Lecture Notes
Distributed System

Hinnerk van Bruinehsen
Tobias Höppner
Tobias Famulla
Johannes Dillmann
Julian Dobmann
Jens Fischer

SoSe 2013

Contents

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

There are 4 goals of distributed systems:
1. Making Resources Accessible
2. Distribution Transparency
3. Openness
4. Scalability

2.1 Making Resources Accessible

- provide users (and applications) access to remote resources (printer, storage, computing)
- share ressources in a controlled efficient way

2.2 Distribution Transparency

Hide the fact that processes and resources are physically distributed. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.
- transparancy is desireable, but not always perfectly possible
- tradeoff between transparancy and complexity, maintainablility and performance

Types of transparancy:
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

2.3 Openness

- An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services
- service interfaces (syntax) specified using Interface Definition Language (IDL)
- service specification (semantics) as text

2.4 Scalability

is an important property, distributed systems should be scalable in

size number of nodes, users, resources
geographic spread geographical distribution of users and resources (may lie far apart)
administration manageability, even if even if it spans many independent administrative organizations

2.4.1 Scaling in size

If more users or resources need to be supported, we are often confronted with the limitations of centralized services, data, and algorithms. Scalability in size is limitated by:
centralized services A single server for all users (bottleneck)
centralized data A single on-line telephone book or non-distributed DNS
centralized algorithms Doing routing based on *complete* information

2.4.2 Geographical Scalability

- existing distributed systems were designed for LANs
- → LAN-based systems often use synchronous communication
- → LAN-based systems provide relatively reliable communication
- → in LAN-based systems broadcasting is possible
- all of this is not possible in WAN ⇒ problems in geographical scalability

2.4.3 Scalability in administration

how to scale a distributed system across multiple, independent administrative domains. Important security questions arise:
selfprotection from malicious attacks from the new domain
domainprotection the domain protects itself from attacks from a new domain

2.4.4 Scaling techiques

hiding communication latencies use only asynchronous communication
distribution split components into smaller parts :)
replication of components, chaching, enables load balancing

2.5 Pitfalls

False assumptions that everyone makes when developing a distributed application for the first time:
1. The network is reliable.
2. The network is secure.
3. The network is homogeneous.
4. The topology does not change.
5. Latency is zero.
6. Bandwidth is infinite.
7. Transport cost is zero.
8. There is one administrator.

2.6 Types of distributed systems

2.6.1 Distributed Computing Systems

Cluster Computing Systems

- the underlying hardware consists of a collection of similar workstations or PCs
- these closely connected by means of a high speed local-area network
- each node runs the same operating system
- used for instance for the building of supercomputers using off-the-shelf technology by hooking up a collection of relatively simple computers in a high-speed network
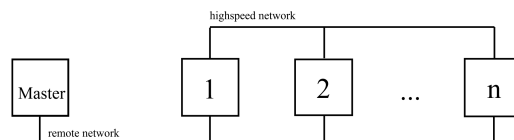
Figure 2.1: cluster computing

Grid Computing Systems

- grid computing systems have a high degree of heterogeneity: no assumptions are made concerning hardware, operating systems, networks, administrative domains, secu- rity policies, etc.
- virtual organization: resources from different organizations are brought together to allow the collaboration of a group of people or institutions
- geographically distributed

2.6.2 Distributed Information Systems

Transaction Processing Systems

A distributed system for processing transactions, e.g. a (distributed) database. Transactions are defined through fulfilling the **ACID** (atomicity, consistency, isolation, durability) properties:

Atomic To the outside world, the transaction, happens indivisibly
Consistens The transaction does not violate system invariants
Isolated Concurrent transactions do not interfere with each other
Duarble Once a transaction commits, the changes are permanent

Enterprise Application Integration

- the more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independent from their databases
- application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems
- main idea was that existing applications could directly exchange information via communication middleware
- see RPC

Distributed Pervasive Systems

- The distributed systems we have been discussing so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network
- in distributed pervasive systems, instability is the default behavior
- often small, wireless, adhoc, no administration
- Examples: Home automation, Health systems, Sensor Networks

Chapter 3

Architectures of distributed Systems

- how to split software into components
  ⇒ Softwarearchiticture
- 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 threeough 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
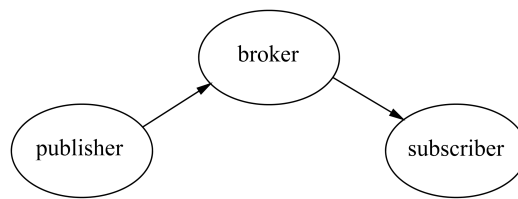
Figure 3.1: publish subsribe system

Event-based and data-based can be combined
$\Rightarrow$ shared Data space
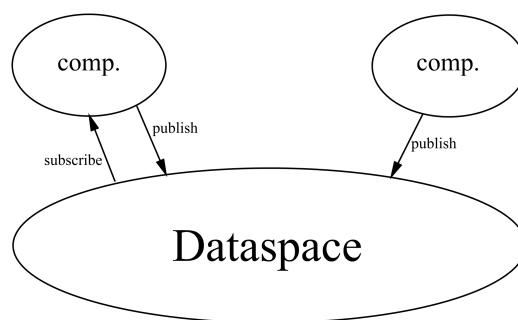
Figure 3.2: shared data space

3.1 System architectures

1. centralized architectures
   client - server

   (i) single point of failure
   (ii) performance (server is bottleneck)
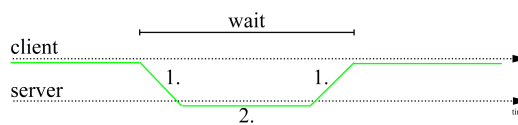
Figure 3.3: client server simple waiting situation

   (a) communication problems

   (b) server problems

can request be repeated without harm?
$\Rightarrow$ request is idempotent

  (iii) aplication layering
        Layers:
         1.) User interface
         2.) processing
         3.) data level
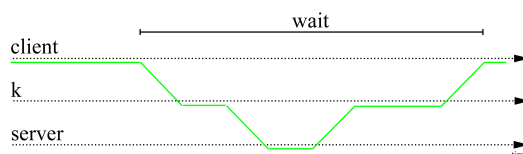
Figure 3.4: application layer

$\Rightarrow$ a lot of waiting
$\Rightarrow$ does not scale

2. Decentralized architectures

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

Structured P2P architectures

- most popular technique is distributed hashtables (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 assigened 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 succ(k)=id
returns the name of the node k is assigened to
to find data item k the function LOOKUP(k) returns the adress of succ(k) in O(log(N))(later!)

membership management
join:
create SHA1 identifier
LOOKUP(id) = succ(id)
contact succ(id) and pred(id) to join ring

leave:
node id informs succ(id) and pred(id) and assigns it's data to succ(id)

Content adressable network (CAN)

- d-dimensional cartesian space
- every node draws random number
- space is divided among nodes
- every data draws identifier (coodinates) 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
        $\Rightarrow$ 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

4.1   Kinds of simulation

cycle-based

- fixed cycles; every cycle the part in "next_cycle" is run
- **Observer:** after every cycle

event-based

- exchange of messages, the part in "processevent" is run (e.g. message delays)
- **Observer:** at specified point of time

4.2   Configuration

Specified are controls:

- protocols
- **Initializer:** run at the beginning of the simulation
- **Observer:** run during simulation, change nothing, at least one needed

experiments (number of repetitions), range (simulation is repeated with different parameters)

4.3   Initializer

- **wirekout** parameter k (number of nodes, normally random), creates a "wireplan" for the simulation
- **peekdistributioninitializer** creates one maximum and sets everything else to zero
- **lineardistributioninitializer** creates linear distribution from start to end value

4.4   Stats, order of parameters

`obs_name time min max n avg variance ct_min ct_max`

4.5   PeerSim simulation life-cycle

PeerSim was designed to encourage modular programming based on objects (building blocks). Every block is easily replaceable by another component implementing the same interface (i.e., the same functionality). The general idea of the simulation model is:

1. choose a network size (number of nodes)
2. choose one or more protocols to experiment with and initialize them
3. choose one or more Control objects to monitor the properties you are interested in and to modify some parameters during the simulation (e.g., the size of the network, the internal state of the protocols, etc)
4. run your simulation invoking the Simulator class with a configuration file, that contains the above information

