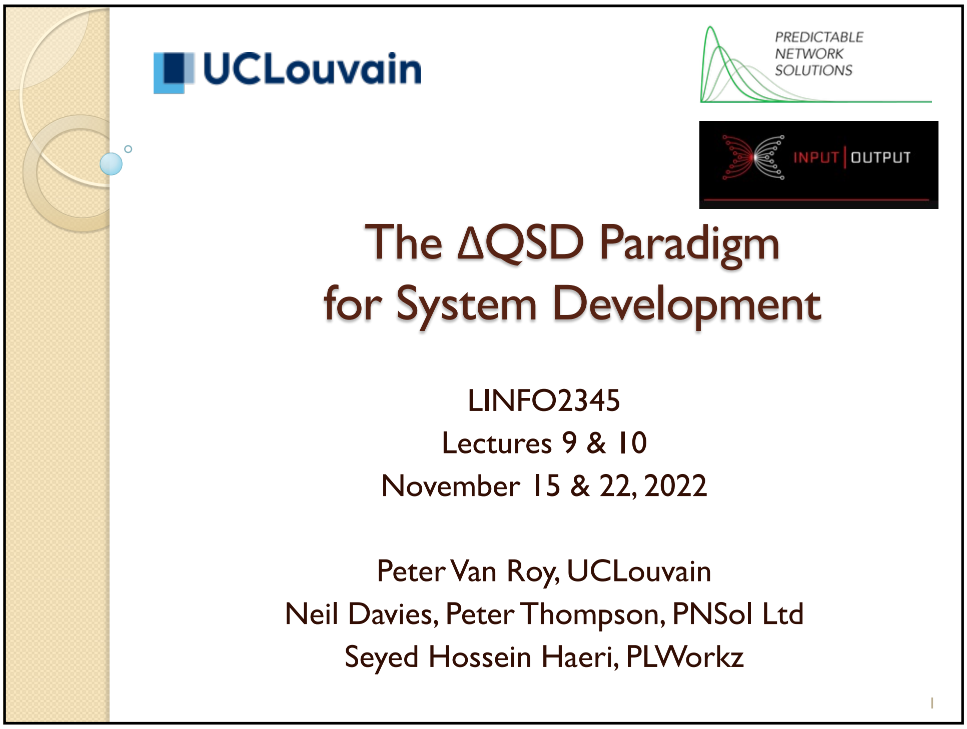


UCLouvain

PREDICTABLE NETWORK SOLUTIONS

INPUT | OUTPUT

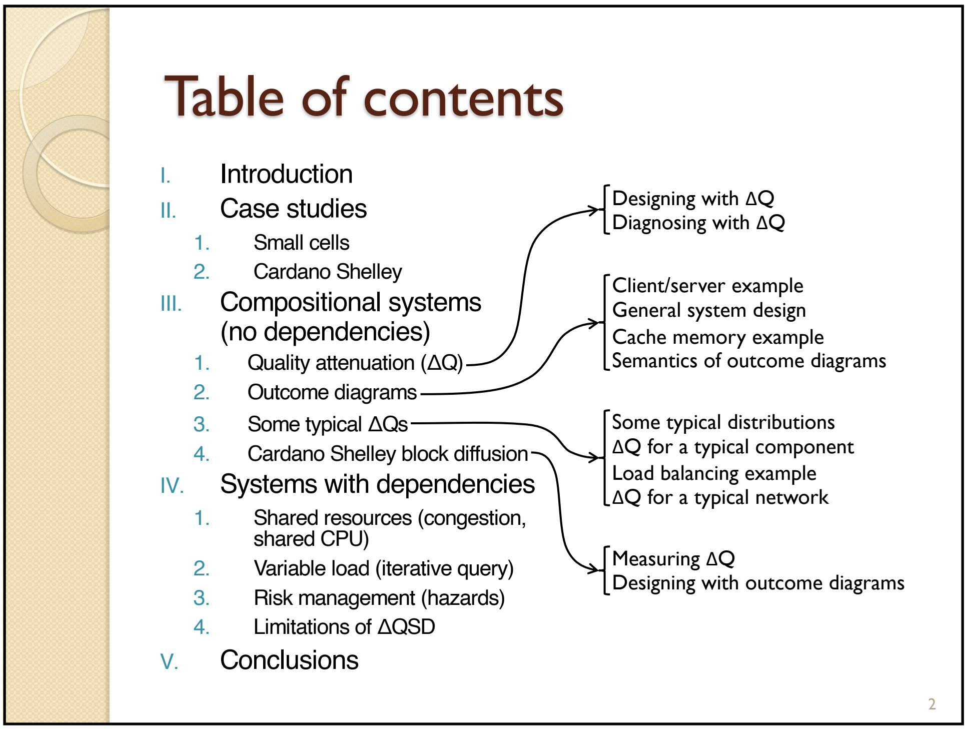
# The ΔQSD Paradigm for System Development

LINFO2345
Lectures 9 & 10
November 15 & 22, 2022

Peter Van Roy, UCLouvain
Neil Davies, Peter Thompson, PNSol Ltd
Seyed Hossein Haeri, PLWorkz

1

1



## Table of contents

I. Introduction
II. Case studies
  1. Small cells
  2. Cardano Shelley
III. Compositional systems (no dependencies)
  1. Quality attenuation (ΔQ)
  2. Outcome diagrams
  3. Some typical ΔQs
  4. Cardano Shelley block diffusion
IV. Systems with dependencies
  1. Shared resources (congestion, shared CPU)
  2. Variable load (iterative query)
  3. Risk management (hazards)
  4. Limitations of ΔQSD
V. Conclusions

Designing with ΔQ
Diagnosing with ΔQ

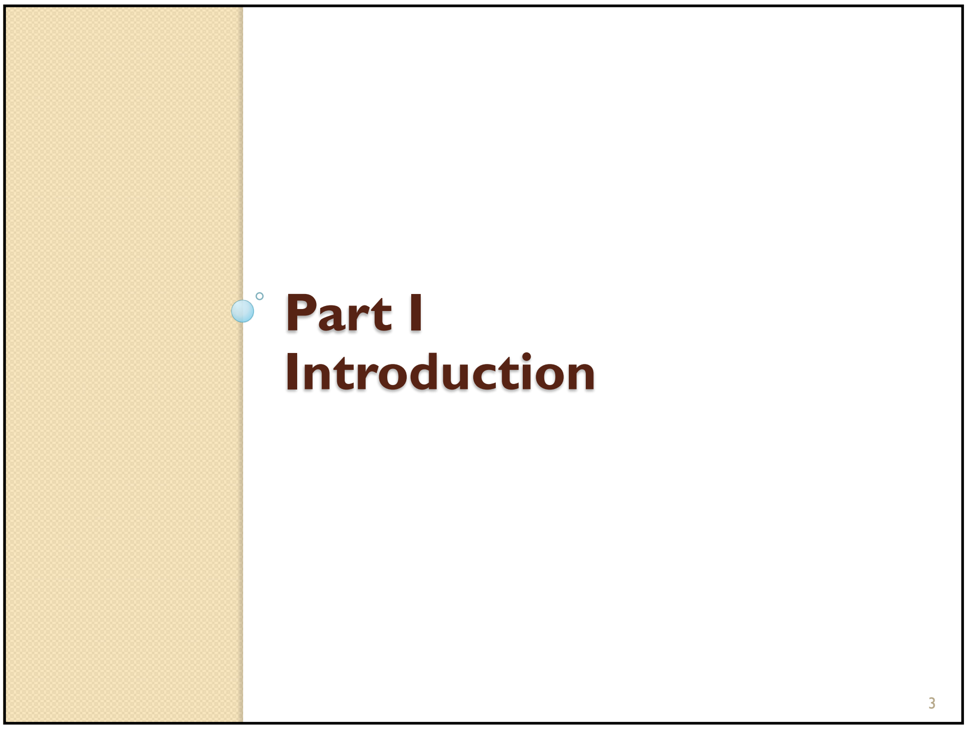
Client/server example
General system design
Cache memory example
Semantics of outcome diagrams

Some typical distributions
ΔQ for a typical component
Load balancing example
ΔQ for a typical network

Measuring ΔQ
Designing with outcome diagrams

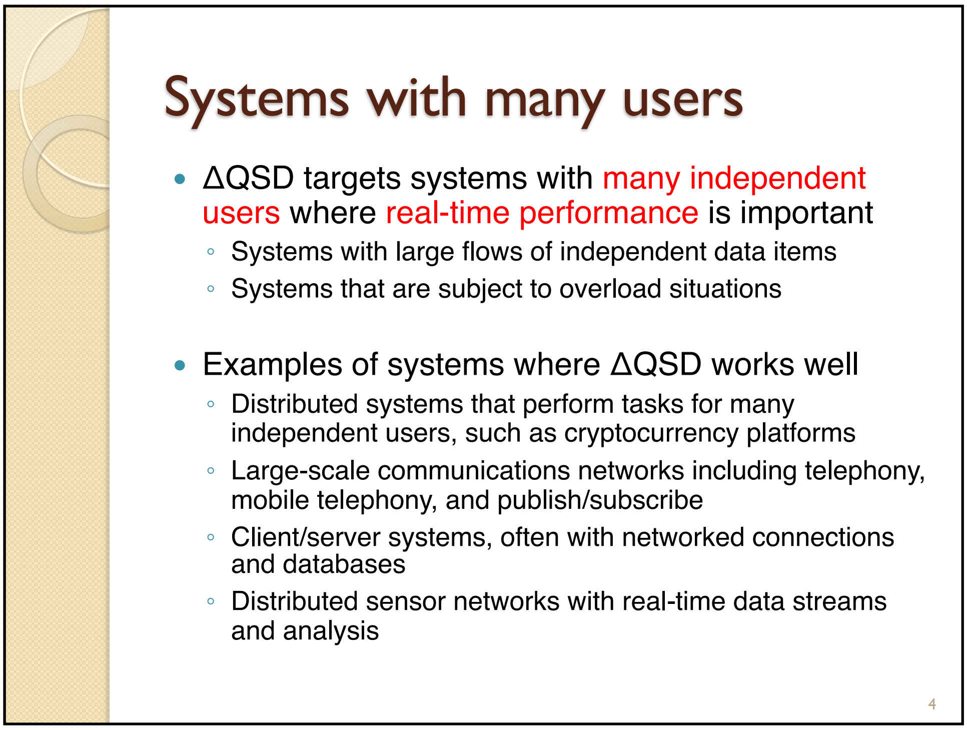
2

2



## Part I Introduction

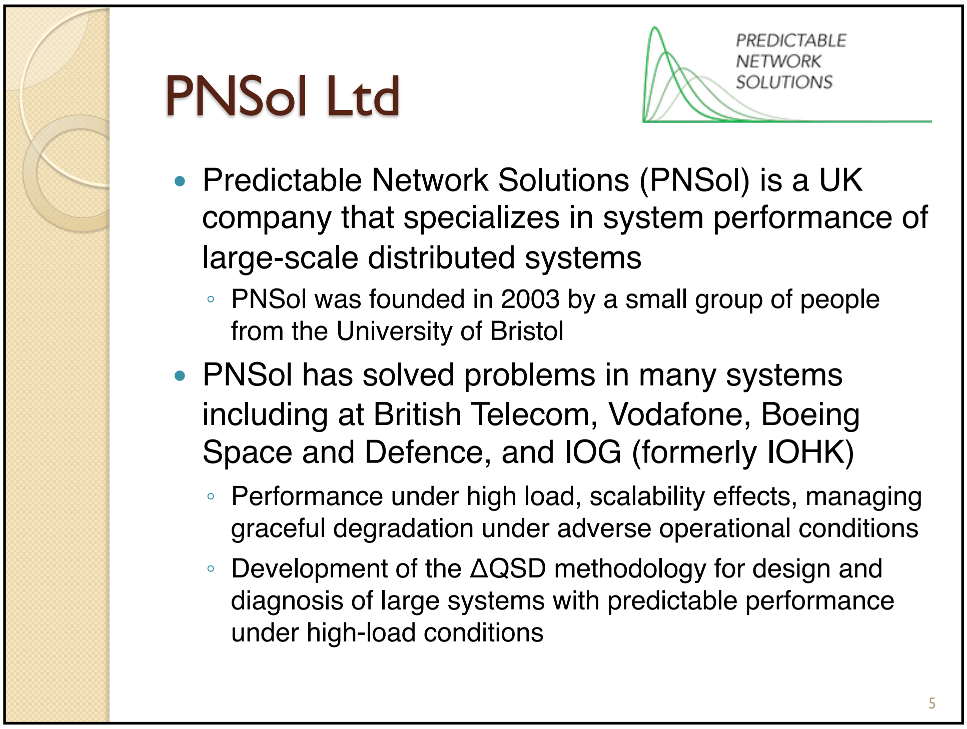
3



## Systems with many users

- ΔQSD targets systems with many independent users where real-time performance is important
  - Systems with large flows of independent data items
  - Systems that are subject to 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
  - Distributed sensor networks with real-time data streams and analysis

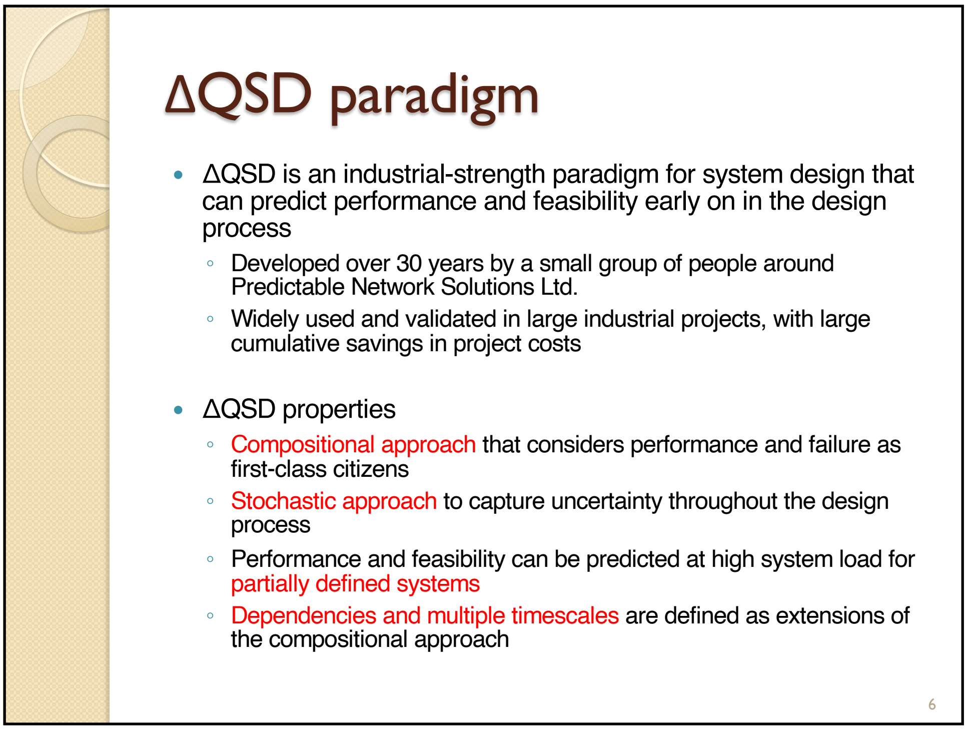
4



# PNSol Ltd

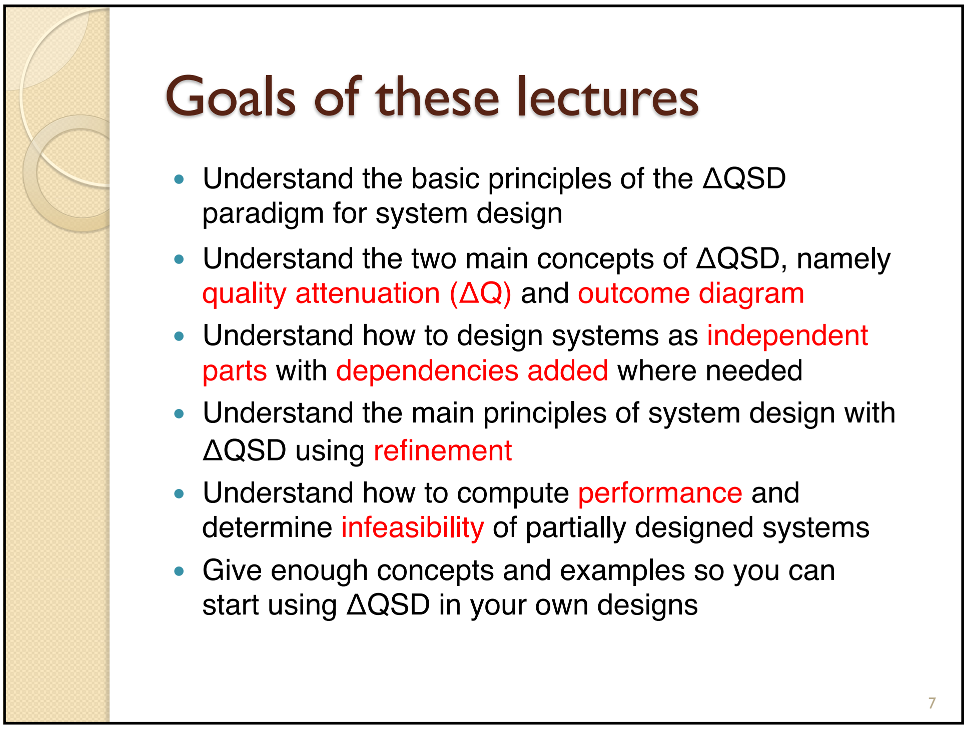
PREDICTABLE
NETWORK
SOLUTIONS

- Predictable Network Solutions (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 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



# ΔQSD paradigm

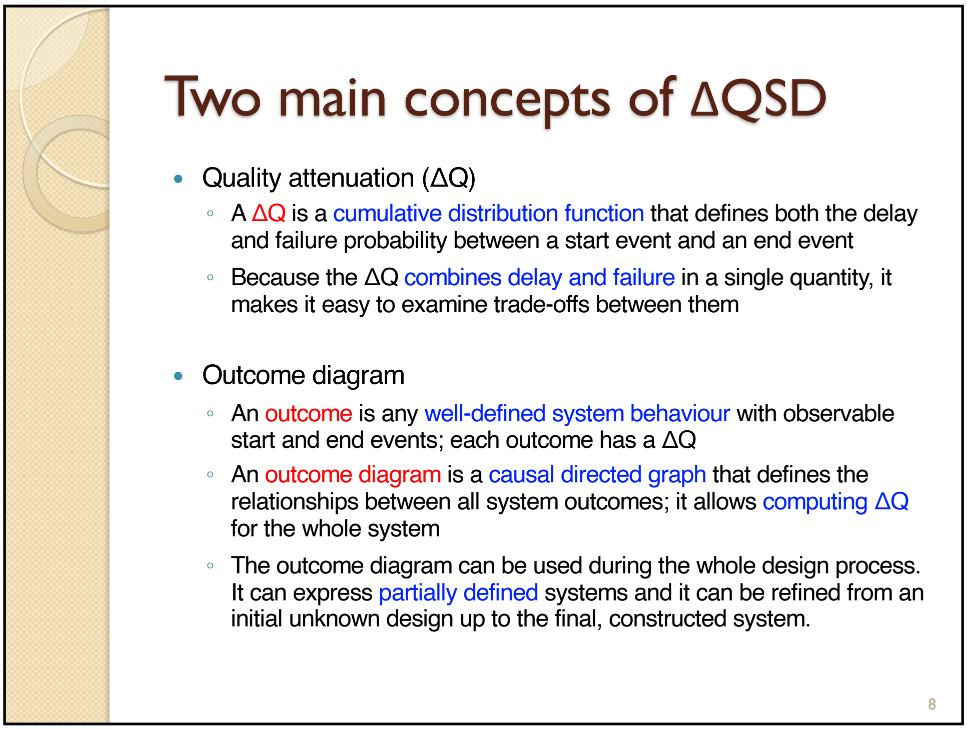
- ΔQSD is an industrial-strength paradigm for system design that can predict performance and feasibility early on in the design process
  - Developed over 30 years by a small group of people around Predictable Network Solutions Ltd.
  - Widely used and validated in large industrial projects, with large cumulative savings in project costs
- ΔQSD properties
  - Compositional approach that considers performance and failure as first-class citizens
  - Stochastic approach to capture uncertainty throughout the design process
  - Performance and feasibility can be predicted at high system load for partially defined systems
  - Dependencies and multiple timescales are defined as extensions of the compositional approach



## Goals of these lectures

- Understand the basic principles of the ΔQSD paradigm for system design
- Understand the two main concepts of ΔQSD, namely quality attenuation (ΔQ) and outcome diagram
- Understand how to design systems as independent parts with dependencies added where needed
- Understand the main principles of system design with ΔQSD using refinement
- Understand how to compute performance and determine infeasibility of partially designed systems
- Give enough concepts and examples so you can start using ΔQSD in your own designs

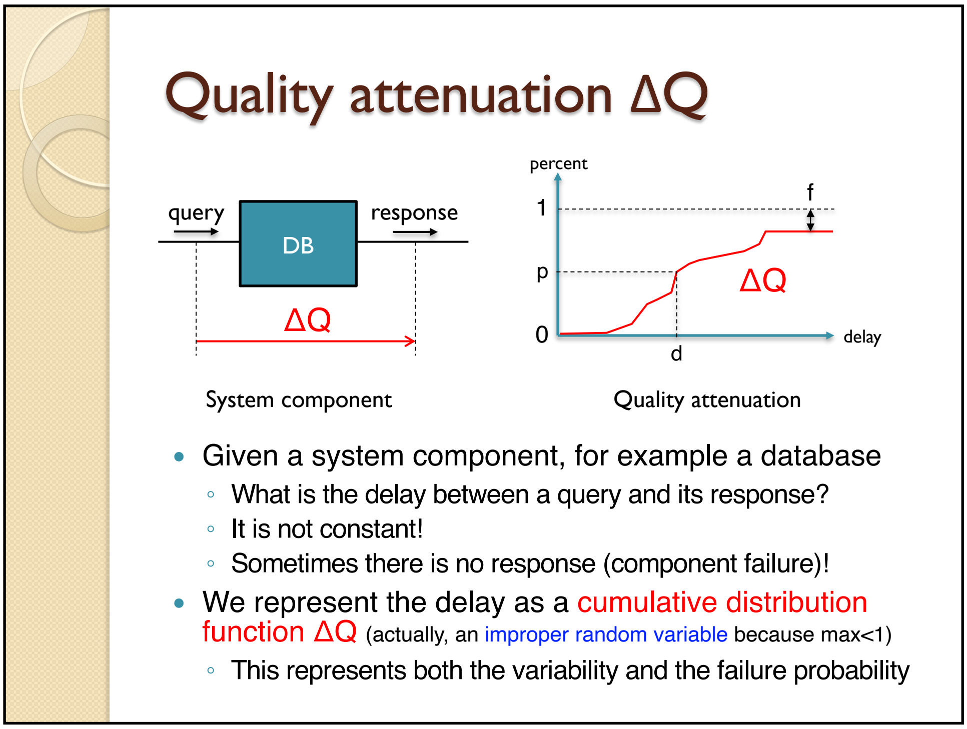
7



## Two main concepts of ΔQSD

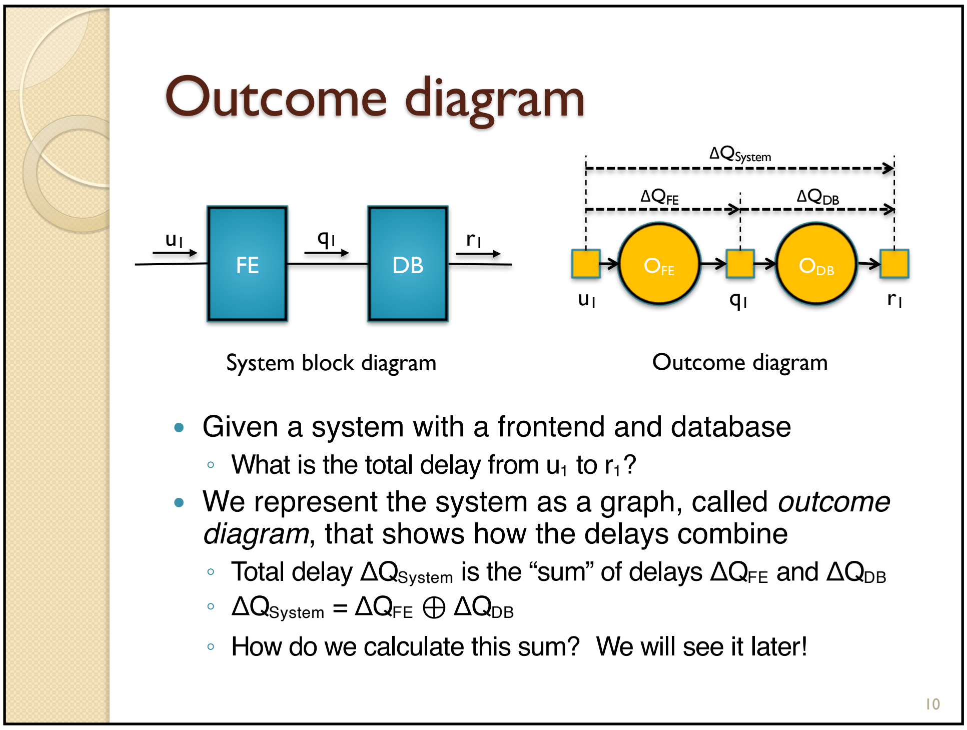
- Quality attenuation (ΔQ)
  - A ΔQ is a cumulative distribution function that defines both the delay and failure probability between a start event and an end event
  - Because the ΔQ combines delay and failure in a single quantity, it makes it easy to examine trade-offs between them
- Outcome diagram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

8



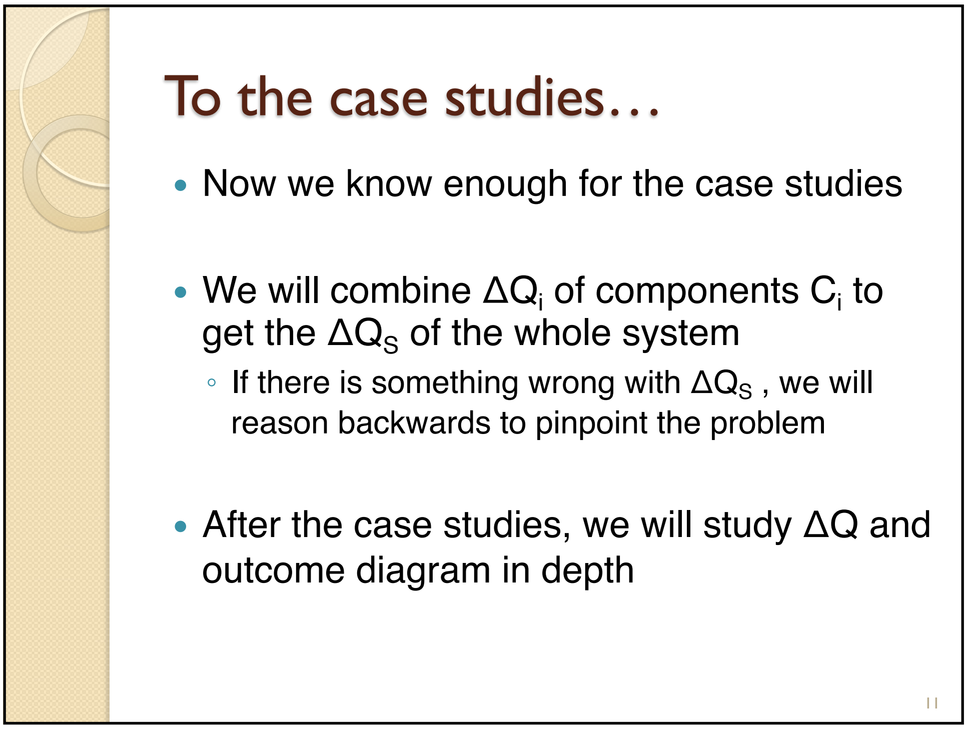
# Quality attenuation ΔQ

- 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 represent the delay as a cumulative distribution function ΔQ (actually, an improper random variable because max<1)
  - This represents both the variability and the failure probability



# Outcome diagram

- Given a system with a frontend and database
  - What is the total delay from $u_1$ to $r_1$?
- We represent the system as a graph, called *outcome diagram*, 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!

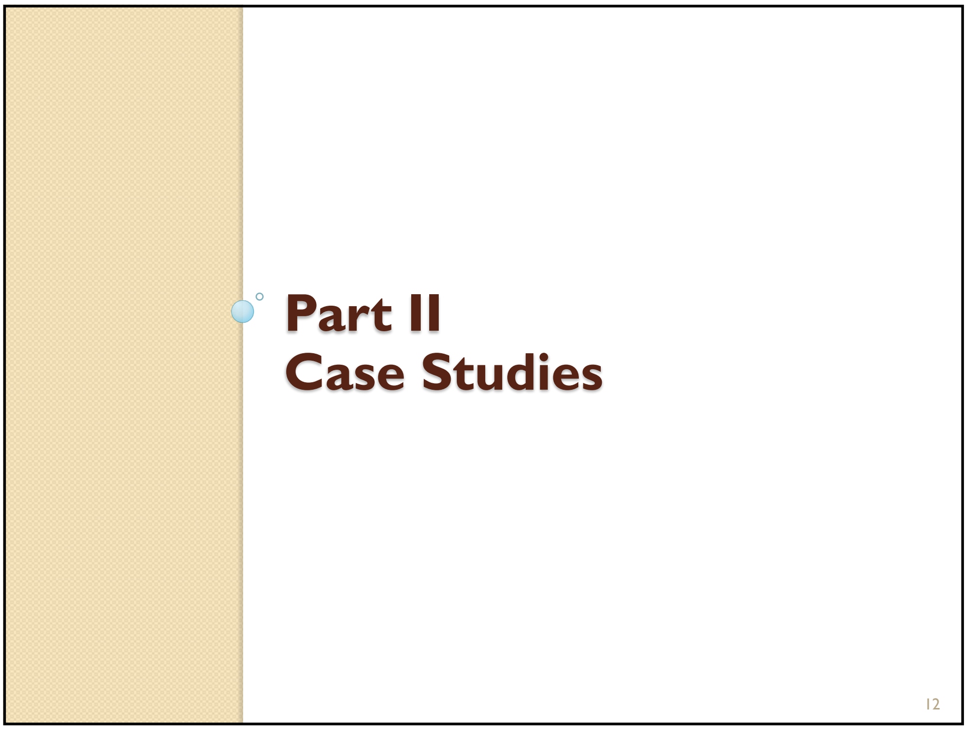


## To the case studies…

- Now we know enough for the case studies
- We will combine $\Delta Q_i$ of components $C_i$ to get the $\Delta Q_S$ of the whole system
  - If there is something wrong with $\Delta Q_S$ , we will reason backwards to pinpoint the problem
- After the case studies, we will study ΔQ and outcome diagram in depth

11

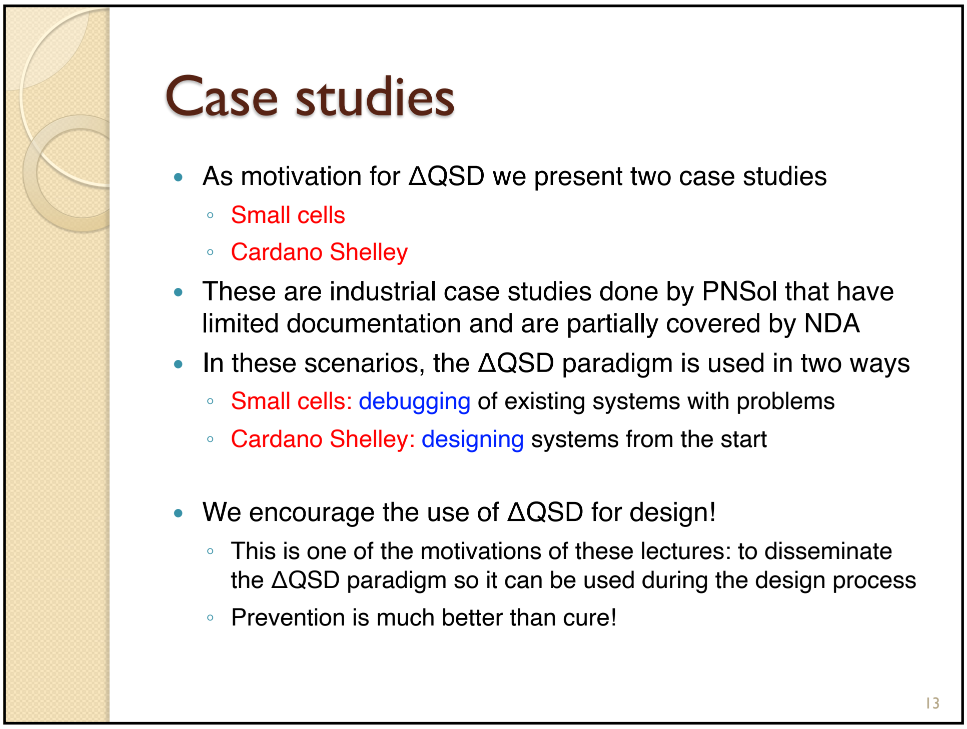
11



## Part II
## Case Studies

12

12



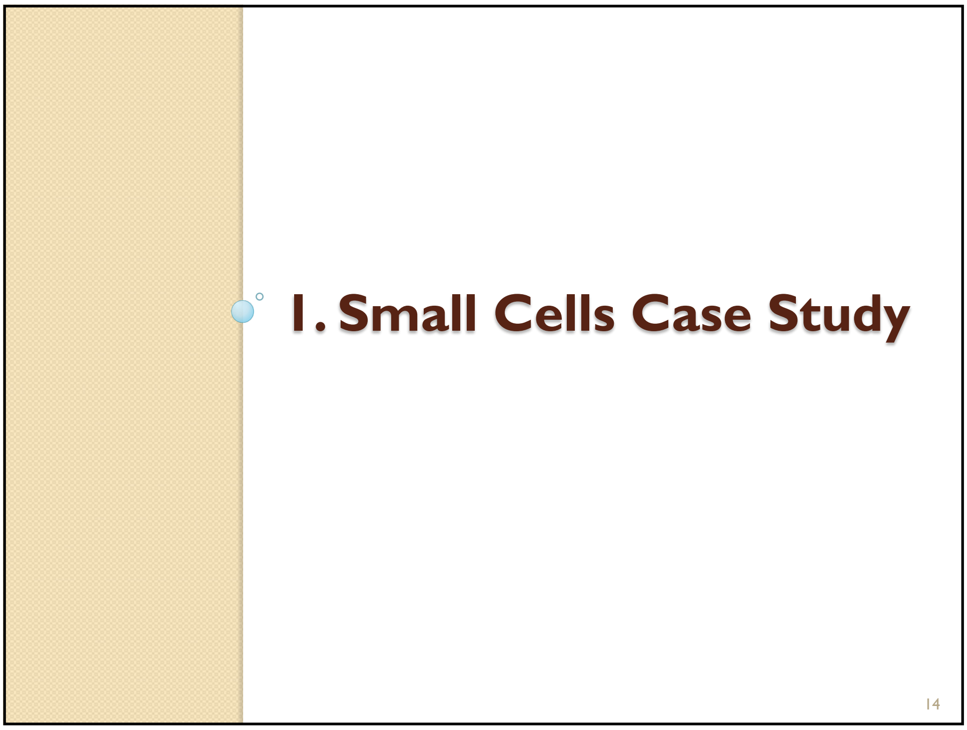
# Case studies

- As motivation for ΔQSD we present two case studies
  - Small cells
  - Cardano Shelley
- These are industrial case studies done by PNSol that have limited documentation and are partially covered by NDA
- In these scenarios, the ΔQSD paradigm is used in two ways
  - Small cells: debugging of existing systems with problems
  - Cardano Shelley: designing systems from the start

- We encourage the use of ΔQSD for design!
  - This is one of the motivations of these lectures: to disseminate the ΔQSD paradigm so it can be used during the design process
  - Prevention is much better than cure!

