Fundamental Models

Fundamental Models- Basics

- A fundamental model should contain only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system's behaviour.
- Purpose of the model:
  - To make explicit all the relevant assumptions about the system.
  - To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are dependent on logical analysis and, where appropriate, mathematical proof.
- There is much to be gained by knowing what our designs do, and do not, depend upon. It allows us to decide whether a design will work if we try to implement it in a particular system: we need to only ask whether our assumptions hold in that system.

Fundamental Models (cont.)

- The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:
  - Interaction:
    - Computation occurs within processes;
    - the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes.
    - In the analysis and design of distributed systems we are concerned especially with these interactions.
    - The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

Fundamental Models (cont.)

- ***Failure:***
  - The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them.
  - Our model defines and classifies the faults. This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

- ***Security:***
  - The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents.
  - Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

Fundamental Models (cont.)

1. Interaction Model:
   - Deals with communication details among the components and their timing and performance details.
2. Failure Model:
   - Provides specification and effects of faults relating to processes and communication channels.
3. Security Model:
   - Specifies possibilities of attacks to processes and communication channels due to openness and modular nature of distributed systems.
   - Provides a basis for the analysis of threats to a system.

Basics of (1) Interaction Model

- Multiple server processes may cooperate to provide service. For example, DNS, which partitions and replicates its data at servers throughout the Internet, and Sun's Network Information Service, which keeps replicated copies of password files at several servers in a local area network.

- A set of peer processes may cooperate to achieve common goal. For example, Audio conferencing, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

- Their behaviour and state can be defined by distributed algorithm.

Basics of (1) Interaction Model (cont.)

- **Distributed Algorithm**
  - Definition of the steps to be taken by each of the processes of which DS is made of, including the transmission of messages.
  - Rate at which each process proceed and the timing of transmission of messages can not be predicted in general.
  - Each process has its own state.
- The rate at which each process proceeds and the timing of the transmission of messages between them cannot in general be predicted. It is also difficult to describe all the states of a distributed algorithm, because it must deal with the failures of one or more of the processes involved or the failure of message transmissions.
- Significant factors affecting interacting processes in a distributed system:
  - Communication performance is often a limiting characteristic.
  - It is impossible to maintain a single global notion of time.

Performance of communication channels:

- The communication channels in our model are realized in a variety of ways in distributed systems – for example, by an implementation of streams or by simple message passing over a computer network.
- **Delay** between start of a message's transmission from one process and beginning of its receipt by another process. It includes:

  - Time taken for the first of a string of bits transmitted through a network to reach its destination.

  - Delay in accessing the network increases with increase in network load. For example, for Ethernet transmission the sending station waits for the network to be free of traffic.

  - Time taken by operating system communication services at both sending and receiving processes, which varies according to current load on operating system.

Performance of communication channels (cont.)

- The **bandwidth** of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.

- **Jitter** is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.

Computer clocks and timing events:

- Each computer in DS has its own internal clock, which can be used by local processes.
- Processes running on different computers associate timestamps with their events.
- If two processes read their clocks at the same time, their local clocks may supply different time values.
- **Clock drift rate:** The relative amount of time with which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.
- For example, computers may use radio receivers to get time readings from the Global Positioning System with an accuracy of about 1 micro-second.

Inter Process Communication (IPC)

- A process can be of two types:
  - Independent process.
  - Co-operating process.
- An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
- Co-operative nature can be utilized for increasing computational speed, convenience, and modularity.

Inter-process communication (IPC)

- A mechanism that allows processes to communicate with each other and synchronize their actions.
- Processes can communicate with each other through both:
  - Shared Memory
  - Message passing
- An operating system can implement

  both methods of communication.

