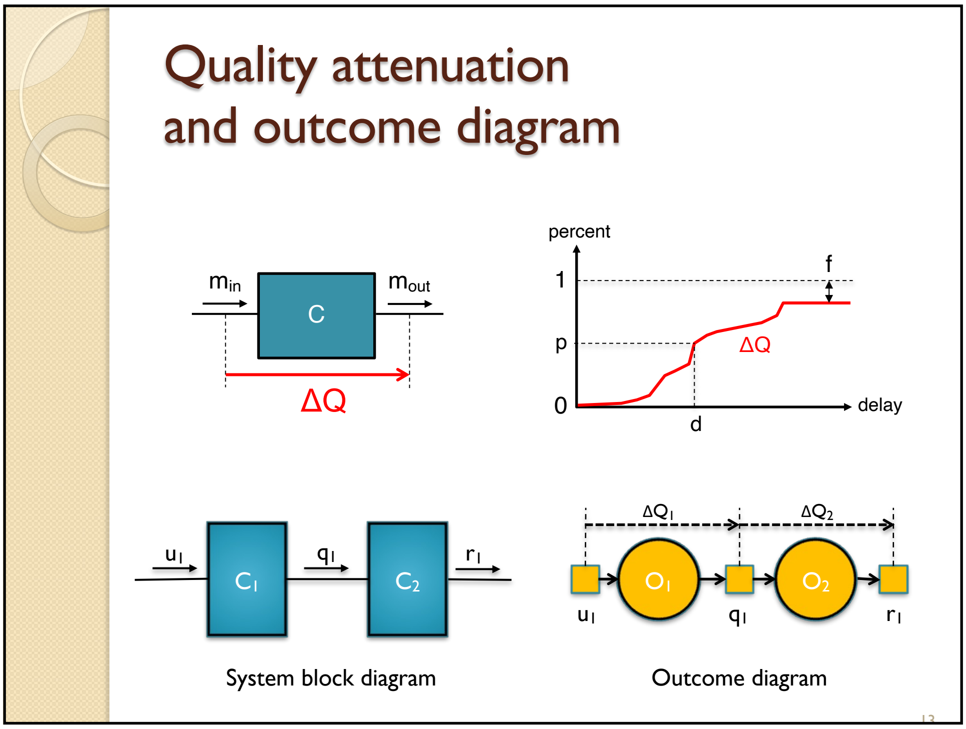


1.1.B Two main concepts of ΔQSD



Two main concepts of ΔQSD

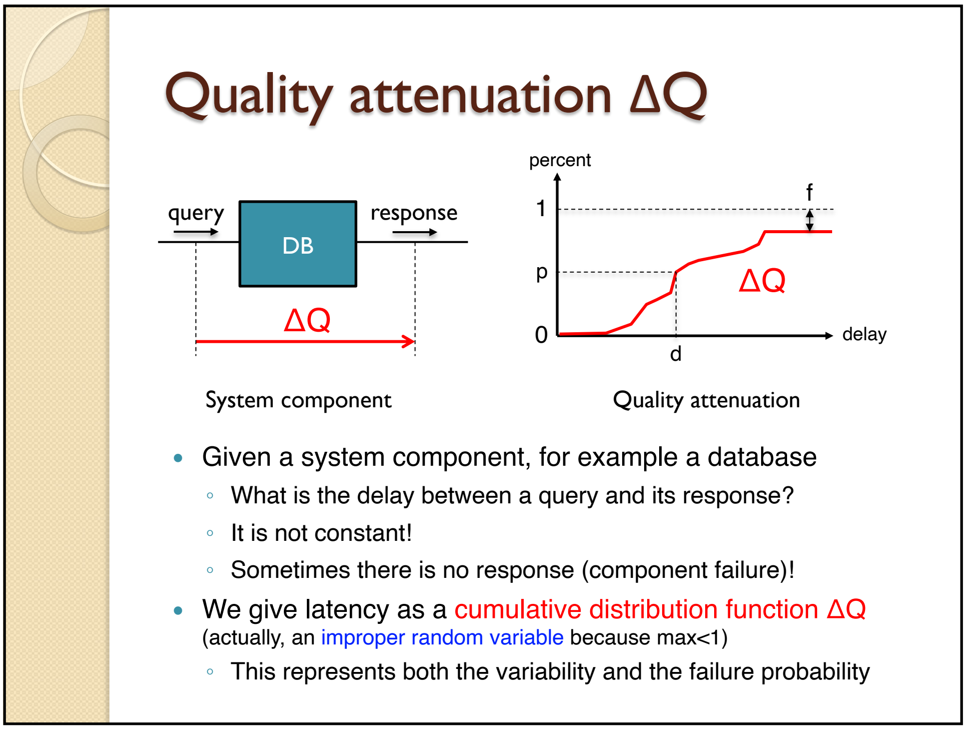
- Quality attenuation (ΔQ): “first-class latency and failure”
  - A ΔQ is a cumulative distribution function that defines both latency and failure probability between a start and an end event
  - Because the ΔQ combines latency and failure in a single quantity, it makes it easy to examine trade-offs between them
- Outcome diagram: “system behaviours seen from outside”
  - 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 during the whole design process. It can express partially defined systems that are refined from an initial unknown design up to the final constructed system.



Quality attenuation and outcome diagram

System block diagram

Outcome diagram

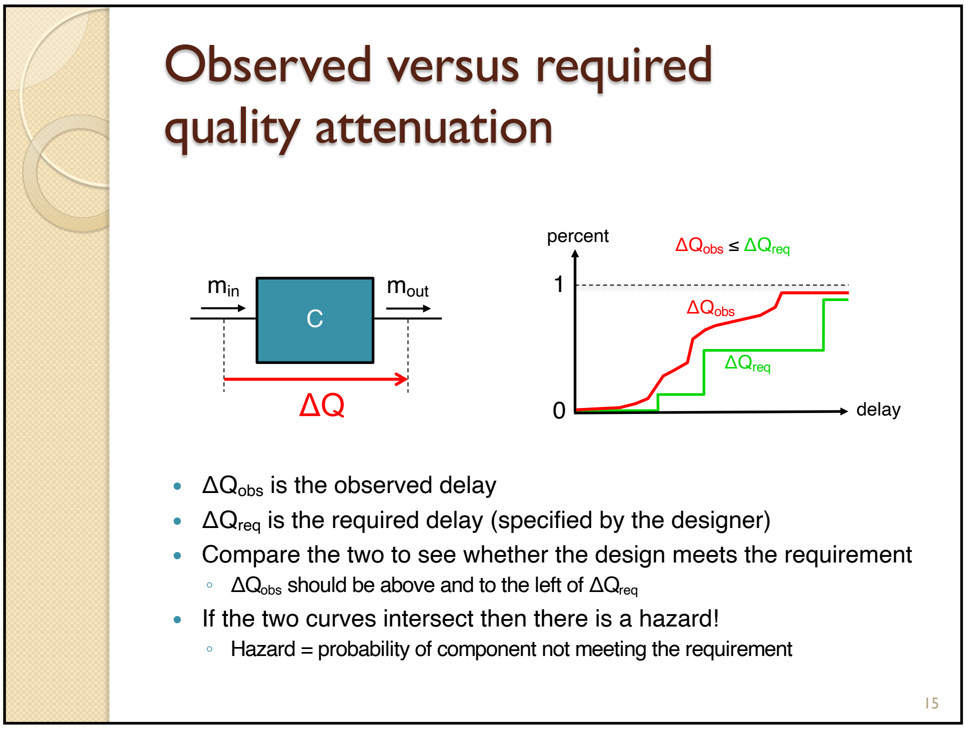


Quality attenuation ΔQ

System component

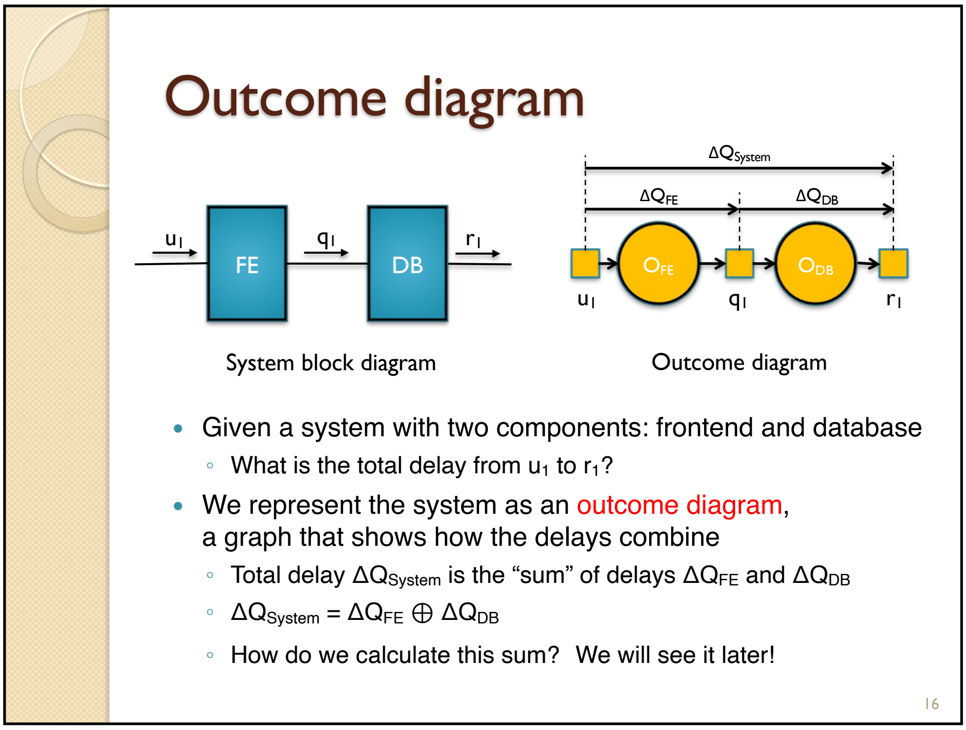
Quality attenuation

- Given a system component, for example a database
  - What is the delay between a query and its response?
  - It is not constant!
  - Sometimes there is no response (component failure)!
- We give latency as a cumulative distribution function ΔQ (actually, an improper random variable because max<1)
  - This represents both the variability and the failure probability



Observed versus required quality attenuation

- ΔQ_{obs} is the observed delay
- ΔQ_{req} is the required delay (specified by the designer)
- Compare the two to see whether the design meets the requirement
  - ΔQ_{obs} should be above and to the left of ΔQ_{req}
- If the two curves intersect then there is a hazard!
  - Hazard = probability of component not meeting the requirement



Outcome diagram

System block diagram

Outcome diagram

- Given a system with two components: frontend and database
  - What is the total delay from u_1 to r_1?
- We represent the system as an outcome diagram, a graph that shows how the delays combine
  - Total delay ΔQ_{System} is the "sum" of delays ΔQ_{FE} and ΔQ_{DB}
  - $\Delta Q_{System} = \Delta Q_{FE} \oplus \Delta Q_{DB}$
  - How do we calculate this sum? We will see it later!

To the case studies…

- Now we know enough for the case studies
- Diagnosing a misbehaving system
    - In the first stage, measure ΔQ_{system} of the whole system
    - In the second stage, reason backwards to pinpoint the problem by measuring ΔQs of parts of the system
- After the case studies, we will study how to design systems using ΔQ and outcome diagrams

1.2 Case Studies

Case studies

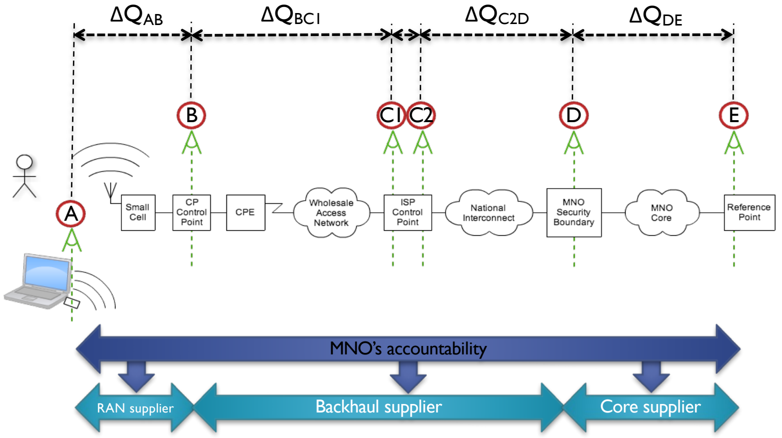
- As motivation for ΔQSD we present two case studies
  - Small cells
  - iPhone launch
- These are industrial case studies done by PNSol that have limited documentation and are partially covered by NDA
- In these scenarios, the ΔQSD paradigm is used to diagnose a misbehaving system
  - Later on we show how to use ΔQSD to design a system with predictable performance at high load (Cardano Shelley)
- It's better to use ΔQSD for design rather than diagnosis
  - Prevention is better than cure!
  - This is one of the motivations of this tutorial: to disseminate the ΔQSD paradigm so it can be used during the design process
    - PNSol is often called in to perform a cure for systems with major problems

1.2.A Small cells case study

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

Who is to blame for my system crashing?

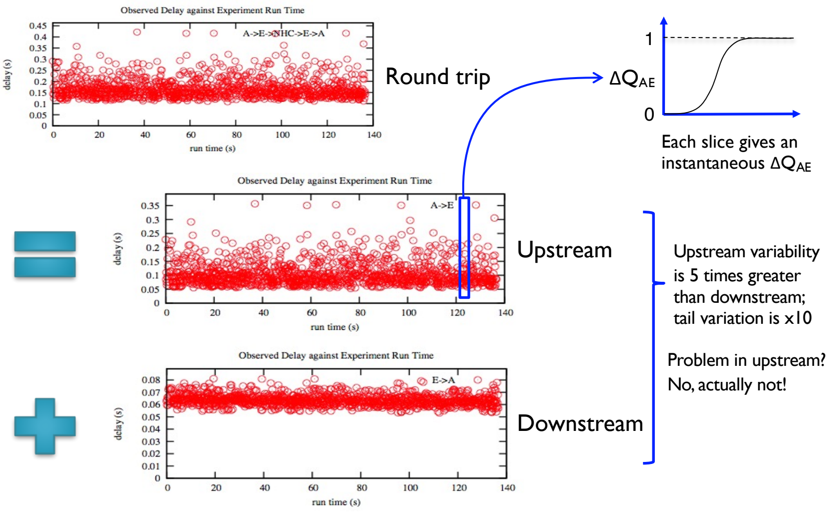


MNO (erroneously) believed that: (1) its contracts would deliver the service & contain the hazards; and (2) there were no residual hazards.

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 → D → C2 → C1 → B → A
  - Measure quality attenuation ΔQ for outcomes
  - Identify issues and anomalies for further investigation
- **Each added observation point *greatly increases* spatial fidelity**
  - Example: even with just **A** and **E** there is definitive knowledge as to whether the effect is occurring upstream or downstream.

Which direction has issues?



Round trip

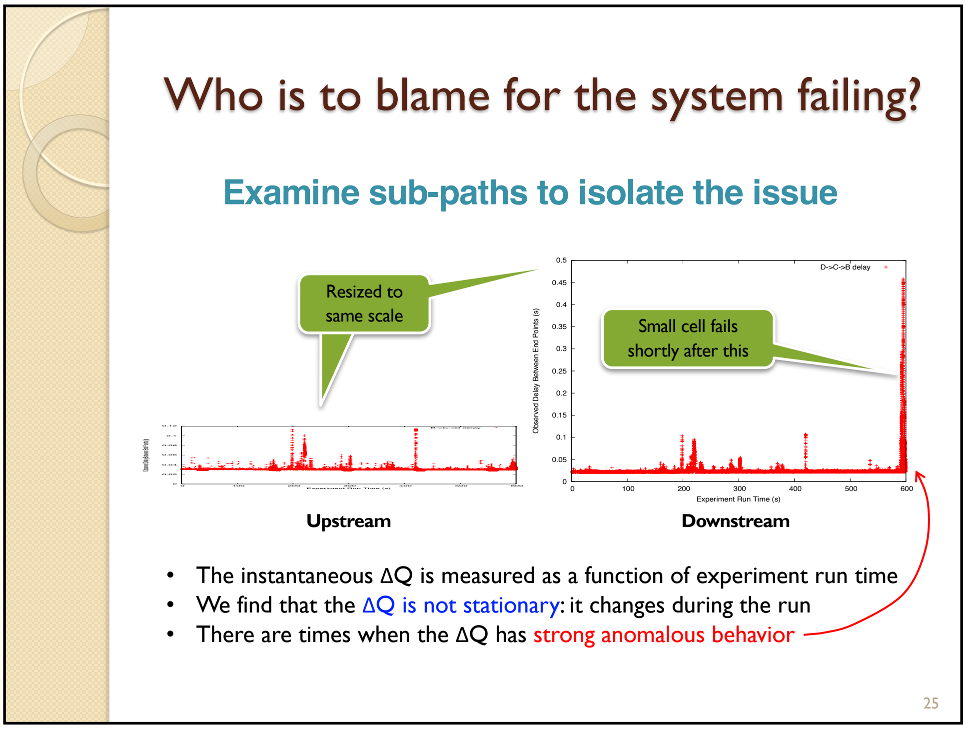
Each slice gives an instantaneous ΔQ_{AE}

Upstream

Downstream

Upstream variability is 5 times greater than downstream; tail variation is x10

Problem in upstream? No, actually not!



Who is to blame for the system failing?

Examine sub-paths to isolate the issue

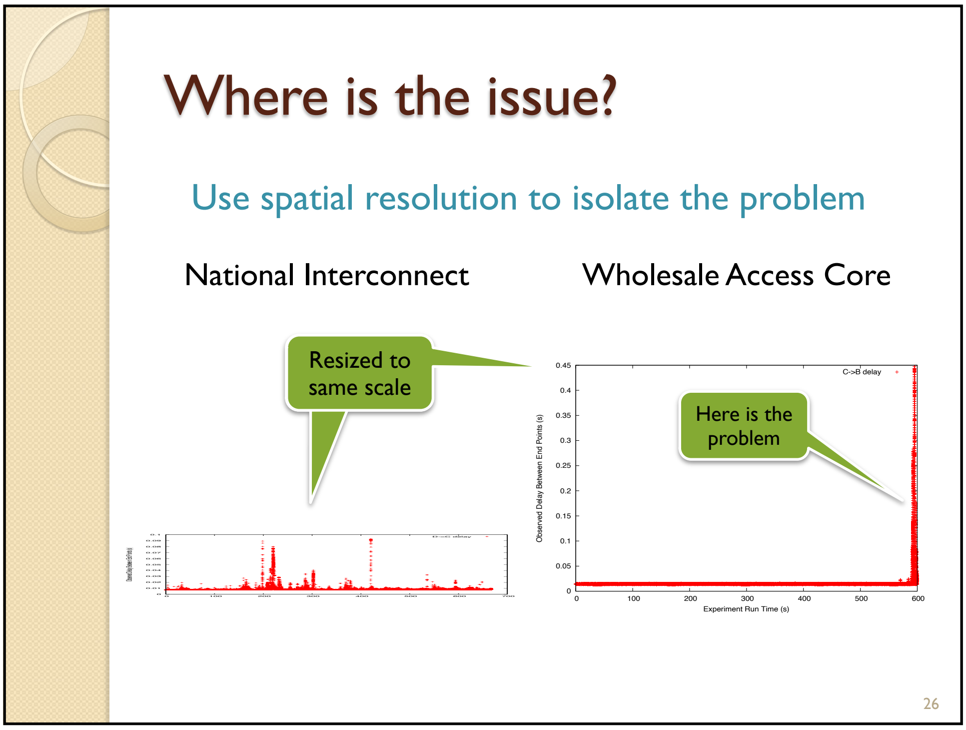
Resized to same scale

Small cell fails shortly after this

Upstream

Downstream

- The instantaneous ΔQ is measured as a function of experiment run time
- We find that the ΔQ is not stationary: it changes during the run
- There are times when the ΔQ has strong anomalous behavior



Where is the issue?