13

13



# 1. Small Cells Case Study

14

14

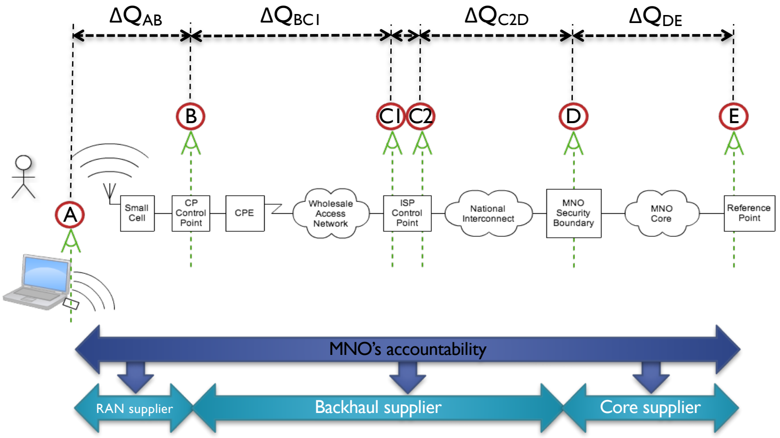
# Small cells case study

- 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

15

15

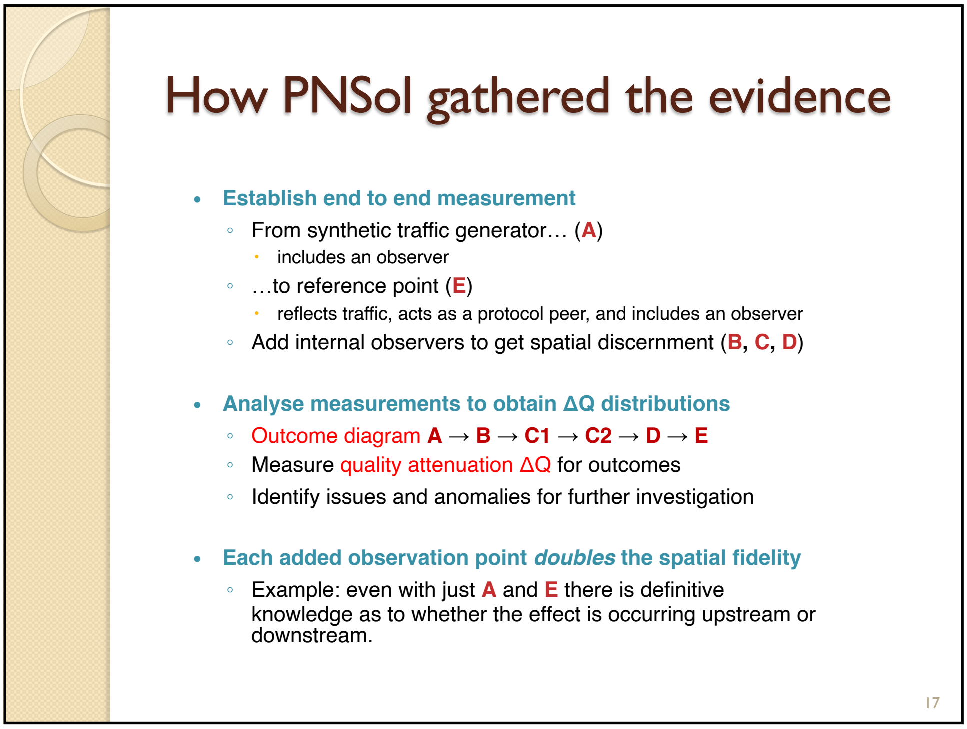
# Who is to blame for my system crashing?



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

16

16



# How PNSol gathered the evidence

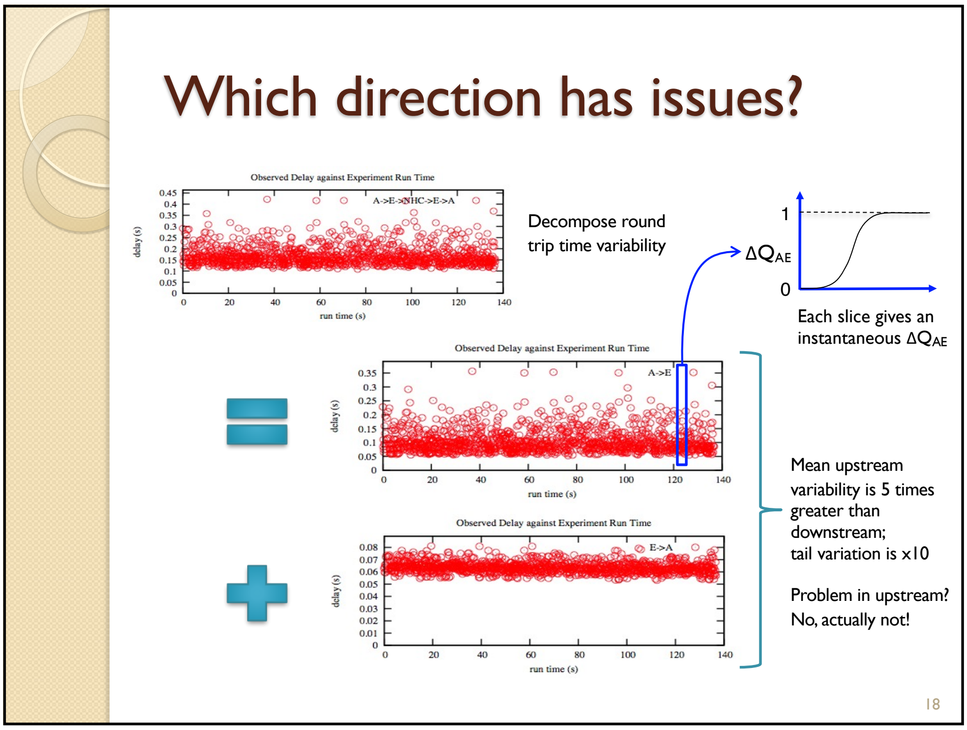
- Establish end to end measurement
  - From synthetic traffic generator… (**A**)
    - includes an observer
  - …to reference point (**E**)
    - reflects traffic, acts as a protocol peer, and includes an observer
  - Add internal observers to get spatial discernment (**B**, **C**, **D**)

- Analyse measurements to obtain ΔQ distributions
  - Outcome diagram **A** → **B** → **C1** → **C2** → **D** → **E**
  - Measure quality attenuation ΔQ for outcomes
  - Identify issues and anomalies for further investigation

- Each added observation point *doubles* the spatial fidelity
  - Example: even with just **A** and **E** there is definitive knowledge as to whether the effect is occurring upstream or downstream.

17

17



# Which direction has issues?

Decompose round trip time variability

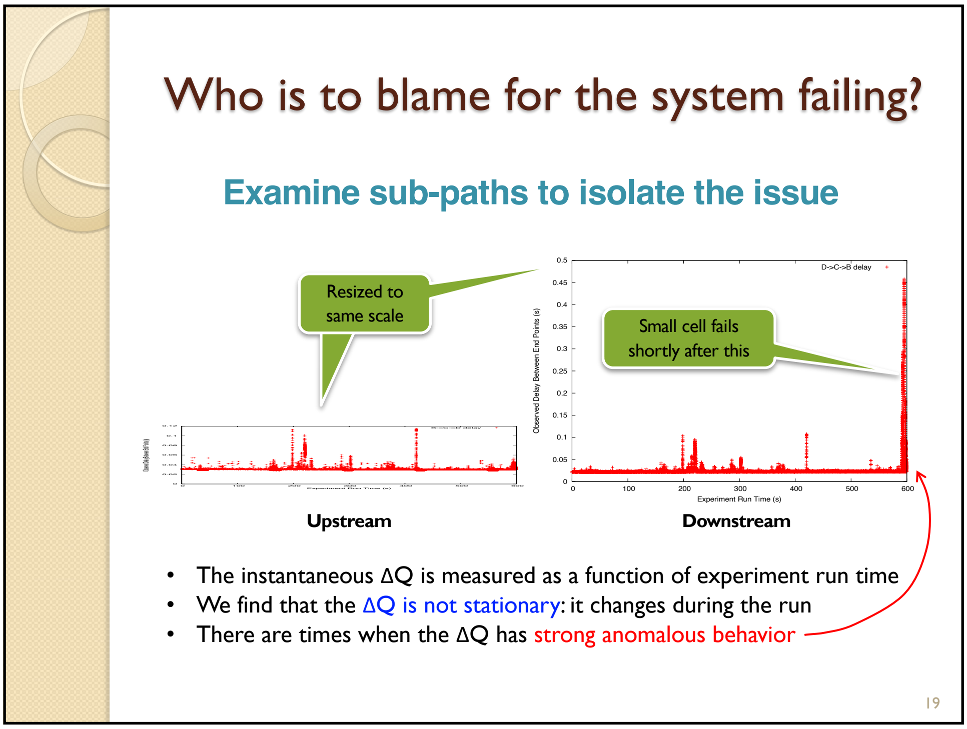
Each slice gives an instantaneous $\Delta Q_{AE}$

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

Problem in upstream?
No, actually not!

18

18



## 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

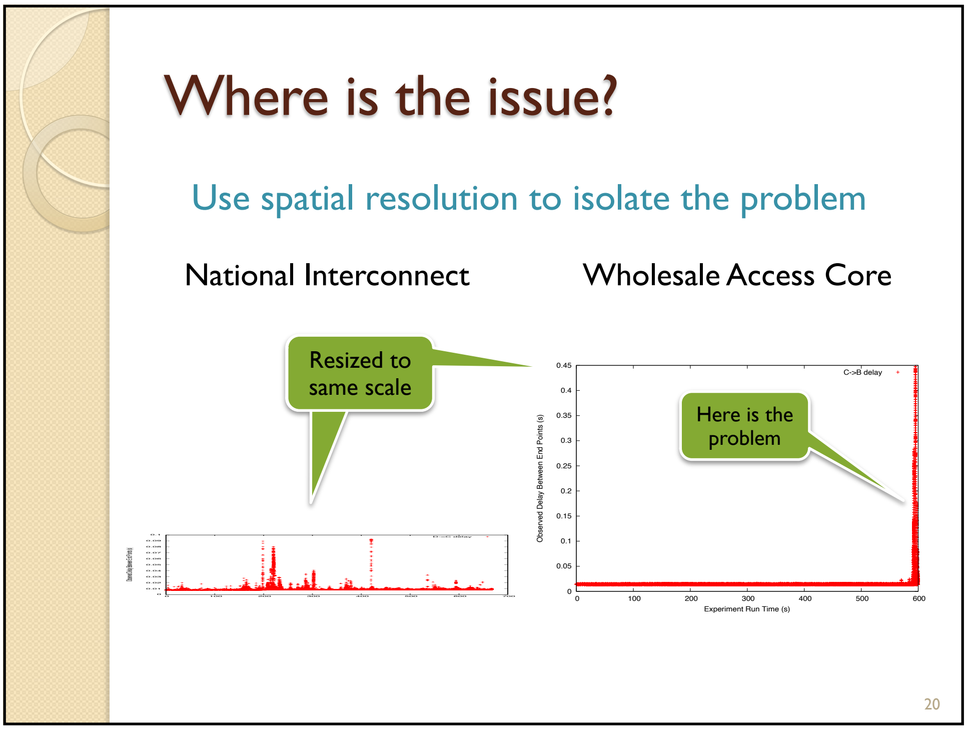
**Upstream**

**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

19

19



## Where is the issue?

### Use spatial resolution to isolate the problem

National Interconnect

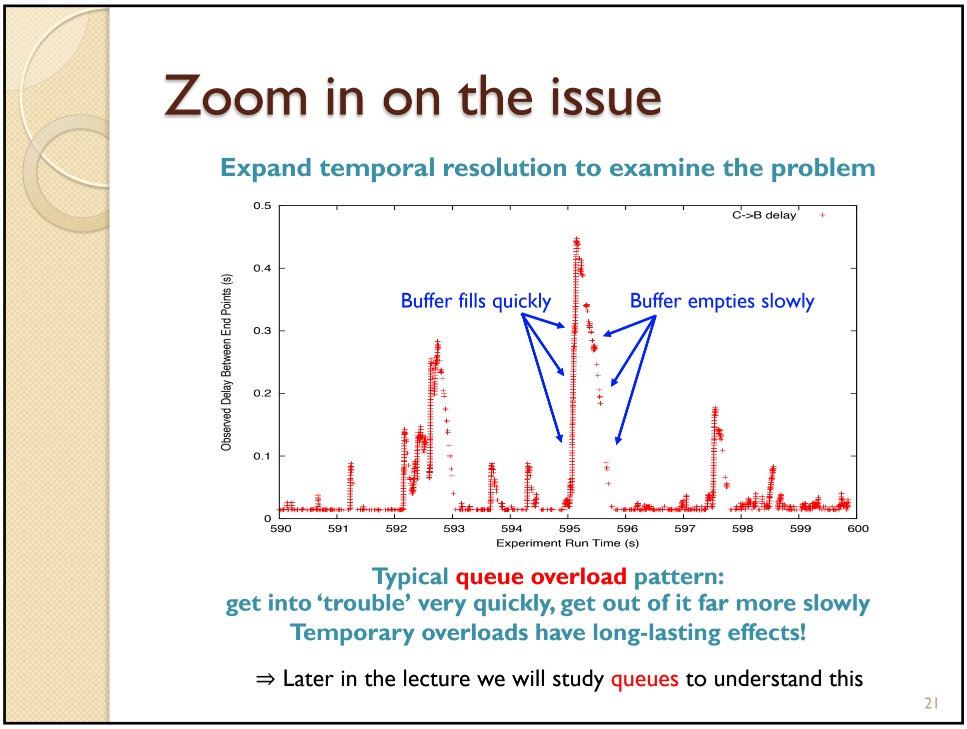
Wholesale Access Core

Resized to same scale

Here is the problem

20

20



# Zoom in on the issue

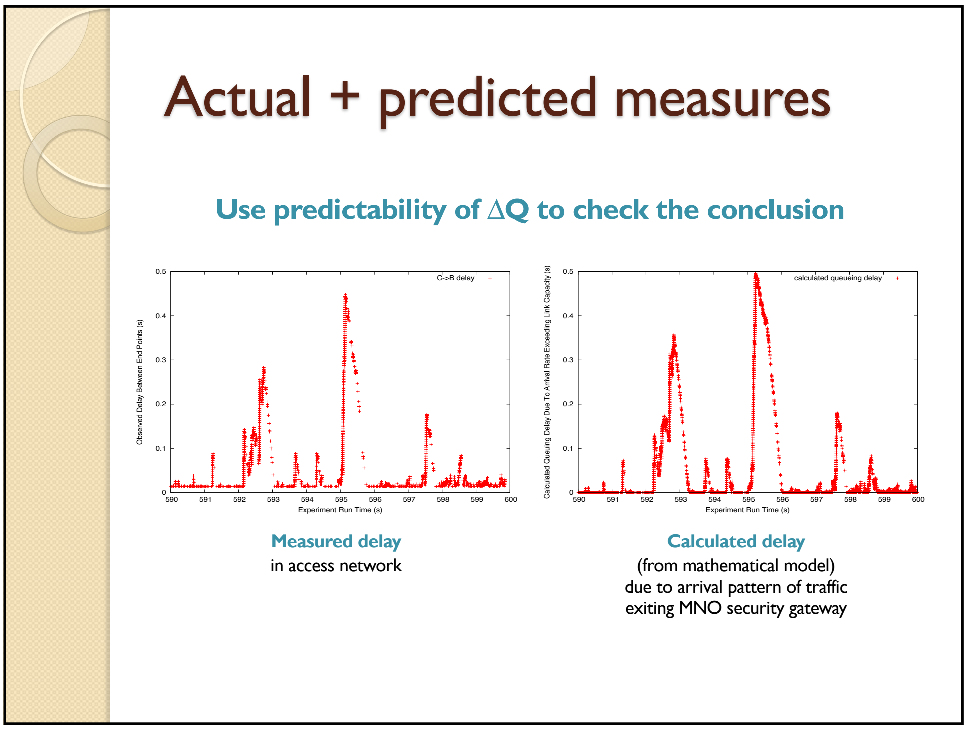
Expand temporal resolution to examine the problem

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

21

21



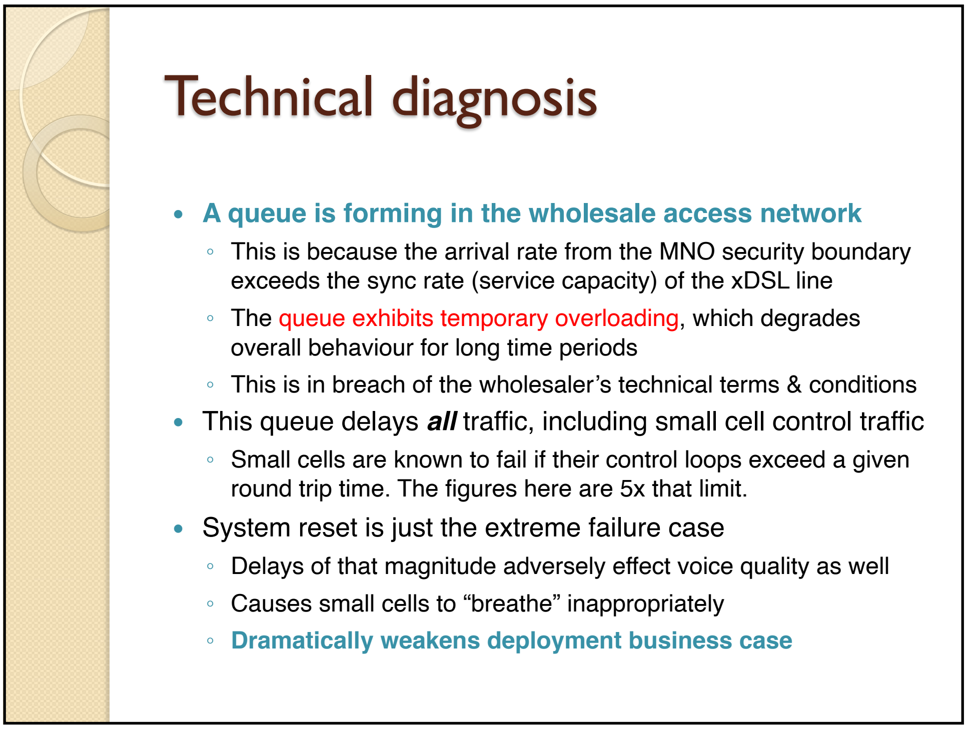
# Actual + predicted measures

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

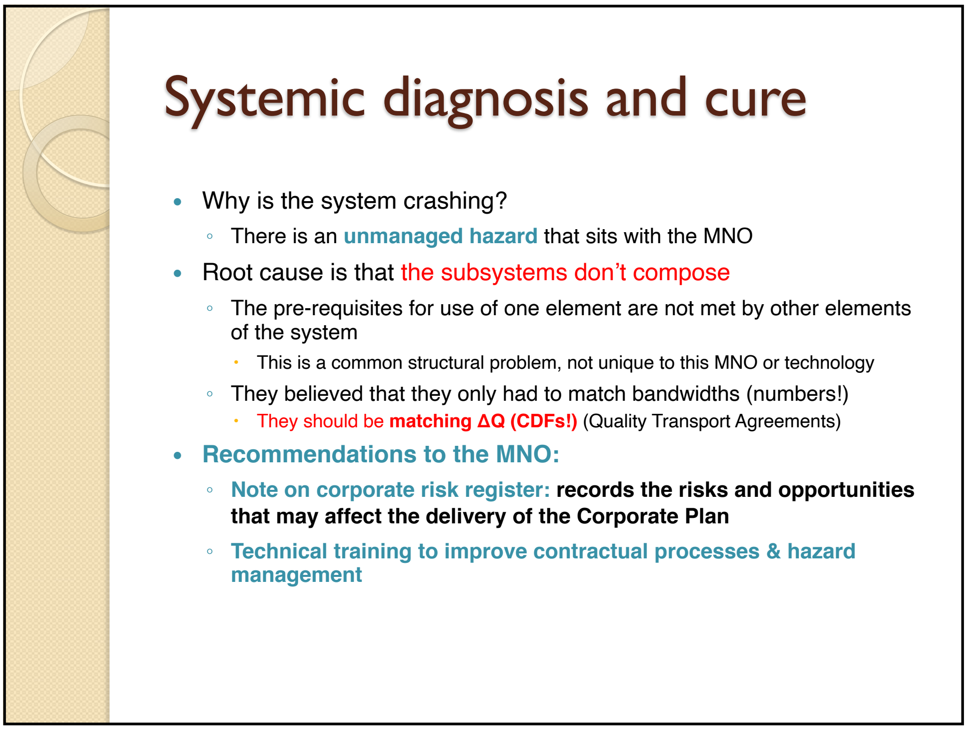
22



# 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**

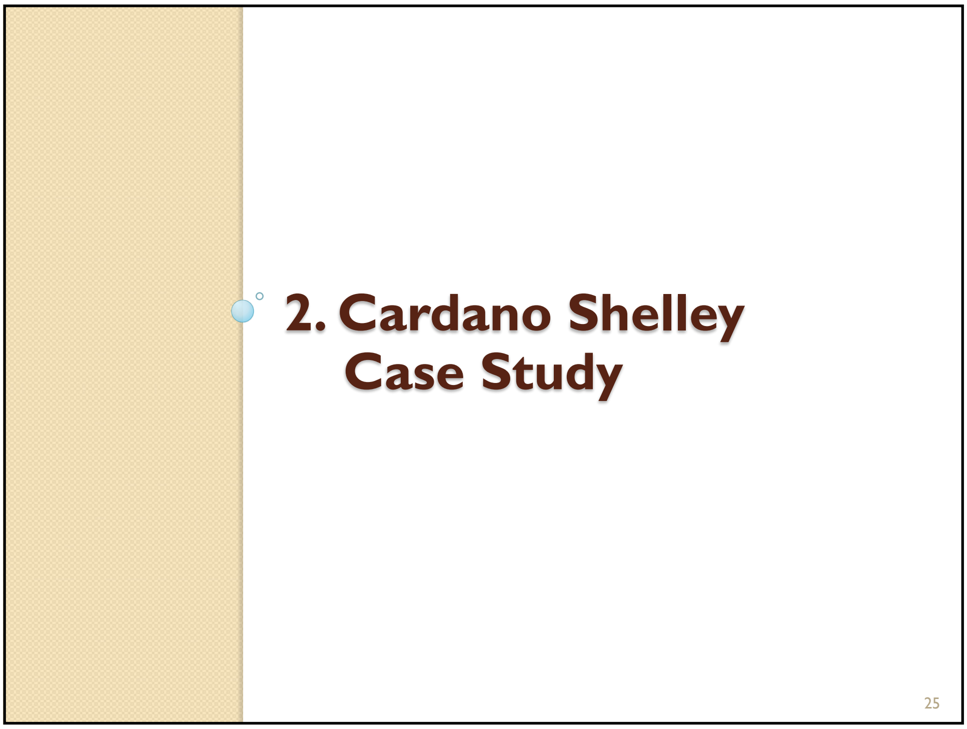
23



# Systemic diagnosis and cure

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

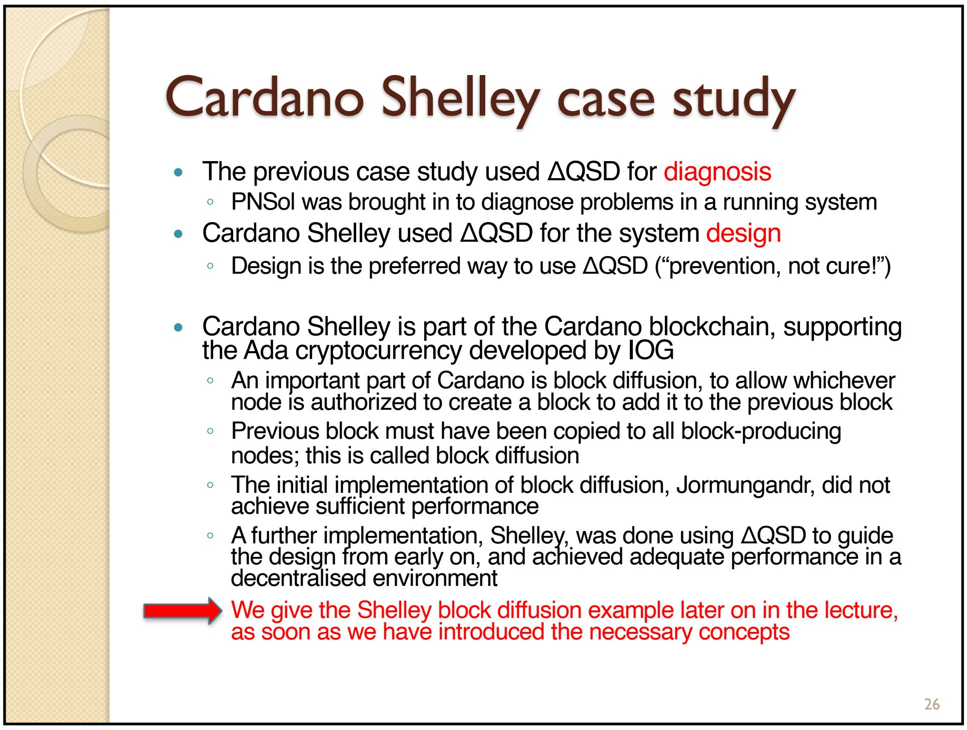
24



# 2. Cardano Shelley Case Study

25

25



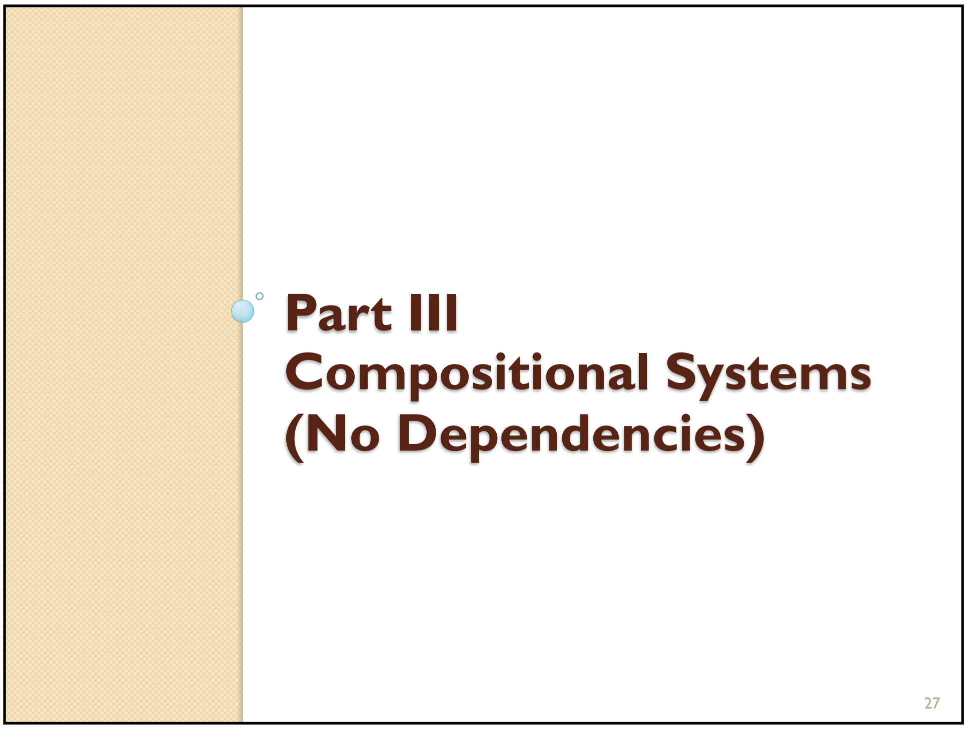
## Cardano Shelley case study

- The previous case study used ΔQSD for diagnosis
  - PNSol was brought in to diagnose problems in a running system
- Cardano Shelley used 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 developed by IOG
  - An important part of Cardano is block diffusion, to allow whichever node is authorized to create a block to add it to the previous block
  - Previous block must have been copied to all block-producing nodes; this is called block diffusion
  - The initial implementation of block diffusion, Jormungandr, did not achieve sufficient performance
  - A further implementation, Shelley, was done using ΔQSD to guide the design from early on, and achieved adequate performance in a decentralised environment

➡ We give the Shelley block diffusion example later on in the lecture, as soon as we have introduced the necessary concepts

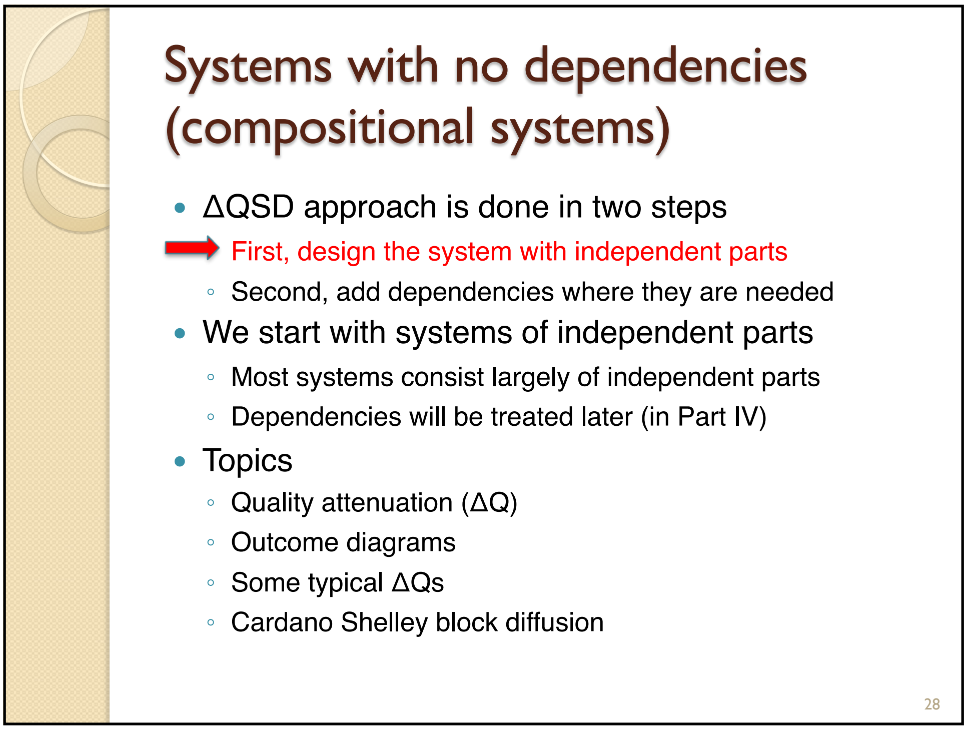
26

26



## Part III
## Compositional Systems
## (No Dependencies)

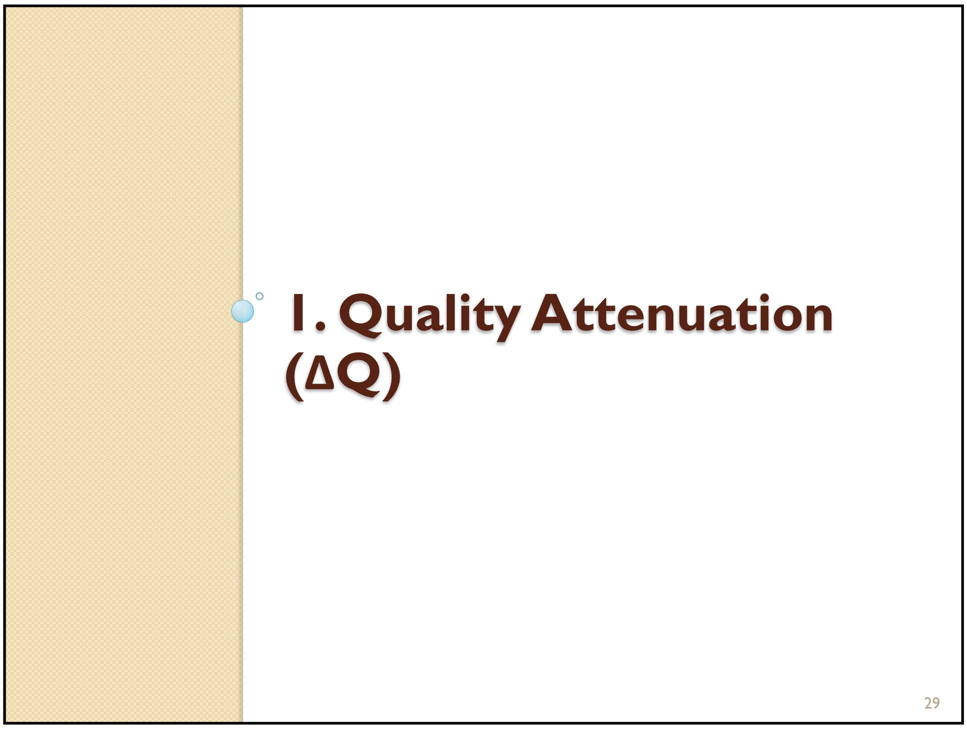
27



## Systems with no dependencies (compositional systems)

- ΔQSD approach is done in two steps
  - ➡ First, design the system with independent parts
  - Second, add dependencies where they are needed
- We start with systems of independent parts
  - Most systems consist largely of independent parts
  - Dependencies will be treated later (in Part IV)
- Topics
  - Quality attenuation (ΔQ)
  - Outcome diagrams
  - Some typical ΔQs
  - Cardano Shelley block diffusion