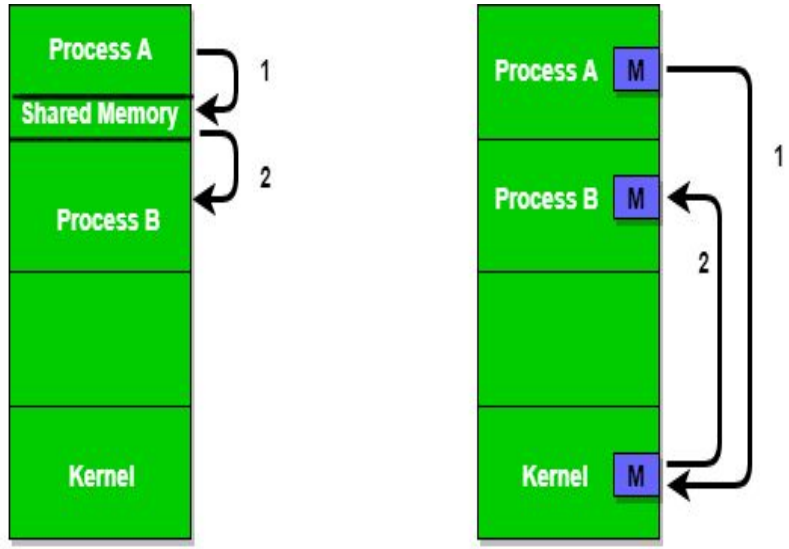


Figure 1 - Shared Memory and Message Passing

IPC: Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer.

The producer produces some items and the Consumer consumes that item.

The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed.

It is a bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it.

IPC: Shared Memory Method

- First, the Producer and the Consumer will share some common memory, then the producer will start producing items.
- If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer.
- Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it.
- If there are items available, Consumer will consume them.

IPC: Shared Memory Method

The pseudo-code to demonstrate shared data between the two Processes:

```
#define buff_max 25
#define mod %
 struct item{
...
  }
int free_index = 0;
int full_index = 0;
```

IPC: Shared Memory Method

- **Producer Process Code**

```
item nextProduced;

   while(1){

      // check if there is no space
      // for production.
      // if so keep waiting.
      while((free_index+1) mod buff_max == full_index);

      shared_buff[free_index] = nextProduced;
      free_index = (free_index + 1) mod buff_max;
   }
```

IPC : Shared Memory Method

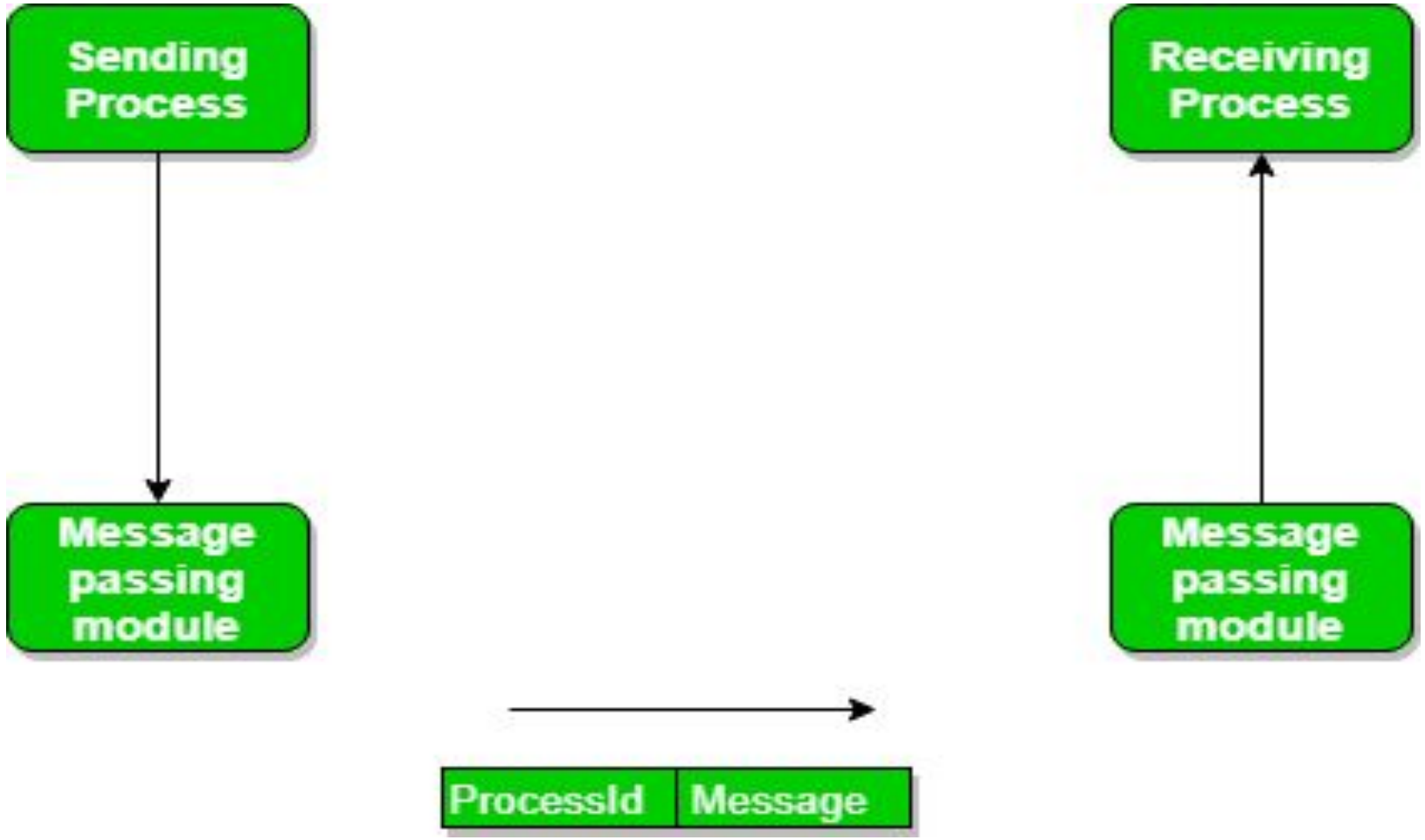
- **Consumer Process Code**

```
item nextConsumed;

  while(1){

     // check if there is an available
     // item  for consumption.
     // if not keep on waiting for
     // get them produced.
     while((free_index == full_index);

     nextConsumed = shared_buff[full_index];
     full_index = (full_index + 1) mod buff_max;
  }
```

