

Lecture Notes

# Distributed System

Hinnerk van Bruinehsen  
Tobias Höppner  
Tobias Famulla

SoSe 2013

# Contents

<b>1</b>	<b>Verteilte Systeme/Distributed Systems</b>	<b>3</b>
1.1	Orga . . . . .	3
1.1.1	Elektisches . . . . .	3
1.1.2	Übungen . . . . .	3
1.1.3	Material/Inhalt . . . . .	3
<b>2</b>	<b>Distributed Systems</b>	<b>5</b>
2.1	Transparency . . . . .	5
<b>3</b>	<b>Architectures of distributed Systems</b>	<b>7</b>
3.1	System architectures . . . . .	8
<b>4</b>	<b>PeerSim</b>	<b>11</b>
<b>5</b>	<b>Processes</b>	<b>12</b>
5.1	Virtualisation . . . . .	13
5.2	Client-/Serverprocesses . . . . .	14
5.3	Code Migration . . . . .	16
<b>6</b>	<b>Communication</b>	<b>18</b>
6.1	RPC - Remote Procedure Call . . . . .	18
6.2	Asynchronous RPC . . . . .	19
6.3	Message oriented communication . . . . .	20
6.3.1	Message-Oriented Transient Communication . . . . .	20
6.3.2	Message-Oriented Persistent Communication . . . . .	21
6.4	stream oriented communication . . . . .	22
6.5	Multicast communication . . . . .	22
6.6	Gossip-based-communication . . . . .	23
<b>7</b>	<b>Naming</b>	<b>25</b>
7.1	Flat naming . . . . .	25
7.1.1	Distributed Hash Tables . . . . .	25
<b>8</b>	<b>Synchronisation</b>	<b>27</b>
8.1	Clock synchronisation algorithms . . . . .	27
8.2	Network Time Protocol (NTP) . . . . .	28
8.3	Berkeley algorithm . . . . .	28
8.4	Lamports Logical Clocks . . . . .	28

8.5	Vector Clocks . . . . .	30
<b>9</b>	<b>Mutual Exclusion . . . . .</b>	<b>32</b>
9.1	Centralized algorithm . . . . .	32
9.2	Decentralized algorithm . . . . .	33
9.3	A distributed algorithm . . . . .	33
9.4	Token Ring Algorithm . . . . .	34
9.5	Centralized algorithm . . . . .	34
9.6	Comparison . . . . .	34
<b>10</b>	<b>Leader Election algorithms . . . . .</b>	<b>35</b>
10.1	leader election in a synchronnous ring . . . . .	35
10.1.1	LCR algorithm . . . . .	35
10.1.2	Algorithm of Hirschberg and Sinclair (HS-Alg) . . . . .	36
10.1.3	Time slice algorithm . . . . .	37
10.1.4	Variable speeds algorithm . . . . .	37
10.2	Leader election in a wireless environment . . . . .	37
10.3	The Bully Algorithm(flooding) (Garcia-Mdina, 1982) . . . . .	38
<b>11</b>	<b>Consistency and Consensus . . . . .</b>	<b>39</b>
11.1	Reliable multicast protocols . . . . .	39
11.1.1	Distributed Commit . . . . .	39

# Chapter 1

## Verteilte Systeme/Distributed Systems

### 1.1 Orga

VL Di 10-12 (nicht am 23.04.)  
Ue Do 10-12

#### 1.1.1 Elektisches

- (kvv)
- Website AG
- Sakai

#### 1.1.2 Übungen

- ca. 5 Übungsblätter, 14-tägig
- Vorträge in Gruppen über „verteilte Systeme“

#### 1.1.3 Material/Inhalt

1. Hälfte Distributed Systems (Tanenbaum, van Steen)
  - Architektur
  - Prozesse
  - Kommunikation
  - Namen
  - Synchronisation
  - Konsistenz
  - Replikation
  - Fehlertoleranz
2. Hälfte Distributed Algorithms (Nancy Lynch)
  - synchronous network algorithms

- network models (leader election, shortest path, distributed consensus, byzantine agreement)
- asynchronous network algorithms (shared memory, mutual exclusion, resource allocation, consensus)
- timing
- network resource allocation
- failure detectors

## Chapter 2

# Distributed Systems

**Def:** A distributed System is a collection of independent computers that appears to it's users as a single coherent system.

Characteristics:

- autonomous components
- appears as single system
- communication is hidden
- organisation is hidden  
(could be high-performance mainframe or sensor net)
- heterogenous system offers homogenous look/interface

Objectives:

- provide resources (printer, storage, computing)
    - share in a controlled, efficient way
    - grant access
- ⇒ connect users and resources

## 2.1 Transparency

hide the fact that processes and resources are physically distributed.

Types of transparency:

**access** hide differences in data representation and how a resource is accessed

**location** hide where a resource is located

**migration** hide that a resource may move to another location

**relocation** hide that a resource may be moved to another location while in use

**replication** hide that a resource is replicated

**concurrency** hide that a resource may be shared by serveral competitive users

**failure** hide the failure and recovery of a resource

transparency is desireable, but not always perfectly possible

tradeoff between transparency and complexity, maintainablility and performance

**Open System**

- service interfaces specified using Interface Definition Language (IDL)

- service specification as text

**Scalability** is an important property

- scalable in size (number of nodes)
- scalable in geographic spread
- scalable in administration

#### Problems

- centralized services
- centralized data
- centralized algorithms

#### Scaling techniques

- use only asynchronous communication
- distribution, split components
- replication of components

#### pitfalls

1. reliable network
2. secure network
3. homogenous network
4. constant topology
5. zero latency
6. infinite bandwidth
7. zero transport cost
8. one administrator!

#### Types of distributed systems

- computing systems
  - cluster computing

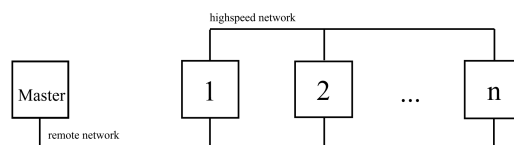


Figure 2.1: cluster computing

- grid computing(virtual organisation, geographically distributed and heterogenous))
- distributed information systems
  - transaction processing systems (database)
    - ACID (atomicity, consistency, isolated, durable)
  - enterprise systems
- Distributed pervasive systems
  - small, wireless, adhoc, no administration
  - home automation, health systems, sensor networks

#### Why do we need distributed systems?

- performance
- distribution inherent
- reliability
- incremental growth (scalability)
- sharing resources

## Chapter 3

# Architectures of distributed Systems

- how to split software into components  
⇒ Softwarearchitecture
  - how to build a system out of the components  
⇒ Systemarchitecture
- Middleware can help to create distribution transparency

Architecturestyles:

- Layered architecture  
⇒ network stack, messages or data flow up and down
  - control flow between layers
  - requests down
  - reply up
- Object-based architectures
  - interaction between components
  - e.g. remote procedure calls
  - can be client-server system
- data-centered architectures
  - data is key element
  - communication over data, distributed database
  - web-systems mostly data-centric
- event-based architecture
  - publish-subscribe systems
  - processes communicates through events
  - publisher announces events at broker
    - ⇒ loose coupling (publisher and subscriber need not to know each other), decoupled in space
    - ⇒ scalability better than client-server, parallel processing, caching

