

The Road to Distributed Programming: From Network Transparency to Structured Overlay Networks and Onward to Self Management

March 2006

Peter Van Roy

Université catholique de Louvain
Louvain-la-Neuve, Belgium



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

1

Solving the distributed programming problem



- How can we *really* simplify distributed programming, to find a definitive solution?
- This talk gives a personal view of this problem, based on research done since 1995
- This talk takes the broad view to solve the problem: in particular, language design and distributed algorithms are at the core
- I give an overview of our work and insights and show that much remains to be done

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

2

Major steps along the way



- Client/server techniques
- Network transparency and network awareness
 - Given proper language design and distributed algorithms to support it, network transparency becomes practical (despite the naysayers)
- Abstractions for distributed programming
 - For example, transactions as a way to both overcome network delays and machine crashes
 - For example, asynchronous communication between objects as the default (instead of shared-state concurrency such as monitors)
- Decentralized computing
 - Structured overlay networks that provide communication and storage
 - This is an outgrowth of peer-to-peer, adding guarantees and efficiency
- Self management
 - “Abnormal” behavior is normal; the system must continue to work
 - Self management uses feedback loops that pervade the system
 - Combining structured overlay networks with a component model

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

3

Programming language



- The right programming language is critical for success
- It's not a question of details such as whether Java, C#, Python, or Ruby are better (in fact, they are all unsatisfactory)
- It's more fundamental: the right language needs a layered structure with three or four layers:
 - Declarative (functional) language at the core
 - Declarative (deterministic) concurrency: no race conditions!
 - Message-passing concurrency (active objects)
 - Shared-state concurrency (using transactions, not monitors)
- Why is this the right structure? There are many reasons!
- With a language designed in this way, many things become simple (including distributed programming)
- Does such a language exist?
 - Partly: the closest approximations are Oz, Alice, E, and Erlang

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

4

Network transparency



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

5

Basic principles



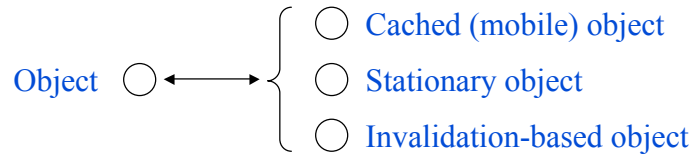
- *Refine* language semantics with a distributed semantics
 - Separates **functionality** from **distribution structure** (network behavior, resource localization)
- Three properties are crucial:
 - **Transparency**
 - Language semantics **identical** independent of distributed setting
 - Controversial, but let's see how far we can push it, *if* we can also think about language issues
 - **Awareness**
 - Well-defined distribution behavior for each language entity: simple and predictable
 - **Control**
 - Choose different distribution behaviors for each language entity
 - Example: objects can be stationary, cached (mobile), asynchronous, or invalidation-based, with same language semantics

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

6

Adding distribution



- Each language entity is implemented with one or more distributed algorithms. The choice of distributed algorithm allows **tuning of network performance**.
- Simple programmer interface: there is just **one basic operation**, passing a language reference from one process (called “**site**”) to another. This conceptually causes the processes to form one large store.
- How do we pass a language reference? We provide an **ASCII representation of language references**, which allows passing references through any medium that accepts ASCII (Web, email, files, phone conversations, ...)
- How do we do fault tolerance? By **reflecting** the faults in the language.

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

7

Example: sharing an object (1)



```
class Coder
  attr seed
  meth init(S) seed:=S end
  meth get(X)
    X=@seed
    seed:=(@seed*23+49) mod 1001
  end
end
```

```
% Create a new object C
C={New Coder init(100)}
```

```
% Create a ticket for C
T={Connection.offer C}
```

- Define a simple random number class, Coder
- Create one **instance**, C
- Create a **ticket** for the instance, T
- The ticket is an **ASCII representation of the object reference**

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

8

Example: sharing an object (2)



```
% Use T to get a reference to C
C2={Connection.take T}
```

```
local X in
  % invoke the object
  {C2 get(X)}
  % Do calculation with X
  ...
end
```

- Let us use the object C on a second site
- The second site gets the value of the ticket T (through the Web or a file, etc.)
- We convert T back to an object reference, C2
- C2 and C are references to the same object

What distributed algorithm is used to implement the object?

