D Distributed Systems

D.1 Overview

- Definition and Motivation
- Taxonomy
- Communication Models
- Selected Problems of Distributed Systems
- Object-Based Distributed Systems

D.2 References

General:

Special Problems:

D.3 Definition and Motivation

★ “Distributed System” My definition
- Hardware and software of multiple cooperating computers

★ “Distributed System” Definition according to Tanenbaum and van Renesse
- It looks like an ordinary centralized system.
- It runs on multiple, independent CPUs.
- The use of multiple processors should be invisible (transparent).

★ “Distributed System” Definition according to Mullender
- Additionally: Not any single points of failures

1 Advantages

★ Efficiency to cost ratio
- High performance computers are very expensive
- Microprocessors became very cheap
- Multiple microprocessors can easily have more computing power than a high performance computer and cost much less.

★ Costs
- Distributed systems can be much cheaper at same capacity.
- Expensive devices (e.g., color printers) can be shared by many users.

★ Efficiency
- Distributed systems can be much more efficient than any available high performance computer.
1 Advantages (2)

- Centralized CPU vs. personal computer
  - Response time of centralized systems is very bad at high load.
  - Personal computers are available for a single user.
  - More computing power available for a single user: better user interfaces, etc.

- Load Balancing
  - Unlike individual PCs, a distributed system can grant peak performance to a single user without annoying other users.

1 Advantages (3)

- Scalability
  - "No" restriction on the maximum size of the system.

- Extensibility, incremental growth
  - It is easier to add a new computer to a distributed system than to extend a high performance machine.

1 Advantages (4)

- Availability
  - Distributed systems can have redundant components (CPUs, memory, communication channels, etc.)
  - System just runs on if a component fails.

- Reliability
  - Reliability needs availability.
  - Reliable systems mask failures (e.g., CPU failure, communication failures, etc.)
  - Distributed systems can be made very reliable. However, this is a difficult task.

2 Disadvantages

- Concurrency
  - Distributed systems are inherently concurrent.
  - Controlling concurrency is complex.
  - Combining well-understood components can generate new problems not apparent to the components.

- Propagation of effect
  - One malfunctioning computer can bring down the whole system.
  - There can be unforeseen dependences between components.

- Security
  - It is harder to secure a physically distributed system.
  - Communication channels can be wire tapped and eavesdropped.
  - Data access could not be controlled on certain sites.
2 Disadvantages (2)

▲ Efficiency
◆ Distributed systems can only gain efficiency for the total output of the entire system. If you cannot parallelize your application you cannot benefit from the available high performance.

▲ Load Balancing
◆ It is hard to balance the load because the physical distribution of resources may not match the distribution of demands.

▲ Scalability
◆ A working system with ten nodes may fail miserably when it grows to a hundred nodes.

▲ Complexity
◆ All in all, a distributed system is much more complex than a centralized one (e.g., dealing with partial failures, concurrency, load balancing, etc.)

D.4 Taxonomy

■ Classification according to Flynn (1972)
◆ SISD – Single Instruction Stream, Single Data Stream
  all current single CPU computers (PCs, Mainframes)
◆ SIMD – Single Instruction Stream, Multiple Data Streams
  high performance computers, vector computers
◆ MISD – Multiple Instruction Streams, Single Data Stream
  no known system available that implements this category
◆ MIMD – Multiple Instruction Streams, Multiple Data Streams
  systems with independent CPUs

■ Distributed systems are always seen as MIMD computers
1 Multiprocessors (2)

- Connection-oriented systems
  - For more than 64 processors bus-based systems fail
  - Cross-bar switch
- Omega switching network

- Cross-bar switches need $n^2$ switches
- Omega networks need $n \log_2 n$ switches
- Slow memory access
- Solution: hierarchical systems (NUMA = Non uniform memory access)

2 Multicomputers

- Each CPU has its own private memory
- Bus-based multicomputers
  - Workstations in a LAN
  - CPUs connected to a fast communication bus

3 Network Operating Systems

- Early distributed systems
- Loosely-coupled systems
  - Multicomputers usually in a LAN
  - One (but not necessarily the same) operating system on each system
  - Users act locally
  - Users have access to remote systems
    - Remote login: `rlogin faui04a`
    - Remote copy: `rcp faui04a:aFile myCopy`
    - Shared file systems
    - Shared devices (e.g., printers)
3 Network Operating Systems (2)

- Shared file systems
  - Users can operate on remote files as on local files
  - File server provides remote access to local files
  - Local file name is not necessarily equal to remote file name

4 True Distributed Systems

- Same operating system on each node
- System behaves like a uniprocessor
  - Users should not see any differences if they access the system from another node.
  - The identity of the local computer is not important.
  - File sharing semantics is usually well-defined.

