MODELS OF DISTRIBUTED SYSTEMS

1. **Architectural Models**
2. **Interaction Models**
3. **Fault Models**

Basic Elements

- **Resources** in a distributed system are shared between **users**. They are normally encapsulated within one of the computers and can be accessed from other computers by communication.
- Each resource is managed by a program, the *resource manager*; it offers a communication interface enabling the resource to be accessed by its users.
- Resource managers can be in general modelled as *processes*.
  If the system is designed according to an object-oriented methodology, resources are encapsulated in *objects*.

Architectural Models

How are responsibilities distributed between system components and how are these components placed?

- Client-server model
- Peer-to-peer

Variations of the above two:

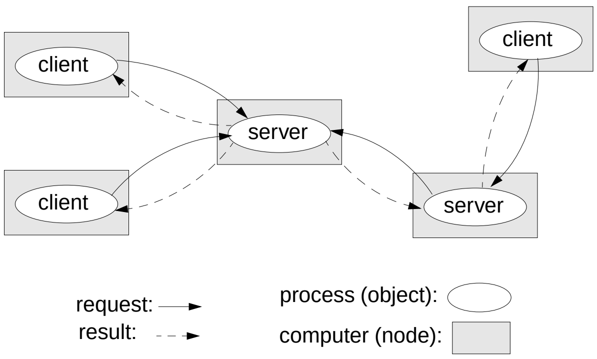
- Proxy server
- Mobile code
- Mobile agents
- Network computers
- Thin clients
- Mobile devices

Client - Server

☞ The system is structured as a set of processes, called *servers*, that offer services to the users, called *clients*.

- The client-server model is usually based on a simple request/reply protocol, implemented with *send/receive* primitives or using *remote procedure calls* (RPC) or *remote method invocation* (RMI):
  - the client sends a request (invocation) message to the server asking for some service;
  - the server does the work and returns a result (e.g. the data requested) or an error code if the work could not be performed.

Client - Server (cont'd)

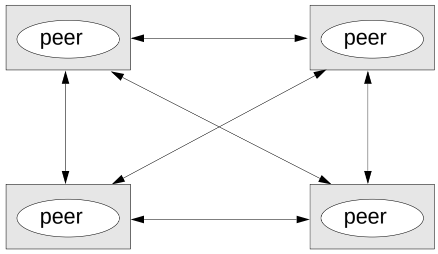


- A server can itself request services from other servers; thus, in this new relation, the server itself acts like a client.

Peer-to-Peer

☞ All processes (objects) play similar role.

- Processes (objects) interact without particular distinction between clients and servers.
- The pattern of communication depends on the particular application.
- A large number of data objects are shared; any individual computer holds only a small part of the application database.
- Processing and communication loads for access to objects are distributed across many computers and access links.
- This is the most general and flexible model.



Peer-to-Peer (cont'd)

☞ Some problems with client-server:

- Centralisation of service ⇒ poor scaling
  - Limitations:
    - capacity of server
    - bandwidth of network connecting the server

☞ Peer-to-Peer tries to solve some of the above

- It distributes shared resources widely

⇓

share computing and communication loads.

☞ Problems with peer-to-peer:

- High complexity due to
  - cleverly place individual objects
  - retrieve the objects
  - maintain potentially large number of replicas.

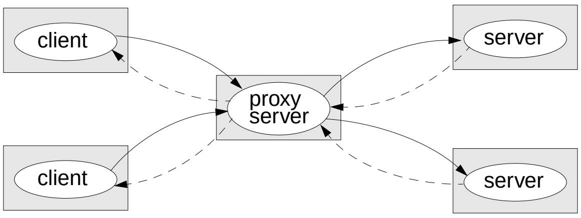
Variations of the Basic Models

☞ Client-server and peer-to-peer can be considered as basic models.

- Several variations have been proposed, with considering factors such as:
  - multiple servers and caches
  - mobile code and mobile agents
  - low-cost computers at the users' side
  - mobile devices

Proxy Server

☞ A proxy server provides copies (replications) of resources which are managed by other servers.



- Proxy servers are typically used as caches for web resources. They maintain a cache of recently visited web pages or other resources.
  When a request is issued by a client, the proxy server is first checked, if the requested object (information item) is available there.
- Proxy servers can be located at each client, or can be shared by several clients.
- The purpose is to increase performance and availability, by avoiding frequent accesses to remote servers.