IPC: Messaging Passing Method

- In this method, processes communicate with each other without using any kind of shared memory.
- If two processes p1 and p2 want to communicate with each other, they proceed as follows:
  - Establish a communication link (if a link already exists, no need to establish it again.)
  - Start exchanging messages using basic primitives.
    We need at least two primitives:
    – **send**(message, destination) or **send**(message)
    – **receive**(message, host) or **receive**(message)

IPC: Messaging Passing Method



IPC: Messaging Passing Method

- The message size can be of fixed size or of variable size.
- If it is of fixed size, it is easy for an OS designer but complicated for a programmer
- If it is of variable size then it is easy for a programmer but complicated for the OS designer.
- A standard message can have two parts: **header and body.**
  - The **header part** is used for storing message type, destination id, source id, message length, and control information.
  - The control information contains information like what to do if runs out of buffer space, sequence number, priority.
- Generally, message is sent using FIFO style.

IPC: Messaging Passing Method

Message Passing through Communication Link:

While implementing the link, there are some questions that need to be kept in mind like :

- How are links established?
- Can a link be associated with more than two 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?

IPC: Messaging Passing Method

Message Passing through Exchanging the Messages:

- **Synchronous and Asynchronous Message Passing:**

IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system.

In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking.

Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available.

Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null.

However, the sender expects acknowledgment from the receiver in case the send fails.

IPC: Messaging Passing Method

- There are basically three preferred combinations:
  - Blocking send and blocking receive
  - Non-blocking send and Non-blocking receive
  - Non-blocking send and Blocking receive (Mostly used)

IPC: Messaging Passing Method

- **Direct message passing**:The process which wants to communicate must explicitly name the recipient or sender of the communication.
  e.g. **send(p1, message)** means send the message to p1.
  Similarly, **receive(p2, message)** means to receive the message from p2.
  - In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links.
- **Indirect message passing**, processes use mailboxes (also referred to as ports) for sending and receiving messages.
  - Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes.
  - Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox.