4.6   Interfaces

Node, CDProtocol, Linkable, Control

4.7   Example: Average

4.7.1   AverageFunction

```
1  package example.aggregation;
2
3  import peersim.cdsim.CDProtocol;
4  import peersim.config.FastConfig;
5  import peersim.core.CommonState;
6  import peersim.core.Linkable;
7  import peersim.core.Node;
8  import peersim.vector.SingleValueHolder;
9
10 /**
11  * This class provides an implementation for the averaging function in the
12  * aggregation framework. When a pair of nodes interact, their values are
13  * averaged. The class subclasses {@link SingleValueHolder} in order to provide
14  * a consistent access to the averaging variable value.
15  *
16  * Note that this class does not override the clone method, because it does not
17  * have any state other than what is inherited from {@link SingleValueHolder}.
18  */
19 public class AverageFunction extends SingleValueHolder implements CDProtocol {
20
21     /**
22      * Creates a new {@link example.aggregation.AverageFunction} protocol
23      * instance.
24      *
25      * @param prefix
26      *            the component prefix declared in the configuration file.
27      */
28     public AverageFunction(String prefix) {
29         super(prefix);
30     }
31
32     /**
```

```
33          * Using an underlying {@link Linkable} protocol choses a neighbor and
34          * performs a variance reduction step.
35          *
36          * @param node
37          *                the node on which this component is run.
38          * @param protocolID
39          *                the id of this protocol in the protocol array.
40          */
41         public void nextCycle(Node node, int protocolID) {
42             int linkableID = FastConfig.getLinkable(protocolID);
43             Linkable linkable = (Linkable) node.getProtocol(linkableID);
44             if (linkable.degree() > 0) {
45                 Node peer = linkable.getNeighbor(CommonState.r.nextInt(linkable
46                         .degree()));
47
48                 // Failure handling
49                 if (!peer.isUp())
50                     return;
51
52                 AverageFunction neighbor = (AverageFunction) peer
53                         .getProtocol(protocolID);
54                 double mean = (this.value + neighbor.value) / 2;
55                 this.value = mean;
56                 neighbor.value = mean;
57             }
58         }
59     }
```

4.7.2 AverageObserver

```
1  package example.aggregation;
2
3  import peersim.config.*;
4  import peersim.core.*;
5  import peersim.vector.*;
6  import peersim.util.IncrementalStats;
7
8  /**
9   * Print statistics for an average aggregation computation. Statistics printed
10  * are defined by {@link IncrementalStats#toString}
11  */
12 public class AverageObserver implements Control {
13     /**
14      * Config parameter that determines the accuracy for standard deviation
15      * before stopping the simulation. If not defined, a negative value is used
16      * which makes sure the observer does not stop the simulation
17      *
18      * @config
19      */
20     private static final String PAR_ACCURACY = "accuracy";
21
22     /**
23      * The protocol to operate on.
24      *
25      * @config
26      */
27     private static final String PAR_PROT = "protocol";
28
29     /**
30      * The name of this observer in the configuration. Initialized by the
31      * constructor parameter.
32      */
```

```java
33      private final String name;
34
35      /**
36       * Accuracy for standard deviation used to stop the simulation; obtained
37       * from config property {@link #PAR_ACCURACY}.
38       */
39      private final double accuracy;
40
41      /** Protocol identifier; obtained from config property {@link #PAR_PROT}. */
42      private final int pid;
43
44      /**
45       * Creates a new observer reading configuration parameters.
46       */
47      public AverageObserver(String name) {
48          this.name = name;
49          accuracy = Configuration.getDouble(name + "." + PAR_ACCURACY, −1);
50          pid = Configuration.getPid(name + "." + PAR_PROT);
51      }
52
53      /**
54       * Print statistics for an average aggregation computation. Statistics
55       * printed are defined by {@link IncrementalStats#toString}. The current
56       * timestamp is also printed as a first field.
57       *
58       * @return if the standard deviation is less than the given
59       *         {@value #PAR_ACCURACY}.
60       */
61      public boolean execute() {
62          long time = peersim.core.CommonState.getTime();
63
64          IncrementalStats is = new IncrementalStats();
65
66          for (int i = 0; i < Network.size(); i++) {
67
68              SingleValue protocol = (SingleValue) Network.get(i)
69                      .getProtocol(pid);
70              is.add(protocol.getValue());
71          }
72
73          /* Printing statistics */
74          System.out.println(name + ": " + time + " " + is);
75
76          /* Terminate if accuracy target is reached */
77          return (is.getStD() <= accuracy);
78      }
79  }
```

17

Chapter 5

Processes

processes
-execution of program
-processor creates virtual processor

-for each program everyting 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

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

Summary:

- Serveral threads share one process.
- Processes have own address space, threads don't.
- Processes and threads each have userland and kernel implementations

2 possible solutions:

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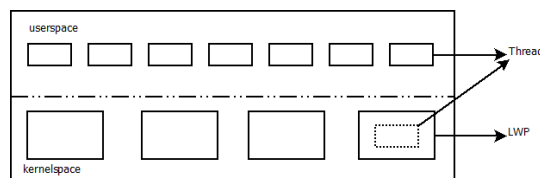   user level thread package

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 $\Rightarrow$ 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
  $\Rightarrow$ 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 recieves 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
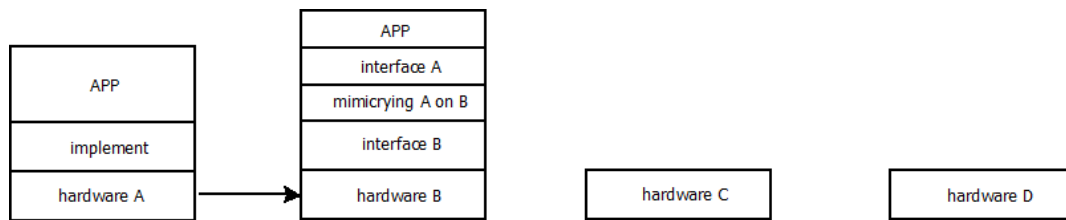
Figure 5.2: virtualisation

V pretends there are more resources then available.

Reasons for the need for Virtualization
-hardware changes much faster then SW
$\Rightarrow$ improves portability
-networks consist of different hardware
$\Rightarrow$ enables portability of programs for all
usage (distributed applications, network protocols)

2 Types of Architectures for Virtualisation:

1. Runtime system providing syscall API/ABI
    Virtualization of processes (e.g. Wine)

    - syscalls are interpreted (like 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 simultaneosly
    $\Rightarrow$ VMM such as VMware, Xen

5.2 Client-/Serverprocesses

Clients:
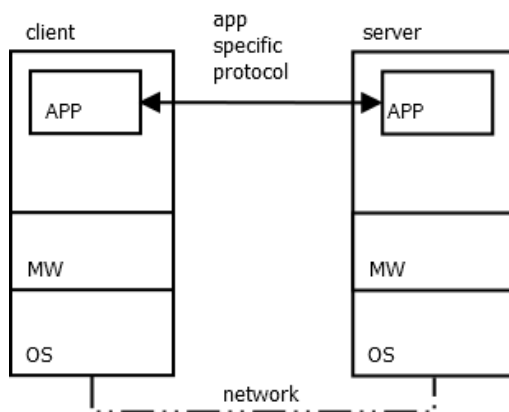
Figure 5.3: a) app specific communication
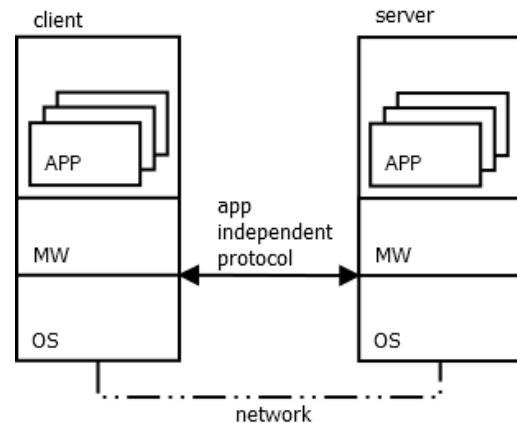
Figure 5.4: b) machine only communication

- b) allows to store data at the server
- **thin client** e.g. X-windows
- 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
- stateless servers, keeps no information on state of client $\rightarrow$ change state without informing the client, e.g. web server

Figure 5.5: listener server

Figure 5.6: superserver

Figure 5.7: stateless server

Figure 5.8: distributed 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
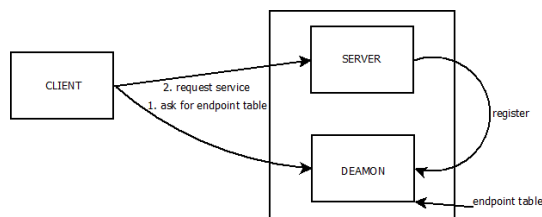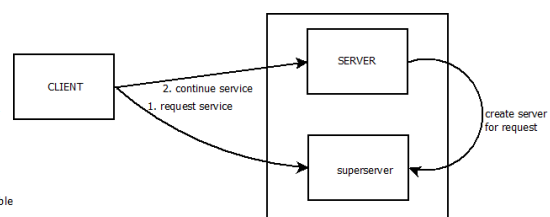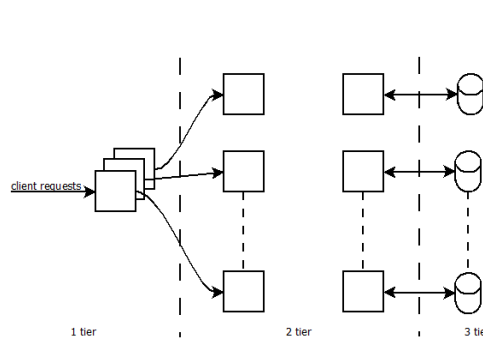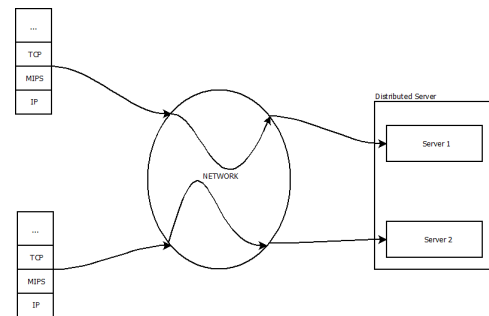- cookies allow to share information for server upon next visit client sends it'S cookies, allows state information for stateless server

