CS 5523: Operating Systems

Instructor: Dr. Tongping Liu

Final Reviews (Comprehensive)

Final Exam: May 5, 2015, Tuesday
6:00pm – 8:30pm
Lecture06: Distributed Systems

- Distributed Systems: examples and definition
- OS structures in distributed systems
  - Distributed OS
  - Networked OS
  - Middleware Based systems
- Design objectives of distributed systems
  - Resource availability, transparency, openness and scalability
- Different distributed systems
- Misconceptions about distributed systems
Different Types of Distributed Systems

- **Distributed Computing** Systems: HP computing task
  - Cluster computing: similar components
  - Grid computing / Cloud computing: different components

- **Distributed Information** Systems
  - Web servers
  - Distributed database applications

- **Distributed Pervasive** Systems: instable
  - Smart home systems
  - Electronic health systems: monitor
  - Sensor networks: surveillance systems
Scalability in Distributed Systems

- Three aspects of scalability
  - **size**: number of users and/or processes
  - **geographical**: Maximum distance between nodes
  - **administrative**: Number of administrative domains

- Most systems account only, to a certain extent, for size scalability: powerful servers (supercomputer)

- Challenge nowadays: geographical and administrative scalability
Techniques for Scalability

- **Hiding communication latency: (Geographical)**
  - Use *asynchronous* communication:
    - +: separate handler for incoming response and do something while waiting.
    - -: what if there is nothing else to do

- **Distribution: splitting it to small parts**
  - Domain naming systems (DNS)
  - Decentralized data, information systems (WWW)
  - Decentralized algorithm (Distance Vector)

- **Replicate:**
  - Increase availability
  - Load balance
Lecture07: Network Communication

- Layered network models
  - OSI 7-layer model (Open System Interconnection)
- Ethernet: local area network
- Inter-network Protocols (IP)
  - Addressing and routing etc.
- TCP/UDP protocols
  - communication ports and sockets
- Multicast: more than one recipients
Approaches for Packet Delivery

**Datagram (vs. mailed letters) - UDP**
- each packet contains full network address of source-to-destination;
- no setup of paths, one-at-a-time, hop-by-hop transmission of packets,
- unreliable, e.g., Internet IP datagram in network layer

**Virtual circuits (vs. phone call) - TCP**
- set up end-to-end path, packets contains virtual circuit #,
- more reliable,
- links can be shared
TCP: Transmission Control Protocol

- TCP is *connection-oriented*.
  - 3-way handshake used for connection setup
  - Acknowledge each pack (pigback)

Connection Setup
3-way handshake

Acknowledgement
data packets
Lecture08: Application-Level Communications

- Fundamentals
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous

- Models for application communications
  - **RPC: remote procedure call**
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication
Should Servers Re-Do Operations?

- **Idempotent** operations: which can be performed **repeatedly** with the **same** effect.
  - suppose $x$ is input message $\rightarrow f(f(x)) = f(x)$
  - No state needs to maintain on the server

- Are the following operations idempotent?
  - HTTP GET ...
  - UNIX file operations: read, write etc.

  - **yes**
  - **NO**
Server Invocation Semantics (cont.)

- **Maybe**: if no reply, the client does not know if method was executed or not
- **At least once**: will guarantee that invocation be carried out at least once, but possibly more
- **At most once**: Will guarantee that RPC has been carried out at most once, but possibly none at all
  - Detect duplicated requests with sequence numbers
  - **No guarantees**: When a server crashes, the client gets no help and no promises about what happened

- Local invocation: **exactly once** - ideal case
Types of Communications

- **Asynchronous communication**
  - Sender *continues* immediately after it has submitted the request (unblocked, need a local buffer at the sender)

- **Synchronous communication**
  - Sender *blocks* until the sender receives an OK to continue; *where the OK may come?*
RPC Mechanism

Client computer

- Local return
- Unmarshal results
- Receive reply

Server computer

- Local call
- Marshal arguments
- Send request
- Execute procedure
- Unmarshal arguments
- Marshal results
- Select procedure
- Receive request
- Send reply

Communication module

client stub proc.

server stub proc.
Different problems of RPC

- Problem 1: data representations (e.g. Big Endian vs. Little Endian)
  - Using the receiver’s data representation
  - common external data representation

- Problem 2: Un/Marshaling
  - How to properly interpret messages

- Problem 3: Passing reference parameters
  - Forbid reference parameters
  - Copy the entire data structure (e.g. an entire array may be sent if the size is known). In the case of the server input parameter, it does not need to be copied back
Transmission of Continuous Media

- Different timing guarantees: 3 types of transmission

- **Asynchronous**: no restrictions with respect to *when* data is to be delivered

- **Synchronous**: define a maximum end-to-end delay for individual data packets

- **Isochronous**: define a maximum and minimum end-to-end delay (*jitter* is bounded)
Stream

- **Definition:** A (continuous) data stream is a connection-oriented communication facility that supports *isochronous* data transmission.

