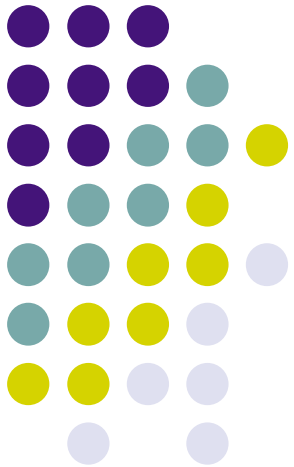
THE ADVENTURES OF SELFMAN

Reflections on Self Management in Software Development

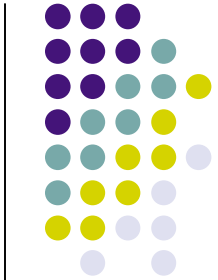


Peter Van Roy

Sept. 28, 2007
ICMS, Edinburgh
Scottish programming language seminar

Université catholique de Louvain
Louvain-la-Neuve, Belgium

THE ADVENTURES OF SELFMAN

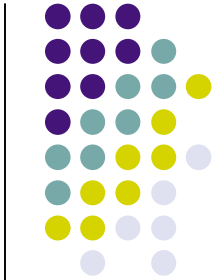


Overview

- Motivation
  - Why software design needs self management
- Feedback loops
  - Why feedback loops are everywhere
  - Biological regulatory systems
- Program design with feedback loops
- Architecture of self-managing systems
- Decentralized distributed systems
- Conclusions
  - Software systems should be self managing
  - This is ongoing work in the SELFMAN project

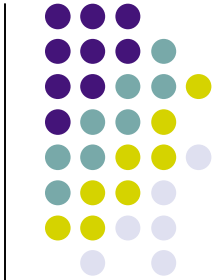
THE ADVENTURES OF SELFMAN

Software and the Red Queen



- Software is fragile!
  - A single bit error can cause a catastrophe
- Hardware and operating systems have been reliable enough so that this has not unduly hampered the quantity of software being written
  - Hardware is verified to a high degree, it is much more reliable than software
  - Good operating systems provide strong encapsulation at their core (virtual memory, multitasking) and this has been polished for many years
  - New techniques in fault tolerance (e.g., distributed algorithms, Erlang) and in programming (e.g., structured programming, OOP, the usual bunch of modern methodologies – agile, extreme, etc.) have arguably kept pace so far
- We are in a Red Queen situation: running as hard as we can to stay in the same place
- So what is the next challenge and the next technique that will keep pace with it?

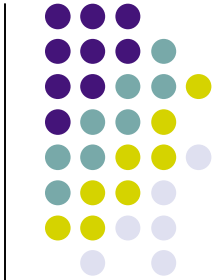
THE ADVENTURES OF SELFMAN



The next challenge (1)

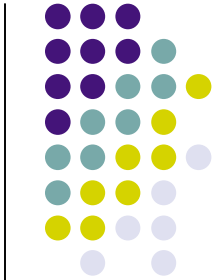
- Software complexity is ramping up quickly due to:
  - The sufficient bandwidth and reliability of the Internet to support distributed applications
  - The increasing number of devices connected to the Internet
  - The increasing computing power of these devices
- Many new applications are appearing: file-sharing (Napster, Gnutella, Morpheus, Freenet, etc.), collaborative tools (Skype, various Messengers), MMORPGs (World of Warcraft, Dungeons & Dragons, etc.), research testbeds (SETI@home, PlanetLab, etc.)
  - These applications are like services: they should be long-lived
  - Their architectures are a mix of client/server and peer-to-peer
  - These applications are still rather conservative: they do not take full advantage of the new complexity space

THE ADVENTURES OF SELFMAN



The next challenge (2)

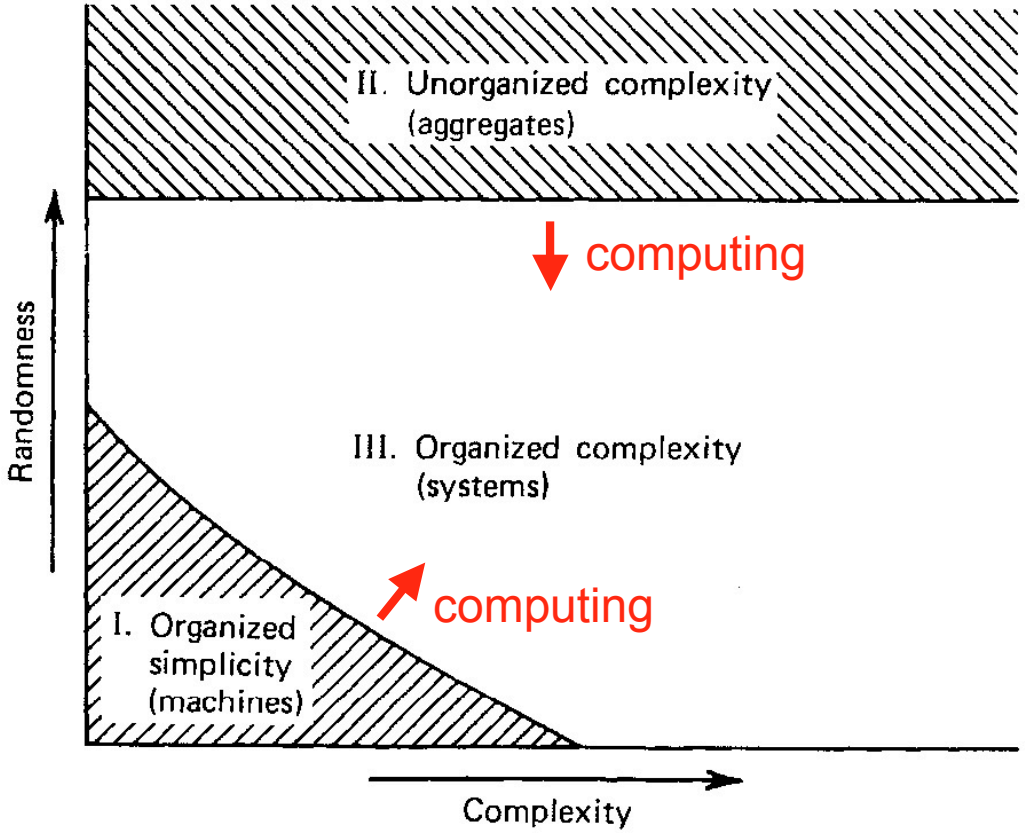
- The main problem that comes from the increase in complexity is that software errors cannot be eliminated [Armstrong 2003]
  - We have to cope with them
- Programming large-scale distributed systems introduces other problems too:
  - Scale: large numbers of independent nodes
  - Partial failure: part of the system fails
  - Security: multiple security domains
  - Resource management: resources are localized
  - Performance: harnessing multiple nodes or spreading load
  - Global behavior: emergent behavior of the system as a whole
- Global behavior is particularly relevant
  - Example: the power grid [Fairley 2005]



The next solution

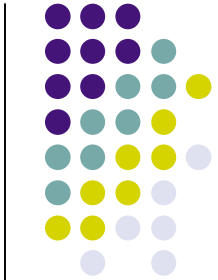
- Now that we have set the stage, what solution do we propose?
- For inspiration, we go back fifty years, to the first work on cybernetics and general system theory
  - Designing systems that regulate themselves (self-managing systems) [Wiener 1948, Ashby 1956, von Bertalanffy 1969]
- A system is a set of components (called subsystems) that are connected together to form a coherent whole
  - Can we predict the system's behavior from its subsystems?
  - Can we design a system with desired behavior?
- No general theory has emerged (yet) from this work
  - We do not intend to develop such a theory
  - Our aim is narrower: to build self-managing *software* systems
    - Such systems have a chance of coping with the new complexity

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
- Programming research is (one of) the vanguards of system theory
  - The other is computational science (model design and simulation)

THE ADVENTURES OF SELFMAN

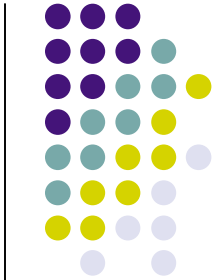


Some recent work

- IBM's Autonomic Computing initiative (2001)
  - Reduce management costs by removing humans from system management loops
  - The role of humans is then to manage policy and not to manage the mechanisms that implement it
- Structured overlay networks ([Stoica *et al* 2001], …)
  - Inspired by popular peer-to-peer applications
  - Provide low-level self management of routing, storage, and smart lookup in large-scale distributed systems
- Research in ambient and adaptive computing
  - E.g., adaptive user interfaces, ubiquitous computing
- Self management has a bigger role than just these areas

THE ADVENTURES OF SELFMAN

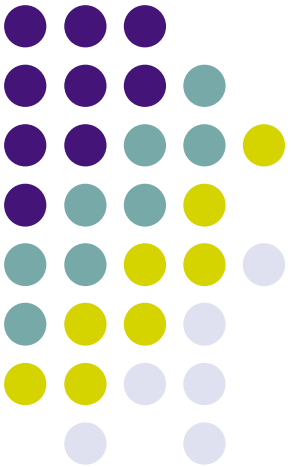
Designing self-managing software systems



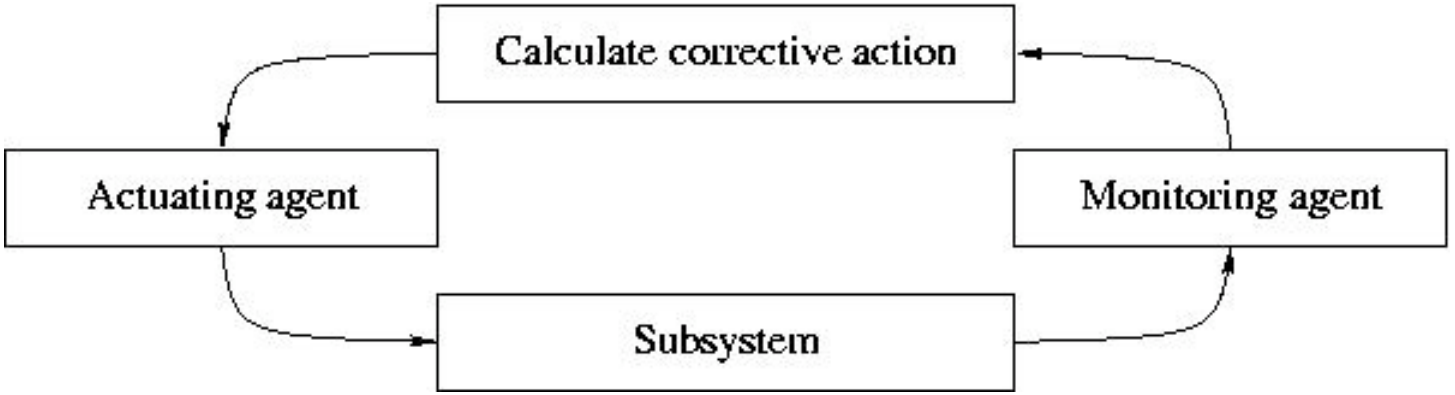
- From system theory, we take some fundamental principles
  - Programming with feedback loops
  - Focus on global (emergent) properties
  - Architectural framework
- We will use these principles as a basis for practical software development
  - This talk will give some ideas and many examples; our work in this area is just starting
    - All comments welcome!
  - We will emphasize how to program with feedback loops
    - Slogan: no open-ended software

THE ADVENTURES OF SELFMAN

Feedback loops



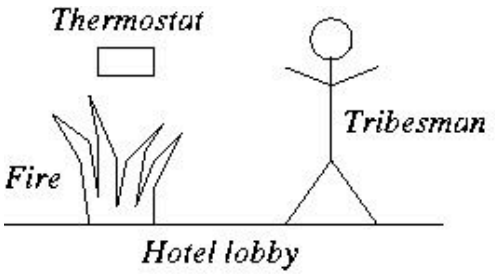
Feedback loops

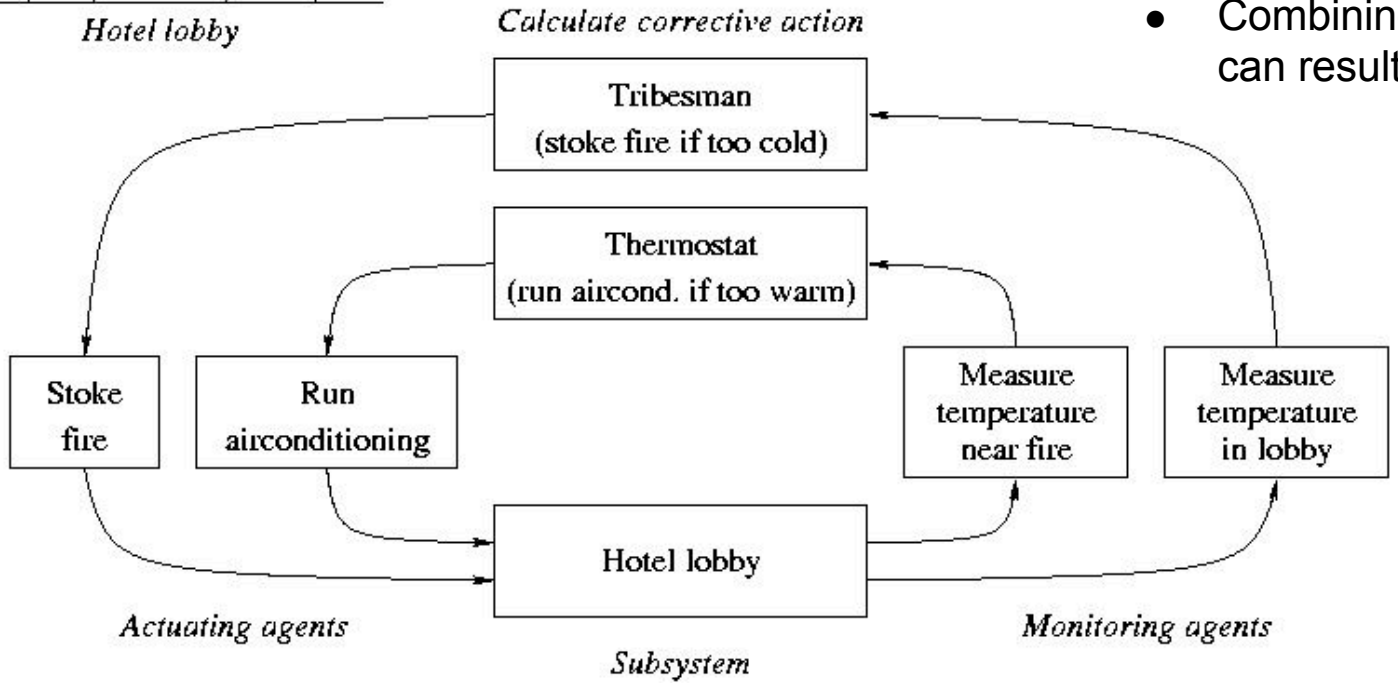


- A feedback loop consists of three elements that interact with a subsystem: a monitoring agent, a correcting agent, and an actuating agent
  - The elements and the subsystem are concurrent components interacting through asynchronous message passing
- Feedback loops can interact in two ways:
  - two loops that affect interdependent system parameters (stigmergy)
  - one loop that directly controls another loop (management)