Event-based and data-based can be combined



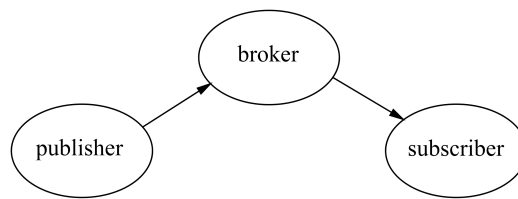


Figure 3.1: publish subscribe system

⇒ shared Data space

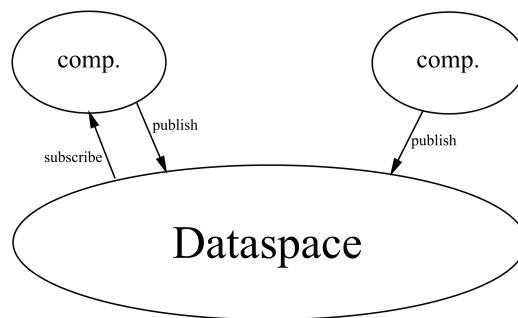


Figure 3.2: shared data space

### 3.1 System architectures

#### 1. centralized architectures

client - server

- (i) single point of failure
- (ii) performance (server is bottleneck)

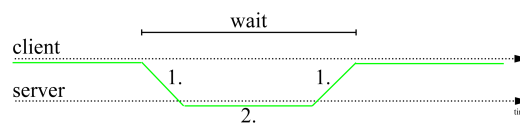


Figure 3.3: client server simple waiting situation

- (a) communication problems
- (b) server problems

can request be repeated without harm?

⇒ request is idempotent

(iii) application layering

Layers:

- 1.) User interface
- 2.) processing
- 3.) data level

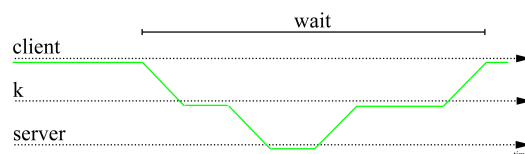


Figure 3.4: application layer

⇒ a lot of waiting

⇒ does not scale

## 2. Decentralized architectures

- vertical distribution (layering)  
different logic on different machines
- horizontal distribution  
replicated client/server operating on different data  
⇒ overlay-underlay hides physical structure by adding logical structure

### Structured P2P architectures

- most popular technique is distributed hash tables (DHT)
- randomly 128 bit or 160 bit ke for data and nodes. Two or more keys are very unlikely
- Chord system arranges items in a ring
- data item  $k$  is assigned to node with smallest identifier  $id \geq k$

ie item 1 belongs to node 1

item 2 belongs to node 2

for each item  $k_i$   $\text{succ}(k)=id$

returns the name of the node  $k$  is assigned to

to find data item  $k$  the function  $\text{LOOKUP}(k)$  returns the adress of  $\text{succ}(k)$  in  $O(\log(N))$ (later!)

### membership management

join:

create SHA1 identifier

$\text{LOOKUP}(id) = \text{succ}(id)$

contact  $\text{succ}(id)$  and  $\text{pred}(id)$  to join ring

leave:

node  $id$  informs  $\text{succ}(id)$  and  $\text{pred}(id)$  and assigns it's data to  $\text{succ}(id)$

## Content adressable network (CAN)

- d-dimensional cartesian space
- every node draws random number
- space is divided among nodes
- every data draws identifier (coordinates) which assigns a node
- join
  - select random point
  - half the square in which id falls
  - assign item to centers
- leave
  - one node takes the rectangle
  - ⇒ reassign rectangles periodically

## Unstructured P2P Network

- random graph
  - each node maintains a list of c neighbours
  - partial view or neighbourhood list with age
  - nodes exchange neighbour information
- active thread  
select peer

### PUSH

select c/2 youngest entries+myself  
send to peer

### PULL

receive peer buffer  
construct new partial view  
increment age

passive thread  
recieve buffer from peer

### PULL:

select c/2  
send to peer  
construct new partial view increment age

## **Chapter 4**

# **PeerSim**

# Chapter 5

## Processes

### processes

- execution of program
- processor creates virtual processor
- for each program everything is stored in process table
- transparent sharing of resources,(processor, memory) separation
- each virtual processor has it's own independent adress space
- process switch is expensive, (save cpu context, pointers, translation lookaside buffer (TLB), memory management unit (MMU))
- perhaps even swaps to disk, if memory exhausted

2 possible solutions:

1. scheduler activation, upcall to achieve process switch
2. light-weight processes (LWP)  
user level thread package

### threads

- several threads share CPU
- thread context has little memory information, perhaps mutex lock
- threads avoid blocking application (e.g. spreadsheet, computation of dependent cells, intermediate backup)
- thread switch is fast
- user level threads allow parallel computation of program sections
- I/O or other blocking system calls block all threads, but thread creation/deletion is kernel task = expensive
- advantages of threads over processes vanishes

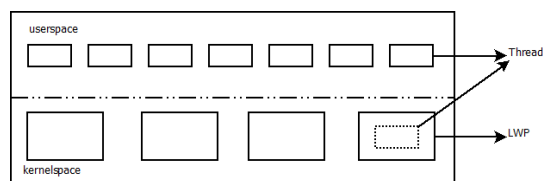


Figure 5.1: light-weight processes can run threads

execute scheduler and run thread of parent  
may block on systemcall, then other LWP may run  
triggered from userspace

Advantages of LWP and user-level thread package:

1. creation, deletion etc is easy, no kernel intervention
2. blocking syscall does not suspend process if enough LWPs are available
3. applications do not see LWP. They only see user-level threads
4. LWP can run on different processors in multiprocessor systems

Disadvantages:

1. LWP creation as expensive as creation of kernel-level thread

Advantages:

- a blocking systemcall blocks only thread, not process ⇒ system call for communication in distributed systems

Multiple threads in clients and servers

### **Clients:**

- multiple thread may hide communication delay (distribution transparency)
- web browser opens several connections to load parts of a document/page
- web server may be replicated in same or different location  
⇒ truly parallel access to items and parallel download

### **Servers:**

- single threaded, e.g. file server  
thread serves incoming request, waits for disk, returns file  
serves next
- multithreaded  
dispatcher thread receives request  
hands over to worker thread  
waits for disk etc.  
dispatcher takes next request
- finite state machine  
only one thread  
examines request, either read from ... or from disk  
during wait stores requests in table  
serves next request  
manage control either new request or reply from disk  
act accordingly  
process acts as finite state machine that receives messages and acts/changes state

### **Summary:**

model	characteristics
single thread	no parallelism, blocking syscalls
multi thread	parallelism, blocking syscalls
finite state machine	parallelism, non-blocking syscalls

## **5.1 Virtualisation**

V pretends there are more resources than available.

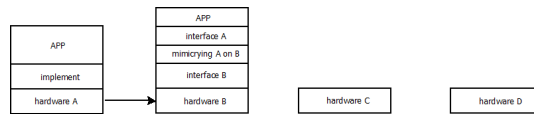


Figure 5.2: virtualisation

#### Reasons for the need for Virtualization

- hardware changes much faster then SW
- ⇒ improves portability
- networks consist of different hardware
- ⇒ enables portability of programs for all usage (distributed applications, network protocols)

#### 2 Types of Architectures for Virtualisation:

1. Runtime system providing instruction set  
Virtualization of processes (e.g. Wine)
  - interpreted as Java
  - emulated as for Windows applications on UNIX-platform processes VM
2. Virtualisation shields hardware and offers instruction set of the same or other hardware
  - can host different OS that run simultaneously
  - ⇒ VMM such as VMware, Xen

