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. Indirect Communication
7. Group Communication
8. Publish-Subscribe Systems
Communication Models and their Layered Implementation

Applications & Services

RMI, RPC

Request&Reply

Operating System&Network Protocol

Hardware: Computer&Network

Middleware
In this chapter: communication between distributed objects by means of two models: remote method invocation (RMI) and remote procedure call (RPC).

RMI, as well as RPC, are implemented on top of request and reply primitives.

Request and reply are implemented on top of the network protocol (e.g. TCP or UDP in case of the Internet).
Middleware and distributed applications are implemented on top of a network protocol. Such a protocol is implemented as several layers.

**In case of the Internet:**

<table>
<thead>
<tr>
<th>Applications &amp; Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware</td>
</tr>
<tr>
<td>TCP or UDP</td>
</tr>
<tr>
<td>IP</td>
</tr>
</tbody>
</table>

- lower level layers

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

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 mechanisms on top of IP to meet reliability guarantees.
  - **Sequencing:**
    A sequence number is attached to each transmitted segment (packet). At the receiver side, packets are delivered in order of this number.
  
  - **Flow control:**
    The sender takes care not to overwhelm the receiver. 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 timeout, it is retransmitted. Using sequence number, the receiver detects and rejects duplicates.

  - **Buffering:**
    Buffering balances the flow. If the receiving buffer is full, incoming segments are dropped. They will be retransmitted by the sender.

  - **Checksum:**
    Each segment carries a checksum. If the received segment does not match the checksum, it is dropped (and will be retransmitted).
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 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 on top of the network protocol (in the middleware).
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 Primitives in 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 *method invocation* or *procedure call*; 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

Client

Object A
Invocation
Marshaling
Unmarshaling
Remote reference
module

Proxy for B
Unmarshal
arguments
Remote reference
module

Communication
module

Invocation
Unmarshal
arguments

Remote reference
module

Server

Object B
Invocation
Remote reference
module

Skeleton for B
Unmarshal
arguments

Communication
module

Messages over network

Remote reference
module

Unmarshal
results

Remote reference
module

Local reference ↔ remote reference
Question 1

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

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

- Object A and Object B belong to the application.

- Remote reference module and communication module belong to 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 are 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

Client stub

Sever stub

Remote reference module
Communication module

Messages over network

Call
Return

Marshal arguments

Unmarshal results

Call
Server

Unmarshal arguments

Marshal results

Return

Request
Reply
RMI Semantics and Failures

If everything works OK, RMI behaves exactly like a local invocation.

What if certain failures occur?
RMI Semantics and Failures

If everything works OK, RMI behaves exactly like a local invocation.

What if certain failures occur?

- Classes of failures that have to be handled by an RMI protocol:
  1. Lost request message
  2. Lost reply message
  3. Server crash
  4. Client crash

- We 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 request message again.*
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 request message again.*

**Problem!**

What if the request message 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 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.
Lost Reply Message

*The client can not distinguish the loss of a request from that of a reply;* it simply resends the request because no answer has been received!

- 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.
Lost Reply Message

*The client can not distinguish the loss of a request from that of a reply; it simply resends the request because no answer has been received!*

- 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:** 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

- *Exactly once semantics* can be implemented in the case of lost (request or reply) messages if *both duplicate filtering and history are provided and the message is resent until an answer arrives*:
  - Eventually a reply arrives at the client and the call has been executed correctly - exactly one time.

- However, the situation is different if we assume that the server can crash.
Server Crash

a) The normal sequence:

```
Request → Server
          | receive
          | execute
          | reply
Reply ←
```
Server Crash

a) The normal sequence:

request
receive
execute
reply

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

request
receive
execute
crash

no reply
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 lost memory and doesn’t remember that it has executed the operation):

c) The server crashes before executing the operation:
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 *lost memory* and doesn’t remember that it has executed the operation):

Big problem!
The client cannot distinguish between these cases

What to do if the client noticed that the server might be down? (it didn’t answer to repeated requests)?

c) The server crashes before executing the operation:
Server Crash

**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.
Server Crash

**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.
Server Crash

**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 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 resending requests, filtering out duplicates, and using history.

- If server crashes *with loss of memory* are considered, only *at least once* and *at most once* semantics are achievable in the best case.
Conclusion with RMI Semantics and Failures

- In practical applications, servers can survive crashes without loss of memory. Transaction based sophisticated commitment protocols are implemented in distributed database systems in order to achieve this goal. In such systems 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:
    - 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.

- RMI implementation and error handling differs between systems. Sometimes several semantics are implemented among which the user is allowed to select.
The communication primitives studied so far are based on *direct coupling* between sender and receiver: the sender has a reference/pointer to the receiver and specifies it as an argument of the communication primitive.

The sender writes something like:

```
--------------------------------------------------
send (request) to server_reference;
--------------------------------------------------
```

Very Rigid!
Direct vs. Indirect Communication

An alternative: *Indirect communication*

- No direct coupling between sender and receiver(s).
- Communication is performed via an intermediary.