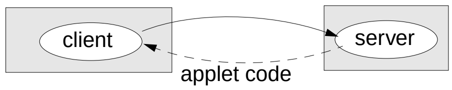
Mobile Code

☞ Mobile code: code that is sent from one computer to another and run at the destination.

Advantage: remote invocations are replaced by local ones.

Typical example: Java applets.

Step 1: load applet



Step 2: interact with applet

Mobile Agents

☞ Mobile agent: a running program that travels from one computer to another carrying out a task on someone’s behalf.

- A mobile agent is a complete program, code + data, that can work (relatively) independently.
- The mobile agent can invoke local resources/data.

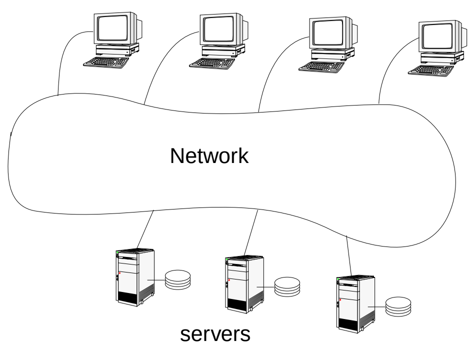
Typical tasks:

- Collect information
- Install/maintain software on computers
- Compare prises from various vendors bay visiting their sites.

Attention: potential security risk (like mobile code)!

Network Computers

Network computers



☞ Network computers do not store locally operating system or application code. All code is loaded from the servers and run locally on the network computer.

Advantages:

- The network computer can be simpler, with limited capacity; it does not need even a local hard disk (if there exists one it is used to cache data or code).
- Users can log in from any computer.
- No user effort for software management/ administration.

Thin Clients

☞ The thin client is a further step, beyond the network computer:

- Thin clients do not download code (operating system or application) from the server to run it locally. *All code is run on the server, in parallel for several clients*.
- The thin client only runs the user interface!

Advantages:

- All those of network computers but the computer at the user side is even simpler (cheaper).

☞ Strong servers are needed!

Mobile Devices

☞ Mobile devices are hardware, computing components that move (together with their software) between physical locations.

- This is opposed to software agents, which are software components that migrate.
- Both clients and servers can be mobile (clients more frequently).

☞ Particular problems/issues:

- Mobility transparency: clients should not be aware if the server moves (e.g., the server keeps its Internet address even if it moves between networks).
- Problems due to variable connectivity and bandwidth.
- The device has to explore its environment:
  - Spontaneous interoperation: associations between devices (e.g. clients and servers) are dynamically created and destroyed.
  - Context awareness: available services are dependent on the physical environment in which the device is situated.

Interaction Models

How do we handle time? Are there time limits on process execution, message delivery, and clock drifts?

- Synchronous distributed systems
- Asynchronous distributed systems

Synchronous Distributed Systems

Main features:

- Lower and upper bounds on execution time of processes can be set.
- Transmitted messages are received within a known bounded time.
- Drift rates between local clocks have a known bound.

Important consequences:

1. In a synchronous distributed system there is a notion of global physical time (with a known relative precision depending on the drift rate).
2. Only synchronous distributed systems have a predictable behaviour in terms of timing. *Only such systems can be used for hard real-time applications*.
3. In a synchronous distributed system it is possible and safe to use timeouts in order to detect failures of a process or communication link.

☞ It is difficult and costly to implement synchronous distributed systems.

Asynchronous Distributed Systems

☞ Many distributed systems (including those on the Internet) are asynchronous.

- No bound on process execution time (nothing can be assumed about speed, load, reliability of computers).
- No bound on message transmission delays (nothing can be assumed about speed, load, reliability of interconnections)
- No bounds on drift rates between local clocks.

Important consequences:

1. In an asynchronous distributed system there is no global physical time. Reasoning can be only in terms of logical time (see lecture on time and state).
2. Asynchronous distributed systems are unpredictable in terms of timing.
3. No timeouts can be used.

Asynchronous Distributed Systems (cont'd)

☞ Asynchronous systems are widely and successfully used in practice.

In practice timeouts are used with asynchronous systems for failure detection.
However, additional measures have to be applied in order to avoid duplicated messages, duplicated execution of operations, etc.

