Sistemas Distribuídos

System Models

António Rui Borges

Summary

- *Models*
- *Types of models*
- *Architectural models*
  - *Client-server*
  - *Peer communication*
  - *Publisher-subscriber*
- *Communication primitives*
- *Fundamental models*
  - *Interaction*
  - *Failure*
  - *Security*
- *Suggested reading*

Models

Distributed systems are aimed to operate in the real world, which means that they must be designed to work properly even when subjected to a wide range of operating environments and/or facing difficult scenarios and predictable menaces.

One uses *description models* to enumerate the common properties and the design assumptions characteristic of a particular class of systems.

Thus, the goal of a model is to provide a simplified and abstract, but consistent, description of relevant features of the system under discussion.

Types of models

- *architectural models* – they define the way how the different components of the system are mapped into the nodes of the underlying parallel platform and how they interact among themselves
  - *mapping wise*: they aim to establish efficient patterns of data distribution and processing loads
  - *interaction wise*: they describe the functional role assigned to each component and their communication patterns
- *fundamental models* – they discuss the systemic characteristics affecting dependability
  - *interaction type*: they deal with issues such as communication bandwidth and latency
  - *failure handling*: they specify the kind of failures that may occur in the intervening processes and communication channels
  - *security*: they discuss possible menaces which are posed to the distributed system and which affect its performance

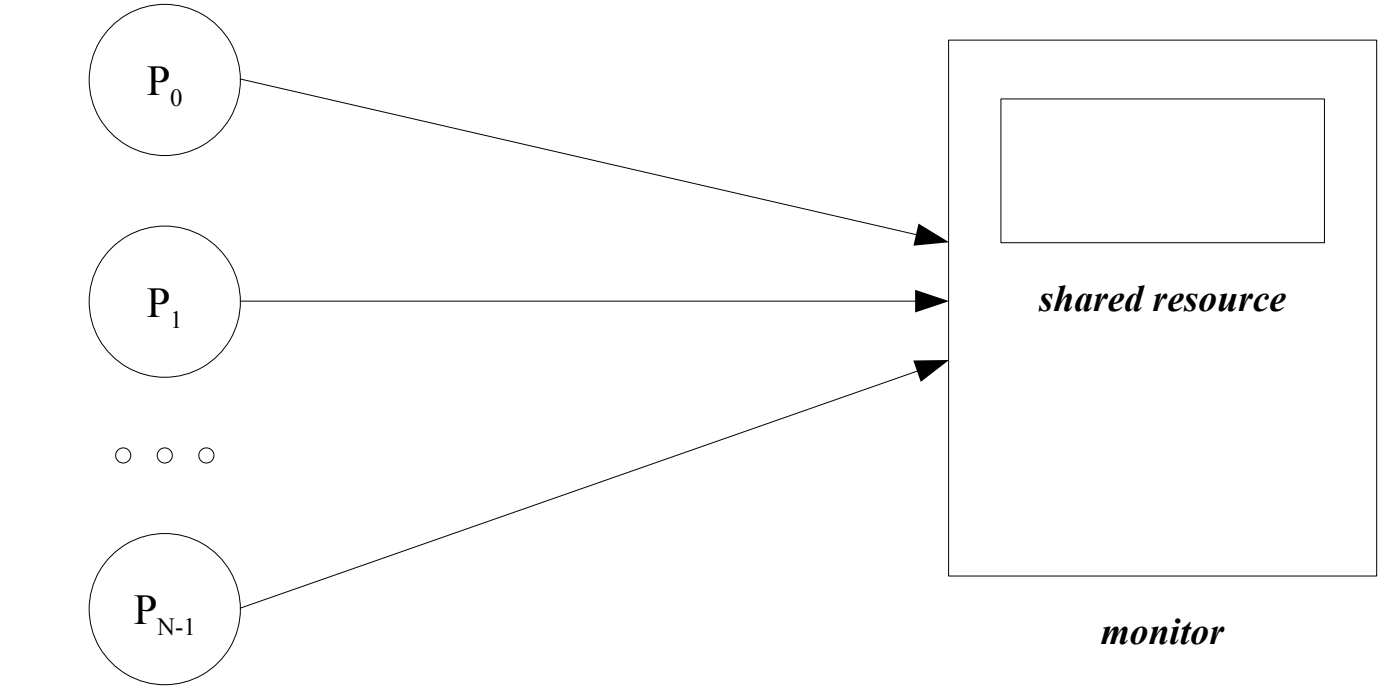
Architectural models

In a *distributed system*, the intervening processes cooperate among themselves in the undertaking of a common task. Responsibility division and mapping on specific processing nodes of the underlying parallel platform are among the most distinctive project issues.

A working approach consists in considering first a concurrent solution to the problem (valid in a monoprocessor computer system) and perform next a set of transformations leading to the migration of the solution to a computer network.

In designing the concurrent solution, any of the communication paradigms may be used, *shared variables* or *message passing*, as they are equivalent. The former, however, leads to more intuitive implementations.

Concurrent solution - 1



Concurrent solution - 2

- the shared resource belongs to the addressing space of all the intervening processes
- a *monitor* protects the access to the resource enforcing mutual exclusion
- process synchronization is done inside the monitor through *condition variables*
- the interaction model is *reactive*: each process runs until blocking or termination
- interaction itself is based on calling operations upon the monitor
- alternatively, if the programming language does not have a concurrency semantics, the shared resource may be protected by an *access* semaphore (or some similar device) and synchronization be carried out by semaphores as well, located now outside the critical region to prevent deadlock

