Chapter IV
INTER-PROCESS COMMUNICATION

Jehan-François Pâris
jfparis@uh.edu
Chapter overview

- **Types of IPC**
  - Message passing
  - Shared memory

- **Message passing**
  - Blocking/non-blocking, ...
  - Datagrams, virtual circuits, streams
  - Remote procedure calls
Message passing (I)

- Processes that want to exchange data send and receive *messages*

- Any *message exchange* requires
  - A *send*
    
    ```
    send(addr, msg, length);
    ```
  - A *receive*
    
    ```
    receive(addr, msg, length);
    ```
Message passing (II)
Advantages

- **Very general**
  - Sender and receivers can be on different machines

- **Relatively secure**
  - Receiver can inspect the messages it has received before processing them
Disadvantages

- **Hard to use**
  - Every data transfer requires a `send()` and a `receive()`
  - Receiving process must *expect* the `send()`
    - Might require forking a special thread
Shared Memory

- Name says it
  - Two or more processes share a part of their address space
Advantages

- *Fast and easy to use*
  - The data are there
  but
  - Some concurrent accesses to the shared data can result into small disasters
  - Must synchronize access to shared data

- Topic will be covered in next chapter
Disadvantages

- *Not a general solution*
  - Sender and receivers must be on the *same machine*

- *Less secure*
  - Processes can directly access a part of the address space of other processes
MESSAGE PASSING

- Defining issues
  - Direct/Indirect communication
  - Blocking/Non-blocking primitives
  - Exception handling
  - Quality of service
    - Unreliable/reliable datagrams
    - Virtual circuits, streams
Direct communication (I)

- Send and receive system calls always specify **processes** as destination or source:
  - `send(process, msg, length);`
  - `receive(process, msg, &length);`

- Most basic solution because there is
  - No intermediary between sender and receiver
An analogy

- Phones without switchboard
  - Each phone is hardwired to another phone
Direct communication (II)

- Process executing the receive call must know the identity of all processes likely to send messages
  - Very bad solution for servers
    - Servers have to answer requests from arbitrary processes
Indirect communication (I)

- Send and receive primitives now specify an intermediary entity as destination or source: the *mailbox*
  
  \[
  \text{send(mailbox, msg, size);} \\
  \text{receive(mailbox, msg, &size);}
  \]

- Mailbox is a system object created by the kernel at the request of a user process
An analogy (I)

- Phones with a switchboard
  - Each phone can receive calls from any other phone
An analogy (II)

- Each phone has now a **phone number**
  - Callers dial that number, not a person’s name

- Taking our phone with us allows us to receive phone calls **from everybody**
Indirect communication (II)

- Different processes can send messages to the same mailbox
  - A process can receive messages from processes it does not know anything about
  - A process can wait for messages coming from different senders
    - Will answer the first message it receives
Mailboxes

- Mailboxes can be
  - *Private*
    - Attached to a specific process
      - *Think of your cell phone*
  - *Public*
    - System objects
      - *Think of a house phone*
Private mailboxes

- Process that requested its creation and its children are the only processes that can receive messages through the mailbox are that process and its children.

- Cease to exist when the process that requested its creation (and all its children) terminates.

- Often called **ports**

- **Example:** BSD **sockets**
Public mailboxes

- Owned by the *system*
- Shared by all the processes having the right to receive messages through it
- Survive the termination of the process that requested their creation
- Work best when all processes are on the *same machine*

*Example:* System V UNIX *message queues*
Blocking primitives (I)

- A *blocking send* does not return until the receiving process has received the message
  - No *buffering* is needed
  - *Analogous to what is happening when you call somebody who does not have voice mail*
Blocking primitives (II)

- **A blocking receive** does not return until a message has been received.
- *Like waiting by the phone for an important message or staying all day by your mailbox waiting for the mail carrier.*
Blocking primitives (III)
Non-blocking primitives (I)

- A **non-blocking send** returns as soon as the message has been accepted for delivery by the OS
  - Assumes that the OS can store the message in a *buffer*
  - *Like mailing a letter: once the letter is dropped in the mailbox, we are done*
    - The mailbox will hold your letter until a postal employee picks it up
Non-blocking primitives (II)

