Week 3: Inter-Process Communications & Remote Procedure Call

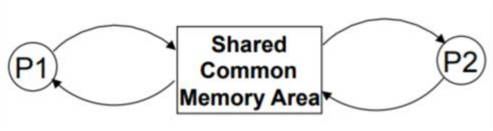
Learning Outcomes:

- Introduce the concept of inter-process communications (IPC)
- Introduce the concept of remote procedure call (RPC)
- Discuss Assignment 1

Inter-process Communication (IPC)

Means information sharing among two or more processes. Consists of two methods:

1. Original sharing (Shared data approach)



2. Copy sharing (Message passing approach)

We focus on the message passing approach (we’re doing MPI), using a message-passing system to allow processes to communicate. It also serves as a suitable infrastructure for building other IPC systems such as RPC (Remote Procedure Call) or DSM (Distributed Shared Memory).

Features of an Effective Message Passing System

- ✓ Simplicity
- ✓ Uniform semantics – same primitives for local & remote communication
- ✓ Efficiency via:
  1. Reducing the number of messages as much as possible
  2. Avoiding repeating cost of establishing & terminating connections between same pair of processes for each and every message exchange between them
  3. Minimizing cost of maintaining connections
  4. Piggybacking acknowledgement of previous message with the next message
- ✓ Reliability – Loss & duplicate message handling
- ✓ Correctness: Atomicity, Ordered delivery, survivability

- ✓ Security – Authentication of sender & receiver + Encryption of messages
- ✓ Portability – system should be lightweight

IPC Message Structure

Header:

1. Addresses
    - ✓ Sender address
    - ✓ Receiver address
2. Sequence Number
3. Structural Information
    - ✓ Type
    - ✓ Number of bytes
4. Message

Issues in an IPC Protocol

1. Identity – Who is sender?, who is receiver?
2. Network Topology – 1 receiver or many?
3. Flow Control – Guaranteed by the receiver? should sender wait for reply?
4. Error Control & Channel management – Node crash? Receiver not ready? Several outstanding messages to the receiver?

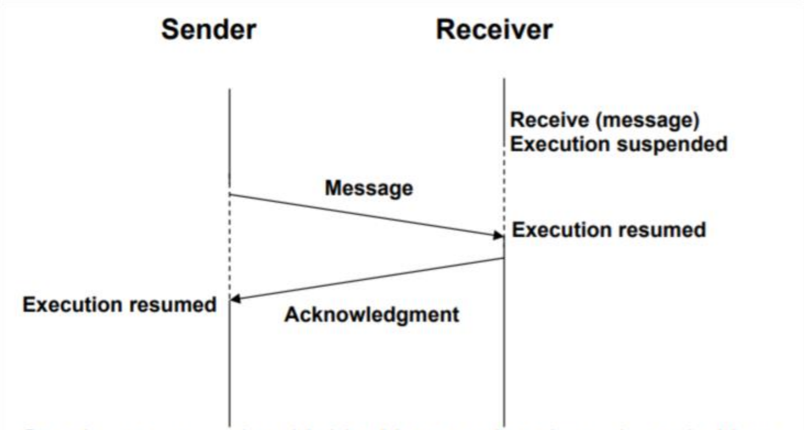
Synchronization in IPC

Send primitive consists of Blocking & Non-blocking

Receive primitive consists of Blocking & Non-blocking. Non-blocking can utilize Polling (continuously ping for response) or Interrupts (check for a message during program interruption).

Synchronous & Asynchronous Communication

When both the send & receive primitives of a communication between two processes use blocking semantics, the communication is synchronous, otherwise it is asynchronous.



Synchronous mode with blocking send and receive primitives

Synchronous VS Asynchronous Communication

Synchronous offers:

1. Simplicity & ease of implementation (ordering & execution is more stable)
2. Reliability
3. No backward error recovery needed

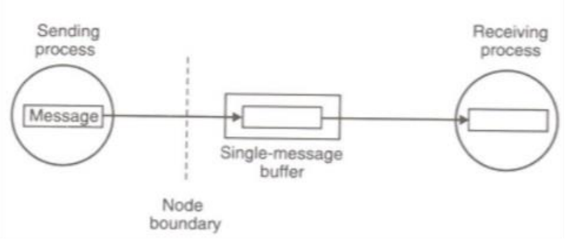
Asynchronous offers:

1. Higher concurrency (parallel computing!)
2. More flexible than synchronous
3. harder to manage
4. Lower deadlock risk than synchronous

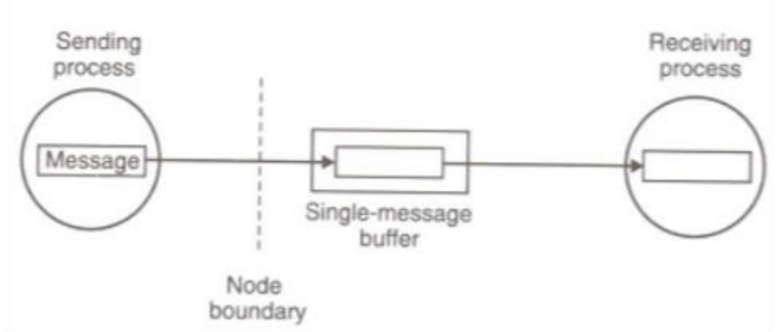
Buffering

Buffer is where the data we send lives. There is a send buffer & a receive buffer.

In Synchronous communication, we may not need a buffer (null), or we may need a single-message buffer (to issues that may occur with consistency)



In Asynchronous communication, the buffer capacity is unbounded, consisting of a multiple message buffer (of finite length), which hopefully, stores messages that will eventually be received (and cycled out).



