Distributed Systems

Presentation 1
What is Distributed Computing

Distributed Computing

- Tightly-coupled System
  Parallel Computing

- Loosely Coupled System
  Distributed Computing
  - Grid Computing
  - P2P Computing
  - Mobile Computing
  - Pervasive Computing
Models of Distributed Computing

Computation Models
- Message-passing Systems
- Shared memory Systems

Timing Models
- Asynchronous
- Synchronous
Message-passing System Model

Configuration

Event

- Computation Event
- Delivery Event

Execution: Sequence of configuration alternating with events, which satisfies all required safety conditions for a particular system type.

Admissible Execution: An Execution that satisfies all required liveness conditions.

Schedule
Complexity Measures

- Message Complexity
- Time complexity
Algorithm: Spanning tree broadcast algorithm.

Initially \(<M>\) is in transit from \(p_r\) to all its children in the spanning tree.

Code for \(p_r\):

upon receiving no message:

send \(<M>\) to all children

terminate

Code for \(p_i, \ 0 \leq i \leq n - 1, \ i \neq r:\)

upon receiving \(<M>\) from parent:

send \(<M>\) to all children

terminate
Example: Algorithm 2

Algorithm: algorithm to construct a spanning tree:

code for processor $p_i$, $0 \leq i \leq n - 1$.

Initially $parent = NIL$, $children = NIL$, and $other = NIL$.

upon receiving no message:

if $p_i = p_r$ and $parent = NIL$ then \hspace{1em} // root has not yet sent $<M>$

send $<M>$ to all neighbors

$parent := p_i$

upon receiving $<M>$ from neighbor $p_j$:

if $parent = NIL$ then \hspace{1em} // $p_i$ has not received $<M>$ before

$parent = P_j$

send $<parent>$ to $p_j$

send $<M>$ to all neighbors except $p_j$

else send $<already>$ to $p_j$
upon receiving <parent> from neighbor $p_j$:
   add $p_j$ to *children*
   if (*children* U *other*) contains all neighbors except *parent* then
      terminate

upon receiving <already> from neighbor $p_j$:
   add $p_j$ to *other*
   if (*children* U *other*) contains all neighbors except *parent* then
      terminate
Leader election problem can’t be solved for Anonymous ring.

Two types of algorithms for Leader Election Problem

- Uniform
- Non-uniform
Asynchronous Ring

$O(n^2)$ uniform algorithm
O(n log n) algorithm

**Algorithm:** Asynchronous leader election: code for processor \( p_i, 0 \leq i \leq n-1 \).

Initially, \( asleep = \text{true} \)

upon receiving no message:

if \( asleep \) then

\( asleep = \text{false} \)

send \((\text{probe, } id, 0, 1)\) to left and right

upon receiving \((\text{probe, } j, k, d)\) from left (resp., right):

if \( j = id \) then terminate as the leader

if \( j > id \) and \( d < 2^k \) then

send \(<\text{probe, } j, k, d + 1>\) to right (resp., left) // forward the message

if \( j > id \) and \( d = 2^k \) then

send \(<\text{reply, } j, k>\) to left (resp., right) // reply to the message

// message is swallowed
O(n log n) algorithm (Cont.)

upon receiving (reply, j, k) from left (resp., right):

if j != id then
    send (reply, j, k) to right (resp., left)  
    // forward the reply
else  // reply is for own probe
    if already received (reply, j, k) from right (resp., left) then
        if k != log (n-1)
            send (probe, id, k +1,1)  // phase k winner
        else
            declare itself as the leader.
            send termination message to all the other processors.

The lower bound of the message complexity for the Leader election algorithm is O(n log n).
Synchronous Ring

The upper bound and the lower bound of the message complexity Leader Election Algorithm is $O(n)$.

A non-uniform algorithm
Distributed Systems

Presentation 2
Components:

n Processors

m Registers

Each register has a type, which specifies:

1. The values that can be taken on by the register
2. The operations that can be performed on the register
3. The value to be returned by each operation (if any)
4. The new value of the register resulting from each operation
Shared Memory System Model (Cont.)

