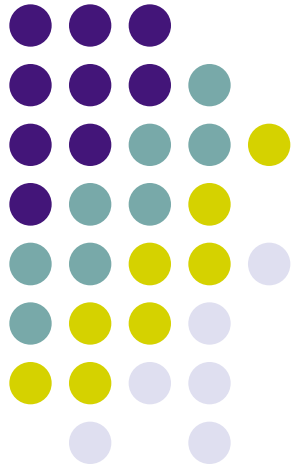
Designing robust distributed systems with weakly interacting feedback structures

EPITA

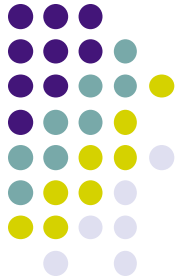
April 24, 2013

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain
Louvain-la-Neuve, Belgium

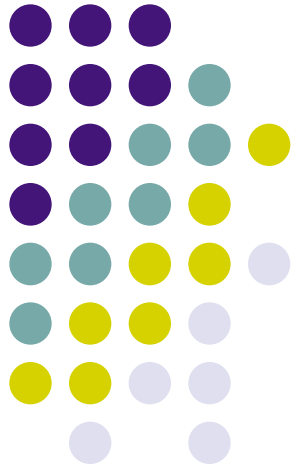


Overview

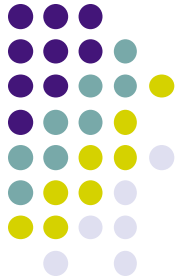


- As Internet services become larger, they become more complex and their environment becomes more hostile
  - Partial failures, software errors, communication problems, churn, attacks
  - Problems due to global behavior (oscillations, traffic jams, multicast storms, thundering herds, chaotic fluctuations, thrashing, cascading failures)
- How can we design Internet services to provide predictable behavior in such conditions?
  - Motivating examples from biology (and some well-designed computing systems)
  - Proposed architecture for scalable services as a set of weakly interacting feedback structures with dependencies
- Preliminary evaluation based on Scalaris key/value store (SELFMAN project)
  - Scalaris provides high-performance transactions on a structured overlay network
  - Scalaris contains five feedback structures and their dependencies: connectivity, routing, load balancing, replication, and transactions
- We are currently formalizing this approach and tying it to existing quantitative approaches

Motivating examples from biology and computing

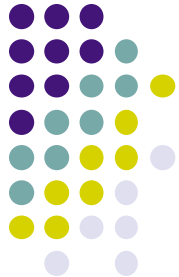


Motivating examples

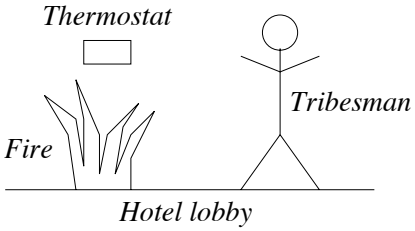


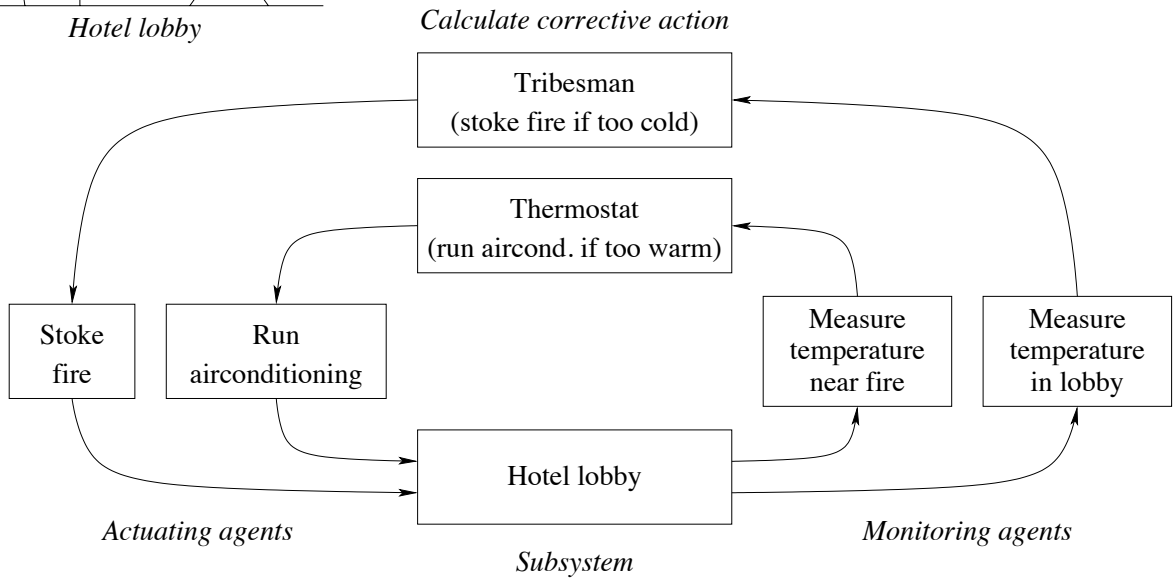
- Many systems exist that survive in hostile environments
  - Biological organisms
  - Some computing systems
  - Human organizations
- It is a good idea to study these systems to derive general design principles
- We give five examples to introduce the main ideas
  - Hotel lobby example → Debugging of feedback structures
  - Human respiratory system → Design rules, state diagram
  - TCP protocol operation → Systems with many parts
  - Human endocrine system → Concurrent component model
  - Human organizations → Design patterns for feedback structures

Simple example: hotel lobby (from [Wiener 1948])



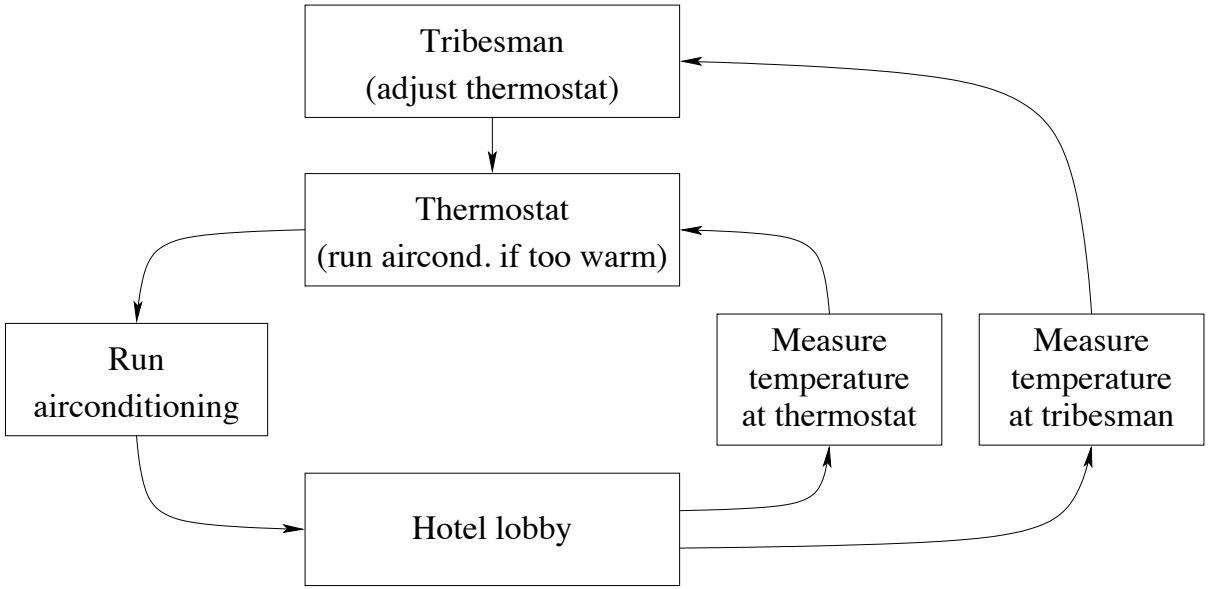
- Two loops interacting through a common subsystem (stigmergy)





- This is unstable!
  - The tribesman stokes the fire but gets colder and colder because the airconditioning works harder and harder
- Wiener leaves the fix as homework for the reader (!)
- One possible solution: outer loop (tribesman) controls the other by simply adjusting the thermostat
  - One loop controls the other

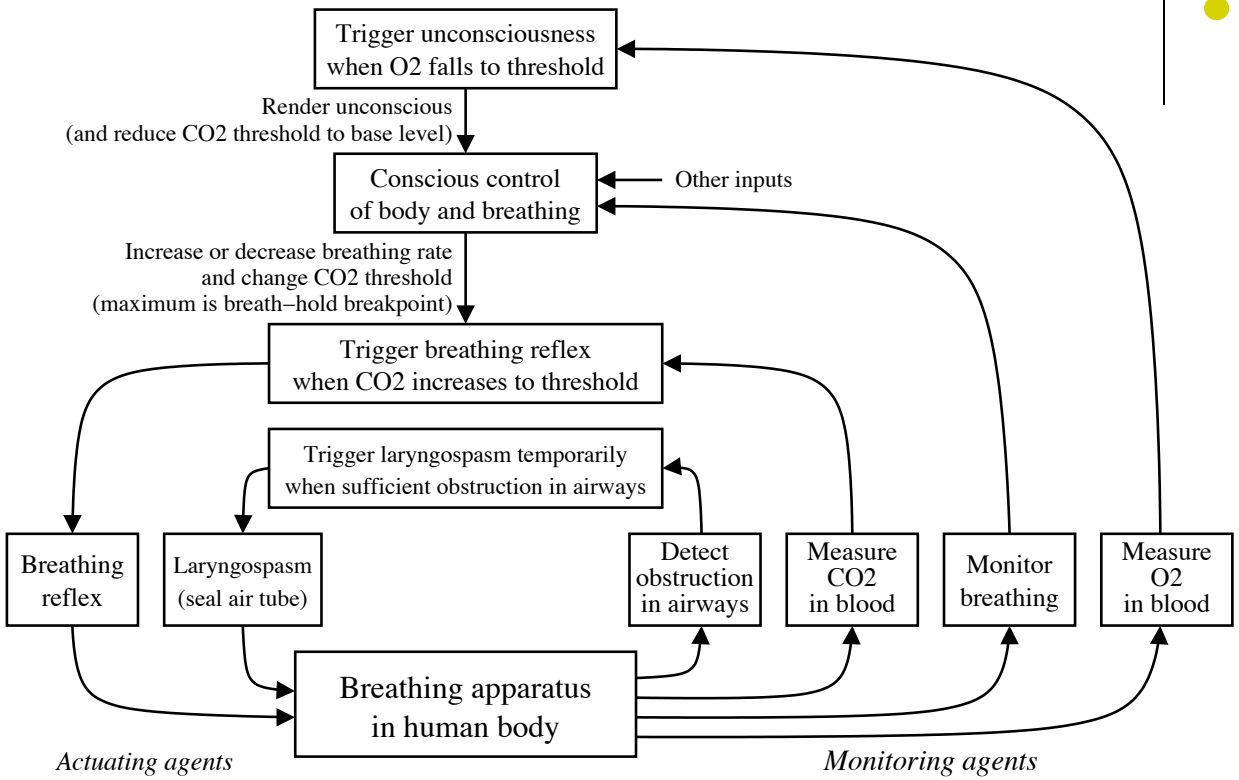
Hotel lobby solution



- Instead of stoking a fire, the tribesman simply adjusts the thermostat. The resulting system is stable.
- This uses management (one loop controls another) instead of stigmergy (two loops interact through the environment)
- Design pattern: use the system, don't try to bypass it