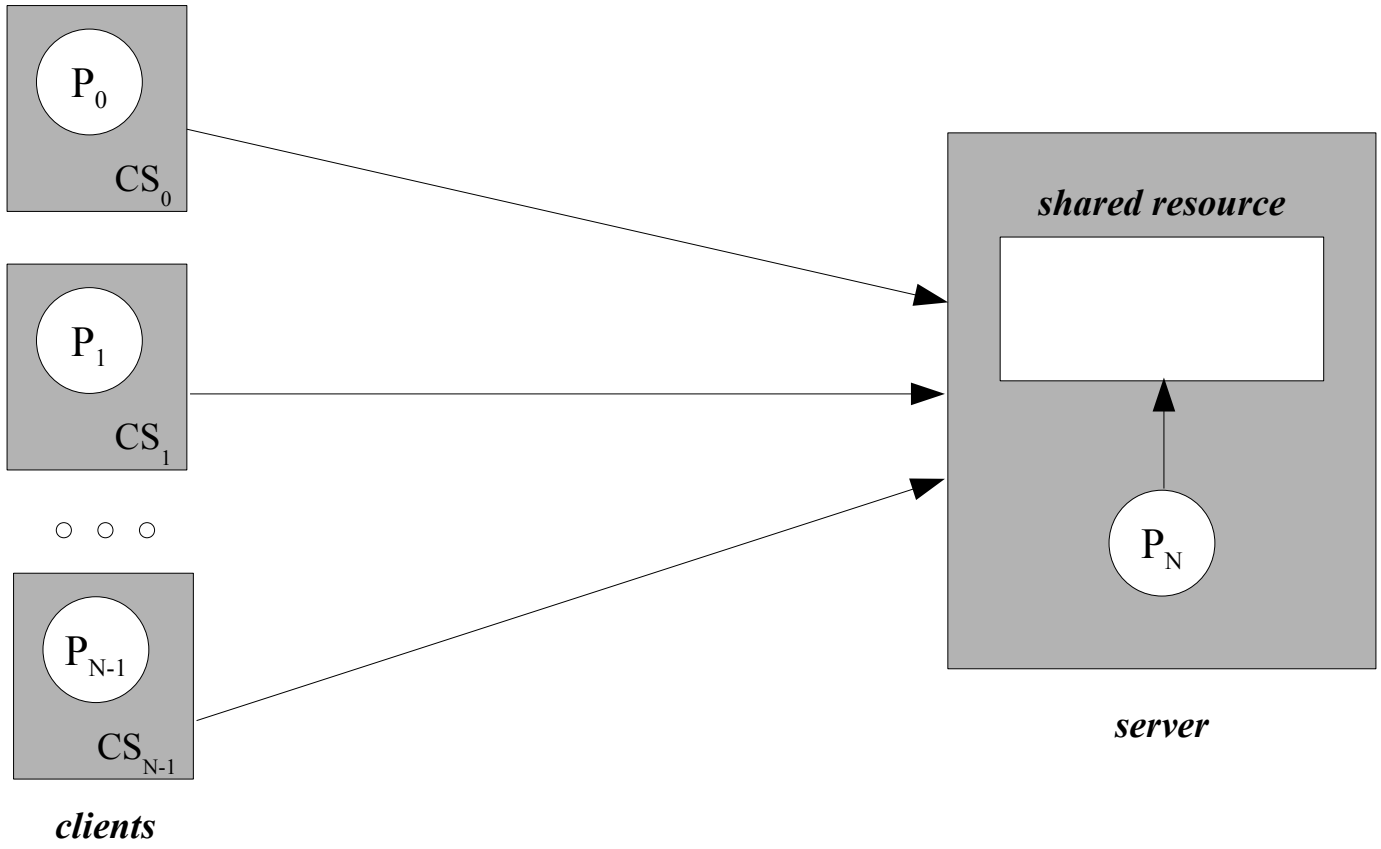
Client-server model – 1

On migrating processes and the shared resource to different computer systems, their addressing spaces become disjoint and communication must take place, in a implicit or explicit way, by message passing on a common communication channel.

A critical issue to consider is that, being the shared resource a passive entity, the access management to it requires the creation of a new process, whose duty is not only communicating with the other processes, but also executing locally the operations called on the shared resource.

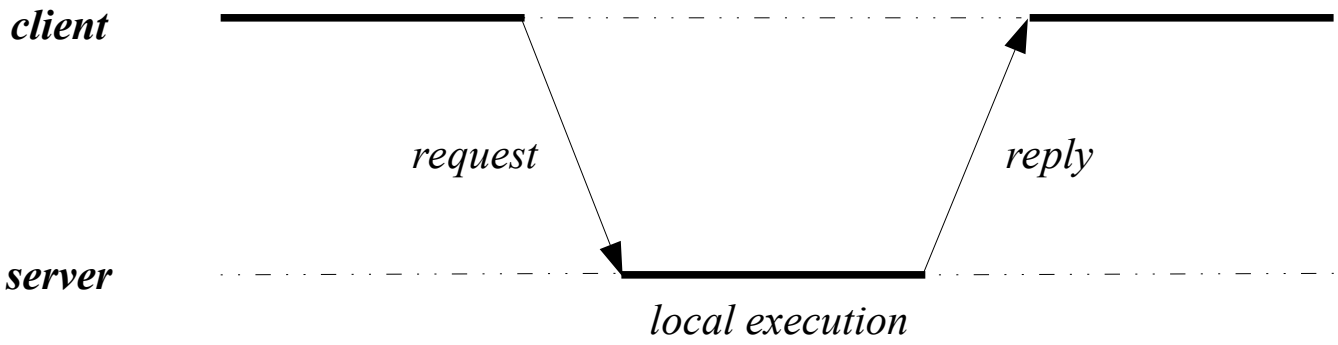
Therefore, an operational model where resource manipulation can be understood as *service rendering* becomes apparent. The process managing locally the shared resource is seen as the service render, usually called *server*. The remaining processes, which want to access the resource, are seen as the entities that request the service, called *clients*.

