CHAPTER 4: INTERPROCESS COMMUNICATION AND COORDINATION

Chapter outline

- Discuss three levels of communication: basic message passing, request/reply and transaction communication based on message passing
- Discuss name services for communication
- Show examples of process coordination using message passing

Basic message passing communication

Communication primitives:

send(destination, message)
receive(source, message)

channel naming = process name, link, mailbox, port

- direct communication: symmetric/asymmetric process naming, link
- indirect communication: many-to-many mailbox, many-to-one port
Message buffering and synchronization

sender  source  network  destination  receiver
kernel   message  kernel

1. **Nonblocking send**, 1+8: Sender process is released after message has been composed and copied into sender’s kernel.

2. **Blocking send**, 1+2+7+8: Sender process is released after message has been transmitted to the network.

3. **Reliable blocking send**, 1+2+3+6+7+8: Sender process is released after message has been received by the receiver’s kernel.

4. **Explicit blocking send**, 1+2+4+5+6+7+8: Sender process is released after message has been received by the receiver process.

5. **Request and reply**, 1-4, service, 5-8: Sender process is released after message has been processed by the receiver and response returned to the sender.
Message passing API

- **Pipe**: A FIFO byte-stream unidirectional link for related processes
- **Message queue**: A structured variable length message queue
- **Named Pipe**: A special FIFO file pipe using path name for unrelated processes under the same domain
- **Socket**: A logical communication endpoint for communication between autonomous domains

Connectionless socket communication
Connection-oriented socket communication
Secure Socket Layer protocol

- **Privacy**: use symmetric private-key cryptography
- **Integrity**: use message integrity check
- **Authenticity**: use asymmetric public-key cryptography

CLIENT

- **ClientHello**
  - randomC, CipherSuites
  - randomS, CipherSuite, session id
  - server public key

- **ClientKeyExchange**
  - encrypted pre-mastersecret

- **ChangeCipherSpec**

- **Finished**
  - hashed message and secret

- **Socket Message**
  - encrypted and signed

SERVER

- **ServerHello**

- **ServerKeyExchange**

- **ChangeCipherSpec**

- **Finished**

- **Socket Message**
Group communication and multicast

- Best effort
- All or none
- Orderly delivery
  - FIFO
  - Causal order
  - Total order
Causal order

- Accept message \( m \) if \( T_i = S_i + 1 \) and \( T_k \leq S_k \) for all \( k \neq i \).
- Delay message \( m \) if \( T_i > S_i + 1 \) or there exists a \( k \neq i \) such that \( T_k > S_k \).
- Reject the message if \( T_i \leq S_i \).

Total order

![Diagram showing causal order and total order with message delivery times and multicast message acknowledge and commit times.](image)
Request/reply communication

Remote Procedure Calls (RPCs)

- Parameter passing and data conversion
- Binding
- Compilation
- Exception and failure handling
- Security
RPC Binding

RPC compilation

server procedures

interface specification

client main program

RPC generator

server stub

header file

client stub

RPC run-time library

compilation

server program

client program

server machine address
(directory server (binder or trader))

register service

register program, version, and port

client handle

create

port #

client machine

port mapper

client stub

RPC generator

compilation

client program

server program

RPC generator
RPC exception and failure

- **Exception**: in-band or out-band signaling
- **Link failure**: retransmission, sequence number and idempotent requests, use of transaction id $xid$
- **Server crash**:
  - *at least once*: server raises an exception and client retries
  - *at most once*: server raises an exception and client gives up
  - *maybe*: server raises no exception and client retries
- **Client crash**:
  - orphan killed by client
  - orphan killed by server
  - orphan killed by expiration
Secure RPC

- $C_s$ and $S_s$ are 128-bit random numbers.
- $C_p = \alpha^{C_s \mod M}$ and $S_p = \alpha^{S_s \mod M}$, where $\alpha$ and $M$ are known constants.

\[
\begin{align*}
SK_{cs} &= S_p^{C_s} = (\alpha^{S_s})^{C_s} = \alpha^{S_s \cdot C_s} \\
SK_{sc} &= C_p^{S_s} = (\alpha^{C_s})^{S_s} = \alpha^{C_s \cdot S_s}
\end{align*}
\]
Transaction Communication

ACID properties

- Atomicty
- Consistency
- Isolation
- Durability

Two-phase commit protocol

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>CoORDINATOR</th>
<th>PARTICIPANT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- precommit the transaction</td>
<td>- received request message</td>
</tr>
<tr>
<td></td>
<td>- send request to all participants</td>
<td>- if ready</td>
</tr>
<tr>
<td></td>
<td>- collect all replies</td>
<td>- then precommit and send YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- else abort transaction and send NO</td>
</tr>
<tr>
<td>Phase 2</td>
<td>- if all votes are unanimous YES</td>
<td>- receive decision</td>
</tr>
<tr>
<td></td>
<td>then commit and send COMMIT</td>
<td>- if COMMIT then commit</td>
</tr>
<tr>
<td></td>
<td>else abort and send ABORT</td>
<td>- if ABORT then abort</td>
</tr>
<tr>
<td></td>
<td>- received response</td>
<td>- send response</td>
</tr>
</tbody>
</table>
Failure and recovery of the 2PC protocol

Coordinator failure recovery actions

Participant failure recovery actions
Name and Directory Services

Object attributes and name structures

<table>
<thead>
<tr>
<th>Service/object Attributes</th>
<th>Name Structures</th>
<th>Attribute Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; attributes &gt;</td>
<td>flat structure</td>
<td>physical</td>
</tr>
<tr>
<td>&lt; name, attributes, address &gt;</td>
<td>hierarchical structure name-based resolution (e.g., white pages)</td>
<td>organizational</td>
</tr>
<tr>
<td>&lt; name, type, attributes, address &gt;</td>
<td>structure-free attribute-based resolution (e.g., yellow pages)</td>
<td>functional</td>
</tr>
</tbody>
</table>

Name space and information base

[Diagram showing a tree structure with nodes labeled DSA 2, DSA 1, DSA 3, Root, Country, Organization, and User]
Name resolution

Recursive chaining

Transitive chaining

Referral

Multicast
Distributed Mutual Exclusion

- *Contention-based:*
  - Timestamp prioritized
  - Voting

- *Control (Token)-based:*
  - Ring structure
  - Tree structure
  - Broadcast structure

**Tree-structure token passing**
**Broadcast structure token passing**

<table>
<thead>
<tr>
<th>Process 1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>20</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

| Process 2 | 14| 21| 10| 8 |

| Process 3 | 15| 21| 11| 9 |

| Process 4 | 15| 21| 10| 9 |

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>20</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Sequence vectors $S_i$

Token vector $T$

Token queue $Q$
Leader Election

Complete topology

The Bully algorithm

Logic ring topology

The initiator node sets participating = true and
send (id) to its successor node;

For each process node,

receive (value);

case

value > id : participating := true, send (value);

value < id and participating == false : participating := true, send (id);

value == id : announce leader;

end case

Tree topology