- Configuration
  \[ C = (q_0, \ldots, q_{n-1}, r_0, \ldots, r_{m-1}) \]

- Events

- Execution
  \[ C_0, \emptyset_1, C_1, \emptyset_2, C_2, \emptyset_3, \ldots \]

- Schedule
  \[ \sigma = i_1, i_2, \ldots \]
Complexity Measures

- Space Complexity
- Time Complexity
The Mutual Exclusion Problem

Critical Section

Program of a processor is partitioned into the following sections:

• Entry
• Critical
• Exit
• Remainder

Assumptions:

- The variables accessed in the entry and Exit sections are not accessed in the Critical and the Remainder Section.
- No processor stays in the Critical section forever.
Conditions of Mutual Exclusion

- Mutual exclusion
- No Deadlock
- No Lockout
Example: Binary Test&Set Registers

**test&set(V : memory address)** returns binary value:

- temp = V
- V = 1
- return (temp)

**reset(V : memory address):**

- V = 0

**Algorithm: Mutual exclusion using a test&set register:** (code for every processor)

Initially V equals 0

**(Entry):**
- wait until test&set( V) = 0

**(Critical Section)**

**(Exit):**
- reset(V)

**(Remainder)**
Example: Read-Modify-Write Registers

\( \text{rmw}(V : \text{memory address}, f: \text{function}) \text{ returns value:} \)

\[
\begin{align*}
t & = V \\
V & = f(V) \\
\text{return (temp)}
\end{align*}
\]

**Algorithm:** Mutual exclusion using a read-modify-write register (code for every processor)  
Initially \( V = <0, 0> \)

(Entry):

\[
\begin{align*}
\text{position} & = \text{rmw}(V, <V.\text{first}, V.\text{last} + 1>) & \text{// enqueueing at the tail} \\
\text{repeat} \\
\text{queue} & = \text{rmw}(V, V) & \text{// read head of queue} \\
\text{until} (\text{queue.first} = \text{position.last}) & \text{// until becomes first}
\end{align*}
\]

(Critical Section)

(Exit):

\[
\text{rmw}(V, <V.\text{first} + V.\text{last}>) & \text{// dequeueing}
\]

(Remainder)
Local spinning

**Algorithm: Mutual exclusion using local spinning:** (code for every processor)

Initially \( \text{Last} = 0; \) Flags\([0]\) = has-lock; Flags\([i]\) = must-wait, \(0 < i < n\).

**(Entry):**

\[
\text{my-place} := \text{rmw} (\text{Last}, \text{Last} + 1 \mod n)
\]

wait until ( Flags[my-place] = has-lock)

Flags[my-place] = must-wait

**(Critical Section)**

**(Exit):**

Flags[my-place + 1 \mod n] = has-lock

**(Remainder)**
Distributed Systems

Presentation 3
Failures in Synchronous Systems

- Crash Failure
- Byzantine Failure
Synchronous Systems with Crash Failure

Assumptions

Communication graph is complete.

Links are completely reliable.

Formal Model

For a f-resilient system, at most f processors can fail.

In the last round in which a faulty processor has a computation event, an arbitrary set of the outgoing messages are delivered.

The Consensus Problem

- Termination
- Agreement
- Validity
A Simple Algorithm

**Algorithm:** Consensus algorithm in the presence of crash failures:

code for processor $p_i$, $0 \leq i \leq n - 1$.

Initially $V = \{x\}$

// $V$ contains $p_i$'s input

round $k$, $1 \leq k \leq f + 1$:

send \{v in $V$ : $p_i$ has not already sent v\} to all processors

receive $S_j$ from $p_j$, $0 \leq j \leq n - 1$, $j \neq i$

\[
V = \bigvee_{j=0}^{n-1} S_j
\]

if $k = f + 1$ then $y = \min(V)$ // decide

The above algorithm solves the consensus problem in the presence of crash failures within $f + 1$ rounds.
Lower Bound on the Number of Rounds

**Theorem:** Any consensus algorithm for \( n \) processors that is resilient to \( f \) crash failures requires at least \( f + 1 \) rounds in some admissible execution, for all \( n \geq f + 2 \).
Synchronous Systems with Byzantine Failure

Formal Model

The Consensus Problem

- Termination
- Agreement
- Validity

Lower Bound on the Number of Faulty Processors

Theorem: In a system with n processors and f Byzantine processors, there is no algorithm that solves the consensus problem if n < 3f.
An Exponential Algorithm

f is the upper bound on the number of failures.

\[ n \geq 3f + 1. \]

The algorithm takes exactly \( f + 1 \) rounds.

The exponential information gathering Tree
Validity Condition

**Lemma.** For every tree node label \( (pi) \) of the form \( (pi)'j \), where \( p_j \) is non-faulty, \( \text{resolve}_i(pi) = \text{tree}_j(pi') \), for every non-faulty processor \( p_i \).

Agreement Condition

**Common**

**Common Frontier**

**Lemma.** Let \((pi)\) be a node. If there is a common frontier in the sub-tree rooted at \((pi)\), then \((pi)\) is common.

**Theorem:** There exists an algorithm for \( n \) processors that solves the consensus problem in the presence of Byzantine failures within \( f + 1 \) rounds using exponential size messages, if \( n > 3f \).
Algorithm: A polynomial consensus algorithm in the presence of Byzantine failures: \((n > 4f)\)

code for \(p_i\), \(0 \leq i \leq n - 1\).