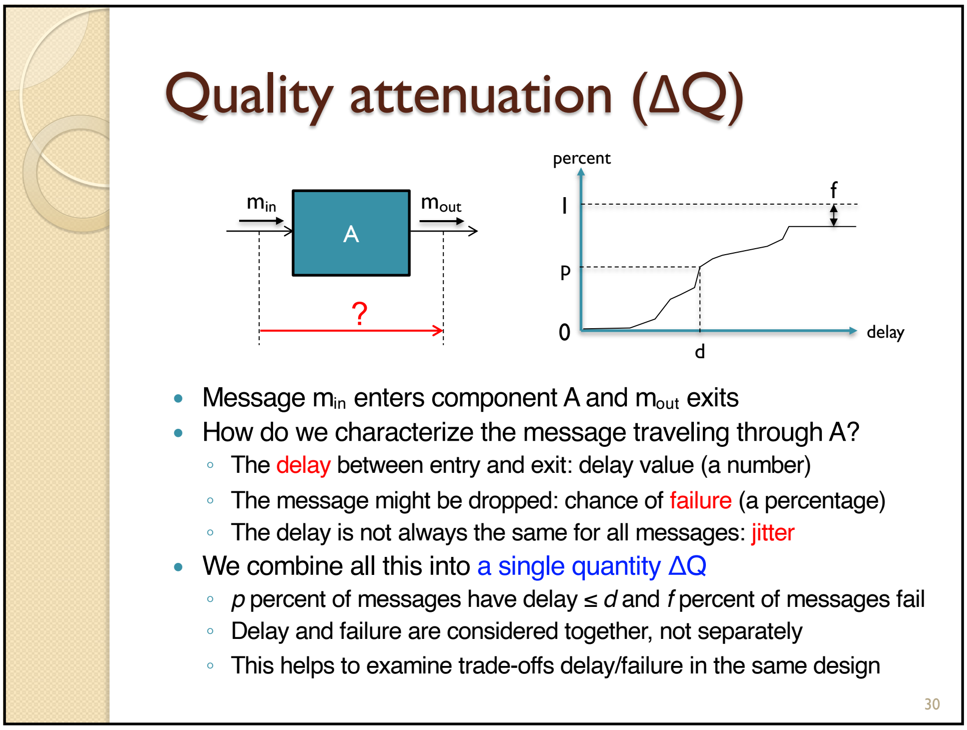
28



# I. Quality Attenuation (ΔQ)

29

29



# Quality attenuation (ΔQ)

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

30

30

# Combining delay and failure

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

31

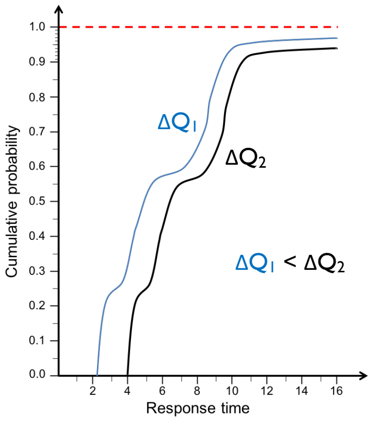
31

# Partial order of ΔQ comparison

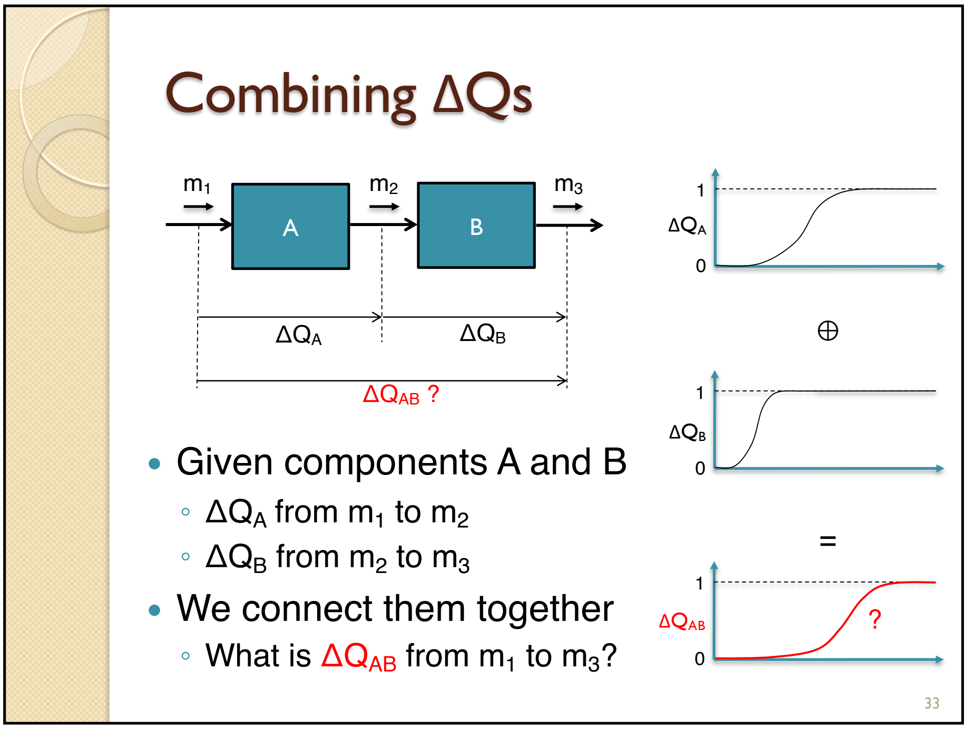
If we compare the CDFs of two ΔQs, then one is *less than* the other if its CDF is everywhere to the left and above the other

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

This provides a criterion for 'good enough' performance



32

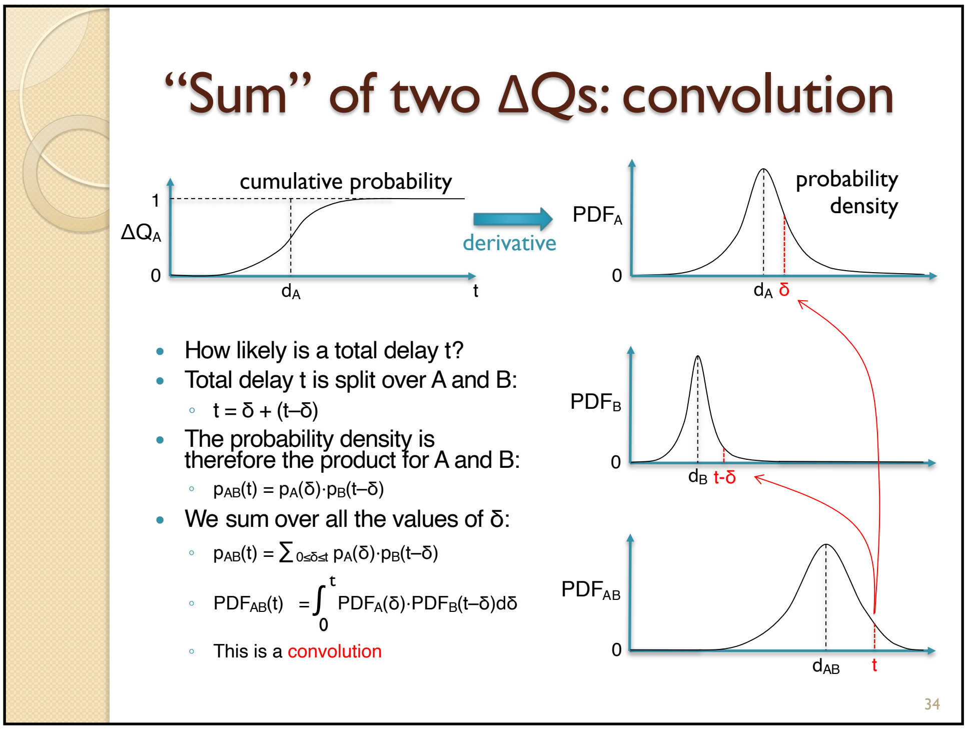


# Combining ΔQs

- 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 $m_3$?

33

33

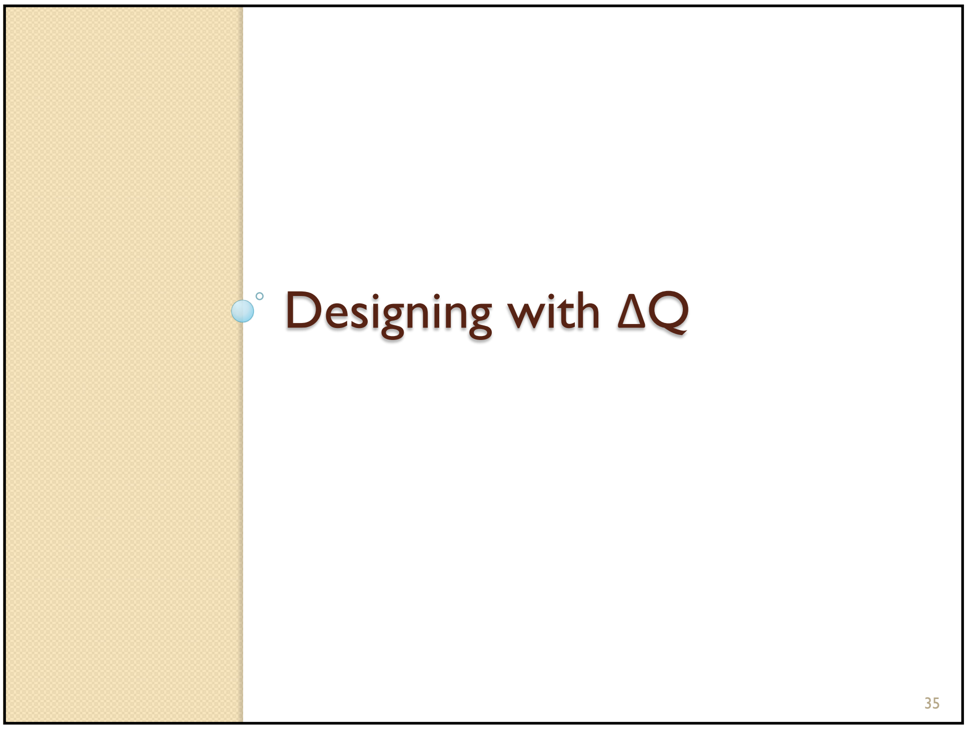


# "Sum" of two ΔQs: convolution

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

34

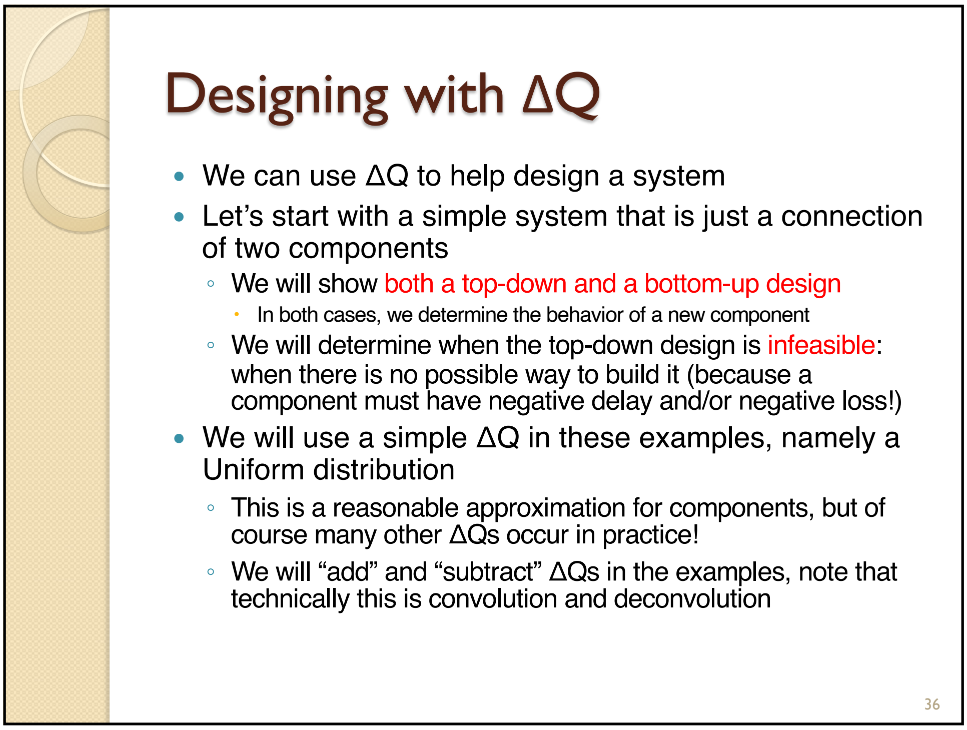
34



# Designing with ΔQ

35

35

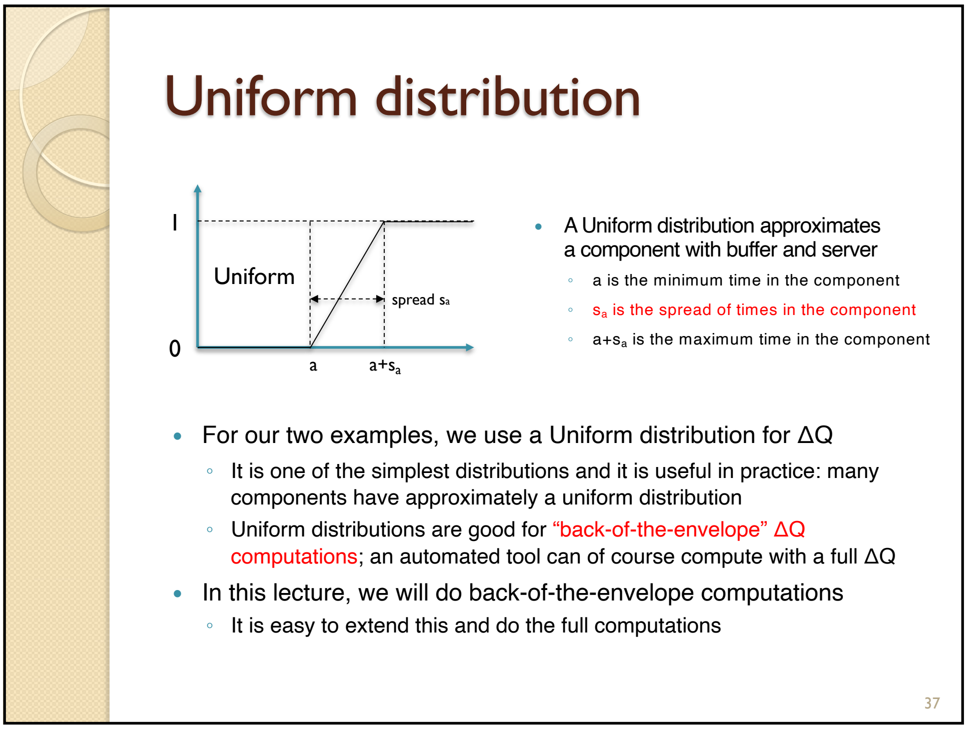


## Designing with ΔQ

- 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

36

36



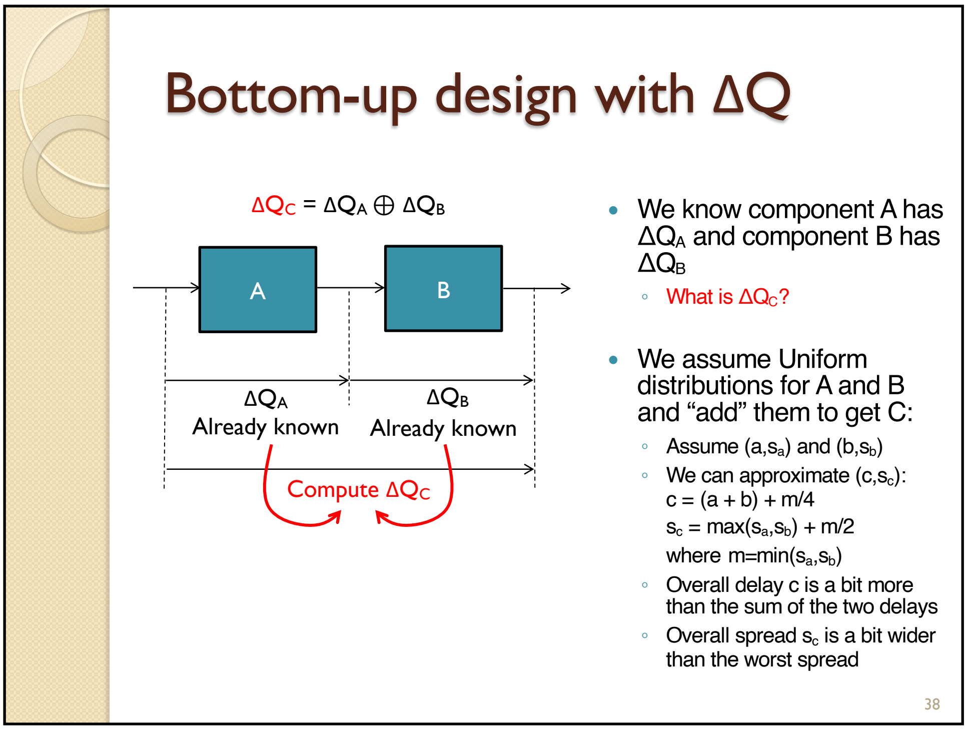
# Uniform distribution

- 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

37

37



# Bottom-up design with ΔQ

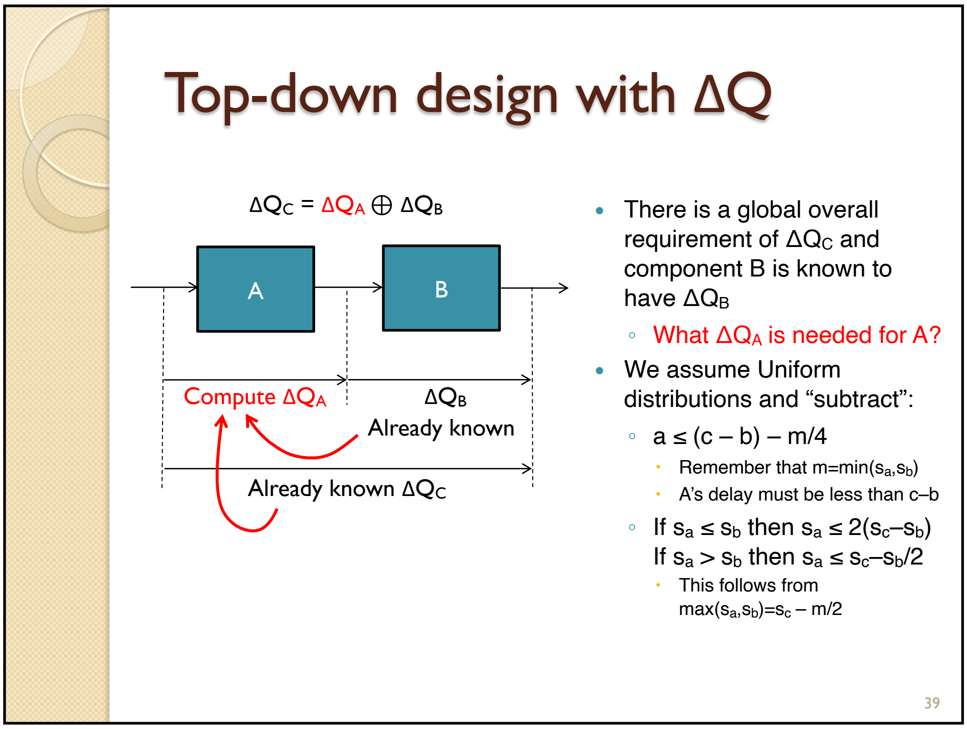
$\Delta Q_C = \Delta Q_A \oplus \Delta Q_B$

- We know 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

38

38



# Top-down design with ΔQ

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

A

B

Compute $\Delta Q_A$

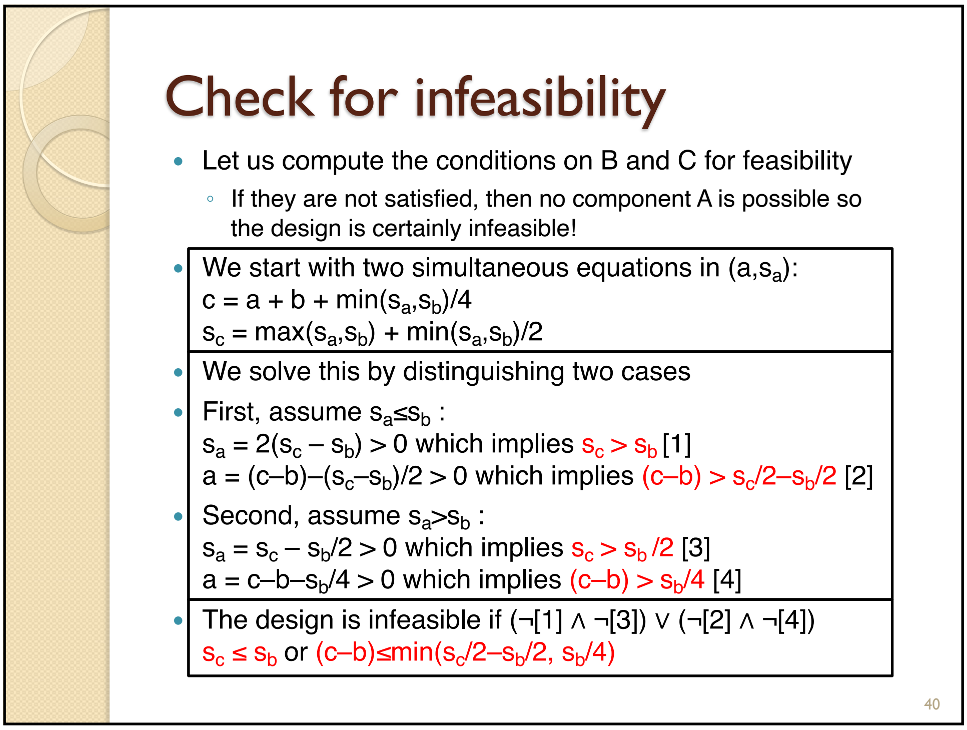
$\Delta Q_B$
Already known

Already known $\Delta Q_C$

- 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 $\max(s_a, s_b) = s_c - m/2$

39

39

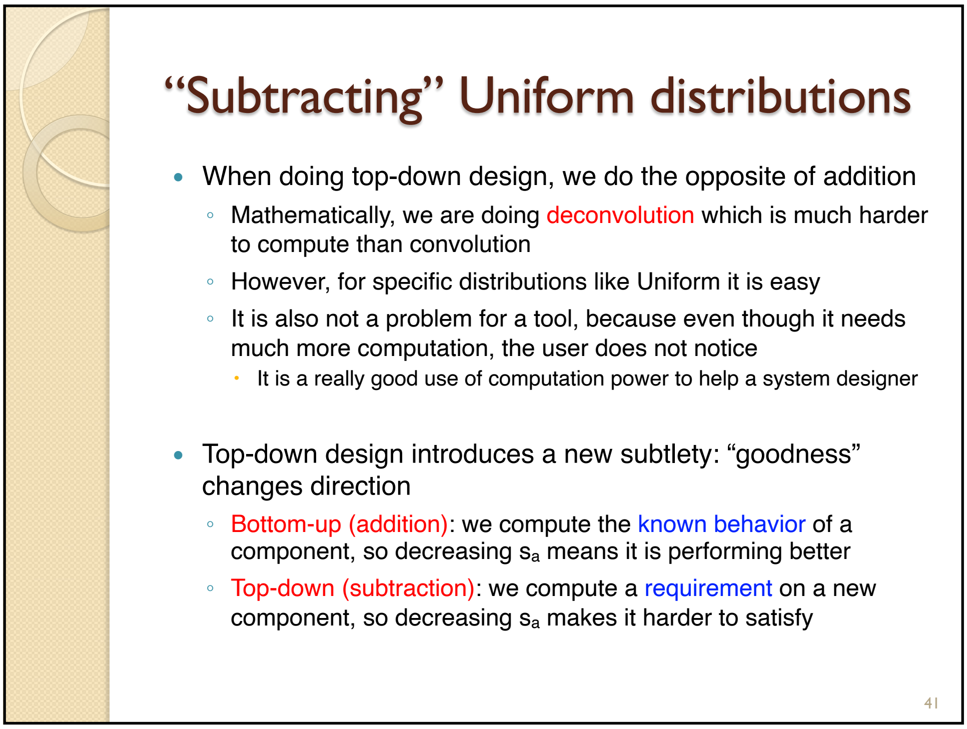


# Check for infeasibility

- 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] ∧ ¬[3]) ∨ (¬[2] ∧ ¬[4])
  $s_c \leq s_b$ or $(c-b) \leq \min(s_c/2 - s_b/2,\ s_b/4)$

40

40

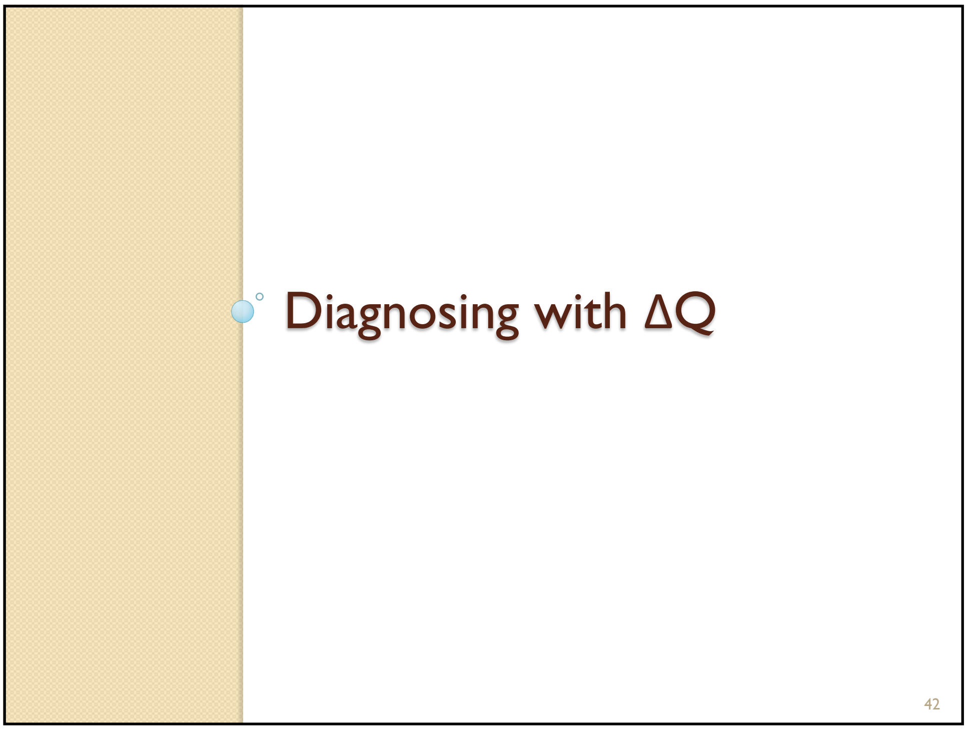


# “Subtracting” Uniform distributions

- 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

41

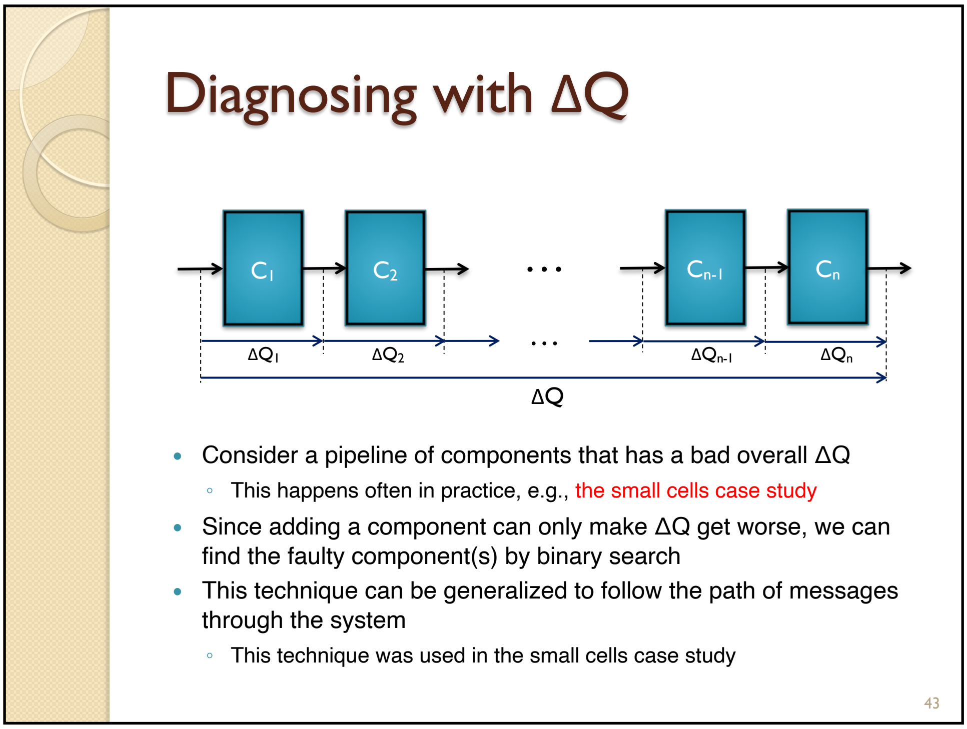
41



# Diagnosing with ΔQ

42

42

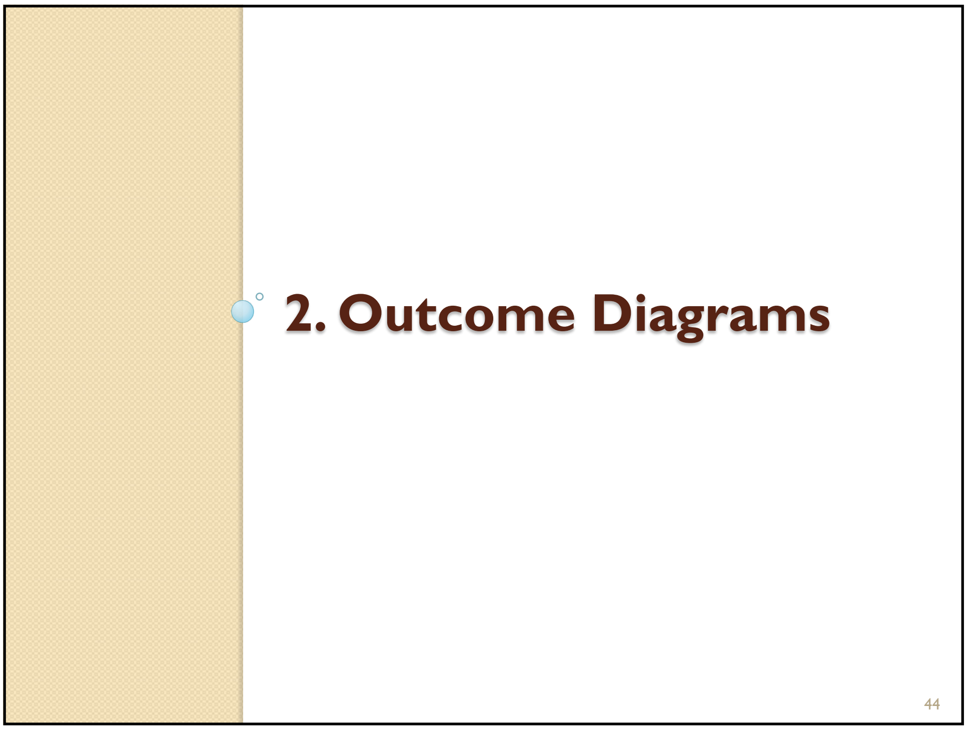


## Diagnosing with ΔQ

- 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
  - This technique was used in the small cells case study

43

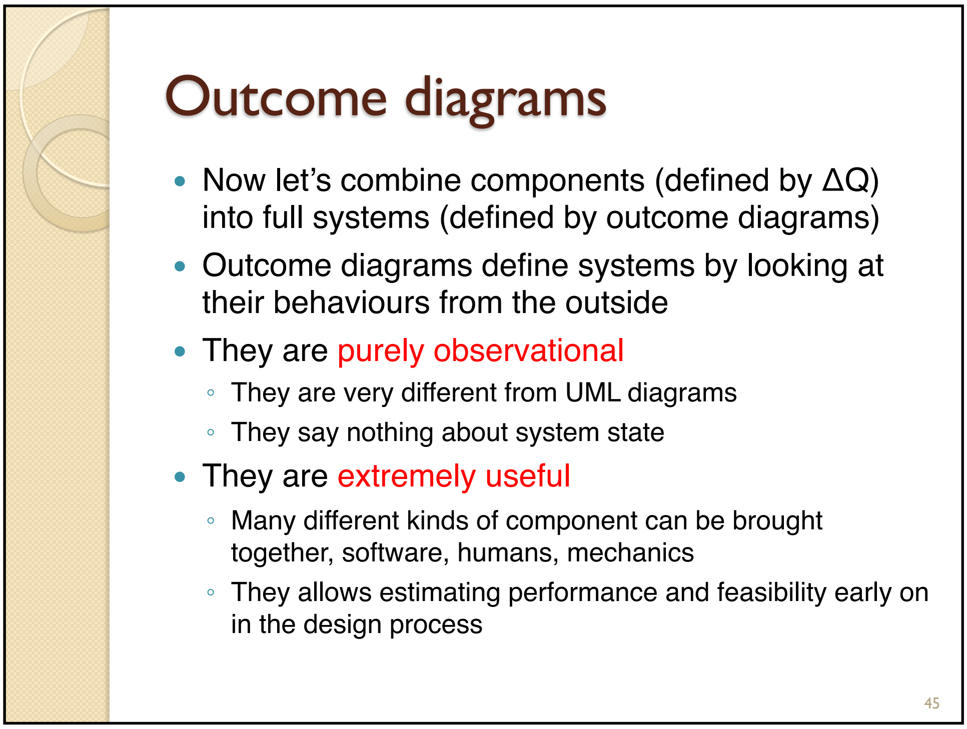
43



## 2. Outcome Diagrams

44

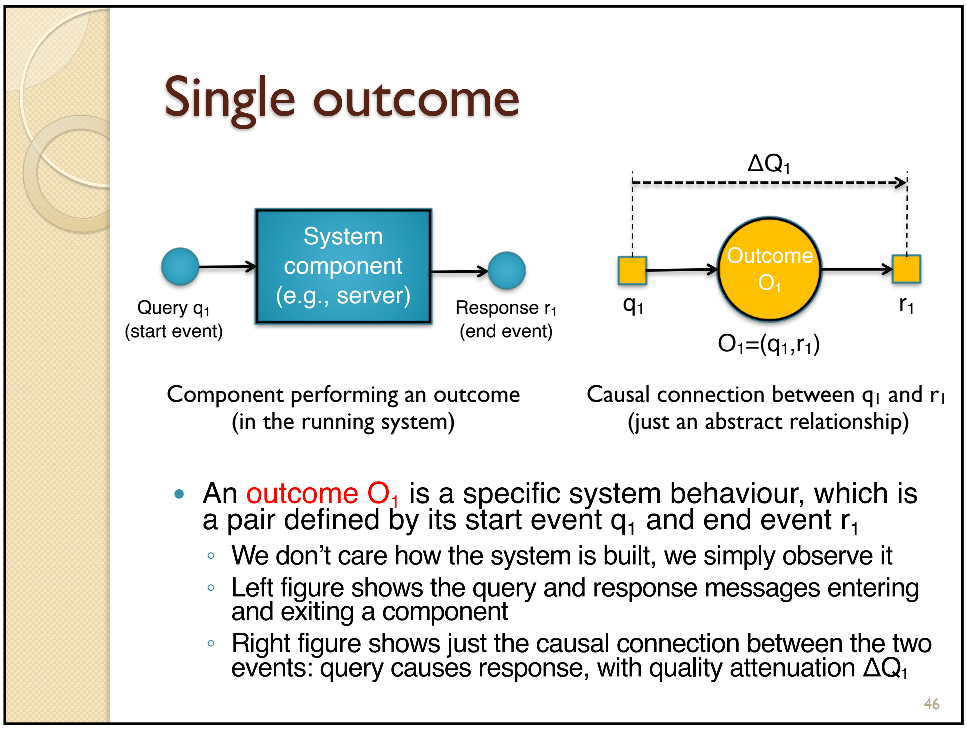
44



# 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
  - They say nothing about system state
- They are extremely useful
  - Many different kinds of component can be brought together, software, humans, mechanics
  - They allows estimating performance and feasibility early on in the design process