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

9

Example: sharing an object (3)



- C and C2 are the **same object**: there is a distributed algorithm guaranteeing coherence
- Many distributed algorithms are possible, as long as the language semantics are respected
- By default, we use a **cached object**: the object state moves synchronously to the invoking site. This makes the semantics easy, since all object execution is local (e.g., exceptions raised in local threads). A cached object is a kind of mobile object.
- Other possibilities are a **stationary object** (behaves like a server, similar to RMI), an **invalidation-based object**, etc.

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

10

Example: sharing an object (4)



- **Cached objects:**
 - The object state is mobile; to be precise, the *right to update the object state* is mobile, moving atomically to the invoking site
 - The object class is stateless (a record with method definitions, which are procedures); it therefore has its own distributed algorithm: it is copied once to each process referencing the object
 - The protocol for cached objects is simple: maximum of three messages for each atomic move. Faults (e.g., crashes) are reflected in a simple way.

Asynchronous objects (1)



- Cached objects still have the synchronous behavior of centralized objects (they keep the same semantics)
 - This means that a *round trip delay* is needed upon the first invocation
- In a distributed system, asynchronous communication is more natural
 - To achieve it, we use **dataflow variables**: single-assignment variables that can be in one of two states, **unbound** (the initial state) or **bound**
- The use of a dataflow variable is transparent: it can be used **as if it were the value!**
 - If the value is not yet available when it is needed, then the thread that needs it will simply suspend until the value arrives
 - Example:

```
thread X=100 end          Y=X+100
(binds X)                (uses X)
```

- A **distributed protocol** is used to implement this behavior

Asynchronous objects (2)



- Used just like normal objects
- Return values are passed with dataflow variables:

C={NewAsync Coder init(100)}
(create on site 1)

{C get(X1)}
{C get(X2)}
{C get(X3)}
X=X1+X2+X3
(call from site 2)

- Can synchronize on error
 - Exception raised by object: {C get(X1) E}
(synchronize on E)
 - Error due to system fault (crash or network problem):
 - Attempt to use return variable (X1 or E) will signal error (lazy detection)
 - Eager detection also possible

March 2006

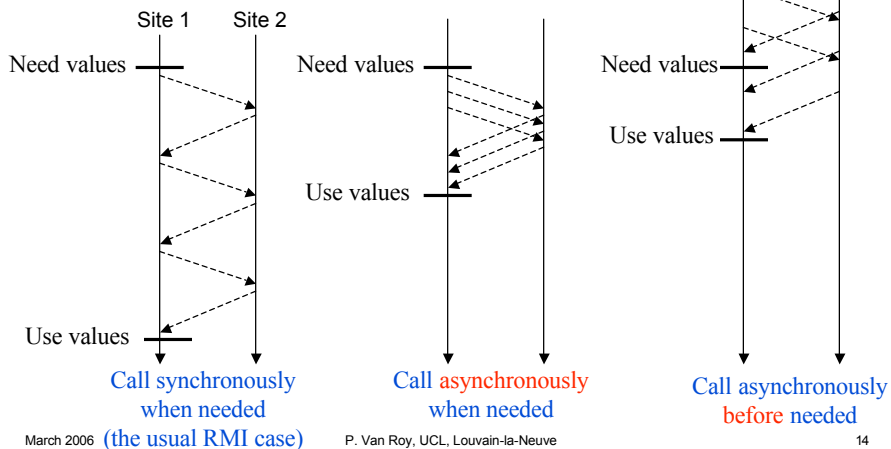
P. Van Roy, UCL, Louvain-la-Neuve

13

Asynchronous objects (3)



Improved network performance without changing the program!



