Distributed Systems

Thoai Nam
Faculty of Computer Science and Engineering
HCMC University of Technology
Chapter 2: Communication

- Issues in communication
- Message-oriented Communication
- Remote Procedure Calls
  - Transparency but poor for passing references
- Remote Method Invocation
  - RMIs are essentially RPCs but specific to remote objects
  - System wide references passed as parameters
- Stream-oriented Communication
Communication Protocols

- Protocols are agreements/rules on communication
- Protocols could be connection-oriented or connectionless

Diagram:

```
  Application (1)  -->  Application protocol (7)
     |                        |                        |
     v                        v                        v
Presentation (6)  -->  Presentation protocol (5)
     |                        |                        |
     v                        v                        v
Session (5)      -->  Session protocol (4)
     |                        |                        |
     v                        v                        v
Transport (4)    -->  Transport protocol (3)
     |                        |                        |
     v                        v                        v
Network (3)      -->  Network protocol (2)
     |                        |                        |
     v                        v                        v
Data link (2)    -->  Data link protocol (1)
     |                        |                        |
     v                        v                        v
Physical (1)     -->  Physical protocol
```

Network
Layered Protocols

- A typical message as it appears on the network.

![Diagram of layered protocols]

- Data link layer header
- Network layer header
- Transport layer header
- Session layer header
- Presentation layer header
- Application layer header

Bits that actually appear on the network
Client-Server TCP

(a) Normal operation of TCP.

1. Client: SYN
2. Server: SYN, ACK(SYN)
3. Client: ACK(SYN)
4. Server: request
5. Server: FIN
6. Client: ACK(req+FIN)
7. Server: answer
8. Server: FIN
9. Client: ACK(FIN)

(b) Transactional TCP.

1. Client: SYN, request, FIN
2. Server: SYN, ACK(FIN), answer, FIN
3. Server: ACK(FIN)
4. Time
5. Client: ACK(FIN)
Middleware Protocols

- Middleware: layer that resides between an OS and an application
  - May implement general-purpose protocols that warrant their own layers. Ex: distributed commit
Client-Server Communication Model

- Structure: group of servers offering service to clients
- Based on a request/response paradigm
- Techniques:
  - Socket, remote procedure calls (RPC), Remote Method Invocation (RMI)
Issues in Client-Server Communication

- Addressing
- Blocking versus non-blocking
- Buffered versus unbuffered
- Reliable versus unreliable
- Server architecture: concurrent versus sequential
- Scalability
Addressing Issues

- **Question:** how is the server located?

- **Hard-wired address**
  - Machine address and process address are known a priori

- **Broadcast-based**
  - Server chooses address from a sparse address space
  - Client broadcasts request
  - Can cache response for future

- **Locate address via name server**
Blocking versus Non-blocking

- **Blocking communication (synchronous)**
  - Send blocks until message is actually sent
  - Receive blocks until message is actually received

- **Non-blocking communication (asynchronous)**
  - Send returns immediately
  - Return does not block either

- **Examples**
Buffering Issues

- **Unbuffered communication**
  - Server must call receive before client can call send

- **Buffered communication**
  - Client send to a mailbox
  - Server receives from a mailbox
Reliability

- **Unreliable channel**
  - Need acknowledgements (ACKs)
  - Applications handle ACKs
  - ACKs for both request and reply

- **Reliable channel**
  - Reply acts as ACK for request
  - Explicit ACK for response

- **Reliable communication on unreliable channels**
  - Transport protocol handles lost messages
Remote Procedure Calls

- Goal: Make distributed computing look like centralized computing
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language

- Issues
  - How to pass parameters
  - Bindings
  - Semantics in face of errors

- Two classes: integrated into prog, language and separate
Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read

- Main program's local variables

b) The stack while the called procedure is active

- Main program's local variables
- bytes
- buf
- fd
- return address
- read's local variables

Stack pointer

(a) (b)
Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures

- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling

- Related issue: global variables are not allowed in RPCs
Client and Server Stubs

- Principle of RPC between a client and server program.

![Diagram showing the principle of RPC between a client and server. The diagram illustrates the flow of a request from the client to the server, followed by a response back to the client, with the server making a local call and returning results. The process is indicated with arrows for client and server actions, and dashed lines for the wait for result and return from call phases.]
Stubs

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called *marshalling*
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  - Simplifies programmer task
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client
Example of an RPC

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream
Problem: how does a client locate a server?
- Use Bindings

Server
- Export server interface during initialization
- Send name, version no, unique identifier, handle (address) to binder

Client
- First RPC: send message to binder to import server interface
- Binder: check to see if server has exported interface
  » Return handle and unique identifier to client
Case Study: SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN’s eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures
Binder: Port Mapper

- Server start-up: create port
- Server stub calls `svc_register` to register prog. #, version # with local port mapper
- Port mapper stores prog #, version #, and port
- Client start-up: call `clnt_create` to locate server port
- Upon return, client can call procedures at the server
 Rpcgen: generating stubs

- Q_xdr.c: do XDR conversion
- Detailed example: later in this course
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
Lightweight RPCs

- 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
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
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
Message-oriented Transient Communication

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

Server

- socket -> bind -> listen -> accept -> read -> write -> close
- Synchronization point

Client

- socket -> connect -> write -> read -> close

Communication
## 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>