THE ADVENTURES OF SELFMAN

Self Management and the Future of Software Design

October 25, 2006

Peter Van Roy
Coordinator, SELFMAN project
Université catholique de Louvain
Louvain-la-Neuve, Belgium

European Commission

Information Society

Software and the Red Queen

- Software is fragile!
  - A single bit error can cause a catastrophe
- Hardware has been reliable enough so that this has not unduly hampered the quantity of software being written
  - We are in a Red Queen situation: running as hard as we can to stay in the same place
  - New techniques (structured programming, OOP, the usual bunch of modern methodologies – agile, extreme, etc.) have arguably kept pace so far
- 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 increased connection of small devices to the Internet
- 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.)
  - A mix of client/server and peer-to-peer architectures
  - These applications are still rather conservative: they do not take advantage of the new complexity space

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
- In addition, programming large-scale distributed systems introduces other problems
  - 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]

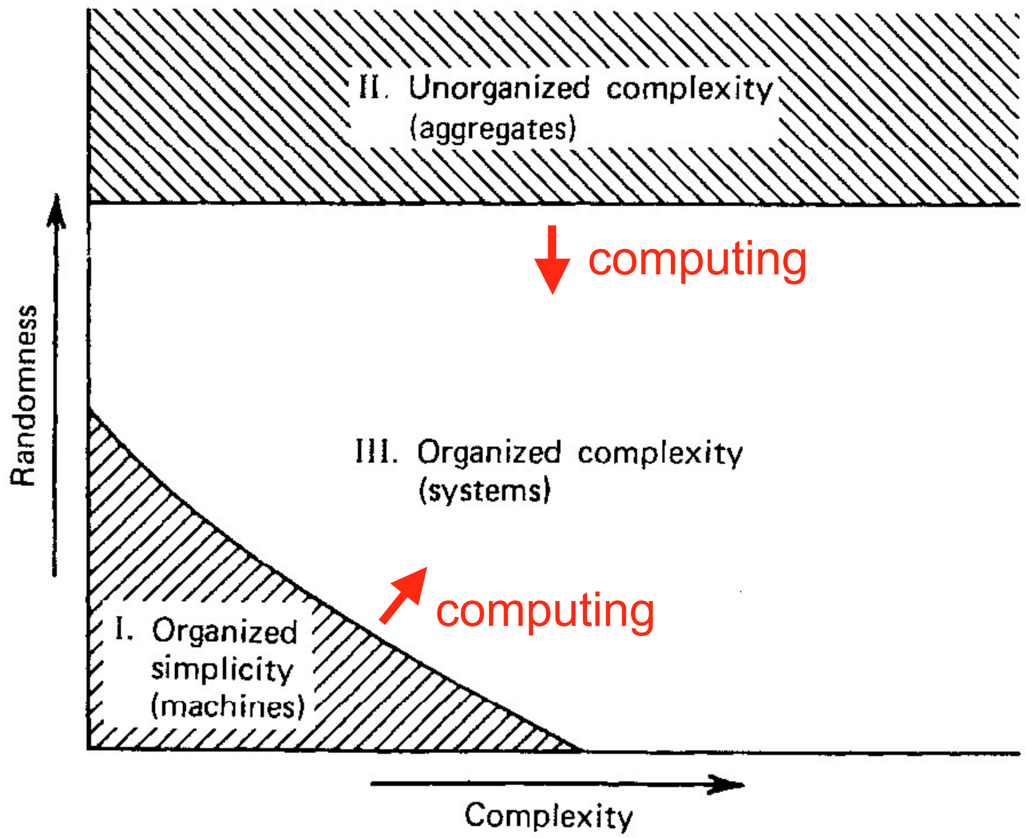
The next solution

- Now that we have set the stage, what solution do we propose?
- 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

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
- Is there a bigger role for self management?

Types of systems



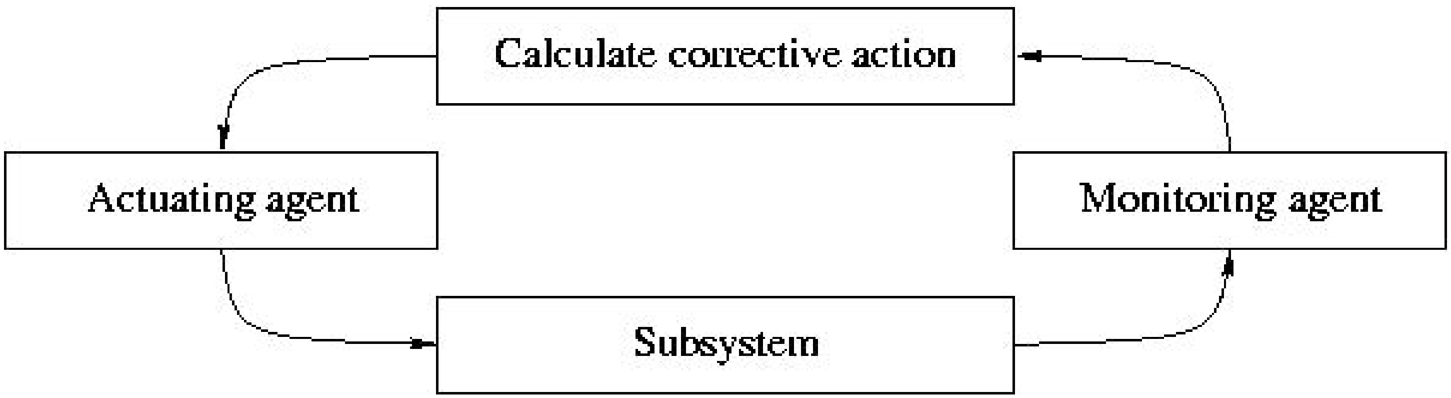
- This diagram is from [Weinberg 1977] *An Introduction to General Systems Thinking*
- The discipline of computing is pushing the boundaries of the two shaded areas inwards
- Software development methodologies are the vanguards of system theory