Distributed Servers

- servers in different locations that have different ip-adresses in DNS under the same name
- MIPv6: mobility support for IPv6
- mobile node has home network with stable home adress (HoA)
- special router is home agent and takes care of traffic to the mobile node
- mobile node receives care-of-adress (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 $\Rightarrow$ performace
- (security)

Models of Migration

- or process model
    1. code segment, instructions
    2. resource segement, references to external resources, ie.e. file, printer, devices
    3. execution segement, execution state process, stack, private data, programm 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
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 reference, 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 operating system sends the message to the remote operating system.
4. The remote operating system 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 operating system.
8. The server's as sends the message to the client's operating system.
9. The client's operating system 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)
- transfered 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), represetation of integers (one's complement vs two's complement) or endianness (little endian vs. big endian)

Passing Reference Parameters

- extremly 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
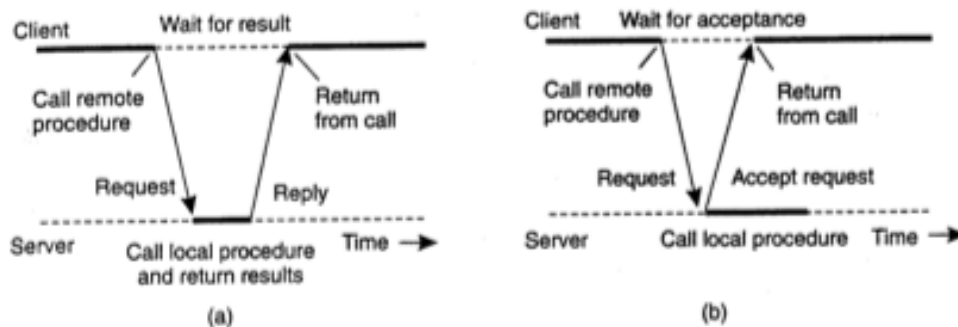
Figure 6.1: a: synchronous b: asynchronous RPC

- 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, vorrübergehend

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.
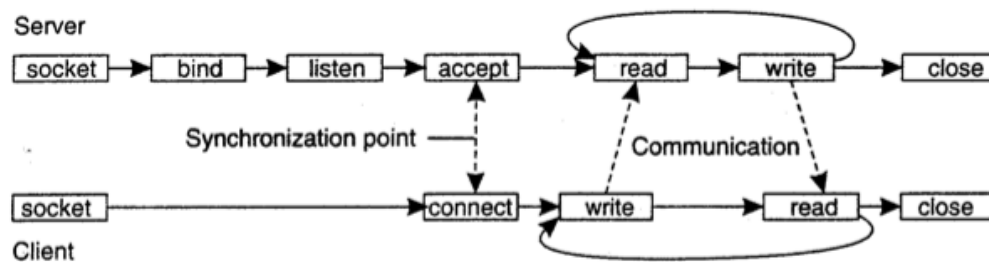
Figure 6.2: Connection-oriented communication pattern using sockets

- socket: create a new communication end point
- bind: attach a local addres to a socket
- listen: announce willingness to accept connections
- accept: block caller until a connection request arrives
- connect: actively attemt 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)

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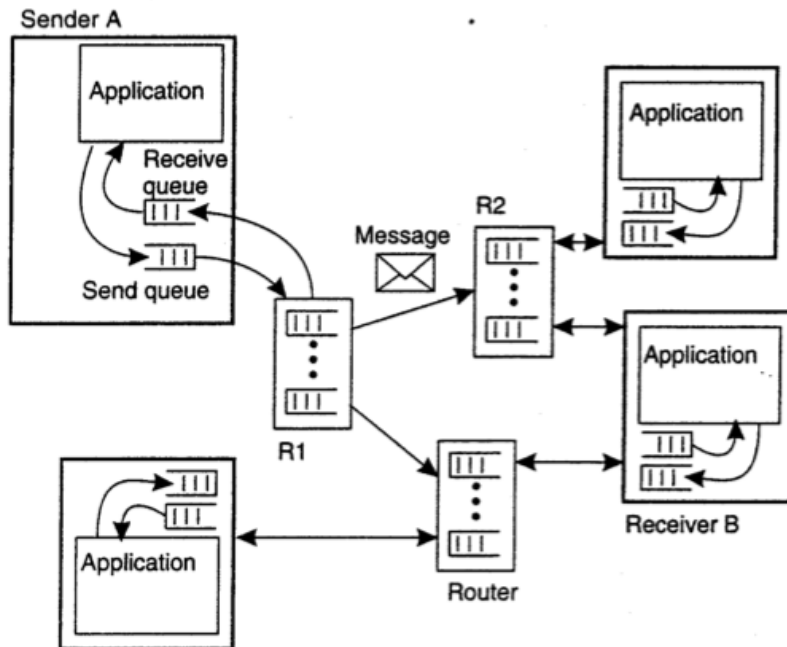
Figure 4·20. The general organization of a message-queuing system with routers.

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
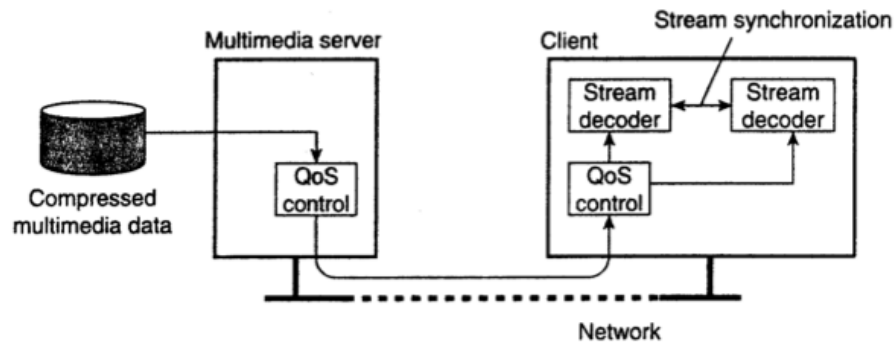
Figure 6.4: A general architecture for streaming stored multimedia data over a network

• form of communication in which timing plays a crucial role
• in continuous media, the temporal relationships between different data items are fundamental to correctly interpreting what the data actually means
• multimedia data will need to be compressed substantially in order to reduce the required network capacity
• simple stream: consists of only a single sequence of data
• complex stream: consists of several related (often time dependent) simple streams (substreams)
• QoS
    1. The required bit rate at which data should be transported.
    2. The maximum delay until a session has been set up (i.e., when an application can start sending data).
    3. The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
    4. The maximum delay variance, or jitter.
    5. The maximum round-trip delay.
• synchronisation of streams
    – Synchronization of streams deals with maintaining temporal relations between streams
    – Synchronization takes place at the level of the data units of which a stream is made up
    – In other words, we can synchronize two streams only between data units
    – Example: Playing a movie in which the video stream needs to be synchronized with the audio

6.5   Multicast communication

6.5.1   Application Level Multicasting

• sending data to multiple receivers
• For many years, this topic has belonged to the domain of network protocol

- With the advent of peer-to-peer technology various application-level multicasting techniques have been introduced
- The basic idea in application-level multicasting is that nodes organize into an overlay network, which is then used to disseminate information to its members
- connections between nodes in the overlay network may cross several physical links, and as such, routing messages within the overlay may not be optimal in comparison to what could have been achieved by network-level routing

Approaches to building the overlay

- tree: nodes may organize themselves directly into a tree, meaning that there is a unique (overlay) path between every pair of nodes
- mesh network: nodes organize into a mesh network in which every node will have multiple neighbors and, in general, there exist multiple paths between every pair of nodes
- mesh network generally provides higher robustness

Example: Construct overlay tree for chord

- node that wants to start multicast generates key 128bit/16obit (mid) randomly
- lookup of succ(mid) finds node responsible for key mid
  $\Rightarrow$ succ(mid) becomes root of tree
- join multicast: lookup (mid) creates lookup message with join request routed from P to succ(mid)
- request is forwarded by Q (first time forward), Q becomes forwarder
  $\Rightarrow$ P child of Q
- request is then forwarded by R, R becomes forwarder
  $\Rightarrow$ Q becomes child of R
- if Q or R is already forwarder: no forward
  $\Rightarrow$ Q becomes child of R
- multicast: lookup(mid) sends message to the root
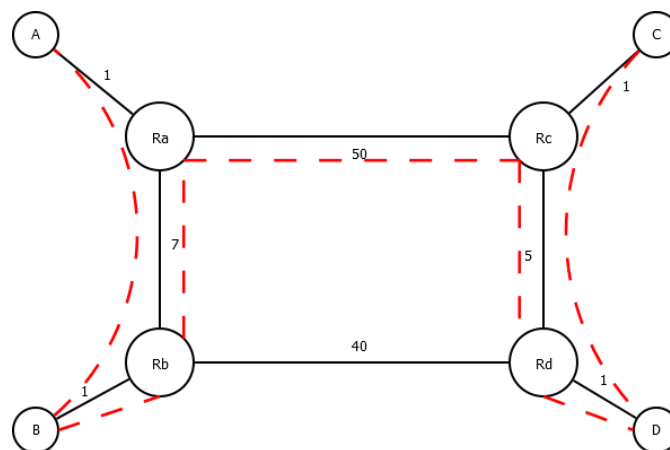 multicast from root