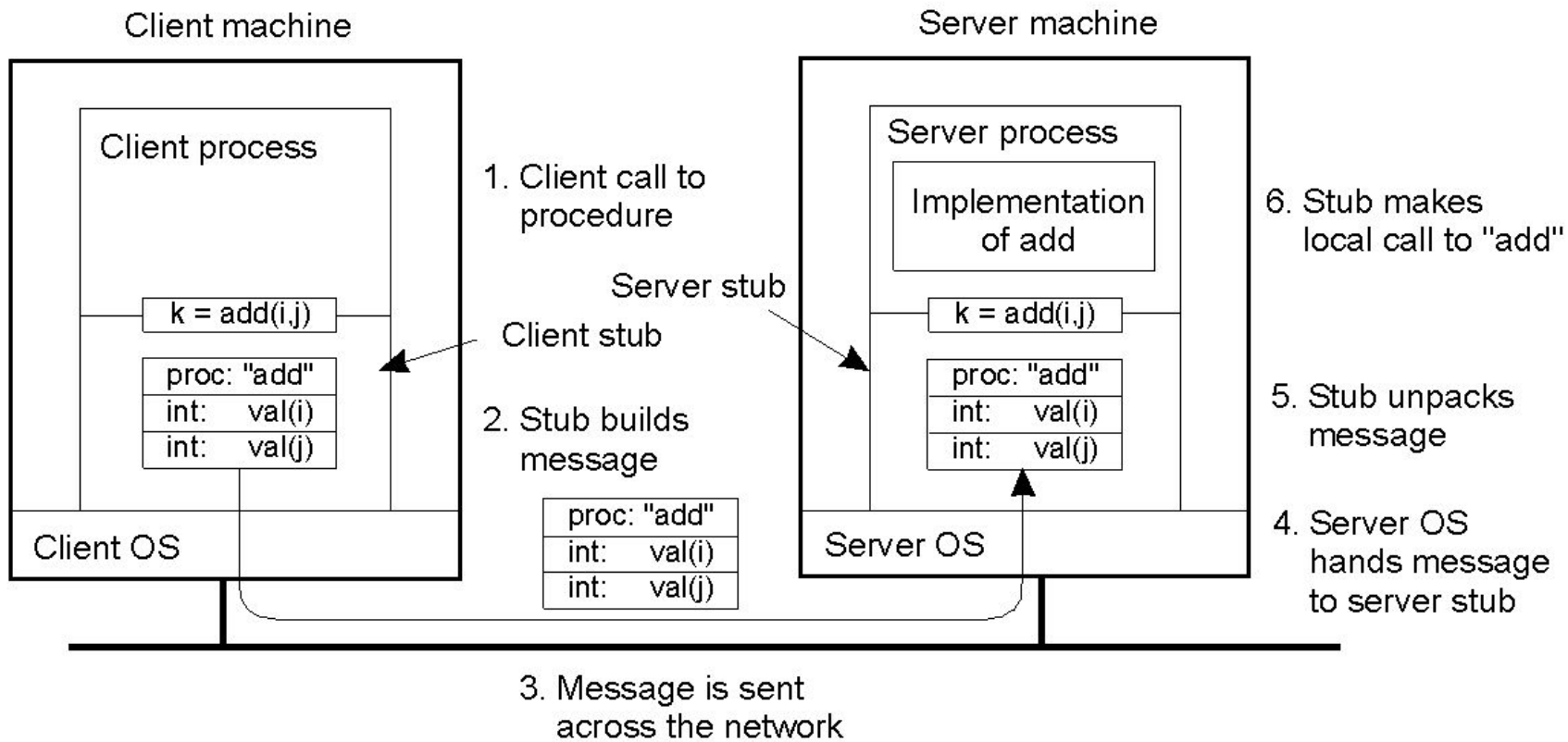
IPC: Messaging Passing Method

- Producer-Consumer problem using the message passing concept: The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox.

```
void Producer(void){

    int item;
    Message m;

    while(1){

      receive(Consumer, &m);
      item = produce();
      build_message(&m , item ) ;
      send(Consumer, &m);
    }   }
```

```
void Consumer(void){

    int item;
    Message m;

    while(1){

      receive(Producer, &m);
      item = extracted_item();
      send(Producer, &m);
      consume_item(item);
    }
  }
```