THE ADVENTURES OF SELFMAN

Designing self-managing software systems

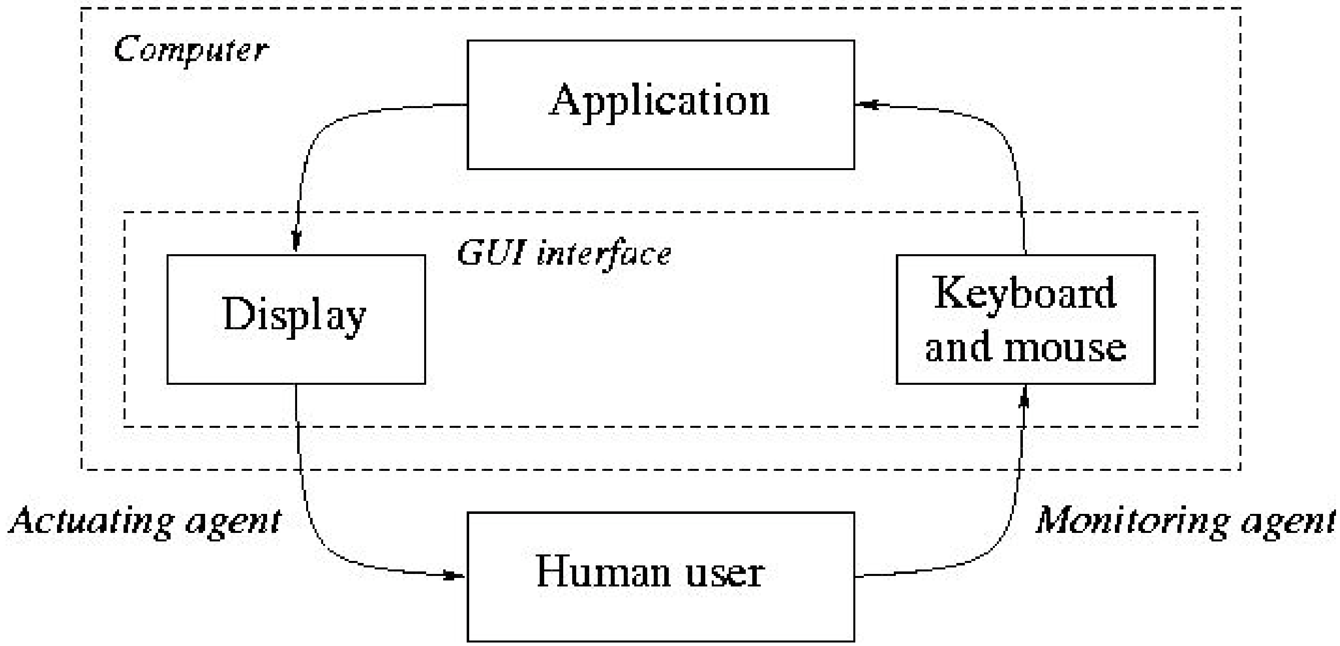
- From system theory, we take the fundamental principles
  - Programming with feedback loops
  - Focus on global (emergent) properties
  - Architectural framework
- We will use these principles as a basis for practical software development
  - This talk will give a few ideas on how to do this; our work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

Feedback loops



- A feedback loop consists of three elements that interact with a subsystem: a monitoring agent, a correcting agent, and an actuating agent
- Feedback loops can interact in two ways:
  - two loops that affect interdependent system parameters (stigmergy)
  - one loop that directly controls another loop (management)

Feedback loops are everywhere

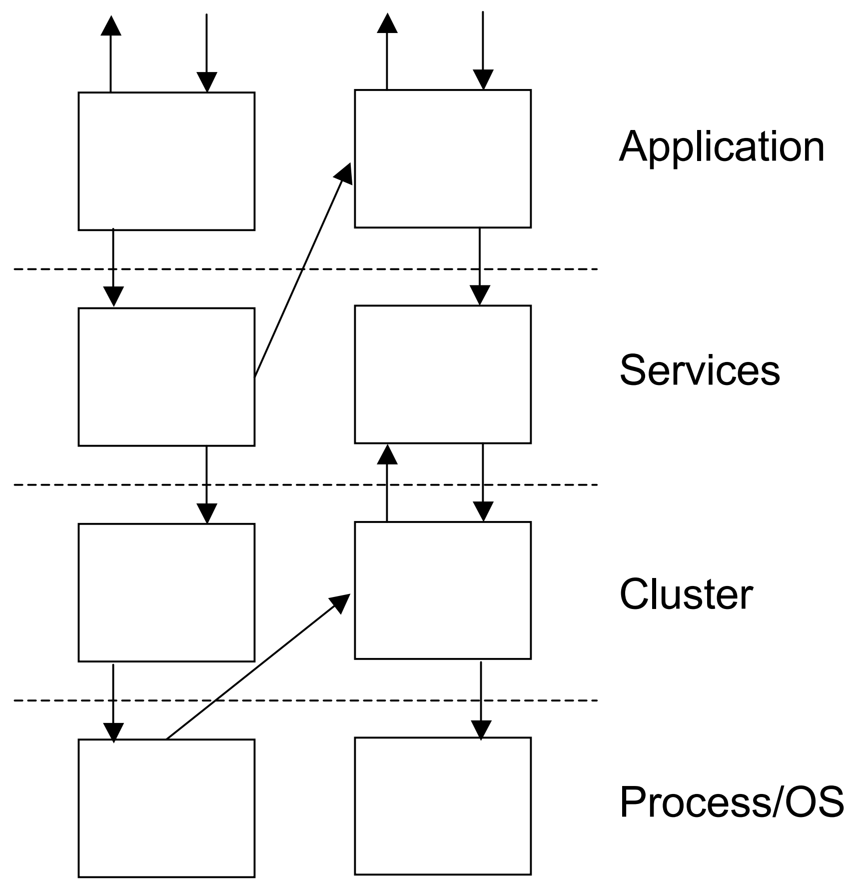


- Feedback loops are literally everywhere, if you look at a system with the right mindset
- A single-user application is a simple example