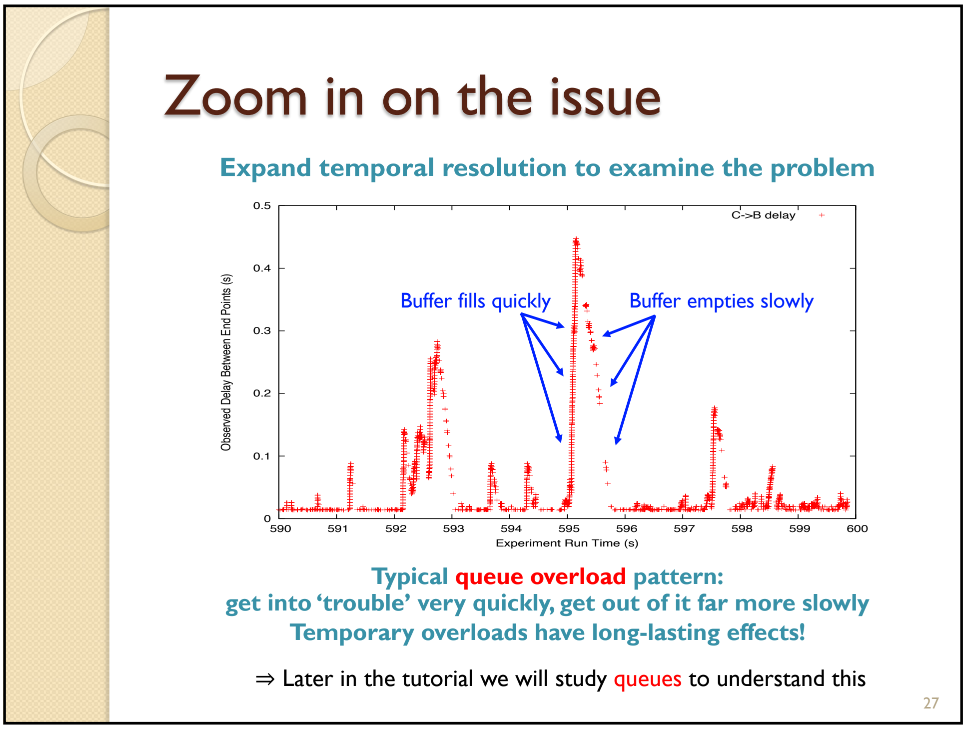
Use spatial resolution to isolate the problem

National Interconnect

Wholesale Access Core

Resized to same scale

Here is the problem

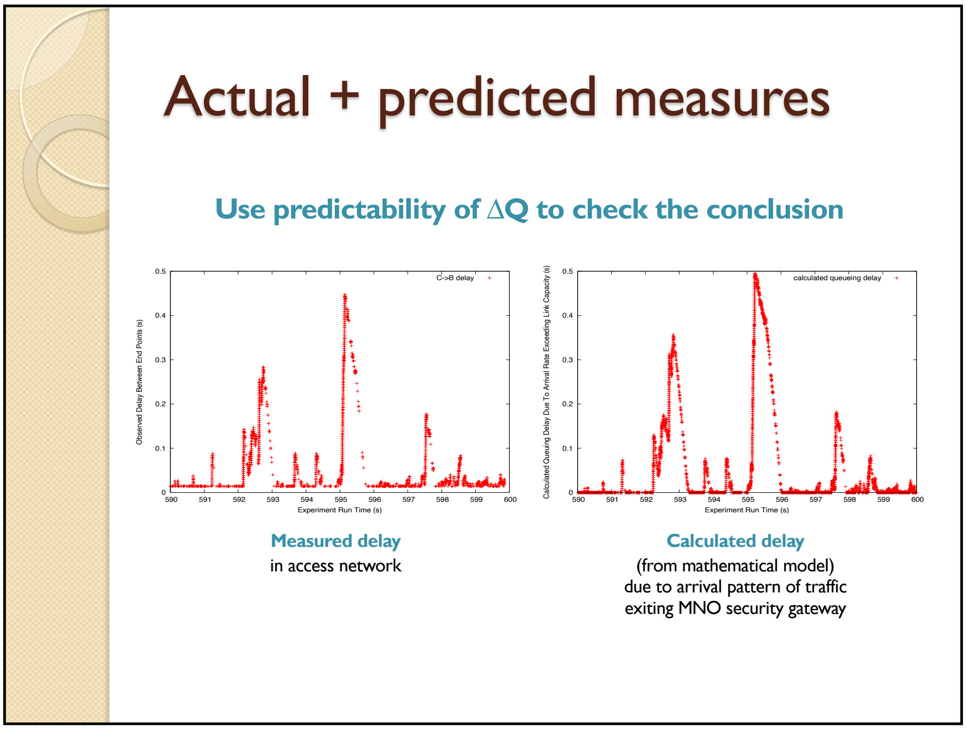


Zoom in on the issue

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



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

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**

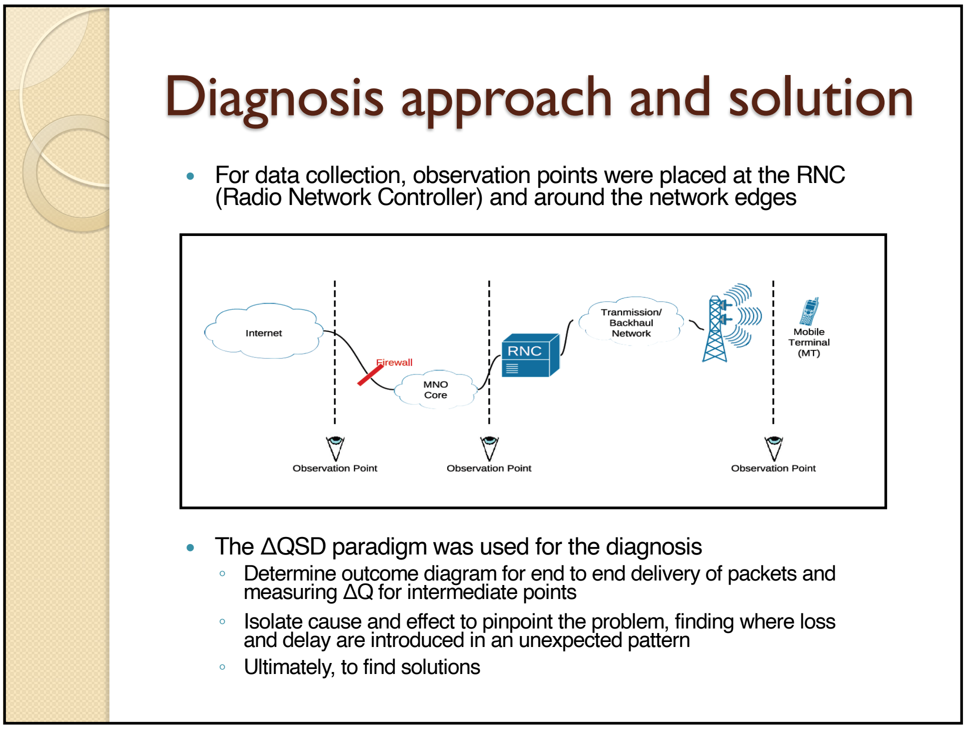
Systemic diagnosis and cure

- 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
        - Common structural problem, not unique to this MNO or technology
    - The MNO believed they only had to match bandwidths (numbers!)
        - They should match **ΔQ (CDFs!)** (Quantitative Timeliness Agreement)
- **Recommendations to the MNO:**
    - **Note on corporate risk register: records the risks and opportunities that affect the delivery of the Corporate Plan**
    - **Technical training to improve contractual processes & hazard management**

1.2.B iPhone launch case study

iPhone launch case study

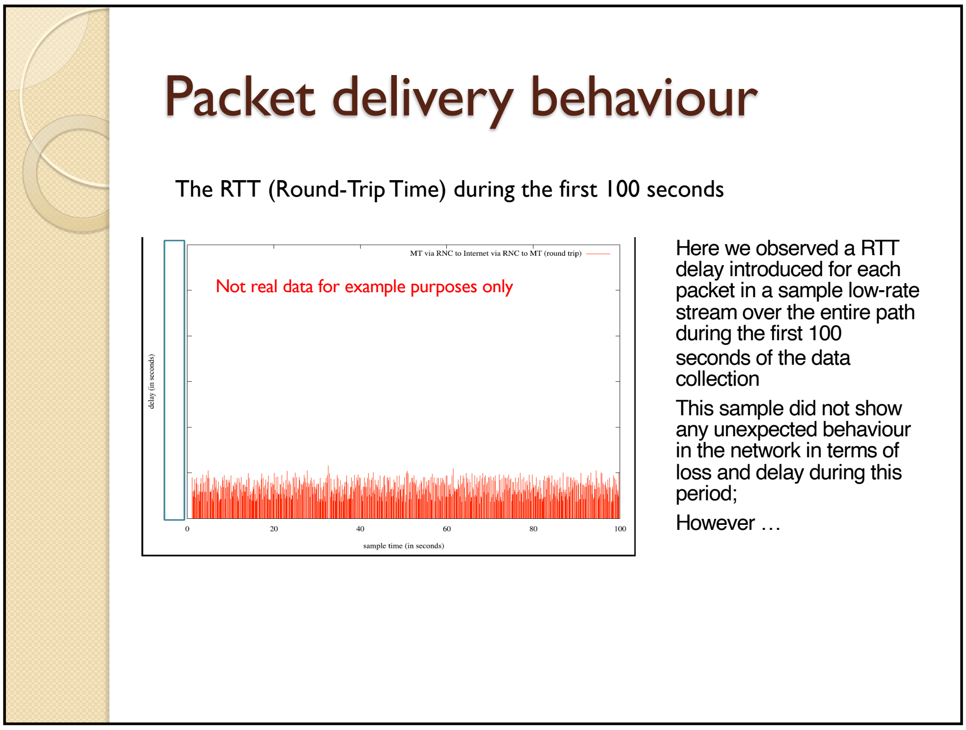
- iPhone was initially supported in UK by one MNO
- A second MNO prepared to enter this market
  - Before the launch, the performance was known to be bad for the second MNO, and the first MNO had gleefully prepared a major ad campaign focusing on this fact
    - Both MNOs are large UK operators who will 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



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



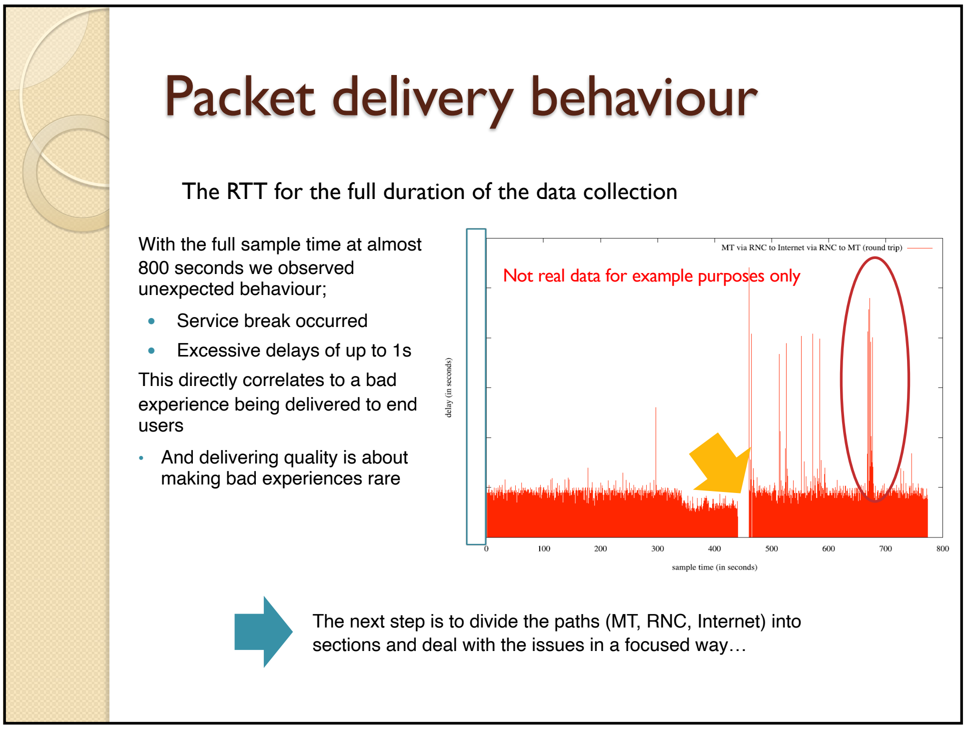
Packet delivery behaviour

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 …



Packet delivery behaviour

The RTT for the full duration of the data collection

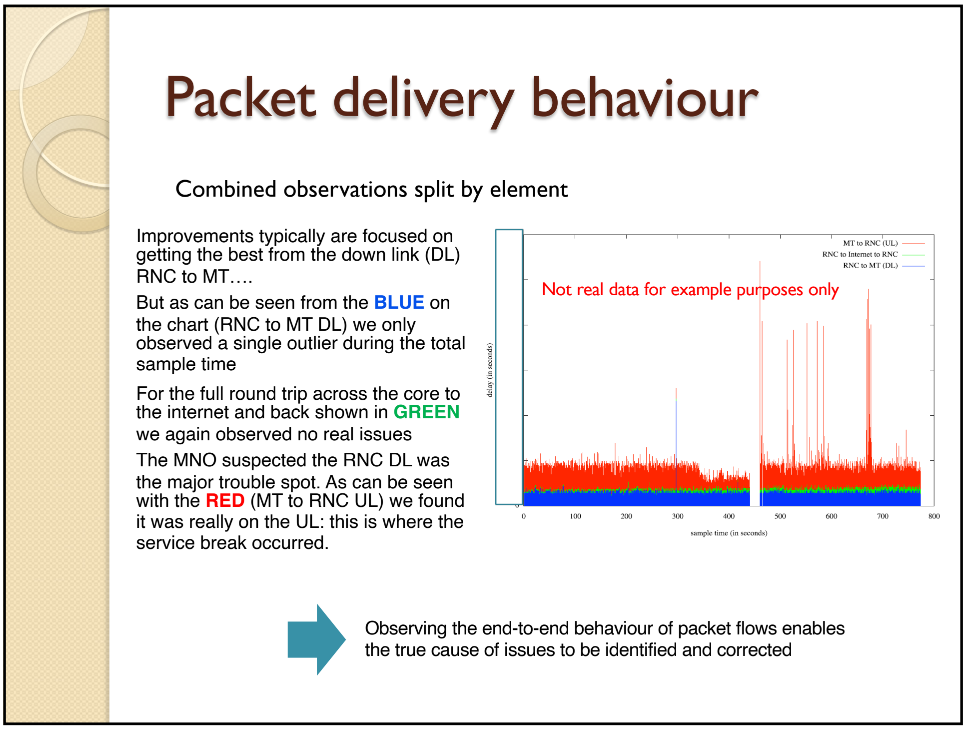
With the full sample time at almost 800 seconds we observed unexpected behaviour;

- 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…



Packet delivery behaviour

Combined observations split by element

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

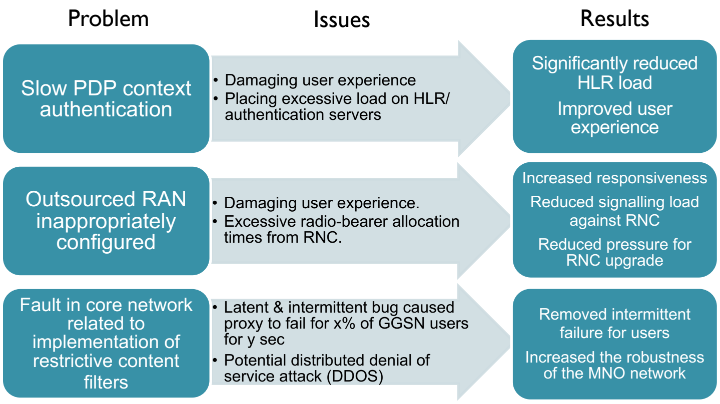
But as can be seen from the **BLUE** on the chart (RNC to MT DL) we only observed a single outlier during the total sample time

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.

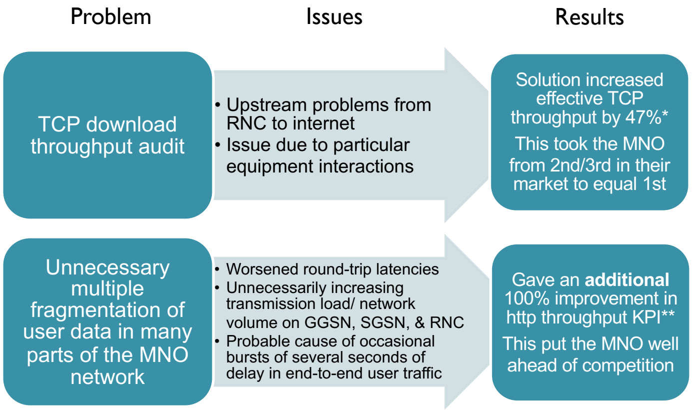
Observing the end-to-end behaviour of packet flows enables the true cause of issues to be identified and corrected

iPhone launch findings using ΔQSD (1 of 2)



Problem	Issues	Results
Slow PDP context authentication	• Damaging user experience • Placing excessive load on HLR/ authentication servers	Significantly reduced HLR load Improved user experience
Outsourced RAN inappropriately configured	• Damaging user experience. • Excessive radio-bearer allocation times from RNC.	Increased responsiveness Reduced signalling load against RNC Reduced pressure for RNC upgrade
Fault in core network related to implementation of restrictive content filters	• Latent & intermittent bug caused proxy to fail for x% of GGSN users for y sec • Potential distributed denial of service attack (DDOS)	Removed intermittent failure for users Increased the robustness of the MNO network

iPhone launch findings using ΔQSD (2 of 2)



Problem	Issues	Results
TCP download throughput audit	• Upstream problems from RNC to internet • Issue due to particular equipment interactions	Solution increased effective TCP throughput by 47%* This took the MNO from 2nd/3rd in their market to equal 1st
Unnecessary multiple fragmentation of user data in many parts of the MNO network	• Worsened round-trip latencies • Unnecessarily increasing transmission load/ network volume on GGSN, SGSN, & RNC • Probable cause of occasional bursts of several seconds of delay in end-to-end user traffic	Gave an **additional** 100% improvement in http throughput KPI** This put the MNO well ahead of competition

* as measured by independent testing
** by internal MNO measurement

1.3 Lab 1:
Introduction to Jupyter

Lesson 2
Compositional systems

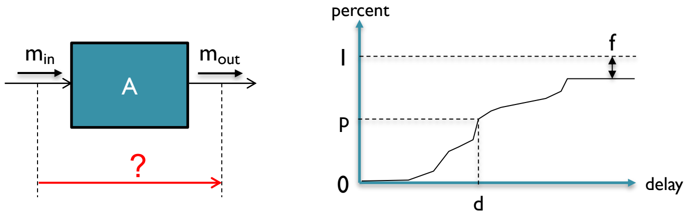
Systems with independent parts (compositional systems)

- ΔQSD approach is done in three steps
  - First, design the system with independent parts
  - Second, add couplings where they are needed
  - Third, add multiple levels to handle multiple timescales
- We start with systems of independent parts
  - Most systems consist largely of independent parts
  - Coupling and multiple levels will be treated later (in Lessons 3 and 4)
- Topics for Lesson 2
  - Quality attenuation (ΔQ)
  - Bottom-up and top-down design
  - Outcome diagrams
  - Example system designs

2.1 Quality attenuation (ΔQ)

2.1.A Introduction

Quality attenuation (ΔQ)



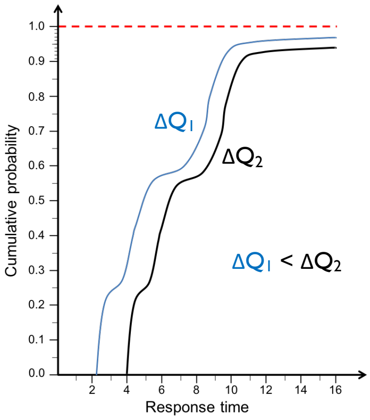
- Message m_{in} enters component A and m_{out} exits
- How do we characterize the message traveling through A?
  - The latency between entry and exit: delay value (a number)
  - The message might be dropped: chance of failure (a percentage)
  - The latency is not always the same for all messages: jitter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

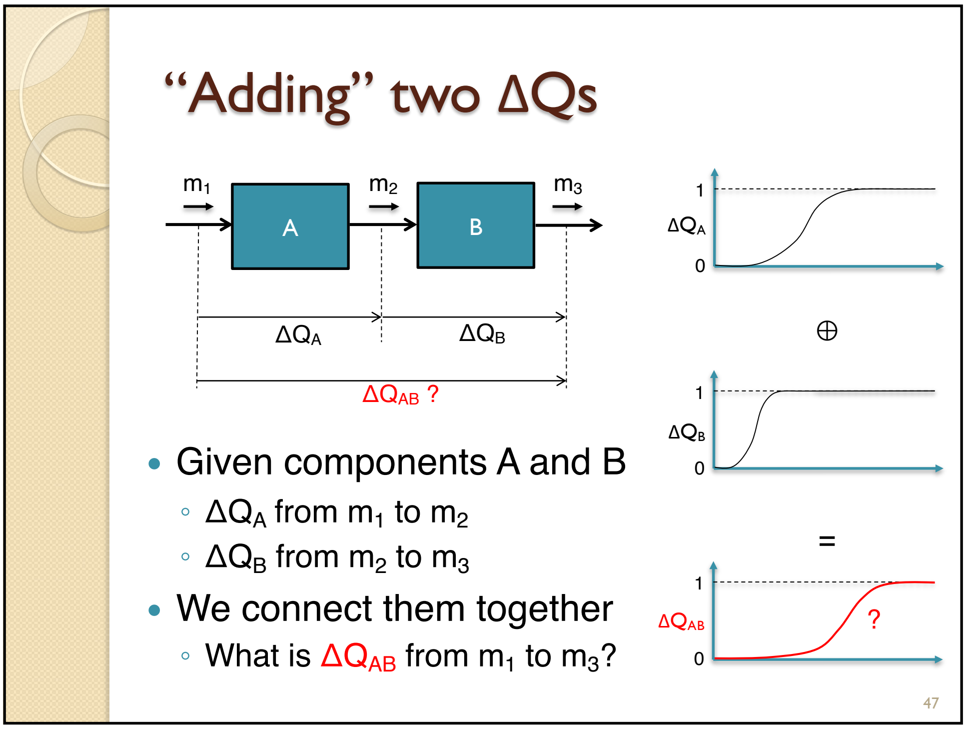
Combining delay and failure

- Delay and failure are combined in one quantity ΔQ
  - Two parts of system design that are usually separate are considered together
  - This lets us examine trade-offs between delay and failure
- 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

Comparing ΔQs

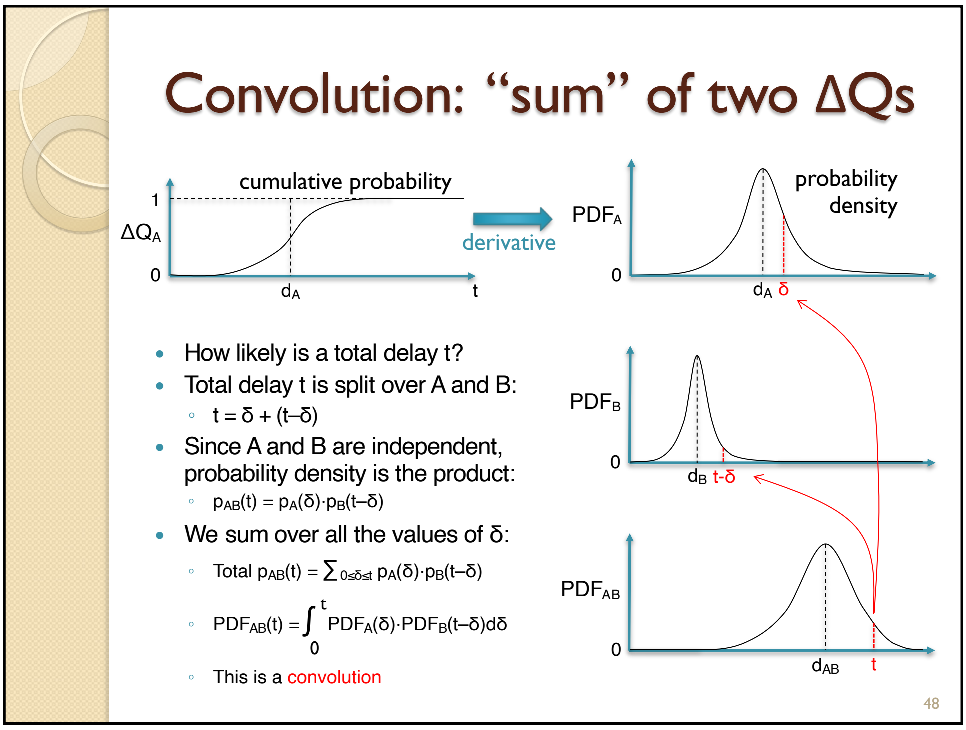
- We can compare two ΔQs: one is *less than* the other if its CDF is everywhere to the left and above the other
  - Mathematically, this relation between two ΔQs is a *partial order*
  - If the ΔQs intersect then they are not ordered
- A system satisfies its specification if the 'delivered ΔQ' is less than the 'required ΔQ'





“Adding” two ΔQs

- Given components A and B
  - ΔQ_A from m_1 to m_2
  - ΔQ_B from m_2 to m_3
- We connect them together
  - What is ΔQ_{AB} from m_1 to m_3?



