UCLouvain

icteam

The Essence of Scalability

March 6, 2024

Peter Van Roy
ICTEAM Institute
Université catholique de Louvain

PREDICTABLE NETWORK SOLUTIONS

Overview

- Context
  - ΔQSD paradigm for system design
  - Collaboration with PNSol Ltd
- System design for scalability
  - What is scalability?
  - Two great frauds in system design
  - A simple system model
- Designing for overload
  - Temporary overload
  - Permanent overload
- Beyond inductive reasoning
- Conclusions

Context

Background

- My background (www.info.ucl.ac.be/~pvr/pldc.html)
  - I am a computer scientist who is interested in programming languages and distributed computing. Both of these topics are stepping stones to the main problem that interests me, namely how to design and understand large systems.

- PNSol (www.pnsol.com)
  - Predictable Network Solutions Ltd (PNSol) specializes in system performance of large-scale distributed systems
  - This talk is dedicated to the people at PNSol, in particular Neil Davies and Peter Thompson, who have spent their lives slaving behind NDAs to make sure that the Internet infrastructure keeps working

ΔQSD paradigm

- This talk is based on the ΔQSD paradigm for system design
  - It has been developed over 30 years by a small group of people around Predictable Network Solutions Ltd.
  - It is widely used and has been validated in large industrial projects, with large cumulative savings in project costs
  - This talk is part of a collaborative project to make ΔQSD available to the wider system design community

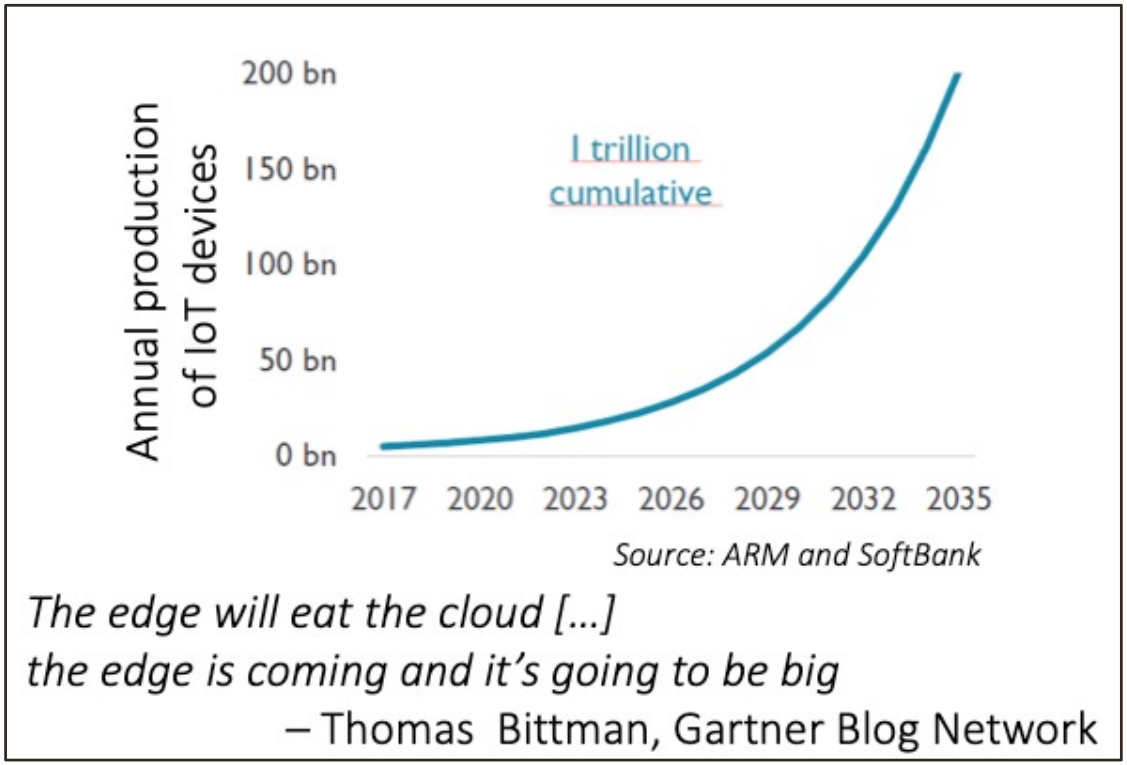
- ΔQSD targets systems with many independent users where performance and reliability are important
  - ΔQSD targets systems with large flows of data items that are subject to unexpected overload situations
  - Examples are large-scale communications networks, large-scale client/servers, cloud analytics platforms, distributed sensor networks, and cryptocurrency platforms
  - ΔQSD is a stochastic and compositional approach for system design that can predict performance (latency and throughput) and feasibility at high system load during the design pcoess

System design for scalability

What is scalability?

- A system is **scalable** if it is able to handle growing amounts of work in an acceptable manner
  - Desired system properties (such as performance and reliability) are "acceptable" functions of the system size *n*
- One way to achieve scalability is to add resources to the system in an appropriate manner to achieve the desired properties as the amount of work grows
  - Aggregate performance can increase by using parallel computing
  - Reliability can remain high by using redundancy
- Achieving scalability becomes more difficult as scale increases
  - The system must be able to handle increasing stress
  - New and unexpected problems appear at large scales
  - As the Internet grows, scalability continues to be an important issue

Exponential growth of ICT infrastructure



David Bol, RISC-V (Paris), Oct. 1-3, 2019

- Exponential growth of ICT (Information and Communications Technology) infrastructure, starting with the Internet, is the motor for scalability
- The Internet has been growing exponentially since its conception in 1970
- Nowadays, the main growth is concentrated at the edge

To boldly go where no system has gone before…

- Scalable systems must support parallel execution for performance and redundancy for fault tolerance
  - These are necessary conditions, but they are not sufficient
  - A scalable system, simply because it is scalable, will often be pushed to the edge, beyond its capacity to do work
- Scalable systems must therefore support overload
  - Handling overload is essential, yet it is rarely discussed
  - We will see that there are two fundamentally different kinds of overload: temporary overload and permanent overload
- How can we design for overload?
  - The two main tools, induction and probability, lose most of their power
  - We must use principles that are independent of induction
  - Experience shows that when a system goes beyond what has been done before, new problems and new behaviors appear

The two great frauds

Two great frauds in system design

- There are two great frauds that must be exposed!