THE ADVENTURES OF SELFMAN

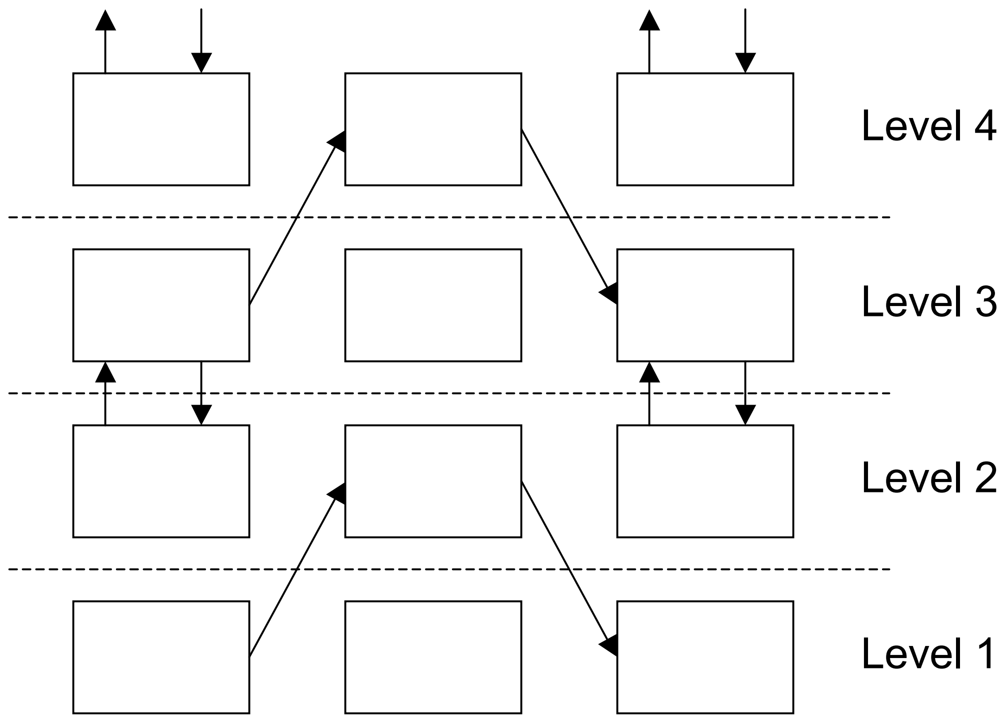
Feedback loops are needed at all levels

- 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

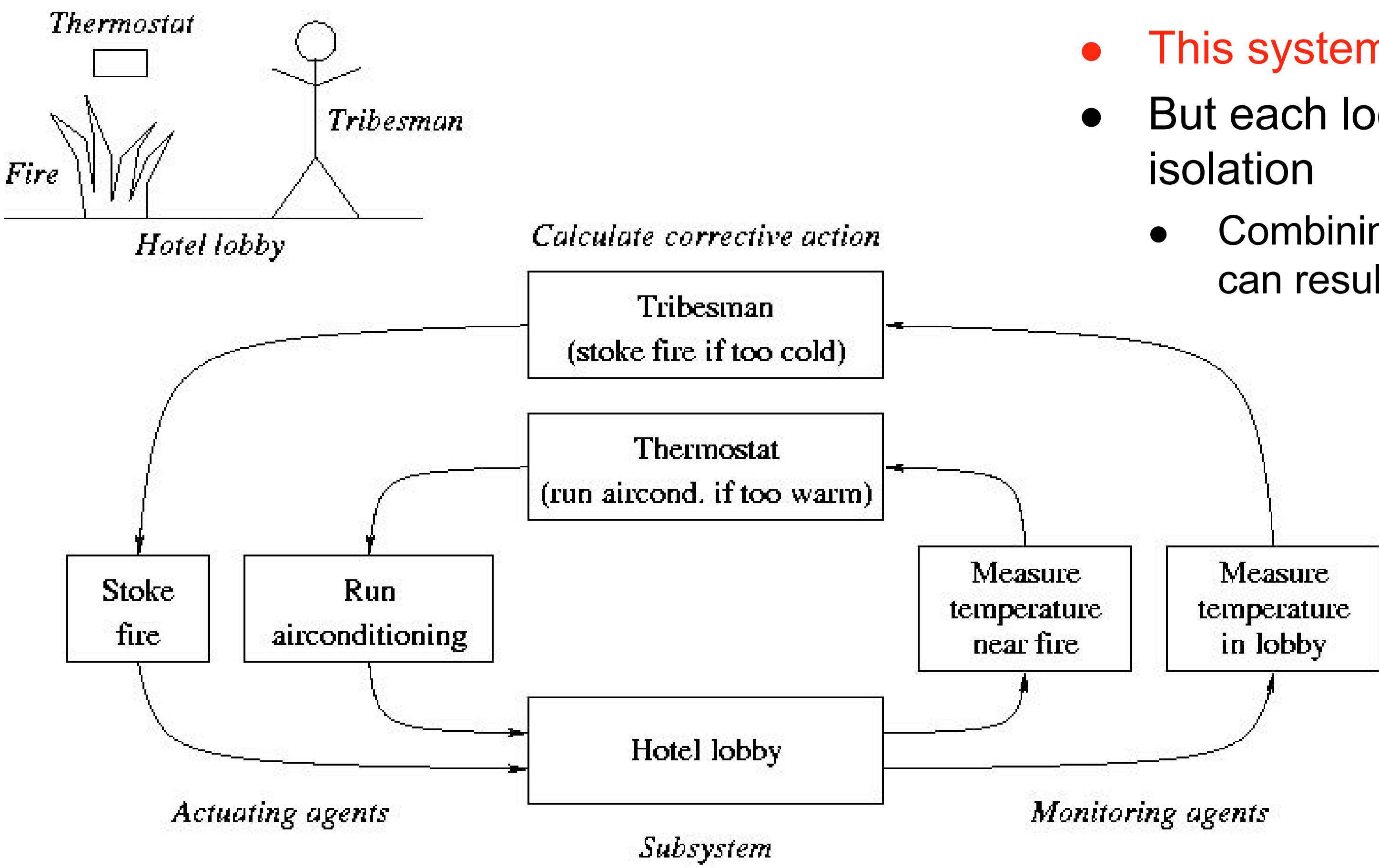


Complexity of interacting feedback loops

- 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**

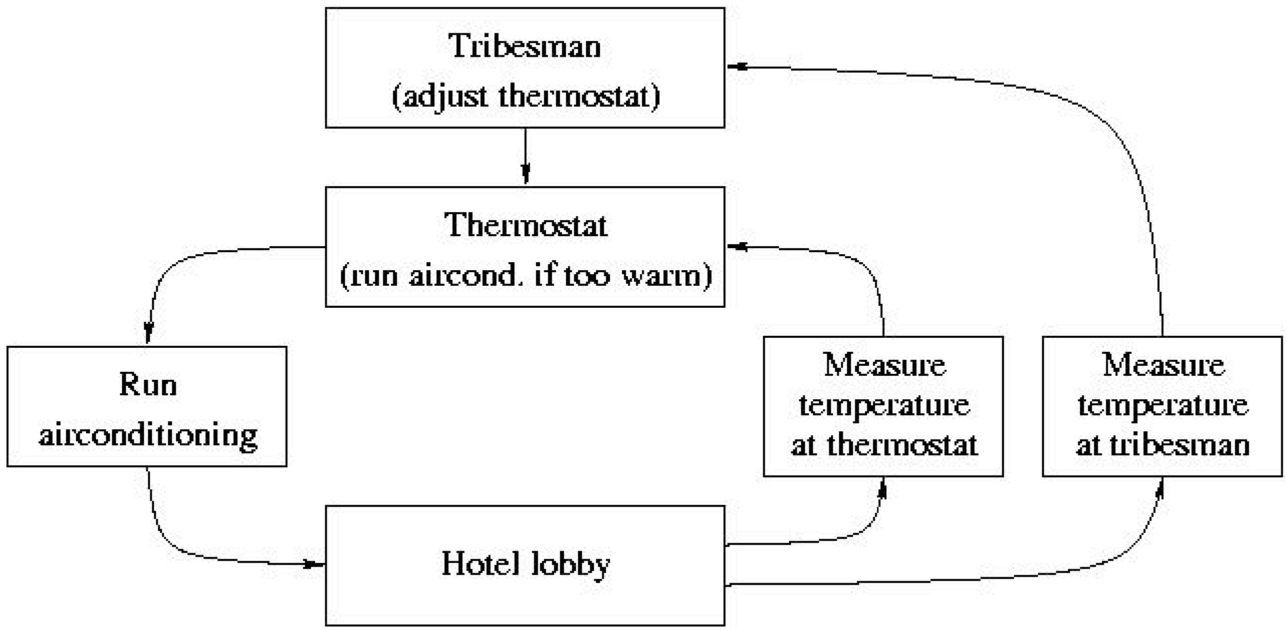


Example of stigmergy (Wiener)