Convolution: “sum” of two ΔQs

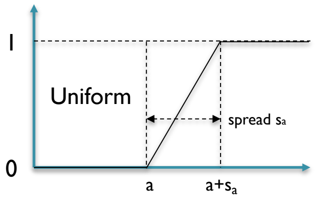
- How likely is a total delay t?
- Total delay t is split over A and B:
  - $t = \delta + (t-\delta)$
- Since A and B are independent, probability density is the product:
  - $p_{AB}(t) = p_A(\delta)\cdot p_B(t-\delta)$
- We sum over all the values of δ:
  - Total $p_{AB}(t) = \sum_{0\le\delta\le t} p_A(\delta)\cdot p_B(t-\delta)$
  - $PDF_{AB}(t) = \int_0^t PDF_A(\delta)\cdot PDF_B(t-\delta)d\delta$
  - This is a convolution

2.1.B Designing with ΔQ

Designing 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

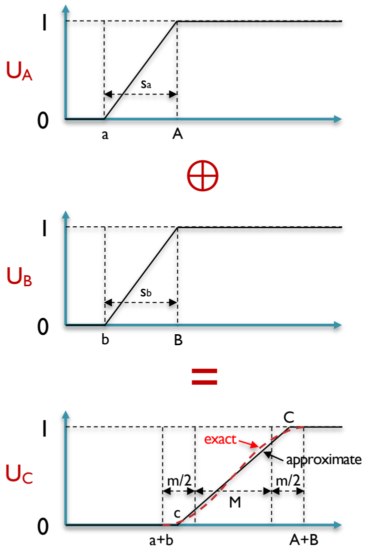
Uniform distribution



- 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

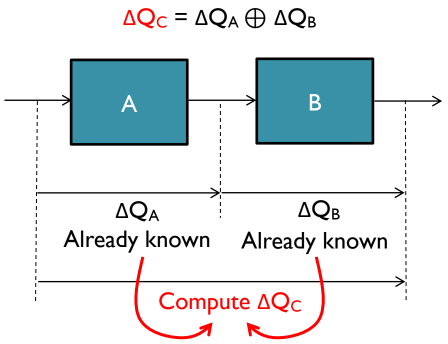
- For our two examples, we use a Uniform distribution for ΔQ
  - It is one of the simplest distributions and it is useful in practice: many components have approximately a uniform distribution
  - Uniform distributions are good for "back-of-the-envelope" ΔQ computations; an automated tool can of course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

Convolution of Uniform distributions



- Formulas: (approximate)
  - $U_A = (a,s_a)$
    $U_B = (b,s_b)$
    $U_C = U_A \oplus U_B = (c,s_c)$
  - $M = \max(s_a , s_b)$
    $m = \min(s_a , s_b)$
  - $c = (a + b) + m/4$
    $C = (A + B) - m/4$
    $s_c = \max(s_a , s_b) + m/2$
- In other words:
  - Starting times are added, plus a little more
  - Spread is the maximum of the spreads, plus a little more
- 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

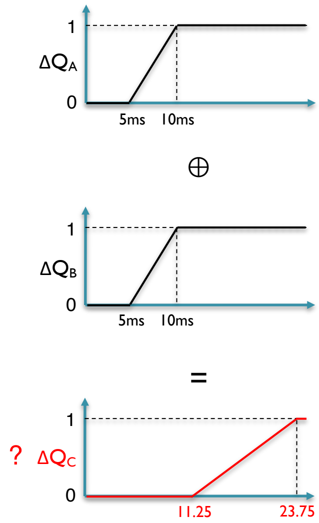
Bottom-up design with ΔQ



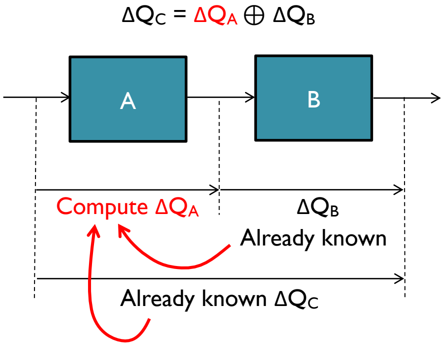
- Component A has ΔQ_A and component B has ΔQ_B
  - What is ΔQ_C?
- We assume Uniform distributions for A and B and "add" them to get C:
  - Assume (a,s_a) and (b,s_b)
  - We can approximate (c,s_c):
    $c = (a + b) + m/4$
    $s_c = \max(s_a,s_b) + m/2$
    where $m=\min(s_a,s_b)$
  - Overall delay c is a bit more than the sum of the two delays
  - Overall spread s_c is a bit wider than the worst spread

Numerical bottom-up example

- We know A and B
  - $a=5$, $s_a=5$
  - $b=5$, $s_b=5$
  - $m=\min(s_a,s_b)=5$
- Compute for C:
  - $c = (a+b)+m/4 = 11.25$ms
  - $s_c = \max(s_a,s_b)+m/2 = 12.5$ms
- Note bigger c and s_c!
  - $c = 11.25$ not 10
  - $s_c = 12.5$ not 10



Top-down design with ΔQ

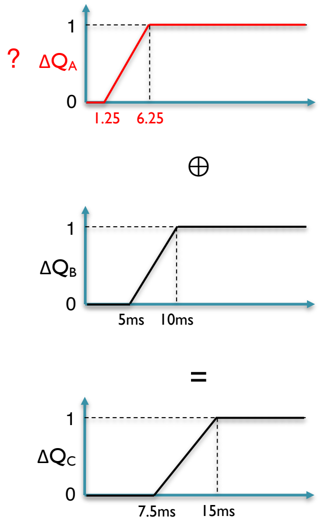


$\Delta Q_C = \Delta Q_A \oplus \Delta Q_B$

- 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”:
    - $a \le (c - b) - m/4$
        - Remember that $m=\min(s_a,s_b)$
        - A’s delay must be less than c–b
    - If $s_a \le s_b$ then $s_a \le 2(s_c - s_b)$
      If $s_a > s_b$ then $s_a \le s_c - s_b/2$
        - This follows from $\max(s_a,s_b)=s_c - m/2$

Numerical top-down example

- We know B and C
- Assume $s_a \approx s_b$
    - $b=5$, $s_b=5$
    - $c=7.5$, $s_c=7.5$
    - $m=\min(s_a,s_b)=5$
- Compute for A:
    - $a \le (c-b)-m/4 = 1.25$ms
    - $s_a \le 2(s_c - s_b) = 5$ms
- Note strict bound on A!
    - a=1.25 not 2.5



Infeasibility check for top-down

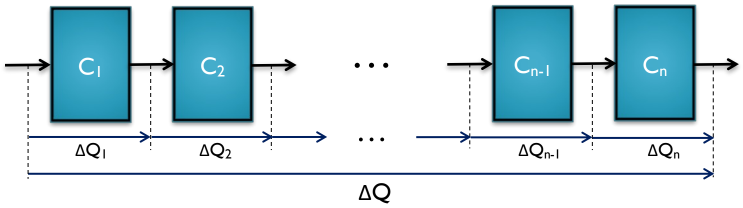
- Let us compute the conditions on B and C for feasibility
  - If they are not satisfied, then no component A is possible so the design is certainly infeasible!
- We start with two simultaneous equations in (a,s_a):
  $c = a + b + \min(s_a,s_b)/4$
  $s_c = \max(s_a,s_b) + \min(s_a,s_b)/2$
- We solve this by distinguishing two cases
- First, assume $s_a \leq s_b$:
  $s_a = 2(s_c - s_b) > 0$ which implies $s_c > s_b$ [1]
  $a = (c-b)-(s_c-s_b)/2 > 0$ which implies $(c-b) > s_c/2-s_b/2$ [2]
- Second, assume $s_a > s_b$:
  $s_a = s_c - s_b/2 > 0$ which implies $s_c > s_b/2$ [3]
  $a = c-b-s_b/4 > 0$ which implies $(c-b) > s_b/4$ [4]
- The design is infeasible if (¬[1] ∨ ¬ [2]) ∧ (¬[3] ∨ ¬[4])
  Basically, each of the two cases is impossible

"Subtracting" Uniform distributions

- When doing top-down design, we do the opposite of addition
  - Mathematically, we are doing deconvolution which is much harder to compute 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
    - It is a really good use of computation power to help a system designer
- 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

2.1.C Diagnosing with ΔQ

Diagnosing with ΔQ



- Consider a pipeline of components that has a bad overall ΔQ
  - This happens often in practice, e.g., the small cells case study
- Since adding a component can only make ΔQ get worse, we can find the faulty component(s) by binary search
- This technique can be generalized to follow the path of messages through the system

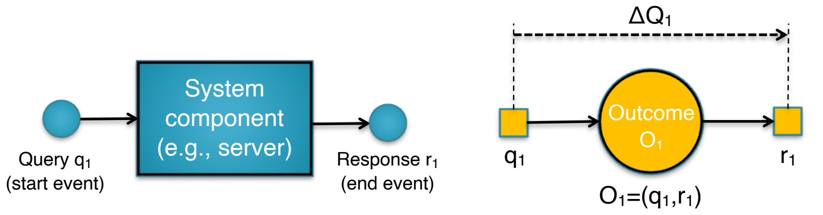
2.2 Outcome diagrams

2.2.A Introduction

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
    - UML diagrams define what happens inside the system being modelled
  - Outcome diagrams say nothing about system state
- They are extremely useful
  - Many different kinds of component can be brought together, software, humans, mechanical devices
  - They allow estimating performance and feasibility early on in the design process

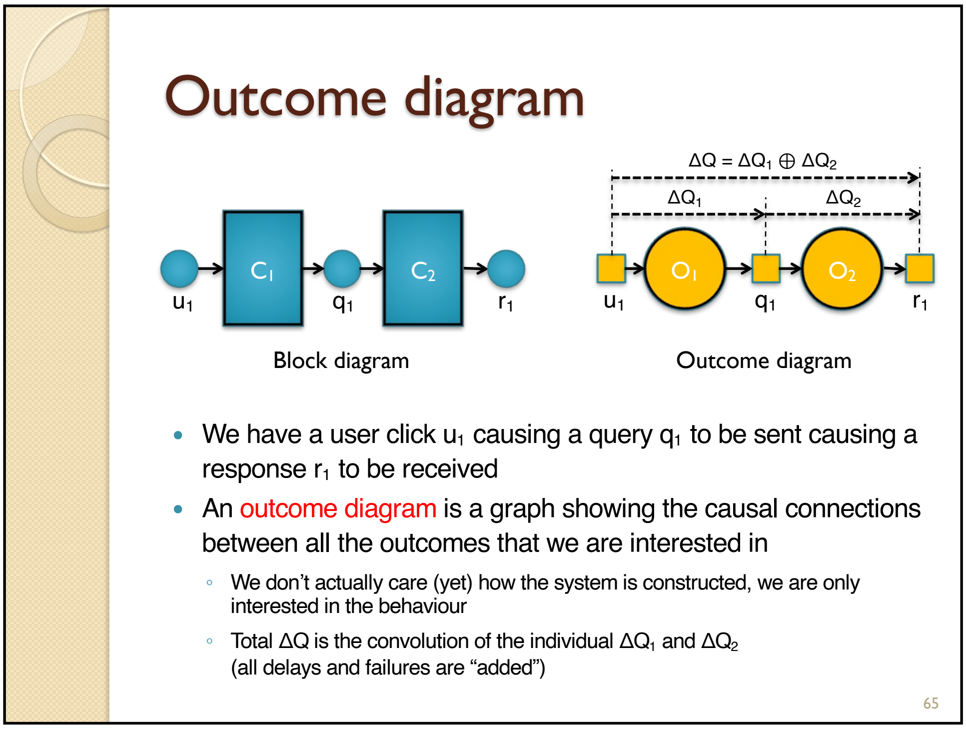
Single outcome



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 a pair 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

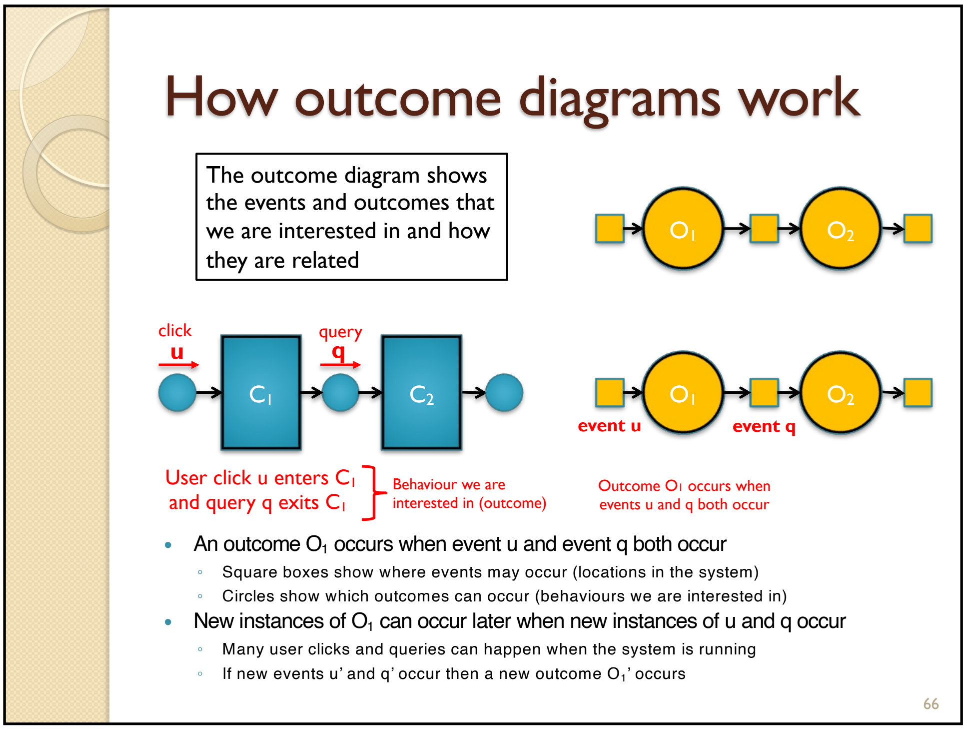


Outcome diagram

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")



How outcome diagrams work

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

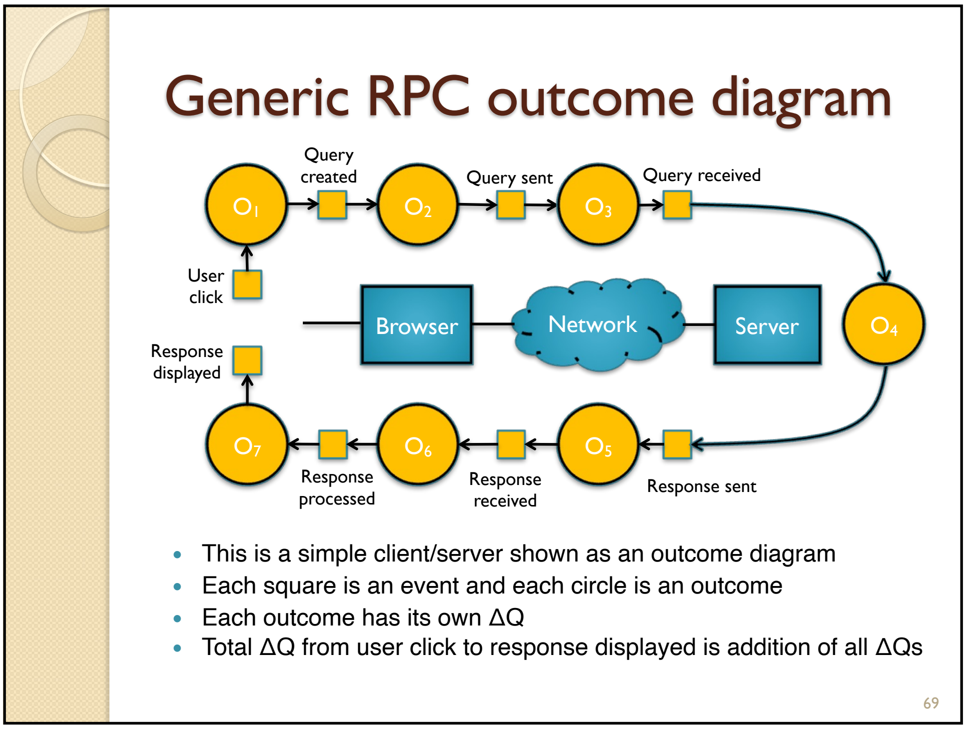
User click u enters C_1 and query q exits C_1 } Behaviour we are interested in (outcome)

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

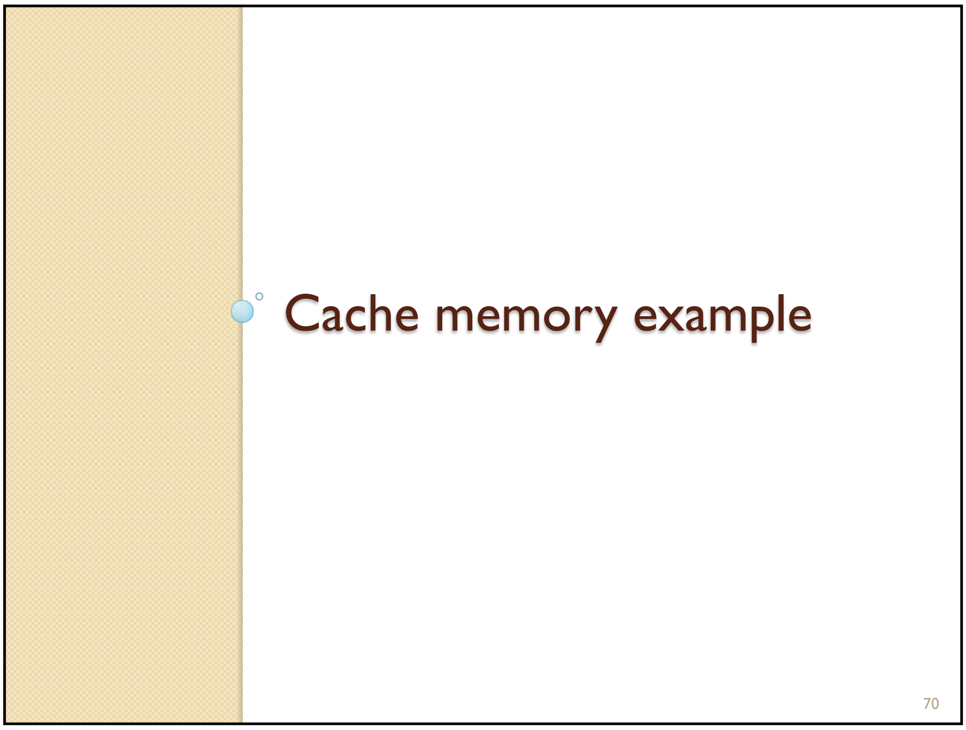
2.2.B Examples

Client/server example



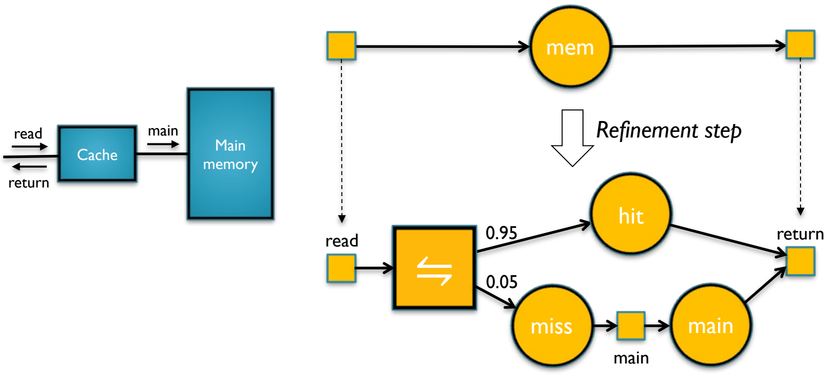
Generic RPC outcome diagram

- This is a simple client/server shown as an 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