Initially \(\text{pref}[i] = x\) // initial preference for self is for own input
and \(\text{pref}[j] = \text{DEFAULT}\) for any \(j \neq i\) // default for others

**round 2k - 1, 1 \leq k \leq f + 1:** // first round of phase \(k\)

send \(<\text{pref}[i]>\) to all processors
receive \(<v_j>\) from \(p_j\) and assign to \(\text{pref}[j]\), for all \(0 \leq j \leq n - 1, j \neq i\)

let \(maj\) be the majority value of \(\text{pref}[0], \ldots, \text{pref}[n - 1]\) (DEFAULT if none)
let \(mult\) be the multiplicity of \(maj\)

**round 2k, 1 \leq k \leq f + 1:** // second round of phase \(k\)

if \(i = k\) then send \(<maj>\) to all processors //king of this phase
receive \(<\text{king-maj}>\) from \(p_k\) (DEFAULT if none)

if \(mult > n/2 + f\)
    then \(\text{pref}[i] = maj\)
else \(\text{pref}[i] = \text{king-maj}\)

if \(k = f + 1\) then \(y = \text{pref}[i]\) // decide
Validity Property

**Lemma.** If all non-faulty processors prefer \( v \) at the beginning of phase \( k \), then they all prefer \( v \) at the end of phase \( k \), for all \( k, 1 \leq k \leq f + 1 \).

Agreement Property

**Lemma.** Let \( g \) be a phase whose king \( p_g \) is non-faulty. Then all non-faulty processors finish phase \( g \) with the same preference.

**Theorem:** There exists an algorithm for \( n \) processors that solves the consensus problem in the presence of Byzantine failures within \( 2(f + 1) \) rounds using constant size messages, if \( n > 4f \).
Impossibilities in Asynchronous Systems

Shared Memory

The Wait-Free case

*Theorem:* There is no wait-free algorithm for solving the consensus problem in an asynchronous shared memory system with $n$ processors.

The General Case

*Theorem:* There is no consensus algorithm for a read/write asynchronous shared memory system that can tolerate even a single crash failure.

Message Passing

*Theorem 5.25* There is no algorithm for solving the consensus problem in an asynchronous message-passing system with $n$ processors, one of which may fail by crashing.
Distributed Systems

Presentation 4
Capturing Causality

Causality relations in asynchronous message-passing system

Some Basic Concepts:

A **partial order** is a binary relation $R$ over a $P$ which is *reflexive*, *anti-symmetric*, and *transitive*.

**Partially Ordered Set**

**Example:** The set of natural numbers equipped with the (divides) relation.

A **Total order, Linear order or Simple order** on a set $P$ is any binary relation $R$ on $P$ that is *anti-symmetric*, *transitive*, and *total*.

**Totally Ordered Set**

**Example:** real numbers ordered by the standard less than ($<$) or greater than ($>$) relations.
The Happens-Before Relation

Given two events $e_1$ and $e_2$ in an execution, $e_1$ happens before $e_2$, denoted by $e_1 \Rightarrow e_2$, if one of the following conditions holds:

1. $e_1$ and $e_2$ are events by the same processor $p_i$, and $e_1$ occurs before $e_2$ in that execution.

2. $e_1$ is the send event of the message $m$ from $p_i$ to $p_j$, and $e_2$ is the receive event of the message $m$ by $p_j$.

3. There exists an event $e$ such that $e_1 \Rightarrow e$ and $e \Rightarrow e_2$.

Happens-Before relation is an Partial Order.
Casual Shuffle

**Definition.** Given an execution segment \( \alpha = \text{exec}(C, \sigma) \), a permutation \( \Pi \) of a schedule \( \sigma \) is a causal shuffle of \( \alpha \) if

1. For all \( i, 0 < i < n-1 \), \( \sigma \mid i = \Pi \mid i \), and

2. If a message \( m \) is sent during processor \( p_i \)'s (computation) event \( \phi \) in \( \alpha \), then in \( p_i \), \( \phi \) precedes the delivery of \( m \).

**Lemma.** Let \( \alpha = \text{exec}(C, \sigma) \) be an execution fragment. Then any permutation of the events in \( \sigma \) that is consistent with the happens-before relation of \( \alpha \) is a causal shuffle of \( \alpha \).

