Unit 2:

Communication in Distributed Systems

References:

1. G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**
2. Andrew S. Tanenbaum and Maarten van Steen; **Distributed Systems: Principles and Paradigms, 2nd Edition.**

2. Communication in Distributed Systems

2.1 Inter-process Communication

- 2.1.1 Message passing
- 2.1.2 Remote Procedure Calls (RPC)
- 2.1.3 Communication between distributed objects
- 2.1.4 Remote Method Invocation (RMI)
- 2.1.5 Events and Notifications

2.2 Group Communication

- 2.2.1 Multicast
- 2.2.2 Publish/subscribe systems
- 2.2.3 Consistency models.

IPC - Introduction

- Process of communication between two processes that reside in same or different systems.

- E.g. communication of client and server
  - Here a process of client communicates with another process of server for a specific purpose.

IPC - Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation (Communication)
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Dangers of process cooperation (Communication)
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing

Purposes for IPC

- Data Transfer
- Sharing Data
- Event notification
- Resource Sharing and Synchronization
- Process Control

IPC Mechanisms in DS

- *Mechanisms used for communication and synchronization*
  - the different ways of communication between the processes.

1. **Message Passing**
2. RPC

Message Passing

- In a *Message passing* there are no shared variables. IPC facility provides two operations (primitives) for fixed or variable sized message:
  - *Send()*

```
send(void *sendbuf, int nelems, int dest)
```

  - *Receive()*

```
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
P0                      P1
a = 100;                receive(&a, 1, 0)
send(&a, 1, 1);         printf("%d\n", a);
```

Message Passing

- If processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link*
  - exchange messages via *send and receive*

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., syntax and semantics, abstractions)

Implementation Questions

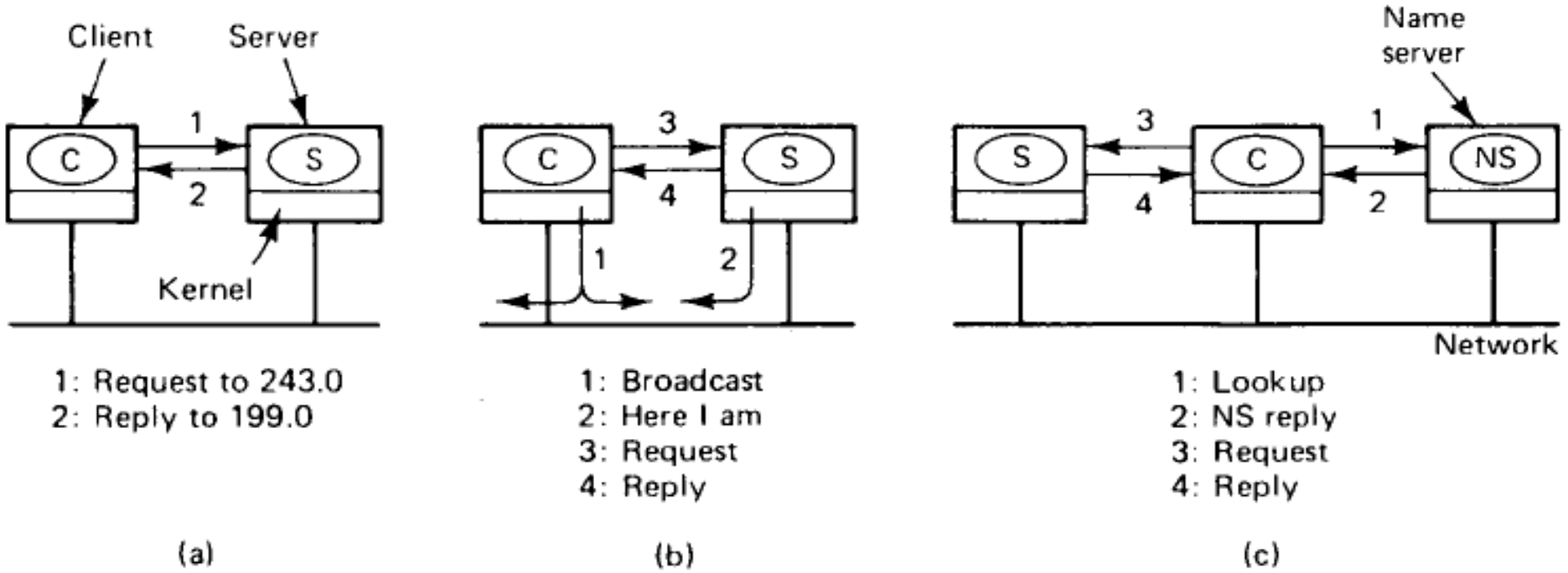
- How are links established?
- Can a link be associated with more than two processes?
- How are links made known to processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Message Passing Systems

1. Addressing – Direct and Indirect
2. Communications
    1. Direct and Indirect
    2. Synchronous and Asynchronous
    3. Persistent and Transient
    4. Discrete and Streaming
3. Synchronization of Primitives
4. Message Format: fixed or variable sized messages

Addressing

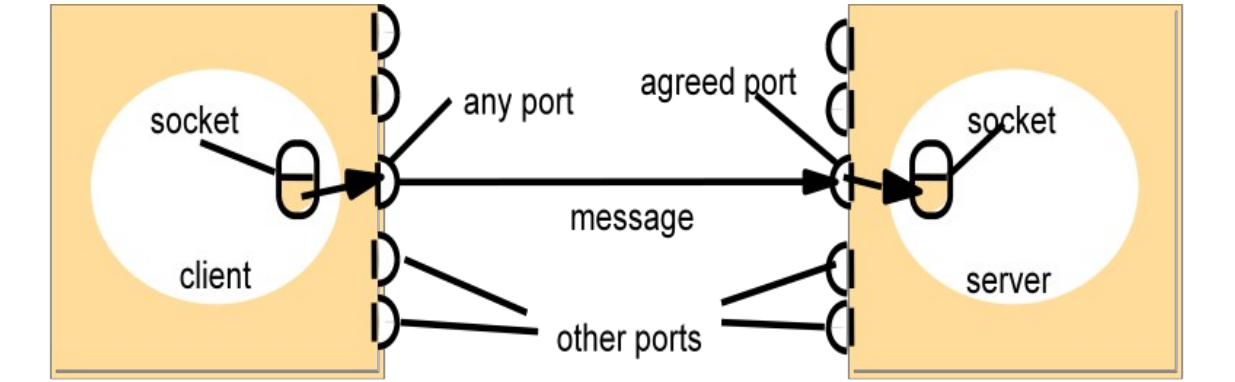
- Direct addressing
  - Send primitive include the identifier of the receiving process
  - Receive can be handled in two ways
    - Receiving process explicitly designates the sending process (effective for cooperating processes) – Symmetric Addressing
    - Receiving process is not specifying the sending process (known as implicit addressing); in this case, the source parameter of receive primitive has a value returned when the receive operation has been completed – Asymmetric Addressing
- Indirect addressing
  - The messages are not sent directly from sender to receiver, but rather they are sent to a shared data structure consisting of queues that temporarily can hold messages; those are referred to as ***mailboxes***.
  - Two communicating process:
    - One process sends a message to a mail box
    - Receiving process picks the message from the mailbox



a) Machine Process addressing
b) Process addressing with broadcasting
c) Address lookup via name server

Communications

1. Direct and Indirect
2. Synchronous and Asynchronous
3. Persistent and Transient
4. Discrete and Streaming



Sockets

Direct Communication – *Sockets*

- Processes must name each other *explicitly*:
  - Symmetric Addressing
    - *send (message,size,P)* – send to process P
    - *Receive(message,size,Q)* – receive from Q
  - Asymmetric Addressing
    - *send (message,size,P)* – send to process P
    - *Receive(message,size,id)* – receive from any system that sets id = sender