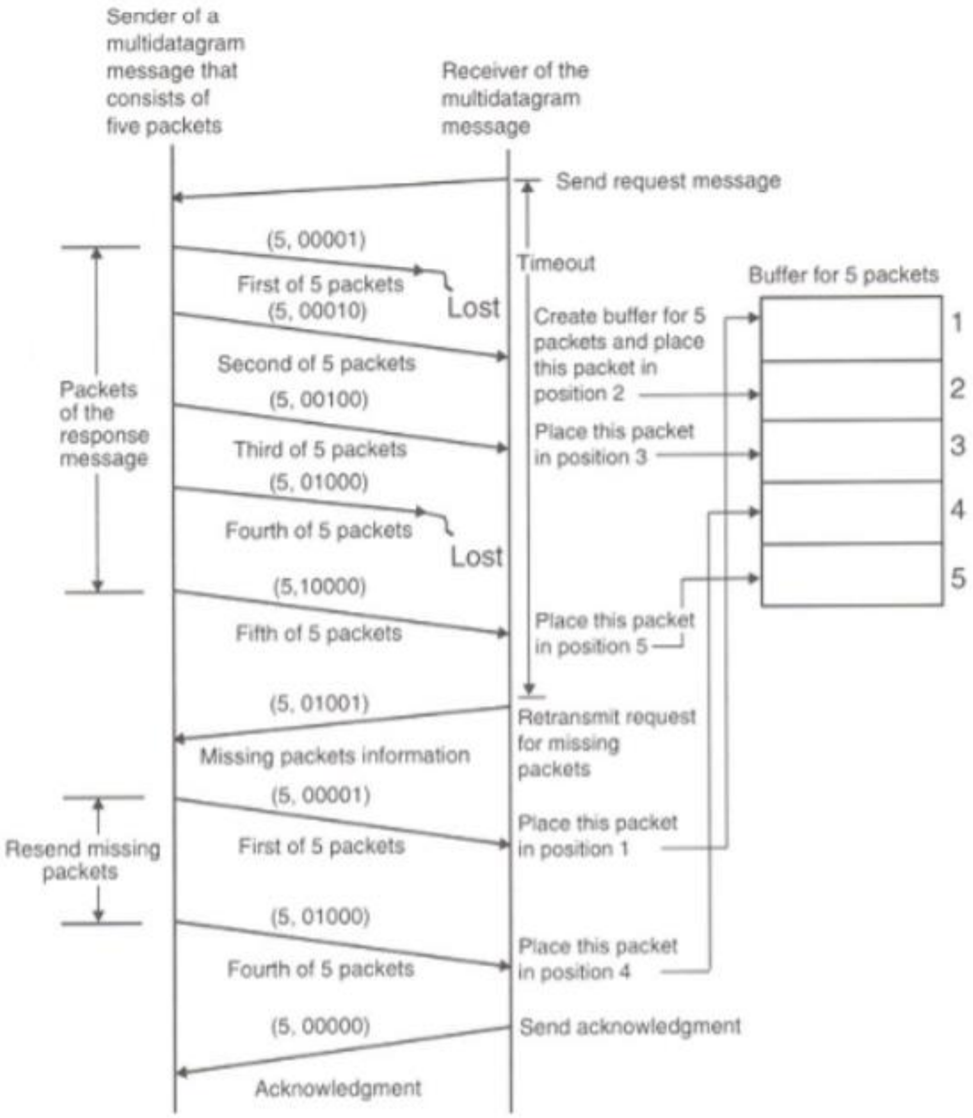
Multi-datagram Messages (On a Network)

Data sent within networks typically have an upper bound size for a packet. This is MTU (maximum transfer unit). Thus, a message size which is greater than MTU has to be fragmented, and set in a packet, known as a datagram.

Thus, messages may be single-datagram messages or multi-datagram messages. Assembling & disassembling messages is the responsibility of the MPS.

Example Message containing 5 packets

Can see that each packet has a sequence number, and packet 1 & 4 has been lost during transmission. Receiver retransmits request for missing packets and sends acknowledgement after receiving all.

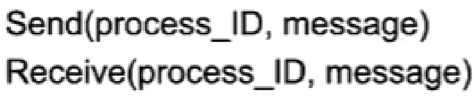


Encoding/Decoding

- ✓ Needed if sender & receiver have different architecture.
- ✓ Modern systems have a globally shared address space

Process Addressing

Explicit Addressing – explicitly state which process we want to send messages to and receive messages from.



Send(process_ID, message)
Receive(process_ID, message)

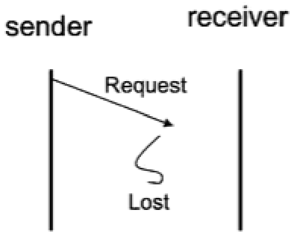
Implicit Addressing – Send messages to a global communicator, and receivers can read it.

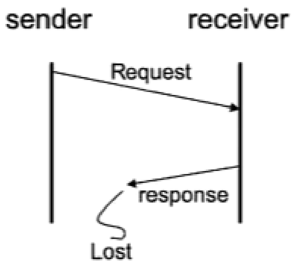
Send_any(service_ID, message)
Receive_any(process_ID, message)

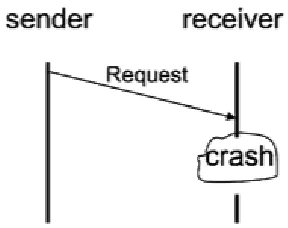
Failure Handling

3 Types:

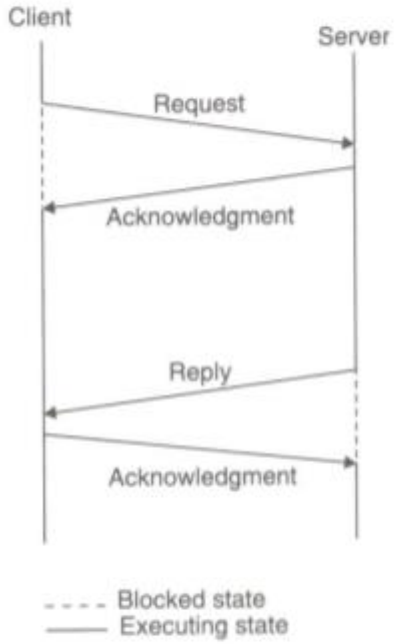
1. Loss of request message
2. Loss of response message
3. Unsuccessful execution of request





