Distributed Systems

COMP90015 2019 SM1

Models

Lectures by Aaron Harwood

Models Overview

- Introduction
- Architectural models
- Fundamental models
- Summary

Introduction

A *pyhsical model* considers: - underlying hardware elements

An *architectural model* considers:

- *Architectural elements* - components of the system that interact with one another
- *Architectural patterns* - the way components are mapped to the underlying system
- *Associated middleware solutions* - existing solutions to common problems

Several well accepted architectural models are available for distributed systems (e.g client-server, peer-to-peer).

Fundamental models define:

- The non-functional aspects of the distributed system such as
  - reliability
  - security
  - performance

Communicating Entities

From a system perspective:

- processes
- threads
- nodes, e.g. sensors

From a programming perspective:

- Objects, distributed object system
- Components, like objects but with dependencies made more explicit
- Objects and components are usually used within an organization, while web services are usually seen as providing public interfaces

Interfaces

Distributed processes can not directly access each others internal variables or procedures. Passing parameters needs to be reconsidered, in particular, call by reference is not supported as address spaces are not the same between distributed processes. This leads to the notion of an *interface*. The set of functions that can be invoked by external processes is specified by one or more *interface definitions*. Defining interfaces is an important part of designing objects, components and web services.

- Programmers are only concerned with the abstraction offered by the interface, they are not aware of the implementation details.
- Programmers also need not know the programming language or underlying platform used to implement the service.
- So long as the interface does not change (or that changes are backwards compatible), the service implementation can change transparently.

Communication paradigms

From low level to high level:

- Interprocess communication are the underlying primitives -- relatively low-level (perhaps the lowest level) of support for communication between processes in a distributed system, e.g. shared memory, sockets, multicast communication
- Remote invocation -- based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method, e.g. request-reply protocols, remote procedure calls, remote method invocation
- Indirect communication
  - *space uncoupling* -- senders do not need to know who they are sending to
  - *time uncoupling* -- senders and receivers do not need to exist at the same time
  - for example: group communication, publish-subscribe systems, message queues, tuple spaces, distributed shared memory

Roles and responsibilities

- Client, a process that initiates connections to some other process
- Server, a process that can receive connections from some other process
- Peer, can be seen as taking both the role of client and server, connecting to and receiving connections from other peers

E.g. we say that a process is acting as a client to another process which is acting as a server, to mean that the first process will be the one responsible for initiating the connection.

Placement

- mapping services to multiple servers
  - a single service may not make use of a single process and multiple processes may be distributed across multiple machines
- caching
  - storing data at places that are typically closer to the client or whereby subsequent accesses to the same data will take less time
- mobile code
  - transferring the code to the location that is most efficient, e.g. running a complex query on the same machine that stores the data, rather than pulling all data to the machine that initiated the query
- mobile agents
  - code and data together
  - e.g. used to install and maintain software on a users computer, the agent continues to check for updates in the background