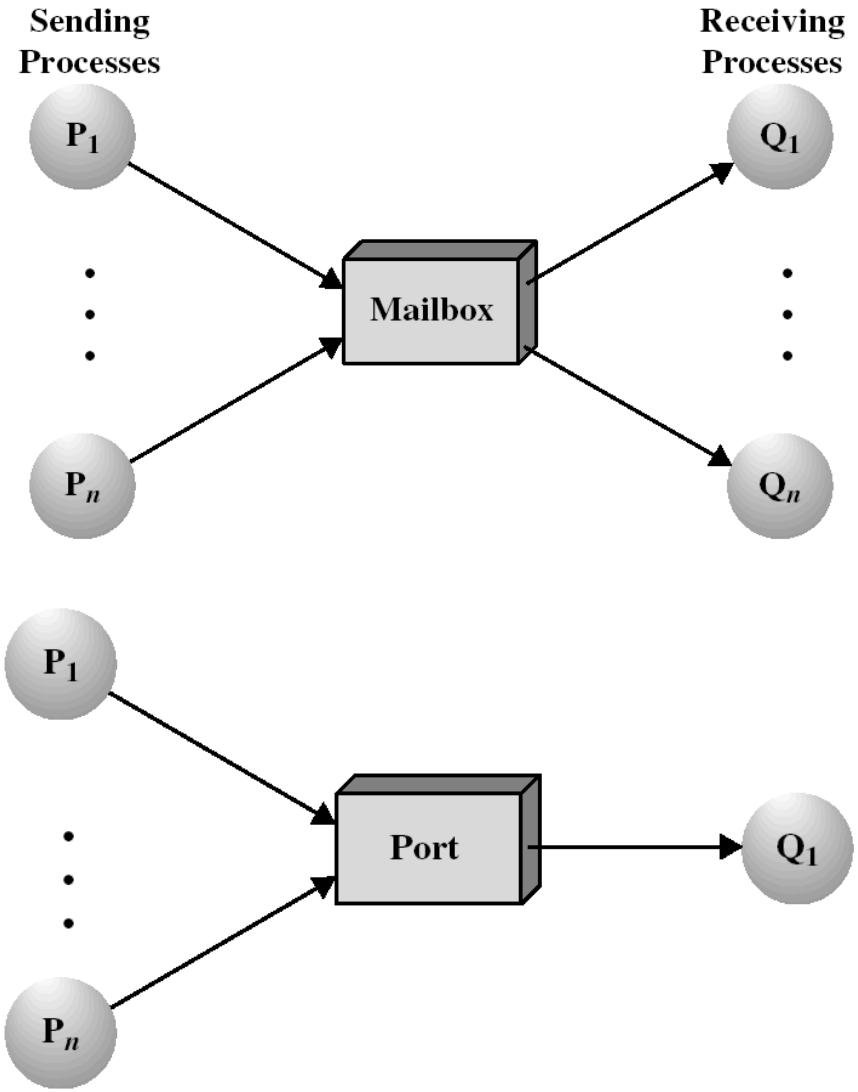
- Properties of communication link
  - Links established automatically between pairs
  - processes must know each others ID
  - Exactly one link per pair of communicating processes

- Disadvantage: a process must know the name or ID of the process(es) it wishes to communicate with

Indirect Communication - *Pipes*

- Messages are sent to or received from *mailboxes* (also referred to as *ports*).
  - *Send(message,size,A)* – send a message to mailbox A
  - *Receive(message,size,A)* – receive a message from mailbox A
- Each mailbox has a unique id.
- Processes can communicate only if they share a mailbox.
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with more than 2 processes.
  - Each pair of processes may share several communication links.

Indirect process communication



- Indirect addressing decuples the sender form the receiver allowing for greater flexibility.
- Relationship between sender and receiver:
  - One to one
    - Private communications link to be set up between two processes
  - Many to one
    - Useful for client server interaction; one process provides services to other processes; in this case, the mailbox is known as port
  - One to many
    - Message or information is broadcasted across a number of processes
  - Many to many

Indirect Communication

- Mailbox sharing:
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation – need of synchronization by mutual exclusion
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was

Synchronous vs Asynchronous Primitives

- Synchronous
  - SEND primitive is blocked until corresponding RECEIVE primitive is executed at the target computer
- Asynchronous
  - Messages are buffered
  - SEND primitive is not blocked even if there is no corresponding execution of the RECEIVE primitive
  - The corresponding RECEIVE primitive can be either blocking or non-blocking

Asynchronous vs Synchronous Communication

- **Synchronous**: sender is blocked until
  - The OS or middleware notifies acceptance of the message, *or*
  - The message has been delivered to the receiver, *or*
  - The receiver processes it & returns a response. (Also called a rendezvous) –this is what we've been calling synchronous up until now.
- **Asynchronous**: (non-blocking) sender resumes execution as soon as the message is passed to the communication/middleware software
  - Message is buffered temporarily by the middleware until sent/received

Persistent versus Transient Communication

- **Persistent**: messages are held by the middleware comm. service until they can be delivered. (Think email)
  - Sender can terminate after executing send
  - Receiver will get message next time it runs
- **Transient**: Messages exist only while the sender and receiver are running
  - Communication errors or inactive receiver cause the message to be discarded.
  - Transport-level communication is transient

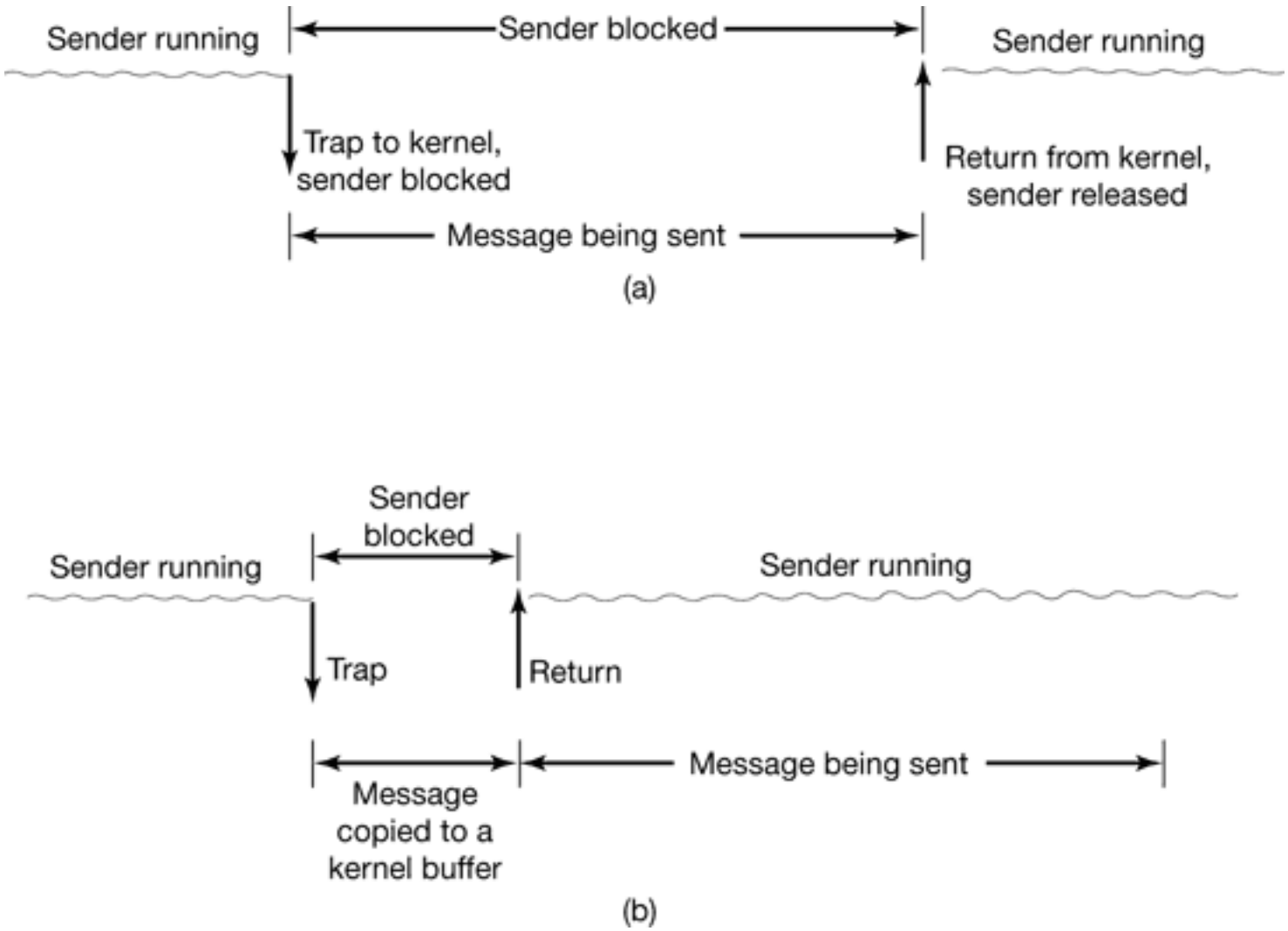
Synchronizing the Primitives

1. Blocking vs Non-Blocking Primitives
2. Buffered vs Non-buffered primitives
3. Reliable vs Non-Reliable Primitives

Blocking vs Non-Blocing Primitives

- Blocking (Synchronous)
  - **Blocking send**: sender blocked until message received by mailbox or process
  - Blocking receive: receiver blocks until a message is available

- Non-Blocking (Asynchronous)
  - Nonblocking send: sender resumes operation immediately after sending
  - Nonblocking receive: receiver returns immediately with either a valid or null message.