Fault Models

What kind of faults can occur and what are their effects?

- Omission faults
- Arbitrary faults
- Timing faults

☞ Faults can occur both in processes and communication channels. The reason can be both software and hardware faults.

☞ Fault models are needed in order to build systems with predictable behaviour in case of faults (systems which are fault tolerant).

☞ Of course, such a system will function according to the predictions, only as long as the real faults behave as defined by the "fault model". If not ....

☞ These issues will be discussed in some of the following chapters and in particular in the chapter on "Recovery and Fault Tolerance".

Omission Faults

☞ A processor or communication channel fails to perform actions it is supposed to do.
This means that the particular action is not performed!

- We do not have an omission fault if:
  - An action is delayed (regardless how long) but finally executed.
  - An action is executed with an erroneous result.

☞ With synchronous systems, omission faults can be detected by timeouts.
- If we are sure that messages arrive, a timeout will indicate that the sending process has crashed. Such a system has a *fail-stop* behaviour.

Arbitrary (Byzantine) Faults

☞ This is the most general and worst possible fault semantics.

Intended processing steps or communications are omitted or/and unintended ones are executed. Results may not come at all or may come but carry wrong values.

Timing Faults

☞ Timing faults can occur in synchronous distributed systems, where time limits are set to process execution, communications, and clock drifts.

A timing fault occurs if any of this time limits is exceeded.

Summary

- Models can be used to provide an abstract and simplified description of certain relevant aspects of distributed systems.
- Architectural models define the way responsibilities are distributed among components and how they are placed in the system.

  We have studied three architectural models:
  1. Client-server model
  2. Peer-to-peer
  3. Several variations of the two
- Interaction models deal with how time is handled throughout the system.

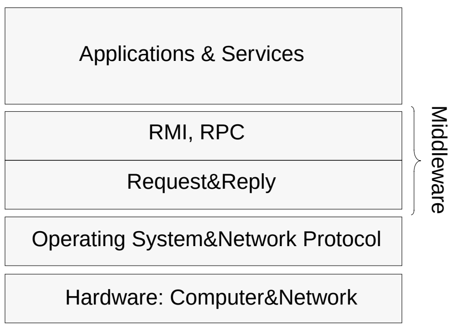
  Two interaction models have been introduced:
  1. Synchronous distributed systems
  2. Asynchronous distributed systems
- The fault model specifies what kind of faults can occur and what their effects are.

  Fault models:
  1. Omission faults
  2. Arbitrary faults
  3. Timing faults

COMMUNICATION IN DISTRIBUTED SYSTEMS

1. **Communication System: Layered Implementation**
2. **Network Protocol**
3. **Request and Reply Primitives**
4. **RMI and RPC**
5. **RMI and RPC Semantics and Failures**
6. **Group Communication**

Communication Models and their Layered Implementation

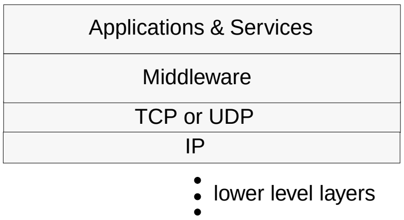


- This chapter concentrates on communication between distributed objects by means of two models: **remote method invocation** (**RMI**) and **remote procedure call** (**RPI**).
- RMI, as well as RPC, are based on *request* and *reply* primitives.
- *Request* and *reply* are implemented based on the network protocol (e.g. TCP or UDP in case of the Internet).

Network Protocol

☞ Middleware and distributed applications have to be implemented on top of a network protocol. Such a protocol is implemented as several layers.

In case of the Internet:



- TCP (Transport Control Protocol) and UDP (User Datagram Protocol) are both transport protocols implemented on top of the Internet protocol (IP).

Network Protocol (cont'd)

☞ TCP is a reliable protocol.