THE ADVENTURES OF SELFMAN

Stigmergy example (Wiener)

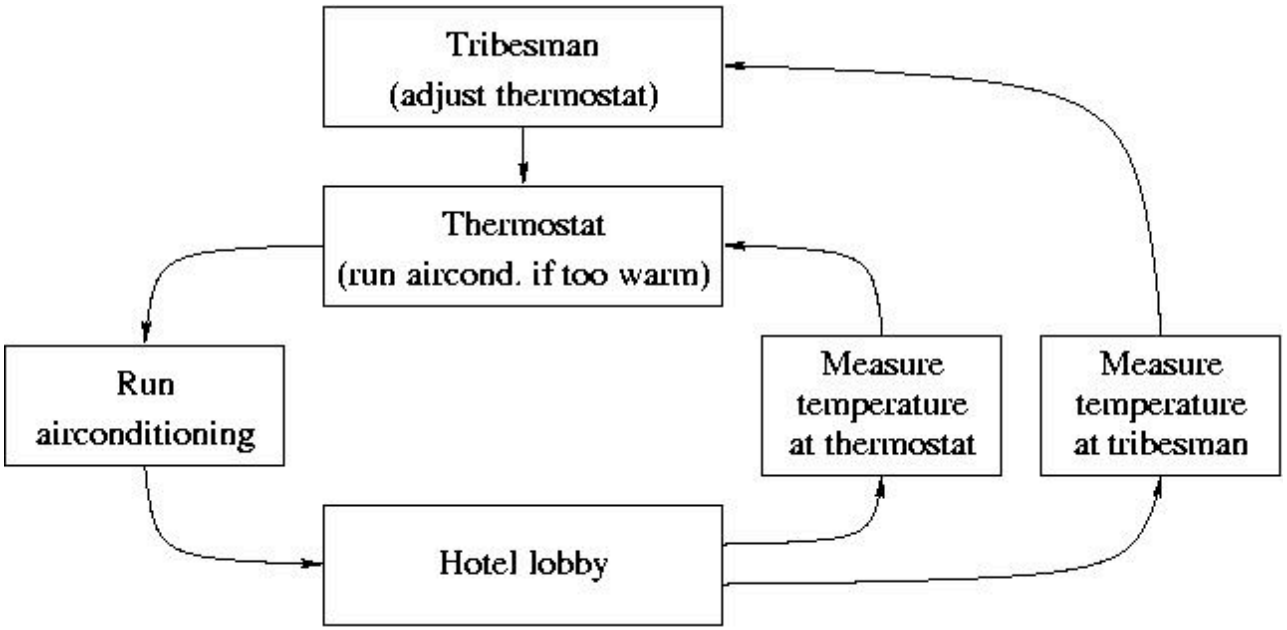




- This system is unstable!
- But each loop is stable in isolation
  - Combining stable loops can result in instability

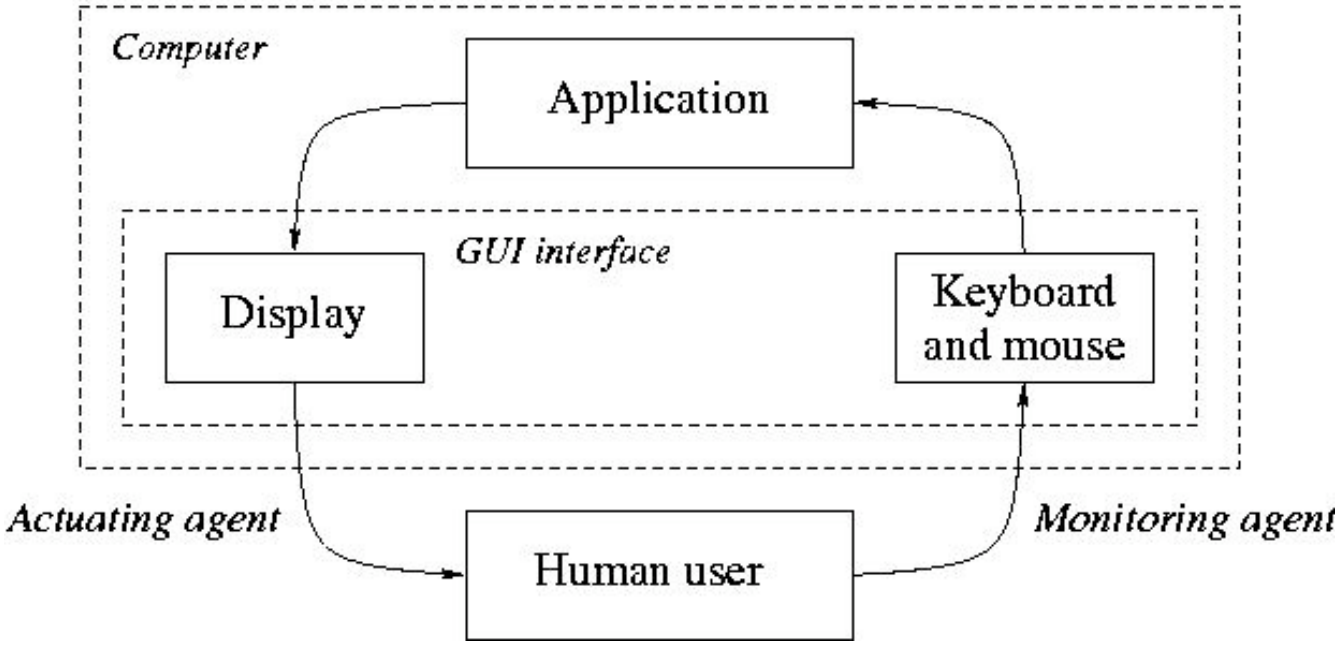
THE ADVENTURES OF SELFMAN

Correct solution uses management



- Instead of stoking a fire, the tribesman simply adjusts the thermostat. The resulting system is stable.
- This uses management instead of stigmergy
- Design rule: use the system, don't try to bypass it

Feedback loops are everywhere



- Feedback loops are literally everywhere, if you look at a system with the right mindset
- A single-user application is a simple example

THE ADVENTURES OF SELFMAN

Feedback loops are really everywhere!

- Real life is literally filled with variations on the feedback principle.
  - Bending a plastic ruler: single stable state. The ruler resists with a force that increases with degree of bending, until equilibrium (or until ruler breaks: change of phase)
    - The ruler is a simple self-adaptive system.
    - The feedback loop: force imposed on ruler, ruler reacts with counteracting force, this may affect the force, etc.
  - Clothes pin: one stable and one unstable state. Can be kept temporarily in the unstable state by pinching. Release the force and it will go back to (a possibly more complex) stable state.
  - Safety pin: two stable states, open and closed. Within each stable state, the system is adaptive like the ruler. Example of feedback loop with management: the outer control chooses the stable state.
- Anything with duration is managed by a feedback loop
  - Lack of feedback means there is a runaway reaction (explosion)

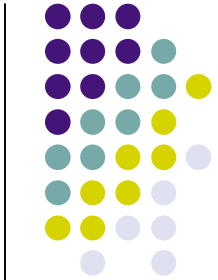
THE ADVENTURES OF SELFMAN

Civilization relies on feedback loops

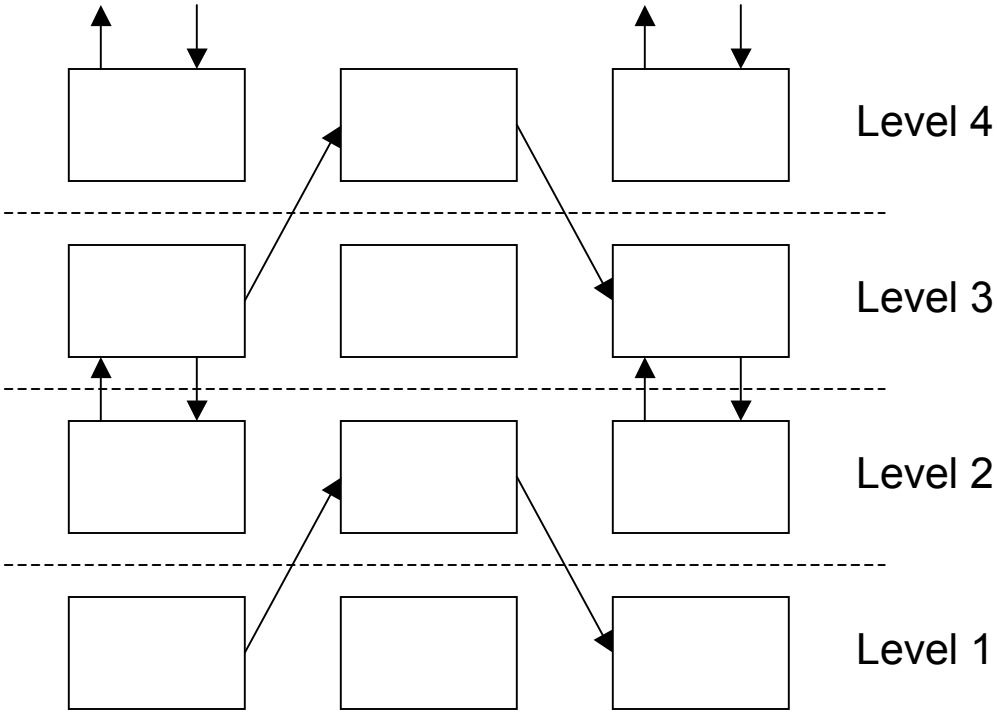
- Most products of human civilization use an implicit management feedback loop, called “maintenance”, done by a human
  - Changing lightbulbs, replacing broken windows, filling up a car
- Each human mind is at the center of an enormous number of these feedback loops
  - Most require very little conscious thinking, since they have become “habits”: programmed into the brain below consciousness
  - Each human being creates huge numbers of such habit programs
- If there are too many feedback loops to manage the human complains that “life is too complicated”!
  - “Civilization advances by reducing the number of feedback loops that have to be explicitly managed” (Van Roy’s corollary to A. N. Whitehead’s dictum)
  - A dishwashing machine reduces work of washing dishes, but it needs to be bought, maintained, replaced, etc. Is it worth it? Is the total effort reduced?

THE ADVENTURES OF SELFMAN

Complexity of interacting feedback loops

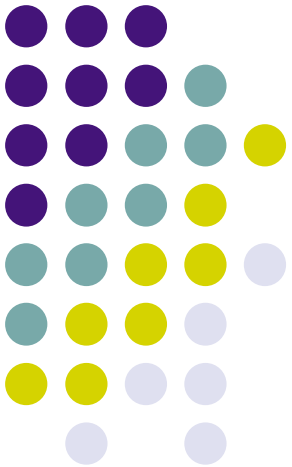


- Problems of global behavior
  - Does it converge or diverge?
  - Does it oscillate or behave chaotically?
- Analysis not always easy
  - Linear and monotonic loops are easy; unfortunately software is usually nonlinear
- What are the **rules of good feedback design**?
  - We need to understand how to program with feedback loops
  - Analogous to structured and object-oriented programming
- **Let us start by looking at some real systems**



THE ADVENTURES OF SELFMAN

Biological regulatory systems

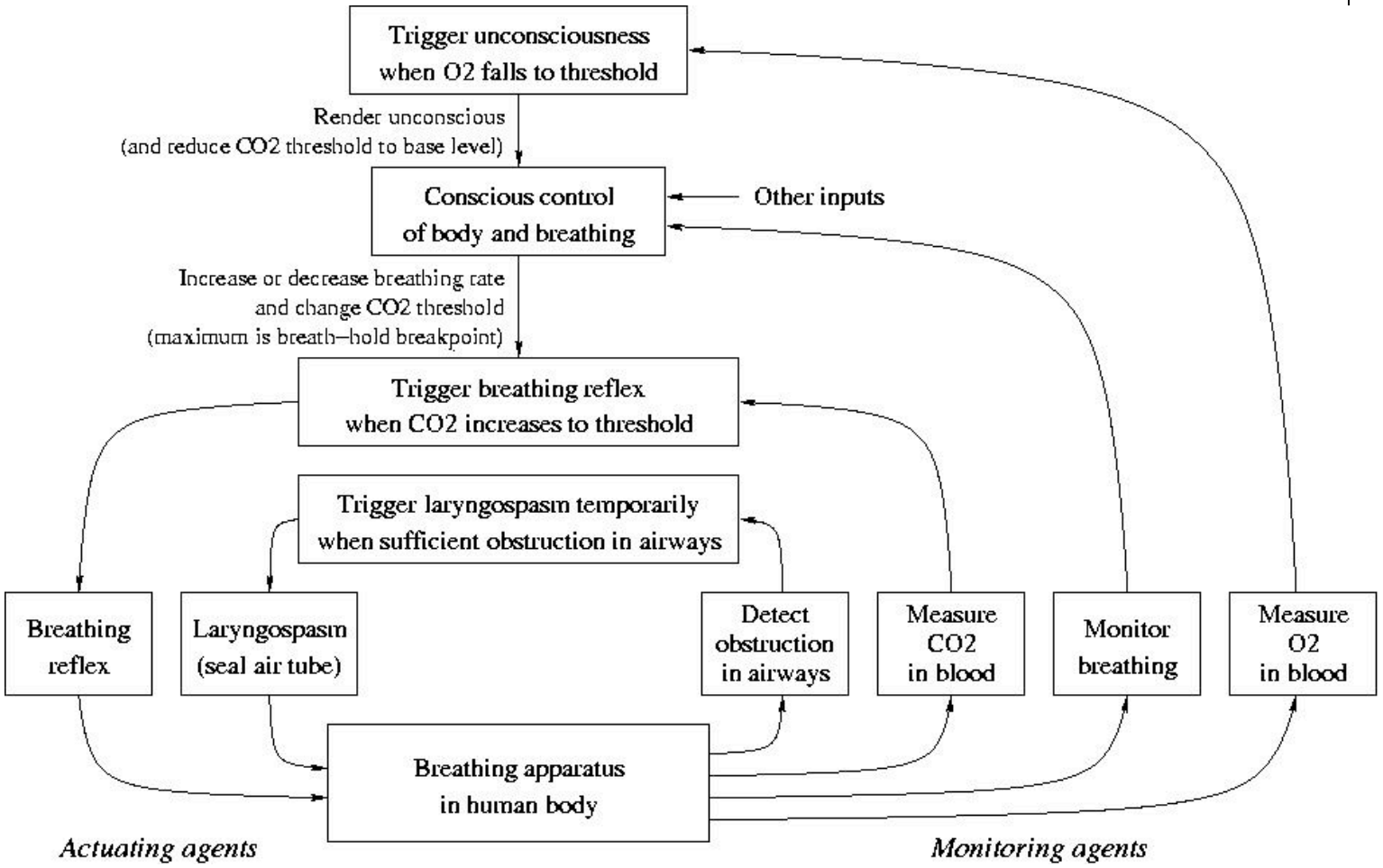


Biological regulatory systems

- We start our investigation by looking at some existing systems built with feedback loops
- Biological systems a good example, and they have the great advantage that they work!
  - Design rules inferred by studying them may be good ones
- Examples:
  - Human respiratory system
  - Human endocrine system