(a) A blocking send primitive. (b) A nonblocking send primitive.

Advantages and Disadvantages

- Blocking
  - Advantage : Program's behavior is predictable
  - Drawback → Lack of flexibility in programming
- Non-Blocking
  - Advantage : Programs have maximum flexibility in performing computation and communication in any order
  - Drawback
  - Sender can not modify the message buffer until the message has been sent
  - Overwriting the message during transmission are too horrible.

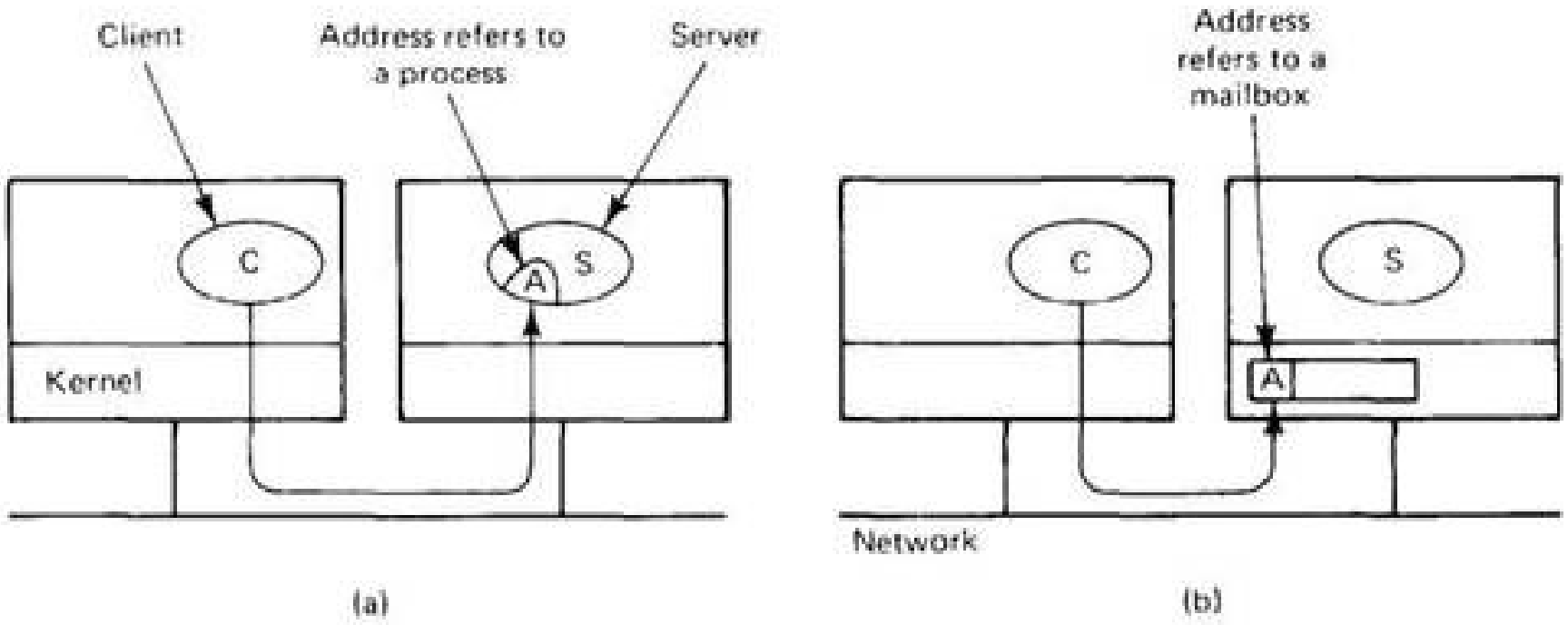
Blocking vs Non-Blocing Primitives

- Blocking send (CPU idle during message transmission)
- Non blocking send with copy (CPU time waste for extra copy)
- Non blocking send with interrupt (→ Programming becomes tricky and difficult)

Buffered versus Unbuffered Primitives

- Unbuffered
  - Receiver tells the kernel it's waiting
  - The kernel stores one message to one address
  - The receiver unblocks and retrieves the message
  - Could lose a message if sender sends before receiver calls receive
- Buffered
  - Kernel stores received messages in a "mailbox"
  - Advantages:
    - No missed communications
    - Kernel knows where to store messages
  - Disadvantages:
    - Buffer space management
    - Mailboxes can fill up

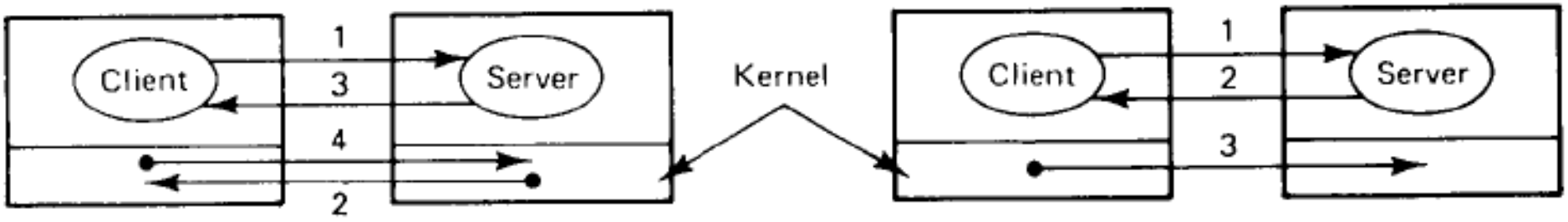
Buffered versus Unbuffered Primitives(2)



a) Unbuffered message passing. (b) Buffered message passing.

Reliable versus Unreliable Primitives

- Unreliable
  - Without request and acknowledgement
- Three Approaches of reliability
  - Strobe (no guarantee about msg delivered e.g. post office)
  - Single handshaking (req-ack)
  - Double handshaking
- Example:
  - Unreliable: Nonblocking-unbuffered
  - Reliable: blocking-buffered and others.
- Problem:
  - The network itself can lose messages



1. Request (client to server)
2. ACK (kernel to kernel)
3. Reply (server to client)
4. ACK (kernel to kernel)

1. Request (client to server)
2. Reply (server to client)
3. ACK (kernel to kernel)

Double Handshaking (the sending kernel can resend the request to guard against the possibility of lost message)

Single handshaking

Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.