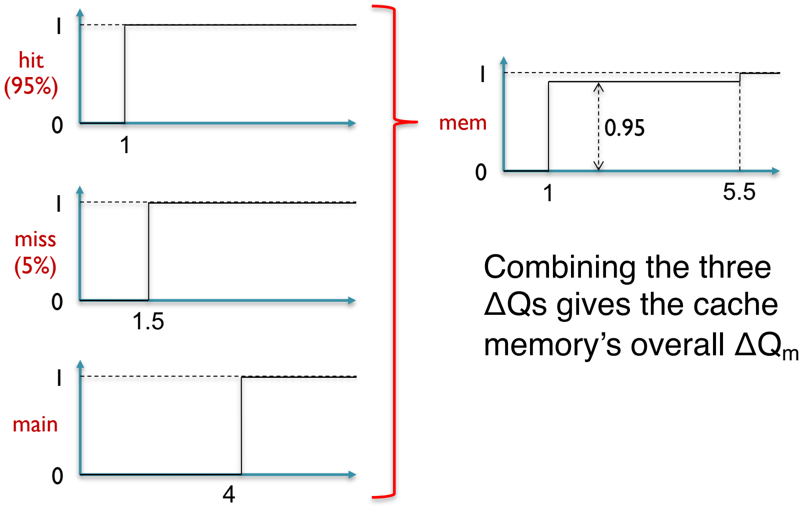
Cache memory example

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

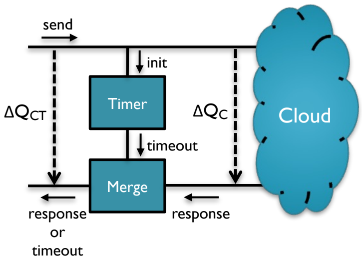
Cache quality attenuation

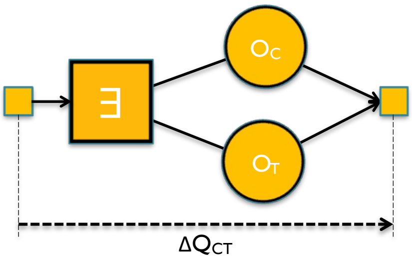


Combining the three ΔQs gives the cache memory's overall ΔQ_{mem}

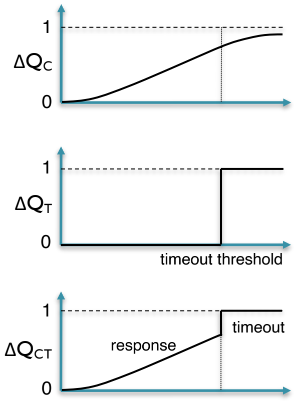
Timeout example

Timeout example



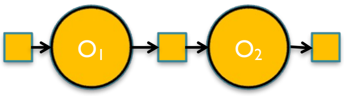


- Timeout is modeled using first-to-finish
- Assume a send request to “Cloud” that waits for a response or a timeout
- This gives: $\Delta Q_{CT} = \Delta Q_C + \Delta Q_T - \Delta Q_C \times \Delta Q_T$



Basic operators to build systems

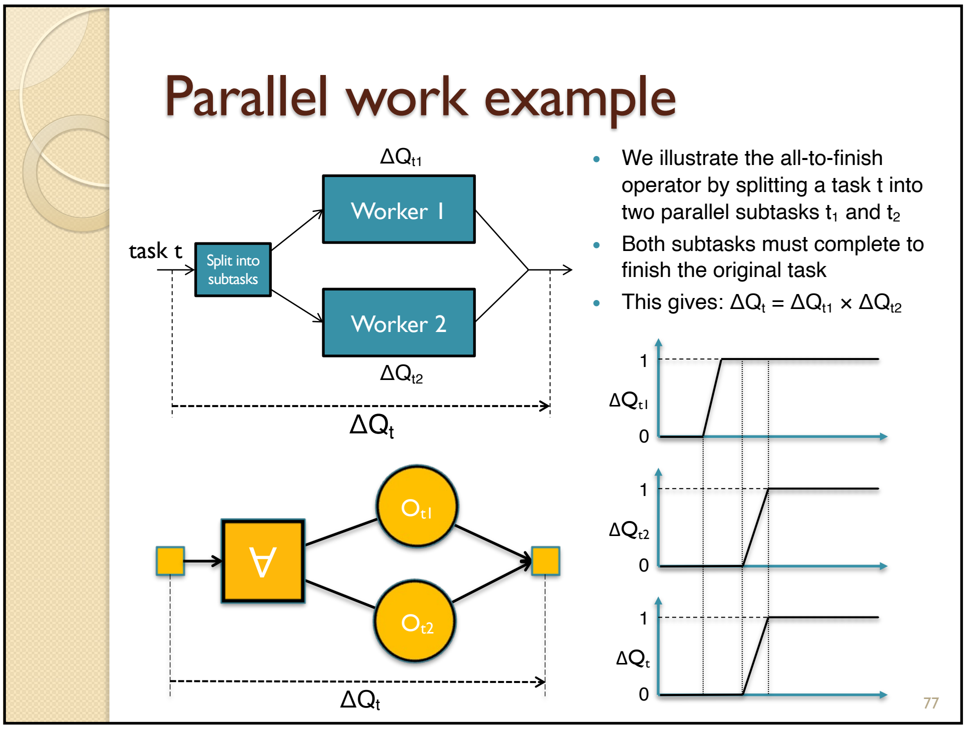
- To connect two components, we used a system operator
  - Sequential composition



- To build the cache and the timeout, we used two more system operators
  - Probabilistic choice ⇋
  - First-to-finish ∃
- We need one more operator
  - All-to-finish ∀

Parallel work example

Parallel work example



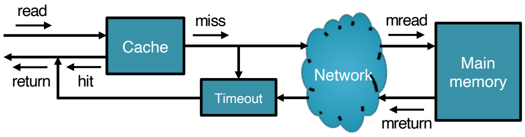
- We illustrate the all-to-finish operator by splitting a task t into two parallel subtasks t_1 and t_2
- Both subtasks must complete to finish the original task
- This gives: $\Delta Q_t = \Delta Q_{t1} \times \Delta Q_{t2}$

Networked memory example

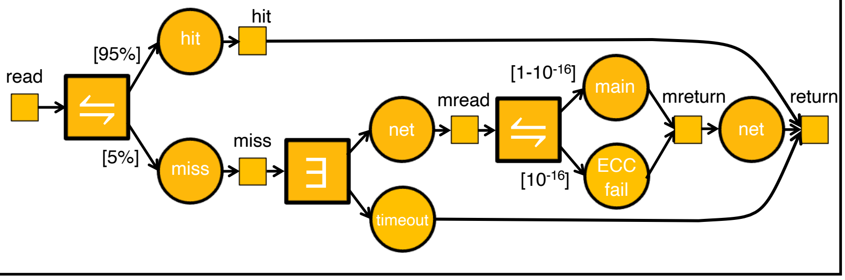
Networked memory example

- We can now define a bigger system:
  - A networked memory with local cache and timeout
  - As exercise, compute the ΔQ of this system

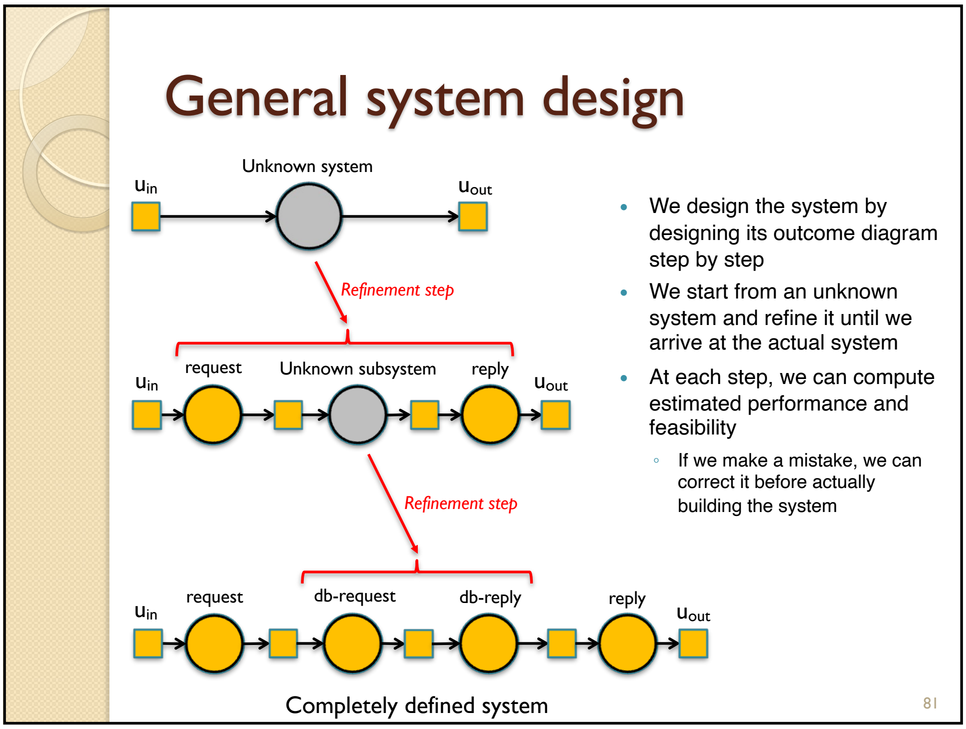
Block diagram



Outcome diagram



2.2.C General system design



General system design

Unknown system

u_{in} u_{out}

Refinement step

request Unknown subsystem reply

u_{in} u_{out}

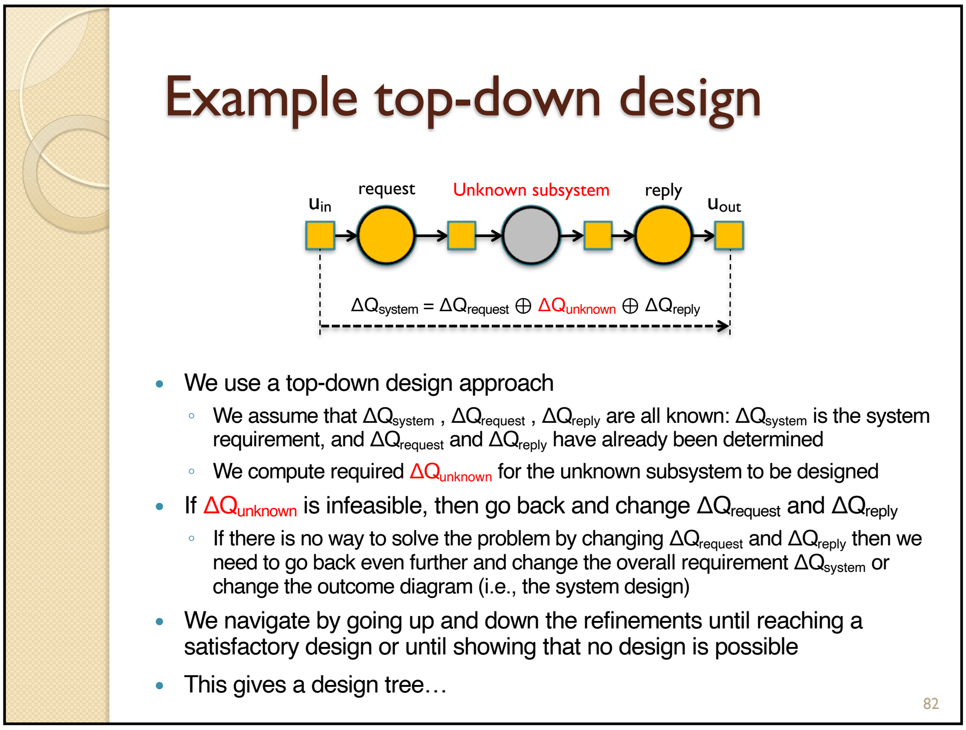
Refinement step

request db-request db-reply reply

u_{in} u_{out}

Completely defined system

- We design the system by designing its 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



Example top-down design

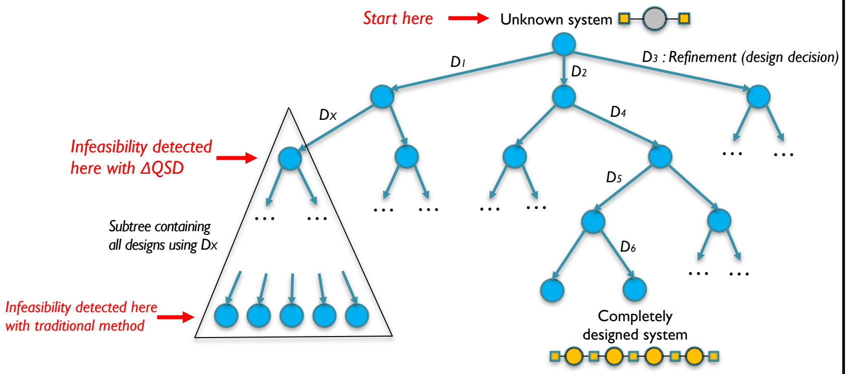
request Unknown subsystem reply

u_{in} u_{out}

$\Delta Q_{system} = \Delta Q_{request} \oplus \Delta Q_{unknown} \oplus \Delta Q_{reply}$

- 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…

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

2.3 Lab 2: Designing with ΔQSD

Lesson 3

- Systems with coupling
- Theory of ΔQSD

3.1 Introduction

Systems with coupling

- ΔQSD approach is done in three steps
  - First, design the system with independent parts (lesson 2)
  - ➡ Second, add couplings where they are needed (lesson 3)
  - Third, add multiple levels and multiple timescales (lesson 4)
- Realistic systems have some coupled parts
  - Most of the system consists of independent parts
  - A few couplings are added, for example where two message streams use the same database
- Topics for Lesson 3
  - Shared resources
  - Dependencies (iterative query example)
  - Theory of ΔQSD

Nonlinearity

- The outcome diagrams of Lesson 2 describe systems with independent components
  - Overall ΔQ is a linear combination of the component ΔQs
    - It is compositional: we can decompose the outcome diagram into parts, compute ΔQs separately, and then combine them
  - Nevertheless, ΔQ of a component is a nonlinear function of the load
    - There is an overflow effect: when offered load goes beyond capacity, delay and failure rate of a component increase quickly
- Lesson 3 adds even more nonlinearity
  - Adding dependencies between components is another source of nonlinearity
  - A shared resource can overflow and cause a major change in ΔQ
- Lesson 4 studies overload and how to handle it
  - Temporary overload and permanent overload

3.2 Shared resources

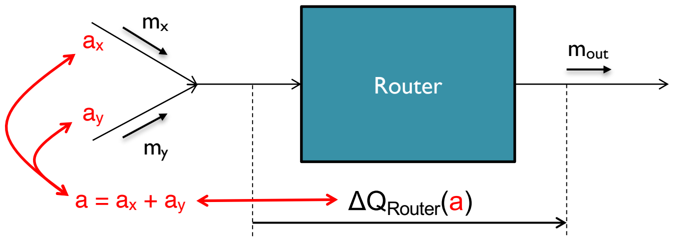
Resource properties

- Computing ΔQ is simple if all components are independent
    - This is the default compositional approach we saw so far
- But real systems have shared resources
    - A resource is part of the system that can potentially be shared
    - Sharing is modeled by additional variables and their equations
    - Computing ΔQ is done by adding the equations to the solver
- Two key resource properties
    - Ephemeral: A resource is *ephemeral* if it is available at a particular time instant and if not used at that time, it is lost.
    - Threshold: A resource is *threshold* if exceeding a particular limit causes a ΔQ to become bottom (failure: no result). If there is still some functionality, it is not a threshold resource.

Kinds of shared resources

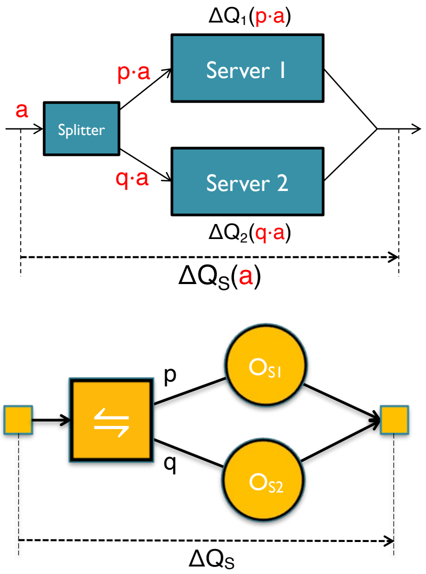
- Ephemeral, not threshold
    - (1) A network connection. When capacity of the line is exceeded or there is congestion, the ΔQ has larger failure rate, but it still works.
    - (2) A shared CPU. When too many processes use same CPU, they slow down but still keep going.
- Ephemeral, threshold
    - (1) Working set of a process. When size of working set exceeds maximum memory available, system will thrash and effectively stops.
    - (2) Mains electricity at an outlet. When too much power is drawn, a circuit breaker trips and power is zero.
- Not ephemeral, not threshold
    - Tidal energy generator with battery storage. Battery is charged periodically, can always take energy from battery. Battery energy goes down until next charge cycle.
- Not ephemeral, threshold
    - Battery power supply. Battery can supply energy at any time, until it runs out (total energy needed exceeds energy stored in battery).