- **Common stream characteristics**
  - Streams are unidirectional
  - A single *source*, and one or more *sinks*
  - Often, either the sink and/or source is a wrapper around hardware (e.g., CD device, TV monitor, dedicated storage)

- **Two types of streams:**
  - *Simple*: single flow of data, e.g., audio or video
  - *Complex*: multiple data flows, e.g., stereo audio or combination audio/video
Lecture09: Remote Objects and RMI

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects;
RMI Overview

How do clients know where the remote objects are?

How do clients know where the remote objects are?

- Binding…
  - RMI register: the string name of the object, the remote object itself
  - The registry returns to the caller a reference (called stub) to the remote object.
  - Invoke methods on the object (through the stub).
RPC vs. RMI

- **Similarity:**
  - Marshaling and parameter passing

- **Difference:**
  - RPC: C based, structure based semantics. RMI: java and object-oriented
  - RPC: call remote functions, passed everything. RMI: remote/system-wide object reference and invoke methods. We can also pass and return objects that can be distributed among many JVM instances, much more powerful.
  - RMI can support dynamic invocations.

```plaintext
fobject.append(int)        Invoke(fobject, id(append), int)
```
Proxy and Skeleton

Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.
Steps in RMI

1. **Request**
   - Object A sends a request to proxy for B.

2. **Remote Reference**
   - Proxy for B sends a remote reference to the server's skeleton module.

3. **ROR**
   - Naming Service resolves the remote reference to an actual reference.

4. **Remote Reference**
   - Server's remote reference module sends the request to the server's skeleton module.

5. **ROR**
   - Object B's class is instantiated and the method is dispatched.

6. **Reply**
   - Method is executed and the result is returned to the client.

7. **Remote Reference**
   - Remote reference module in the client receives the reply.

8. **Reply**
   - Reply is forwarded to the client.

9. **Remote Reference**
   - Remote reference module in the client sends a reply to the server's remote reference module.

10. **Remote Reference**
    - Server's remote reference module sends a reply to the server's skeleton module.

11. **Dispatcher**
    - Skeleton module dispatches the reply to the client's proxy for B.

12. **Remote Reference**
    - Proxy for B sends a reply to Object A.

13. **Remote Reference**
    - Object B's remote reference module sends a reply to the server's remote reference module.

14. **Reply**
    - Server's remote reference module sends a reply to the server's skeleton module.

15. **Remote Reference**
    - Skeleton module sends a reply to the client's proxy for B.

16. **Reply**
    - Proxy for B sends a reply to Object A.

17. **Remote Reference**
    - Object A's remote reference module sends a reply to the client's remote reference module.

18. **Reply**
    - Remote reference module in the client receives the reply.
Lecture 10: Naming and Name Service

- Name and name services
  - Naming space and implementation
- Flat name and resolutions
  - Forwarding pointers
  - Distributed Hash Table (DHT): Chord
- Structure name
  - Name resolution: iterative vs. recursive
  - Case study: DNS
- Attributed-based naming
  - Directory service
  - Case study: X.500 and LDAP
Identifier

- A special name to **uniquely** identify an entity (SSN, MAC)
- A true identifier has the following three properties:
  - **P1:** Each identifier refers to at most one entity
  - **P2:** Each entity is referred to by at most one identifier
  - **P3:** An identifier always refers to the same entity (no reuse)

Addresses and identifiers are important and used for different purposes, but they are often represented in machine readable format (MAC, memory address)
Naming Systems and Their Goals

Naming Systems
- Flat names
- Structured names
- Attributed-based names

Goals
- Scalable to arbitrary size
- Have a long lifetime
- Be highly available
- Have fault isolation
- Tolerate mistrust
Flat Naming

- **Flat name**: random bits of string, no structure
  - E.g., SSN, MAC address

- **Resolution problem**:
  Given a flat (unstructured) name, how can we find/locate its associated access point and its address?

- **Solutions**:
  - Simple solutions (broadcasting)
  - Home-based approaches
  - Distributed Hash Tables (structured P2P)
  - Hierarchical location service
Home-Based Approaches

How to deal with scalability problem when locating mobile entities?

- Let a **home** keep track of where the entity is!