Transparencies
- Location transparency — location of resources is irrelevant
- Migration transparency — resources may move
- Replication transparency — resources may be replicated
- Concurrency transparency — multiple accesses to a resource at a time
- Parallelism transparency — activities may be executed in parallel
1 Protocol Layers (3)

- Presentation Layer
  - Transparency of different internal representations of data

- Application Layer
  - Set of application protocols
    - Electronic mail protocol
    - File transfer protocol
    - etc.

2 Classification

- Synchronicity
  - Is the sender blocked until the receiver gets a message, or not?

- Pattern of Interaction
  - Message Passing — a message is sent from one party to the other
  - Request-Reply (Client-Server) Interaction — there is a message to the receiver and a message back to the original sender

- Addressees
  - One receiver
  - Multiple receivers (group communication, multicast, broadcast)

3 Rendezvous Model

- Message passing; synchronous send
  - Sender waits until message is received
  - Receiver may be blocked until a message arrives
  - Needs no buffer space

Datagram Message

- Message passing; asynchronous send
  - Sender can proceed immediately
  - Receiver may be blocked until a message arrives
  - Needs buffer space for not yet received messages
4 Synchronous Request-Reply Model

- Request-reply interaction; synchronous send
- Client waits until reply message is received
- Server may be blocked until a request message arrives
- Client and server do not work concurrently
- Well known representative is the RPC (remote procedure call)

5 Asynchronous Request-Reply Model

- Request-reply interaction; asynchronous send
- Client and server can work concurrently
- Basis for group communication

6 Reliability

- It is possible that messages get lost if we do not use a reliable connection
  - Reliable connections introduce acknowledge messages (ACK)
  - For simple message passing this means a lot of overhead
- Combining reliability with the request-reply interaction model

- Possible errors
  - Server crash
    - failure model is: total amnesia
    - (server loses all knowledge of former requests)
  - Request message gets lost
  - Reply message gets lost
- Ideal semantics
  - exactly-once
    - The request is processed exactly once at the server side.

6 Reliability (2)

- At-Least-Once Semantics
  - Request is processed once or more times
  - Client will never notice an error message, but it may notice that the request was processed multiple times: operations need to be idempotent.

- Implementation
  - If the client does not get a reply message after some time (time-out), it resends the request.
    - There is no additional functionality needed at the server side.
    - However, the server can ignore resent requests if it can detect them.
6 Reliability (3)

- At-Most-Once Semantics
  - The request is processed once or not at all.

- Simple implementation (at the client side only)
  - If the reply message does not arrive within a certain period of time an error is returned to the caller (at-most-once semantics).
  - Otherwise, the result is returned (exactly-once semantics).

- More complex implementation
  - Client repeats request message after time-out (hides message losses on the wire).
  - Client has to identify server crashes (error code to the caller, at-most-once semantics).
  - Server keeps reply messages (enables resending if message gets lost)
  - Server has to identify and ignore old requests after server crash.
  - If the result is returned we have exactly-once semantics.

6 Reliability (4)

- Request message gets lost

6 Reliability (5)

- Processing has not yet finished

6 Reliability (6)

- Reply message gets lost

- Server keeps reply message and resends it
6 Reliability (7)

- Server crashes

```
Client
send
request message
receive
time-out
error reply message
return error

Server
```

- Server identifies old requests (old generation number) and returns error code (at-most-once semantics)

7 Remote Procedure Calls (2)

- Implementing RPCs using stub procedures

```
Client
  call A
  Stub Procedure A
  send/receive
  Request Reply Protocol

Server
  Procedure A
  call
  Stub Procedure A
  receive/reply
  Request Reply Protocol
```

acc. to Nehmer 1995

7 Remote Procedure Calls (3)

- Client stub procedure
  - Marshalling of parameters (composing a request message)
  - Sending request message
  - Waiting for reply message
  - Unmarshalling of the result
  - Implementing delivery semantics

