CODE BEAM LITE

STOCKHOLM

12 MAY 2023

CODE BEAM LITE STOCKHOLM

Little Things and Big Things: Lessons for Building Large Systems

Friday, 12 May 2023 13:55

Peter Van Roy
Université catholique de Louvain

PREDICTABLE NETWORK SOLUTIONS

UCLouvain

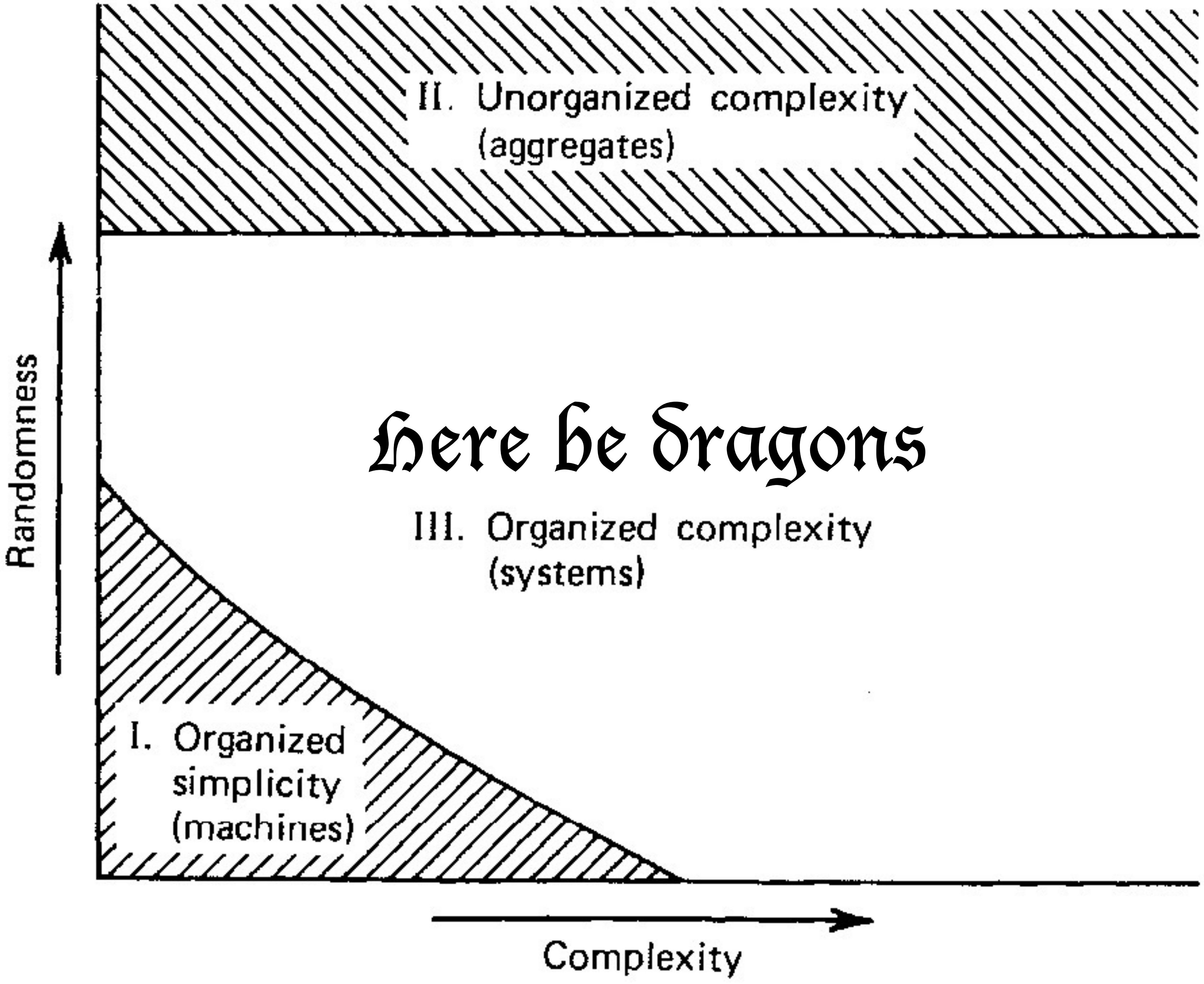
Overview

- The Christmas tree
- Starting points
  - The first law of scalability
  - The two great frauds
- A real industrial example
  - The small cells case study
- System design for scalability
  - Design for overload (temporary overload)
  - Multilevel system (permanent overload)
  - Taming nondeterminism
- Large-scale phenomena
  - Buridan's principle
  - Sudden changes
  - Consistency
  - Small-world networks
- Conclusions

Background

- My background (www.info.ucl.ac.be/~pvr/pldc.html)
  - I am a computer scientist who is interested in programming languages and systems. 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)
  - This talk is dedicated to the people at PNSol, in particular Neil Davies and Peter Thompson, who have spent their lives working behind the scenes to make sure that our communications infrastructure keeps working and who have developed an innovative new system design approach that has shown its mettle in many industrial projects



From *An Introduction to General Systems Thinking* (Gerald Weinberg, 1975)

The Christmas tree

The Christmas tree

- System design is like a Christmas tree with lots of pretty ornaments
  - Too many for one talk!
  - We have not finished decorating the tree yet; this talk gives a snapshot of the current state of the tree
- In this talk I will focus on a few of the nicest ornaments
  - Some important principles of large system design
- At the end of the talk, I give references for people who want to look up some of the other ornaments

Starting points

The first law of scalability

The first law of scalability

- At each new scale, the situation changes…

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)

  - No problem is ever solved for all scales (despite contrary claims)
- This is a basic law of scalability in all areas
  - We see this every day in any kind of system that can get big
    - Biological systems take the lead in complexity
    - Computing systems take the lead for man-made systems
- How does the situation change as the system grows?
  - Goethe has explained it …

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)

Goethe's flower and the first law

- A growing system is moving toward something
  - A well-designed system "knows things" that the designer doesn't
    - It may jump off a cliff or travel to the moon
  - Our goal is to make sure it behaves reasonably!
    - We don't know what this means exactly, but we can reduce the chance that something bad happens
- It is like a gardener 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 trimmed as it grows
  - A rose bush can become enormously large
    - Thousands of roses on one plant, beautiful!
    - Fragile stems become thick wooden trunks

The two great frauds

The two great frauds

- 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 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 will not 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!
    - Sometimes it can be used when combining independent events

Black swans

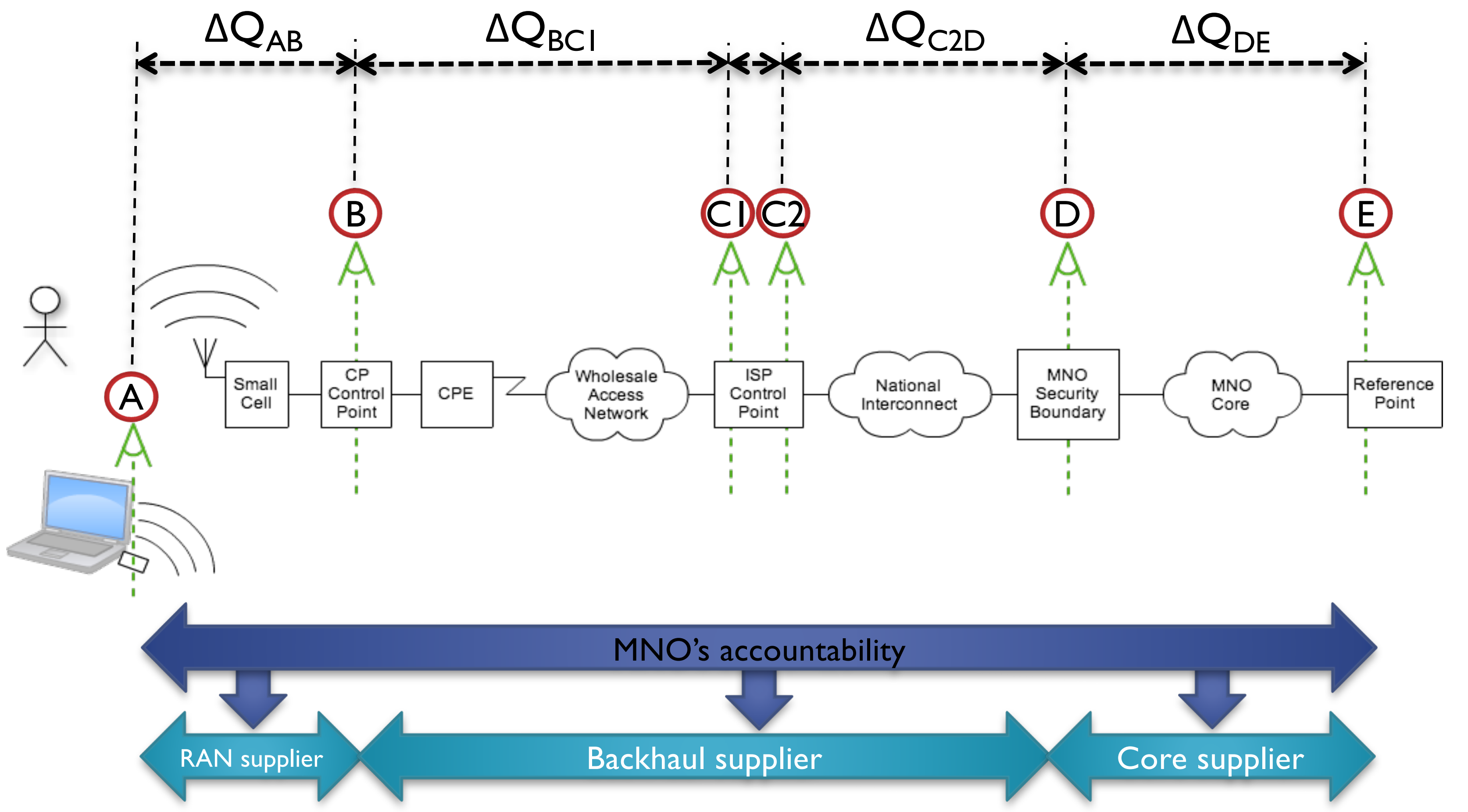
- Systems are designed by relying on induction
  - However, induction often has a built-in limit and fails beyond
  - Year 2000 Bug: it was “far away” but it has arrived (I was there!)
  - Dinosaurs and banks: “too big to fail” but they fail
- In computer systems this is both ubiquitous and hidden
  - 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