THE ADVENTURES OF SELFMAN

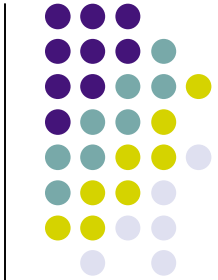
Human respiratory system



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: “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

Human endocrine system



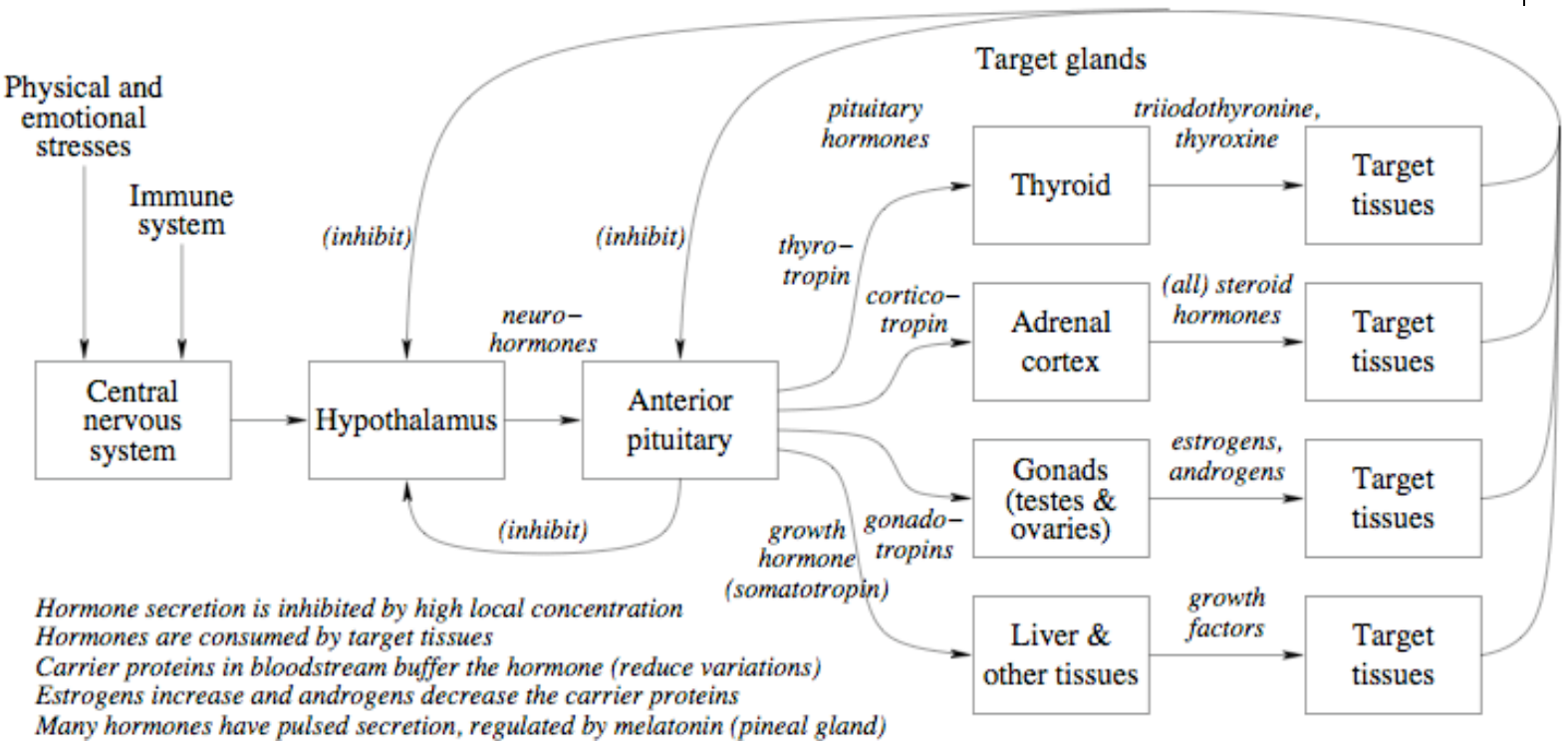
- The endocrine system regulates many quantities in the human body
- It used chemical messengers called hormones which are secreted by specialized glands and which exercise their action at a distance, using the blood stream as a diffusion channel
- By studying the endocrine system, we can obtain insights 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
  - Pattern: two hormones that work in opposite directions
- More complex regulatory mechanisms exist, e.g., hypothalamus-pituitary-target organ axis
- There is interaction between nervous transmission and hormonal transmission

THE ADVENTURES OF SELFMAN

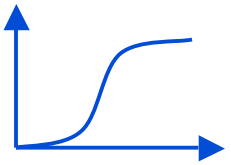
Hypothalamus-pituitary-target organ axis



- Two superimposed groups of negative feedback loops, a third short negative loop, a fourth loop from the central nervous system (from [Encyc. Brit. 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
    - Time scales: the loops are slow; the central nervous system is fast



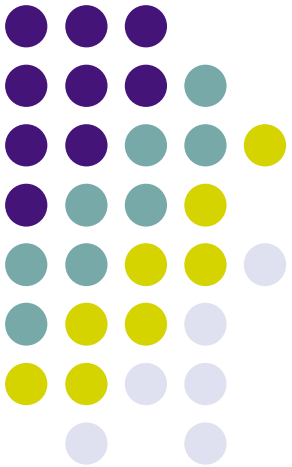
THE ADVENTURES OF SELFMAN

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

THE ADVENTURES OF SELFMAN

Program design with feedback



Program design with feedback

- The style of system design illustrated by biological systems can be applied to programming
- Programming then consists of building hierarchies of interacting feedback loops
- Examples
  - Simple loop: Self-adaptive user interface
  - Distribution: TCP (reliable point-to-point byte stream)
  - Fault tolerance: Erlang fault-tolerance architecture
  - Environment: Subsumption architecture

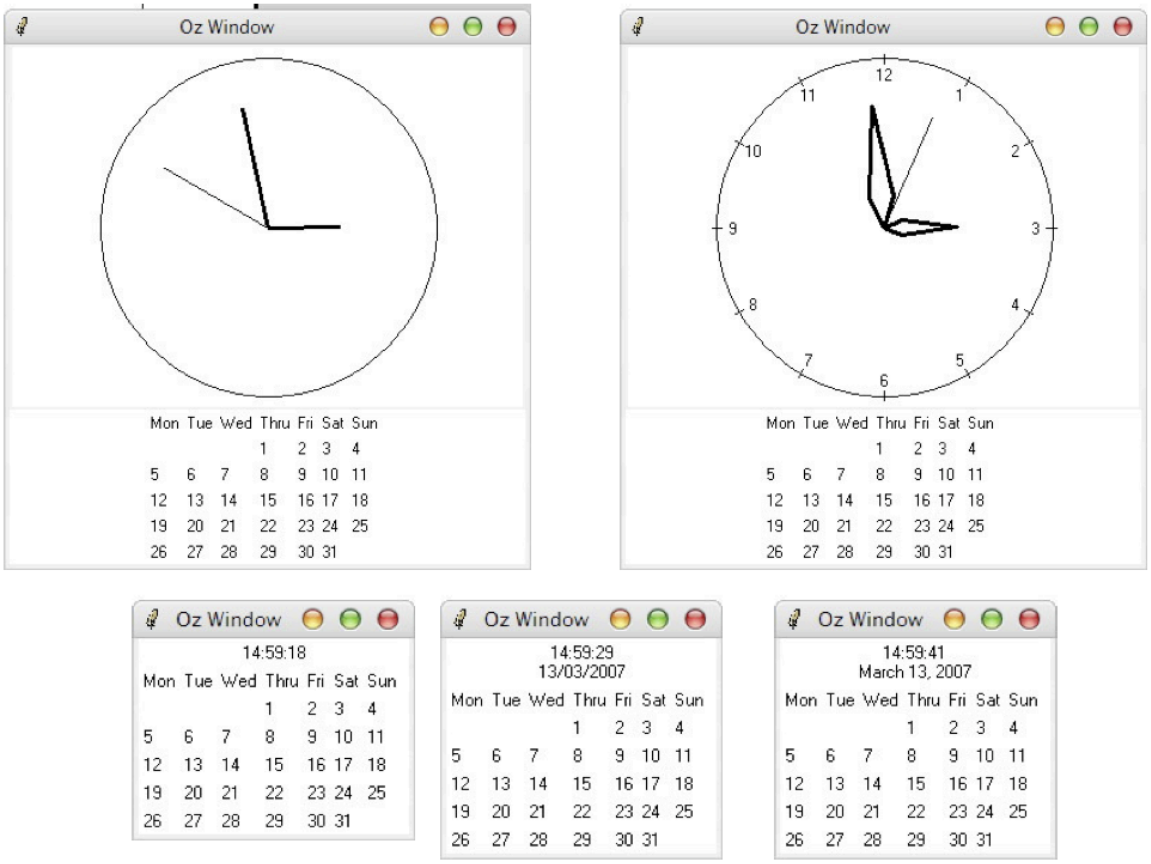
THE ADVENTURES OF SELFMAN

Example: self adaptation of user interfaces

- A simple example that uses one feedback loop
  - With a user interface toolkit that implements monitoring and actuating abilities
- EBL/Tk toolkit is designed to do exactly this
  - Monitoring: changes in environment including migration, using event-based architecture (window resize, migration, mouse, any environmental change)
  - Actuating: recalculation of display using hybrid declarative-imperative approach, transparent migration, platform independence
- Implemented for Mozart Programming System
  - Donatien Grolaux, "Transparent migration and adaptation in a Graphical User Interface toolkit", Ph.D. thesis, Sept. 2007

THE ADVENTURES OF SELFMAN

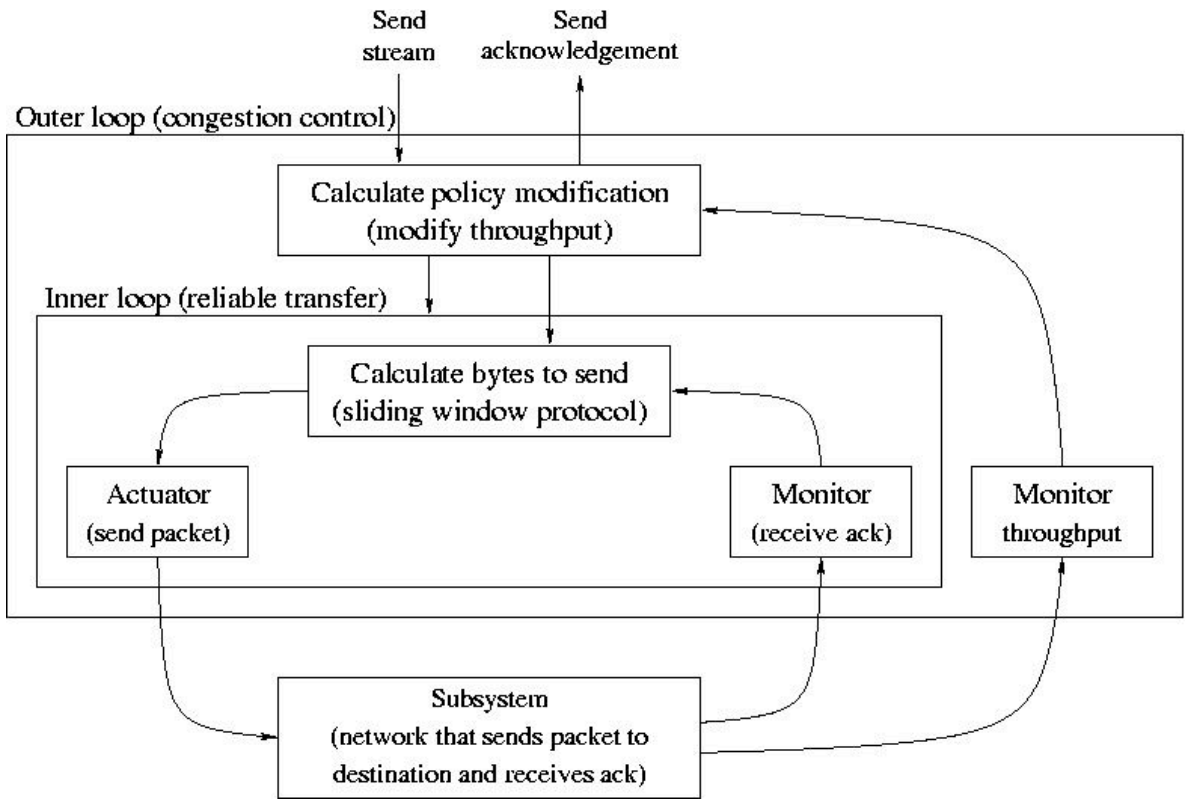
Self-adaptable clock utility



- Clock dynamically adapts display according to window size and device capability (including resize and migration to other devices)
- One feedback loop: resize/migration event → choose display → new display

