Distributed Coordination

Prof. Nalini Venkatasubramanian

(includes slides borrowed from Prof. Petru Eles, lecture slides from Couloiris, Dollimore and Kindberg textbook, MIT course notes, slides and animations from UIUC CS425 Indranil Gupta)
From Distributed Operating Systems to Distributed Middleware

Manages a collection of independent computers and makes them appear to the users of the system as if it were a single computer.

**Multiprocessors**
- Tightly coupled
- Shared memory

**Multicomputers**
- Loosely coupled
- Private memory
- Autonomous

Parallel Architecture

- CPU
- Cache
- CPU
- Cache
- Memory

Distributed Architecture

- CPU
- Memory
- CPU
- Memory
- CPU
- Memory
Early Systems - Cluster of Workstations

- How to find an idle workstation?
- How is a process transferred from one workstation to another?
- What happens to a remote process if a user logs onto a workstation that was idle, but is no longer idle now?
- Other models - processor pool, workstation server...

Examples: Berkeley NOW Project, U Wisconsin CONDOR system
Cluster Computing

Loosely connected set of machines that behave as a unit
Design Issues (cont.)

- **Transparency**
  - Location transparency
    - processes,cpu’s and other devices, files
  - Replication transparency (of files)
  - Concurrency transparency
    - (user unaware of the existence of others)
  - Parallelism
    - User writes serial program, compiler and OS do the rest

- **Performance**
  - Metrics
    - Throughput, response time
  - Load Balancing (static, dynamic)
  - Communication is slow compared to computation speed
    - fine grain, coarse grain parallelism tradeoff

*Also: Scalability, Reliability, Flexibility, Heterogeneity, Security*
Design Elements

● Process Management
  ● Task Partitioning, allocation, synchronization, load balancing, migration

● Communication
  ● Two basic IPC paradigms used in distributed systems
    ● Message Passing (RPC) and Shared Memory
  ● synchronous, asynchronous

● FileSystems
  ● Naming of files/directories
  ● File sharing semantics
  ● Caching/update/replication
Remote Procedure Call

- Basis of early 2 tier client-server systems
- A convenient way to construct a client-server connection without explicitly writing send/receive type programs (helps maintain transparency).
- Ideas initiated by RFCs in the 70s
  - RFC 684 (Rick Schantz, BBN) - “A Commentary on Procedure Calling as Network protocol, 1975
  - RFC 707 - “A High level Framework for Network Based Resource Sharing”
- Landmark paper by Birrell and Nelson in 1980’s - CEDAR project at Xerox
- EDEN (OS), Emerald (PL), AnsaWARE (traders)
- SUN, DEC, IBM HP NEC
  - RPC projects…
- CORBA, Java RMI,
- Web Services

Implementing Remote procedure calls
Remote Procedure Calls (RPC)

- General message passing model for execution of remote functionality.
  - Provides programmers with a familiar mechanism for building distributed applications/systems
- Familiar semantics (similar to LPC)
  - Simple syntax, well defined interface, ease of use, generality and IPC between processes on same/different machines.
- It is generally synchronous
- Can be made asynchronous by using multi-threading
LPC vs. RPC

Host A
P1
main()
int f1()
int f2()

LPC

P2
int f2()

RPC

Host A
P1
main()
int f1()
int f2()

LPC

P2
int f2()

RPC request message
RPC reply message

Host B
RPC Challenges

Achieving exactly the same semantics for LPC and RPC is hard

- Disjoint address spaces
- Consumes more time (due to communication delays)
- Failures (hard to guarantee exactly-once semantics)
  - Function may not be executed if
    - Request (call) message is dropped
    - Reply (return) message is dropped
    - Called process fails before executing called function
    - Called process fails after executing called function
    - Hard for caller to distinguish these cases
  - Function may be executed multiple times if request (call) message is duplicated
RPC Needs: Syntactic and Semantic Transparency

- Resolve differences in data representation (CDR)
- Support multi-threaded programming
- Provide good reliability
- Provide independence from transport protocols
- Ensure high degree of security
- Locate required services across networks
- Support a variety of execution semantics
  - **At most once** semantics (e.g., Java RMI)
  - **At least once** semantics (e.g., Sun RPC)
  - Maybe, i.e., best-effort (e.g., CORBA)

<table>
<thead>
<tr>
<th>Retransmit request</th>
<th>Filter duplicate requests</th>
<th>Re-execute function or retransmit reply</th>
<th>RPC Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Re-execute</td>
<td>At least once</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Retransmit</td>
<td>At most once</td>
</tr>
<tr>
<td>No</td>
<td>NA</td>
<td>NA</td>
<td>Maybe</td>
</tr>
</tbody>
</table>
Implementing RPC - Mechanism

- Uses the concept of stubs; A perfectly normal LPC abstraction by concealing from programs the interface to the underlying RPC
- Involves the following elements
  - The client, The client stub
  - The RPC runtime
  - The server stub, The server
RPC – How it works II

client process

server process

client

procedure call

client stub

locate
(un)marshal
(de)serialize
send (receive)

communication module

server

procedure

dispatcher

selects stub

communication module

server stub

(un)marshal
(de)serialize
receive (send)
RPC - Steps

- Client procedure **calls** the client stub in a normal way
- Client stub **builds** a message and **traps** to the kernel
- Kernel **sends** the message to remote kernel
- Remote kernel **gives** the message to server stub
- Server stub **unpacks** parameters and **calls** the server
- Server **computes** results and **returns** it to server stub
- Server stub **packs** results in a message and **traps** to kernel
- Remote kernel **sends** message to client kernel
- Client kernel **gives** message to client stub
- Client stub **unpacks** results and **returns** to client
Different architectures use different ways of representing data

- **Big endian**: Hex 12-AC-33 stored with 12 in lowest address, then AC in next higher address, then 33 in highest address
  - IBM z, System 360
- **Little endian**: Hex 12-AC-33 stored with 33 in lowest address, then AC in next higher address, then 12
  - Intel

Caller (and callee) process uses its own *platform-dependent* way of storing data

Middleware has a common data representation (CDR)
- Platform-independent

Caller process converts arguments into CDR format
- Called “Marshalling”

Callee process extracts arguments from message into its own platform-dependent format
- Called “Unmarshalling”

Return values are marshalled on callee process and unmarshalled at caller process
How Stubs are Generated

● Through a compiler
  ● e.g. DCE/CORBA IDL – a purely declarative language
  ● Defines only types and procedure headers with familiar syntax (usually C)

● It supports
  ● Interface definition files (.idl)
  ● Attribute configuration files (.acf)

● Uses Familiar programming language data typing

● Extensions for distributed programming are added
RPC - IDL Compilation - result

development environment

client process

client code

language specific call interface

client stub

server process

server code

language specific call interface

server stub

IDL

IDL sources

IDL compiler

interface headers

Wolfgang Gassler, Eva Zangerle
RPC - binding

● Static binding
  ● hard coded stub
    ● Simple, efficient
    ● not flexible
  ● stub recompilation necessary if the location of the server changes
  ● use of redundant servers not possible

● Dynamic binding
  ● name and directory server
    ● load balancing
  ● IDL used for binding
  ● flexible
  ● redundant servers possible
RPC - dynamic binding

**client process**

- **client**
  - procedure call

- **client stub**
  - bind
  - (un)marshal
  - (de)serialize
  - 4 Find/bind
  - 7 send
  - 12 receive

**server process**

- **server**
  - procedure

- **server stub**
  - register
  - (un)marshal
  - (de)serialize
  - 11 register
  - 10 send
  - 9 receive

- **dispatcher**
  - selects stub

**Communication Module**

- 5
- 6
- 8
- 12

**Name and Directory Server**

*Wolfgang Gassler, Eva Zangerle*
RPC NG: DCOM & CORBA

- Object models allow services and functionality to be called from distinct processes.