Non-Buffered Blocking Message Passing Operations

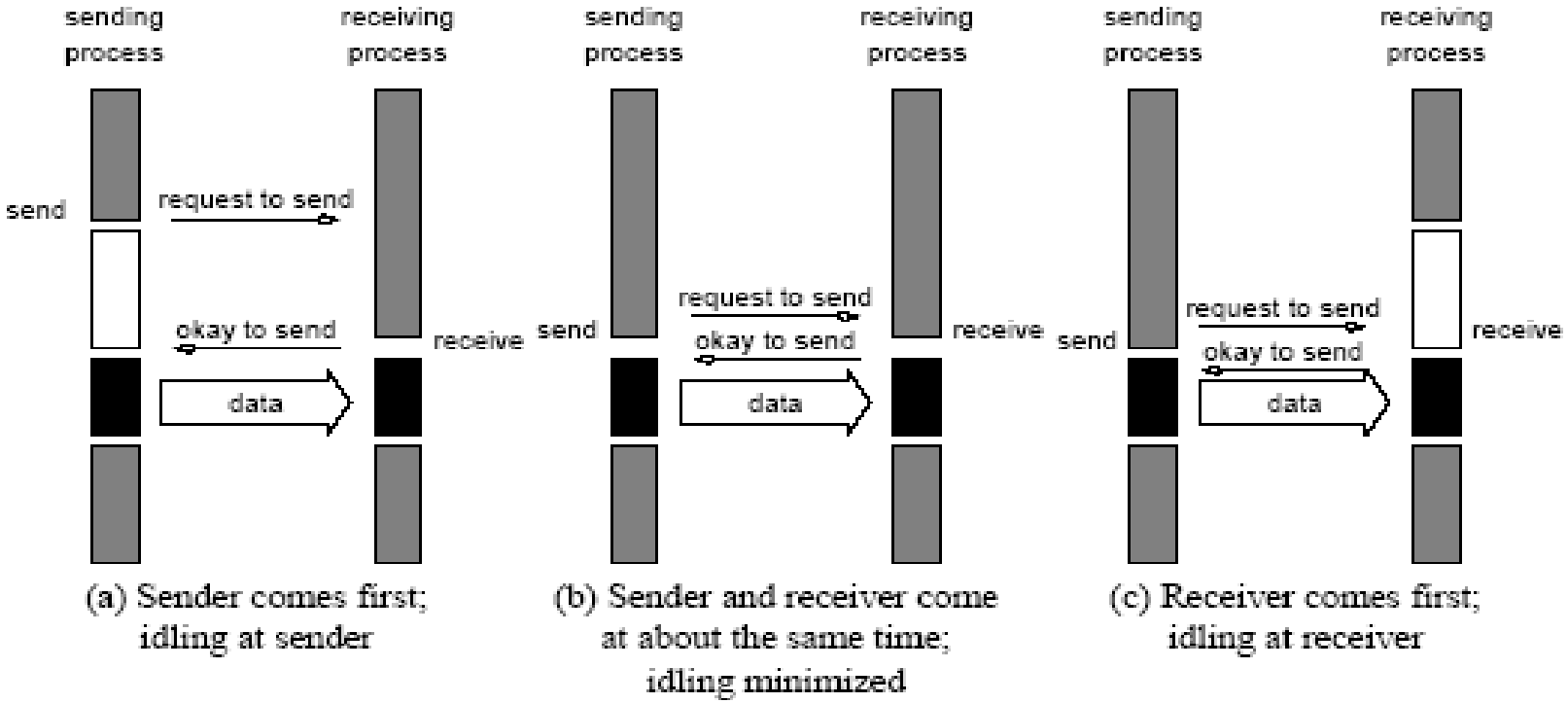


Fig: Handshake for a blocking non-buffered send/receive operation.

It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering reduce idling at the expense of copying overheads.

Buffered Blocking Message Passing Operations

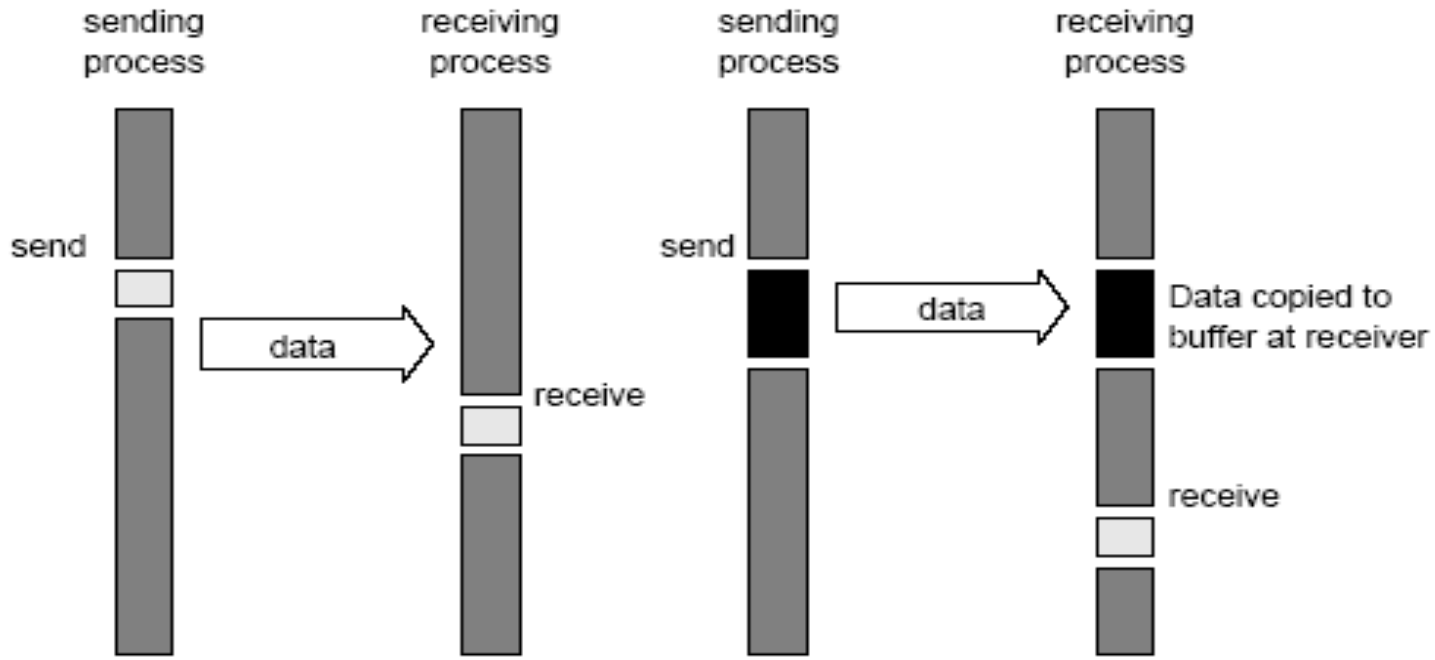


Fig: Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Implementing Client Server Model (1)

- Virtually all networks have maximum packet size, typically a few thousand bytes at most.
- Message larger than this must split up into multiple packets and sent separately.
- Some packet may arrive in wrong order.
- Message number could be one solution.
- Acknowledgement could be another solution.
  - Individual packet acknowledgement
  - Entire message acknowledgement

Implementing Client Server Model (2)

- If a packet is lost, only that packet has to be retransmitted
- It has the disadvantage of requiring more packets on the network.

- Entire message acknowledgement have more complicated recovery when a packet is lost.

- Another issues is the underlying protocol used in client server communication.

Implementing Client Server Model

Item	Option 1	Option 2	Option 3
Addressing	Machine number	Sparse process addresses	ASCII names looked up via server
Blocking	Blocking primitives	Nonblocking with copy to kernel	Nonblocking with interrupt
Buffering	Unbuffered, discarding unexpected messages	Unbuffered, temporarily keeping unexpected messages	Mailboxes
Reliability	Unreliable	Request-Ack-Reply Ack	Request-Reply-Ack

Table: Four design issues for the communication primitives and some of the principle choices available

IPC Mechanisms in DS

- *Mechanisms used for communication and synchronization*
  - the different ways of communication between the processes.

1. **Message Passing**
2. RPC

Remote Procedure Call (RPC)

- RPC is an interaction between a client and a server
- Client invokes procedure on sever
- Server executes the procedure and pass the result back to client
- Calling process is suspended (blocked) and proceeds only after getting the result from server

RPC - Motivation

- Transport layer message passing consists of two types of primitives: send and receive
  - May be implemented in the OS or through add-on libraries
- Messages are composed in user space and sent via a send() primitive.
- When processes are expecting a message they execute a receive() primitive.
  - Receives are often blocking
- Messages lack *access transparency*.
  - Differences in data representation, need to understand message-passing process, etc.
- Programming is simplified if processes can exchange information using techniques that are similar to those used in a shared memory environment.

The Remote Procedure Call (RPC) Model

- A high-level network communication interface
- Based on the single-process procedure call model.
- **Client request**: formulated as a procedure call to a function on the server.
- **Server's reply**: formulated as function return

Conventional Procedure Calls

1. Initiated when a process calls a function or procedure
2. The caller is “suspended” until the called function completes.
3. Arguments & return address are pushed onto the process stack.
4. Variables local to the called function are pushed on the stack
5. Control passes to the called function
6. The called function executes, returns value(s) either through parameters or in registers.
7. The stack is popped.
8. Calling function resumes executing