- **Space uncoupling**: sender does not know the identity of receiver(s).
- **Time uncoupling**: sender and receiver(s) have independent lifetimes: they do not need to exist at the same time.

We look at two examples:

1. Group communication
2. Publish-subscribe systems
The assumption with client-server communication and RMI (RPC) is that two parties are involved: the client and the server.

Sometimes 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 a group and then it is delivered to all members of the group ⇒ multiple receivers in one operation.
Group Communication

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 sent to all processes managing replicas.
Group Communication

**Group membership management:** maintains the view of group membership, considering members joining, leaving, or failing.

**Services provided by group membership management:**

- **Group membership changes:**
  - create/destroy process groups;
  - add/withdraw processes to/from group.

- **Failure detection:**
  - Detects processes that crash or become unavailable (due to e.g. communication failure);
  - Excludes processes from membership if crashed or unavailable.

- **Notification:**
  - Notifies members of events e.g. processes joining/leaving group.

- **Group address expansion:**
  - Processes sending to group specify group identifier; address expansion provides the actual addresses for the multicast operation delivering the message to each group members.
Group Communication

Essential features:

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

Essential features:

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

- **Ordering**
  - **FIFO ordering**: Messages originating from a given sender are delivered in the order they have been sent, to all members of the group.
Group Communication

Essential features:

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

- **Ordering**
  - **FIFO ordering**: Messages originating from a given sender are delivered in the order they have been sent, to all members of the group.
  - **Total-ordering**: When several messages, from different senders, are sent to a group, the messages reach all the members of the group in the same order.
Publish-Subscribe Systems

The general objective of publish-subscribe systems is to let information propagate from publishers to interested subscribers, in an anonymous, decoupled fashion.

- **Publishers** publish events
- **Subscribers** subscribe to and receive the events they are interested in.

- Subscribers are not directly targeted from publishers but indirectly via the *notification service*.
- Subscribers express their interest by issuing subscriptions for specific notifications, independently from the publishers that produce them; they are asynchronously notified for all notifications, submitted by any publisher, that match their subscription.
Publish-Subscribe Systems

Notification Service: is a propagation mechanism that acts as a logical intermediary between publishers and subscribers, to avoid each publisher to have to know all the subscriptions for each possible subscriber.

- Both publishers and subscribers communicate only with a single entity, the notification service, that
  - stores the subscriptions associated with each subscriber;
  - receives all the notifications from publishers;
  - dispatches the notifications to the correct subscribers.
A subscription is respectively installed and removed on the notification service as result of subscriber processes executing:

\[ \text{subscribe()} \]
\[ \text{unsubscribe()} \]
A publisher submits a piece of information by executing the `publish()` operation on the notification service.

The notification service dispatches a piece of information to a subscriber by executing the `notify()` on it.

A publisher produces an event (publication), while the notification service issues the corresponding notification on interested subscribers.
Publish-Subscribe Systems

**Example:** stock trading system:

- BMW
- GM
- IBM
Publish-Subscribe Systems

**Example:** stock trading system:

1. S1 has subscribed to 'IBM', with a filter indicating that it should be notified only if the stock increases by at least 25;
Publish-Subscribe Systems

Example: stock trading system:

1. S1 has subscribed to 'IBM', with a filter indicating that it should be notified only if the stock increases by at least 25;
2. S2 and S3 have subscribed to 'IBM' and 'GM' respectively, without filter.
Publish-Subscribe Systems

**Example:** stock trading system:

1. S1 has subscribed to 'IBM', with a filter indicating that it should be notified only if the stock increases by at least 25;
2. S2 and S3 have subscribed to 'IBM' and 'GM' respectively, without filter.
3. P1 is publishing the new value of 'IBM'.

**Diagram:**

- **P1** publishes 'IBM', 95
- **S1** subscribes to 'IBM', change>25
- **S2** subscribes to 'GM'
- **S3** subscribes to 'IBM'

**Values:**
- BMW: 11
- GM: 90
- IBM: 91
Publish-Subscribe Systems

**Example**: stock trading system:

1. S1 has subscribed to 'IBM', with a filter indicating that it should be notified only if the stock increases by at least 25;
2. S2 and S3 have subscribed to 'IBM' and 'GM' respectively, without filter.
3. P1 is publishing the new value of 'IBM'.
4. S1 is not notified because its filter is not satisfied; S2 is not notified because it's not interested in 'IBM'; S3 is notified.
One of the main problems with publish-subscribe systems is to achieve scalability of the notification service.

- **Centralized implementations**: are the simplest, however, scalability is limited by the processing power of the machine that hosts the service.

- **Distributed implementations**: the notification service is realised as a network of distributed processes, called brokers; the brokers interact among themselves with the common aim of dispatching notifications to all interested subscribers.

  - Such a solution is scalable but is more challenging to implement; it requires complex protocols for the coordination of the various brokers and the diffusion of the information.