- Systems obey inductive reasoning – *false!*
  - Past experience is a bad guide for future systems, especially if the future system is going beyond the past one

  ⇒ Black swans

- Systems obey probability distributions – *false!*
  - Probability distributions are introduced to simplify analysis, but they often do not exist in reality
  - Assuming that a probability distribution exists is a very strong assumption (frequency limit exists) and is very probably wrong

  ⇒ Dijkstra's demon

Dijkstra's demon

- Dijkstra's demon continues to haunt computer systems
  - Unlike Maxwell's demon, it cannot be exorcised
- Consider the guarded command (generalized **if** statement):

$$\textbf{if } B_1 \rightarrow S_1 \textbf{ [] } B_2 \rightarrow S_2 \textbf{ [] } B_3 \rightarrow S_3 \textbf{ [] } \ldots \textbf{ [] } B_n \rightarrow S_n \textbf{ fi}$$

- Each guard B_i is a boolean condition and S_i is a statement
  - Select any true guard and execute its statement
  - This is a nondeterministic choice!
- What can we assume about the selection – fairness? No!

> "The most effective way out is to assume the UM [Unbounded Machine] not equipped with an unbiased coin, but with a totally erratic daemon"
> – *A discipline of programming* (Edsger Dijkstra, 1976)

- Do not use probability to prove correctness
  - All paths must be verified, even extremely rare ones
  - Be very wary about using probability!
    - It can sometimes be used when combining independent events

Black swans

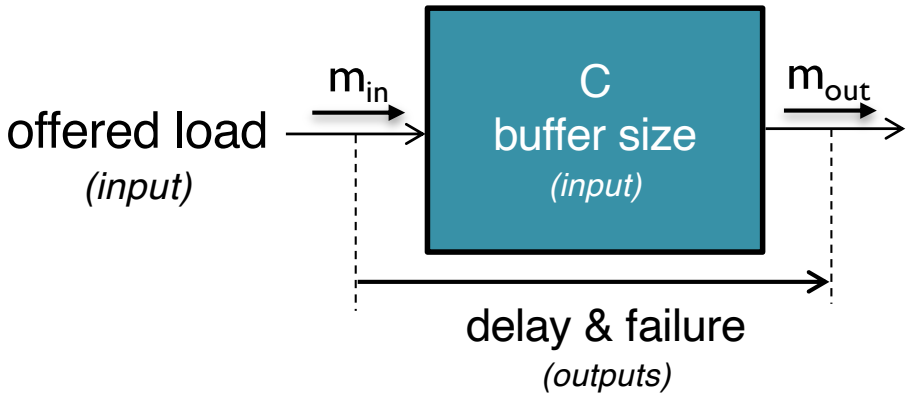
- Systems are designed by relying on induction
  - However, induction often has a built-in limit and fails beyond it
  - Year 2000 Bug: it was “far away” but it arrived (I was there!)
  - Dinosaurs and banks: “too big to fail” but they fail
- In computer systems this is both ubiquitous and hidden
  - All systems have finite resource limits (memory, speed) that are far away in “normal usage” but reached when system is stressed
  - Typically, the system will fail in exactly the case where it is needed most (“Red Wedding” situation)
- Black swan: an unexpected event that is obvious in hindsight
  - Large systems must be designed to survive unexpected events
  - See *The Black Swan* (Nassim Nicholas Taleb, 2010)
- We will see how to design systems to survive black swans

System model

A simple system model

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

A system as a component



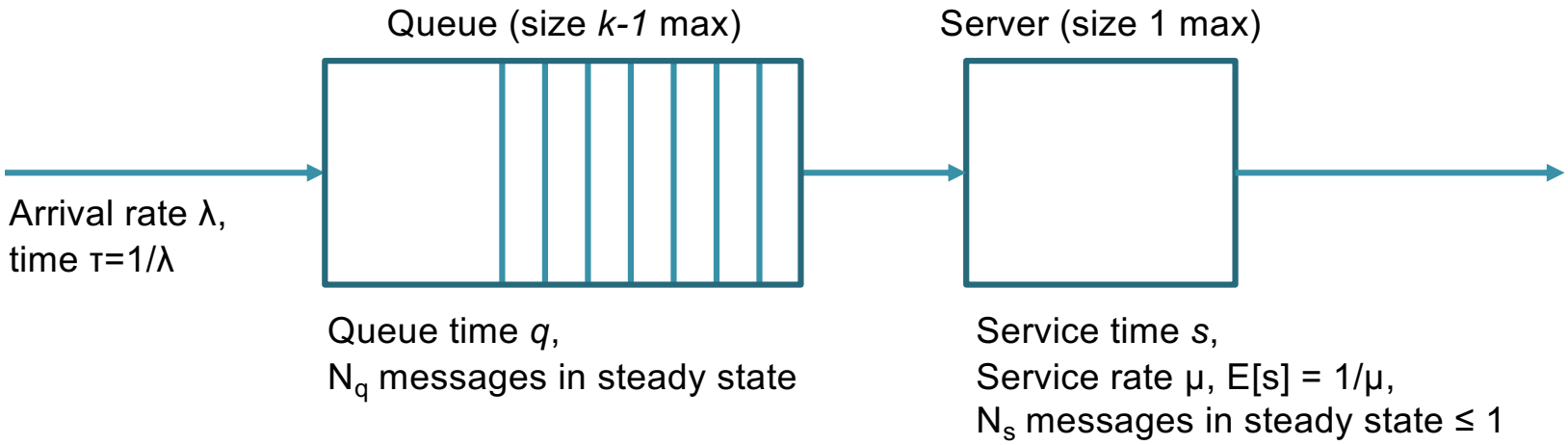
- We will model our system as a simple component
  - The component has a buffer (queue) and a server
- A typical component has four parameters of interest

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

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

- Delay and failure are function of load and buffer size

A component as M/M/1/K queue



- We model a component as an M/M/1/K queue
  - M: arriving messages have Exponential distribution with rate λ
  - M: service time has Exponential distribution with rate μ
  - 1: one message can be served at a time
  - *K*: total buffer size is *k* (buffer size = queue size *k-1* + server size 1)
- Offered load $a = \lambda/\mu$ (arrival rate / service rate, normalized arrival rate)
- The two knobs we control are offered load *a* and buffer size *k*
  - When the buffer is full, new arrivals are dropped (failure)

