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
    $\Rightarrow$ 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	
relocation	
replikation	
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
  - 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
  - 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

0.3.1 System architectures

1. centralized architectures
   client - server
   (i) single point of failure
   (ii) performance (server is bottleneck)
   can request be repeated without harm?
   ⇒ request is idempotent
   (iii) aplication layering
   Layers:
   1.) User interface
   2.) processing
   3.) data level
   ⇒ 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
  $\Rightarrow$ 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

0.4 PeerSim

0.5 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

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

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

0.6 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