UCL Université catholique de Louvain

LIGHTKONE
Lightweight computation for networks at the edge

Reflections on scalability and consistency

Peter Van Roy

W-PSDS 2019, Lyon, France
Oct. 1, 2019

Overview

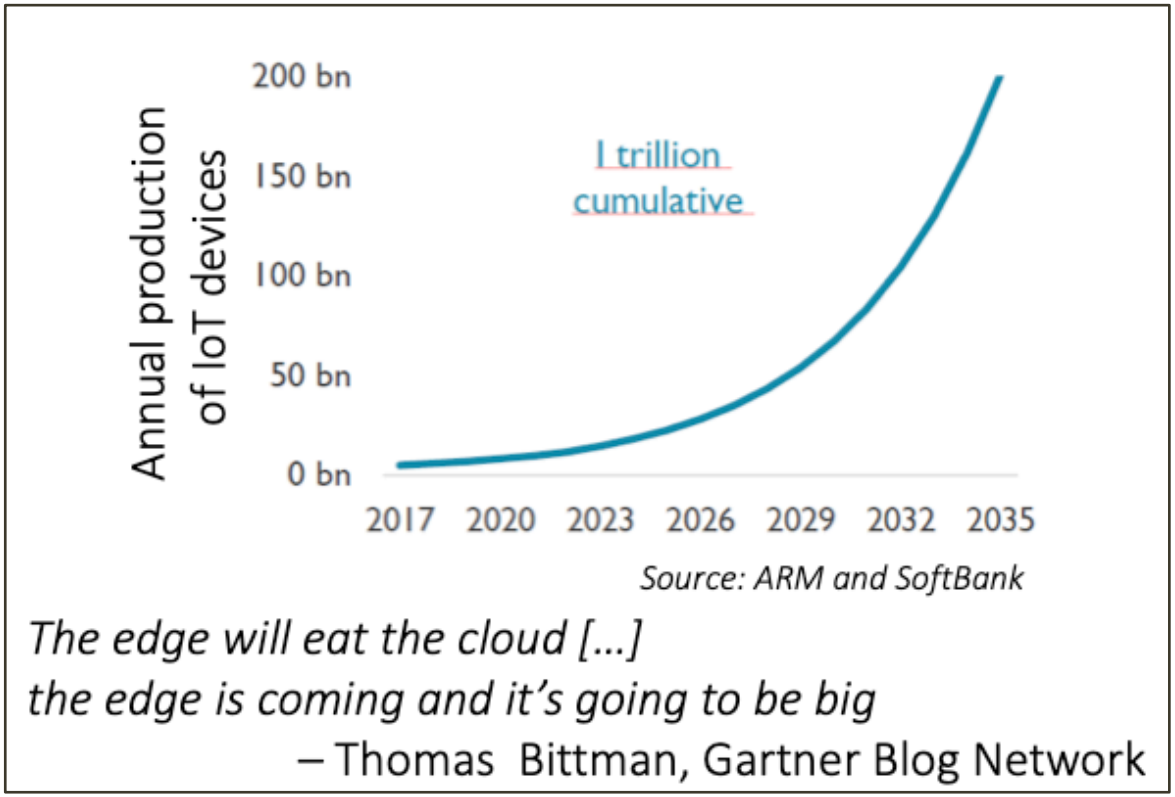
- Basic principles
  - Understanding scalability, Internet growth and IoT, CAP theorem
- Large-scale phenomena
  - Buridan's principle, black swans, real-world graphs, Heisenberg applications
- Building scalable systems
  - Autonomic computing and feedback structures, convergent data management, simplified distributed systems (confluence and linearity)
- Conclusions
  - Ongoing work on convergent data management and confluent distributed systems

Understanding scalability

- A system is scalable if it maintains a desirable property as the scale increases
  - Scale can be defined in several useful ways: number of nodes, size of data, number of users, geographic separation
  - We will use a rough definition where scale corresponds to total aggregate work (data, computation) done by a system
- When scale increases, strange things happen
  - This talk is not a standard technical talk
  - We will explain various phenomena that show up at high scale and give some approaches how to design systems to accommodate these phenomena
  - In my view, anyone who is serious about scalability should know about these phenomena and these techniques!
  - This talk is also intended to provoke discussion and solicit pointers to other work on scalability

Internet scale

- The Internet is an excellent testbed for scalability because it has been growing exponentially since around 1970 and still is
- Nowadays, this growth is concentrated at the edge:



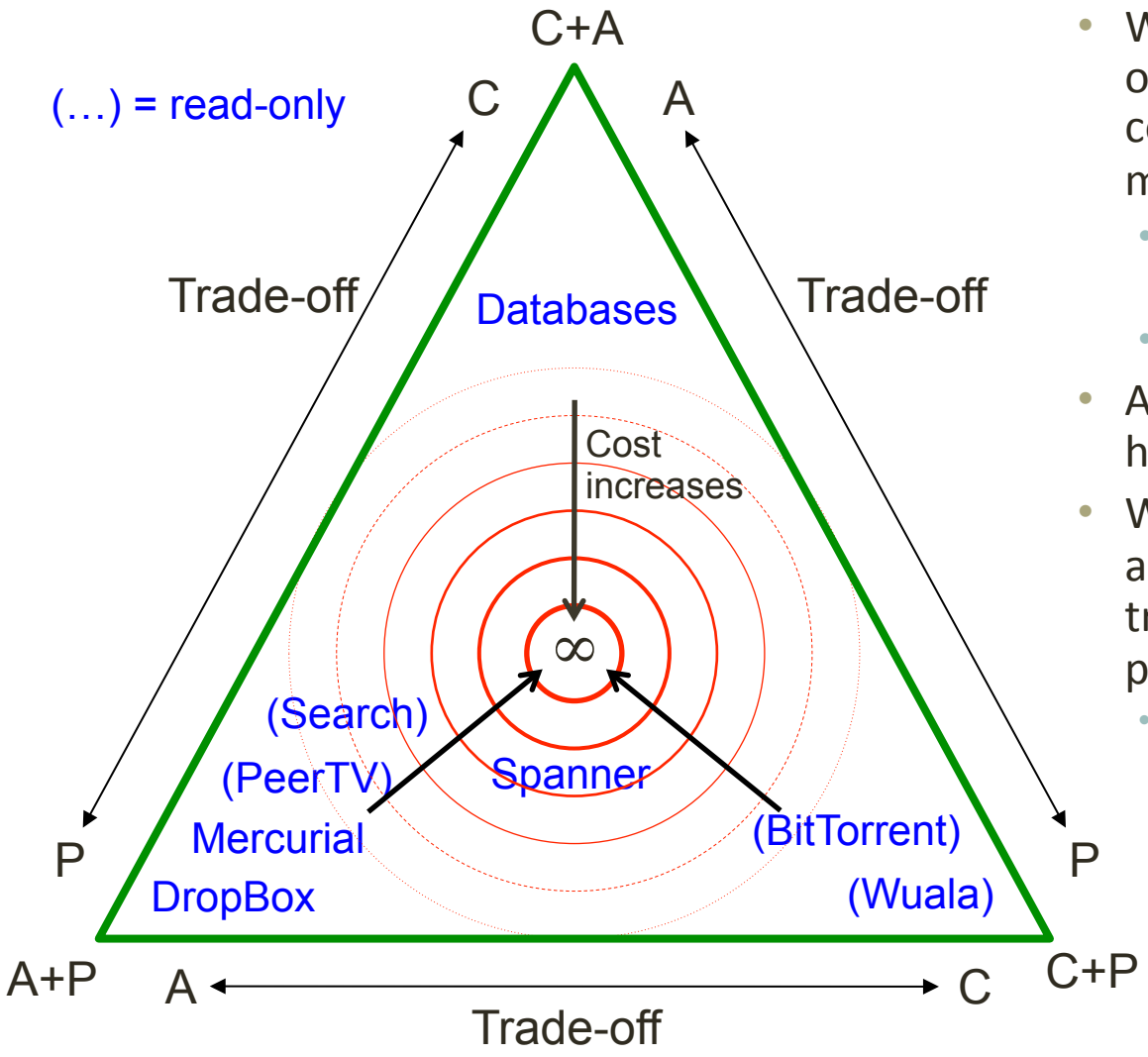
From talk of David Bol, RISC-V (Paris), Oct. 1-3, 2019