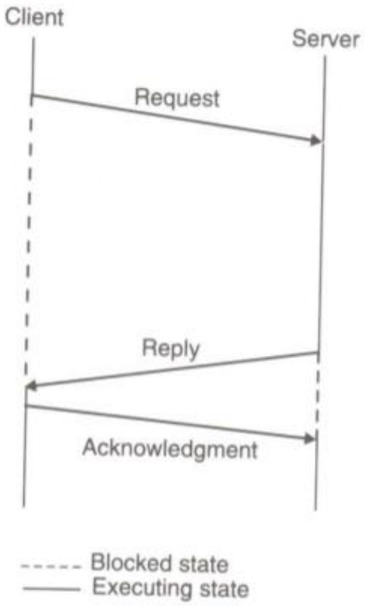


4-message reliable IPC protocol



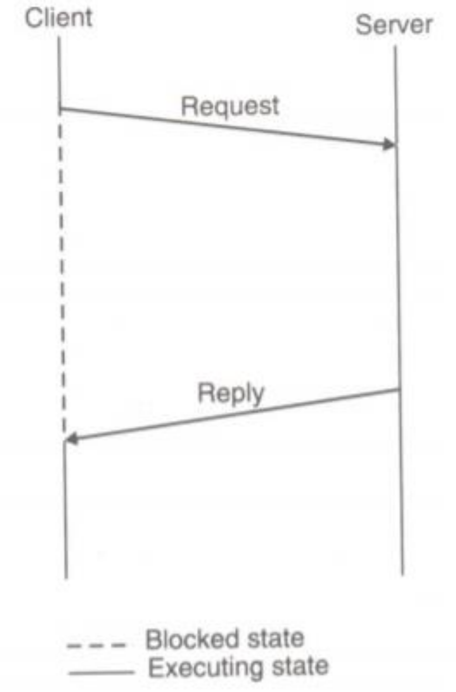
3-message reliable IPC protocol

Server does not acknowledge request. Client may utilize a timeout protocol, sending the request again if it never receives a response.

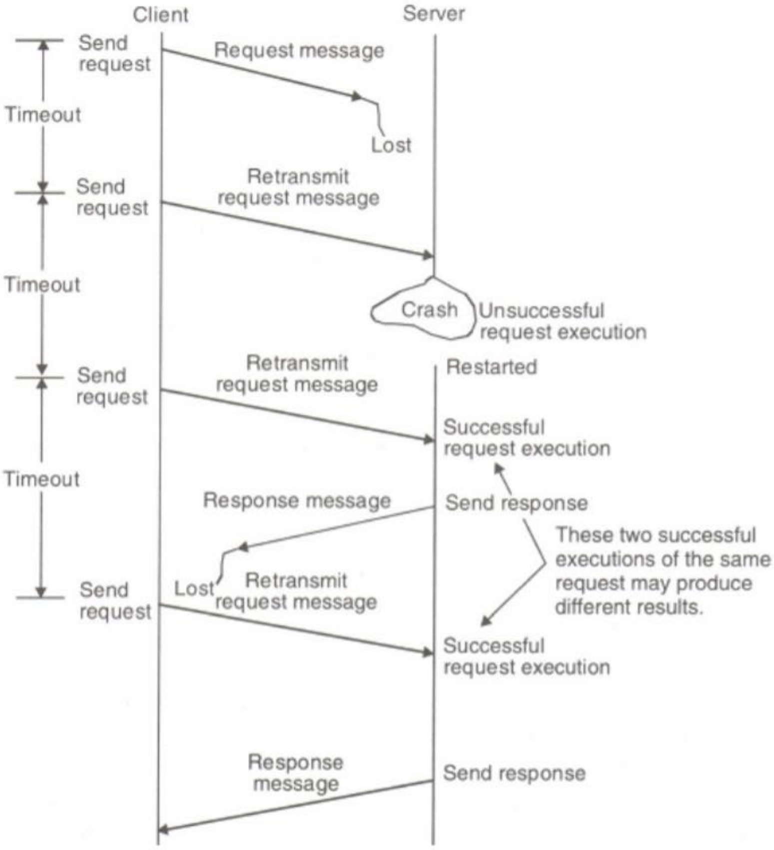


2-message reliable IPC protocol

Not very reliable.