## 5.2 Client-/Serverprocesses

#### Clients:

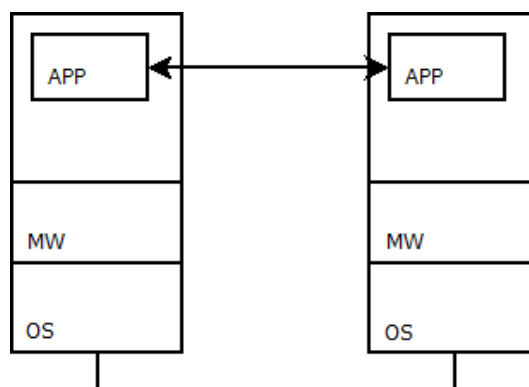


Figure 5.3: app specific communication

- b) allows to store data at the server
- **thin client** e.g. X-windows

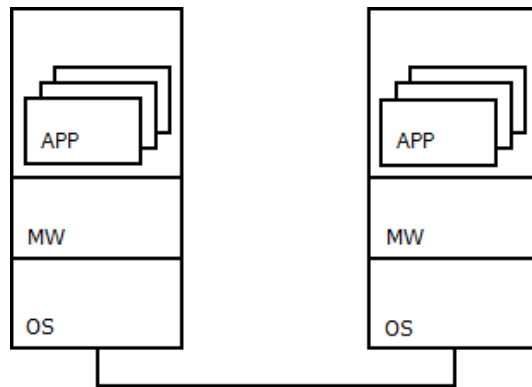


Figure 5.4: machine only communication

- thin client should separate application logic from user interaction
- often not implemented  $\Rightarrow$  poor performance
- compression of interaction commands as solution
- compound documents where user interaction triggers several processing steps on the server. must be implemented (e.g. rotation of picture changes placement in texts)

#### Servers:

- serves requests on behalf of the client (can server one request at a time)
- Types of servers
  - **iterative Server** handles requests itself
  - **concurrent server** passes requests to worker, e.g. multithreaded server
- server listens to port, endpoint to the client; some ports are reserved for special services

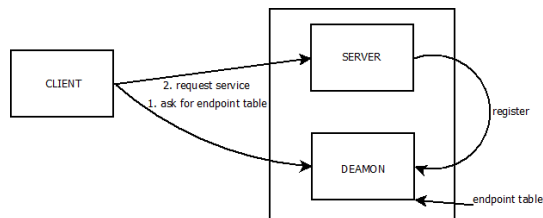


Figure 5.5: listener server

- superserver listens to several ports, replacing several (mostly idle) servers
- stateless servers, keeps no information on state of client  $\rightarrow$  change state without informing the client, e.g. web server
- soft state server, maintains client state for limited time, e.g. servers informing about updates
- stateful server keeps information about client (file server keeps (client, file) table), often better performance, fault-tolerance poorer
- cookies allow to share information for server upon next visit client sends its cookies, allows state information for stateless server

#### Distributed Servers



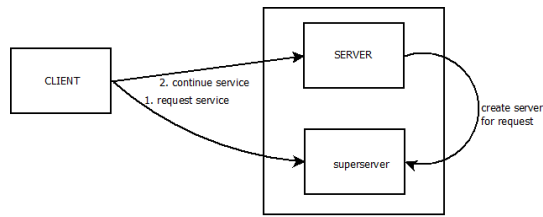


Figure 5.6: superserver

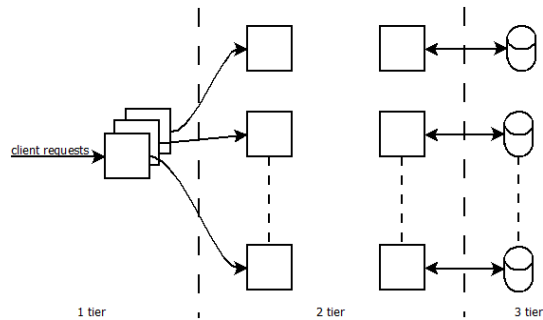


Figure 5.7: stateless server

- servers in different locations that have different ip-addresses in DNS under the same name
- MIPv6: mobility support for IPv6
- mobile node has home network with stable home address (HoA)
- special router is home agent and takes care of traffic to the mobile node
- mobile node receives care-of-address (CoA), never seen by client
- route optimisation avoids routing through home agent

### 5.3 Code Migration

- Code migration on (running) process - Why?
- service placement in distributed system  $\Rightarrow$  minimize communication cost
- load balancing in multiprocessor machine or cluster  $\Rightarrow$  performance
- (security)

#### Models of Migration

- or process model
  1. code segment, instructions
  2. resource segment, references to external resources, i.e. file, printer, devices
  3. execution segment, execution state process, stack, private data, program counter
- **Migration types**
  - weak mobility, transfer code, (1), mabe 3)), which executes from beginning (i.e. java applets)
  - strong mobility, transfer 1)3), stop executions, transfer, resume

Migrating resource segment 2) is difficult

Consider process to resource binding

1. binding by identifier, URL, ftp-server-name

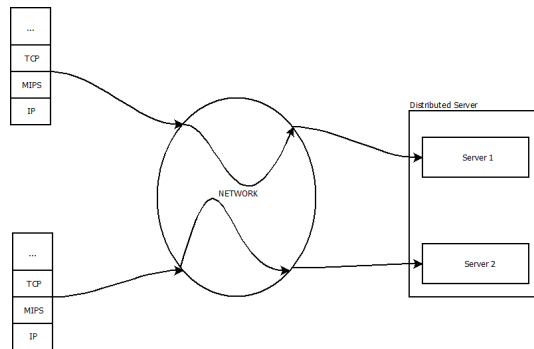


Figure 5.8: distributed server

2. binding by value, libraries for programming
3. binding by type, local device, monitor

### Resource-machine-binding

1. unattached
2. fastend
3. fixed

pass tp resource binding	unattached	fastened	fixed
by identifier	MV	GR(or MV)	GR
by value	CP	GR(or CP)	GR
by type	RB	RB(or GR,CP)	RB(or GR)

MV:move, GR, global refer-

ence, CP: copy value, RB: rebind to locally available resource

## Chapter 6

# Communication

- Communication in distributed systems is always based on low-level message passing as offered by the underlying network
- message passing is harder than using primitives based on shared memory, as in nondistributed systems
- low-level communication facilities of computer networks are in many ways not suitable due to their lack of distribution transparency.

### 6.1 RPC - Remote Procedure Call

- allow programs to call procedures located on other machines
- When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B.
- Remote procedure call uses stubs to pack parameters in message
- client stub: packs the parameters into a message and requests that message to be sent to the server
- server stub: transforms requests coming in over the network into local procedure calls
- No message passing at all is visible to the programmer
- neither client nor server need to be aware of the intermediate steps or the existence of the network

#### **A remote procedure call occurs in the following steps:**

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's as sends the message to the remote as.
4. The remote as gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.

8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

## Parameter Marshaling

parameter marshaling: packing parameters into a message is called

### Passing Value Parameters

- values are packed into messages (client) and unpacked from messages (server)
- transferred byte-by-byte
- as long as the client and server machines are identical this model works fine
- in a large distributed system, it is common that multiple machine types are present
- ⇒ problems because of different character encoding (EBCDIC vs ASCII), representation of integers (one's complement vs two's complement) or endianness (little endian vs. big endian)

### Passing Reference Parameters

- extremely difficult
- pointers are meaningful only within the address space of the process in which it is being used
- replace with copy/restore: copy the datastructure, send it to the server, work on it, send it back, restore at the client