**Lemma.** Let \( \alpha = \text{exec}(C, \sigma) \) be an execution fragment. Let \( \Pi \) be a causal shuffle of \( \sigma \). Then \( \alpha' = \text{exec}(C, \Pi) \) is an execution fragment and is similar to \( \alpha \).
Logical Clocks

Logical Timestamp $LT(e)$

To capture the happens-before relation, we require an irreflexive partial order “$<$“ on the timestamps, such that for every pair of events, $e_1$ and $e_2$,

$$\text{if } e_1 \Rightarrow e_2, \text{ then } LT(e_1) < LT(e_2)$$

**Theorem.** Let $\alpha$ be an execution, and let $e_1$ and $e_2$ be two events in $\alpha$. If $e_1 \Rightarrow e_2$, then $LT(e_1) < LT(e_2)$.

If $LT(e_1) \geq LT(e_2)$ then $e_1 \nRightarrow e_2$

It is possible that $LT(e_1) < LT(e_2)$, but $e_1 \nRightarrow e_2$

Happens-before relation is a partial order, but the logical timestamps are totally ordered “$<$“ relation.
Non-causality: Two events $e_1$ and $e_2$ are concurrent in execution $\alpha$, denoted by $e_1 \parallel_\alpha e_2$, if $e_1 \not\Rightarrow e_2$ and $e_2 \not\Rightarrow e_1$.

A partial ordering is needed to describe the non-causality.
Vector Clocks

Vector timestamps provide a way to capture causality and non-causality.

Vector Clock $\text{VC}_i$

For every processor $p_j$, in every reachable configuration, $\text{VC}_j[i] < \text{VC}_i[i]$, for all $i, 0 < i < n - 1$.

**Theorem.** Let $\alpha$ be an execution, and let $e_1$ and $e_2$ be two events in $\alpha$. If $e_1 => e_2$, then $\text{VC}(e_1) < \text{VC}(e_2)$.

**Theorem.** Let $a$ be an execution, and let $e_1$ and $e_2$ be two events in $a$. If $\text{VC}(e_1) < \text{VC}(e_2)$, then $e_1 => e_2$. 
Given two events $e_1$ and $e_2$ in an execution $\alpha$, $e_1$ happens before $e_2$, denoted $e_1 \Rightarrow e_2$, if one of the following conditions holds:

1. $e_1$ and $e_2$ are events by the same processor $p_i$, and $e_1$ occurs before $e_2$ in $\alpha$.

2. $e_1$ and $e_2$ are conflicting events, that is, both access the same shared variable and one of them is a write, and $e_1$ occurs before $e_2$ in $\alpha$.

3. There exists an event $e$ such that $e_1 \Rightarrow e$ and $e \Rightarrow e_2$.

The notion of a causal shuffle can be adapted to the shared memory model.
Clock Synchronization

Hardware Clock

**Assumption:** Hardware clocks have no drifts.

**Definition.** A view with clock values of a processor $p_i$ (in a model with hardware clocks) consists of an initial state of $p_i$, a sequence of events (computation and deliver) that occur at $p_i$ and a hardware clock value assigned to each event.

**Definition.** A timed view with clock values of a processor $p_i$ (in a model with hardware clocks) is a view with clock values together with a real time assigned to each event. The assignment must be consistent with the hardware clock having the form $HC_i(t) = t + c_i$; for some constant $c_i$. 

\[ HC_i(t) = t + c_i \]
Merging of the Time Views of the Processors

**Definition.** Let $\alpha$ be a timed execution with hardware clocks and let $x$ be a vector of $n$ real numbers. Define $\text{shift}(\alpha, x)$ to be $\text{merge}(\eta_0, \eta_1, \ldots, \eta_{n-1})$, where $\eta_i$ is the timed view obtained by adding $x_i$ to the real time associated with each event in $\alpha | i$.

<table>
<thead>
<tr>
<th>Real Time</th>
<th>$t - x_i$</th>
<th>$t$</th>
<th>$t + x_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{HC}_i(t) = T$

<table>
<thead>
<tr>
<th>$x_i &gt; 0$</th>
<th>$p_i$ shifted</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$x_i &lt; 0$</th>
<th>$p_i$ shifted</th>
</tr>
</thead>
</table>

$\text{HC}_i(t - x_i) = T^*$

$\text{HC}_x(t + x_i) = T$

**Lemma.** Let $\alpha$ be a timed execution with hardware clocks $\text{HC}_i$, $0 < i < n - 1$, and $x$ be a vector of $n$ real numbers. In $\text{shift}(\alpha, x)$:

(a) the hardware clock of $p_i$, $\text{HC}'_i$, is equal to $\text{HC}_i - x_i$, $0 < i < n - 1$, and

(b) every message from $p_i$ to $p_j$ has delay $\delta - x_i + x_j$, where $\delta$ is the delay of the message in $a$, $0 < i, j < n - 1$. 
Clock Synchronization Problem

Hardware Clock $HC_i(t)$

Adjusted Clock $AC_i(t)$

$$AC_i(t) = HC_i(t) + \text{adj}_i(t).$$

**Achieving $\varepsilon$-Synchronized Clocks:** In every admissible timed execution, there exists real time $t_f$ such that the algorithm has terminated by real time $t_f$, and, for all processors $p_i$ and $p_j$, and all $t > t_f$, $|AC_i(t) - AC_j(t)| < \varepsilon$.