45



# Single outcome

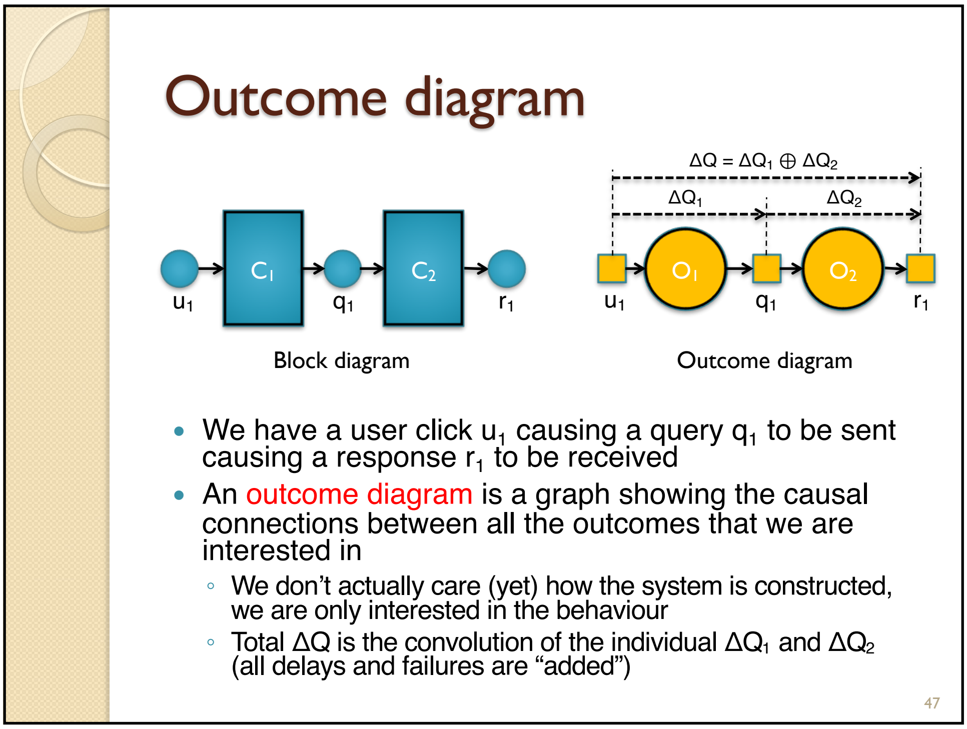
System component (e.g., server)
Query $q_1$ (start event)
Response $r_1$ (end event)
$\Delta Q_1$
Outcome $O_1$
$q_1$
$r_1$
$O_1=(q_1,r_1)$

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 $\Delta Q_1$

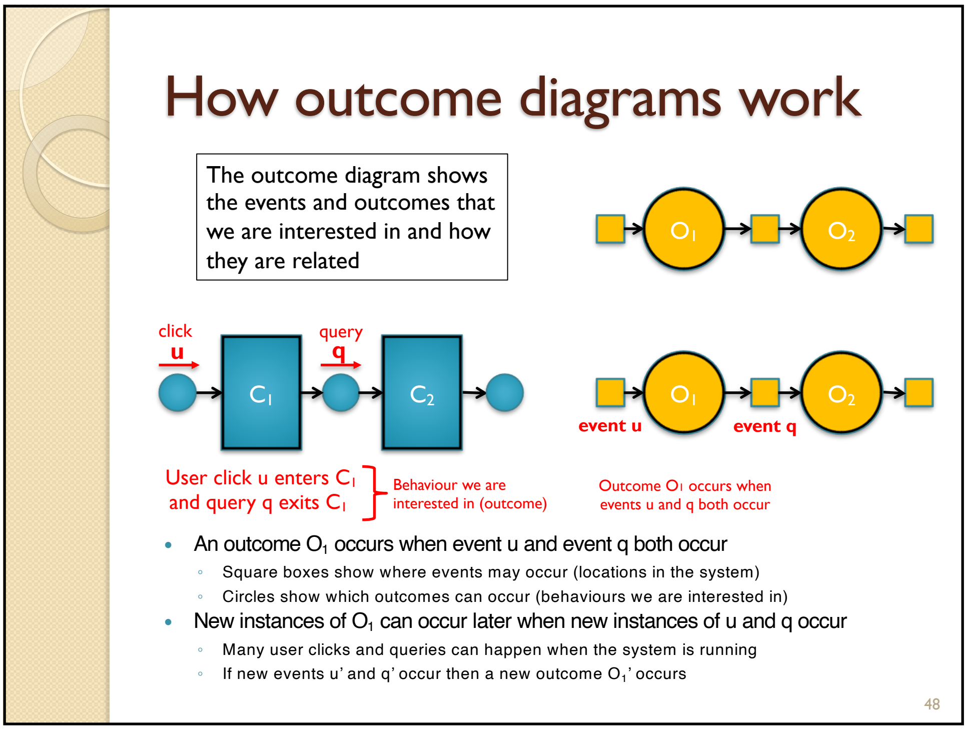
46



# 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")

47



# How outcome diagrams work

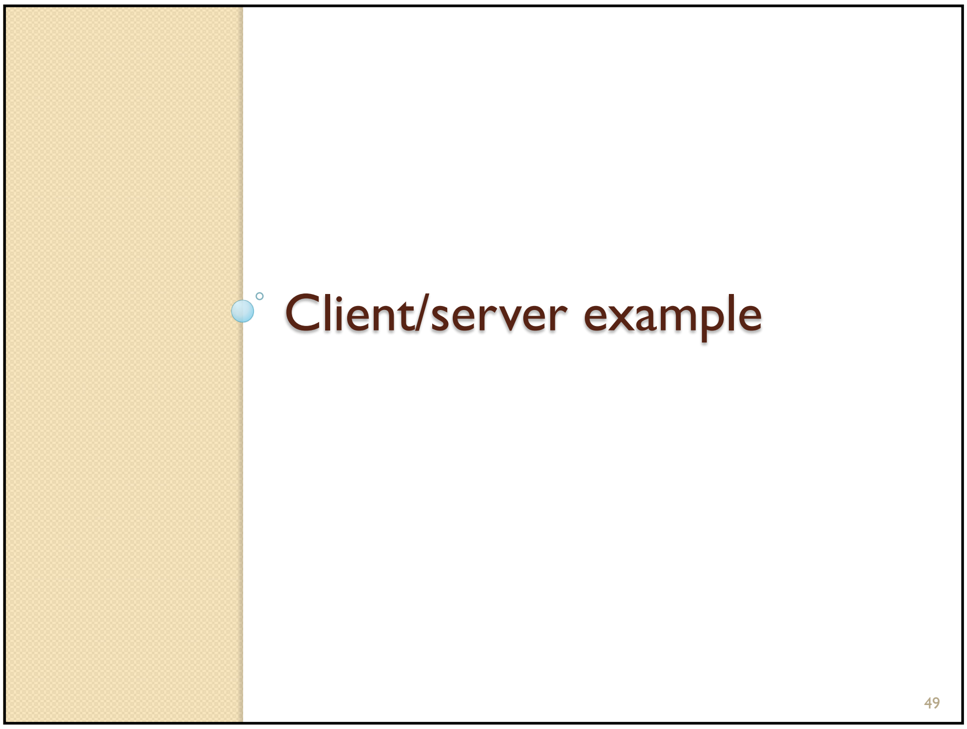
The outcome diagram shows the events and outcomes that we are interested in and how they are related

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

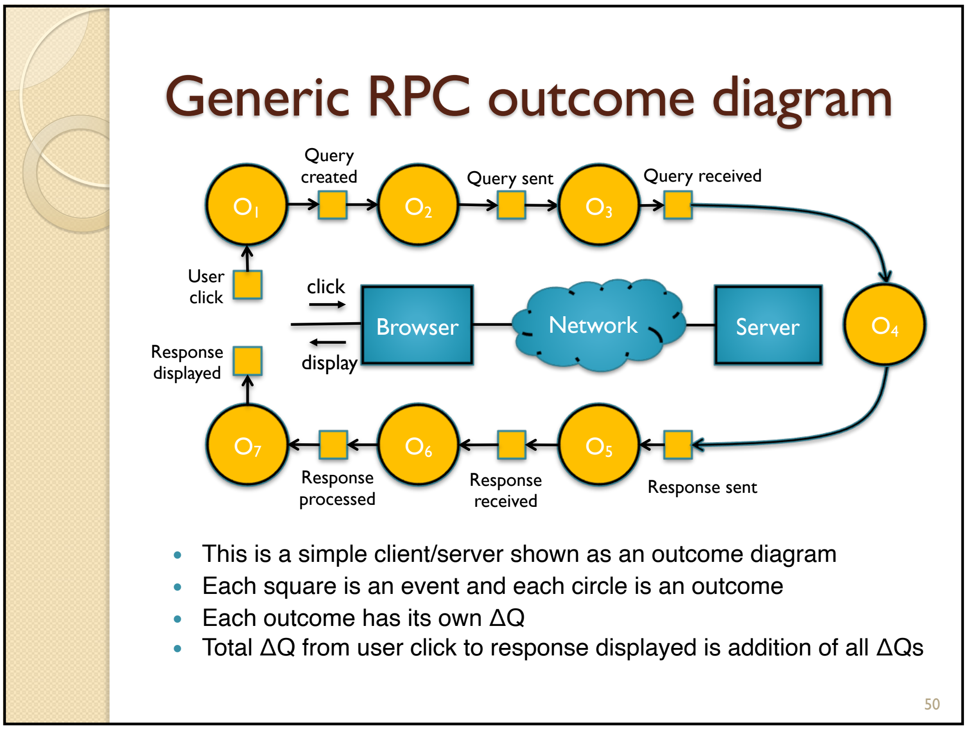
Outcome $O_1$ occurs when events u and q both occur

- 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

48

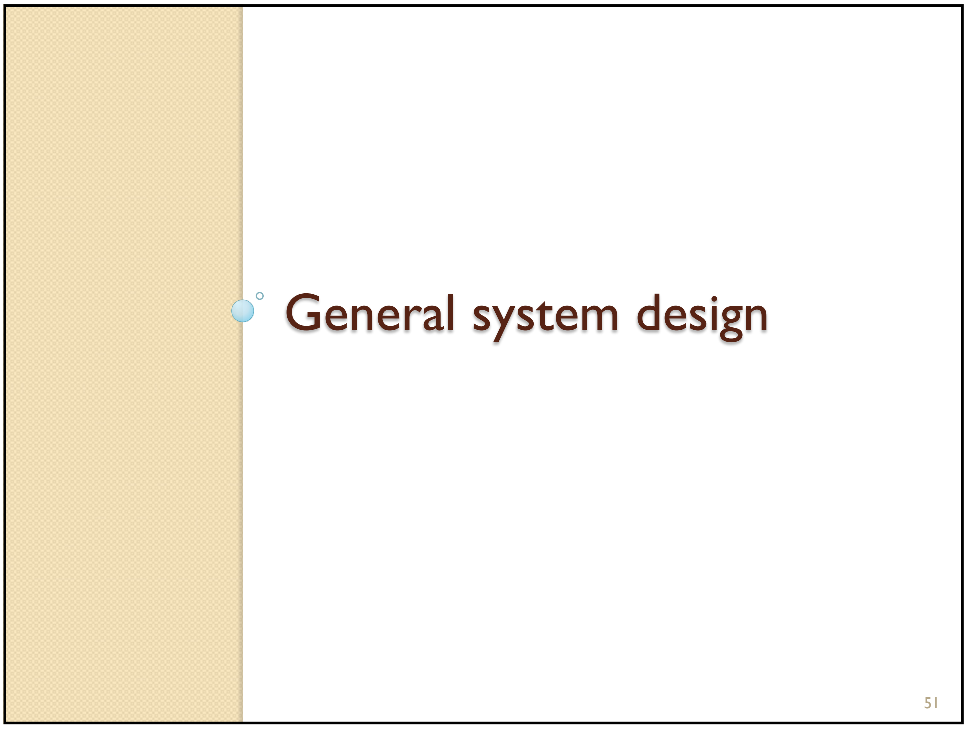


# Client/server example



# Generic RPC outcome diagram

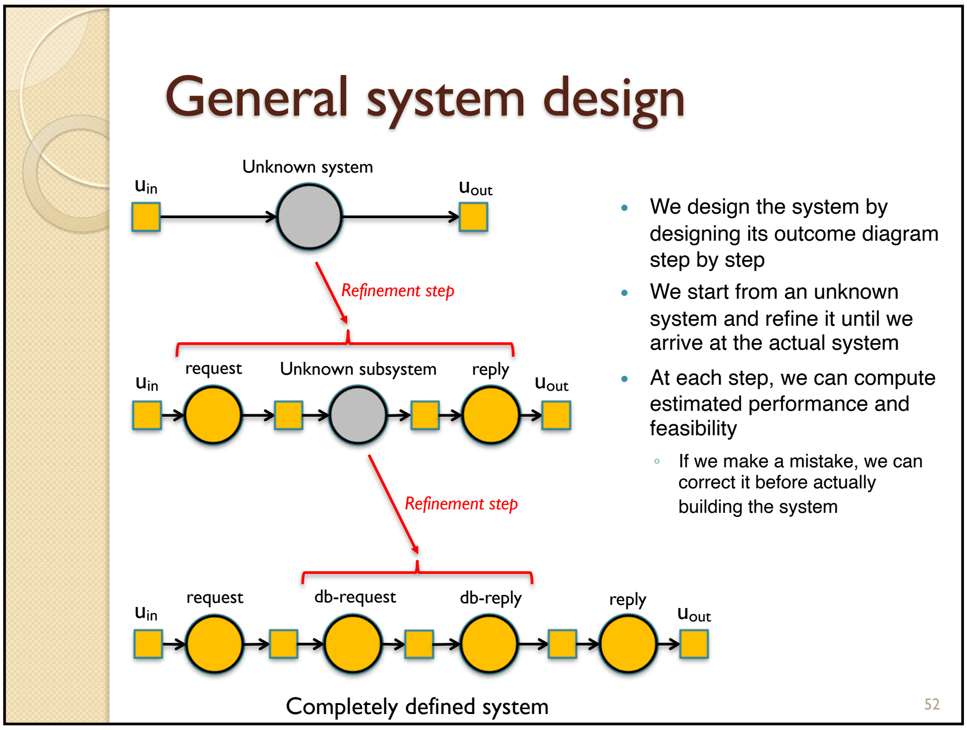
- This is a simple client/server shown as an outcome diagram
- Each square is an event and each circle is an outcome
- Each outcome has its own ΔQ
- Total ΔQ from user click to response displayed is addition of all ΔQs



# General system design

51

51



# General system design

Unknown system

$u_{in}$ $u_{out}$

*Refinement step*

$u_{in}$ request Unknown subsystem reply $u_{out}$

*Refinement step*

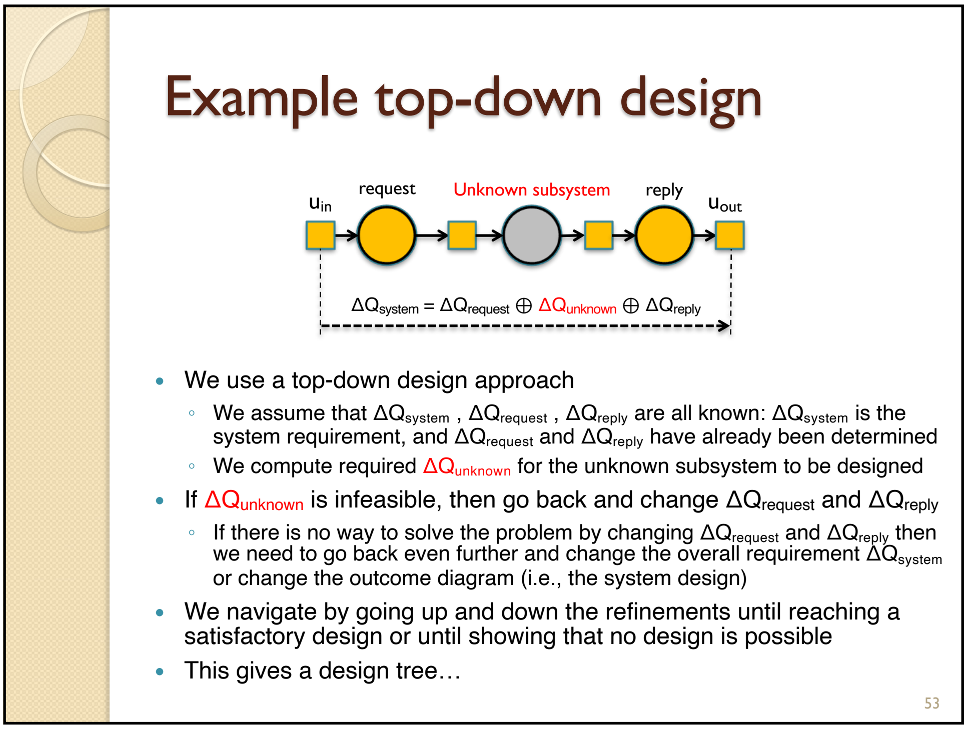
$u_{in}$ request db-request db-reply reply $u_{out}$

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

52

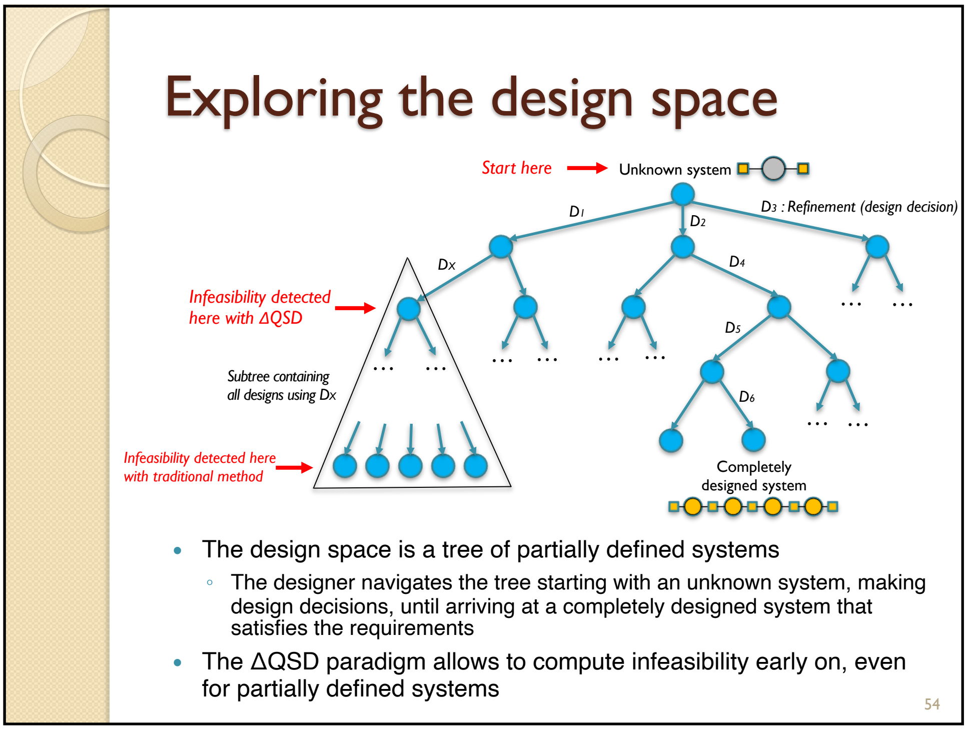
52



# Example top-down design

$u_{in}$
request
Unknown subsystem
reply
$u_{out}$
$\Delta Q_{system} = \Delta Q_{request} \oplus \Delta Q_{unknown} \oplus \Delta Q_{reply}$

- We use a top-down design approach
  - We assume that $\Delta Q_{system}$ , $\Delta Q_{request}$ , $\Delta Q_{reply}$ are all known: $\Delta Q_{system}$ is the system requirement, and $\Delta Q_{request}$ and $\Delta Q_{reply}$ 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 $\Delta Q_{reply}$
  - If there is no way to solve the problem by changing $\Delta Q_{request}$ and $\Delta Q_{reply}$ then we need to go back even further and change the overall requirement $\Delta Q_{system}$ 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…



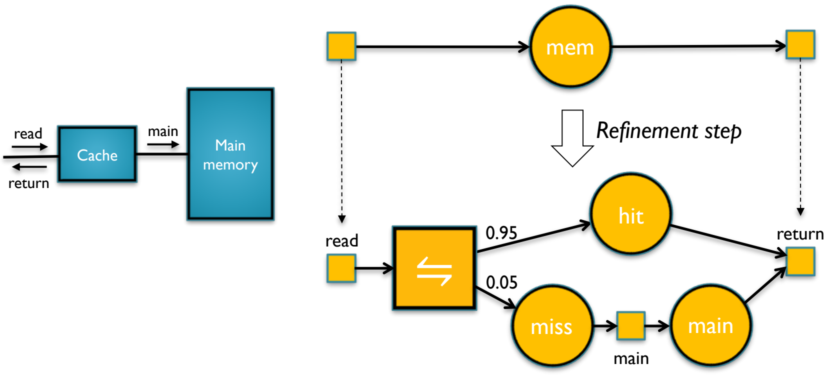
# Exploring the design space

Start here
Unknown system
$D_1$
$D_2$
$D_3$ : Refinement (design decision)
$D_x$
$D_4$
$D_5$
$D_6$
Infeasibility detected here with ΔQSD
Subtree containing all designs using $D_x$
Infeasibility detected here with traditional method
Completely designed system
…

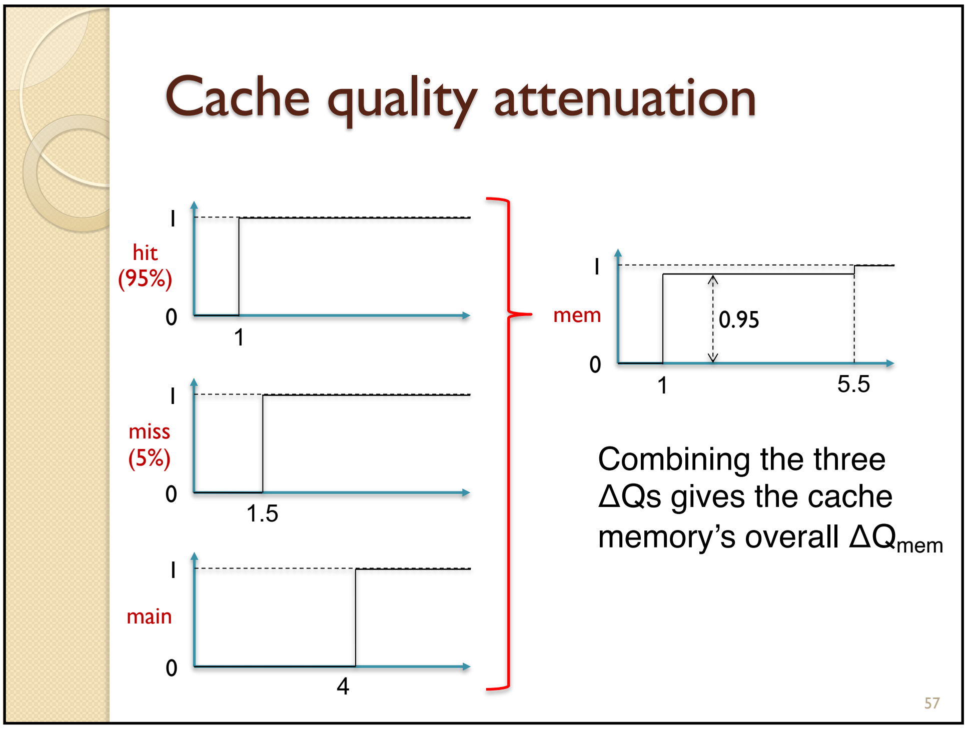
- The design space is a tree of partially defined systems
  - The designer navigates the tree starting with an unknown system, making design decisions, until arriving at a completely designed system that satisfies the requirements
- The ΔQSD paradigm allows to compute infeasibility early on, even for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

## 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

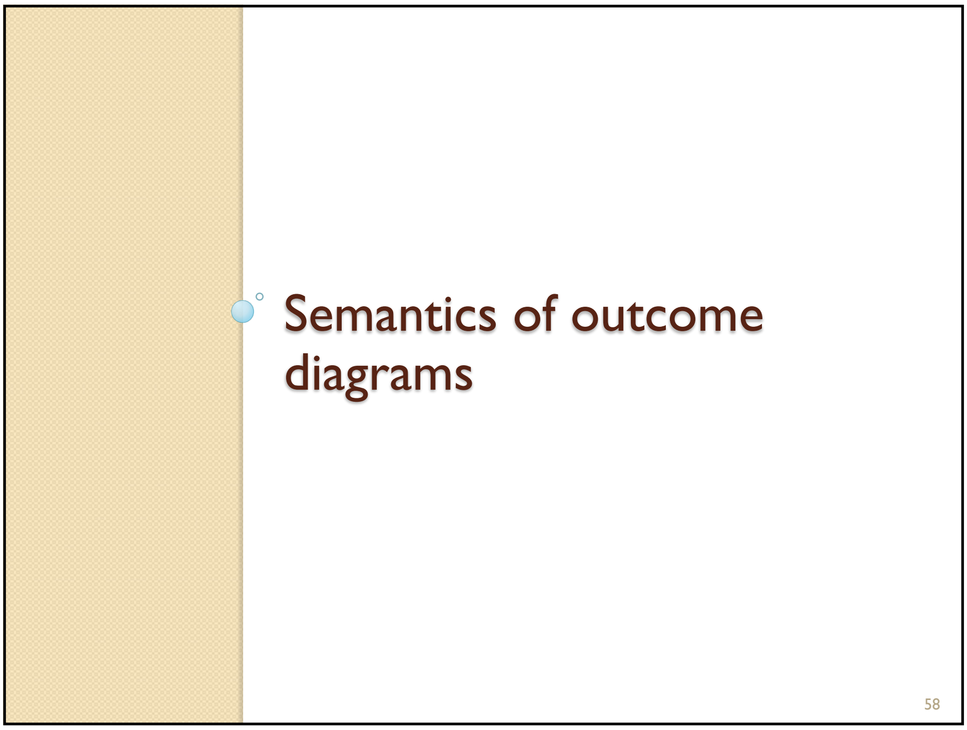


# Cache quality attenuation

Combining the three ΔQs gives the cache memory's overall $\Delta Q_{mem}$

57

57



# Semantics of outcome diagrams

58

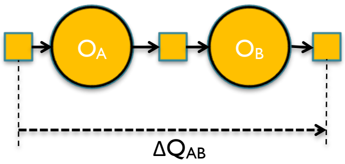
58

# Semantics of outcome diagrams

- Given an outcome diagram and the ΔQs of all outcomes in the diagram, we can compute the ΔQ of the complete diagram
  - Recall that ΔQ(t) is a function of delay t that represents the cumulative probability distribution of the delay (technically, 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 ΔQ 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

59

# Sequential composition



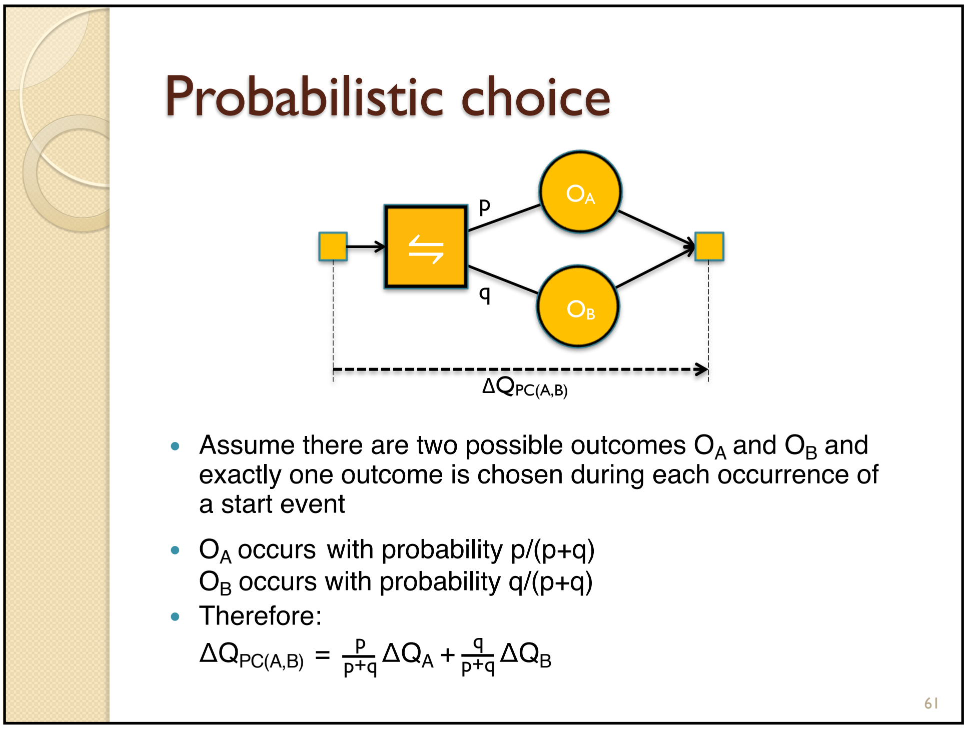
- Assume two outcomes $O_A$ and $O_B$ where the end event of $O_A$ is the start event of $O_B$
- The probability distribution of $O_{AB}$ is the convolution of the probability distributions of $O_A$ and $O_B$
- Therefore:

$\Delta Q'_{AB} = \Delta Q'_A \oplus \Delta Q'_B$

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

60

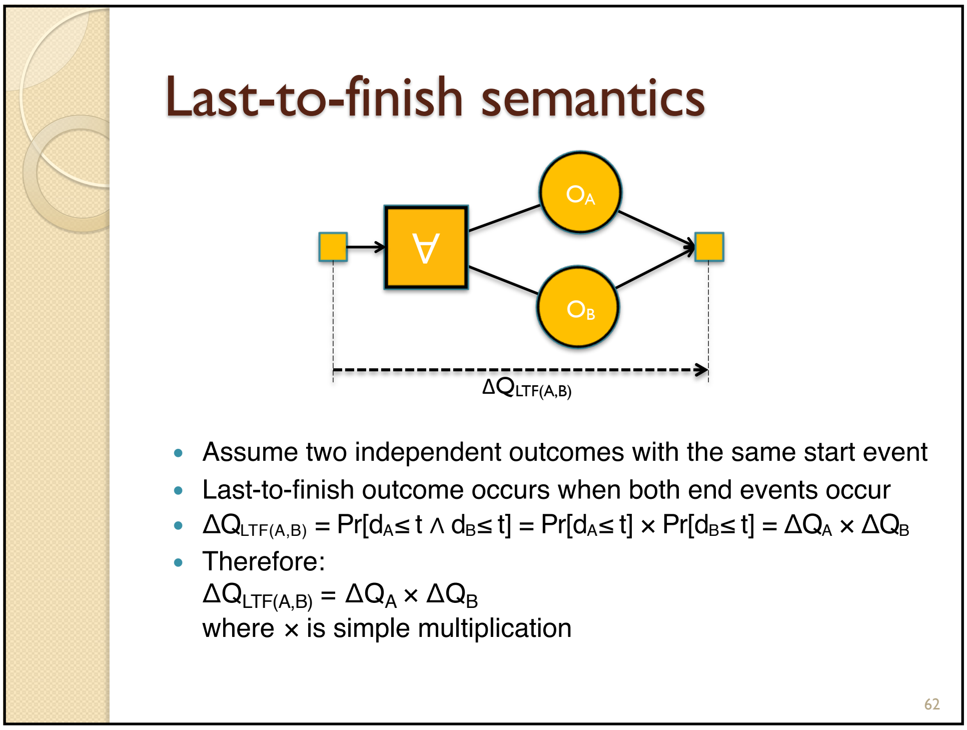


# Probabilistic choice

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

61

61

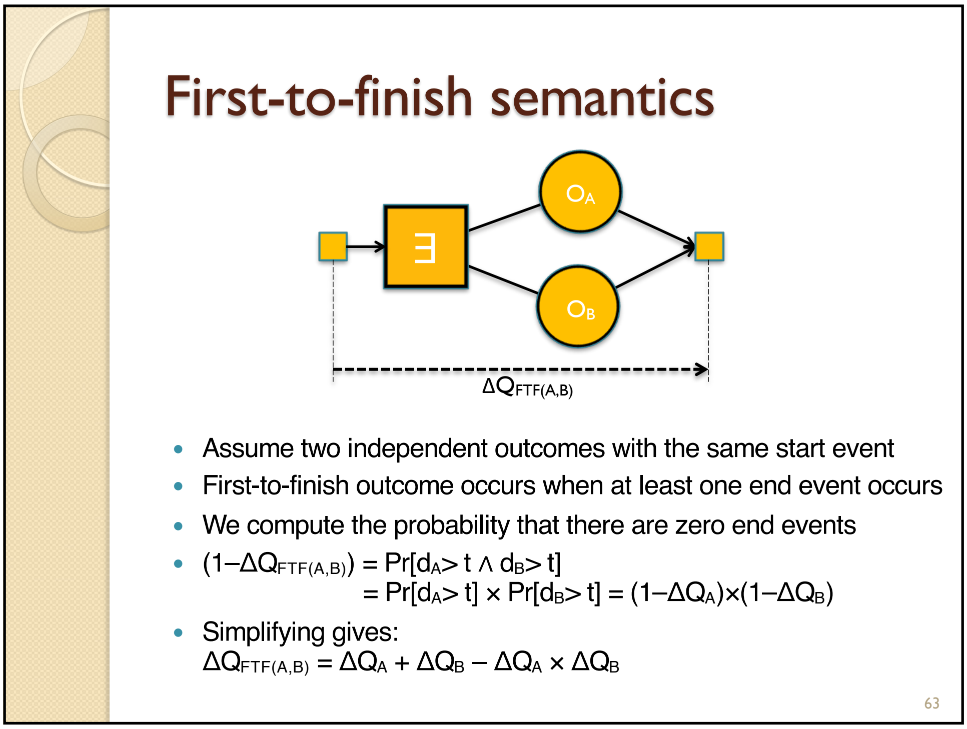


# Last-to-finish semantics

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

62

62

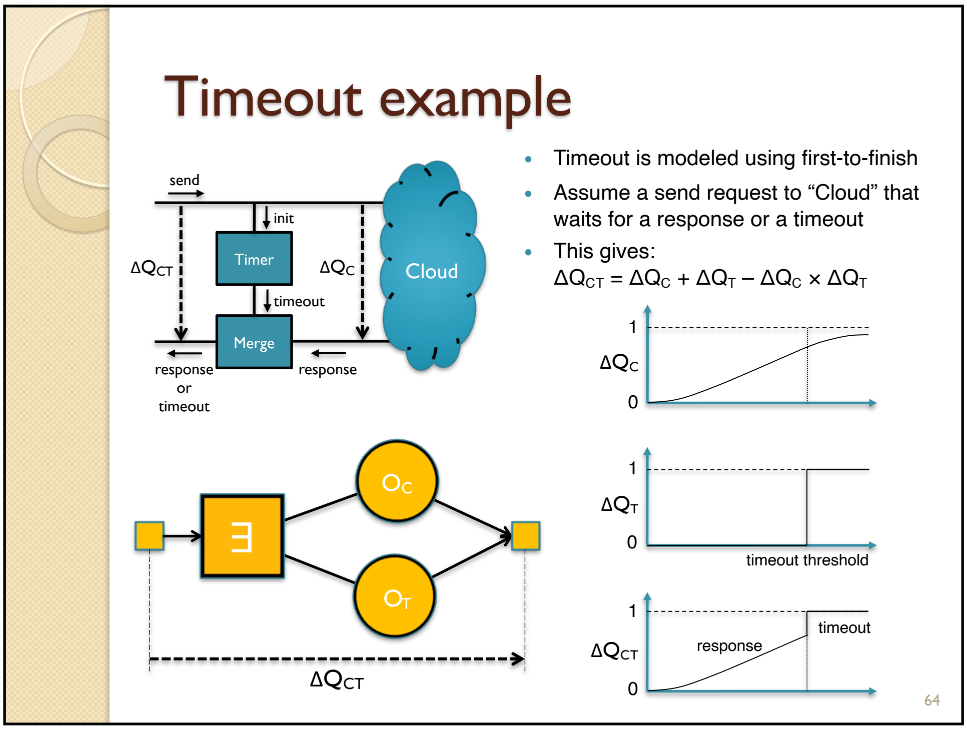


# First-to-finish semantics

- 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:
  $\Delta Q_{FTF(A,B)} = \Delta Q_A + \Delta Q_B – \Delta Q_A \times \Delta Q_B$