## 6.2 Asynchronous RPC

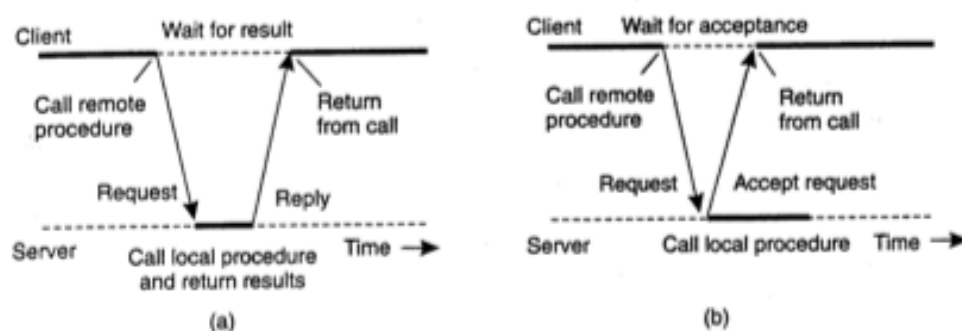


Figure 6.1: a: synchronous b: asynchronous RPC

- in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned
- asynchronous RPCs: the server immediately sends a reply back to the client the moment the RPC request is received. Reply acts as an acknowledgment.
- client will continue without further blocking as soon as it has received the server's acknowledgment

- Examples: transferring money from one account to another, adding entries into a database, starting remote services, batch processing...
- Asynchronous RPCs can also be useful when a reply will be returned but the client doesn't need to wait for it and can do nothing in the meantime
- One-Way RPCs: the client does not wait for an acknowledgment from the server
- deferred synchronous RPC: organize the communication between the client and server through two asynchronous RPCs
- foo

## 6.3 Message oriented communication

General Idea: avoid synchronous communication which blocks sender (RPC)

### 6.3.1 Message-Oriented Transient Communication

transient: flüchtig, vorübergehend

#### Berkeley Sockets

A socket is a communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual communication end point that is used by the local operating system for a specific transport protocol.

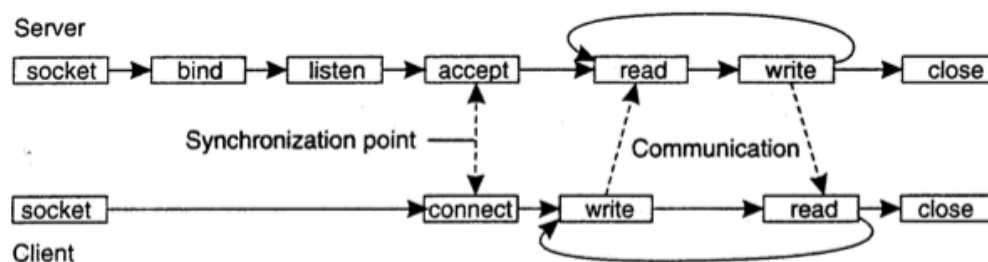


Figure 6.2: Connection-oriented communication pattern using sockets

- socket: create a new communication end point
- bind: attach a local address to a socket
- listen: announce willingness to accept connections
- accept: block caller until a connection request arrives
- connect: actively attempt to establish a connection
- send: send some data over the connection
- receive: receive some data over the connection
- close: release the connection

#### Message-passing-interface (MPI)

- standard for message passing

- designed for parallel applications
- communication within groups of processes
- A (*groupID*, *processID*) pair uniquely identifies the source or destination of a message (used instead of a transport-level address)

### 6.3.2 Message-Oriented Persistent Communication

aka Message-queuing-system, Message-oriented-middleware (MoM)

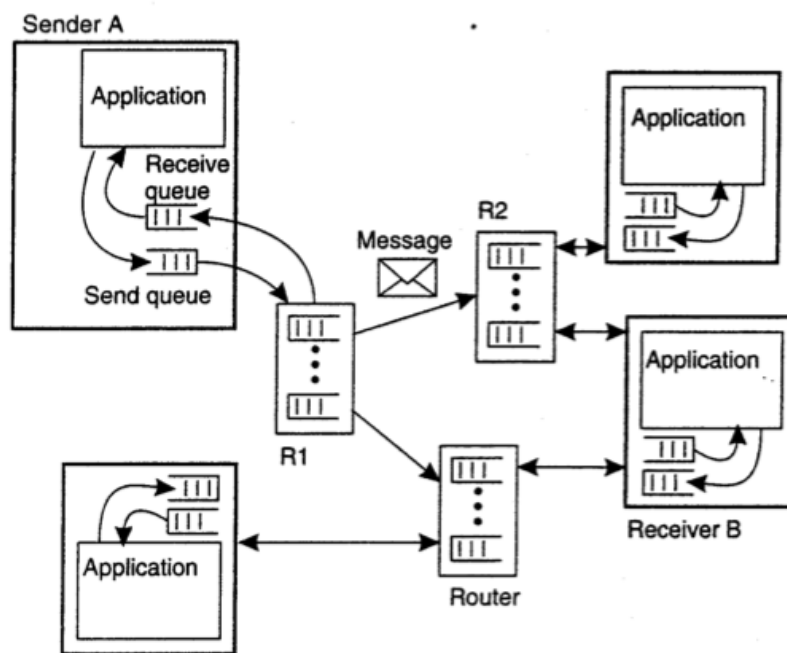


Figure 4-20. The general organization of a message-queuing system with routers.

Figure 6.3: general organization of a message-queuing system with routers

- asynchronous persistent communication
- offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission
- transfer may take minutes, not milliseconds
- applications communicate by inserting messages into queues
- messages are only put into and read from local queues
- the message-queuing system takes care that messages are transferred from their source to their destination queue
- message carries destination address
- queue managers
  - a queue manager interacts directly with the application that is sending or receiving a message

- also special queue managers that operate as routers, or relays: they forward incoming messages to other queue managers
- message brokers transform type A into type B, using a set of rules
  - application-level gateway in a message-queuing system
  - convert incoming messages so that they can be understood by the destination application
  - transform messages of type A into type B, using a set of rules
- Examples: Email, workflow, batch processing, queries accross several databases

## 6.4 stream oriented communication

- temporal relationship between items important
- multimedia data is compressed
- QoS is important
  - bit rate
  - max delay for session setup
  - max end-to-end delay
  - max delay variance (jitter)
  - max round trip delay
- networking solution such as differentiated services
- synchronisation of streams

## 6.5 Multicast communication

- application level multicast uses overlay
- tree, unique path between each pair of nodes
- mesh, more robust, fault-tolerant

**Example:** Construct overlay tree for chord

- node that wants to start multicast generates key 128bit/160bit (nid) randomly
- lookup of succ(nid) finds node responsible for key mid  
⇒ succ(nid) becomes root of tree
- join: lookup (nid) creates lookup message with join request routed from P to succ(nid)
- request is forwarded Q (first time forward), Q becomes forwarder  
⇒ P child of Q
- request is first time forwarded by R, R becomes forwarder  
⇒ Q becomes child of R
- multicast: lookup(nid) sends message to the root  
multicast from root

**Efficiency?**

Quality of application level tree

1. Link stress, number of traversals of same link per packet
2. stretch, relative delay penalty (RDP)
 
$$\frac{\text{transmission time in overlay}}{\text{transmission time in delay/network}} \Rightarrow \text{minimize aggregated stretch, average RDP over all note pairs}$$
3. tree cost, minimize aggregated link cost, link cost = cost between end points  
⇒ find minimal spanning tree