- Server stub procedure
  - Receiving request message
  - Unmarshalling of parameters
  - Invoking server procedure
  - Marshalling of the result
  - Sending reply message
  - Implementing delivery semantics
7 Remote Procedure Calls (4)

▲ Problems with RPCs
   ◆ Marshalling of parameters
      • Number and types must be known (comp. with C: `printf("Count %d\n", count)`)
   ◆ Parameter passing semantics
      • Call-by-value: no problem
      • Call-by-reference: How to implement?
   ◆ No global variables
   ◆ Semantics
      • Server crashes; no exactly-once semantics
   ◆ Performance
      • No concurrency
      • Large parameter data
      • Short procedures

7 Remote Procedure Calls (5)

■ Automatic generation of stub procedures
   ◆ Tools generate code for:
      • parameter marshalling
      • client stub procedure
      • server stub procedure
      • server loop waiting for request messages

■ Binding client stubs to server stubs
   ◆ Server stub has a network address that must be known to the client stub
   ◆ Problem: How does the client know its server?

★ Name server
   ◆ Symbolic names are converted to network addresses

8 Name Server and Binding

■ Well known name server converts names to addresses
   ◆ Client knows a unique name for its server and the address of a name server
   ◆ Name server converts this name to a dynamic network address
   ◆ Client can always bind to the server
   ◆ Server has to register its dynamic network address with the name server

![Diagram]

- 1. Register
- 2. Query
- 3. RPC
- Client
- Name Server
- Server

9 Group Communication

■ Motivation
   ◆ Often more than one server needs to be informed
      • multiple servers administrate a resource
      • multiple redundant servers (no "single point of failure")

■ Terminology
   ◆ Unicast
      • One receiver (1:1)
   ◆ Anycast
      • One receiver of many (1:1 of n)
   ◆ Multicast
      • Multiple receivers (1:n)
   ◆ Broadcast
      • All receivers of a special group (1:n)
9 Group Communication (2)

- Implementation of multicast
  - Using a hardware-based multicast
    - e.g., Ethernet multicast
  - Using a hardware-based broadcast
    - e.g., Ethernet broadcast
    - filtering of not addressed parties at receiver side
  - Using unicast messages
    - sending an individual message to each party

- Kernel discards message

acc. to Tanenbaum 1995

9 Group Communication (3)

- Primitives for group communication
  - Message passing
    - Same primitives as for unicasts (send, receive) and multiple addressees for send
    - Different primitives: group_send, group_receive
  - Request-reply interaction
    - Multiple rcv_reply invocations to get all reply messages

- Variants of group communication semantics
  - Reliability: none, k-reliable, atomic/reliable
  - Message ordering: none, FIFO order, causal order, total order

9 Group Communication (4)

- Reliability
  - None: messages may arrive or may not arrive at a receiver
  - K-reliable: at least k members of the group receive the message
  - Atomic/reliable: all members or none of them receive the message

- Message ordering
  - None: messages arrive in arbitrary order at a receiver
  - FIFO order: messages arrive in the order sent by the sender
  - Causal order: causality of messages is reflected in the order of arrival
    - If a member of the group receives a message A and then sends a message B to the group, each group member will first receive A and then message B.
  - Total order: as causal order, but additionally not causally dependent messages arrive in the same order at each receiver

9 Group Communication (5)

- Examples for different message ordering

- FIFO Order

- Causal Order

- Total Order
**D.6 Selected Problems of Distributed Systems**

1. **Logical Clocks**
   - Usually the precise absolute time is not necessary
     - We only need to know when one event causally depends on another
     - \( a \rightarrow b \) is read “\( b \) is causally dependent on \( a \)”
     - If \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \) (transitivity)
     - If neither \( a \rightarrow b \) nor \( b \rightarrow a \) is true then \( a \) and \( b \) are said to be concurrent
   - Clock condition:
     - If an event \( b \) causally depends on an event \( a \) then timestamp of \( a \) must be less than the timestamp of \( b \)
     - \( a \rightarrow b \) \( \Rightarrow \) \( T(a) < T(b) \)
   - Algorithm of Lamport (1978)
     - Messages as the only means for communication
     - Fulfills clock condition

---

**Example: UNIX make command**

- Editor runs on machine A
- Compiler runs on machine B

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>32</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>30</td>
<td>48</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>36</td>
<td>56</td>
<td>70</td>
<td>80</td>
</tr>
<tr>
<td>42</td>
<td>64</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>48</td>
<td>72</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>54</td>
<td>80</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

![Clocks and file creation](image)

- Make command will not notice necessary update!