Figure 6.5: The relation between links in an overlay and actual network-level routes

Efficiency

- building a tree is not difficult once we have organized the nodes into an overlay
- building an efficient tree may be difficult
- The quality of an application-level multicast tree is generally measured by three different metrics
    1. Link stress: defined per link and counts how often a packet crosses the same link
    2. Stretch / Relative Delay Penalty (RDP)
       ratio in the delay between two nodes in the overlay, and the delay that those two nodes would experience in the underlying network: $\frac{\text{transmission time in overlay}}{\text{transmission time in delay/network}}$
       $\Rightarrow$ minimize aggregated stretch, average RDP over all note pairs
    3. tree cost: global metric, generally related to minimizing the aggregated link costs
       link cost = cost between end points
       $\Rightarrow$ find minimal spanning tree

6.5.2 Gossip-based-communication

- epidemic behaviour: model information dissemination in a (large) distributed system after the spreading of infectious diseases
- infected node: a node that holds data that it is willing to spread
- susceptible: a node does not yet have the new data
- removed: an updated node that is not willing or able to spread its data
- we assume we can distinguish old from new data, for example, because it has been timestamped or versioned

Anti-entropy-model

A node P picks another node Q at random, and subsequently exchanges updates with Q. There are three approaches to exchanging updates:
   1. P only pushes its own updates to Q
       - updates can be propagated only by infected nodes
       - if many nodes are infected, the probability of each one selecting a susceptible node is relatively small
       - $\Rightarrow$ chances are that a particular node remains susceptible for a long period simply because it is not selected by an infected node
   2. P only pulls in new updates from Q
       - spreading updates is essentially triggered by susceptible nodes
       - high probability to to contact an infected node and pull in the updates
       - $\Rightarrow$ pull-based approach works much better when many nodes are infected
   3. P and Q send updates to each other (i.e., a push-pull approach)
       - 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

- if node P has just been updated for data item x, it contacts an arbitrary other node Q and tries to push the update to Q
- if Q was already updated by another node, P may lose (with some probability) interest in spreading the update any further

- P then becomes removed
-
- Fraction of nodes that never obtain data: $s = e^{-(k+1)(1-s)}$
  e.g. $k = 4, ln(s) = 4,97$
  $\Rightarrow s = 0,007$
  less than $0,7\%$ remain without data

Removing Data

- problem: deletion of a data item destroys all information on that item
  $\Rightarrow$ when a data item is simply removed from a node, that node will eventually receive old copies
  of the data item and interpret those as updates on something it did not have before
- $\Rightarrow$ record the deletion of a data item as just another update, and keep a record of that deletion

Chapter 7

CHORD: Distributed Hash Tables

Chapter in the Book:
- Naming
    - Flat naming
        * Distributed Hash Tables

- Chord uses an m-bit identifier space (128 or 160 Bit) to assign randomly-chosen identifiers to nodes as well as keys to specific entities
- An entity with key k falls under the jurisdiction of the node with the smallest identifier id $\geq$ k (referred to as succ(k)

7.1   Joining

- node p wants to join
- p requests lookup for $succ(p)$
    - actually, p creates SHA1 identifier id, e.g. from its IP, which is then looked up
- contact $succ(id)$ and $pred(id)$ to join ring
- create finger table for p either by calculating all the entries or by copying the finger table of $succ(p)$ an let the stabilization protocol do the rest

7.2   Leaving

- node p informs $succ(p)$ and $pred(p)$
- $p's$ data is assigned to $succ(p)$

7.3   Routing

- searching, lookung
- Resolve key k to address of $succ(k)$

7.3.1 Option 1: Naive Approach (Linear search)

- each node p keeps succ(p+1) and pred(p)
- each node forwards request for key k to a neighbor
- if pred(p) < k ≤ p, return(p)
  ⇒ not scalable

7.3.2 Option 2: Finger-table based lookup

- better solution: each Chord node maintains <u>finger table</u> of lenght m

$$\forall\, 1 \leq i \leq m : FT[i] = succ(p + 2^{i-1}) \mod 2^m$$

- the i-th entry points to the first node succeeding p by at least 2^{i-1}
- Example: Node 1: FT[1]=succ($p + 2^{i-1}$) = succ($1 + 2^{1-1}$) = succ($1 + 1$) = succ(2)
  (smallest id, sucht that id ≥ 2)
- to lookup key k, node p forwards request to node q with index j in $p's$ finger table:
  $q = FT_p[j] \leq k \leq FT_p[j+1]$
- lookup generally requires $O(log(N))$ steps, N nodes in system

7.4 Stabilization

- goals:
  - keep the finger tables up to date
  - make new nodes known to the rest of the network
  - preserve the structure of the ring
- two tasks, periodically run at each node
  1. fix_fingers
     - check (li.e. lookup) one, some or all entries in the the finger table
     - update the finger table accordingly
  2. stabilize
     - lookup $pred(succ(p))$ (should be p)
     - otherwise: update successor and predecessor information in p and the neighbors

Example 1: Routing

The example refers to Figure 5.4, p. 190, Tanenbaum.
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

Example 2: Superpeers

Assume a system for which k bits of an m-bit identifier space have been reserved for assigning to superpeers. If identifiers are randomly assigned, how many superpeers can one expect to have in an

N-node system?

Superpeers: 2^k, therefore 2^{m-k} normal nodes.

Probability that a node is a supernode: $n \cdot \frac{2^k}{2^m} = 2^{k-m} \cdot n$, 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

Synchronistation

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: $\forall\, p, t : C_p(t) = t$
    $\Longleftrightarrow C_p'(t) = \frac{dC_p(t)}{dt} = 1$
    $\widehat{=}$ frequency of clock C_p at time t

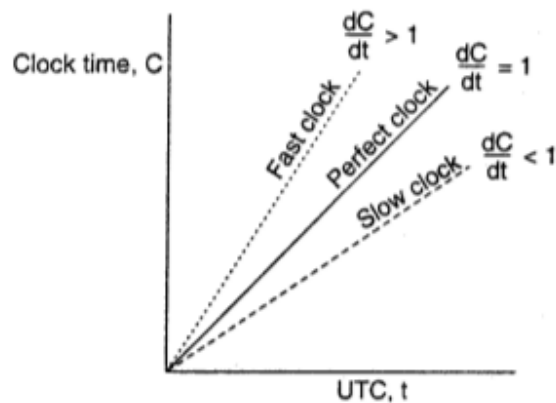Figure 6-5. The relation between clock time and UTe when clocks tick at different rates.

Figure 8.1: fast, slow & perfect clock

- $C_p'(t) - 1 \widehat{=}$ skew of p's clock, difference to perfect clock.
- $C_p(t) - t \widehat{=}$ offset
- real timers do not interrupt exactly H times per second
- maximum drift ρ is a constant guaranteed/specified by the vendor
    $1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$

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

  example: given values for C_1 and C_2:

$C_1 = \rho = 0.001$
$C_2 = \rho = 0.001$
$\frac{dC}{dt} = \frac{1}{2*0.001} = \frac{1}{0.002} = \frac{1}{2} * 10^3$
Clocks C_1 and C_2 need to be synchronized every $500s$ to keep time in the same second.