$\varepsilon$ is the **Clock skew**.

Maximum message delay $d$

Uncertainty in the message delay $u$
The Two Processors Case

How to estimate the delay in the delivery of the message?

The best estimated delay is \((d - u/2)\).

\[
d - u \leq \delta \leq d \implies |\delta - (d - u/2)| \leq u/2
\]
The best skew that can be achieved in the worst case by a clock synchronization algorithm for two processors is \( \frac{u}{2} \).

\[ \alpha' = \text{shift} (\alpha, <-u, 0>) \]
**Algorithm:** A clock synchronization algorithm for n processors:

code for processor pi, 0 < i < n - 1.

initially diff[i] = 0

at first computation step:

send HC (current hardware clock value) to all other processors.

upon receiving message T from some pj:

\[
\text{diff}[j] := T + d - u/2 - HC
\]

if a message has been received from every other processor then

\[
adj = \frac{1}{n} \sum_{k=0}^{n-1} \text{diff}[k]
\]

The above algorithm achieves \(u(1 – 1/n)\)-synchronization for n processors.

**Theorem:** For every algorithm that achieves \(\varepsilon\)-synchronized clocks, \(\varepsilon\) is at least \(u(1 – 1/n)\).
Timeout Parameter

![Graph showing the relationship between Prob, Timeout Parameter, and Delay.](image-url)
Distributed Systems

Presentation 5
Objectives

- To study tools and abstraction for simplifying the design of distributed algorithms.

- To modify our model to handle specifications and implementation of distributed algorithms.

- To put our focus on the interface between an algorithm (equivalently, the processor) and the external world.
Problem Specification

A problem is specified at the interface between an algorithm and the external world.

A **Problem Specification** $P$ is

- A set of inputs $in(P)$
- A set of outputs $out(P)$
- A set of allowable sequences $seq(P)$

**Example:** Mutual Exclusion Problem.

Inputs: $T_i$ and $E_i$

Outputs: $C_i$ and $R_i$

A sequence $\alpha$ of inputs and outputs is in the set of allowable sequences iff

- $\alpha | i$ cycles through $T_i$, $C_i$, $E_i$, $R_i$ in that order
- Whenever $C_i$ occurs, the most recent preceding output for any other $j$ is not $C_j$
Objective:

To provide communication system in software

Communication System is interposed between the processors.

The communication system will be different for different situation

• Different interface
• Different ordering
• Reliability
Asynchronous Point-to-point Message Passing

The interface to an asynchronous point-to-point message-passing system is with two types of events:

- \( \text{send}_i(M) \)
- \( \text{recv}_i(M) \)

There exists a mapping \( \kappa \) from the set of messages appearing in all the \( \text{recv}_i(M) \) events, for all \( i \), to all the set of messages appearing in \( \text{send}_i(M) \) events, for all \( i \), such that each message \( m \) in a recv event is mapped to a message with the same content appearing in an earlier send event, and the following three properties are satisfied:

- Integrity
- No Duplicates
- Liveness
The interface to a basic asynchronous broadcast service is with two types of events:

- $bc\text{-}send_i(m)$
- $bc\text{-}recv_i(m, j)$

There exists a mapping $\kappa$ from each $bc\text{-}recv_i(m, j)$ events to an earlier $bc\text{-}send_j(m)$ events, with the following three properties:

- Integrity
- No Duplicates
- Liveness
A system consists of a collection of $n$ processors (or nodes), $p_0$ through $p_{n-1}$, a communication system $C$ linking the nodes, and the environment $E$. 

Node Input
Configuration

Execution

- Configuration $C_0$ is an initial configuration.

- For each $i \geq 1$, event $\phi_i$ is enabled in configuration $C_{i-1}$ and configuration $C_i$ is the result of $\phi_i$ acting on $C_{i-1}$. In more detail, every state component is the same in $C_i$ as it is in $C_{i-1}$, except for the (at most two) processes for which $\phi_i$ is an event.

- For each $i \geq 1$, if event $\phi_i$ is not a node input, then $i > 1$ and it is on the same node as event $\phi_{i-1}$. Thus the first event must be a node input, and every event that is not a node input must immediately follow some other event on the same node.

- For each $i \geq 1$, if event $\phi_i$ is a node input, then no event (other than a node input) is enabled in $C_{i-1}$. Thus a node input does not occur until all the other events have "played out" and no more are enabled.