CAP theorem

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 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
- The CAP Theorem applies for all systems, at all levels of abstraction, and at all sizes
  - It can be applied in many places in the same system
  - The whole system is a rainbow of interacting instances of CAP

The CAP triangle



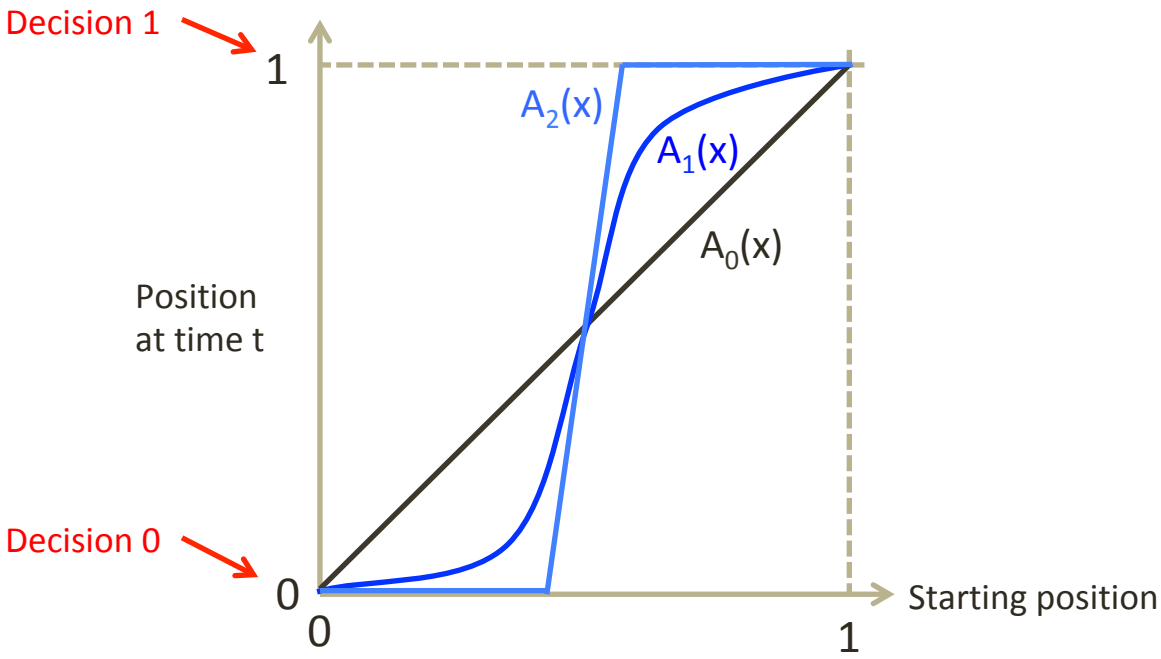
- We give an intuitive overview of the consequences of CAP by means of a CAP triangle
  - Cost increases toward the center
  - The center itself is empty!
- All parts of the CAP triangle have their uses
- We have arranged some applications around the triangle according to perceived functionality
  - Very little systematic study has been done about navigating in this triangle

Buridan’s principle

Buridan's principle

- 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
  - This principle has surprising consequences, as shown in the paper "Buridan's Principle", by 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
  - A car at an unguarded railroad crossing, the driver stops at the crossing and proceeds when it is safe. The driver must decide whether to wait for the train or to cross the tracks.
  - 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, because more information must be analyzed.

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

Relationship to scale

- Distributed systems often must make decisions
  - Distributed bank account: is the account positive or negative.
  - Voting system: who wins the vote.
- The input data is (approximately) continuous and can be distributed over the whole system
- To make the decision, more information / computation is required as the input data is closer to 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

Black swans

… 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)

Black swans

- Systems are designed by relying on induction, “what worked in the past will continue to work in the future”
  - It is very common to assume that induction will always work
- However, induction often has a built-in limit and fails beyond
  - Year 2000 Bug: it was “far away” but it has arrived
  - Dinosaurs and banks: “too big to fail” but they will 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 situations)

The two great "frauds"

- Systems obey inductive reasoning – *false!*
  - Past experience with systems is a bad guide for future systems, especially if the future system is going beyond the past one
- Systems obey probability distributions – *false!*
  - Probability distributions are introduced to simplify analysis, but often do not exist in reality
  - Assuming a probability distribution exists is a very strong assumption (frequency limit exists) and is very probably wrong
- Black swan: unexpected large events that falsify induction and are obvious in hindsight
  - Large systems are fundamentally irregular and must be designed to survive extreme cases
  - See "The Black Swan", by Nassim Nicholas Taleb (2010)

Degrees of increasing irregularity in a large ystem

1. Existence of a probability distribution
   - Statistical physics holds, all microstates have equal probability, behavior is thermodynamic (describable by macroscopic state variables)
   - Unfortunately, most simulations and models are stuck here!
2. Critical point
   - Minor fluctuations can be amplified without bounds
   - The limit of statistical physics
   - Many computing systems have critical points (garbage collectors, dynamic hash tables, wide-area routing, virtual memory)
3. No probability distribution exists ("Black Swans")
   - We know only the range of behavior, frequency limits do not exist
     - Dijkstra's demon: in a guarded command, all guards can be chosen
     - The program must be designed so that it works even if a demon makes the worst possible choices in each guarded command
   - Complex systems, program verification, distributed algorithmics