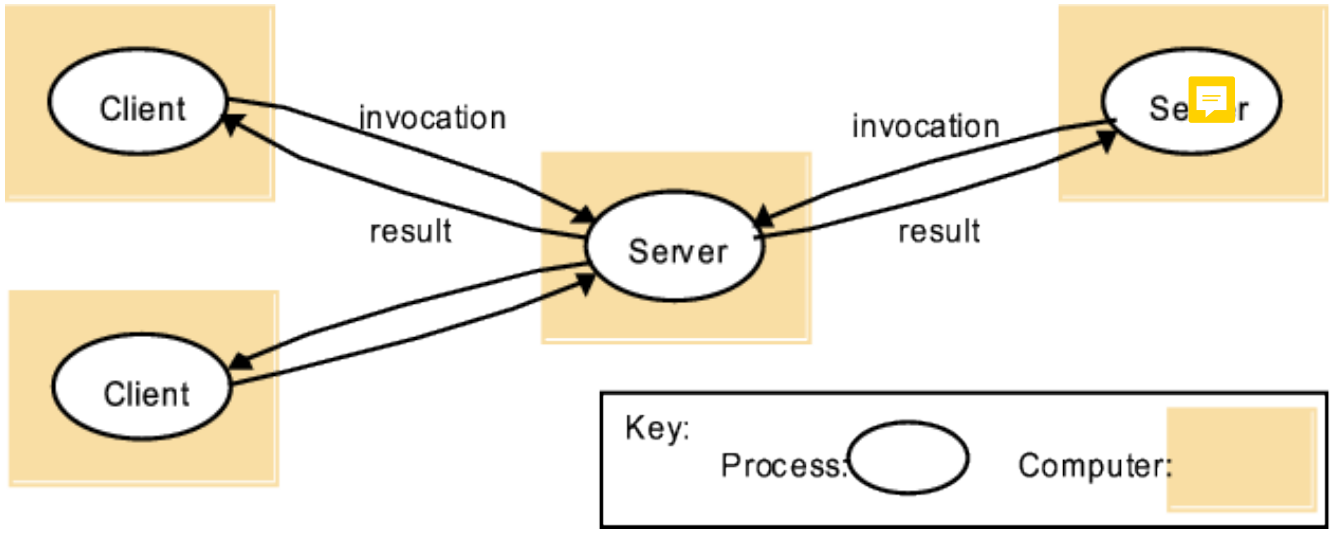
Architectural Patterns

The two widely used distributed architectures are:

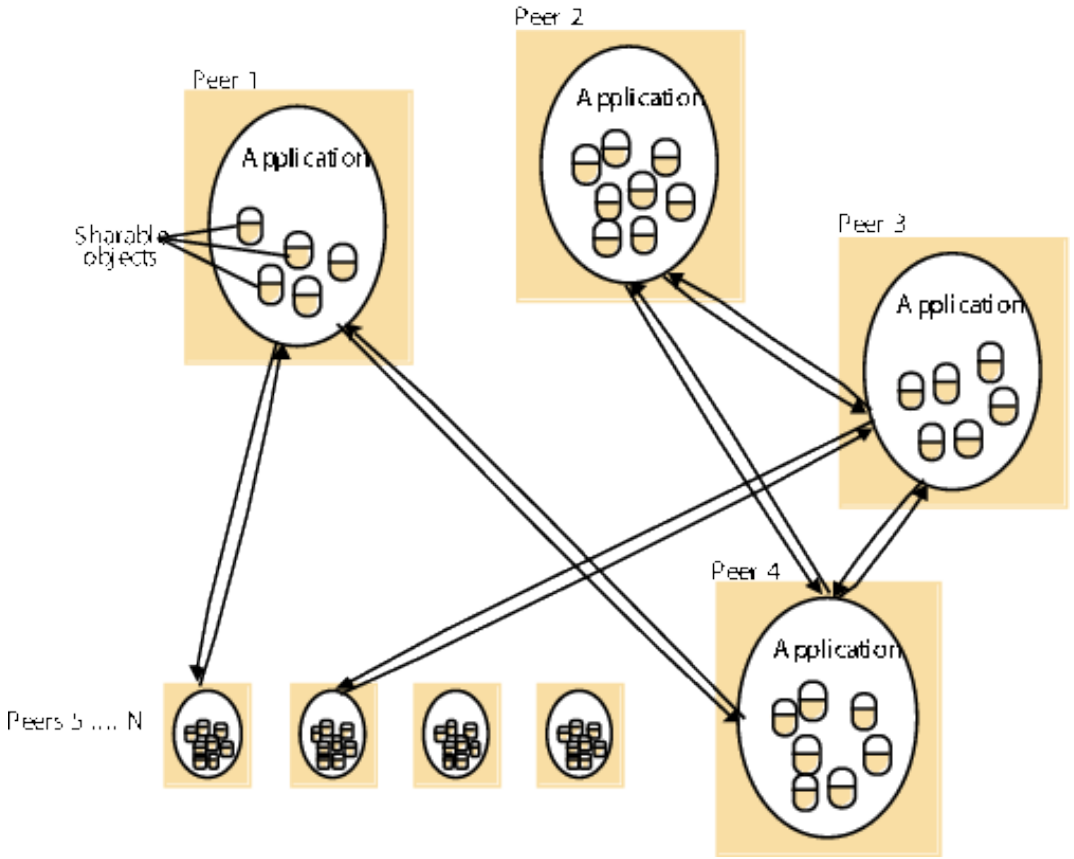
- **Client-server**: Clients invoke services in servers and results are returned. Servers in turn can become clients to other services.
- **Peer-to-peer**: Each process in the systems plays a similar role interacting cooperatively as peers (playing the roles of client and server simultaneously).

Many variations of these architectures are present in today's distributed systems.