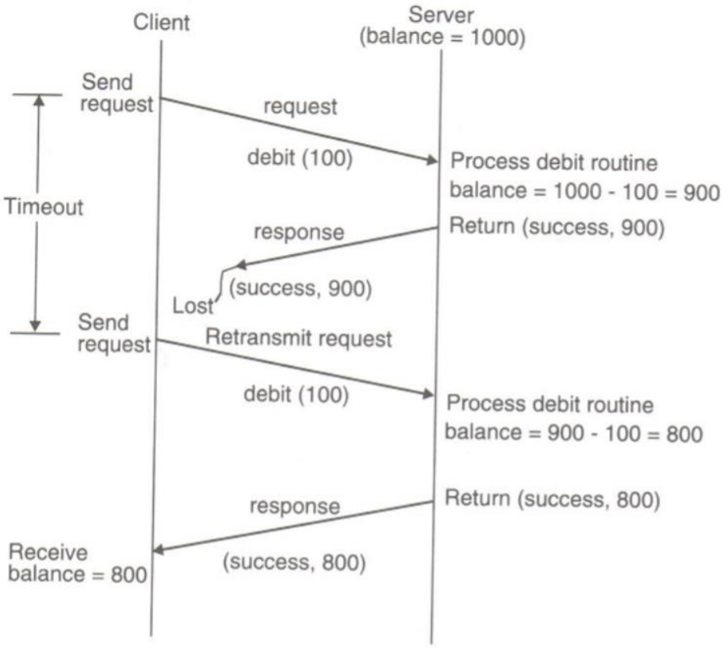


Example of 4-message IPC protocol



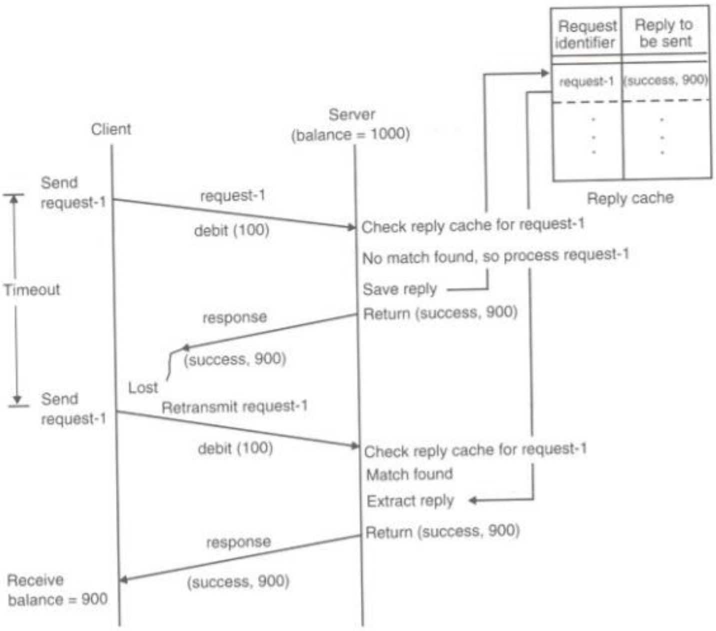
Idempotency – Same results should output when supplying the same input. Similar to purity of functions. Idempotent operations are encouraged.

A Non-Idempotent Routine



Implementing Idempotency

- ✓ Adding sequence numbers in operations to distinguish between retransmitted requests/responses.
- ✓ Using 'Reply cache'

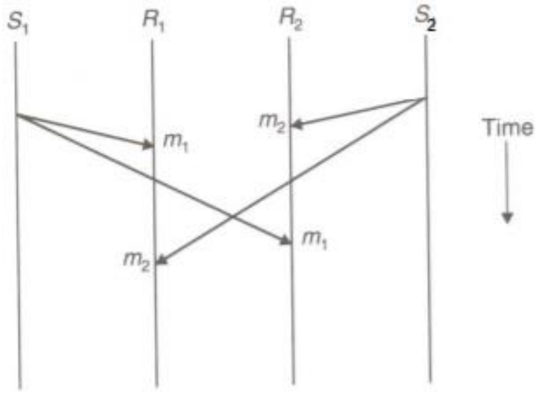


Group Communication

1. One to Many
    - ✓ Group management
    - ✓ Group addressing
    - ✓ Buffered & unbuffered multicast
    - ✓ Send-to-all & Bulletin-Board semantics
    - ✓ Flexible reliability in multicast communication
    - ✓ Atomic multicast
2. Many to One
3. Many to Many
    - ✓ Issues related to 1:M and M:1 also applies here
    - ✓ Ordered message delivery is an important issue

Semantics for ordered delivery in M:M communication

1. No Ordering

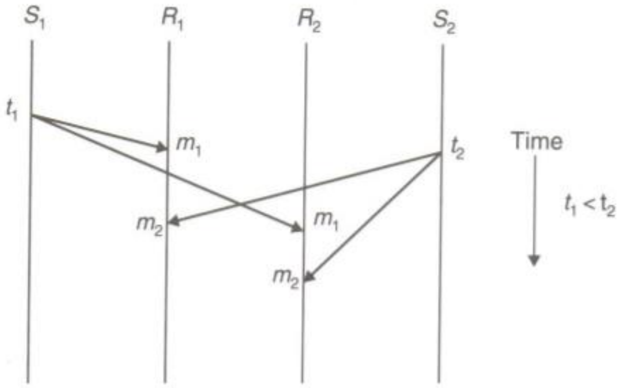


4 Processes. 2 sending processes S1 & S2, 2 receiving processes R1 & R2. S1 & S2 both performed a 1:M message.

In this example, S2 sent message first, then S1 sent message second,

However, R2 received S2’s message first, then S1’s message second.

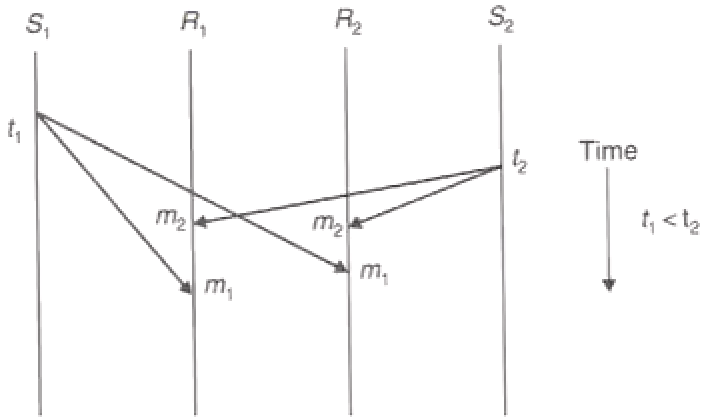
2. Absolute Ordering



- ✓ All messages are delivered to all receiver processes in the exact order they were sent
- ✓ Using global timestamp as message identifiers with sliding window protocol
- ✓ S1 sent first, so both R1 and R2 receive S1’s message first

