Communication Models

Advanced Operating Systems

Reza Azimi

Shiraz University, Spring 2010

Reasons for Communication

- **Sharing Information**
  - Processes of several users exchange information. Examples: sending emails, sharing files, etc.

- **Parallelization**
  - Multiple processes work on the same problem running on different processors
  - Need to communicate data and the results of computation

- **Modularity**
  - A program is broken into several processes for reasons of robustness and security

- **Multiprogramming**
  - Several processes of a single user are active simultaneously. Example: cut&pasting information from the browser to a text or photo editor
Communication Models

- **Shared Memory**
  - A region of memory is shared between processes.
  - Communication is done by direct `read/write` (CPU) instructions.
  - *Implicit* communication
    - A process can write to the shared memory without notifying the others.

- **Message Passing**
  - Messages are exchanged between communicating processes.
  - *Explicit* communication
    - All message `send` and `receive` operations are visible to the communicating processes.
Message Passing

- Basic Operations
  - Send
  - Receive

- Naming Models
- Synchronization Models
- Buffering Models

---

Naming Models: Direct Naming

- Every process has a unique identifier (PID, in UNIX systems)
- The process id is used for addressing.

Sender (pid = "Q")
/* send a message to process P */
send(P, message)

Recipient (pid = "P")
/* receive a msg from Q (symmetric communication) */
receive(Q, message);

... or
/* receive a msg from any process (asymmetric communication) */
receive(id, message);

What's the disadvantage of this model?
What happens if the PID of one process changes?
Naming Models: Indirect Naming

- A "mailbox" or "port" is used for sending/receiving messages
  - The mailbox or port has its own unique identifier.
  - The mailbox or port is owned either by the system or by a process.
  - The sender and receiver process use port id for communication rather than each other’s pid (more permanent info).

- A Process can
  - Create a mailbox or attach to a port.
  - Send and receive messages through the mailbox or the port
  - Delete a mailbox or detach from a port.

Synchronization Models

- Blocking (synchronous)
  - Send: The sending process is blocked until the message is received by the receiving process.
  - Receive: The receiver blocks until a message is available.

- Non-blocking (asynchronous)
  - Send: The sending process sends the message and resumes operation
  - Receive: The receiver does not wait for the message to arrive;
    - if the message has already arrived it will be delivered to the receiver, otherwise, a null message is retrieved by the receiver.
Blocking vs. Non-blocking

- **Blocking**
  - **Advantage: Simplicity**
    - Sender: Once the send operation returns, you know the data is received by the receiver process.
    - Recipient: You either have your data, or wait for it!
  - **Disadvantage: Inefficient**
    - Both sender and receiver may be blocked while they can do other things.

- **Non-blocking**
  - **Advantage: Efficiency**
    - While the message is on the fly, you manage to do other things.
  - **Disadvantage: Program Complexity**
    - You need to check once in a while whether your message is actually sent or arrived (polling), or write a callback function to notify you.
    - You need to schedule your own tasks more or less explicitly.

Message Passing: Buffering Models

- **Zero Capacity**
  - No queuing of the messages. Only one message on each channel. The sender cannot send any other message before the current one is received by the receiver process.

- **Bounded Capacity**
  - A maximum of N messages can be in flight. If the queue is full, the sender will be blocked until one message is received by the receiver.

- **Unbounded Capacity**
  - The queue length is virtually infinite
**IPC in Client-Server Systems**

- The sender and receiver processes reside in different computers.

- Communication Steps
  - The clients initiate communication by establishing a channel to the server.
  - Once the channel is established, the client sends request to the server.
  - The server receives the request and sends the response over the channel.

**Examples of IPC Mechanisms**

- Signals (software interrupts)
- Pipes
- Shared Memory
- Sockets
- Semaphores and Mutexes
- Remote Procedure Calls (RPC)
Example: Sockets

Step 1: Creating Sockets

```
int sd; /* socket descriptor */
sd = socket(AF_INET, SOCK_STREAM, 0);
```

- check for errors as in the sample code!
Step 2 for Servers: Binding

```c
/* socket address structure */
struct sockadd_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));

/* Internet as te protocol family */
servaddr.sin_family = AF_INET;

/* Binding to any IP address on the server */
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* binding to your specific port */
servaddr.sin_port = htons(port);

bind(sd, (struct sockaddr *) &servaddr, sizeof(servaddr));
```

Step 3 for Servers: Listening

```c
int sd; /* socket descriptor */
...
listen(sd, 5);
```

- **Socket Descriptor**
- **Backlog Size:** Maximum Number of Pending Connections

---

*Advanced Operating Systems, Simon University, Winter 88/Spring 89, Reza Azimi*
Step 4 for Servers: Accepting Conn.

```c
int sd; /* socket descriptor */
int connected_sd;
...
connected_sd = accept(sd, NULL, NULL);
```

Step 2 for Clients: Connect

```c
/* socket address structure */
struct sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));

/* Internet as te protocol family */
servaddr.sin_family = AF_INET;
/* the port server is listening to */
servaddr.sin_port = htons(port);
/* Set the server IP address */
if ( inet_aton(szAddress, &servaddr.sin_addr) <= 0 )
    printf("Invalid remote IP address.\n");
/* connect() to the remote echo server */
connect(conn_s, (struct sockaddr *) &servaddr,
        sizeof(servaddr));
```
Writing to Sockets

```c
int sd;   /* socket descriptor */
char buffer[BUFF_SIZE];
int num_bytes, num_written;
...
num_written = write(sd, buffer, num_bytes);
```

- **Asynchronous write:**
  - The write system call returns before the recipient has received the message.