- DCOM/COM+(Win2000) and CORBA IIOP extend this to allow calling services and objects on different machines.

- More OS features (authentication, resource management, process creation, ...) are being moved to distributed objects.
Sample RPC Middleware Products

- **JaRPC (NC Laboratories)**
  - libraries and development system provides the tools to develop ONC/RPC and extended .rpc Client and Servers in Java.

- **powerRPC (Netbula)**
  - RPC compiler plus a number of library functions. It allows a C/C++ programmer to create powerful ONC RPC compatible client/server and other distributed applications without writing any networking code.

- **Oscar Workbench (Premier Software Technologies)**
  - An integration tool. OSCAR, the Open Services Catalog and Application Registry is an interface catalog. OSCAR combines tools to blend IT strategies for legacy wrappering with those to exploit new technologies (object oriented, internet).

- **NobleNet (Rogue Wave)**
  - simplifies the development of business-critical client/server applications, and gives developers all the tools needed to distribute these applications across the enterprise. NobleNet RPC automatically generates client/server network code for all program data structures and application programming interfaces (APIs)— reducing development costs and time to market.

- **NXTWare TX (eCube Systems)**
  - Allows DCE/RPC-based applications to participate in a service-oriented architecture. Now companies can use J2EE, CORBA (IIOP) and SOAP to securely access data and execute transactions from legacy applications. With this product, organizations can leverage their current investment in existing DCE and RPC applications.
Some recent views on RPC

- Convenience over correctness?

- Time to retire RPC?

Video: Mythbusting Remote Procedure Calls

New messaging formats
  -- Google ProtoBuf, Apache Thrift (Facebook)
Protocol Buffer

- Designed ~2001 because everything else wasn’t that good those days
- Production, proprietary in Google from 2001-2008, open-sourced since 2008
- Battle tested, very stable, well trusted
- Every time you hit a Google page, you’re hitting several services and several PB code
- PB is the glue to all Google services
- Official support for four languages: C++, Java, Python, and JavaScript
- Does have a lot of third-party support for other languages (of highly variable quality)
- Current Version - protobuf-2.4.1
- BSD License
Apache Thrift

- Designed by an X-Googler in 2007
- Developed internally at Facebook, used extensively there
- An open Apache project, hosted in Apache's Inkubator.
- Aims to be the next-generation PB (e.g. more comprehensive features, more languages)
- IDL syntax is slightly cleaner than PB. If you know one, then you know the other
- Supports: C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages
- Offers a stack for RPC calls
- Current Version - thrift-0.8.0
- Apache License 2.0
Comparing messaging formats

JSON  🌟  Protobuf
RPC - Extensions

- conventional RPC: sequential execution of routines
- client blocked until response of server
- asynchronous RPC – non blocking
  - client has two entry points (request and response)
  - server stores result in shared memory
  - client picks it up from there
RPC servers and protocols...

- RPC Messages (call and reply messages)
- Server Implementation
  - Stateful servers
  - Stateless servers
- Communication Protocols
  - Request(R) Protocol
  - Request/Reply(RR) Protocol
  - Request/Reply/Ack(RRA) Protocol

- Idempotent operations - can be repeated multiple times, without any side effects
- Examples (x is server-side variable)
  - x=1;
- Non-examples
  - x=x+1;
  - x=x*2
Distributed Shared Memory (DSM)

Tightly coupled systems
Use of shared memory for IPC is natural

Loosely coupled distributed-memory processors
Use DSM – distributed shared memory
A middleware solution that provides a shared-memory abstraction.
Issues in designing DSM

- Synchronization
- Granularity of the block size
- Memory Coherence (Consistency models)
- Data Location and Access
- Replacement Strategies
- Thrashing
- Heterogeneity
Synchronization

- Inevitable in Distributed Systems where distinct processes are running concurrently and sharing resources.

- Synchronization related issues
  - Clock synchronization/Event Ordering (recall happened before relation)
  - Mutual exclusion
  - Deadlocks
  - Election Algorithms
Distributed Mutual Exclusion

- Mutual exclusion
  - ensures that concurrent processes have serialized access to shared resources - the critical section problem
- Shared variables (semaphores) cannot be used in a distributed system
  - Mutual exclusion must be based on message passing, in the context of unpredictable delays and incomplete knowledge
  - In some applications (e.g. transaction processing) the resource is managed by a server which implements its own lock along with mechanisms to synchronize access to the resource.
Requirements/Conditions

- **Safety Property (Mutual Exclusion)**
  - At any instant, only one process can execute the critical section.

- **Liveness Property (Progress)**
  - A process requesting entry to the CS is eventually granted it (as long as any process executing in its CS eventually leaves it).
  - Implies freedom from deadlock and starvation

- **Fairness (Bounded Waiting)**
  - Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.
Distributed Mutual Exclusion Techniques

● **Non-token Based Approaches**
  ● Each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control suite or by distributed agreement
    ● **Central Coordinator Algorithm**
    ● Ricart-Agrawala Algorithm (timestamp based)
    ● Lamport’s Algorithm (timestamp based)

● **Token-based approaches**
  ● A logical token representing the access right to the shared resource is passed in a regulated fashion among processes; whoever holds the token is allowed to enter the critical section.
    ● Token Ring Algorithm
    ● Ricart-Agrawala Second Algorithm
    ● Suzuki-Kazami Broadcast Algorithm

● **Quorum-based approaches**
  ● Need to coordinate with only some members in a group (quorum).
    ● Maekawa’s Algorithm -- voting sets
Performance Metrics for Mutual Exclusion Algorithms

- **Message complexity**
  - The number of messages required per CS execution by a site.

- **Synchronization delay**
  - After a site leaves the CS, it is the time required before the next site enters the CS.

- **Response time**
  - The time interval a request waits for its CS execution to be over after its request messages have been sent out.

- **System throughput**
  - The rate at which the system executes requests for the CS. The formula is:
    \[
    \text{System throughput} = \frac{1}{SD + E}
    \]
  - where SD is the synchronization delay and E is the average critical section execution time.
Central Coordinator Algorithm

- A central coordinator grants permission to enter the critical section (CS).
- To enter the CS, a process
  - sends a request message to the coordinator and
  - waits for a reply (during this waiting period the process can continue work)
- The reply from the coordinator gives a process the right to enter the CS.
- After completing work in the CS, the process notifies the coordinator with a release message.
Central Coordinator Algorithm

- **Advantages**
  - Scheme is easy to implement
  - Requires only 3 messages to get access to a critical section (request, OK, release)

- **Issues with centralized coordinator**
  - A performance bottleneck
  - Critical point of Failure
    - New coordinator must be elected when existing coordinator fails
    - New coordinator can be one of the processes competing for access to the CS
    - A *leader election* algorithm must select one and only one coordinator.
Lamport’s Algorithm

- Basic Idea
  - Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
  - Every site $S_i$ keeps a queue, request $\text{queue}_i$, which contains mutual exclusion requests ordered by their timestamps.
  - This algorithm requires communication channels to deliver messages the FIFO order.
Lamport’s Algorithm

- **Requesting the critical section**
  - When a site $S_i$ wants to enter the CS, it broadcasts a REQUEST($ts_i, i$) message to all other sites and places the request on request queue$_i$. (($ts_i, i$) denotes the timestamp of the request.)
  - When a site $S_j$ receives the REQUEST($ts_i, i$) message from site $S_i$, it places site $S_i$’s request on request queue of $j$ and returns a timestamped REPLY message to $S_i$

- **Executing the critical section**
  - Site $S_i$ enters the CS when the following two conditions hold:
    - L1: $S_i$ has received a message with timestamp larger than ($ts_i, i$) from all other sites.
    - L2: $S_i$’s request is at the top of request queue$_i$.

