



# Introduction to Distributed Systems

---

*Department of*  
**Computer Science & Engineering**

notesforvtu.in

[www.cambridge.edu.in](http://www.cambridge.edu.in)



# Why Distributed Systems?

---

- **Career Opportunities:** Companies like Google, Amazon, Microsoft, and numerous startups build their infrastructure on distributed systems. Understanding these concepts can open doors to careers in software engineering, cloud computing, and big data.
- **Real-World Applications**
- **Cutting-Edge Technologies**
- **Research and Innovation**

# What is a Distributed System?

---

**A Distributed System is one in which hardware and software components located at networked computers communicate and coordinate their actions only by passing messages.**

# Components of a Distributed System

---

- **Nodes:** Individual computers or processes in the distributed system.
- **Communication Network:** Infrastructure that allows nodes to exchange messages (e.g., internet, local network).
- **Middleware:** Software that provides common services and capabilities to applications beyond what's offered by the operating system.

# Characteristics of a Distributed System

- **Concurrency:** Multiple processes execute simultaneously across different machines.
- **Scalability:** Ability to grow and handle increased loads by adding more resources.
- **Fault Tolerance:** Ability to continue functioning despite failures of some components.
- **Transparency:** The system hides the complexity of distribution from users and applications, making it seem like a single system.

# Challenges of a Distributed System

- **Heterogeneity:** The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks.
- **Openness:** The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways.
- **Security:** Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance.
- **Concurrency:** Multiple processes execute simultaneously across different machines.

# Challenges of a Distributed System

- **Transparency:** Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.
- **Failure handling:** Designing the system to handle node failures and network issues gracefully.
- **Scalability:** Ensuring that the system can handle increased load without significant performance degradation.
- **Quality of service**

# Applications of Distributed System

---

- **Cloud Computing:** Services like AWS, Azure, and Google Cloud leverage distributed systems to provide scalable and reliable services.
- **Big Data:** Frameworks like Hadoop and Apache Spark use distributed systems to process and analyze large volumes of data.
- **Blockchain:** Distributed ledger technologies like Bitcoin and Ethereum use distributed systems to maintain a secure and decentralized record of transactions.

# Examples

---

- **File Systems:** Distributed file systems like Google File System (GFS) and Hadoop Distributed File System (HDFS) store and manage data across multiple machines.
- **Databases:** Distributed databases such as Cassandra and MongoDB manage data across multiple nodes, ensuring high availability and scalability.

# Chapter 5 – Remote Invocation

---

- This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system

# Request-Reply Protocols

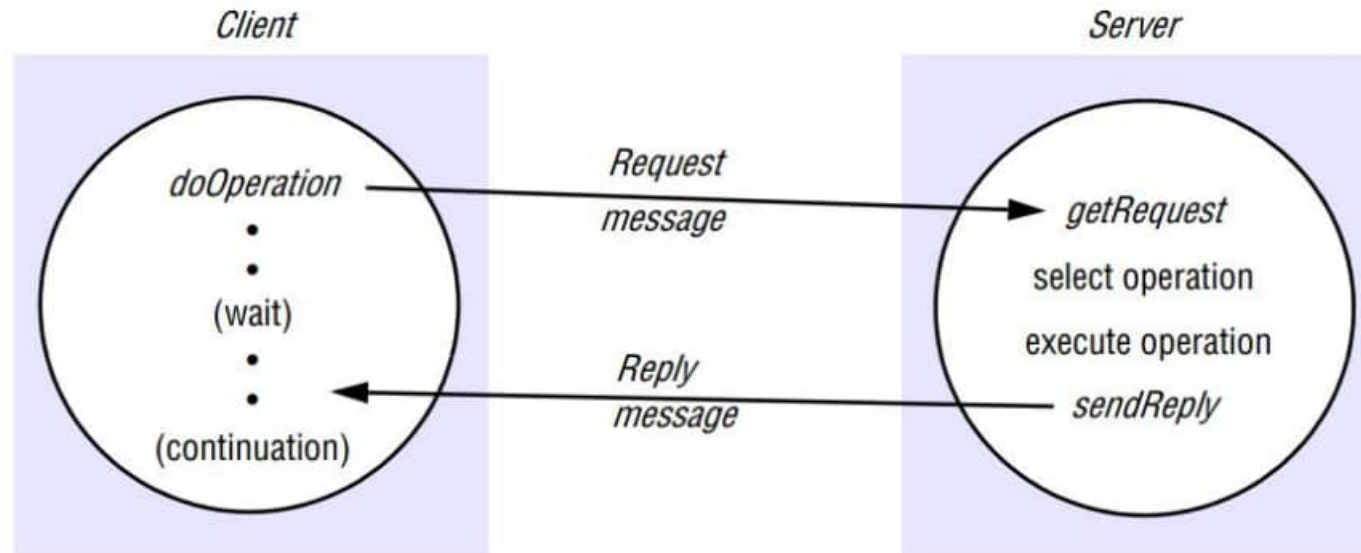
A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol. In particular:

- Acknowledgements are redundant, since requests are followed by replies.
- Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply.
- Flow control is redundant for the majority of invocations, which pass only small arguments and results.

# Request-Reply Protocols

- The protocol we describe here is based on a trio of communication primitives, `doOperation`, `getRequest` and `sendReply`, as shown in Figure.

Request-reply communication



# Request-Reply Protocols

- The protocol we describe here is based on a trio of communication primitives, `doOperation`, `getRequest` and `sendReply`, as shown in Figure.

## Operations of the request-reply protocol

*public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)*

Sends a request message to the remote server and returns the reply.

The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

*public byte[] getRequest ();*

Acquires a client request via the server port.

*public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*

Sends the reply message *reply* to the client at its Internet address and port.

# Request-Reply Protocols

- The **doOperation** method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply.
- **getRequest** is used by a server process to acquire service requests.
- **sendReply** is used to send the reply message to the client.

# Message identifiers

A message identifier consists of two parts:

1. a requestId, which is taken from an increasing sequence of integers by the sending process;
2. an identifier for the sender process, for example, its port and Internet address.

# Message identifiers

A message identifier consists of two parts:

1. a requestId, which is taken from an increasing sequence of integers by the sending process;
2. an identifier for the sender process, for example, its port and Internet address.

# Failure model of the request-reply protocol

That is:

- They suffer from omission failures.
- Messages are not guaranteed to be delivered in sender order.

# Timeouts & Discarding duplicate request messages

- The timeout may have been due to the request or reply message getting lost.
- In cases when the request message is retransmitted, the server may receive it more than once. This can lead to the server executing an operation more than once for the same request.

# Request-reply message structure

The information to be transmitted in a request message or a reply message is shown in Figure

|                 |                                  |
|-----------------|----------------------------------|
| messageType     | <i>int (0=Request, 1= Reply)</i> |
| requestId       | <i>int</i>                       |
| remoteReference | <i>RemoteRef</i>                 |
| operationId     | <i>int or Operation</i>          |
| arguments       | <i>// array of bytes</i>         |

# Failure model of the request-reply protocol

---

If the three primitives `doOperation`, `getRequest` and `sendReply` are implemented over UDP datagrams, then they suffer from the same communication failures.

That is:

- They suffer from omission failures.
- Messages are not guaranteed to be delivered in sender order.

# Timeouts

---

- There are various options as to what doOperation can do after a timeout.
- The simplest option is to return immediately from doOperation with an indication to the client that the doOperation has failed.
- To compensate for the possibility of lost messages, doOperation sends the request message repeatedly until either it gets a reply or it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages.

# Discarding duplicate request messages

---

- In cases when the request message is retransmitted, the server may receive it more than once.
- the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates.