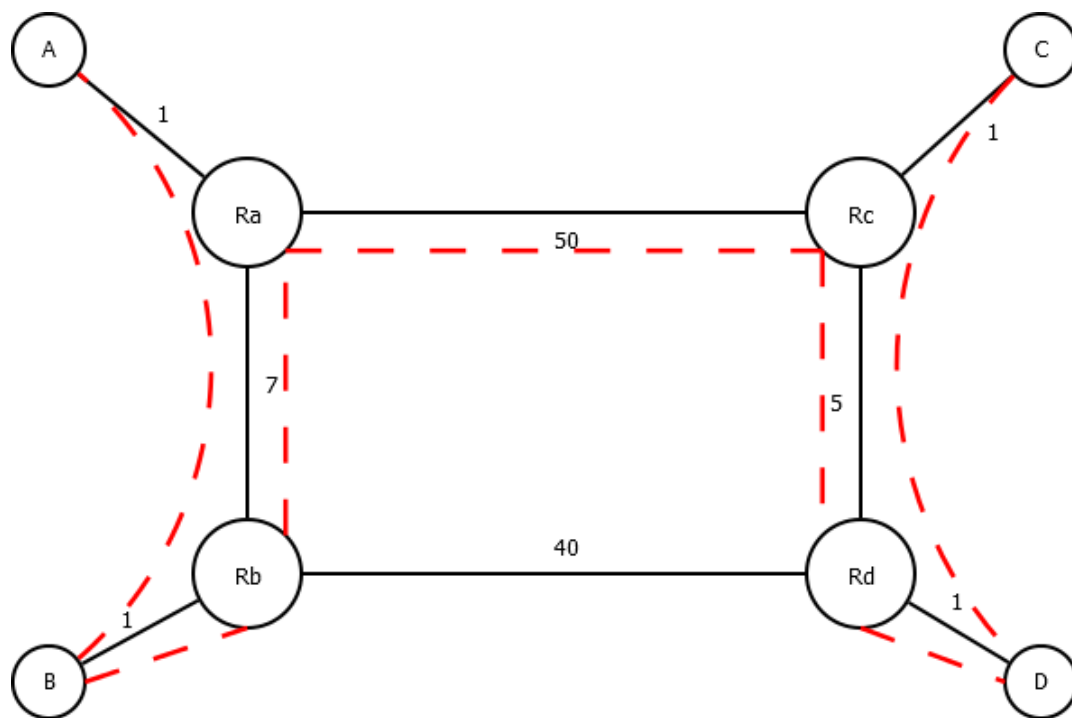


Figure 6.4: example of an overlay network

## 6.6 Gossip-based-communication

- epidemic behaviour
- a node does not have new data (susceptible), it has the data (infected) or is unwilling to spread (removed)  
Anti-entropy-model  
P chooses randomly Q
  1. P pushes its data to Q
  2. P pulls Q's data
  3. P and Q exchange data
- if many nodes are infected probability for selecting susceptible node is low  
⇒ low probability of data dissemination
- pull works when many nodes are infected. Susceptible node determines spread. They have a high probability to contact infected nodes
- if only one node is infected push/pull is best
- Round is period in which each node at least once selects a neighbor  
number of rounds needed to spread  $\approx \mathcal{O}(\log(N))$ ,  $N$  is number of nodes

### Rumor spreading, gossiping:

function of nodes that never obtain data:  $s = e^{\mathbb{W}(k \wedge 1)(t \mathbb{W} s)}$

e.g.  $k = 4$ ,  $\ln(s) = 4.97$



$\Rightarrow s = 0,007$

less than 0,7 remain without data

removing data is difficult : delete message is send via gossiping

# Chapter 7

## Naming

### 7.1 Flat naming

#### 7.1.1 Distributed Hash Tables

- m-bit identifier (128 or 160 Bit)
- entity with key k is under jurisdiction of node with smallest identifier  $id \geq k$   
 $\Rightarrow succ(k)$
- resolve key k to address of  $succ(k)$
- option 1: each node p keeps  $succ(p)$ ,  $pred(p)$  node forwards request for key k to a neighbor  
if  $pred(p) < k \leq p$ , return(p)  
 $\Rightarrow$  not scalable
- better solution: each Chord node maintains finger table of length m

$$\forall 1 \leq i \leq m \times FT[i] = succ(p \wedge 2^{i\wedge 1}) \bmod 2^m$$

$FT[i] = succ(p \wedge 2^{i\wedge 1}) = succ(p \wedge 1) = succ(2)$  (smallest id, such that  $id \geq 2$ )  
i-th entry points to  $2^{i\wedge 1}$  ahead of p

- to lookup k node p forwards request to p with index j in ps finger table:  
 $q = FT_p[j] \leq k < FT_p[j+1]$
- example:  
resolve k = 26 from node 1  
 $k = 26 > FT_1[5] \Rightarrow$  forward request to node  
 $18 = FT_1[5]$ 
  - node 18 selects node  $20 = FT_{18}[2] \leq k < FT_{18}[3]$
  - node 20 selects node 21  $\Rightarrow$  28 which is responsible for key 26
  - lookup generally requires  $O(\log(N))$  steps, N nodes in system
  - join/leave is rather simple
  - keeping finger table up to date is expensive
- example2:

30	
1	37
2	37
3	37
4	38
5	45
6	0

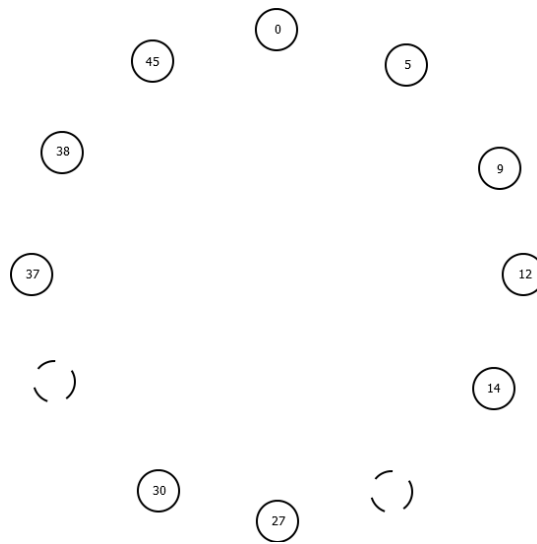


Figure 7.1: Ring

Superpeers:  $2^k$ , therefore  $2^{m \setminus k}$  normal nodes.  
 Probability that a node is a supernode:  $n \cdot \frac{2^k}{2^m} = 2^{k \setminus m} \cdot n$ , with  $m$  number of bits,  $k$  is number of bits that mark superpeers,  $n$  is number of nodes (not maximum possible but the actual number).  
 Number of superpeers to be expected:  $E(x) = n \cdot p$

## Chapter 8

# Synchronisation

### 8.1 Clock synchronisation algorithms

System model: each machine has timer that causes  $H$  interrupts per second

- clock  $C$  adds up ticks (interrupts)
- $C_p(t)$  is clock time on machine  $p$
- perfect clock:  $\forall p, t \times C_p(t) = t$   
 $\iff C'_p(t) = \frac{dC_p(t)}{dt} = 1$   
 $\equiv$  frequency of clock  $C_p$  at time  $t$

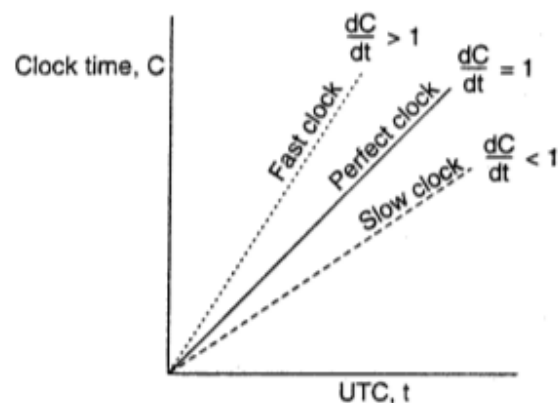


Figure 6-5. The relation between clock time and UTC when clocks tick at different rates.

Figure 8.1: fast, slow & perfect clock

- $C'_p(t) \setminus 1 \equiv$  skew of  $p$ 's clock, difference to perfect clock.
- $C_p(t) \setminus t \equiv$  offset
- real timers do not interrupt exactly  $H$  times per second
- maximum drift  $\rho$  is a constant guaranteed/specified by the vendor

$$1 \vee \rho \leq \frac{dC_p(t)}{dt} \leq 1 \wedge \rho$$

- at time  $\Delta t$  after two clocks were synchronized the drift can be max:  
 $|C_2(\Delta t) \vee C_1(\Delta t)| \leq 2\rho \Delta t$
- if the difference should never exceed  $\delta$  then synchronisation every  $\frac{\delta}{2\rho}$  seconds is needed
- time always moves forward.