Client-server model – 2



Client-server model – 3

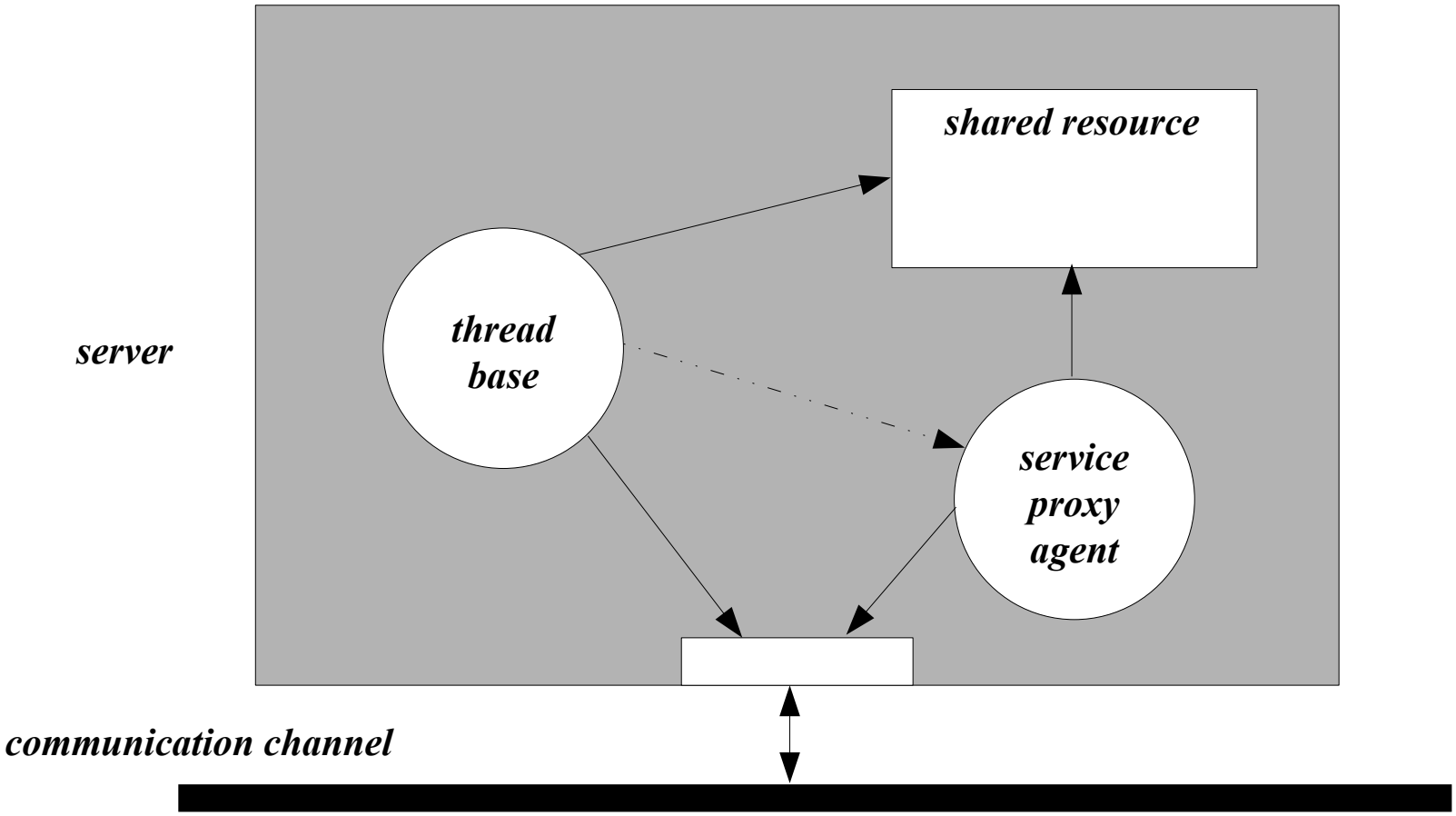
- operations called on the shared resource may be divided in
  - *request*: *client* process calls *server* process asking it to execute an operation on its behalf
  - *local execution*: execution by *server* process of the requested operation upon the shared resource
  - *reply*: *server* process answers to *client* process communicating the result of the operation



Client-server model – 4

- the *server* process has typically two roles, acting as a
  - *communication manager*: waits for service requests by *client* processes
  - *service proxy agent*: executes the operations on the shared resource as a proxy of the *client* processes
- the communication model is asymmetric
  - *server* → ***public*** / *clients* → ***private***: service operation requires that the service be known by all the interested parties, while service users do not need to be previously known by the service suppliers
  - *server* → ***eternal*** / *clients* → ***mortals***: service availability require it to be permanently operational, while its users only manifest themselves from time to time
  - *centralized control*: all service operations are centered in a single entity

Basic server architecture – 1

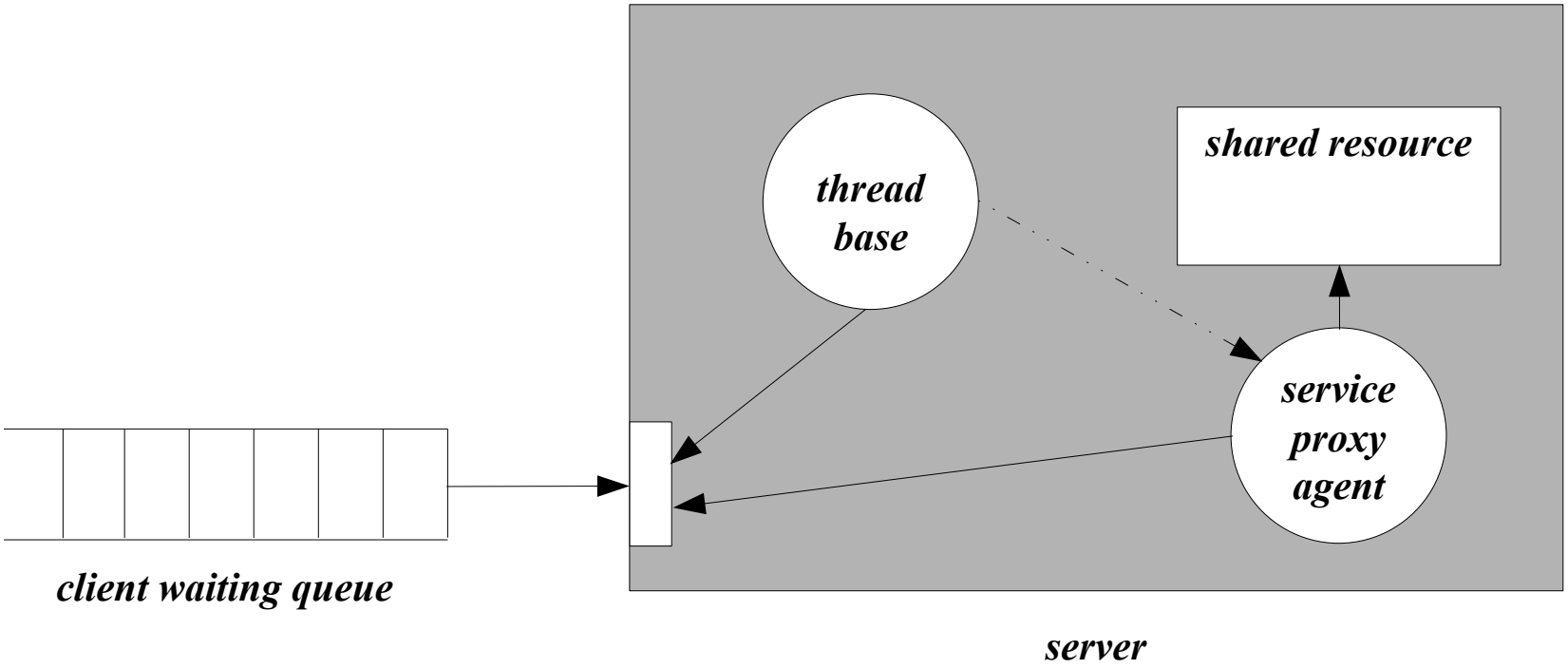


Basic server architecture – 2

- role of thread *base*
  - instantiate the shared resource
  - instantiate the communication channel and map it to a public known address
  - start listening at the communication channel
  - when a connection from a *client* process is established, create a *service proxy agent* thread to deal with the *client* request
- role of thread *service proxy agent*
  - determine the operation *client* process wishes to be performed in the shared resource (*request*)
  - execute the operation on its behalf (*local execution*)
  - communicate operation result to *client* process (*reply*)

Variants of the client-server model – 1

Request serialization



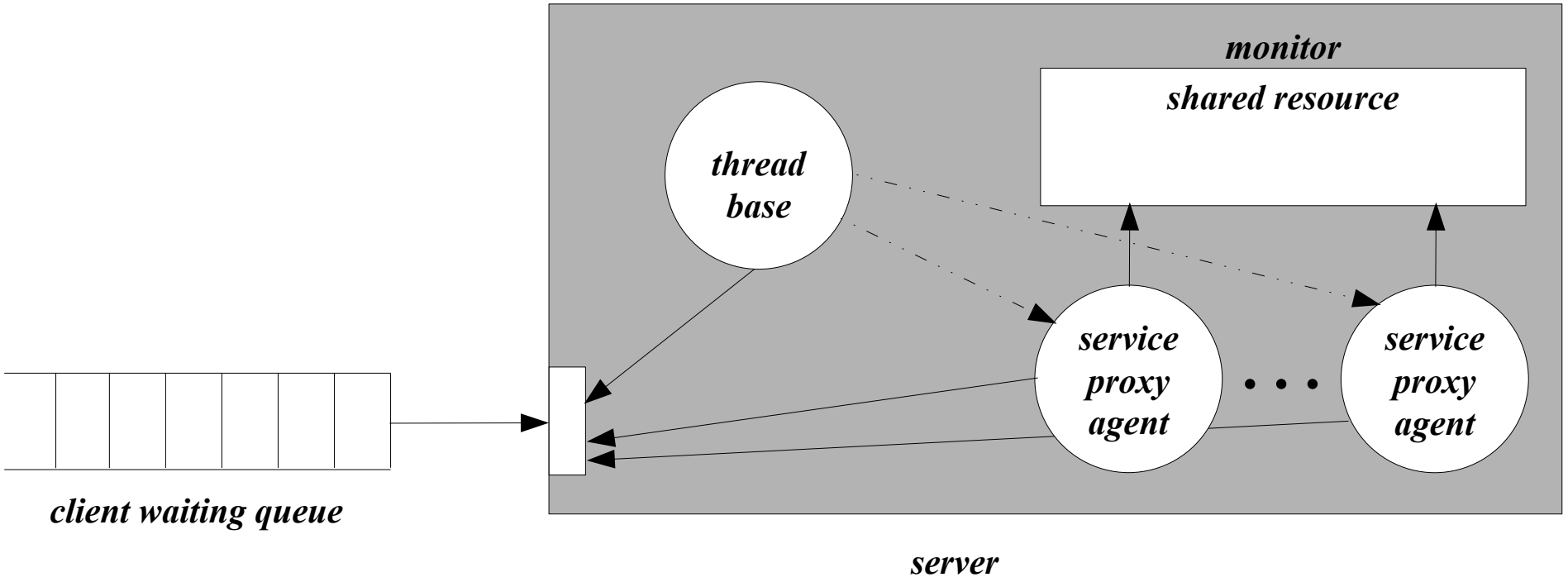
Variants of the client-server model – 2

Type 1 variant (***request serialization***)

- only a *client* process is serviced at a time: this means that the thread *base*, upon receiving a connection request, instantiates a *service proxy agent* and waits for its termination before start listening again
- the shared resource does not need any special protection to ensure mutual exclusion on access, since there is a single active *service proxy agent* thread
- it is a very simple, but rather inefficient, model since
  - the service time is not minimized, because one does not take advantage of the interaction dead times due to the lack of competition
  - it gives rise to *busy waiting* in the attempt to synchronize multiple client processes on the same shared resource.

Variants of the client-server model – 3

Server replication



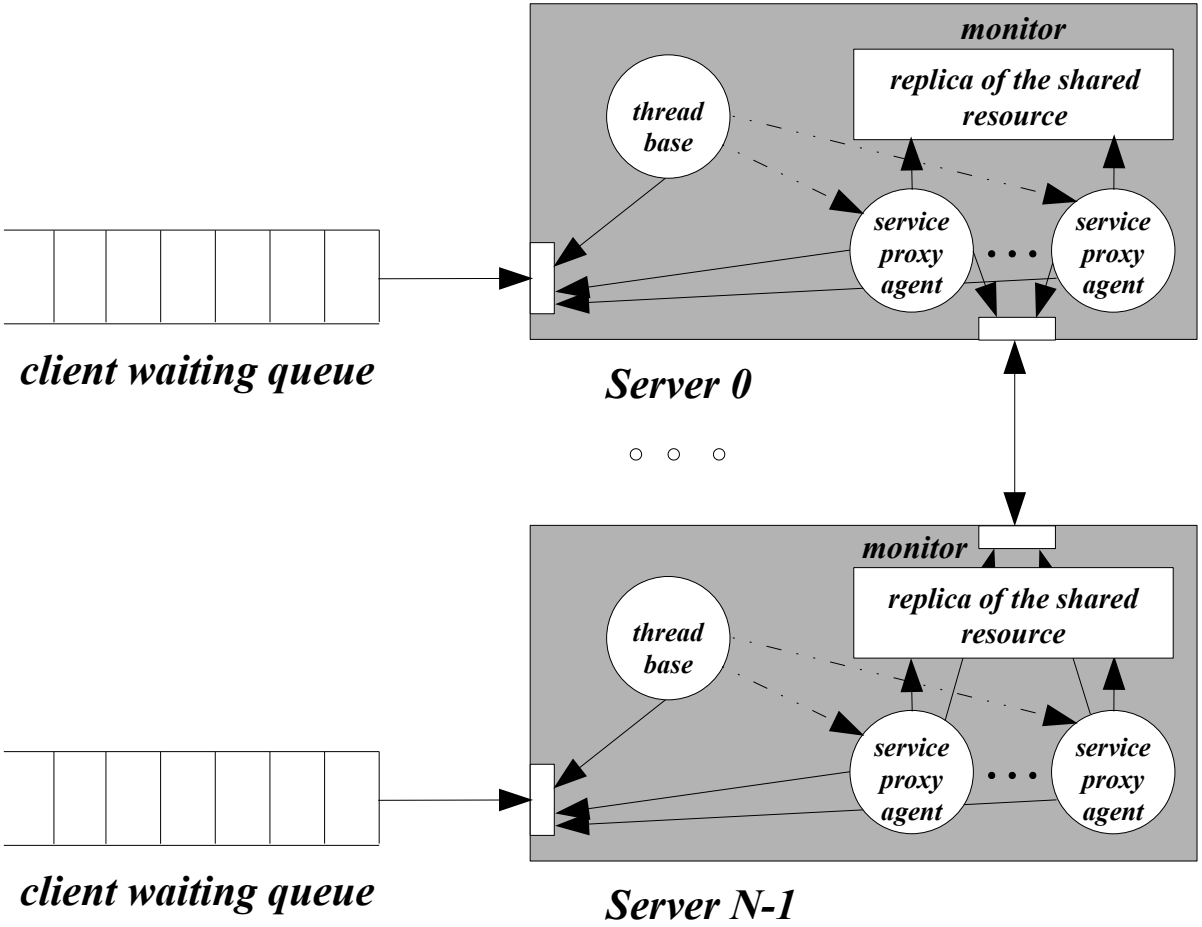
Variants of the client-server model – 4

Type 2 variant (***server replication***)

- *client* processes are serviced concurrently: this means that the thread *base*, upon receiving a connection request, instantiates a *service proxy agent* and starts listening again
- the shared resource is transformed into a monitor to ensure mutual exclusion on access, since there are now multiple active *service proxy agent* threads at the same time
- it is the traditional way *servers* are set up, as one tries to make the most of the resources of computer system where the server is located
  - the service time is minimized, because one takes advantage of the interaction dead times through concurrency
  - it enables the synchronization of different *client* processes on the same shared resource.

Variants of the client-server model – 5

Resource replication

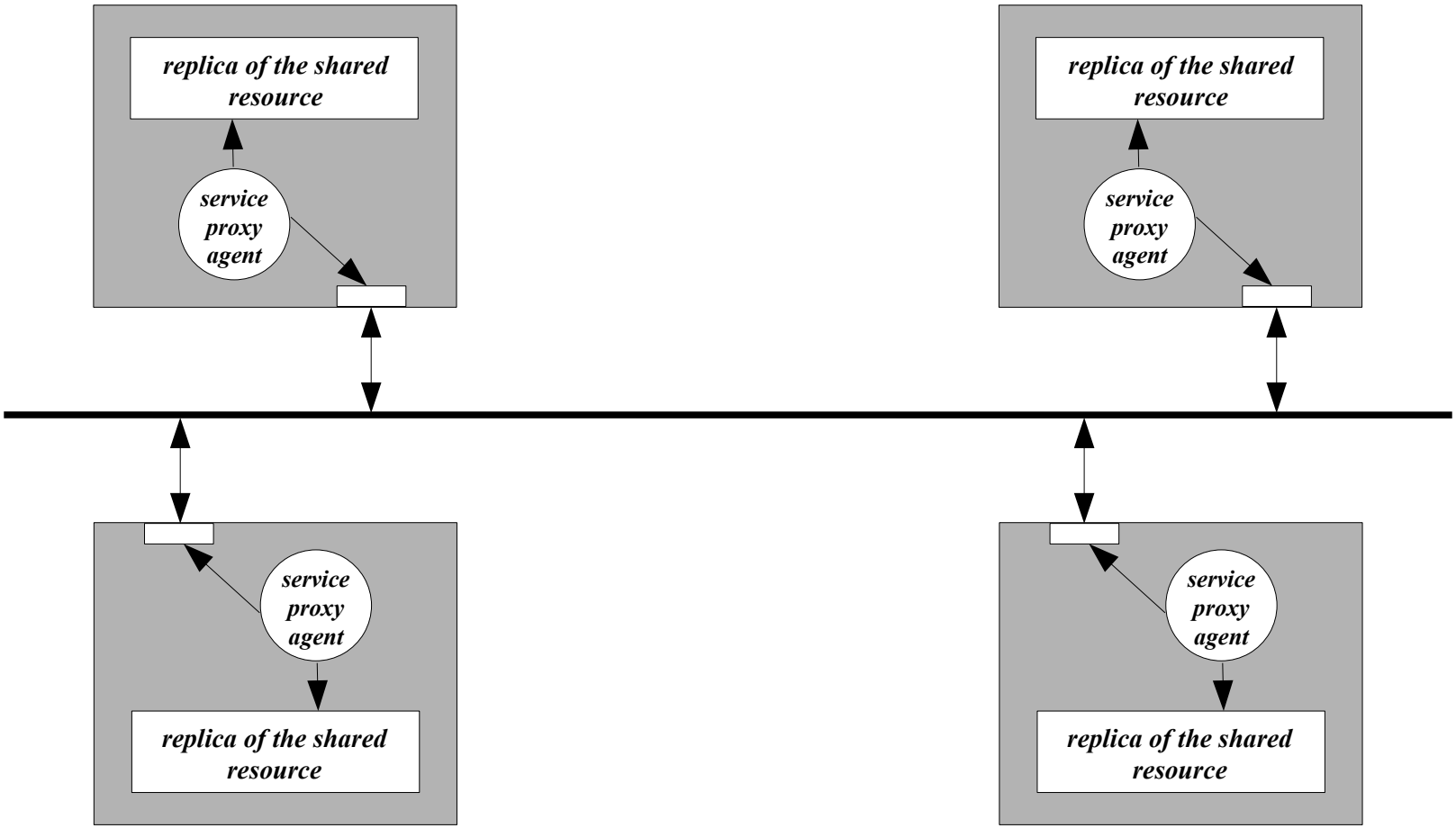


Variants of the client-server model – 6

Type 3 variant (***resource replication***)

- the service is simultaneously made available in several computer systems, each running a type 2 variant of the *server*
- the shared resource is, thus, replicated in each server, giving rise to multiple copies
- it is a sophisticated model which aims both to maximize service availability and to minimize the service time, even in load peak situations (thus, potentiating scalability)
  - the service is kept operational against the failure of particular *servers*
  - *client* requests are distributed among the available servers using a policy, provided by the DNS service, of geographical association for global requests and of rotational association for local requests
  - when there is an alteration of local data at one of the replicas of the shared resource, the need to keep the different replicas consistent arises and has to be dealt with.

Peer communication – 1



Peer communication – 2

- the service is simultaneously made available in several computer systems, each playing the same role; that is, as far as the service is concerned, there is no difference among the different processing nodes, called here *peers*
- the shared resource is, thus, replicated in each peer, giving rise to multiple copies
- it is a sophisticated model which aims both to maximize service availability and to minimize the service time, even in load peak situations (thus, potentiating scalability)
  - the service is kept operational against the failure of particular *peers*
  - in special situations, one of the peers has to assume a leading role so that the system as a whole may make a smooth transition from one stable state into another; the leader's choice is typically subjected to an election process where consensus must be reached.

Communication primitives - 1

Communication through *message passing* assumes two entities: the *forwarder*, which sends the message; and the *recipient*, that receives it.

The *send* primitive has at least two parameters: the destination address and a reference to a buffer in user space containing the data to be sent. Similarly, the *receive* primitive also has at least two parameters: the source address and a reference to a buffer in user space where the received data is to be stored.

Typically, the communication is *buffered* by the operating system: that is, on a send operation, data are first transferred into a kernel buffer before being properly delievered to the network; on a receive operation, data are first stored in a kernel buffer, upon reception, and are only transferred into the user buffer when the *receive* primitive is called.

Communication primitives - 2

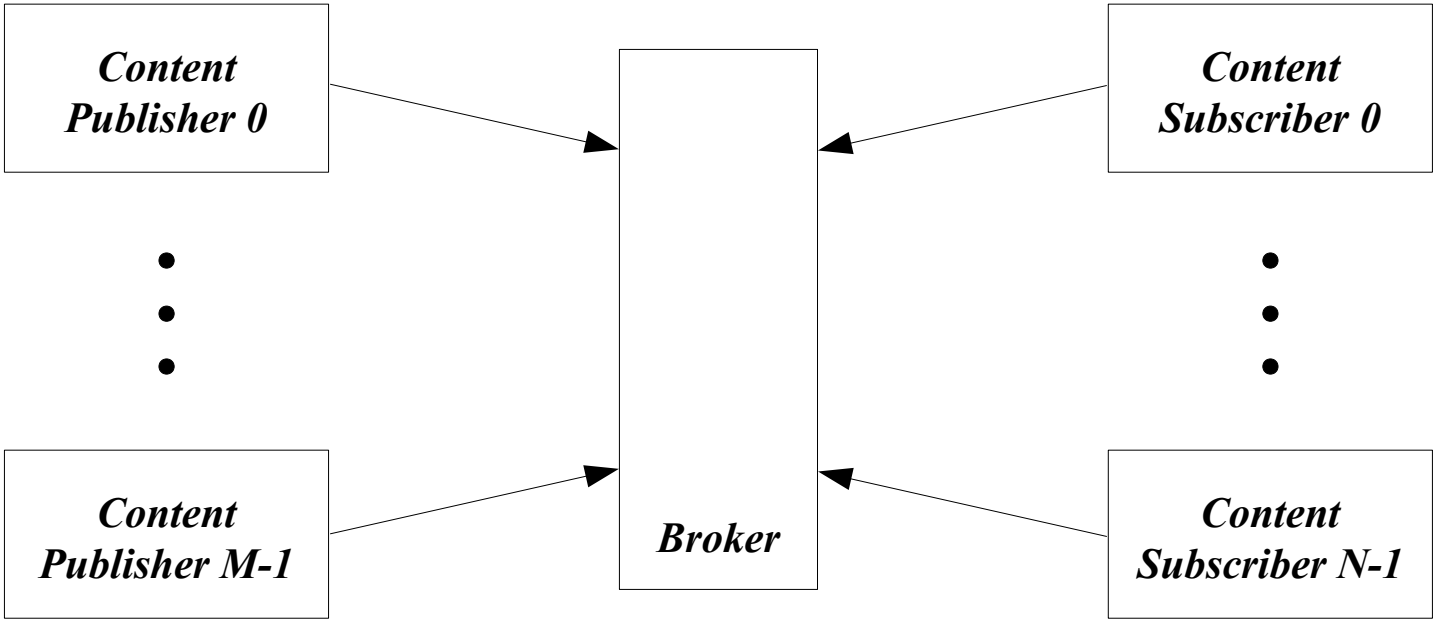
Communication primitives may be divided into

- *synchronous* – if *send* and *receive* primitives are coupled together: the send operation only completes when the forwarding processing node is aware that, on the recipient processing node, the receive operation has also been completed
- *asynchronous* – if *send* and *receive* primitives are totally uncoupled: the send operation completes as soon as data are transferred from the buffer in user space; there is no asynchronous receive primitive.

They may be further divided into

- *blocking* – if control only returns to the invoking process after the operation, whether synchronous or asynchronous, has completed
- *non-blocking* – if control returns to the invoking process immediately after the primitive is called, even if the respective operation has not yet been completed.

Publisher-subscriber model - 1

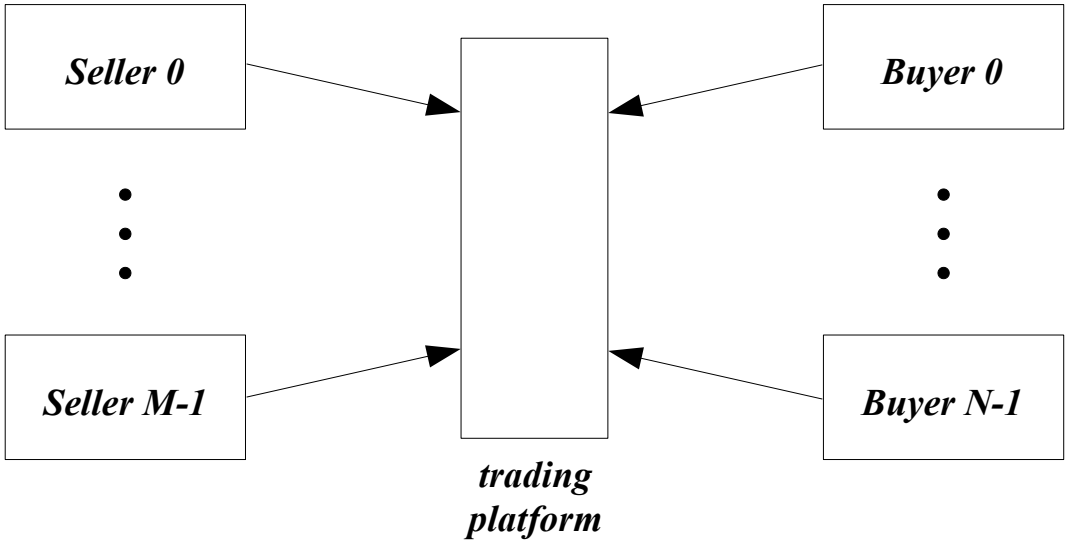


Publisher-subscriber model - 2

- there are multiple service providers, the *publishers*, and multiple service recipients, the *subscribers*, which are totally decoupled from one another through the mediation of an intermediary service, the *broker*
  - *publishers* produce information according to different topics and act as clients of the *broker* which store it and make it available to *subscriber* groups that have explicitly subscribed the topic it is associated with
  - *subscribers* act as clients both of the *publishers* and of the *broker*: the *publishers*, as they consume specific information produced by them; the *broker*, as they let it know of the topics they are interested in and that they should be alerted of when new data are made available
- the key feature is that, in contrast with the conventional *client-server model*, there is no synchronous interaction taking place here.

Publisher-subscriber model - 3

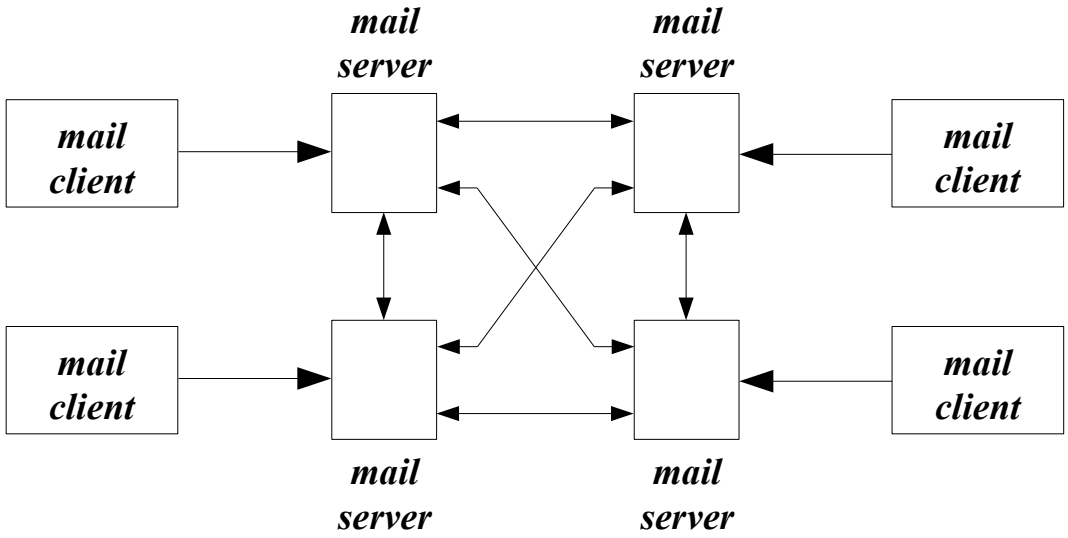
Electronic trading or **e-trading**



- the *sellers* act as *publishers* who post data about the products they want to sell at the trading platform
- the *buyers* act as *subscribers* who request the trading platform information about the products they want to buy
- the *trading platform* is the *broker* who manages possible transactions among them.

Publisher-subscriber model - 4

Electronic mail or **e-mail**



- the *mail servers* act as both *brokers* to local *mail clients* and other *mail servers*, for managing local messages, and as *publishers* to others *mail servers*, for forwarding remote messages
- the *mail clients* act as both *publishers* for sending messages and as *subscribers* for receiving messages.

Fundamental models

In a distributed system, where processes cooperate and communicate over a network, performance is directly related to the effectiveness of message exchange. When a remote service is part of it, the speed at which the exchange takes place is determined not only by the load and the performance of the server, but also by the routing and transferring capabilities of the interconnecting network and the delays in all the software components involved. To achieve short interactive response times, systems must be composed of relatively few software layers and the amount of data transferred per interaction should be small.

But fast response is not all that matters for a good *quality of service* to be attained. *Reliability*, *security* and *adaptability* to meet changing system configurations and resource availability are also considered important. In applications which deal with *time-critical* data, the availability of the necessary computing and network resources at the appropriate times is paramount.

Performance of a communication channel

Communication channels may be implemented either by data streams which connect the communication endpoints, or by simple message passing over the computer network.

The following properties are relevant

- *latency* – the delay between the times the sender process starts transmitting the message and the receiver process starts receiving it; it depends on the time the operating system services at both ends take to process the data, the delay to access the network and the routing overhead
- *bandwidth* – the total amount of information that can be transmitted over the computer network in a given time
- *jitter* – the time variation it takes to deliver a sequence of similar messages between the two endpoints.

Variants of the interaction model

In a distributed system, it is very difficult to set precise limits on the time taken for process execution, message delivery and local clock drift.

Tho opposing extreme situations are

- *synchronous distributed systems*
  - the time to execute each process step has known lower and upper bounds
  - each message transmitted over a communication channel is received within a known upper bound
  - each process has a local clock whose drift rate from *real time* has a known upper bound
- *asynchronous distributed systems*
  - the time to execute each process step has an arbitrary, but finite, upper bound
  - each message transmitted over a communication channel is received within an arbitrary, but finite, upper bound
  - each process has a local clock whose drift rate from *real time* is arbitrary.

Failure model

In a distributed system, both the intervening processes and the communication channels may *fail*. This means that their behavior may move apart from the pattern which was defined as desirable or correct.

Failures are usually classified as

- *omission failures* – when the actions prescribed to take place simply do not occur
- *timing failures* – when the actions prescribed to take place do not obey to the previously established time limits
- *arbitrary or bysantine failures* – when an unexpected error can temporally or permanently occur in result of a malfunction of any system component.

Failure classification

Failure Class	*Affects*	*Description*
fail-stop	process	a process halts and remain halted (it is assumed that processes either function correctly, or else stop; other processes may detect its state)
crash	process	a process halts and remain halted (other processes can not detect its state)
omission	channel	a message placed in the outgoing message buffer of the fowarder never arrives at the incoming message buffer of the recipient
send omission	process	a process completes a send operation, but no message is placed in the outgoing message buffer
receive omission	process	a message is placed in the incoming message buffer, but a receive operation by the recipient does not get it
clock	process	the process local clock exceeds the bounds on its drift rate from real time
performance	process	the process exceeds its bounds on the execution time between two operations
performance	channel	a message transmission time exceeds the stated bound
arbirary (or bysantine)	either process or channel	a process / channel exhibits an erratic behavior: it may transmit arbitrary messages at arbitrary times, commit omissions; or a process may stop or take an incorrect action

Security model - 1

The security of a distributed system can be achieved by securing the processes and the communication channels used in their interactions and by protecting against unauthorized access to the resources they encapsulate.

Resources are intended to be used in different ways by different users. They can either hold data private to specific users, accessible to special classes of users or shared by everybody who deals with the system. To support such an environment, *access rights* are defined and specify what operations are available and who is allowed in a differential way to perform them on the resource.

Users are, thus, included in the model as beneficiaries of resource access rights. An authority, called the *principal* and being either a user or a process, is associated with each operation invocation and operation result.

The *server* is responsible for verifying the identity of the principal behind each service request and checking if it has the required access rights so that the operation may be performed on its behalf; otherwise, reject the request. The *client*, on the other hand, must verify the identity of the principal behind the server to be sure that the reply comes from the intended one.

Security model - 2

To model security threats, an *enemy*, or an *adversary* as it is sometimes called, is postulated. The *enemy* is capable of sending any message to any process and/or reading any message exchanged by a pair of processes. The *enemy* may attack a distributed system from a computer platform that is either legitimately connected to the network, or that is connected in an authorized manner.

The threats may be divided in

- *threats to processes* – a process that is designed to handle incoming requests may receive a message which it can not necessary identify the forwarder
- *threats to communication channels* – an enemy may copy, alter and/or inject messages as they travel over the computer network, thus, putting at risk the privacy, the integrity and the availability of system information.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 2: *System models*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 2: *Architectures*
    - Sections 2.1 to 2.3