A real industrial example: the small cells case study

A real industrial example

- Small cells case study (from PNSol)
  - A major MNO deployed small cells: low-powered cellular radio access nodes with range 10m-3km, backhaul using consumer DSL broadband
    - This is an actual industrial example from the company PNSol. We are collaborating with PNSol on ΔQSD, a system development approach that allows predicting performance at high load before the system is built.
  - The system worked but did not scale, despite all components having enough bandwidth!
    - Voice quality had major problems, cells were failing
- We will investigate and learn an important lesson
  - No spoilers! You will understand very soon.
  - We present some theory to explain the situation

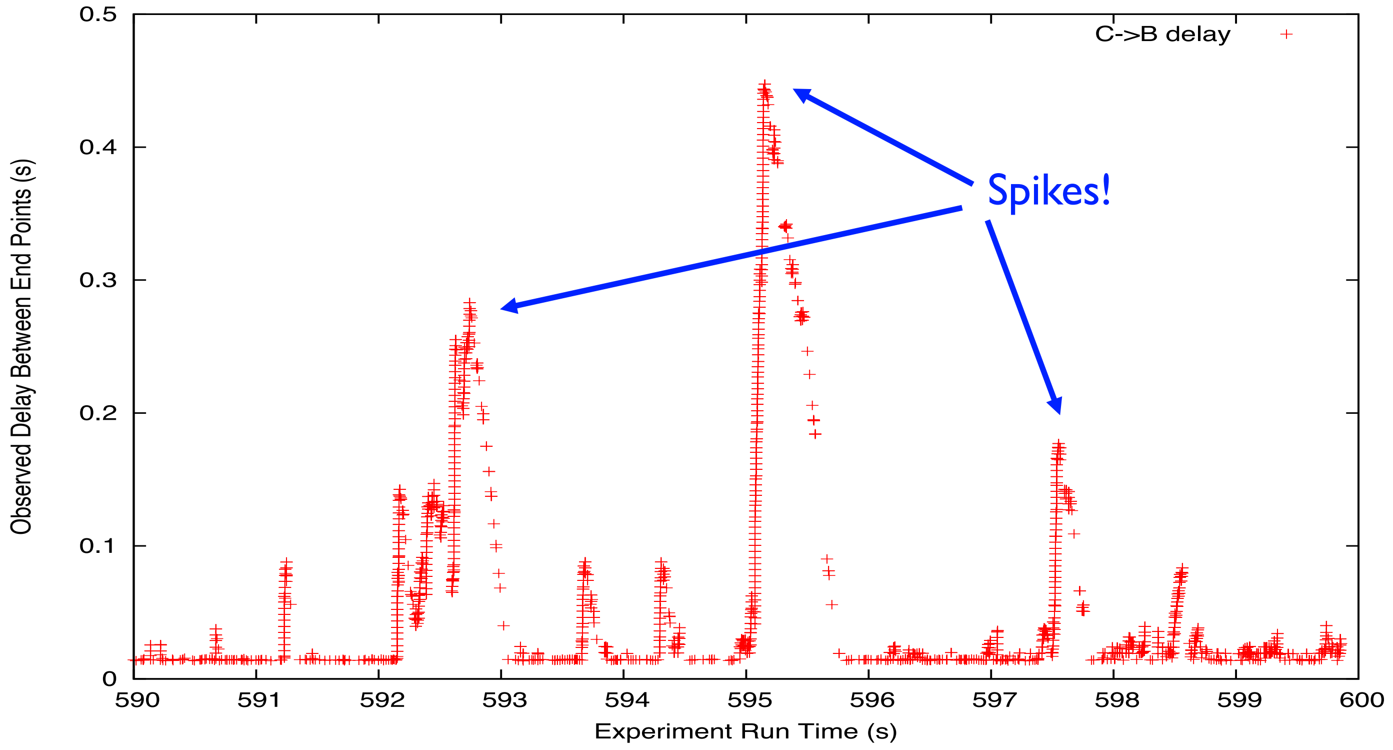
Mobile network architecture



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

Zoom in on the problem

Packet delays were measured at high resolution

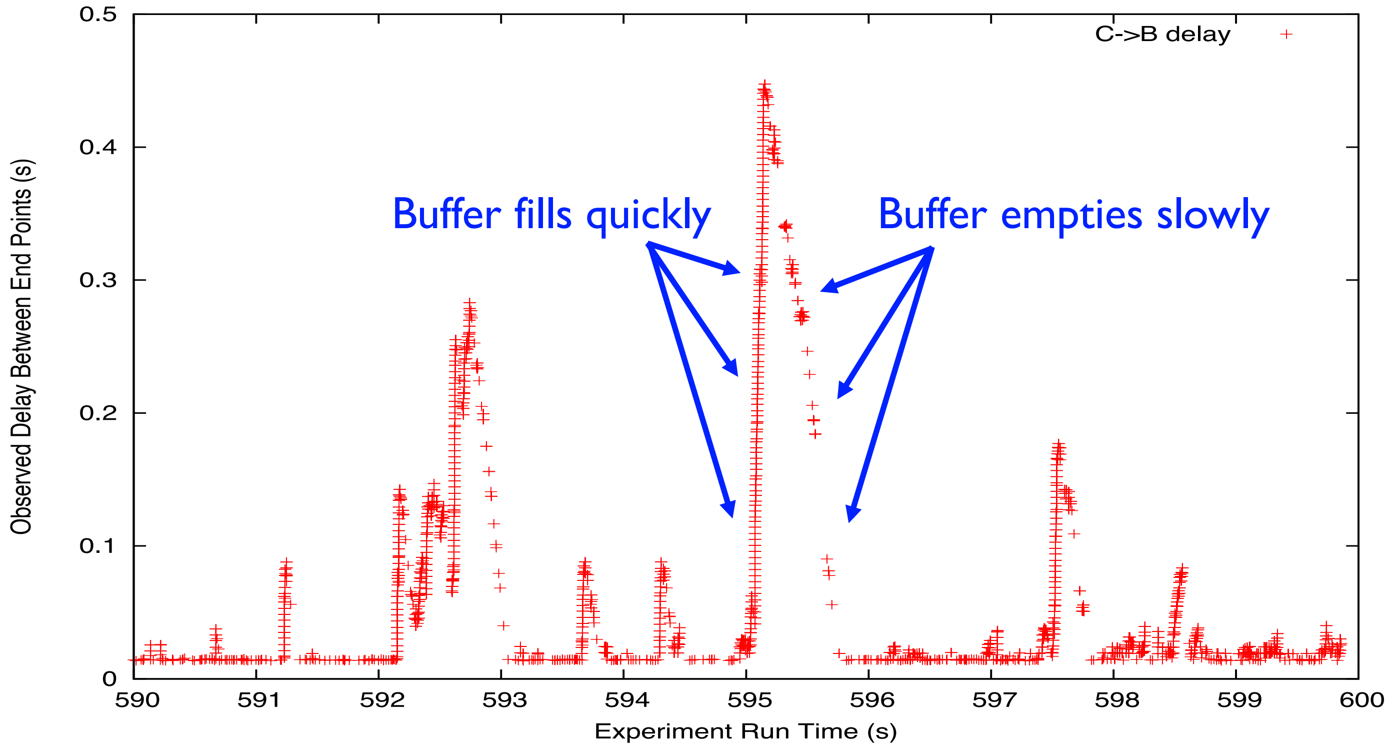


Occasionally, there are spikes with enormously increased delay. The spikes can last for seconds and cause the small cells to fail. Small cells are known to fail if their control loops are too slow.

What is causing these spikes?

Zoom in on the problem

Examine the structure of a spike

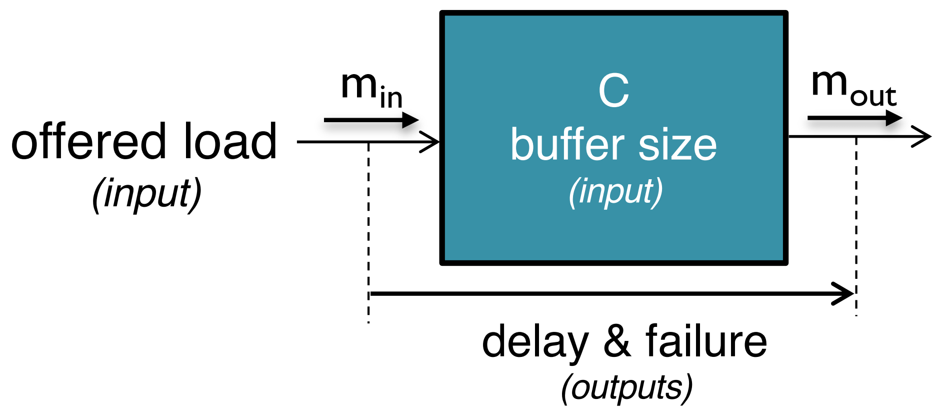


The delay increases very quickly and decreases slowly.
This is a queue overload problem:
temporary overload causes a buffer to fill up quickly,
but emptying the buffer takes much more time

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.
- The root cause is that the subsystems don't compose
  - Pre-requisites for use of one element are not met by other elements
    - Common structural problem, not unique to this MNO or technology
  - The MNO believed they only had to match bandwidths (numbers)
    - They should match delay probabilities (cumulative distribution functions)!