THE ADVENTURES OF SELFMAN

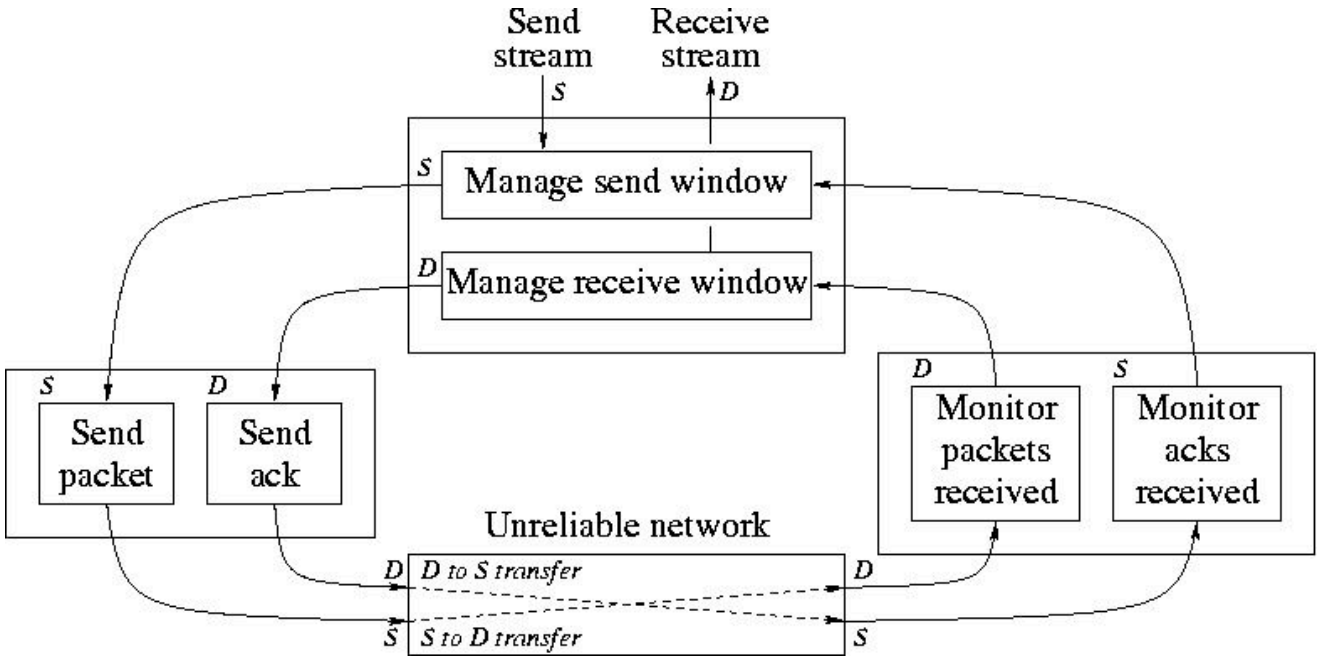
Example: TCP



- 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

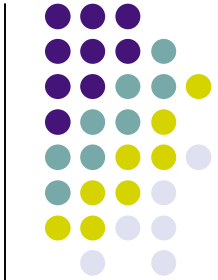
THE ADVENTURES OF SELFMAN

Interaction between feedback loops and distribution



- The previous slide only showed what happens at the source node
- We expand the inner loop to show execution on both nodes. This shows two feedback loops (S loop and D loop), one running at the source and one running at the destination. The loops interact through stigmergy.

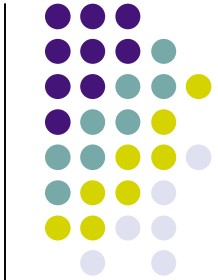
THE ADVENTURES OF SELFMAN



Feedback loops and distribution

- The interaction between feedback loops and distribution is not well understood
- Distributed algorithmics has studied special cases of this interaction
  - Fault tolerance
  - Self-stabilizing systems
  - Structured overlay networks
- Feedback loops are useful for much more than fault tolerance!
  - We will take a closer look at structured overlay networks as an example of a decentralized distributed system

THE ADVENTURES OF SELFMAN

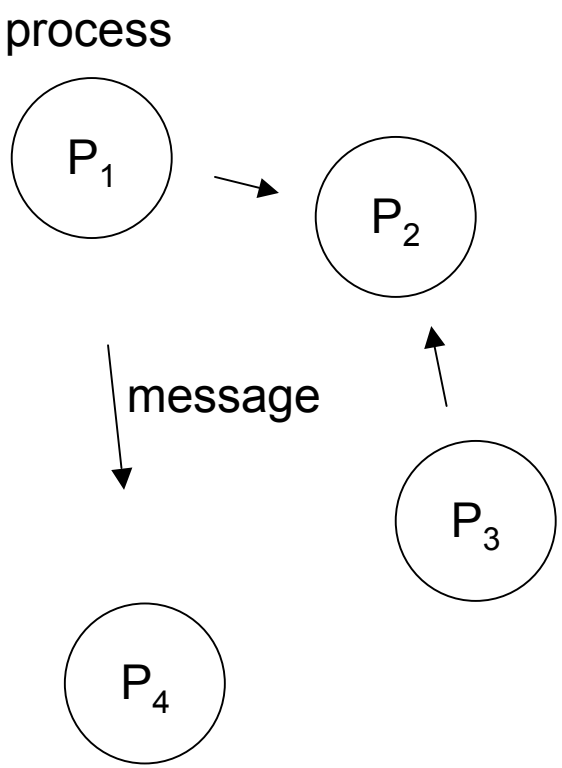


Other examples of self-managing software systems

- Erlang fault-tolerance architecture [Armstrong 2003]
  - Erlang is designed explicitly to build applications that survive software faults
    - Hypothesis: Software faults are inevitable
  - The Erlang system has been used to build highly available products: AXD301 ATM switch, Bluetail Mail Robustifier, SSL accelerator
- Subsumption architecture [Brooks 1986]
  - To build systems that show intelligent behavior by decomposing complex behaviors into layers of simple behaviors
  - Knowledge is represented indirectly through the environment
  - Used successfully to program physical robots

THE ADVENTURES OF SELFMAN

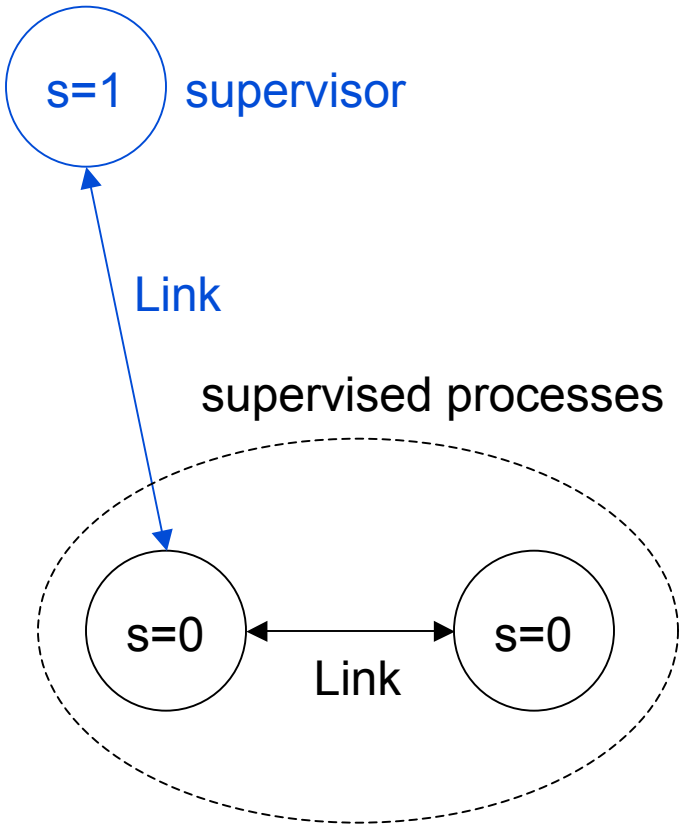
Erlang



- Erlang is a language used to develop highly reliable software systems
- An Erlang program consists of a set of running "processes" (lightweight threads with independent address spaces) that send messages asynchronously
- Fault tolerance consists of three levels:
  - Primitive failure detection through process linking: when one process fails, another is notified
  - Supervisor trees to structure the program
  - Stable storage to restart after crashes (single or multiple disk)

THE ADVENTURES OF SELFMAN

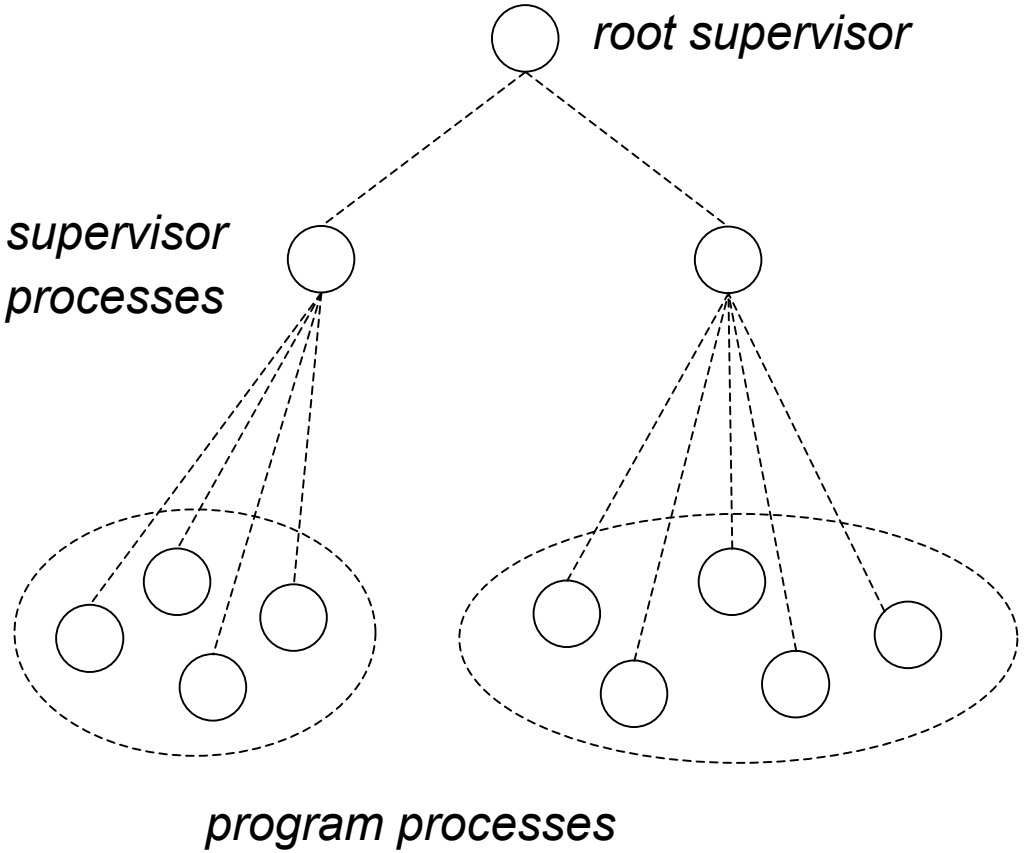
Primitive failure detection



- Two processes can be linked: if one fails then both are terminated
  - Failure is a permanent crash failure, detected by the run-time system
  - "Let it fail" philosophy: if anything goes wrong, just crash and let another process correct the problem
- If a linked process has its supervisor bit set, then it is sent a message instead of failing
- This primitive failure detection can be seen as monitoring in a feedback loop

THE ADVENTURES OF SELFMAN

Supervisor trees



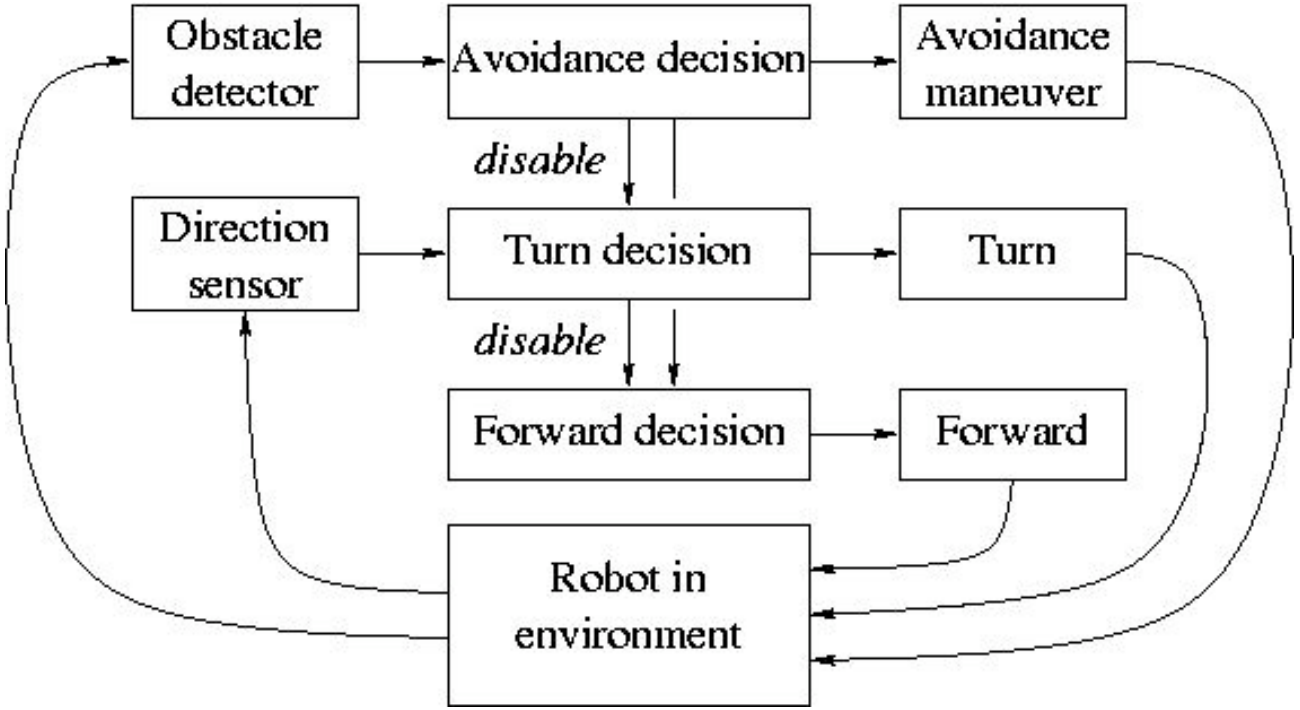
- The program consists of a large number of processes
- Program processes are organized in pools
  - Each pool is observed by a supervisor process linked to all of them
  - An AND supervisor stops and restarts all its children if one crashes
  - An OR supervisor restarts just the crashed child
- The supervisors themselves are observed by a root supervisor
- Each internal node in the supervisor tree corresponds to a feedback loop

THE ADVENTURES OF SELFMAN

Subsumption architecture

- The subsumption architecture is a way to implement complex, “intelligent” behaviors by decomposing them into simpler behaviors
- The system consists of layers where each layer provides a simple ability
- Layers are given priorities: when a layer can act, it disables the lower layers
- Layers interact through stigmergy