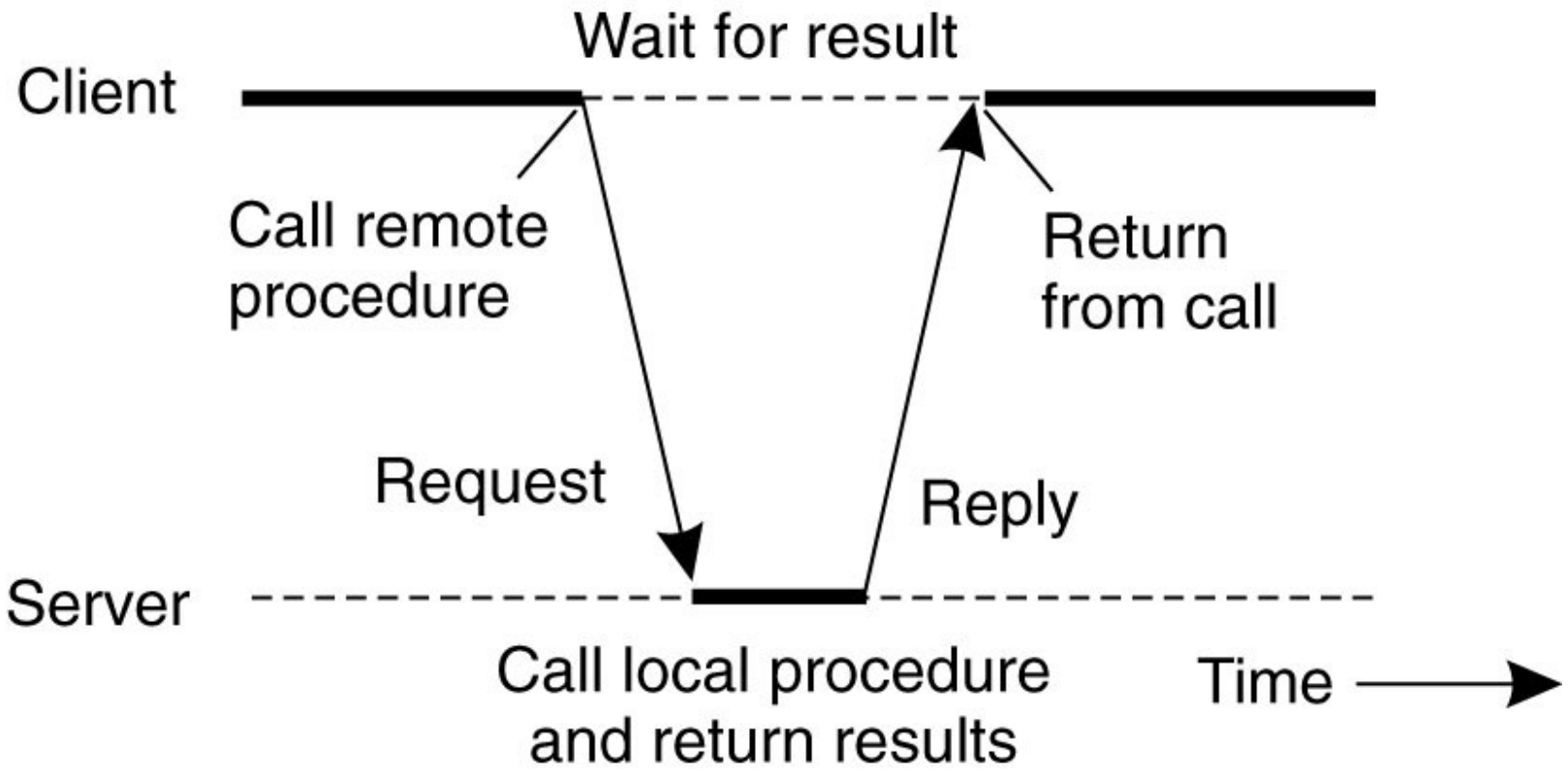


Figure: Principle of RPC between a client and server program.

Types of RPC

1. **Lightweight RPC :** It is a method of optimizing RPCs when client and server processes are both running on the same machine.
   a. Instead of sending an explicit network message, the client just passes a buffer from client to the server via a shared memory region where client puts in the RPC request and the parameters and tells the server to access it from there.
   b. No explicit message passing is needed as memory is shared to send and receive messages.

2. **Synchronous RPC:** It is a blocking call, i.e. when a client has made a request to the server, the client will wait until it receives a response from the server.

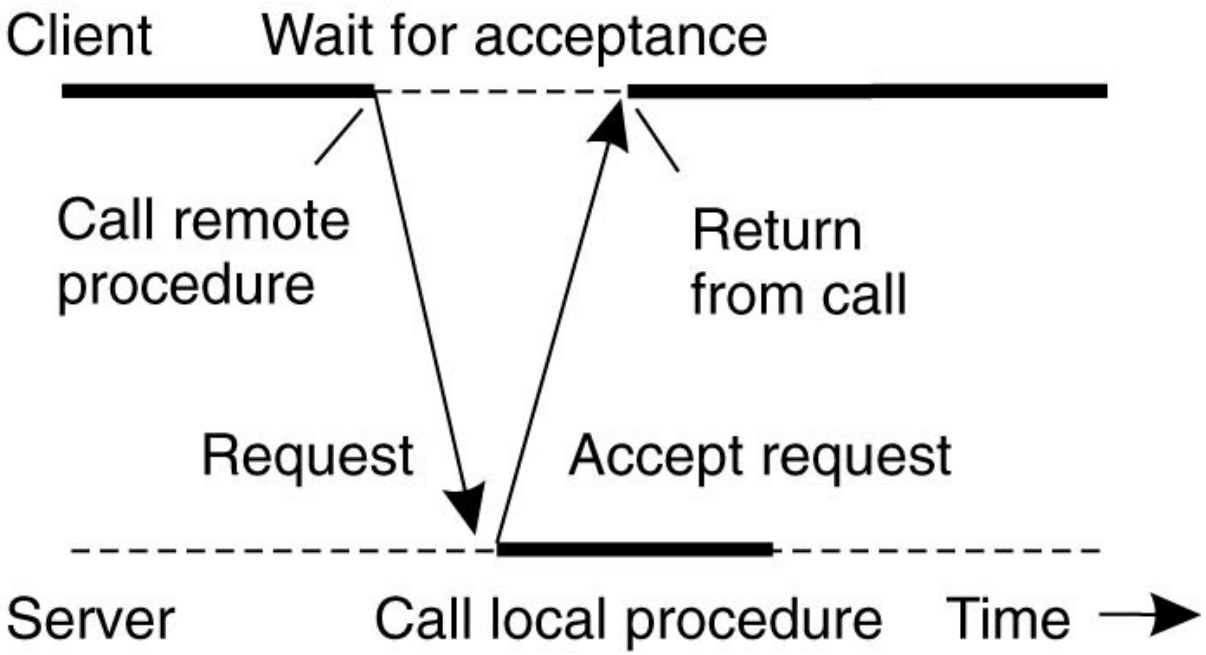
Types of RPC…

3. Asynchronous RPC: It is non blocking call.

- Client makes a RPC call and it waits only for an acknowledgement from the server and not the actual response.
- The server then processes the request asynchronously and send back the response asynchronously to the client which generates an interrupt on the client to read response received from the server.
- This is useful when the RPC call is a long running computation on the server, meanwhile client can continue execution.

Types of RPC…

- **Asynchronous RPC**



(b)

Types of RPC…

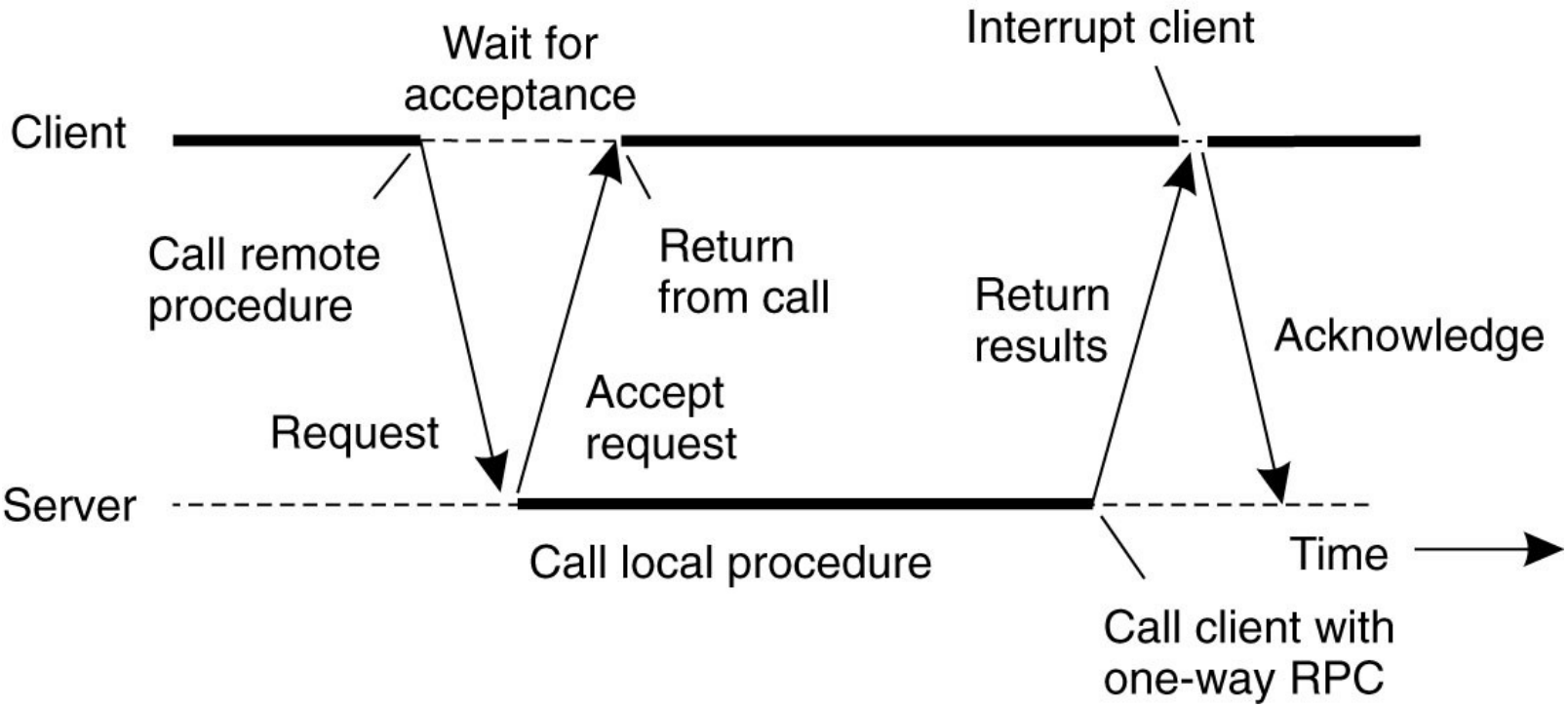
4. Deferred Synchronous RPC: Client and server interact through two asynchronous RPCs.