- How will the clients continue to communicate?
  - Home agent gives the new location to the client so it can directly communicate
    - efficient but not transparent
  - Home agent forwards the messages to new location
    - Transparent but may not be efficient
Distributed Hash Tables (DHT)

- Recall Chord from Chapter 2, which organizes many nodes into a logical ring
  - Each node is assigned a random $m$-bit identifier.
  - Every entity is assigned a unique $m$-bit key.
  - Entity with key $k$ falls under jurisdiction of node with smallest $id \geq k$ (called its successor)

- **Linearly** resolve a key $k$ to the address of $succ(k)$
  - Each node $p$ keeps two neighbors: $succ(p+1)$ and $pred(p)$
  - If $k > p$ then
    - forward to $succ(p+1)$
  - if $k \leq pred(p)$ then
    - forward $k$ to $pred(p)$
  - If $pred(p) < k \leq p$ then
    - return $p$’s address (p holds the entity)
Each node $p$ maintains a **finger table**

- at most $m$ entries (short cuts) with exponentially increasing size
  
  $FT_p[i] = \text{succ}(p + 2^{i-1})$

- $FT_p[i]$ points to the first node succeeding $p$ by at least $2^{i-1}$

- To look up a key $k$, node $p$ forwards the request to node with index $j$ satisfying
  
  $q = FT_p[j] \leq k < FT_p[j+1]$  
  
  (e.g., node 0 sends req $\rightarrow 4 \rightarrow 6$)

- If $p < k < FT_p[1]$, the request is also forwarded to $FT_p[1]$

- Need at most $O(\log N)$ steps, where $N$ is the number of nodes in the systems
DHT: Example
Structure Name

- **A name graph**
  - **Leaf node** represents a (named) entity.
  - **A directory node** is an entity that refers to other nodes: contains a (directory) table of (edge label, node identifier) pairs.
DNS issues

- Name tables change infrequently, but when they do, caching can result in the delivery of stale data.
  - Clients are responsible for detecting this and recovering
- Its design makes changes to the structure of the name space difficult. For example:
  - merging previously separate domain trees under a new root
  - moving sub-trees to a different part of the structure

Overall: it still runs well after 30 years, no need to be replaced!
Lecture 11: DS Synchronizations

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
Computer Clocks and Timing Events

- Processes on different computers can timestamp their events using their own clocks
  - Clocks on different computers may give different times
  - Computer clocks drift from perfect time and their drift rates differ from one another
NTP: basic idea
At least one machine has a UTC receiver

- Suppose we have a server with UTC receiver.
- The server has an accurate clock.
- So clients can simply contact it and get the accurate time (every $\delta/2\rho$ sec)

- A gets T1, T2, T3, T4.
- How should A adjust its clock?
- The problem is the delay which causes inaccuracy
NTP: basic idea
Suppose propagation delay is the same in both ways?

- Assume $dT_{\text{req}} = dT_{\text{res}}$
- A can estimate its offset value to B as $\theta$

$$\theta = T_3 + \frac{(T_2-T_1) + (T_4-T_3)}{2} - T_4 = \frac{(T_3-T_4) + (T_2-T_1)}{2}$$

$\theta > 0$, A is slower
$\theta < 0$, A is faster, but time cannot run backward?

Confuse: the object file is earlier than the source

Introduce the difference gradually (e.g., if time generate 100 interrupts, instead of 10ms, we add 9ms for each interrupt.)
Berkeley Algorithm
No machine has UTC receiver

- Time does not need to be the actual time…
- As long as all machines agree, then that is OK for many applications
- Gradually advance or slow down the clock…
Logical Time

- The order of two events occurring at two different computers cannot be determined based on their “local” time.

- **Problem:** How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation

- *The notion of logical time/clock is fairly general and constitutes the basis of many distributed algorithms*
“Happened Before” Relation

- Lamport first defined a “happened before” relation (→) to capture the causal dependencies between events.

- **Same process**: A → B, if A and B are events in the same process and A occurred before B.

- **Different processes**: A → B, if A is the event of sending a message \( m \) in a process and B is the event of the receipt of the same message \( m \) by another process.

- If A→B, and B → C, then A → C (happened-before relation is transitive).
“Happened Before”: Partial Order

- $a \rightarrow b$ (at $p_1$); $c \rightarrow d$ (at $p_2$); $b \rightarrow c$; also $d \rightarrow f$
- Not all events are related by the “$\rightarrow$” relation
  - $a$ and $e$ (different processes and no message chain)
  - they are not related by “$\rightarrow$”
  - they are said to be **concurrent** (written as $a \parallel e$)
Logical Clock: Example
Problem with Lamport’s Clocks

Observation: Lamport’s clocks do not guarantee that if $C(a) < C(b)$ that $a$ causally preceded $b$:

- Event $a$: $m_1$ is received at $T = 16$.
- Event $b$: $m_2$ is sent at $T = 20$.