# Styles of exchange protocols

---

Three protocols, that produce differing behaviors in the presence of communication failures are used for implementing various types of request behavior. They were originally identified by Spector [1982]:

- the request (R) protocol;
- the request-reply (RR) protocol;
- the request-reply-acknowledge reply (RRA) protocol.

# Styles of exchange protocols

## RPC exchange protocols

| <i>Name</i> | <i>Messages sent by</i> |               |                          |
|-------------|-------------------------|---------------|--------------------------|
|             | <i>Client</i>           | <i>Server</i> | <i>Client</i>            |
| R           | <i>Request</i>          |               |                          |
| RR          | <i>Request</i>          | <i>Reply</i>  |                          |
| RRA         | <i>Request</i>          | <i>Reply</i>  | <i>Acknowledge reply</i> |

# Remote Procedure Call (RPC)

---

- In RPC, procedures on remote machines can be called as if they are procedures in the local address space.
- The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.
- This concept was first introduced by Birrell and Nelson [1984] and paved the way for many of the developments in distributed systems programming.

# Design issues for RPC

---

Before looking at the implementation of RPC systems, we look at three issues that are important in understanding this concept:

- the style of programming promoted by RPC – programming with interfaces;
- the call semantics associated with RPC;
- the key issue of transparency and how it relates to remote procedure calls.

# Interfaces in distributed systems

---

- In a distributed program, the modules can run in separate processes.
- In the client-server model, in particular, each server provides a set of procedures that are available for use by clients.
- The term service interface is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

# CORBA IDL example

```
// In file Person.idl  
struct Person {  
    string name;  
    string place;  
    long year;  
};  
interface PersonList {  
    readonly attribute string listname;  
    void addPerson(in Person p) ;  
    void getPerson(in string name, out Person p);  
    long number();  
};
```

# CORBA(Common Object Request Broker Architecture)

- It is a standard defined by the Object Management Group (OMG) that allows pieces of programs, known as objects, to communicate with one another regardless of where they are located (locally or across a network) and regardless of the programming language used to write them.
- CORBA achieves this through its middleware framework, enabling interoperability between distributed systems.

# Idempotency

---

- In Distributed Systems: Idempotency is crucial in systems where operations might be retried due to network failures. If an operation is idempotent, it can safely be retried without concern for unintended side effects.

# RPC call semantics

---

The main choices are:

**Retry request message:** Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

**Duplicate filtering:** Controls when retransmissions are used and whether to filter out duplicate requests at the server.

**Retransmission of results:** Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

# RPC call semantics

---

The choices of RPC invocation semantics are defined as follows:

- Maybe semantics
- At-least-once semantics
- At-most-once semantics

# Maybe semantics

---

With maybe semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

- omission failures if the request or result message is lost;
- crash failures when the server containing the remote operation fails.

# At-least-once semantics

---

With at-least-once semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.

At-least-once semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message.

# At-least-once semantics

---

At-least-once semantics can suffer from the following types of failure:

- crash failures when the server containing the remote procedure fails;
- arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned.

# Idempotent operation

---

An Idempotent operation is one that can be performed repeatedly with the same effect as if it had been performed exactly once.

Non-idempotent operations can have the wrong effect if they are performed more than once.

