Today’s Topic

• Other RPC Models
• Remote Method Invocation (RMI)
• Message-Oriented Communication

Other RPC Models

• Asynchronous RPC
  – Request-reply behavior not needed when there is no result to return
  – Server can reply as soon as request is received
• Deferred synchronous RPC
  – Use two asynchronous RPCs
  – Client needs a reply but doesn’t want to wait for it; server sends reply via another asynchronous RPC
• One-way RPC
  – Client does not even wait for an ACK from the server
  – Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).
Asynchronous RPC

- A client and server interacting through two asynchronous RPCs

Deferred Synchronous RPC

- A client and server interacting through two asynchronous RPCs
Remote Method Invocation (RMI)

- RPCs applied to **objects, i.e., instances of a class**
  - Interface resides on one machine, implementation on another

Proxies and Skeletons

- **Proxy:** client stub
  - Sets up and tears down connection with the server
  - Marshall parameters
- **Skeleton:** server stub
  - Demarshall parameters and passes parameters to server and sends result to proxy
Binding a Client to an Object

• A client must first bind to an object before invoking its methods.
• Binding: contact the directory service to find out where the object resides, then load the interface (proxy) into client’s address space

Parameter Passing

• RMI supports system-wide object references
  - Parameters can be object references
  - [Java] pass local objects by value, pass remote objects by reference
Message-Oriented Transient Communication

- Many distributed systems are built on top of simple message-oriented model offered by the transport layer
  - Example: sockets API

Client-Server Communication using TCP Sockets
Socket Primitives for TCP

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Receive</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Message-Passing Interface (MPI)

- Sockets designed for network communication (e.g., TCP/IP)
  - Support simple send/receive primitives
- Sockets not suitable for protocols in clusters of workstations
  - Need an interface with more advanced primitives
- Large number of incompatible proprietary communication libraries and protocols
  - Need for a standard interface
- Message-passing interface (MPI) – a standard for message passing
  - Hardware independent
  - Low overhead
  - Designed for parallel applications (uses transient communication)
- Key idea: communication between groups of processes
  - Each endpoint is a (groupId, processId) pair
# MPI Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_bsend</td>
<td>Append outgoing message to a local send buffer</td>
</tr>
<tr>
<td>MPI_send</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_sendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recev</td>
<td>Receive a message; block if there is none</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>

# Message-Oriented Persistent Communication

- Message queuing systems
  - Support asynchronous persistent communication
  - Intermediate storage for message while sender/receiver are inactive
  - Example application: email
- Communicate by inserting messages in queues
- Sender is only guaranteed that message will be eventually inserted in recipient’s queue
  - No guarantees on when or if the message will be read
  - “Loosely coupled communication”
Message-Queuing Model

[Diagram showing various states of sender and receiver: running vs. passive, with arrows indicating transitions.]

Message-Queuing Model

[Diagram showing the interaction between sender and receiver, with labels for Look-up transport-level address of queue and Address look-up database.]

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue</td>
</tr>
</tbody>
</table>
General Architecture of a Message-Queuing System

Message Brokers