Fault tolerance



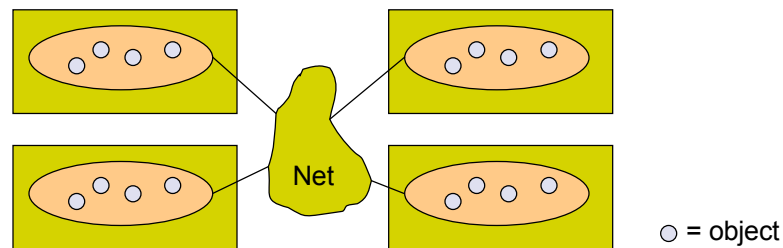
- **Reflective failure detection**
 - Reflected into the language, at level of single language entities
 - Two kinds: **permanent process failure** and **temporary network failure**
 - Both synchronous and asynchronous detection
 - Synchronous: exception when attempting language operation
 - Asynchronous: language operation blocks; user-defined operation started in new thread
 - Our experience: **asynchronous is better** for building abstractions
- Building fault-tolerant abstractions
 - Using reflective failure detection we can build abstractions in Oz
 - Example: **transactional store**
 - Set of objects, replicated and accessed by transactions
 - Provides both fault tolerance and network delay compensation
 - Lightweight: no persistence, no dependence on file system

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

15

Transactional store with the GlobalStore abstraction



- The GlobalStore is looks like a set of objects with a transactional interface
 - The application affects the GlobalStore through transactions
 - The GlobalStore affects the application through notifications
- Objects are replicated, giving both fault tolerance and latency tolerance
 - Current implementation handles only machine failure

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

16

Conclusions for network transparency



- Network transparency is a practical way to program open, robust distributed applications with a small number of computing nodes
- Everything is implemented in the Mozart Programming System version 1.3.1
 - Including stationary, cached mobile, and asynchronous object
 - Including dataflow variables with distributed rational tree unification
 - Including distributed garbage collection with weighted reference counting and time-lease
 - Transactional object store was implemented but is no longer supported – it will be superseded by peer-to-peer (see later in the talk)
- Current work
 - General distribution subsystem (DSS) that generalizes and factorizes out the distribution support from the emulator
 - Extensions for a structured overlay network (P2PS library) and a service architecture (P2PKit) are in the works (see later in the talk)

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

17

Decentralized (peer-to-peer) computing



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

18

Peer-to-peer systems (1)



- Network transparency works well for a small number of nodes; what do we do when the number of nodes becomes large?
 - This is what is happening now
- We need a **scalable way to handle large numbers of nodes**
- Peer-to-peer systems provide one solution
 - A distributed system that connects resources located at the edges of the Internet
 - Resources: storage, computation power, information, etc.
 - Peer software: all nodes are functionally equivalent
- Dynamic
 - Peers join and leave frequently
 - Failures are unavoidable

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

19

Peer-to-peer systems (2)



- Unstructured systems
 - Napster (first generation): still had centralized directory
 - Gnutella, Kazaa, ... (second generation): neighbor graph, fully decentralized but no guarantees, often uses superpeer structure
- **Structured overlay networks** (third generation)
 - Using non-random topologies
 - Strong guarantees on routing and message delivery
 - Testing on realistically harsh environments (e.g., PlanetLab)
 - DHT (Distributed Hash Table) provides lookup functionality
 - Many examples: Chord, CAN, Pastry, Tapestry, P-Grid, DKS, Viceroy, Tango, Koorde, etc.

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

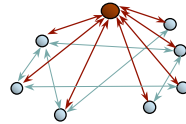
20

Examples of P2P networks



- Hybrid (client/server)

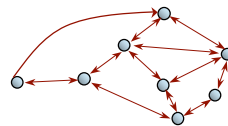
- Napster



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

- Unstructured P2P

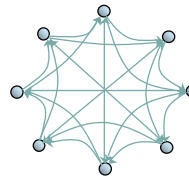
- Gnutella



$R = ?$ (variable)
 $H = 1 \dots 7$
 (but no guarantee)

- Structured P2P

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



$R = \log N$
 $H = \log N$
 (with guarantee)

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

21

Properties of structured overlay networks



- Scalable
 - Works for any number of nodes
- Self organizing
 - Routing tables updated with node joins/leaves
 - Routing tables updated with node failures
- Provides guarantees
 - If operated inside of failure model, then communication is guaranteed with an upper bound on number of hops
 - Broadcast can be done with a minimum 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