- **Releasing the critical section**
  - Site $S_i$, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
  - When a site $S_j$ receives a RELEASE message from site $S_i$, it removes $S_i$’s request from its request queue.
  - When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.
Performance – Lamport’s Algorithm

- For each CS execution Lamport’s algorithm requires
  - $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages.
  - Thus, Lamport’s algorithm requires $3(N - 1)$ messages per CS invocation.

- Optimization
  - In Lamport’s algorithm, REPLY messages can be omitted in certain situations.
  - For example, if site $S_j$ receives a REQUEST message from site $S_i$ after it has sent its own REQUEST message with timestamp higher than the timestamp of site $S_i$’s request, then site $S_j$ need not send a REPLY message to site $S_i$.
  - This is because when site $S_i$ receives site $S_j$’s request with timestamp higher than its own, it can conclude that site $S_j$ does not have any smaller timestamp request which is still pending.
  - With this optimization, Lamport’s algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages per CS execution.
Ricart-Agrawala Algorithm 1

- In a distributed environment it seems more natural to implement mutual exclusion, based upon distributed agreement - not on a central coordinator.

- It is assumed that all processes keep a (Lamport’s) logical clock which is updated according to the clock rules.
  - The algorithm requires a total ordering of requests. Requests are ordered according to their global logical timestamps; if timestamps are equal, process identifiers are compared to order them.

- The process that requires entry to a CS multicasts the request message to all other processes competing for the same resource.
  - Process is allowed to enter the CS when all processes have replied to this message.
  - The request message consists of the requesting process’ timestamp (logical clock) and its identifier.

- Each process keeps its state with respect to the CS: released, requested, or held.
Ricart-Agrawala Algorithm

- A request issued by a process Pj is blocked by another process Pi only if Pi is holding the resource or if it is requesting the resource with a higher priority (i.e. smaller timestamp) than Pj.

- Issues
  - Expensive in terms of message traffic.
    - requires 2(n-1) messages for entering the CS; (n-1) requests and (n-1) replies.
  - The failure of any process involved makes progress impossible.
    - Special recovery measures must be taken.
Ricart-Agrawala Algorithm 1

☞ Rule for process initialization
   /* performed by each process \(P_i\) at initialization */
   [RI1]: \(\text{state}_{P_i} := \text{RELEASED}\). 

☞ Rule for handling incoming requests
   /* performed by \(P_i\) whenever it received a request \((T_{P_j}, j)\) from \(P_j\) */
   [RH1]: if \(\text{state}_{P_i} = \text{HELD}\) or ((\(\text{state}_{P_i} = \text{REQUESTED}\) and ((\(T_{P_i}, i) < (T_{P_j}, j))\)) then
   Queue the request from \(P_j\) without replying.
   else
   Reply immediately to \(P_j\),
   end if.

☞ Rule for releasing a CS
   /* performed by \(P_i\) after it finished work in a CS */
   [RR1]: \(\text{state}_{P_i} := \text{RELEASED}\).
   \(P_i\) replies to all queued requests.
Example: Ricart-Agrawala Algorithm

Request message<br>\(<T, Pi> = <102, 32>\)
Example: Ricart-Agrawala Algorithm

N32 state: Held. Can now access CS

Reply messages
Example: Ricart-Agrawala Algorithm 1

N12 state: Requested

Request message \(<115, 12>\)

N6

N32

N32 state: Held.
Can now access CS

N80 state: Requested

Request message \(<110, 80>\)

N5
Example: Ricart-Agrawala Algorithm

N32 state: Held. Can now access CS

N80 state: Requested

N12 state: Requested

Request messages

Reply messages

Request message <110, 80>

Request message <115, 12>
Example: Ricart-Agrawala Algorithm

N12 state: Requested

N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N80 state: Requested

N12

Request message
<115, 12>

Reply messages

N3

N6

N32

Request message
<110, 80>

N80

N5
Example: Ricart-Agrawala Algorithm

N12 state: Requested

Request message: <115, 12>

N6

N3 state: Held.
Can now access CS
Queue requests: <115, 12>, <110, 80>

Reply messages:

N32

N32 state: Held.
Can now access CS
Queue requests: <115, 12>, <110, 80>

N80 state: Requested
Queue requests: <115, 12> (since > (110, 80))
Example: Ricart-Agrawala Algorithm

- **N12 state**: Requested
- **N12** requests: 115, 12
  - Reply message: Request message 115, 12

- **N3**

- **N6**

- **N80 state**: Requested
  - Queue requests: 115, 12

- **N32 state**: Held.
  - Can now access CS
  - Queue requests: 115, 12, 110, 80
  - Request message 110, 80

- **N5**

- **Example**: Ricart-Agrawala Algorithm
N12 state: **Requested** (waiting for N80’s reply)

N32 state: **Released**.
Multicast Reply to <115, 12>, <110, 80>

N80 state: **Held**. Can now access CS.
Queue requests: <115, 12>

*Example: Ricart-Agrawala Algorithm*
Token-Based Mutual Exclusion

- Token Ring Algorithm
- Ricart-Agrawala Second Algorithm
- Suzuki-Kazami Broadcast Algorithm
**Token Ring Algorithm**

● Simple - arrange the n processes in a logical ring.

● The logical ring topology is created by giving each process the address of one other process which is its neighbor in the clockwise direction.

● Logical ring topology is unrelated to the physical interconnection between computers.
Token Ring-based Mutual Exclusion

Currently holds token, can access CS

Token:  

N12
N3
N6
N32
N80
N5
Token Ring-based Mutual Exclusion

Cannot access CS anymore

Here’s the token!

Token:

N12
N3
N6
N32
N80
N5
Token Ring-based Mutual Exclusion

Currently holds token, can access CS

Token:
Token Ring Algorithm

- The token is initially given to one process.
- The token is passed from one process to its neighbor around the ring.
- When a process requires entry to the CS, it waits to receive a token from its neighbor (right) and retains it; enters the CS; when done, it passes the token to its left neighbor (clockwise).
- When a process receives the token, but does not require entry to CS, it passes it along the ring.
Token Ring Algorithm

- It can take from 1 to \( n-1 \) messages to obtain a token. Messages are sent around the ring even when no process requires the token (additional load on network).

- Works well in heavily loaded situations when there is a high probability that the process which gets the token wants to enter the CS. Works poorly in lightly loaded cases.