Temporary overload

Designing for overload

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

Capacity races

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

Design for temporary overload

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

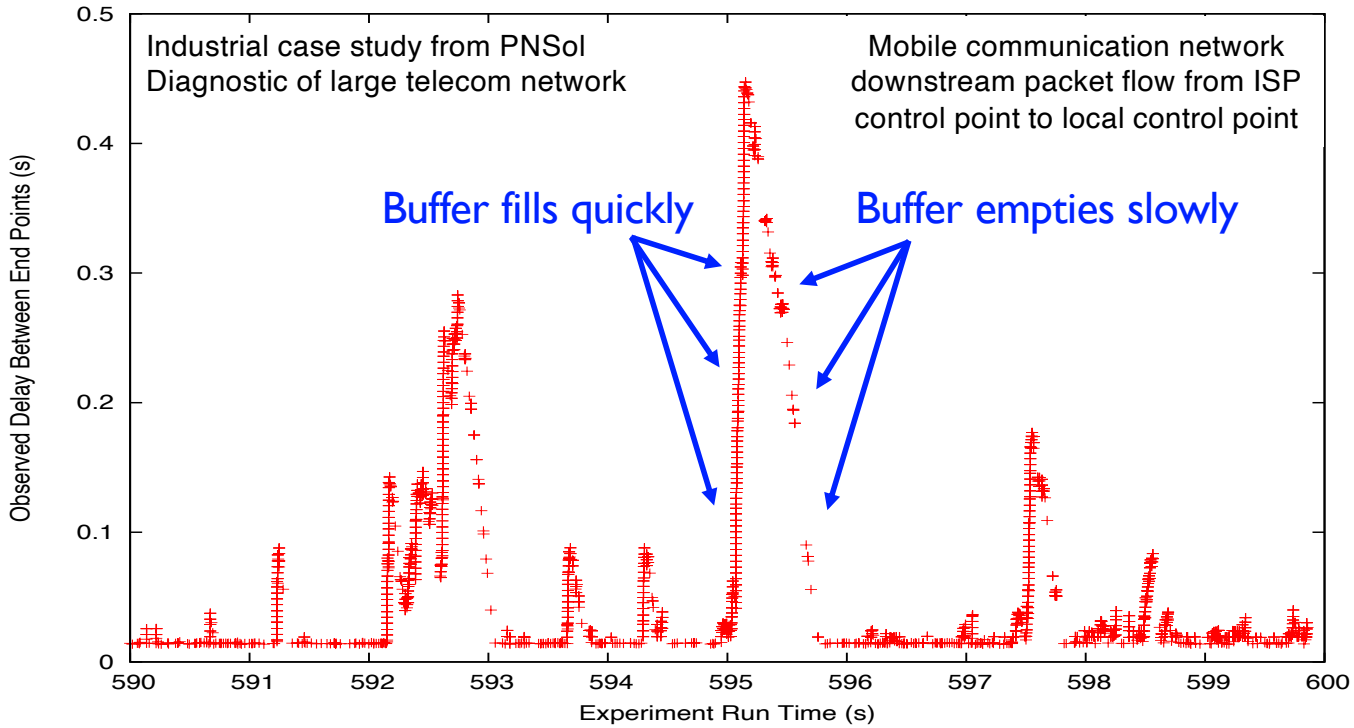
Effect of temporary overload

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

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

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

Long-lasting increase in delay



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

Tweaking load and buffer size

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

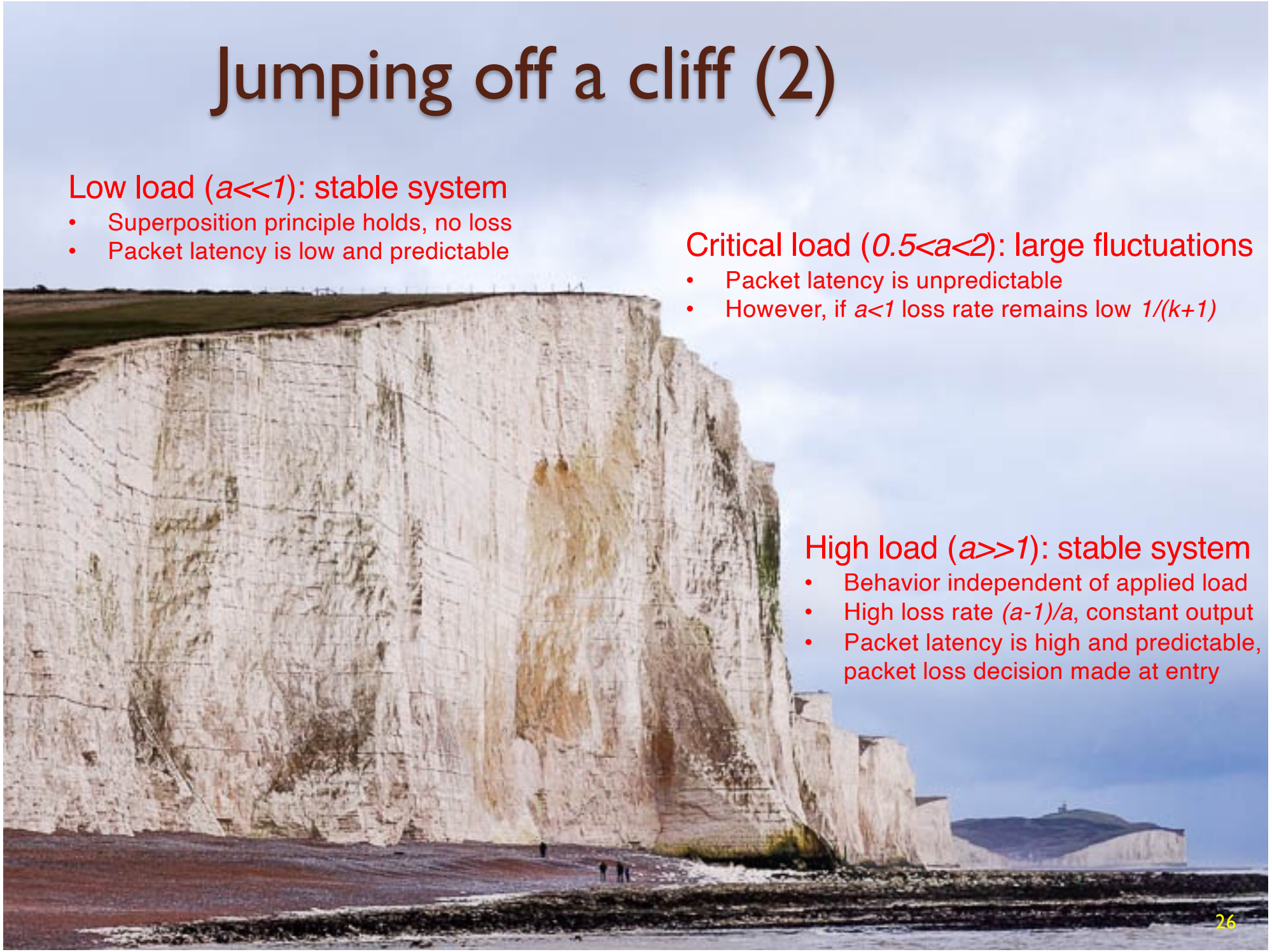
Jumping off a cliff (1)