Schedule
Admissibility Conditions

• An execution is **fair** if every event, other than a node input, that is continuously enabled eventually occurs.

• An execution is **user compliant** for problem specification $P$, if the environment satisfies the input constraints of $P$.

• An execution $\alpha$ is **correct** for communication system $C$ if bot($\alpha$) is an element of seq($C$).

We define an execution to be **($P, C$)-admissible** if it is **fair**, **user compliant** for problem specification $P$, and **correct** for communication system $C$. 
Global Simulation

Communication system \( C_1 \) globally simulates (or simply simulates) communication system \( C_2 \) if there exists a collection of processes, one for each node, called \( Sim \) (the simulation program) that satisfies the following:

1. The top interface of \( Sim \) is the interface of \( C_2 \).
2. The bottom interface of \( Sim \) is the interface of \( C_1 \).
3. For every \((C_2, C_1)\)-admissible execution \( \alpha \) of \( Sim \), there exists \( \sigma \) sequence \( \sigma \) in \( \text{seq}(C_2) \) such that \( \sigma = \text{top}(\alpha) \).
Local Simulation

- An execution $\alpha$ is locally user compliant for problem specification $P$ if, the environment satisfies the input constraints of $P$ on a per node basis, but not necessarily globally.

- An execution is $(P, C)$-locally-admissible if it is fair, locally user compliant for $P$, and correct for the communication system $C$.

Communication system $C_1$ locally simulates communication system $C_2$ if there exists a collection of processes, one for each node, called $Sim$ (the simulation program) that satisfies the following:

1. The top interface of $Sim$ is the interface of $C_2$

2. The bottom interface of $Sim$ is the interface of $C_1$.

3. For every $(C_2, C_1)$-locally-admissible execution $\alpha$ of $Sim$, there exists a sequence $\sigma$ in $\text{seq}(C_i)$ such that $\sigma | i = \text{top}(\alpha) | i$ for all $i$, $0 \leq i \leq n - 1$. 
Distributed Systems

Presentation 6
Quality of Service

- The type of ordering
- The degree of fault tolerance

The interface to a basic asynchronous broadcast service is with two types of events:

- $bc\text{-}send_i(m, \text{qos})$
- $bc\text{-}recv_i(m, j, \text{qos})$
Broadcast Service Quality: Ordering

**Single-Source FIFO:** For all messages $m_1$ and $m_2$ and all processors $p_i$ and $p_j$, if $p_i$ sends $m_1$ before it sends $m_2$, then $m_2$ is not received at $p_j$ before $m_1$ is.

**Totally Ordered:** For all messages $m_1$ and $m_2$ and all processors $p_i$ and $p_j$, if $m_1$ is received at $p_i$ before $m_2$ is, then $m_2$ is not received at $p_j$ before $m_1$ is.

Given a sequence of bc-send and bc-recv events, message $m_1$ is said to happen before message $m_2$ if either:

- The bc-recv event for $m_1$ happens before the bc-send event for $m_2$, or
- $m_1$ and $m_2$ are sent by the same processor and $m_1$ is sent before $m_2$.

**Causally Ordered:** For all messages $m_1$ and $m_2$ and every processor $p_i$, if $m_1$ happens before $m_2$, then $m_2$ is not received at $p_i$ before $m_1$ is.
What are the relationships between these three ordering requirements?

- Causally ordered implies single-source FIFO, but does not imply totally ordered.
- Totally ordered does not imply causally ordered or single-source FIFO,
- Single-source FIFO does not imply causally ordered or totally ordered.

If a broadcast service provides total ordering as well as single-source FIFO ordering, then it is causally ordered.
There must be a partitioning of the processor indices into "faulty" and "nonfaulty" such that there are at most $f$ faulty processors, and the mapping $k$ from $bc$-$recv(m)$ events to $bc$-$send(m)$ events must satisfy the following properties:

- Integrity
- No Duplicates
- Non faulty Liveness
- Faulty Liveness

Different kinds of Broadcast

- Atomic broadcast or Total broadcast.
- FIFO atomic broadcast
- Causal atomic broadcast
Implementing a Broadcast Service

Assumption: Underlying message system is asynchronous and point-to-point.

Basic Broadcast Service

Implemented on top of an asynchronous point-to-point message system with no failures.

Single Source FIFO Ordering

Implemented on top of basic broadcast.
Totally Ordered Broadcast

An Asymmetric Algorithm

- implemented on top of Basic Broadcast
- relies on a central coordinator.

A symmetric Algorithm

- implemented on the top of the single-source FIFO broadcast.
Algorithm 1: Totally ordered broadcast algorithm: code for $p_i$, $0 \leq i \leq n - 1$.