Real-world graphs

Real-world graphs

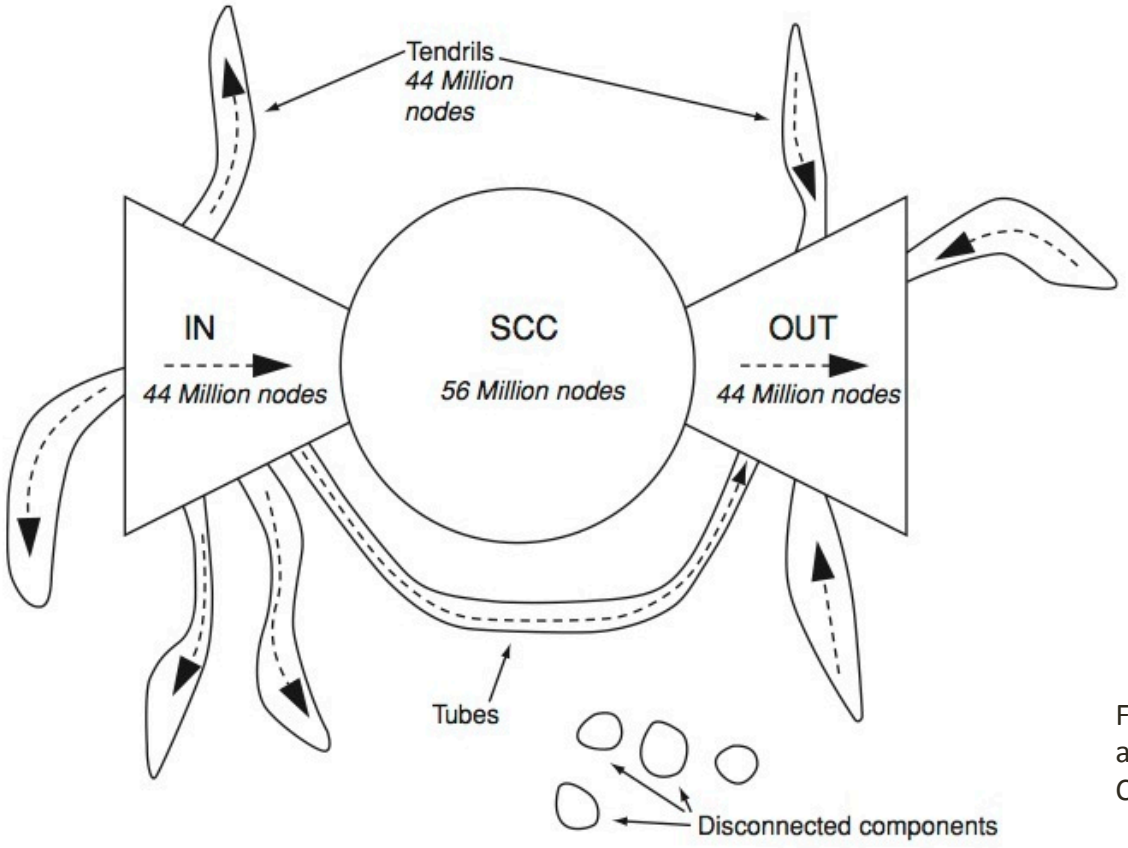
- Large applications on the Internet will often have large numbers of users whose behavior can significantly influence the large-scale behavior of the system
  - Especially important is information dissemination among users
- Small-world graphs
  - The connectivity graph among the application's users will almost always be a small-world graph: small average shortest path length, large clustering coefficient (more clustered than random, smaller paths than neighbor)
  - Navigation is often easy (user search using partial information)
- Power-law structure
  - Fraction of Web pages with k in-links is proportional to $1/k^2$
  - Consequence of information dissemination during formation

See « Networks, Crowds, and Markets », by David Easley and Jon Kleinberg (2010)

Consistency in real-world graphs

- CAP theorem states that consistency cannot be achieved for available, partition-tolerant systems
  - Users' information dissemination makes this more precise
- Limiting communication between users causes pluralistic ignorance, where different parts of the network have very different information, and this can last indefinitely long
- Limiting communication can cause sudden changes
  - Epidemic dissemination (synchronization, oscillation, stability)
  - Collapse of giant components (one connected component that contains a significant fraction of all nodes)
- Information content can be changed during dissemination
  - Cascades (herding), when decisions are made in sequential order
  - Tipping points, decisions need to convince a large initial group

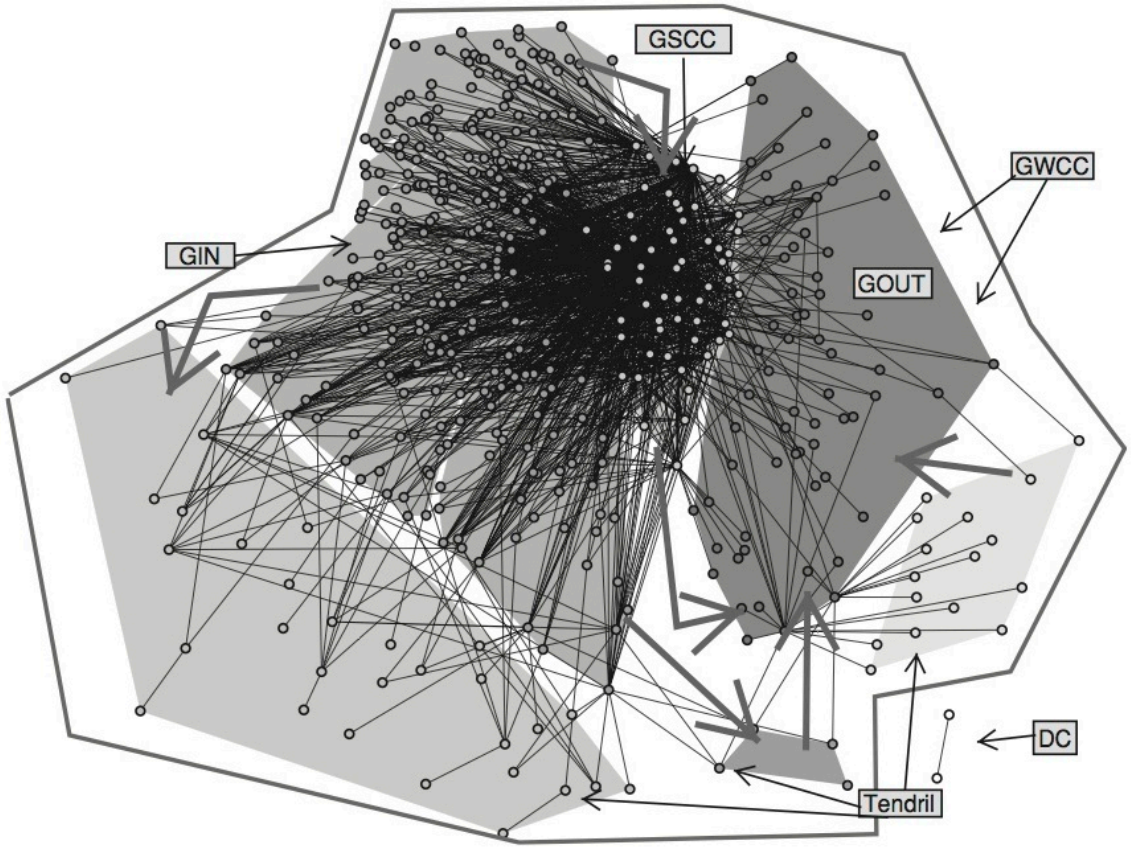
Example: Web bow-tie



From Broder et al (2000), as printed in Networks, Crowds, and Markets

- Large decentralized information networks such as the Web will often have a global structure with a central strongly-connected component (from 2000, but still valid today)
- The structure is maintained as the network continuously changes

Example: financial networks



From Bech & Atalay (2008), as printed 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

