Distributed Systems

Aims:

- *Resource Sharing*: costly resources (high-quality printers and expensive programs) can be shared
- *Computation Speedup*: Algorithms can run concurrently
- *Reliability*: Failure of one site does not imply failure of the whole system
  Redundancy prerequisite

Especially last point difficult to fulfill

Design Issues

Several levels of Distribution possible
listed from tightly coupled to loosely coupled

- Shared memory
- Shared file system
- Bus Systems
- Switching Systems

Key Problem Areas:

- Transparency: pretend to be one computer
- Reliability: want availability, fault tolerance
- Performance
- Scalability: avoid centralised tables and algorithms

Communication

Simplest way of communication:
Client/Server Model

Idea: group of processes (servers) used by clients
Advantage: Simple communication
Simple enough to study several problems
First problem: Addressing! Possible Solutions:

- Hardware address into client code: inflexible
- Broadcasting: works only on local networks
- Name Servers: Ask special host
  Example: Domain Name Service, DNS

Second Problem:
Blocking (synchronous) vs. non-blocking primitives
Conceptual ease vs. performance

Third Problem:
Reliable vs. unreliable primitives
Where does the error correction go?
Kernel (once for all) vs. application (possibly more efficient)

Remote Procedure Call

Very simple idea: execute procedure on different host
Goal: total transparency

Basic Schema:

- Client sends arguments to server
- Server executes call
- Server sends results back

Difference to local call hidden in kernel routines
Details complicated:
- Have to transfer parameters
- deal with failures

Problems with parameter passing

- Different representation on different machines: either common format (inefficient), or store format in message
- What to do with call-by-reference parameters? Can copy arrays, but not arbitrary data structures

Failure problems more complicated

Have several cases

- Client cannot locate server: Generate exception
  $\Rightarrow$ transparency lost
- Lost Request Messages: use timer
- Lost Reply Messages: client timer insufficient:
  could execute operation more than once
  Solution: use sequence numbers
- Server Crashes: Sequence numbers not enough: When did crash occur?
  Can guarantee *at least once*, `at most once` semantics, but not *exactly once* semantics $\Rightarrow$ have to have call-specific remedies
- Client Crash: leaves orphans (unwanted computations)
  No general way of getting rid of them

Generating Timestamps

Unique timestamps needed for co-ordination
No problem in monoprocessor system: use system clock
Not possible in distributed systems
One way out:

- Each host maintains logical clock which is advanced with each event
- All message from host contain logical clock
- When message with greater logical clock is received, increase own logical clock to this value
- with two identical timestamps, let host number decide

Mutual Exclusion algorithms

A Distributed Version (Ricart and Agrawala)
Assume reliable messages, unique timestamps
Following steps:

- Process trying to enter critical section:
  sends to all other processes name of section and unique timestamp
- Process receiving such a message:
  - Sends back OK if not interested in critical section
  - Queues message if already in critical section
  - Receiver wants to enter critical section
    $\Rightarrow$ enters critical section if its request has lower timestamp and queues message, otherwise sends OK

Grants mutual exclusion without deadlock or starvation

Problems:

- Requires that everyone knows about all other hosts
- Algorithm fails if one host fails
  $\Rightarrow$ Reconstruction of network necessary in this case

Suitable for networks where configuration is stable

A Token Ring Algorithm

Assumption: Network organised on a (physical or logical) ring, i.e. each node has unique successor in line
Simple algorithm:

- At initialisation, generate token
- Pass token around continuously
- Process wanting to enter a critical region waits for token
- After leaving critical section, process passes token to next neighbour

Properties:

- Detecting lost tokens difficult: Time spent in critical region unbound
- Detecting dead processes easy if sent token is acknowledged

Election Algorithms

Problem: Select new co-ordinator

Assumption:

Know id of *every* host on the network

First example: Bully algorithm

- P sends message to all hosts with higher number
- No response $\Rightarrow$ P wins and is co-ordinator
- Answer received $\Rightarrow$ host with higher number has taken over