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

22



Self organization

- Maintaining the routing 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

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

23

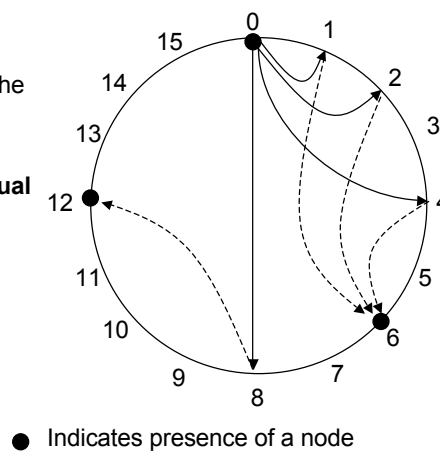


Chord: lookup illustrated

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



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

24

Peer-to-peer applications



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

25

What is P2P good for?



- P2P is the **basic structure of the Internet**
 - TCP/IP is a peer-to-peer protocol
- P2P concept was forgotten for a long time during the reign of TCP/IP
 - Client/server (on top of TCP/IP) was king
- Now P2P has come **back in the limelight** because of **file-sharing applications**
 - Napster, Gnutella, Kazaa, etc., etc.,
- But P2P is useful for much more than this!

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

26

One simple use



- People often talk about P2P and scalability in one breath
- But one basic use of P2P has nothing to do with scalability
 - P2P applications don't need a server
 - Two computers (e.g., PDAs) can connect without having to go through a server
- This use is so compelling that one company started a project with us just because of it
 - They do mobile applications on PDAs and their middleware needed a server!
- But there is much more

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

27

Basic P2P services



- Structured overlay networks provide two classes of services
 - Distributed directory or DHT
 - Name-based communication
- Let's see what we can build with these to start with

March 2006

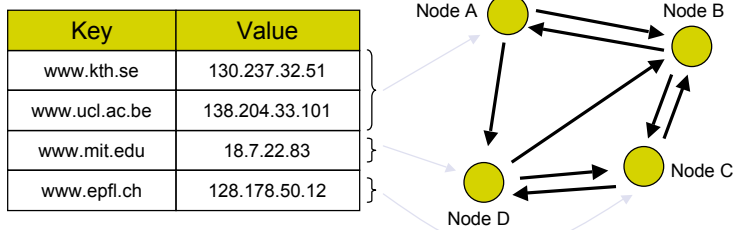
P. Van Roy, UCL, Louvain-la-Neuve

28

Distributed directory



- A service that enables the distribution of an ordinary directory (or any other table) onto a set of cooperating nodes
- Each node stores only a fraction of the data
- Provides a basic lookup service; if data is not found then lookups are routed to other nodes



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

29

Name-based communication (1)



- Standard Internet communication (TCP/IP, UDP) is not suited for 21st century computing
 - Firewalls, NATs, changing IP addresses
- P2P can overcome these complications
- How?
 - Use the directory
 - Map between names and location
 - Bypass firewalls and NATs by routing through neighbors

March 2006

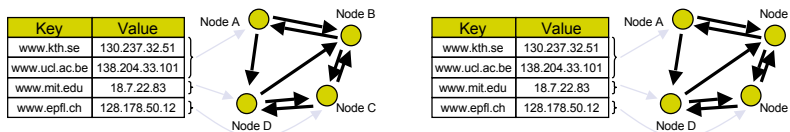
P. Van Roy, UCL, Louvain-la-Neuve

30

Name-based communication (2)



- What about group communication?
- Use the overlay to broadcast to all nodes
 - Create multiple groups, broadcast within each
- P2P functionality is a primitive that can be instantiated as often as needed



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

31

Some applications (1)



- Distributed authorization
 - Swedish defense project
 - Store certificates in the directory
 - No central server; survives even if nodes are attacked
 - Similar to original goal for Arpanet
- Global backup
 - Setup phase: clients install backup tool, decide on amount of space to share, choose files for backup
 - Regular operation: data is encrypted and stored in the directory