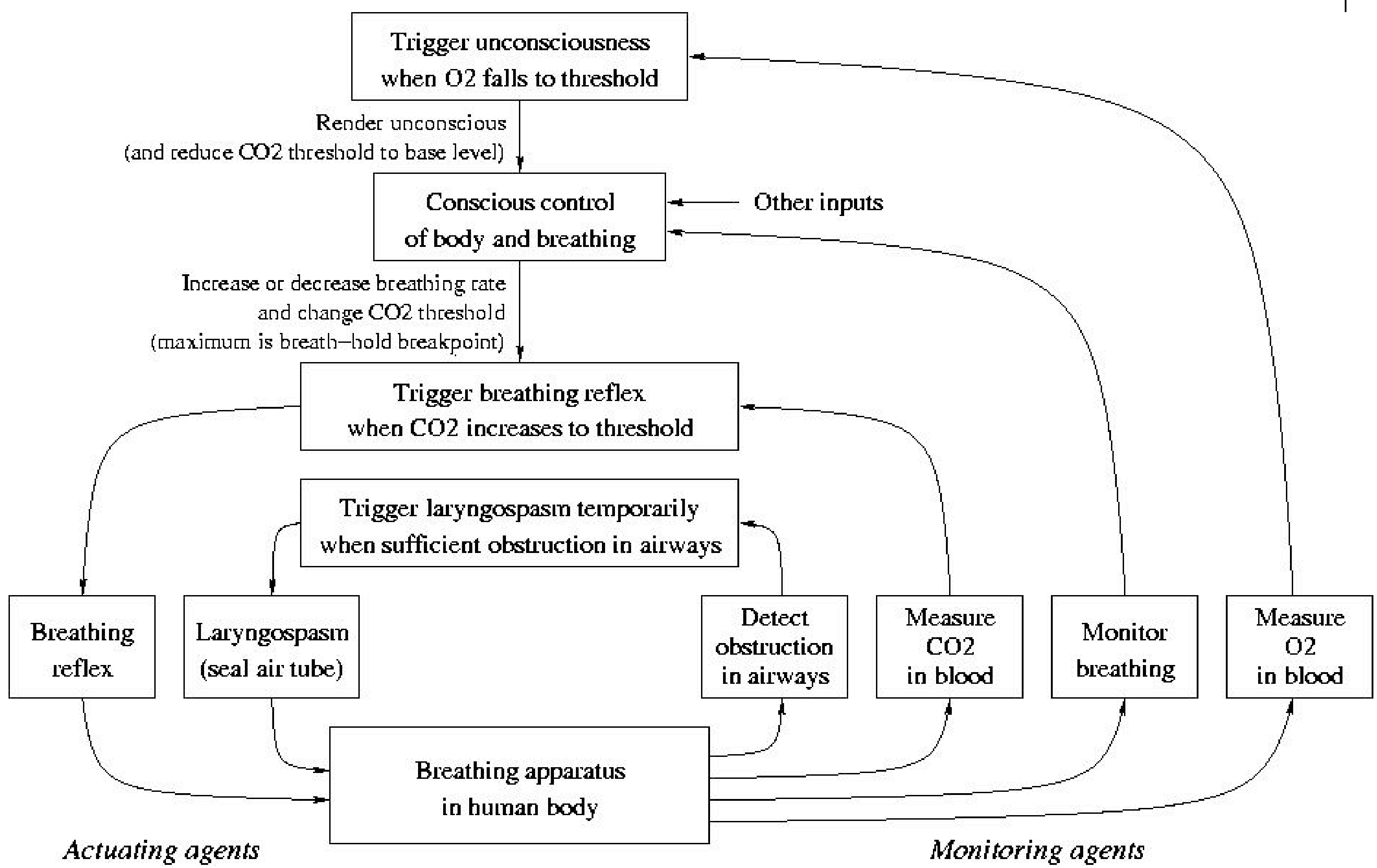
- This system is unstable!
- But each loop is stable in isolation
  - Combining stable loops can result in instability