Human respiratory system

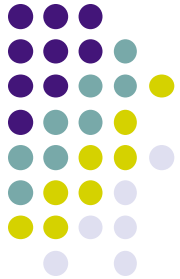
The operation of the human respiratory system is given as one feedback structure, inferred from a precise medical description of its behavior



Some design rules:

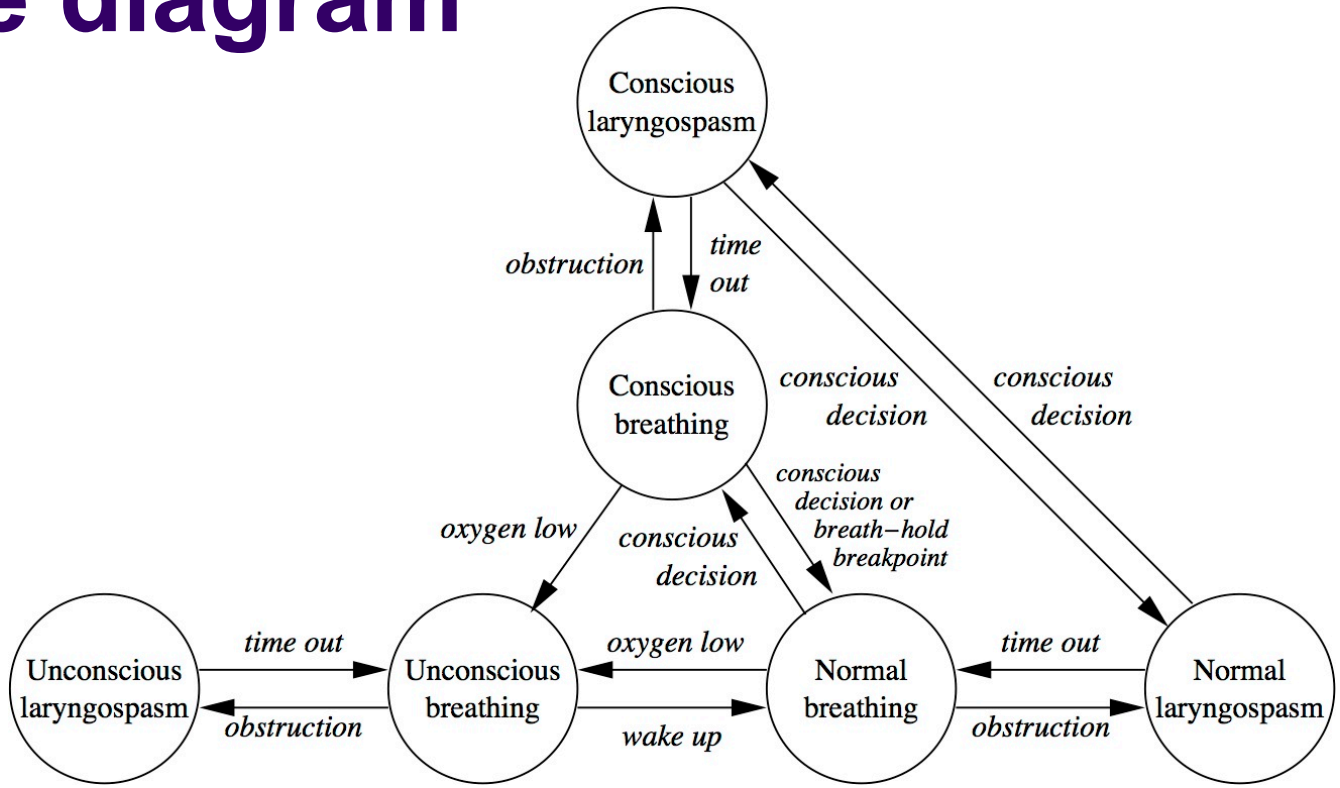
- Default behavior: rhythmic breathing reflex
- Complex component: conscious control can override and plan lifesaving actions
- Abstraction: conscious control does not need to know details of breathing reflex
- Fail-safe: conscious control can itself be overridden (falling unconscious)
- Time scales: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

Discussion of respiratory system



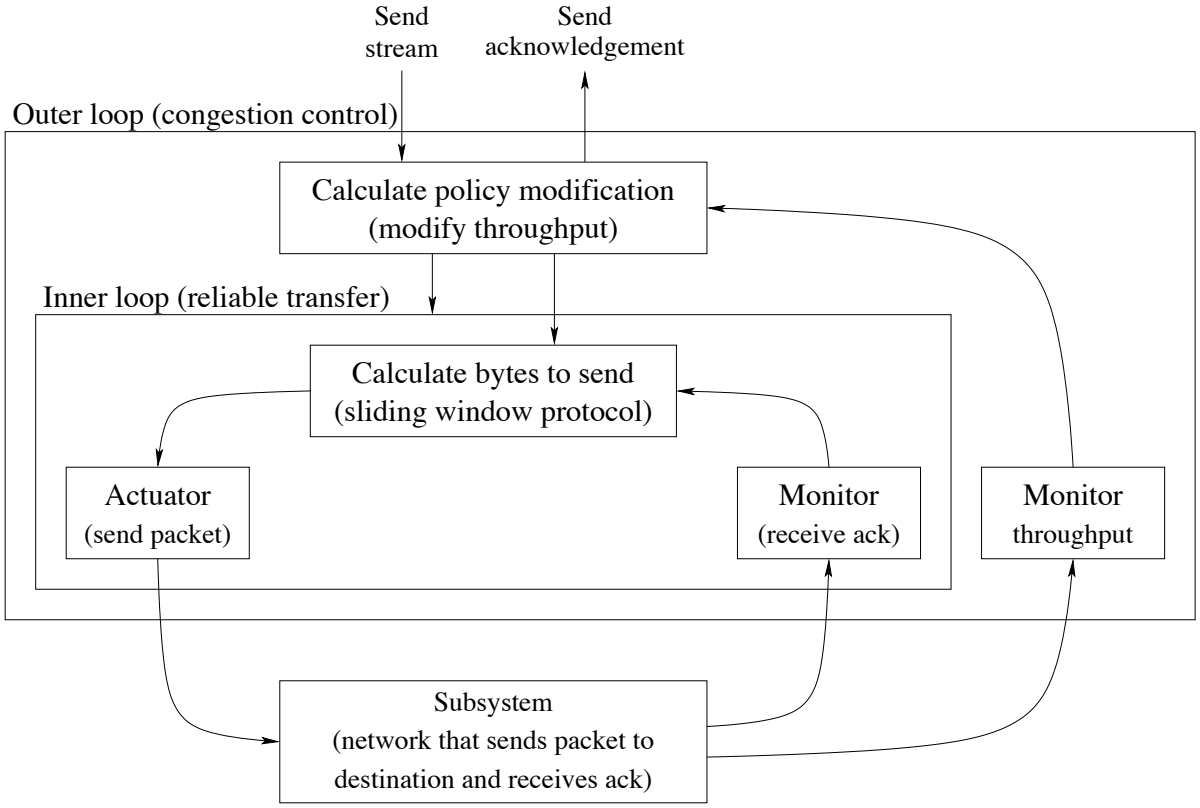
- Four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious)
  - This design is derived from a precise textual medical description [Wikipedia 2006: Entry "Drowning"]
- Holding your breath can have two effects
  - Breath-hold threshold is reached first and breathing reflex happens
  - O_2 threshold is reached first and you fall unconscious, which reestablishes the normal breathing reflex
- Some plausible design rules inferred from this system
  - Conscious control is sandwiched in between two simpler loops: the breathing reflex provides abstraction (consciousness does not have to understand details of breathing) and falling unconscious provides protection against instability
  - Conscious control is a powerful problem solver but it needs to be held in check

Respiratory system state diagram



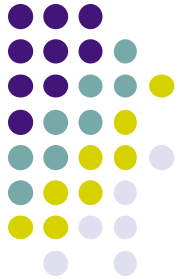
- The behavior of the human respiratory system modeled as a state diagram
- Dominant subset = active subset of feedback loops = state
  - At any time, one subset is active, depending on operating conditions
  - Each subset corresponds to a state in the state diagram

TCP as feedback structures



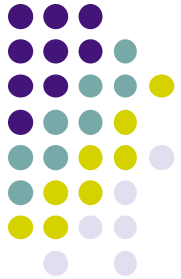
- This example shows a reliable byte stream protocol with congestion control (a variant of TCP)
  - This diagram is for the sending side
- The congestion control loop manages the reliable transfer loop
  - By changing the sliding window's buffer size
- With *n* connections there are *n* feedback structures interacting through a shared network (stigmergy)
  - This is an example of a system with many WIFS
  - Each FS has its own state