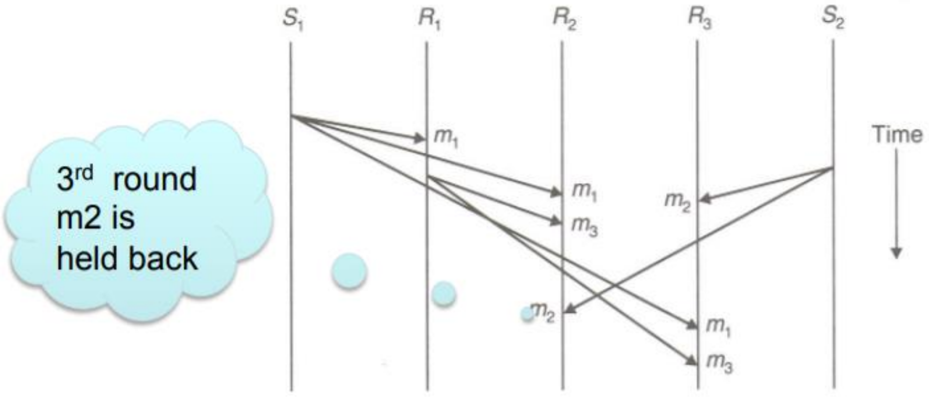
3. Consistent Ordering

- ✓ All messages are delivered to all receivers in the same order. However, this order may be different from the order in which the messages were sent.
- ✓ S1 sent message first, but in both cases it was received second, but at least its consistent among the receiving processes



4. Causal Ordering

- ✓ Ordering which occurs when a message must be transmitted before another message can be sent
- ✓ R1 only sends message m3, after it has received message m1.
- ✓ m3 is only sent after m1 has been received by R1.
- ✓ Hence R2 must receive message m1 first, and then message m3 second.



3rd round m2 is held back

Reason for RPC

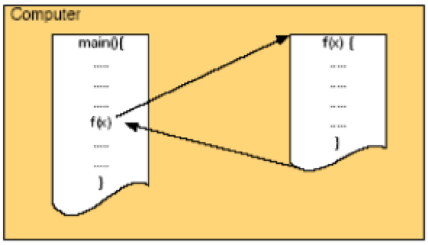
- ✓ IPC of a DS can be handled using an IPC protocol based on a message passing system.
- ✓ However, an independently developed IPC protocol is tailored to one specific application, and does not provide a foundation for building a variety of distributed applications
- ✓ Need was desired for a general IPC protocol, which led to RPC

Remote Procedure Call (RPC)

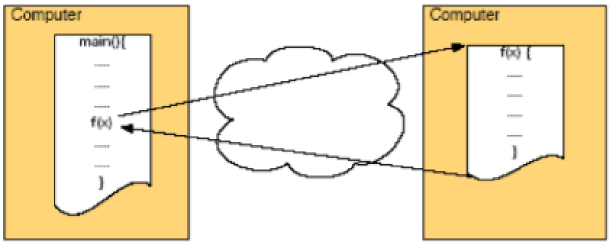
- ✓ the solution for a fairly large number of distributed applications
- ✓ widely accepted IPC mechanism in DS.
- ✓ Features:
  - ➢ Simple call syntax
  - ➢ Familiar semantics
  - ➢ Specifications of a well-defined interface
  - ➢ Ease of use
  - ➢ Generality
  - ➢ Efficiency

RPC Model

- ✓ Similar to “Procedure Call” model except now we invoke procedures on a remote system.
- ✓ Local Procedure Call – “Caller and Callee are within a single process on a given host”
- ✓ Intent of RPC is to make it appear to the programmer that a local procedure call is taking place

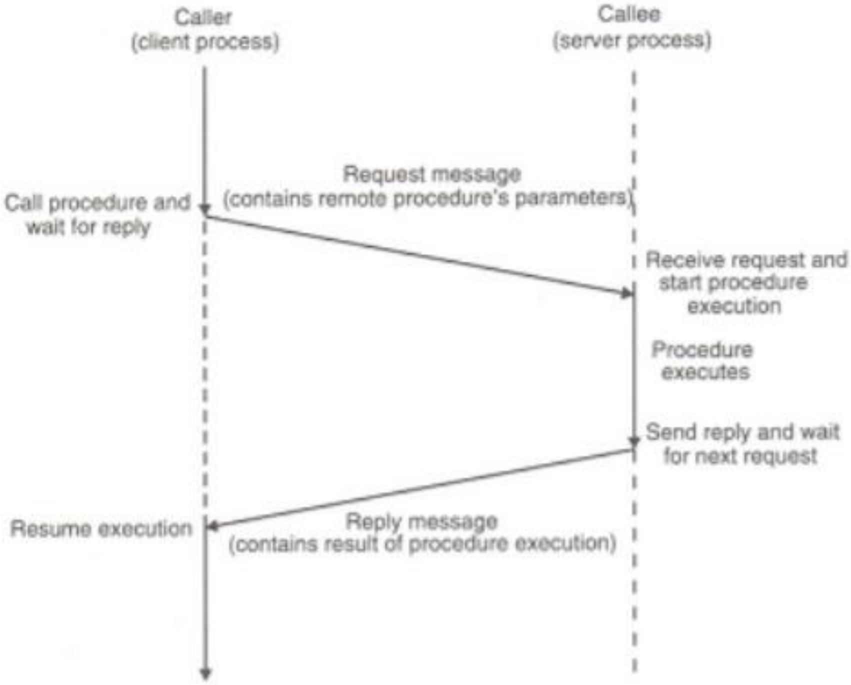


Local Procedure Call



Remote Procedure Call

Typical workflow of RPC



1. Caller issue remote procedure request
2. Callee (remote processing system) starts procedure execution and sends result process back to caller