Example 1: congestion



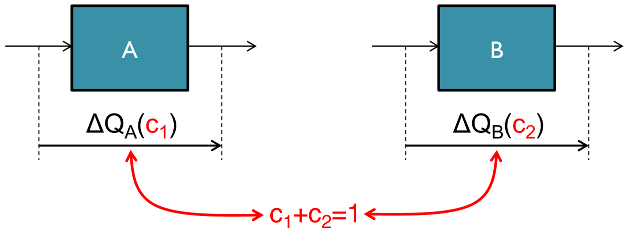
- Assume two message streams entering the same component (e.g., a router)
    - Total load is the sum of the two incoming loads: $a = a_x + a_y$
    - Sharing is modeled as the sum of loads
- Congestion, i.e., buffer overflow and message drop, is computed from ΔQ_{Router} using the queue model we saw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

Example 2: load balancing



- We illustrate the queue model by doing load balancing
- Load a is split between $p\cdot a$ and $q\cdot a$ for the two servers
  - Modeled with probabilistic choice
  - Servers have equal capacity with normalized load $a=1$, so $p=q=0.5$
- All quality attenuations are function of load
- We have the equation:
  $\Delta Q_S(a) = p\cdot \Delta Q_1(p\cdot a) + q\cdot \Delta Q_2(q\cdot a)$
- For good performance, both servers must never be overloaded, which gives:
  - $p\cdot a < 0.8$ and $q\cdot a < 0.8$
  - This results in $a < 1.6$
- This example can be extended in many ways, for example to divide packets into low and high priority

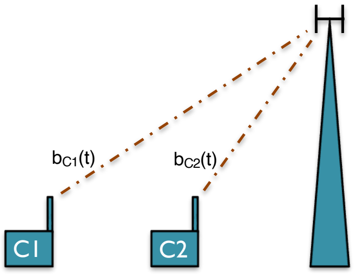
Example 3: shared CPU



- 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

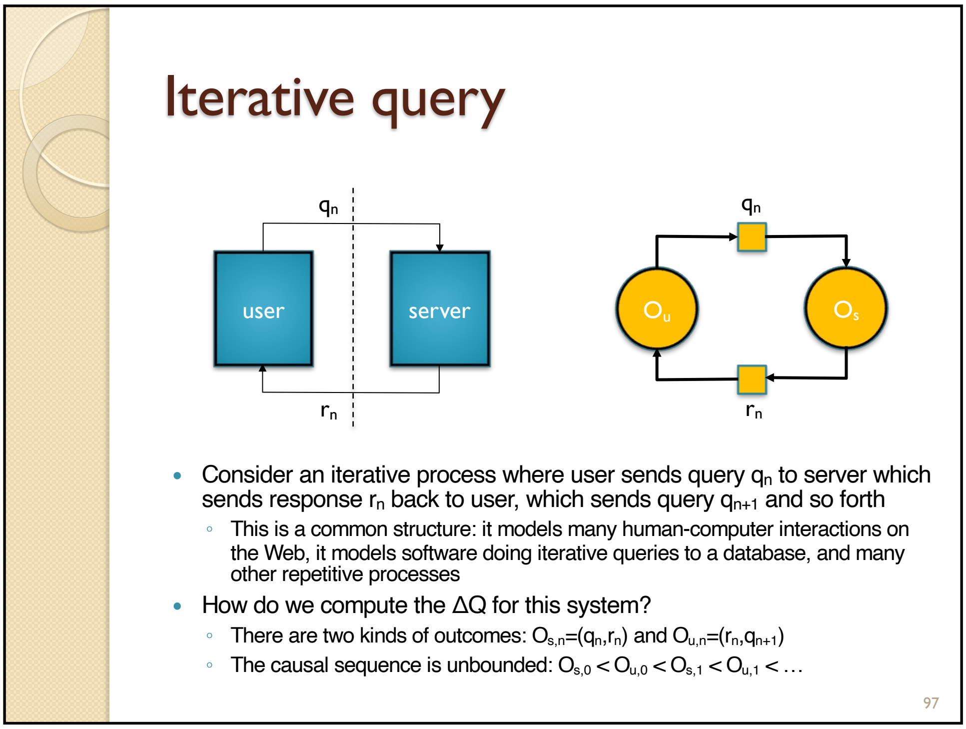
Example 4: shared cell tower

- Two carriers share a cell tower
- Each carrier is guaranteed at least 50% of the tower capacity
- Each is allowed to use the capacity unused by the other



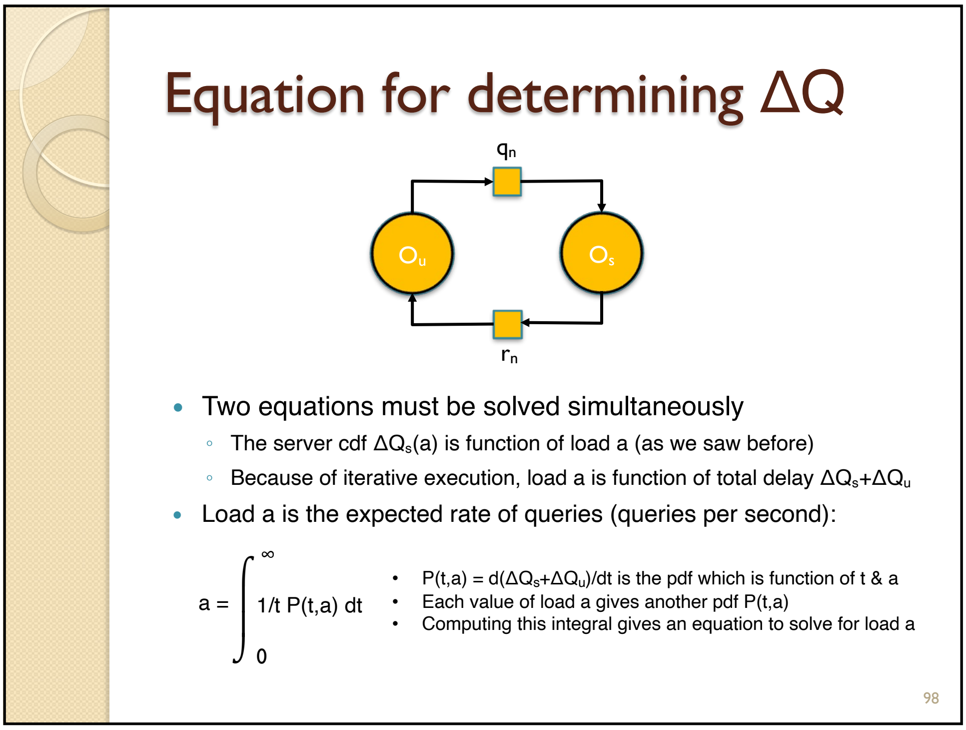
- Assume carrier 1 uses 30% and carrier 2 uses 60% of the capacity
- If carrier 1 suddenly decides to use 50% then carrier 2 will immediately drop to 50% and some packets it has in transit will be lost
- This is a nonlinear resource dependency:
  - Each carrier requests loads $a_{C1}(t)$ and $a_{C2}(t)$ with condition $0 \leq a_{C1}(t), a_{C2}(t) \leq 1$
  - Shared tower grants loads $b_{C1}(t)$ and $b_{C2}(t)$, total tower load is $b_{C1}(t)+b_{C2}(t)$
  - If $a_{C1}(t) \leq 0.5$ then { $b_{C1}(t)=a_{C1}(t)$; if $a_{C2}(t) \geq 0.5$ then $b_{C2}(t)=\min(a_{C2}(t),1-a_{C1}(t))$ }
  - If $a_{C2}(t) \leq 0.5$ then { $b_{C2}(t)=a_{C2}(t)$; if $a_{C1}(t) \geq 0.5$ then $b_{C1}(t)=\min(a_{C1}(t),1-a_{C2}(t))$ }
  - If $a_{C1}(t)>0.5$ and $a_{C2}(t)>0.5$ then { $b_{C1}(t)=0.5$; $b_{C2}(t)=0.5$ }

3.3 Dependencies: iterative query



Iterative query

- 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: $O_{s,0} < O_{u,0} < O_{s,1} < O_{u,1} < \ldots$



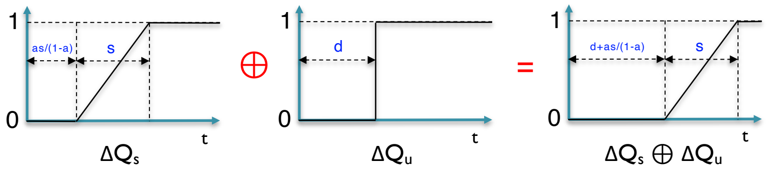
Equation for determining ΔQ

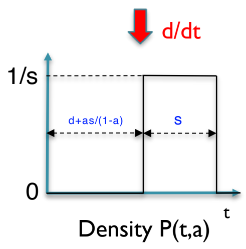
- Two equations must be solved simultaneously
  - The server cdf $\Delta Q_s(a)$ is function of load a (as we saw before)
  - Because of iterative execution, load a is function of total delay $\Delta Q_s+\Delta Q_u$
- Load a is the expected rate of queries (queries per second):

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

- $P(t,a) = d(\Delta Q_s+\Delta Q_u)/dt$ 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

Solving the equation



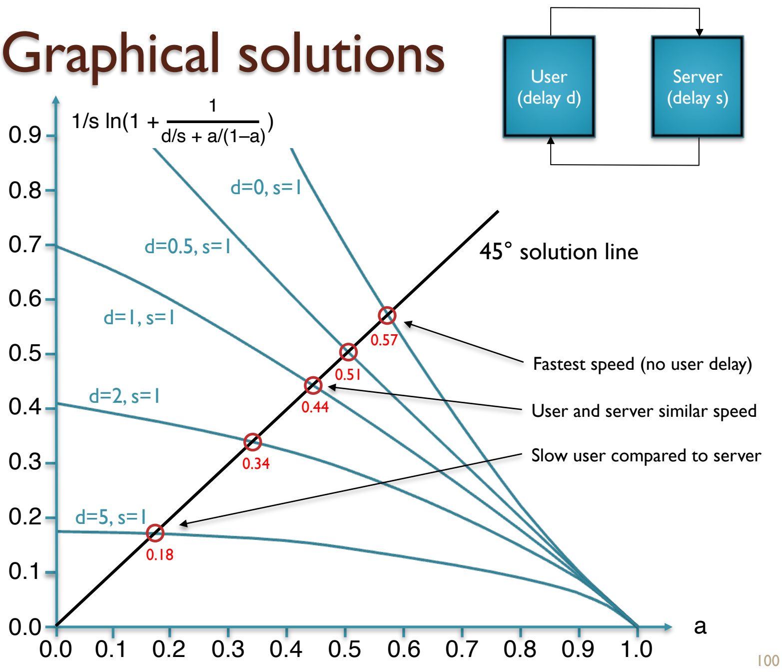


- Working out the integral gives:

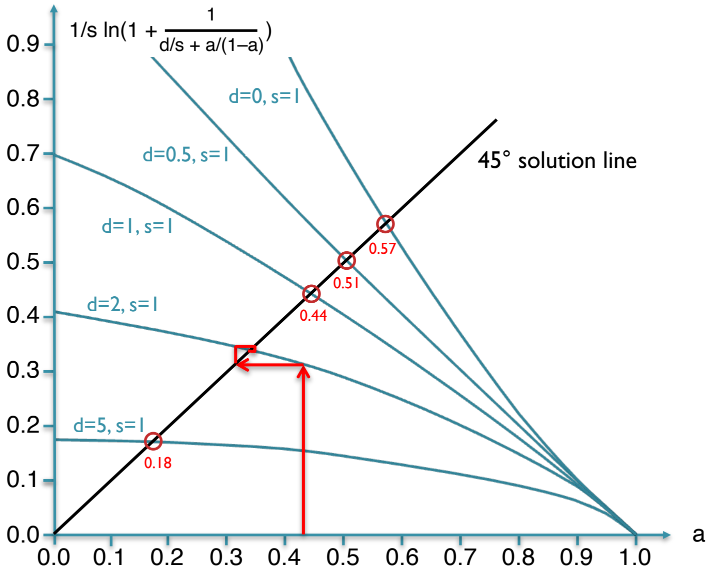
  $a = 1/s \ln(1 + \frac{1}{d/s + a/(1-a)})$

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

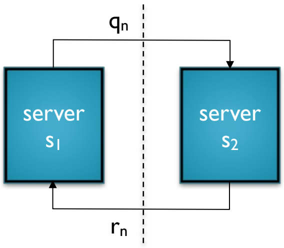
Graphical solutions



Graphical iteration



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

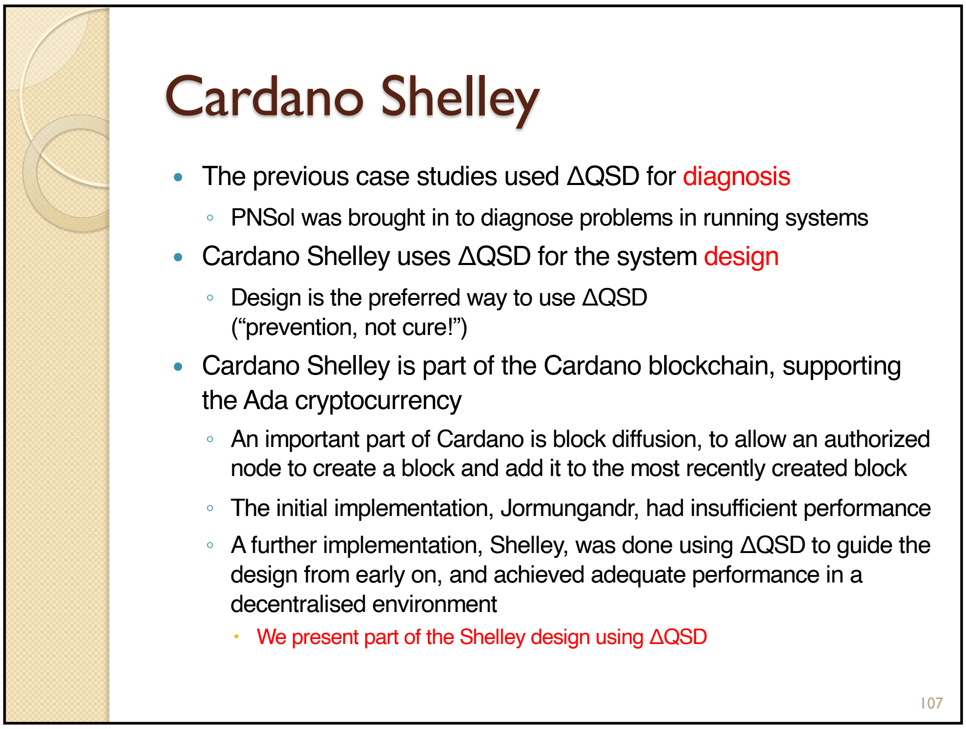
3.4 Theory of ΔQSD

3.5 Lab 3:
More design with ΔQSD

Lesson 4

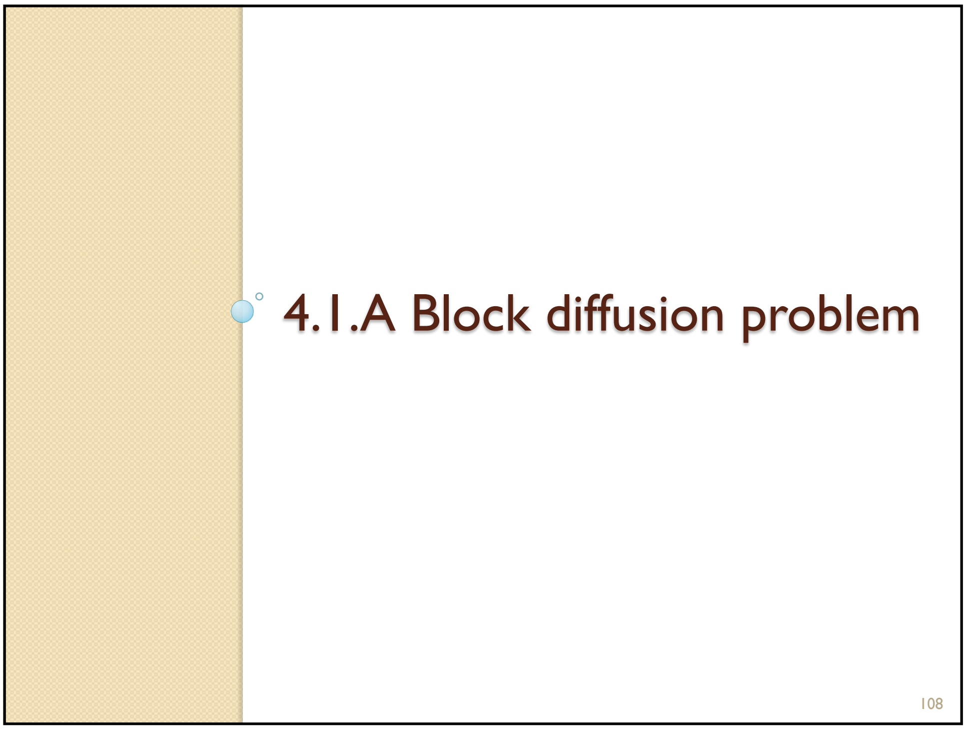
- Cardano Shelley
- Handling overload
- Conclusions

4.1 Cardano Shelley design example



Cardano Shelley

- The previous case studies used ΔQSD for diagnosis
  - PNSol was brought in to diagnose problems in running systems
- Cardano Shelley uses ΔQSD for the system design
  - Design is the preferred way to use ΔQSD
    (“prevention, not cure!”)
- Cardano Shelley is part of the Cardano blockchain, supporting the Ada cryptocurrency
  - An important part of Cardano is block diffusion, to allow an authorized node to create a block and add it to the most recently created block
  - The initial implementation, Jormungandr, had insufficient 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 present part of the Shelley design using ΔQSD



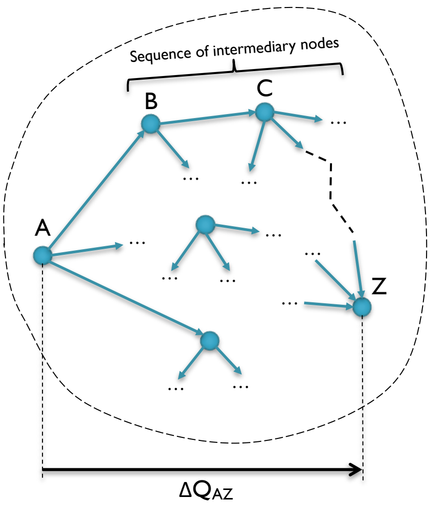
4.1.A Block diffusion problem

Context of block diffusion

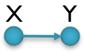
- Blockchain management in Cardano
    - We will use ΔQSD to solve a design problem in the Cardano blockchain, which is an open-source platform using proof of stake
    - A blockchain is a distributed ledger comprising a chain of data blocks that are cryptographic witnesses to correctness of preceding blocks
        - Ledger = A book in which financial transactions are recorded
    - 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 the most recent 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

Block diffusion problem statement

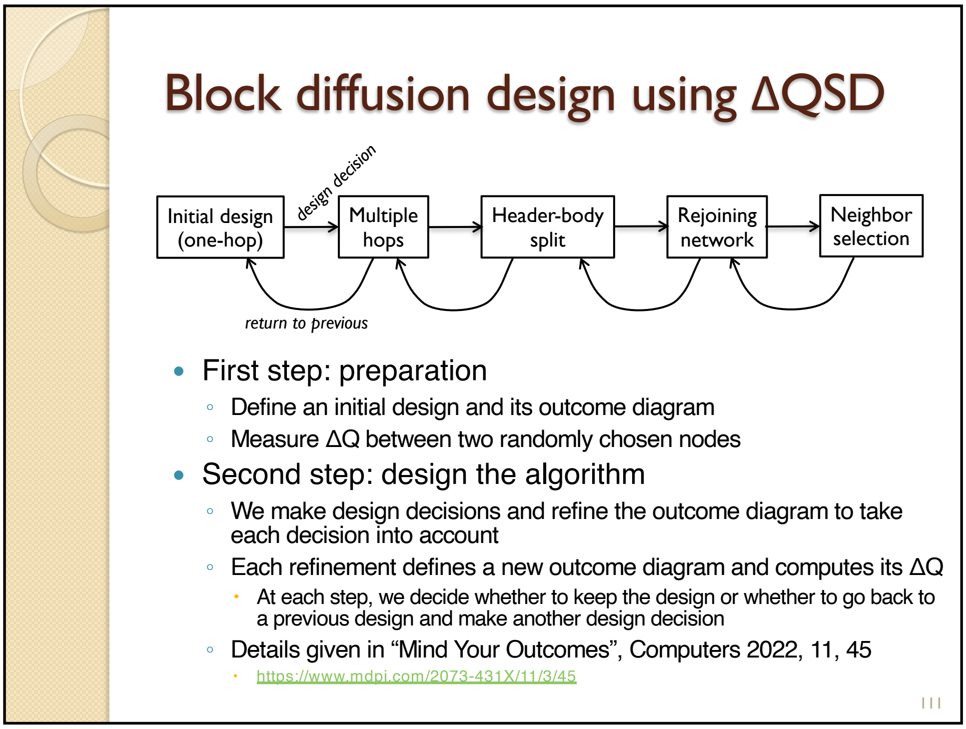
Node graph of Cardano blockchain



- 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



Block diffusion design using ΔQSD

Initial design (one-hop)

design decision

Multiple hops

Header-body split

Rejoining network

Neighbor selection

return to previous

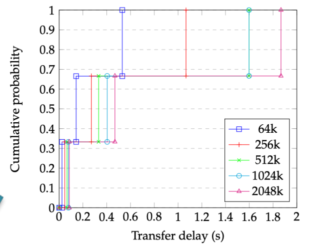
- First step: preparation
  - Define an initial design and its outcome diagram
  - Measure ΔQ between two randomly chosen nodes
- Second step: design the algorithm
  - We make design decisions and refine the outcome diagram to take each decision into account
  - Each refinement defines a new outcome 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

4.1.B Measuring ΔQ

First step: measuring ΔQ

- First step is to measure ΔQ between two Internet nodes
  - 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)

Measured ΔQ for fixed paths

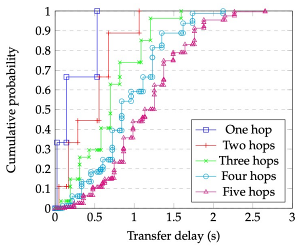


One-hop ΔQ per block size

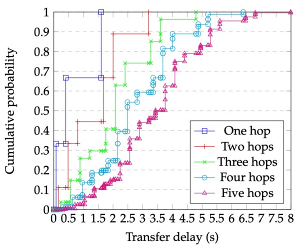
Weighted sum of short, medium, and long hops, computed using probabilistic choice operator

Multi-hop computed using convolution

Multi-hop computed using convolution

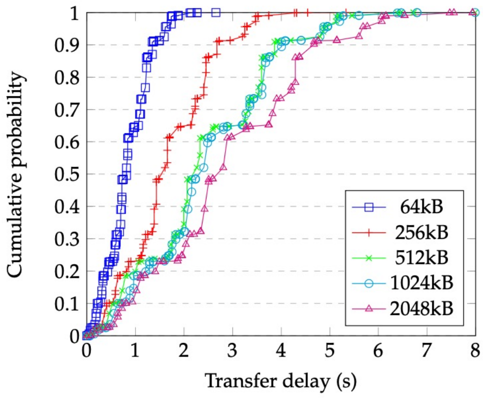


Multi-hop ΔQ for 64KB



Multi-hop ΔQ for 1024KB

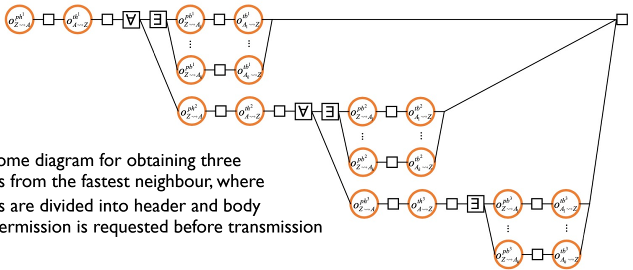
Measured ΔQ for varying paths



- 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

4.1.C Designing with the outcome diagram

Second step: design process

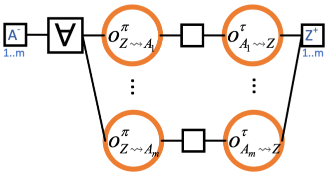


Outcome diagram for obtaining three blocks from the fastest neighbour, where blocks are divided into header and body and permission is requested before transmission