- TCP guarantees the delivery to the receiving process of all data delivered by the sending process, in the same order.
- TCP implements additional mechanisms on top of IP to meet reliability guarantees.
  - Sequencing:
    A sequence number is attached to each transmitted segment (packet). At the receiver side, no segment is delivered until all lower-numbered segments have been delivered.
  - Flow control:
    The sender takes care not to overwhelm the receiver (or intermediate nodes). This is based on periodic acknowledgements received by the sender from the receiver.
  - Retransmission and duplicate handling:
    If a segment is not acknowledged within a specified timeout, the sender retransmits it. Based on the sequence number, the receiver is able to detect and reject duplicates.
  - Buffering:
    Buffering is used to balance the flow between sender and receiver. If the receiving buffer is full, incoming segments are dropped. They will not be acknowledged and the sender will retransmit them.
  - Checksum:
    Each segment carries a checksum. If the received segment doesn't match the checksum, it is dropped (and will be retransmitted)

Network Protocol (cont'd)

☞ UDP is a protocol that does not guarantee reliable transmission.

- UDP offers no guarantee of delivery.
  According to the IP, packets may be dropped because of congestion or network error. UDP adds no additional reliability mechanism to this.
- UDP provides a means of transmitting messages with minimal additional costs or transmission delays above those due to IP transmission.
  Its use is restricted to applications and services that do not require reliable delivery of messages.
- If reliable delivery is requested with UDP, reliability mechanisms have to be implemented at the application level.

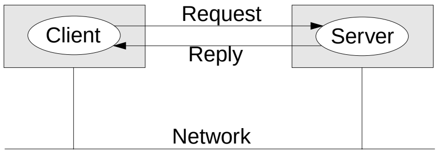
Request and Reply Primitives

☞ Communication between processes and objects in a distributed system is performed by message passing.

- In a typical scenario (e.g. client-server model) such a communication is through request and reply messages.

Request-Reply Communication in a Client-Server Model

The system is structured as a group of processes (objects), called *servers*, that deliver services to *clients*.



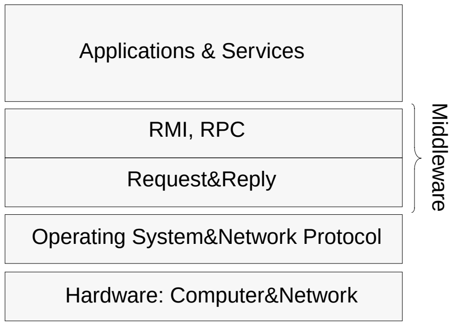
The client:

- - - - - - - - - - - - - - - - - -
send (request) to server_reference;
receive(reply);
- - - - - - - - - - - - - - - - - -

The server:

- - - - - - - - - - - - - - - - - -
receive(request) from client-reference;
execute requested operation
send (reply) to client_reference;
- - - - - - - - - - - - - - - - - -

Remote Method Invocation (RMI) and Remote Procedure Call (RPC)

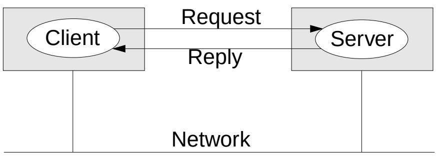


The goal: make, for the programmer, distributed computing look like centralized computing.

The solution:

- Asking for a service is solved by the client issuing a simple *method invocation* or *procedure call*; because the server can be on a remote machine this is a *remote* invocation (call).
- *RMI (RPC) is transparent*: the calling object (procedure) is not aware that the called one is executing on a different machine, and vice versa.

Remote Method Invocation



The client writes:

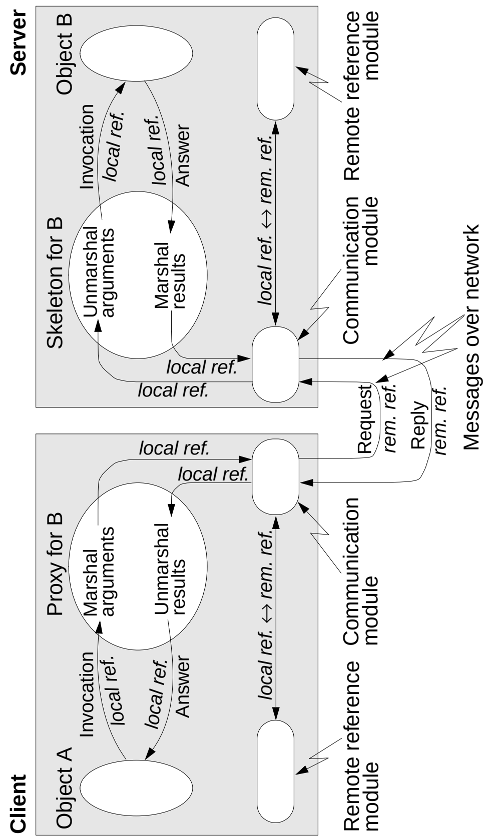
- - - - - - - - - - - - - - - - - -
server_id.service(values_to_server, result_arguments);
- - - - - - - - - - - - - - - - - -

The server contains the method:

public service(**in** type1 arg_from_client; **out** type2 arg_to_client)
{ - - - };

- The programmer is unaware of the request and reply messages which are sent over the network during execution of the RMI.