---

**Example (2)**

- Without logical clocks:
  - Send event happens before arrival: send time must be less than arrival time!
- With logical clocks:
  - Solution: adjust local clock
1 Logical Clocks (3)

- Lamport's algorithm
  - Each process has its own logical clock (a counter LC that is used for timestamping of events)
  - Logical clock ticks for each local event
    - Local event: \( \text{LC} := \text{LC} + 1 \)
    - Send event: \( \text{LC} := \text{LC} + 1; \text{send} ( \text{message}, \text{LC} ) \)
    - Receive event: \( \text{LC} := \max(\text{LC}, \text{LC}_S) + 1 \)
  - Fulfills clock condition
  - Reverse clock condition is not fulfilled!
    - \( T(a) < T(b) \Rightarrow a \rightarrow b \)

1 Logical Clocks (5)

- Does it help for the “make” example?
  - Without logical clocks:
    - A: write test.c (timestamp 10)
    - FS: test.c written
    - A: make starts compiler
    - B: write test.o (timestamp 26)
    - A: write test.o (timestamp 24)
    - FS: test.o written
    - FS: test.o written
  - With logical clocks:
    - A: write test.c (timestamp 10)
    - FS: test.c written
    - A: make starts compiler
    - B: write test.o (timestamp 26)
    - A: write test.o (timestamp 24)
    - FS: test.o written
    - FS: test.o written

1 Logical Clocks (4)

- How does it help?
  - Logging processes: timestamp log messages with local clock

1 Logical Clocks (6)

- Does it help for the “make” example?
  - Without logical clocks:
    - A: write test.c (timestamp 10)
    - FS: test.c written
    - A: make starts compiler
    - B: write test.o (timestamp 26)
    - A: write test.o (timestamp 24)
    - FS: test.o written
    - FS: test.o written
  - With logical clocks:
    - A: write test.c (timestamp 10)
    - FS: test.c written
    - A: make starts compiler
    - B: write test.o (timestamp 26)
    - A: write test.o (timestamp 24)
    - FS: test.o written
    - FS: test.o written

- NO!!
2 Clock Synchronization

- Local clocks are realized in software
  - Time chip signals interrupt that counts clock ticks
  - Local clock has a drift to UTC (Universal Coordinated Time)
  - Synchronize local clocks to minimize drift to UTC
  - Sources: DCF77, GEOS, GPS, Atomic clock

3 Vector Time

- Sometimes we would like to know whether two events are causally dependent by looking at their timestamps
  - Corresponds to reverse clock condition
  - Impossible to derive with logical clocks
  - Vector time introduced by Mattern (1989)
    - Each process $i$ of $k$ processes maintains a clock vector $V_i$ of $k$ clocks
    - Local event: $V_i[j] := V_i[j] + 1$
    - Send event: $V_i[j] := V_i[j] + 1$; send($V_i[j] + 1$)
    - Receive event: $V_i[j] := V_i[j] + 1$; receive($V_i[j] + 1$)
      - $V_i[j] := \max(V_i[j], V_i[j])$

- Comparing two time vectors:
  - $a \leq b \iff \forall i : a[i] \leq b[i]$
  - $a < b \iff (a \leq b) \land (a \neq b)$
  - $a \parallel b \iff (a < b) \land (a < b)$

4 Mutual Exclusion

- Semaphore needs coherent shared memory
  - Multicomputers cannot use a semaphore
  - Centralized semaphore server and request-reply interaction
    - Centralized component (coordinator) acts like a semaphore
    - Every process has to contact the coordinator to get access to a critical region

- Process $B$ sends a release message to the coordinator after leaving the critical region
  - Single point of failure
4 Mutual Exclusion (2)

- Distributed algorithm
  - Lamport (1978)
  - Improved by Ricart and Agrawala (1981)

- Algorithm by Ricart and Agrawala
  - Total ordering of events
    - Lamport’s logical clock value plus process ID \((time, pid)\)
    - The tuple makes timestamps of different events different and comparable (if time is equal process ID of different events is not)
  - Group of processes that may enter a critical region
  - Process that wants to enter the region has to send a message to all others:
    - \texttt{group\_send(LC, pid)}
    - Send must be reliable
    - Process waits until all other group member grant permission to enter the critical region