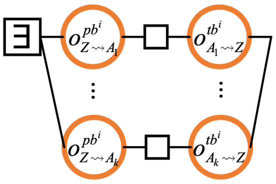
- For each design decision
  - Determine a new outcome diagram
  - Evaluate the effectiveness (ΔQ) using the outcome diagram
- This leads step by step to a final outcome diagram, which corresponds to the complete distributed system
  - Let us explain one of the steps, namely obtaining several blocks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

Obtaining three blocks (1)

All-to-finish operator

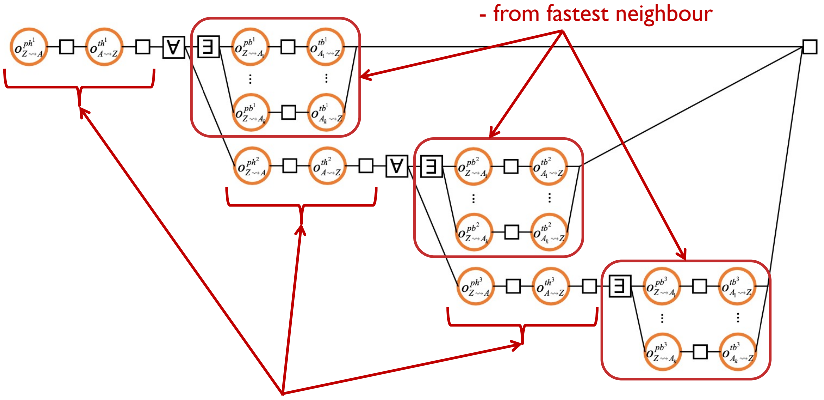


First-to-finish operator



- We remind you of 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 (∃)

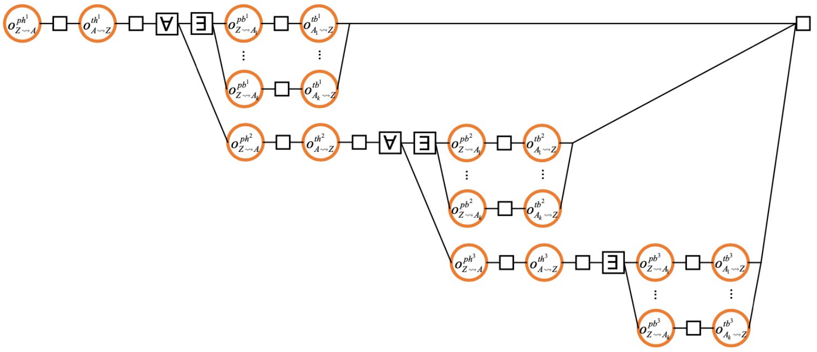
Obtaining three blocks (2)



Obtain one block body
- from fastest neighbour

Obtain three blocks in order:
- permission request before transmission authorized
- header obtained before body
- body and next block combined using ∀

Obtaining three blocks (3)



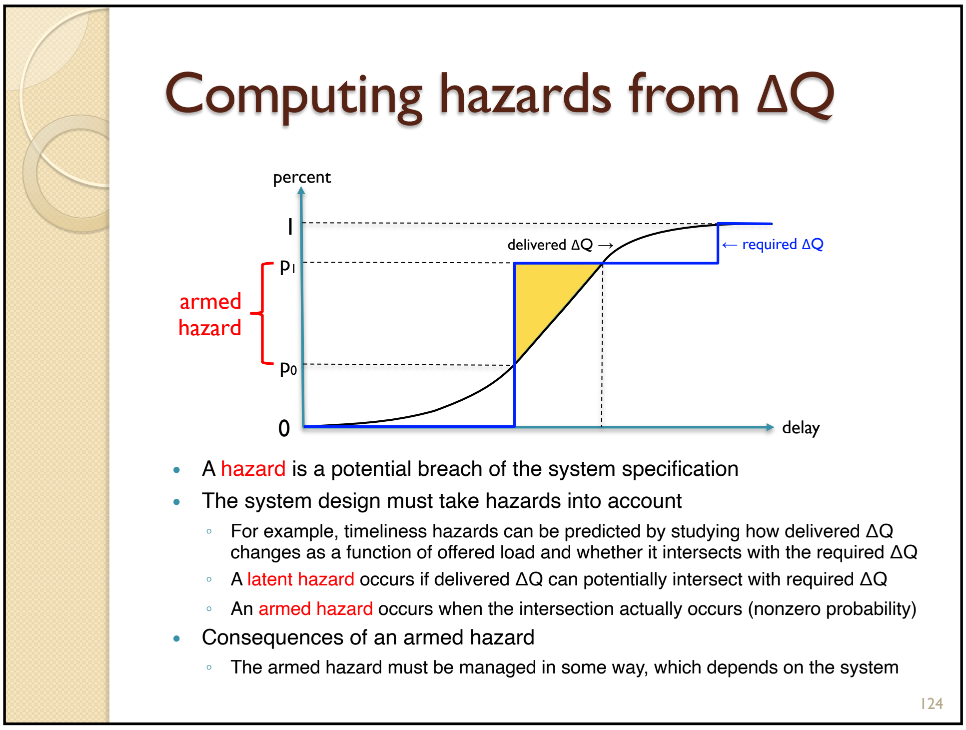
- The resulting outcome diagram correctly models the causality and performance of the block transfer; ΔQ is easily computed
- The outcome diagram is complex but it can be simplified by introducing abstractions
- A software tool would have no problem with it, of course

4.2 Handling overload

Handling overload

- The system is designed for a maximum load
    - As a prudent designer, you overdimension the system
    - This does not solve the problem of overload!
    - There will always be overload, so you must design for it!
        - In fact, if you overdimension, the problem can be worse since users will assume the system is much more capable than it is
- The question is then: how to design your system to be predictable when overload happens
- There are two cases that can occur:
    - Temporary overloads: system must react in a reasonable way (discussed in this section)
    - Permanent (long-lasting or repeated) overloads: this is a timescale issue and must be handed to the next level (discussed in the next section)

4.2.A Hazard management



Computing hazards from ΔQ

- A hazard is a potential breach of the system specification
- The system design must take hazards into account
  - For example, timeliness hazards can be predicted by studying how delivered ΔQ changes as a function of offered load and whether it intersects with the required ΔQ
  - A latent hazard occurs if delivered ΔQ can potentially intersect with required ΔQ
  - An armed hazard occurs when the intersection actually occurs (nonzero probability)
- Consequences of an armed hazard
  - The armed hazard must be managed in some way, which depends on the system

Overload and timeliness hazards

- Overload is a key enabler of timeliness hazards
  - Delivered $\Delta Q_{del}(a)$ is a time-dependent function of offered load a
  - High load causes ΔQ_{del} to get worse, which can arm the hazard (ΔQ_{del} intersects ΔQ_{req})
  - Design for overload needs to be seen in the light of possible armed hazards and how they are managed

- We define an order for performance hazards
  - 0: Causality (system respects causal order)
  - 1: Capacity (expected average load, linear behaviour)
  - 2: Schedulability (expected load variability)
  - 3: Behaviour (internal correlations)
  - 4: Stress (external correlations)

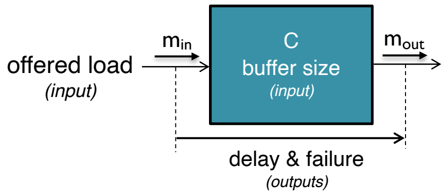
$\left.\begin{matrix}3\\4\end{matrix}\right\}$ Overload dependent

4.2.B System model

A simple system model

- For our analysis we define a simple system model
  - A model simple enough to analyse easily and complex enough to correspond to reality
  - We have used many system models in the past, and a simple model gives the correct intuition
- A system is a component from queueing theory
  - We assume input consists of many independent tasks, which allows us to use probability theory for the analysis
- Note that ΔQSD can use a more general model
  - Outcome diagram: a directed graph that gives causal relationships between a system's basic operations
    - Outcome diagrams allow precise computation of system feasibility and performance at high load conditions including when there is coupling (dependencies or shared resources), but we do not need that precision

A system as a component



offered load
(input)

m_{in}

C
buffer size
(input)

m_{out}

delay & failure
(outputs)

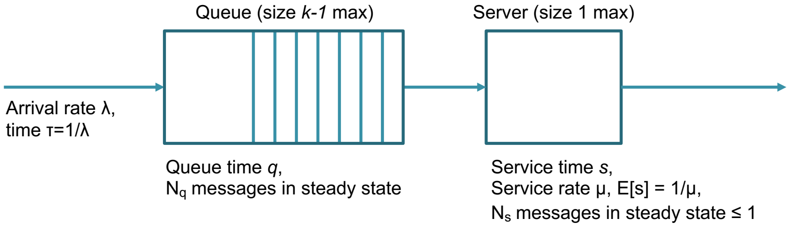
- We will model our system as a simple component
  - The component has a buffer (queue) and a server
- A typical component has four parameters of interest

Inputs
- Offered load *a* : arrival rate of messages (tasks)
- Buffer size *k* : number of messages stored inside

Outputs
- Delay *d* : time delay between input and output message
- Failure rate *f* : percentage of messages dropped

- Delay and failure are function of load and buffer size

A component as M/M/1/K queue



Queue (size *k-1* max)

Server (size 1 max)

Arrival rate λ,
time τ=1/λ

Queue time *q*,
N_q messages in steady state

Service time *s*,
Service rate μ, E[s] = 1/μ,
N_s messages in steady state ≤ 1

- 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 $a = \lambda/\mu$ (arrival rate / service rate, normalized arrival rate)
- The two knobs we control are offered load *a* and buffer size *k*
  - When the buffer is full, new arrivals are dropped (failure)

4.2.C Temporary overload

Temporary overload

- We are ambitious fellows
  - We are building a big system, bigger than anything that has been built before!
- We are experienced engineers
  - We use our experience to build the system: this is called inductive reasoning
  - But inductive reasoning is flawed
    - If we are not careful, it goes wrong when we go beyond where designer experience has gone before
- There is always a cliff
  - The system works perfectly until we fall off the cliff!
  - How can we design the system to be predictable and reasonable when it falls off the cliff?

Capacity races

- Your system suffers from temporary overloads
  - Solve it by increasing system capacity, a no-brainer, right?
  - Some big companies, who will remain unnamed, solve it by multiplying capacity by 2. With k levels of management, this gives a factor of 2^k overdimensioning. This does not solve the problem, but it may push its occurrence beyond where one can identify who is responsible.
- Surprise! This does not solve the problem.
  - When system capacity increases, users become more demanding. Users who never used video suddenly decide to use video. There is a large pent-up reservoir of demand.
  - It is like drug addiction. Increasing dosage gives temporary relief but does not solve the problem.
    - It actually makes the problem worse in the long term
- So what is the solution?
  - The solution is to design for temporary overload
  - How is it done?

Design rules

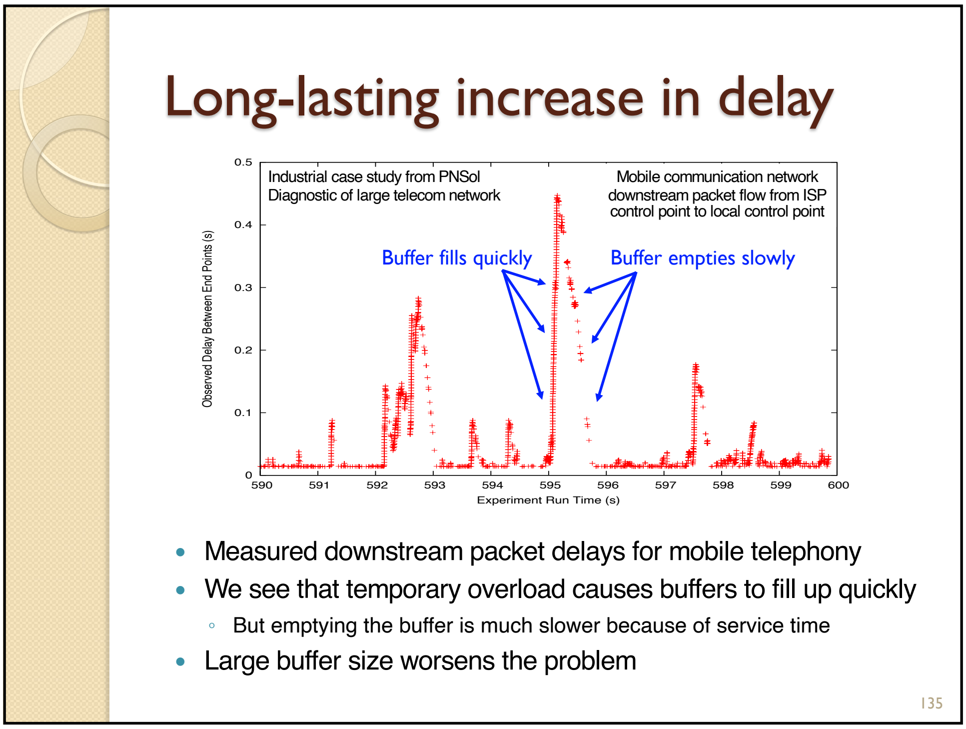
- Three key design rules:
  1. When overloaded, the system may behave badly but it must never break
     - When the overload goes away, the system goes back to normal
     - The system should be “ballistic”: be predictable (albeit bad) in open loop
  2. When overloaded, the system must provide some guaranteed minimum functionality
     - For example, high priority packets will pass
  3. When overloaded, the system is still accessible to management
     - There is a management interface where the behavior is observable and controllablekey

Effect of temporary overload

- Low load ($a<0.8$)
  - Component has enough power to service all messages
  - An underloaded component behaves very well

- High load ($a\geq0.8$)
  - Component is overloaded and will drop some messages
  - When $a \approx 1$ (starts around 0.8) things quickly get worse!
  - When a $\gg$ 1, failure rate tends to 100%, delay increases to k

- Switchover occurs between $a=0.5$ and $a=1$
  - As load increases beyond 0.5, the system quickly gets very bad
  - ⇨ A temporary overload gives a long-lasting increase in delay
  - Why is that?

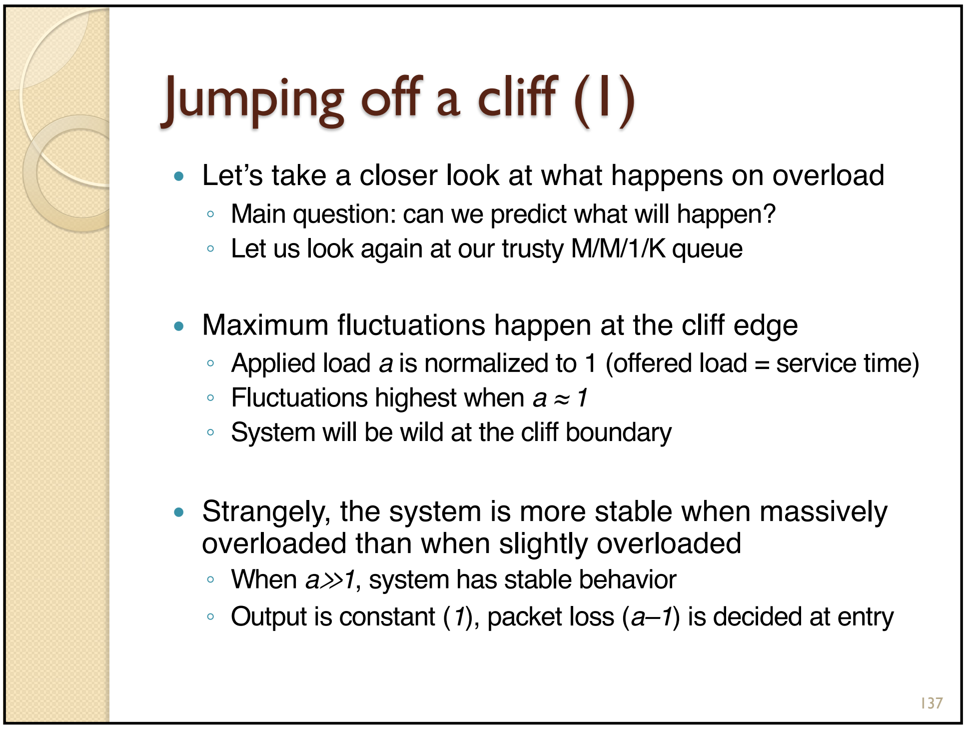


Long-lasting increase in delay

- Measured downstream packet delays for mobile telephony
- We see that temporary overload causes buffers to fill up quickly
  - But emptying the buffer is much slower because of service time
- Large buffer size worsens the problem

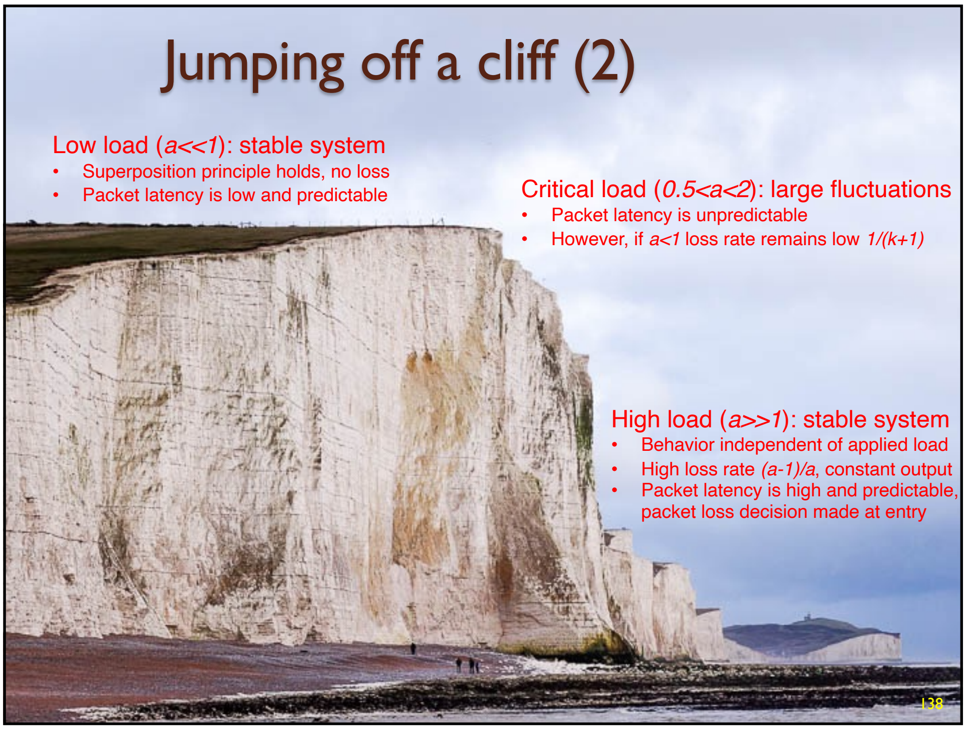
Tweaking load and buffer size

- The two main knobs we control are *a* and *k*
- Changing offered load *a*
  - When *a* approaches 1, the exponential distribution of interarrival times causes “bunches” of tasks to arrive which causes more and more temporary overloads, increasing both failure and delay
  - Decreasing *a* is the only way to reduce both failure and delay
  - During normal operation, *a* should be well below 1
- Changing buffer size *k*
  - Buffer size affects the trade-off of failure rate versus delay
  - Small buffer: increases failure rate and reduces delay
    - Typical scenarios: interactive video, gaming
  - Large buffer: reduces failure rate and increases delay
    - Typical scenarios: file download, data transmission



Jumping off a cliff (1)

- Let's take a closer look at what happens on overload
  - Main question: can we predict what will happen?
  - Let us look again at our trusty M/M/1/K queue
- Maximum fluctuations happen at the cliff edge
  - Applied load *a* is normalized to 1 (offered load = service time)
  - Fluctuations highest when $a \approx 1$
  - System will be wild at the cliff boundary
- Strangely, the system is more stable when massively overloaded than when slightly overloaded
  - When $a \gg 1$, system has stable behavior
  - Output is constant (1), packet loss ($a-1$) is decided at entry

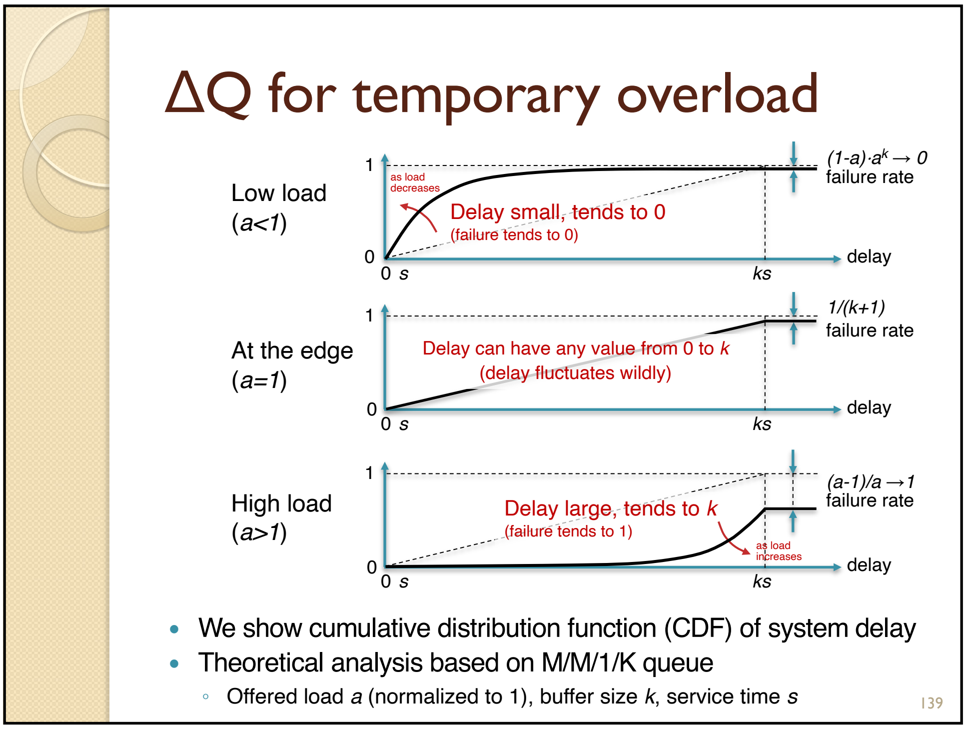


Jumping off a cliff (2)

Low load ($a \ll 1$): stable system
- Superposition principle holds, no loss
- Packet latency is low and predictable

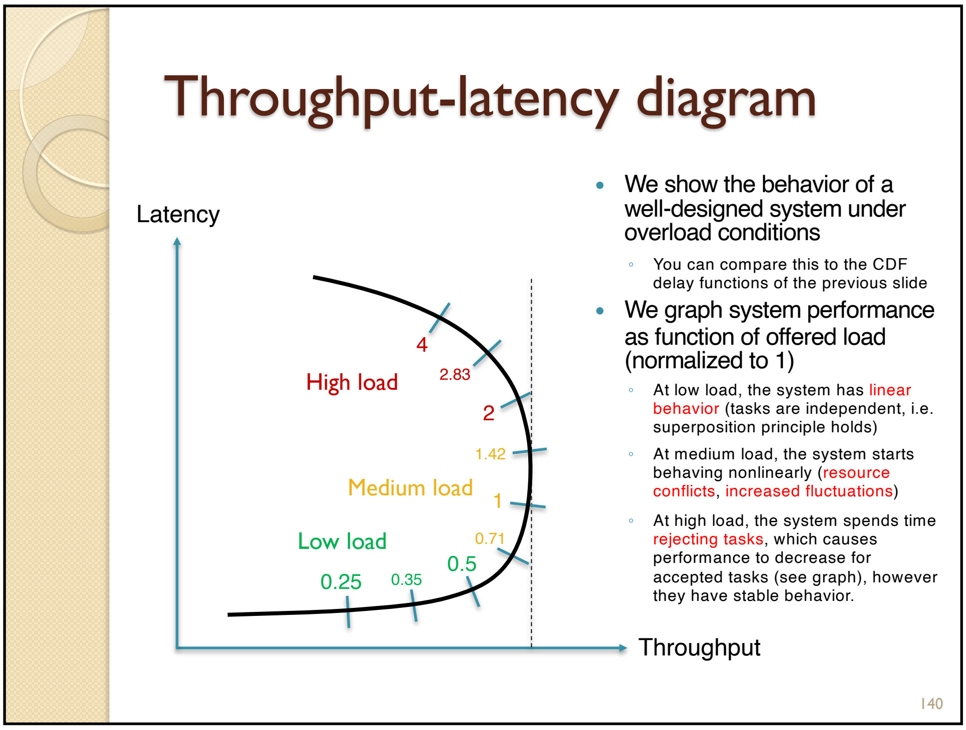
Critical load ($0.5<a<2$): large fluctuations
- Packet latency is unpredictable
- However, if $a<1$ loss rate remains low $1/(k+1)$

High load ($a \gg 1$): stable system
- Behavior independent of applied load
- High loss rate $(a-1)/a$, constant output
- Packet latency is high and predictable, packet loss decision made at entry



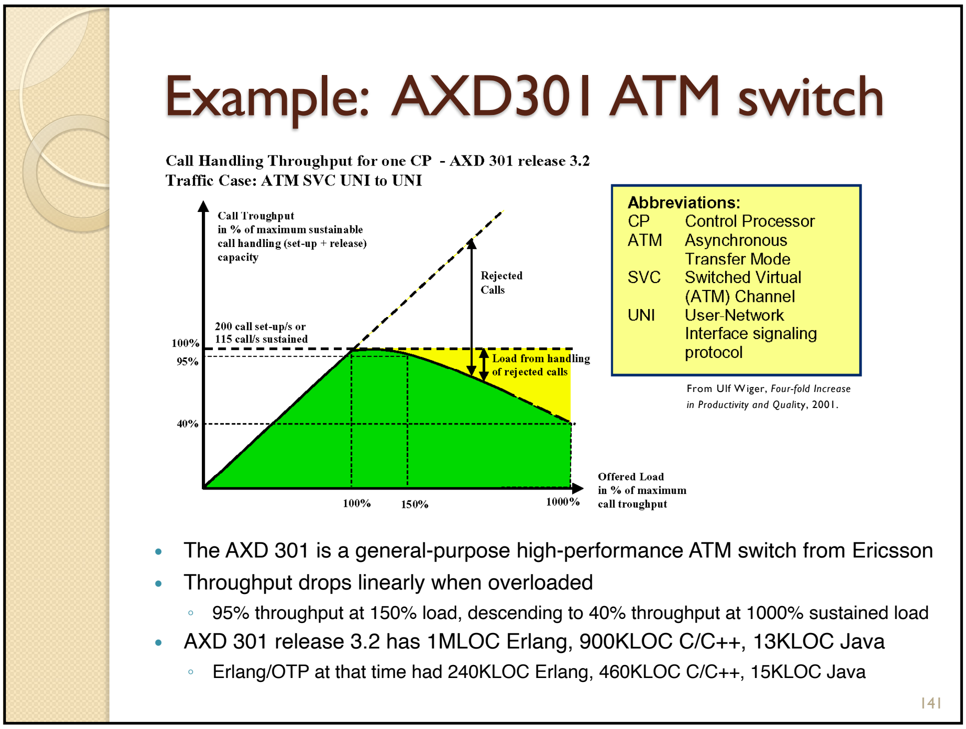
ΔQ for temporary overload

- We show cumulative distribution function (CDF) of system delay
- Theoretical analysis based on M/M/1/K queue
    - Offered load a (normalized to 1), buffer size k, service time s



Throughput-latency diagram

- We show the behavior of a well-designed system under overload conditions
    - You can compare this to the CDF delay functions of the previous slide
- We graph system performance as function of offered load (normalized to 1)
    - At low load, the system has linear behavior (tasks are independent, i.e. superposition principle holds)
    - At medium load, the system starts behaving nonlinearly (resource conflicts, increased fluctuations)
    - At high load, the system spends time rejecting tasks, which causes performance to decrease for accepted tasks (see graph), however they have stable behavior.



Example: AXD301 ATM switch

Abbreviations:

CP	Control Processor
ATM	Asynchronous Transfer Mode
SVC	Switched Virtual (ATM) Channel
UNI	User-Network Interface signaling protocol

From Ulf Wiger, *Four-fold Increase in Productivity and Quality*, 2001.

- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - Erlang/OTP at that time had 240KLOC Erlang, 460KLOC C/C++, 15KLOC Java

4.2.D Permanent overload

Permanent overload

- If your system suffers overload too often (repeated or long-lasting), we say that it has a permanent overload
  - Permanent overload is generally handled by adding resources to the system, but this is never a permanent solution!
- Multilevel system
  - Permanent overload is always possible, so there must be active mechanisms outside the system: a hierarchy of independent observers where each observer manages the hazards that are not managed at a lower level
  - The bottom level is the base system that we saw before
  - Each new level of hazard will have its own timescale
- "Mitigate or propagate"
  - It is important to know when to hand over control to a higher level
- Erlang/OTP supports this design approach
  - But it is difficult to keep the levels truly independent

Multilevel systems

Multilevel systems

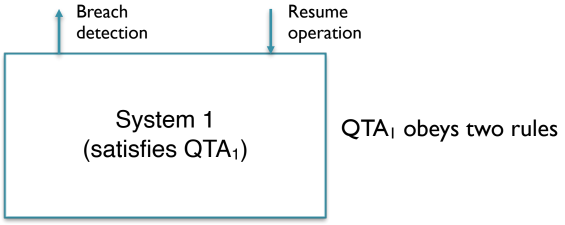
- ΔQSD approach is done in three steps
  - First, design the system with independent parts
  - Second, add dependencies where they are needed
  - ➡ Third, add multiple levels to make the system resilient
- Highly reliable systems need multiple levels
  - The main system will break when too much goes wrong
  - When this happens, control passes to another level
  - In general, widely different problems occur at different timescales, which need different solutions

Original system

Original system
(satisfies QTA)

- Our starting point is a system that satisfies a QTA
  - The QTA (Quantitative Timeliness Agreement) specifies what the base system does and its limits
  - In the previous section we saw how this system behaves under temporary overload
- Under temporary overload, the QTA may no longer be satisfied
  - The system no longer provides its contractual service to its users
  - Is the temporary overload actually a permanent overload?
- Multilevel design targets this situation

Lowest level


Breach detection

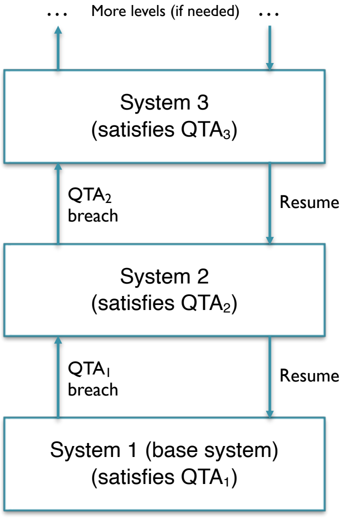
Resume operation

System 1 (satisfies QTA_1)

QTA_1 obeys two rules

- The original system becomes the lowest level in a multilevel system
  - To assume its new role, it must satisfy several new properties
- Management ability:
  - For breach detection, it must be extended with real-time observation points
  - For resuming operation, it must be extended with reconfiguration and restart
- QTA_1 properties: (used to minimize breaches)
  - When System 1 is overloaded, it may behave badly but it should not “break”, i.e., if the overload disappears the system recovers
  - When System 1 is overloaded, it provides a guaranteed minimum functionality
  - These properties are only possible if System 1 is not physically damaged; in the contrary situation control passes directly to a higher level

Higher levels


… More levels (if needed) …

System 3 (satisfies QTA_3)

QTA_2 breach

Resume

System 2 (satisfies QTA_2)

QTA_1 breach

Resume

System 1 (base system) (satisfies QTA_1)

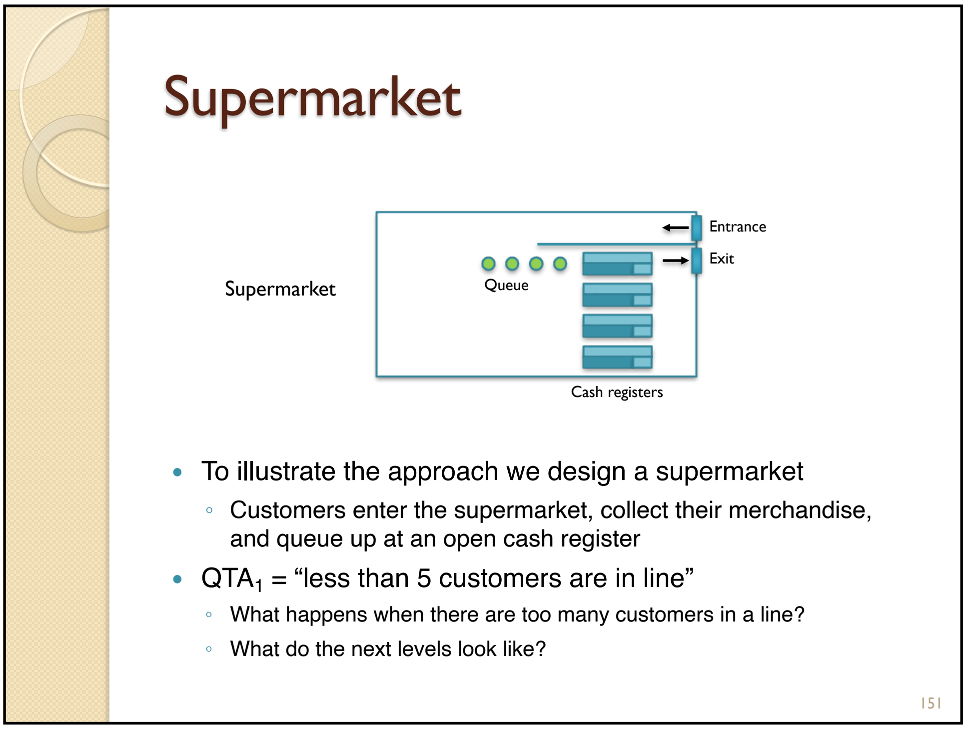
- During normal operation, System 1 does the work
  - The other systems monitor this but normally do not intervene
  - Upon QTA_1 breach, System 2 is notified
- System 2 has several options:
  - Take over from System 1, temporarily or permanently
  - Reconfigure System 1 and then resume it
  - Replace System 1 by another system and then resume it
- If System 2 cannot fix the problem then QTA_2 is breached
  - System 3 is notified and handles the problem at a higher level

Levels must be independent

Leonid Rogozov lying down talking to his friend Yuri Vereschagin at Novolazarevskaya Station
– BBC World Service, 5 May 2015

- Leonid Rogozov was a Soviet surgeon in Antarctica (1960-61)
  - He developed appendicitis during his stay and, since he was the only surgeon, he did his own appendectomy. He was awake and performed local anesthesia of the abdominal wall. He used a mirror to observe his insides and instructed assistants (a driver and a meteorologist) to provide instruments. Despite general weakness and nausea, the operation was complete in two hours. He resumed duties in two weeks.
- Needless to say this is not recommended!
  - The system that treats the fault must not be the same as the faulty system
  - Ideally, independence must be complete. The best existing system for this is Erlang.

Examples



Supermarket

Supermarket

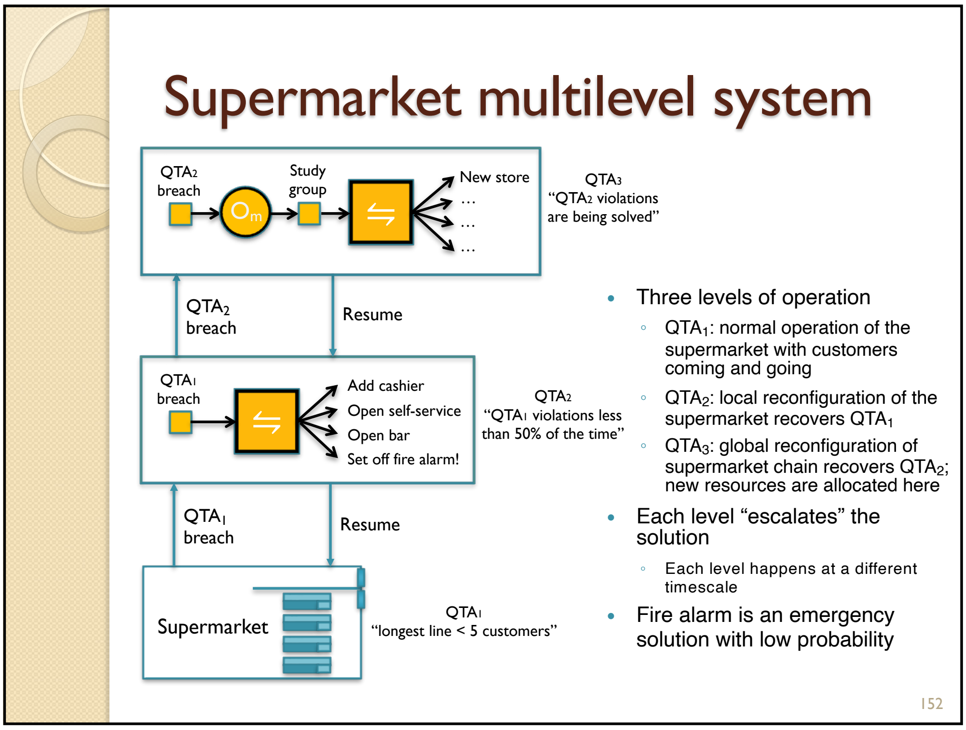
Entrance

Exit

Queue

Cash registers

- To illustrate the approach we design a supermarket
  - Customers enter the supermarket, collect their merchandise, and queue up at an open cash register
- QTA_1 = “less than 5 customers are in line”
  - What happens when there are too many customers in a line?
  - What do the next levels look like?



Supermarket multilevel system

QTA_2 breach

O_m

Study group

New store

…

…

…

QTA_3 “QTA_2 violations are being solved”

QTA_2 breach

Resume

QTA_1 breach

Add cashier

Open self-service

Open bar

Set off fire alarm!

QTA_2 “QTA_1 violations less than 50% of the time”

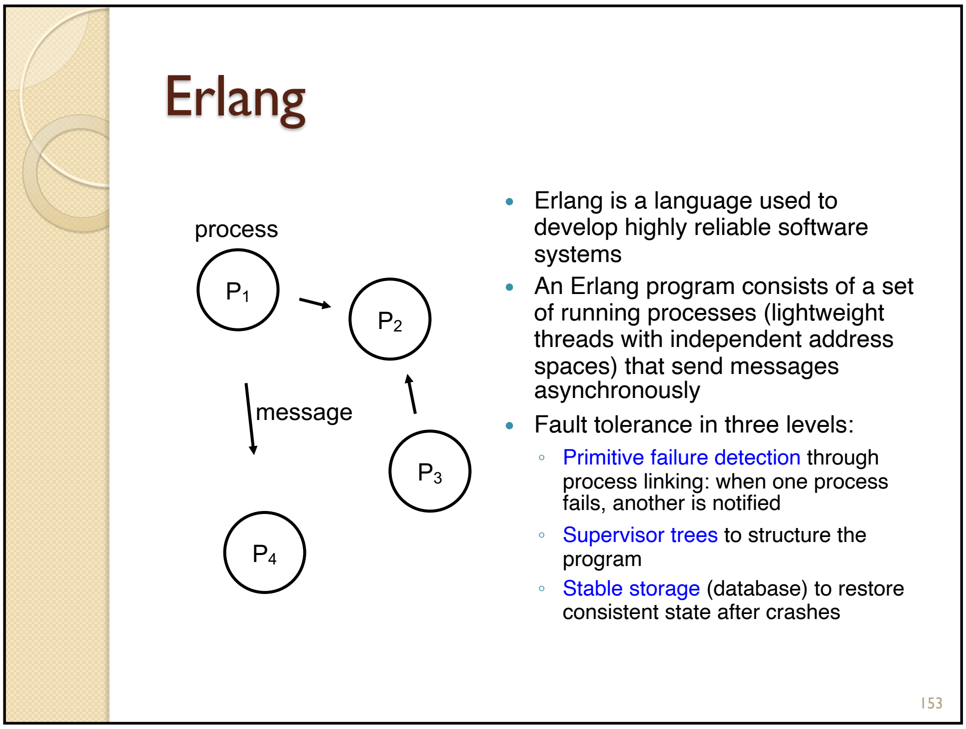
QTA_1 breach

Resume

Supermarket

QTA_1 “longest line < 5 customers”

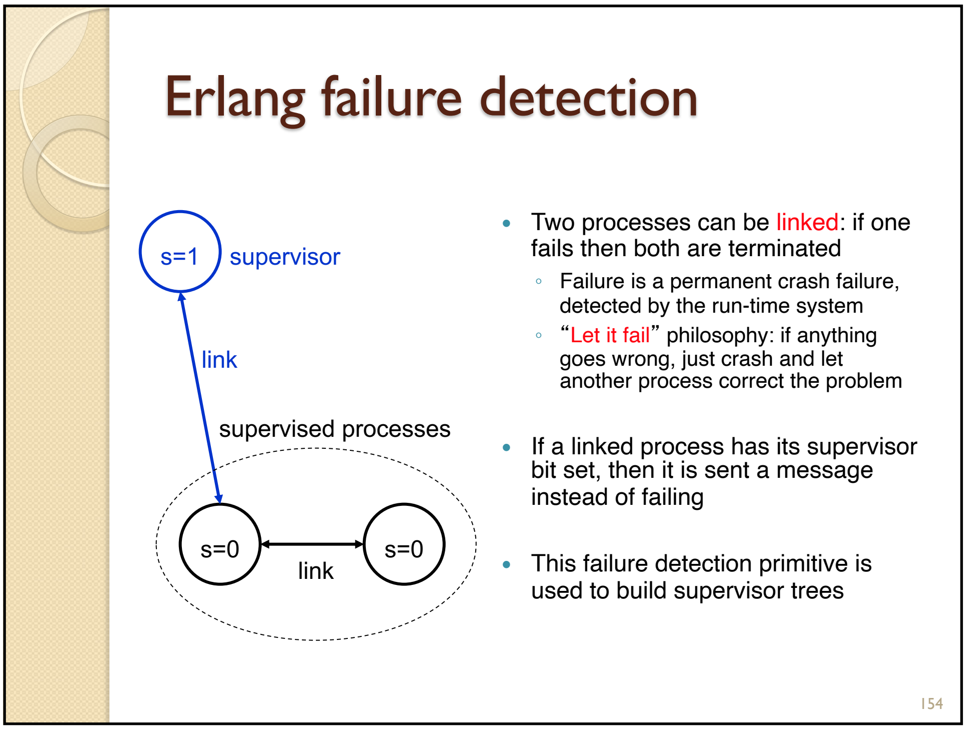
- Three levels of operation
  - QTA_1: normal operation of the supermarket with customers coming and going
  - QTA_2: local reconfiguration of the supermarket recovers QTA_1
  - QTA_3: global reconfiguration of supermarket chain recovers QTA_2; new resources are allocated here
- Each level “escalates” the solution
  - Each level happens at a different timescale
- Fire alarm is an emergency solution with low probability