**example:** given values for  $C_1$  and  $C_2$ :

$$C_1 = \rho = 0.001$$

$$C_2 = \rho = 0.001$$

$$\frac{dC}{dt} = \frac{1}{20.001} = \frac{1}{0.002} = \frac{1}{2} 10^3$$

Clocks  $C_1$  and  $C_2$  need to be synchronized every 500s to keep time in the same second.

## 8.2 Network Time Protocol (NTP)

- nodes contact time server that has an accurate clock
- time server passive

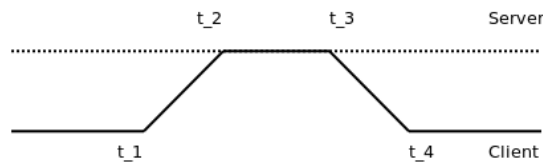


Figure 8.2: ntp

A estimates its offset to B as

$$\Theta = \frac{(T_2 \vee T_1) \wedge (T_4 \vee T_3)}{2}$$

assuming communication time is symmetric

delay:

$$\delta = (T_4 \vee T_1) \vee (T_3 \vee T_2) = (T_2 \vee T_1) \wedge (T_4 \vee T_3)$$

- A probes B, B probes A
- NTP stores 8 pairs  $(\Theta, \delta)$  per node pair using  $\min(\delta)$  for smallest delay
- either A or B can be more stable
- reference node has stratum 1 (clock has stratum 0) (stratum = # Servers to a reference clock)
- lower stratum level is better, will be used.

## 8.3 Berkeley algorithm

- assumes no node has 'good' time
- time server polls all nodes for their time
- takes average and adjusts speed of nodes correspondingly
- all nodes agree on time, which may not be correct

## 8.4 Lamports Logical Clocks

- logical time need not correct in real time.

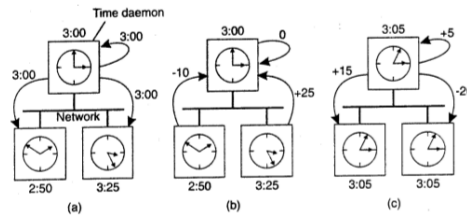


Figure 8.3: (a) The time daemon asks all the other machines for their clock values (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

- needs 'happens before' relation  $a \rightarrow b$ 
  1. if  $a, b$  are events in the same process and  $a$  happens before  $b$ , then  $a \rightarrow b$  is true
  2. if  $a$  denotes the event of sending a message and  $b$  the event of receiving this message by another process then  $a \rightarrow b$  is true
- happens before is transitive:  
 $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$
- concurrency:  
 if  $x, y$  happen in different processes and neither  $x \rightarrow y$  nor  $y \rightarrow x$ , then  $x, y$  are concurrent (which means, it is not known who comes first)
- $\forall$  events  $a$ , we can assign it a time  $C(a)$  on which all processes agree.
- if  $a \rightarrow b$  then  $C(a) < C(b)$
- 4 properties of logical time
  1. No two events get assigned the same time.
  2. Logical times of events in each process are strictly increasing
  3. logical time of send event is strictly smaller than receive event for the same message
  4. for any  $t \in T$  only finitely many events get assigned logical times smaller than  $t$ .
- Example:

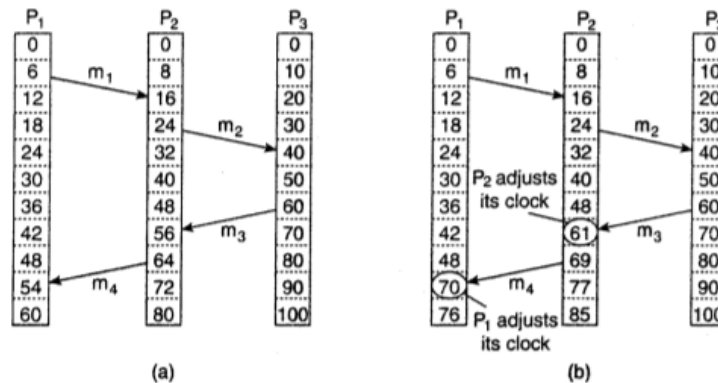


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

Figure 8.4: ...

### Algorithm

$C_i$  = local counter of process  $P_i$

1. Before executing an event (sending a msg over the network, delivering a msg to an app, some internal events)  $P_i$  executes  
 $C_i \leftarrow C_i \wedge 1$
2. When process  $P_i$  sends message  $m$  to  $P_j$  it sets the timestamp  $ts(m)$  of  $m$  to the current time  
 $ts(m) \leftarrow C_i$ .
3. upon receipt of a message  $m$  process  $P_j$  adjust its time to  $C_j \leftarrow \max C_j, ts(m)$ , then executes step 1 and delivers message

### Examples

- Consider a bank with two data centers A and B, that need to be kept consistent. Each request uses the nearest copy.  
Assume a customer has \$1000,- in his bank account and decides to add \$100,- using copy A. At the same time 1% interest is added to copy B. What happens? How can we solve the problem?
- Totally ordered multicast
  - every message is sent to all receivers+itself with timestamp
  - Assumption 1: messages from the same sender are received in the order they were sent, and that no messages are lost
  - Assumption 2: no two events get assigned the same time
  - When msg is received
    1. put it into a local priority queue ordered to the msg's timestamp
    2. multicast an acknowledgment to all other processes ( following Lamport  $\Rightarrow ts(msg) < ts(ack)$  )
  - eventually all queues are identical  $\Rightarrow$  total order
  - Process delivers a queued msg to the app only when
    1. msg is at head of queue
    2. ack received by every process

## 8.5 Vector Clocks

- Problem with Lamport: does not capture causality
- $T_{sent}(m_i) < T_{recv}(m_i) \stackrel{?}{\Rightarrow} T_{recv}(m_i) < T_{sent}(m_j)$
- In the case  $m_i = m_1$  and  $m_j = m_3$  we know at  $P_2$  that  $m_j$  was sent after  $m_i$  was received. This may indicate that sending of  $m_j$  has something to do with the receiving of  $m_i$
- Vector Clocks:  
 $VC(a) < VC(b)$  means, that event a is known to causally precede event b.
- Each process  $P_i$  maintains a Vector  $VC_i$  with the following properties:
  1.  $VC_i[i]$  is the number of events that occurred so far at  $P_i$   
 $VC_i[i]$  is the logical clock at  $P_i$
  2. if  $VC_i[j] = k$  then  $P_i$  knows, that  $k$  events have occurred at  $P_j$ . It is thus  $P_i$ 's knowledge of the local time at  $P_j$
- To maintain properties:
  1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event),  $P_i$  executes  $VC_i[i] = VC_i[i] \wedge 1$
  2. When process  $P_i$  sends a message  $m$  to  $P_j$ , it sets  $m$ 's (vector) timestamp  $ts(m)$  equal to  $VC_i$  after having executed the previous step
  3. Upon the receipt of a message  $m$ , process  $P_j$  adjusts its own vector by setting  $VC_j[k] = \max(VC_j[k], ts(m)[k])$  for each  $k$ , after which it executes the first step and delivers the

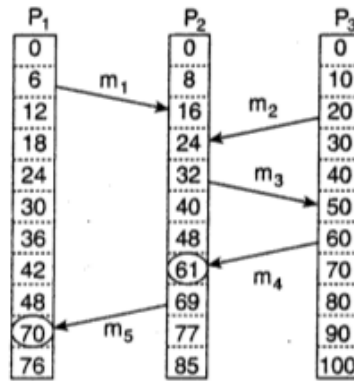


Figure 6-12. Concurrent message transmission using logical clocks.