4 Mutual Exclusion (3)

- If a process receives a message it does the following:
  - The receiver is not in the critical region and does not want to enter it:
    - send(OK) to the original sender
  - The receiver is in the region:
    - the message is enqueued
  - The receiver is waiting for entering the critical region:
    - The receiver compares the timestamps of the incoming message with the timestamp of its own request message
      - The own timestamp is lower:
        - the message is enqueued
      - The own timestamp is higher:
        - send(OK) to the original sender
  - After leaving a critical region a process sends back an OK for all enqueued request messages and deletes those messages

4 Mutual Exclusion (4)

- Two processes want to enter the critical region at the same time
  - The process with the lowest timestamp will win
4 Mutual Exclusion (5)

- Is it really better?
  - $n$ points of failures
  - $2(n-1)$ messages
  - Group membership must be known to all other processes

- Hardly better than the centralized version
  - Shows that it is possible to solve the problem by a distributed algorithm
  - Good example for distributed algorithms

5 Election Algorithms

- Problem
  - Find out a (new) coordinator, initiator, sequencer, or something similar
  - After the run of the algorithm
    - one group member should be the coordinator,
    - all other group member should know who was elected.
  - Multiple processes may start the election, but only one process will be elected.

6 Deadlock Detection

- Problem
  - Find out whether some processes are involved in a deadlock
  - Traversing the distributed dependency graph

7 Distributed Garbage Collection

- Problem
  - Find out data object that are not referenced any more
  - Traversing the distributed reference graph

8 Echo Algorithms

- Problem
  - Distributed information to all of not fully interconnected processes and compute a function (e.g. maximum of the output of all processes)

D.7 Object-Based Distributed Systems

- So far: processes
  - Processes & message passing
  - Processes & remote procedure calls

- Object-based programming
  - Objects
  - Classes
  - Methods, method invocation

- Inheritance (object-oriented programming)

- Systems that are distributed and object-based
1 Holistic Approach

- Objects as distributable entities
  - Objects are distributed on several nodes and never split
  - Objects communicate with each other
  - Remote method invocation

Node A | Node B
---|---
Object | Reference

Object Mobility

- Objects may migrate from one node to the other
- Stubs have to be created for all references of the moved object
- Local stub pairs can be abbreviated

Node A | Node B
---|---
Object | Reference

Disadvantages

- No transparent replication as object is a centralized entity
- In general:
  - Quality-of-service requirements often need object code at the client side!
    - Replication
    - Caching
    - Bandwidth reservation
    - etc.
2 Fragmented-Object Approach

- Distributed objects consist of fragments that can be spread over multiple nodes
  - Fragments communicate with each other
  - Method invocation is always done locally (local fragment is needed)

Node A -> Node B
Node B -> Node C

Fragmented Object

---

2 Fragmented-Object Approach (2)

★ Advantages
  - More general; includes the centralized object approach
    - one fragment is the main object
    - other fragments are stubs
  - Arbitrary communication between fragments
    - group communication for fragments replicating the object’s state
    - real-time or transactional communication
    - communication with the object is always local
  - "Intelligent stubs"
    - local fragment can replicate or cache data of the object
    - local fragment can compute methods that do need little of the object’s data

Node A -> Node B
Node B -> Node C

Fragmented Object

---

2 Fragmented-Object Approach (3)

▲ Disadvantages
  - Programmer has to build up the object-internal communication by his own
    - tools and libraries may help (e.g., stub fragment generator)
    - special name services may be needed
  - System does not know about stubs
    - Somehow, the system has to load the fragment code from somewhere whereas it otherwise only has to generate a stub.

Node A -> Node B
Node B -> Node C

Fragmented Object

---

2 Fragmented-Object Approach (4)

■ Object mobility
  - Mobility is relative because the object is always accessed via a local fragment
  - Fragments may be mobile: fragments need to be replaced by one another

Node A -> Node B
Node B -> Node C

Fragmented Object
2 Fragmented-Object Approach (5)

- Example:
  - A new main fragment is built up at the side of stub fragment, takes over the essential data from the old main fragment, and replaces the stub.
  - The old main fragment is replaced by a new stub fragment.

![Fragmented Object Diagram](image-url)