- A *non-blocking receive* returns as soon as it has either retrieved a message or learned that the mailbox is empty
  - *Like checking whether your mail has arrived or not*
Non-blocking primitives (III)

Sender

Buffer

msg

Receiver

receive(...) acts as a retrieve(...)

send(...)
Simulating blocking receives

- Can simulate a blocking receive with a non-blocking receive within a loop:

  ```
  do {
      code = receive(mbox, msg, size);
      sleep(1); // delay
  } while (code == EMPTY_MBOX);
  ```

- Known as a **busy wait**
  
  - Costlier than a **blocking wait**
Simulating blocking sends

- Can simulate a blocking send with two non-blocking sends and a blocking receive:
  - Sender sends message and requests an acknowledgement (ACK)
  - Sender waits for ACK from receiver using a blocking receive
  - Receiver sends ACK
The standard choice

- In general we prefer
  - *Indirect naming*
  - *Non-blocking sends*
    - Sender does not care about what happens once the message is sent
    - *Similar to UNIX delayed writes*
  - *Blocking receives*
    - Receiver needs the data to continue
Buffering

- **Non-blocking primitives** require **buffering** to let OS store somewhere messages that have been sent but not yet received.

- These buffers can have
  - **Bounded capacity**
    - Refuse to receive messages when the buffer is full
  - Theoretically **unlimited capacity**.
An explosive combination (I)

- **Blocking receive** does not go well with **direct communication**
  - Processes cannot wait for messages from several sources without using special parallel programming constructs:
    - **Dijkstra's alternative command**
An explosive combination (II)

- Using blocking receives with direct naming does not allow the receiving process to receive any messages from any process but the one it has specified

receive(Q, msg)
Exception condition handling

- Must specify what to do if one of the two processes dies
  - Especially important whenever the two processes are on two different machines
- Must handle
  - Host failures
  - Network partitions
Quality of service

- When sender and receiver are on different machines, messages
  - Can be lost, corrupted or duplicated
  - Arrive out of sequence

- Can still decide to provide reliable message delivery
Positive acknowledgments

- Basic technique for providing reliable delivery of messages
- Destination process sends an acknowledgment message (ACK) for every message that was correctly delivered
  - Damaged messages are ignored
- Sender resends any message that has not been acknowledged within a fixed time frame
First scenario

Sender

Sends message

Sends ACK

Receiver
Second scenario

Sender

Sends message

Message is lost: no ACK is sent

Resends message

Receiver
Third scenario (I)

- Sender sends message.
- Receiver sends ACK.
- ACK is lost.
- Sender resends message.
Third scenario (II)

- Receiver **must** acknowledge a second time the message
  - Otherwise it would be resent one more time

- Rule is
  - *Acknowledge any message that does not need to be resent!*
Classes of service

- **Datagrams:**
  - Messages are send one at time

- **Virtual circuits:**
  - Ordered sequence of messages
  - *Connection-oriented* service

- **Streams:**
  - Ordered sequence of bytes
  - Message boundaries are ignored
Datagrams

- Each message is sent *individually*
  - Some messages can be *lost*, other *duplicated* or arrive *out of sequence*
  - *Equivalent of a conventional letter*

- **Reliable datagrams:**
  - resent until they are acknowledged

- **Unreliable datagrams**
Unreliable datagrams (I)

- Messages are not acknowledged
- Works well when message requests a reply
  - Reply is *implicit ACK* of message

Client

Sends request

Server

Sends reply (and ACKs the request)
Unreliable datagrams (II)

- Exactly what we do in real life:
  - *We rarely ACK emails and other messages*
  - *We reply to them!*

- Sole reason to ACK a request is when it might take a long time to reply to it
UDP

- **User Datagram Protocol**
- Best known datagram protocol
- Provides an unreliable datagram service
  - Messages can be *lost*, *duplicated* or arrive *out of sequence*
- Best for short interactions
  - One request and one reply
Virtual circuits (I)