Implementation of RMI



Implementation of RMI (cont'd)

Who are the players?

- Object A asks for a service
- Object B delivers the service

Who more?

- The proxy for object B
  - If an object A holds a remote reference to a (remote) object B, there exists a proxy object for B on the machine which hosts A. The proxy is created when the remote object reference is used for the first time. For each method in B there exists a corresponding method in the proxy.
  - The proxy is the local representative of the remote object ⇒ the remote invocation from A to B is initially handled like a local one from A to the proxy for B.
  - At invocation, the corresponding proxy method *marshals* the arguments and builds the message to be sent, as a request, to the server. After reception of the reply, the proxy *unmarshals* the received message and sends the results, in an answer, to the invoker.

Implementation of RMI (cont'd)

- The skeleton for object B
  - On the server side, there exists a skeleton object corresponding to a class, if an object of that class can be accessed by RMI. For each method in B there exists a corresponding method in the skeleton.
  - The skeleton receives the request message, unmarshals it and invokes the corresponding method in the remote object; it waits for the result and marshals it into the message to be sent with the reply.
  - A part of the skeleton is also called *dispatcher*. The dispatcher receives a request from the *communication module*, identifies the invoked method and directs the request to the corresponding method of the skeleton.

Implementation of RMI (cont'd)

- Communication module
  - The communication modules on the client and server are responsible of carrying out the exchange of messages which implement the request/reply protocol needed to execute the remote invocation.
  - The particular messages exchanged and the way errors are handled, depends on the RMI semantics which is implemented (see slide 40).
- Remote reference module
  - The remote reference module translates between local and remote object references. The correspondence between them is recorded in a *remote object table*.
  - Remote object references are initially obtained by a client from a so called *binder* that is part of the global name service (it is not part of the remote reference module). Here servers register their remote objects and clients look up after services.

Implementation of RMI (cont'd)

☞ Question 1

What if the two computers use different representation for data (integers, chars, floating point)?

- The most elegant and flexible solution is to have a standard representation used for all values sent through the network; the proxy and skeleton convert to/from this representation during marshalling/unmarshalling.

☞ Question 2

Who generates the classes for proxy and skeleton?

- In advanced middleware systems (e.g. CORBA) the classes for proxies and skeletons can be generated automatically. Given the specification of the server interface and the standard representations, an interface compiler can generate the classes for proxies and skeletons.

Implementation of RMI (cont'd)

☞ Object A and Object B belong to the application.

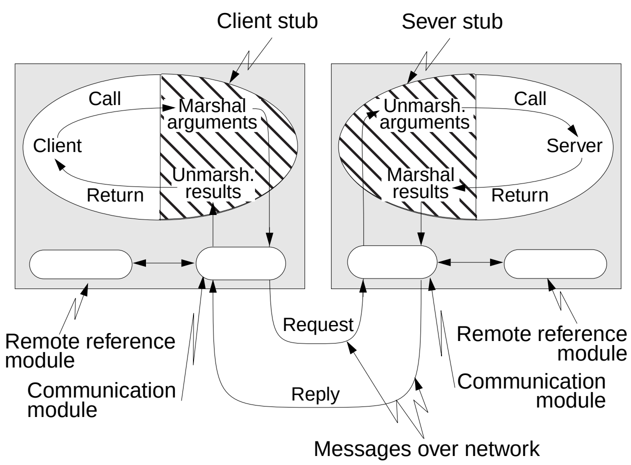
☞ Remote reference module and communication module belong to the middleware.

☞ The proxy for B and the skeleton for B represent the so called *RMI software*. They are situated at the border between middleware and application and usually can be generated automatically with help of available tools that are delivered together with the middleware software.