Initially $ts[j] = 0$, $0 \leq j \leq n - 1$, and $pending$ is empty.

when $bc$-send$_i(m, to)$ occurs:

\[
\begin{align*}
& ts[i] := ts[i] + 1 \\
& \text{add } (m, ts[i], i) \text{ to pending} \\
& \text{enable } bc$-send$_i(<m, ts[j]>, ssf)
\end{align*}
\]

when $bc$-recv$_j(<m,T>, j, ssf), j \neq i$, occurs:

\[
\begin{align*}
& ts[j] := T \\
& \text{add } (m, T, j) \text{ to pending} \\
& \text{if } T > ts[i] \text{ then} \\
& \quad ts[i] := T \\
& \quad \text{enable } bc$-send$_i(<ts-up,T>, ssf)
\end{align*}
\]

when $bc$-recv$_j(<ts-up, T>, j, ssf), j \neq i$, occurs:

\[
\begin{align*}
& ts[j] := T
\end{align*}
\]

enable $bc$-recv$_i(m, j, to)$ when

\[
\begin{align*}
& <m, T, j> \text{ is the entry in } pending \text{ with the smallest } (T, j) \\
& T \leq ts[k] \text{ for all } k \\
\end{align*}
\]

result: remove $<m, T, j>$ from pending
Algorithm 2  Causally ordered broadcast algorithm: code for p_i, 0 < i < n - 1.

Initially vt[j] = 0, 0 ≤ j ≤ n - 1, and pending is empty

when bc-send_i(m, co) occurs:

vt[i] = vt[i] + 1
enable bc-recev_i({m, co})
enable bc-send_i((m, vt), basic)

when bc-recev_j(<m, v>), j, basic), j ≠ i, occurs:

add <m, v, j> to pending

enable bc-recev; (m, j, co) when:
(m, v, j) is in pending

v[j] = vt[j] + 1
v[k] ≤ vt[k] for all k ≠ i

result: remove <m, v, j> from pending

vt[j] := vt[j] + 1
Causality without Total Ordering
Algorithm 3 Reliable broadcast algorithm: code for $p_i$, $0 < i < n - 1$.

when $bc$-send$_i(m$, reliable$)$ occurs:

  enable $bc$-send$_i(<m$, $i>$, basic$)$

when $bc$-recv$_i(<m$, $k>$, $j$, basic$)$ occurs:

  if $m$ was not already received then

    enable $bc$-send$_i(<m$, $k>$, basic$)$

    enable $bc$-recv$_i(m$, $k$, reliable$)$
Quality of Service

- The type of ordering
- The degree of fault tolerance

The interface to a basic asynchronous broadcast service is with two types of events:

- $bc\text{-}send_i(m, G, qos)$
- $bc\text{-}recv_i(m, j, qos)$
Ordering and reliability

Ordering

- Single Source FIFO
- Totally Ordered
- **Multiple-Group Ordering:** Let \( m_1 \) and \( m_2 \) be messages. For any pair of processors \( p_i \) and \( p_j \), if the events \( mc\text{-recv}(m_1) \) and \( mc\text{-recv}(m_2) \) occur at \( p_i \) and \( p_j \), then they occur in the same order.
- Causally Ordered

Reliability

- Integrity
- No Duplicates
- Nonfaulty Liveness
- Faulty Liveness
Distributed Systems

Presentation 7
Distributed shared memory is a model for interprocess communication that provides the illusion of a shared memory on top of a message passing system.

The simulation program, which runs on top of the message system providing the illusion of shared memory is called the **Memory Consistency System (MCS)**.
Shared Object

Operation – Pairs of invocation and matching responses

Sequential Specification – Set of operations and a set of legal sequences of operations.

Example: Read/Write object X

- The invocation for a read is \text{read}_i(X) and responses are \text{return}_i(X, v), where i indicates the node and v the return value.

- The invocations for a write have the form \text{write}_i(X, v), where v is the value to be written, and the response is \text{ack}_i(X).

- A sequence of operations is legal if each read returns the value of the most recent preceding write, if there is one, and otherwise returns the initial value.
Inputs – invocations on shared objects

Outputs – responses from the shared object

For a sequence $\sigma$ to be in the allowable set, the following properties must be satisfied:

**Correct interaction:** For each $p_i$, $\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation. This condition imposes constraints on the inputs.

**Liveness:** Every invocation has a matching response.

**Linearizability:** There exists a permutation $\Pi$ of all the operations in a such that

1. For each object $O$, $\Pi|O$ is legal (i.e., is in the sequential specification of $O$)
2. If the response of operation $o_1$ occurs in $\sigma$ before the invocation of operation $o_2$, then $o_1$ appears before $o_2$ in $\Pi$. 
Examples:

Processor $p_0$ and $p_1$

Shared registers $x$ and $y$, both initially 0.