- Let's take a closer look at what happens on overload
  - Main question: can we predict what will happen?
  - Let us look closer at our trusty M/M/1/K queue

- Maximum fluctuations happen at the cliff edge
  - Applied load *a* is normalized to 1 (offered load = service time)
  - Fluctuations highest when $a \approx 1$
  - System will be wild at the cliff boundary

- Strangely, the system is more stable when massively overloaded than when slightly overloaded
  - When $a \gg 1$, system has stable behavior
  - Output is constant (*1*), packet loss ($a-1$) is decided at entry

Jumping off a cliff (2)



Low load (*a<<1*): stable system

- Superposition principle holds, no loss
- Packet latency is low and predictable

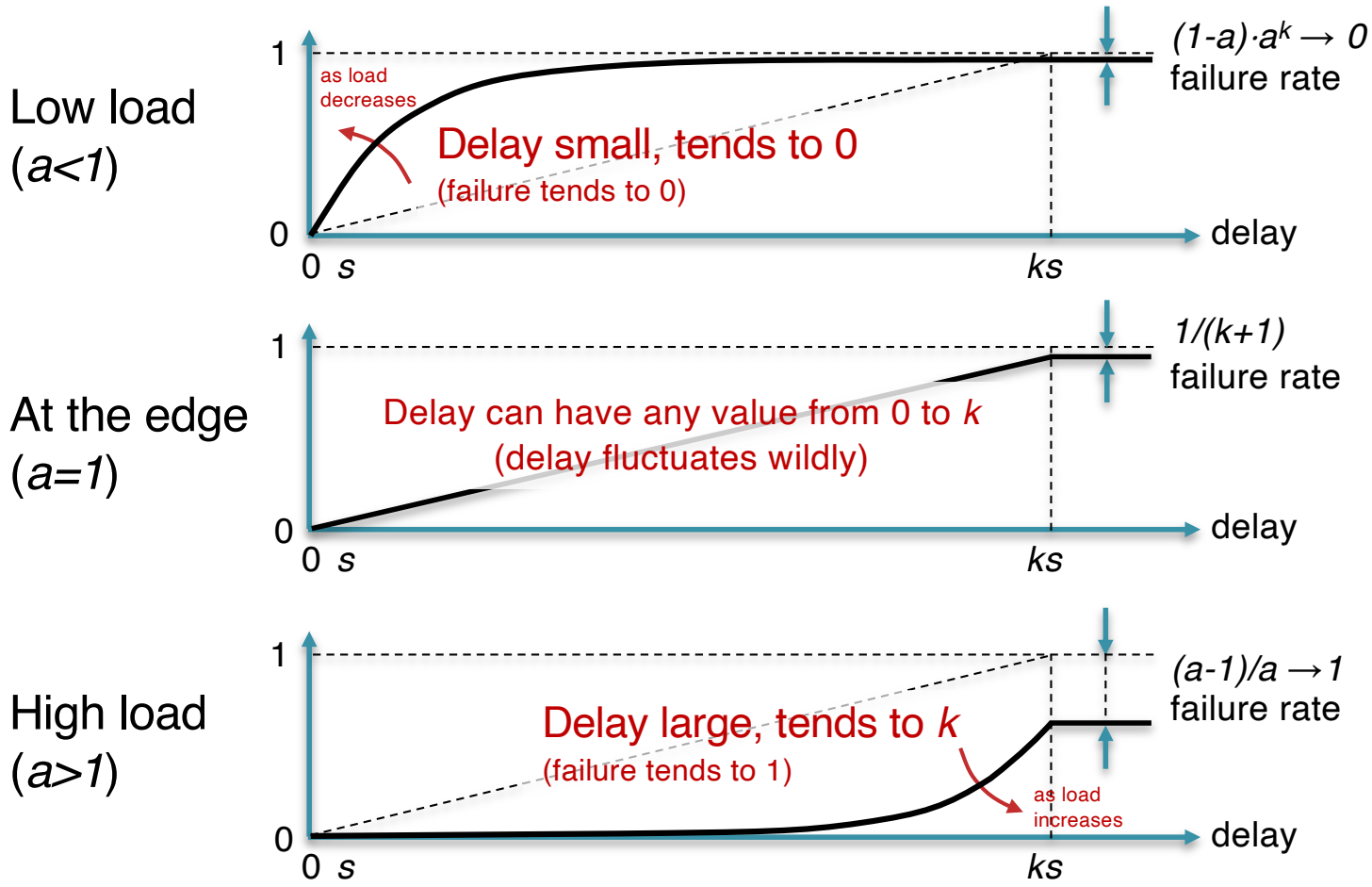
Critical load (*0.5<a<2*): large fluctuations

- Packet latency is unpredictable
- However, if *a<1* loss rate remains low *1/(k+1)*