- If a process fails → no progress can be made until a reconfiguration is applied to extract the process from the ring.

- If the process holding the token fails → a unique process has to be picked to regenerate token and pass it along the ring (election algorithm).
Ricart-Agrawala Second Algorithm

- A process is allowed to enter the critical section when it gets the token.
  - Initially the token is assigned arbitrarily to one of the processes.

- In order to get the token it sends a request to all other processes competing for the same resource.
  - The request message consists of the requesting process’ timestamp (logical clock) and its identifier.

- When a process $P_i$ leaves a critical section
  - it passes the token to one of the processes which are waiting for it; this will be the first process $P_j$, where $j$ is searched in order $[i+1, i+2, \ldots, n, 1, 2, \ldots, i-2, i-1]$ for which there is a pending request.
  - If no process is waiting, $P_i$ retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.

- How does $P_i$ find out if there is a pending request?
  - Each process $P_i$ records the timestamp corresponding to the last request it got from process $P_j$, in request$_{Pi}[j]$. In the token itself, token$_{[j]}$ records the timestamp (logical clock) of $P_j$’s last holding of the token. If request$_{Pi}[j] >$ token$_{[j]}$ then $P_j$ has a pending request.
Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.
Ricart-Agrawala Second Algorithm (cont'd)

The Algorithm

- Rule for process initialization
  
  / * performed at initialization */

  [RI1]: \textit{state}_{P_i} := \text{NO-TOKEN} for all processes \( P_i \), except one single process \( P_x \) for which \( \text{state}_{P_x} := \text{TOKEN-PRESENT} \).

  [RI2]: \textit{token}[k] initialized 0 for all elements \( k = 1 \ldots n \).

  \textit{request}_{P_i}[k] initialized 0 for all processes \( P_i \) and all elements \( k = 1 \ldots n \).

- Rule for access request and execution of the CS
  
  / * performed whenever process \( P_i \) requests an access to the CS and when it finally gets it; in particular \( P_i \) can already possess the token */

  [RA1]: \textbf{if} \( \text{state}_{P_i} = \text{NO-TOKEN} \) \textbf{then}

  \hspace{1em} \( P_i \) sends a request message to all processes;
  \hspace{1em} the message is of the form \( (T_{P_i}, i) \), where
  \hspace{1em} \( T_{P_i} = C_{P_i} \) is the value of the local logical clock,
  \hspace{1em} and \( i \) is an identifier of \( P_i \).
  \hspace{1em} \( P_i \) waits until it receives the token.

  \hspace{1em} \textbf{end if.}

  \hspace{1em} \textit{state}_{P_i} := \text{TOKEN-HELĐ}.

  \hspace{1em} \( P_i \) enters the CS.
Ricart-Agrawala Second Algorithm (cont’d)

- Rule for handling incoming requests
  /* performed by \( P_i \) whenever it received a request \((T_{P_j}, j)\) from \( P_j \) */
  [RH1]: request\( j \) := max\( (\text{request} \[ j \], T_{P_j}) \).
  [RH2]: if state\(_{P_i}\) = TOKEN-PRESENT then
    \( P_i \) releases the resource (see rule RR2).
  end if.

- Rule for releasing a CS
  /* performed by \( P_i \) after it finished work in a CS or when it holds a token without using it and it got a request */
  [RR1]: state\(_{P_i}\) = TOKEN-PRESENT.
  [RR2]: for \( k = [i+1, i+2, \ldots, n, 1, 2, \ldots, i-2, i-1] \) do
    if request\( k \) > token\( k \) then
      state\(_{P_i}\) := NO-TOKEN.
      token\( i \) := \( C_{P_k} \), the value of the local logical clock.
      \( P_i \) sends the token to \( P_k \).
      break. /* leave the for loop */
    end if.
  end for.
Ricart-Agrawala Second Algorithm (cont’d)
Ricart-Agrawala Second Algorithm (cont’d)

- The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires $n$ messages for entering a CS: $(n-1)$ requests and one reply.

- The failure of a process, except the one which holds the token, doesn’t prevent progress.
Coordination in practice - Distributed Lock Managers

Distributed Concurrency executing processes/applications must share mutable resources in a **conflict-free, highly available, and scalable manner**

- one application compacts file/volume; another writes files to volume.

**Popular approach:** Centralized coordination with replicas; uses consensus protocols to maintain HA logs.

- Initially used in clustered file systems for disk cache coherency (DEC VMS systems)
- Borrow Ideas from distributed database technologies to create distributed lock managers (hierarchy of locks for database, table, record, field)...
- Incorporate ideas from consensus protocols -- Paxos, Raft.. to maintain highly available logs.
Modern Lock Managers

Typically adapted for more loosely coupled systems

- **Google Chubby** - coarse-grained locking to create a reliable distributed file system. Used in GFS, BigTable to synchronize accesses to shared resources.
- **Apache ZooKeeper** - created at Yahoo, is open-source software for distributed locking, uses Paxos variant.
- **Redis (Remote Dictionary Server)** -- open-source key-value store; used to implement the Redlock algorithm for distributed lock management.
- **Apache Etc.d** - open-source distributed reliable key-value store, Raft-based consensus
- **Cinder** - OpenStack Block Storage Service, Avoid multiple entities from manipulating the same volume resource(s) at the same time while still being scalable and highly available. Provides file locks, volume state transitions.
- **Ironic** - Avoid multiple conductors from manipulating the same bare-metal instances and/or nodes at the same time while still being scalable and highly available. Uses consistent hashing principles to distribute work (open-source)
- **Heat**: Prevents multiple engines working on the same stack (or nested stack of) by locking a stack using a database lock. Lock a stack using a database lock and disallow other engines from working on that same stack. Uses token-ring network (open-source).
- **HashiCorp's Consul** - open-source software and can be used to perform distributed locks as well.
- **Taooka distributed lock manager** uses the "try lock" methods to avoid deadlocks. It can also specify a TTL for each lock with nanosecond precision.
Uber: An interesting usecase

Uber Marketplace Meetup: Using Distributed Locking to Build Reliable Systems

https://www.youtube.com/watch?v=MDuagr729aU
Distributed Deadlocks

- Deadlocks is a fundamental problem in distributed systems.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.
- Conditions for a deadlocks
  - Mutual exclusion, hold-and-wait, No-preemption and circular wait.
Modeling Deadlocks

- In addition to the standard assumptions (no shared memory, no global clock, no failures), we make the following assumptions:
  - The systems have only reusable resources.
  - Processes are allowed to make only exclusive access to resources.
  - There is only one copy of each resource.
  - A process can be in two states: *running* or *blocked*.
    - In the *running* state (also called *active state*), a process has all the needed resources and is either executing or is ready for execution.
    - In the *blocked* state, a process is waiting to acquire some resource.
  - The state of the system can be modeled by directed graph, called a *wait for graph (WFG)*.
    - In a WFG, nodes are processes and there is a directed edge from node P1 to node P2 if P1 is blocked and is waiting for P2 to release some resource.
  - A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.
Figure 1: An Example of a WFG
Techniques for Handling Deadlocks

- Note: No site has accurate knowledge of the current state of the system

- Techniques
  - Ignore it...
  - Deadlock Prevention (collective/ordered requests, preemption)
    - Often Inefficient,
  - Deadlock Avoidance
    - A resource is granted to a process if the resulting global system state is safe
    - Requires advance knowledge of processes and their resource requirements
    - Impractical
  - Deadlock Detection and Recovery
    - Maintenance of local/global WFG and searching of the WFG for the presence of cycles (or knots), local/centralized deadlock detectors
    - Recovery by operator intervention, break wait-for dependencies, termination and rollback
Distributed deadlock prevention

• Deny circular wait
• Assign a unique timestamp to each transaction
• Ensure that the *Global Wait-For Graph* can only proceed from *young to old* or from *old to young*
Deadlock prevention

• When a process is about to block waiting for a resource used by another
  – Check to see which has a larger timestamp (which is older)

• Allow the wait only if the waiting process has an older timestamp (is older) then the process waited for

• Following the resource allocation graph, we see that timestamps always have to increase, so cycles are impossible.