Understanding a component



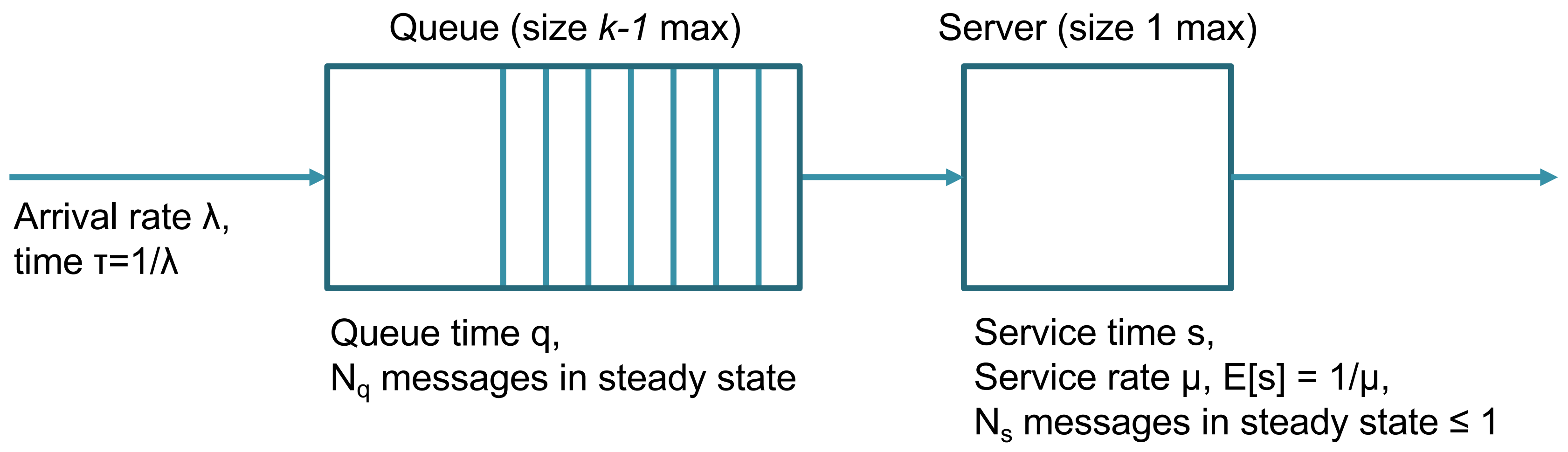
- To understand better, let us look closely at a component
  - To get intuition, we model the component as a queue
- A typical component has four parameters of interest

Inputs
- Offered load *a* : arrival rate of messages
- 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

Component as M/M/1/K queue



Queue (size *k-1* max)
Server (size 1 max)
Arrival rate λ, time τ=1/λ
Queue time q, N_q messages in steady state
Service time s, Service rate μ, E[s] = 1/μ, N_s messages in steady state ≤ 1

- We model a component as an M/M/1/K queue
  - M: arriving messages have Exponential distribution with rate λ
  - M: service time has Exponential distribution with rate μ
  - 1: one message can be served at a time
  - *k*: total buffer size is *k* (buffer size = queue size *k-1* + server size 1)
- Offered load *a = λ/μ* (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)

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 rate 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!
  - Failure rate tends goes up to 100% for high load!
  - Delay increases to k when a exceeds 1
- Quick switchover somewhere between *a=0.5* and *a=1*
  - As load increases beyond 0.5, the system quickly gets very bad

⇨ Even a temporary overload causes a big, long-lasting degradation
- *This is the cause of the problem in the small cells case study*

System design for scalability

System design for scalability

- We focus on fundamental issues of system design that are important for scalability
  - Design for overload (temporary overloads)
  - Multilevel systems (permanent overloads)
  - Taming nondeterminism

- We will not explain the mechanics of fault tolerance and increasing performance
  - Fault tolerance needs redundancy
    - Especially well-known in the Erlang community
    - Large systems are always repairing themselves
  - Increasing performance needs parallelism
    - This is well known everywhere

Design for overload (temporary overloads)

Design for overload

- You design your system to handle a given 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 behave reasonably (discussed in this section)
  - Permanent (long-lasting) overloads: this is a timescale issue and must be handed to the next level (discussed in the next section)

Temporary overloads

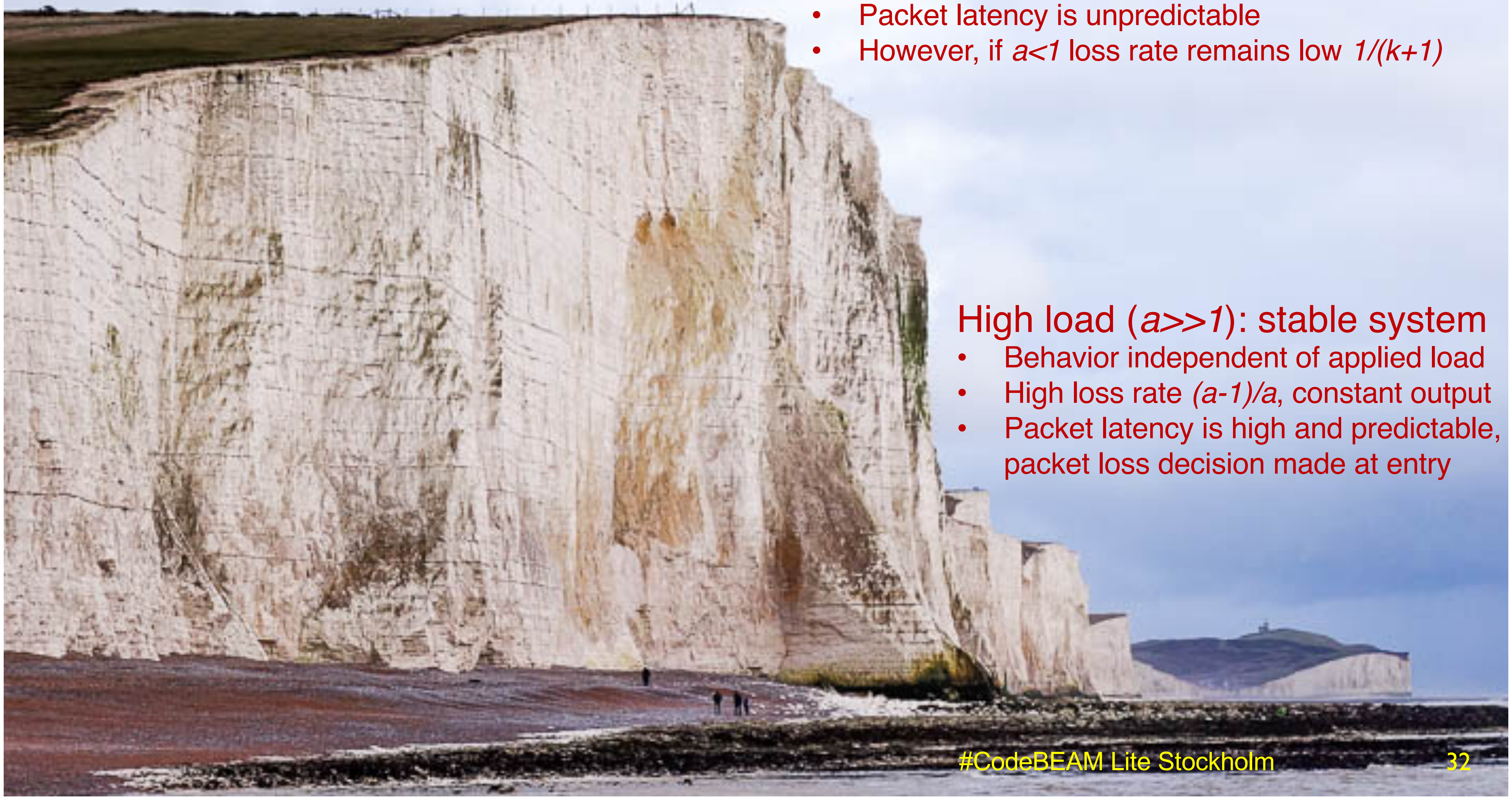
- Three key design rules:
  1. When overloaded, the system may behave badly but it must never break
     - When the overload goes away, the system goes back to normal
     - System should be “ballistic”: perform predictably in open loop
  2. When overloaded, the system must provide some guaranteed minimum 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

Jumping off a cliff (1)

- What happens when the system is overloaded?
  - The system may behave badly but it should be predictable!
  - Let us look again at our trusty M/M/1/K queue

- Maximum fluctuations happen at the 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 much 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

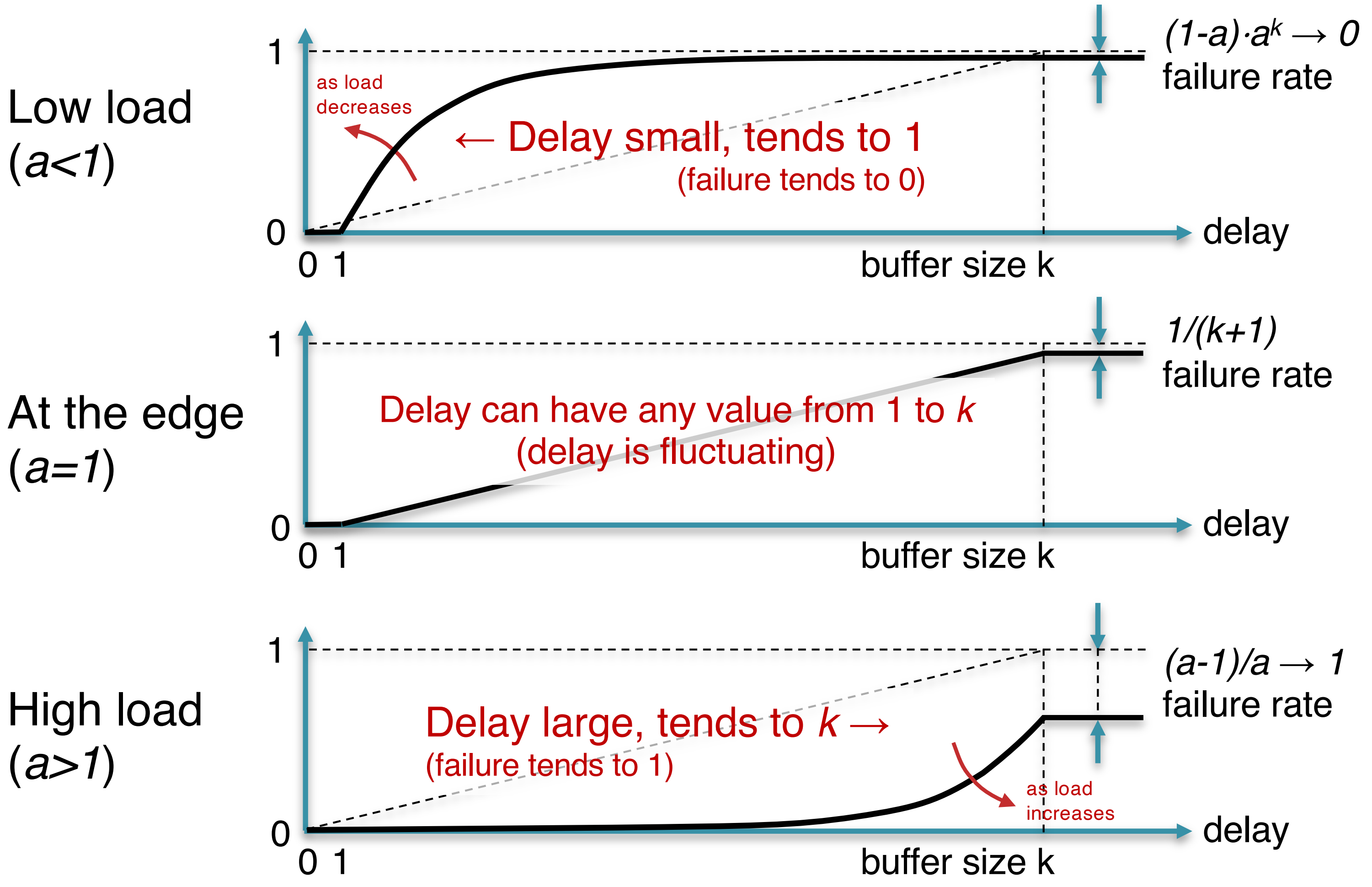
Critical load (*0.5<a<2*): large fluctuations

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

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

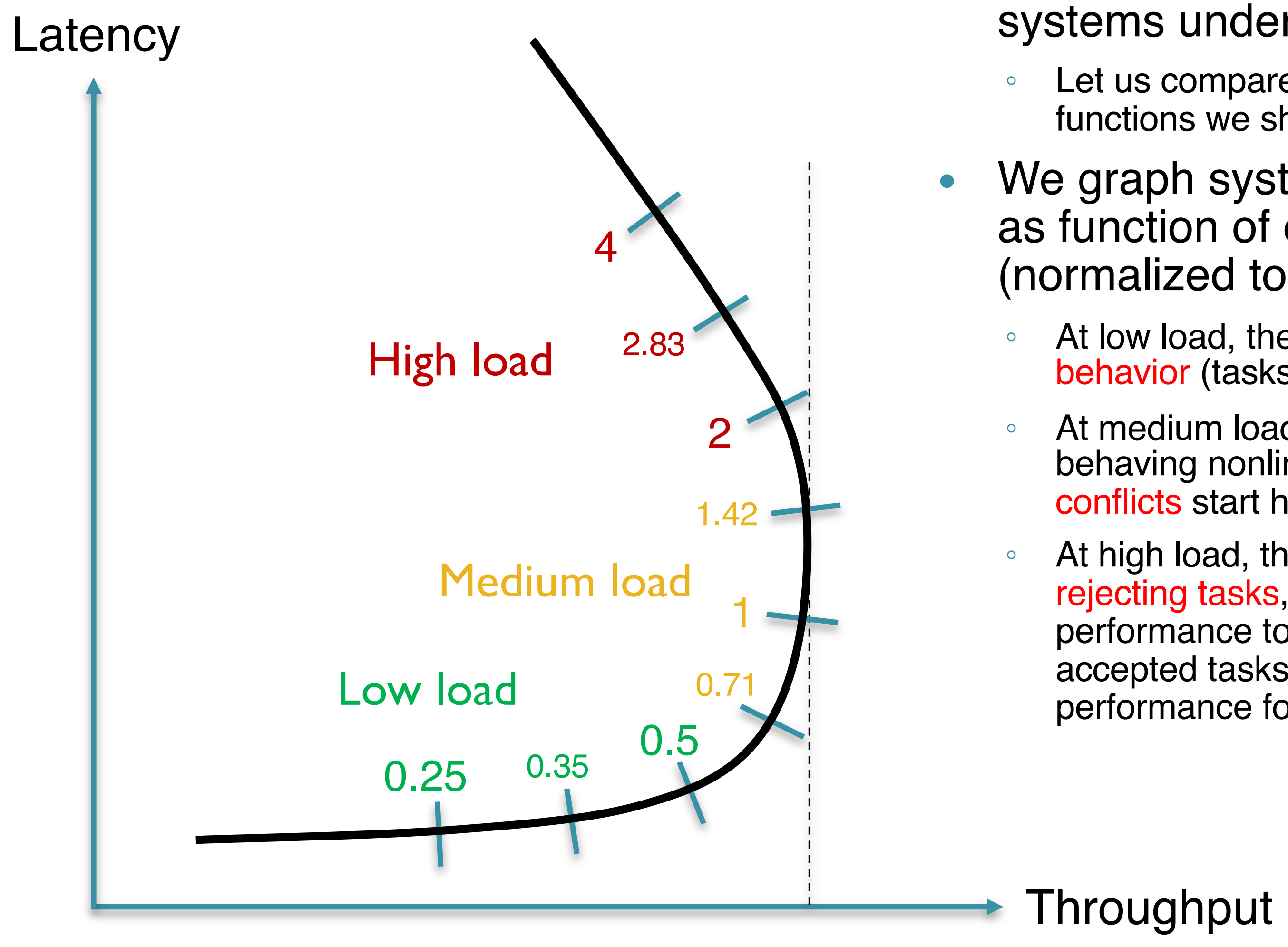
- Behavior independent of applied load
- High loss rate *(a-1)/a*, constant output
- Packet latency is high and predictable, packet loss decision made at entry

Theoretical analysis



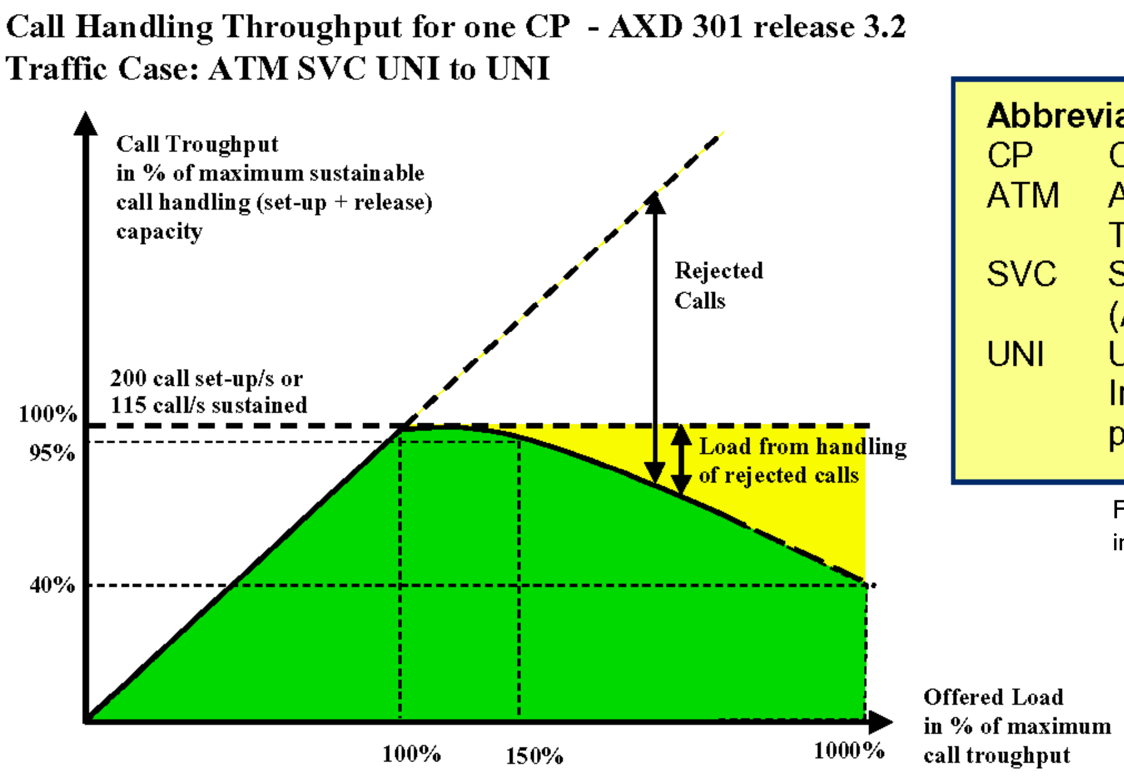
- We show cumulative distribution function (CDF) of system delay
- Theoretical analysis based on M/M/1/K queue (as before)
  - Offered load *a* (normalized to 1), buffer size *k*

Throughput-latency diagram



- A common way to illustrate systems under load
    - Let us compare it to the CDF delay functions we showed before
- We graph system performance as function of offered load (normalized to 1)
    - At low load, the system has linear behavior (tasks are independent)
    - At medium load, the system starts behaving nonlinearly (resource conflicts start happening for all tasks)
    - At high load, the system spends time rejecting tasks, which causes performance to decrease for accepted tasks. Graph shows performance for accepted tasks only!

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.

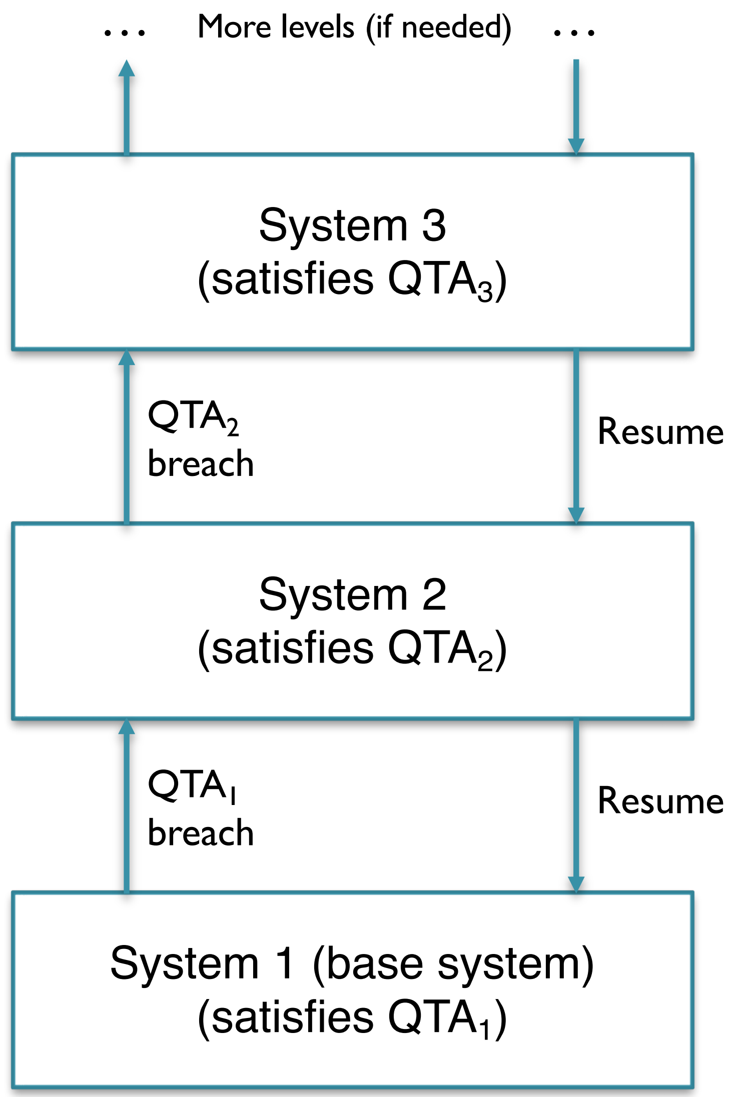
- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - Erlang/OTP at that time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

Multilevel system (permanent overloads)

Multilevel system

- If your system suffers overload too often, we say that it has a permanent overload
  - This is usually unacceptable, so we need to take action!
  - The action to be taken depends on the timescale
- Multilevel system
  - The system is built as several levels
  - The bottom level is the base system that we saw before
  - Higher levels are designed to support other timescales
- "Mitigate or propagate"
  - It is critically important to know when to hand over control to a higher level
- Erlang supports this design approach
  - But be careful to keep the levels truly separate

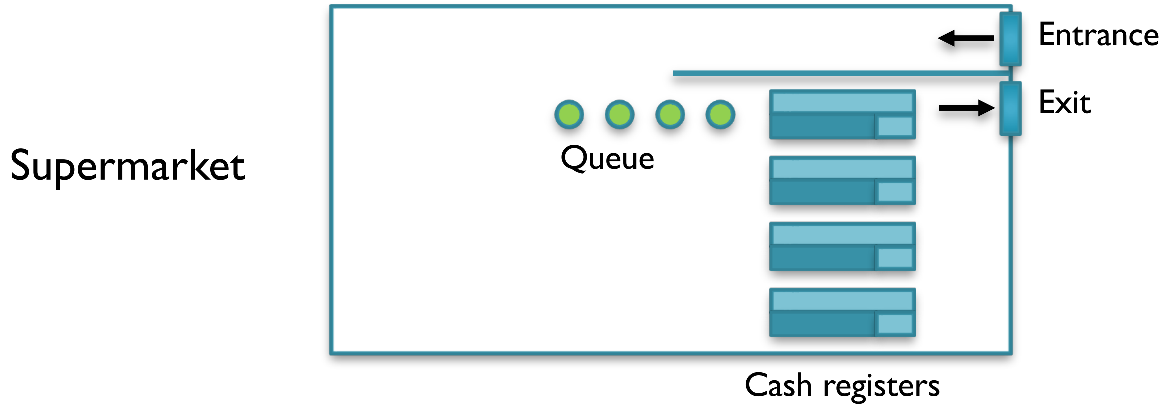
Multilevel system



. . . More levels (if needed) . . .
System 3 (satisfies QTA_3)
QTA_2 breach
Resume
System 2 (satisfies QTA_2)
QTA_1 breach
Resume
System 1 (base system) (satisfies QTA_1)

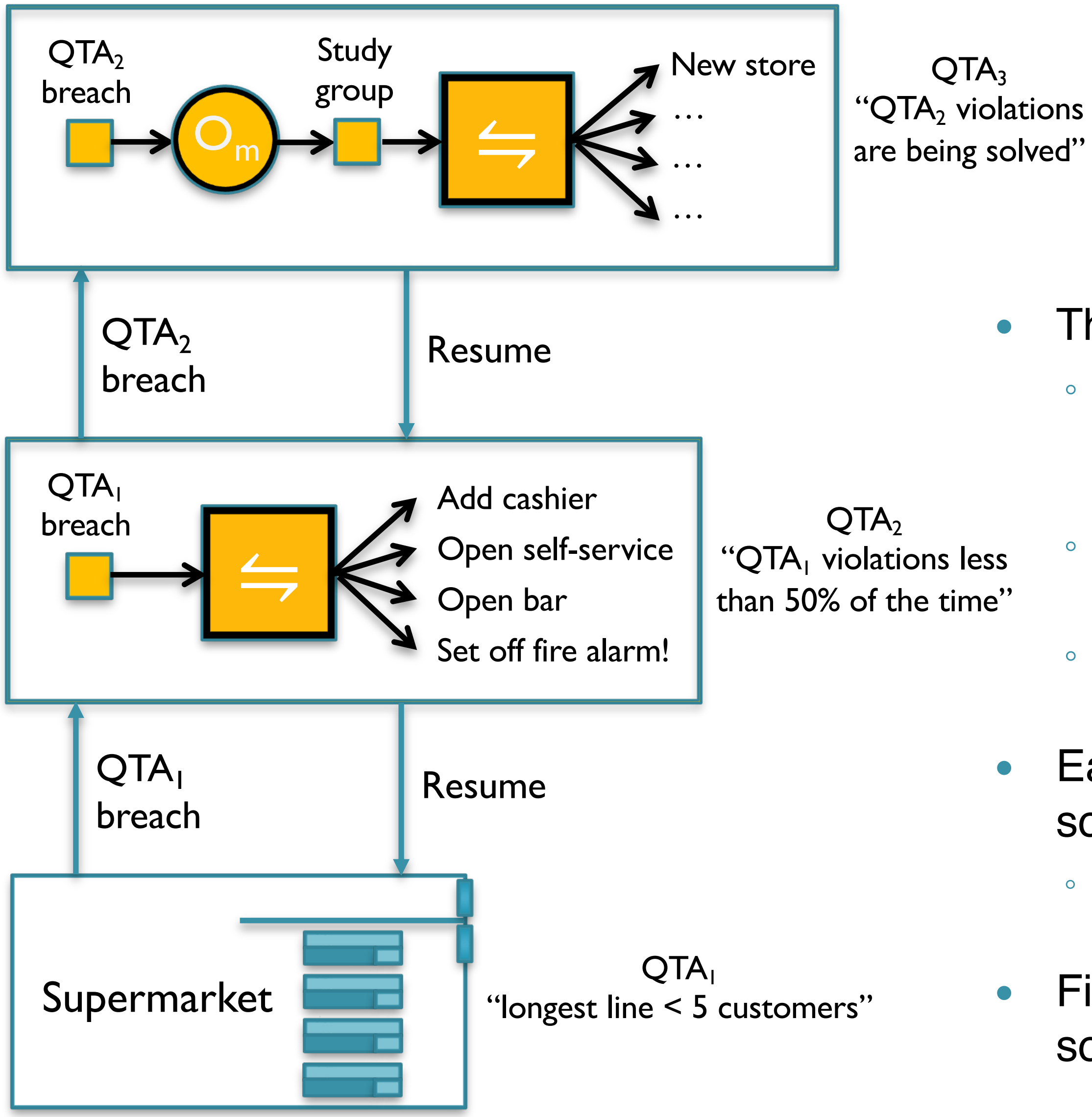
- During normal operation, System 1 does the work
  - The other systems monitor this but normally do not intervene
  - Upon QTA_1 breach, System 2 is notified (QTA = Quantitative Timeliness Agreement)
- System 2 has several options:
  - Take over from System 1, temporarily or permanently
  - Reconfigure 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

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
- Each level "escalates" the solution
  - Each level happens at a different timescale
- Fire alarm is an emergency solution with low probability

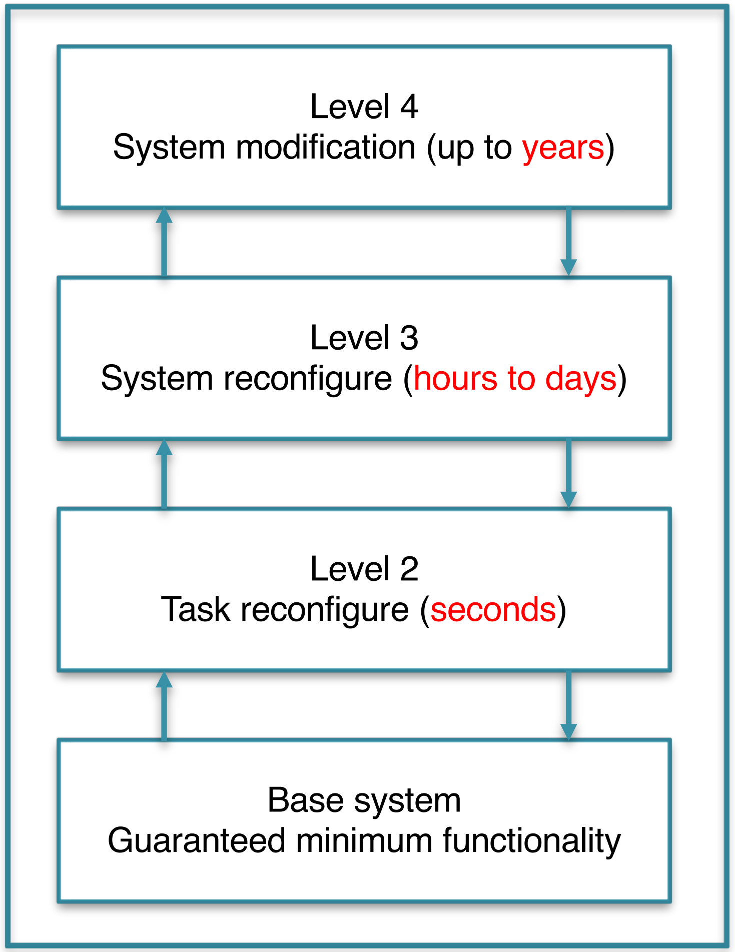
Mitigate or propagate?

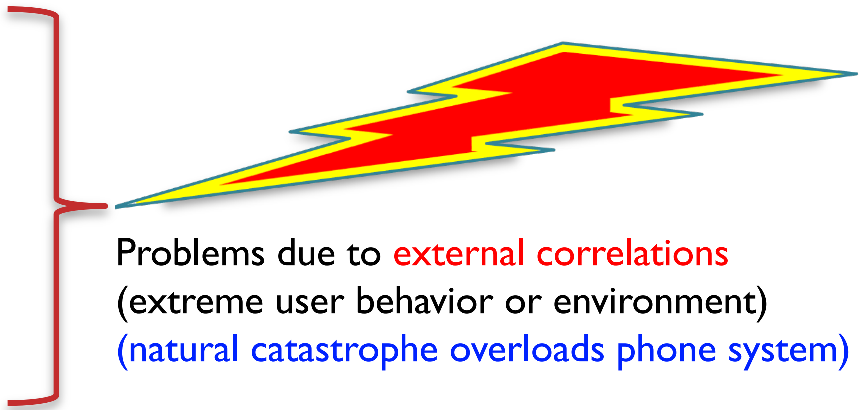
- How do we design each level?
  - Each level is designed to satisfy a specification, called QTA (Quantitative Timeliness Agreement)
  - QTA breach detection activates the next level

- 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 it 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)

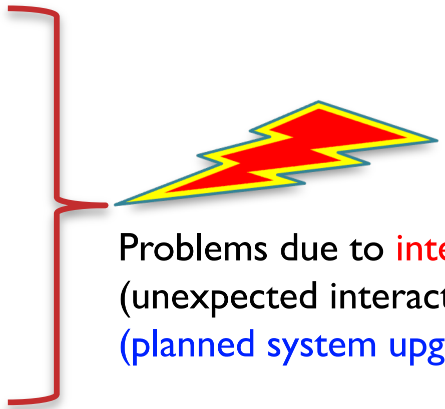
Correlations

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


Level 4
System modification (up to years)
Level 3
System reconfigure (hours to days)
Level 2
Task reconfigure (seconds)
Base system
Guaranteed minimum functionality



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)

Multiple timescales

- Base system is designed to obey three rules:
  1. When overloaded, the system may behave badly but it must never break
     - 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)
  3. When overloaded, the system must still be observable and controllable
- Level 2: reconfigure with respect to primitive tasks (seconds)
  - Drop nonessential traffic; stop admitting new tasks; kick out running tasks
- Level 3: reconfigure overall system (hours to days)
  - Depending on timescale: admission control, cold standbys, data center elasticity, software rejuvenation, put human in the loop
- Level 4: system modification (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

Taming nondeterminism

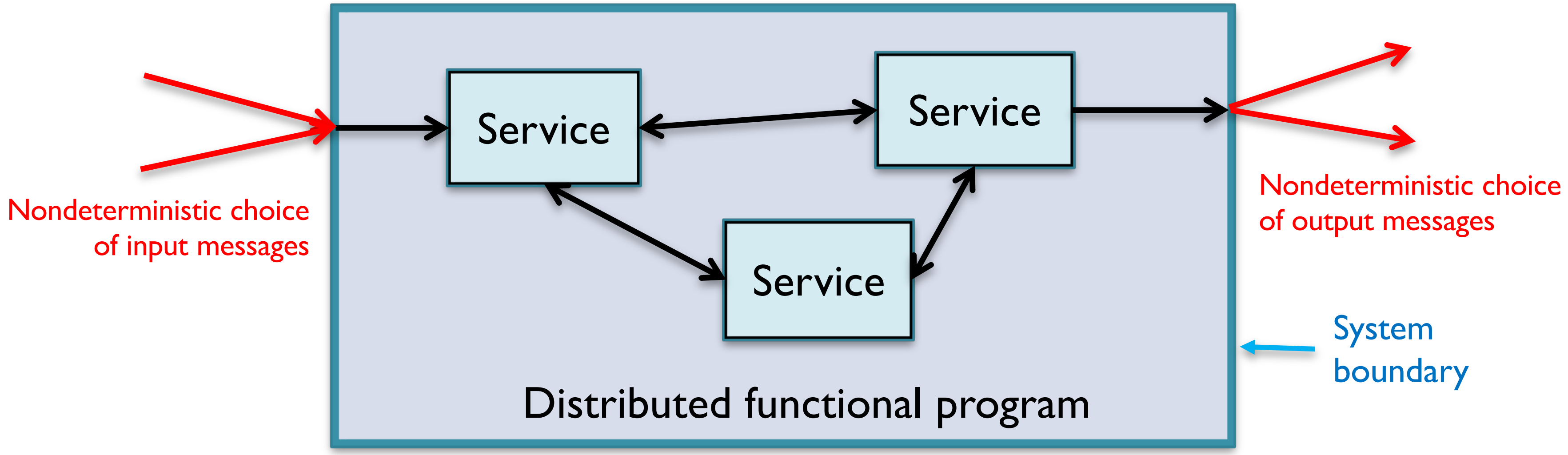
Nondeterminism is ubiquitous

- Distributed systems have a lot of nondeterminism

  ⇨ Nondeterminism is defined as choices that the program must make at run-time that cannot be made by the developer

  - Every Erlang process is a source of nondeterminism, since it can receive messages from multiple sources
  - Commonly used distributed algorithms, like consensus, are inherently nondeterministic
- But nondeterminism is hard to manage
  - Debugging nondeterministic systems is incredibly hard
  - The bigger the system gets, the worse the problem gets
- Can we reduce the amount of nondeterminism?
  - How much of the nondeterminism is really necessary?
    - Much Erlang code is *de facto* deterministic (e.g., behaviors)
    - ! I claim that most of the nondeterminism in today's large distributed systems is superfluous! Let's take a closer look…

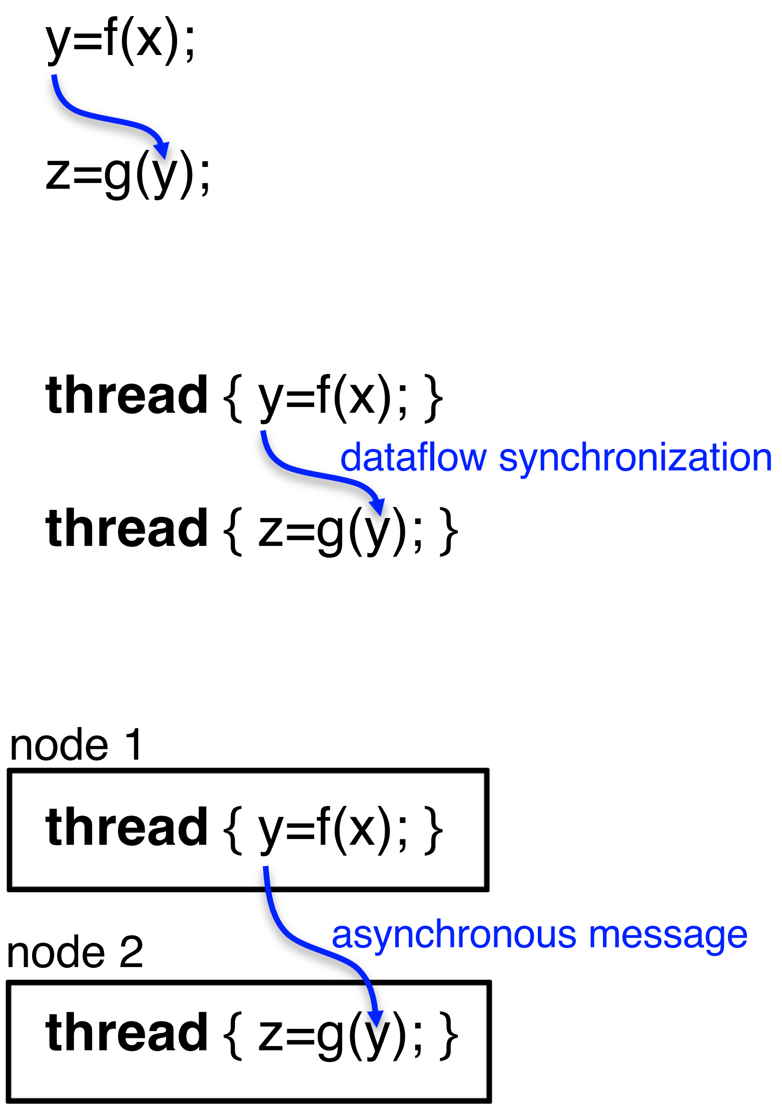
Reducing the nondeterminism


Service
Service
Service
Distributed functional program
Nondeterministic choice of input messages
Nondeterministic choice of output messages
System boundary

- Nondeterminism is only really needed at the system boundary
  - When the system receives or sends messages from outside, it chooses origin or destination nondeterministically, depending on conditions like timing & causality.
- Inside, nondeterminism is not needed at all
  - Inside, the system is a distributed functional program (completely deterministic)
  - Can this really work?
    - What about concurrency?
    - What about asynchronous messages between nodes?
    - They are not a problem! Let's see why…

From functional to distributed

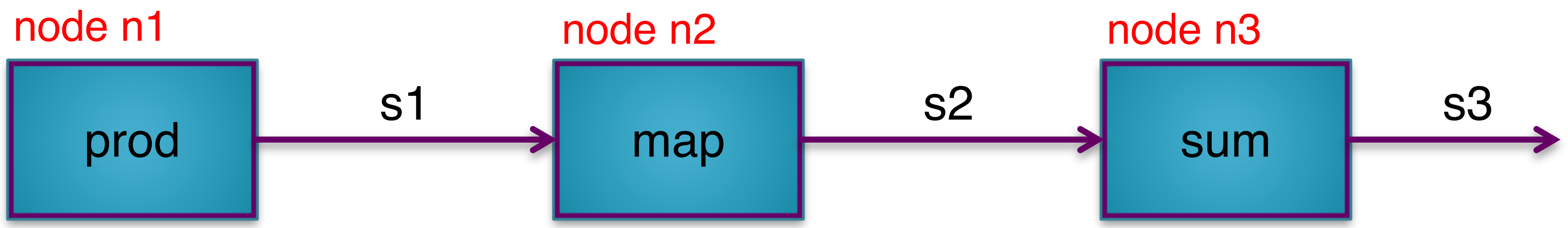
- Functional programming
  - Pure functions (λ calculus)
  - Church-Rosser property: same results for all reduction orders (confluence)
- Concurrent functional programming
  - Each function in a thread and connected using dataflow (read waits until bound)
  - Church-Rosser property still holds
- Distributed functional programming
  - Threads on different nodes, binding sends a message asynchronously
  - Church-Rosser property still holds



Distributed functional program

- Using agents, streams, and threads:
  - Agent = tail-recursive function executing in its own thread
  - Stream = list created by an agent & read by another agent (dataflow)
  - Thread = sequential execution of instructions

```
{ var s1, s2, s3;
   node n1 { s1=prod(1); }
   node n2 { s2=map(s1,fun (x) { x*x; }; }
   node n3 { s3=sum(s2,0); }
}
```



```
fun prod(n) {
    delay(1000);
    n|prod(n-1);
}
```

```
fun map(s, f) {
    case s of h|t then
        f(h)|map(t, f);
    [] nil then
        nil;
    end;
}
```

```
fun sum(s, a) {
    case s of h|t then
        h+a|sum(t,h+a);
    [] nil then
        nil;
    end;
}
```

How to build distributed systems

- The right way to build distributed systems:
  - Internally, the system is a distributed functional program
    - No problem with concurrency or asynchronous messages
  - Nondeterminism is needed only at the system boundary
    - Where the system interacts with the external world
  - Choose carefully your distributed algorithms!
    - Avoid consensus, replace it by CRDTs if possible
- Is this really the right way?
  - Today’s big systems are moving in this direction.
    For example, big data analytics systems originating with MapReduce are all mostly functional.
    - On a personal note, the Lasp system designed by C. Meiklejohn and myself is distributed, reliable, scalable, and completely functional
  - For Erlang, it is already good style that your programs should be as deterministic as possible

Large scale phenomena

Large scale phenomena

- The rest of the talk is not yet integrated into a uniform framework
  - We are still working on this part of the approach
  - It's important to be aware of these phenomena and the conditions in which they appear
- We show phenomena that appear at large scale
  - Buridan's principle
  - Sudden changes ("hiccups")
  - Consistency
  - Small-world networks
- This is a non-exhaustive list!

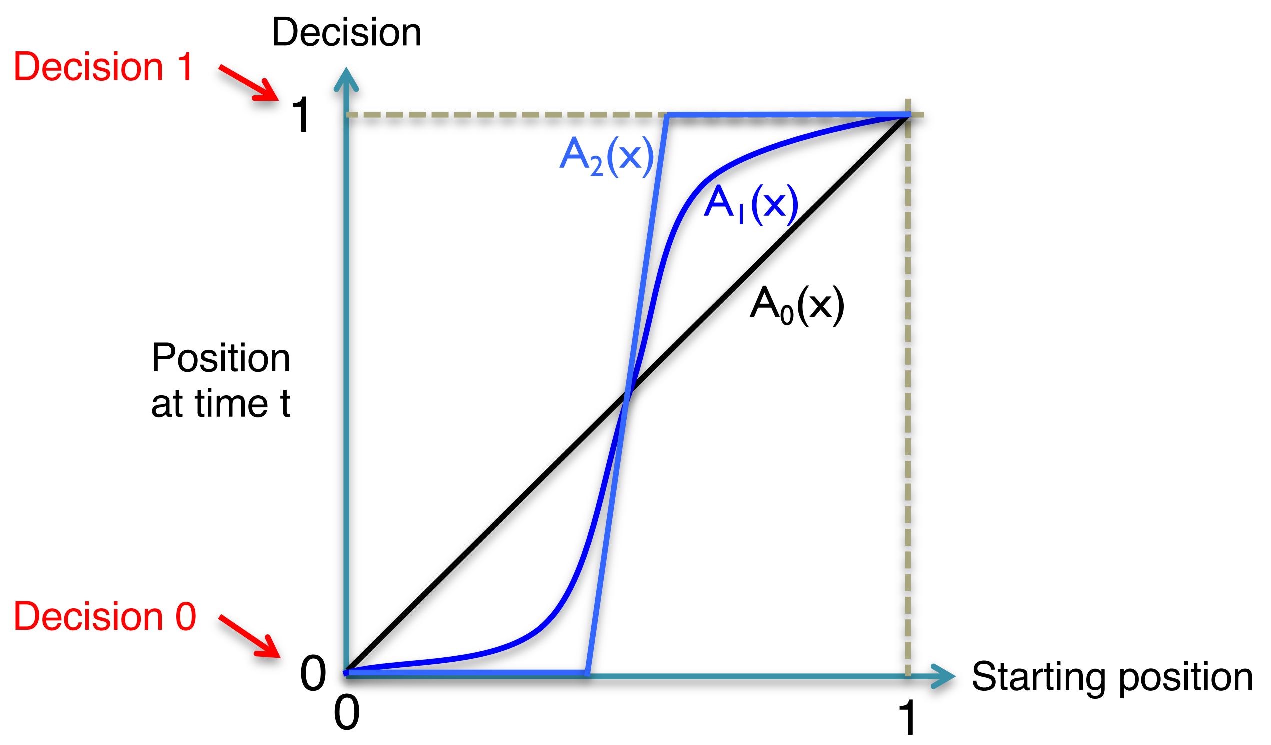
Buridan’s principle

Buridan's principle

Also known as a donkey

- Philosopher Jean Buridan stated that an ass placed equidistant between two bales of hay must starve to death because it has no reason to choose one bale over the other
  - See *Buridan's Principle* (Leslie Lamport, 1984)
- Assume a system has to make a discrete decision from continuous input. Then we can prove the following:
  - A *discrete* decision based upon an input with a *continuous* range of values cannot be made within a bounded length of time
- There are many examples of this principle:
  - In an election between two candidates, if the election is very close it takes longer and longer to decide. US presidential election of 2000: so close that eventually it went to the Supreme Court.
  - A jury must decide whether a student passes or fails his academic year. As the average gets closer to 50%, the decision becomes harder and harder.

Proof of Buridan's principle



- $A_t(x)$ is the position at time t with starting position x
- As t increases, $A_t(x)$ must converge to 0 or 1 for all x
- Since $A_t(x)$ is continuous in t and x, it is clear that there exist x for which t will be arbitrarily large

Buridan effect at large scales

- Distributed systems often make decisions
  - Distributed bank account: is the account positive or negative?
  - Distributed voting system: who wins the election?
- The input data is (approximately) continuous and may be distributed over the whole system
- To make the decision, more information and computation are required as the input data approaches the decision boundary
  - Far from the boundary, only local information is needed
  - Very close to the boundary, the whole system is involved
- Lesson for system design: prevention or cure!
  - Design a scalable system to stay far away from boundaries
  - Some boundaries are inevitable: in that case, try to predict when decisions are needed and « prefetch » the information (escrow)

Sudden changes

Sudden changes

- Systems with much widespread data can have sudden rapid changes ("hiccups")

⇨ Domino effects (when data is highly interdependent)

⇨ Giant component collapses

  - Epidemics (diffusion of new information)
  - Cascades (decisions with limited information)
  - Tipping points (jumps from one stable state to another)

- Some can be avoided, some cannot
  - It's important to be aware of the conditions that favor them

- We illustrate some of them

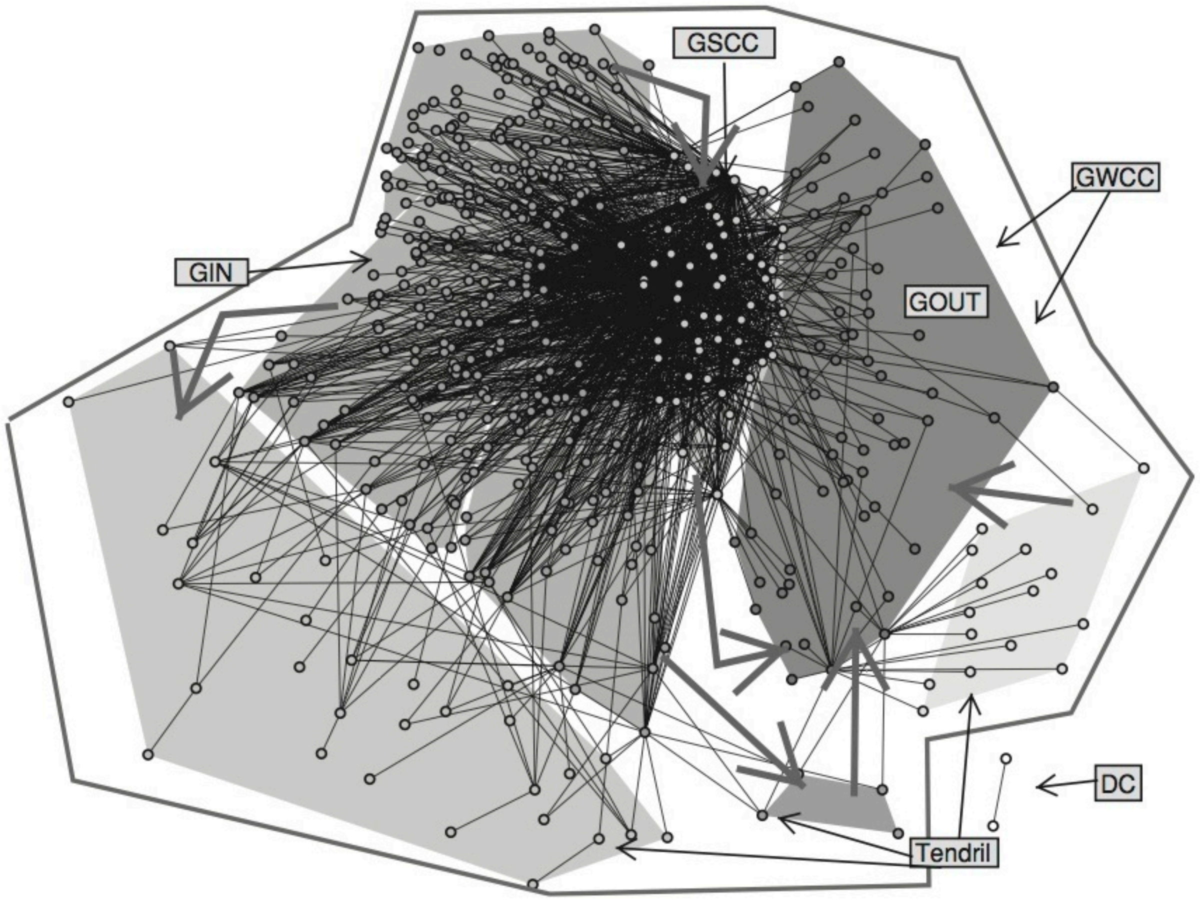
Domino effects

- A real-world network can often be modeled as a directed graph, where each node points to other nodes (like the Web)

- We define a strongly connected component (SCC) in a directed graph as a subset of nodes such that:
  1. Each node in the subset has a path to each other node in the subset
  2. The subset is maximal, i.e., it is not part of a larger subset with the property that each node can reach the other

- In the real world, a strongly connected component can reveal a hidden weakness!
  - For example, bank A takes out a loan from bank B, which makes B dependent on A: if A fails, B may also fail
  - It turns out that many banks in the US banking system are part of a SCC: if one fails, the others may fail like a stack of dominos

Fragility of financial networks



From Bech & Atalay (2008), as reprinted in Networks, Crowds, and Markets

- Network of loans among US financial institutions, revealing its strongly connected core
- This reveals a structural fragility in the financial system

Giant component collapse

- Consider a social network where the nodes are all people on Earth and there is an edge between two people if they are friends
    - Is this graph connected? Probably not! (Isolated tropical island)
    - But still, many people have friends far away
- Large social networks often have a giant component, which is a component that contains a significant fraction of the graph's nodes. This is a deliberately informal concept (not precise)!
    - If there exists a giant component, then there is almost always only one. Reasoning: if there were two, then adding just one edge between them would cause them to fuse, "collapsing" into one component.
- Real-world examples of collapsing giant components are often significant events and even catastrophic
    - When European explorers arrived in the Americas, this caused such a collapse, with dramatic effects on technology and disease (epidemics)

Consistency

Consistency

- A large data set will always have inconsistencies
  - There are many reasons for this, such as disconnections, real-world changes, and distance
- We distinguish two cases
  - Inconsistency in a network with partitions
    - CAP Theorem
  - Inconsistency in a connected network
    - Pluralistic ignorance
- Both can be partly corrected, but never completely!
  - By adding or improving information diffusion
  - Correction can be slow (diffusion) or fast (collapse)

CAP theorem

- The CAP theorem was conjectured by Eric Brewer at PODC in 2000 and proved by Seth Gilbert and Nancy Lynch in 2002
  - For an asynchronous network, it is impossible to implement an object that guarantees the following three properties in all fair executions:
    - Consistency: all operations are atomic (totally ordered)
    - Availability: every request eventually returns a result
    - Partition tolerance: any messages may be lost
  - We can only have two out of three of these properties!
- In a large data set, there will always be partitions
  - Then the CAP theorem says the following:
    - All requests that return will be correct, but some will not return (C + P)
    - All requests will return a reply but it can be wrong (A + P)
  - If we assume all partitions are eventually repaired, then all requests will return a correct answer (C+A) but it may take a very long time!

Pluralistic ignorance

- Pluralistic ignorance is when different parts of a connected network have inconsistent information
  - It can occur if communication in the network is limited

- In the real world, this typically happens in two situations
  - Repressive governments actively increase pluralistic ignorance
  - Information bubbles form in social networks because of the recommendation algorithm (maximize “active engagement”)

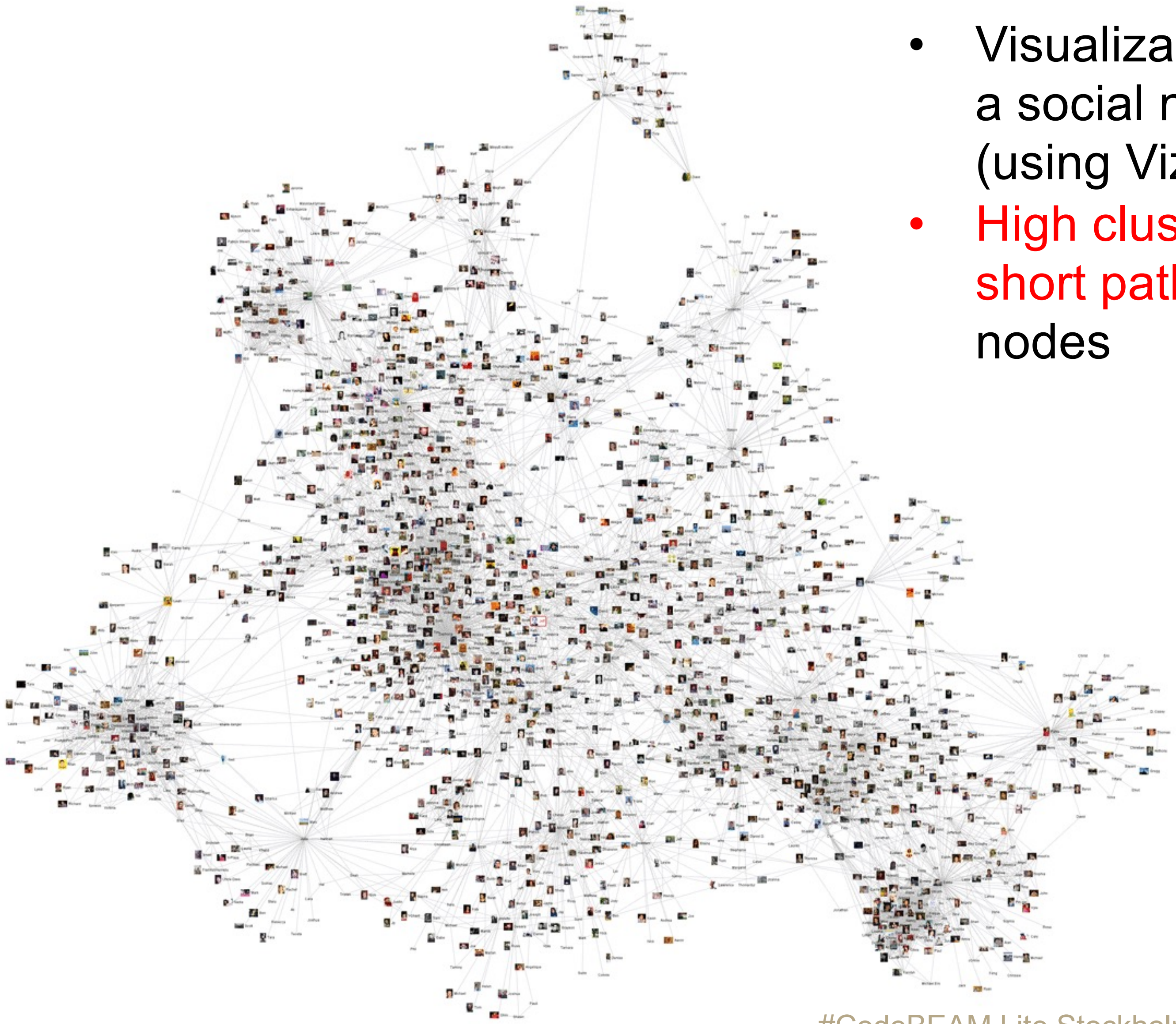
- Pluralistic ignorance may last for a very long time
  - Tightly-knit communities hinder the spread of new information
  - There are techniques to overcome this:
    - Make the information easier to transmit or more attractive
    - Convince a small number of key nodes in the network (influencers) to spread the information

Small-world networks

Small-world networks

- Large applications on the Internet will often have a few users whose behavior can significantly influence the large-scale system behavior
  - This is a general property of the real world, mirrored in the system
  - What is going on?
- Small-world graphs
  - The connectivity graph among the application's users will almost always be a small-world graph, which is defined with two properties:
    - Small average shortest path length
    - Large clustering coefficient
  - Navigation is often easy (user search using partial information)
    - There are both short links and long links
- Power-law structure
  - Fraction of nodes with k in-links is proportional to $1/k^2$
  - Many nodes have high popularity (heavy tail)

Example of a real SWN



- Visualization of a social network (using Vizster tool)
- High clustering with short paths between nodes

Conclusions

Conclusions

- Large systems must be scalable
  - If they are large then their exact size is dependent on external factors which are often highly fluctuating
- Building these systems is difficult
  - Going beyond what has been done before invalidates induction, and probability distributions are a dubious simplification
- Despite this, we know some useful principles
  - Three basic design rules for predictable overload behavior
  - Multilevel systems for multiple timescales
  - Reducing nondeterminism as much as possible
  - Important large system phenomena
- This is a snapshot of an ongoing project with PNSol
  - We are developing the ΔQSD system design paradigm in an ongoing collaboration with the company PNSol

The Christmas tree redux

- The complete Christmas tree does not exist yet (sorry!)
  - We are working on it in an ongoing project with PNSol and IOG

- Some references for more information on the topics of this talk:
  - Peter Van Roy, Neil Davies, Peter Thompson, and Seyed Hossein Haeri. *ΔQSD: Designing Systems with Predicable Latency at High Load.* HiPEAC tutorial, Jan. 18, 2023.
  - Seyed Hossein Haeri, Peter Thompson, Neil Davies, Peter Van Roy, Kevin Hammond, and James Chapman. *Mind Your Outcomes: The ΔQSD Paradigm for Quality-Centric Systems Development and Its Application to a Blockchain Case Study.* Computers 2022, 11, 45.
  - David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010.
  - Predictable Network Solutions, Ltd (PNSol). See: www.pnsol.com.
  - Peter Van Roy. *Why Time is Evil in Distributed Systems and What To Do About It.* Keynote talk, CodeBEAM 2019, Stockholm, Sweden, May 16, 2019.
  - Peter Van Roy. *Reflections on Scalability and Consistency*. Keynote talk, Workshop on Planetary Scale Distributed Systems (W-PSDS 2019), Lyon, France, Oct. 1, 2019.
  - Peter Van Roy, Seif Haridi, and Alexander Reinefeld. *Designing Robust and Adaptive Distributed Systems with Weakly Interacting Feedback Structures*. Research Report 2011-01, ICTEAM Institute, Université catholique de Louvain, Jan. 2011.
  - Peter Van Roy. *A Little Book of Insights*. Jan. 2023.