High load (*a>>1*): stable system

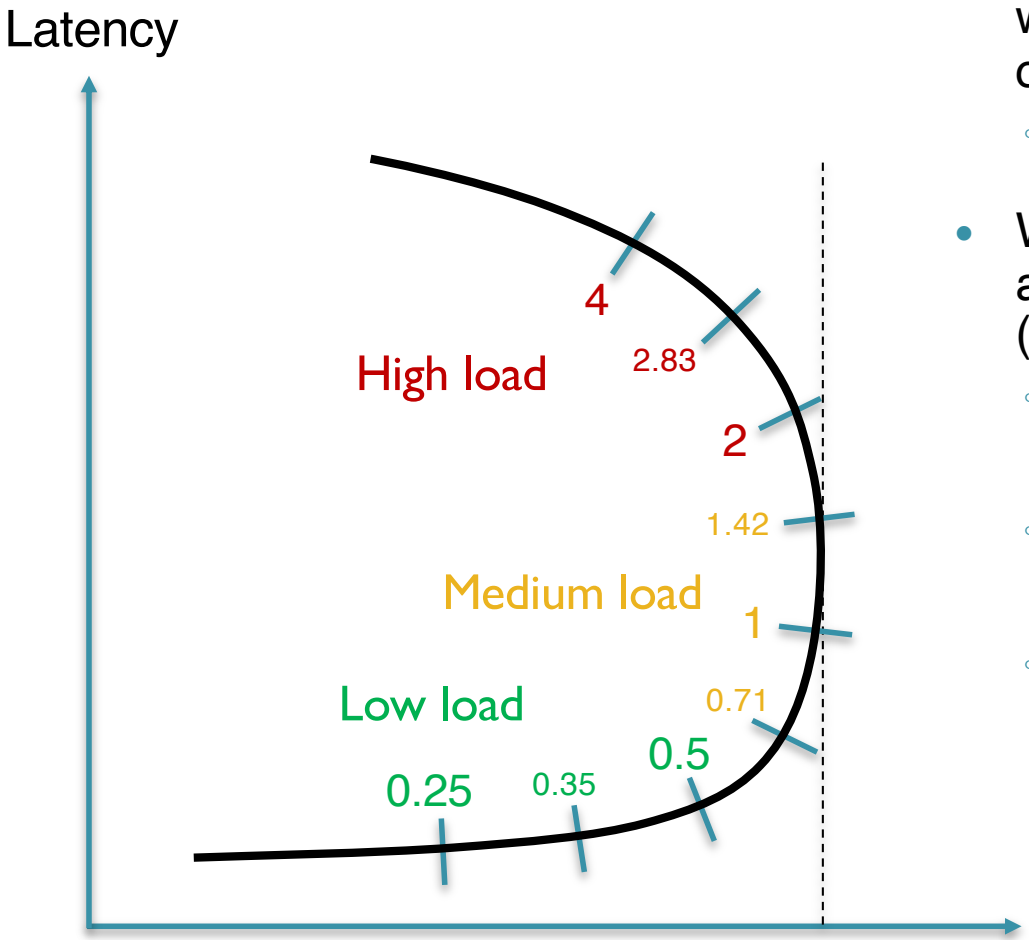
- Behavior independent of applied load
- High loss rate *(a-1)/a*, constant output
- Packet latency is high and predictable, packet loss decision made at entry

Theoretical analysis



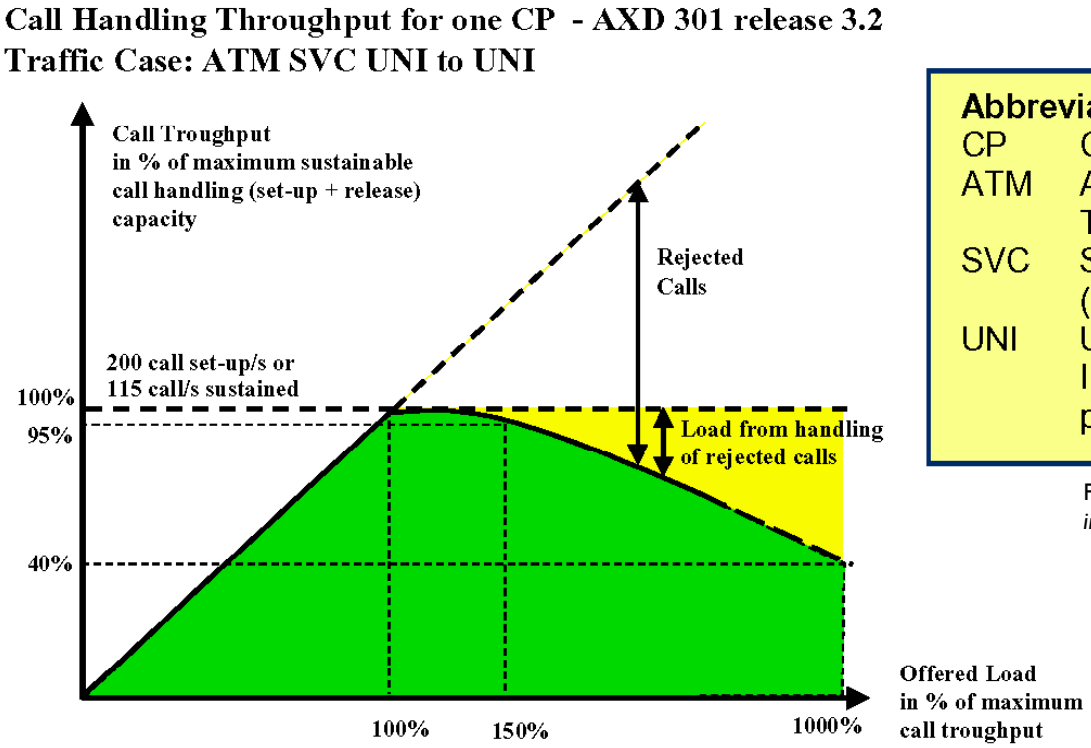
- We show cumulative distribution function (CDF) of system delay
- Theoretical analysis based on M/M/1/K queue
  - Offered load *a* (normalized to 1), buffer size *k*, service time *s*

Throughput-latency diagram



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

Example: AXD301 ATM switch



Abbreviations:

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

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

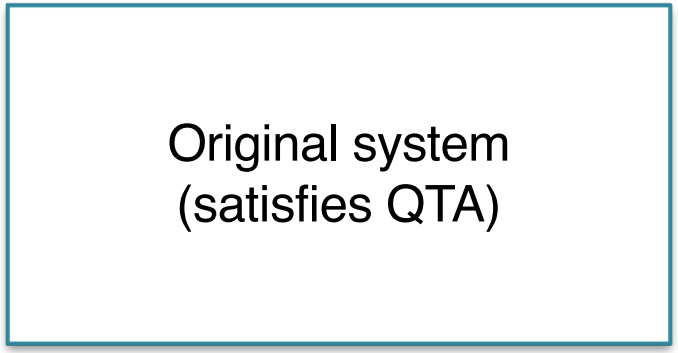
- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - Erlang/OTP at that time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