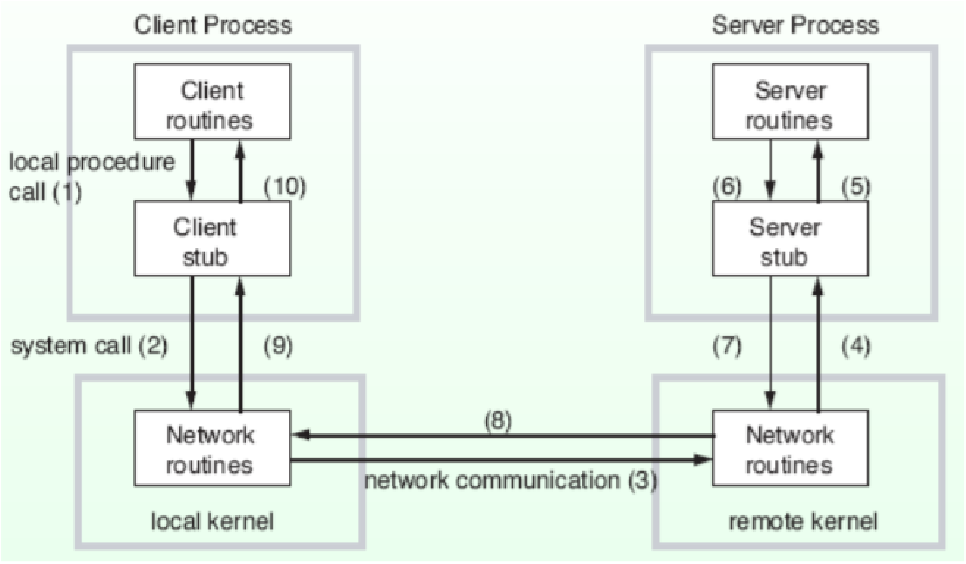
Implementing RPC Mechanism

- ✓ Similar to networking stacks, we have layers as ‘stubs’ in our RPC protocol.
- ✓ Stubs provide a perfectly normal(local) procedure call abstraction
- ✓ Implementation involves five elements:
  - ➢ Client
  - ➢ client stub
  - ➢ RPC Runtime
  - ➢ server stub
  - ➢ server

Stubs

Client & sever stubs are generated from interface definition of server routines by development tools, similar to class definition in C++ & Java.

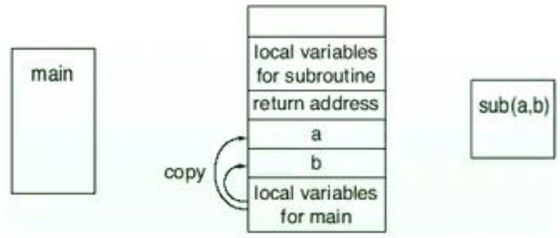
Diagram example



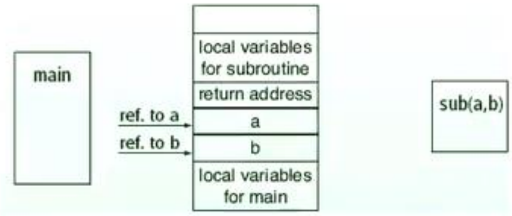
Parameter Passing Mechanisms

When a procedure is called, parameters are passed to the procedure as arguments. There are 3 methods of passing parameters.

1. Call-by-value – supplying an actual value
   The calling procedure may modify value, but modifications do not affect the original value at the calling side

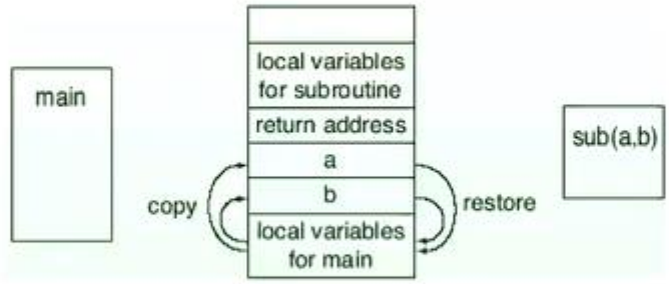


2. Call-by-reference – supplying the memory address of a variable, enabling the calling procedure to manipulate the original values at the calling side



3. Call-by-copy/restore

- ✓ Values of arguments are copied to the stack and passed to the calling procedure.
- ✓ When the processing of procedure completes, the values are copied back to the original values at the calling side.

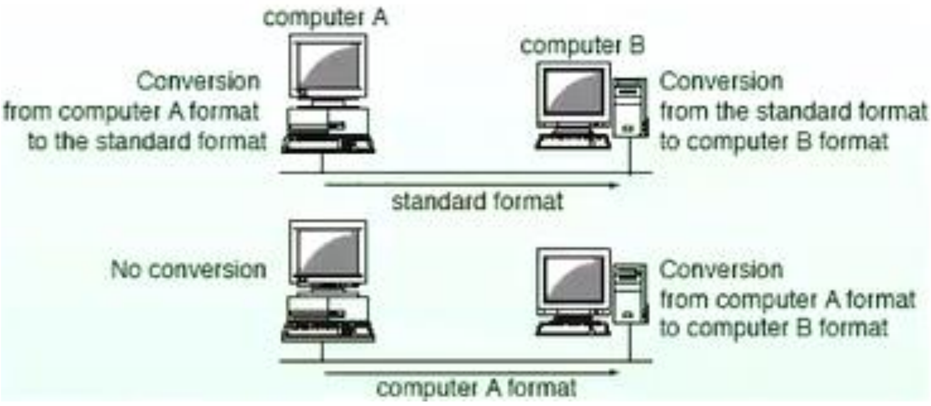


Parameter Passing in RPC

- ✓ We can implement all 3 mechanisms shown above.
- ✓ Usually call-by-value and call-by-copy/restore are used
- ✓ Call-by-reference is difficult to implement. all data which may be referenced must be copied to the remote host and the reference to the copied data is used.