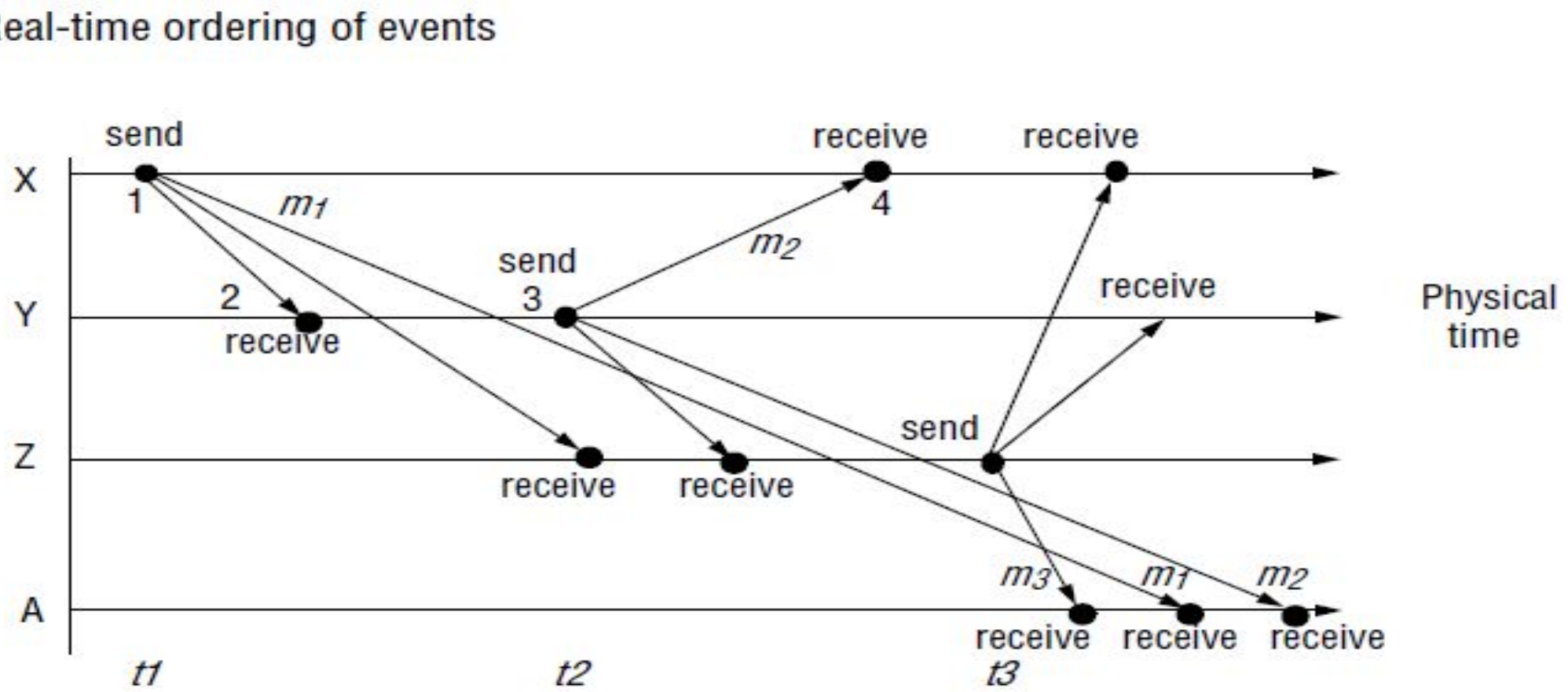
Steps involved in doing remote computation through RPC



Event Ordering

- In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.
- For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:
  - User X sends a message with the subject Meeting.
  - Users Y and Z reply by sending a message with the subject Re: Meeting.

Event Ordering (cont.)



Event Ordering (cont.)

- In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the independent delays in message delivery, the messages may be delivered as shown in Figure. Some users may view these two messages in the wrong order. For example, user A might see:

		Inbox:
Item	*From*	*Subject*
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

Event Ordering (cont.)

- If the clocks on X's, Y's and Z's computers could be synchronized, then each message could carry the time on the local computer's clock when it was sent.
- For example, messages *m1*, *m2* and *m3* would carry times *t1*, *t2* and *t3* where $t1<t2<t3$. The messages received will be displayed to users according to their time ordering. If the clocks are roughly synchronized, then these timestamps will often be in the correct order.

Event Ordering (cont.)

- Since clocks cannot be synchronized perfectly across a distributed system, Lamport [1978] proposed a model of *logical time* that can be used to provide an ordering among the events at processes running in different computers in a distributed system.

- Logical time allows the order in which the messages are presented to be inferred without recourse to clocks.

- Let us understand how some aspects of logical ordering can be applied to our email ordering problem.

Event Ordering (cont.)

- Logically, we know that a message is received after it was sent. Therefore we can state a logical ordering for pairs of events shown in Figure, for example, considering only the events concerning X and Y:
  - X sends *m1* before Y receives *m1*; Y sends *m2* before X receives *m2*.

- We also know that replies are sent after receiving messages, so we have the following logical ordering for Y:
  - Y receives *m1* before sending *m2*.