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

32

Some applications (2)



- Global file system
 - Similar to AFS and NFS
 - Files stored in the directory
 - What is new here?
 - File system logic is self-managed
 - Add/remove servers on the fly
 - Automatically handles failures
 - Automatically load-balances
 - No manual configuration needed

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

33

Some applications (3)



- P2P cache
 - A distributed cache
 - Every node in an organization runs a client
 - Want to browse a Web page?
 - If it exists in the directory: download it from a peer
 - Otherwise, fetch it and cache it
 - No central proxy needed
- P2P Web server
 - Distributed Web server; pages stored in directory
 - What is new here?
 - Again, Web server logic is self-managed
 - Automatically load-balances
 - Add/remove servers on the fly
 - Automatically handles failures

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

34

Conclusions for structured overlay networks



- Structured overlay networks are a basic building block for building distributed applications on large numbers of nodes
- As such, they complement the client/server and network-transparent systems
- But they only provide basic services
 - We need to build on top of them!

Self management





The need

- Because of the growth of the Internet, individual computers are so numerous that they tend to vanish
 - Programs that run on a single computer will become the exception
 - Programs will be spread over many computers
- This brings enormous complexity problems
 - Network transparency solves a few: application code remains simple despite the distribution
 - Structured overlay networks solve a few more: self-organizing communications and storage infrastructure
 - But the many services and applications, all built on top, introduce much more complexity
 - A solution: self management
- We need a general architecture for building self-managing systems
 - A key part of this architecture is a powerful component model

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

37



Self management

- The system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies
 - Human intervention is lifted to the level of the policies
- Typical self-management operations include: add/remove nodes, tune performance, auto-configure, failure detection & recovery, intrusion detection & recovery, software rejuvenation
- **Self management exists at all levels**
 - Such as: application level, service levels, cluster level, process/OS level
- For large-scale systems, environmental changes that require recovery by the system become normal and even frequent events
 - **“Abnormal” events are normal occurrences** (e.g., failure is a normal occurrence)

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

38

Axes of self management

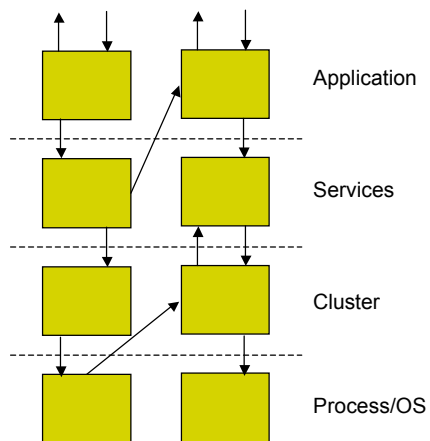


- **Self configuration**: adding or removing nodes during execution, version updating during execution, lifecycle maintenance
- **Self healing (“fault tolerance”)**: according to failure model (node failure, network failure)
- **Self tuning**: load balancing and overload management
- **Self protection (“security”)**: according to threat model (simple model: nodes themselves are trustworthy)

Different layers, different requirements



- **Application level**
 - 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



Mechanisms for self management



- Self management adds **feedback loops** throughout the system
 - Detection of anomaly → calculation of a correction → application of the correction
- In non-self-managing systems, the feedback loops often go through human beings
 - In self-managing systems, the human is no longer inside the loop but manages the loop from the outside
 - **The human manages the policy, the system implements the mechanism**
- Problems of **global behavior**: convergence, oscillation, chaos, divergence
 - Techniques from cybernetics and general system theory
 - Software systems are usually **non-linear** and **non-monotonic**. Is a linear or monotonic approximation possible? Can the system be forced into a linear or monotonic mode?