Question: Do we need to convert the values of parameter arguments into a standard format to transmit over the network?

- ✓ Yes, many machines use different character codes, e.g. IBM mainframes use EBCDIC, while PCs use ASCII
- ✓ If a standard format is not used, two message conversions are necessary
- ✓ If format info is attached to a message, only one conversion at receiver will suffice, however, receiver must be able to handle many different formats.

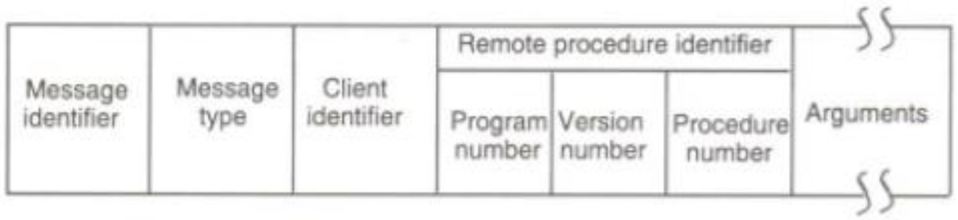


RPC Messages

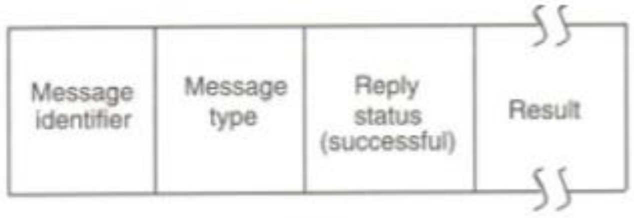
Generally, two types: Call, and Reply messages.

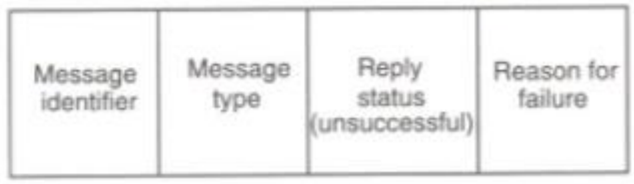
Message Identifier is usually a sequence number.

Call Message format



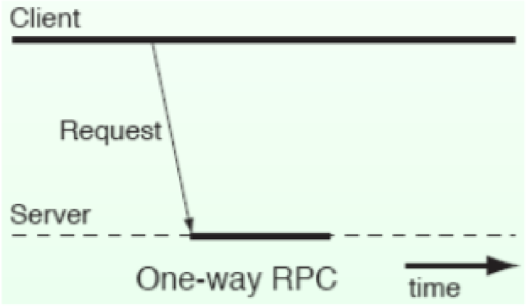
Reply Message format (successful & unsuccessful)





One-way RPC

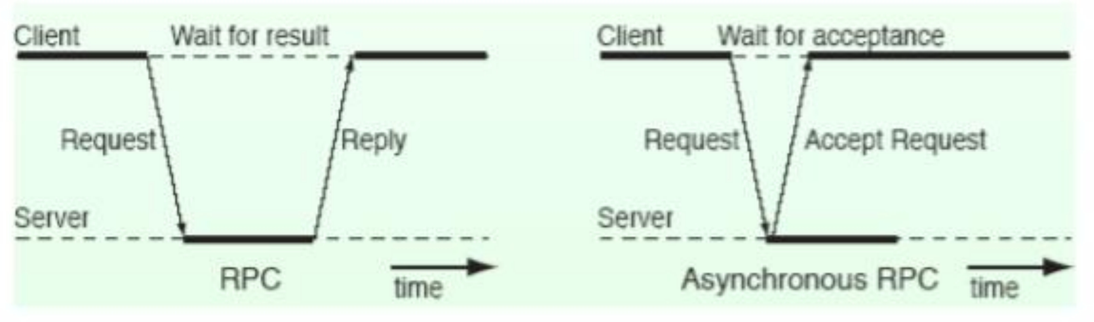
✓ In one-way RPC, the client immediately continues after sending the request to the server



Asynchronous RPC

- ✓ For synchronous RPC, when a client requests a remote procedure, the client wait until a reply comes back
- ✓ If no result is returned, there will be unnecessary wait time overhead
- ✓ In asynchronous RPC, the server immediately sends accept message when it receives a request

Synchronous vs Asynchronous RPC Diagram



Optimizing RPC for better performance/efficiency

- ✓ Distribute RPC requests across multiple processors/servers
- ✓ Have a single server processing requests from multiple clients simultaneously
- ✓ Have client perform serial work on an infrastructure which is optimized for that work, then have servers designed for high-performance, highly parallel throughput
- ✓ Proper selection of timeout values
- ✓ Proper design of RPC protocol specification

