Remote Invocation

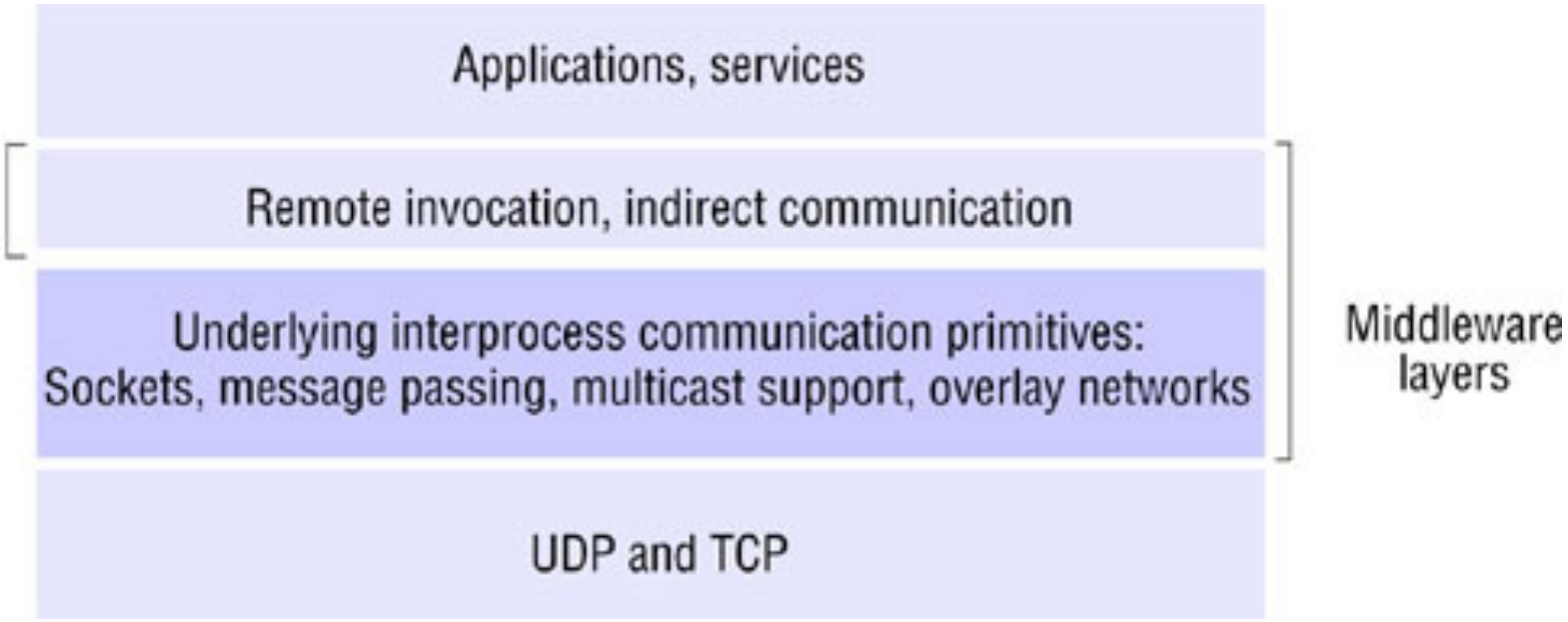
CS432: Distributed Systems
Spring 2017

Reading

- Coulouris (5th Edition): 5.1, 5.3, 5.4, 5.5
- Tanenbaum (2nd Edition): 4.2

Middleware Layers

This Lecture



Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012

Outline

- **Introduction**
- Request-Reply Protocols
- Remote Procedure Call
- Remote Method Invocation
- Case Study: Java RMI

Communication Paradigms

- **Interprocess Communication**: refers to the relatively low-level support for communication between processes in distributed systems (e.g., message-passing primitives, direct access to the API offered by Internet protocols)
- **Remote invocation**: represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method
- **Indirect communication**: such as group communication, publish-subscribe, distributed shared memory

Remote Invocation

- Request-reply protocols: a pattern imposed on an underlying message-passing service to support client-server computing
- Remote procedure call:
  - Procedures in processes on remote computers can be called as if they are procedures in the local address space
  - RPC system hides aspects of distribution, including the encoding and decoding of parameters and results, and passing of messages
  - Supports client-server computing: servers offering a set of operations through a service interface and clients calling these operations
  - Offer access and location transparency
- Remote method invocation: resembles remote procedure calls but in a world of distributed objects