Permanent overload

Permanent overload

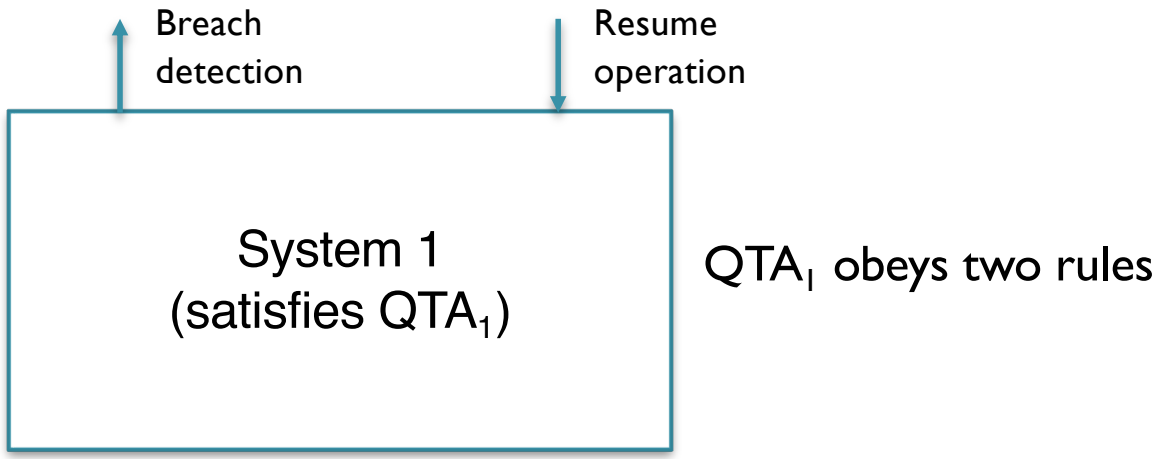
- If your system suffers overload too often (repeated or long-lasting), we say that it has a permanent overload
  - Permanent overload is generally handled by adding resources to the system, but this is never a permanent solution!
- Multilevel system
  - Permanent overload is always possible, so there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

Original system



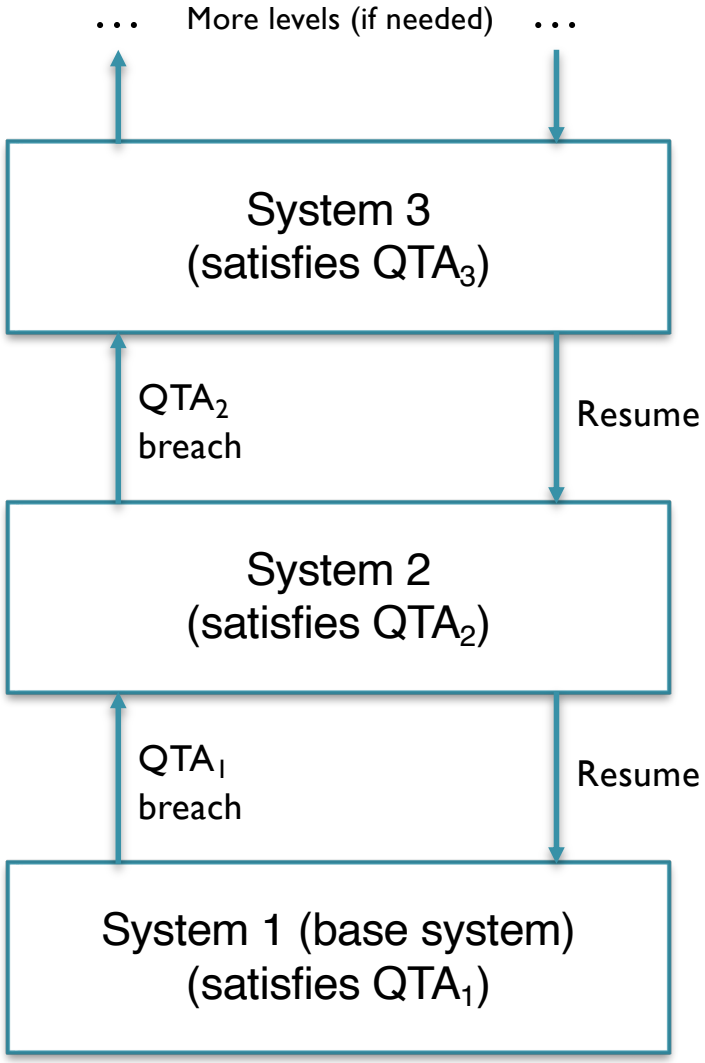
- Our starting point is a system that satisfies a QTA
  - The QTA (Quantitative Timeliness Agreement) specifies what the base system does and its limits
  - In the previous section we saw how this system behaves under temporary overload
- Under temporary overload, the QTA may no longer be satisfied
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

Lowest level in multilevel system



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

Multilevel system



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

Levels must be independent

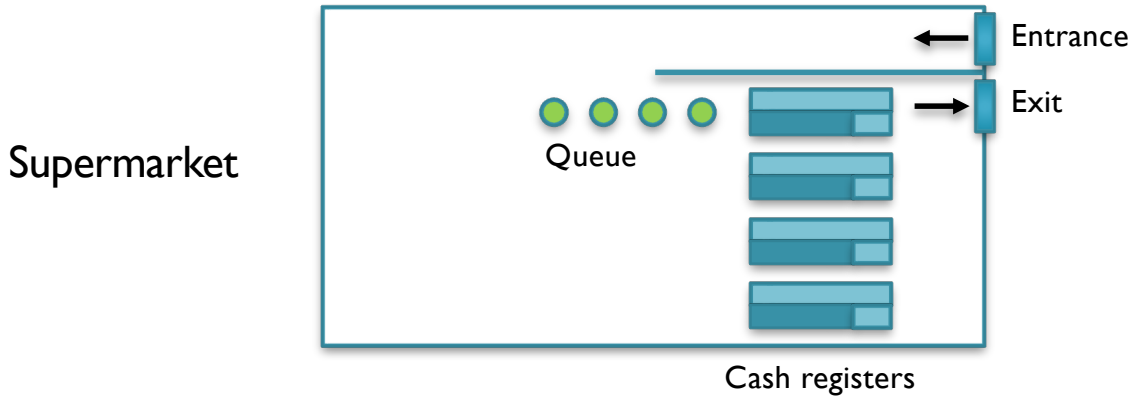
Leonid Rogozov lying down talking to his friend Yuri Vereschagin at Novolazarevskaya Station
– BBC World Service, 5 May 2015