- Logical time takes this idea further by assigning a number to each event corresponding to its logical ordering, so that later events have higher numbers than earlier ones. For example, Figure shows the numbers 1 to 4 on the events at X and Y.

Failure Model

- In a distributed system both processes and communication channels may fail.
- Fail means they may depart from what is considered to be correct or desirable behaviour.
- The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures.
- Distinguishes between the failures of processes and communication channels.
- Failure can be any one of these:
  - **omission failures**: a special class of performance failures where results are either incorrect or infinitely late
  - **arbitrary failures**: In a process, arbitrary behaviour may include setting incorrect data values, returning a value of incorrect type, stopping or taking incorrect steps.
  - **timing failures**: upper and lower bounds on the time to execute each step of a process; a bound on the transmission time of each message over a channel; a bound on the drift rate of the local clock of each process.

Failure Model: Omission Failure

- when a process or communication channel fails to perform actions that it is supposed to do.
- Process omission failures:
  - A process makes an omission failure when it crashes — it is assumed that a crashed process will make no further progress on its program. A crash is considered to be clean if the process either functions correctly or has halted. A crash is termed a fail-stop if other processes can detect with certainty that the process has crashed.
    - a process has crashed mean that it has halted and will not execute any further steps of its program ever.
    - Other processes may be able to detect such a crash by the fact that the process repeatedly fails to respond to invocation messages.
    - This method of crash detection relies on the use of ***timeouts*** – *that is, a method in which one process allows a fixed period of time for* something to occur.
    - In an asynchronous system a timeout can indicate only that a process is not responding – *it may have crashed or may be slow, or the messages may not have arrived.*

Failure Model: Omission Failure

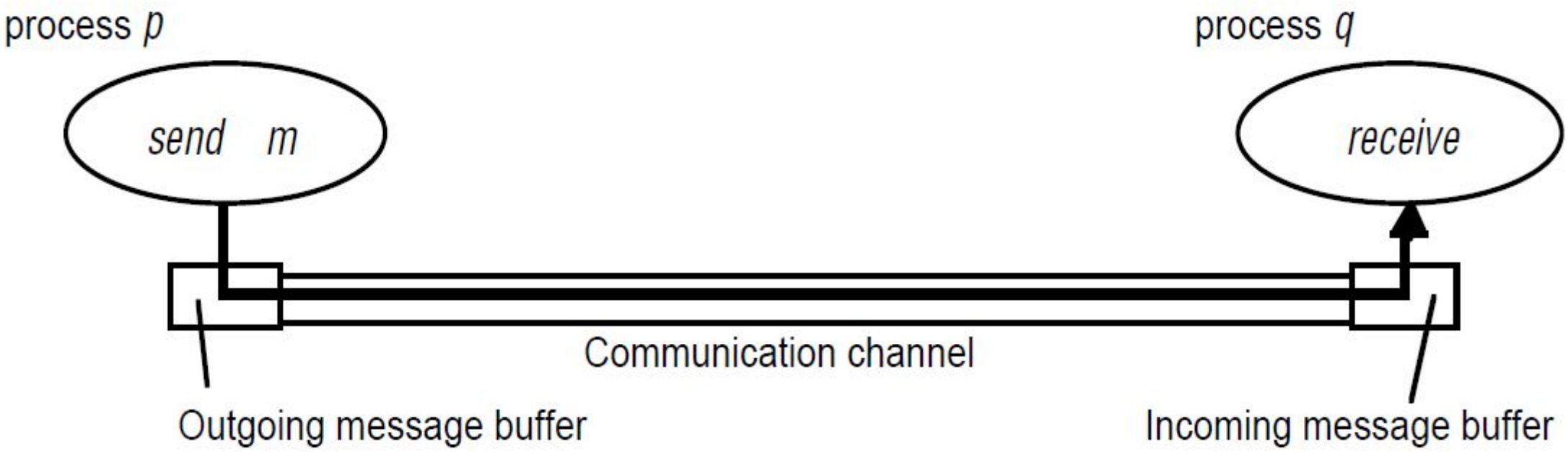
- A process crash is called ***fail-stop*** *if other processes can detect certainly that the* process has crashed.
- Fail-stop behaviour can be produced in a synchronous system if the processes use timeouts to detect when other processes fail to respond and messages are guaranteed to be delivered.
- For example, if processes *p and q are programmed for q to* reply to a message from *p, and if process p has received no reply from process q in a* maximum time measured on *p's local clock, then process p may conclude that process q has failed.*
- *Difficulty arises in detecting failures in an* asynchronous system.