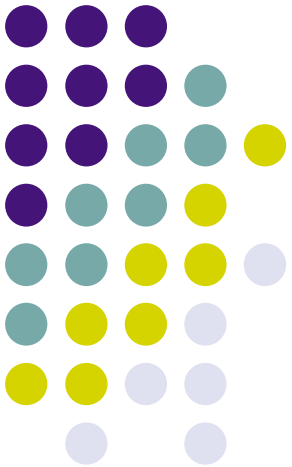
An obstacle-avoiding robot



- Each layer provides a competence
- Each layer can override the lower layers
- If a higher layer fails, some competence remains

THE ADVENTURES OF SELFMAN

Architecture of self-managing systems

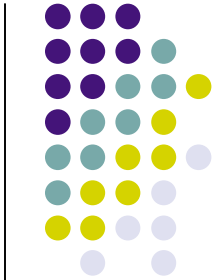


THE ADVENTURES OF SELFMAN

Self-management architecture

- Axes of self management
  - Cooperation/competition
  - Simple/complex component
  - Time scales
  - Noise
- Three-layered architecture
  - Event layer
  - Feedback loop layer
  - Market (collective intelligence) layer
- Collective intelligence
  - El Farol bar example

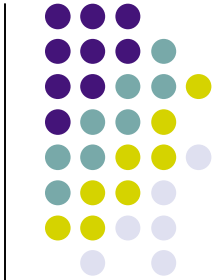
THE ADVENTURES OF SELFMAN



Axes of self management

- Cooperation/competition
  - One designer/user versus many designers/users
- Simple/complex components
  - Complex: contains human intelligence or some other form of nontrivial reasoning (e.g., digital assistant for Minesweeper with constraint-based reasoning or chess program)
  - Many systems have both simple and complex components, e.g., social systems with both humans and computers, human regulatory systems
  - Fault tolerance: malicious component is simple ("noise" is not malicious), security: malicious component is complex
- Time scales
- Abstraction degree
  - How "leaky" are the abstractions?
  - Biological and social abstractions tend to be leaky; computer abstractions tend not to be
  - Security architectures are different in both cases

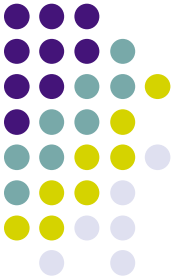
THE ADVENTURES OF SELFMAN



Complex versus simple

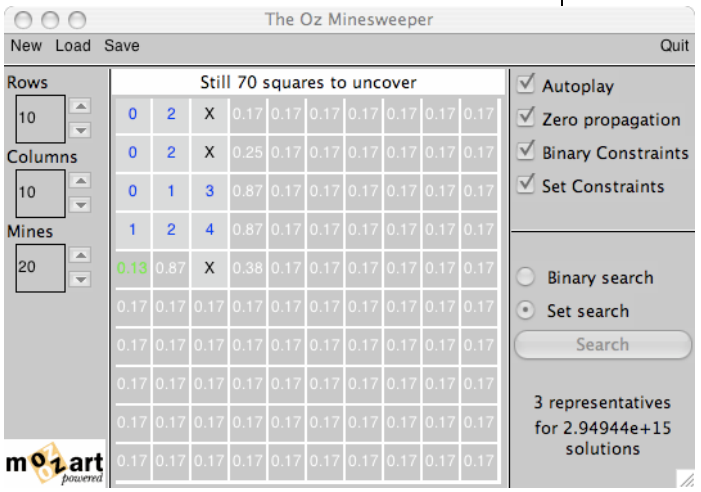
- A complex component can radically affect the behavior of the system
  - Complex cooperative component can stabilize an otherwise highly unstable system
  - Complex competitive component can unstabilize an otherwise highly stable system
- All four combinations do appear
  - {complex,simple} x {cooperative,competitive}
- How to design a system that has complex components?
  - If the component is imposed: defensive design (e.g., collective intelligence)
  - If the component is designed: it can improve system behavior, but fail-safe mechanisms must be built in (complex components will only enhance behavior in part of the configuration space)

THE ADVENTURES OF SELFMAN



Some complex components

- Human intelligence
  - Main strength: adaptability (dynamic creation of new feedback loops)
- Program intelligence
  - Can easily go beyond human intelligence in special areas!
    - **Turing test is moot**: complex boxes are 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



THE ADVENTURES OF SELFMAN

Time scale example (Wiener)

- Braking of a car (without ABS!)
  - Driver tests road traction by quick braking attempts
  - Driver then uses this information to help brake
- Short and long time scales are often independent (little leakage)
  - Use short time scale to gain information about the environment
  - This information helps the long time scale

THE ADVENTURES OF SELFMAN

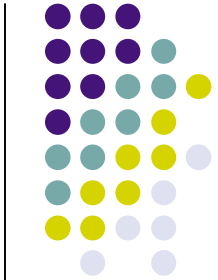
Three-layered architecture

Market architecture with utility function

Feedback loop architecture

Event architecture (concurrent components)

- Market architecture
  - Collective intelligence: optimizing a local utility (selfish behavior) will optimize the global utility
  - Works for competitive systems
- Feedback loop architecture
  - All components are part of feedback loops
  - Works well for cooperative systems
- Event architecture
  - Concurrent components that interact through events
  - Publish/subscribe events: any component that subscribes to a published type will receive the event
  - Eventually perfect failure detector with suspect and resume events

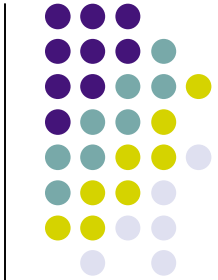


Feedback loop layer

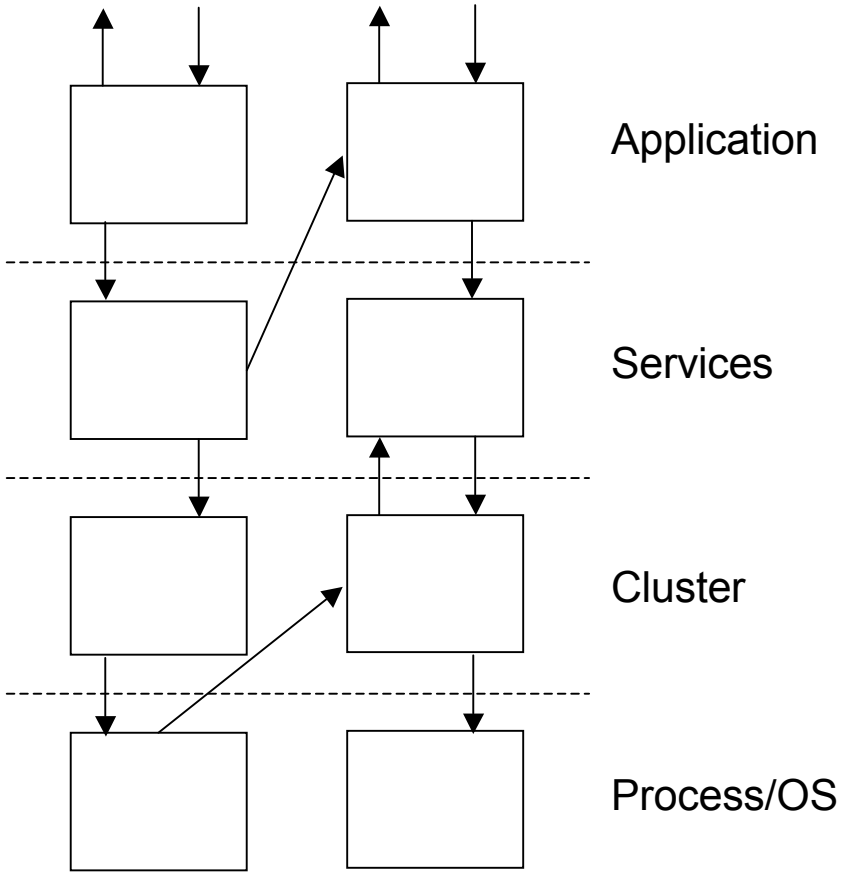
- A self-managing software system can be organized as a set of agents (instances of concurrent components) that communicate through asynchronous message passing
  - Event-based and publish/subscribe communication are adequate mechanisms
- The system is a hierarchy of interacting feedback loops, where each loop is implemented by several concurrent agents
- To allow the system to monitor and reconfigure itself, components must be first-class entities that allow higher-order component programming (e.g., the Fractal model [Bruneton *et al* 2004])
- Global properties of the system (total effect of all feedback loops) need to be monitored, e.g., using diffusion algorithms or belief propagation
  - There is a close relationship between global property monitoring and feedback monitoring

THE ADVENTURES OF SELFMAN

Observation: feedback loops are needed at all levels

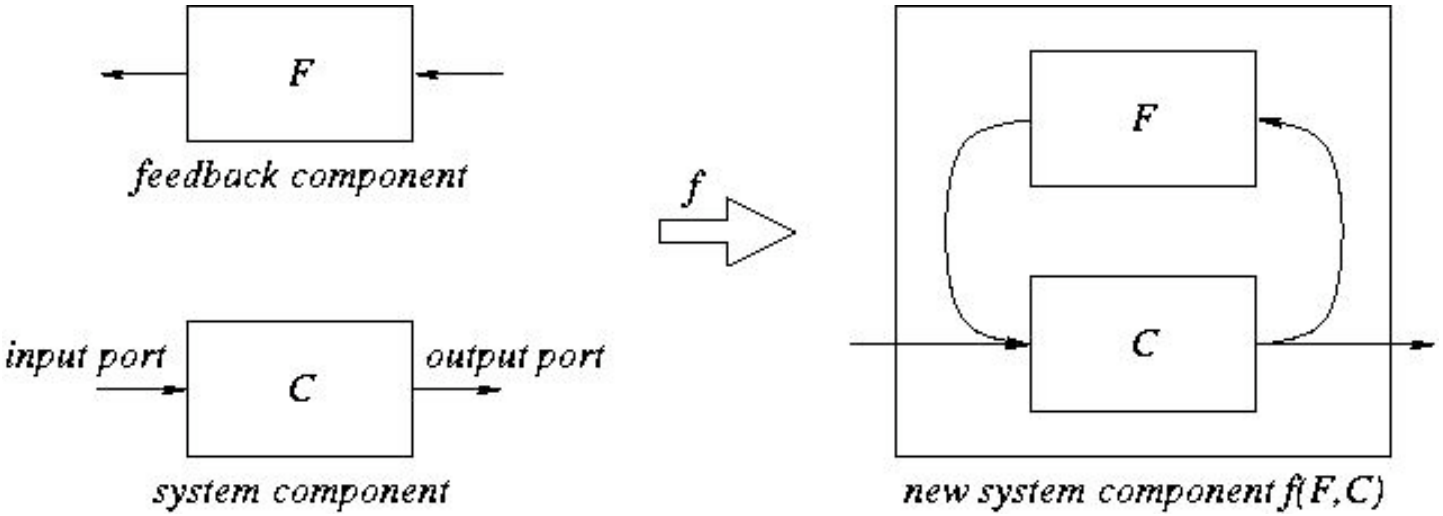


- Application level
  - User interaction
  - Self-describing components/software
  - "Autonomic Computing" techniques: removing humans from the loop
- Service levels
  - Loosely-coupled service infrastructure
  - Search and discovery of resources
  - Robust, self-organizing communication
  - Data management and replication
  - Redundancy-based fault tolerance
- Cluster level
  - Tightly-coupled infrastructure
  - Self-management services (e.g., demand prediction)
  - Scheduling services
  - Node replication and replacement
- Process/OS level
  - Node protection mechanisms (e.g., intrusion detection)
  - Software rejuvenation
  - Fault detection and alerting



THE ADVENTURES OF SELFMAN

Programming with feedback loops



- We can build feedback loops with a component combinator *f*
- We need different combinators depending on whether *C* or *F* is an explicit or implicit system (e.g., environment) and whether the loop is managed or not
- The semantics must take into account the input and output interleaving and the feedback delay

THE ADVENTURES OF SELFMAN

Programming with feedback loops in Mozart

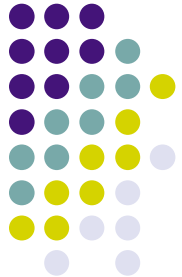
- We have programmed this in the Mozart Programming System using higher-order functions, lightweight concurrency, and dataflow synchronization
  - Mozart Programming System: an advanced multiparadigm programming platform (see www.mozart-oz.org)
- Component interface: one input port (accepts input events) and one output stream (produces ordered sequence of output events)
- Component behavior:
  - State × Event → State × Event* × (R^+,Event)*
  - Given an input state and an input event, create an output state, new output events, and new time-delayed input events
- Component combinators can be written in a few lines of code
- All the examples we have shown can be programmed (or simulated) easily

THE ADVENTURES OF SELFMAN

Mozart Programming System

- Mozart implements the Oz language
  - Oz is a multiparadigm language with lightweight threads, dataflow, symbolic, functional, logic, and object-oriented programming
  - Oz has a simple formal semantics
  - Mozart is a high-quality open-source implementation of Oz developed since the early 1990s (www.mozart-oz.org)
  - See “Concepts, Techniques, and Models of Computer Programming” (MIT Press, 2004)
- Mozart has advanced support for distributed programming
  - Network-transparent distribution with reflective failure detection
  - Recent development of Mozart Distribution Subsystem (Ph.D. work of Raphaël Collet and Erik Klintskog)
    - Choice of distribution protocols for language entities
    - Asynchronous event-based interface to failure detection
    - Kill operation to manage fault tolerance
    - Support for temporary failures (imperfect failure detection: Internet failures)
- Mozart has advanced support for logic and constraint programming
  - For building complex components (e.g., minesweeper assistant shown earlier)

THE ADVENTURES OF SELFMAN



Collective intelligence

- How do we get selfish agents to work together for the common good?
- The system (called a “collective”):
  - Has a global utility function that measures the system’s performance
  - Is composed of many selfish agents that each tries to optimize its private utility
- Who does what
  - The system designers define the agent’s private utility and how the agent’s actions affect its private utility
  - The agents choose their actions within the system
- The goal: agents acting to optimize their private utilities should also optimize the global utility
  - There is no other mechanism to force cooperation
  - This is in fact how society is organized. For example, employees act to optimize their salaries/work satisfaction and this benefits the company.