# At-most-once semantics

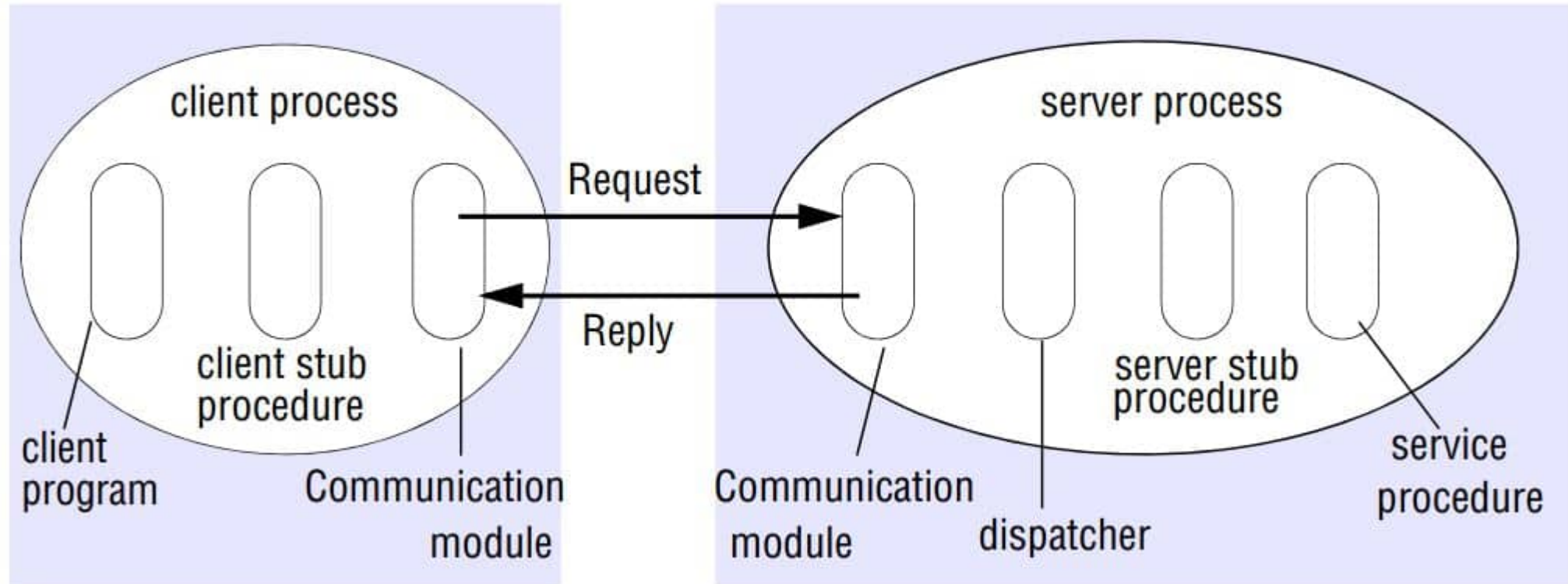
---

With at-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

Sun RPC provides at-least-once call semantics.

# Implementation of RPC

## Role of client and server stub procedures in RPC



# Remote Method Invocation (RMI)

---

- Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects.
- In RMI, a calling object can invoke a method in a potentially remote object.
- As with RPC, the underlying details are generally hidden from the user.

# Remote Method Invocation (RMI)

The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach.
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

# Remote Method Invocation (RMI)

The advantages of RMI are as follows:

- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

# Design issues for RMI

---

RMI shares the same design issues as RPC in terms of programming with interfaces, call semantics and level of transparency.

The key added design issue relates to the object model and, in particular, achieving the transition from objects to distributed objects.