Figure 8.5: ...

message to the application.

- timestamp  $ts(m)$  tells the receiver how many events in other processes have preceded the sending of  $m$ , and on which  $m$  may causally depend.
- casually-ordered multicast is weaker than totally ordered multicast
- delivery of message  $m$  from  $P_i$  to  $P_j$  to application layer will be delayed until:
  1.  $ts(m)[i] = VC_j[i] \wedge 1$   
=  $m$  is the next message that  $P_j$  was expecting from process  $P_i$
  2.  $ts(m)[k] \leq VC_j[k] \quad \forall k \neq i$   
=  $P_j$  has seen all the messages that have been seen by  $P_i$  when it sent message  $m$
- Better to be implemented on application layer, because the app knows which messages are causally related.

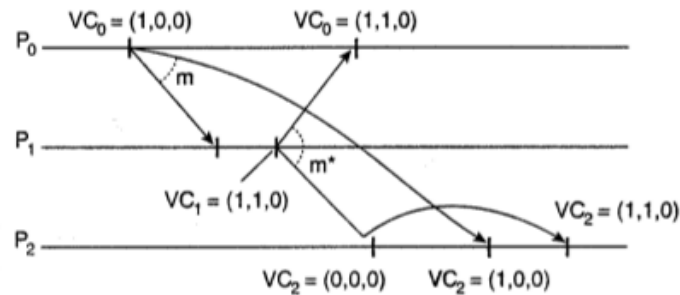


Figure 6-13. Enforcing causal communication.

Figure 8.6: ...



## Chapter 9

# Mutual Exclusion

Access to shared resources

2 types of algorithms: token-based and permission-based

- token is simple, reliability problem (lost token)
- permission difficult in distributed systems

### 9.1 Centralized algorithm

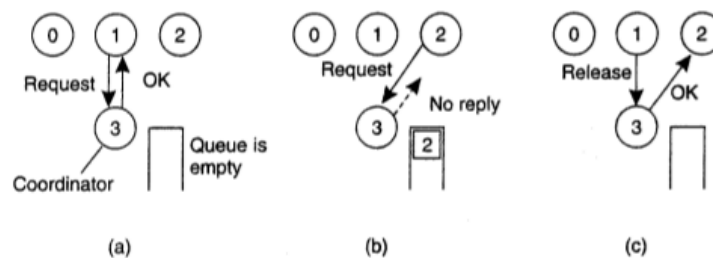


Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Figure 9.1: ...

- one process is coordinator
- coordinator allows access only to one process
- fair, requests are processed in order of arrival
- no starvation
- easy to implement
- coordinator is single point of failure
- (handle message loss with ack)
- dead coordinator looks like permission denied

## 9.2 Decentralized algorithm

- Each resource is replicated  $n$  times,  $rname\_i$  is the name of the replica
- each replica has it's own controller, the name is a hash of the  $rname\_i$
- if  $rname$  is known, each process can generate the address of the controllers
- access to resource when  $m > n/2$  controllers grant it
- Let  $p$  probability that a coordinator resets during  $\Delta t$   
 $P[k] = \text{prob}\{k \text{ out of } m \text{ coordinators reset during } \Delta t\} = \binom{m}{k} p^k (1-p)^{m-k}$
- at least  $2m \forall n \geq n \wedge 2 \forall n = 2$  coordinators need to reset in order to violate the voting. This happens with probability  $\sum_{k=2m \forall n}^n P[k]$   
 e.g.  $\Delta t = 10s, n = 32, m = 0,75n$   
 Probability of violation in  $10^{40}$
- if a process gets less than  $m$  votes access to the resource is denied
- random backoff, retry  
 many requests, noone gets access
- heavy load  $\Rightarrow$  drop in utilisation

## 9.3 A distributed algorithm

- deterministic
- uses total ordering of events
- process that wants to access a resource sends out message containing (resourcename, process no, current localtime) to all other processes and itself
- process receives a message. Either:
  1. returns OK, if does not want a resource
  2. queues request, if it has resource
  3. compares timestamps, sends OK if timestamp is smallest, queues request and sends no reply else
- grants mutual exclusion without deadlocks or starvation

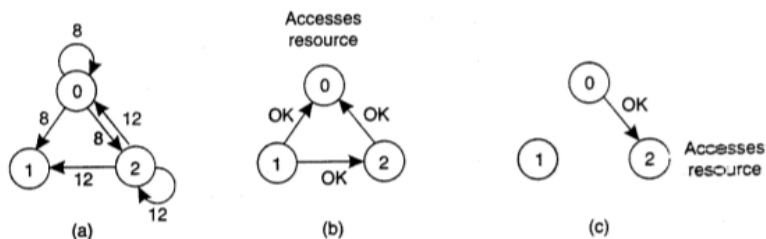


Figure 6-15. (a) Two processes want to access a shared resource at the same moment. (b) Process 0 has the lowest timestamp. so it wins. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

Figure 9.2: ...

Problems:

- note failure  $\Rightarrow$  dito
- load, all processes take part in decisions (needs  $2(n-1)$  messages for  $n$  processes)
- algorithm is slower, more complicated, more expensive, less robust than centralised alg.
- not a good algorithm

## 9.4 Token Ring Algorithm

## 9.5 Centralized algorithm

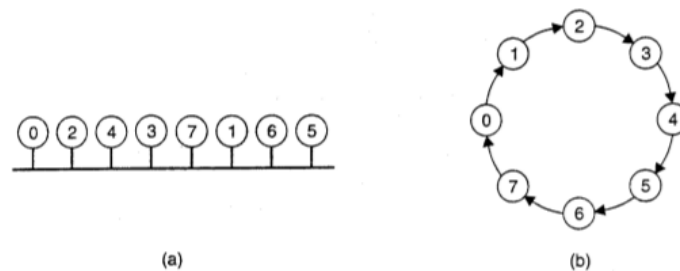


Figure 6-16. (a) An unordered group of processes on a network.. (b) A logical ring constructed in software.

Figure 9.3: ...

- processes form a logical ring
- token circulates
- owner of token can access resource
- simple and efficient
- not fair under heavy load

Problems:

- token loss  $\rightarrow$  regenerating?
- crashing nodes

## 9.6 Comparison

Algorithm	messages per entry/exit	Delay before access	Problems
Centralised	3	2	coordinator crash
Decentralised	$3mk$	$2m$	starvation, low efficiency
Distributed	$2(n \setminus 1)$	$2(n \setminus 1)$	crash of any process
Token Ring	1 to $\infty$	0 to $\infty$	lost token, process crash, fairness?

## Chapter 10

# Leader Election algorithms

### 10.1 leader election in a synchronnous ring

Network is a graph  $G$  consisting of  $n$  nodes connected by unidirectional links. Use  $\text{mod } n$  for labels

- elected node is "leader"
- leader election is not possible for identical processes/nodes

#### 10.1.1 LCR algorithm

(Lelan, Chang, Roberts)

- unidirectional communication
- ring size unknown
- only leader produces output
- algorithm compares UID

```
1 For each node
2   a = a UID, initially i's UID
3   send = a UID or NULL, initially i's UID
4   status = {unknown, leader} initially unknown
5
6 message generation
7   send = current value of send to node i+1
8
9 state transitions
10  send = NULL
11  if incoming message is v (a UID) then
12    v > u: send v
13    v = u: status = leader
14    v < u: do nothing
```

#### Correctness

Let max index of process with  $\max(\text{UID})$  let  $u_{\max}$  is its UID  
Show:

- (i) process max outputs "leader" after  $n$  rounds
- (ii) no other process does the same

We clarify:

(iii) After  $n$  rounds  $\text{status}_{\text{max}} = \text{leader}$   
and

(iv) For  $0 \leq r \leq n-1$  after  $r$  rounds

$$\text{send}_{\text{max}} = u_{\text{max}}$$

find UID at distance  $r$  from  $i_{\text{max}}$  as it has to go once around.