March 2006

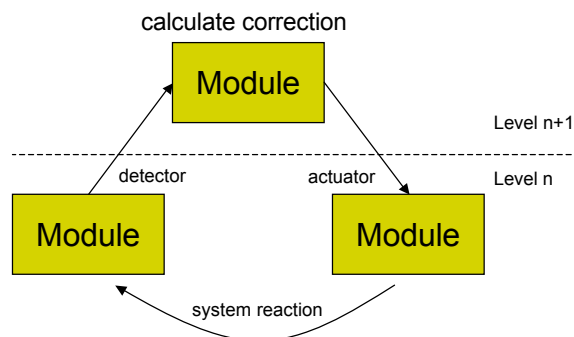
P. Van Roy, UCL, Louvain-la-Neuve

41

Simple feedback loop



- Self management through **feedback loops**
- Program modules need **detectors** (introspection) and **actuators** (control)
- Whole modules might be devoted to being detectors or actuators
 - For example, failure detection, which can be complicated to calculate (heartbeat detection, belief propagation)
- Feedback loops can interact in two ways:
 - two loops that affect interdependent system parameters (**stigmergy**)
 - one loop that directly controls another loop (**management**)



March 2006

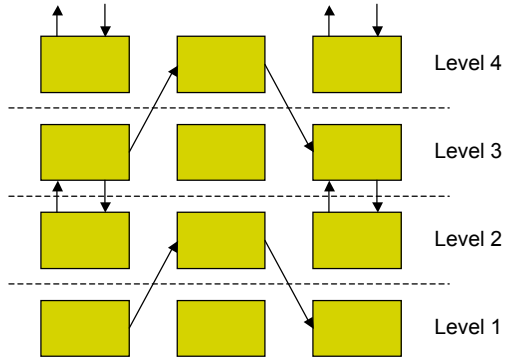
P. Van Roy, UCL, Louvain-la-Neuve

42

Complexity of interacting feedback loops



- Feedback loops exist throughout the system
- 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
 - Interaction between nested feedback loops (cf. Norbert Wiener's classic example of a **fire in an airconditioned hotel**)
- What are the **rules of good feedback design**?
 - Analogous to structured and object-oriented programming
 - **We need to understand how to build general self-managing systems**

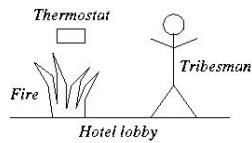


March 2006

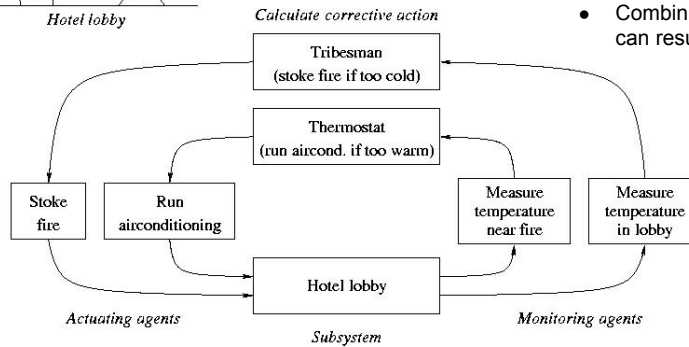
P. Van Roy, UCL, Louvain-la-Neuve

43

Example of stigmergy (Wiener)