Human endocrine system



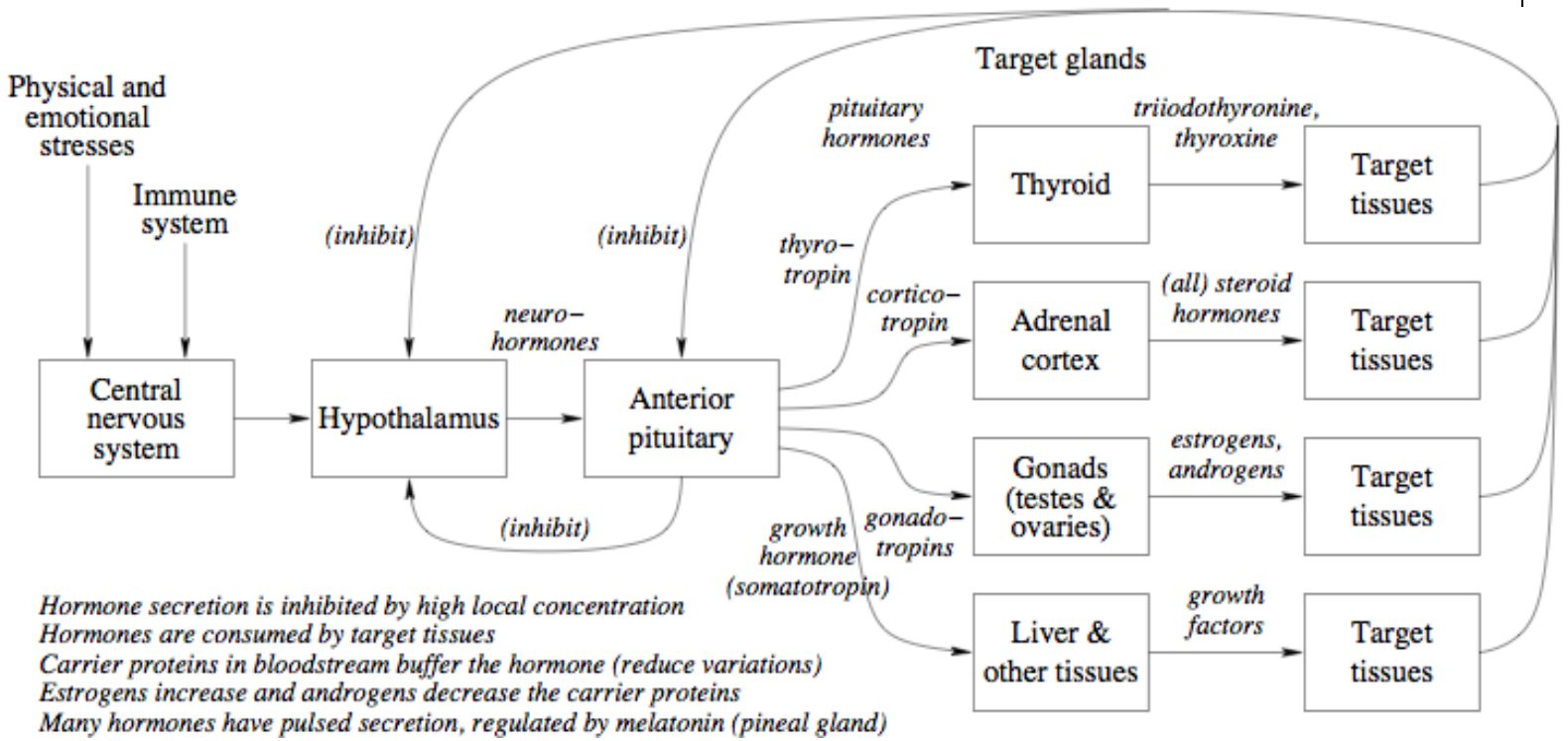
- The endocrine system regulates many quantities in the human body
- It uses chemical messengers called hormones which are secreted by specialized glands and which exercise their action at a distance, using the blood stream as a diffusion channel
- By studying the endocrine system, we can obtain insights in how to build large-scale self-regulating distributed systems

Feedback loops in the endocrine system



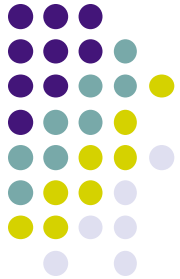
- There are many feedback loops and systems of interacting feedback loops in the endocrine system
  - Provides homeostasis (stability) and reaction to stresses
- Much regulation is done by simple negative feedback loops
  - Glucose level in blood is regulated by hormones glucagon & insulin. In the pancreas, A cells secrete glucagon and B cells secrete insulin. Increase in glucose in blood causes decrease in glucagon and increase in insulin. These hormones act on the liver, which releases glucose in the bloodstream.
  - Calcium level in blood is regulated by parathyroid hormone (parathormone) and calcitonine (also in opposite directions), which act on the bone
- More complex regulatory mechanisms exist, e.g., hypothalamus-pituitary-target organ axis
- There is interaction between nervous transmission and hormonal transmission

Hypothalamus-pituitary-target organ axis (endocrine system)

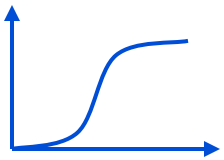


- Two superimposed groups of negative feedback loops, a third short negative loop, a fourth loop from the central nervous system [Encyclopaedia Britannica 2005]
- This diagram shows only the main components and their interactions; there are many more parts giving a much more complex full system

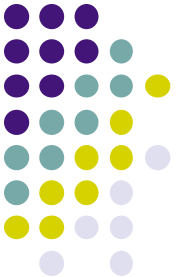
Discussion of endocrine system



- This system is quite complex
  - Many interacting feedback loops, many "short circuits", many special cases, much interaction with other systems (nervous, immune)
    - Negative feedback for most, also saturation (logistic curve)
    - Evolution is not always a parsimonious designer!
      - Only criterion: it has to work
  - Several feedback loops are channeled through a single point, the hypothalamus-pituitary complex in the brain
    - So that the central nervous system can manage these loops
    - Different time scales are used: the loops are slow; the central nervous system is fast



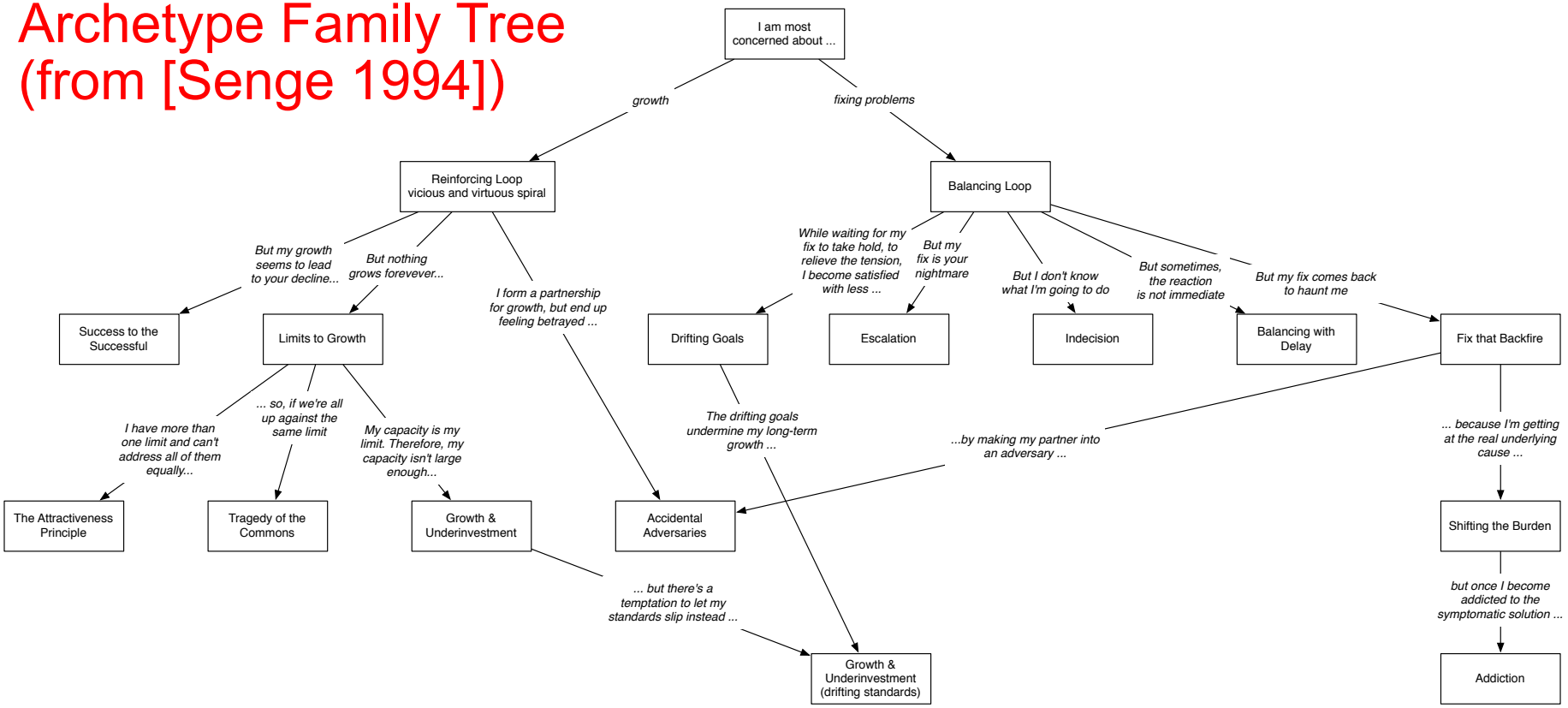
Computational architecture of human endocrine system



- Local and global components
  - Local: gland, organ, or clumps of cells
  - Global (diffuse): large part of the body
- Point-to-point and broadcast channels
  - Fast point-to-point: nerve fiber, e.g., from spinal chord to muscle
  - Slower broadcast: hormone diffused by blood circulation
    - With buffering (reducing variations): carrier proteins
- Regulatory mechanisms can be modeled by interactions between components and channels
  - There are often intermediate links
  - Abstraction (encapsulation) is almost always approximate

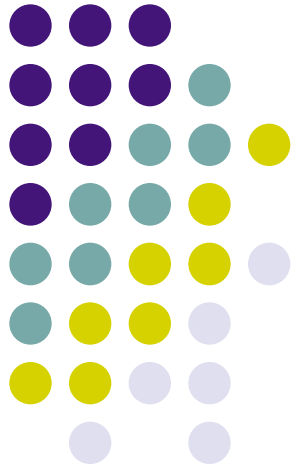
Design patterns for feedback structures

Archetype Family Tree (from [Senge 1994])

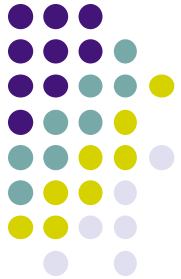


- We can arrange feedback structures in a tree according to their relationships and the problems they solve

Designing scalable systems

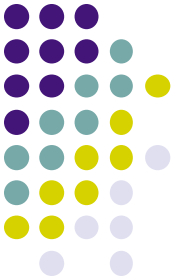


Designing scalable systems



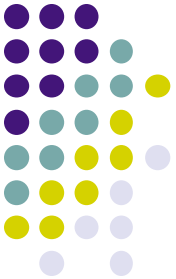
- Essential ingredients
  - Design principles inferred from existing working systems and validated subsequently
  - The CAP theorem is an essential tool
    - It holds at all scales and all levels of abstraction
- First step
  - The default is a set of independent parts
  - We add coordination between these parts
  - It is important to add as little coordination as possible
- Next step: weakly interacting feedback structures

The CAP theorem



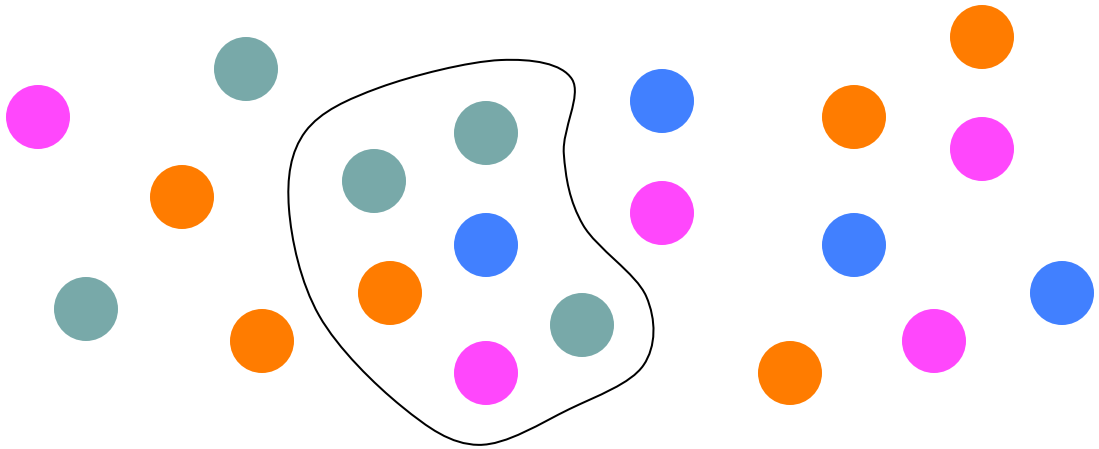
- The CAP theorem is an essential tool for any scalable system design
  - The CAP theorem was conjectured by Eric Brewer at PODC in 2000 and proved by Seth Gilbert and Nancy Lynch in 2002
- For an asynchronous network, it is impossible to implement an object that guarantees the following properties in 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