• Alternatively: allow processes to wait only if the waiting process has a higher (younger) timestamp than the process waiting for.
Wait-die algorithm

- Old process wants resource held by a younger process
  - old process waits

- Young process wants resource held by older process
  - young process kills itself

Only permit older processes to wait on resources held by younger processes.
Wound-wait algorithm

• Instead of killing the transaction making the request, kill the resource owner
• Old process wants resource held by a younger process
  – old process kills the younger process
• Young process wants resource held by older process
  – young process waits

Only permit younger processes to wait on resources held by older processes.
Deadlock Detection

Correctness Criteria:

- **Safety (No false deadlocks)**
  - The algorithm should not report deadlocks which do not exist (called *phantom or false deadlocks*).

- **Progress (No undetected deadlocks)**
  - The algorithm must detect all existing deadlocks in finite time. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.
Global Deadlock Detection

- Centralized Algorithm

● Choose one site to be the **coordinator**
● Each site maintains its own WFG
● Every site sends its WFG to **coordinator**
● **Coordinator** constructs a global WFG
  ○ Messages sent to **coordinator** when edges to (resource request/hold/release) WFG are added/deleted
  ○ Can List be sent periodically?
● **Coordinator** checks for cycle in global WFG
  ○ if yes, possibly deadlock
● **Coordinator** may choose victim
Global WFG - Example

Local WFG on Node A

Local WFG on Node B

Global WFG
Centralized deadlock detection

Two events occur:
1. Process $P_1$ releases resource $R$ on system $A$
2. Process $P_1$ asks system $B$ for resource $T$

Two messages are sent to the coordinator:
1 (from $A$): release $R$
2 (from $B$): wait for $T$

If message 2 arrives first, the coordinator constructs a graph that has a cycle and hence detects a deadlock. This is **phantom deadlock**.

A **phantom deadlock** is sometimes known as a **false deadlock**.
Avoiding phantom deadlocks

- Impose globally consistent total ordering on all processes
- Coordinator asks each process if there are release messages
Deadlock Detection: Models and Types of Algorithms

- Multiple models
  - Single-resource, AND, OR, AND-OR, P-out-of-Q model

- Classes of Deadlock Detection Algorithms
  - Path-pushing
    - distributed deadlocks are detected by maintaining an explicit global WFG (constructed locally and pushed to neighbors)
  - Edge-chasing
    - the presence of a cycle in a distributed graph structure is verified by propagating special messages called probes, along the edges of the graph. The formation of cycles can be detected by a site if it receives the matching probe sent by it previously.
  - Diffusion computation
    - deadlock detection computation is diffused through the WF
  - Global state detection
    - Take a snapshot of the system and examining it for the condition of a deadlock.
Distributed Deadlock Detection: Chandy-Mishra-Haas Algorithm

Chandy-Misra-Haas algorithm

Edge Chasing

When requesting a resource, generate a probe message
- Send to all process(es) currently holding the needed resource
- Message contains three process IDs: \{\textit{blocked\_ID}, \textit{my\_ID}, \textit{holder\_ID}\}

1. Process that originated the message (\textit{blocked\_ID})
2. Process sending (or forwarding) the message (\textit{my\_ID})
3. Process to whom the message is being sent (\textit{holder\_ID})
Distributed Deadlock Detection: Chandy-Mishra-Haas Algorithm

- When *probe* message arrives, recipient checks to see if it is waiting for any processes

- If so, update & forward message: \{blocked\_ID, my\_ID, holder\_ID\}
  - Replace *my\_ID* field by its own process ID
  - Replace *holder\_ID* field by the ID of the process it is waiting for
  - Send messages to each process on which it is blocked

- If a message goes all the way around and comes back to the original sender, a cycle exists
  - *We have deadlock*
Distributed Deadlock Detection: Chandy-Mishra-Haas Algorithm

• Process 0 needs a resource process 1 is holding
• That means process 0 will block on process 1
  – Initial message from P₀ to P₁: (0,0,1)
  – P₁ sends (0,1,2) to P₂; P₂ sends (0,2,3) to P₃
• Message (0,8,0) returns back to sender
  – cycle exists: deadlock
Election Algorithms

- Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role.

- Examples with mutual exclusion
  - Central coordinator algorithm
    - At initialization or whenever the coordinator crashes, a new coordinator has to be elected.
  - Token ring algorithm
    - When the process holding the token fails, a new process has to be elected which generates the new token.
Election Algorithms

- It doesn’t matter which process is elected.
  - What is important is that one and only one process is chosen (we call this process the coordinator).
  - All processes agree on this decision.
  - Let all processes know about this leader.

What happens if the leader fails?
- Election is typically started after the failure occurs.
  - The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out.
    - A process that gets no response for a period of time suspects a failure and initiates an election process.

- An election process is typically performed in two phases:
  - Select a leader with the highest priority.
  - Inform all processes about the winner.

- Assume that each process has a unique number (identifier).
  - In general, election algorithms attempt to locate the process with the highest number, among those which currently are up.
Assumptions - System Model

- $N$ processes.
- Each process has a unique id.
- Messages are eventually delivered.
- Failures may occur during the election protocol.

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes are allowed to call an election simultaneously.
  - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it.
The Leader Election Problem

- A run of the election algorithm must always guarantee at the end:
  - **Safety**: For all non-faulty processes $p$: ($p$'s elected = ($q$: a particular non-faulty process with the best attribute value) or Null)
  - **Liveness**: For all election runs: (election run terminates)
    & for all non-faulty processes $p$: $p$'s elected is not Null

- At the end of the election protocol, the non-faulty process with the **best (highest)** election attribute value is elected.
  - Common attribute: leader has highest id
  - Other attribute examples: leader has highest IP address, or fastest cpu, or most disk space, or most number of files, etc.
Leader Election: The Ring Algorithm

- $N$ processes are organized in a logical ring
  - $i$-th process $p_i$ has a communication channel to $p_{(i+1) \mod N}$
  - All messages are sent clockwise around the ring.
The Ring Election Protocol

- Any process $p_i$ that discovers the old coordinator has failed initiates an “Election” message that contains $p_i$’s own id:attr. This is the initiator of the election.
- When a process $p_i$ receives an “Election” message, it compares the attr in the message with its own attr.
  - If the arrived attr is greater, $p_i$ forwards the message.
  - If the arrived attr is smaller and $p_i$ has not forwarded an election message earlier, it overwrites the message with its own id:attr, and forwards it.
  - If the arrived id:attr matches that of $p_i$, then $p_i$’s attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an “Elected” message to its neighbor with its id, announcing the election result.
- When a process $p_i$ receives an “Elected” message, it
  - sets its variable $elected_i$ id of the message.
  - forwards the message unless it is the new coordinator.
The Ring-based Algorithm