Second Example: Ring Algorithm

- any host may send *Election* message
- passed around the net, which each hostid added
- If original host gets message, determines co-ordinator and sends new message around
- After it has gone round, host removes it

Transaction Protocols

Need higher level of abstraction

Example: booking a flight:
Agreement on status of transaction necessary:
either completed or not happened at all

want to model this
Assumptions:

- reliable communication, but hosts may fail
  (lost messages handled by lower levels)
- Have stable storage surviving host crashes and disk failures

Properties of transactions

- *Atomic*: transaction indivisible
- *Consistent*: maintain system invariants
- *Isolated*: Concurrent transaction do not interfere
- *Durable*: Once transaction finished, changes are permanent

Implementation

Two methods used for working on data:

- *Private Workspace*:
  Copy files to host executing the transaction
  Copy results (or original files) back afterwards
- *Writeahead log*:
  modify files locally, but keep log of changes
  makes undo possible later

The Two-Phase Commit Protocol

Aim: Achieve atomicity
Requires central co-ordinator
Protocol works as follows:
(**C** co-ordinator, **S** subordinate)

- **C** writes "Prepare" in the log
- **C** sends "Prepare" message to subordinate
- **S** sends "Ready" message when it is happy to commit
- **C** collects replies
- **C** writes log record
- **C** sends "Commit" message
- **S** writes "Commit" in the log
- **S** Commit
- **S** sends finished message

Concurrency control

Need some way of serialising parallel events
First way: *File Locking*
Each host maintains list who is accessing files at the moment
Only one process allowed to access file
To avoid inconsistency:

- First aquire all locks (Growing Phase)
- Perform operations
- Release locks (Shrinking Phase)

Achieves serialisability
Deadlock possible:
Standard avoidance techniques:

- global order of files
- If lock not available, release all others and wait for random interval

Optimistic concurrency control
Check which files have been written
Any files written twice $\Rightarrow$ Abort transaction
Advantage: Deadlock free, allows maximum parallelism
Problem: Does not work with high load

Refinement: Use timestamps and abort only if transaction with higher timestamp wants to write

Dealing with failures

Distributed systems should cope with failure of one site, loss of messages etc.

Kinds of failures:

- Silent: host does not respond
- Byzantine: host sends false information

Failure detection done by handshaking protocol:

- Site A send "Are-you-up"-message to B
- B answers immediately "I am up"
- If answer not received within certain time, try again, possibly via a different route
- give up after fixed number of attempts

works for silent, not for Byzantine failures

Byzantine failures

Assumption: Processors faulty, communication reliable
Question: Can we achieve agreement between the working processors?
Possible under certain conditions (Lamport)
More than two-thirds of processors working properly
To get idea, consider 1 faulty, 4 processors in total.
Have following steps:

- Every processor sends to other one value of local variable
- All processors collect values received
- Processors pass on all values to all other processors
- Each processor decides by majority voting on values received

Distributed File Systems

Special Problems:

- *Naming:* identify files systemwide
- how are concurrent reads and writes executed?

Naming

Aims:

- *Location Transparency:*
  name does not give any hint on location
- *Location independence:*
  files can be moved without name being changed

standard approach: *Remote mounting*

Make remote file system available under local name

Achieves only location transparency

Latter difficult to achieve (requires name server)

Implementation Issues

Main issue: stateless vs. stateful servers
(Should server keep information about requests?)
Properties of stateless servers

- Fault tolerance
- No `open`/`close`-requests needed
- No problems with client crashes

Advantage of stateful servers

- Read ahead possible
- Idempotency easier
- File locking possible

Main problem: Cache consistency, especially for stateless servers

NFS

Idea: make file systems available on other hosts
Works on different architectures
$\Rightarrow$ Need well-defined protocol
RPC's used for this purpose

Stateless system
$\Rightarrow$ no `open`/`close` RPC's
Each RPC contains absolute address in file
Caching employed:

- Server does normal caching (no ill-effects)
- Client caches reads and writes
  $\Rightarrow$ obtain inconsistency
  Data sent to server only when
  - $>$ 8k written
  - file closed on client
  - timeout reached

  `open` on client checks server