- The client sends a RPC request to the server, and client waits only for acknowledgement of received request from server, post that the client carries on with its computation.
- Once the server processes the request, it sends back the response to the client which generates an interrupt on client side, the client then sends an response received acknowledgement to the server

5. One-Way RPC:

- The client sends an RPC request and doesn't wait for an acknowledgement from the server, it just sends an RPC request and continues execution.
- The reply from the server is handled through interrupt generated on receipt of response on client side.
- The downside here is that this model is not reliable. If it is running on non-reliable transport medium such as UDP, there will be no way to know if the request was received by the server.

Type of RPC…



- Fig: A client and server interacting through two asynchronous RPCs.

RPC and Client-Server

- RPC forms the basis of most client-server systems.
- Clients formulate requests to servers as procedure calls
- **Server's reply**: formulated as function return

Problem with Conventional RPC

Access Transparency in not maintained

(Solution: Using Stubs)

Transparency Using Stubs

- Stub procedures (one for each RPC i.e. client and server)
- For procedure calls, control flows from
  - Client application to client-side stub
  - Client stub to server stub
  - Server stub to server procedure
- For procedure return, control flows from
  - Server procedure to server-stub
  - Server-stub to client-stub
  - Client-stub to client application

Modern RPC

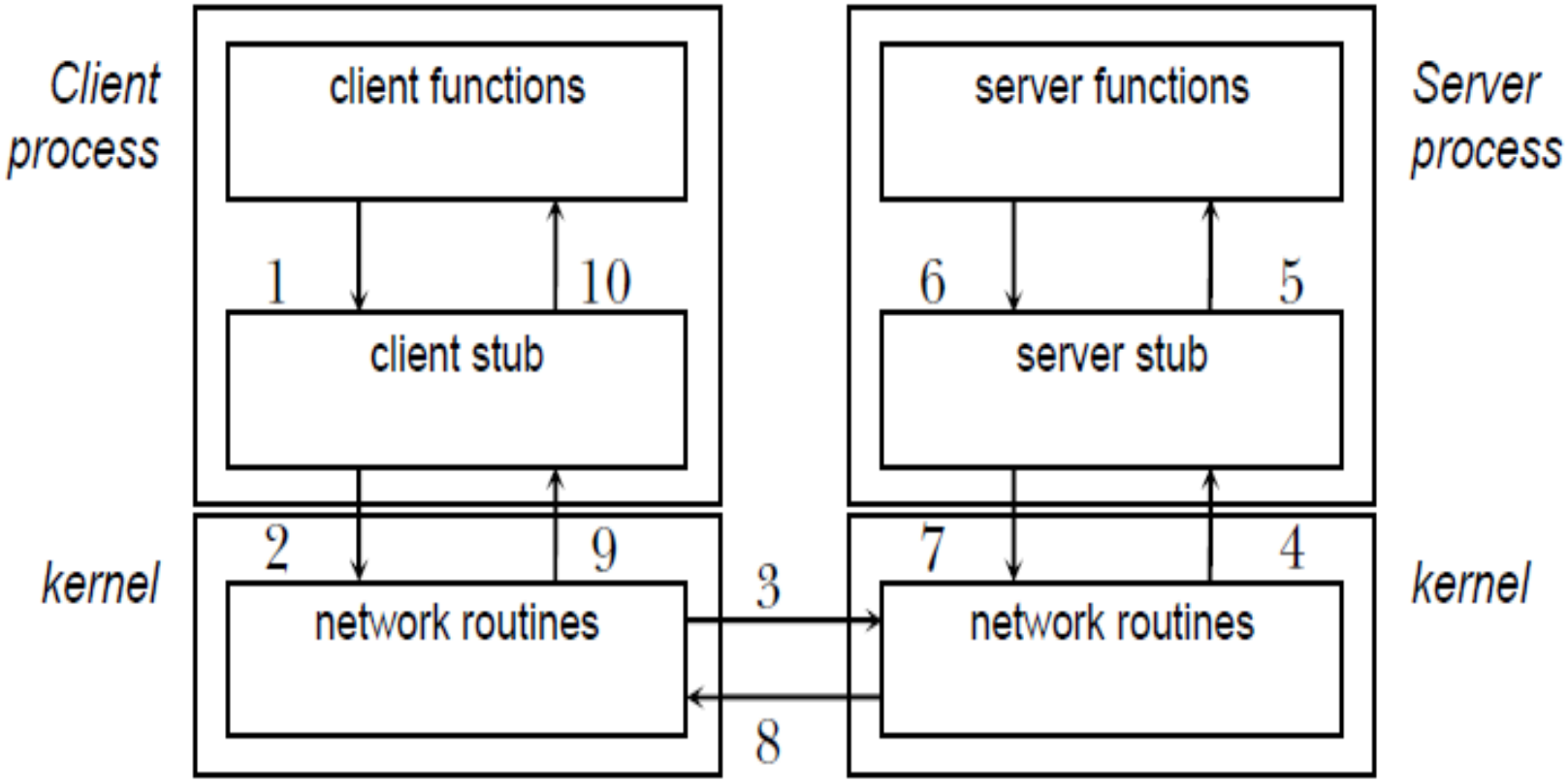


Figure Functional steps in a remote procedure call

Modern RPC

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message including parameters, name or number of procedure to be called etc and calls the local operating system. The packaging of arguments into a network message is called *marshaling*.
3. The client's as sends the message to the remote OS via a system call to the local kernel. To transfer the message some protocol (either connectionless or connection-oriented) are used.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the waiting client procedure.