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

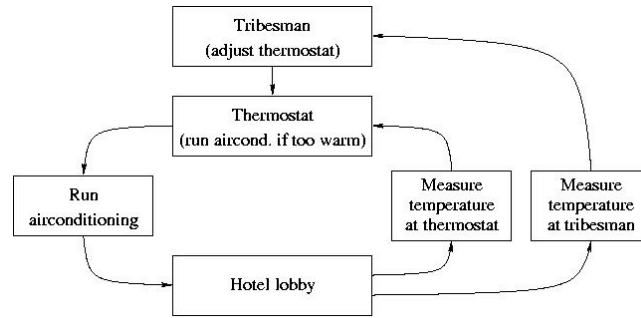


March 2006

P. Van Roy, UCL, Louvain-la-Neuve

44

Correct solution



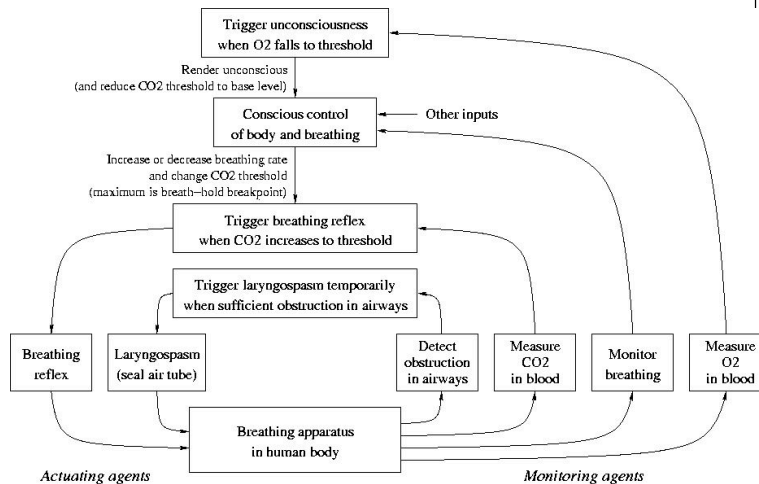
- Instead of stoking a fire, the tribesman simply adjusts the thermostat. The resulting system is stable.
- This uses management instead of stigmergy
- Design rule: **use the system, don't try to bypass it**

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

45

The human respiratory system



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

46

Discussion of respiratory system



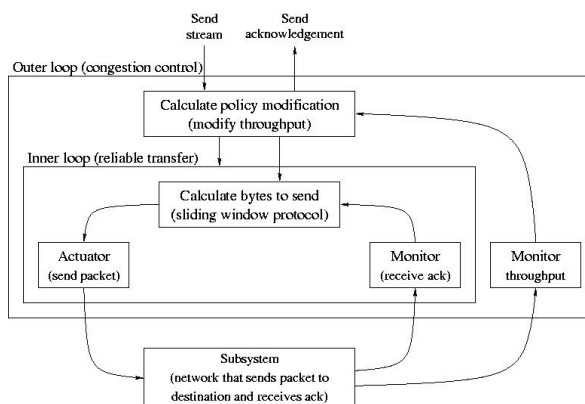
- **Four feedback loops:** two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling unconscious)
 - This design is derived from a precise textual medical description [Wikipedia 2006: "Drowning"]
- Holding your breath can have two effects
 - Breath-hold threshold is reached first and breathing reflex happens
 - O₂ 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** (fail safe)
 - Conscious control is a powerful problem solver but it needs to be held in check

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

47

Program design with feedback loops



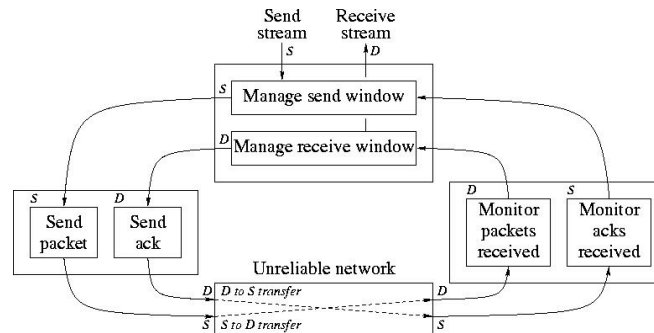
- The style of system design illustrated by the respiratory system can be applied to programming
- Programming then consists of building hierarchies of interacting feedback loops
- This example shows a reliable byte stream protocol with congestion control (a variant of TCP)
 - The source node of the protocol is shown
- The congestion control loop manages the reliable transfer loop
 - By changing the sliding window's buffer size

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

48

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.

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

49

Feedback loops and distribution



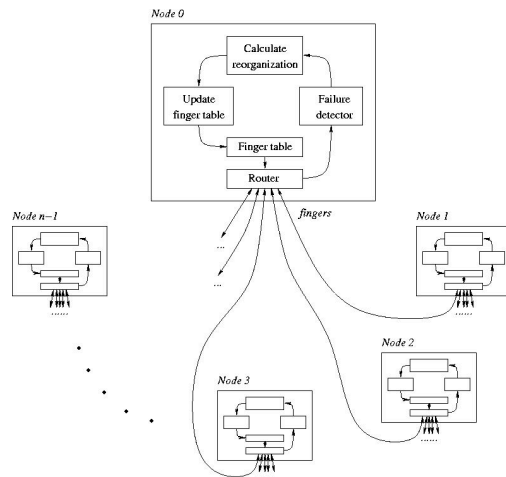
- The interaction between feedback loops and distribution is not well understood
- Distributed algorithmics has studied special cases of this interaction
 - Fault tolerance
 - Self-stabilizing systems
 - Structured overlay networks
- Feedback loops are useful for much more than fault tolerance!
 - Let us take a closer look at **structured overlay networks**