We cannot conclude that $a$ causally precedes $b$. 
Vector Clocks

Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

- $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$.

- If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$. It is $P_i$’s knowledge of the local time at $P_j$.
Causally-Ordered Multicasting

For $P_2$, when receive $m^*$ from $P_1$, $ts(m^*) = (1,1,0)$, but $VC_2 = (0,0,0)$; $m^*$ is delayed as $P_2$ didn’t see message from $P_0$ before;

When $P_2$ receive $m$ from $P_0$, $ts(m)=(1,0,0)$, with $VC_2=(0,0,0)$ → both R1 and R2 is ok, and $m$ is delivered → $VC_2 = (1,0,0)$, then $m^*$ is delivered
Mutual Exclusion in Distributed Systems

To ensure exclusive access to some resource for processes in a distributed system
  ➢ Permission-based vs. token-based approaches

Solutions
  ➢ Centralized server;
  ➢ Decentralized, using a peer-to-peer system;
  ➢ Distributed, with no topology imposed;
  ➢ Completely distributed along a (logical) ring;
Lecture12: Consistency & Replication

- Motivations for replications
  - **Performance** and/or **fault-tolerance**

- **Data-Centric** Consistency Models
  - Continuous Consistency, Consistent Ordering of Operations

- **Client-Centric** Consistency Models
  - Eventual Consistency
  - Monotonic Reads, Monotonic Writes
  - Read Your Writes, Writes Follow Reads

- **Replica Management**
  - Replica-Server Placement, Content Replication & Placement
  - Content Distribution

- **Consistency Protocols**
  - Implementation of the consistency models
Why Replications are Needed?

Data are replicated

➢ To increase the reliability of a system:
  ✓ If one crash, we can switch to another one
  ✓ Provide better protection on the data

➢ To improve performance ➔ **Scalability**
  ✓ Scaling in numbers and in geographical area (e.g., place copies of data close to the processes using them. So clients can quickly access the content.)

Problems

➢ How to keep replicas *consistent*
  ✓ Distribute replicas
  ✓ Propagate modifications

➢ Cost >> benefit if access-to-update is very low
Replication as Scaling Technique

What if there is an update?

- Update all in an atomic way (sync replication)
- To keep replicas consistent → conflicting operations are done in the same order everywhere
  - Read–write conflict: read and write operations act concurrently
  - Write–write conflict: two concurrent write operations

Solution

- Loosen the consistency constraint so that hopefully global synchronization can be avoided
Consistency Models

- Data-Centric
  - Multiple writers may update the data store simultaneously

- Client-Centric
  - Lack of simultaneous updates.
Consistent Ordering of Operations

How to reach a global order of operations applied to replicated data so we can provide a system-wide consistent view on data store?

- Comes from concurrent programming
  - Sequential consistency
  - Causal consistency

\( W_i(x)a \): a write by process \( P_i \) to data item \( x \) with the value \( a \)
Sequential Consistency (1)

- The result of any execution is the same as if the (R/W) operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program – by Lamport.

- Behavior of two processes operating on the same data item. The horizontal axis is time.

\[
\begin{align*}
\text{P1:} & & \text{W(x)} & \text{a} \\
\text{P2:} & & \text{R(x)NIL} & \text{R(x)a}
\end{align*}
\]

it took sometime to propagate new value of x
Sequential Consistency (2)

Any valid interleaving of R and W is acceptable as long as all processes see the same interleaving of operations.

Everyone sees all W in the same order.

(a) A sequentially consistent data store.

(b) A data store that is **NOT** sequentially consistent. Why?

---

- Any valid interleaving of R and W is acceptable as long as all processes see *the same interleaving of operations*.
- Everyone *sees all W in the same order*
Causal Consistency (1)

- Weakening sequential consistency
  - NOT all, only causally related W → seen in same order
- It implies:
  - Writes that are potentially causally related must be seen by all processes in the same order.
  - Concurrent writes may be seen in a different order on different machines.
- If event b is caused by an earlier event a, a→b
  - P1: Wx P2: Rx then Wy, then Wx → Wy (potentially causally related)
Causal Consistency (2)

This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

(a) A violation of a causally-consistent store

(b) Causally but not sequentially consistent events.