- Establish a *logical connection* between the sender and the receiver
- Messages are *guaranteed* to arrive in sequence without lost messages or duplicated messages
  - *Same as the words of a phone conversation*
Virtual circuits (II)

- Require setting up a virtual connection **before** sending any data
  - Costlier than datagrams
- Best for transmitting large amounts of data that require sending several messages
  - *File transfer protocol* (FTP)
  - *Hypertext transfer protocol* (HTTP)
Streams

- Like virtual circuits
- Do **not** preserve message boundaries:
  - Receiver sees a *seamless stream of bytes*
- Offspring of UNIX philosophy
  - Record boundaries do not count
  - Message boundaries should not count
TCP

- Transmission Control Protocol
- Best known stream protocol
- Provides a reliable stream service
- Said to be *heavyweight*
  - Requires three messages (*packets*) to establish a virtual connection
Datagrams and Streams

- **Datagrams:**
  - Unreliable
  - Not ordered
  - Lightweight
  - Deliver messages
- **Example:**
  - UDP

- **Streams:**
  - Reliable
  - Ordered
  - Heavyweight
  - Stream-oriented
- **Example:**
  - TCP
Remote Procedure Calls
Motivation (I)

- Apply to **client-server** model of computation
- A typical client-server interaction:

```c
send_req(args); rcv_req(&args);
process(args, &results);
send_reply(results);
rcv_reply(&results);
```
Motivation (II)

- Very similar to a procedure call to a local procedure:

  - \texttt{xyz(args, \&results);}

  - \texttt{xyz(...) \{}\
    - \ldots
    - return;
  \}\ // \texttt{xyz}

- Try to use the same formalism
The big idea

- We could write

```c
rpc(xyz, args, &results);
xyz(...) {
    ....
    return;

    ....
    return;

} // xyz
```

and let system take care of all message passing details
Advantages

- Hides all details of message passing
  - Programmer can focus on the logic of her application
- Provides a higher level of abstraction
- Extends a well-known model of programming
  - Anybody that can use procedures and function can quickly learn to use remote procedure calls
Disadvantage

- The illusion is *not perfect*
  - RPCs do not always behave exactly like regular procedure calls
    - Client and server do not share the same address space
  - Programmer must remain aware of these subtle and not so subtle differences
General Organization

User Program

User Stub

calls

(Server Stub)

calls

(Server Procedure)

(system generated)
What the programmer sees

User Program

Does a RPC

Server Procedure

All IPC between client and server are *hidden*
The user program

- Contains the user code
- Calls the user stub
  
  
  ```
  rpc(xyz, args, &results);
  ```

- *Appears* to call the server procedure
The user stub

- Procedure generated by RPC package:
  - Packs arguments into request message and performs required data conversions (argument marshaling)
  - Sends request message
  - Waits for server's reply message
  - Unpacks results and performs required data conversions (argument unmarshaling)
The server stub

- Generic server generated by RPC package:
  - Waits for client requests
  - Unpacks request arguments and performs required data conversions
  - Calls appropriate server procedure
  - Packs results into reply message and performs required data conversions
  - Sends reply message
The server procedure

- Procedure called by the server stub
- Written by the user
- Does the actual processing of user requests
Differences with regular PC

- Client and server processes *do not share* the *same address space*
- Client and server can be on *different machines*
- Must handle *partial failures*
No shared address space

This means

- *No global variables*
- *Cannot pass addresses*
  - Cannot pass arguments by reference
  - Cannot pass dynamic data structures through pointers
The solution

- RPC *can pass arguments by value and result*
  - Pass the *current value* of the argument to the remote procedure
  - *Copy* the *returned value* in the user program

- Not the same as passing arguments by reference
Passing by reference

**Caller:**

```c
... i = 0; abc(&i); ...
```

Procedure `abc()` will directly increment variable `i`

```c
abc(int *k){
    (*k)++; 
}
```
Passing by value and result

**Caller:**

```c
... i = 0;
abc(&i);
...
```

Variable `i` is updated after caller receives server’s reply

```c
abc(int *k){
(*k)++;
}
```
An example (I)