63

63

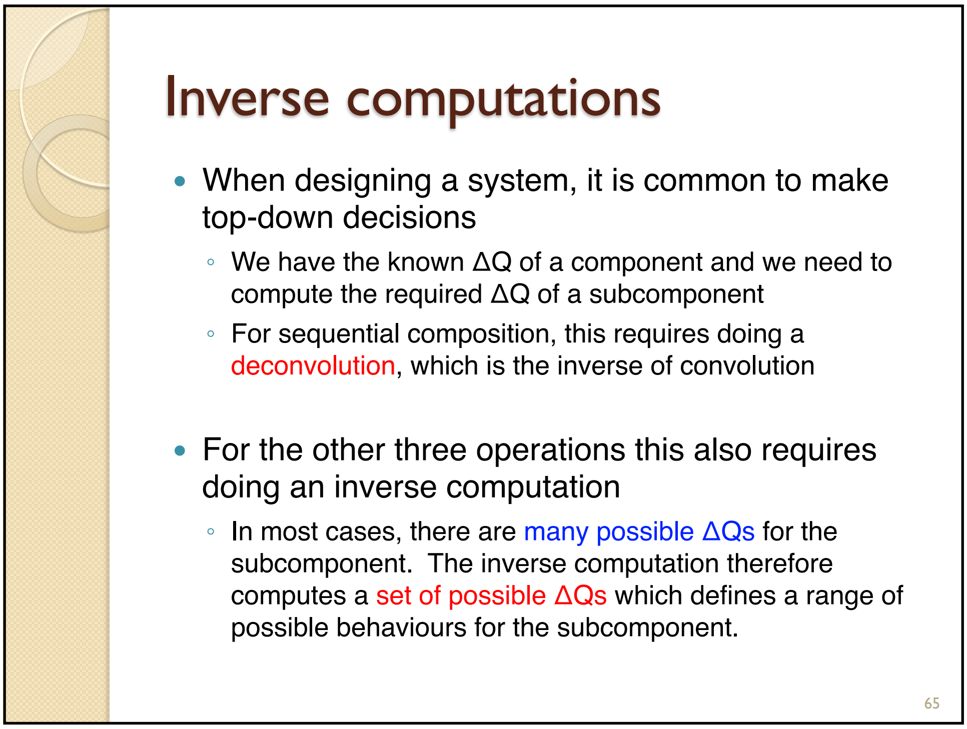


# Timeout example

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

64

64

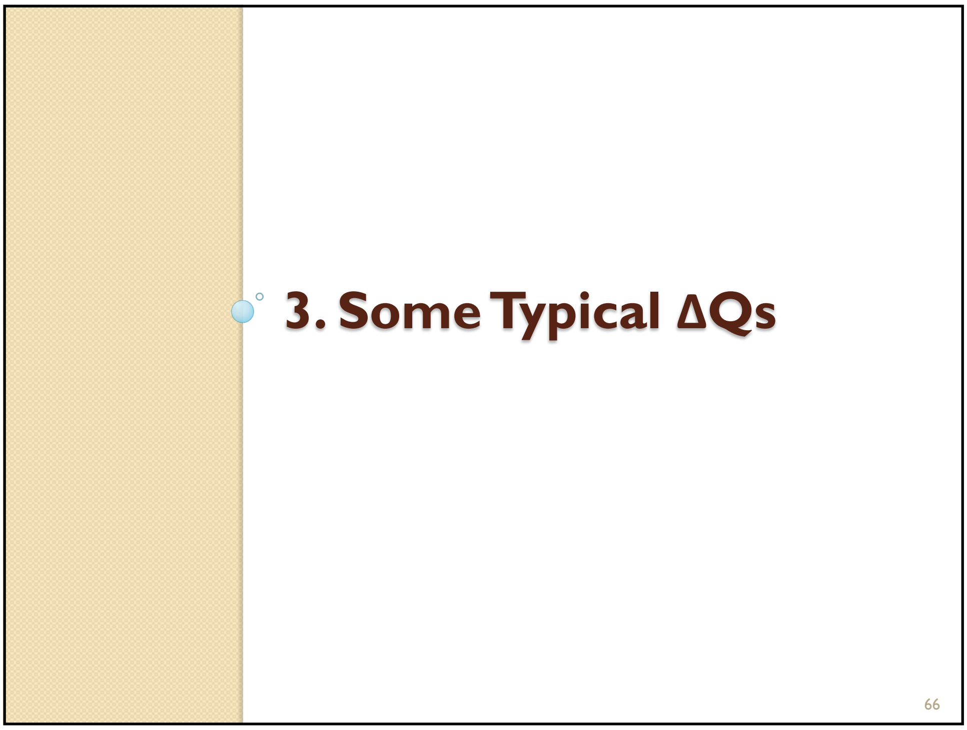


# Inverse computations

- When designing a system, it is common to make top-down decisions
  - We have the known ΔQ of a component and we need to compute the required ΔQ of a subcomponent
  - For sequential composition, this requires doing a deconvolution, which is the inverse of convolution

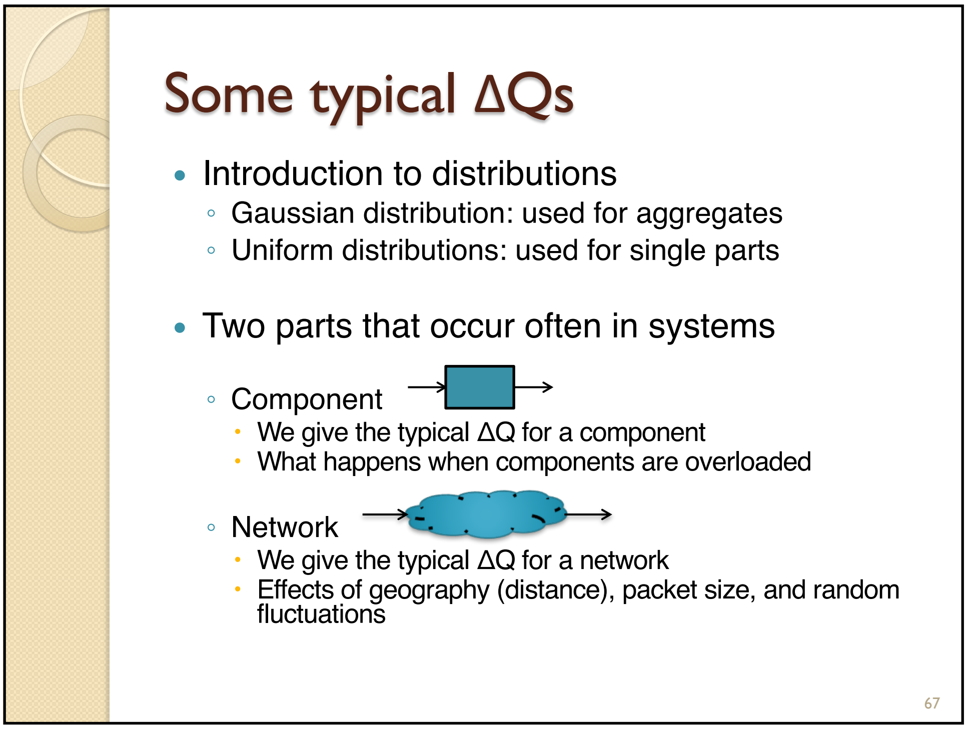
- For the other three operations this also requires doing an inverse computation
  - In most cases, there are many possible ΔQs for the subcomponent. The inverse computation therefore computes a set of possible ΔQs which defines a range of possible behaviours for the subcomponent.

65



# 3. Some Typical ΔQs

66



# Some typical ΔQs

- 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 ΔQ for a component
    - What happens when components are overloaded
  - Network
    - We give the typical ΔQ for a network
    - Effects of geography (distance), packet size, and random fluctuations

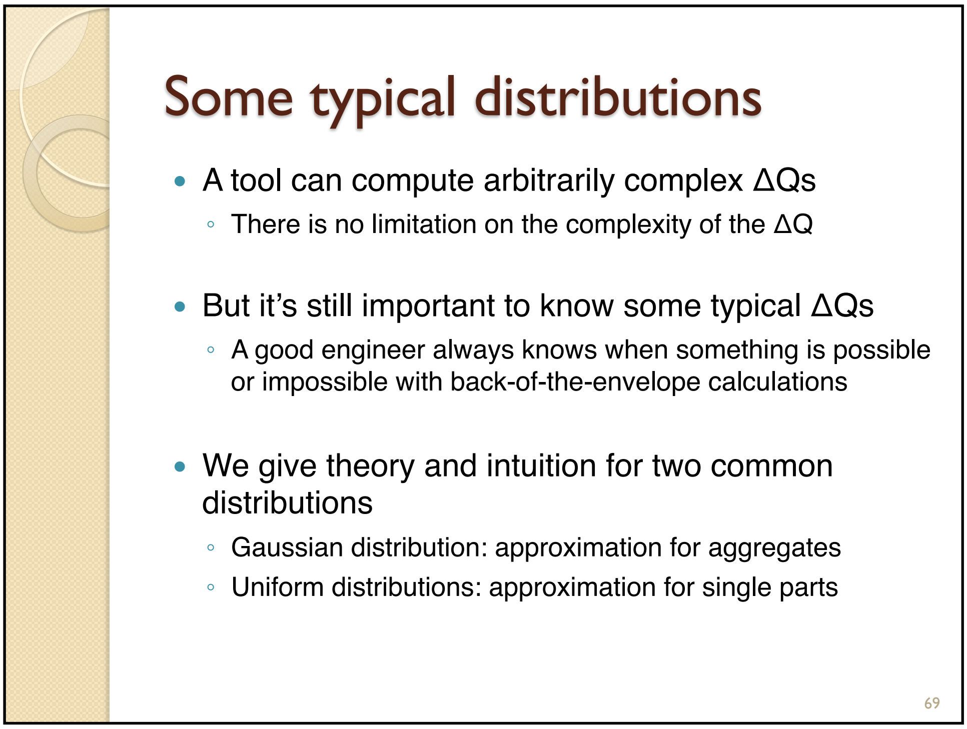
67

67

# Some typical distributions

68

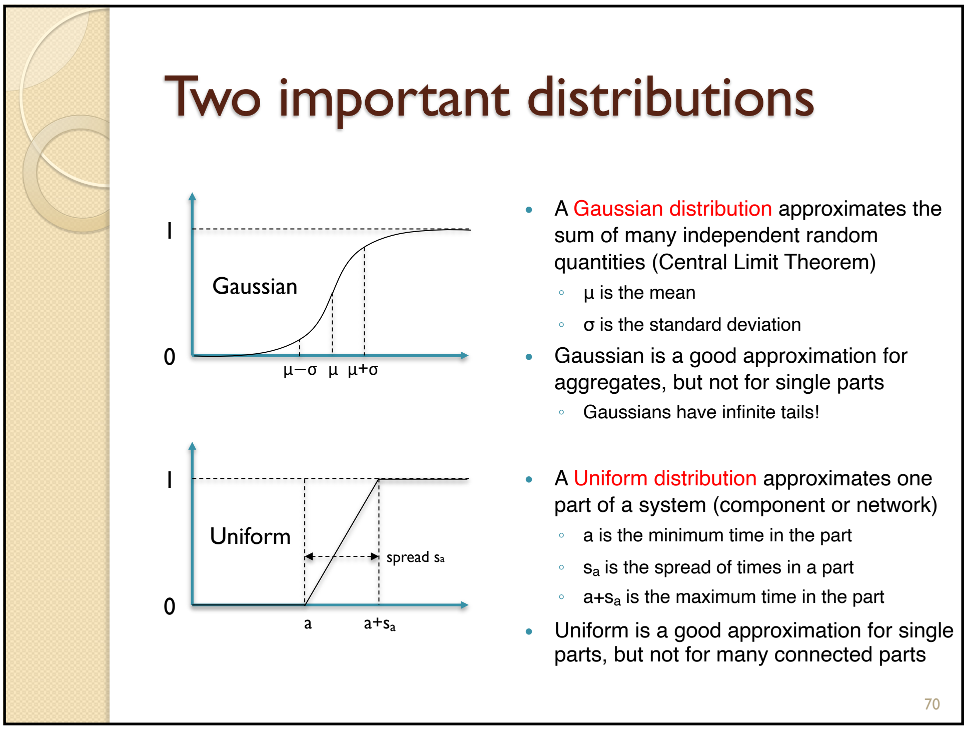
68



# Some typical distributions

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

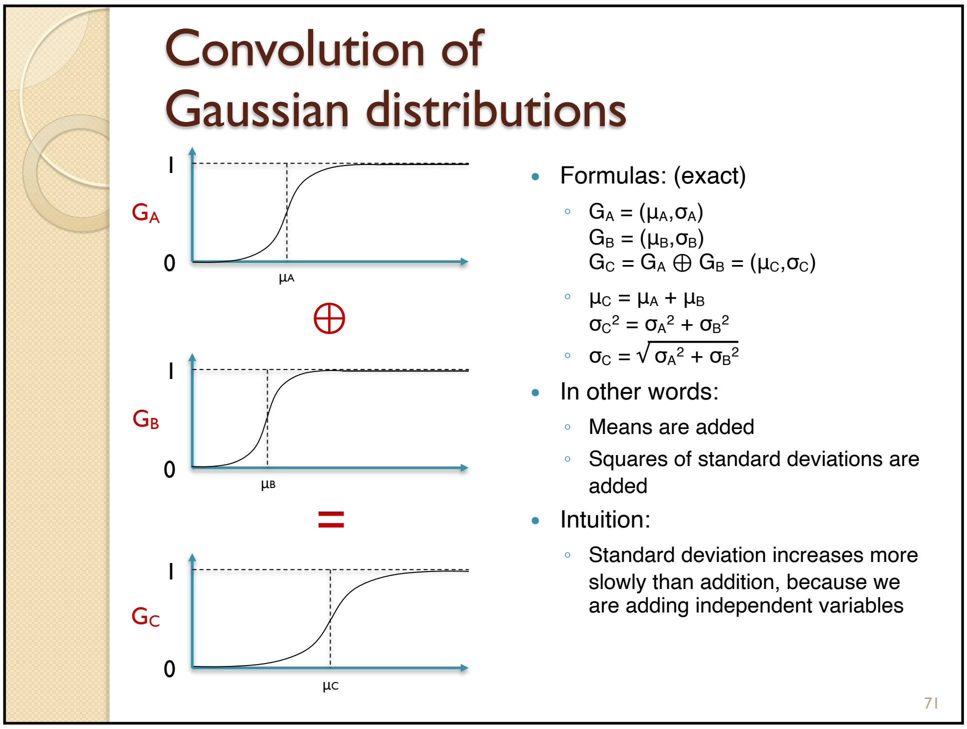
69



# Two important distributions

- A Gaussian distribution approximates the sum of many independent random quantities (Central Limit Theorem)
  - μ is the mean
  - σ 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

70

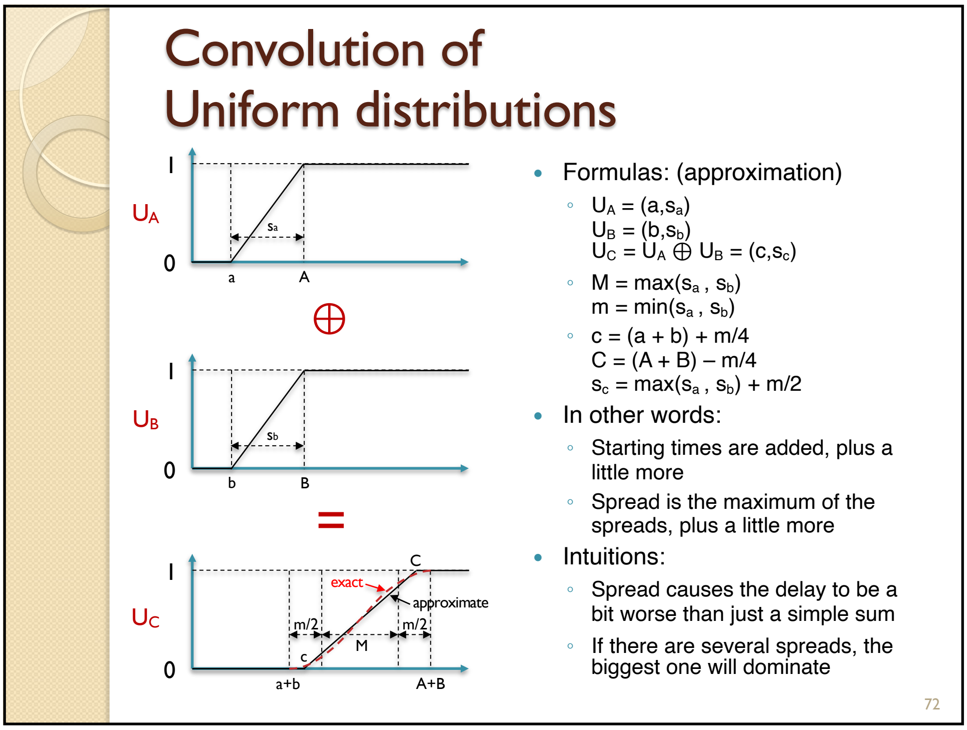


# Convolution of Gaussian distributions

- 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

71

71



# Convolution of Uniform distributions

- Formulas: (approximation)
  - $U_A = (a,s_a)$
    $U_B = (b,s_b)$
    $U_C = U_A \oplus U_B = (c,s_c)$
  - $M = \max(s_a , s_b)$
    $m = \min(s_a , s_b)$
  - $c = (a + b) + m/4$
    $C = (A + B) – m/4$
    $s_c = \max(s_a , s_b) + m/2$
- 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

72

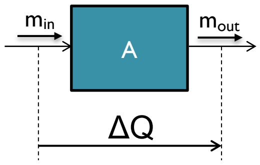
72

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

73

73

## 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

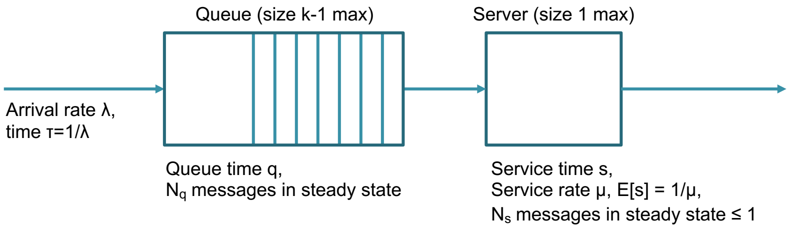
$\Delta$Q {Failure rate f, Delay d}

- These four parameters are all related
  - $\Delta$Q is function of offered load and buffer size

74

74

# M/M/1/K queue



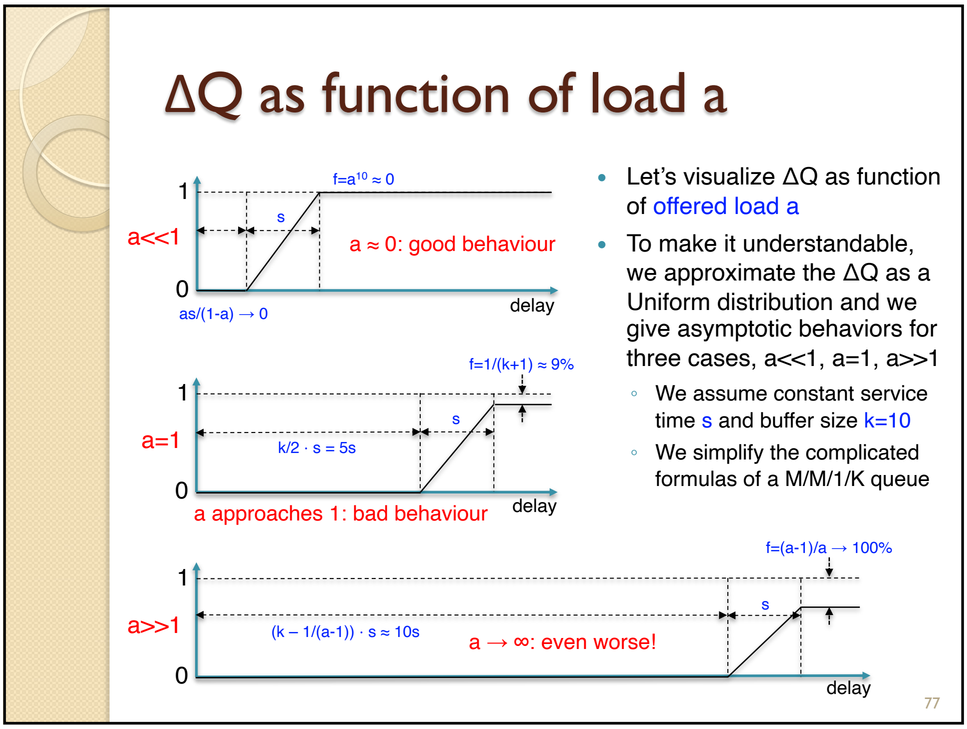
- We model a component as an M/M/1/K queue
  - M: arriving messages have Exponential distribution with rate λ
  - M: service time has Exponential distribution with rate μ
  - 1: one message can be served at a time
  - k: total buffer size is k (buffer size = queue size k-1 + server size 1)
- Offered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

# Effect of offered load a

- The offered load is the most important parameter
  - a<1: the component has enough power to service all messages
  - a>1: the component is overloaded and performs very badly
- Low load (a<0.8)
  - Failure tends to 0, delay tends to 1 (as k increases)
  - An underloaded component behaves very well

!

- High load (a≥0.8)
  - When a gets close to 1 (around 0.8) things quickly get worse!
  - When a>>1, failure rate tends to (a-1)/a, up to 100% for high load!
  - Delay increases very quickly when a approaches 1
    - When a=1, delay is already k/2, half of buffer size, which can be huge
- Quick switchover somewhere between a=0.5 and a=1
  - As the load increases beyond 0.5, the system quickly gets very bad
  - The exact threshold depends on what you consider bad!
  - Even a temporary overload causes a big, long-lasting degradation
    - *This is the cause of the problem in the small cells case study*

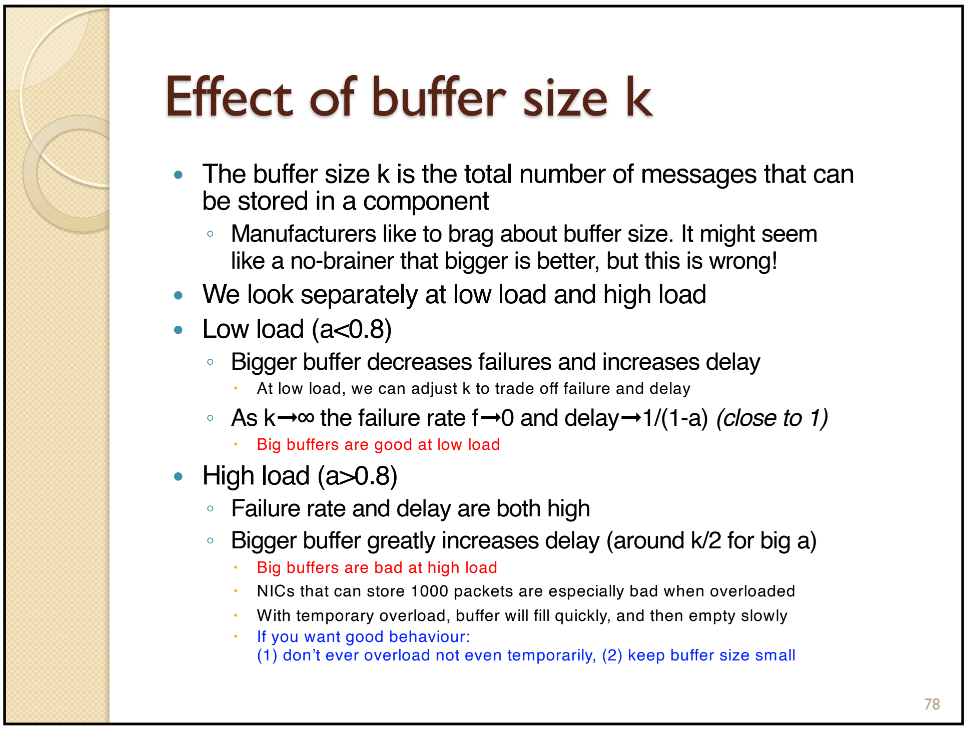


# ΔQ as function of load a

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

77

77

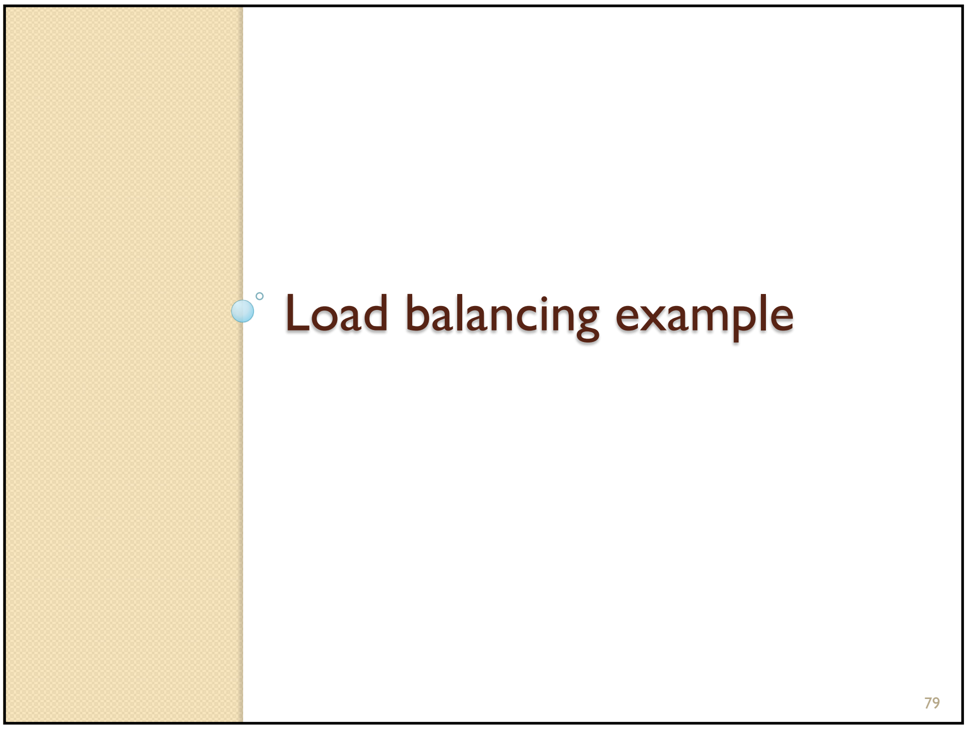


# Effect of buffer size k

- The buffer size k is the total number of messages that can be stored in a component
  - Manufacturers like to brag about buffer size. It might seem like a no-brainer that bigger is better, but this is wrong!
- We look separately at low load and high load
- Low load (a<0.8)
  - Bigger buffer decreases failures and increases delay
    - At low load, we can adjust k to trade off failure and delay
  - As k→∞ the failure rate f→0 and delay→1/(1-a) *(close to 1)*
    - 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:<br>(1) don't ever overload not even temporarily, (2) keep buffer size small

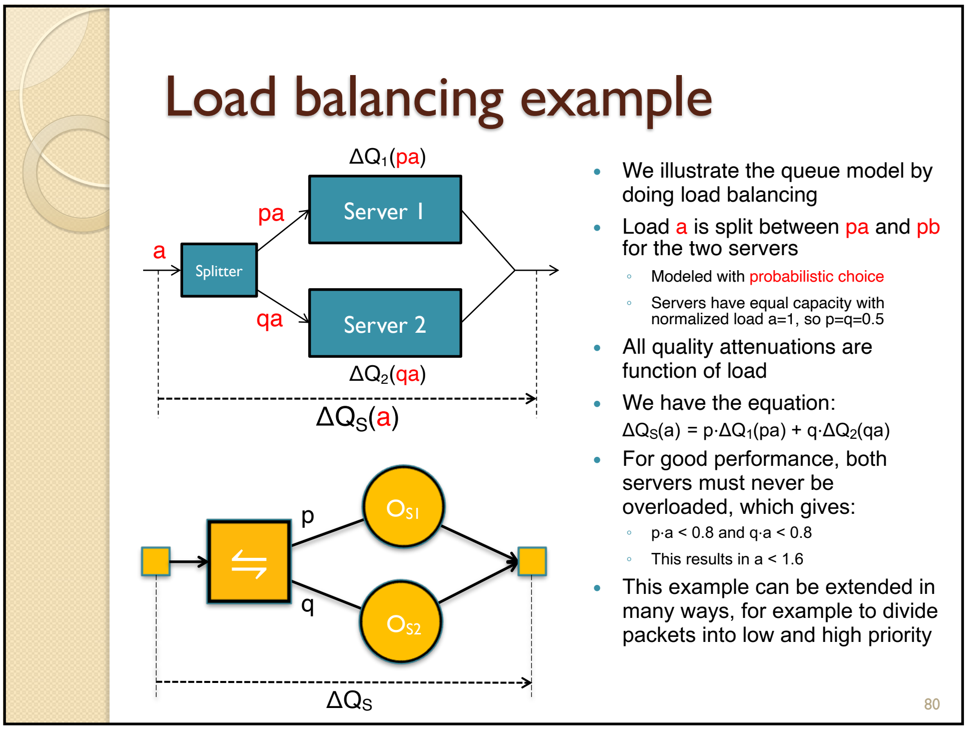
78

78



# Load balancing example

79



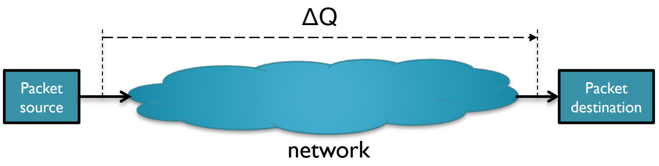
## Load balancing example

- We illustrate the queue model by doing load balancing
- Load a is split between pa and pb 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: $\Delta Q_S(a) = p{\cdot}\Delta Q_1(pa) + q{\cdot}\Delta Q_2(qa)$
- 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

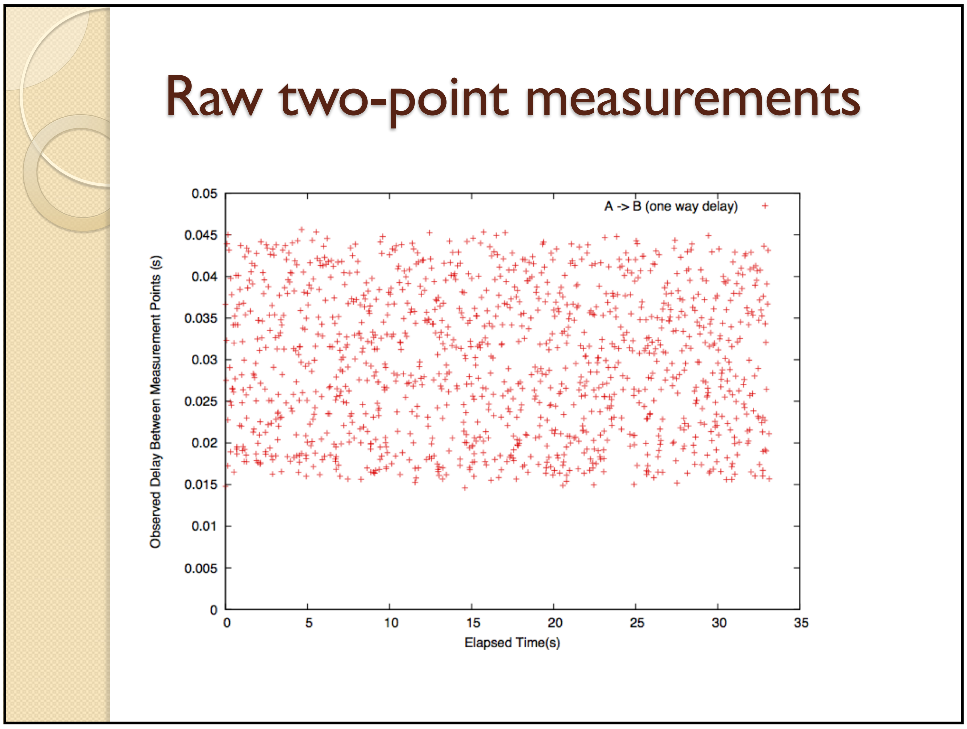
80

# ΔQ for a typical network

## ΔQ for network packets

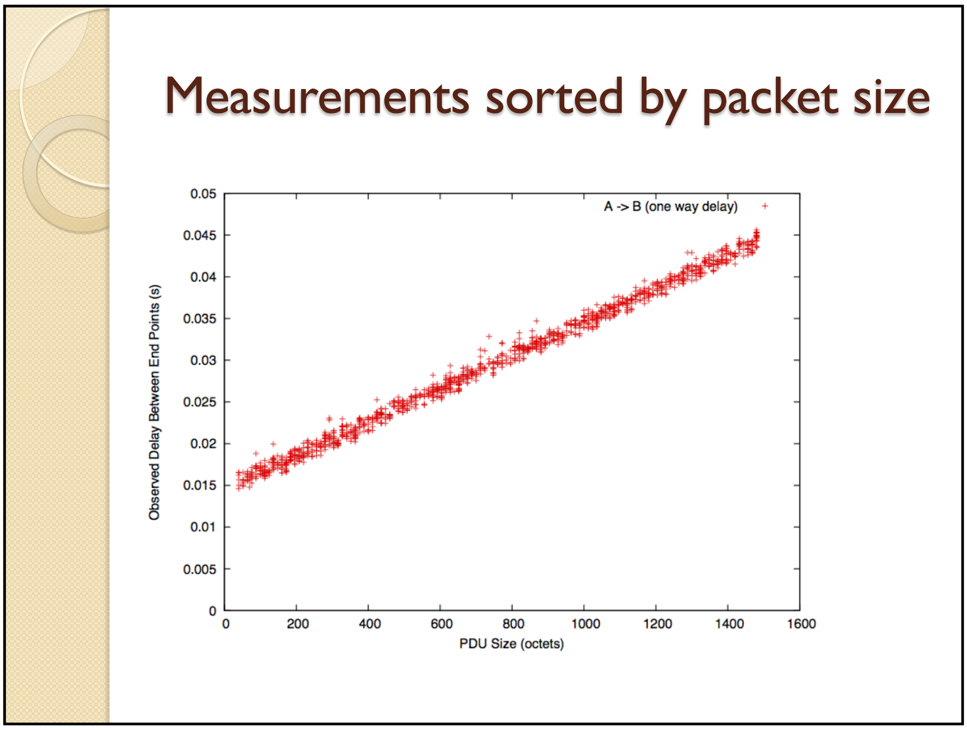


- We can study what the real ΔQ is for networks delivering packets
- Experience shows that the ΔQ has three parameters G, S, V:
  - $\Delta Q = \Delta Q_G \oplus \Delta Q_S \oplus \Delta Q_V$
- Again, we add ΔQs using convolution
  - Because of the simple structure, the equations are simple



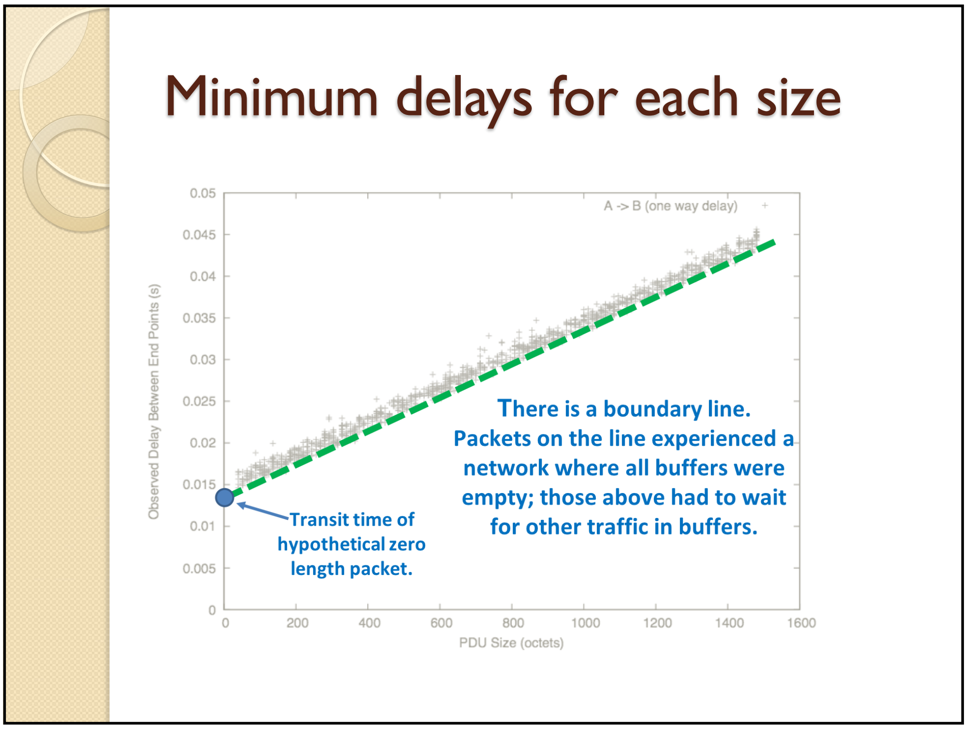
# Raw two-point measurements

83



# Measurements sorted by packet size

84

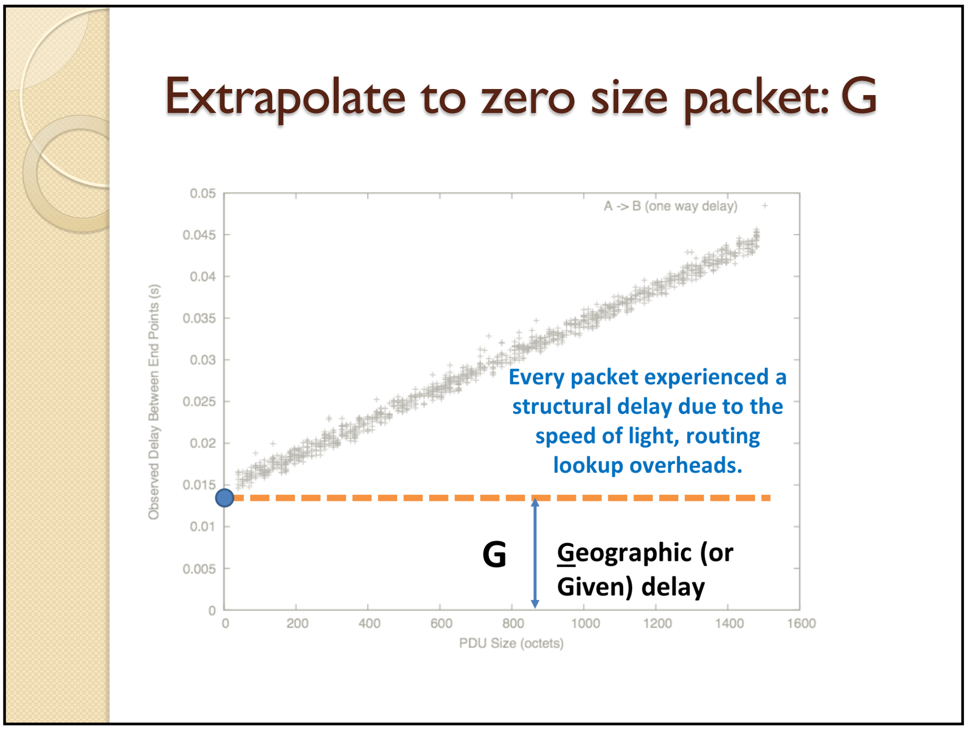


# Minimum delays for each size

There is a boundary line. Packets on the line experienced a network where all buffers were empty; those above had to wait for other traffic in buffers.