The History of an RMI

1. The calling sequence in the client object activates the method in the proxy corresponding to the invoked method in B.
2. The method in the proxy packs the arguments into a message (marshalling) and forwards it to the communication module.
3. Based on the remote reference obtained from the remote reference module, the communication module initiates the request/reply protocol over the network.
4. The communication module on the server's machine receives the request. Based on the local reference received from the remote reference module the corresponding method in the skeleton for B is activated.
5. The skeleton method extracts the arguments from the received message (unmarshalling) and activates the corresponding method in the server object B.
6. After receiving the results from B, the method in the skeleton packs them into the message to be sent back (marshalling) and forwards this message to the communication module.
7. The communication module sends the reply, through the network, to the client's machine.
8. The communication module receives the reply and forwards it to the corresponding method in the proxy.
9. The proxy method extracts the results from the received message (unmarshalling) and forwards them to the client.

Remote Procedure Call



RMI Semantics and Failures

- If everything works OK, RMI behaves exactly like a local invocation. *What if certain failures occur*?

We consider the following classes of failures which have to be handled by an RMI protocol:

1. Lost request message
2. Lost reply message
3. Server crash
4. Client crash

☞ We will consider an *omission failure* model.
This means:
- Messages are either lost or received correctly.
- Client or server processes either crash or execute correctly. After crash the server can possibly restart with or without loss of memory.

Lost Request Messages

- The communication module starts a timer when sending the request;
  if the timer expires before a reply or acknowledgment comes back, the communication module sends the message again.

Problem: what if the request was not truly lost (but, for example, the server is too slow) and the server receives it more than once?

- We have to avoid that the server executes certain operations more than once.
- Messages have to be identified by an identifier and copies of the same message have to be filtered out:
  - If the duplicate arrives and the server has not yet sent the reply ⇒ simply send the reply.
  - If the duplicate arrives after the reply has been sent ⇒ the reply may have been lost or it didn't arrive in time (see next slide).

Lost Reply Message

The client can not really distinguish the loss of a request from that of a reply; it simply resends the request because no answer has been received in the right time.

☞ If the reply really got lost, when the duplicate request arrives at the server it already has executed the operation once!

☞ In order to resend the reply the server may need to reexecute the operation in order to get the result.

Danger!

- Some operations can be executed more than once without any problem; they are called *idempotent operations* ⇒ no danger with executing the duplicate request.
- There are operations which cannot be executed repeatedly without changing the effect (e.g. transferring an amount of money between two accounts) ⇒ *history* can be used to avoid re-execution.

History: the history is a structure which stores a record of reply messages that have been transmitted, together with the message identifier and the client which it has been sent to.

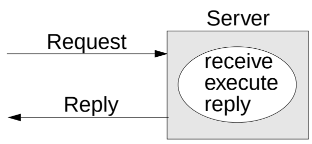
Conclusion with Lost Messages

☞ Based on the previous discussion ⇒ correct, exactly once semantics (see slide 46) can be implemented in the case of lost (request or reply) messages. *If all the measures are taken* (duplicate filtering and history):
  - When, finally, a reply arrives at the client, the call has been executed correctly (exactly one time).
  - If no answer arrives at the client (e.g. because of broken line), an operation has been executed at most one time.

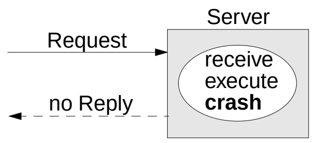
☞ However, the situation is different if we assume that the server can crash.

Server Crash

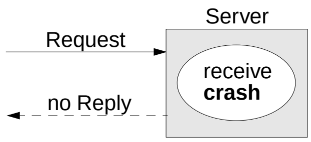
a) The normal sequence:



b) The server crashes after executing the operation but before sending the reply (as result of the crash, the server doesn't remember that it has executed the operation):



c) The server crashes before executing the operation:



Server Crash (cont'd)

Big problem!

The client cannot distinguish between cases *b* and *c*!

However they are very different and should be handled in a different way!

What to do if the client noticed that the server is down (it didn't answer to a certain large number of repeated requests)?

Server Crash (cont'd)

Alternative 1: *at least once semantics*

- The client's communication module sends repeated requests and waits until the server reboots or it is rebound to a new machine; when it finally receives a reply, it forwards it to the client.

⇩

When the client got an answer, the RMI has been carried out at least one time, but possibly more.