Communication omission failures

- Communication omission failures may occur in the sending process (send-omission failures), the receiving process (receive omission failures) or the channel (channel omission failures).

- The communication primitives are *send and receive.*

Communication omission failures



- *A process **p** performs a send by inserting the message m in its outgoing message* buffer.
- The communication channel transports *m to q's incoming message buffer.*
- Process ***q*** *performs a receive by taking m from its incoming message buffer and* delivering it.
- The outgoing and incoming message buffers are typically provided by the operating system.

Communication omission failures

- **Omission failure** if it does not transport a message from *p's outgoing message buffer to q's incoming message buffer.*
- *This is* known as 'dropping messages' and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network transmission error.
  - loss of messages between the sending process and the outgoing message buffer as ***send omission failures***.
  - *loss of messages between the incoming message buffer and the* receiving process as ***receive-omission failures***.
  - *loss of messages in between as* ***channel omission failures***.

Arbitrary failures

- **The term *arbitrary or Byzantine failure is used to describe the worst*** possible failure semantics, in which any type of error may occur.
- For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.
- Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.
- Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once.
- ***Arbitrary failures of communication channels are rare***

Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send,* but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing failures

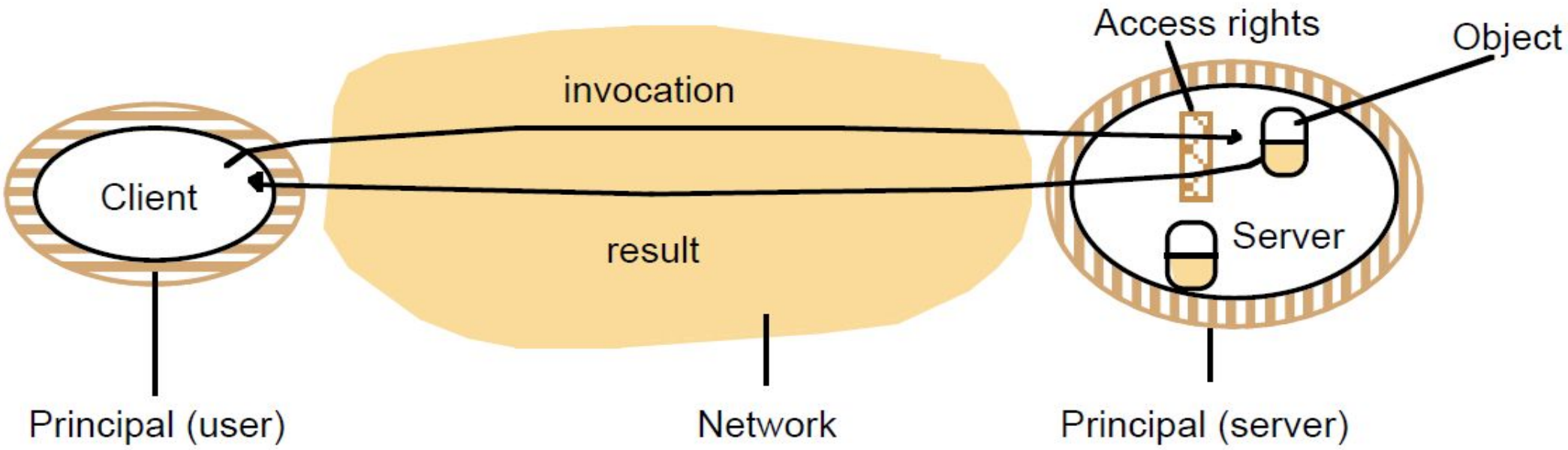
Applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Timing is particularly relevant to multimedia computers with audio and video channels. Video information can require a very large amount of data to be transferred.
Delivering such information without timing failures can make very special demands on both the operating system and the communication system.

Security model

- The security of a distributed system can be achieved by securing the **processes** and the **channels** used for their interactions and by protecting the objects that they encapsulate against unauthorized access.