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

Erlang's practical realization

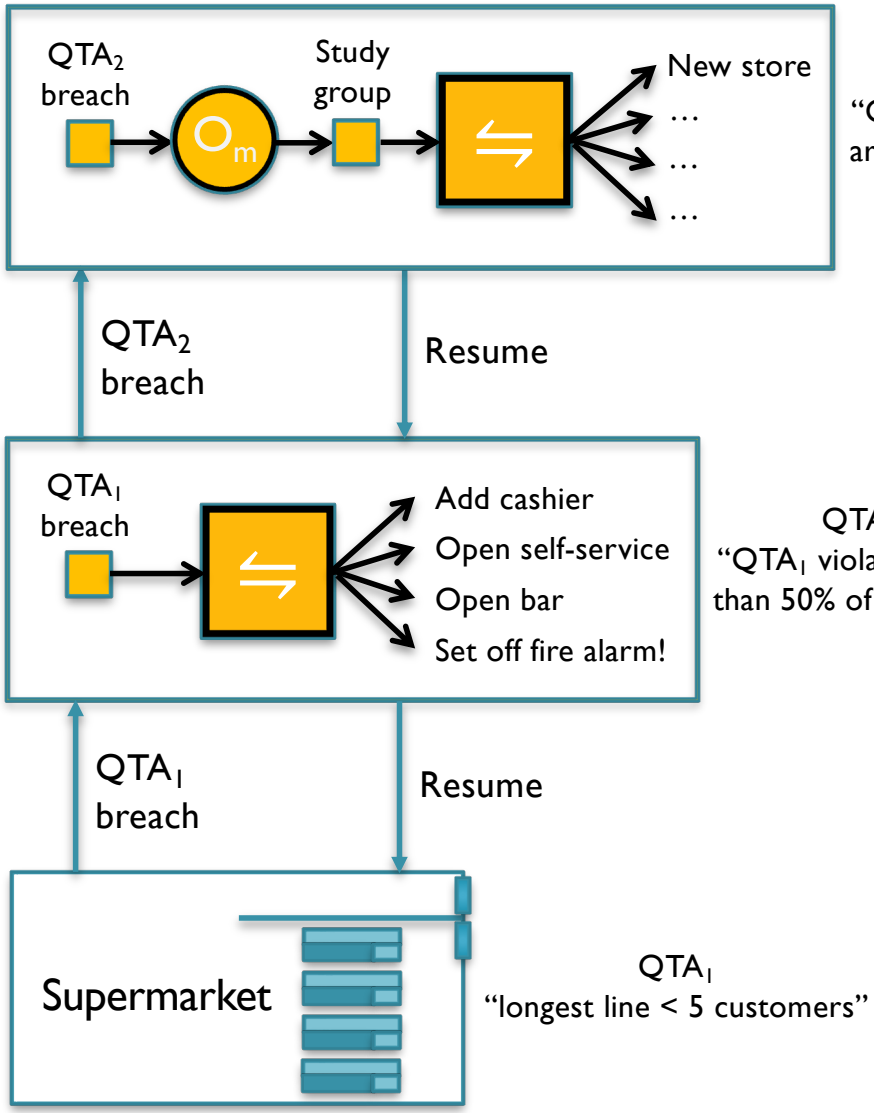
- Erlang/OTP is a development platform for robust networked applications that largely supports the multilevel approach
- Primitive concepts and operations
  - Fine-grain concurrency
  - Processes share no state (mostly independent)
  - Processes are fail-fast (do not repair themselves)
  - Process crashes send notifications (fault detection)
  - Modules can have two code versions (for hot code update)
- High-level libraries
  - Hot code update for applications that must never go down
  - Supervisor trees for fault handling (but not fully independent!)
  - Behaviors (libraries) simplify application development
  - Support for graceful degradation on overload

Supermarket example



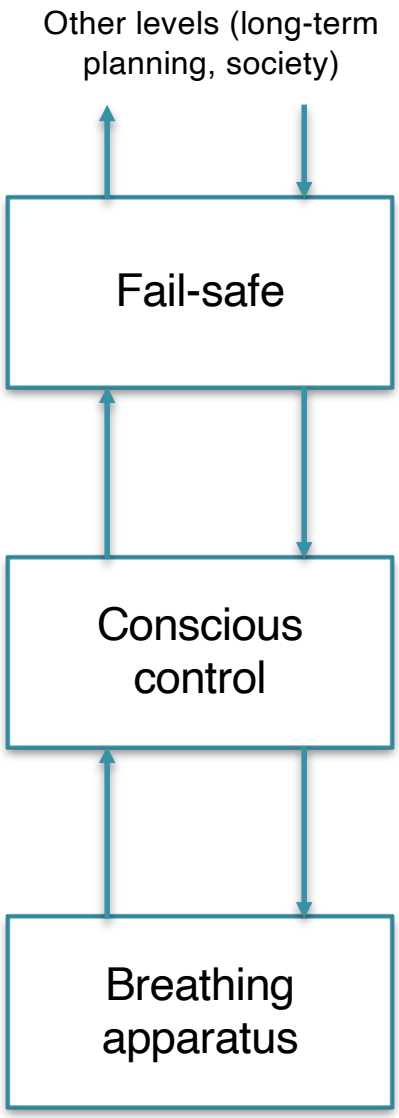
- To illustrate the approach we design a supermarket
  - Customers enter the supermarket, collect their merchandise, and queue up at an open cash register
- QTA_1 = "less than 5 customers are in line"
  - What happens when there are too many customers in a line?
  - What do the next levels look like?

Supermarket multilevel system



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

Human respiratory system



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

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

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

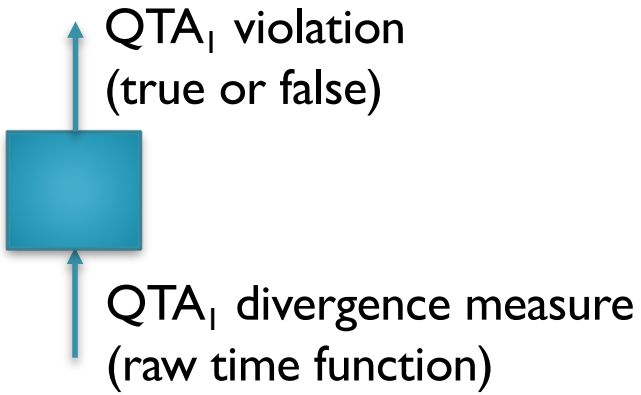
- The human respiratory system is a multilevel system
  - Designed and debugged by billions of years of evolution
  - This diagram summarizes a precise medical description (Wikipedia entry “Drowning”)
- Conscious control is a powerful problem solver but it must be used correctly
  - Breathing reflex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

