0.1 Verteilte Systeme/Distributed Systems

0.1.1 Orga

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

Elektisches

- (kvv)
- Website AG
- Sakai

Übungen

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

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

0.2 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

Transparency:
hide the fact that processes and resources are physically distributed.
Types of transparancy:

access hide differences in representation and how a resource is accessed

location

migration move ressources

relocation move ressources while using

replication

concurrency

failure

transparancy is desireable, but not always perfectly possible
tradeoff between transparancy 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 techiques

- use only asynchronous communication
- distribution, split components

- replication of components

pitfalls

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

Types of distributed systems

- computing systems
  - cluster computing

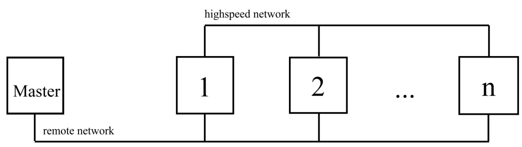

Abbildung 1: cluster computing

  - grid computing(virtual organisation, geographically distributed and heterogenous))
- distributed inforamtion 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

0.3 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

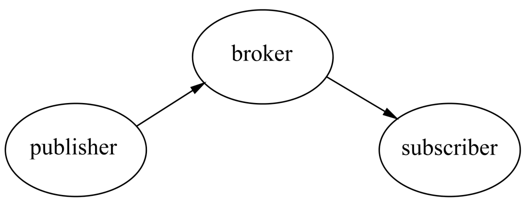

Abbildung 2: publish subsribe system

  - processes communicates threough 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
⇒ shared Data space

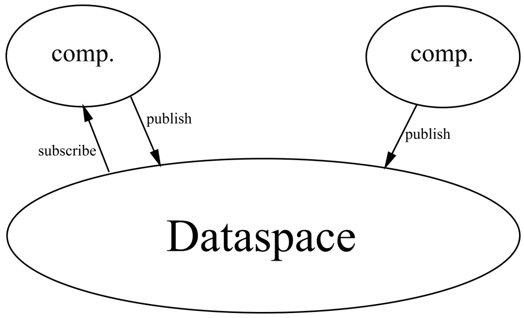

Abbildung 3: shared data space

0.3.1 System architectures

1. centralized architectures
   client - server

   (i) single point of failure

   (ii) performance (server is bottleneck)

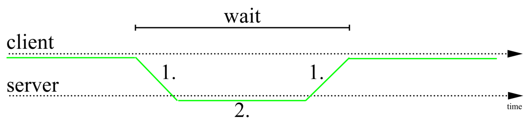

Abbildung 4: client server simple waiting situation

   (a) communication problems

   (b) server problems

   can request be repeated without harm?
   ⇒ request is idempotent

   (iii) aplication layering
   Layers:

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

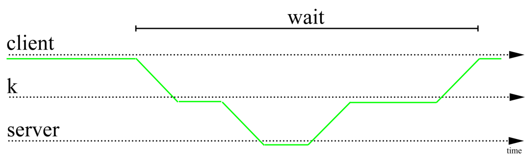

Abbildung 5: application layer

   ⇒ 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 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
       ⇒ 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

0.4 PeerSim

0.5 Processes

processes	**threads**
-execution of program	-several threads share CPU
-processor creates virtual processor	-thread context has little memory information, perhaps mutex lock
-for each program everyting is stored in process table	-threads avoid blocking application (e.g. spreadsheet,computation of dependent cells, intermediate backup)
-transparent sharing of resources,(processor, memory) separation	-thread switch is fast
-each virtual processor has it's own independent adress space	-user level threads allow parallel computation of program sections
-process switch is expensive, (save cpu context, pointers, translation lookaside buffer (TLB), memory management unit (MMU))	I/O or other blocking system calls block all threads, but thread creation/deletion is kernel task = expensive
-perhaps even swaps to disk, if memory exhausted	advantages of threads over processes vanishes

2 possible solutions:

1. scheduler activation, upcall to achieve process switch
2. light-weight processes (LWP)
   user level thread package
   execute scheduler and run thread of parent
   may block on systemcall, then other LWP may run
   triggered from userspace

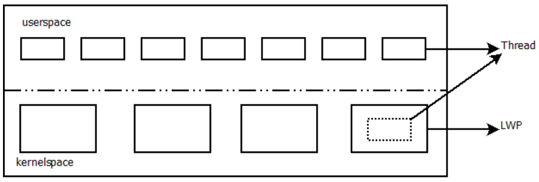

Abbildung 6: light-weight processes can run threads

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 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

0.5.1 Virtualisation

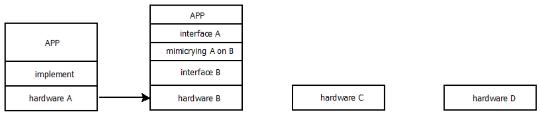

Abbildung 7: virtualisation

V pretends there are more resources then available.

Reasons for the need for V.
-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
   - 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 simultaneosly
   ⇒ VMM such as VMware, Xen

0.5.2 Client-/Serverprocesses

CLients:

- b) allows to store data at the server
- **thin client** e.g. X-windows
- thin client should separate application logic from user interaction
- ooften not implemented ⇒ 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

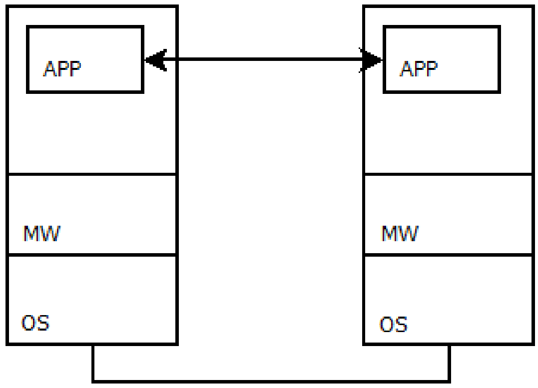

Abbildung 8: app specific communication

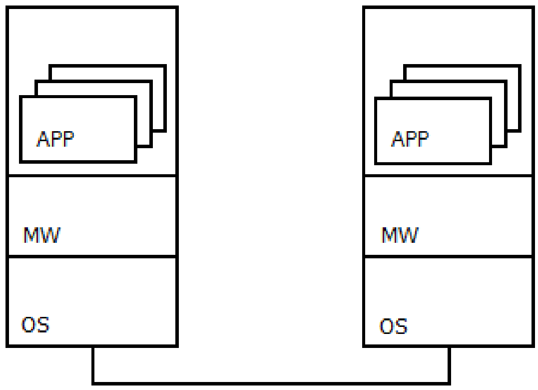

Abbildung 9: machine only communication

- 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

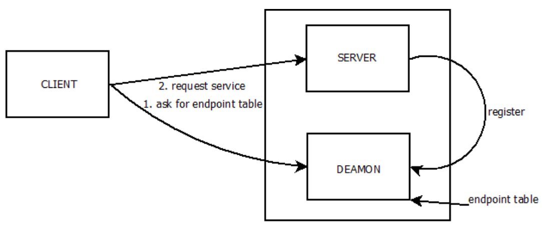

Abbildung 10: listener server

- superserver listens to several ports, replacinf several (mostly idle) servers
- stateless servers, keeps no information on state of client → 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