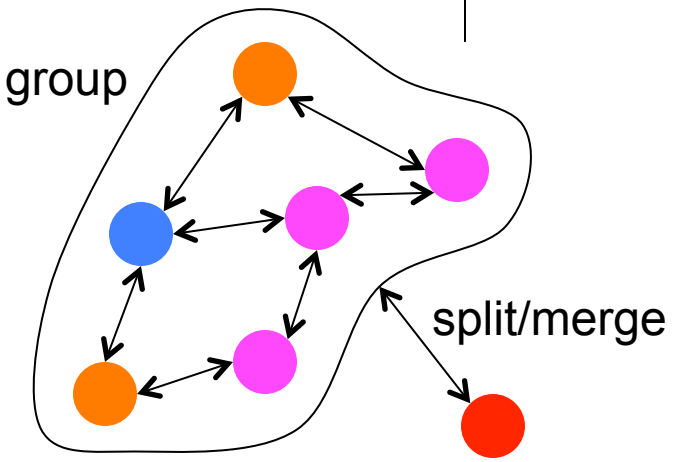
Designing with CAP



- C is hard to achieve → (P+A, no C) is the default
  - Consistency requires global coordination
- Avoid needing C if possible
  - We can achieve robustness (P) and performance (A)
    - DropBox and Web cache give P and A, but not C
    - Wuala and BitTorrent are read-only, achieve C easily
    - Mercurial is consistent if connected (C+A), but is still usable if disconnected (P+A)
- But if we really need C
  - Give up A → Waiting sometimes needed
  - Give up P → Fragile system
    - Distributed database guarantees C but will block if there is a partition
  - Accept weaker C → Eventual consistency
- We can have our cake and eat it too, if we pay the price
  - Highly reliable communication channels and fault tolerance
  - We get C and A, and we "seem" to get P as well (actually, we just have less partitions)
    - Scalaris, Beernet: peer-to-peer with majority consensus (Paxos) gives robustness
    - Cassandra: run on cloud, not peer-to-peer (does not support loose coupling)

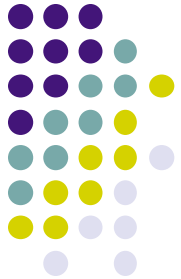
The default is a set of independent parts





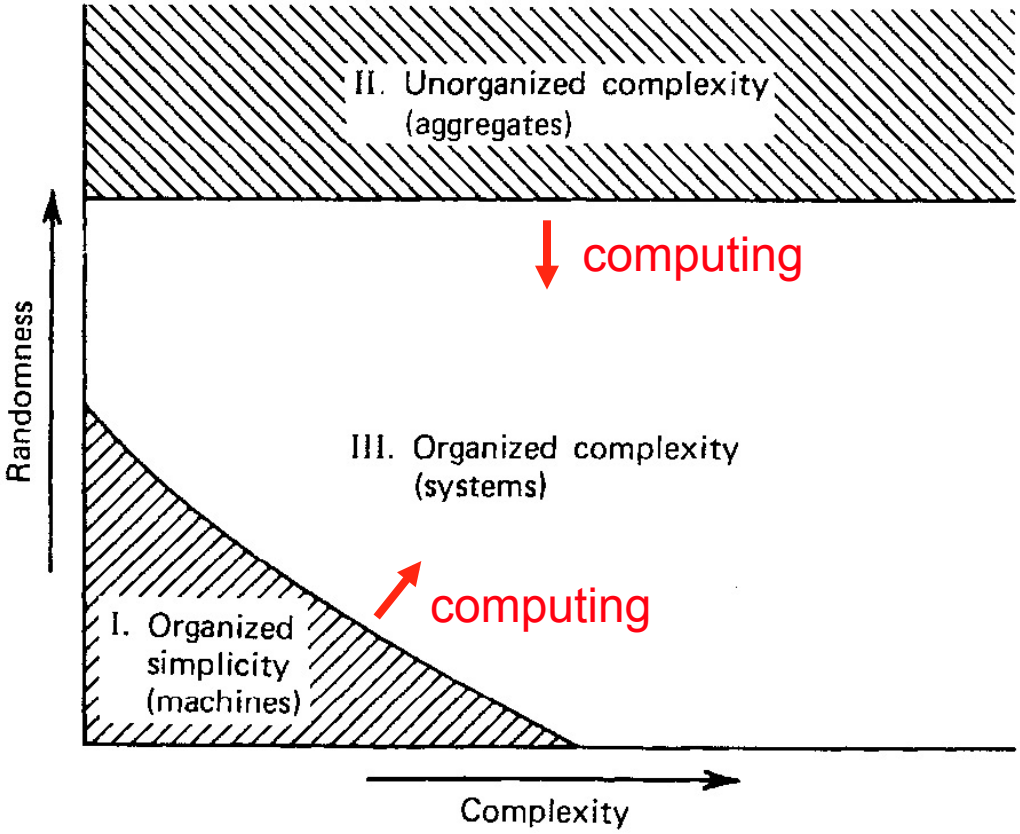
- Every scalable design starts as a decentralized system (P+A, no C)
  - A system of independent parts
- Nodes occasionally interact (add some C) → collaboration, emergence
  - Split protocol: what happens when a node leaves a group (may be abrupt)
  - Merge protocol: what happens when a node joins a group
- Merge is based on data coherence and may need input from highest level
- Many examples: biology, peer-to-peer, map-reduce, gas/liquid/solid, …

Mostly independent parts



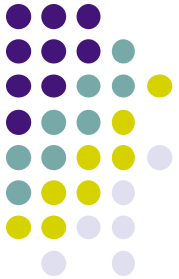
- Large systems consist of independent parts with weak interactions
  - Gas in a box: molecules mostly independent, occasional interaction when two molecules collide.
  - Peer-to-peer network: peers mostly independent, occasional interaction between neighbors only. Can provide efficient and robust communication and storage infrastructure (see later).
  - Gossip algorithm: nodes mostly independent, occasional interaction between random pairs. Can efficiently solve many global problems such as diffusion, search, aggregation, monitoring, and topology management.
- This seems to be a general principle
  - Systems with many parts that interact strongly are avoided by nature

Types of systems



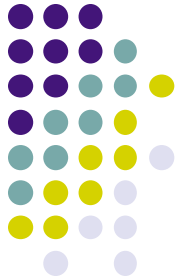
- This diagram is from [Weinberg 1977] *An Introduction to General Systems Thinking*
- The discipline of computing is pushing the boundaries of the two shaded areas inwards
- Software development and computational science are the vanguards of system theory
- However, there seems to be something inherently unpleasant about the white area in the middle
  - It is extremely difficult to analyze systems with many strongly interacting parts; science has barely touched it
  - Even biological organisms avoid it (they are mostly decomposable)

Adding consistency/ coordination

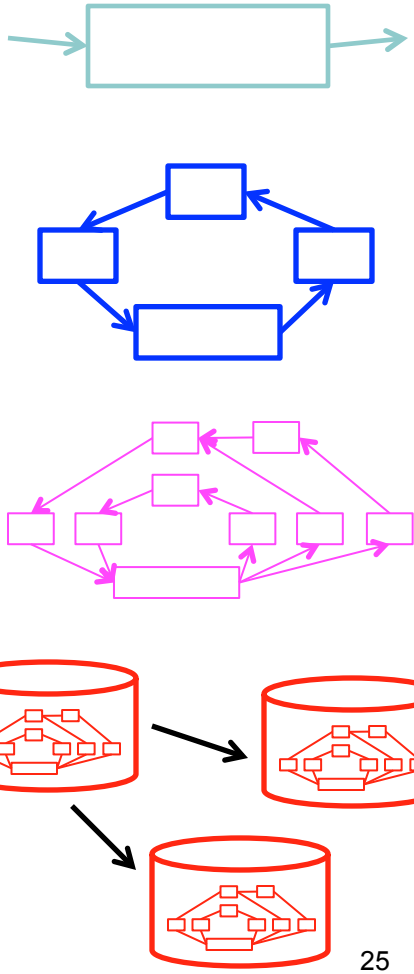


- We start with a decentralized system (P+A, no C)
  - How much C do we need and how do we add it?
  - General principle: as little as possible (weak interaction)
- The rest of the talk explores how to add C
- Main design principle:
  weakly interacting feedback structures
- We validate the approach on a real system
  - Scalaris, a transactional key/value store

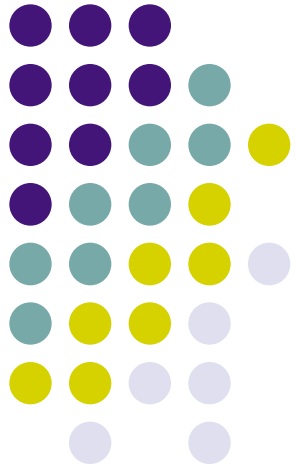
A scalable architecture in four steps



- Concurrent component
  - An active entity communicating with its neighbors through asynchronous messages
  - Intelligence is concentrated in complex components
- Feedback loop
  - Monitor, corrector, and actuator 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
  - 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