Correct solution



- 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

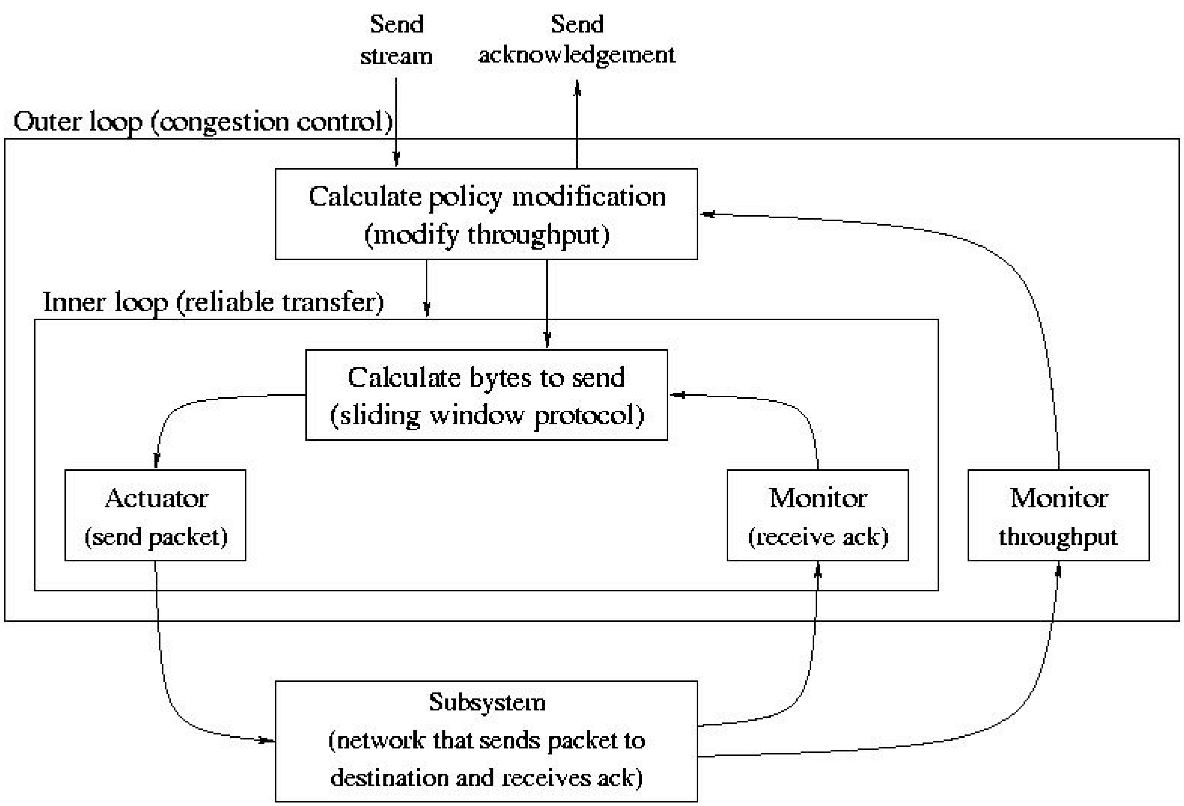
The human respiratory system



Discussion of respiratory system

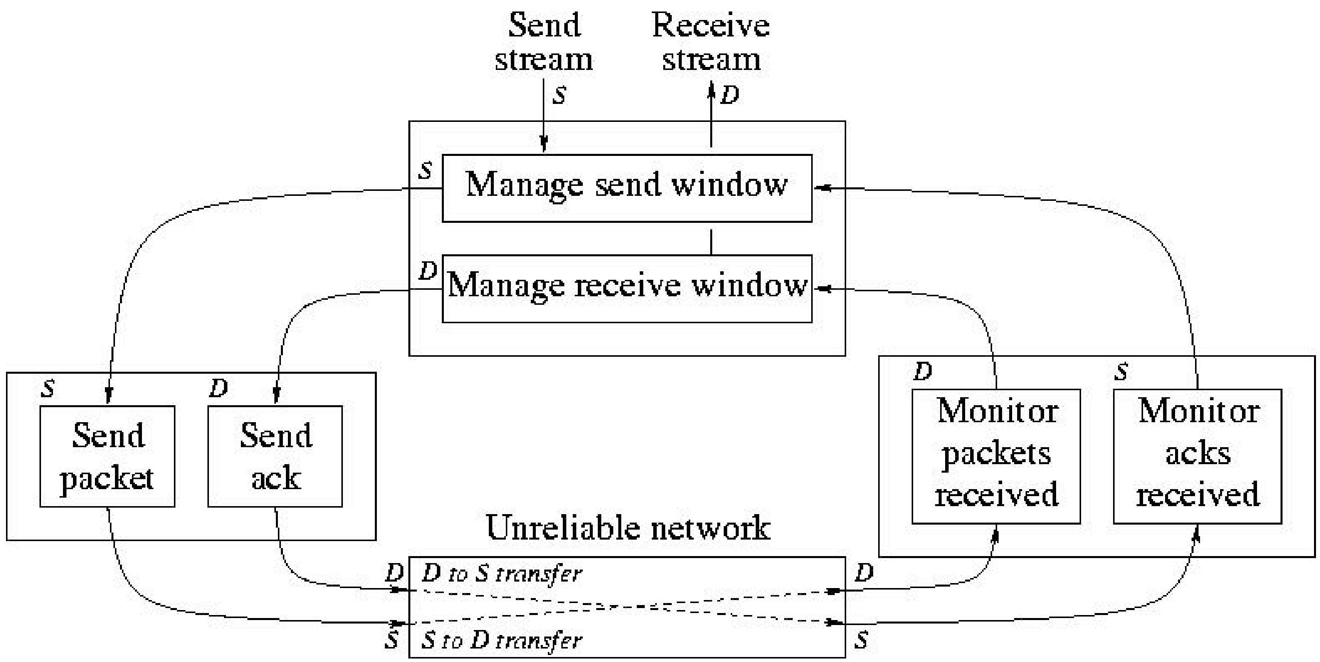
- Four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious)
  - This design is derived from a precise textual medical description [Wikipedia 2006: “Drowning”]
- 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

Program design with feedback loops



- The style of system design illustrated by the respiratory system can be applied to programming
- Programming then consists of building hierarchies of interacting feedback loops
- This example shows a reliable byte stream protocol with congestion control (a variant of TCP)
- The congestion control loop manages the reliable transfer loop

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.

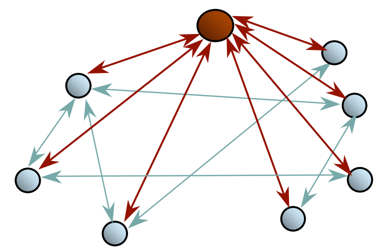
Feedback loops and distribution

- 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!
  - Let us take a closer look at structured overlay networks

THE ADVENTURES OF SELFMAN

Structured overlay networks: inspired by peer-to-peer

- Hybrid (client/server)
  - Napster

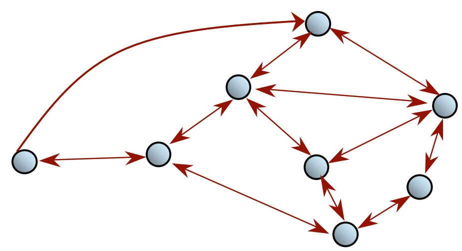


R = N-1 (hub)

R = 1 (others)

H = 1

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

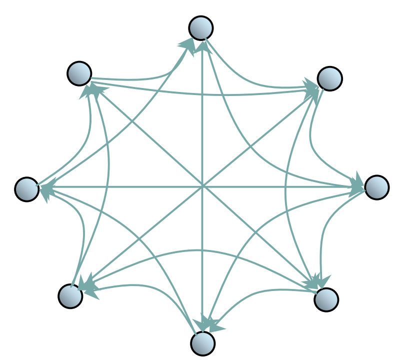


R = ? (variable)

H = 1…7

(but no guarantee)