8.2   Network Time Protcol (NTP)

- nodes contact time server that has an accurate clock
- time server passive
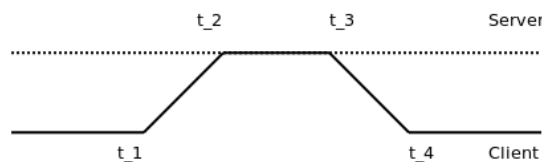
Figure 8.2: ntp

A estimates its offset to B as
$\Theta = \frac{(T_2-T_1)+(T_4-T_3)}{2}$
assuming communication time is symmetric
delay:
$\delta = (T_4 - T_1) - (T_3 - T_2) = (T_2 - T_1) + (T_4 - T_3)$

- A probes B, B probes A
- NTP stores 8 pairs (Θ, δ) per node pair using $\min(\delta)$ for smallest delay
- either A or B can be more stable
- reference node has <u>stratum 1</u> (clock has stratum 0) (stratum = # Servers to a reference clock)
- lower stratrum 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.
- needs 'happens before' relation a $\to$ b

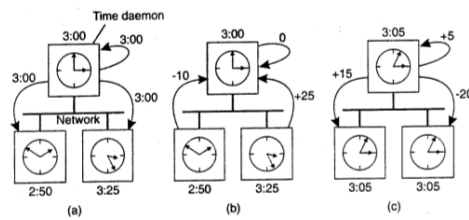Figure 8.3: Berkley Clocks: (a) The time daemon asks all the other machines for their clock values (b) The machines anwer. (c) The time daemon tells everyone how to adjust their clock.
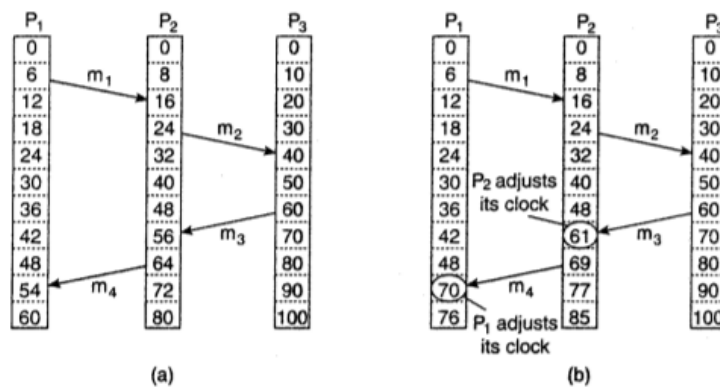

Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

Figure 8.4: Lamport's clock

1. if a,b are events in the same process and a happens before b, than 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 which one comes first)
- $\forall$ events a, we can assign it a time $C(a)$ on which all processes aggree.
- if $a \rightarrow b$ then $C(a) < C(b)$
  if $C(a) < C(b)$ then not $a \xrightarrow{?} b$
  if $C(a) \not< C(b)$ then $a \nrightarrow b$

- 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 then t.

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 $C_i \leftarrow C_i + 1$
2. When process P_i sends message m to P_j it sets the timestamp $ts(m)$ of m to the current time $ts(m) \leftarrow C$.
3. upon receipt of a message m process P_j adjust its time to $C_j \leftarrow \max\left(C_j, ts(m) + 1\right)$

8.4.1   Totally ordered multicast
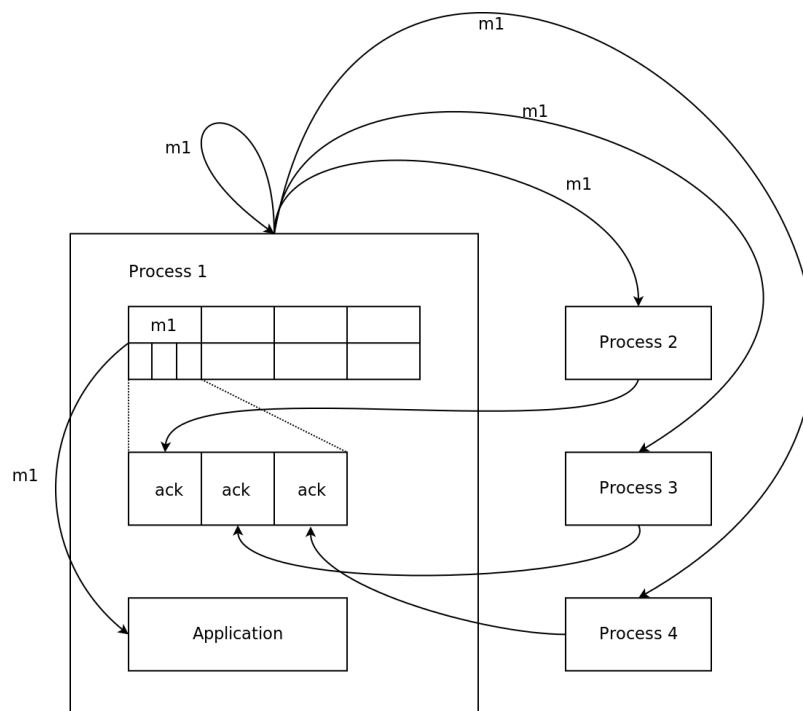
Figure 8.5: Totally Ordered Multicast

Example

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?

Algorithm

- 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

38

- 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 msgs 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 from every process

8.5 Vector Clocks

- Problem with Lamport: does not capture causality
- By construction, we know that for each message $T_{sent}(m_i) < T_{recv}(m_i)$. But what can we conclude in general from $T_{recv}(m_i) < T_{sent}(m_j)$ [Lamport]
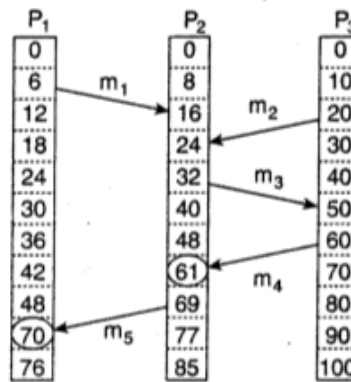

Figure 6-12. Concurrent message transmission using logical clocks.

- 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 occured so far at P_i
       $VC_i[i]$ is the logial clock at P_i
    2. if $VC_i[j] = k$ then P_i knows, that k events have occured 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 $VC_i[i] = VC_i[i] + 1$
    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 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.

8.5.1 Causally-ordered multicast

- Causally-ordered mulsticast is weaker than totally ordered multicast
- delivery of message m from P_i to P_j to application layer will be delayed until:
    1. $ts(m)[i] = VC_j[i] + 1$
       = m is the next message that P_j was expecting from process P_i
    2. $ts(m)[k] \leq VC_j[k] \ \forall \, k \neq i$
       = 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.
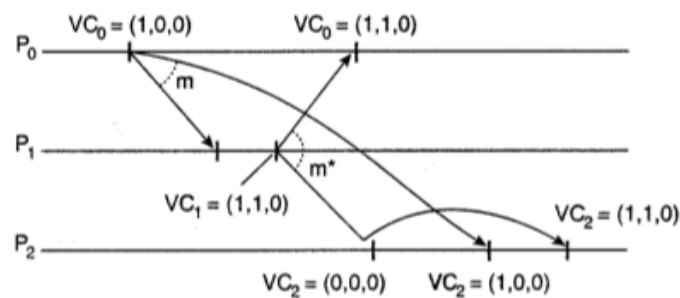
Figure 6-13. Enforcing causal communication.

8.5.2 Thoughts on causality and ordering

- causal ordering and (total) ordering are not the same
- causal ordering asks, which causal relationship the ordered items have
- lamport clocks and vector clocks both implement partial causal ordering
- vector clocks implement a stronger form of partial causal ordering (Tanenbaum p. 248 speaks here of total ordering, WP speaks of partial ordering)
- both can be forced to adhere to total ordering of events by implementing a mechanism deciding the ordering of concurrent events (if you do so, the total ordering then does not necessarily imply a causal ordering)

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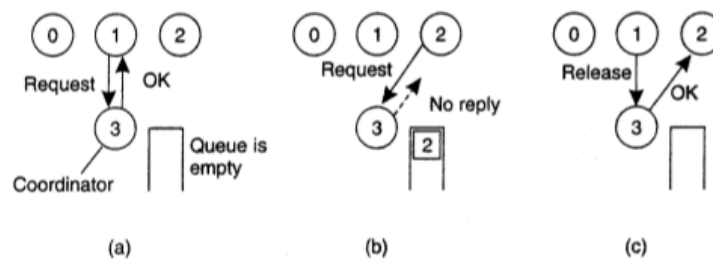
Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to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

- one process is coordinator
- coordinator alloes access onl 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

- Let m be the count of the coordinators, that answered and allowd entry to the critical section

- 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 Δt
  $P[k] = prob\{k$ out of m coordinators reset during $\Delta t\} = \left(\binom{m}{k}\right) p^k (1-p)^{m-k}$
- at least $2m - n \geq n + 2 - n = 2$ coordinators need to reset in order to violate the voting. This
  happens with probability $\sum\limits_{k=2m-n}^{n} P[k]$
  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
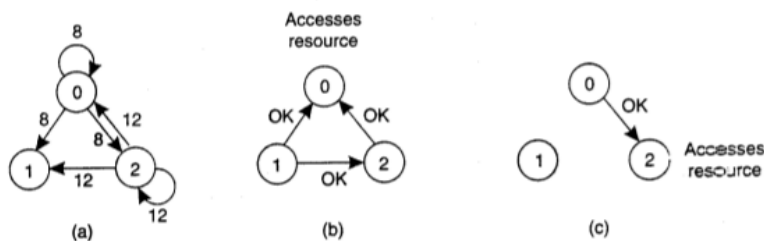
Figure 6-15. (a) Two processes want to access a shared resource at the same moment. (b) Process 0 has the lowest timestamp. so it wins. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

Problems:
- node 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

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

Problems:
- token loss -> regenerating?
- crashing nodes

9.5   Comparison

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

Chapter 10

Leader Election algorithms

10.1   leader election in a synchronous ring

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

- elected node is "leader"
- leader election is not possible for identical processes/nodes
- $\rightarrow$ processes have unique id (UID)