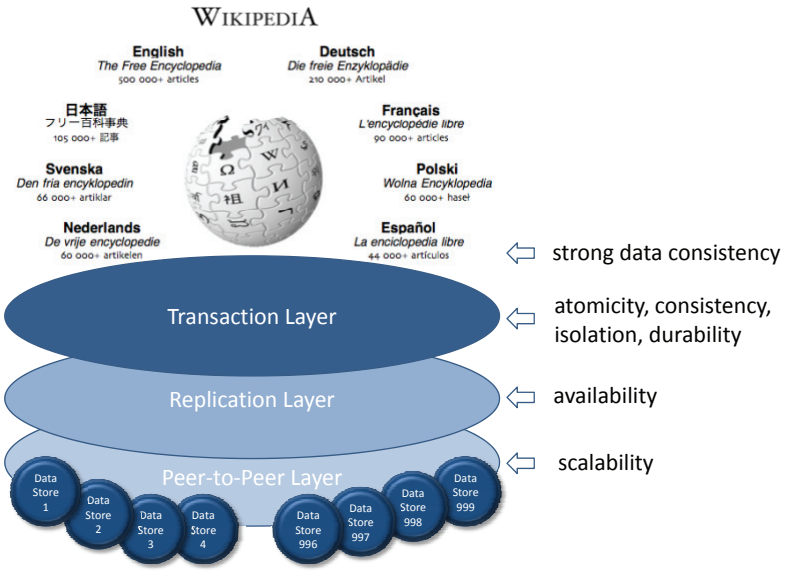


Scalaris with feedback structures



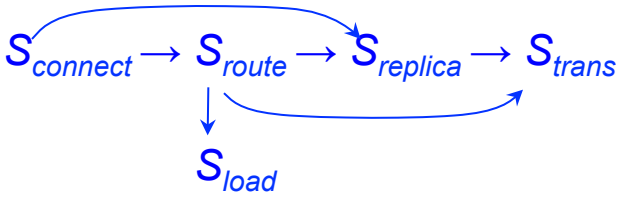
THE ADVENTURES OF SELFMAN

A peer-to-peer key/value Store: Scalaris



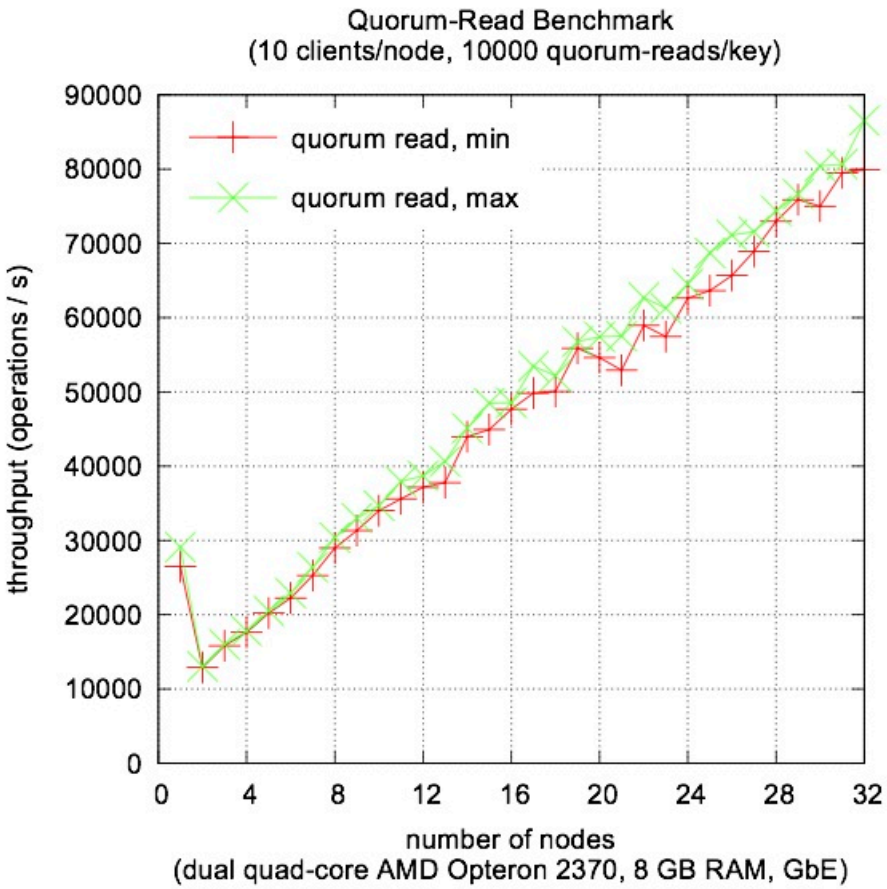
$S_{scalaris} = S_{key\text{-}value} \wedge S_{connect} \wedge S_{route} \wedge S_{load} \wedge S_{replica} \wedge S_{trans}$

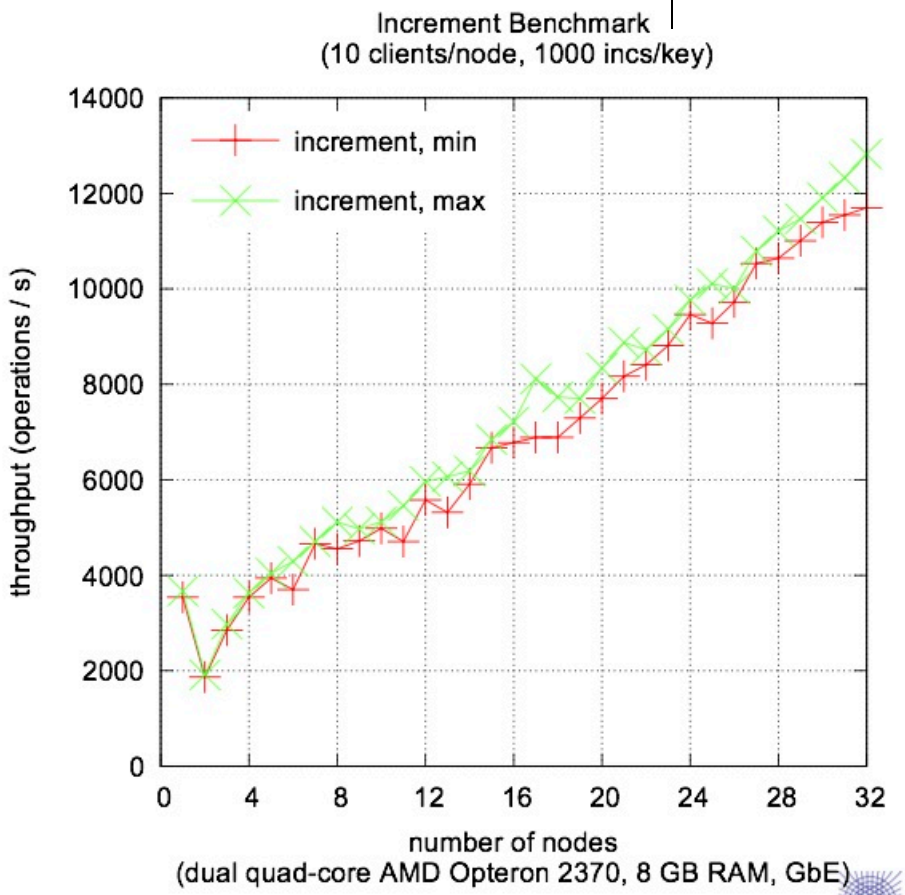
The Scalaris specification is a conjunction of six properties. Each non-functional property is implemented by one feedback structure.



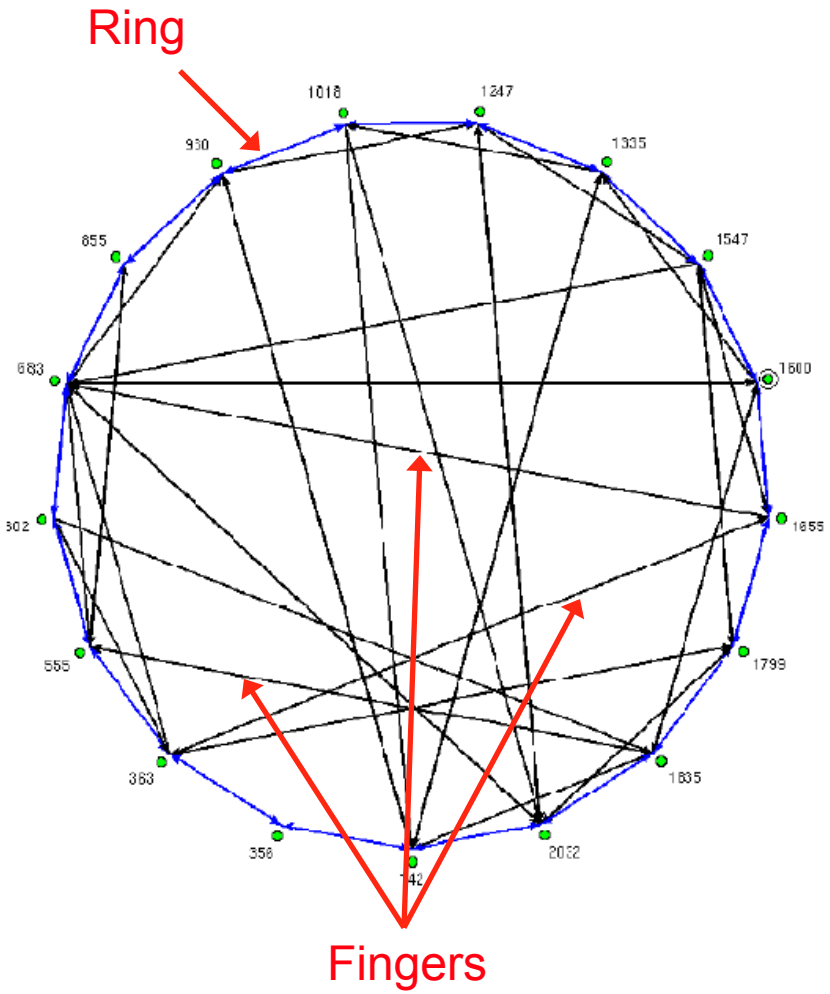
- Scalaris is a high-performance self-managing key/value store that provides transactions and is built on top of a structured overlay network
  - A major result of the European SELFMAN project (www.ist-selfman.org)
  - 4000 read-modify-write transactions per second on two dual-core Intel Xeon at 2.66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

Scalaris scalability



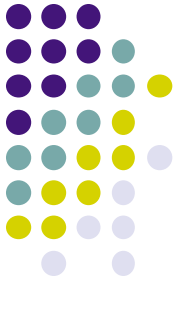


Scalaris is based on a structured overlay network

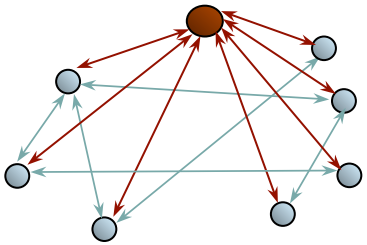


- Structured overlay networks are often based on a ring
  - By far the most popular structure, it has many variants and has been extensively studied
- Self organization is done at two levels:
  - The ring ensures connectivity: it must always exist despite node joins, leaves, and failures
  - The fingers provide efficient routing: they can be temporarily in an inconsistent state

Structured overlay networks: inspired by peer-to-peer

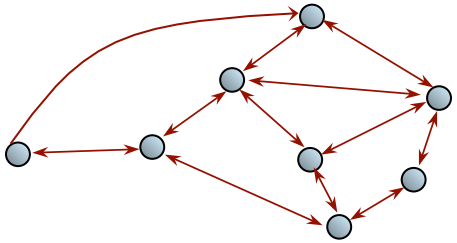


- Hybrid (client/server)
  - Napster



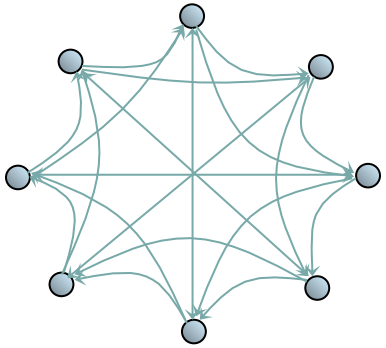
R = *N*-1 (hub)
R = 1 (others)
H = 1

- Unstructured overlay
  - Gnutella, Kazaa, Morpheus, Freenet, …
  - Uses flooding



R = ? (variable)
H = 1…7
(but no guarantee)

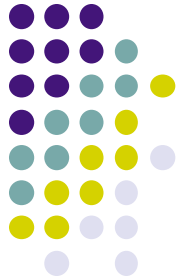
- Structured overlay
  - Exponential network
  - DHT (Distributed Hash Table), e.g., Chord, DKS, Scalaris, Beernet, etc.

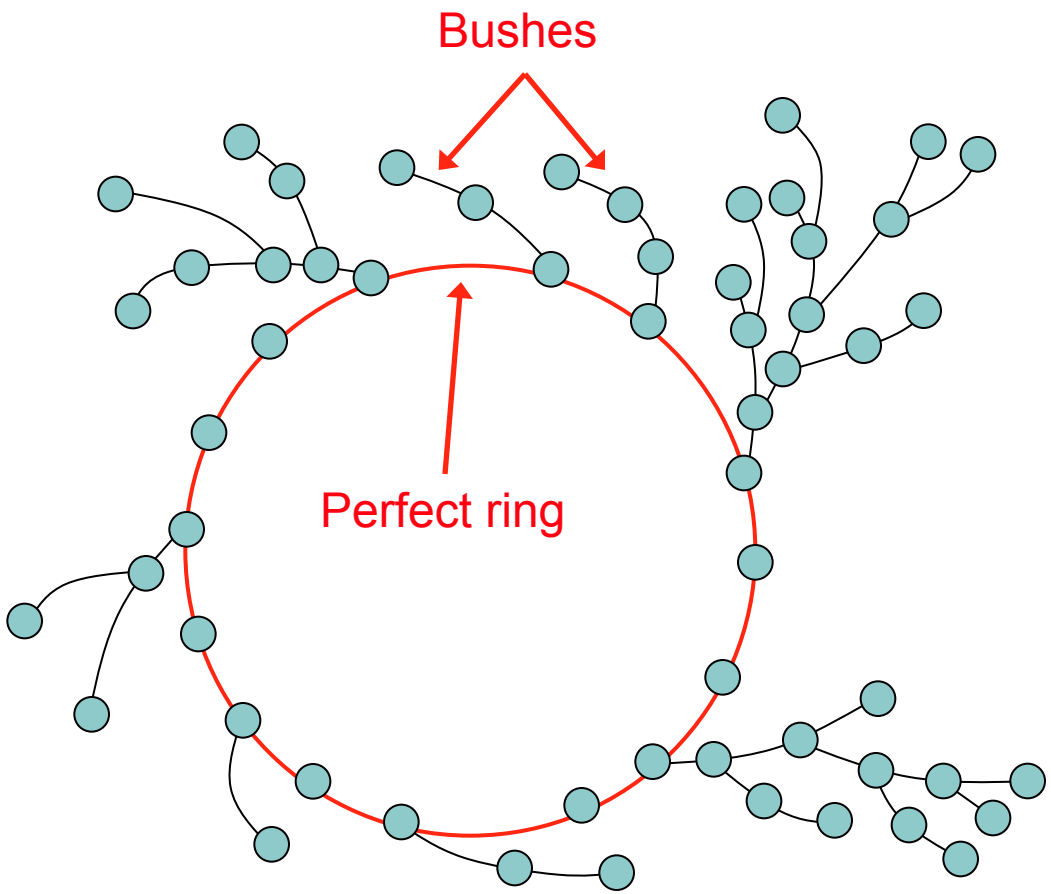


R = log *N*
H = log *N*
(with guarantee)

THE ADVENTURES OF SELFMAN

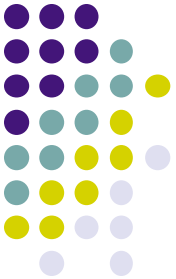
A "relaxed" structured overlay network



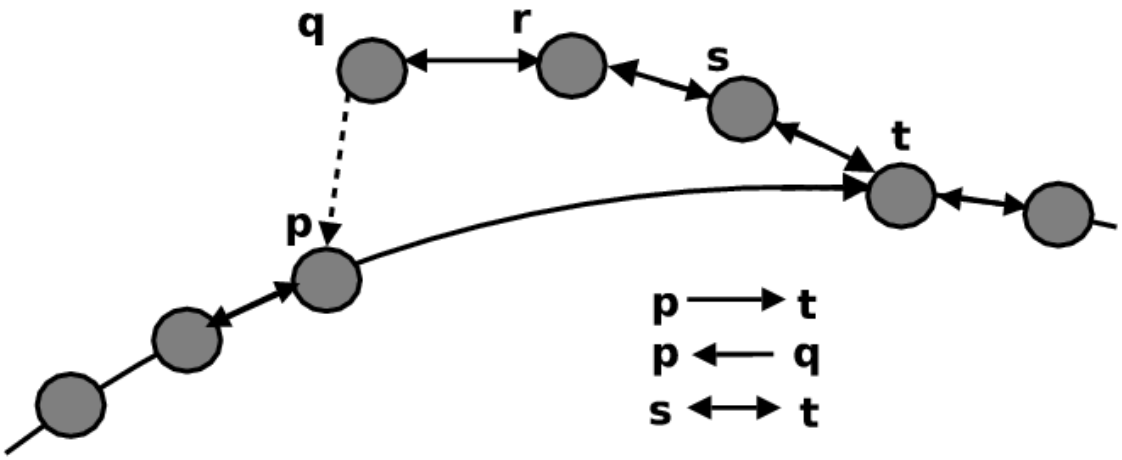


- The relaxed ring is completely asynchronous
  - Join and leave are completely asynchronous (as opposed to Scalaris, where they are synchronous)
  - The bushes appear only if there are failure suspicions
  - Beernet implements the relaxed ring
- There is a perfect ring (in red) as a subset of the relaxed ring
- The relaxed ring is always converging to a perfect ring
  - The bushiness depends on churn (rate of change of the ring, leaves/joins) and failure suspicion rate (communication delays)

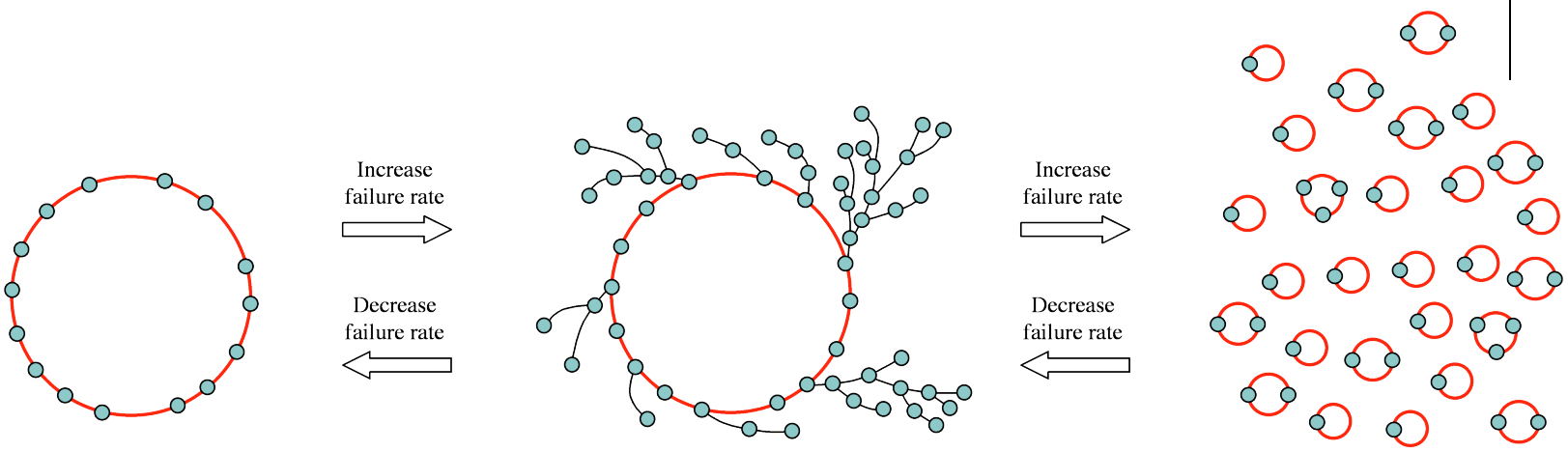
More on the relaxed ring



- False failure suspicions are common on the Internet
  - We do not want to eject the node from the ring when this happens
- The relaxed ring solves this by doing ring maintenance in asynchronous fashion [Mejias 2008]
  - Nodes communicate through message passing
  - For a join, instead of one step involving 3 peers (as in Scalaris, Chord, or DKS), we have two steps each with 2 peers → we do not need locking or a periodic stabilization algorithm
- Invariant: Every peer is in the same ring as its successor

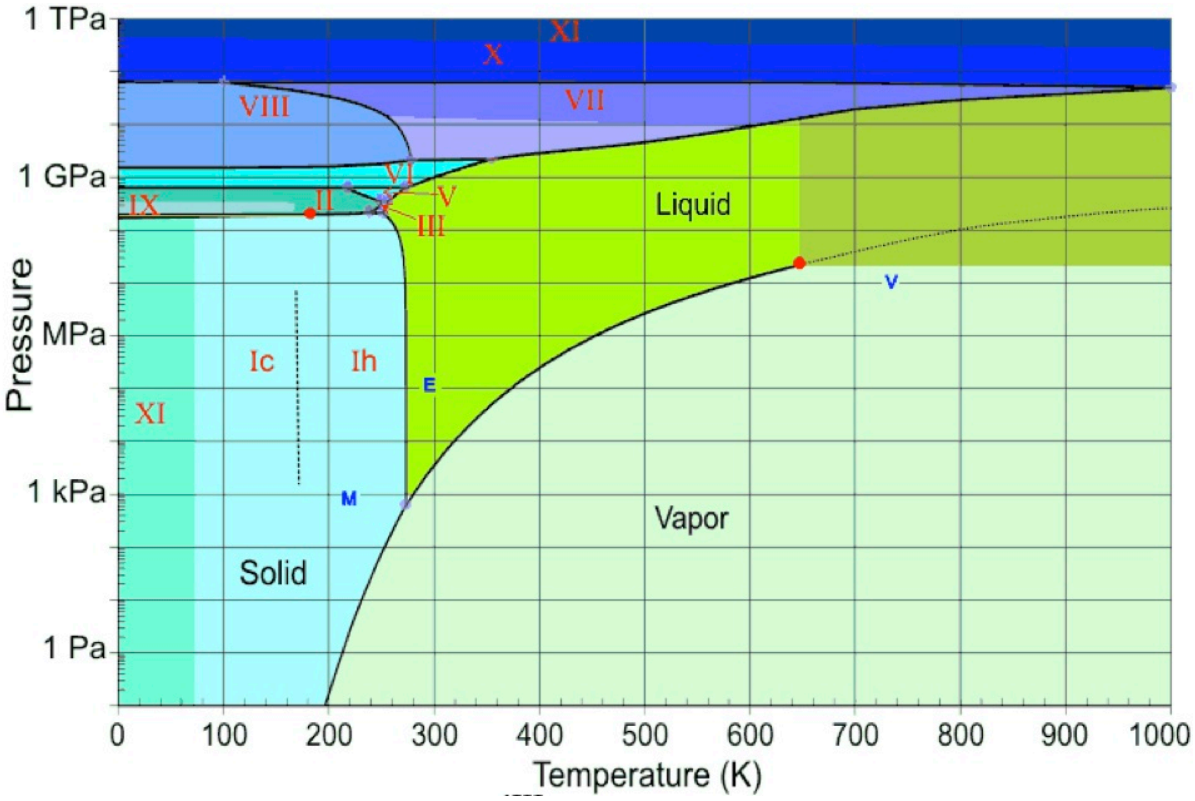


Phases in the relaxed ring



- The relaxed ring has (at least) three phases
  - Uses ring merge algorithm developed in SELFMAN [Shafaat 2009]
  - We are studying how the ring reacts to external stress (phase transitions)
- Key questions:
  - How do the phases show up at the application layer? ("qualitative changes")
  - How do we know when we are near a phase transition? ("early bubbling")