# The object model

Object references

Interfaces

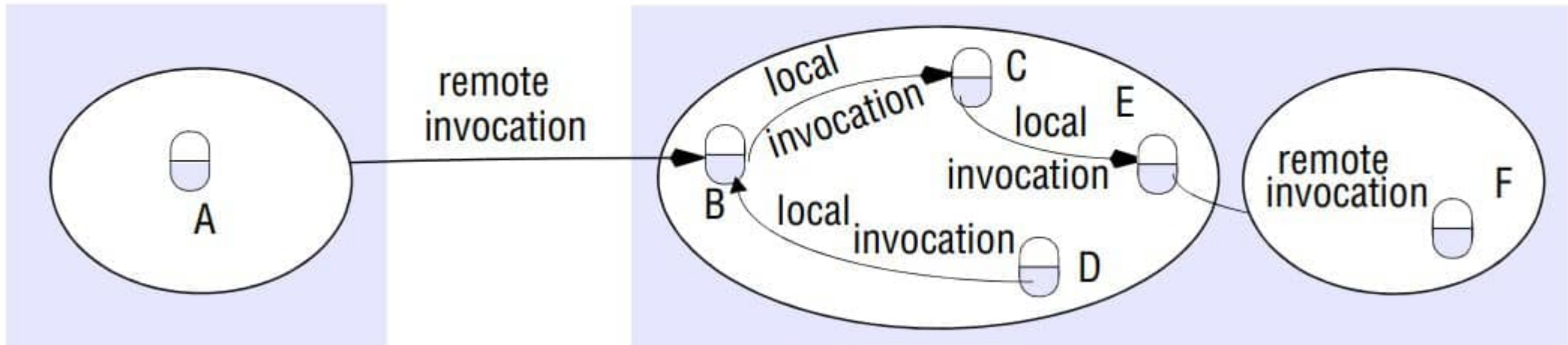
Actions

Exceptions

Garbage collection

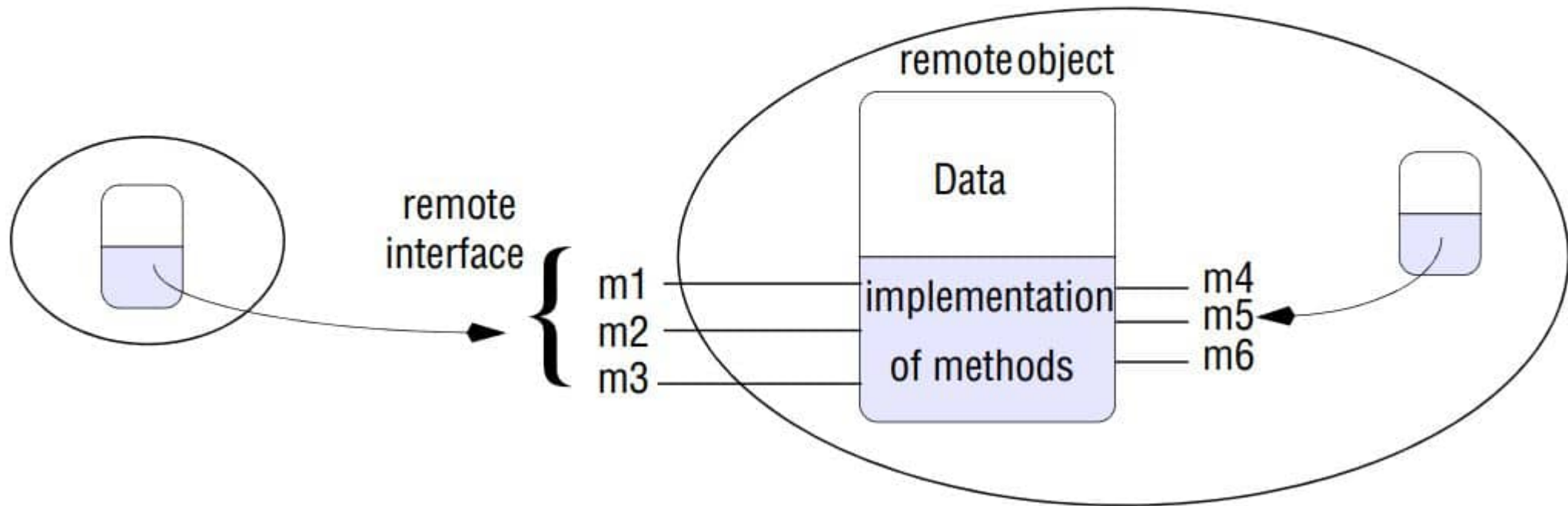
# The distributed object model

## Remote and local method invocations



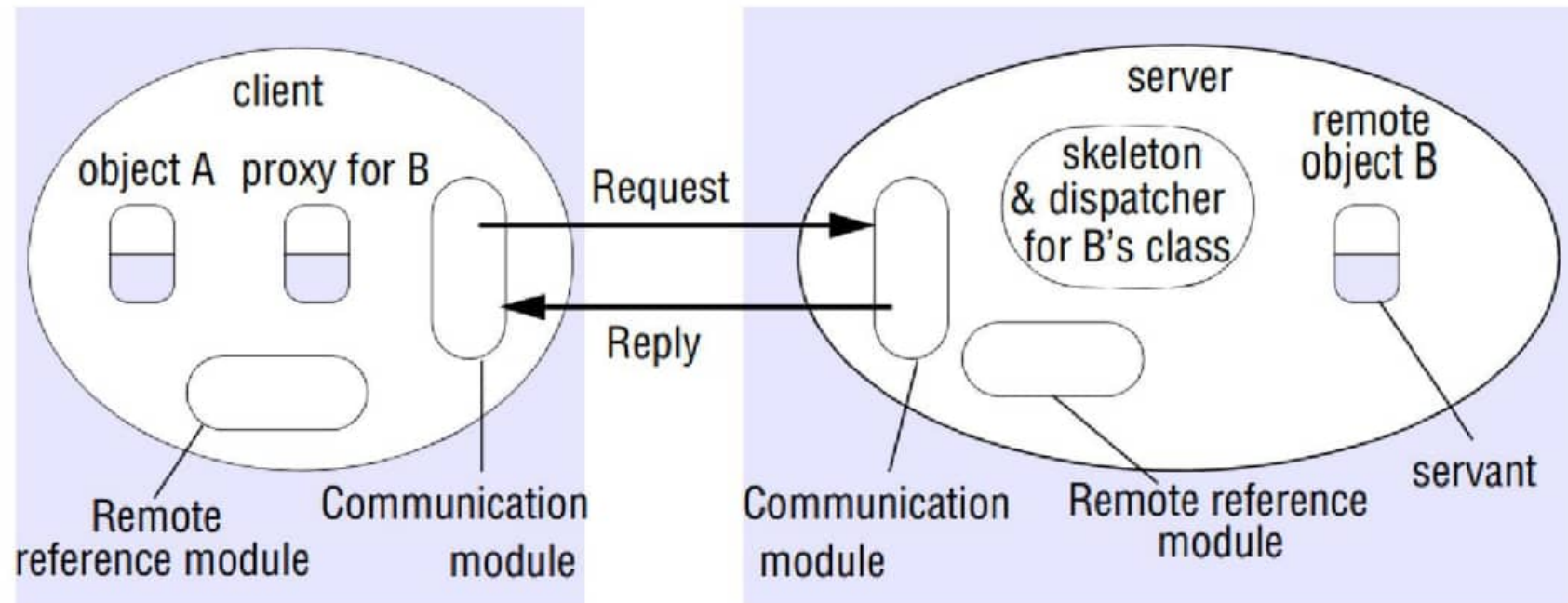
# The distributed object model

A remote object and its remote interface



# Implementation of RMI

The role of proxy and skeleton in remote method invocation



# Implementation of RMI

**Communication module** : The two cooperating communication modules carry out the request-reply protocol, which transmits request and reply messages between the client and server.

**Remote reference module**: A remote reference module is responsible for translating between local and remote object references and for creating remote object references. (**remote object table**)

**Servants**: A servant is an instance of a class that provides the body of a remote object.

# Remote reference module

---

To support its responsibilities, the remote reference module in each process has a **remote object table** that records the correspondence between local object references in that process and remote object references (which are system-wide).

The table includes:

- An entry for all the remote objects held by the process.
- An entry for each local proxy.

# The RMI software

---

This consists of a layer of software between the application-level objects and the communication and remote reference modules.

The roles of the middleware objects are as follows:

- **Proxy** : The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker
- **Dispatcher** : The dispatcher receives request messages from the communication module.
- **Skeleton** : A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the servant.

# Distributed garbage collection

---

The aim of a distributed garbage collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered.