

# ΔQSD: Designing Systems with Predictable Latency at High Load

January 18, 2023

Peter Van Roy  
Université catholique de Louvain

Neil Davies, Peter Thompson  
Predictable Network Solutions Ltd.

Seyed Hossein Haeri  
PLWorkz

1

1

## Organization of the tutorial

- Lecture 1: Case Studies
  1. Small cells
  2. iPhone launch
  3. Cardano Shelley block diffusion
- Lecture 2: Compositional systems
  1. Quality attenuation ( $\Delta Q$ )
  2. Outcome diagrams
  3. Shelley block diffusion algorithm
  4. Some typical  $\Delta Q$ s
- Lecture 3: Systems with dependencies
  1. Shared resources
  2. Iterative query
- Lecture 4: Multilevel systems
  1. Managing risk
  2. Multilevel system design
  3. Supermarket example
  4. Design for overload
- Conclusions
- ΔQSD is a system design paradigm that can predict system behaviour at high load. It was developed by PNSol over 30 years and validated in large-scale industrial systems
  - This tutorial is part of an ongoing project to disseminate ΔQSD for the benefit of the wider system design community
- This tutorial is work in progress: I welcome errata and constructive comments
- Caveat
  - I am not the inventor of ΔQSD. I am a computer scientist with long experience in system design based on distributed systems and programming languages. I created this tutorial as part of my experience in learning ΔQSD, because I consider ΔQSD to be an interesting and innovative approach that deserves to be more widely known.

2

2

## Systems with many users

- $\Delta$ QSD 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  $\Delta$ QSD 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

3

3

## PNSol Ltd

[www.pnsol.com](http://www.pnsol.com)



- 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  $\Delta$ QSD methodology for design and diagnosis of large systems with predictable performance under high-load conditions

4

4

## $\Delta$ QSD paradigm

- $\Delta$ QSD 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
- $\Delta$ QSD 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

5

5

## Goals of these lectures

- Understand the two main concepts of  $\Delta$ QSD: **quality attenuation ( $\Delta$ Q)** and **outcome diagram**
- Understand how to design systems as **independent parts** with **added dependencies**
- Understand how to design systems by refining **partially defined systems**
- Understand how to compute **latency and throughput** and **infeasibility** during the design
- Give enough concepts and examples so you can start using  $\Delta$ QSD in your own designs

6

6

## Introduction

7

7

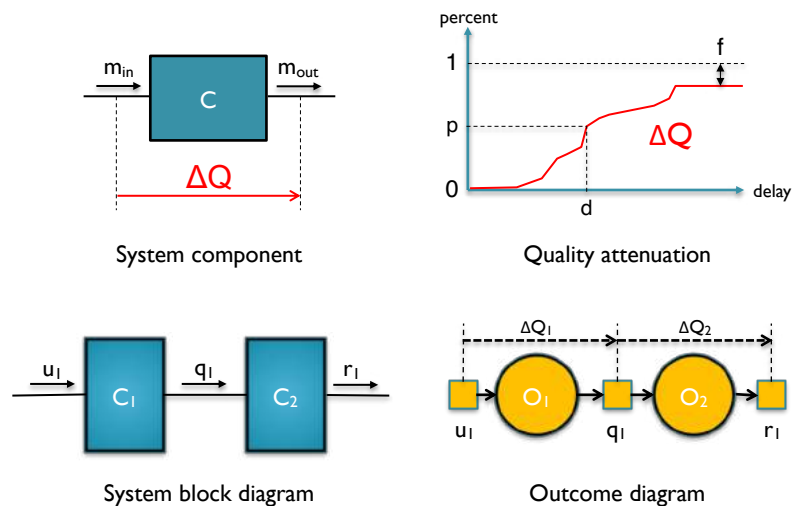
## Two main concepts of $\Delta QSD$

- Quality attenuation ( $\Delta Q$ ): “first-class latency and failure”
  - A  $\Delta Q$  is a **cumulative distribution function** that defines both latency and failure probability between a start and an end event
  - Because the  $\Delta Q$  **combines latency and failure** in a single quantity, it makes it easy to examine trade-offs between them
- Outcome diagram: “system observed from outside”
  - An **outcome** is any **well-defined system behaviour** with observable start and end events; each outcome has a  $\Delta Q$
  - An **outcome diagram** is a **causal directed graph** that defines the relationships between all system outcomes; it allows **computing  $\Delta Q$**  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.

8

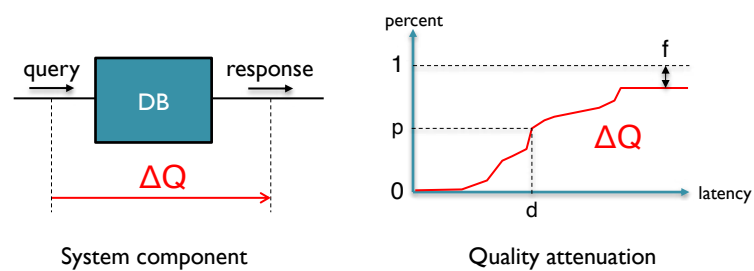
8

## Quality attenuation and outcome diagram



9

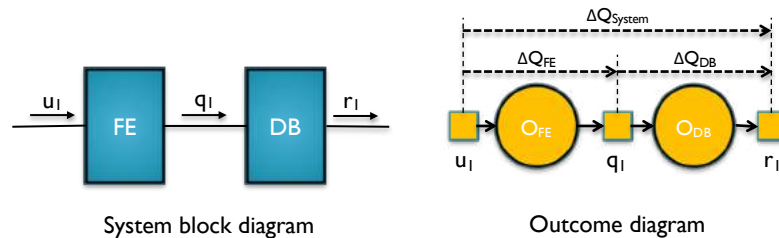
## Quality attenuation $\Delta Q$



- Given a system component, for example a database
  - What is the latency 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  $\Delta Q$**  (actually, an **improper random variable** because  $\max < 1$ )
  - This represents both the variability and the failure probability

10

## Outcome diagram



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

11

11

## To the case studies...

- Now we know enough for the case studies
- We combine  $\Delta Q_i$  of components  $C_i$  to get the  $\Delta Q_{system}$  of the whole system
  - If there is something wrong with  $\Delta Q_{system}$  then we reason backwards to pinpoint the problem
- After the case studies, we will study how to design systems using  $\Delta Q$  and outcome diagrams

12

12

# Lecture I

## Case Studies

13

13

## Case studies

- As motivation for  $\Delta$ QSD 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  $\Delta$ QSD paradigm is used in two ways
  - **Diagnostic use:** debugging of existing systems with problems (for small cells and iPhone launch)
  - **Design use:** designing systems using  $\Delta$ QSD from the start (for Cardano Shelley)
- It's better to use  $\Delta$ QSD for design rather than diagnosis
  - Prevention is better than cure!
  - This is one of the motivations of this tutorial: to disseminate the  $\Delta$ QSD paradigm so it can be used during the design process
    - PNSol is often called in to perform a cure for systems that have major problems

14

14

## I. Small Cells Case Study

15

15

## Small cells case study

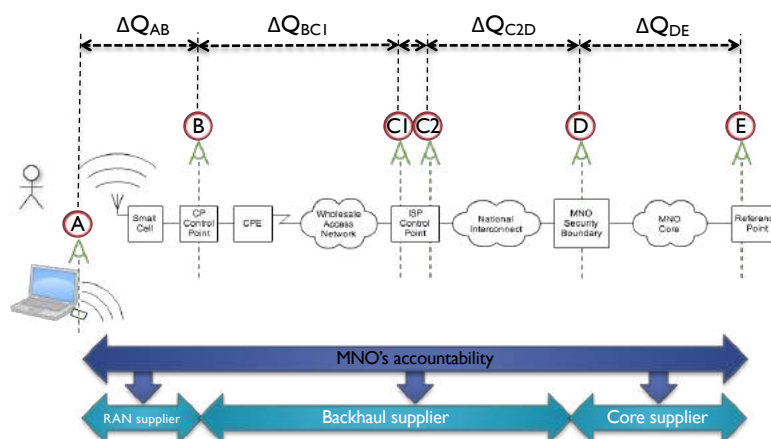
- A major MNO (Mobile Network Operator), who shall remain unnamed, deployed small cells
  - Small cell: low-powered cellular radio access nodes with range 10m-3km
  - Backhaul using consumer DSL broadband
- The system worked but did not scale
  - Voice quality had major problems, cells were failing
  - 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  **$\Delta Q$**  across system to pinpoint the problem
  - Focus on problematic behavior shown by  $\Delta Q$
  - $\Delta QSD$  led to successful diagnosis and cure proposal

16

16



## 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.**

17

17

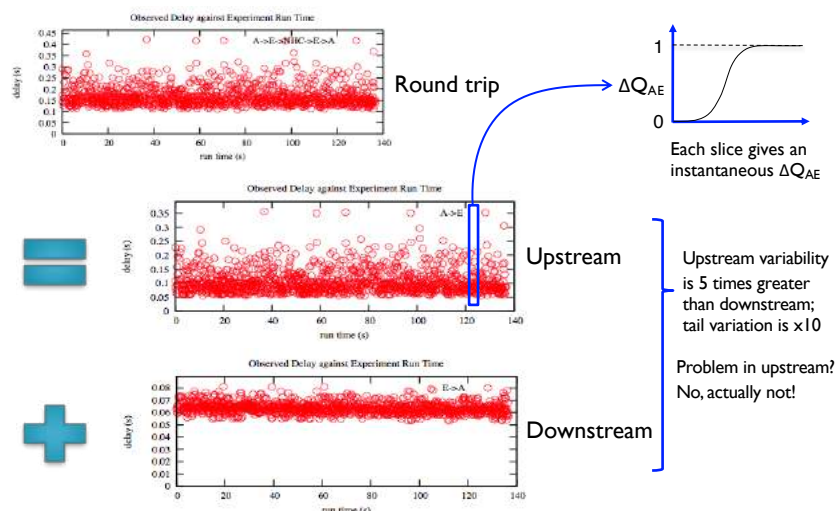
## How PNSol gathered the evidence

- **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**  
 $A \rightarrow B \rightarrow C1 \rightarrow C2 \rightarrow D \rightarrow E \rightarrow D \rightarrow C2 \rightarrow C1 \rightarrow B \rightarrow A$
  - 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.

18

18

## Which direction has issues?

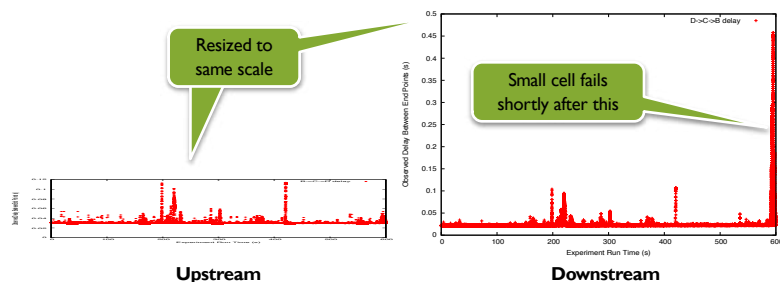


19

19

## Who is to blame for the system failing?

### Examine sub-paths to isolate the issue



- The instantaneous  $\Delta Q$  is measured as a function of experiment run time
- We find that the  $\Delta Q$  is **not stationary**; it changes during the run
- There are times when the  $\Delta Q$  has **strong anomalous behavior**

20

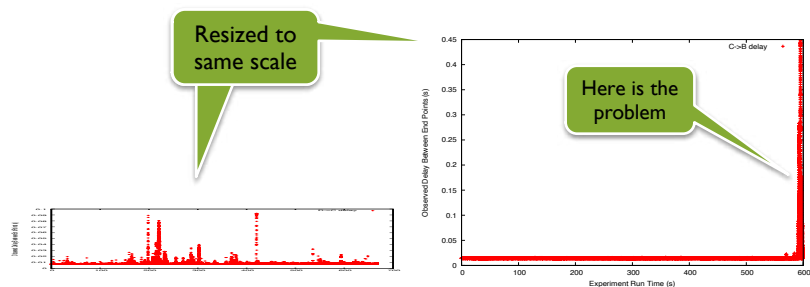
20

## Where is the issue?

Use spatial resolution to isolate the problem

National Interconnect

Wholesale Access Core

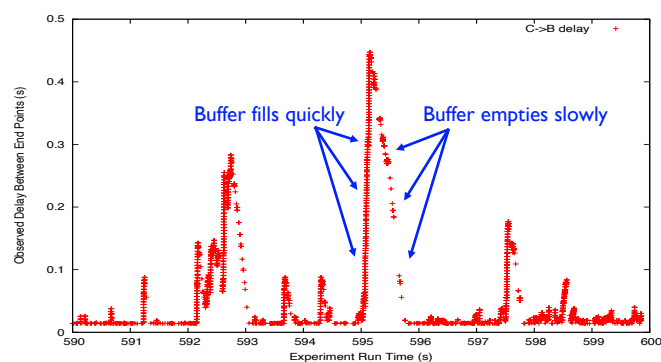


21

21

## 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!

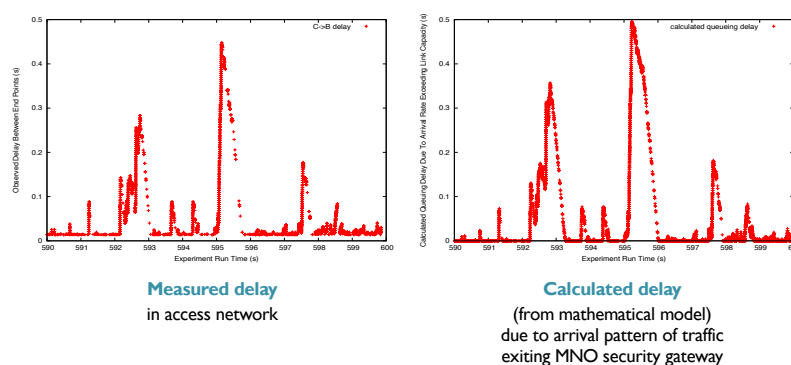
⇒ Later in the lecture we will study **queues** to understand this

22

22

## Actual + predicted measures

Use predictability of  $\Delta Q$  to check the conclusion



23

## 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**

24

## 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  $\Delta Q$  (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**

25

## 2. iPhone Launch Case Study

26

26

## iPhone launch case study

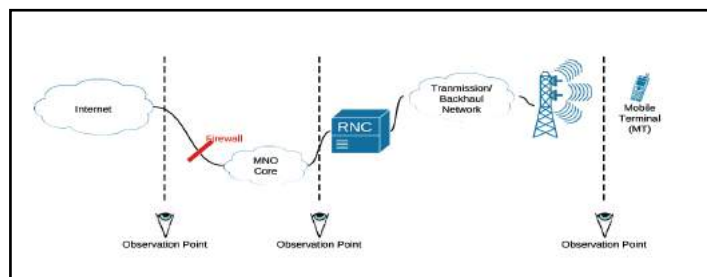
- iPhone was initially supported in UK by one MNO
- A second MNO prepared to enter this market
  - Before the launch, the performance was known to be bad for the second MNO, and the first MNO had gleefully prepared a major ad campaign focusing on this fact
    - Both MNOs are large UK operators who will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

27

27

## Diagnosis approach and solution

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



- The  $\Delta$ QSD paradigm was used for the diagnosis
  - Determine outcome diagram for end to end delivery of packets and measuring  $\Delta$ Q 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

28

## 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 ...

29

## Packet delivery behaviour

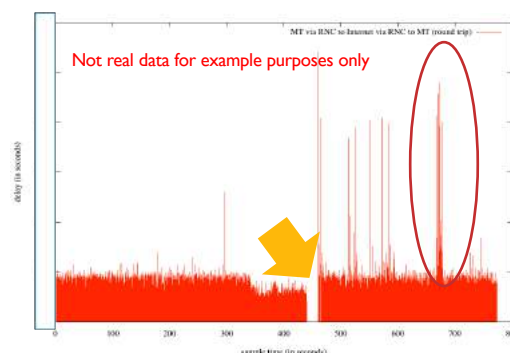
The RTT for the full duration of the data collection

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...

30



## Packet delivery behaviour

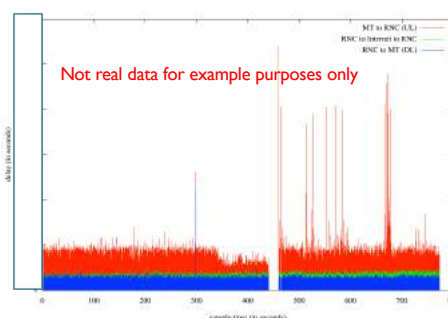
Combined observations split by element

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

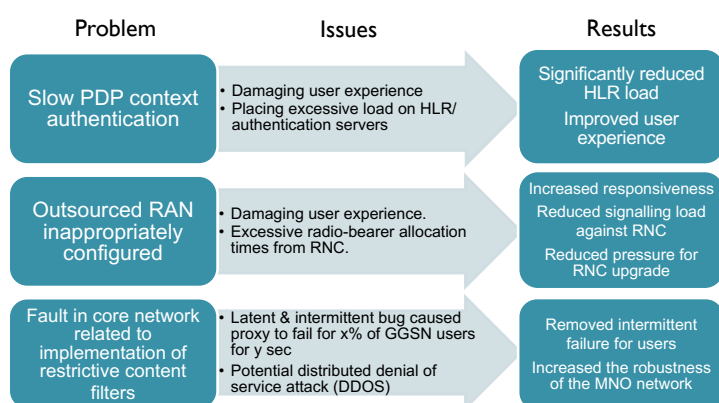
The MNO suspected the RNC DL was the major trouble spot. As can be seen with the **RED** (MT to RNC UL) we found it was really on the UL: this is where the service break occurred.



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

31

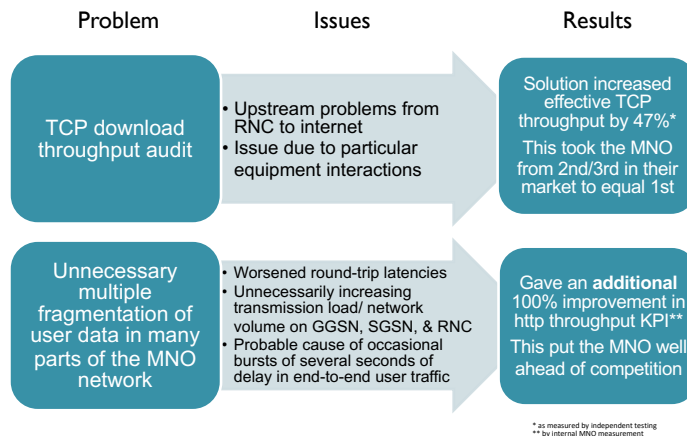
## iPhone launch findings using $\Delta$ QSD (1 of 2)



32



## iPhone launch findings using $\Delta$ QSD (2 of 2)



33

## 3. Cardano Shelley Case Study

34

34

## Cardano Shelley case study

- The previous case studies used  $\Delta$ QSD for **diagnosis**
  - PNSol was brought in to diagnose problems in running systems
- Cardano Shelley used  $\Delta$ QSD for the system **design**
  - Design is the preferred way to use  $\Delta$ QSD (“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  $\Delta$ QSD to guide the design from early on, and achieved adequate performance in a decentralised environment
    - We present part of the Shelley design using  $\Delta$ QSD

35

35

## Context of block diffusion

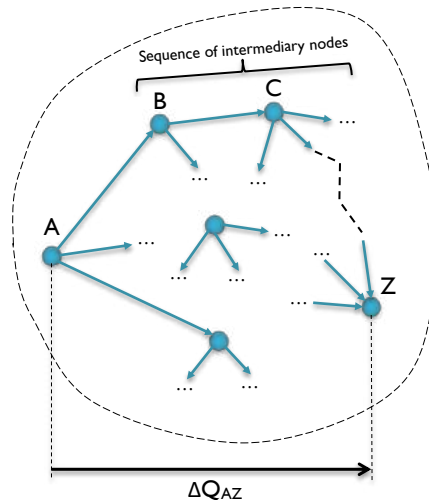
- Blockchain management in Cardano
  - We will use  $\Delta$ QSD to solve a design problem in the Cardano cryptocurrency, which is implemented using a blockchain
  - A blockchain is a distributed ledger comprising a set of data blocks that are cryptographic witnesses to correctness of preceding blocks
  - 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  $\Delta$ QSD to ensure that the algorithm satisfies stringent timeliness constraints


36

36

## Block diffusion problem statement

Node graph of Cardano blockchain



- Problem:
  - Determine  $\Delta Q_{AZ}$  for randomly chosen nodes A and Z, as function of design
  - Determine design so that  $\Delta Q_{AZ}$  satisfies performance constraints
  - $\Delta Q_{XY}$  is known (measured) 
- Design parameters:
  - Frequency of block production
  - Node connection graph
  - Block size
  - Block forwarding protocol
  - Block processing time

37

37

## Block diffusion design using $\Delta QSD$

- First step:  $\Delta Q$  measurement
  - Measure primitive  $\Delta Q$  for simple cases
  - Compute overall  $\Delta Q$  for two arbitrary nodes across the Internet
- Second step: Algorithm design
  - Define simple initial design and its outcome diagram
  - Performance is  $\Delta Q$  as function of load
  - Make a design decision and refine the outcome diagram
    - Each refinement defines a new outcome diagram
    - Compute new performance and compare with requirements
    - Decide whether to keep the design or not
  - Continue until design is satisfactory
- We will show part of this process in Lecture 2
  - When we have defined the concepts we need

38

38

## Lecture 2 Compositional Systems

39

39

### Systems with no dependencies (compositional systems)

- $\Delta$ QSD 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 handle multiple timescales
- We start with systems of independent parts
  - Most systems consist largely of independent parts
  - Dependencies and multiple levels will be treated later (in Lectures 3 and 4)
- Topics for Lecture 2
  - Quality attenuation ( $\Delta Q$ )
  - Outcome diagrams
  - Cardano Shelley block diffusion
  - Some typical  $\Delta Q$ s

40

40

## Lecture 2 contents

1. Quality attenuation ( $\Delta Q$ )
  1. Designing with  $\Delta Q$
  2. Diagnosing with  $\Delta Q$
2. Outcome diagrams
  1. Client/server example
  2. Cache memory example
  3. General system design
  4. Semantics of outcome diagrams
3. Cardano Shelley block diffusion algorithm
  1. Measuring  $\Delta Q$
  2. Designing with outcome diagrams
4. Some typical  $\Delta Q$ s
  1. Some typical distributions
  2.  $\Delta Q$  for a typical component
  3. Load balancing example
  4.  $\Delta Q$  for a typical network

41

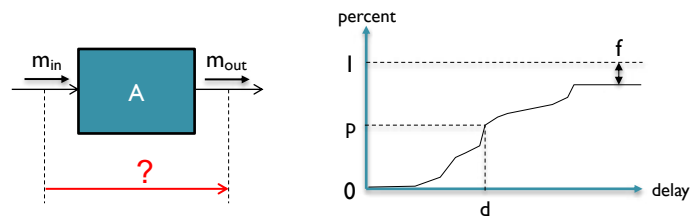
41

## I. Quality Attenuation ( $\Delta Q$ )

42

42

## Quality attenuation ( $\Delta Q$ )



- 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  $\Delta Q$** 
  - $p$  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

43

43

## Combining delay and failure

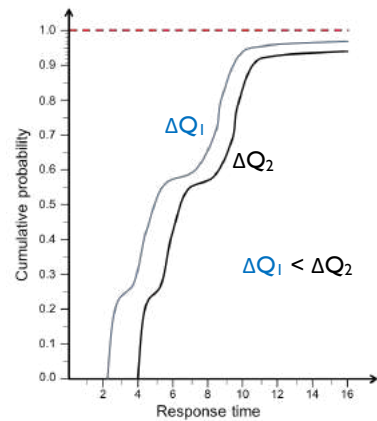
- Delay and failure are combined in one quantity  $\Delta Q$ 
  - Two parts of system design that are usually separate are considered together
  - This lets us examine trade-offs between delay and failure
- Performance and fault tolerance should not be separate
  - They are two sides of the same coin
  - For example, failure can be reduced by increasing delay, which is all part of one  $\Delta Q$ 
    - **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  $\Delta Q$  quantity

44

44

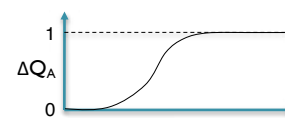
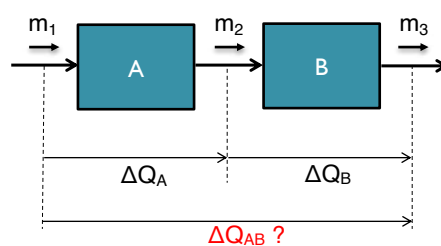
## Comparing $\Delta Q$ s

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

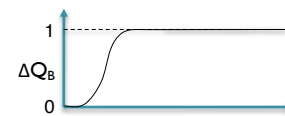


45

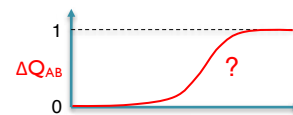
## “Adding” two $\Delta Q$ s



⊕



=

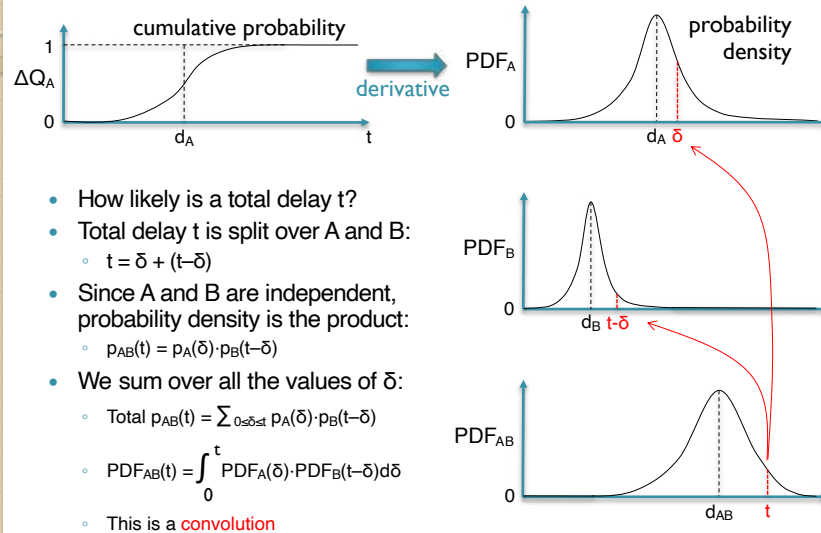


- Given components A and B
  - $\Delta Q_A$  from  $m_1$  to  $m_2$
  - $\Delta Q_B$  from  $m_2$  to  $m_3$
- We connect them together
  - What is  $\Delta Q_{AB}$  from  $m_1$  to  $m_3$ ?

46

46

## Convolution: “sum” of two $\Delta Q$ s



- 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  $\delta$ :
  - Total  $p_{AB}(t) = \sum_{0 \leq \delta \leq t} p_A(\delta) \cdot p_B(t - \delta)$
  - $PDF_{AB}(t) = \int_0^t PDF_A(\delta) \cdot PDF_B(t - \delta) d\delta$
  - This is a **convolution**

47

47

## Designing with $\Delta Q$

48

48



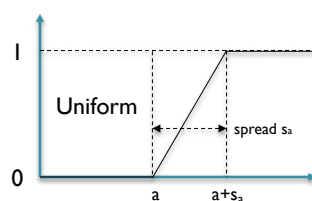
## Designing with $\Delta Q$

- We can use  $\Delta Q$  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  $\Delta Q$  in these examples, namely a Uniform distribution
  - This is a reasonable approximation for components, but of course many other  $\Delta Q$ s occur in practice!
  - We will “add” and “subtract”  $\Delta Q$ s in the examples, note that technically this is convolution and deconvolution

49

49

## Uniform distribution



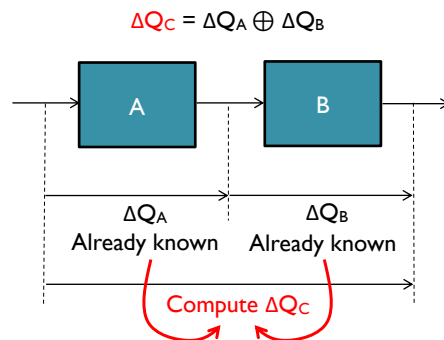
- A Uniform distribution approximates a component with buffer and server
  - $a$  is the minimum time in the component
  - $s_a$  is the spread of times in the component
  - $a+s_a$  is the maximum time in the component

- For our two examples, we use a Uniform distribution for  $\Delta Q$ 
  - 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”  $\Delta Q$  computations; an automated tool can of course compute with a full  $\Delta Q$
- In this lecture, we will do back-of-the-envelope computations
  - It is easy to extend this and do the full computations

50

50

## Bottom-up design with $\Delta Q$



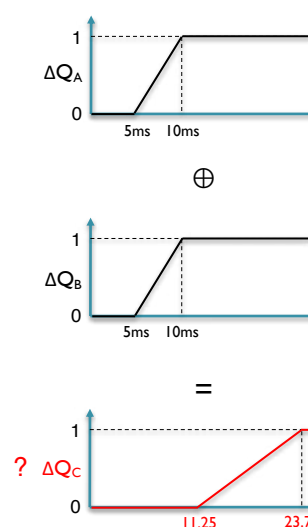
- Component A has  $\Delta Q_A$  and component B has  $\Delta Q_B$ 
  - What is  $\Delta Q_C$ ?
- We assume Uniform distributions for A and B and “add” them to get C:
  - Assume  $(a, s_a)$  and  $(b, s_b)$
  - We can approximate  $(c, s_c)$ :  
 $c = (a + b) + m/4$   
 $s_c = \max(s_a, s_b) + m/2$   
 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

51

51

## Numerical bottom-up example

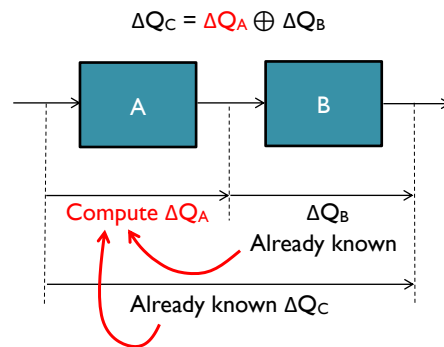
- We know A and B
  - $a=5, s_a=10$
  - $b=5, s_b=10$
  - $m = \min(s_a, s_b) = 5$
- Compute for C:
  - $c = (a+b) + m/4 = 11.25\text{ms}$
  - $s_c = \max(s_a, s_b) + m/2 = 12.5\text{ms}$
- Note bigger  $c$  and  $s_c$ !
  - $c = 11.25$  not 10
  - $s_c = 12.5$  not 10



52

52

## Top-down design with $\Delta Q$



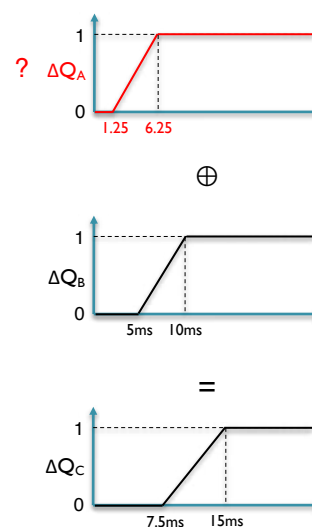
- There is a global overall requirement of  $\Delta Q_C$  and component B is known to have  $\Delta Q_B$ 
  - What  $\Delta Q_A$  is needed for A?
- We assume Uniform distributions and “subtract”:
  - $a \leq (c - b) - m/4$ 
    - Remember that  $m = \min(s_a, s_b)$
    - A's delay must be less than  $c - b$
  - If  $s_a \leq s_b$  then  $s_a \leq 2(s_c - s_b)$   
 If  $s_a > s_b$  then  $s_a \leq s_c - s_b/2$ 
    - This follows from  $\max(s_a, s_b) = s_c - m/2$

53

53

## 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:
  - $a \leq (c - b) - m/4 = 1.25\text{ms}$
  - $s_a \leq 2(s_c - s_b) = 5\text{ms}$
- Note strict bound on A!
  - $a=1.25$  not 2.5



54

54

## 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!

- |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>We start with two simultaneous equations in <math>(a, s_a)</math>:<br/> <math>c = a + b + \min(s_a, s_b)/4</math><br/> <math>s_c = \max(s_a, s_b) + \min(s_a, s_b)/2</math></li> </ul>                                                                                                                                                                                                                                                                                                                                                               |
| <ul style="list-style-type: none"> <li>We solve this by distinguishing two cases</li> <li>First, assume <math>s_a \leq s_b</math> :<br/> <math>s_a = 2(s_c - s_b) &gt; 0</math> which implies <math>s_c &gt; s_b/2</math> [1]<br/> <math>a = (c-b) - (s_c - s_b)/2 &gt; 0</math> which implies <math>(c-b) &gt; s_c/2 - s_b/2</math> [2]</li> <li>Second, assume <math>s_a &gt; s_b</math> :<br/> <math>s_a = s_c - s_b/2 &gt; 0</math> which implies <math>s_c &gt; s_b/2</math> [3]<br/> <math>a = c - b - s_b/4 &gt; 0</math> which implies <math>(c-b) &gt; s_b/4</math> [4]</li> </ul> |
| <ul style="list-style-type: none"> <li>The design is infeasible if <math>(\neg[1] \vee \neg[2]) \wedge (\neg[3] \vee \neg[4])</math><br/>           which is implied by <math>s_c \leq s_b/2 \vee (c-b) \leq \min(s_c/2 - s_b/2, s_b/4)</math></li> </ul>                                                                                                                                                                                                                                                                                                                                   |

55

55

## “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

56

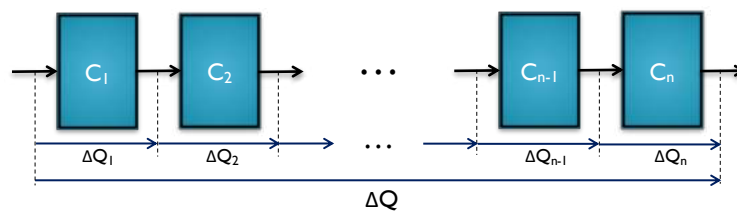
56

## Diagnosing with $\Delta Q$

57

57

## Diagnosing with $\Delta Q$



- Consider a pipeline of components that has a bad overall  $\Delta Q$ 
  - This happens often in practice, e.g., [the small cells case study](#)
- Since adding a component can only make  $\Delta Q$  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
  - This technique was used in the small cells case study

58

58

## 2. Outcome Diagrams

59

59

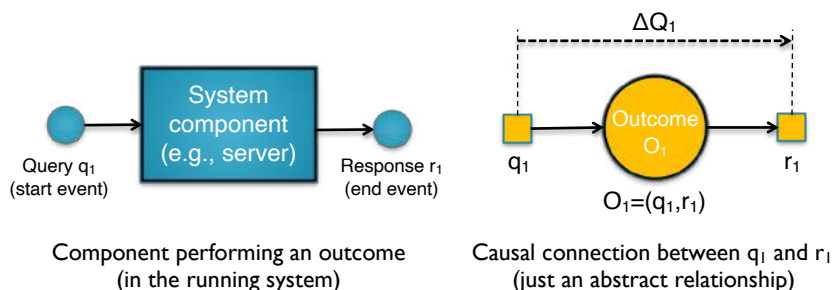
### Outcome diagrams

- Now let's combine components (defined by  $\Delta Q$ ) 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

60

60

## Single outcome

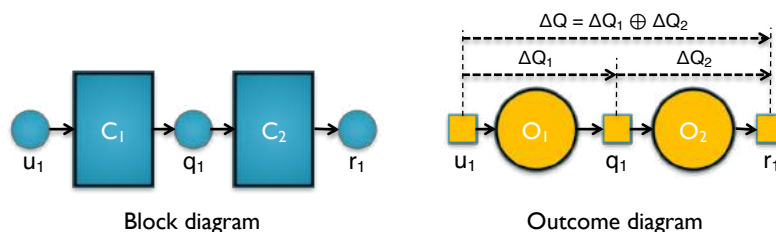


- An **outcome**  $O_1$  is a specific system behaviour, which is a pair defined by its start event  $q_1$  and end event  $r_1$ 
  - We don't care how the system is built, we simply observe it
  - Left figure shows the query and response messages entering and exiting a component
  - Right figure shows just the causal connection between the two events: query causes response, with quality attenuation  $\Delta Q_1$

61

61

## Outcome diagram



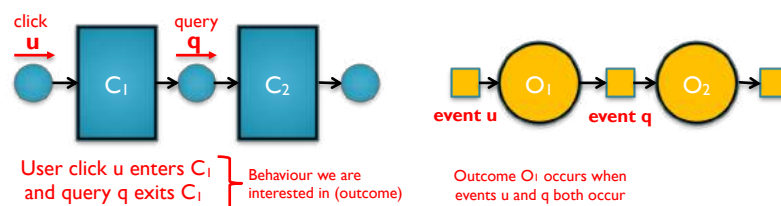
- We have a user click  $u_1$  causing a query  $q_1$  to be sent causing a response  $r_1$  to be received
- An **outcome diagram** is a graph showing the causal connections between all the outcomes that we are interested in
  - We don't actually care (yet) how the system is constructed, we are only interested in the behaviour
  - Total  $\Delta Q$  is the convolution of the individual  $\Delta Q_1$  and  $\Delta Q_2$  (all delays and failures are "added")

62

62

## How outcome diagrams work

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



- 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

63

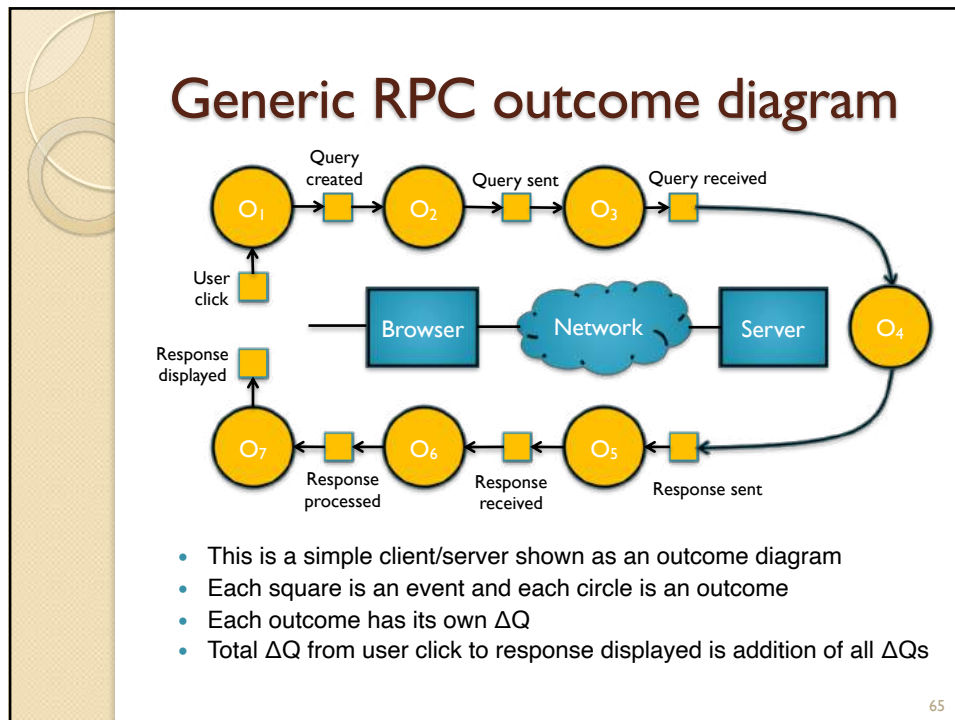
63

Client/server example

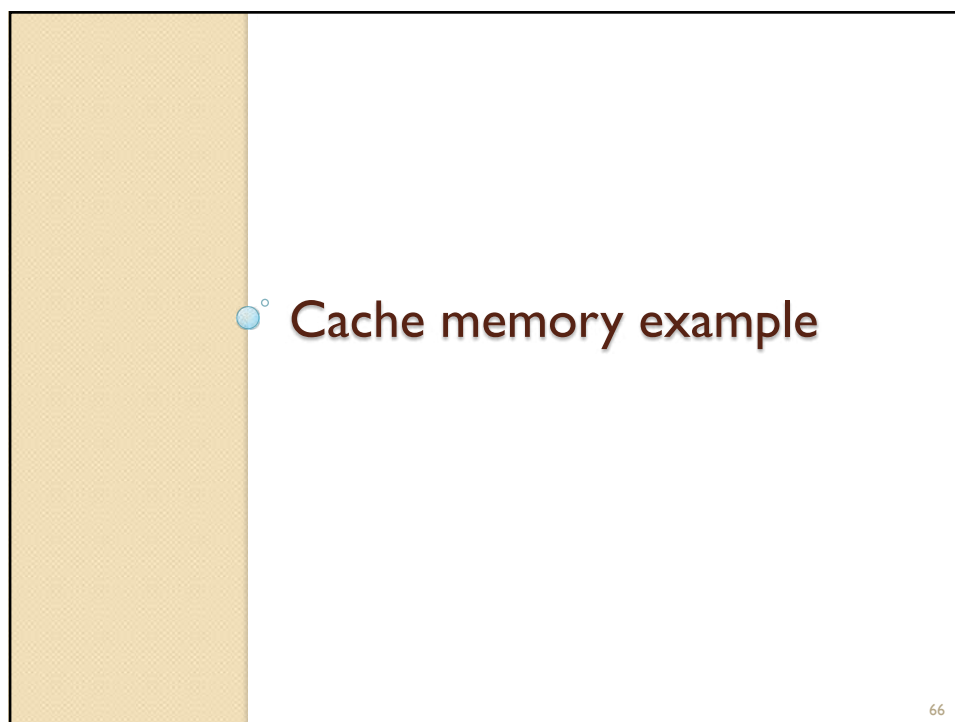
64

64



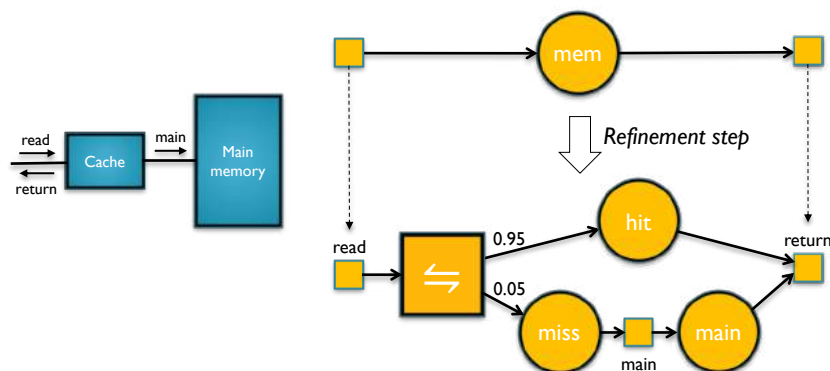


65



66

## Cache memory example

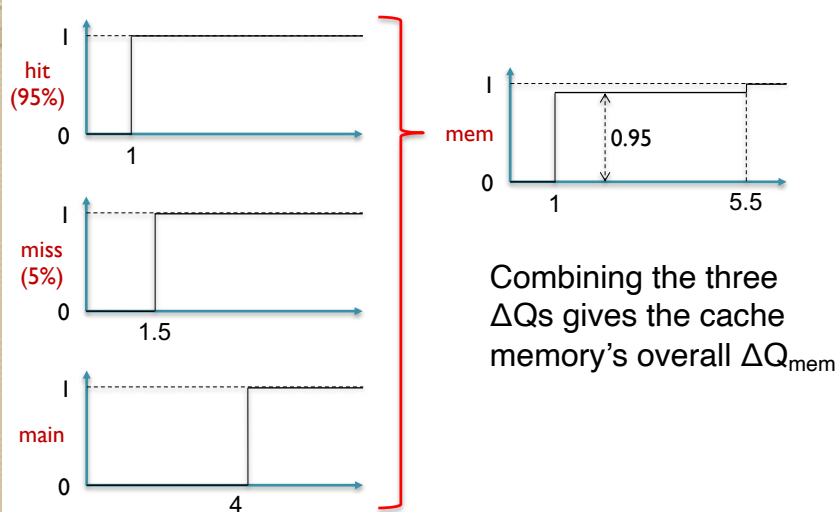


- A cache memory is modeled using **probabilistic choice**
- $\Delta Q_{\text{mem}} = h \cdot \Delta Q_{\text{hit}} + m \cdot (\Delta Q_{\text{miss}} \oplus \Delta Q_{\text{main}})$
- We can see the cache as one component or refine it

67

67

## Cache quality attenuation



68

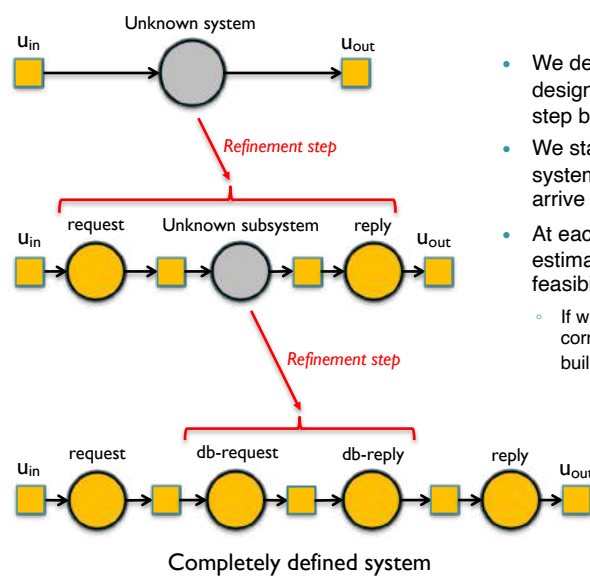
68

## General system design

69

69

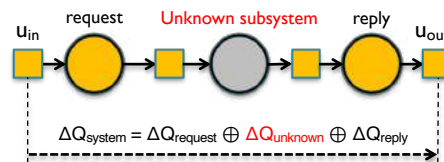
## General system design



70

70

## Example top-down design

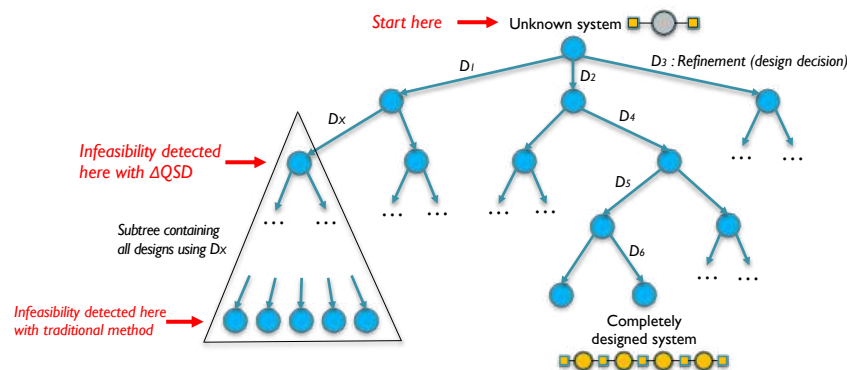


- We use a top-down design approach
  - We assume that  $\Delta Q_{\text{system}}$ ,  $\Delta Q_{\text{request}}$ ,  $\Delta Q_{\text{reply}}$  are all known:  $\Delta Q_{\text{system}}$  is the system requirement, and  $\Delta Q_{\text{request}}$  and  $\Delta Q_{\text{reply}}$  have already been determined
  - We compute required  $\Delta Q_{\text{unknown}}$  for the unknown subsystem to be designed
- If  $\Delta Q_{\text{unknown}}$  is infeasible, then go back and change  $\Delta Q_{\text{request}}$  and  $\Delta Q_{\text{reply}}$ 
  - If there is no way to solve the problem by changing  $\Delta Q_{\text{request}}$  and  $\Delta Q_{\text{reply}}$  then we need to go back even further and change the overall requirement  $\Delta Q_{\text{system}}$  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...

71

71

## 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  $\Delta QSD$  paradigm allows to compute infeasibility early on, even for partially defined systems

72

72

## 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 logical 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

73

73

## ◦ Semantics of outcome diagrams

74

74

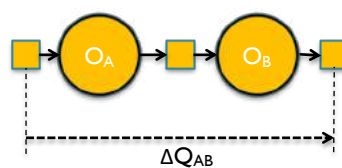
## Semantics of outcome diagrams

- Given an outcome diagram and the  $\Delta Q$ s of all outcomes in the diagram, we can compute the  $\Delta Q$  of the complete diagram
  - Recall that  $\Delta Q(t)$  is a function of delay  $t$  that represents the **cumulative probability distribution** of the delay (formally, it is an improper random variable since the maximum can be  $< 100\%$ )
- Outcome diagrams have four primitive operators
  - Sequential composition (**convolution**)
  - Probabilistic choice (**weighted sum**)
  - Last-to-finish (all-to-finish) (**arithmetic product**)
  - First-to-finish (**dual of arithmetic product**)
- They are defined as a formal language
  - Outcome diagrams are represented formally by outcome expressions with a semantics, which allows a software tool to represent outcome diagrams and do  $\Delta Q$  computations on them
  - We only give the semantics of the four operators in this lecture; to make a practical software tool we need to define more properties

75

75

## Sequential composition

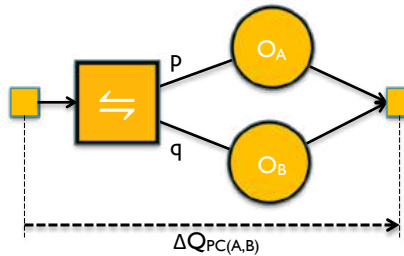


- Assume two outcomes  $O_A$  and  $O_B$  where the end event of  $O_A$  is the start event of  $O_B$
- The probability distribution of  $O_{AB}$  is the convolution of the probability distributions of  $O_A$  and  $O_B$
- Therefore:
 
$$\Delta Q'_{AB} = \Delta Q'_A \oplus \Delta Q'_B$$
 where  $\Delta Q'(t) = d\Delta Q/dt$  and  $\oplus$  is the convolution operator
- Convolution is a commutative mathematical operator, but this does not mean that components can be switched around

76

76

## Probabilistic choice



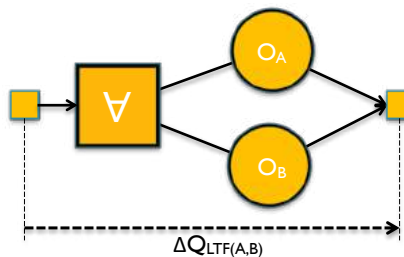
- Assume there are two possible outcomes  $O_A$  and  $O_B$  and exactly one outcome is chosen during each occurrence of a start event
- $O_A$  occurs with probability  $p/(p+q)$   
 $O_B$  occurs with probability  $q/(p+q)$
- Therefore:  

$$\Delta Q_{PC(A,B)} = \frac{p}{p+q} \Delta Q_A + \frac{q}{p+q} \Delta Q_B$$

77

77

## Last-to-finish semantics



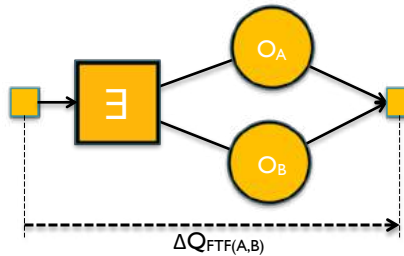
- Assume two independent outcomes with the same start event
- Last-to-finish outcome occurs when both end events occur
- $\Delta Q_{LTF(A,B)} = \Pr[d_A \leq t \wedge d_B \leq t] = \Pr[d_A \leq t] \times \Pr[d_B \leq t] = \Delta Q_A \times \Delta Q_B$
- Therefore:  

$$\Delta Q_{LTF(A,B)} = \Delta Q_A \times \Delta Q_B$$
 where  $\times$  is simple multiplication

78

78

## First-to-finish semantics

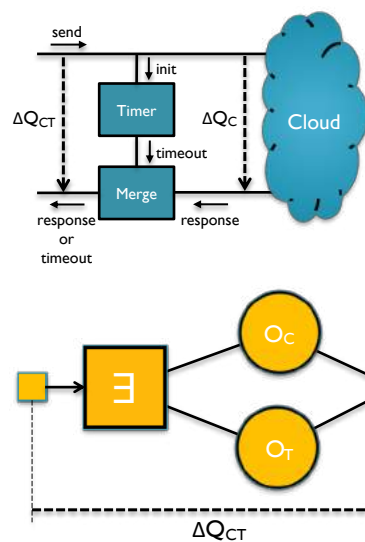


- Assume two independent outcomes with the same start event
- First-to-finish outcome occurs when at least one end event occurs
- We compute the probability that there are zero end events
- $(1 - \Delta Q_{FTF(A,B)}) = \Pr[d_A > t \wedge d_B > t]$   
 $= \Pr[d_A > t] \times \Pr[d_B > t] = (1 - \Delta Q_A) \times (1 - \Delta Q_B)$
- Simplifying gives:  
 $\Delta Q_{FTF(A,B)} = \Delta Q_A + \Delta Q_B - \Delta Q_A \times \Delta Q_B$

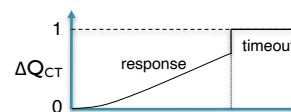
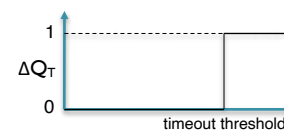
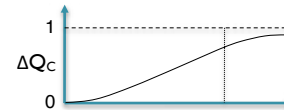
79

79

## 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:  
 $\Delta Q_{CT} = \Delta Q_C + \Delta Q_T - \Delta Q_C \times \Delta Q_T$



80

80



## Inverse computations

- When designing a system, it is common to make top-down decisions
  - We have the known  $\Delta Q$  of a component and we need to compute the required  $\Delta Q$  of a subcomponent
  - For sequential composition, this requires doing a **deconvolution**, which is the inverse of convolution
- For the other three operations this also requires doing an inverse computation
  - In most cases, there are **many possible  $\Delta Q$ s** for the subcomponent. The inverse computation therefore computes a **set of possible  $\Delta Q$ s** which defines a range of allowable behaviours for the subcomponent.

81

81

## 3. Cardano Shelley Block Diffusion Algorithm

82

82

## Context of block diffusion

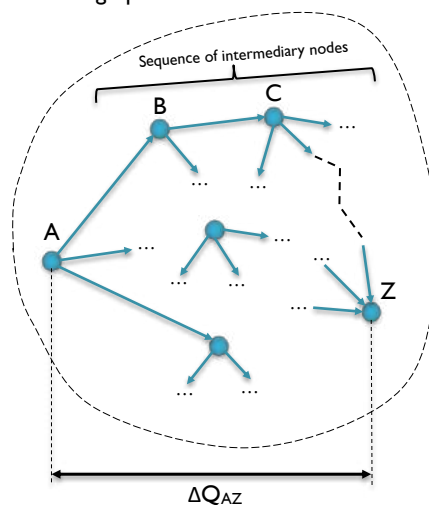
- Blockchain management in Cardano
  - We will use  $\Delta QSD$  to solve a design problem in the Cardano blockchain, which is an open-source platform using proof of stake
  - A blockchain is a **distributed ledger** comprising a chain of data blocks that are cryptographic witnesses to correctness of 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  $\Delta QSD$  to ensure that the algorithm satisfies stringent timeliness constraints


83

83

## Block diffusion problem statement

Node graph of Cardano blockchain

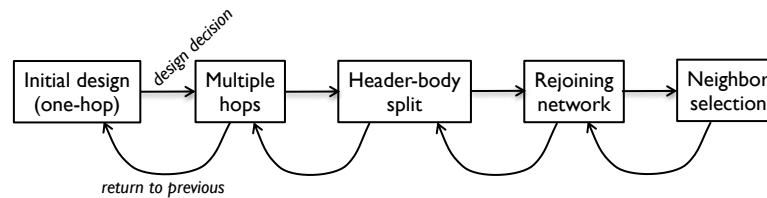


- Problem:
  - Determine  $\Delta Q_{AZ}$  for randomly chosen nodes A and Z, as function of design
  - Determine design so that  $\Delta Q_{AZ}$  satisfies performance constraints
  - $\Delta Q_{XY}$  is known (measured) 
- Design parameters:
  - Frequency of block production
  - Node connection graph
  - Block size
  - Block forwarding protocol
  - Block processing time

84

84

## Block diffusion design using $\Delta QSD$



- First step: preparation
  - Define an initial design and its outcome diagram
  - Measure  $\Delta Q$  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  $\Delta Q$ 
    - 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>

85

85

## Measuring $\Delta Q$

86

86

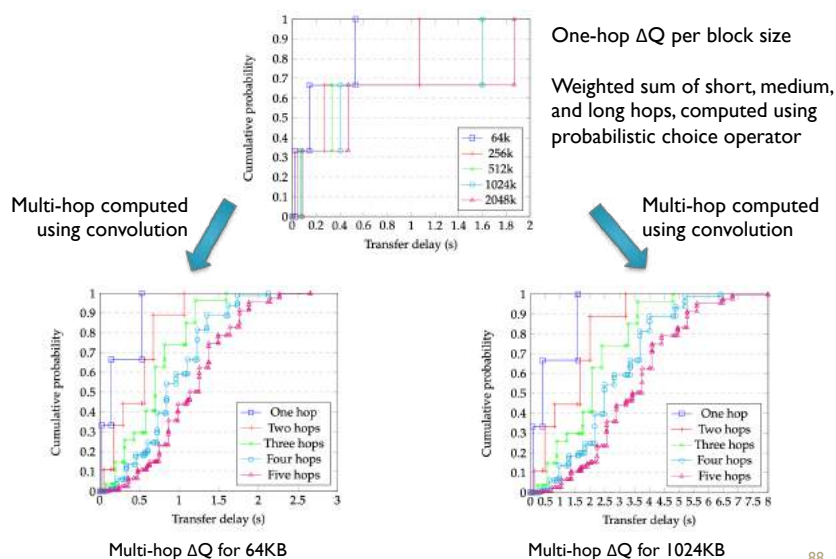
## First step: measuring $\Delta Q$

- First step is to measure  $\Delta Q$  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  $\Delta Q$ s are approximately step functions
  - **Multi-hop  $\Delta Q$ s** computed from single-hop (sequential composition operator, i.e., convolution)
  - **Random path  $\Delta Q$ s** computed from multi-hop (probabilistic choice operator, i.e., weighted sum)

87

87

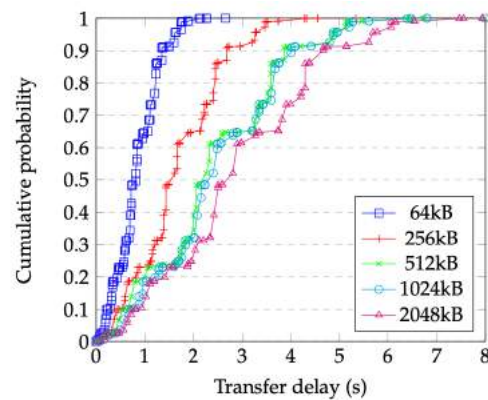
## Measured $\Delta Q$ for fixed paths



88

88

## Measured $\Delta Q$ for varying paths



- $\Delta Q$  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

89

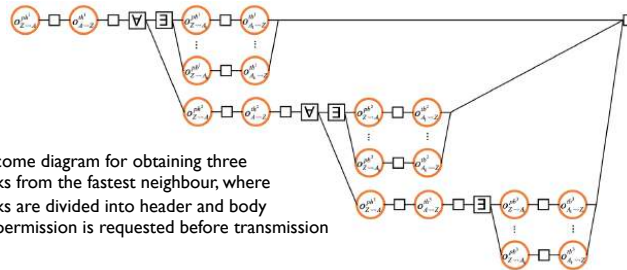
89

- Designing with outcome diagrams

90

90

## 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

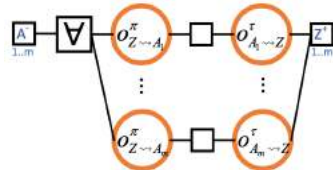
- For each design decision
  - Determine a new outcome diagram
  - Evaluate the effectiveness ( $\Delta Q$ ) using the outcome diagram
- This leads step by step to a final outcome diagram, which corresponds to the complete distributed system
  - Let us explain **one of the steps**, namely **obtaining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

91

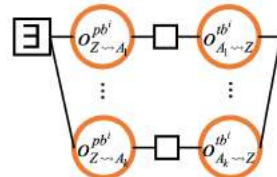
91

## Obtaining three blocks (I)

All-to-finish operator



First-to-finish operator

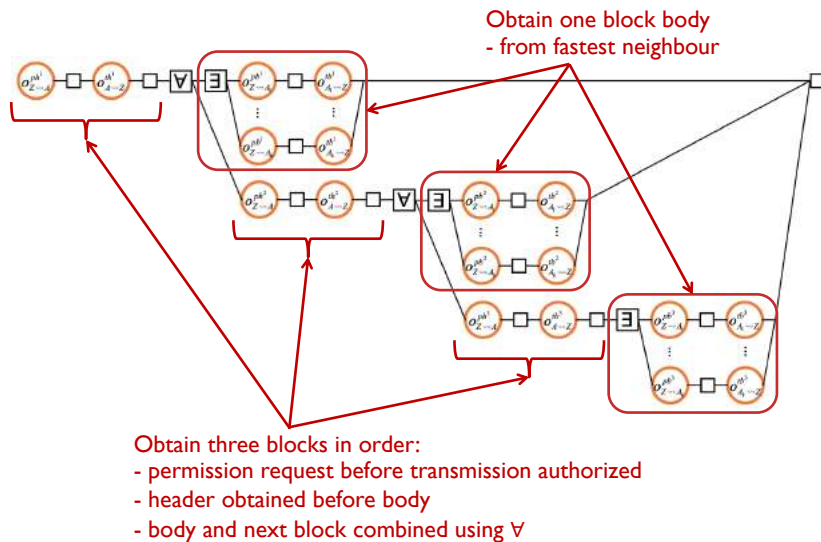


- We remind you of the two operators that are needed
- Obtaining one block from each neighbour uses the **all-to-finish operator** ( $\forall$ )
- Obtaining fastest block from one neighbour uses **first-to-finish operator** ( $\exists$ )

92

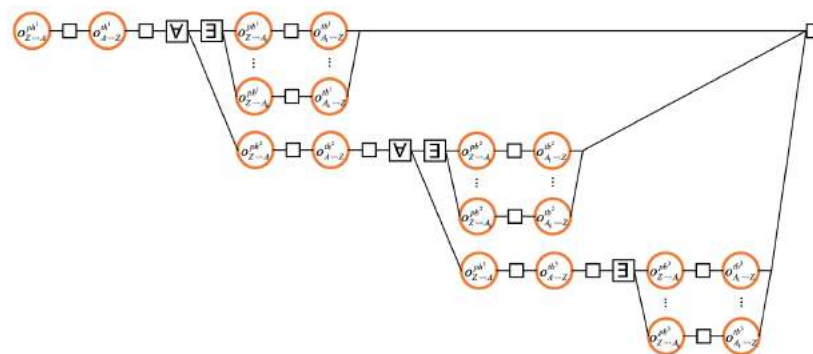
92

## Obtaining three blocks (2)



93

## Obtaining three blocks (3)



- The resulting outcome diagram correctly models the causality and performance of the block transfer;  $\Delta Q$  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

94

94





## 4. Some Typical $\Delta Q$ s

95

95

## Some typical $\Delta Q$ s

- Introduction to distributions
  - Gaussian distribution: used for aggregates
  - Uniform distributions: used for single parts
- Two parts that occur often in systems
  - Component 
    - We give the typical  $\Delta Q$  for a component
    - What happens when components are overloaded
  - Network 
    - We give the typical  $\Delta Q$  for a network
    - Effects of geography (distance), packet size, and random fluctuations

96

96





## Some typical distributions

97

97



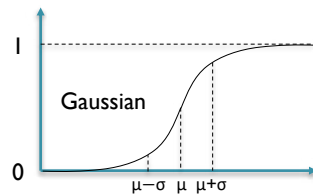
## Some typical distributions

- A tool can compute arbitrarily complex  $\Delta Q$ s
  - There is no limitation on the complexity of the  $\Delta Q$
- But it's still important to know some typical  $\Delta Q$ s
  - A good engineer always knows when something is possible or impossible with back-of-the-envelope calculations
- We give theory and intuition for two common distributions
  - Gaussian distribution: approximation for aggregates
  - Uniform distributions: approximation for single parts

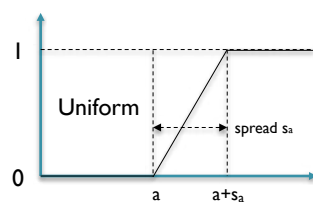
98

98

## Gaussian and Uniform distributions



- A **Gaussian distribution** approximates the sum of many independent random quantities (Central Limit Theorem)
  - $\mu$  is the mean
  - $\sigma$  is the standard deviation
- Gaussian is a good approximation for aggregates, but not for single parts
  - Gaussians have infinite tails!

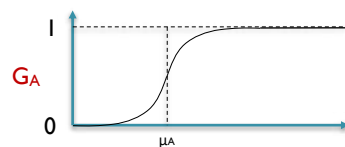


- A **Uniform distribution** approximates one part of a system (component or network)
  - $a$  is the minimum time in the part
  - $s_a$  is the spread of times in a part
  - $a+s_a$  is the maximum time in the part
- Uniform is a good approximation for single parts, but not for many connected parts

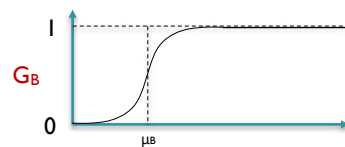
99

99

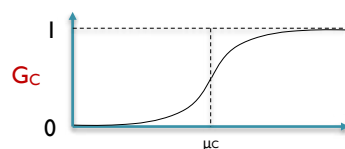
## Convolution of Gaussian distributions



⊕



=

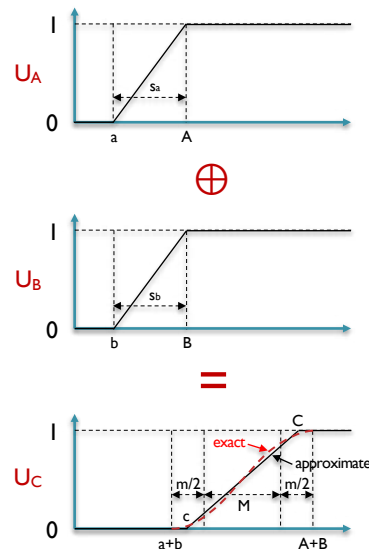


- Formulas: (exact)
  - $G_A = (\mu_A, \sigma_A)$
  - $G_B = (\mu_B, \sigma_B)$
  - $G_C = G_A \oplus G_B = (\mu_C, \sigma_C)$
  - $\mu_C = \mu_A + \mu_B$
  - $\sigma_C^2 = \sigma_A^2 + \sigma_B^2$
  - $\sigma_C = \sqrt{\sigma_A^2 + \sigma_B^2}$
- In other words:
  - Means are added
  - Squares of standard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

100

100

## 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

101

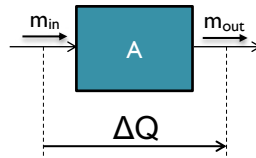
101

•  $\Delta Q$  for a typical component  
(from queuing theory)

102

102

## A component as a queue

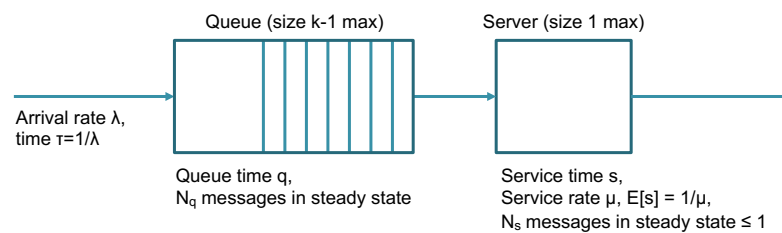


- Let's get some more intuition on how a component works
  - To get this intuition, we model the component as a queue
- A typical component has four parameters of interest
  - **Offered load a**: arrival rate / service rate of messages
  - **Buffer size k**: number of messages stored inside
  - **Failure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

103

103

## M/M/1/K queue



- We model a component as an M/M/1/K queue
  - M: arriving messages have Exponential distribution with rate  $\lambda$
  - M: service time has Exponential distribution with rate  $\mu$
  - 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)
- 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)
  - $\Delta Q$ , i.e., failure rate  $f$  and average delay  $d$ , is function of  $a$  and  $k$

104

104

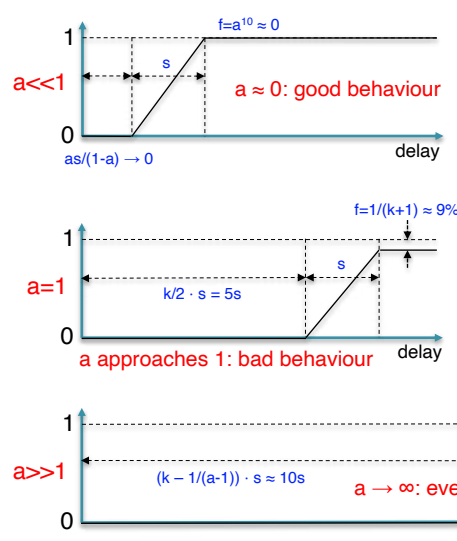
## Effect of offered load $a$

- 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 ( $a \geq 0.8$ )**
  - When  $a$  gets close to 1 (around 0.8) things quickly get worse!
  - When  $a \gg 1$ , 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*

105

105

## $\Delta Q$ as function of load $a$



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

106

106

## 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  $k \rightarrow \infty$  the failure rate  $f \rightarrow 0$  and delay  $\rightarrow 1/(1-a)$  (close to 1)
    - Big buffers are good at low load
- High load ( $a > 0.8$ )
  - Failure rate and delay are both high
  - Bigger buffer greatly increases delay (around  $k/2$  for big  $a$ )
    - Big 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:
      - (1) don't ever overload not even temporarily, (2) keep buffer size small

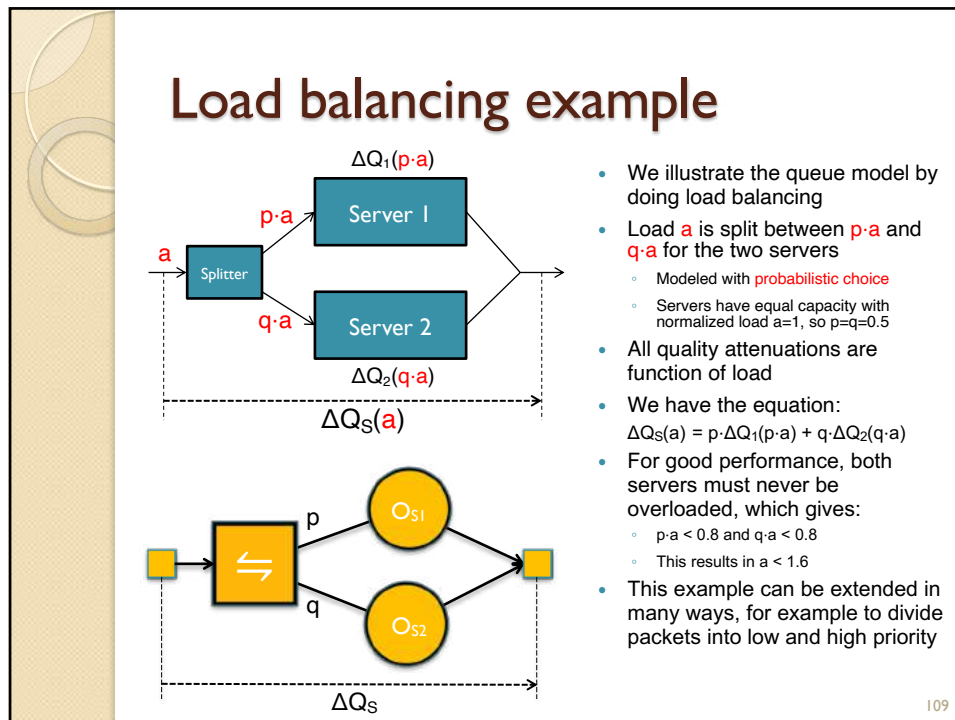
107

107

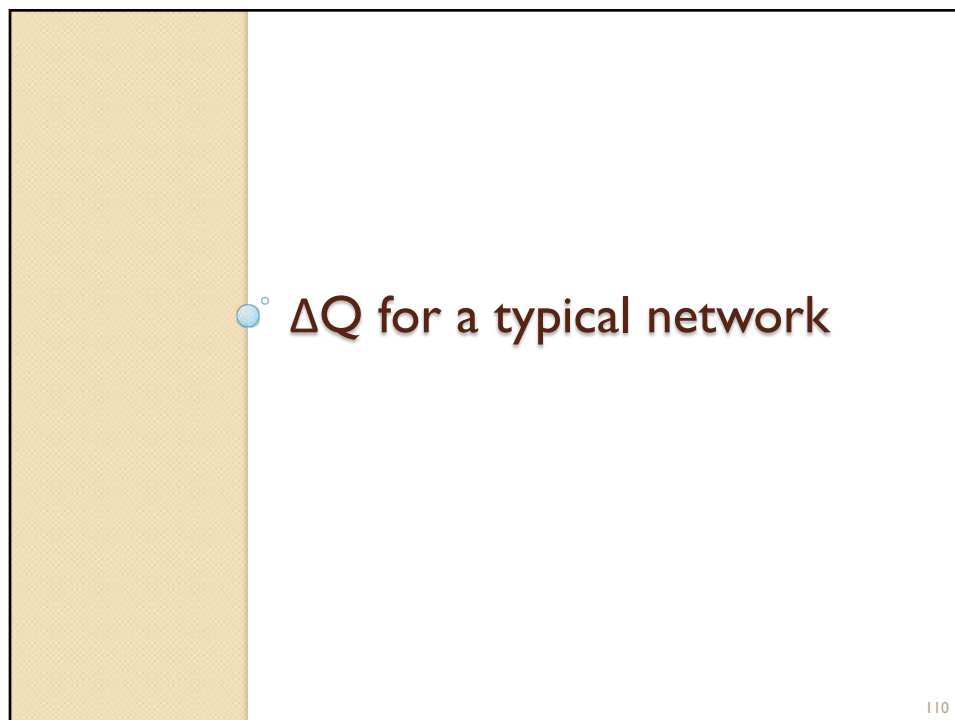
## Load balancing example

108

108

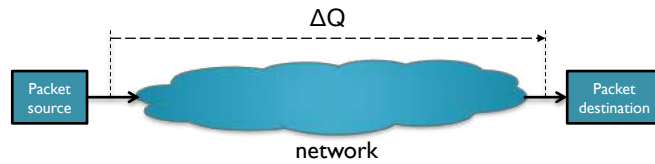


109



110

## $\Delta Q$ for network packets

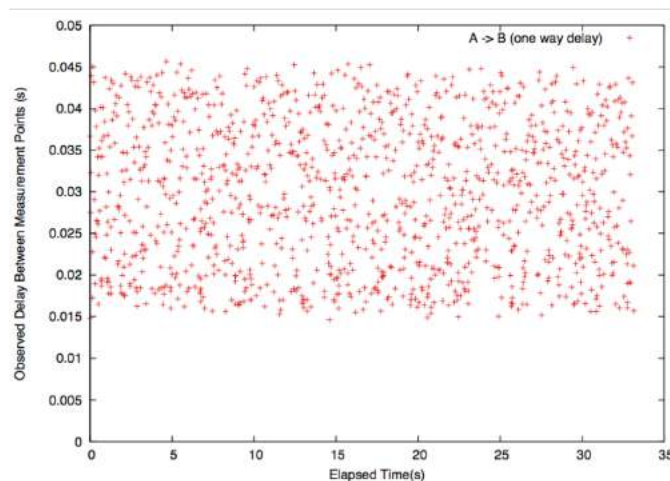


- We can study what the real  $\Delta Q$  is for networks delivering packets
- Experience shows that the  $\Delta Q$  has four parameters G, S, V, L:
  - $\Delta Q = \Delta Q_G \oplus \Delta Q_S \oplus \Delta Q_V \oplus \Delta Q_L$
- Again, we add  $\Delta Q$ s using convolution
  - Because of their simple structure, convolution is easy

III

111

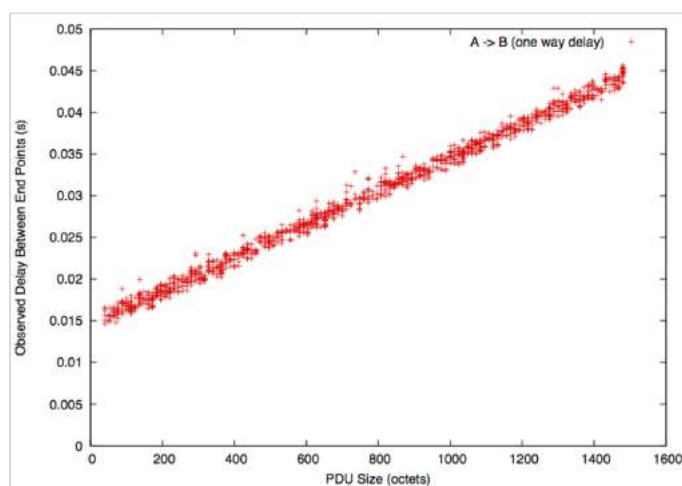
## Raw two-point measurements



112

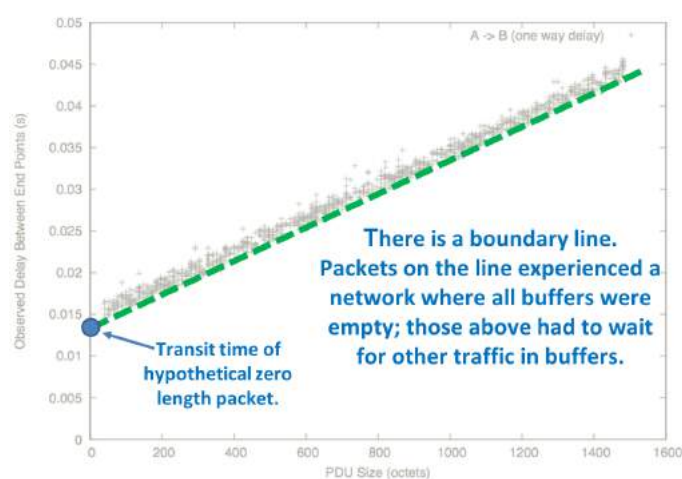


## Measurements sorted by packet size



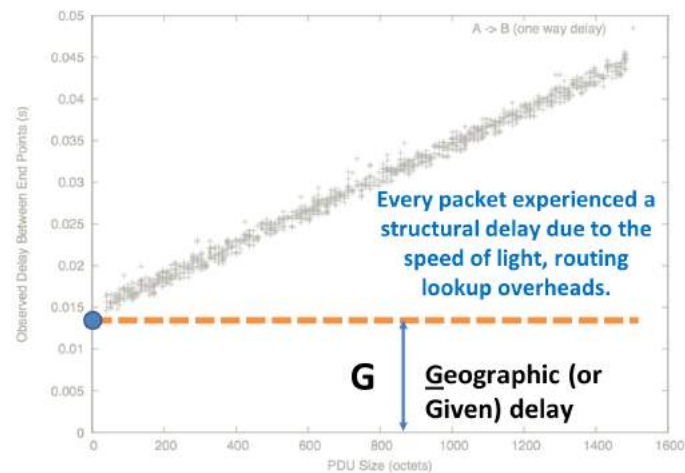
113

## Minimum delays for each size



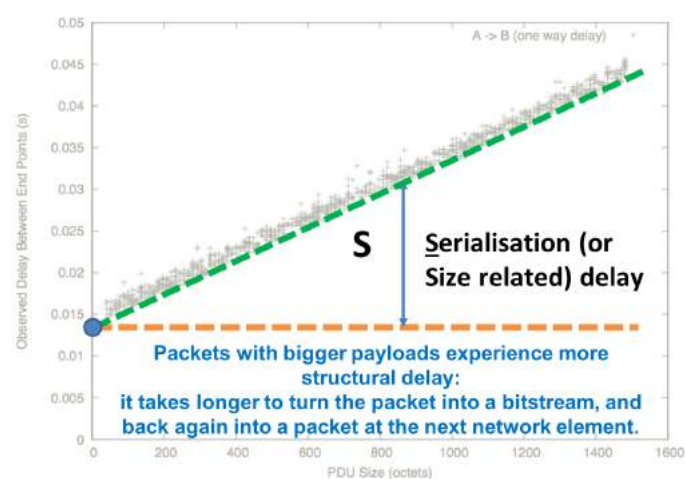
114

## Extrapolate to zero size packet: G



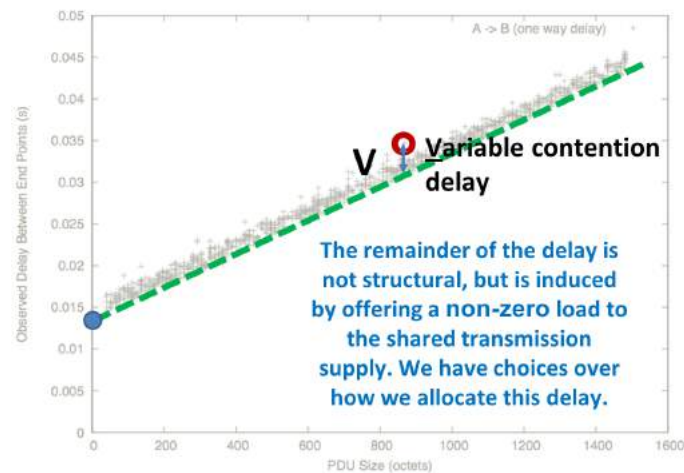
115

## Extract S



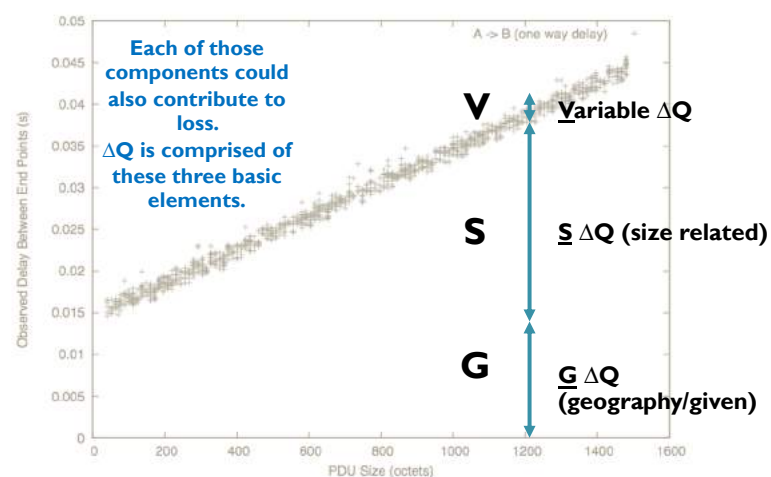
116

## V is what remains



117

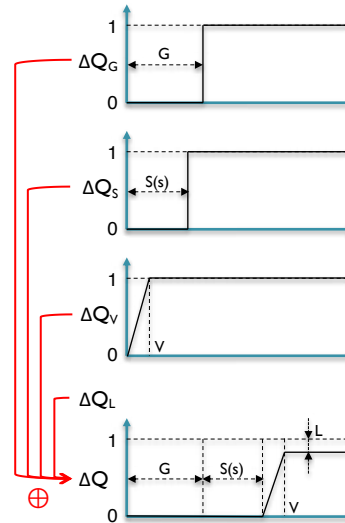
## G, S, V from measured $\Delta Q$



118

## Overall network $\Delta Q$

- Total network  $\Delta Q$  is the sum of the three parts:
  - Geographic delay  $G$
  - Size-related delay  $S(s)$  function of packet size  $s$ 
    - $\Delta Q_s$  is a step function if there is a constant packet size, otherwise it has another shape
  - Variability  $V$  function of contention and noise
- In addition, there is a percentage  $L$  of lost packets



119

119

120

120

## Lecture 3 Systems with Dependencies

121

121

## Systems with dependencies

- $\Delta$ QSD 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 handle multiple timescales
- Realistic systems have some dependent parts
  - Most of the system consists of independent parts
  - A few dependencies are added, for example where two message streams use the same database
- Topics for Lecture 3
  - Shared resources
  - Variable load (iterative query example)

122

122

## Nonlinearity

- The outcome diagrams of Lecture 2 describe systems with independent components
  - Overall  $\Delta Q$  is a **linear combination** of the component  $\Delta Q$ s
    - It is **compositional**: we can decompose the outcome diagram into parts, compute  $\Delta Q$ s separately, and then combine them
  - Nevertheless, this  $\Delta Q$  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
- Lecture 3 adds even more nonlinearity
  - Adding dependencies between components is another source of nonlinearity
  - A shared resource can also overflow and cause a major change in  $\Delta Q$

123

123

## Shared resources

124

124

## Shared resources

- Computing  $\Delta Q$  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  $\Delta Q$  is done by adding the equations to the solver
- 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  $\Delta Q$  to become bottom (failure: no result). If there is still some functionality, it is not a threshold resource.

125

125

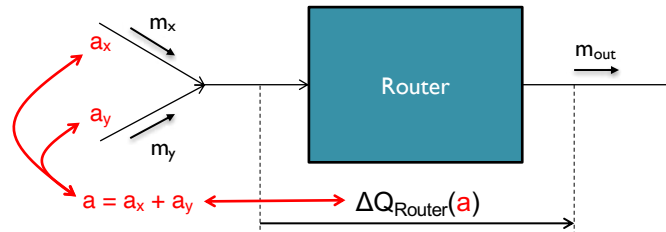
## Examples of shared resources

- **Ephemeral, not threshold**
  - (1) A network connection. When capacity of the line is exceeded or there is congestion, the  $\Delta Q$  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).

126

126

## Example 1: congestion

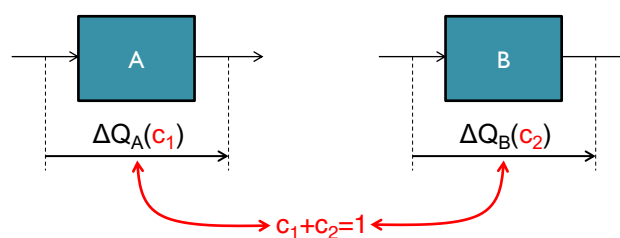


- Assume two message streams entering the same component (e.g., a router)
  - Total load is the sum of the two incoming loads:  $a = a_x + a_y$
  - Sharing is modeled as the sum of loads
- Congestion, i.e., buffer overflow and message drop, is computed from  $\Delta Q_{Router}$  using the queue model 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

127

127

## 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  $c_1 + c_2 = 1$
  - $\Delta Q$  of each component is function of its processor utilisation
- This gives extra arguments  $c_1$  and  $c_2$  to the  $\Delta Q$ s and an equation (constraint) linking them

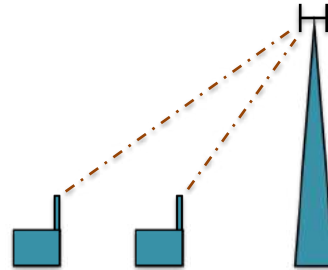
128

128



## Example 3: shared cell tower

- Two carriers share a cell tower
- Each carrier is guaranteed 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 70% 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: (with loads  $a_{C1}(t)$  and  $a_{C2}(t)$ )
  - $\forall t. \text{ If } a_{C1} \geq 0.5 \text{ then } a_{C1} \leq 0.5 + \max(0, 0.5 - a_{C2})$
  - $\forall t. \text{ If } a_{C2} \geq 0.5 \text{ then } a_{C2} \leq 0.5 + \max(0, 0.5 - a_{C1})$

129

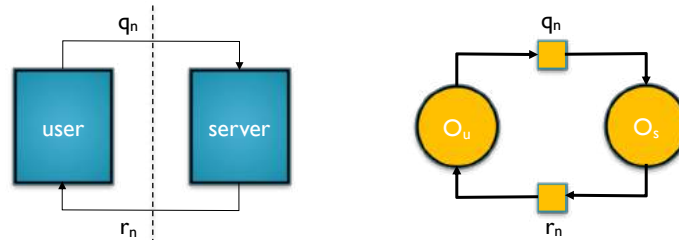
129

## Iterative query

130

130

## Systems with iterative queries

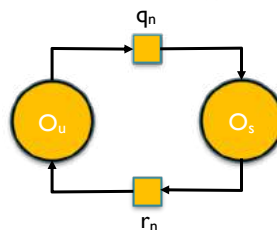


- Consider an iterative process where user sends query  $q_n$  to server which sends response  $r_n$  back to user, which sends query  $q_{n+1}$  and so forth
  - This is a common structure: it models many human-computer interactions on the Web, it models software doing iterative queries to a database, and many other repetitive processes
- How do we compute the  $\Delta Q$  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:  $O_{s,0} < O_{u,0} < O_{s,1} < O_{u,1} < \dots$

131

131

## $\Delta Q$ 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 the expected rate of queries (queries per second):

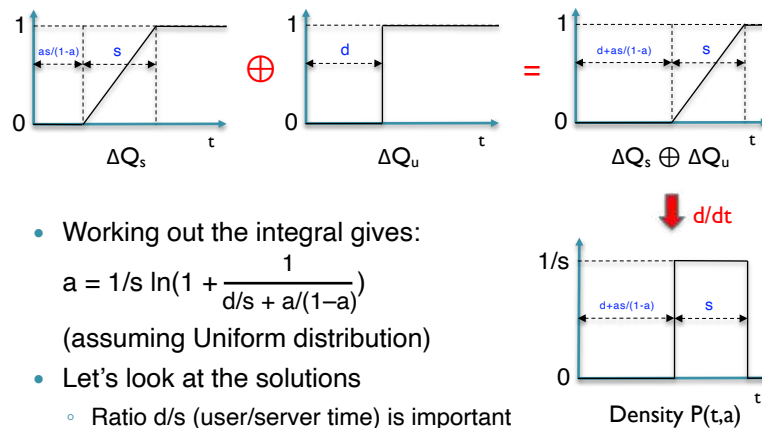
$$a = \int_0^{\infty} \frac{1}{t} P(t, a) dt$$

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

132

132

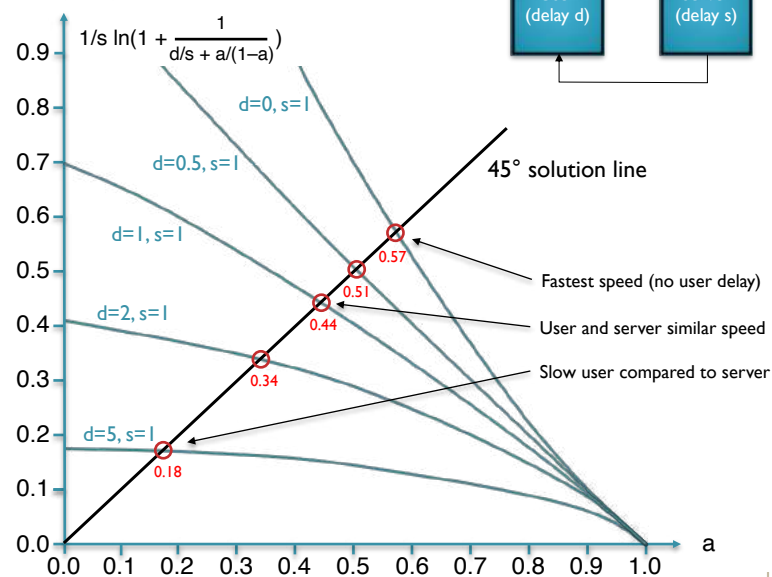
## Solving the equations



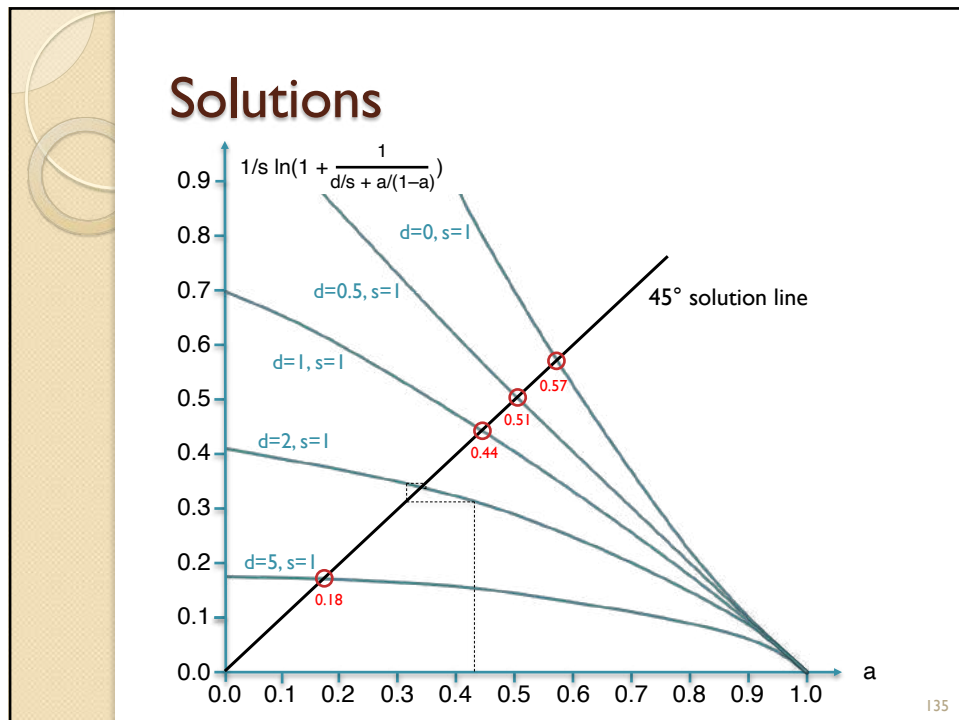
133

133

## Solutions



134



135

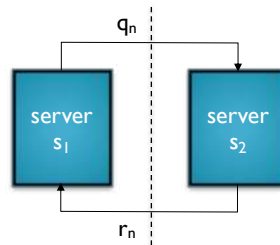
## How to measure load

- 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 **work done per unit of time**
    - 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  $2d$

136

136

## 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

137

137

138

138



## Lecture 4 Multilevel Systems

139

139



## There is always a cliff

- 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
    - It goes wrong when we go beyond where it has gone before
- There is always a cliff
  - But we don't know where it is
  - The system works perfectly until we fall off the cliff!

140

140

## 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
- Topics for Lecture 4
  - Managing risk
  - Multilevel system design
  - Supermarket scenario
  - Multiple timescales

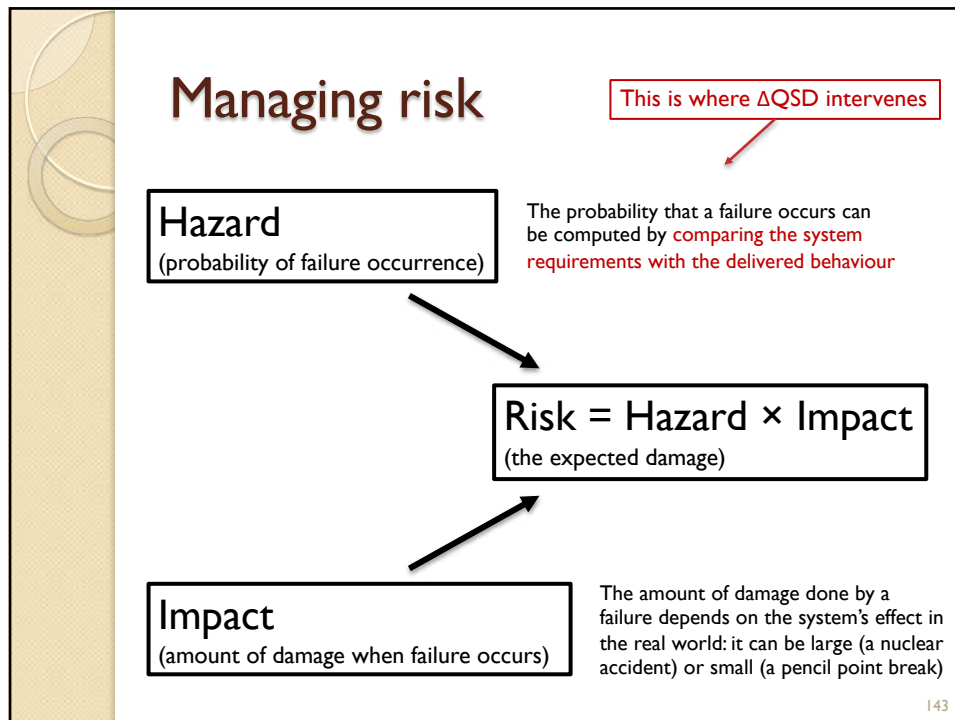
141

141

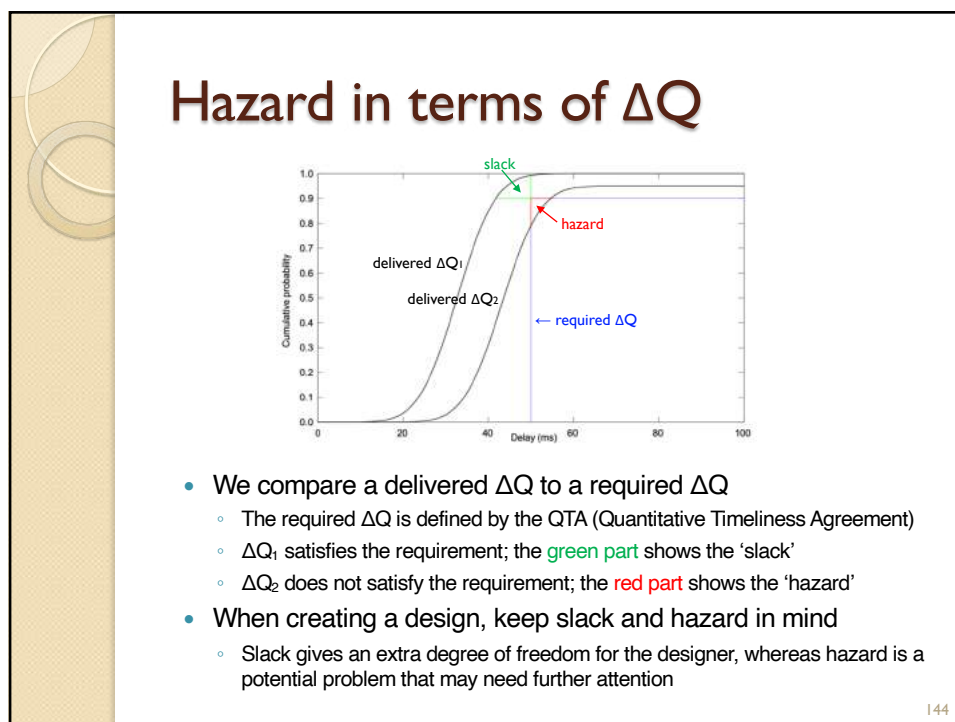
## Managing risk

142

142



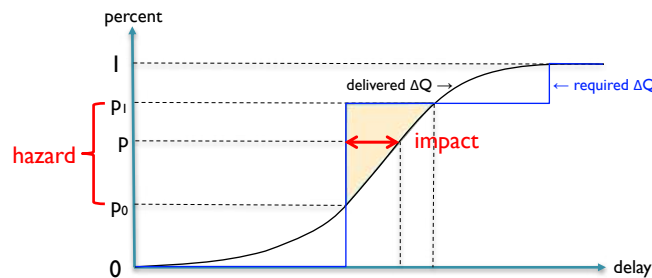
143



144



## Computing hazard and risk



- Risk = impact times probability of occurrence
  - Hazard = probability of occurrence of the failure =  $p_1 - p_0$
  - Impact = cost (i.e., extra delay) when the failure occurs =  $i(p)$
- Because  $\Delta Q$  is a probability distribution, this is an integral
- $r = \int i \, dp$ 
  - Total risk is area of orange triangular part
  - Risk = expected damage, i.e. expected extra delay

145

145

## Managing hazards

- The number of possible hazards is unbounded
  - "Limited only by your imagination"
  - Theoretically, the system should be designed to handle all possible hazards, but this is too costly and time-consuming
  - How do we bring order into this situation?
- New hazards appear in new systems
  - If you are building a system that goes beyond previous systems, then new hazards will appear
  - New hazards are hard to predict, but  $\Delta QSD$  gives some useful tools: it can compute  $\Delta Q$  for high loads and it organizes hazards according to their order
- Managing the hazards
  - To manage a hazard, you look at all the ways that it can occur and you reduce the probability that one of these will happen

146

146

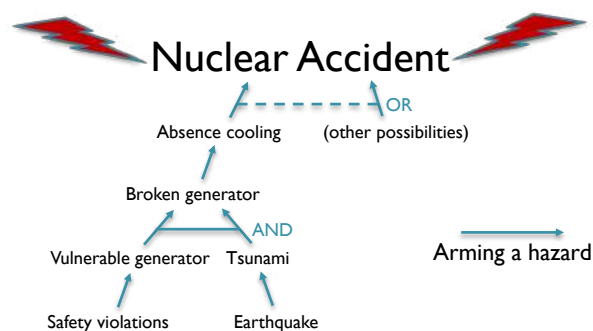
## Avoid “arming the hazards”

- A hazard can be quantified through its preconditions
  - Enabling a precondition gives the hazard a nonzero probability
    - This is called **arming the hazard**
  - The system should be designed to avoid arming the hazards
- Example: the 2011 Fukushima nuclear disaster
  - Caused by an earthquake and a followup tsunami
  - The earthquake caused the reactor to shut down normally
  - But the tsunami broke the generators needed to cool the core
    - Safety regulations should have avoided this, but they were violated
  - Absence of cooling caused a severe nuclear accident
- There was a chain of preconditions arming the hazard
  - Safety violations armed the broken generator hazard
  - Broken generator armed the nuclear accident hazard

147

147

## Causal graph of preconditions



- We show a graph of preconditions for the Fukushima accident
  - Safety violations → Vulnerable generator
  - Vulnerable generator  $\wedge$  Tsunami → Broken generator
- The hazards are armed like a chain of falling dominos
  - Each precondition arms the next one

148

148

## Performance hazards

- Now let us focus on performance hazards
  - Hazards that affect performance of a system
- There is a hierarchy of performance hazards
  - Depending on how the system is stressed
  - Does the system work as expected under normal load?
  - What happens when hazards are armed:
    - By internal preconditions (e.g., scheduled maintenance)?
    - By external preconditions (e.g., load peaks)?
- We classify hazards according to preconditions
  - How easy it is to control the preconditions to avoid arming
  - Some preconditions are controlled by correct design
  - Some preconditions are hard to control (external effects)

149

149

## Order of performance hazards

	Order	Subject of concern
Compositional	0: Causality	<b>Causal behavior</b> is the only requirement. If $\Delta Q$ is best possible, can the system deliver its successful top-level outcomes, i.e., can the system ever work if causality is respected?
	1: Capacity	<b>Markovian</b> (independent) and <b>linear</b> (superposition) behaviour. Will the delivered $\Delta Q$ be within requirements at expected loads, i.e., constant average load within capacity constraints?
Dependent	2: Schedulability	<b>Expected variability</b> in behaviour which can be managed by proper scheduling. Can the QTAs be maintained during reasonable operational stress, i.e., expected load variability?
	3: Behaviour	Is the system sensitive to <b>internal correlation effects</b> , i.e., interactions between subsystems due to internal effects? For example, all devices doing http lookup at midnight.
	4: Stress	Is the system sensitive to <b>external correlation effects</b> , 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
- $\Delta Q$  computation techniques depend on the order of hazards
  - Orders 0 and 1 assume independence; orders 2, 3, 4 introduce sharing

150

150

## Multilevel system design

151

151

## Multilevel system design

- In lectures 2 and 3 we saw how to design systems with predictable performance
  - Outcome diagrams show the causal connections
  - We add dependencies where they occur
- This works for systems with limited perturbations
  - But it is not good enough for safety-critical systems that must continue to work under all circumstances
  - You might think it is a good idea to add mechanisms inside the system to avoid arming the hazards
    - This does not work!
- Multilevel systems
  - The right solution is to add mechanisms **outside the system**
    - **Build a hierarchy of observers** where each observer manages the hazards that are not managed by observers at a lower level
  - The original system is the base of the hierarchy

152

152

## Base system

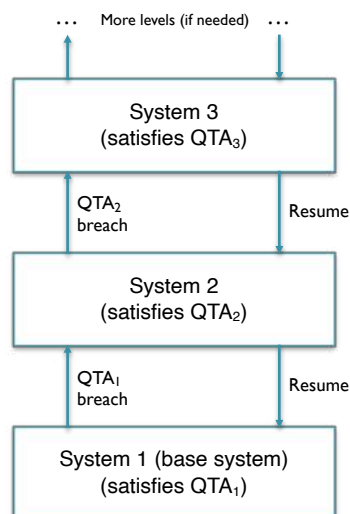
Base system  
(satisfies QTA)

- Our starting point is a base system that satisfies a QTA
  - The QTA (Quantitative Timeliness Agreement) specifies what the base system does and its limits
  - In lectures 2 and 3 we saw how to design this system using outcome diagrams with dependencies
- What happens when the QTA is no longer satisfied?
  - Some hazard has occurred
  - The system no longer provides its service to its users
- Multilevel design targets this situation

153

153

## Multilevel system

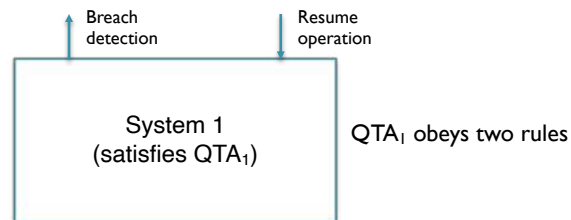


- During normal operation, System 1 does the work
  - The other systems monitor this but normally do not intervene
  - Upon QTA<sub>1</sub> breach, System 2 is notified
- System 2 has several options:
  - Take over from System 1, temporarily or permanently
  - Reconfigure system 1 and then resume it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

154

154

## System I design constraints



- A base system in a multilevel system must be designed right
- 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

155

155

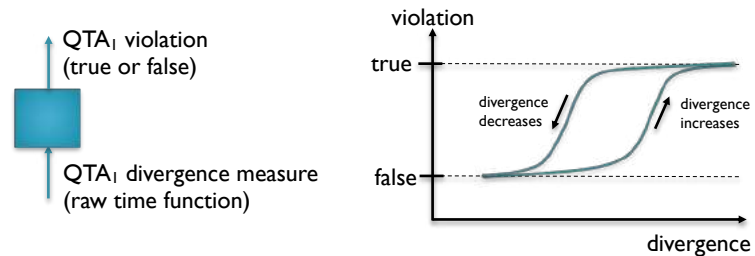
## “mitigate or propagate”

- The general principle of a multilevel system is called “mitigate or propagate”
  - First try to solve the problem inside the current system (mitigate), if this is not possible go to the next (propagate)
- “mitigate”
  - Each level first tries to solve the problem by itself
  - This is why we give rules for  $QTA_1$ : you can behave badly but you should not break, and you should always provide some minimal functionality
  - Don’t try to mitigate too much; there may be interactions between the mitigation strategies
- “propagate”
  - Propagation can be simpler than mitigation, because each system is cleanly separated from the others (modular)

156

156

## Breach detection



- Base system does continuous measurement of QTA<sub>I</sub> divergence
  - Quantified hazard, by comparison of QTA<sub>I</sub> and delivered  $\Delta Q$
- 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

157

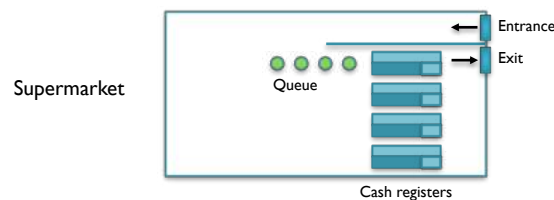
157

## Supermarket example

158

158

## Supermarket example

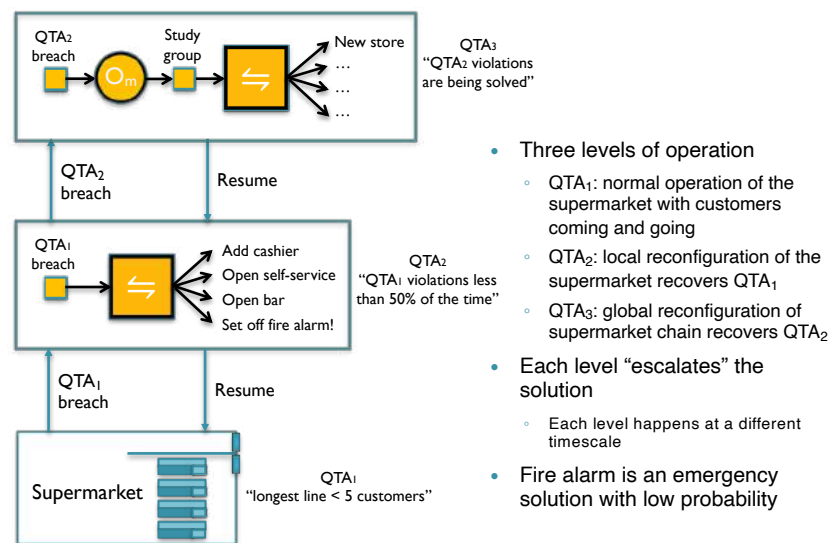


- To illustrate the approach we design a supermarket
  - Customers enter the supermarket, collect their merchandise, and queue up at an open cash register
- QTA<sub>1</sub> = "less than 5 customers are in line"
  - What happens when there are too many customers in a line?
  - What do the next levels look like?

159

159

## Supermarket layered system

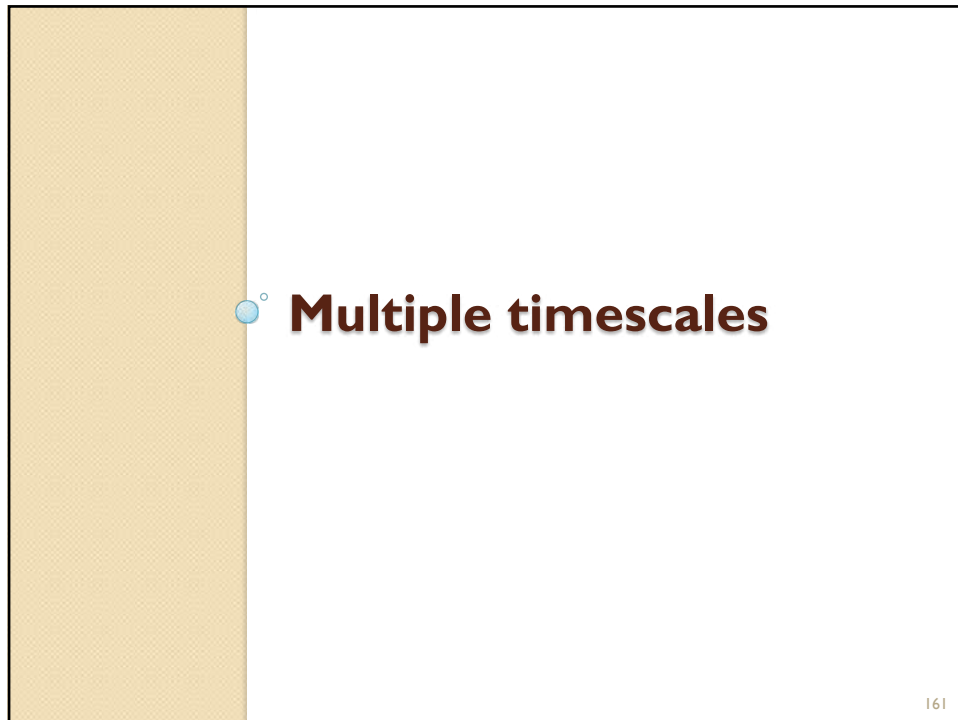


- Three levels of operation
  - QTA<sub>1</sub>: normal operation of the supermarket with customers coming and going
  - QTA<sub>2</sub>: local reconfiguration of the supermarket recovers QTA<sub>1</sub>
  - QTA<sub>3</sub>: global reconfiguration of supermarket chain recovers QTA<sub>2</sub>
- Each level "escalates" the solution
  - Each level happens at a different timescale
- Fire alarm is an emergency solution with low probability

160

160





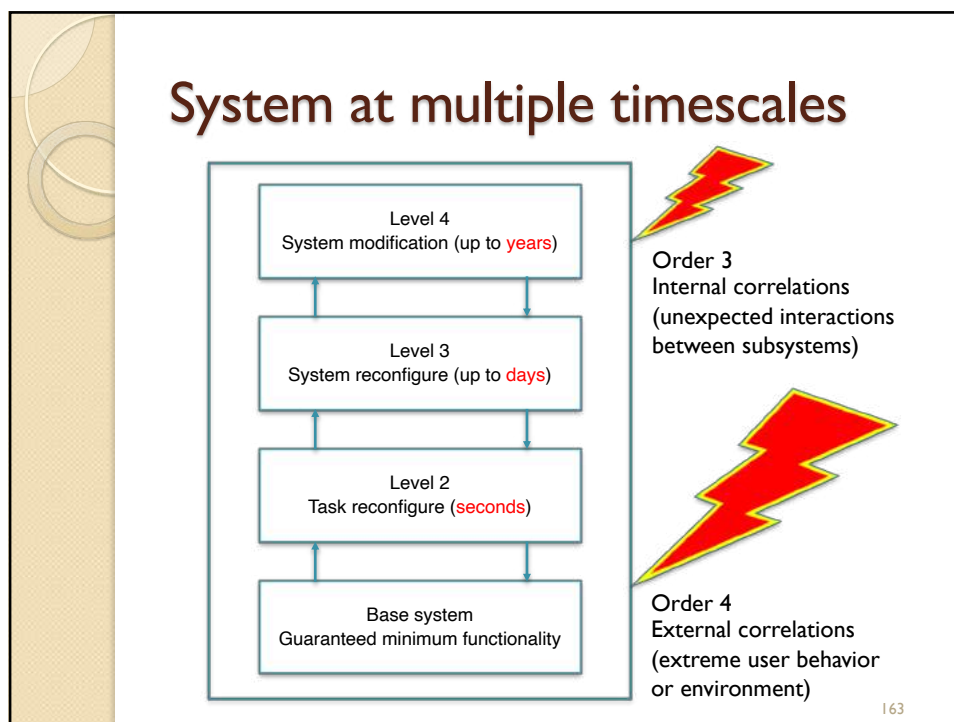
161

 A presentation slide with a light beige background and a darker beige vertical bar on the left. The title "Multiple timescales" is centered in a dark brown font. Below the title is a bulleted list. The slide number "162" is in the bottom right corner.
 

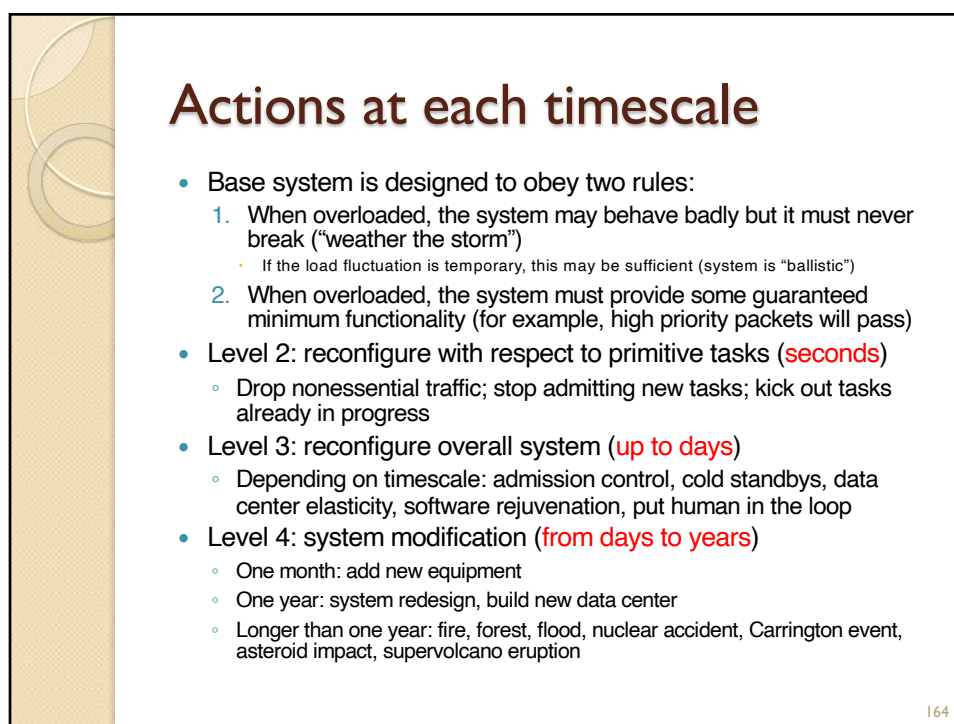
## Multiple timescales

- The system must be designed to deal with overload (hazard orders 3 and 4)
  - Ideally the load never approaches 1
    - As we saw before, when  $a > 0.8$  things get bad very quickly
  - But it will almost certainly happen **if we wait long enough**
    - 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**, where each timescale is handled at its own level
    - Time to react depends on timescale of the hazard
    - $\Delta QSD$  is used to define the actions at each level
  - Software must be as idempotent as possible and non-idempotent parts should be isolated

162



163



164

## Conclusions

165

165

## Advantages of $\Delta$ QSD

- $\Delta$ QSD 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** (cdf's, 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
- $\Delta$ QSD **cleanly factors the design** into three parts
  - Compositional system made of independent parts
  - Adding dependencies between components
  - Adding multilevel risk management

166

166

## Limitations of $\Delta$ QSD

- $\Delta$ QSD **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
- $\Delta$ QSD works best for **systems with independent actions**
  - For systems that execute long sequences of dependent actions, the predictions will be less accurate
- Achieving  $\Delta$ QSD'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

167

167

## Conclusions and future steps

- This tutorial introduces  $\Delta$ QSD but there is much more:
  - Practical measurement and computation of  $\Delta$ Q
  - 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  $\Delta$ QSD
  - Experience reports for large industrial projects
- Ongoing project to formalize  $\Delta$ QSD 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>

168

168