THE ADVENTURES OF SELFMAN

Example: El Farol bar problem

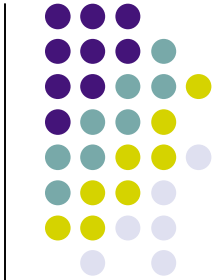
- People go to El Farol once a week to have fun
  - B. Arthur introduced this problem in [Arthur 1994]
- Each person picks which night to attend the bar
  - If the bar is too crowded or too empty it is no fun
  - Otherwise they have fun (receive a reward)
- Each person makes one decision per week
  - All they know is last week's attendance
  - People don't interact to make their decision (only stigmergy)!
- What strategy should each person use to maximize his/her fun?
  - To avoid a "Tragedy of the Commons" where everybody maximizing their local utility causes minimization of the global utility

THE ADVENTURES OF SELFMAN

Collective intelligence solution

- Global utility G = $\Sigma_{w\ (week)}$ W(w) (week utility)
  - Week utility W(w) = $\Sigma_{d\ (weekday)}\ \phi_d(a_d)$ where a_d is total attendance on weekday d for week w
  - $\phi_d(y) = \alpha_d y \exp(-y/c)$ (picked by system designer: low when y is too low or too high, optimum somewhere in the middle)
- Local utility = agent reward function
  - The reward that the agent gets for its choice
  - The system designer picks this function: we will see how to do it!
- Each agent would like to maximize its reward. For example, it can use a learning algorithm:
  - It picks a night randomly according to a Boltzmann distribution distributed according to the energies in a 7-vector
  - When it gets its reward, it updates the 7-vector accordingly
  - (Real agents may use other algorithms. We pick this one just so that we can simulate the problem [Wolpert et al 1999].)

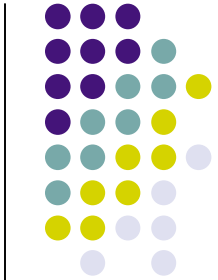
THE ADVENTURES OF SELFMAN



What is the agent's reward?

- How do we design the agent's reward function?
  - There are many bad utility functions
    - For example, Uniform Division: divide $\phi_d(y)$ uniformly among all agents present on day y. This is particularly awful!
  - One utility that works surprisingly well is called the Wonderful Life utility:
    - $R_{WL}(w) = W(w) - W_{\text{agent is absent}}(w)$
    - I.e., we calculate W(w) when the agent is missing (dropped from the attendance vector)
    - $R_{WL}(w)$ is the difference that the agent's existence makes (hence the name "Wonderful Life", from the Frank Capra movie)
- With the Wonderful Life utility, if each agent maximizes its reward, the global utility will also be maximized

THE ADVENTURES OF SELFMAN



How can we use this?

- How can we use this idea for building collective services?
  - The agent chooses its action and the system calculates the reward
    - The system is built using cryptographic protocols so that the agent cannot “hack” its reward
  - We assume that agents will try to maximize their rewards
- This does not solve all security problems
  - For example, collusion: when many agents get together to break the system
- But it can be useful
  - In many cases, the agents cannot or will not talk to each other
    - Collective intelligence is one way to get them to cooperate
  - One example is the “grey goo” problem in the Second Life application: self-replicating objects that use up resources
    - A solution could be based on collective intelligence

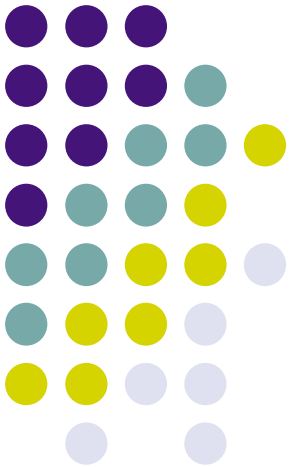
THE ADVENTURES OF SELFMAN

Related ideas

- Game theory: studies strategic interactions between rational players
  - "Rational" strategies for players to maximize their payoffs
  - Strictly dominant strategy for a player:
    - Always gives better result than dominated strategy, no matter what the other players do
  - Pareto efficiency:
    - Cannot change a strategy to make one player better off without making some other player worse off
  - Nash equilibrium:
    - Each player has a local maximum (cannot unilaterally change its strategy without reducing its payoff)
    - Much more controversial: not always the best strategy for the player!
  - These are mathematical notions, not always applicable to the real-world
- Agoric systems: software design based on market principles
  - Markets can achieve "intelligent" regulation that cannot be achieved by a subset of the market (like a regulatory agency or a manager) [Hayek]

THE ADVENTURES OF SELFMAN

Decentralized distributed systems



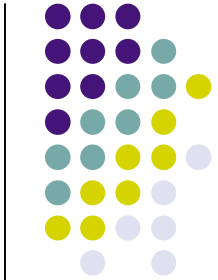
THE ADVENTURES OF SELFMAN

Building robust distributed systems

- How can one build robust distributed systems?
  - One approach is to make them decentralized and self-managing
    - No single point of failure, every node can play any role
  - A good example is the structured overlay network, which is an example of a peer-to-peer network with strong self-organizing properties
- We examine one design, the relaxed ring [Mejias et al 2007]
  - A self-organizing decentralized distributed system that works in the presence of Internet-style failures (permanent node failures and temporary network failures, with false suspicions)
  - The relaxed ring is a form of structured overlay network
- We use structured overlay networks (SONs) as the basis for self-managing distributed systems
  - We build the three-layer architecture on top of them

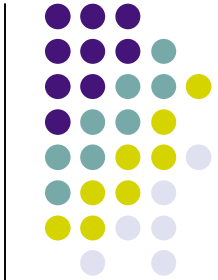
THE ADVENTURES OF SELFMAN

Context of the work

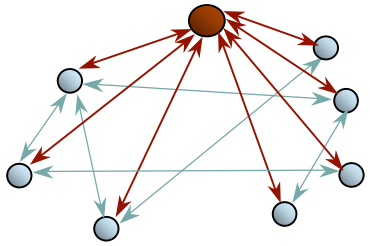


- This work is done in the following projects
  - SELFMAN project (2006-2009) on self management of large-scale distributed systems (see www.ist-selfman.org)
  - EVERGROW integrated project (2004-2007) on complexity and large-scale distributed systems (see www.evergrow.org)
  - CoreGRID network of excellence (2004-2008) on grid and peer-to-peer systems (www.coregrid.net)
- Our main implementation work is done in SELFMAN
  - Build a self-managing system based on a structured overlay network
  - We have developed the P2PS library in the Mozart Programming System (www.mozart-oz.org)
- We have implemented the relaxed ring in P2PS and we have built a visualization tool to observe its behavior
  - We are studying its behavior (on PlanetLab) and simulating it (on EVERGROW cluster)
- The P2PS library will be the basis for building decentralized services (storage and transactions) and applications using them
  - In the second and third years of SELFMAN

Structured overlay networks: inspired by peer-to-peer

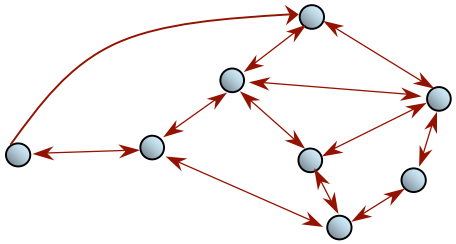


- Hybrid (client/server)
  - Napster



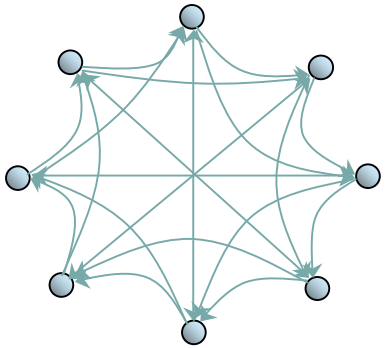
R = *N*-1 (hub)
R = 1 (others)
H = 1

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



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

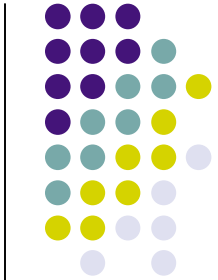
- Structured overlay
  - Exponential network
  - DHT (Distributed Hash Table), e.g., Chord, DKS, P2PS



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

THE ADVENTURES OF SELFMAN

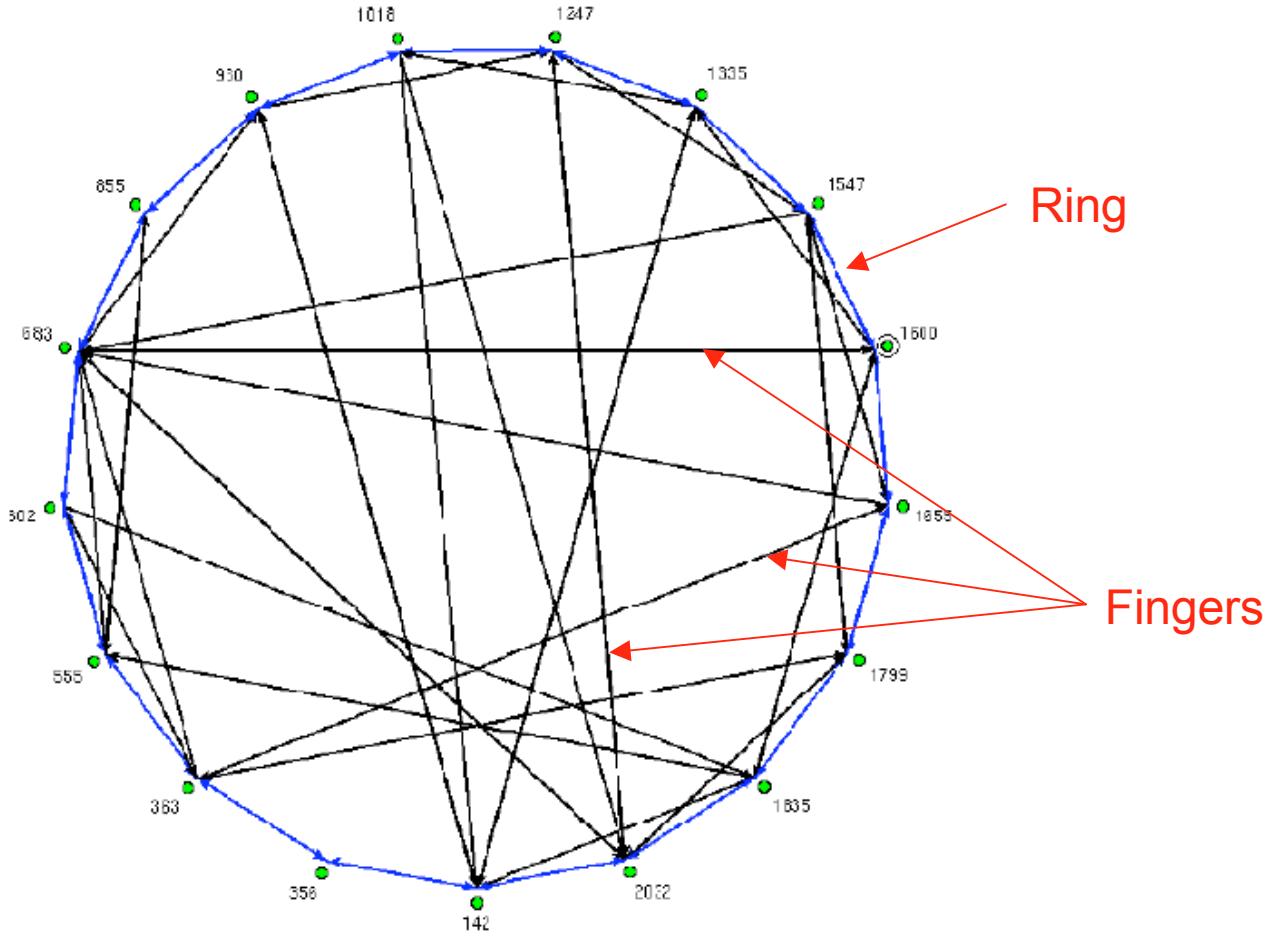
Properties of structured overlay networks



- Scalable
  - Works for any number of nodes
- Self organizing
  - Routing tables (fingers) updated with node joins/leaves
  - Routing tables updated with node failures
- Provides guarantees and efficiency (unlike flooding approach)
  - If operated inside of failure model, then communication is guaranteed with an upper bound on number of hops
  - Broadcast can be done with a minimal number of messages
- Provides basic services
  - Name-based communication (point-to-point and group)
  - DHT (Distributed Hash Table): efficient storage and retrieval of (key,value) pairs

THE ADVENTURES OF SELFMAN

Based on a ring topology



P2PS organization with fingers from Tango protocol

THE ADVENTURES OF SELFMAN

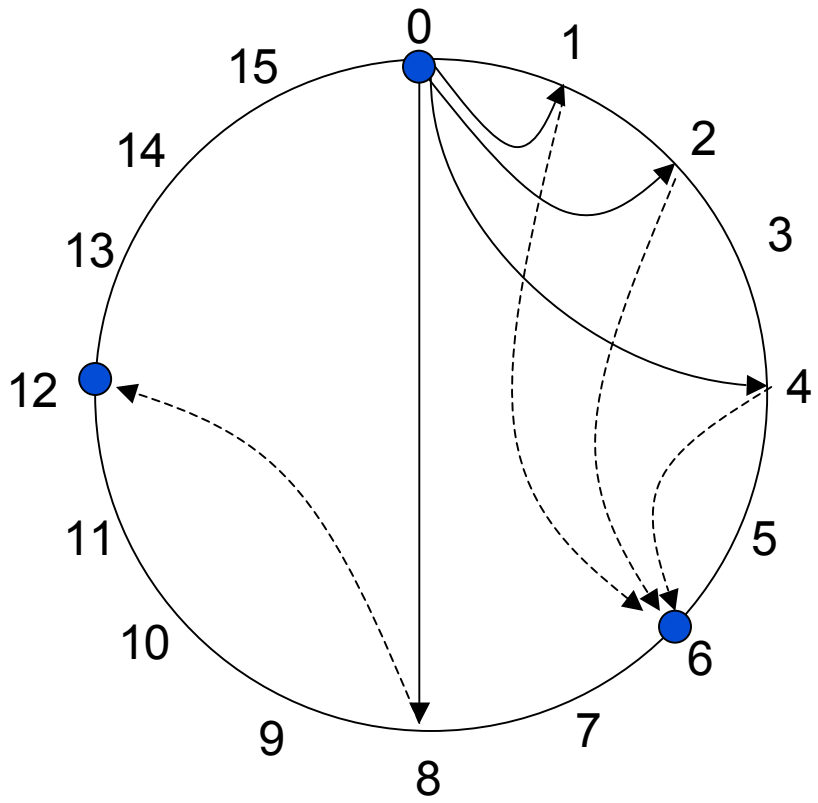
Lookup illustrated in Chord

We illustrate lookup in Chord, a simple SON. Nodes sparsely populate a circular identifier space.