- Procedure **doubleincrement**

  ```c
  doubleincrement(int *p, int *q) {
    (*p)++ ; (*q)++ ;
  } // doubleincrement
  ```

- Calling

  ```c
  doubleincrement(&m, &m);
  ```

  should increment **m twice**
An example (II)

- Calling
  
  \[\text{doubleincrement}(&m, &m)\];

  passing arguments by \textit{value and return} only increments \(m\) \textit{once}

- Let us consider the code fragment

  \[
  \begin{align*}
  \text{int } m &= 1; \\
  \text{doubleincrement}(&m, &m);
  \end{align*}
  \]
Passing by reference

**Caller:**

```c
... int m = 1; double increment(&m,&m); ...
; m
```

Pass TWICE the ADDRESS of m

Variable m gets incremented TWICE
Passing by value and result

**Caller:**

```c
... 
int m = 1; 
double increment(&m, &m); 
... 
;
```

Pass twice the VALUE of m: 1 and 1

Return two NEW VALUES: 2 and 2
Passing dynamic types (I)

- Cannot pass dynamic data structures through pointers
  - Must send a copy of data structure

- For a linked list
  - Send array with elements of linked list plus unpacking instructions
Passing dynamic types (II)

- We want to pass

- We send to the remote procedure

- Header identifies linked list with four elements
The NYC Cloisters

Rebuilt in NYC from actual cloister stones
Architecture considerations

- The machine representations of floating point numbers and byte ordering conventions can be different:
  - **Little endians** start with *least* significant byte:
    - *Intel's 80x86*, *AMD64 / x86-64*
  - **Big-endians** start with *most* significant byte:
    - *IBM z* and *OpenRISC*
If you really want to know

- Big endians

```
00 01 10 11
```

- Little endians

```
11 10 01 00
```
The solution

- Define a network order and convert all numerical variables to that order
  - Use `hton` family of functions
  - Same as requiring all air traffic control communications to be in English
  - If you want to know, the network order is big-endian
Detecting partial failures

- The client must detect server failures
  - Can send *are you alive?* messages to the server at fixed time intervals
  - *That is not hard!*
Handling partial executions

- Client must deal with the possibility that the server could have crashed *after* having partially executed the request
  - ATM machine calling the bank computer
    - Was the account debited or not?
First solution (I)

- **Ignore** the problem and *always resubmit* requests that have not been answered
  - Some requests may be executed more than once

- Will work *if* all requests are *idempotent*
  - Executing them several times has the same effect as executing them exactly once
First solution (II)

- Examples of idempotent requests include:
  - Reading \( n \) bytes from a fixed location
  - \textit{NOT} reading next \( n \) bytes
  - Writing \( n \) bytes starting at a fixed location
  - \textit{NOT} writing \( n \) bytes starting at current location

- Technique is used by all RPCs in the Sun Microsystems’ \textit{Network File System} (NFS)
Second solution

- Attach to each request a *serial number*
  - Server can detect replays of requests it has previously received and refuse to execute them
  - *At most once* semantics

- Cheap but not perfect
  - Some requests could end being partially executed
Third solution

- Use a *transaction mechanism*
  - Guarantees that each request will *either be fully executed or have no effect*
  - *All or nothing* semantics

- **Best** and **costliest** solution

- Use it in all *financial transactions*
An example

- Buying a house using *mortgage money*
  - Cannot get the mortgage without having a title to the house
  - Cannot get title without paying first previous owners
  - Must have the mortgage money to pay them

- Sale is a complex atomic transaction
Another example
Realizations (I)

- **Sun RPC:**
  - Developed by Sun Microsystems
  - Used to implement their Network File System

- **MSRPC (Microsoft RPC):**
  - Proprietary version of the DCE/RPC protocol
  - Was used in the Distributed Component Object Model (DCOM).
Realizations (II)

- **SOAP:**
  - Exchanges XML-based messages
  - Runs on the top of HTTP
    - Very portable
    - Very verbose

- **JSON-RPC:**
  - Uses *JavaScript Object Notation* (JSON)