Transit time of hypothetical zero length packet.

85

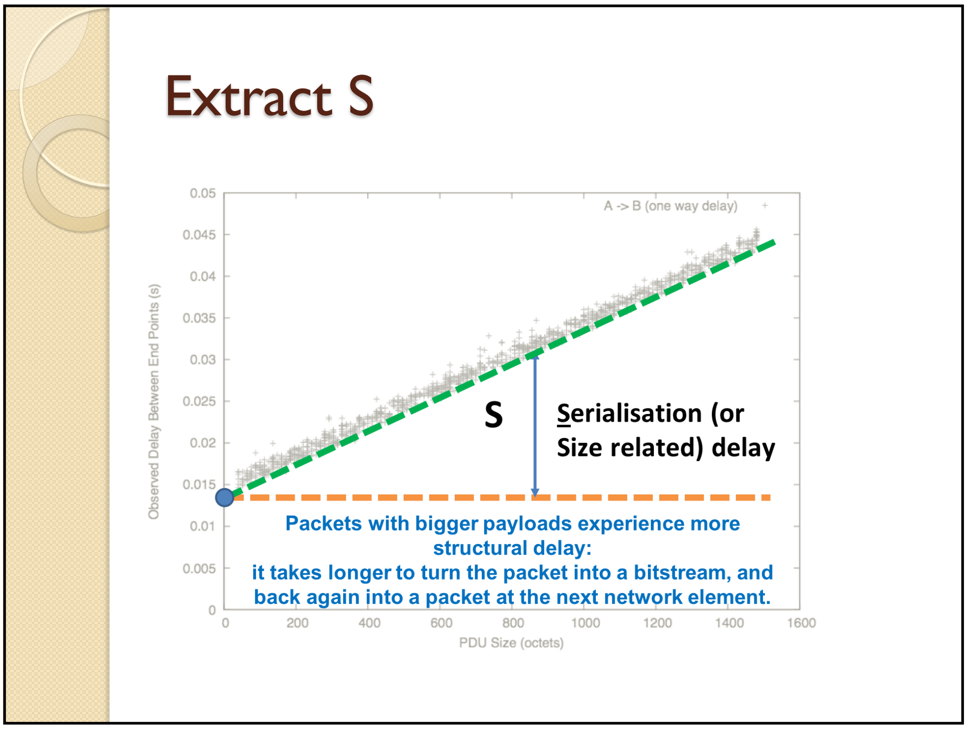


# Extrapolate to zero size packet: G

Every packet experienced a structural delay due to the speed of light, routing lookup overheads.

Geographic (or Given) delay

86

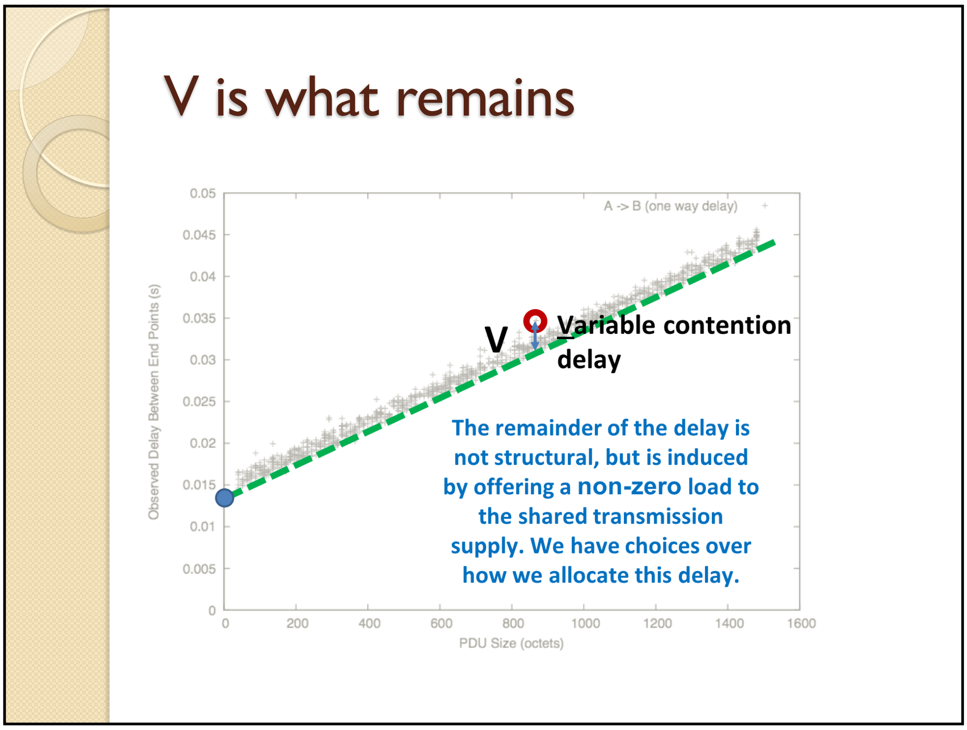


# Extract S

Serialisation (or Size related) delay

Packets with bigger payloads experience more structural delay:
it takes longer to turn the packet into a bitstream, and back again into a packet at the next network element.

87

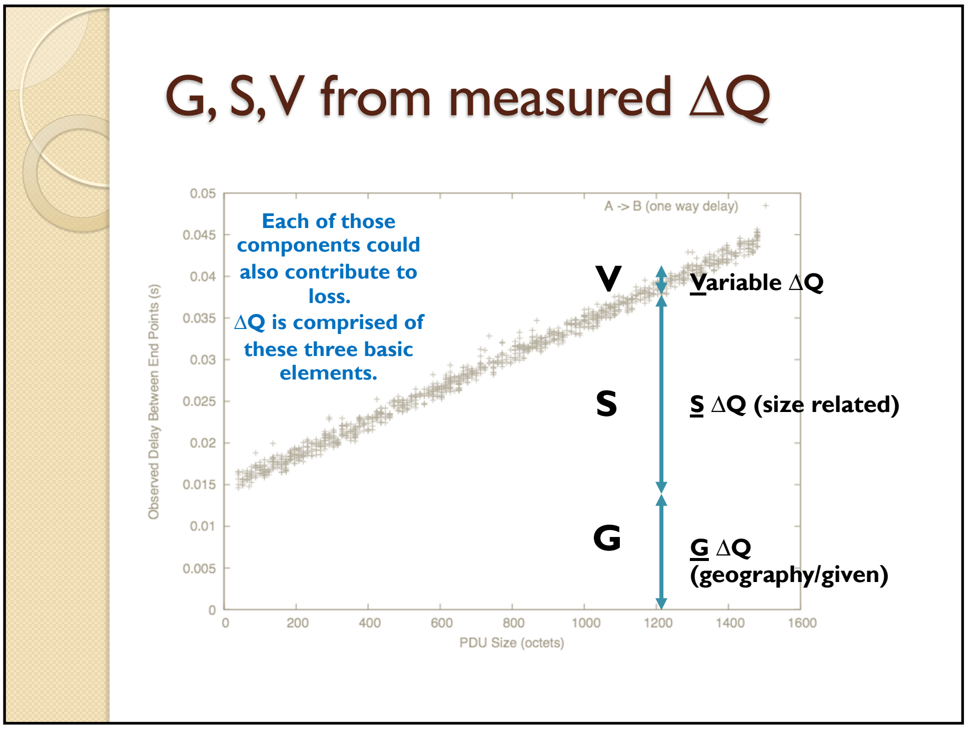


# V is what remains

Variable contention delay

The remainder of the delay is not structural, but is induced by offering a non-zero load to the shared transmission supply. We have choices over how we allocate this delay.

88



# G, S,V from measured ΔQ

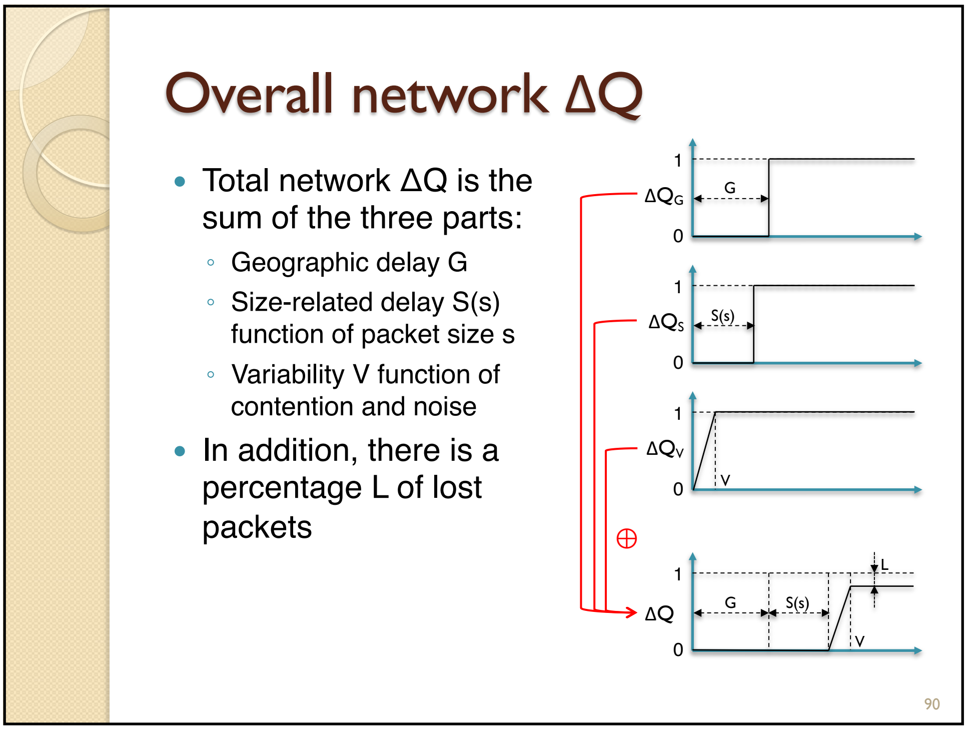
Each of those components could also contribute to loss.
ΔQ is comprised of these three basic elements.

V — Variable ΔQ

S — S ΔQ (size related)

G — G ΔQ (geography/given)

89

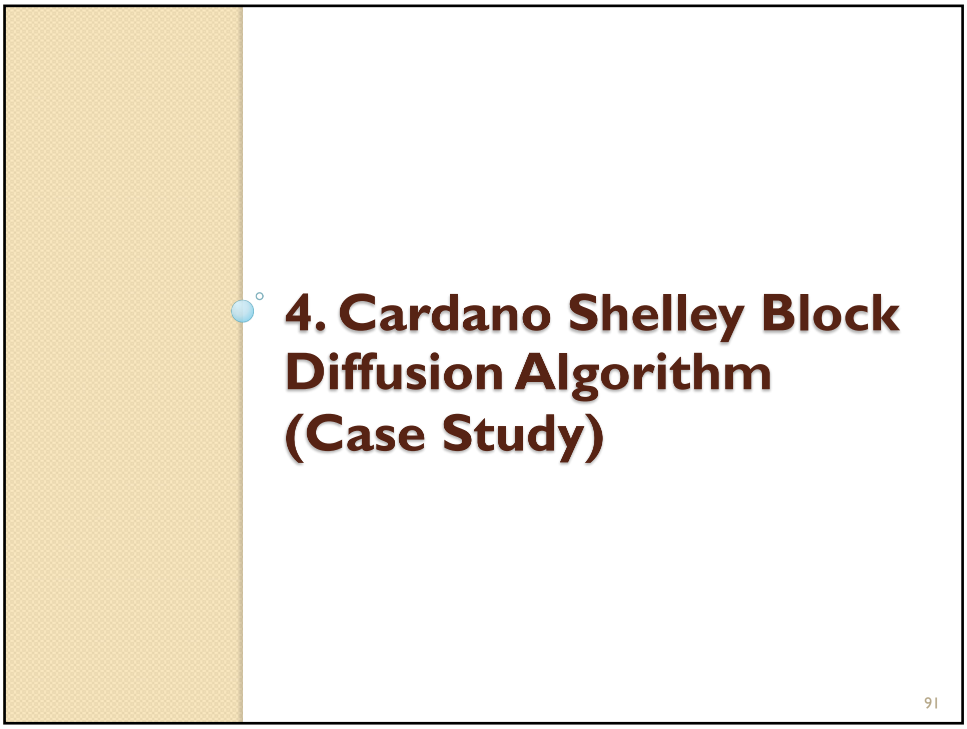


# Overall network ΔQ

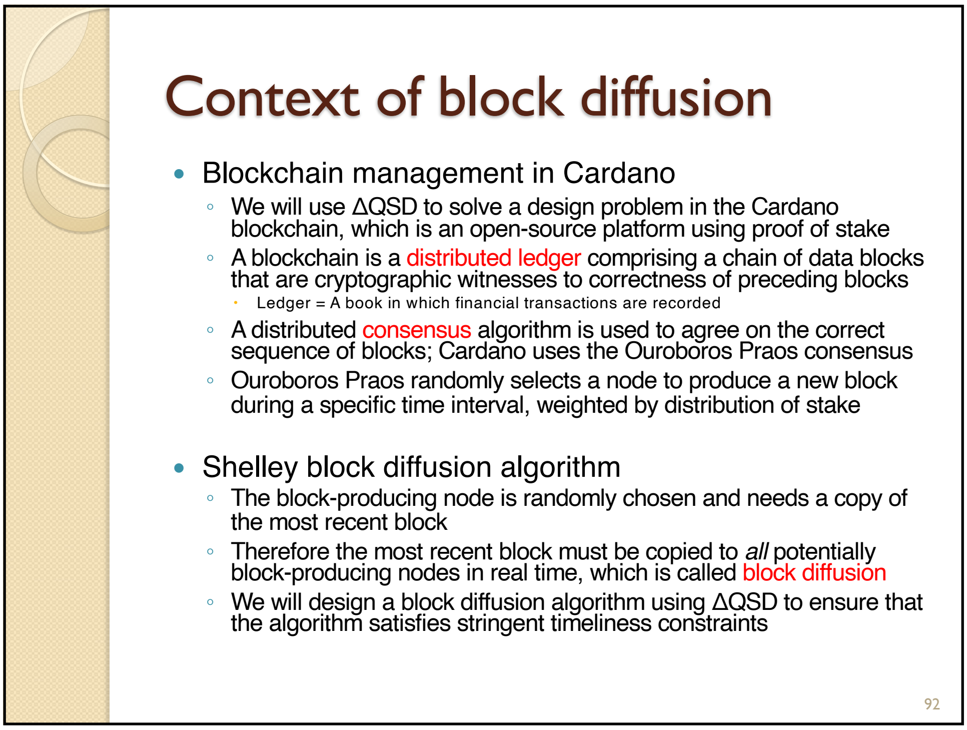
- Total network ΔQ is the sum of the three parts:
  - Geographic delay G
  - Size-related delay S(s) function of packet size s
  - Variability V function of contention and noise
- In addition, there is a percentage L of lost packets

90

90



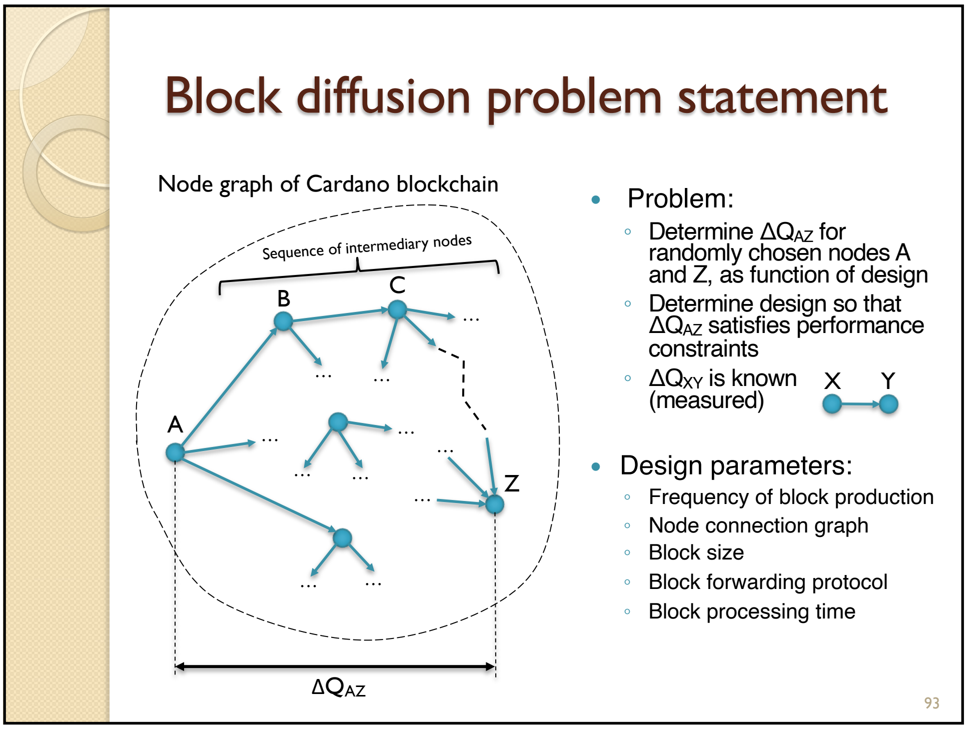
# 4. Cardano Shelley Block Diffusion Algorithm (Case Study)



## 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

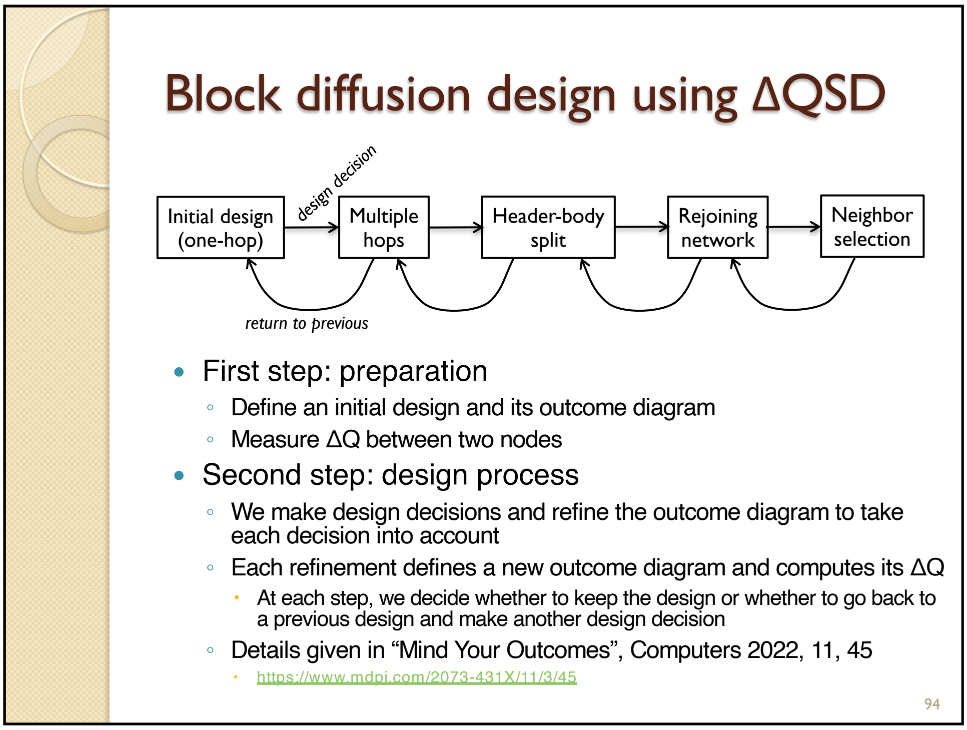


# Block diffusion problem statement

Node graph of Cardano blockchain
Sequence of intermediary nodes
A
B
C
Z
…
$\Delta Q_{AZ}$

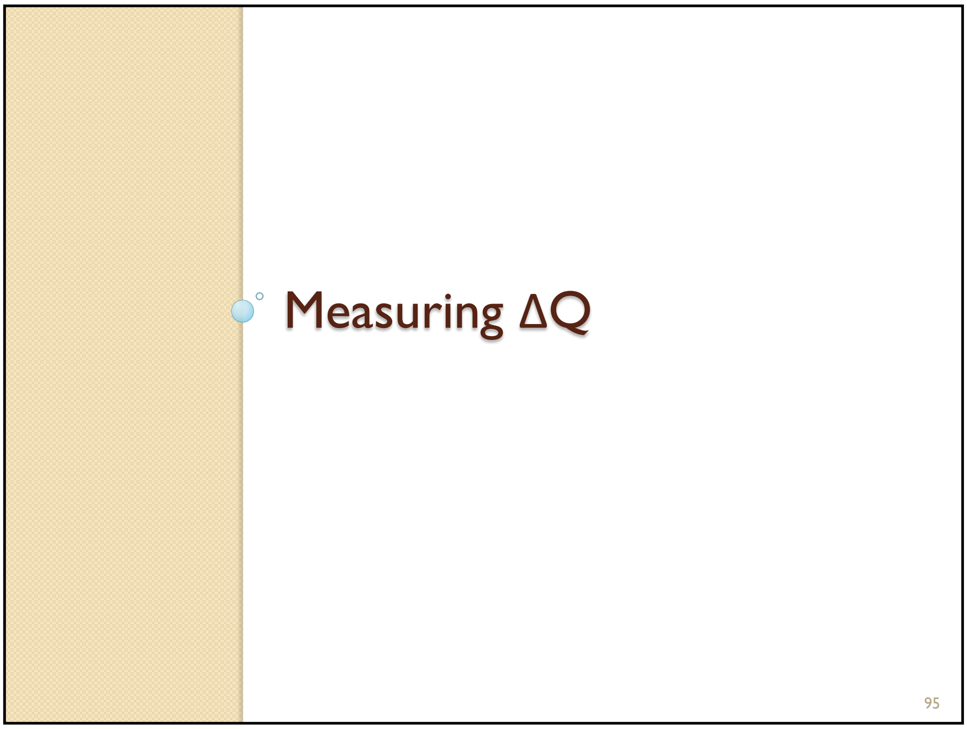
- Problem:
  - Determine $\Delta Q_{AZ}$ for randomly chosen nodes A and Z, as function of design
  - Determine design so that $\Delta Q_{AZ}$ satisfies performance constraints
  - $\Delta Q_{XY}$ is known (measured)