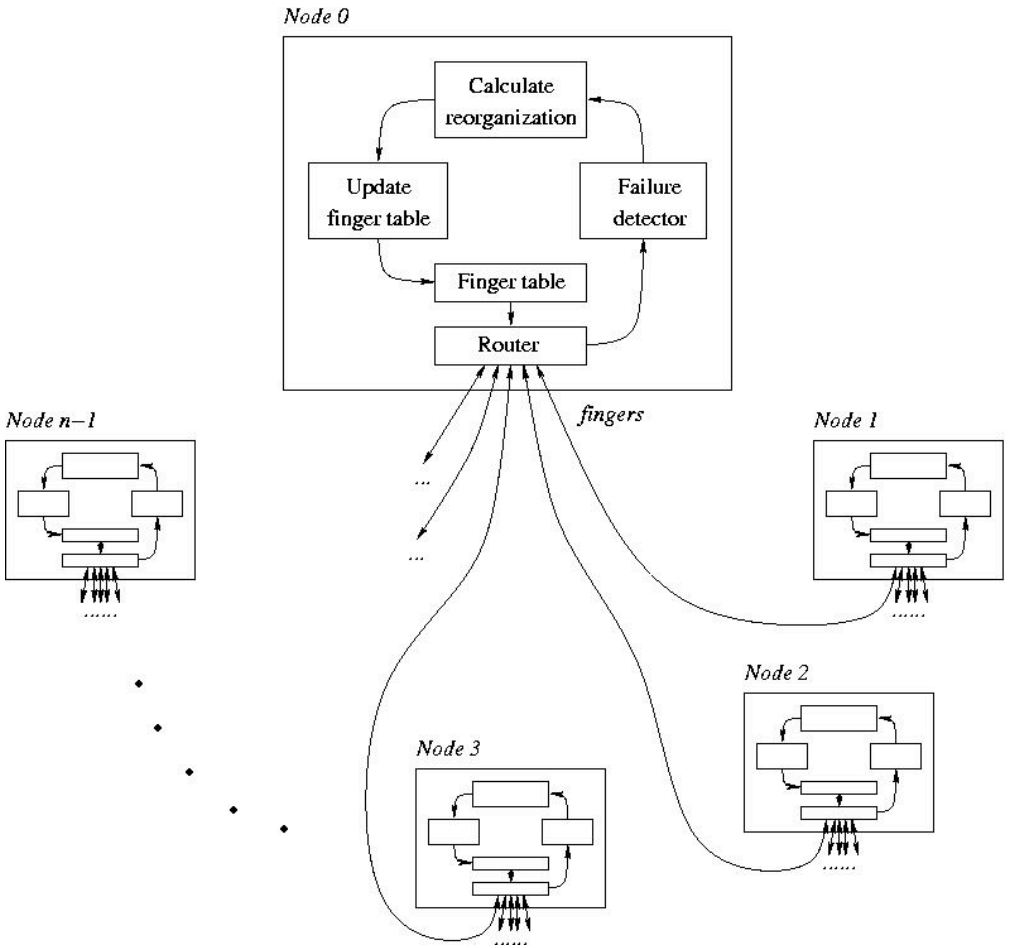
Given a key, find the value associated to the key (here, the value is the IP address of the node that stores the key**)**

Assume node 0 searches for the value associated to key K with identifier 7

Interval	node to be contacted
[0,1)	0
[1,2)	6
[2,4)	6
[4,8)	6
[8,0)	12

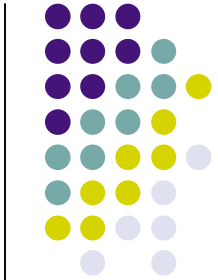


Feedback loops in the ring



- The primitive functionality of a SON is to self-organize its nodes to provide reliable and efficient routing, despite nodes continuously joining, leaving, and failing
- Study of SONs has blossomed since the development of Chord in 2001 [Stoica *et al* 2001]
- SON operation is based on ***three convergence properties***:
  - Within each node, the finger table converges to a correct content
  - Globally, the finger tables converge together to improve routing efficiency
  - When routing, a message in transit converges to its destination node
- Proving correctness:
  - Need atomic join/leave/fail operations
  - Need ability to work with strongly complete failure detection
  - First proved in [Ghodsi 2006]

THE ADVENTURES OF SELFMAN

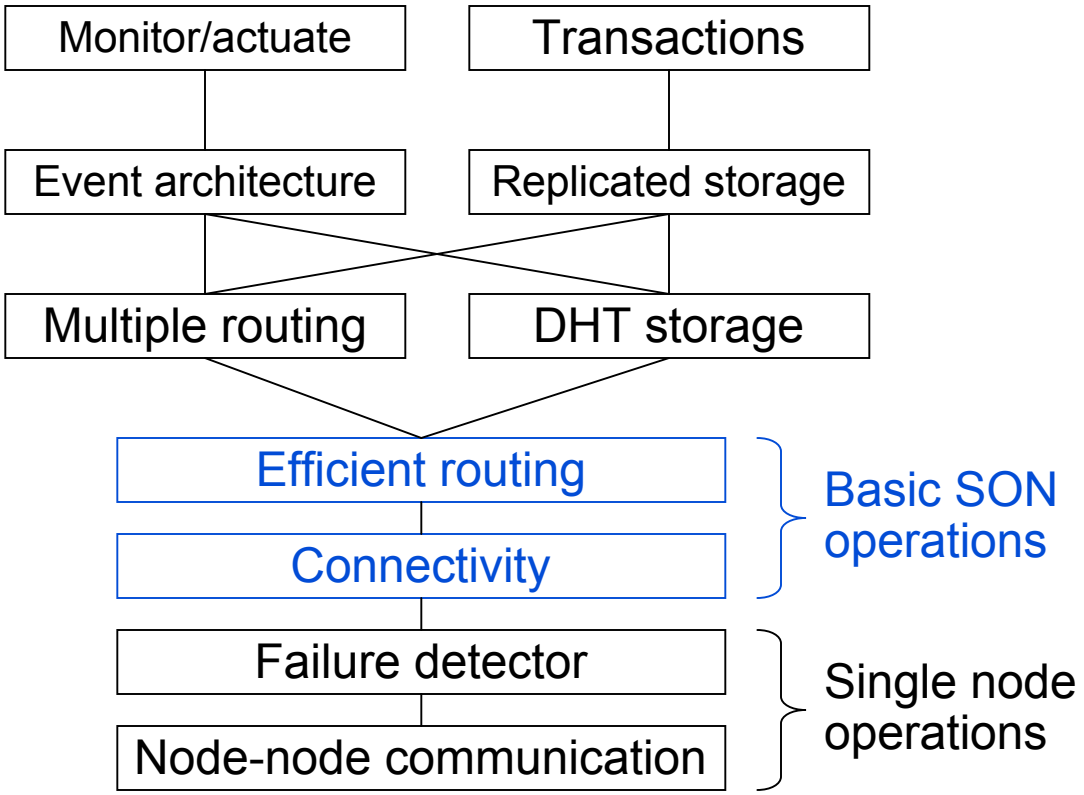


Ring maintenance

- In a SON based on a ring structure, self-organization is done at two levels:
  - The ring ensures connectivity: it must always exist despite joins, leaves, and failures
  - The fingers provide efficient routing: they can be temporarily in an inconsistent state
- These two layers are part of a more general layered architecture…

THE ADVENTURES OF SELFMAN

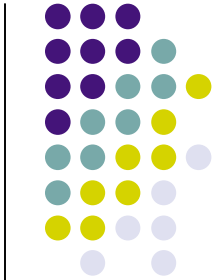
SON layered architecture for self-managing systems



- An instance of the three-layered architecture shown before
- Implementation
  - Lower 4 layers exist in P2PS library
  - Other layers in progress
- Adding services
  - One instance per node
- Self management
  - Inside a layer (e.g., connectivity)
  - Between layers (monitoring and actuating services)

THE ADVENTURES OF SELFMAN

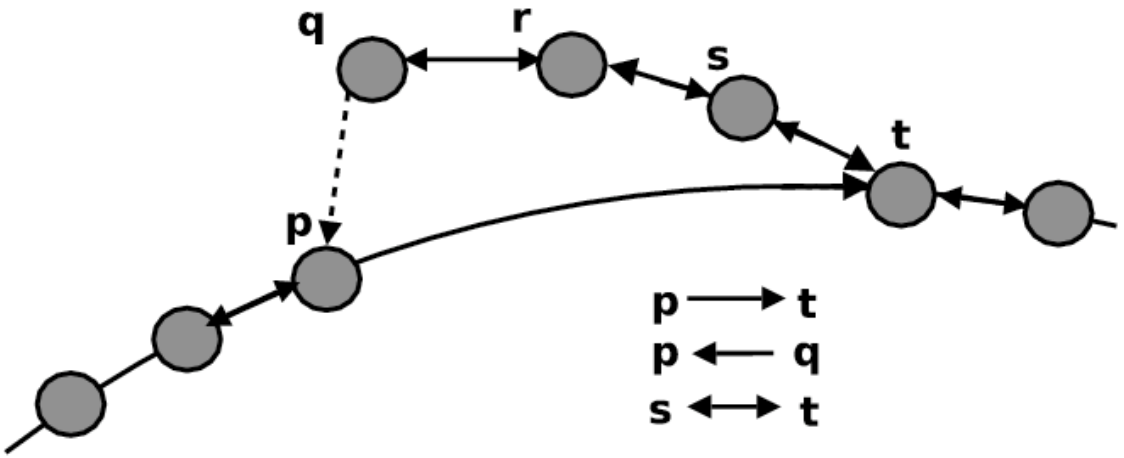
Doing ring maintenance



- Ring maintenance is not a trivial issue
  - Peers can join and leave at any time
  - Peers that crash are like peers that leave but without notification
  - Temporarily broken links create false suspicions in failure detection
- Crucial properties to be guaranteed
  - Lookup consistency
  - Ring connectivity

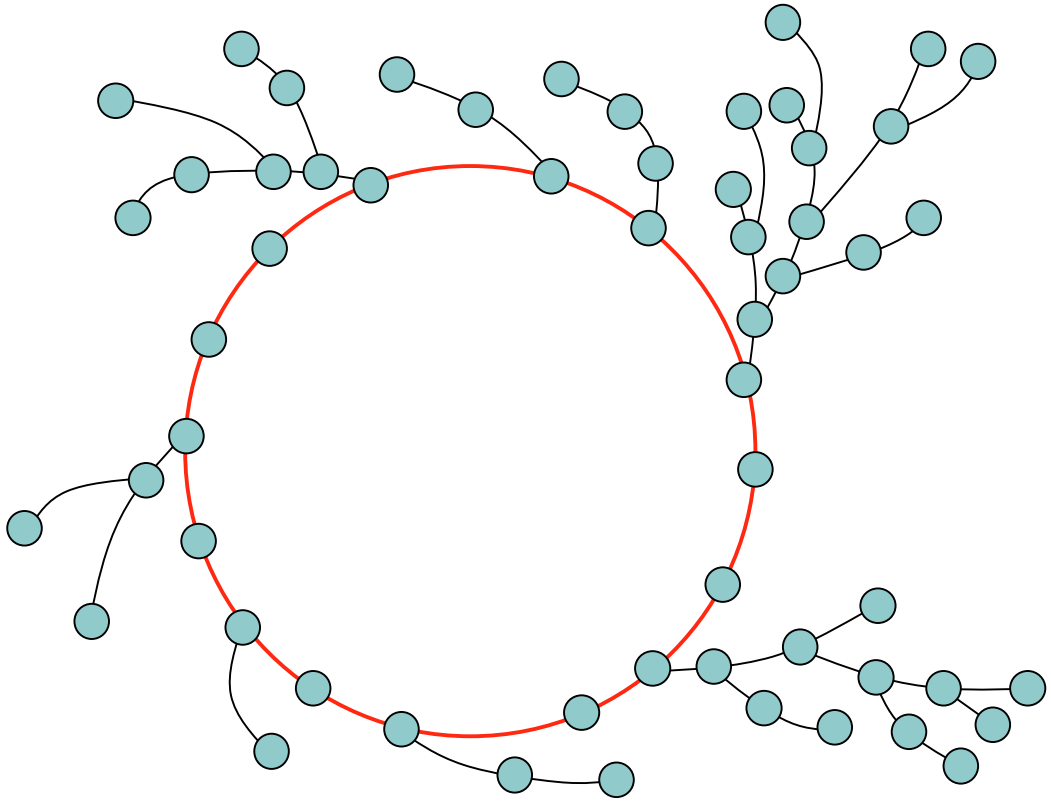
The relaxed-ring architecture

- Relaxed ring maintenance is completely asynchronous
- Nodes communicate through message passing
  - For a join, instead of one step involving 3 peers (as in DKS, also developed in SELFMAN), we have two steps each with 2 peers → we do not need locking
- Invariant: Every peer is in the same ring as its successor
- A peer can never indicate another peer as the responsible node (a peer knows only its own responsibility, which starts with the key of the predecessor + 1)



THE ADVENTURES OF SELFMAN

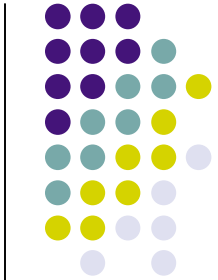
Example of a relaxed ring



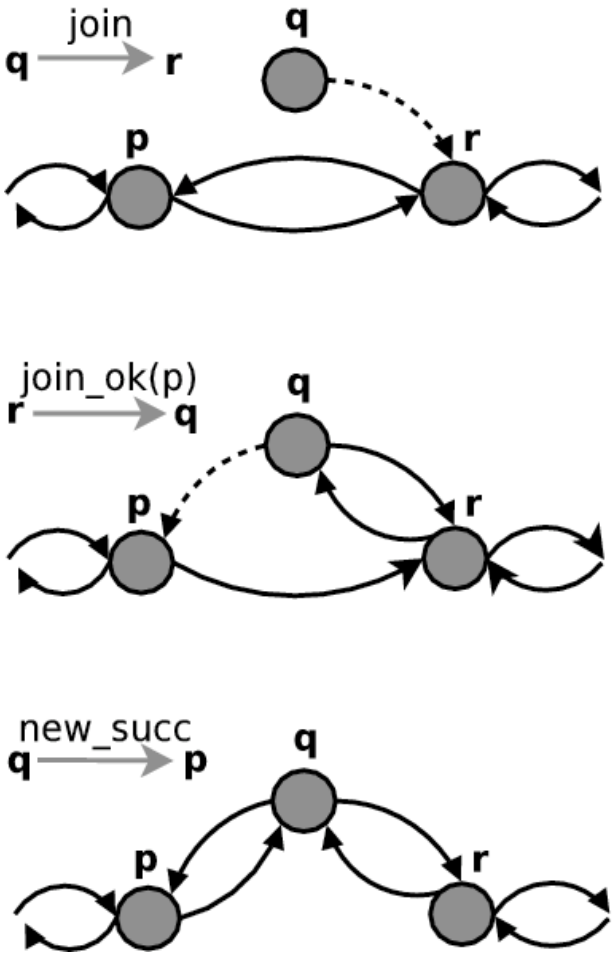
- It looks like a ring with “bushes” sticking out
- The bushes appear only if there are failure suspicions
  - Usually the ring is not as bushy as in this example!
- There always exists a perfect ring (in red) as a subset of the relaxed ring
- The relaxed ring is always converging toward a perfect ring
  - The number of bushes existing at any time depends on the churn (rate of change of the ring, failures/joins per time)