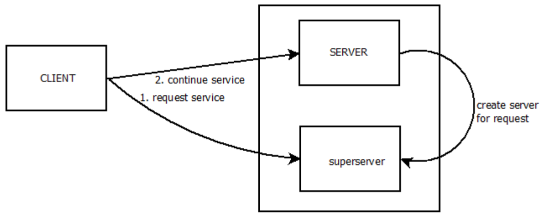

Abbildung 11: superserver

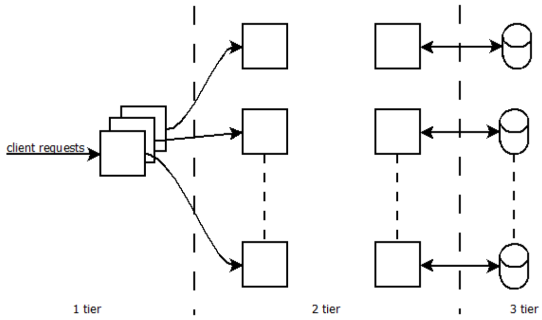

Abbildung 12: stateless server

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

Distributed Servers

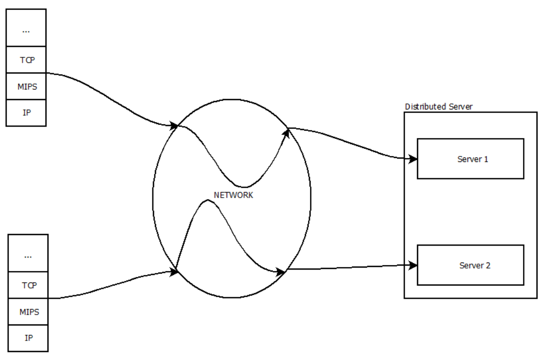

Abbildung 13: distributed server

- 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

0.5.3 Code Migration

- Code migration on (running) process - Why?

- service placement in distributed system ⇒ minimize communication cost
- load balancing in multiprocessor machine or cluster ⇒ 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

0.6 Communication

we skip networking → Telematik

Consider:

- Remote procedure call
- Message-oriented communication
- Strem-oriented communication
- Multicast communication

0.6.1 RPC

Remote procedure call uses stubs to pack parameters in message
value parameters
$\Rightarrow$ value packed in messages, transfer byte-by-byte$\Rightarrow$ problem can be little endian vs big endian systems
reference parameters are: extremely difficult; create array and pass by value; how to handle graps, linked lists. . .

Remote procedure calls

0.6.2 Asynchronous RPC

- hide communication, communication transparency

0.6.3 Message oriented communication

- avoids synchronous communication which blocks sender Berkely sockets UNIX TCP/IP
  server |socket|->|bind|->|listen|->|accept|->|read|<->|write|->|close|
  client |socket|——————————->|connect|->|read|<->|write|->|close|

0.6.4 Message-passing-interface (MPI)

- standad for communication
- communication within group of processes
- each group/member has identifier

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

- asynchronous persistent communication

- store messages
- transfer may take minutes, not milliseconds
- applications communicate by inserting messages into queues
- messages are inserted into local queue
- message carries destination adress
- queue manager may act as relays, router

- message broker transform type A into type B, using a set of rules
- applications are email, workflow, batch processing, queries accross several databases

0.6.6 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

0.6.7 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
  $\Rightarrow$ 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
  $\Rightarrow$ P child of Q
- request is first time forwarded by R, R becomes forwarder
  $\Rightarrow$ 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)
   $\frac{\text{transmission time in overlay}}{\text{transmission time in delay/network}} \Rightarrow$ minimize aggregated stretch, average RDP over all note pairs
3. tree cost, minimize aggregated link cost, link cost = cost between end points
   $\Rightarrow$ find minimal spanning tree

0.6.8 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 probabiltry 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: $s = e^{-(k+1)(1-s)}$
e.g. $k = 4, ln(s) = 4,97$
$\Rightarrow s = 0,007$
less than $0,7$ remain without data
removing data is difficult : delete message is send via gossiping

0.7 Naming

Flat naming

0.7.1 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
  ⇒ 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) $\leq$ k $\leq$ p, return(p)
  ⇒ not scalable
- better solution: each Chord node maintains finger table of lenght m
  FT[i]=succ($p + 2^{i-1}$) = succ($p + 1$) = succ(2) (smallest id, sucht that id $\geq$ 2)
  i-th entry points to 2^{i-1} ahead of p
- to lookup k node p forwards request to p with index j in ps finger table:
  $q = FT_p[j] \leq k \leq FT_p[j + 1]$