$\sigma_1 = \text{write}_0(x,1) \text{ write}_1(y,1) \text{ ack}_0(x) \text{ ack}_1(y) \text{ read}_0(y) \text{ read}_1(x) \text{ return}_0(y,1) \text{ return}_1(x,1)$

$\Pi_1 = w_0 w_1 r_0 r_1$

- Linearizable.

$\sigma_2 = \text{write}_0(x,1) \text{ write}_1(y,1) \text{ ack}_0(x) \text{ ack}_1(y) \text{ read}_0(y) \text{ read}_1(x) \text{ return}_0(y,0) \text{ return}_1(x,1)$

- Not Linearizable.
A sequence $\sigma$ of invocations and responses is sequentially consistent if there exists a permutation $\Pi$ of the operations in $a$ such that

1. For every object $O$, $\Pi | O$ is legal, according to the sequential specification of $O$.

2. If the response for operation $o_1$ at node $p_i$ occurs in $\sigma$ before the invocation for operation $o_2$ at node $p_i$, then $o_1$ appears before $o_2$ in $\Pi$, equivalently, $\sigma | i = \Pi | i$. 
Example:

$$\sigma_2 = \text{write}_0(x, 1) \text{ write}_1(y, 1) \text{ ack}_0(x) \text{ ack}_1(y) \text{ read}_0(y) \text{ read}_1(x) \text{ return}_0(y, 0) \text{ return}_1(x, 1)$$

$$\Pi_2 = w_0 r_0 w_1 r_1$$

Sequentially consistent.

$$\sigma_3 = \text{write}_0(x, 1) \text{ write}_1(y, 1) \text{ ack}_0(x) \text{ ack}_1(y) \text{ read}_0(y) \text{ read}_1(x) \text{ return}_0(y, 0) \text{ return}_1(x, 0)$$

Not sequentially consistent.
Algorithm

**Assumption:** Underlying message passing communication system supports totally ordered broadcast.

\[ \text{bc-send}_i (m, \text{total}) \rightarrow \text{tbc-send}_i (m) \]

\[ \text{bc-recv}_i (m, \text{total}) \rightarrow \text{tbc-recv}_i (m) \]

There is a local copy of every shared object in the state of the MCS process at every node.
Algorithm: Linearizability

when \text{read}_i(x) \text{ occurs}:

\hspace{1em} \text{enable tbc-send}_i(x).

when \text{write}_i(x, v) \text{ occurs}:

\hspace{1em} \text{enable tbc-send}_i(x,v).

when \text{tbc-recv}_i(x, v) \text{ from } p_j \text{ occurs}:

\hspace{1em} \text{copy}[x] := v

\hspace{2em} \text{if } j = i \text{ then enable } \text{ack}_i(x)

when \text{tbc-recv}_i(x) \text{ from } p_j \text{ occurs}:

\hspace{2em} \text{if } j = i \text{ then enable } \text{return}_i(\text{copy}[x])
Algorithm: Sequentially Consistently Local Read

code for processor $p_i$, $0 \leq i \leq n - 1$.

Initially copy[$x$] holds the initial value of shared object $x$, for all $x$.

when $\text{read}_i(x)$ occurs:
   enable $\text{return}_i(x, \text{copy}[x])$

when $\text{write}_i(x, v)$ occurs:
   enable $\text{tbc-send}_i(x,v)$

when $\text{tbc-recv}_i(x, v)$ from $p_j$ occurs:
   $\text{copy}[x] := v$
   if $j = i$ then enable $\text{ack}_i(x)$
Algorithm: Sequentially Consistent Local Write

code for processor \( p_i \), \( 0 \leq i \leq n - 1 \).

Initially \( \text{copy}[x] \) holds the initial value of shared object \( x \), for all \( x \), and \( \text{num} = 0 \).

when \( \text{read}_i(x) \) occurs:
   if \( \text{num} = 0 \) then enable \( \text{return}_i(x, \text{copy}[x]) \).

when \( \text{write}_i(x, v) \) occurs:
   \( \text{num} := \text{num} + 1 \)
   enable \( \text{tbc-send}_i(x, v) \)
   enable \( \text{ack}_i(x) \)

when \( \text{tbc-recv}_i(x, v) \) from \( p_j \) occurs:
   \( \text{copy}[x] := v \)
   if \( j = i \) then
      \( \text{num} = \text{num} - 1 \)
      if \( \text{num} = 0 \) and a read on \( x \) is pending then
         enable \( \text{return}_i(x, \text{copy}[x]) \).
Thank You