- We assume that the processes are arranged in a logical ring
  - Each process knows the address of one other process, which is its neighbor in the clockwise direction.
- The algorithm elects a single coordinator, which is the process with the highest identifier.
- Election is started by a process which has noticed that the current coordinator has failed.
  - The process places its identifier in an election message that is passed to the following process.
  - When a process receives an election message
    - It compares the identifier in the message with its own.
    - If the arrived identifier is greater, it forwards the received election message to its neighbor
    - If the arrived identifier is smaller it substitutes its own identifier in the election message before forwarding it.
    - If the received identifier is that of the receiver itself, this will be the coordinator.
- The new coordinator sends an elected message through the ring.
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Election: 80

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Initiates the election

Goal: Elect highest id process as leader
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader

Elected: 80

Initiates the election
Leader Election: Ring Algorithm Example

Elected: 80

Goal: Elect highest id process as leader

Initiates the election
Leader Election: Ring Algorithm Example

Goal: Elect highest id process as leader

N80 elected = 80

N32 elected = 80

N12 elected = 80

N3 elected = 80

N6 elected = 80

N5 elected = 80

Initiates the election
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example

N12
N6
N80
N3
N32
N5
Initiates the election

elected = 80

elected = 80

elected = 80

elected = 80

elected = 80

elected = 80

elected = 80

elected = 80
Worst-case Analysis

- Let’s assume no failures occur during the election protocol itself, and there are $N$ processes
- How many messages?
- Worst case occurs when the initiator is the ring successor of the would-be leader

Goal: Elect highest id process as leader
Worst/Best/Avg Case Analysis

Worst-case Analysis
- \((N-1)\) messages for Election message to get from Initiator to would-be coordinator
- \(N\) messages for Election message to circulate around ring without message being changed
- \(N\) messages for Elected message to circulate around the ring
- Message complexity: \((3N-1)\) messages
- Completion time: \((3N-1)\) message transmission times
- Thus, if there are no failures, election terminates (liveness) and everyone knows about highest-attribute process as leader (safety)

Best Case Analysis
- Initiator is the would-be leader.
- Message complexity: \(2N\) messages
- Completion time: \(2N\) message transmission times

Average Case Analysis
- \((2N + N/2)\) messages; \((2N + N/2)\) completion time
Leader Election: The Ring Algorithm

Multiple Initiators

- Several elections can be active at the same time.
  - Messages generated by later elections should be killed as soon as possible.
- Processes can be in one of two states
  - Participant or Non-participant.
    - Initially, a process is non-participant.
- The process initiating an election marks itself participant.
- Rules
  - Include initiators id with all messages
  - For a participant process, if the identifier in the election message is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election).
  - When forwarding an election message, a process marks itself participant.
  - When sending (forwarding) an elected message, a process marks itself non-participant.
Leader Election: The Ring Algorithm
Multiple Initiators

- By default, the state of a process is NON-PARTICIPANT.

- Rule for election process initiator
  
  
  /* performed by a process $P_i$, which triggers the election procedure */

  [RE1]: $state_{P_i} := PARTICIPANT$.

  [RE2]: $P_i$ sends an election message with $message.id := i$ to its neighbour.

- Rule for handling an incoming election message
  
  /* performed by a process $P_j$, which receives an election message */

  [RH1]: if $message.id > j$ then
  
  $P_j$ forwards the received election message.
  $state_{P_j} := PARTICIPANT$.

  elseif $message.id < j$ then
    if $state_{P_j} = NON-PARTICIPANT$ then
      $P_j$ forwards an election message with $message.id := j$.
      $state_{P_j} := PARTICIPANT$
    end if
  else
    $P_j$ is the coordinator and sends an elected message with $message.id := j$ to its neighbour.
    $state_{P_j} := NON-PARTICIPANT$.
  end if.

- Rule for handling an incoming elected message
  
  /* performed by a process $P_i$, which receives an elected message */

  [RD1]: if $message.id \neq i$ then
    $P_i$ forwards the received elected message.
    $state_{P_j} := NON-PARTICIPANT$.
  end if.
Failures?

- One option: have predecessor (or successor) of would-be leader N80 detect failure and start a new election run
  - May re-initiate election if
    - Receives an Election message but times out waiting for an Elected message
    - Or after receiving the Elected:80 message
  - But what if predecessor also fails?
  - And its predecessor also fails? (and so on)
Fixing for failures (2)

- Second option: use the failure detector

- Any process, after receiving Election:80 message, can detect failure of N80 via its own local failure detector
  - If so, start a new run of leader election

- But failure detectors may not be both complete and accurate
  - Incompleteness in FD => N80’s failure might be missed => Violation of Safety
  - Inaccuracy in FD => N80 mistakenly detected as failed
    - => new election runs initiated forever
    - => Violation of Liveness
Why is Election so Hard?

- Because it is related to the consensus problem!
- If we could solve election, then we could solve consensus!
  - Elect a process, use its id’s last bit as the consensus decision
- But since consensus is impossible in asynchronous systems, so is election!
Leader Election: 
The Bully Algorithm

- A process has to know the identifier of all other processes
  - (it doesn’t know, however, which one is still up); the process with the highest identifier, among those which are up, is selected.

- Any process could fail during the election procedure.

- When a process Pi detects a failure and a coordinator has to be elected
  - It sends an election message to all the processes with a higher identifier and then waits for an answer message:
  - If no response arrives within a time limit
    - Pi becomes the coordinator (all processes with higher identifier are down)
    - it broadcasts a coordinator message to all processes to let them know.
  - If an answer message arrives,
    - Pi knows that another process has to become the coordinator it waits in order to receive the coordinator message.
    - If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) Pi resends the election message.

- When receiving an election message from Pi
  - a process Pj replies with an answer message to Pi and
  - then starts an election procedure itself( unless it has already started one) it sends an election message to all processes with higher identifier.

- Finally all processes get an answer message, except the one which becomes the coordinator.
The Bully Algorithm

By default, the state of a process is ELECTION-OFF.

Rule for election process initiator
/* performed by a process $P_i$, which triggers the election procedure, or which starts an election after receiving itself an election message */

[RE1]: $state_{P_i} :=$ ELECTION-ON.

$P_i$ sends an election message to all processes with a higher identifier.

$P_i$ waits for an answer message.

if no answer message arrives before time-out then

$P_i$ is the coordinator and sends a coordinator message to all processes.

else

$P_i$ waits for a coordinator message to arrive.

if no coordinator message arrives before time-out then

restart election procedure according to RE1

end if

end if.

Rule for handling an incoming election message
/* performed by a process $P_i$ at reception of an election message coming from $P_j$ */

[RH1]: $P_i$ replies with an answer message to $P_j$.

[RH2]: if $state_{P_i} :=$ ELECTION-OFF then

start election procedure according to RE1

end if
Bully Algorithm: Example

Detects failure of N80
Bully Algorithm: Example

Detects failure of N80

Election

Detects failure of N80
Bully Algorithm: Example

N80

Election

N32

Election

N12

OK

N6

Waiting…

N3

N5
Bully Algorithm: Example
Bully Algorithm: Example

Coordinator: N32

Times out waiting for N80’s response

Election is completed
Failures during Election Run
Bully Algorithm: Example

Waiting…

Times out, starts new election run
Bully Algorithm: Example

Times out, starts another new election run
The Bully Algorithm

- If $P_6$ crashes before sending the coordinator message, $P_4$ and $P_5$ restart the election process.