- 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
  - keeeping figer table up to date is expensive

0.8 Synchronistation

0.8.1 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: $C_p(t) = t \forall p, t$
  $\iff C'_p(t) = \frac{dC_p(t)}{dt} = 1$
  $\widehat{=}$ frequency of clock C_p at time t
- $C'_p(t) - 1 \widehat{=}$ skew of p's clock, difference to perfect clock.
- $C_p(t) - t \widehat{=}$ offset
- real timers do not interrupt H timespers maximum drift p such
  $1 - \rho \leq \frac{d(H)}{dt} \leq 1 + p$
- at time δt two clocks that are drifting apart can be
  $|C_2(\Delta t) - C_1(\Delta t) \leq 2\rho\Delta t|$
- if the difference should never exceed δ_i then synchronisation every $\frac{\delta}{2\rho}$ seconds is needed
- time allways moves forward.

0.8.2 Network Time Protcol (NTP)

- nodes contact time server that has an accurate clock
- time server pasive
  A estimates its offset to B as
  $\Theta = T_3 - \frac{(T-2-T_1)+(T_4-T_3)}{2}$
  assuming communication time is symmetric
  delay:
  $\delta = \frac{(T-2-T_1)+(T_4-T_3)}{2}$
- A probes B, B probes A
- NTP stores 8 pairs (Θ, δ) per node pair using min(δ) for smallest delay
- either A or B can be more stable
- reference node has strattime 1 (clock has starttime 0)
- lower starttime level is better, will be used.

0.8.3 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

0.8.4 Logical Clocks - YEAH ALP5! -.-

- logical time need not correct in real time.
- needs 'happens before' relation a → b
- happens before means:
  1. if a,b are events in the same process and a happens before b, than a → 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 → (Anmerk. von Tobi: is true?)
- happens before is transitive:
  $a \to b \wedge b \to c \Rightarrow a \to c$
- concurrency:
  if x, y happen in different processes and neither $x \to y$ nor $y \to x$, then x, y are concurrent (which means, it is not know who comes first)
- if $a \to b$ then $C(a) \to C(b)$
- 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 sendevent is strictly smaller than receive event for the same message
  4. for any $t \in T$ only finetely many events get assigned logical times smaller then t.
- Examle:

Algorithm

1. Before eacht event P_1 executes
   $C_i \leftarrow C_i + 1$
2. When Process P_i sends message m to P_j it sets the timestamp of $m, ts(m)$ to the current time $ts(m) \leftarrow C$.
3. upon receipt of a message m process P_j adjust its time to $C_j \leftarrow \max C_j, ts(m)$, then executes step 1 and delivers message

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? Totally ordered multicast

every message is sent to all receivers+itself with timestamp

- messages are stored in queues and acknowledged by timestamp
- queues are Lamports logical clocks
- eventually all queues are identical ⇒ total order

0.9 Vector Clocks

- Lamport's logical clock causally order
- $T_{sent}(m_i) < T_{recv}(m_i)$ does $T_{recv}(m_i) < T_{sent}(m_j)$ tell something about m_i, m_j
  use Vector Clocks
- each process P maintains VC
  1. $VC_i[i]$ is I of events that occured so far at P_i VC in the logial clock of P_i
  2. $VC_i[j] = k$, P_i stores k events at P_j.
     useful for causally ordered multicast

0.10 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

1. Centralised algorithm
   - 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
2. Decentralised 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 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

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:

     (a) returns OK, if does not want a resource
     (b) queues request, if it has resource
     (c) compares timestamps, sends OK if timestamp is smallest, queues request and sends no reply else

   - grants mutual exclusion without deadlocks or starvation

   Problems:

   - note failure ⇒ 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

4. Token Ring Algorithm

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

   Problems:

   - token loss
   -

Comparison

Alogirithm	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?