Show (iv) for all  $r$ : Induction  
then (iii)

### Complexity

- time complexity is  $n$  rounds
- communication complexity  $\mathcal{O}(n^2)$
- not very expensive in time many messages

## 10.1.2 Algorithm of Hirschberg and Sinclair (HS-Alg)

- reduces number of messages to  $\mathcal{O}(n \log n)$

```

1 each process has states with components
2   u, UID: initially i's UID
3   send+ containing NULL or (UID, flag {in, out}, hopcount): initially
   (i's UID, out, 1)
4   send as send+
5   status  $\in \{ \text{unknown}, \text{leader} \}$  initially unknown phase  $\in \mathbb{N}$ : initially 0
6
7 message generation
8   send current send+ to process  $i+1$ 
9   send current send as send to process  $i-1$ 
10
11 state transitions
12   send+ = NULL
13   send = NULL
14   if message from  $(i-1)$  is  $(v, \text{out}, h)$  then
15      $v > u \wedge h > 1$ : send+ =  $(v, \text{out}, h-1)$ 
16      $v > u \wedge h = 1$ : send =  $(v, \text{in}, 1)$ 
17      $v = u$  status = leader
18   if message from  $i+1$  is  $(v, \text{out}, h)$  then
19      $v > u \wedge h > 1$ : send =  $(v, \text{out}, h-1)$ 
20      $v > u \wedge h = 1$ : send+ =  $(v, \text{in}, 1)$ 
21      $v = u$  status = leader
22   if message from  $i-1$  is  $(v, \text{in}, 1)$   $\wedge v \neq u$  then
23     send+ =  $(v, \text{in}, 1)$ 

```

```

24 if message from i+1 is (v,in,1) & v≠u then
25   send W=(v,in,1)
26 if both messages from i and i+1 are (u,in,1) then
27   phase++
28   send+=(u,out, 2^{phase})
29   send W=(u,out, 2^{phase})

```

### Complexity

Total number of phases is at most  $\lceil \log(n) \rceil$

the total number of messages is at most  $n \lceil \log(n) \rceil \approx \mathcal{O}(n \log n)$

Total time complexity is at most  $3n$  if  $n$  power of 2 otherwise is  $5n$

### 10.1.3 Time slice algorithm

- ring size  $n$  is known
- unidirectional
- elects minimum

\*

```

1 phases with n rounds
2 in phase r consisting of rounds (v-1)n+1, ..., vn
3 only a token carrying UID v is permitted
4 if a process with UID v exists, then it elects itself leader and
   sends a token with its UID

```

**Complexity:** number of messages is  $n$ , time complexity  $n \cdot u_{min}$

### 10.1.4 Variable speeds algorithm

- each process  $i$  creates a token to travel around the ring, carrying UID  $u$  of origin
- tokens travel at different speed
- token carrying UID  $v$  travels 1 message every  $2^v$  rounds
- each process memorises smallest UID
- return to origin elects UID

### Complexity

...

How many messages in total?  $\sum_{k=1}^n \frac{1}{2^k} (< 2n)$

Time complexity:  $n \cdot 2^{u_{min}}$

## 10.2 Leader election in a wireless environment

- consider time needed for communication
- nodes set up a tree
- select based on information like battery lifetime
- node issues leader request to all its neighbours
- becomes parent if there is none yet

### **10.3 The Bully Algorithm(flooding) (Garcia-Mdina, 1982)**

- process P holds election
  1. P sends ELECT message to all processes
  2. P wins if there is no response  $\Rightarrow$  P is leader
  3. if Q answers, Q takes over

## Chapter 11

# Consistency and Consensus

- Distributed Systems use replication of data to improve performance and/or reliability
- replication for scalability
  - How to keep replicas consistent?
  - Many types of consistency
    - data-centric-consistency
    - client-centric-consistency
    - monotonic reads: successive reads return the same or newer value
    - monotonic write: a write op must be completed before the next write by the same process
    - read-your-own-write: write is always seen by read of same process
    - write-follows-read: write on previous read takes place on the same or more recent value
- Do not discuss replica placement

### 11.1 Reliable multicast protocols

- atomic multicast requirement
  - all requests arrive at all servers in the same order

#### 11.1.1 Distributed Commit

- an operation is performed by group or non of the nodes of the group
- reliable multicast operation = delivery of message
- distributed transaction: operation = execution of transaction
- uses coordinator
- one-phased commit
- two-phase commit (2PC) (Jim Gray, 1978)
  - distributed transaction involves several processors each on a different machine
  - 2 phases with each 2 steps:

1. coordinator  $\xrightarrow{\text{vote request}}$  all participants
2. participant  $\xrightarrow[\text{vote abort}]{\text{vote commit}}$  coordinator



3. if all commit  
 coordinator  $\xrightarrow{\text{global}\forall\text{commit}}$  all participants  
 else  
 coordinator  $\xrightarrow{\text{global}\forall\text{abort}}$  all participants

4. if commit, then participants locally commit  
 else participants locally abort

Problems if failures occur

- \* coordinator blocks in: wait
- \* participant blocks in: ready, init
- ⇒ blocking commit protocol
- use timeouts to unblock
- repeat request
- in state ready P can contact Q
  - if Q is in contact, then coordinator died after sending to Q
  - before sending to P ⇒ P can commit
  - if Q is in abort ⇒ abort
  - if Q is in init ⇒ abort
  - if Q is in ready → abort or no decision contact R

• Three-phase commit (3PC) (Steen, 1981)

- avoids blocking in the presence of fail-stop crashes
- states satisfy the following conditions
  1. there is no state from which directly follow commit or abort follows
  2. there is no state in which it is not possible to make a final decision and from which a transaction to a commit state can be made
 ⇒ necessary and sufficient conditions for non-blocking commit protocol
- abort branch as in 2PC
- blocking states: participant: init → abort  
 coordinator: wait → abort  
 precommit, knowing P voted for commit  
 ⇒ global-commit+recovery of P  
 participant: ready  
 coordinator failed as in 2PC  
 precommit: contact other participants: if Q in precommit ⇒ commit  
 if Q is in init ⇒ abort
- Q can be in INIT only if no participant is in precommit
- participant can reach precommit only if coordinator was in precommit already
- In 2PC a crashed participant could recover to commit, while all others are still in ready
- if one process is in ready recovery can be only to states ready, init, abort, precommit  
 ⇒ surviving processes can come to final solution

• Paxos (Leslie Lamport, late 80s)

- does not block with at most  $n/2-1$  failures
- Paxos adds to 2PC:
  - \* ordering of proposals

\* majority voting for acceptance

Duelling proposer

1. Aufgabe: Terminiologie (wichtige Konzepte, erklären, vergleichen, bla,bla) dann durch die Themen des Semsters, übungszettel, gerne ausrechnen (Fingertable, Metriken von overlaynetzen, komplexität von protokollen (wie viele nachrichten braucht ein protokoll), logische uhren (stellen oder so)), erklären, Peersimaufgabe(n) (programm angucken, was macht das programm?, überblick, wie modifizieren für fkt x, cycle driven vs event driven (was wofür)), last auf dem netz, kein gnuplot programm auf papier!!! Hilfsmittel: mitbringen, was man will, außer internet, telefon, freunde usw...