10.1.1   LCR algorithm

(Le Lann, Chang, Roberts)

- unidirectional communication
- ring size unknown
- only leader produces output
- algorithm compares UID
- One or more p_is can take the initiative and start an election, by sending an election message containing their id to p_{i+1}

```
1  For each node
2    u = 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
```
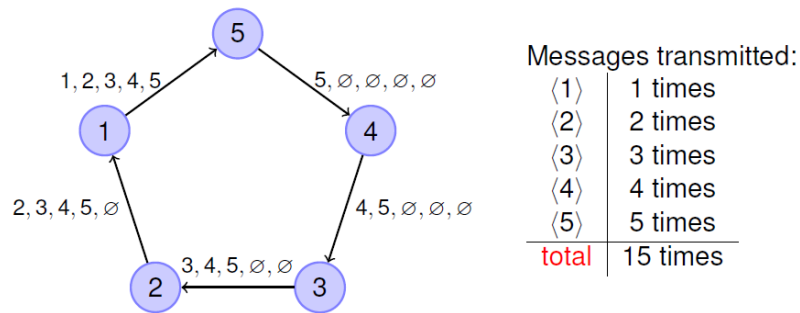
Figure 10.1: LCR algorithm

Correctness

Let max index of process with $max(UID)$ 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 status$_{max}$=leader

and

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

$$send_{max} = u_{max}$$

find UID at distance r from i_{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, but 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 ∈E{unknown, leader} initailly unknown phase ∈ N: initially o
6
7  message generation
```

45

```
 8    send current send+ to process i+1
 9    send current send− to process i−1

11  state transitions
12    send+=NULL
13    send−=NULL
14    if message from (i−1) is (v, out, h) then
15      v>u ∧ h>1: send+ = (v,out,h−1)
16      v>u ∧ h=1: send− = (v,in,1)
17      v=u status = leader
18    if message from i+1 is (v, out, h) then
19      v>u ∧ h>1: send− = (v,out, h−1)
20      v>u ∧ h=1: send+ = (v,in,1)
21      v=u status=leader
22    if message from i−1 is (v,in,1) ∧v ≠ u then
23      send+=(v,in,1)
24    if message from i+1 is (v,in,1) ∧v ≠ u then
25      send−=(v,in,1)
26    if both messages from i−1 and i+1 are (u,in,1) then
27      phase++
28      send+=(u,out, 2^phase)
29      send−=(u,out, 2^phase)
```

Complexity

Total number of phases is at most $8n(1 + \lceil \log(n) \rceil)$
the total number of messages is at most in $(1 + \lceil (\log(n)) \rceil) \approx \mathcal{O}(n \log n)$
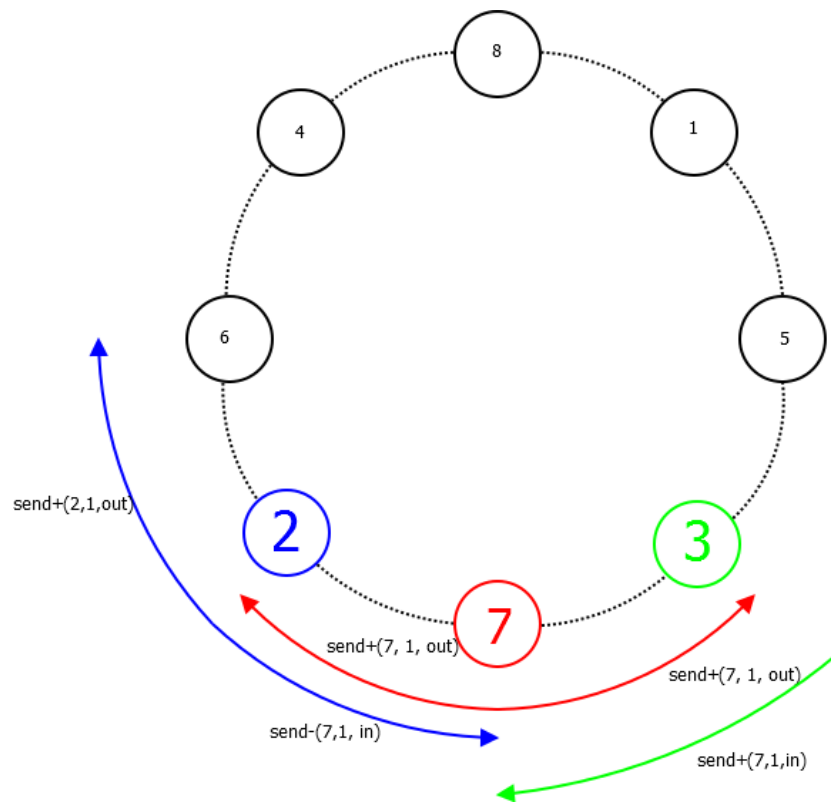Total time complexity is at most $3n$ if n power of 2 otherwise is $5n$
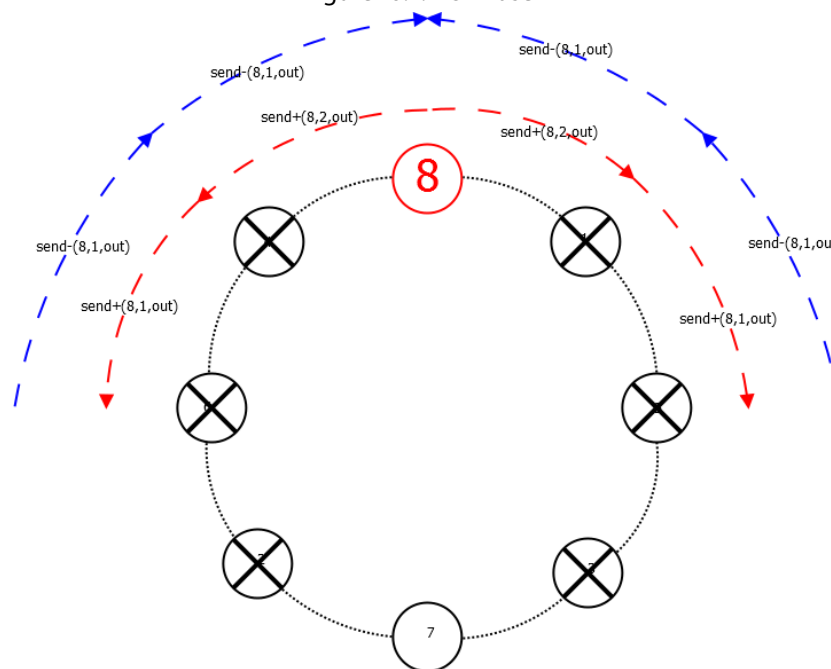
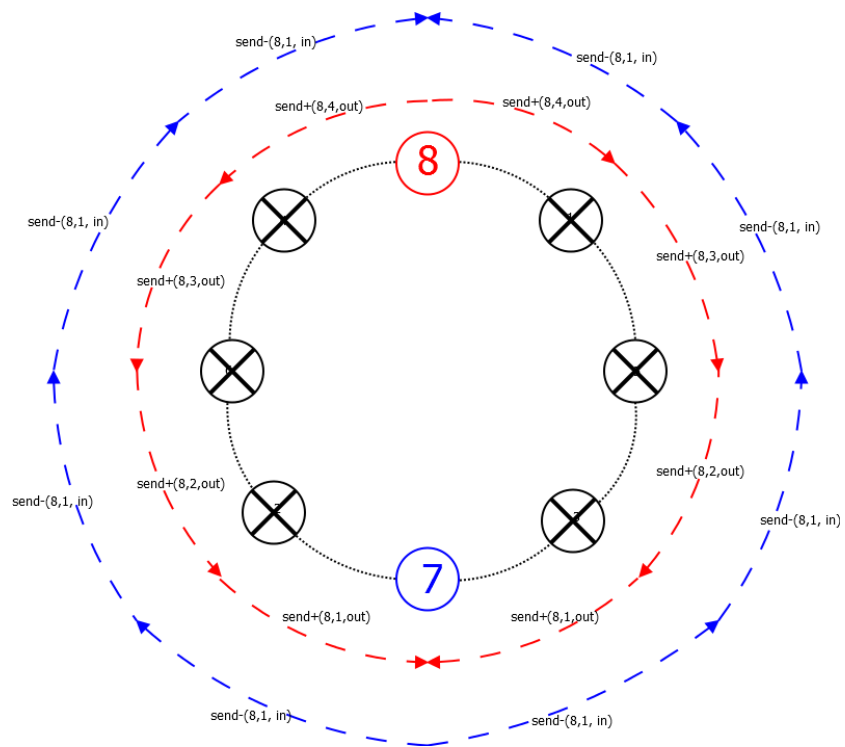Figure 10.2: HS Phase 1
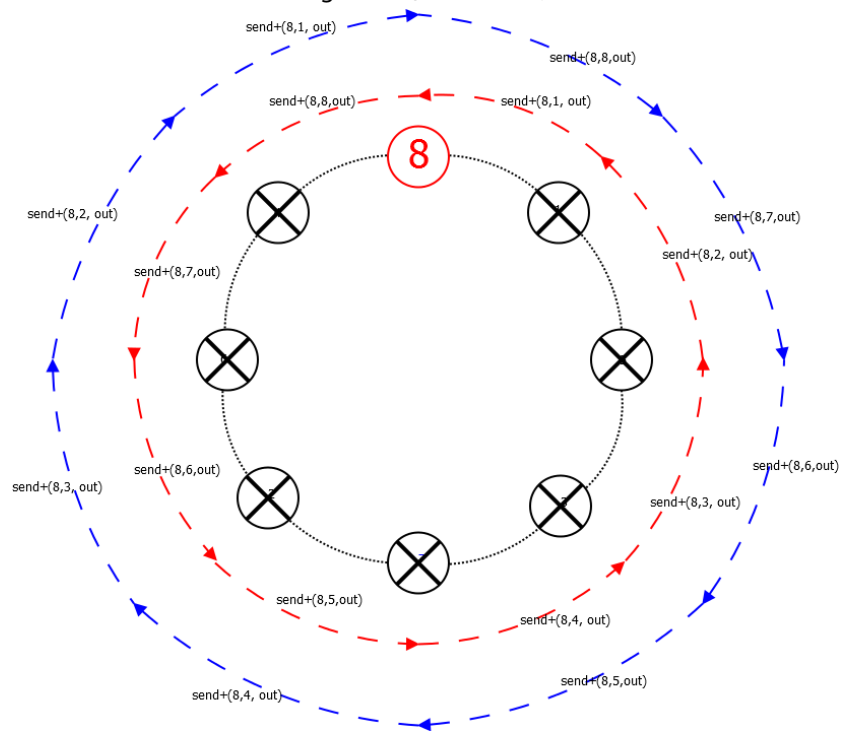

Figure 10.3: HS Phase 2

Figure 10.4: HS Phase 3

Figure 10.5: HS Phase 4

Show that the number of messages sent by the HS is at most $8n(1 + \lceil logn \rceil)$**, which means it is** $O(nlog(n))$**.**

In Phase k können maximal $4 \cdot 2^k$ Nachrichten (in und out) für einen Kandidaten (Knoten, der in seiner Nachbarschaft keinen größeren Knoten hat) existieren.

In Phase $k = 0$ gibt es n Kandidaten $\Rightarrow 2n$ out Nachrichten und maximal $\frac{n}{2} \cdot 2$ in Nachrichten $= 3n$

In Phase $k \geq 1$ gibt es maximal noch $\left\lfloor \frac{n}{2^{k-1}+1} \right\rfloor$ Kandidaten. $2^{k-1} + 1$ ist der minimale Abstand zwischen zwei Gewinnern aus Phase $k - 1$

$\Rightarrow$ die Anzahl der Nachrichten, die in Phase k gesendet werden ist $4 \cdot 2^k \cdot \left\lfloor \frac{n}{2^{k-1}+1} \right\rfloor = 8n \cdot \left\lfloor \frac{2^{k-1}}{2^{k-1}+1} \right\rfloor$

Offensichtlich ist die maximale Anzahl der Phasen ist $\lceil logn \rceil + 1$

Offensichtlich werden in der letzten Phase $2n$ Nachrichten gesendet

$$3n + \sum_{k=1}^{\lceil logn \rceil - 1} \left(8n \cdot \left\lfloor \frac{2^{k-1}}{2^{k-1}+1} \right\rfloor \right) + 2n \leq 5n + 8n \cdot (\lceil logn \rceil - 1)$$

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 the leader
       and sends a token with it's UID
```
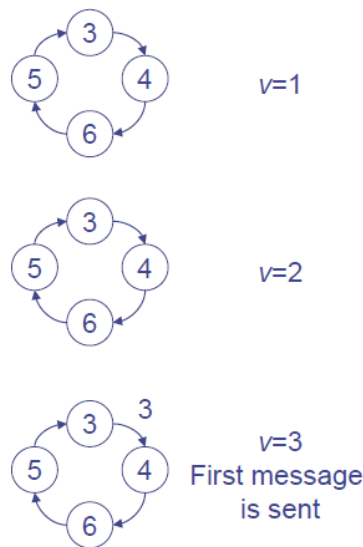