THE ADVENTURES OF SELFMAN

The join algorithm



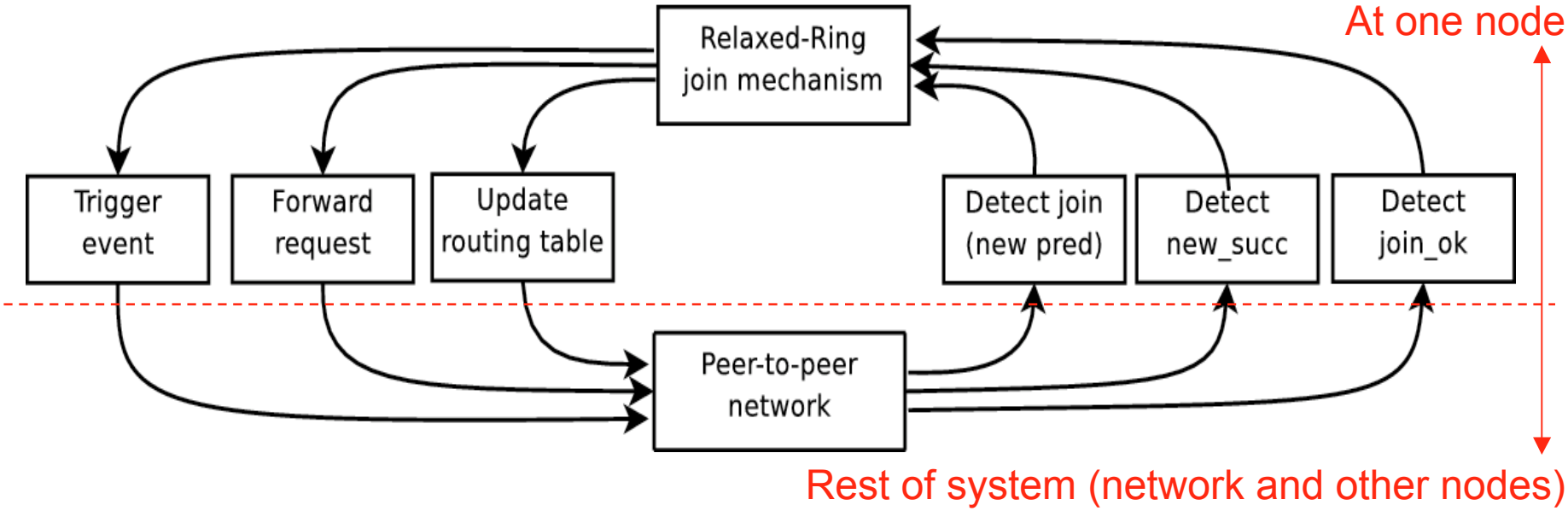
- joining peer ***q*** requests a lookup for its key
- ***q*** sends the ***join*** message to its successor candidate ***r***
- ***r*** accepts ***new pred*** and sends reference ***p*** to ***q***
- ***q*** contacts ***p*** to inform that is its ***new succ***



THE ADVENTURES OF SELFMAN

Join feedback loops

- Model algorithm as a feedback loop
  - Feedback loops are the primitive concept of self-managing systems
  - Boxes are concurrent components, arrows are asynchronous message flows
- Events perturbing the stability of the ring are constantly monitored
- Corresponding corrective actions are triggered



THE ADVENTURES OF SELFMAN

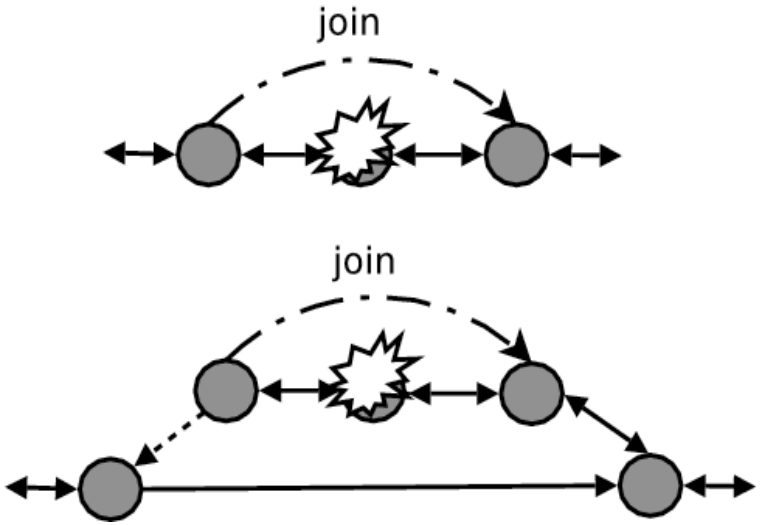
Lookup consistency

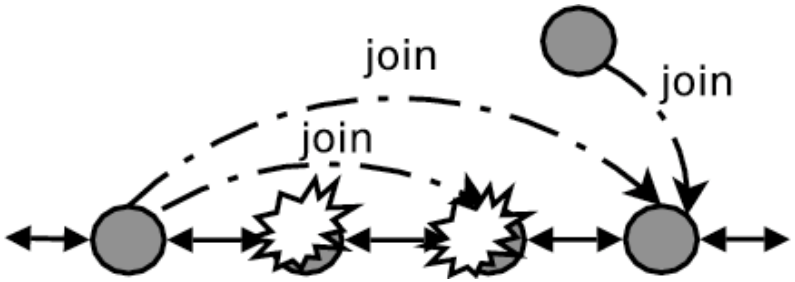
- **Definition:** Lookup consistency means that at any time there is only one responsible node for a particular key ***k***
  - In the case of temporary failures (imperfect failure detection) lookup consistency cannot always be guaranteed: we may temporarily have more than one responsible node
  - Failure model: nodes may fail permanently and network links may fail temporarily, with eventually perfect failure detector (accurate: permanent failure is always detected, eventually perfect: false suspicion is possible, but only temporarily)
- **Theorem:** The relaxed-ring join algorithm guarantees lookup consistency at any time in presence of multiple joining peers
  - This is not true for Chord
  - When there are multiple failing peers, this is not guaranteed but the time interval of inconsistency is small

THE ADVENTURES OF SELFMAN

Failure recovery

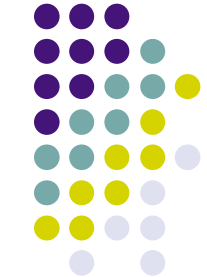
- When a peer detects that its successor has crashed, it contacts the first peer in its successor list for recovery
  - Resilience is based on the length of the successor list



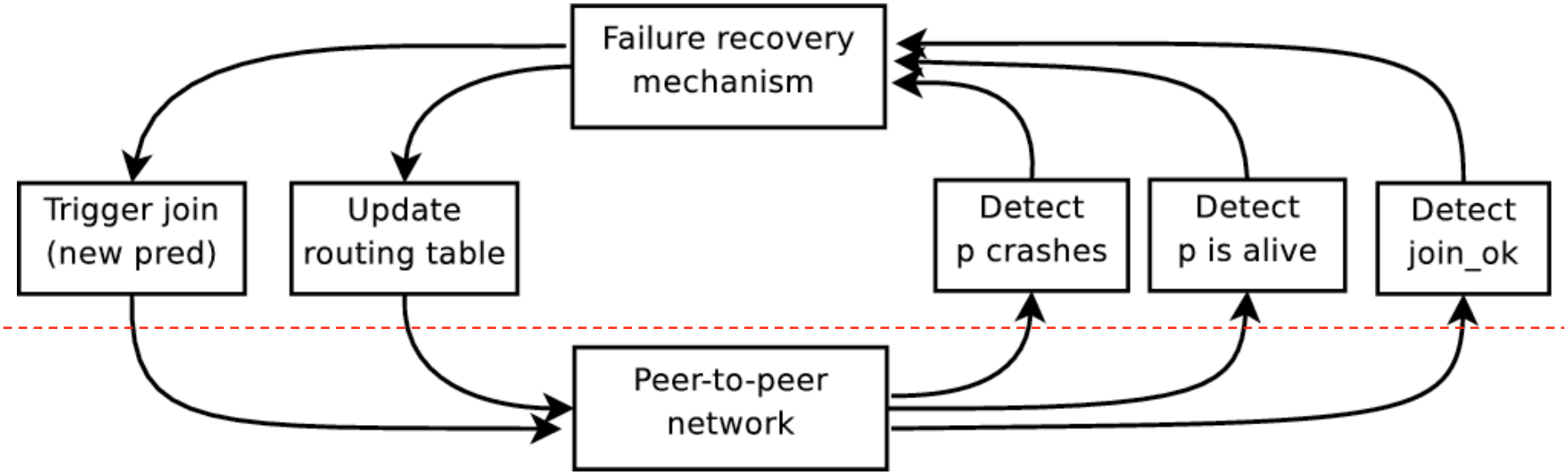


THE ADVENTURES OF SELFMAN

Failure recovery feedback loops

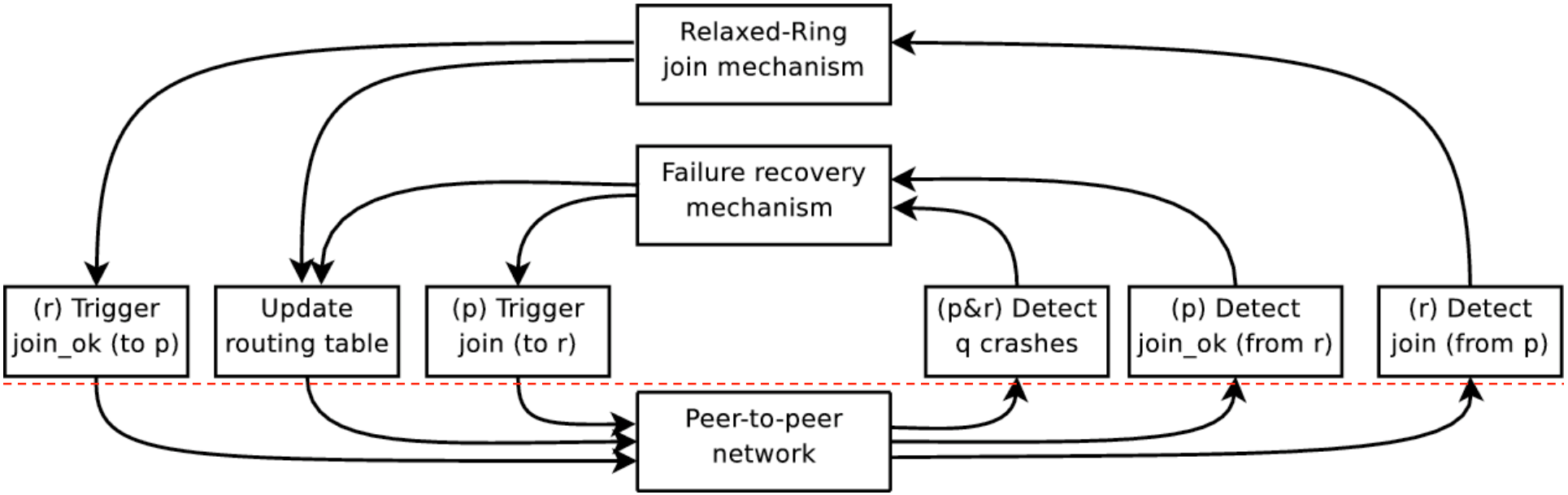


- Failures are perturbations of the relaxed-ring topology
- Join mechanism is triggering for fixing the topology



THE ADVENTURES OF SELFMAN

Failure recovery triggers join



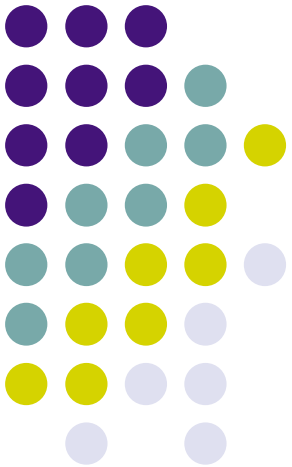
THE ADVENTURES OF SELFMAN

Relaxed ring conclusions

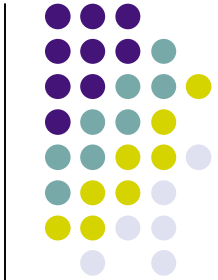
- The relaxed-ring guarantees lookup consistency and ring connectivity while simplifying the ring maintenance and assuming a realistic failure detection
  - Ring connectivity is maintained by relaxing the ring structure; this allows handling imperfect failure detection (false suspicions), i.e., Internet-style failure detection
  - Lookup consistency is maintained on joins, with the single limitation that temporarily the lookup may not succeed
  - Lookup consistency is not maintained on multiple failures, but the inconsistency is temporary
  - Resilience of the system is currently limited to a fixed number of permanent node failures (does not currently handle network partitioning)
- We have implemented the system in P2PS and we will use it in the SELFMAN project to build a self-managing application using storage and transaction services

THE ADVENTURES OF SELFMAN

Conclusions



THE ADVENTURES OF SELFMAN



Research agenda

- Software systems should be self managing
  - Self management has a role to play in general software development; it has too long been relegated to specialized subdomains
  - All parts of the system should be inside one or more feedback loops
  - There is a research agenda here!
- We propose a layered architecture
  - Event-driven foundation (concurrent components, asynchronous comm.)
  - Feedback loops (for cooperative part)
  - Market principles, e.g., collective intelligence (for competitive part)
  - Design for a desired global behavior
- This is work in progress
  - Architecture of a decentralized distributed system (using relaxed ring)
  - We are applying these principles in the SELFMAN project (EU 6FP, since June 2006): www.ist-selfman.org
  - We are combining a structured overlay network (which is already self managing at a low level) with an advanced component model, to achieve a self-management architecture for large-scale distributed systems