Outline

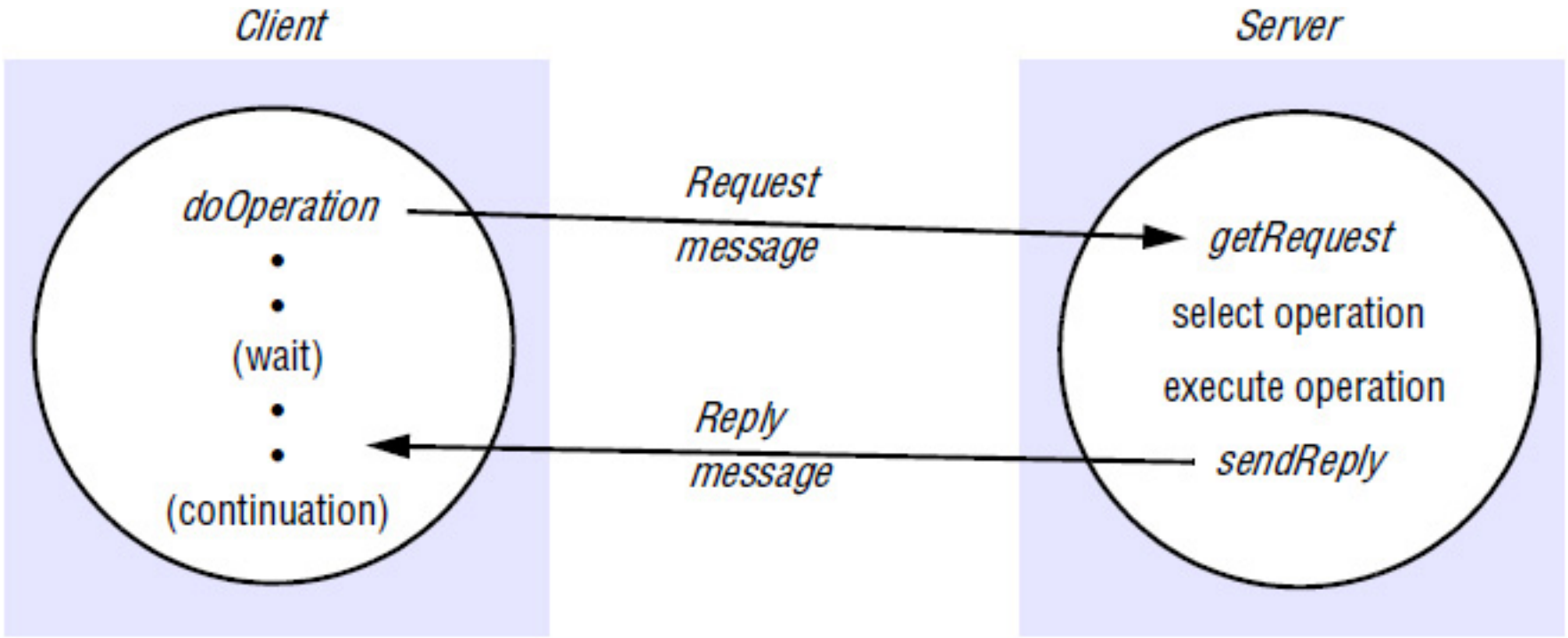
- Introduction
- **Request-Reply Protocols**
- Remote Procedure Call
- Remote Method Invocation
- Case Study: Java RMI

Request-Reply Protocols

- It is designed to support the roles and message exchanges in typical client-server interactions
- Synchronous: because the client process blocks until the reply arrives from the server
- Reliable: because the reply from the server is effectively an acknowledgement to the client
- Asynchronous communication is optional

Request-Reply Communications

- Communication primitives: doOperation, getRequest, and sendReply



Communication Primitives: doOperation

- public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
  - Sends a request message to the remote server and returns the reply
  - Arguments: remote server, ID of the operation to be invoked, and the arguments of that operation
  - RemoteRef: represents references for remote servers including its IP address and port number
  - doOperation invokes receive to get a reply message, from which it extracts the result and returns it to the caller
  - doOperation is blocked until the server performs the requested operation and transmits a reply message to the client process

Communication Primitives: getRequest and sendReply

- public byte[] getRequest ()
  - Used by a server process to acquire service requests via 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
  - When the reply message is received by the client the original doOperation is unblocked and execution of the client program continues

Failure Modes

- Timeout:
  - Return immediately indicating that request failed
  - Retry by sending the request repeatedly
- Duplicate request messages:
  - The server recognizes successive messages (from the same client) with the same request identifier and filters out duplicates
  - If the reply was not sent before, the server sends it when it finishes
  - If reply was already sent: recompute the reply or return a duplicate reply from history

Outline

- Introduction
- Request-Reply Protocols
- **Remote Procedure Call**
- Remote Method Invocation
- Case Study: Java RMI

Remote Procedure Call

- Allowing programs to call procedures located on other machines.
  - When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B
  - Information can be transported from the caller to the callee in the parameters and can come back in the procedure result
- The underlying RPC system hides important aspects of distribution:
  - The encoding and decoding of parameters and results
  - The passing of messages
  - The preserving of the required semantics for the procedure call