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



R = log N

H = log N

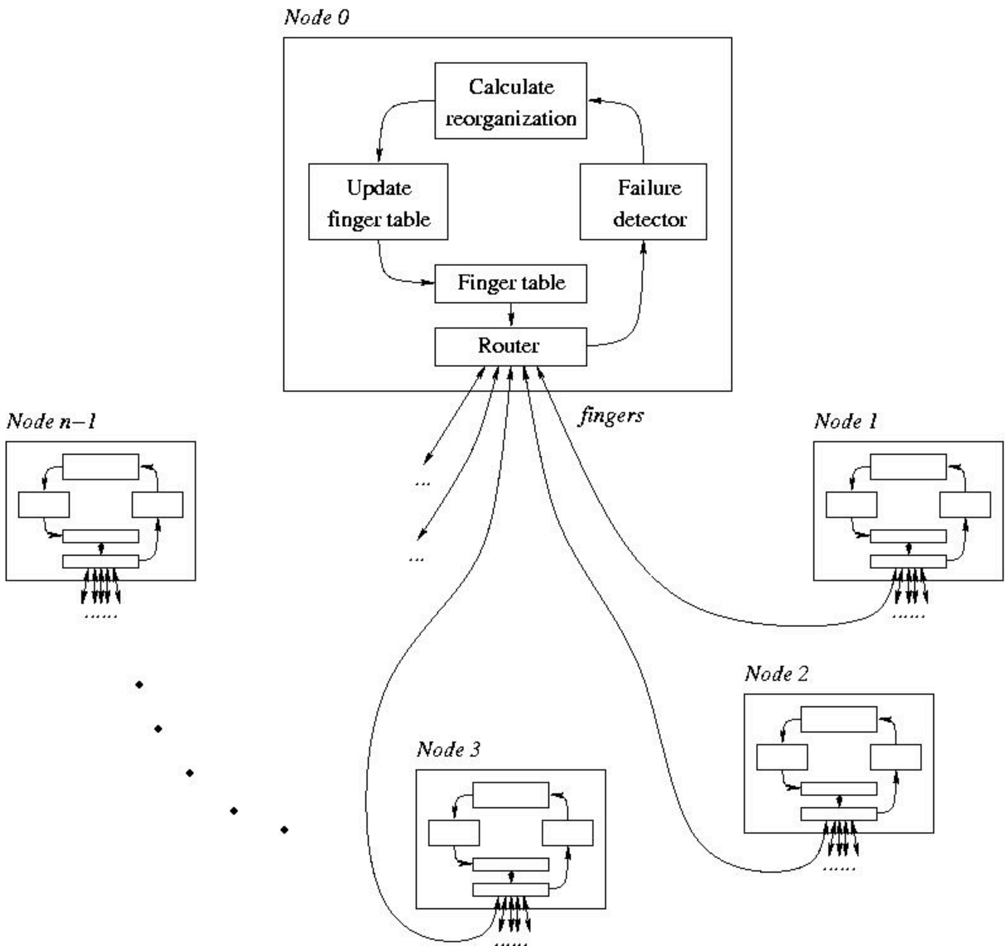
(with guarantee)

THE ADVENTURES OF SELFMAN

Properties of structured overlay networks

- Scalable
  - Works for any number of nodes
- Self organizing
  - Finger tables updated with node joins/leaves
  - Finger 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

Feedback loops in a structured overlay network



- 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]

European Commission

Information Society

Self organization

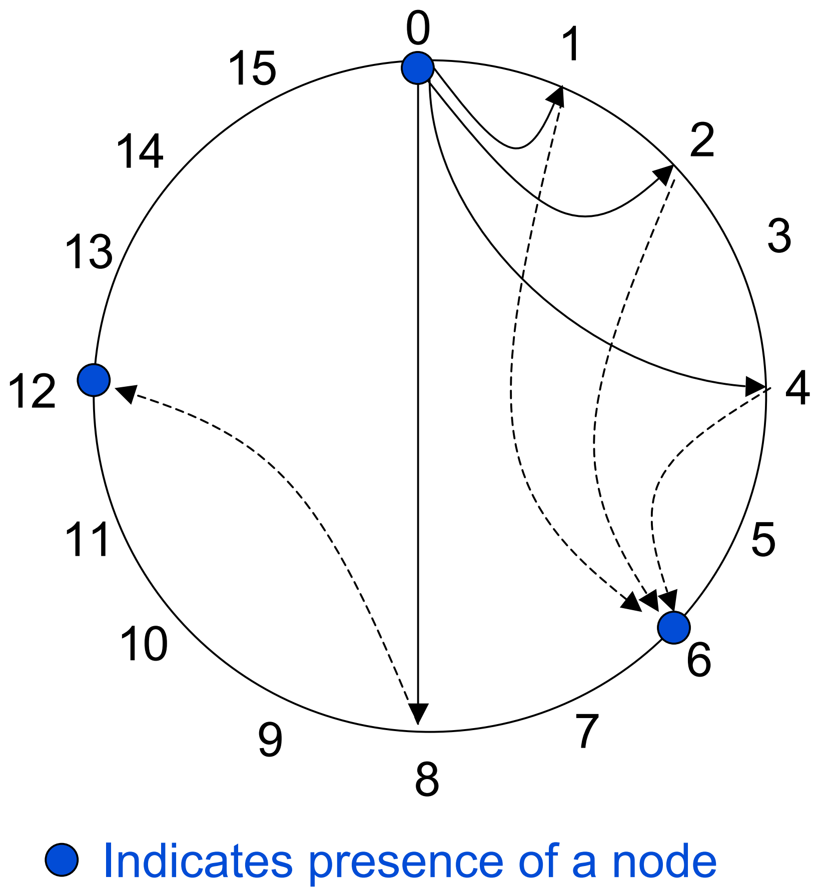
- Self-organizing the finger tables
  - Correction-on-use (lazy approach)
  - Periodic correction (eager approach)
  - Guided by assumptions on traffic
- Cost
  - Depends on structure
  - A typical algorithm, DKS (distributed k-ary search), achieves logarithmic cost for reconfiguration and for key resolution (lookup)
- Example of lookup for Chord, the first well-known structured overlay network

Lookup illustrated in Chord

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 virtual identifier 7

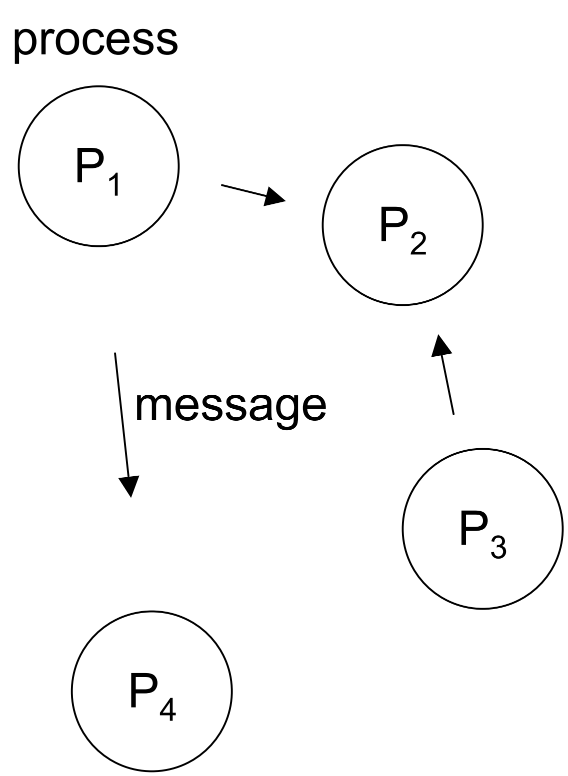
Interval	node to be contacted
[0,1)	0
[1,2)	6
[2,4)	6
[4,8)	6
[8,0)	12