Client-server



Peer-to-Peer

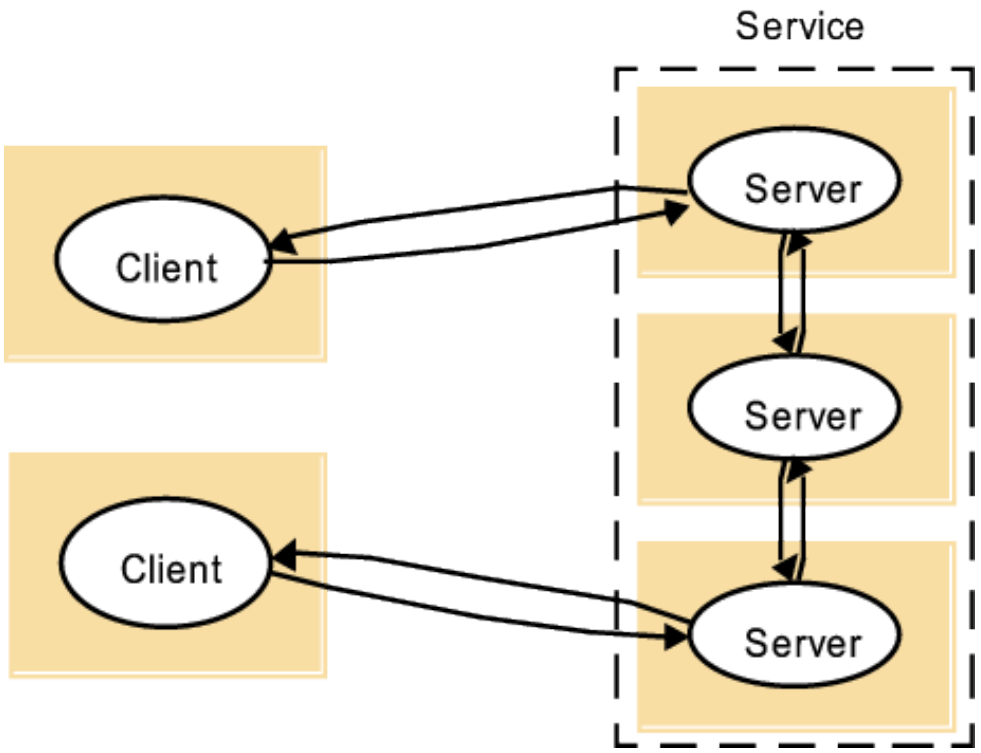

Peer 1
Application
Sharable objects
Peer 2
Application
Peer 3
Application
Peer 4
Application
Peers 5 .... N

Distributed System Architecture Variations

- Services provided by multiple servers
- Proxy servers and caches
- Mobile code
- Mobile agents
- Network computers
- Thin clients
- Mobile devices and spontaneous inter-operation

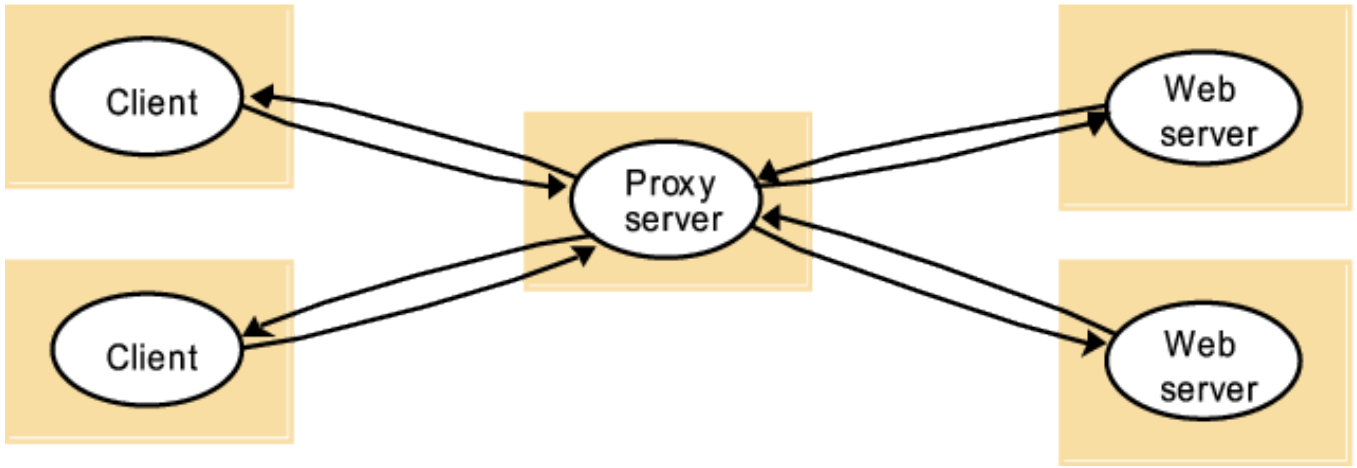
A service provided by multiple servers

Service is provided by several server processes interacting with each other. Objects may be partitioned (e.g web servers) or replicated across servers (e.g. Sun Network Information Service (NIS)).