Erlang

- Erlang is a language used to develop highly reliable software systems
- An Erlang program consists of a set of running processes (lightweight threads with independent address spaces) that send messages asynchronously
- Fault tolerance in three levels:
  - Primitive failure detection through process linking: when one process fails, another is notified
  - Supervisor trees to structure the program
  - Stable storage (database) to restore consistent state after crashes

153

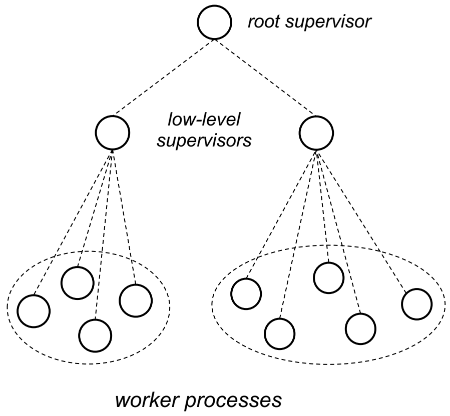


Erlang failure detection

- Two processes can be linked: if one fails then both are terminated
  - Failure is a permanent crash failure, detected by the run-time system
  - "Let it fail" philosophy: if anything goes wrong, just crash and let another process correct the problem
- If a linked process has its supervisor bit set, then it is sent a message instead of failing
- This failure detection primitive is used to build supervisor trees

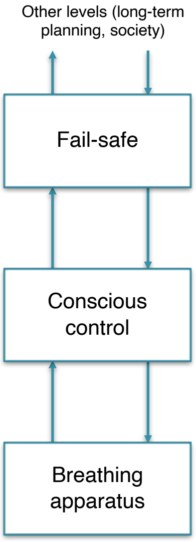
154

Erlang supervisor trees



- The program consists of a large number of processes
- Program processes are organized in pools
  - Each pool is observed by a supervisor process linked to all of them
  - An AND supervisor stops and restarts all its children if one crashes
  - An OR supervisor restarts just the crashed child
- Supervisors are themselves observed by a root supervisor
- The AXD 301 ATM switch we saw before uses this design

Human respiratory system



Other levels (long-term planning, society)

Fail-safe

- Fall unconscious when O_2 falls to threshold
- This reestablishes the breathing reflex

Conscious control

- Override breathing reflex to plan lifesaving actions
- Holding your breath will lead either to breath-hold threshold or O_2 threshold

Breathing apparatus

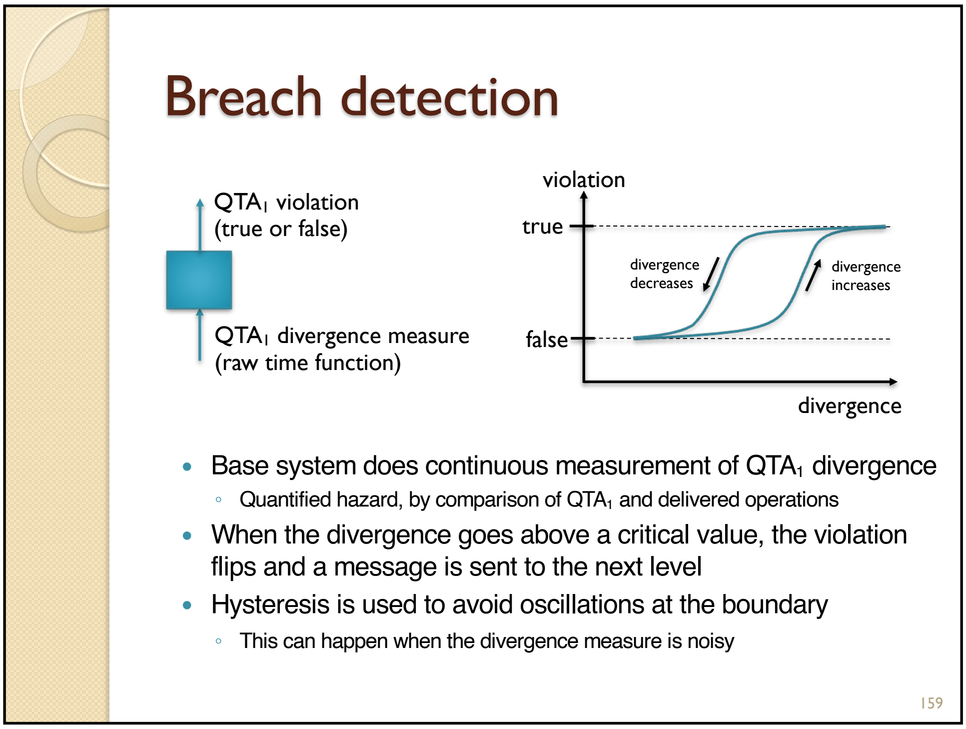
- Trigger breathing reflex when CO_2 reaches threshold
- Trigger laryngospasm when obstruction in airways (quick temporary action)

- The human respiratory system is a multilevel system
  - Designed and debugged by billions of years of evolution
  - This diagram summarizes a precise medical description (Wikipedia entry “Drowning”)
- Conscious control is a powerful problem solver but it must be used correctly
  - Breathing reflex can be controlled without conscious attention to details of muscle movement
  - Falling unconscious provides protection against instability
- Each level has its own timescale

Design rules

Mitigate or propagate?

- Operation of each level
  - Each level has its own QTA
  - QTA breach detection activates the next level
- When does operation move to the next level?
  - Each level is designed to mitigate by default. The three system rules make this possible. But if this becomes too complex (for example, because of interactions between the mitigation strategies), then propagate to the next level.
  - A second criterion is the timescale: propagate if solving the problem requires a different timescale (for example, it needs reconfiguration or elasticity, or a human in the loop)



Breach detection

QTA$_1$ violation
(true or false)

QTA$_1$ divergence measure
(raw time function)

violation

true

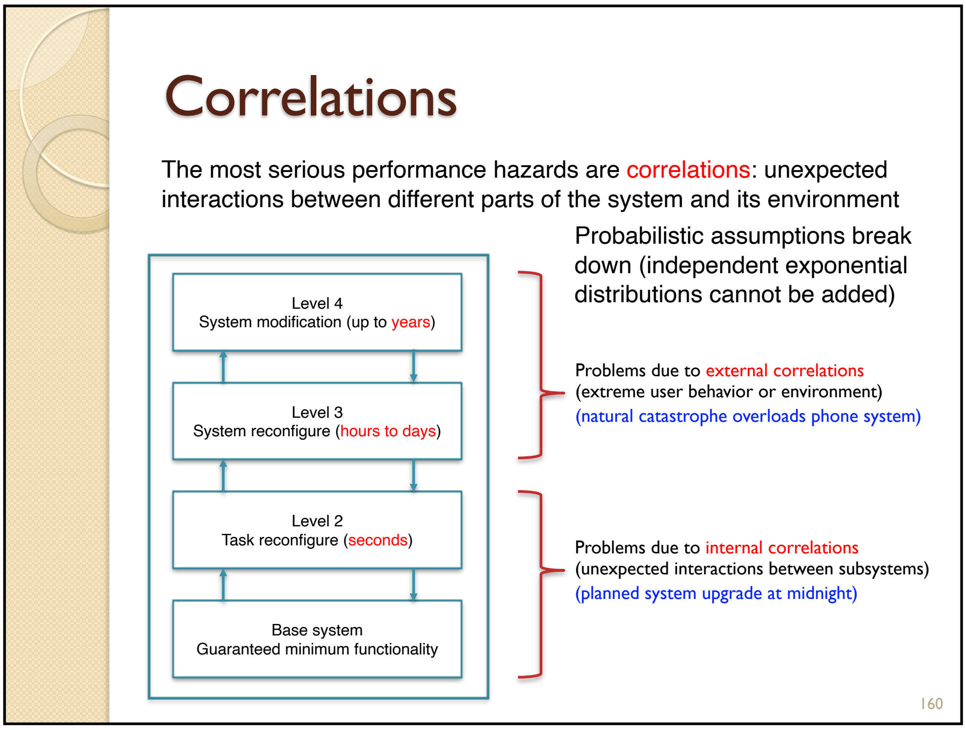
false

divergence decreases

divergence increases

divergence

- Base system does continuous measurement of QTA$_1$ divergence
  - Quantified hazard, by comparison of QTA$_1$ and delivered operations
- When the divergence goes above a critical value, the violation flips and a message is sent to the next level
- Hysteresis is used to avoid oscillations at the boundary
  - This can happen when the divergence measure is noisy



Correlations

The most serious performance hazards are correlations: unexpected interactions between different parts of the system and its environment

Level 4
System modification (up to years)

Level 3
System reconfigure (hours to days)

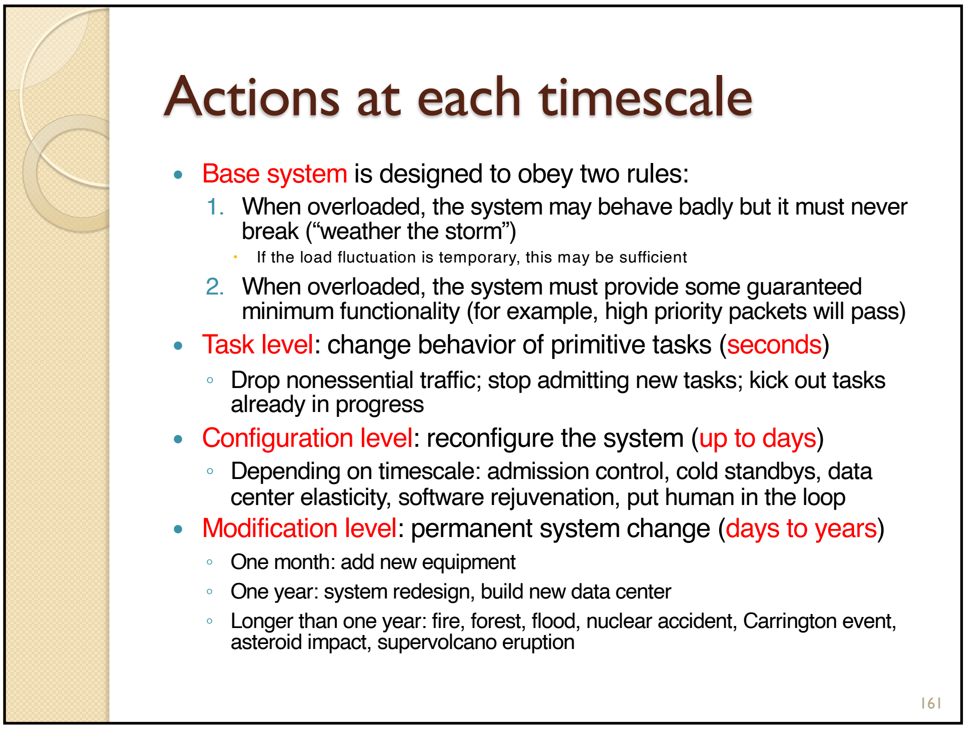
Level 2
Task reconfigure (seconds)

Base system
Guaranteed minimum functionality

Probabilistic assumptions break down (independent exponential distributions cannot be added)

Problems due to external correlations
(extreme user behavior or environment)
(natural catastrophe overloads phone system)

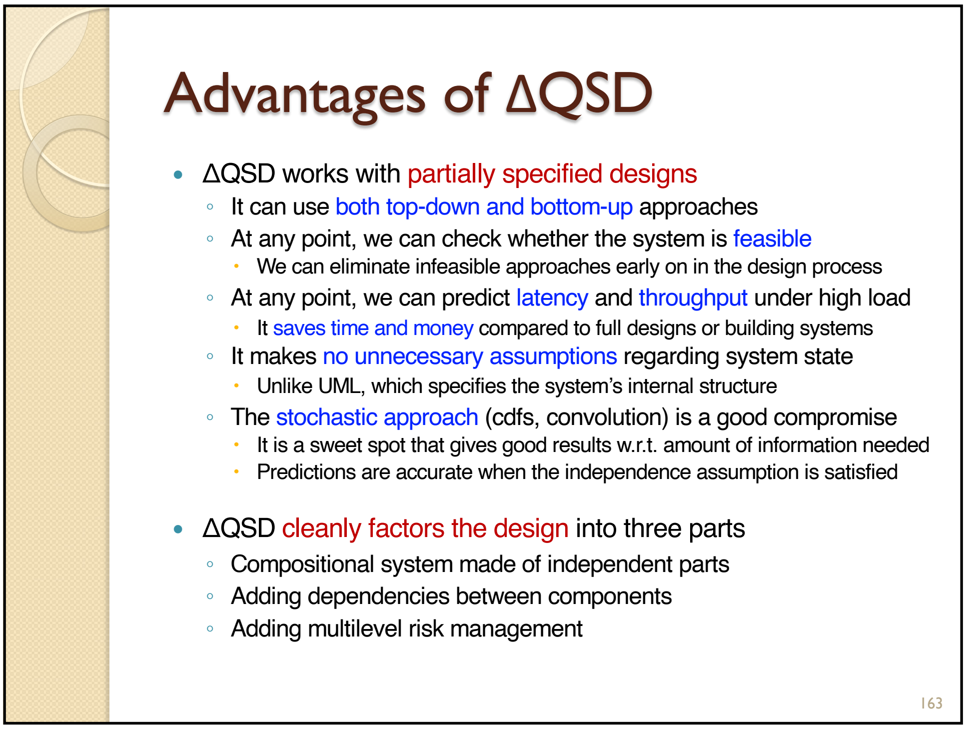
Problems due to internal correlations
(unexpected interactions between subsystems)
(planned system upgrade at midnight)



Actions at each timescale

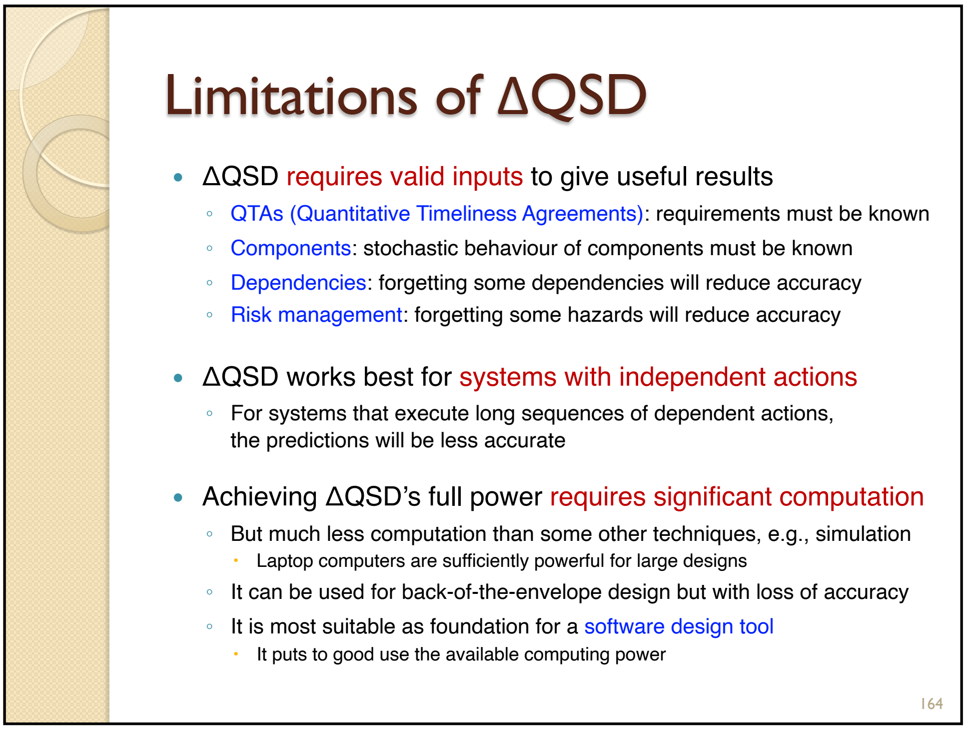
- Base system is designed to obey two rules:
  1. When overloaded, the system may behave badly but it must never break ("weather the storm")
     - If the load fluctuation is temporary, this may be sufficient
  2. When overloaded, the system must provide some guaranteed minimum functionality (for example, high priority packets will pass)
- Task level: change behavior of primitive tasks (seconds)
  - Drop nonessential traffic; stop admitting new tasks; kick out tasks already in progress
- Configuration level: reconfigure the system (up to days)
  - Depending on timescale: admission control, cold standbys, data center elasticity, software rejuvenation, put human in the loop
- Modification level: permanent system change (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

4.3 Conclusions



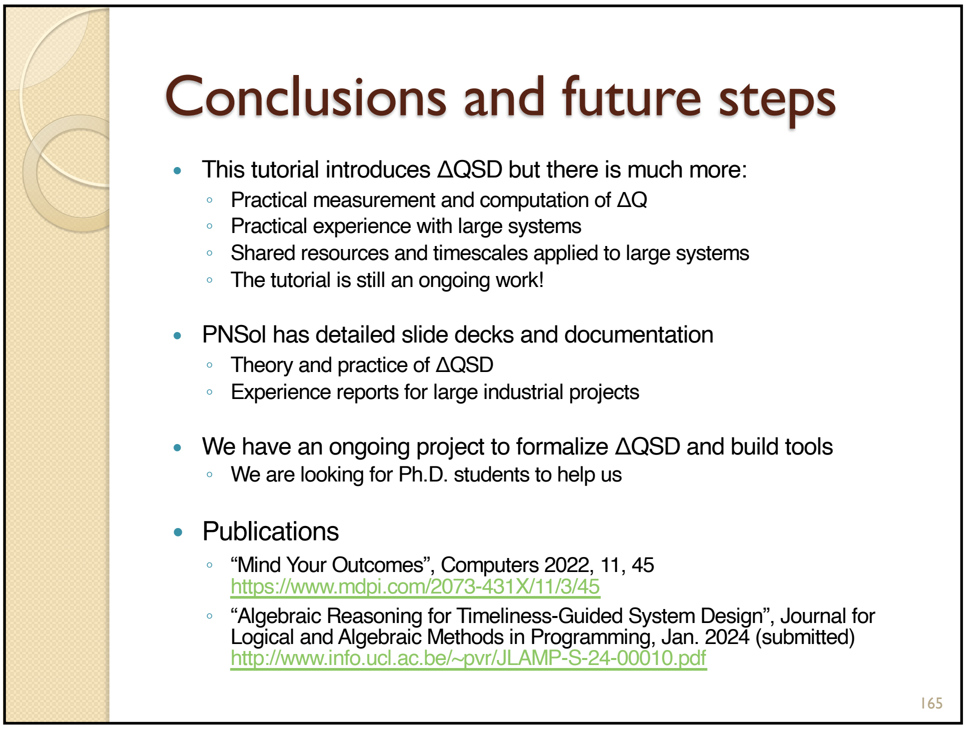
Advantages of ΔQSD

- ΔQSD works with partially specified designs
  - It can use both top-down and bottom-up approaches
  - At any point, we can check whether the system is feasible
    - We can eliminate infeasible approaches early on in the design process
  - At any point, we can predict latency and throughput under high load
    - It saves time and money compared to full designs or building systems
  - It makes no unnecessary assumptions regarding system state
    - Unlike UML, which specifies the system's internal structure
  - The stochastic approach (cdfs, convolution) is a good compromise
    - It is a sweet spot that gives good results w.r.t. amount of information needed
    - Predictions are accurate when the independence assumption is satisfied
- ΔQSD cleanly factors the design into three parts
  - Compositional system made of independent parts
  - Adding dependencies between components
  - Adding multilevel risk management



Limitations of ΔQSD

- ΔQSD requires valid inputs to give useful results
  - QTAs (Quantitative Timeliness Agreements): requirements must be known
  - Components: stochastic behaviour of components must be known
  - Dependencies: forgetting some dependencies will reduce accuracy
  - Risk management: forgetting some hazards will reduce accuracy
- ΔQSD works best for systems with independent actions
  - For systems that execute long sequences of dependent actions, the predictions will be less accurate
- Achieving ΔQSD's full power requires significant computation
  - But much less computation than some other techniques, e.g., simulation
    - Laptop computers are sufficiently powerful for large designs
  - It can be used for back-of-the-envelope design but with loss of accuracy
  - It is most suitable as foundation for a software design tool
    - It puts to good use the available computing power



Conclusions and future steps

- This tutorial introduces ΔQSD but there is much more:
  - Practical measurement and computation of ΔQ
  - Practical experience with large systems
  - Shared resources and timescales applied to large systems
  - The tutorial is still an ongoing work!

- 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

- Publications
  - “Mind Your Outcomes”, Computers 2022, 11, 45
    https://www.mdpi.com/2073-431X/11/3/45
  - “Algebraic Reasoning for Timeliness-Guided System Design”, Journal for Logical and Algebraic Methods in Programming, Jan. 2024 (submitted)
    http://www.info.ucl.ac.be/~pvr/JLAMP-S-24-00010.pdf