Security model: Objects and principals

- A server manages a collection of objects on behalf of some users.
- The users can run client programs that send invocations to the server to perform operations on the objects.
- The server carries out the operation specified in each invocation and sends the result to the client.
- Objects are intended to be used in different ways by different users.
- For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages.
- To support this, *access rights specify who is* allowed to perform the operations of an object – for example, who is allowed to read or to write its state.

Security model: Objects and principals

- The invocation comes from a user and the result from a server.
- Access rights are associated with each invocation and each result the authority on which it is issued.
- **Such an authority is called a *principal. A principal may be a user or a process.***
- The server is responsible for verifying the identity of the principal behind each invocation and checking that they have sufficient access rights to perform the requested operation on the particular object invoked, rejecting those that do not.
- The client may check the identity of the principal behind the server to ensure that the result comes from the required server.

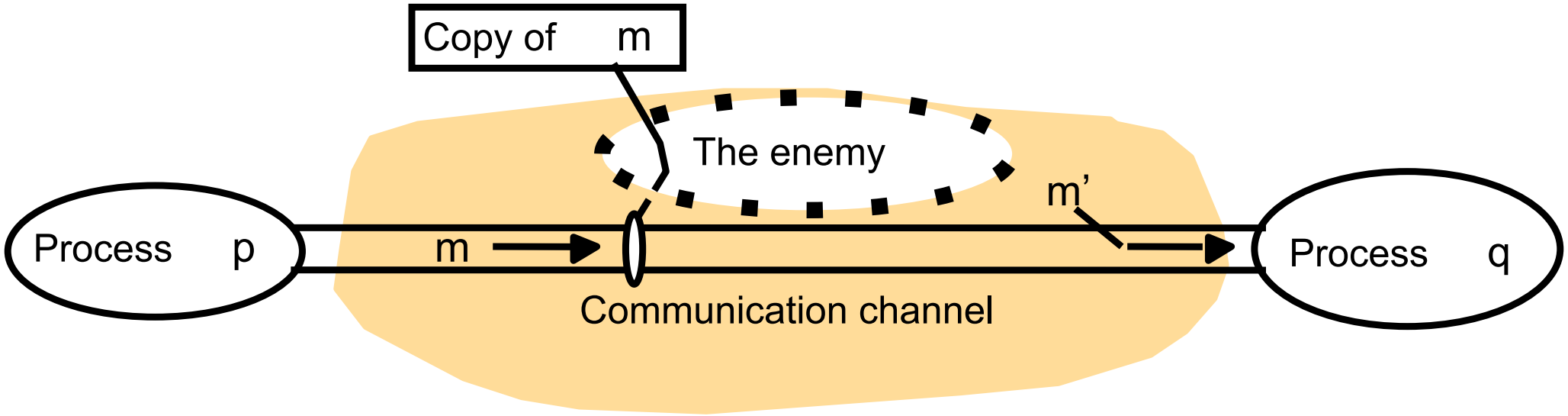
Securing processes and their interactions

- Processes interact by sending messages.
- The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact.
- Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.
- Secrecy or integrity is crucial for the applications that handle financial transactions, confidential information.
- Integrity is threatened by security violations as well as communication failures.
- Threats occur to the processes which handle such applications.
- Threats also occur to the messages travelling between such+66 processes.

Analysis of security threats

- An enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes.
- Such attacks can be made simply by using a computer connected to a network to run a program
  - that reads network messages addressed to other computers on the network,
  - or a program that generates messages that make false requests to services, purporting to come from authorized users.

The enemy



The threats from a potential enemy include

threats to processes

threats to communication channels

Threats to processes

- A process that is designed to handle incoming requests may receive a message from any other process in the distributed system, and it cannot necessarily determine the identity of the sender.
  - Communication protocols such as IP do include the address of the source computer in each message, but it is not difficult for an enemy to generate a message with a forged source address.
  - This lack of reliable knowledge of the source of a message is a threat to the correct functioning of both servers and clients.

Threats to processes: Server

- *Servers: Since a server can receive invocations from many different clients, it cannot* necessarily determine the identity of the principal behind any particular invocation.
- Even if a server requires the inclusion of the principal's identity in each invocation, an enemy might generate an invocation with a false identity.
- Without reliable knowledge of the sender's identity, a server cannot tell whether to perform the operation or to reject it.
- For example, a mail server would not know whether the user behind an invocation that requests a mail item from a particular mailbox is allowedto do so or whether it was a request from an enemy.