Last Class: RPCs

• RPCs make distributed computations look like local computations

• Issues:
  – Parameter passing
  – Binding
  – Failure handling

Today:

• Lightweight RPCs

• Remote Method Invocation (RMI)
  – Design issues
Lightweight RPCs

- Many RPCs occur between client and server on same machine
  - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server S exports interface to remote procedures
- Client C on same machine imports interface
- OS kernel creates data structures including an argument stack shared between S and C

RPC execution
- Push arguments onto stack
- Trap to kernel
- Kernel changes mem map of client to server address space
- Client thread executes procedure (OS upcall)
- Thread traps to kernel upon completion
- Kernel changes the address space back and returns control to client
- Called “doors” in Solaris
• Which RPC to use?  - run-time bit allows stub to choose between LRPC and RPC

Other RPC Models

• Asynchronous RPC
  – Request-reply behavior often not needed
  – Server can reply as soon as request is received and execute procedure later

• Deferred-synchronous RPC
  – Use two asynchronous RPCs
  – Client needs a reply but can’t 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) The interconnection between client and server in a traditional RPC
(b) The interaction using asynchronous RPC

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
  - *Class*: object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another
- RMIs support system-wide object references
  - Parameters can be object references

Distributed Objects

- When a client binds to a distributed object, load the interface ("proxy") into client address space
  - Proxy analogous to stubs
- Server stub is referred to as a skeleton
Proxies and Skeletons

- **Proxy**: client stub
  - Maintains server ID, endpoint, object ID
  - Sets up and tears down connection with the server
  - [Java:] does serialization of local object parameters
  - In practice, can be downloaded/constructed on the fly (why can’t this be done for RPCs in general?)

- **Skeleton**: server stub
  - Does deserialization and passes parameters to server and sends result to proxy

---

**Binding a Client to an Object**

(a) Example with implicit binding using only global references

Distr_object* obj_ref;  // Declare a systemwide object reference
obj_ref = ...;  // Initialize the reference to a distributed object
obj_ref-> do_something();  // Implicitly bind and invoke a method

(b) Example with explicit binding using global and local references

Distr_object obj_ref;  // Declare a systemwide object reference
Local_object* obj_ptr;  // Declare a pointer to local objects
obj_ref = ...;  // Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);  // Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();  // Invoke a method on the local proxy

---

(a) Example with implicit binding using only global references

(b) Example with explicit binding using global and local references
Parameter Passing

- Less restrictive than RPCs.
  - Supports system-wide object references
  - [Java] pass local objects by value, pass remote objects by reference

DCE Distributed-Object Model

a) Distributed dynamic objects in DCE.
b) Distributed named objects
Java RMI

- **Server**
  - Defines interface and implements interface methods
  - Server program
    - Creates server object and registers object with “remote object” registry
- **Client**
  - Looks up server in remote object registry
  - Uses normal method call syntax for remote methods
- **Java tools**
  - Rmiregistry: server-side name server
  - Rmic: uses server interface to create client and server stubs

Java RMI and Synchronization

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
  - How does this work for remote objects?
- **Options:** block at the client or the server
- **Block at server**
  - Can synchronize across multiple proxies
  - Problem: what if the client crashes while blocked?
- **Block at proxy**
  - Need to synchronize clients at different machines
  - Explicit distributed locking necessary
- **Java uses proxies for blocking**
  - No protection for simultaneous access from different clients
  - Applications need to implement distributed locking
Message-oriented Transient Communication

- Many distributed systems built on top of simple message-oriented model
  - Example: Berkeley sockets

Berkeley Socket Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</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
- Abstraction not suitable for other protocols in clusters of workstations or massively parallel systems
  - Need an interface with more advanced primitives
- Large number of incompatible proprietary libraries and protocols
  - Need for a standard interface
- Message-passing interface (MPI)
  - Hardware independent
  - Designed for parallel applications (uses transient communication)
- Key idea: communication between groups of processes
  - Each endpoint is a \((\text{groupId}, \text{processId})\) 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_issend</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there are none</td>
</tr>
<tr>
<td>MPI_irecv</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>