Design Issues for RPC

- 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

Programming with Interfaces

- Service interface refers to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures
- Benefits to programming with interfaces
  - Focus on abstraction offered by the service and hide the implementation details
  - Managing heterogeneity in distributed systems. The programmer does not need to know programming language or underlying platform
  - Support for software evolution as long as interface does not change

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();
};
```

RPC Call Semantics

- Options:
  - Retry request messages: retransmit the message until a reply is received or assume server failed?
  - Filtering duplicates: whether the server filters duplicates or not
  - Retransmission of results: keep a history of results at the server to avoid recomputing?
- RPC supported invocation semantics:
  - Maybe semantics
  - At-least-once semantics
  - At-most-once semantics

Maybe Semantics

- The remote procedure call may be executed once or not at all
- No fault tolerance measures are applied and suffers from these failures:
  - Omission failures if the request or result message is lost: uncertain if the request or reply message is lots; uncertain if execution was performed at server
  - Crash failures when the server containing the remote operation fails (before or after execution)
- Useful only for applications in which occasional failed calls are acceptable

At-Least-Once Semantics

- The invoker receives either a result (the procedure was executed at least once) or an exception (no result was received)
- Achieved by retransmission of request messages (masks omission failure)
- Suffers from these failures:
  - Crash failures when the server containing the remote operation fails (before or after execution)
  - Arbitrary failures: re-execution at the server can cause wrong values to be stored or returned
- Suitable for idempotent operations at the server

At-Most-Once Semantics

- The caller receives either a result (procedure was executed exactly once) or an exception (no result was received — procedure was executed exactly once or not at all)
- Omission, crash, and arbitrary failures are avoided

Summary of Fault Tolerance Measures

Fault tolerance measures			Call semantics
Retransmit request message	**Duplicate filtering**	**Re-execute procedure or retransmit reply**	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Transparency

- Objective: remote procedure calls look like local procedure calls
- Features:
  - Calls to marshalling and message-passing procedures are hidden from the caller
  - Retransmission of requests after timeout is transparent from the caller
- Location and access transparency achieved
- Differences:
  - Remote procedure calls are more vulnerable to failure than local ones
  - Latency of a remote procedure call is several orders of magnitude greater than that of a local one
  - Different style of parameter passing (no call by reference — no addresses as parameters)

Outline

- Introduction
- Request-Reply Protocols
- Remote Procedure Call
- **Remote Method Invocation**
- Case Study: Java RMI

Remote Method Invocation