Phases in large systems

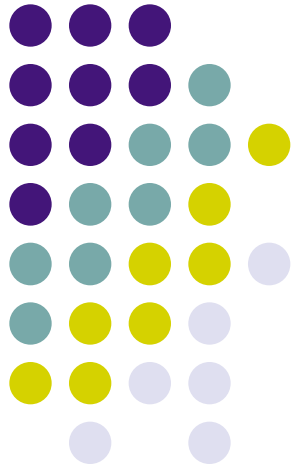


Water phase diagram
(Copyright © Martin Chaplin)

- A phase is a concise characterization of an aggregate behavior in a system consisting of many interacting components
- Phases appear in many large systems
  - Not just physical systems (water) but also computing systems (like peer-to-peer)

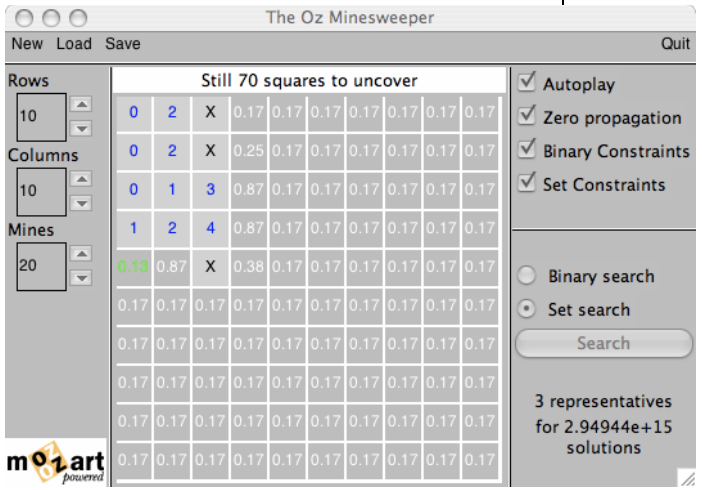
- Different parts of the system can be in different phases
  - Depending on the local operating conditions (environment)
  - Boundaries between phases can be sharp or diffuse
  - Phase transitions and critical points can occur if operating conditions change

Complex components (supplement)

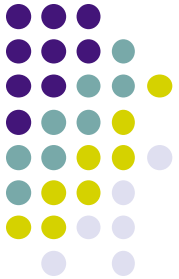


Some complex components

- Human intelligence
  - Main strength: adaptability (dynamic creation of new feedback loops)
- Program intelligence
  - Can easily go beyond human intelligence in many areas!
    - **Turing test is irrelevant**: complex components are already replacing humans in more and more areas
  - Minesweeper digital assistant: uses constraints (easy to program!)
  - Chess: uses alpha-beta search with heuristics
  - Compiler: translates human-readable program into executable form

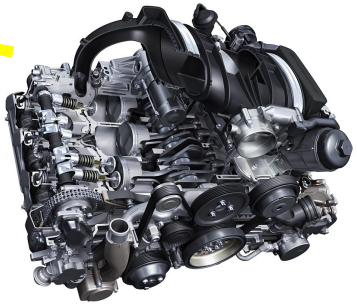


Properties of complex components



- Complex components are essential parts of many large systems
  - For example, conscious control in the human respiratory system
- Complex components *completely solve* a problem inside a specific (small) part of the space of system operating conditions (from the viewpoint of the rest of the system)
  - Conscious control, a chess program, and a compiler are extremely smart *within their operating space*
  - Outside of this space, they can be very stupid and should be inactive (on their own accord or forced)
- Complex components are *completely unpredictable* when viewed from the outside
  - If it were not so, they would not be needed!
  - They can be highly nonlinear and unstable; the rest of the system has to trust them (typically, up to some hardwired fail-safe)

Power is built in, not added on

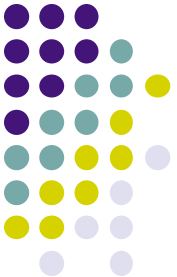


3.6-Liter Biturbo Motor with 353 kW (480 HP)

Porsche Carrera GT

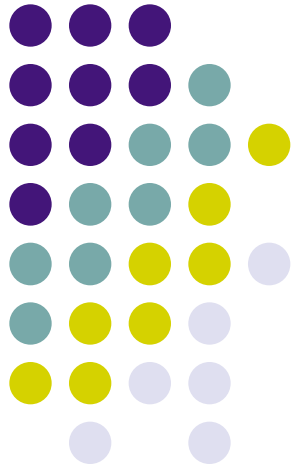
- The power of a system depends on the strength of its complex components
  - The human respiratory system uses conscious control (e.g., to avoid drowning!)
  - Erlang OTP uses supervisor trees and a database to implement robustness
  - Scalaris uses Paxos consensus and replication to implement fast transactions
  - Google Search uses eigenvector calculation of the Web link matrix
  - What does *your* system use?

Why is conscious control so smart?

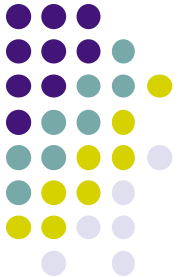


- Cognitive science and neuroscience try to understand why
  - The brain uses brute force, but in a very smart way
- Conscious control is a bricklayer: it continuously builds and organizes new components on top of existing components
  - This process is continuous from birth with compound interest effect, which is why humans are so smart in common-sense tasks
- It continuously brings the most useful concepts to the top (cache organization combined with "grandfather cell")
  - Manipulating common concepts is made easy
- "Mirror neurons": it can use its own components to simulate other humans, which is why humans can empathize so well with others
- It can efficiently execute up to two complex programs at once ("walking and chewing gum"), because of the two-lobed structure of the brain

Clouds and elastic applications (supplement)

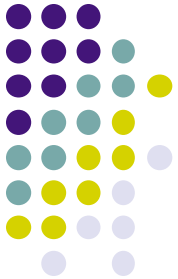


Elastic computing

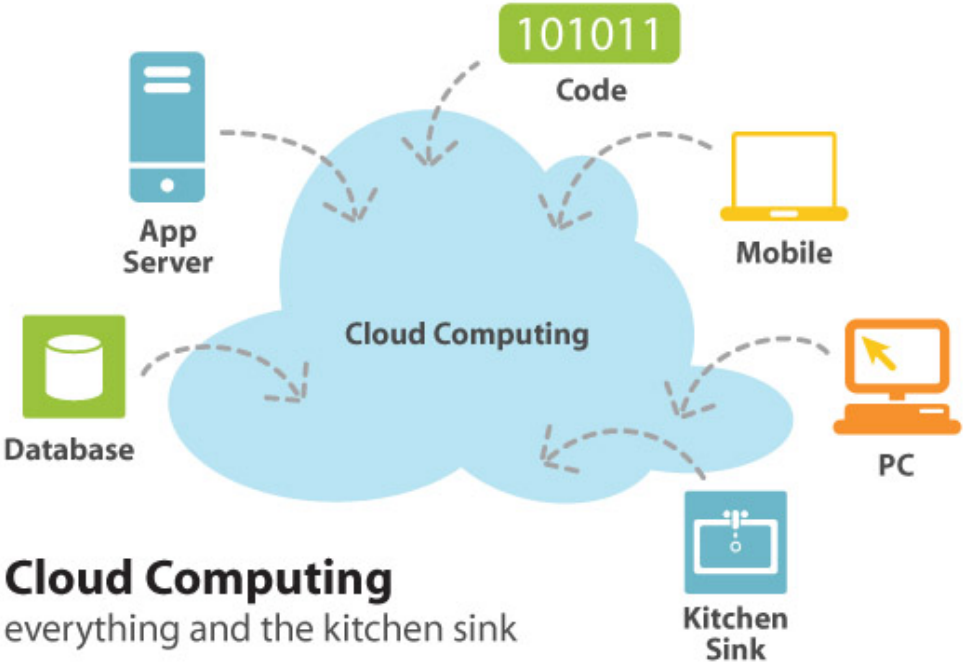


- Two main infrastructures for scalable computing
  - Peer-to-peer: use of client machines
  - Cloud-based: use of datacenters
- Cloud is elastic; peer-to-peer is not
  - Elasticity: the ability to scale resource usage up and down rapidly according to instantaneous demand
  - Elasticity is a new property that did not exist before clouds
- Elasticity makes possible a new kind of application
  - Applications that use enormous computational and storage resources for short times, but at constant (low) cost
  - Applications that use data-intensive algorithms and machine learning

Clouds are the first key: much more than meets the eye!

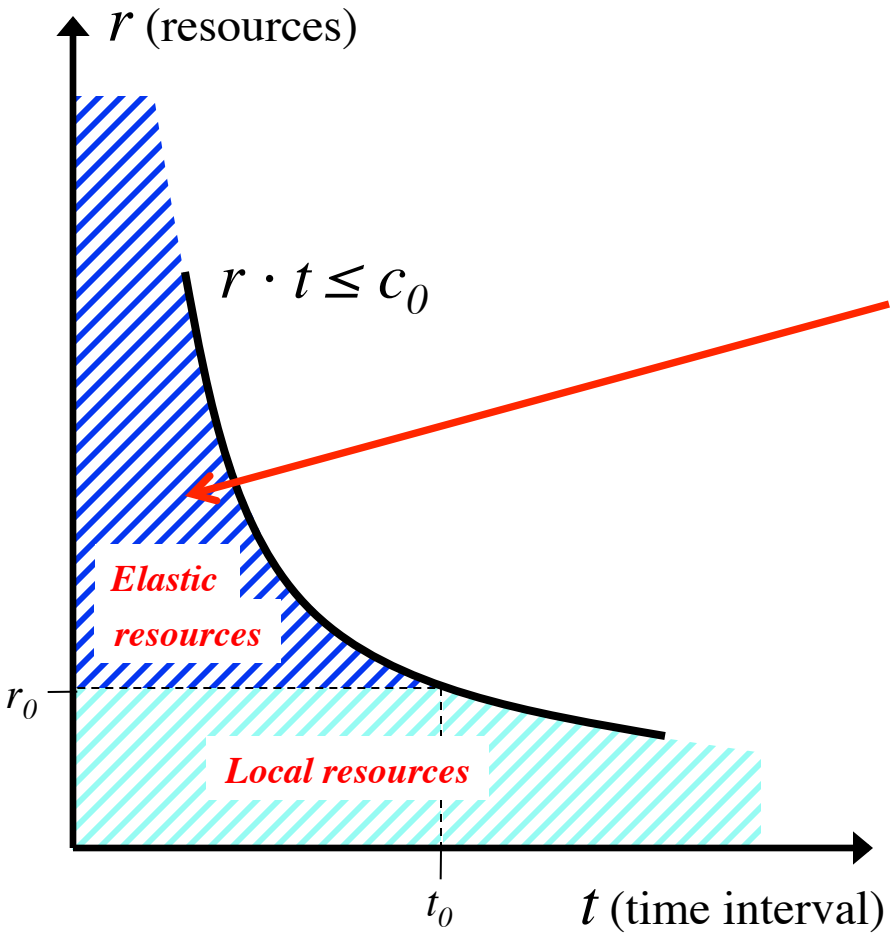


- Cloud computing is a form of client-server where the "server" is a *dynamically scalable network* of loosely coupled heterogeneous nodes that are *owned by a single institution*
- It allows enterprises to offload their computing infrastructure
- It gives mobile devices an easy way to manage data



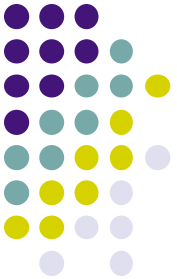
- Is that all that cloud computing offers?
  - No! This is just the tip of the iceberg!
  - Cloud computing is the beginning of a much more profound change

Clouds are elastic!