Heisenberg applications

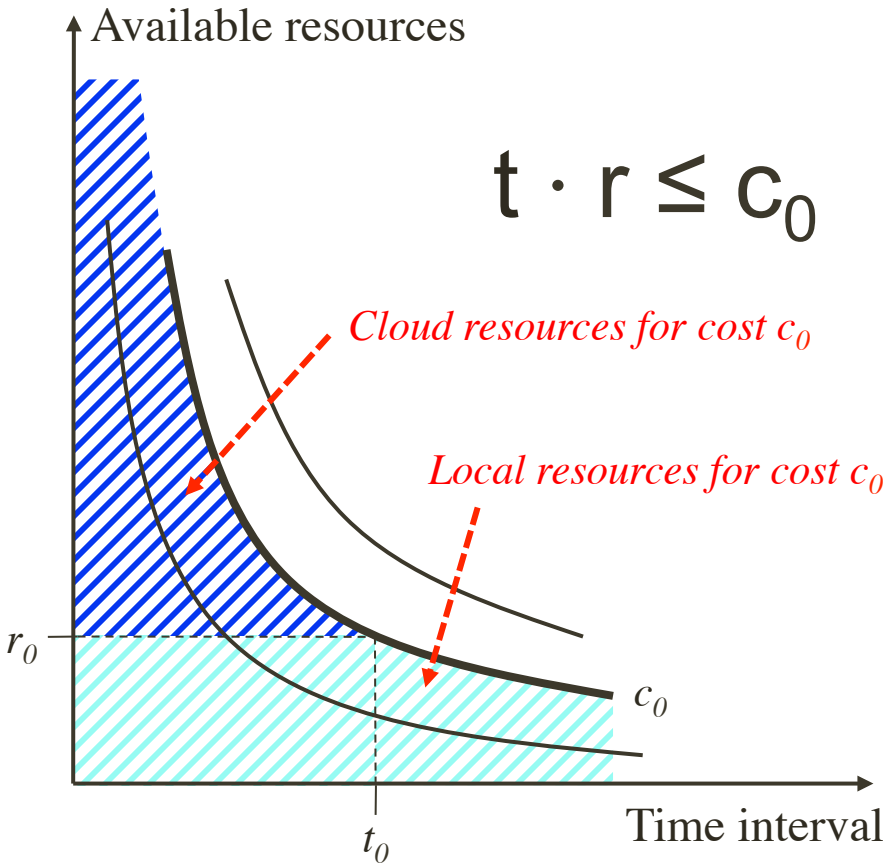
Many applications are bursty

- Many real-world applications require a lot of computational resources for very short time periods
  - For example, interactive applications require massive resources, but only when being used (real-time voice translation when you are speaking)
  - Your computer's CPU usage is bimodal: it is close to 0% most of the time, except when you are doing a compute-intensive task when it is close to 100%
- The solution to this is to provide elasticity
  - Cloud computing: pay only for the resources actually used
    - Economy of scale with many shared users
  - Internet of Things: power up only the nodes actually needed
    - Hardware (silicon) is cheap
- This enables a new kind of application based on burstiness

Computational Heisenberg Principle (1)

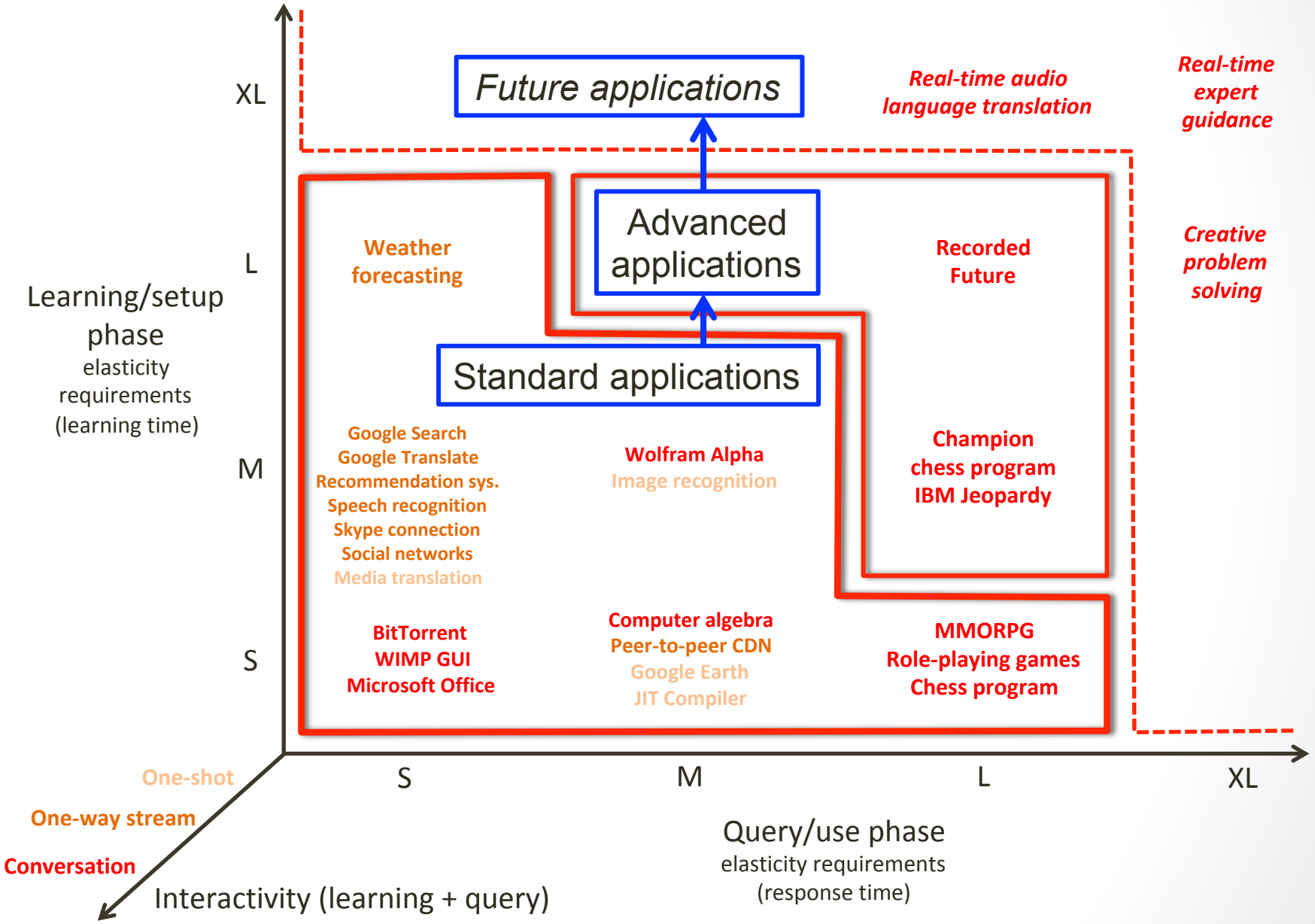
- A cloud has two key properties:
  - Pay per use: pay only for the resources actually used
  - Elasticity: ability to scale resource usage up (and down) rapidly
- For a fixed cost, as the time interval decreases more resources can be made available:
  - **For a given maximum cost, the product of resource amount and usage time is less than a constant**
  - Analogy with Heisenberg's Uncertainty Principle in physics: the product of uncertainty in time and uncertainty in energy is equal to (or greater than) a constant. This increases the probability of events that use arbitrarily high energies if the time period is short enough. As long as the high energies are less than the uncertainty, then they are allowed!
    - This is a property of the system itself, not a limitation of measurement!
    - $\Delta t \cdot \Delta E = c$ and $t_{allow} \leq \Delta t$ and $E_{allow} \leq \Delta E$ implies $t_{allow} \cdot E_{allow} \leq c$
- This opens the door to new applications that could not be done before

Computational Heisenberg Principle (2)