Figure 10.6: Timeslice algorithm

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 differrent speed
- token carrying UID v travels 1 messages every 2^v rounds
- each process memorises smallest UID
- return to origin elects UID

Complexity

…

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

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

10.2 Leader election in a wireless environment

- consider time needed for communication
- every node can initiate an election
- the select is based on information like battery lifetime or processing power
- the nodes only participate in one election, also there are more than one initiating nodes

Process to choose a leader:

Look at figure 10.7

1. one node starts the election and sends ELECT to its neighbours
2. the node, which sended the ELECT message becomes the parent of the node
3. the node sends the request to all neighbours except the parent node
4. the node acknowledges the parent after all the children acknowledged the election
5. if a node recieves an ELECT message, but has already a parent, it responses immediately, that another node is its parent
6. if a the node has only neighbours, which are the parent or have other parents, the node becomes a leaf and sends back its value
7. the waiting nodes send the node id and the highest value of the childs to the parent, after all acknowledged
8. in the end, the initiating node knows the node with the highest value and broadcast to all nodes, that this is the leader
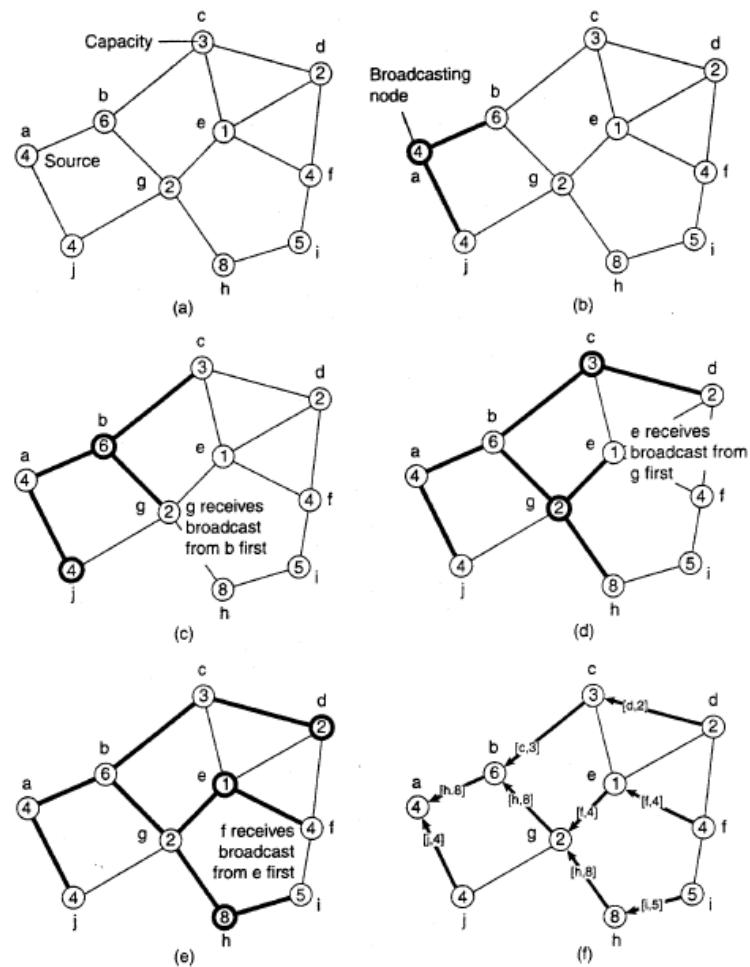
Figure 6.22. Election algorithm in a wireless network, with node *a* as the source.
(a) Initial network., (b)-(e) The build-tree phase (last broadcast step by nodes f
and i not shown). (f) Reporting of best node to source.

Figure 10.7: Election in wireless networks

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

- process P holds election
   1. P sends ELECT message to all processes with higher number
   2. P wins if there is no one responses ⇒ P is leader and sends COORDINATOR message to all available nodes.
   3. if Q (with higher number) answers with OK, Q takes over and sends ELECT to all higher nodes again.
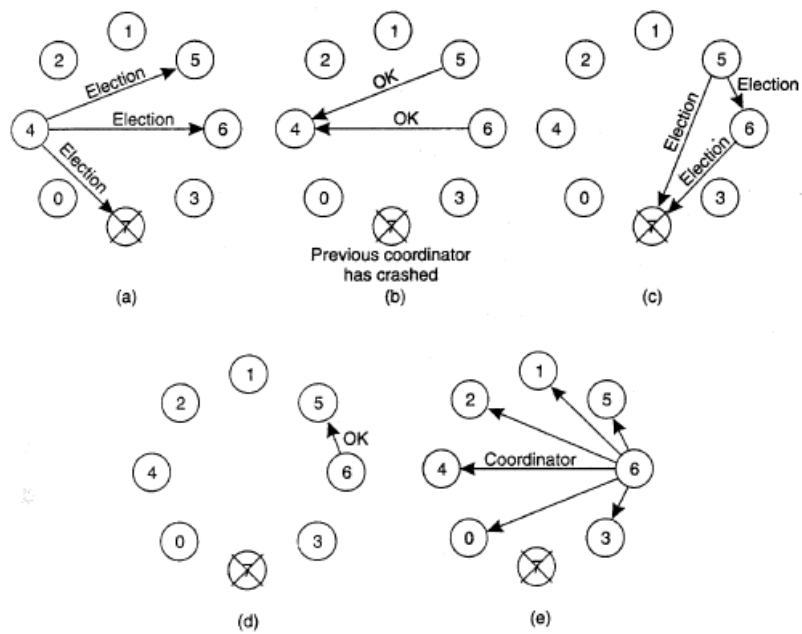
Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond. telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Figure 10.8: The bully election algorithm

Chapter 11

Consistency and Consensus

Anm.: Quelle ist hier der deutsche Tanenbaum/Steen, daher sind manche englischen Begriffe evtl. nicht korrekt gewählt.