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

50

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 (ring management)
 - Need ability to work with strongly complete failure detection
 - First proved in [Ghodsli 2006]

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

51

Architecture for self-managing systems



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

52

Architecture for self-managing systems



- What is a possible architecture for building large-scale distributed systems that are self managing?
 - Requirements: support detectors and actuators at all levels, support reconfiguration and feedback loops, support communication and fault tolerance
- **Proposed solution: Combine structured overlay network (SON) with an advanced component model**
 - The structured overlay network is built with a component model
 - Additional services and the application are installed on top of the structured overlay network
 - The component model supports monitoring and reconfiguration
- Let us examine both structured overlay networks and component models to see what they provide

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

53

Structured overlay networks



- Related to peer-to-peer systems (Gnutella, Kazaa, Freenet, etc.)
- **Mature research:** scalable and **efficient**, provides **guarantees**, provides **basic services** (name-based communication, DHT)
- Already provides low-level self management properties (self organizing with node leave/join and failures, replicated storage)
 - But does **not** provide any lifecycle operations (dynamic reconfiguration, versioning, updating)
- **Proposed solution:** Reformulate SON architecture in terms of components to get lifecycle operations and self-management hooks (detectors, actuators), and as a basis of an architecture providing high-level self management

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

54

Component models



- Components are the building blocks for configuration and lifecycle management of large systems
- Industry standard component models (EJB, CCM) **are not suitable**
 - They are limited to client/server middleware, not scalable, tight coupling instead of loose coupling (e.g., for clusters)
 - They are not compositional, do not have sharing, and support only a limited set of non-functional properties
- Research models **overcome these limitations**
 - **Fractal model**: introspection and reconfiguration abilities, general control of contents and links, both shared and compound components
 - **Mozart model**: components as first-class language entities, stateless, with unique identities, can be calculated at run-time
- **Proposed solution**: Install components on top of structured overlay network: build decentralized distributed applications, use SON communication and storage abilities

SELFMAN project

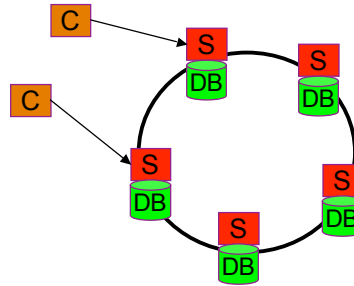


- We are starting a new European project to explore self management in large-scale systems according to this vision
- SELFMAN will combine structured overlay networks with an advanced component model
- SELFMAN will build an application-hosting architecture that provides self management, and build a self-managing multi-tier application as a proof of concept

Self-managing three-tier architecture



- A key application scenario for SELFMAN
- Three-tier: client/server/database
 - **Clients** connect to nearest server node
 - **Server** provides business logic
 - **Database** provides stable storage and querying
- We will take a standard (client/server) three-tier application and make it scalable and self managing



March 2006

P. Van Roy, UCL, Louvain-la-Neuve

57

General conclusions



- Distributed programming is becoming ubiquitous
- The first step is network transparency, but this is only good enough for small systems
 - Mozart Programming System version 1.3.1
- The next step is decentralized computing
 - Structured overlay networks
 - Basic properties such as scalability, efficiency, guarantees
 - Basic functionality: communications and storage
 - P2PS and P2PKit libraries in Mozart
- To build applications on top of this, we need to support self management
 - Otherwise, the complexities require too much human intervention
- We have started a European project, SELFMAN, to explore self management of large-scale distributed systems

March 2006

P. Van Roy, UCL, Louvain-la-Neuve

58