- Similar to RPC, but extends to distributed objects
- A calling object can invoke a method in a potentially remote object
- Similarities
  - Support programming with interfaces
  - Offers range of call semantics: at-least-once, at-most-once
  - Similar level of transparency
- Differences
  - Object oriented programming
  - Object identity concept: all objects in an RMI-based system have unique object references, which can be passed as parameters

Design Issues for RMI

- Object model:
  - Set of data and set of methods
  - Communication by invoking methods (passing arguments and receiving results)
- Distributed objects
- Distributed object model
- Actions in a distributed object system

Object Model

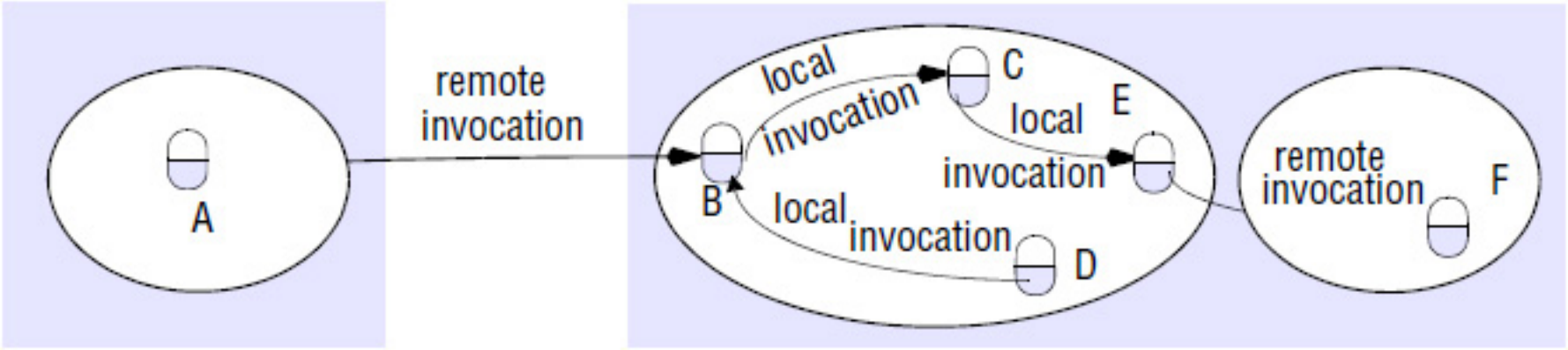
- Object references
  - To invoke a method in an object —> object reference + method name + arguments
- Interfaces
  - Example: Java interface (signatures of methods, no implementation)
- Actions
  - The state of the receiver may be changed
  - A new object may be instantiated (e.g., constructor)
  - Further invocations on methods in other objects
- Exceptions
- Garbage collection (Java vs C++)

Distributed Objects

- State of object: values of its instance variables
- State of program: partitioned into parts, each represents an object
- Architecture models: client/server, replicated objects, migration of objects
- Possibility: copy object locally and directly access it if implementation available
- Concurrent remote invocations to an object methods is possible —> responsibility of the object to protect its state (e.g. using synchronization)

Distributed Object Models