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

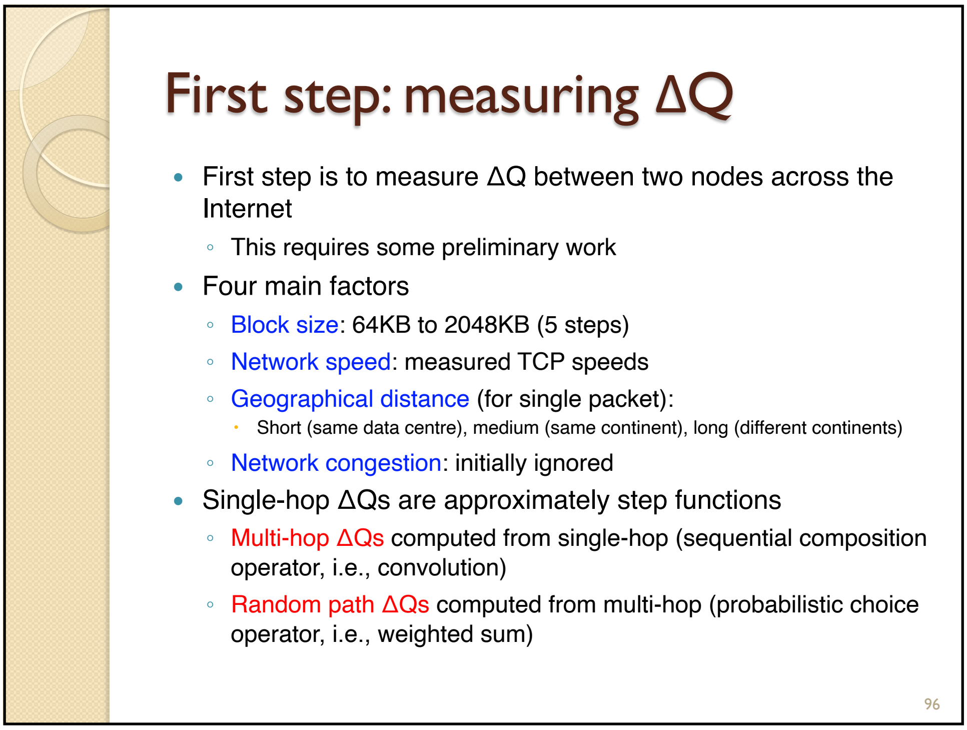


# Block diffusion design using ΔQSD

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

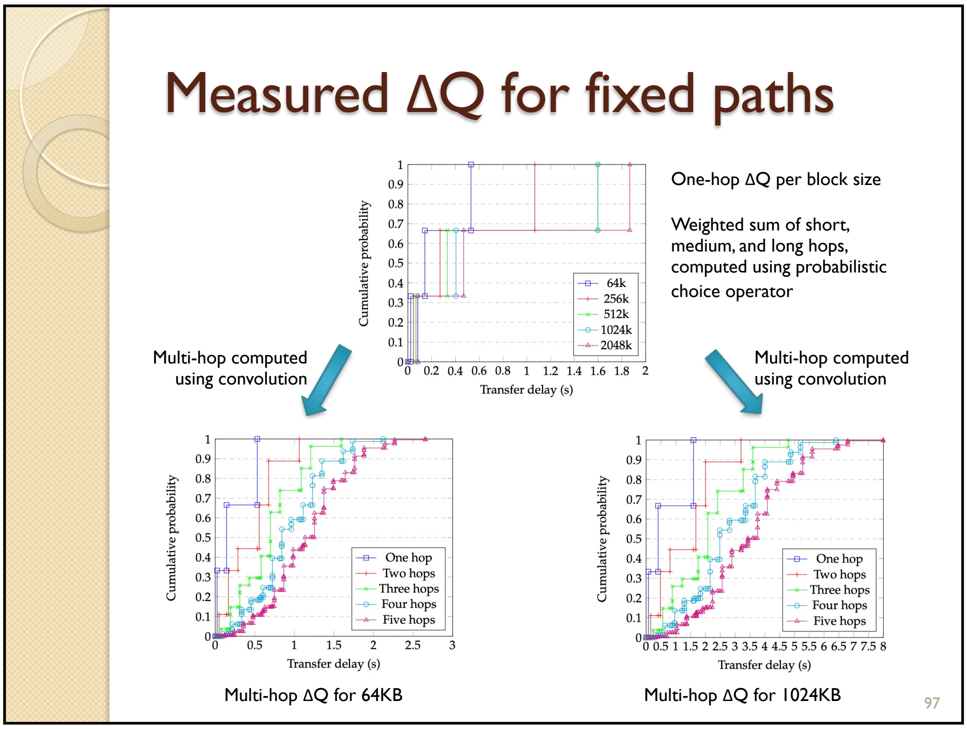


## Measuring ΔQ



## First step: measuring ΔQ

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



## Measured ΔQ for fixed paths

One-hop ΔQ per block size

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

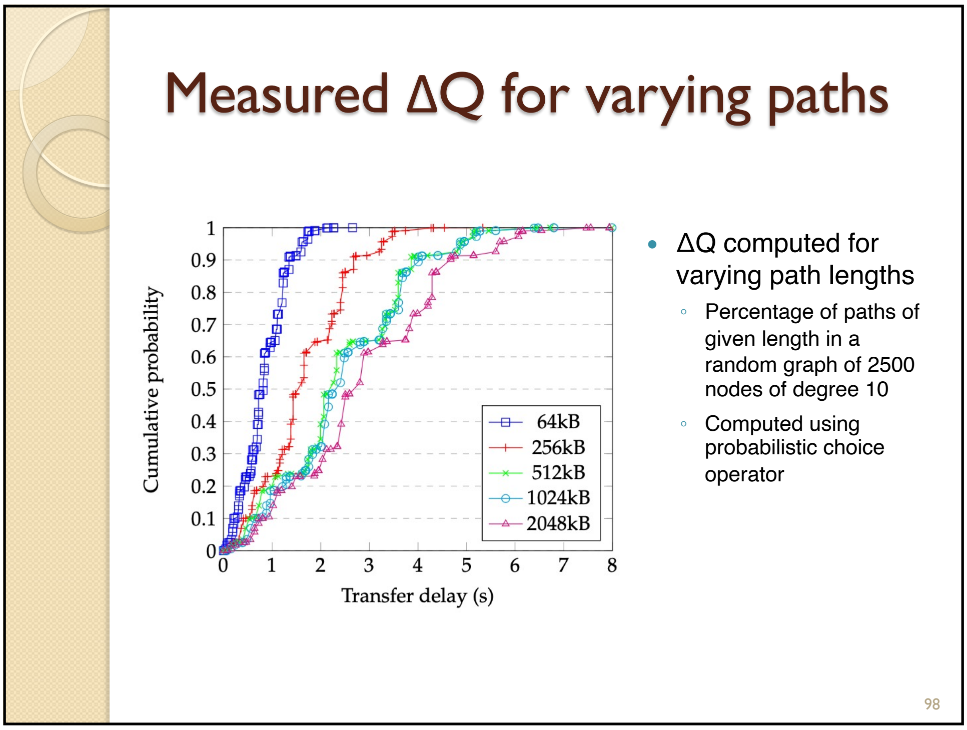
Multi-hop computed using convolution

Multi-hop computed using convolution

Multi-hop ΔQ for 64KB

Multi-hop ΔQ for 1024KB

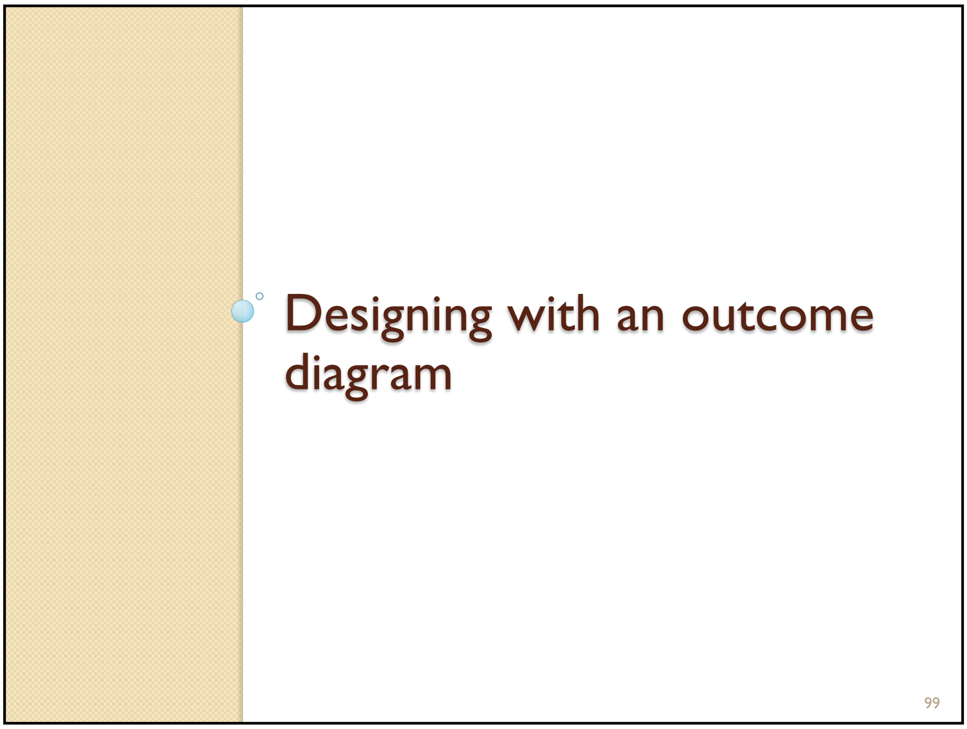
97



## 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

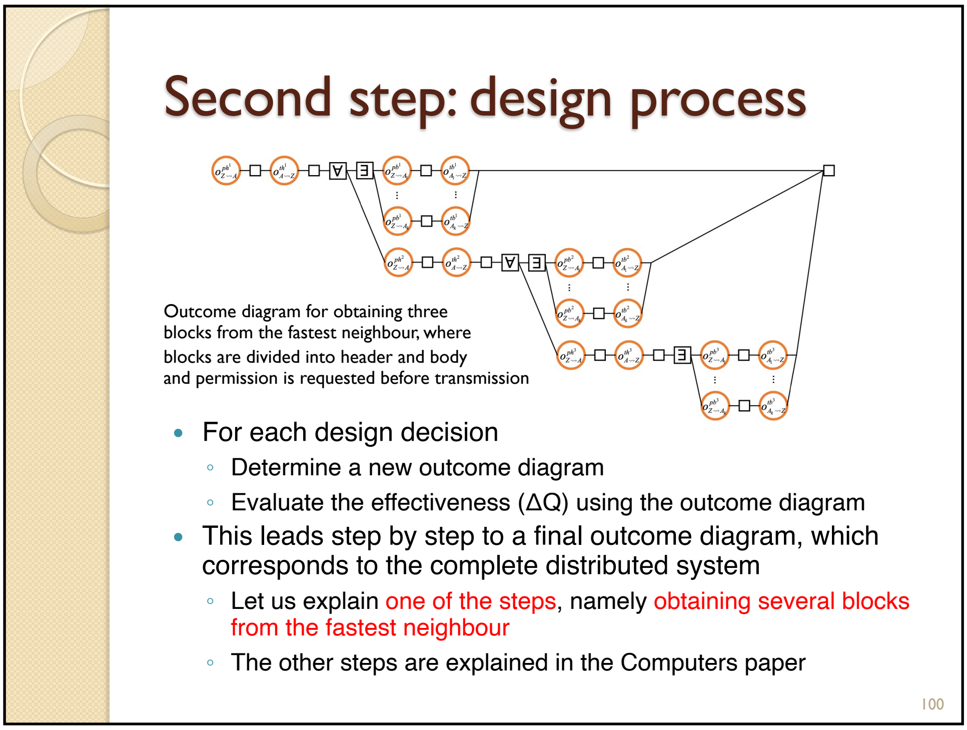
98



# Designing with an outcome diagram

99

99



# Second step: design process

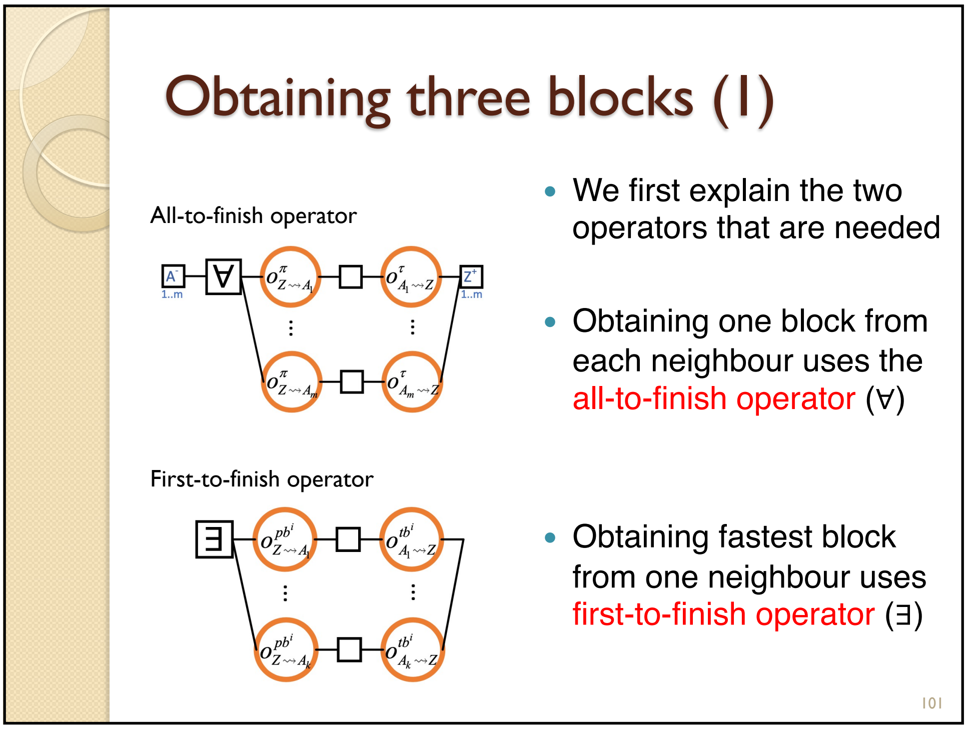
$o^{ph^1}_{Z \rightsquigarrow A}$
$o^{th^1}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^1}_{Z \rightsquigarrow A_1}$
$o^{tb^1}_{A_1 \rightsquigarrow Z}$
$o^{pb^1}_{Z \rightsquigarrow A_k}$
$o^{tb^1}_{A_k \rightsquigarrow Z}$
$o^{ph^2}_{Z \rightsquigarrow A}$
$o^{th^2}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^2}_{Z \rightsquigarrow A_1}$
$o^{tb^2}_{A_1 \rightsquigarrow Z}$
$o^{pb^2}_{Z \rightsquigarrow A_k}$
$o^{tb^2}_{A_k \rightsquigarrow Z}$
$o^{ph^3}_{Z \rightsquigarrow A}$
$o^{th^3}_{A \rightsquigarrow Z}$
∃
$o^{pb^3}_{Z \rightsquigarrow A_1}$
$o^{tb^3}_{A_1 \rightsquigarrow Z}$
$o^{pb^3}_{Z \rightsquigarrow A_k}$
$o^{tb^3}_{A_k \rightsquigarrow Z}$

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

100

100



# Obtaining three blocks (1)

All-to-finish operator

$A^-$
1..m
∀
$o^{\pi}_{Z \rightsquigarrow A_1}$
$o^{\tau}_{A_1 \rightsquigarrow Z}$
$Z^+$
1..m
$o^{\pi}_{Z \rightsquigarrow A_m}$
$o^{\tau}_{A_m \rightsquigarrow Z}$

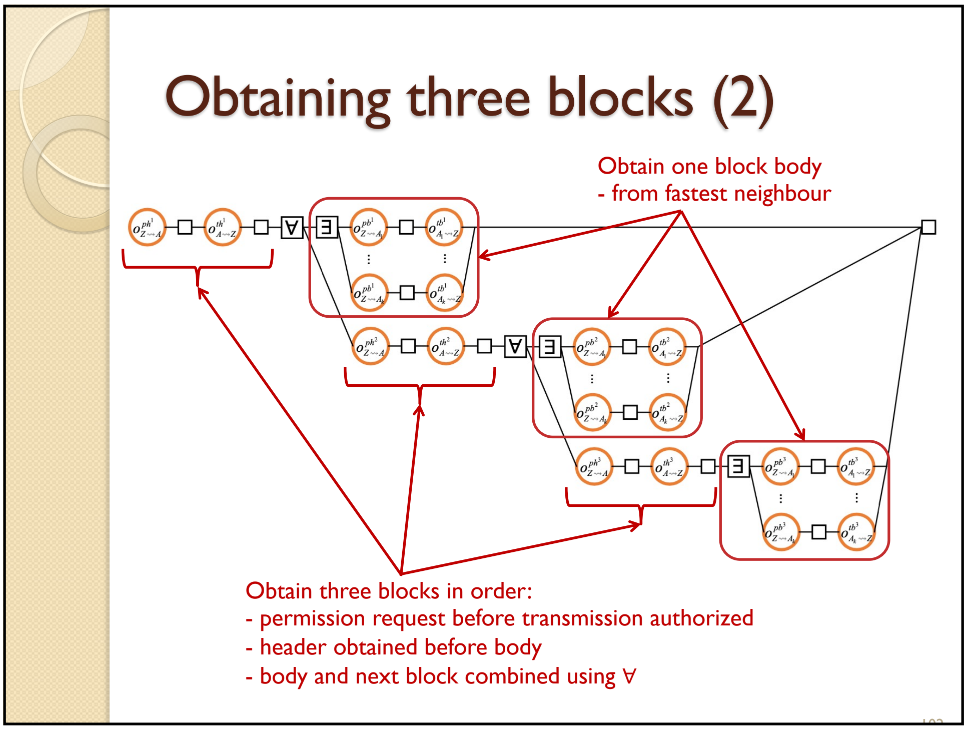
First-to-finish operator

∃
$o^{pb^i}_{Z \rightsquigarrow A_1}$
$o^{tb^i}_{A_1 \rightsquigarrow Z}$
$o^{pb^i}_{Z \rightsquigarrow A_k}$
$o^{tb^i}_{A_k \rightsquigarrow Z}$

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

101

101



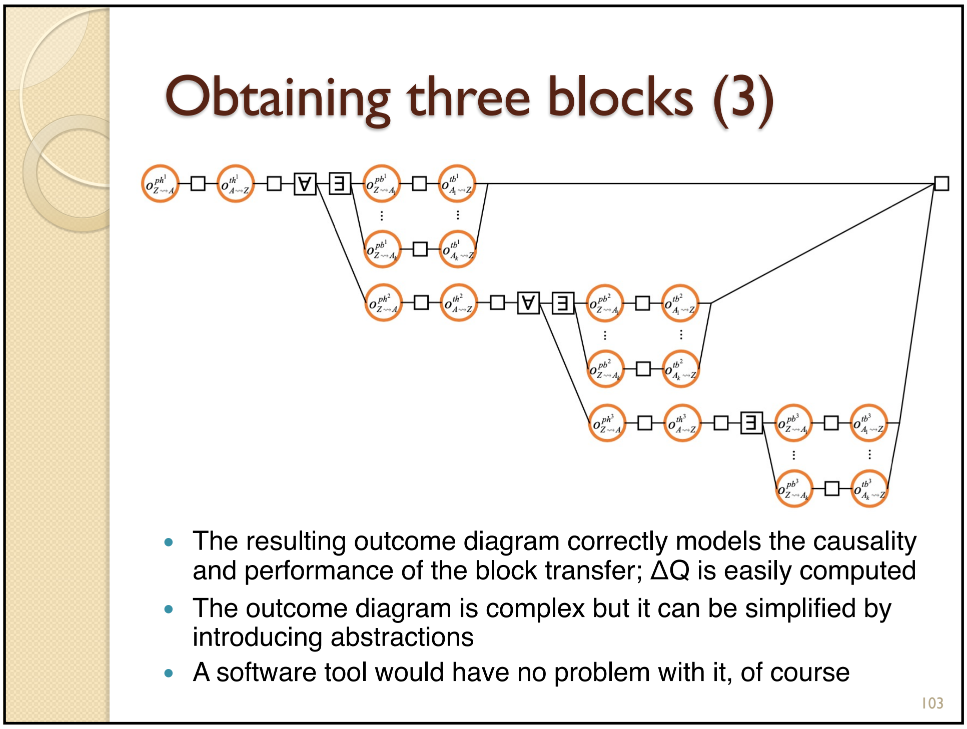
# Obtaining three blocks (2)

$o^{ph^1}_{Z \rightsquigarrow A}$
$o^{th^1}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^1}_{Z \rightsquigarrow A_1}$
$o^{tb^1}_{A_1 \rightsquigarrow Z}$
$o^{pb^1}_{Z \rightsquigarrow A_k}$
$o^{tb^1}_{A_k \rightsquigarrow Z}$
$o^{ph^2}_{Z \rightsquigarrow A}$
$o^{th^2}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^2}_{Z \rightsquigarrow A_1}$
$o^{tb^2}_{A_1 \rightsquigarrow Z}$
$o^{pb^2}_{Z \rightsquigarrow A_k}$
$o^{tb^2}_{A_k \rightsquigarrow Z}$
$o^{ph^3}_{Z \rightsquigarrow A}$
$o^{th^3}_{A \rightsquigarrow Z}$
∃
$o^{pb^3}_{Z \rightsquigarrow A_1}$
$o^{tb^3}_{A_1 \rightsquigarrow Z}$
$o^{pb^3}_{Z \rightsquigarrow A_k}$
$o^{tb^3}_{A_k \rightsquigarrow Z}$

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 ∀

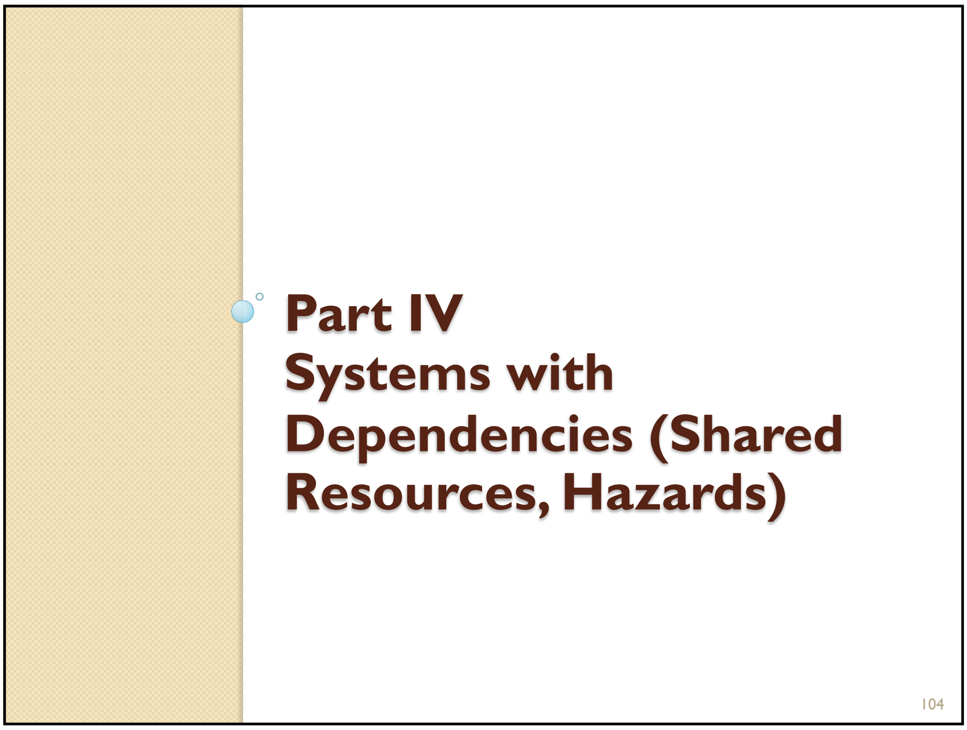
102



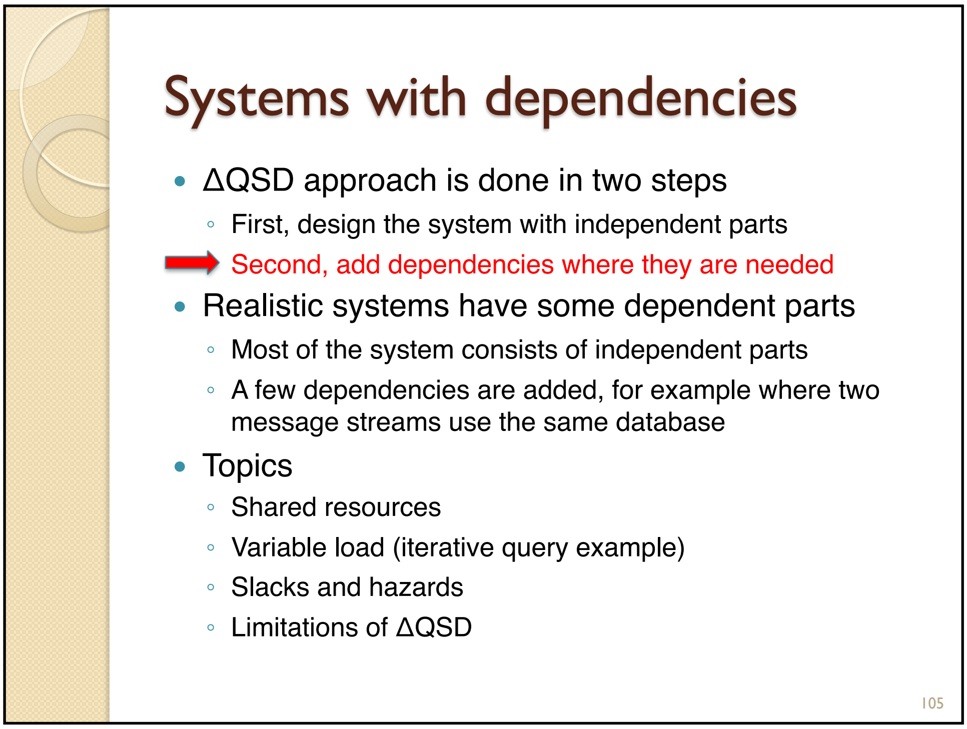
## Obtaining three blocks (3)

$o^{ph^1}_{Z \rightsquigarrow A}$
$o^{th^1}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^1}_{Z \rightsquigarrow A_1}$
$o^{tb^1}_{A_1 \rightsquigarrow Z}$
$o^{pb^1}_{Z \rightsquigarrow A_k}$
$o^{tb^1}_{A_k \rightsquigarrow Z}$
$o^{ph^2}_{Z \rightsquigarrow A}$
$o^{th^2}_{A \rightsquigarrow Z}$
∀
∃
$o^{pb^2}_{Z \rightsquigarrow A_1}$
$o^{tb^2}_{A_1 \rightsquigarrow Z}$
$o^{pb^2}_{Z \rightsquigarrow A_k}$
$o^{tb^2}_{A_k \rightsquigarrow Z}$
$o^{ph^3}_{Z \rightsquigarrow A}$
$o^{th^3}_{A \rightsquigarrow Z}$
∃
$o^{pb^3}_{Z \rightsquigarrow A_1}$
$o^{tb^3}_{A_1 \rightsquigarrow Z}$
$o^{pb^3}_{Z \rightsquigarrow A_k}$
$o^{tb^3}_{A_k \rightsquigarrow Z}$

- 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



# Part IV Systems with Dependencies (Shared Resources, Hazards)

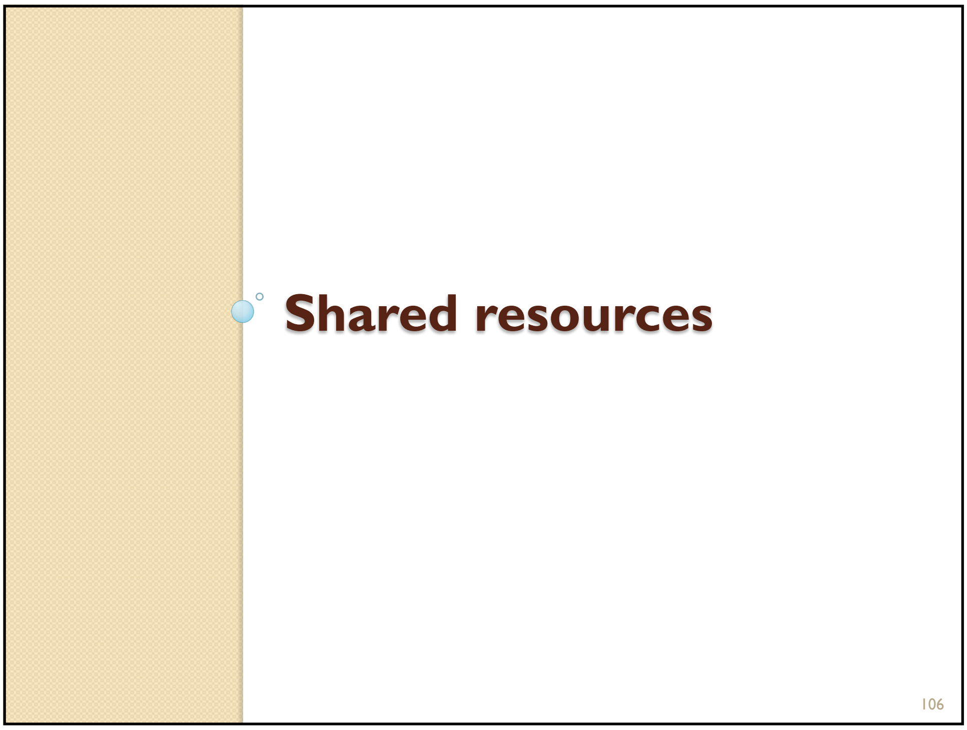


## Systems with dependencies

- ΔQSD approach is done in two steps
  - First, design the system with independent parts
  - ➡ Second, add dependencies where they are needed
- 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
  - Shared resources
  - Variable load (iterative query example)
  - Slacks and hazards
  - Limitations of ΔQSD

105

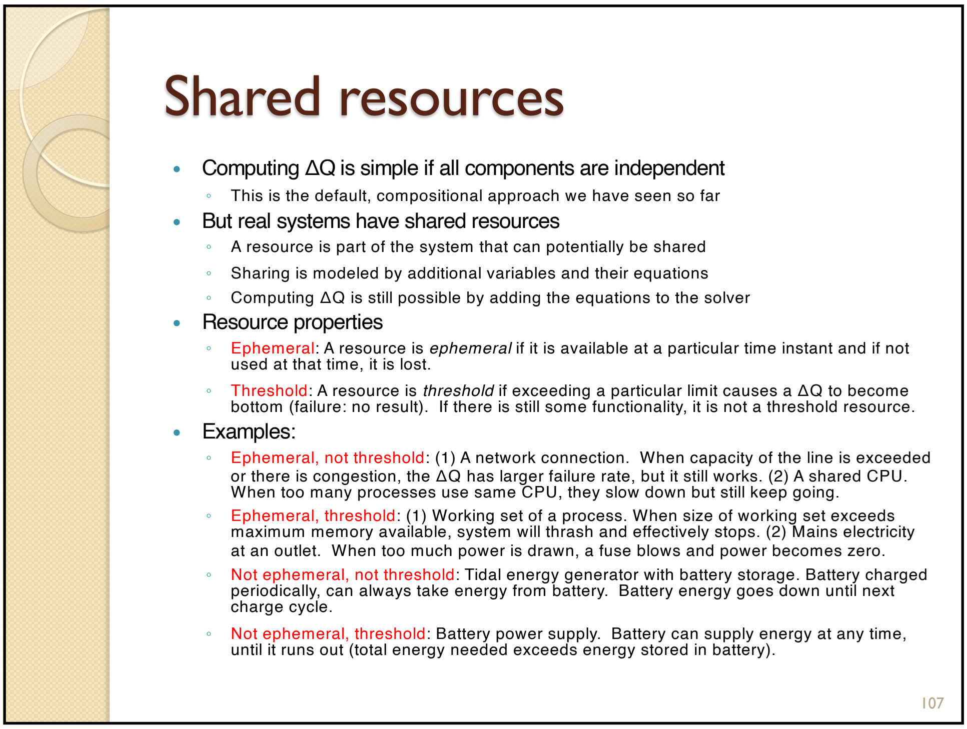
105



## Shared resources

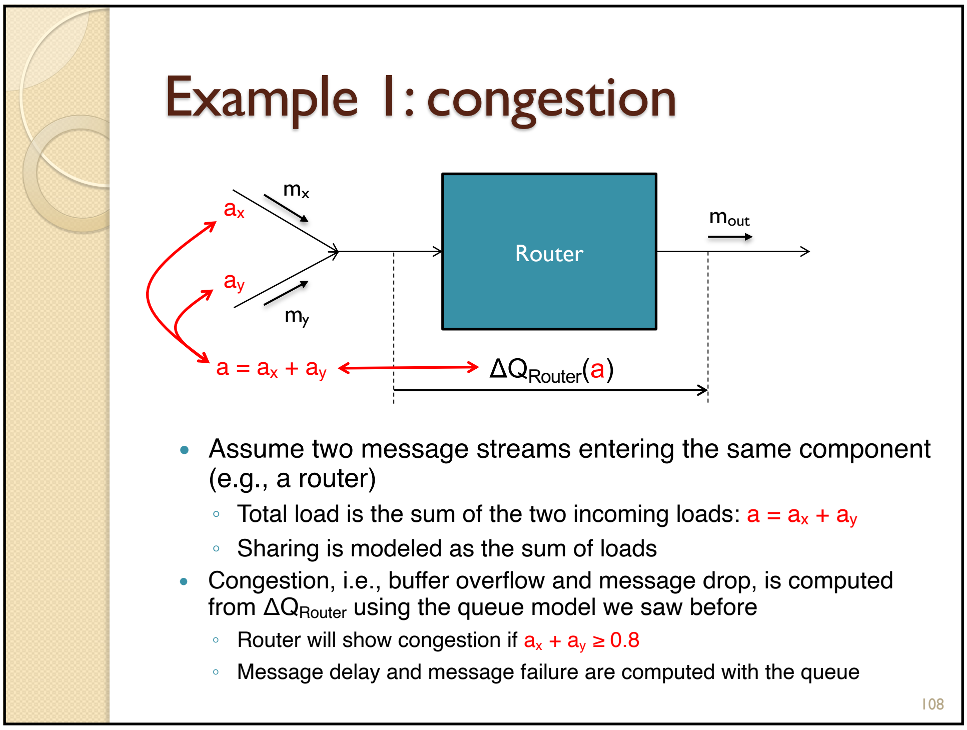
106

106



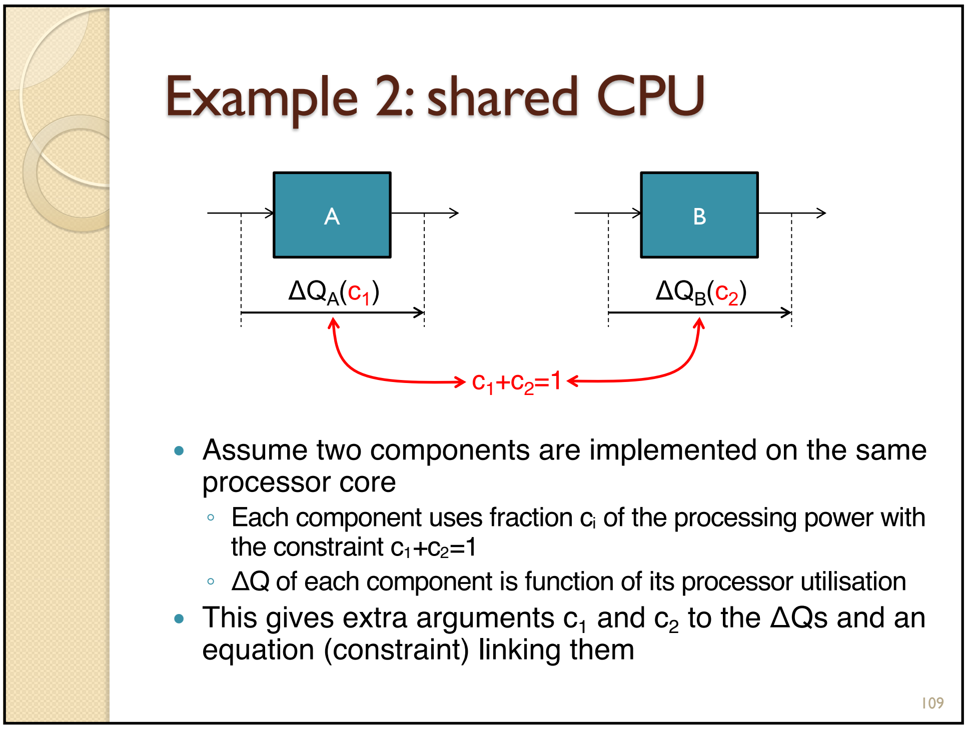
# Shared resources

- Computing ΔQ is simple if all components are independent
  - This is the default, compositional approach we have seen so far
- But real systems have shared resources
  - A resource is part of the system that can potentially be shared
  - Sharing is modeled by additional variables and their equations
  - Computing ΔQ is still possible 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 ΔQ to become bottom (failure: no result). If there is still some functionality, it is not a threshold resource.
- Examples:
  - 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 fuse blows and power becomes zero.
  - Not ephemeral, not threshold: Tidal energy generator with battery storage. Battery 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 $\Delta Q_{Router}$ 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



## Example 2: shared CPU

A

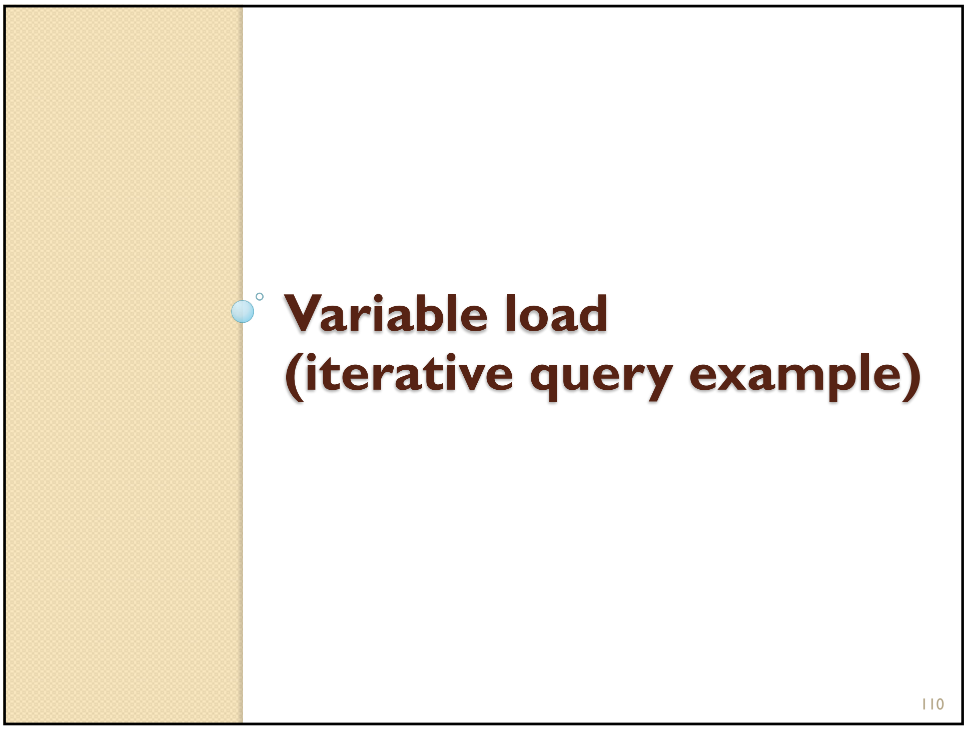
B

$\Delta Q_A(c_1)$

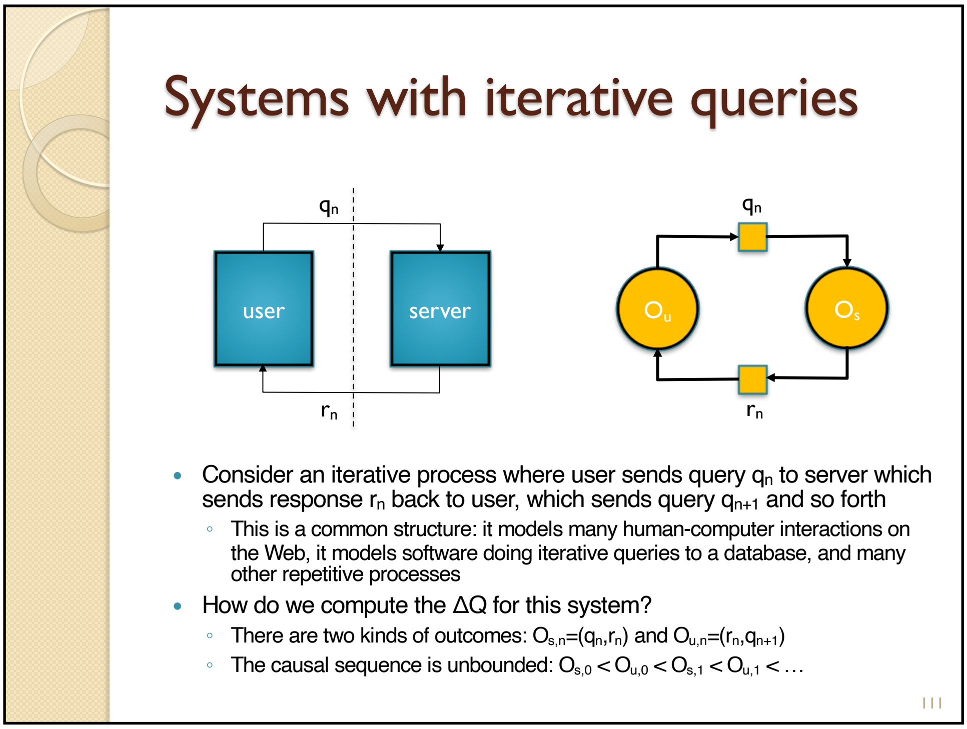
$\Delta Q_B(c_2)$

$c_1+c_2=1$

- 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



## Variable load (iterative query example)



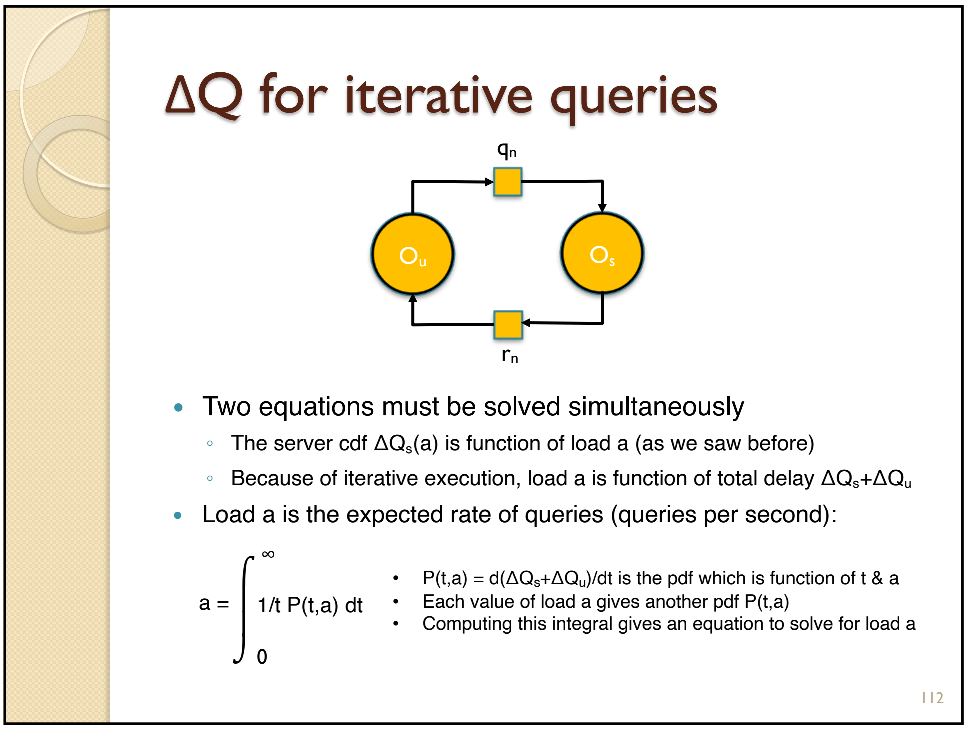
# Systems with iterative queries

$q_n$
user
server
$r_n$
$q_n$
$O_u$
$O_s$
$r_n$

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

111

111



# ΔQ for iterative queries

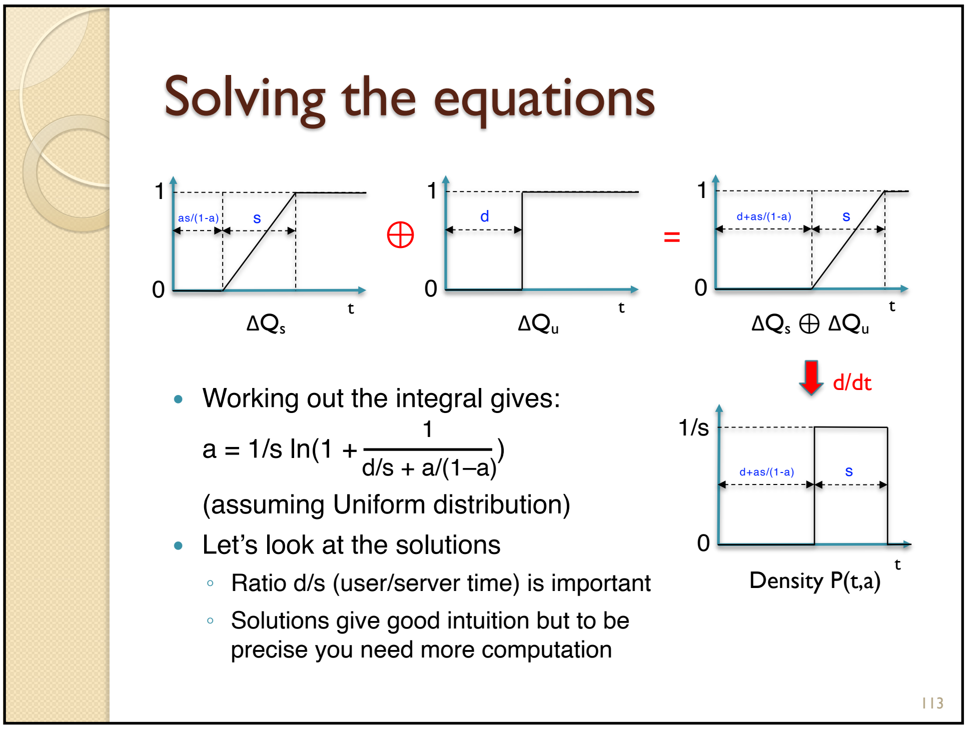
- Two equations must be solved simultaneously
  - The server cdf $\Delta Q_s(a)$ is function of load a (as we saw before)
  - Because of iterative execution, load a is function of total delay $\Delta Q_s+\Delta Q_u$
- Load a is the expected rate of queries (queries per second):