Service
Server
Client
Server
Client
Server

Proxy servers and caches

- Cache is a store of recently used objects that is closer to client
- New objects are added to the cache replacing existing objects
- When an object is requested, the caching service is checked to see if an up-to-date copy is available (fetched in not available)



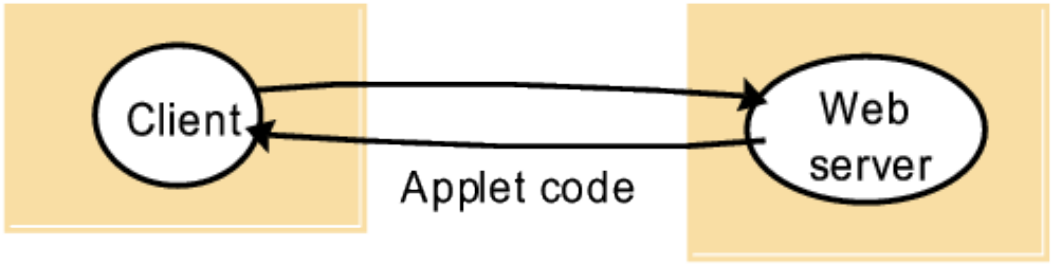
Mobile Code and Agents

Mobile Code is down loaded to the client and is executed on the client (e.g. applet).

Mobile agents are running programs that includes both code and data that travels from one computer to another.

E.g. Web Applets:

a) client request results in the downloading of applet code

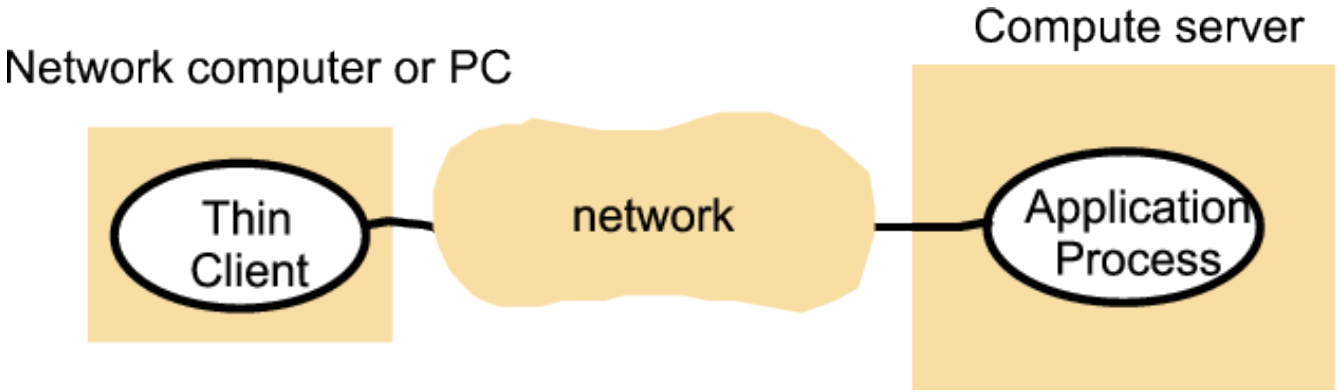


b) client interacts with the applet

Network Computers and Thin clients

- **Network Computers**: download their operating system and application software from a remote file system. Applications are run locally.
- **Thin Clients**: application software is not downloaded but runs on the computer server - e.g. UNIX.

This paradigm is usually not suitable for highly interactive graphical activities.