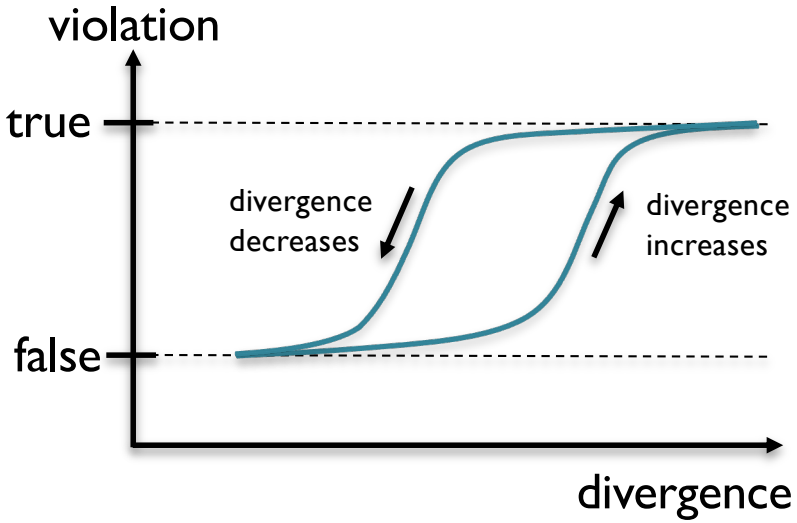
Mitigate or propagate?

- Operation of each level
  - Each level has its own QTA
  - QTA breach detection activates the next level

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

Breach detection

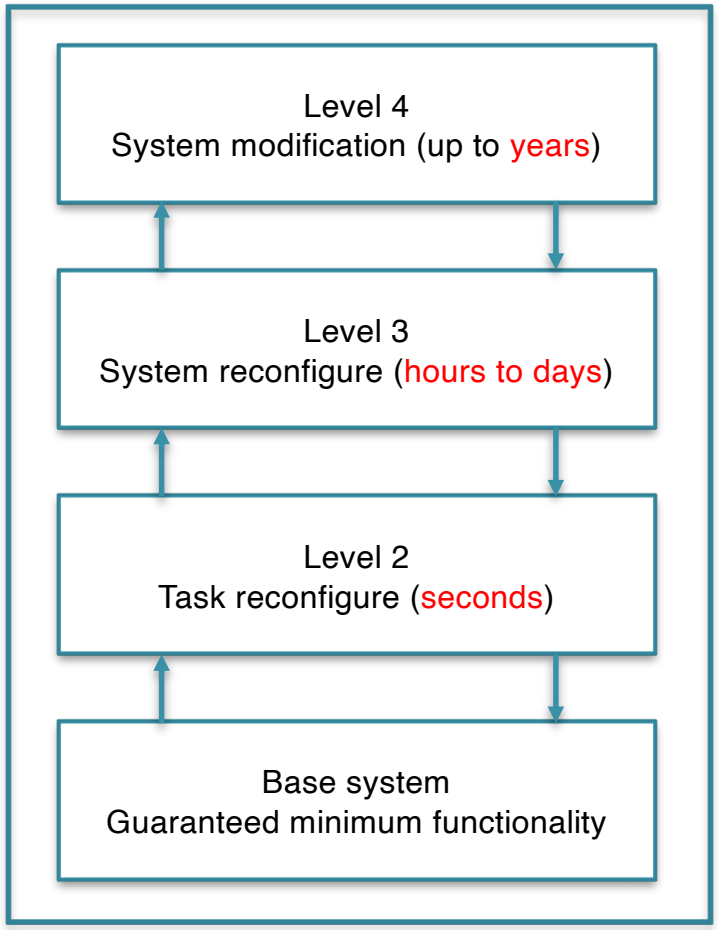




- Base system does continuous measurement of QTA_1 divergence
  - Quantified hazard, by comparison of QTA_1 and delivered operations
- When the divergence goes above a critical value, the violation flips and a message is sent to the next level
- Hysteresis is used to avoid oscillations at the boundary
  - This can happen when the divergence measure is noisy

Correlations

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



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

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

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

Actions at each timescale

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

Beyond inductive reasoning (some poetic reflections)

Beyond inductive reasoning

- At each new scale, new phenomena appear …

Sam Spade: "Ten thousand? We were talking about a lot more money than this."
Kasper Gutman: "Yes, sir, we were, but this is genuine coin of the realm. With a dollar of this, you can buy ten dollars of talk."

– *The Maltese Falcon (Dashiell Hammett, 1941)*

- This seems to be a basic law of nature in all areas
  - We see this every day in any kind of system that can get big
  - Science and engineering are continually confronted with it
    - Computing systems are the most complex man-made systems

Alps viewed from space

- This amazing sight was never seen by humans until spaceflight was invented
- But it has always existed!
  - Nature introduces new ideas at each new scale: new beauty, new science
  - Natural intelligence and natural creativity far outstrip all human activity
  - It is our destiny to strive to understand nature
- Goethe explains it well…

Goethe's flower

… imagine a green plant shooting up from its root, thrusting forth strong green leaves from the sides of its sturdy stem, and at last terminating in a flower. The flower is unexpected and startling, but come it must – nay, the whole foliage has existed only for the sake of that flower, and would be worthless without it.

– *from "Conversations of Goethe with Johann Peter Eckermann" (1930 translation)*

Explaining Goethe's flower

- A scalable system is always moving toward something new
  - A well-designed system will enter terra incognita and experience new problems and new behaviors
    - It may jump off a cliff, travel to the moon, or make a flower
  - Our goal is to make sure it behaves reasonably!
    - We do not know exactly what it will do, but we can reduce the chance that something bad happens
- System designers are like gardeners planting a flower
  - We cannot know how the flower will grow, but we can maximize the chance of a good outcome
    - A large rose bush needs to be fertilized and trimmed as it grows
  - A rose bush can become enormously large
    - Thousands of roses on one plant, beautiful!
    - Fragile stems become thick wooden trunks

Conclusions

Conclusions

- Scalability is a multifaceted problem
  - Achieving it is much more than just adding resources
  - Achieving it becomes more difficult as scale increases
    - Traditional techniques (induction, probability) lose their power
    - New, unexpected phenomena appear at each new scale
  - ICT infrastructure is still growing exponentially, requiring scalability
- Temporary overload
  - We give three design principles for temporary overload
  - Based on a simple system model, we show what can be expected
- Permanent overload
  - The system consists of a hierarchy of independent observers, where each level handles hazards that cannot be handled at a lower one
  - Each level has its own timescale
  - External correlations are the most difficult to handle
- Collaboration with PNSol
  - This work is part of the dissemination of ΔQSD
  - I am giving a tutorial on ΔQSD at ICPE 2024, London, May 7, 2024