Alternative 2: *at most once semantics*

- The client's communication module gives up and immediately reports the failure to the client (e.g. by raising an exception)

⇩

- If the client got an answer, the RMI has been executed exactly once.
- If the client got a failure message, the RMI has been carried out at most one time, but possibly not at all.

Alternative 3: *exactly once semantics*

- This is what we would like to have (and what we could achieve for lost messages): the RMI has been carried out exactly one time.
  However this cannot be guaranteed, *in general*, for the situation of server crashes.

Client Crash

The client sends a request to a server and crashes before the server replies.

⇩

The computation which is active in the server becomes an *orphan* - a computation nobody is waiting for.

Problems:

- wasting of CPU time
- locked resources (files, peripherals, etc.)
- if the client reboots and repeats the RMI, confusion can be created.

The solution is based on identification and killing the orphans.

Conclusion with RMI Semantics and Failures

☞ If the problem of errors is ignored, *maybe semantics* is achieved for RMI:
  - the client, in general, doesn't know if the remote method has been executed once, several times or not at all.

☞ If server crashes can be excluded, *exactly once semantics* is possible to achieve, by using retries, filtering out duplicates, and using history.

☞ If server crashes with loss of memory (case b on slide 44) are considered, only *at least once* and *at most once* semantics are achievable in the best case.

In practical applications, *servers can survive crashes without loss of memory.* In such cases history can be used and duplicates can be filtered out after restart of the server:

- the client repeats sending requests without being in danger operations to be executed more than one time (this is different from alternative 2 on slide 46):
  - If no answer is received after a certain amount of tries, the client is notified and he knows that the method has been executed at most one time or not at all.
  - If an answer is received it is forwarded to the client who knows that the method has been executed exactly one time.

Conclusion with RMI Semantics and Failures (cont'd)

☞ RMI semantics is different in different systems. Sometimes several semantics are implemented among which the user is allowed to select.

☞ And no hope about achieving exactly once semantics if servers crash ?!

In practice, systems can come close to this goal. Such are transaction-based systems with sophisticated protocols for error recovery.

☞ More discussion in chapter on fault tolerance.

Group Communication

☞ The assumption with client-server communication and RMI (RPC) is that two parties are involved: the client and the server.

☞ Sometimes, however, communication involves multiple processes, not only two. A solution is to perform separate message passing operations or RMIs to each receiver.

- With *group communication* a message can be sent to multiple receivers in one operation, called *multicast*.

Why do we need it?

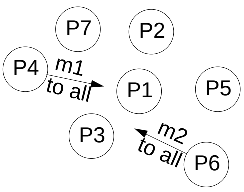
- Special applications: interest-groups, mail-lists, etc.
- Fault tolerance based on replication: a request is sent to *several* servers which all execute the same operation (if one fails, the client still will be served).
- Locating a service or object in a distributed system: the client sends a message to all machines but only the one (or those) which holds the server/object responds .
- Replicated data (for reliability or performance): whenever the data changes, the new value has to be multicast to all processes managing replicas.

Group Communication (cont'd)

Essential features:

- **Atomicity** (all-or-nothing property): when a message is multicast to a group, it will either arrive correctly at all members of the group or at none of them.

- **Ordering**
  - FIFO ordering: The messages *from any one client to a particular server* are delivered in the order sent.
  - Totally-ordered multicast: when several messages are transmitted to a group the messages reach all the members of the group in the same order.



Either each process receives the messages in the order *m1*, *m2* or each receives them in the order *m2*, *m1*.

Summary

- Middleware implements high level communication under the form of Remote Method Invocation (RMI) or Remote Procedure Call (RPC). They are based on request/reply protocols which are implemented using message passing on top of a network protocol (like the Internet).
- Client-server is a very frequently used communication pattern based on a request/reply protocol; it can be implemented using *send*/*receive* message passing primitives.
- RMI and RPC are elegant mechanisms to implement client-server systems. Remote access is solved like a local one.
- Basic components to implement RMI are: the proxy object, the skeleton object, the communication module and the remote reference module.
- An essential aspect is RMI semantics in the presence of failures. The goal is to provide *exactly once* semantics. This cannot be achieved, in general, in the presence of server crashes.
- Client-server communication, in particular RMI, involves exactly two parties. With group communication a message can be sent to multiple receivers.
- Essential features of a group communication facility are: atomicity and ordering.