---

Advanced Operating Systems, Shinaz University, Winter 88/Spring 89, Reza Azimi

Reading From Sockets

```c
int sd;   /* socket descriptor */
char buffer[BUFF_SIZE];
int num_bytes, num_read;
...
num_read = read(sd, buffer, num_bytes);
```

- **Synchronous read:**
  - The write system gets blocked until the a message arrives.

---

Advanced Operating Systems, Shinaz University, Winter 88/Spring 89, Reza Azimi
Socket Programming: Summary

Remote Procedure Call (RPC)

Basic Model:
- Server exports some “services” in the form of some operations/procedures.
- Client uses these services by remotely invoking (calling) the procedures.
- Providing an illusion of making a local call.

Basic Implementation:
- Make a set of message exchanges between client and server look like a traditional procedure call.
Logical view of RPC

RPC in Today’s Systems

- Sun RPC
  - Also known as ONC RPC
  - Used in building many unix network services (e.g. NFS, NIS).
  - Was developed in the 80s.

- CORBA (Obsolete)
  - Common Object Request Broker Architecture

- Java Remote Method Invocation (RMI)

- Microsoft’s DCOM

- IBM’s Web Services
RPC Architecture

RPC Implementation Issues

- Binding
  - How do we locate servers with the services/API we want?
  - How do we specify the service?

- Heterogeneity
  - How to deal with heterogeneity across a distributed system (OS, hardware architecture, language)?

- Error Handling
  - How to handle data losses due to transient network failures?

- Performance
Location Transparency: Good or Bad?

- Basic Idea:
  - *Object names should not reveal their physical location.*
    - Location-dependent names: //server1/tests/doc1.txt
    - Location-independent name: /scr/programs/prog1.exe

- Benefit: uniform name space (for local or remote objects) simplifies distributed system design.

- Flaw: hiding fundamental differences between accessing local and remote objects, in terms of:
  - Access latency
  - Fault rate
  - Availability
  - would make a distributed system very inefficient

Service Binding

- The server **exports** its interface
  - identifies itself to a network name server
  - tells RPC runtime its alive and ready to accept calls

- The client, before issuing any calls, **imports** the service
  - RPC runtime uses the name server to find the location of a server and establish a connection

- The import and export operations are explicit in the server and client programs
  - breakdown of transparency
Defining Server Interfaces

- **Interface Definition Language (IDL)** specifies:
  - Service (procedure) Name
  - Parameters’ type
  - Return value type

- IDL “compiler”
  - Automatically generates code for two **stubs** for each service: a **client stub** and a **server stub**.
  - Masks the details of remote procedure calls
    - Avoid having each program write this code for itself
    - Makes it easy to change the client-server interface later

Server Stubs

- Receives remote requests
- Unpacks the parameters
- Calls the corresponding server-side procedures
- From the client’s side, it looks like the actual service procedure, but it’s not.
Client Stubs

- Packs the parameters into a network message
- Sends the message
- Waits for the reply, and returns the result to the client app

RPC Example

**Server Interface:**

```c
int Temp(str postalCode);
```

**Client Program:**

```c
... temp = server -> Temp(M1C1A4) ...
```

**Server Program:**

```c
int Temp(str postalCode) {
    return (...TempDB lookup...);
}
```

- If the server were implemented as a local regular library, then calling `Temp` would just be a normal procedure call
RPC Example: Call

Client Program:
```
temp = server ->
    Temp(M1C1A4)
```

Server Program:
```
int Temp(str postalCode)
{
    ....
}
```

Client Stub:
```
int Temp(str postalCode)
{
    Alloc message buffer;
    Mark call as "Temp";
    Store postalCode in buffer;
    Send message;
}
```

Server Stub:
```
Temp_Stub(message) {
    Remove postalCode from buffer;
    result = Temp(postalCode);
}
```

RPC Runtime:
```
Forward message to server;
```

RPC Runtime:
```
Receive message;
Dispatch, call Temp_Stub;
```

RPC Example: Return

Client Program:
```
temp = server ->
    Temp(M1C1A4)
```

Server Program:
```
int Temp(str postalCode)
{
    ....
}
```

Client Stub:
```
int Temp(str postalCode)
{
    Create, Send message...
    Remove the return value from the reply
    Return the return value
}
```

Server Stub:
```
Temp_Stub(message) {
    Remove postalCode from buffer;
    result = Temp(postalCode);
    Store result in a reply buffer
}
```

RPC Runtime:
```
Return Reply to Stub;
```

RPC Runtime:
```
Send Reply to Client
```

Advanced Operating Systems, Shiraz University, Winter 88/Spring 89, Reza Azimi
## Marshalling

- **Marshalling** refers to the packing of procedure parameters into a message packet.

- The RPC stubs call type-specific procedures to marshal (or unmarshal) the parameters to a call:
  - The client stub marshals the parameters into a message.
  - The server stub unmarshals parameters from the message and uses them to call the server procedure.

- On return:
  - The server stub marshals the return parameters.
  - The client stub unmarshals return parameters and returns them to the client program.

---

## RPC Semantics: At Least Once

- Remote procedure executed at least once, but maybe more than once.
- Retransmissions on message delay/loss but **no** duplicate filtering.
- This is acceptable for **idempotent** operations (semantics of operation are the same whether executed 1 time or N times); e.g., reading data that is read-only.
RPC Semantics: At Most Once

- Most appropriate for non-idempotent operations.
- Remote procedure executed 0 or 1 time, i.e. exactly once or not at all.
- Use of retransmissions for message delay/loss and duplicate filtering.