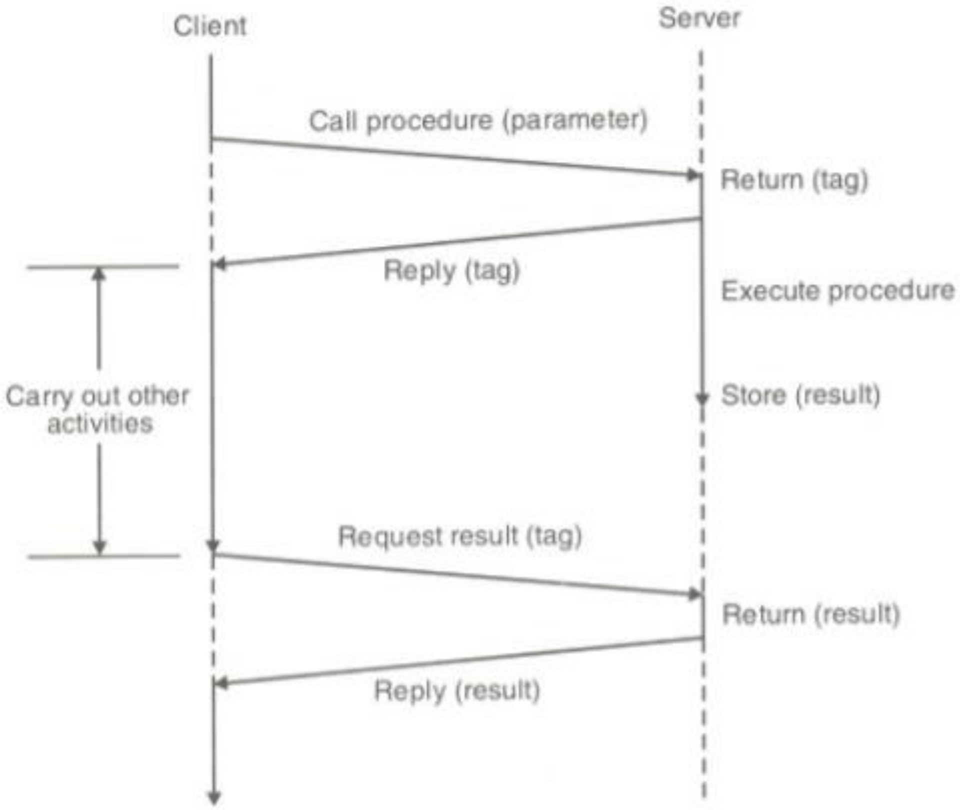
Concurrent Access to Multiple Server

To enable this, one of the following 3 can be adopted:

1. Threads
    - A client process can have multiple threads, which can independently make remote procedure calls to different servers
2. Early reply approach
    - A call is split into two separate RPC calls – one passing parameters and other requesting the result
    - Server must hold the result causing congestion or unnecessary overhead
3. Call buffering approach
    - Clients & severs do not interact directly but via a call buffer server

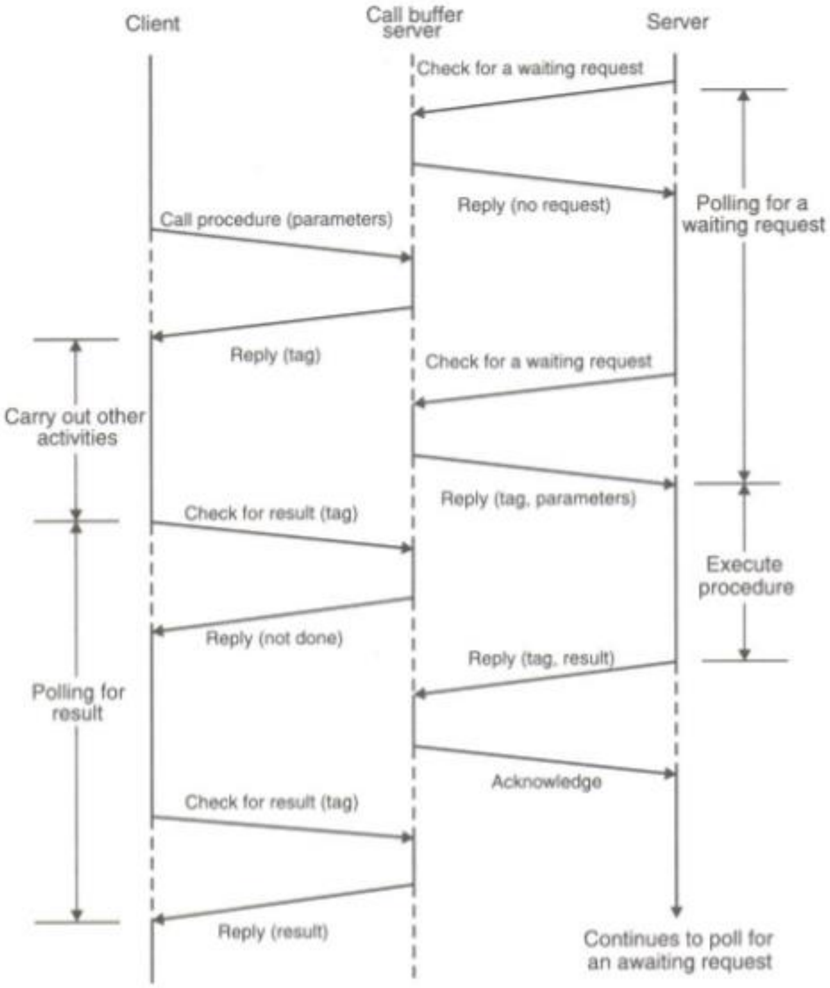
Early Reply Flow Diagram

✓ Client is still processing while server is executing the procedure.



Call Buffering Flow Diagram

- ✓ Client pings the buffer whether its request has been serviced
- ✓ Similar to how RESTFul API's work



Serving Multiple Requests Simultaneously

Delays are common in RPC such as:

- ✓ server needs to wait for shared-resources, delays may occur if workload is not shared properly
- ✓ server calls a remote function involving lots of data or computation or transmission delays

So, it is preferred if the server may accept & process other requests while waiting to complete a request. Multiple-threaded server may be a solution.