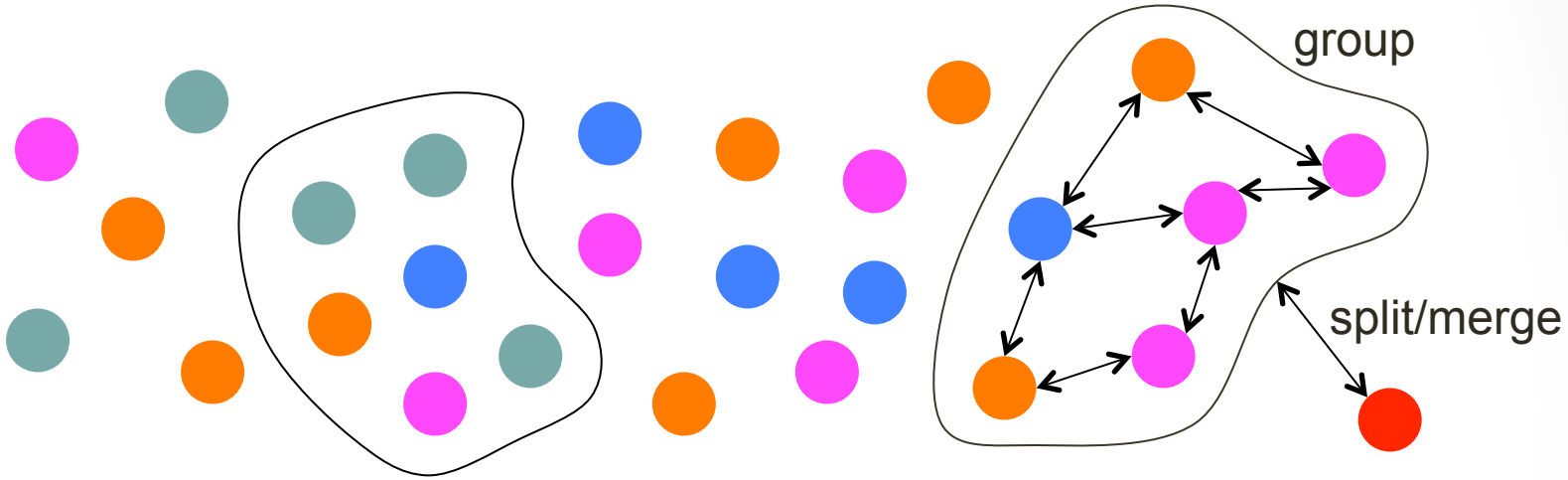
- For given fixed resource cost c_0, what kinds of applications can run?
- Before elasticity: all applications lived in **light blue area** which gives local resources for maximum cost c_0 ($r \leq r_0$)
- With elasticity: **dark blue area** becomes available for the same cost ($r > r_0$)
- The dark blue area is the home of Heisenberg applications
  - Cloud applications (e.g., big data)
  - IoT applications (e.g., data fusion)
- Using machine learning often leads to Heisenberg applications

Heisenberg application space



Building scalable systems

The starting point



- Every scalable design starts as independent pieces (P+A, no C)
- Nodes occasionally interact (add some C)
  → collaboration, emergence
  - Split protocol: what happens when a node leaves a group (may be abrupt)
  - Merge protocol: what happens when a node joins a group
- Many examples: biology, peer-to-peer, map-reduce, …

Building up to a real system

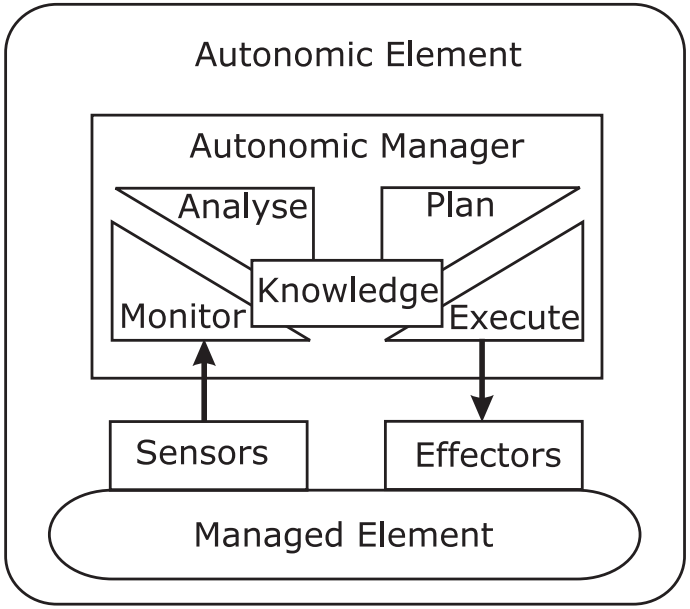
- We start with a decentralized system (P+A, no C)
  - The problem: how much C and how to add it?
- The rest of this section explores how to add C
- Control-oriented approach
  - Autonomic computing
  - Weakly interacting feedback structures
- Data-oriented approach
  - Convergent data management
  - LightKone reference architecture

See « Designing Robust and Adaptive Distributed Systems with Weakly Interacting Feedback Structures », by P. Van Roy, S. Haridi, and A. Reinefeld (2011)

See « LightKone Reference Architecture (LiRA) White Paper », by Ali Shoker *et al* (2019)

Weakly interacting feedback structures

Autonomic computing

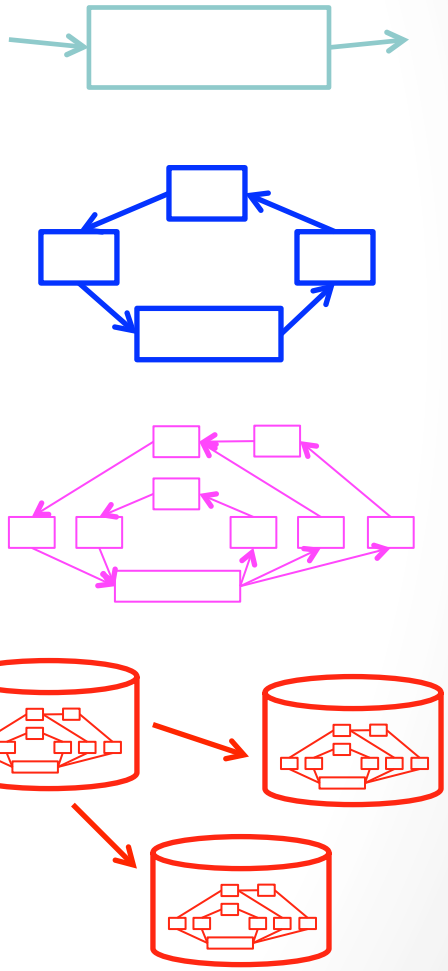


- Autonomic computing, initiated by IBM in 2001, aims to make computer systems self managing, to overcome the growing complexity of systems management as scale increases
- The basic building block of IBM's autonomic system is the MAPE-K feedback loop (Monitor − Analyze − Plan − Execute − Knowledge)
- We start with this approach, and we investigate how to build systems consisting of many interacting MAPE-K loops

A scalable architecture in four steps

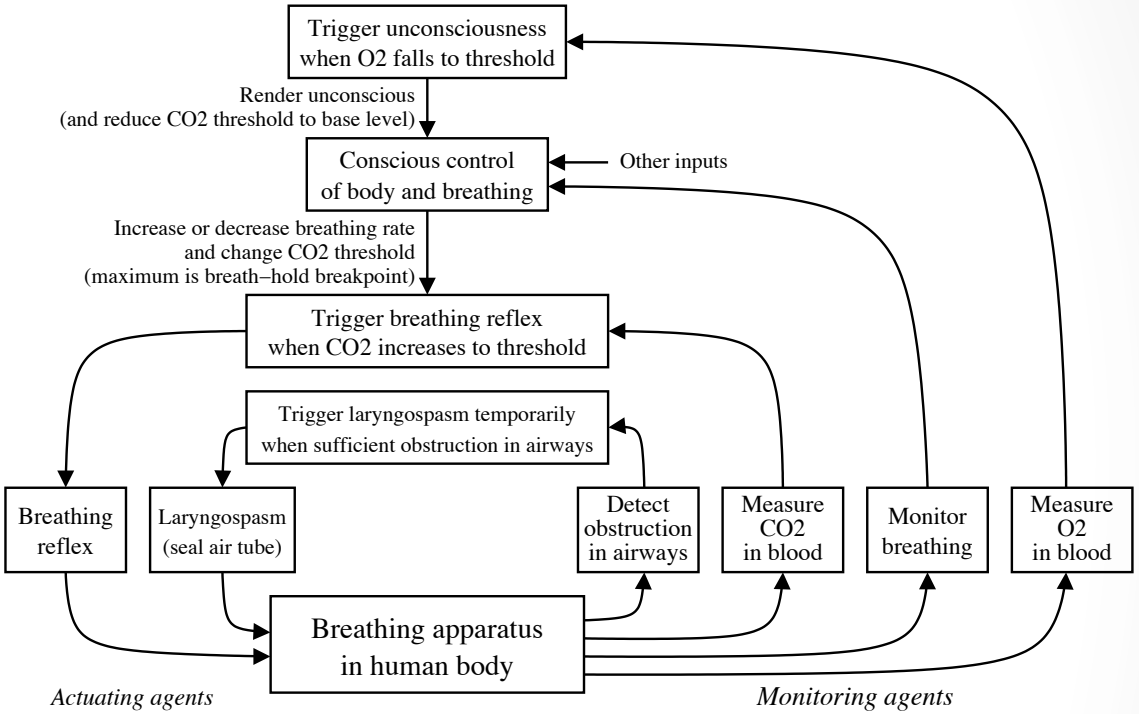
THE ADVENTURES OF SELFMAN

- Concurrent component
  - An active entity communicating with its neighbors through asynchronous messages
  - "Intelligence" concentrated in core components
- Single feedback loop (MAPE-K loop)
  - Manager, sensor, and effector components connected to a subsystem and continuously maintaining one local goal
- Feedback structure
  - A set of feedback loops that work together to maintain one global system property
- Weakly interacting feedback structures (WIFS)
  - The complete system is a conjunction of global properties, each maintained by one feedback structure
  - The feedback structures have dependencies based on the operating conditions