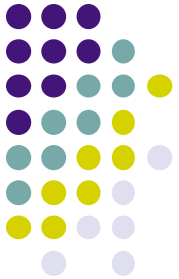
- Elasticity is the ability to ramp up resources quickly to meet demand
  - Like electric power distribution
- With elastic clouds the enormous **dark blue area** becomes available
- Applications that need enormous resources for short times can get them for low cost!
  - Like electric power distribution, pay only for the volume (cost is product of time and number of machines)
  - This is exactly what intelligent applications need!

Machine learning is the second key

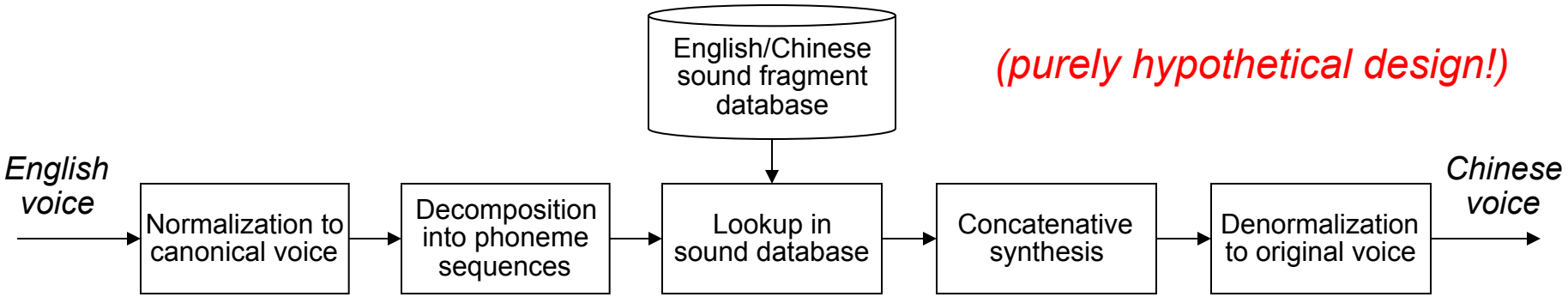


- Machine learning is the discipline that studies how to program computers to evolve behaviors based on example data or past experience
- Machine learning can solve complex problems that we cannot solve in any other way
  - It has many successes in practical applications both big and small, e.g., speech recognition, computer vision (face and handwriting, etc.), social prediction (epidemics, economics, retail, etc.), robot control (drones, cars, etc.), data mining, aiding natural sciences (biology, astronomy, neurology, etc.)
  - It is a major force on the Internet in big companies (Google, Amazon, Netflix, Facebook, etc.) as well as in startups (e.g., RecordedFuture)
- Machine learning will (eventually) transform programming!
  - Programmers will not work on raw data any more; instead they will build machine learning systems
  - "Programming, like all engineering, is a lot of work; machine learning is more like farming, where we let nature do most of the work" – Pedro Domingos

An elastic application (1): real-time voice translation

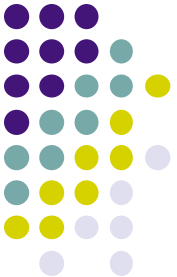


- The pieces of this application already exist; for example the IRCAM research institute has implemented many of them
- It requires combining domain knowledge (in sound and language) with an enormous sound fragment database, hosted on a cloud



- Franz Och, head of translation services at Google, announced that they are working on something similar (Feb. 10, 2010)
- Rick Rashid, head of research at Microsoft, has recently demonstrated a prototype of this application (Nov. 2012)

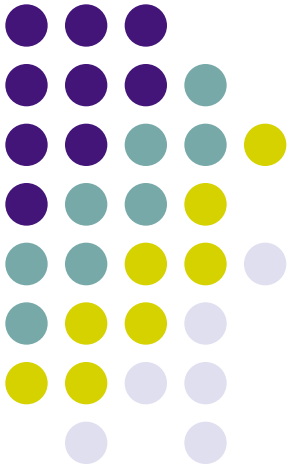
An elastic application (2): ubiquitous augmentation



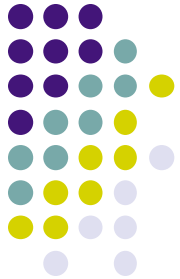
- Your sensory input will be "augmented" in real-time
  - Faces, objects, and names you see will be recognized
  - Selected relevant information will be given spontaneously
  - Foreign languages (text, audio, visual) will be translated
  - When doing an activity, you will be guided to do it expertly
  - When confronted with a problem, solutions will be suggested
- The augmentation will be good enough that it can be always enabled (it doesn't get in your way)
  - It will learn to mesh with your thinking processes productively
  - On the rare occasions that it is disabled, you will feel helpless
    - As if half of your brain just stopped working
    - Like today's Internet addictions, but much worse!

Space of intelligent applications

Conclusions

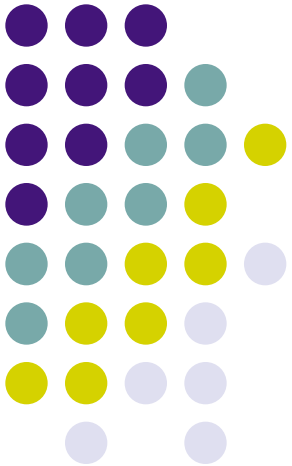


Conclusions

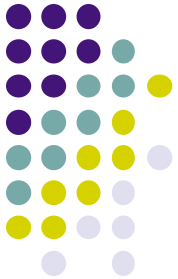


- Design of large distributed systems is difficult
  - Not just because of their own complexity, but because the environment becomes more hostile
- How can we design them?
  - Learn from existing systems that work
  - Inspiration from the SELFMAN project on self-managing systems
  - Inspiration from biological systems
- Proposed architecture
  - Weakly interacting feedback structures with dominant subsets
  - Complex components to solve the problem in limited conditions
  - Phases define behavior over all possible operating conditions
- Validation
  - First validation with the Scalaris and Beernet transactional key/value stores
- Ongoing research
  - Formalization and semantics
  - Tie the approach to existing quantitative techniques (control theory, model checking, system dynamics)
  - Collaboration with system and application builders

References

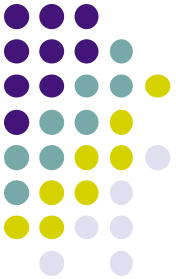


References (1)



- Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*, Ph. D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, Nov. 2003.
- Ken Birman, Gregory Chockler, and Robbert van Renesse. "Toward a Cloud Computing Research Agenda", 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, ACM SIGACT News, 40(2): 68-80 (June 2009).
- Alexandre Bultot. *A Survey of Systems with Multiple Interacting Feedback Loops and Their Application to Programming*, Master's report, Dept. of Comp. Sci. and Eng., UCL, Aug. 2009.
- Rick Cattell. "High Performance Scalable Data Stores", Feb. 22, 2010.
- Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language*, Ph. D. dissertation, Dept. of Comp. Sci. and Eng., UCL, Dec. 2007.
- Michael Fischer, Nancy Lynch, and Michael Paterson. "Impossibility of Distributed Consensus with One Faulty Process", Journal of the ACM, 32(2): 374-382 (April 1985).
- Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", ACM SIGACT News, 33(2): 51-59 (2002).
- Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*, Springer-Verlag, 2006.
- Márk Jelasity and Özalp Babaoglu. "T-Man: Gossip-based Overlay Topology Management", Proc. 3rd Int. Workshop on Engineering Self-Organising Systems (ESOA 2005), Springer-Verlag LNCS volume 3910, 2006, pp. 1-15.
- Boris Mejías. *A Relaxed Ring for Self-Managing Decentralized Systems with Transactional Replicated Storage*, Ph. D. dissertation, Dept. of Comp. Sci. and Eng., UCL, Oct. 2010.
- Gerhard Michal and Dietmar Schomburg. *Biochemical Pathways: An Atlas of Biochemistry and Molecular Biology*, Wiley-Blackwell, 1999 (first edition), 2012 (second edition).

References (2)



- Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. "Enhanced Paxos Commit for Transactions on DHTs", 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), May 17-20, 2010, Melbourne, Australia.
- SELFMAN: Self Management for Large-Scale Distributed Systems Based on Structured Overlay Networks and Components. European 6th Framework Programme, www.ist-selfman.org (2009).
- Peter M. Senge *et al*. *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization*, Nicholas Brealey Publishing, 1994.
- Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. "Dealing with Network Partitions in Structured Overlay Networks", Journal of Peer-to-Peer Networking and Applications, 2(4): 334-347 (2009).
- Steven Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering (Studies in Nonlinearity)*, Perseus Books, 1994.
- Nassim Taleb. *The Black Swan: The Impact of the Highly Improbable*, Penguin Books, 2008.
- Peter Van Roy, Seif Haridi, and Alexander Reinefeld. "Software Design with Weakly Interacting Feedback Structures and Its Application to Distributed Systems", Research Report, Dept. of Comp. Sci. and Eng., UCL, 2011.
- Peter Van Roy. "Programming Paradigms for Dummies: What Every Programmer Should Know", chapter in New Computational Paradigms for Computer Music, G. Assayag and A. Gerzso (eds.), IRCAM/Delatour France, June 2009.
- Gerald M. Weinberg. *An Introduction to General Systems Thinking*, Dorset House Publishing, 1975 (Silver Anniversary Edition 2001).
- Norbert Wiener. *Cybernetics, or Control and Communication in the Animal and the Machine*, MIT Press, Cambridge, MA, 1948.
- Ulf Wiger. "Four-fold Increase in Productivity and Quality – Industrial Strength Functional Programming in Telecom-Class Products", Ericsson Telecom AB, 2001.