- Each process has two types of objects:
  - Objects that can receive both local and remote invocations
  - Objects that can only receive local invocations
- Note: method invocation between objects from different processes is considered remote invocation



Distributed Object Models: Remote Object References

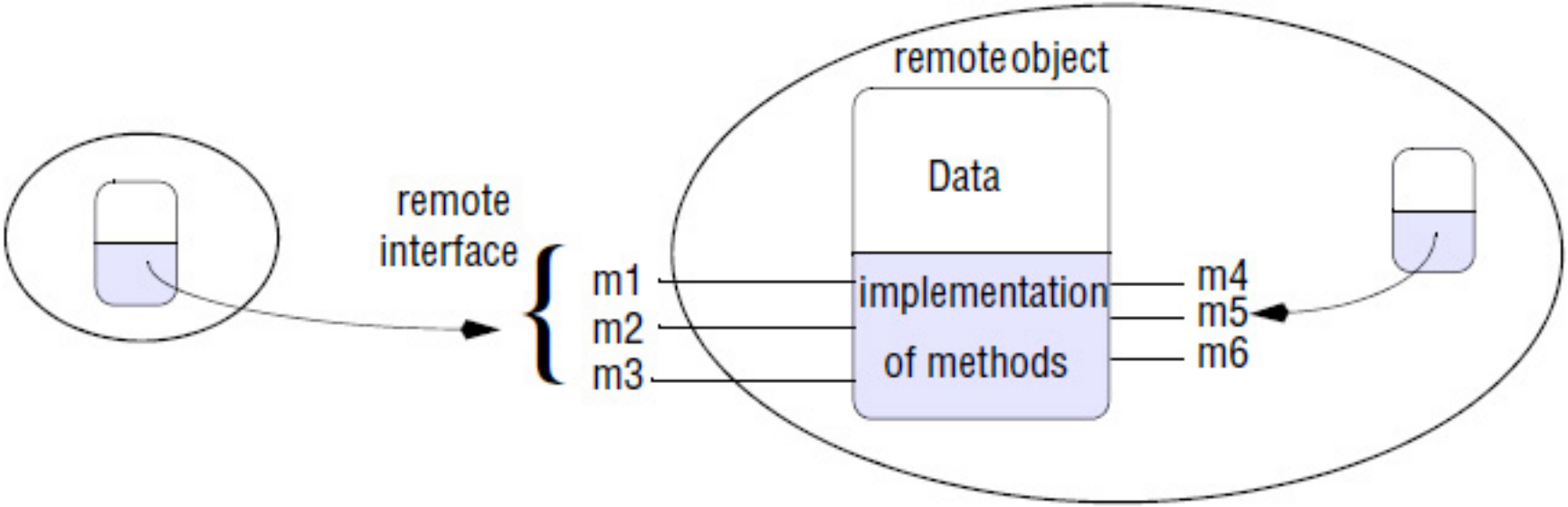
- Other objects can invoke the methods of a remote object if they have access to its remote object reference
- A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object
- Representation:

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

- Remote object reference (1) used when invoking a remote object; (2) used as arguments and results of methods

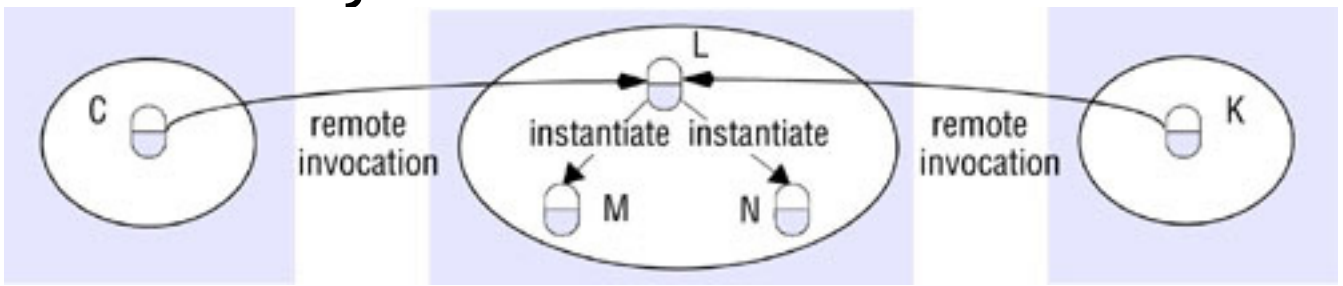
Distributed Object Models: Remote Interfaces

- Remote interfaces: Every remote object has a remote interface that specifies which of its methods can be invoked remotely
- Objects in other processes can invoke only the methods that belong to its remote interface



Actions in Distributed Object System

- Challenge: an action may result in further invocations on methods in other objects, which may be located in different processes or different computers
  - —> remote reference of the object must be available to the invoker
  - Methods that instantiate objects to be accessed by RMI

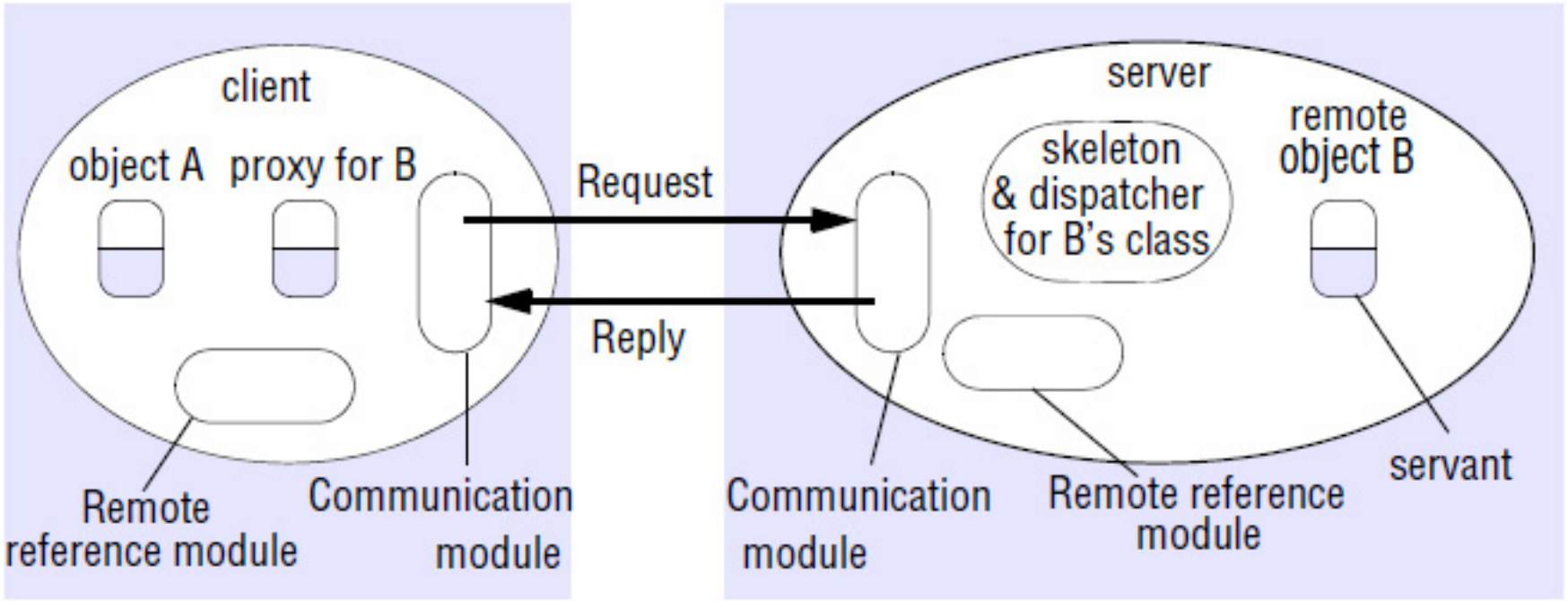


Actions in Distributed Object System: Garbage Collection and Exceptions

- Garbage collection:
  - Goal: garbage collection of remote objects
  - Module that performs distributed garbage collection
- Exceptions:
  - New issues to handle: Failures due to the invoked object being in a different process or computer from the invoker
    - Examples: remote object crashed or busy to reply, invocation or result message is lost
  - Remote method invocation should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked

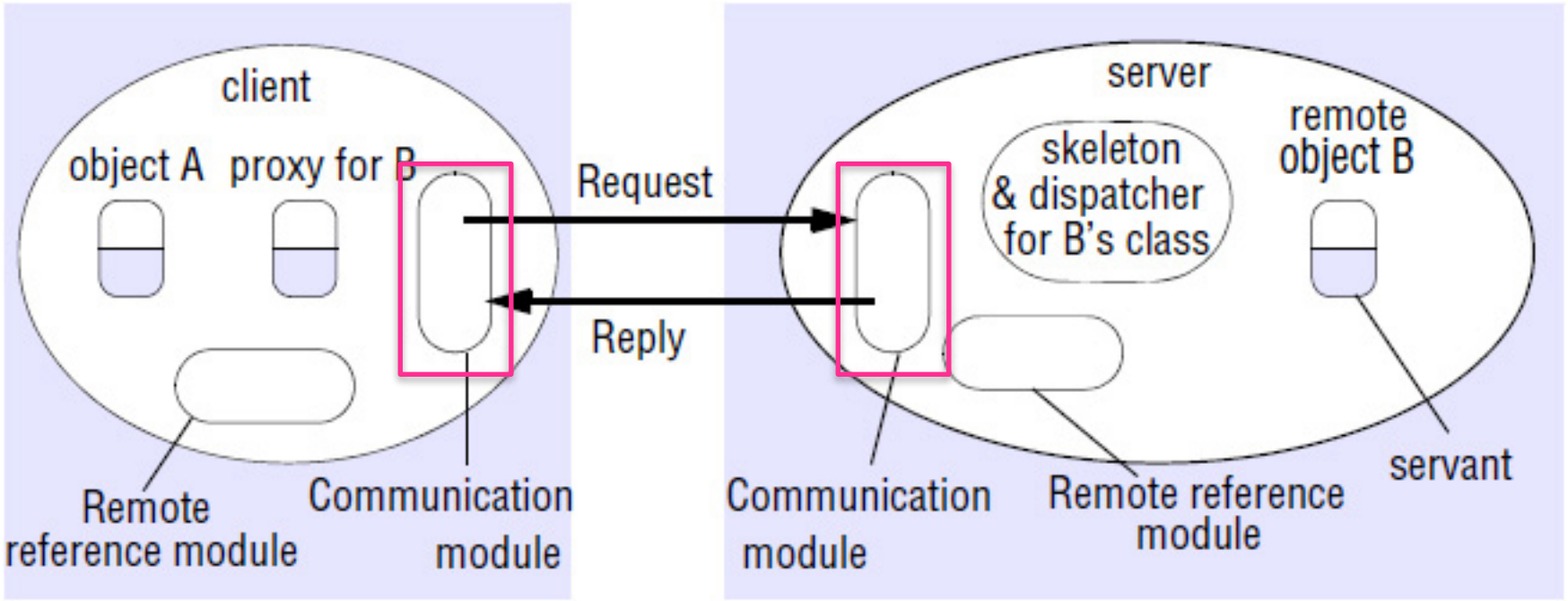
Implementation of RMI

- Object A invokes remote object B



Implementation of RMI

- Communication Module

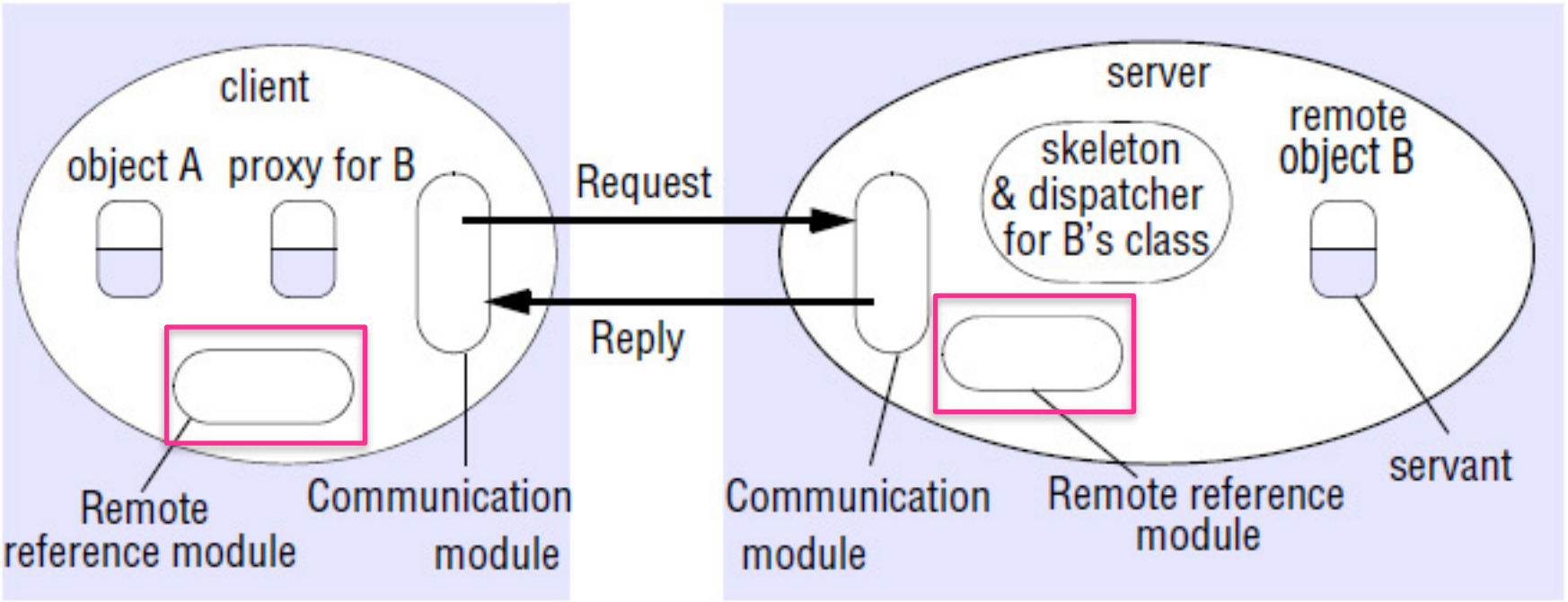


Communication Module

- Two cooperating communication modules carry out the request-reply protocol
- Contents of request and reply messages: message type, requestId, and the remote reference of the object to be invoked

Implementation of RMI

- Remote Reference Module

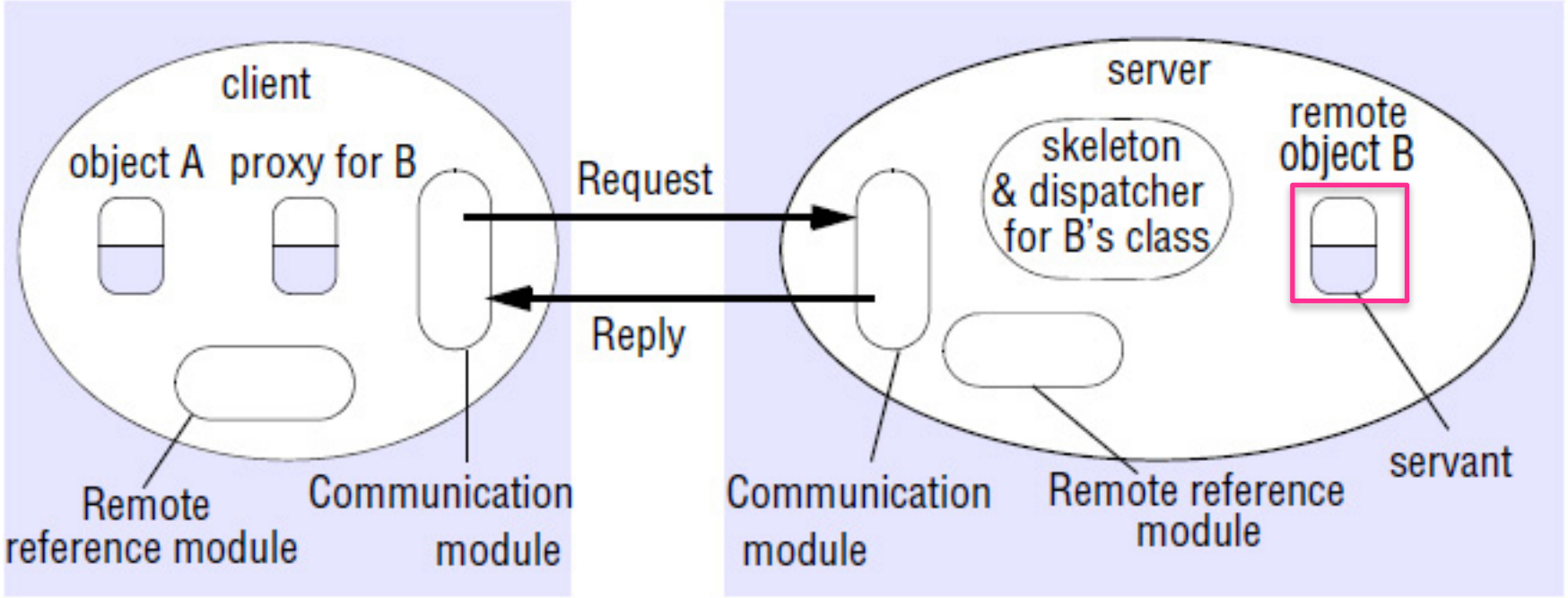


Remote Reference Module

- Responsible for translating between local and remote object references and for creating remote object references
- Maintains a remote object table:
  - An entry for all the remote objects held by the process
  - An entry for each local proxy (discussed later)
- Actions by the remote reference module:
  - When a remote object is to be passed as an argument or a result for the first time, create a remote object reference and add it to the table
  - When a remote object reference arrives in a request or reply message, it is inquired about the object. If not in the table, RMI software creates new proxy and is added to the table

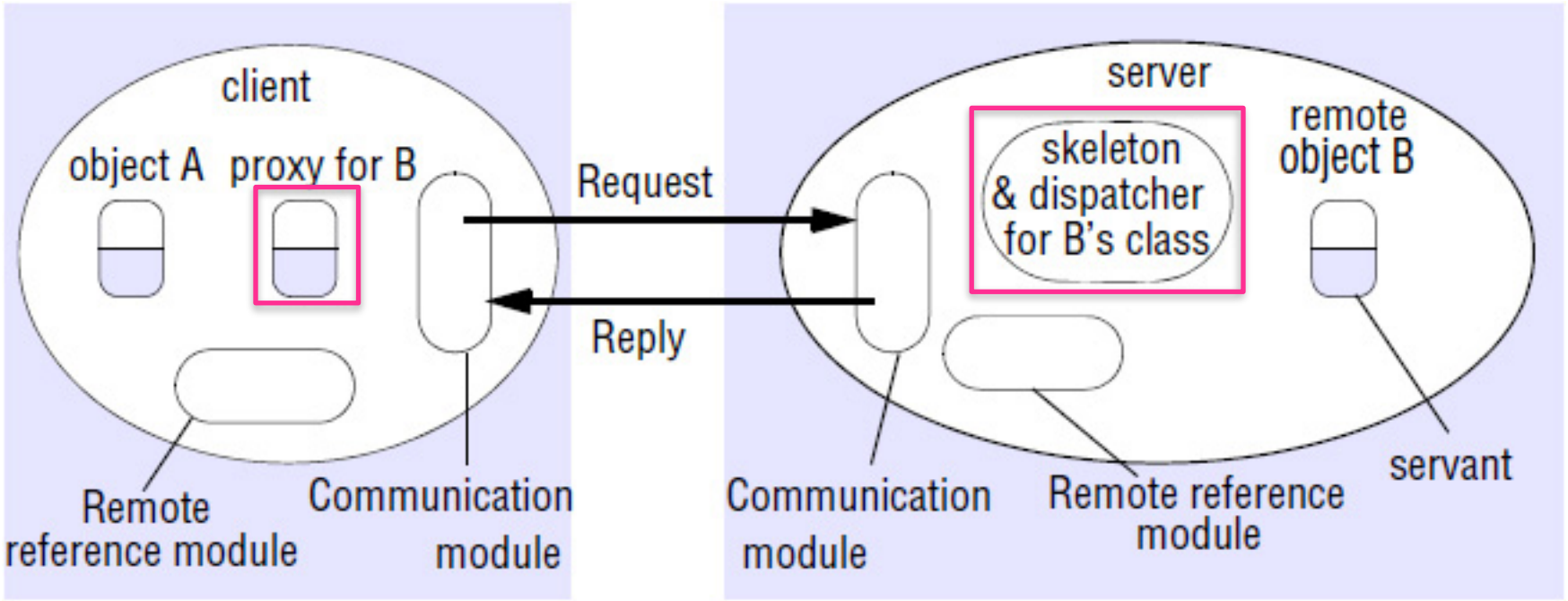
Implementation of RMI

- Servants: an instance of a class that provides the body of a remote object, handles remote requests, created when object is instantiated, garbage collected



Implementation of RMI

- RMI Software



RMI Software

- A layer of software between the application-level objects and the communication and remote reference modules
- Proxy (at client): its role is to make remote method invocation transparent to clients by behaving like a local object to the invoker
- Dispatcher (at server): receives request messages from the communication module and passes it to the correct method using the operationid
- Skeleton: implements the methods in the remote interface
  - unmarshals the arguments in the request message
  - invokes the corresponding method in the servant
  - waits for the invocation to complete and then marshals the result (with any exceptions) to the sending proxy

Distributed Garbage Collection

- Goal:
  - 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,
  - as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered
- Steps:
  - Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects
  - When a client C first receives a remote reference to a particular remote object, B, it makes an addRef(B) invocation to the server of that remote object and then creates a proxy; the server adds C to B.holders
  - When a client C's garbage collector notices that a proxy for remote object B is no longer reachable, it makes a removeRef(B) invocation to the corresponding server and then deletes the proxy; the server removes C from B.holders
  - When B.holders is empty, the server's local garbage collector will reclaim the space occupied by B unless there are any local holders.

Thank You