RPC: Passing Value Parameters

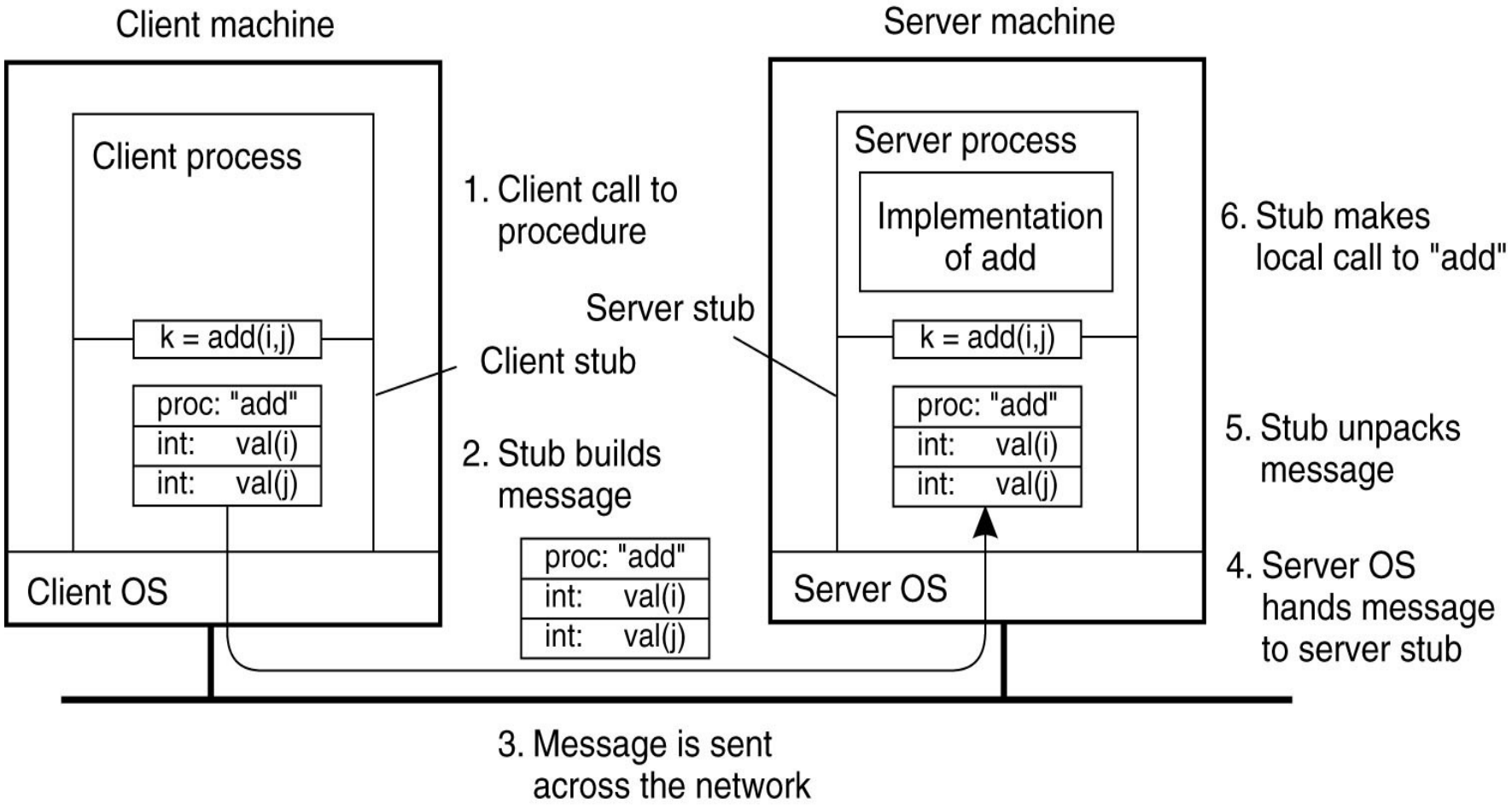


Figure 4-7. The steps involved in a doing a remote computation through RPC.

Issues

- Are parameters call-by-value or call-by-reference?
  - Call-by-value: in same-process procedure calls, parameter value is pushed on the stack, acts like a local variable
  - Call-by-reference: in same-process calls, a pointer to the parameter is pushed on the stack
- How is the data represented?
- What protocols are used?
- For *value parameters*, value can be placed in the message and delivered directly, except …
  - Are the same internal representations used on both machines? (char. code, numeric rep.)
  - Is the representation big endian, or little endian?
  - How integers are represented (1's complement or 2's complement)

How to let two kinds of machines talk to each other?

- A standard should be agreed upon for representing each of the basic data types, given a parameter list (n parameters) and a message.
- devise a network standard or canonical form for integers, characters, Booleans, floating-point numbers, and so on.
- Convert to either little endian/big endian. But inefficient.
- use native format and indicate in the first byte of the message which format this is.

Parameter Passing – Reference Parameters

- Consider passing an array in the normal way:
  - The array is passed as a pointer
  - The function uses the pointer to directly modify the array values in the caller's space
- Pointers = machine addresses; not relevant on a remote machine
- Solution: copy array values into the message; store values in the server stub, server processes as a normal reference parameter.

How can a client locate the server?

- hardwire the server network address into the client.

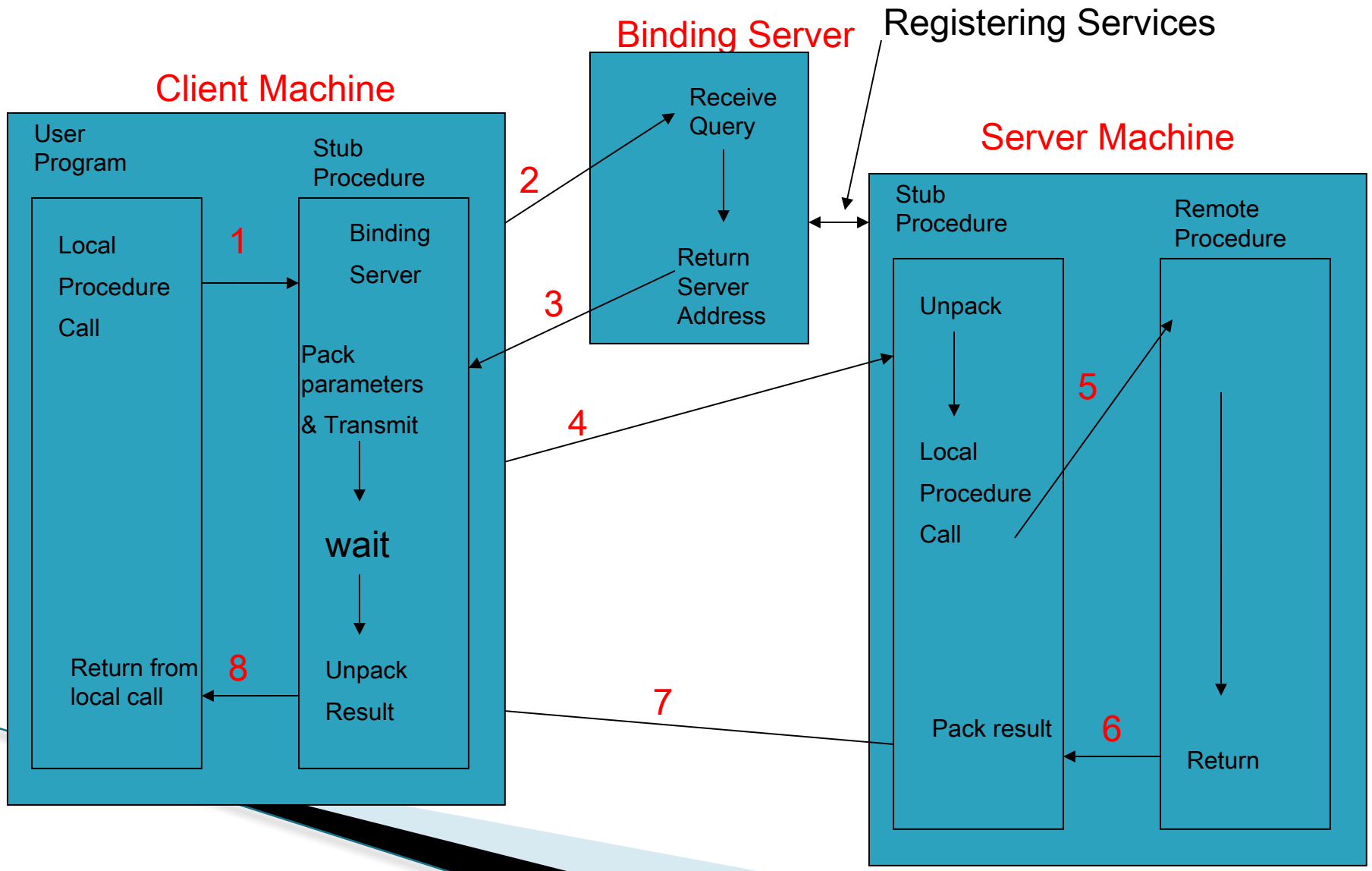
  Disadvantage: inflexible.

- use **dynamic binding** to match up clients and servers.

RPC Issues: Binding

- A local kernel calls remote kernel in RPC along with some parameters through a particular port.
- The local kernel must know the remote's port through which it is communicating with.
- The process of finding out or assigning the port and corresponding system(client or server) is called **binding.**
- Determines remote procedure and machine on which it will be executed
- Checks compatibility of the parameters passed
- **Dynamic Binding**
  - Use Binding Server

Binding


Binding Server
Registering Services
Client Machine
Server Machine
User Program
Stub Procedure
Local Procedure Call
Binding Server
Pack parameters & Transmit
wait
Unpack Result
Return from local call
Receive Query
Return Server Address
Stub Procedure
Remote Procedure
Unpack
Local Procedure Call
Pack result
Return
1
2
3
4
5
6
7
8

How the client locates the server?