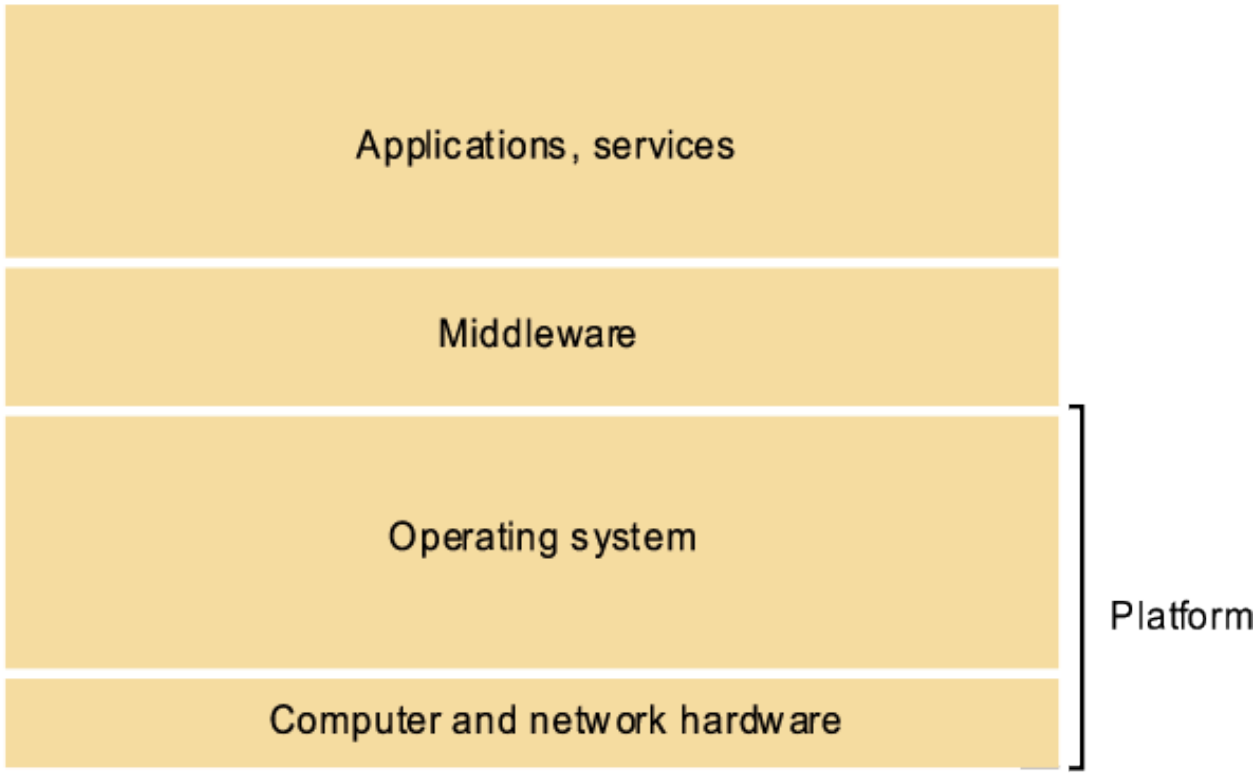
Layering

A software architecture abstracts software into layers or modules in a single computer. Each layer of software provides a service to the next layer. The layers can are referred to as **service layers**.

Two important layers for a distributed system are:

- **Platform**
- **Middleware**

Abstract software layers

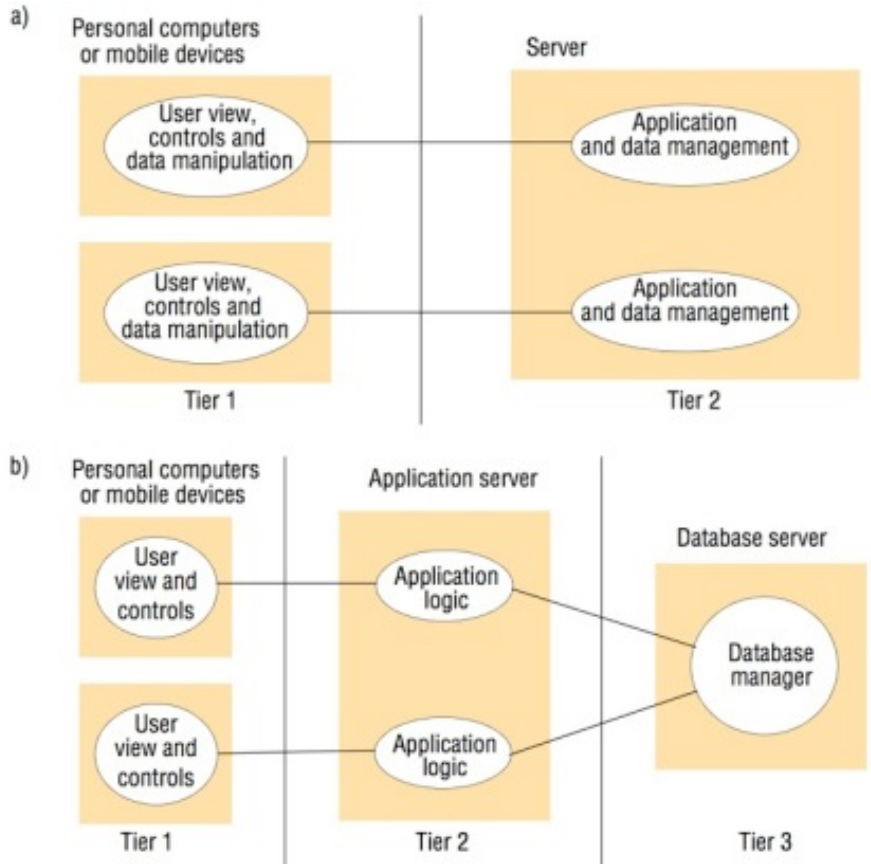


Middleware

- Provides value added services - e.g.
  - Naming
  - security
  - transactions
  - persistent storage and
  - event service
- Adds overhead due to the additional level of abstraction
- Communication cannot be completely hidden from applications since appropriate error handling is important

Tiered architecture

Tiered architectures are complementary to layering. Layering deals with vertical organization of services.



Fundamental Models

Fundamental models allow distributed systems to be analyzed in terms of fundamental properties regardless of the architecture. These models help understand how the non-functional requirements are supported.

The aspects of distributed systems that will be captured in the fundamental models are:

- Interaction
- Failure
- Security

Interaction Model

- Models the interaction between processes of a distributed system - e.g. interaction between clients and servers or peers.
- Distributed algorithms specify:
  - Steps taken by each process in the distributed system
  - The transmission of messages between processes
- Two important aspects of interaction modeling are:
  - Performance of communication channels
  - Computer clocks and timing events

Performance of communication channels

Three important performance characteristics of communication channels:

- **Latency**
- **Bandwidth**
- **Jitter**

Event timing

Each computer in a distributed system has its own internal clock. The timestamps between two processes can vary due to:

- Initial time setting being different
- Differences in clock drift rates

GPS can be used synchronize clocks (accuracy of 1 micro second) but they do not operate inside buildings and are too costly therefore special clock synchronization techniques have to be used.

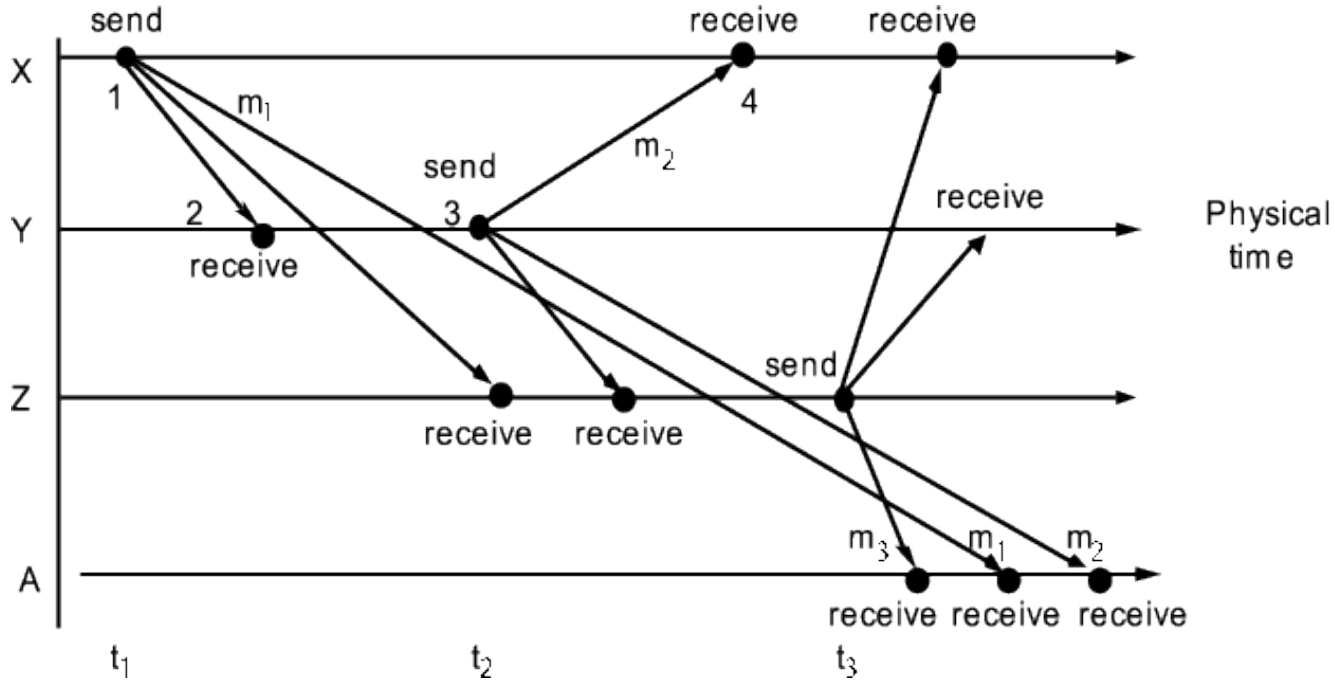
Variations of Interaction Models

Two simple models of distributed system interaction are:

- Synchronous system model - assumes known bounds on:
  - the time to execute each step of a process
  - message transmission delay
  - local clock drift rate
- Asynchronous system model - assumes no bound on:
  - process execution speed
  - message transmission delays
  - clock drift rates

Event Ordering

Event ordering is important in some scenarios. However, this is challenging without a global clock for all processes. Shows the email example where the messages are not received in the logical order.



The concept of *logical time* can be used for ordering events. For example an event number can be generated for a sequence of events, and the events can be interpreted based on this sequence number as opposed to the time of receipt.

Failure Model

The failures in processes and channels are presented using the following taxonomy:

- Omission failures
- Arbitrary failures
- Timing failures

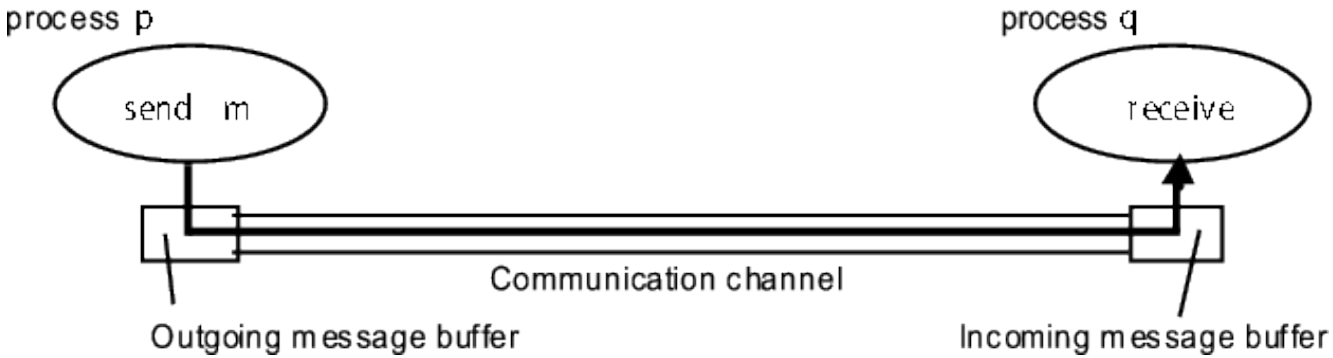
Omissions Failures

Omission failures refers to cases where a process or a communication channel fails to perform what is expected to do.

- Process omission failures:
  - Normally caused by a process crash
  - Repeated failures during invocation is an indication
  - Timeouts can be used to detect this type of crash
  - A crash is referred to as a *fail-stop* if other processes can detect certainly that the process crashed

For communication omission failures consider communication as involving the following steps:

- Process **p** inserting the message **m** to its outgoing buffer (*send*)
- The communication channel transporting **m** from **p**'s outgoing buffer to **q**'s incoming buffer
- Process **q** taking the message from its incoming buffer (*receive*)



Communication omission failure could be due to:

- *Send omission failure*: A message not being transported from sending process to its outgoing buffer
- *Receive omission failure*: A message not being transported from the receiving process's incoming message buffer and the receiving process
- *Channel omission failures*: A message not being transported from **p**'s outgoing message buffer to **q**'s incoming message buffer

Arbitrary Failures (Byzantine failure)

Refers to any type of failure that can occur in a system. Could be due to:

- Intended steps omitted in processing
- Message contents corrupted
- Non-existent messages delivered
- Real messages delivered more than once

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

These failures occur when time limits set on process execution time, message delivery time and clock rate drift. They are particularly relevant to synchronous systems and less relevant to asynchronous systems since the later usually places no or less strict bounds on timing.

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.

Reliability of one-to-one communication

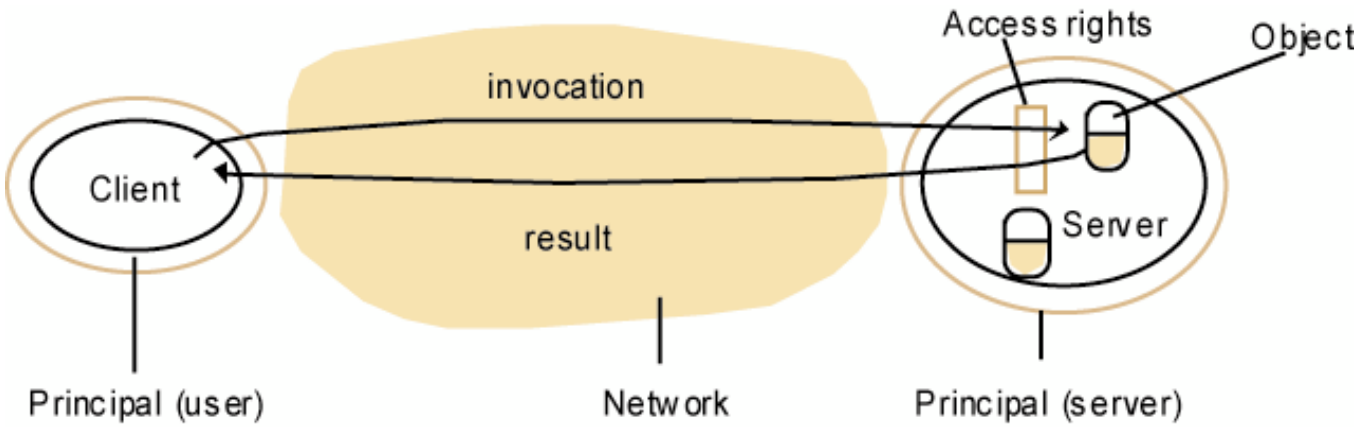
Reliable communication can be defined in terms of two properties:

- **validity**: any message in the outgoing buffer is eventually delivered to the incoming message buffer.
- **integrity**: the message is identical to the one sent, and no messages are delivered twice.

Security Model

Security of a distributed systems is achieved by securing processes, communication channels and protecting objects they encapsulate against unauthorized access.

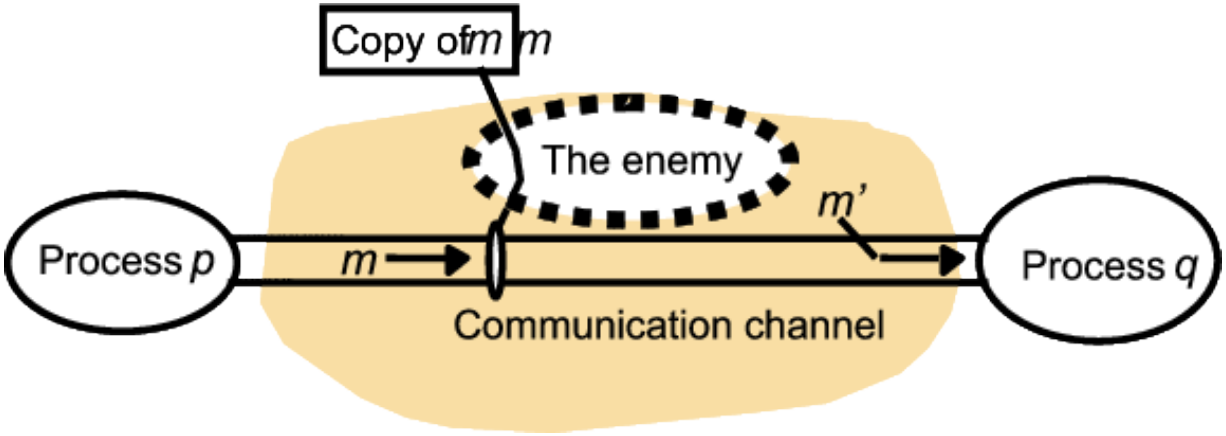
Protecting Objects:



- *Access rights* specify who is allowed to perform operations on an object
- Each invocation and result is associated with a *principal*

Securing Processes and Interactions:

Enemy (adversary) is one capable of sending any message to any process or reading/ copying any message between a pair of processes.



Possible threats from an enemy

- Threats to processes: Servers and clients cannot be sure about the source of the message. Source address can be spoofed.
- Threats to communication channels: Enemy can copy, alter or inject messages
- Denial of service attacks: overloading the server or otherwise triggering excessive delays to the service
- Mobile code: performs operations that corrupt the server or service in an arbitrary way

Addressing security threats

- Cryptography and shared secrets: encryption is the process of scrambling messages
- Authentication: providing identities of users
- Secure Channel: Encryption and authentication are used to build secure channels as a service layer on top of an existing communication channel. A secure channel is a communication channel connecting a pair of processes on behalf of its principles