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

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

112

112



## Solving the equations

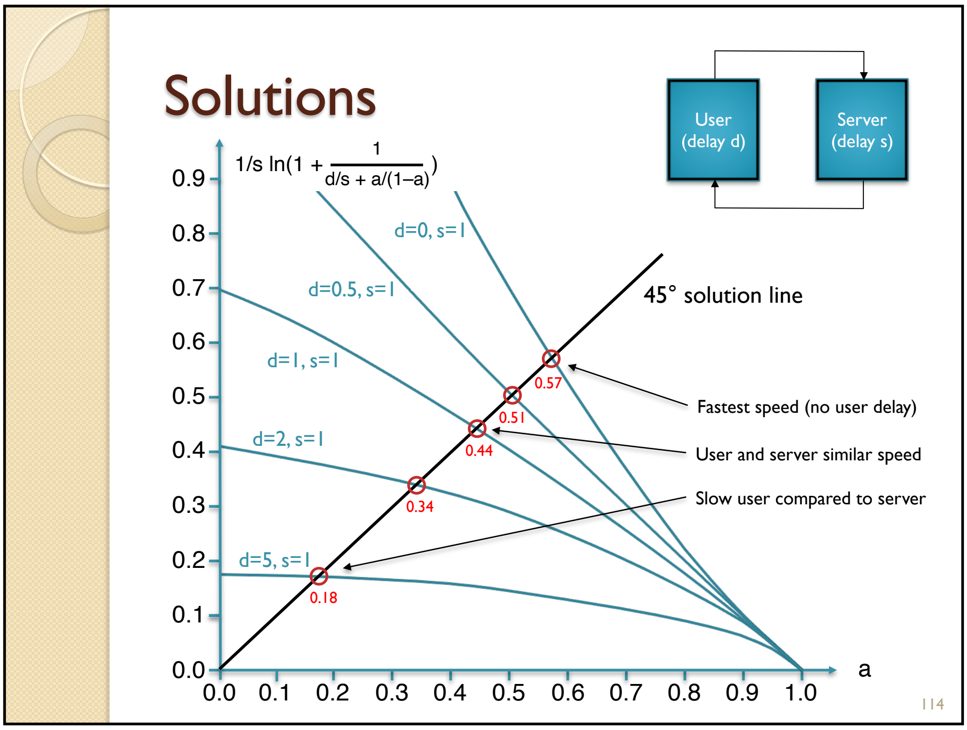
- Working out the integral gives:

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

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

113

113

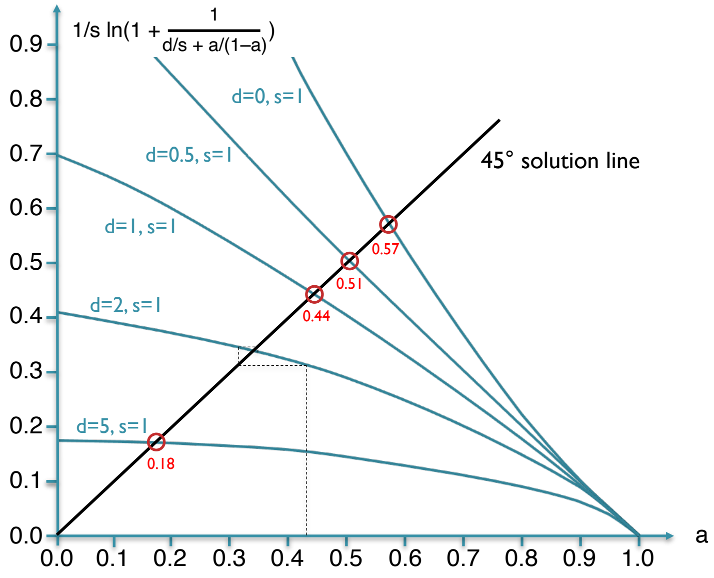


## Solutions

114

114

# Solutions

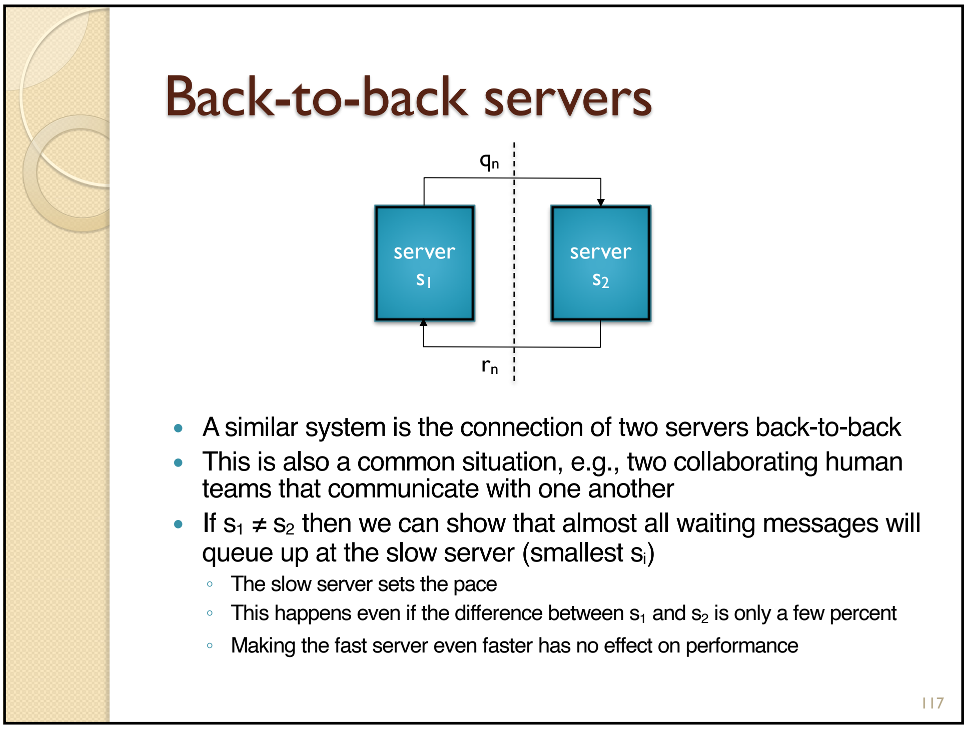


115

# 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

116



## Back-to-back servers

$q_n$
server $s_1$
server $s_2$
$r_n$

- 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

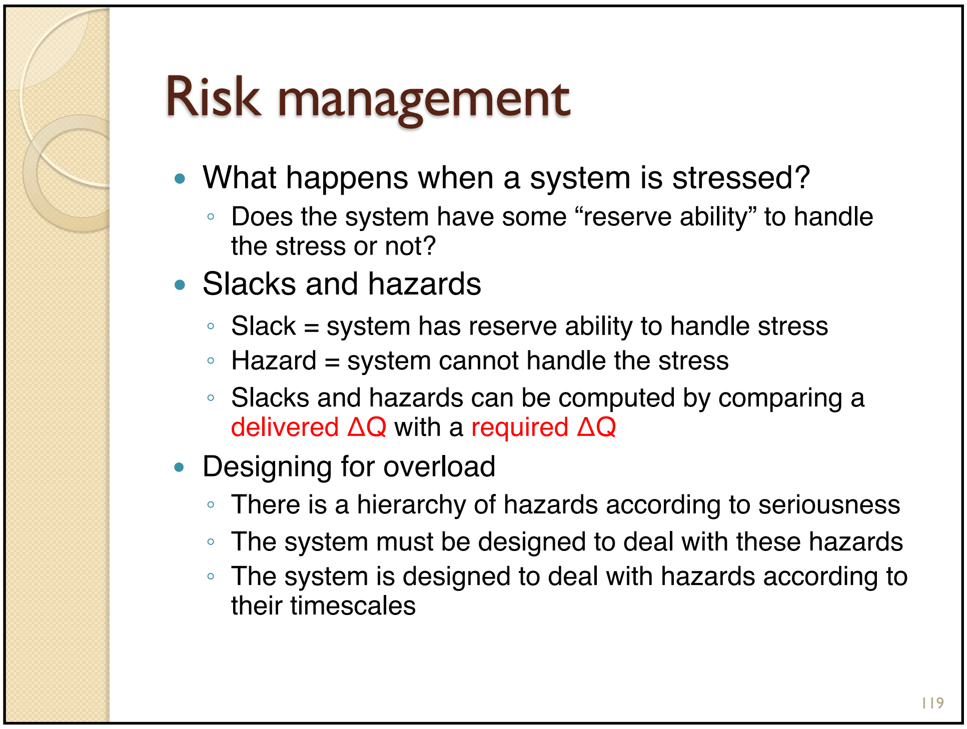
117

117

## Risk management

118

118

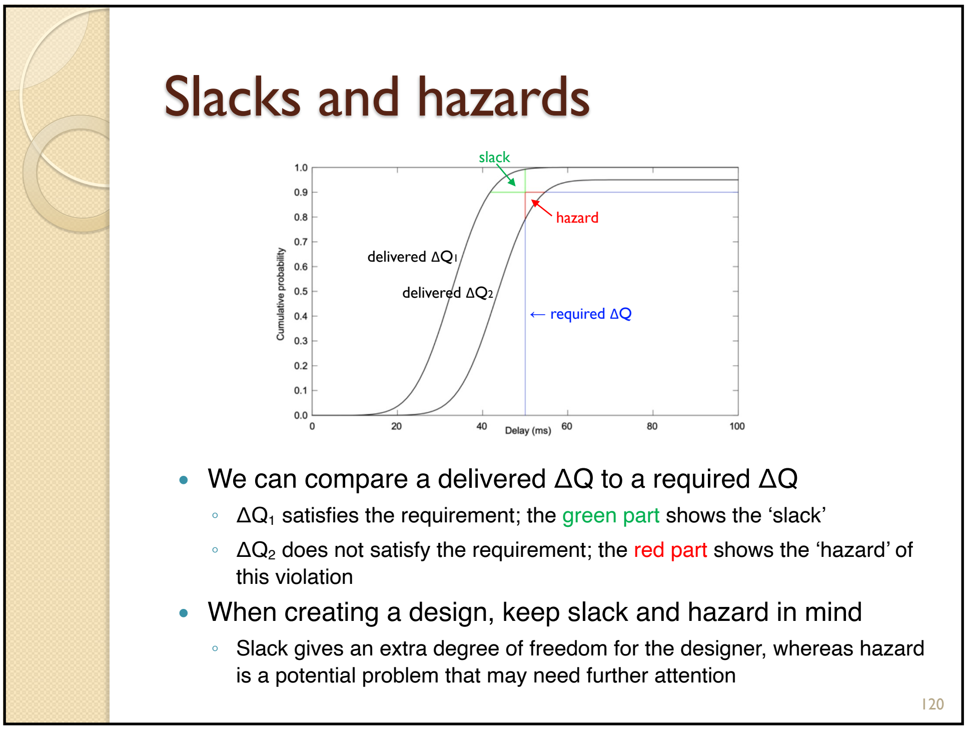


# Risk management

- What happens when a system is stressed?
  - Does the system have some “reserve ability” to handle the stress or not?
- Slacks and hazards
  - Slack = system has reserve ability to handle stress
  - Hazard = system cannot handle the stress
  - Slacks and hazards can be computed by comparing a delivered ΔQ with a required ΔQ
- Designing for overload
  - There is a hierarchy of hazards according to seriousness
  - The system must be designed to deal with these hazards
  - The system is designed to deal with hazards according to their timescales

119

119

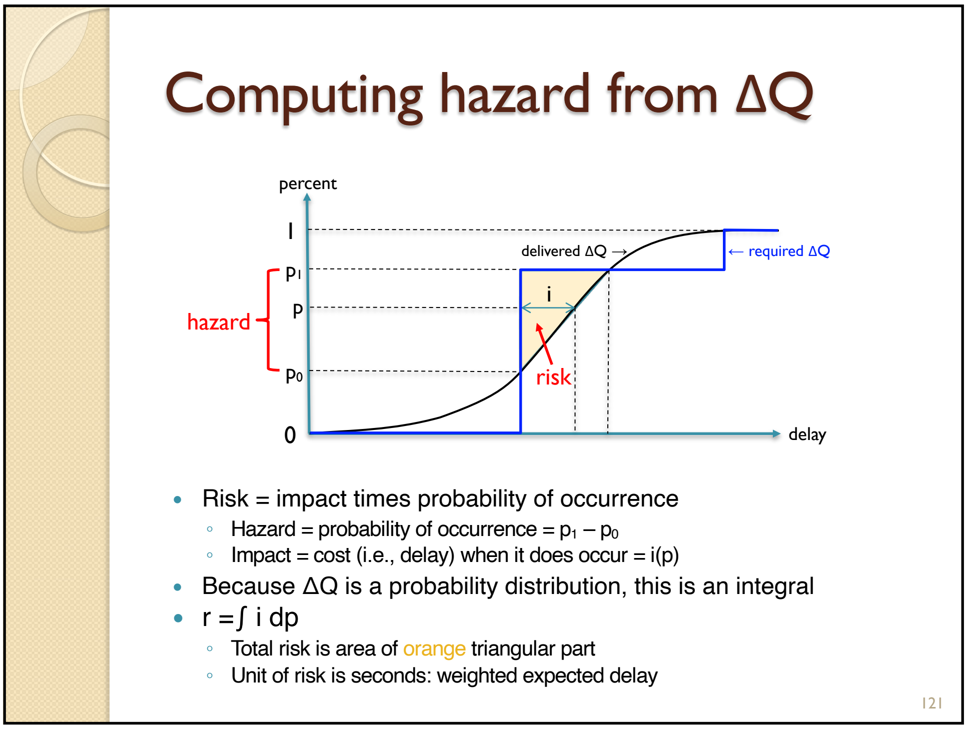


# Slacks and hazards

- We can compare a delivered ΔQ to a required ΔQ
  - $\Delta Q_1$ satisfies the requirement; the green part shows the ‘slack’
  - $\Delta Q_2$ does not satisfy the requirement; the red part shows the ‘hazard’ of this violation
- When creating a design, keep slack and hazard in mind
  - Slack gives an extra degree of freedom for the designer, whereas hazard is a potential problem that may need further attention

120

120

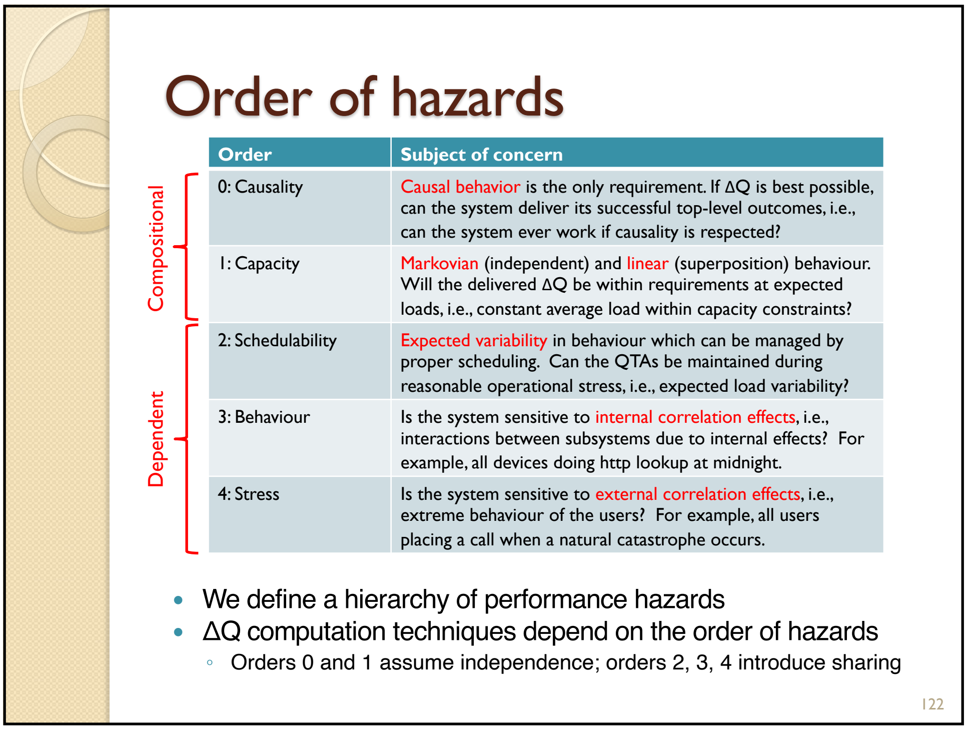


# Computing hazard from ΔQ

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

121

121



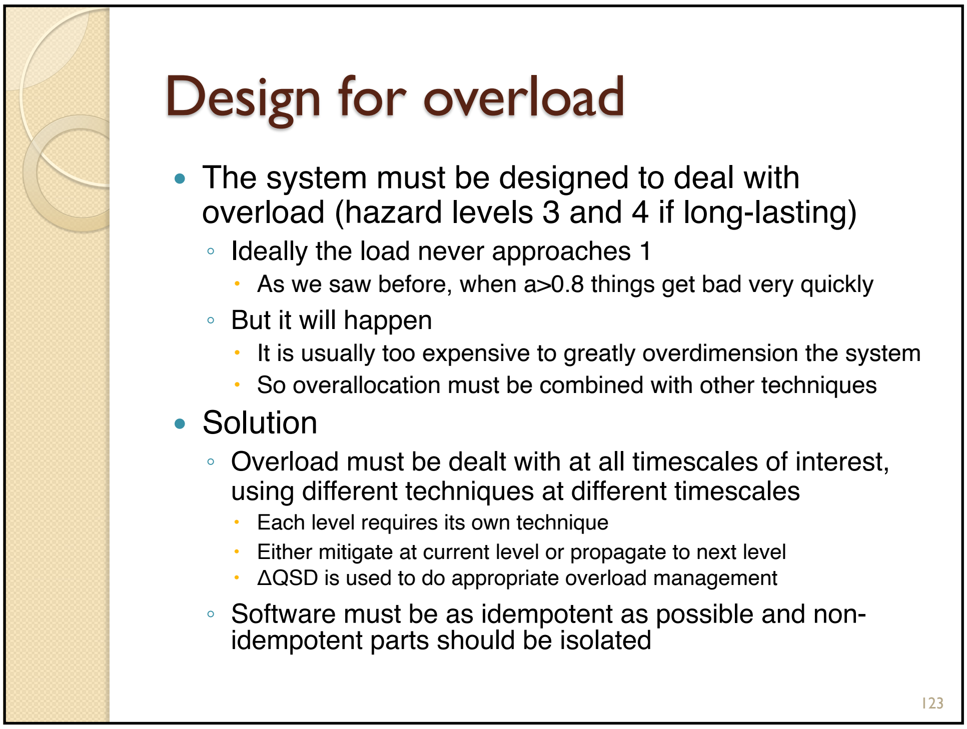
# Order of hazards

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

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

122

122

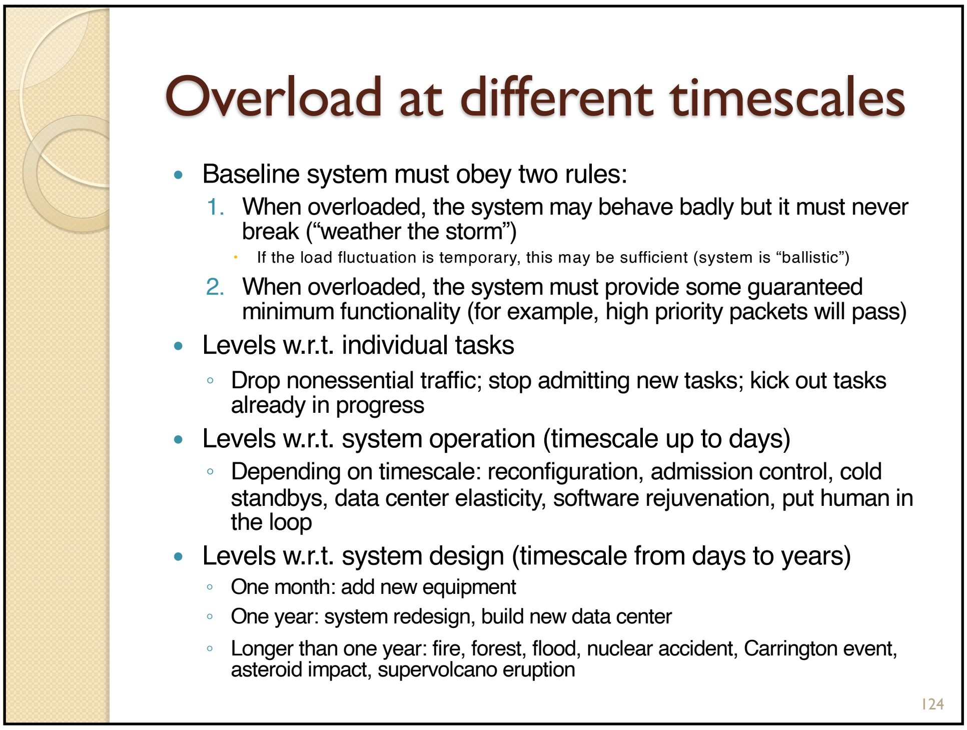


## Design for overload

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

123

123

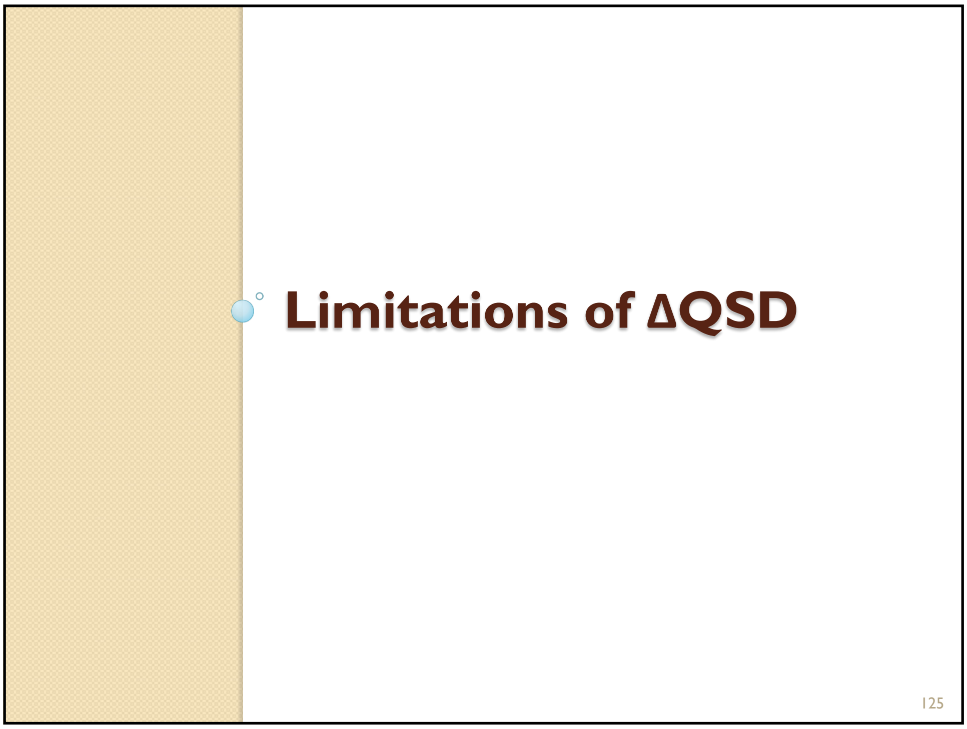


## Overload at different timescales

- Baseline system must 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 (system is “ballistic”)
  2. When overloaded, the system must provide some guaranteed minimum functionality (for example, high priority packets will pass)
- Levels w.r.t. individual tasks
  - Drop nonessential traffic; stop admitting new tasks; kick out tasks already in progress
- Levels w.r.t. system operation (timescale up to days)
  - Depending on timescale: reconfiguration, admission control, cold standbys, data center elasticity, software rejuvenation, put human in the loop
- Levels w.r.t. system design (timescale from days to years)
  - One month: add new equipment
  - One year: system redesign, build new data center
  - Longer than one year: fire, forest, flood, nuclear accident, Carrington event, asteroid impact, supervolcano eruption

124

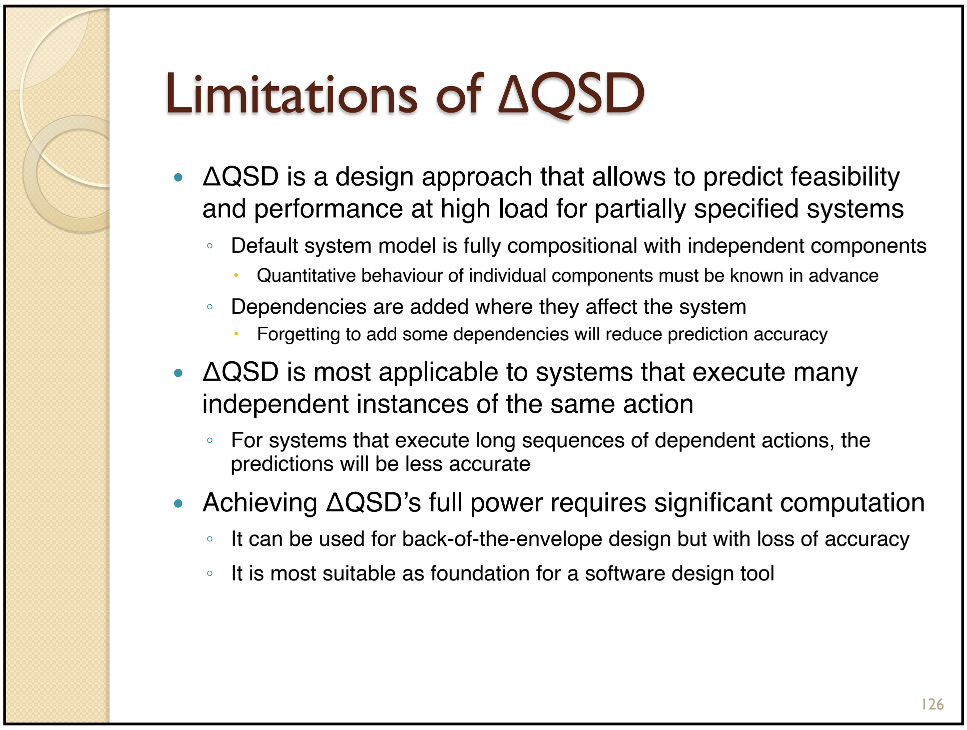
124



# Limitations of ΔQSD

125

125

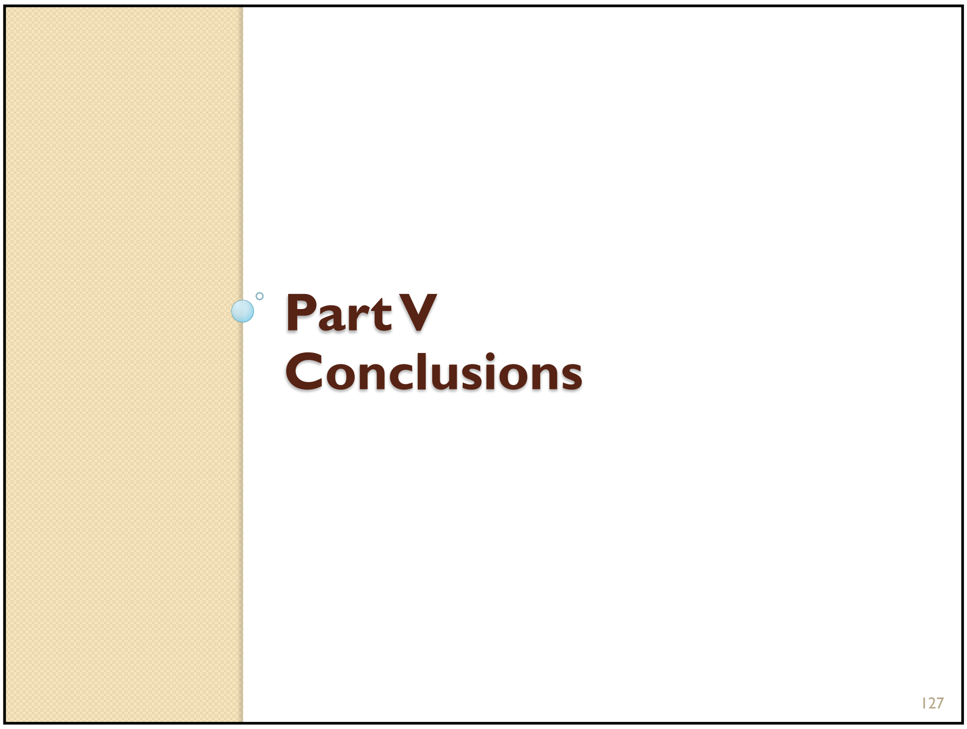


## Limitations of ΔQSD

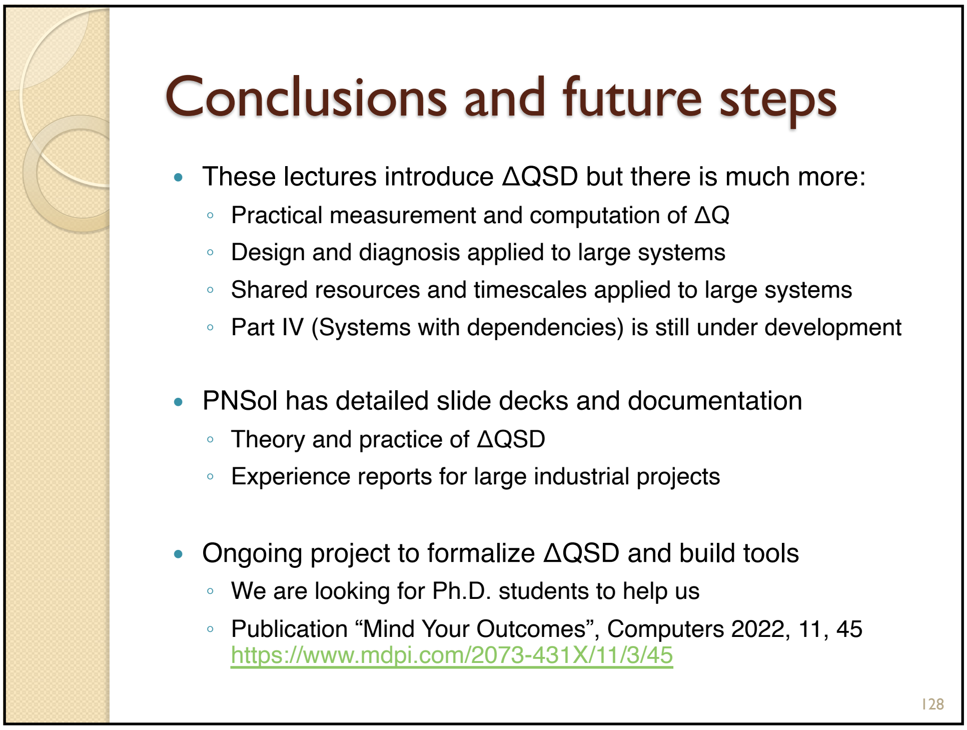
- ΔQSD is a design approach that allows to predict feasibility and performance at high load for partially specified systems
  - Default system model is fully compositional with independent components
    - Quantitative behaviour of individual components must be known in advance
  - Dependencies are added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

126

126



# Part V Conclusions



## Conclusions and future steps

- These lectures introduce ΔQSD but there is much more:
  - Practical measurement and computation of ΔQ
  - Design and diagnosis applied to large systems
  - Shared resources and timescales applied to large systems
  - Part IV (Systems with dependencies) is still under development
- PNSol has detailed slide decks and documentation
  - Theory and practice of ΔQSD
  - Experience reports for large industrial projects
- Ongoing project to formalize ΔQSD and build tools
  - We are looking for Ph.D. students to help us
  - Publication “Mind Your Outcomes”, Computers 2022, 11, 45 https://www.mdpi.com/2073-431X/11/3/45