- The best case: the process with the second highest identifier notices the coordinator’s failure. It can immediately select itself and then send $n-2$ coordinator messages.

- The worst case: the process with the lowest identifier initiates the election; it sends $n-1$ election messages to processes which themselves initiate each one an election $\Rightarrow O(n^2)$ messages.
The Bully Algorithm: Analysis

- **Worst-case completion time:**
  - When the process with the lowest id in the system detects the failure.
  - \((N-1)\) processes altogether begin elections, each sending messages to processes with higher ids.
  - \(i\)-th highest id process sends \((i-1)\) election messages
  - Number of Election messages
    \[= N-1 + N-2 + \ldots + 1 = (N-1) \times \frac{N}{2} = O(N^2)\]

- **Best-case**
  - Second-highest id detects leader failure
  - Sends \((N-2)\) Coordinator messages
  - Completion time: 1 message transmission time
Impossibility?

- Since timeouts built into protocol, in asynchronous system model:
  - Protocol may never terminate => Liveness not guaranteed
- But satisfies liveness in synchronous system model where
  - Worst-case one-way latency can be calculated
    = worst-case processing time + worst-case message latency
Election in Industry

- Several systems in industry use consensus based approaches for election
  - e.g. Paxos is a consensus protocol (safe, but eventually live):
- Google’s Chubby system
- Apache Zookeeper
Distributed Process and Resource Management

- Need multiple other policies to determine when and where to execute processes in distributed systems – useful for load balancing, reliability
  - Load Estimation Policy
    - How to estimate the workload of a node
  - Process Transfer Policy
    - Whether to execute a process locally or remotely
  - Location Policy
    - Which node to run the remote process on
  - Priority Assignment Policy
    - Which processes have more priority (local or remote)
  - Migration Limiting policy
    - Number of times a process can migrate
Load Balancing

- Computer overloaded
  - Decrease load maintain scalability, performance, throughput - transparently
- Load Balancing
  - Can be thought of as “distributed scheduling”
    - Deals with distribution of processes among processors connected by a network
  - Can also be influenced by “distributed placement”
    - Especially in data intensive applications

Load Balancer
- Manages resources
- Policy driven Resource assignment
Load Balancing Issues

- **How**
  - To search for lightly loaded machines

- **When**
  - should load balancing decisions be made
  - to migrate processes or forward requests?

- **Which**
  - processes should be moved off a computer?
  - processor should be chosen to handle a given process or request

- **What should be taken into account when making the above decisions? How should old data be handled**

- **Should**
  - load balancing data be stored and utilized centrally, or in a distributed manner
  - What is the performance/overhead tradeoff incurred by load balancing
  - Prevention of overloading a lightly loaded computer
Static vs. dynamic

- Static load balancing - CPU determined at process creation.
- Dynamic load balancing - processes dynamically migrate to other computers to balance the CPU (or memory) load.
  - **Parallel machines** - *dynamic balancing schemes seek to minimize total execution time of a single application running in parallel on a multiple nodes*
  - **Web servers** - *scheduling client requests among multiple nodes in a transparent way to improve response times for interaction*
  - **Multimedia servers** - *resource optimization across streams and servers for QoS; may require admission control*
Process Migration

• Process migration mechanism
  • Freeze the process on the source node and restart it at the destination node
  • Transfer of the process address space
  • Forwarding messages meant for the migrant process
  • Handling communication between cooperating processes separated as a result of migration
  • Handling child processes

• Process Migration Policies: Depend on
  • CPU Load
  • Memory Requirements
  • I/O Activity -- Where is the physical device?
  • Communication - which processes communicate with each other?
Process Migration and Heterogeneous Systems

- Converts usage of heterogeneous resources (CPU, memory, IO) into a single, homogeneous cost using a specific cost function.
- Assigns/migrates a job to the machine on which it incurs the lowest cost.
  - Can design online job assignment policies based on multiple factors - economic principles, competitive analysis.
  - Aim to guarantee near-optimal global lower-bound performance.
Mosix: Early Cluster Computing

- Mosix (from Hebrew U): Kernel level enhancement to Linux that provides dynamic load balancing in a network of workstations.
- Dozens of PC computers connected by local area network (Fast-Ethernet or Myrinet).
- Any process can migrate anywhere anytime.
Architectures for Migration

<table>
<thead>
<tr>
<th>user level</th>
<th>user level</th>
</tr>
</thead>
<tbody>
<tr>
<td>system calls interface</td>
<td>system calls interface</td>
</tr>
<tr>
<td>communication</td>
<td>communication</td>
</tr>
<tr>
<td>lower kernel</td>
<td>lower kernel</td>
</tr>
</tbody>
</table>

Architecture that fits one system image. Needs location transparent file system. (Mosix early versions)

Architecture that fits entrance dependant systems. Easier to implement based on current Unix. (Mosix later versions)
Mosix: Migration and File Access

Each file access must go back to deputy…

== Very Slow for I/O apps.

Solution: Allow processes to access a distributed file system through the current kernel.
Distributed File Systems (DFS)

A distributed implementation of the classical file system model

- Requirements
  - Transparency: Access, Location, Mobility, Performance, Scaling
  - Allow concurrent access
  - Allow file replication
  - Tolerate hardware and operating system heterogeneity
  - Security - Access control, User authentication

- Issues
  - File and directory naming – Locating the file
  - Semantics – client/server operations, file sharing
  - Performance
  - Fault tolerance – Deal with remote server failures
  - Implementation considerations - caching, replication, update protocols
Issues: File and Directory Naming

- **Explicit Naming**
  - Machine + path  /machine/path
    - one namespace but not transparent

- **Implicit naming**
  - Location transparency
    - file name does not include name of the server where the file is stored
  - Mounting remote filesystems onto the local file hierarchy
    - view of the filesystem may be different at each computer
  - Full naming transparency
    - A single namespace that looks the same on all machines
Distributed File Access Alternatives

Remote access model

Download/upload model

1. File moved to client
2. Accesses are done on client
3. When client is done, file is returned to server
Semantics – File Sharing

One-copy (Sequential) Semantics

- Updates are written to the single copy and are available immediately
- All clients see contents of file identically as if only one copy of file existed
- Issue: Performance
- Soln: Use Caching: after an update operation, no program can observe a discrepancy between data in cache and stored data
  - Extra state; extra traffic

Session semantics

- more relaxed
- Copy file on open, work on local copy and copy back on close
- Changes initially only visible to file that modified it.