Human respiratory system

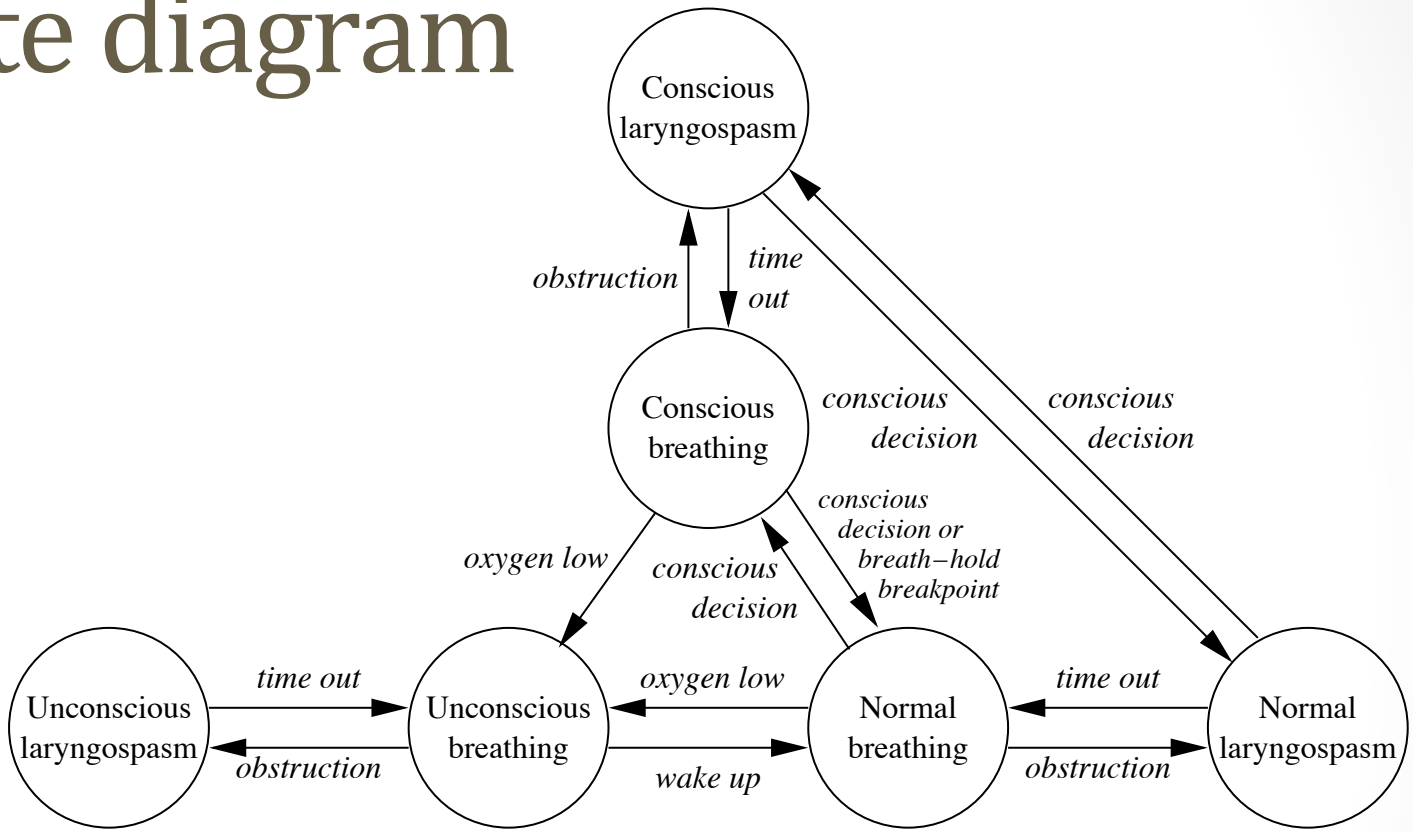
The operation of the human respiratory system is given as one feedback structure, inferred from a precise medical description of its behavior (see entry on "Drowning", Wikipedia)



Some design rules:

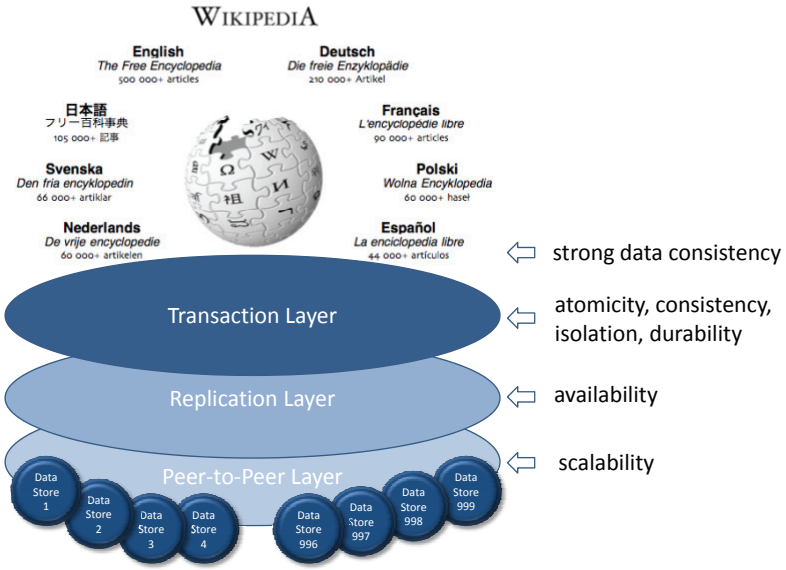
- Default behavior: rhythmic breathing reflex
- Complex component: conscious control can override and plan lifesaving actions
- Abstraction: conscious control does not need to know details of breathing reflex
- Fail-safe: conscious control can itself be overridden (falling unconscious)
- Time scales: laryngospasm is a quick action that interrupts slower breathing reflex

State diagram



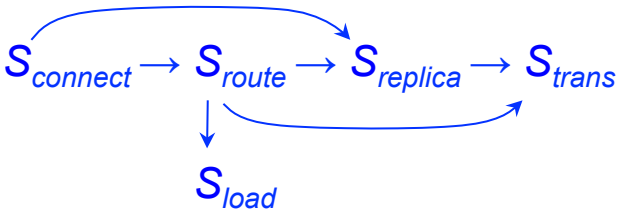
- The human respiratory system can be seen as a state diagram
- Dominant subset = active subset of feedback loops = state
  - At any time, one subset is active, depending on operating conditions
  - Each subset corresponds to a state in the state diagram

A self-managing key/value store: Scalaris



$$S_{scalaris} = S_{key\text{-}value} \wedge S_{connect} \wedge S_{route} \wedge S_{load} \wedge S_{replica} \wedge S_{trans}$$

The Scalaris specification is a conjunction of six properties. Each non-functional property is implemented by one feedback structure.