Related work in self-managing systems

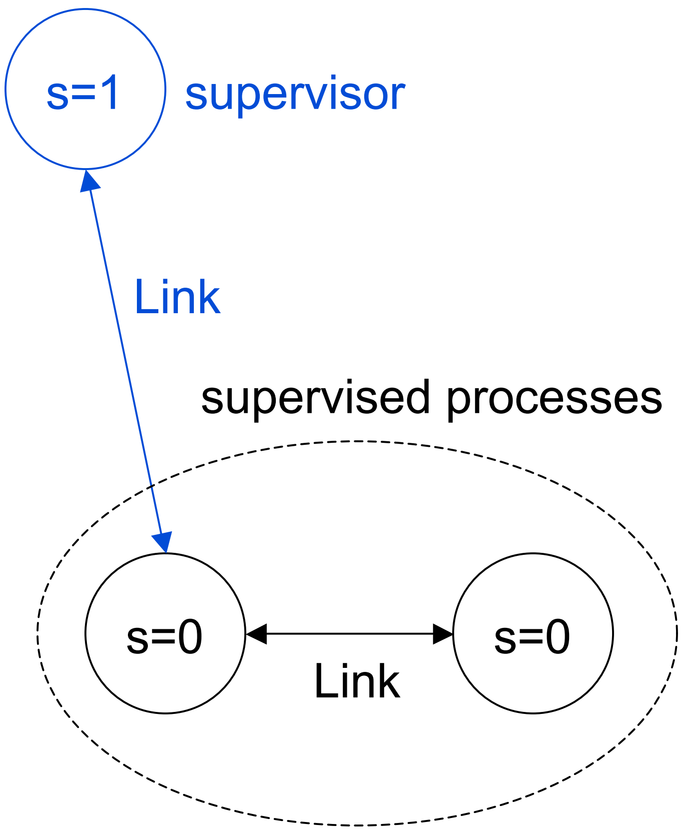
- Erlang fault-tolerance architecture [Armstrong 2003]
  - Erlang is designed explicitly to build applications that survive software faults
    - Hypothesis: Software faults are inevitable
  - The Erlang system has been used to build highly available products: AXD301 ATM switch, Bluetail Mail Robustifier, 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

Erlang



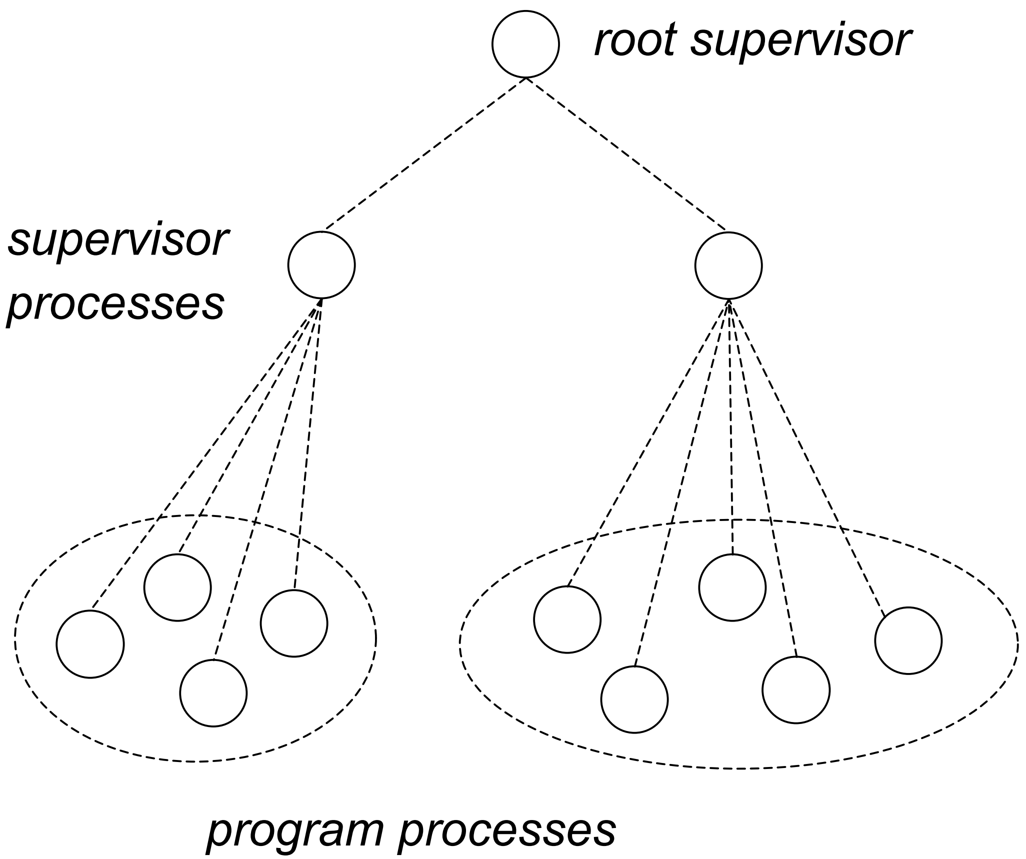
- Erlang is a language used to develop highly reliable software systems
- An Erlang program consists of a set of running “processes” (lightweight threads with independent address spaces) that send messages asynchronously
- Fault tolerance consists of three levels:
  - Primitive failure detection through 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)