Implementing causal consistency requires keeping track of which processes have seen which write → Construct a dependency graph using vector timestamps…
Client-Centric Consistency Models

- Data-centric: aiming at providing a system-wide consistent view on a data store.
  - Assumption: processes can update simultaneously the data store, thus it is necessary to provide consistency
  - Sequential is good but costive, only guarantee when using transactions or locks.

- Client-centric: lacking of simultaneous updates, or we only care about when updates happen
  - From a specific client point of view
Eventual Consistency (1)

- Most processes hardly ever perform updates while a few do updates
- How fast updates should be made available to only reading processes (e.g., DNS)
  - Consider WWW pages, not write-write conflict
    - To improve performance clients cache web pages. Caches might be inconsistent with original page for some time…
    - Eventually all will be brought up to date
  - MongoDB, CouchDB, Amazon DynamoDB and SimpleDB

- Eventual consistency:
  If no updates take place for a long time, all replicas will become consistent
Pull versus Push Protocols

- Pushing updates:
  - server-initiated, in which update is propagated regardless whether target asked for it. + good if r/w is high: read more

- Pulling updates:
  - client-initiated: + good if r/w is low: write more, read less

- We can dynamically switch between pulling and pushing using leases (a hybrid form):

- Lease is a contract in which the server promises to push updates to clients until the lease expires.
Lecture13: Fault Tolerance

- Terminology: fault, error and failures
- Fault recovery techniques
  - Redundancy: time and space
- Recovery and rollback
  - Checkpointing and stable storage
- Process resilience and reliable process groups
- Reliable communications
- Recovery in distributed systems:
  - Consistent checkpointing and message logging
Fault Tolerance Properties

- **Availability**
  - What percentage of time is a system available for use?

- **Reliability**
  - How long can a system can run **continuously without failure**? Fail 1ms out of 1our, availability > 99.9999% but not reliable

- **Safety**
  - Small failures should not have catastrophic effects

- **Maintainability**
  - How easy is it to repair faults?
  - High reliability =\= high availability!

A time interval An instant (point) in time
Fault Management

- **Fault prevention**
  - prevent the occurrence of a fault

- **Fault tolerance**
  - build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask the presence of faults)

- **Fault removal**
  - reduce the presence, number, seriousness of faults

- **Fault forecasting**
  - estimate the present number, future incidence, and the consequences of faults
Fault Tolerance Techniques

- **Redundancy** and agreement
  - Hiding effect of faults

- **Recovery** and rollback
  - Bringing system to a consistent state
Redundancy Techniques

- **Redundancy** is the key technique to tolerance faults

- **Information redundancy**
  - e.g., parity and Hamming codes

- **Time redundancy**
  - e.g., re-execution or execute secondary/backup copy

- **Physical (software/hardware) redundancy**
  - e.g., extra cpus, multi-versions softwares
Triple Modular Redundancy (TMR)

If A2 fails → V1: majority vote → all B get good result

What if V1 fails?!
Correct results are obtain via majority vote

Mask ONE fault

(b)
Level of Redundancy

- Depends on
  - How many faults can a system handle?
  - What kind of faults can happen?

- k-fault tolerant system
  - Can handle k faulty components

- Assume crash failure semantics (i.e., \textit{fail-silent})
  - $k + 1$ components are needed to survive $k$ failures
Level of Redundancy (cont.)

- Assume arbitrary (but non-malicious) failure semantics and group output defined by voting
  - Independent component failures → possible same results
  - \(2k+1\) components are needed to survive \(k\) component failures (majority vote)

- Assume Byzantine (malicious) failure semantics and group output defined by voting
  - Faulty components cooperate to cheat!!!
  - \(3k+1\) components are needed to tolerate \(k\) failures → two-thirds are needed for the agreement from other \(2k+1\) non-faulty components;
Fault Recovery

- **Main idea**: when a failure occurs, we need to bring the system into an **error-free** state

- **Forward recovery**
  - Find a new state from which system continue operation
  - E.g., Error-correction codes
  - Problem: how to correct errors and move to a new state

- **Backward recovery**
  - Bring the system back into a **previous** error-free state
  - E.g., packet retransmission
  - Problem: keeping error-free state (checkpoints)
Recovery with Checkpoints

- Initial state
- Recovery line
- Checkpoint
- Failure

Message sent from P2 to P1
Inconsistent collection of checkpoints

Time
Independent Checkpointing

Each process independently takes checkpoints

- Let $CP[i](m)$ denote $m^{th}$ checkpoint of process $Pi$ and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process $Pi$ sends a message in interval $INT[i](m)$, it piggybacks $(i,m)$
- When process $Pj$ receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

If process $Pi$ rolls back to $CP[i](m-1)$, $Pj$ must roll back to $CP[j](n-1)$.