- Scalaris is a high-performance self-managing key/value store that provides transactions and is built on top of a structured overlay network
  - A major result of the SELFMAN project (www.ist-selfman.org)
  - 4000 read-modify-write transactions/second on two dual-core Intel Xeon 2.66 GHz
- Scalaris has five WIFS: connectivity ($S_{connect}$), routing (S_{route}), load balancing (S_{load}), replica management ($S_{replica}$), and transaction management (S_{trans})

Convergent data management

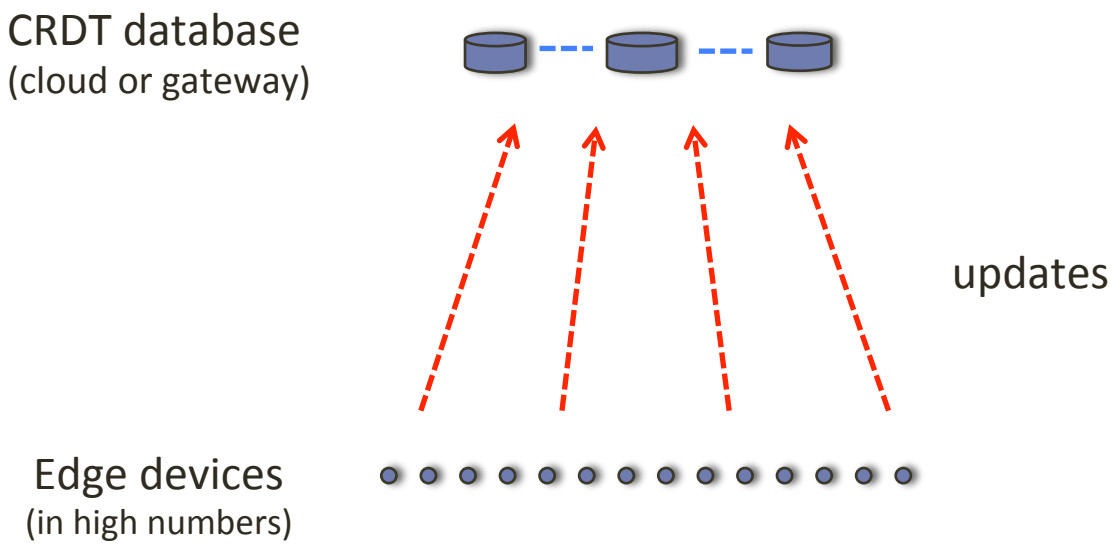
Convergent data management

- Maintaining adequate consistency and performance are two of the most important issues when system scale increases
  - Whereas autonomic computing is an operation-oriented approach, let us now take a data-oriented approach
- Assume our system has a database that is used to centralize the data management
  - We automate the management of "truth" in the database by a system to continuously update the database with information coming from the edge
  - To simplify consistency management, we use convergent data structures that tolerate message and node failures and need very little synchronization (CRDTs)

CRDTs

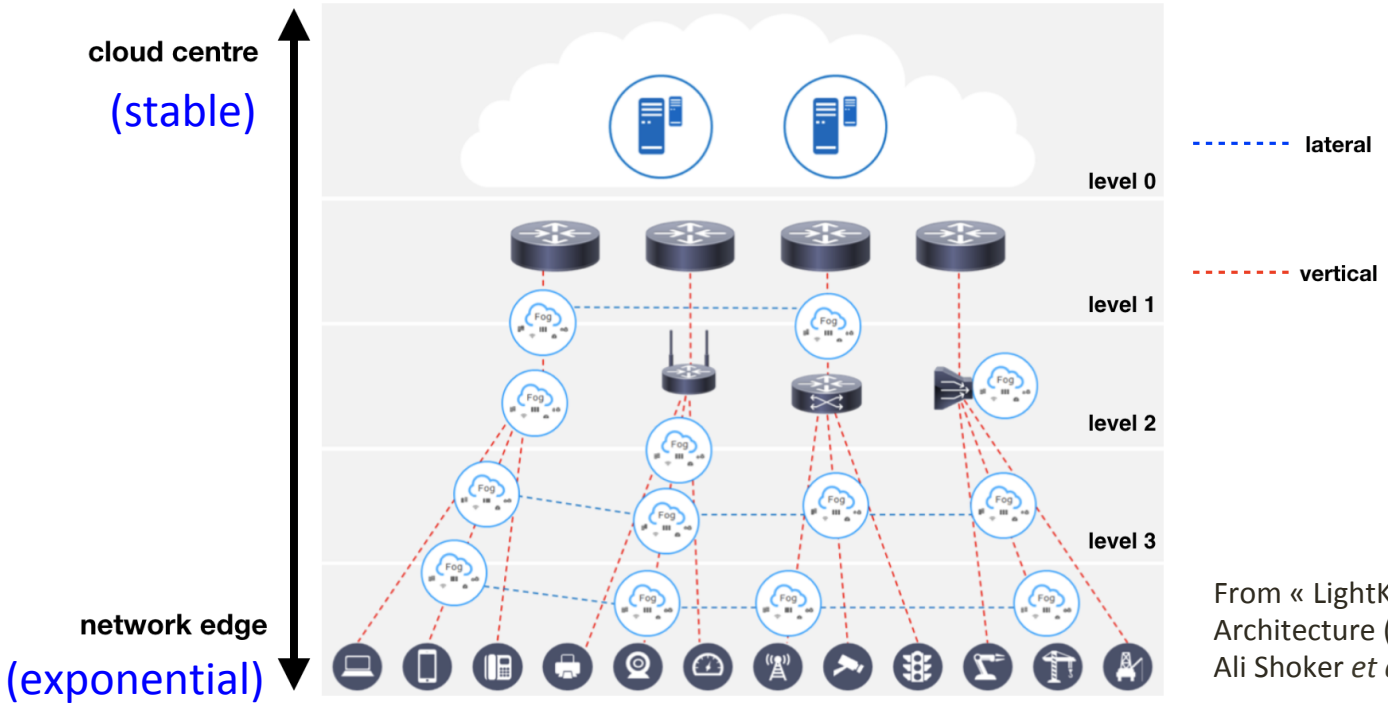
- A Conflict-Free Replicated Data Type is a replicated data structure that maintains consistency between replicas with a very weak synchronization protocol
  - It satisfies Strong Eventual Consistency (SEC): n replicas that receive the same updates (in any order) have equivalent state
  - Internally, all replicas are collecting information monotonically and are always converging to their resulting state
  - Synchronization between replicas is eventual replica-to-replica communication
- Many practical CRDTs exist and are widely used in commercial systems with millions of users
  - Counters, sets, maps, graphs, etc.
- CRDTs are well-suited for convergent data management

Basic convergent data scenario



- The CRDT database receives periodic updates (raw or aggregated) from all edge devices
- Messages may be lost, reordered, or repeated without harm
- The CRDT database will always be converging to the truth at the edge; staleness depends on the update protocol and delay

Lateral data sharing in LiRA



From « LightKone Reference Architecture (LiRA) White Paper », Ali Shoker *et al* (2019)

- LiRA (LightKone Reference Architecture) proposes an architecture with convergent data management, with artefacts (AntidoteDB, Achlys, Legion)
- As the edge continues to grow exponentially, vertical data sharing (red lines) is extended for convergent data management
- In addition, lateral data sharing (blue lines) is added to do convergent data management between devices at the same level

Just-Right Consistency (JRC)