Serializable (Transaction semantics)

- Need file locking protocols implemented - share for read, exclusive for write).
Semantics - Operational

- Support fault tolerant operation
  - At-most-once semantics for file operations
  - At-least-once semantics with a server protocol designed in terms of idempotent file operations
  - Replication (stateless, so that servers can be restarted after failure)
DFS Performance

- Efficiency Needs
  - Latency of file accesses
  - Scalability (e.g., with increase of number of concurrent users)
- RPC Related Issues
  - Use RPC to forward every file system request (e.g., open, seek, read, write, close, etc.) to the remote server
    - Remote server executes each operation as a local request
    - Remote server responds back with the result
  - Advantage:
    - Server provides a consistent view of the file system to distributed clients.
  - Disadvantage:
    - Poor performance
    - Solution: Caching
Traditional File system Operations

- `filedes = open(name, mode)` Opens an existing file with the given name.
- `filedes = creat(name, mode)` Creates a new file with the given name.
- Both operations deliver a file descriptor referencing the open file. The mode is read, write or both.
- `status = close(filedes)` Closes the open file filedes.
- `count = read(filedes, buffer, n)` Transfers n bytes from the file referenced by filedes to buffer.
- `count = write(filedes, buffer, n)` Transfers n bytes to the file referenced by filedes from buffer.
- Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
- `pos = lseek(filedes, offset, whence)` Moves the read-write pointer to offset (relative or absolute, depending on whence).
- `status = unlink(name)` Removes the file name from the directory structure. If the file has no other names, it is deleted.
- `status = link(name1, name2)` Adds a new name (name2) for a file (name1).
- `status = stat(name, buffer)` Gets the file attributes for file name into buffer.
Flat File Service
- Performs file operations
- Uses “unique file identifiers” (UFIDs) to refer to files
- Flat file service interface
  - RPC-based interface for performing file operations
  - Not normally used by application level programs
Architecture (2)

- **Directory Service**
  - Mapping of UFIDs to “text” file names, and vice versa

- **Client Module**
  - Provides API for file operations available to application program
Network File Systems: Sun-NFS

- **Supports heterogeneous systems**
  - **Architecture**
    - Server exports one or more directory trees for access by remote clients
    - Clients access exported directory trees by mounting them to the client local tree
    - Diskless clients mount exported directory to the root directory
  - **Protocols**
    - Mounting protocol
    - Directory and file access protocol - stateless, no open-close messages, full access path on read/write
  - **Semantics** - no way to lock files
Sun Network File System
Mounting of File Systems

- Making remote file systems available to a local client, specifying remote host name and pathname
- Mount protocol (RPC-based)
  - Returns file handle for directory name given in request
  - Location (IP address and port number) and file handle are passed to Virtual File System and NFS client
- Hard-mounted (mostly used in practice)
  - User-level process suspended until operation completed
  - Application may not terminate gracefully in failure situations
- Soft-mounted
  - Error message returned by NFS client module to user-level process after small number of retries
Mounting Example

The file system mounted at /usr/students in the client is actually the sub-tree located at /export/people in Server 1; the file system mounted at /usr/staff in the client is actually the sub-tree located at /nfs/users in Server 2.
Naming (2)

- Mounting nested directories from multiple servers in NFS.
Example 2: Andrew File System

- Supports information sharing on a large scale
  - 10,000+ clients
- Most files are small
  - reads more common than writes; typically one user at a time; access locality expected.
- Uses a session semantics
  - Entire file is copied to the local machine (Venus) from the server (Vice) when open. If file is changed, it is copied to server when closed.
    - Works because in practice, most files are changed by one person
- AFS File Validation (older versions)
  - On open: Venus accesses Vice to see if its copy of the file is still valid. Causes a substantial delay even if the copy is valid.
  - Vice is stateless
Example 3: The Coda Filesystem

- CoDA -- Constant Data Availability
- Descendant of AFS that is substantially more resilient to server and network failures.

General Design Principles
- know the clients have cycles to burn, cache whenever possible, exploit usage properties, minimize system wide change, trust the fewest possible entries and batch if possible

- Directories are replicated in several servers (Vice)

- Support for mobile users
  - When the Venus is disconnected, it uses local versions of files. When Venus reconnects, it reintegrates using optimistic update scheme.
Other DFS Challenges

● Naming
  ● Important for achieving location transparency
  ● Facilitates Object Sharing
  ● Mapping is performed using directories. Therefore name service is also known as *Directory Service*

● Security
  ● Client-Server model makes security difficult
  ● Cryptography based solutions
### Selected NFS operations (2)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>symlink(newdirh, newname, string) -&gt; status</code></td>
<td>Creates an entry <code>newname</code> in the directory <code>newdirh</code> of type symbolic link with the value <code>string</code>. The server does not interpret the <code>string</code> but makes a symbolic link file to hold it.</td>
</tr>
<tr>
<td><code>readlink(fh) -&gt; string</code></td>
<td>Returns the string that is associated with the symbolic link file identified by <code>fh</code>.</td>
</tr>
<tr>
<td><code>mkdir(dirfh, name, attr) -&gt; newfh, attr</code></td>
<td>Creates a new directory <code>name</code> with attributes <code>attr</code> and returns the new file handle and attributes.</td>
</tr>
<tr>
<td><code>rmdir(dirfh, name) -&gt; status</code></td>
<td>Removes the empty directory <code>name</code> from the parent directory <code>dirfh</code>. Fails if the directory is not empty.</td>
</tr>
<tr>
<td><code>readdir(dirfh, cookie, count) -&gt; entries</code></td>
<td>Returns up to <code>count</code> bytes of directory entries from the directory <code>dirfh</code>. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a <code>cookie</code>. The <code>cookie</code> is used in subsequent <code>readdir</code> calls to start reading from the following entry. If the value of <code>cookie</code> is 0, reads from the first entry in the directory.</td>
</tr>
<tr>
<td><code>statfs(fh) -&gt; fsstats</code></td>
<td>Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file <code>fh</code>.</td>
</tr>
</tbody>
</table>
### Selected NFS Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lookup(dirfh, name)</code></td>
<td>Returns file handle and attributes for the file <code>name</code> in the directory <code>dirfh</code>.</td>
</tr>
<tr>
<td><code>create(dirfh, name, attr)</code></td>
<td>Creates a new file name in directory <code>dirfh</code> with attributes <code>attr</code> and returns the new file handle and attributes.</td>
</tr>
<tr>
<td><code>remove(dirfh, name)</code></td>
<td>Removes file name from directory <code>dirfh</code>.</td>
</tr>
<tr>
<td><code>getattr(fh)</code></td>
<td>Returns file attributes of file <code>fh</code>. (Similar to the UNIX <code>stat</code> system call.)</td>
</tr>
<tr>
<td><code>setattr(fh, attr)</code></td>
<td>Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.</td>
</tr>
<tr>
<td><code>read(fh, offset, count)</code></td>
<td>Returns up to <code>count</code> bytes of data from a file starting at <code>offset</code>. Also returns the latest attributes of the file.</td>
</tr>
<tr>
<td><code>write(fh, offset, count, data)</code></td>
<td>Writes <code>count</code> bytes of data to a file starting at <code>offset</code>. Returns the attributes of the file after the write has taken place.</td>
</tr>
<tr>
<td><code>rename(dirfh, name, todirfh, toname)</code></td>
<td>Changes the name of file <code>name</code> in directory <code>dirfh</code> to <code>toname</code> in directory to <code>todirfh</code>.</td>
</tr>
<tr>
<td><code>link(newdirfh, newname, dirfh, name)</code></td>
<td>Creates an entry <code>newname</code> in the directory <code>newdirfh</code> which refers to file <code>name</code> in the directory <code>dirfh</code>.</td>
</tr>
</tbody>
</table>
Automounting (1)

A simple automounter for NFS.