- When the client calls one of the remote procedure “read” for the first time, the client stub sees that is not yet bound to a server.
- The client stub sends message to the binder asking to **import** version 3.1 of the file-server interface.
- The binder checks to see if one or more servers have already **exported** an interface with this name and version number.
- If no server is willing to support this interface, the “read” call fails; else if a suitable server exists, the binder gives its handle and unique identifier to the client stub.
- The client stub uses the handle as the address to send the request message to.

Advantages

- It can handle multiple servers that support the same interface
- The binder can spread the clients randomly over the servers to even the load
- It can also poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance
- It can also assist in authentication. Because a server could specify it only wished to be used by a specific list of users
- **Disadvantage**
  - The extra overhead of exporting and importing interfaces cost time.

RPC semantics in the presence of failures

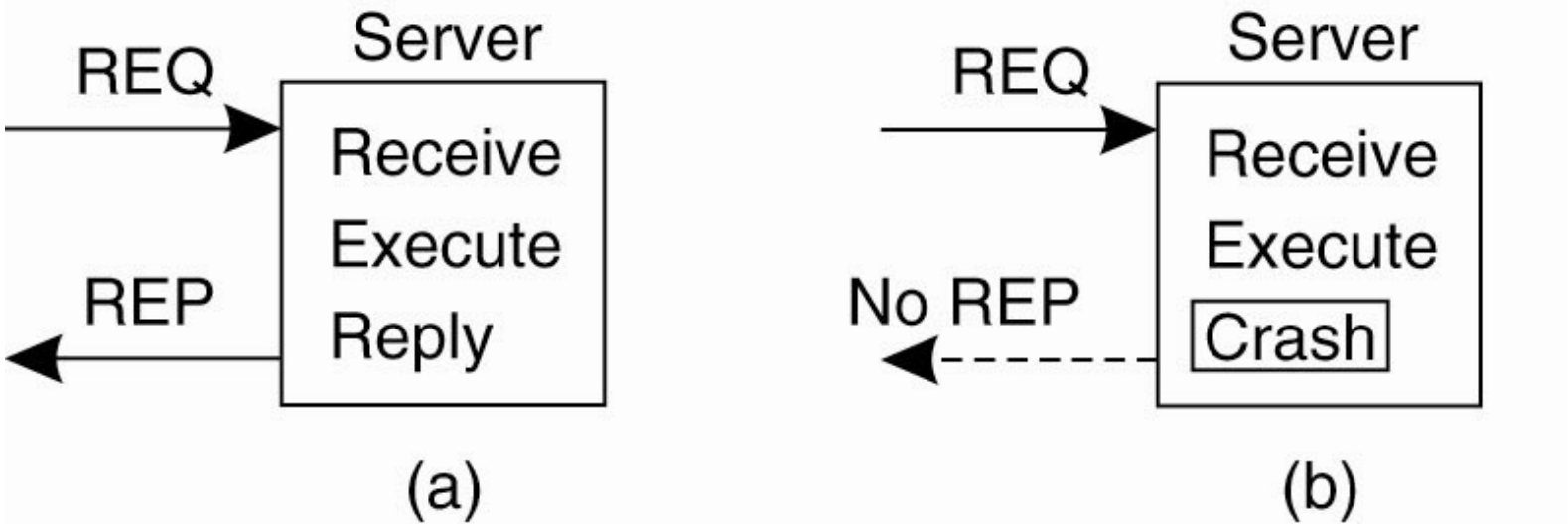
1. The client is unable to locate the server
2. The request message from the client to the server is lost (Lost request message)
3. The reply message from the server to the client is lost (Lost Reply Message)
4. The server crashes after receiving a request (Server Crash)
5. The client crashes after sending a request (Client Crash)

RPC semantics in the presence of failures

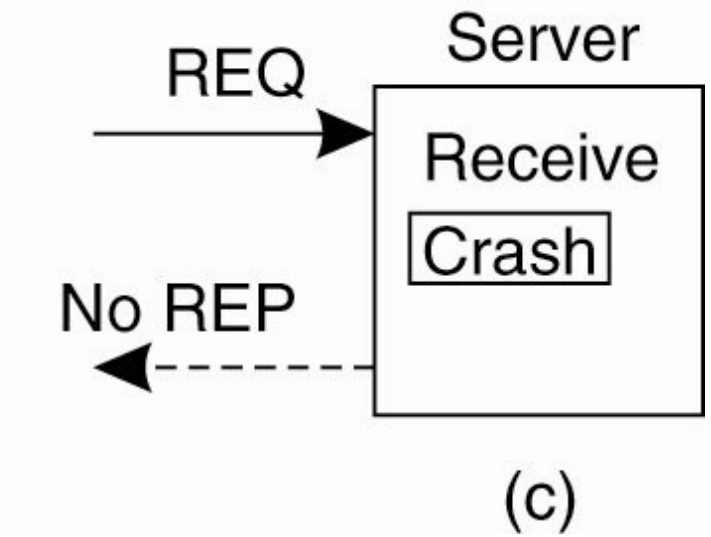
1. Client is unable to Locate Server
   - Possible solution: raise an exception
   - Two drawbacks:
     - Not always easy to write exception handler (for instance there is a big problem if the language used does not support exception handling/signaling of some sort).
     - Use of exception handler may violate the overall requirement of transparency in the distributed system.

2. Lost Request Message
   - Use of timers (to figure out whether a message has been lost).

3. Server Crashes (1)



- A server in client-server communication.
  (a) The normal case.
  (b) Crash after execution.
  (c) Crash before execution.



3. Server Crashes (2)

- The server can crash before the execution or after the execution
- The client cannot distinguish these two.
- The client can:
  - Wait until the server reboots and try the operation again (**at least once semantics**).
  - Gives up immediately and reports back failure (**at most once semantics**).
- Three events that can happen at the server:
  - Send the completion message (M),
  - Print the text (P),
  - Crash (C).

3. Server Crashes(3):Example

- Imagine that the remote operation consists of printing some text, and that the server sends a completion message to the client when the text is printed.
- Also assume that when a client issues a request, it receives an acknowledgment that the request has been delivered to the server.
- There are two strategies the server can follow.
  - It can either send a completion message just before it actually tells the printer to do its work,
  - or after the text has been printed.
- Assume that the server crashes and subsequently recovers.
- It announces to all clients that it has just crashed but is now up and running again.
- The problem is that the client does not know whether its request to print some text will actually be carried out.

3. Server Crashes (3)

- These events can occur in six different orderings:

1. M →P →C: A crash occurs after sending the completion message and printing the text.
2. M →C (→P): A crash happens after sending the completion message, but before the text could be printed.
3. P →M →C: A crash occurs after sending the completion message and printing the text.
4. P→C(→M): The text printed, after which a crash occurs before the completion message could be sent.
5. C (→P →M): A crash happens before the server could do anything.
6. C (→M →P): A crash happens before the server could do anything.

3. Server Crashes (4)

- Different combinations of client and server strategies in the presence of server crashes.

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

RPC Semantics in the Presence of Failures

4. Lost Reply Messages

- Use time-outs (but not certain whether the time outs are due to slow server).
- Some operations can help (those that are idempotent)
- Transactional requests not possible to be deal with! (choose another model).

5. Client Crashes

- If a client sends a request to a server and crashes before the server replies, then a computation is active and no parent is waiting for the result.

- Such an unwanted computation is called an **orphan**.

RPC Semantics in the Presence of Failures

Problems with orphans

- They waste CPU cycles.
- They can lock files or tie up valuable resources.
- If the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result

What to do with orphans?

1. Extermination: Before a client stub sends an RPC message, it makes a log entry telling what it is about to do. After a reboot, the log is checked and the orphan is explicitly killed off.

- **Disadvantage:** The expense of writing a disk record for every RPC; it may not even work, since orphans themselves may do RPCs, thus creating **grandorphans** or further descendants that are impossible to locate.

2. Reincarnation: Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed.

What to do with orphans?(2)

3. Gentle reincarnation: when an epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.

4. Expiration: Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum. On the other hand, if after a crash the server waits a time T before rebooting, all orphans are sure to be gone.

- None of the above methods are desirable.

Communication in Distributed Systems

2.1 Inter-process Communication

- 2.1.1 Message passing
- 2.1.2 Remote Procedure Calls (RPC)
- 2.1.3 Communication between distributed objects
- 2.1.4 Remote Method Invocation (RMI)
- 2.1.5 Events and Notifications

2.2 Group Communication

- 2.2.1 Multicast
- 2.2.2 Publish/subscribe systems
- 2.2.3 Consistency models.