- JRC generalizes convergent data management for application invariants

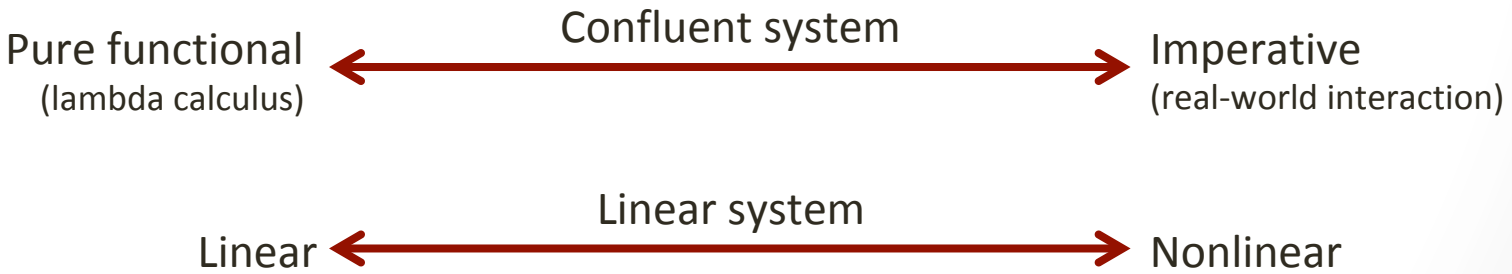
- Different parts of an application need different consistency levels
  - No single model is best: synchronous models are safest, but asynchronous ones are fast and tolerate partitions
  - We want to use synchronous and asynchronous together and get the best of both worlds

- JRC classifies application invariants according to CAP:
  - « Choose any two: Consistency, Availability, Partition-tolerance »
  - AP-compatible patterns: CRDTs with concurrent updates, causal order of operations, highly-available transactions
  - CAP-sensitive pattern: stable precondition before concurrent update (can be verified with tool support, e.g., CEC tool)
  - All other operations must be done sequentially

Simplified distributed systems

Simplifying distributed systems

- Distributed systems have concurrency, message latency, partial failure, and real-world interaction. They are quite difficult to understand and develop in their full generality.
- A useful approach is to focus on simpler kinds of distributed systems
  - The general form can be seen as the simpler kind with small additions
  - Following this approach can greatly help designing distributed systems
- We show two useful kinds of simplified distributed systems
  - Each kind satisfies a strong property that greatly helps development, and the general form can be attained with small additions

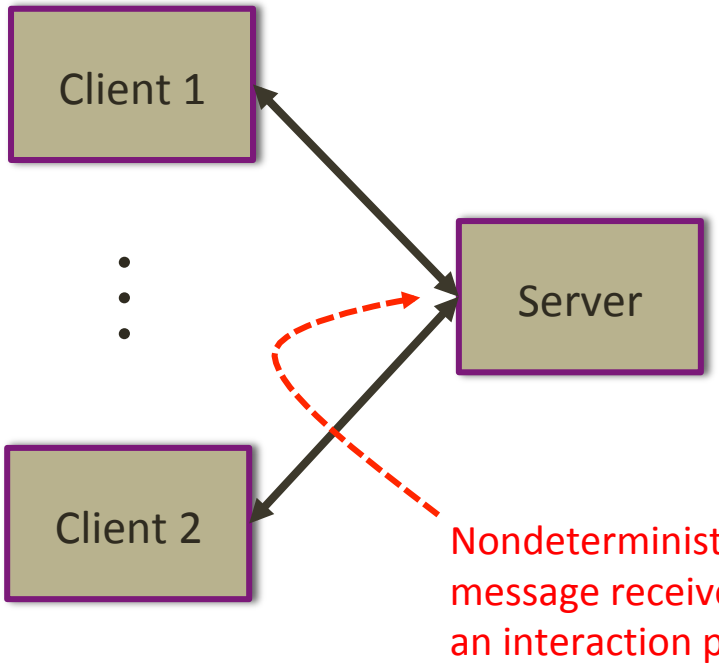


Confluent systems principle

- Functional programming based on λ calculus is confluent:
  - Church-Rosser: Given an initial expression, the final result of a reduction is the same for all reduction orders (up to renaming)
  - A functional program can be seen as a network of concurrent agents, each executing in its own thread. Church-Rosser means that for all scheduler choices, the result is the same.
- We make this program distributed by placing each subexpression on its own node
  - This requires one extra reduction rule, a mobility rule, to colocate expressions on a node for reduction. Church-Rosser still holds.
- The limitation is that it is syntactically known from which subexpression each agent's next input will come
  - For general distributed systems, we need to add interaction points, which depend on reduction order and express real-world interaction

See « Why Time is Evil in Distributed Systems and what to do about it », by Peter Van Roy, invited talk, CodeBEAM 2019, May 16, 2019

Confluent distributed systems



Pseudocode of client/server

```
local s p in
  node p=newport(s) server(state,s) end
  node client(state1,p) end
  node client(state2,p) end
  … /* as many clients as we need */
end

fun client(state,p)
  send(query(state),p)
  client(f_c(state),p)
end

fun server(state,s)
   case s of q|t then
       server(f_s(q,state),t)
   [] nil then
       nil
   end
end
```

One interaction point

- A simple example of this approach is the client/server: this program is completely functional except for one interaction point, namely where the server accepts an incoming message from any client
- The general approach is to write most of the program with distributed functional concurrency and add interaction points where needed

Linear systems principle

- Another useful kind of distributed system is the linear system
  - Linear systems are widely used in mechanical and electrical engineering, but are less used in informatics, because computer programs are inherently discrete and hence nonlinear
  - However, distributed systems are more and more being used in tight connection with the real world, to monitor and control real world systems. These distributed systems could take advantage of linearity, just like in other engineering disciplines.
- The world is a combination of linearity and nonlinearity
  - Linearity = independent parts = whole equals the sum of the parts
  - Nonlinearity = interacting parts = whole is more than the sum of the parts
- Linear systems are much easier to analyze quantitatively than nonlinear ones
  - Because in linear systems, the parts can be analyzed separately and then combined (superposition principle, compositional systems)
  - In addition, some nonlinear systems can be analyzed qualitatively (using a geometric approach). See "Nonlinear Dynamics and Chaos", by S. Strogatz (1994).
  - Nonlinearity cannot be eliminated entirely; often the system is critically based on it
    - For example, intelligence (complex components, machine learning) is often nonlinear
  - That's why biological systems are made of *weakly* interacting subsystems

Linear distributed systems

- Large distributed systems can often be mostly linear
  - This is especially true if the distributed system is connected tightly to the real world (monitoring, analyzing, controlling), e.g., in digital twins
    - Basic physical quantities are additive (mass, force, momentum, energy)
- They can't be completely linear, though
  - Because we need nonlinearity for most nontrivial behavior
  - Discrete decisions (e.g., Buridan's principle) are nonlinear
  - Real-world graph effects are nonlinear
- We should add nonlinearity where needed but no more
  - Today's distributed systems are far too nonlinear and discontinuous
  - They should be mostly linear with small amounts of nonlinearity added where needed
  - This also simplifies resilience, since linearity implies high redundancy

Conclusions

Conclusions

LIGHTKONE
Lightweight computation for networks at the edge

- Internet scale continues to increase, so scalability remains an important research topic
- Building large-scale systems is difficult because new and unusual phenomena appear at each new scale
  - Buridan's principle, black swans, real-world graph phenomena
- We give an overview of some of these phenomena and of some design principles that we have identified
  - Heisenberg applications, weakly interacting feedback structures, convergent data management
- This is part of ongoing work on understanding scalability
  - Convergent data management is ongoing (LightKone project)
  - Confluent distributed systems is ongoing (Van Roy and Haeri)
  - We welcome your reactions and all pointers to related work