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

Supervisor trees

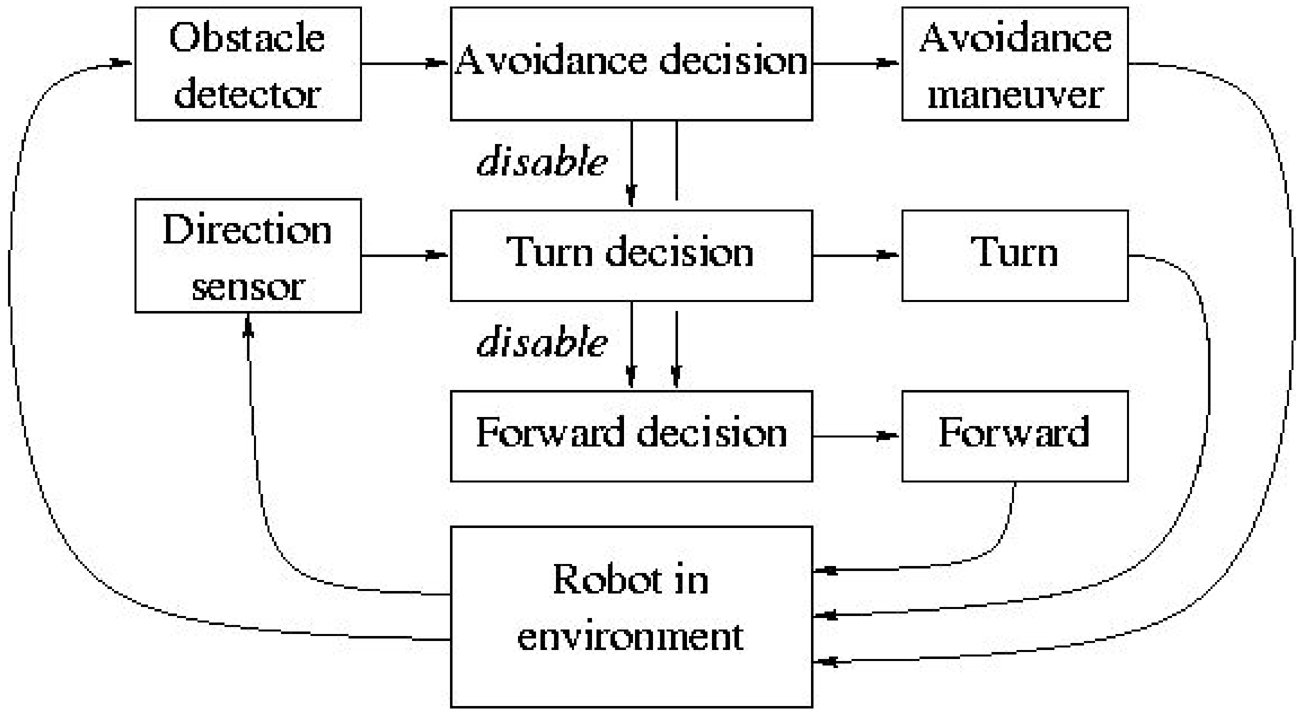


- The program consists of a large number of processes
- Program processes are organized in pools
  - Each pool is observed by a supervisor process linked to all of them
  - An AND supervisor stops and restarts all its children if one crashes
  - An OR supervisor restarts just the crashed child
- The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

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

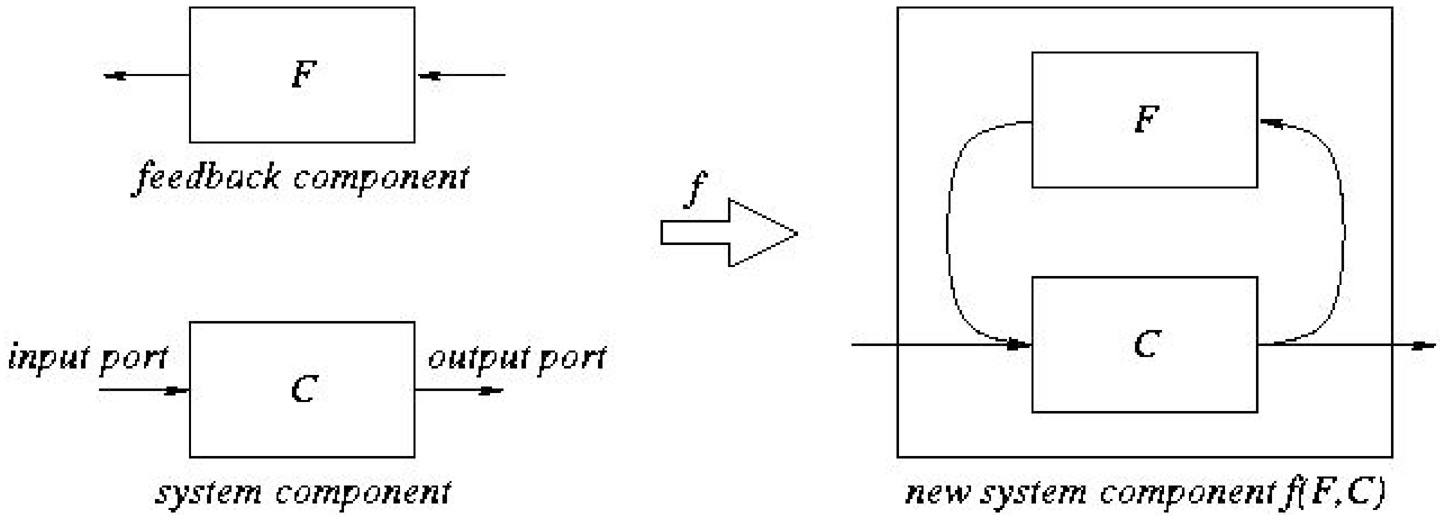
THE ADVENTURES OF SELFMAN

General architectural framework

- What can we deduce from these examples?
- 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

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 Mozart using higher-order functions, lightweight concurrency, and dataflow synchronization
  - Mozart Programming System: an advanced multiparadigm platform
- Component interface: one input port (accepts input events) and one output stream (produces ordered sequence of output events)
- Component behavior:
  - $\text{State} \times \text{Event} \rightarrow \text{State} \times \text{Event}^* \times (R^+, \text{Event})^*$
  - Given an input state and an input event, create an output state, new output events, and new time-delayed input events
  - Time delaying is important when interacting with the external world; it is not needed internally to a program
- Component combinators can be written in a few lines of code
- All the examples we have shown can be programmed easily

Where do we go from here?

- There is a research agenda to be set up!
  - Self management has a role to play in general software development, not just in autonomic computing
  - The overall architecture of a system must be designed using self-management principles
- The SELFMAN project, an EU 6FP project that started in June 2006, will make a first cut at using self management for general software
  - We will combine a structured overlay network (which is already self managing at a low level) with an advanced component model, to achieve a self-management architecture
  - We will build a self-managing three-tier application with a replicated transactional store as proof of concept
  - We will implement in ObjectWeb (industrial middleware) and Mozart (advanced research system)

THE ADVENTURES OF SELFMAN

Programming self-managing systems in Mozart

- Mozart has advanced distribution support
  - 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
    - Event-based interface to failure detection
    - Kill operation
    - Support for temporary failures (imperfect failure detection)
- The distribution support will be extended to support self management of distributed systems
- Redesign of Mozart's P2PS structured overlay network
  - Using concurrent components with event-based communication (Boris Mejias)
  - Support for programming with feedback loops
    - Language support (Yves Jaradin, Jean-Bernard Stefani)

Conclusions

- Self management is useful for all software design, not just for tasks done by a human manager
  - Self management can overcome the fragility of software
- Self-managing software systems consist of hierarchies of interacting feedback loops
  - Programming with feedback loops becomes common and should be supported by the language
  - All parts of the system (except a small kernel) should be inside a feedback loop (slogan: no open-ended code!)
  - It should be feasible to design for a desired global behavior
- We are realizing these ideas in the SELFMAN project, which started in June 2006
  - We are combining ideas from structured overlay networks and advanced component models
  - See http://www.ist-selfman.org

Month 12 deliverables (on Wiki Community Portal)

- **Structured overlay networks** (**Boris Mejias**)
  - D1.1: **Low-level self-management primitives for SON** (node failure / removal / addition, state monitoring, configuration, versioning, updating)
  - D1.3a (**Roland Yap**): **First report on security for SON** (threat model, security mechanisms, monitoring system)
- **Programming framework** (**Peter Van Roy**)
  - D2.1a: **Basic computation model** (components and architectural description language)
  - D2.2a: **Architectural framework specification**
  - D2.3a: **Formal operational semantics** (components and reflection)
- **Transaction model** (**Monika Moser**)
  D3.1a: **First report on formal models for transactions over SON** (resolve tension distributed system ↔ application)
- **User requirements** (**Thierry Coupaye**)
  D5.1: **User requirements for application servers** (from industrial experience)
  Next meeting in Grenoble on Nov. 20 and 21