- 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-constistency
    - client-centric-consistency
    - monotic reads: successive reads return the same or newer value
    - monitic 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
- Object-replication

11.1 Data-centric consistency

The more strict a model is, the stronger are the assumptions, that may be drawn from a series of events, but the harder to implement.

11.1.1 Strict consistency

Every read of x returns the value of the last write on x

Example strict consistency

P1: W(x)a

_____

P2:   R(x)a

Non strict consistent example

P1: W(x)a
_____
P2:   R(x)Null R(x)a

  Simple and clear idea, but this definition requires existence of global time (hard to realise), so that
the *last* write on a variable is unambiguous. This requirement is very hard to accomplish.
  If a data storage fulfills this requirement, then every write must be visible globally (for every process
in the system).

11.1.2   Sequential/linear consistency

Weak version of strict consistency, irrelevance of the chronological order of write events. That means,
that there is no global time or chronology, but some order of the events is globally visible and the
same for every process. Linear consistency just adds the requirement, that the events are ordered by
a globally (mayhaps ambiguous) timestamp.

Example sequential consistency

P1: W(x)a
_____
P2:   W(x)b
_____
P3:   R(x)b   R(x)a
_____
P4:   R(x)b R(x)a

Example non-sequential consistency

P1: W(x)a
_____
P2:   W(x)b
_____
P3:   R(x)b   R(x)a
_____
P4:   R(x)a R(x)b

11.1.3   Causal consistency

Weak version of sequential consistency, only causally dependent events must appear strictly ordered
on a global level. Two causally dependent events may be a write event *W(x)a*, a read event *R(x)a* fol-
lowed by a write *W(x)b*, then the write *W(x)b* may be based on the value writte in *W(x)a*.

Example causal consistency

P1: W(x)a			W(x)c		

P2:	R(x)a	W(x)b			

P3:	R(x)a			R(x)c	R(x)b

P4:	R(x)a			R(x)b	R(x)c

Example non-causal consistency

P1: W(x)a		

P2:	R(x)a	W(x)b

P3:		R(x)b	R(x)a

P4:		R(x)a	R(x)b

11.1.4 FIFO-consistency

Weak version of causal consitency. The writes of a single process (which are ordered and causally dependent) appear in this exact order globally, but the sequence of writes between multiple processes is not strictly ordered. In other words, only the casual dependencies inside one process appear in the correct order, dependencies between different processes may be mixed.

Example FIFO-consistency

P1: W(x)a					

P2:	R(x)a	W(x)b	W(x)c		

P3:			R(x)b	R(x)a	R(x)c

P4:			R(x)a	R(x)b	R(x)c

11.1.5 Weak consistency

The idea of weak consistency is, that also the relatively weak assertions of FIFO-consistency often are too hard to implement or just unnecessary. The globally visible actions are reduced to critical sections (or areas) where the actions become globally visible only after leaving the critical section, so only the results of the actions become visible. To realise that behaviour a mechanism to synchronise on the current state of a synchronisation variable S is introduced, such that processes who are interested in the current status of this variable may update their data, but other processes are not bothered with updates.

In this consistency scheme, consistency is not defined absolutely but only temporarily (at synchronization points).

Example weak consistency

P₁: W(x)a   W(x)b   S
_____

P₂: R(x)a   R(x)b   S
_____

P₃: R(x)b   R(x)a   S

Example non-weak consistency

P₁: W(x)a   W(x)b   S
_____

P₂: S   R(x)a

11.1.6   Release-consistency

In the weak-consistency-scheme there is no difference between synchronising after a write (i.e. a write-through to all replications) and synchronising before reading data, but these two scenarios require different actions on lower layers. Two additional procedures are added *acquire(..)* and *release(..)*.

11.1.7   Entry-consistency

Similar to release-consistency, but synchronisation variables (and procedures acq(..), rel(..)) are responsible for a designated set of variables, not for all variables in the global scope.

11.2   Client-centric consistency

In contrast to the data-centric consistency models client-centric consistency models do not try to maintain a system wide consistent view, but rather try to hide inconsitencies on client level.

11.2.1   Eventual consistency

(popular example: Domain Name Service) Usually big, distributed, replicated databases, which tolerate a high amount of incostency, but eventually become consistent when there are no updates to the database. It is important that clients access the system via fixed nodes, otherwise client updates may take a while, until they are visible for the client itself which is a clear violation of local consistency (read your writes). Remember that in this section the consistency view is only local/client-specific!

11.2.2   Monotone read-consistency

When a datum x is read by a process, then every following read of x results in the same value, or in newer value.

11.2.3 Monotone write-consistency

When a process writes to a datum x, then all previous write operations have to be finished/applied.

11.2.4 Read your writes

The result of a previous write operation of a process on a datum x is always visible in a following read of the same process

11.2.5 Writes follow reads

A write by a process to a datum x, which follows a previous read of x, is performed on this read value, or a newer one.

11.2.6 Implementing client-side consistency

The idea here is to monitor the client's reads and writes and add an ID to each operation. When reading/writing to a server the current status of the client (a list of the reads/writes) is added to the query and the server checks, wether the local storage of the client has to be updatet before executing the query.

11.3 Reliable multicast protocols

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

11.3.1 Distributed Commit

- an operatio 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
  - distributed transaction involves several processors each on a different machine
    2 phases with each 2 steps:

    1. coordinator $\xrightarrow{vote\ request}$ all participants
    2. participant $\xrightarrow[vote\ abort]{vote\ commit}$ coordinator
    3. if all commit
       coordinator $\xrightarrow{global-commit}$ all participants
       else
       coordinator $\xrightarrow{global-abort}$ all participants

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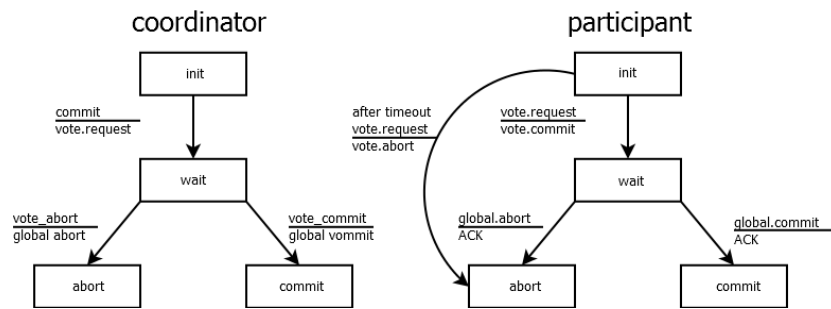   else participants locally abort

Figure 11.1: 2PC

5. two-phase commit (2PC) (Jim Gray, 1978)
   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 con 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
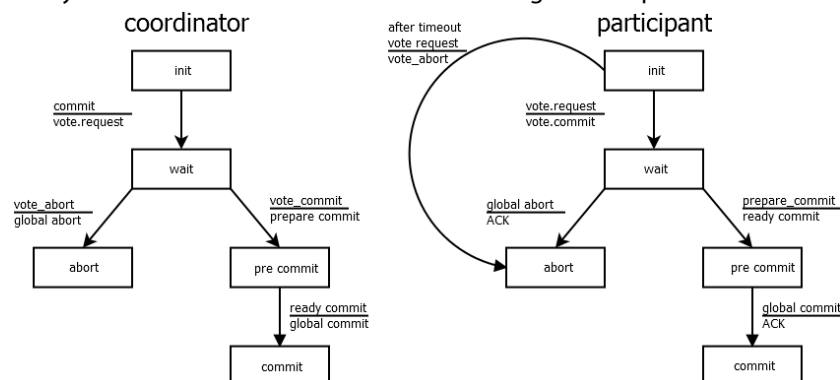 ⇒ necessary and sufficient conditions for non-blocking commit protocol

Figure 11.2: 3PC

58

- – abort branch as in 2PC
- – blocking states: paritcipant: init -> abort
  coordinator: wait –> abort
  precommit, knowing P voted for commit
  $\Rightarrow$ global-commit+recovery of P
  participant: ready
  coordinator failed as in 2PC
  precommit: contact other participants: if Q in precommit $\Rightarrow$ commit
  if Q is in init $\Rightarrow$ 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
  $\Rightarrow$ 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...