Lecture

CMPU4021 Distributed Systems

Remote Invocation

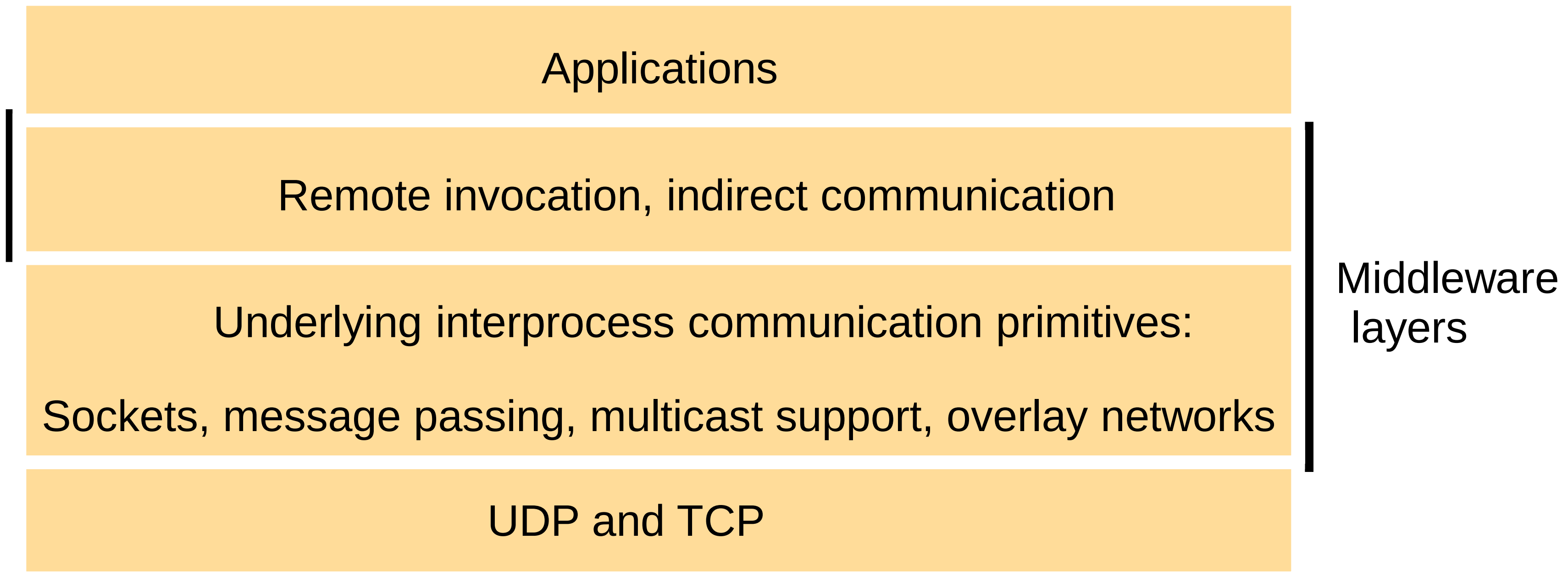
Based on Chapter 5: Coulouris, Dollimore and Kindberg,
Distributed Systems: Concepts and Design

Introduction

- Communication among distributed objects via RMI
  - Recipients of remote invocations are remote objects, which implement remote interfaces for communication
- Reliability
  - Either one or both the invoker and invoked can fail, and status of communication is supported by the interface (e.g., notification on failures, reply generation, parameter processing – marshalling/unmarshalling)
    - *Marshalling* – the process of taking a collection of data items and assembling them into a form suitable transmission in a message (external data representation)
    - *Unmarshalling* – disassembling them on arrival to produce an equivalent collection of data items at the destination.
- Local invocations target local objects, and remote invocations target remote objects

RMI in Middleware Layers

- A suite of API software that uses underlying processes and communication (message passing) protocols to provide its abstract protocol – simple RMI request-reply protocol


Applications
Remote invocation, indirect communication
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
UDP and TCP
Middleware layers

Interfaces

- Interfaces hide the details of modules providing the service; and access to module variables is only indirectly via ‘getter’ and ‘setter’ methods / mechanisms associated with the interfaces
  - e.g., call by value/reference for local calls through pointers vs. input, output, and input paradigms in RMI through message-data and objects

- *Service interfaces*
  - client-server model, specification of the procedures offered by a server
    - defining the types of input and output arguments

- *Remote interfaces*
  - distributed object model, specifies the methods of an object that are available for invocation by objects in other processes
    - defining the types of the input and output arguments of each of them.

Interface definition languages (IDLs)

IDL

- Provides a ‘generic’ template of interfaces for objects in different languages to perform remote invocation among each other
  - E.g., CORBA IDL, Java RMI

- Provides a notation for defining interfaces in which each of the parameters of a method may be described as for *input* or output in addition to having its type specified.

The distributed object model

- RMI
  - invocations between objects in different processes (either on same or different computers) is *remote*. Invocations within the same process are *local*
- Each process contains objects, some of which can receive remote invocations, other only local invocations
- Those that can receive remote invocations are called remote objects
- Objects need to know the remote object reference of an object in another process in order to invoke its methods. How do they get it?
  - The remote interface specifies which methods can be invoked remotely

Distributed objects

- State of an object
  - current values of its variables
- State of an object (in OO context): depends of how the objects are partitioned or distributed in the program space and the programming model
- CS: objects reside with client and server processes or computers
- RMI: method invocations and associated messages are exchanged by c/s depending on where the objects reside
- Distributing objects require ‘encapsulation’ for effective, secure, protected object state
  - All data are accessed directly by their respective methods, and requires authorization.
- Heterogeneity (of data types) is supported via interface-methods

Distributed Objects vs Non-Distributed

Object instantiation

- Cannot instantiate objects directly – Impossible to call new.
- Must go through a factory method - a method that creates servants,
- A factory object is an object with factory methods.
- Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.
  - Such methods are called factory methods, although they are really just normal methods.

Distributed Objects vs Non-Distributed Objects differences

Method invocation:

- Open socket – send method identification data, pass serialised version of parameter objects.
- If parameter object is itself a remotely capable object, we do not pass a serialised version of the object, rather we pass a proxy for that object.
- Can only use message passing.

Parameter passing:

- RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference.

Memory allocation / deallocation:

- Distributed garbage collection

RMI: Objects

- Instances of classes
- Two types of objects
  - Non-remote object: As usual
  - Remote objects: Methods can be invoked remotely (RMI)
- Also: Proxy (*stub*) objects
  - Represent a remote object in a process which wishes to remotely invoke a method in that object

RMI: Object References

- Refer to location of object
- Two types
  - Local references: As usual –memory location in process memory space
  - Remote references: Reference to remote object
- Encapsulated in stub object

RMI: Memory Allocation / Deallocation

- Remove unused objects from memory
- Garbage collection
  - Two types
    - Local garbage collection: As usual –garbage collector counts number of local references and deletes object when this reaches 0
    - Distributed garbage collection: Made difficult by requirement to identify when object is no longer being referenced

Distributed Garbage Collection

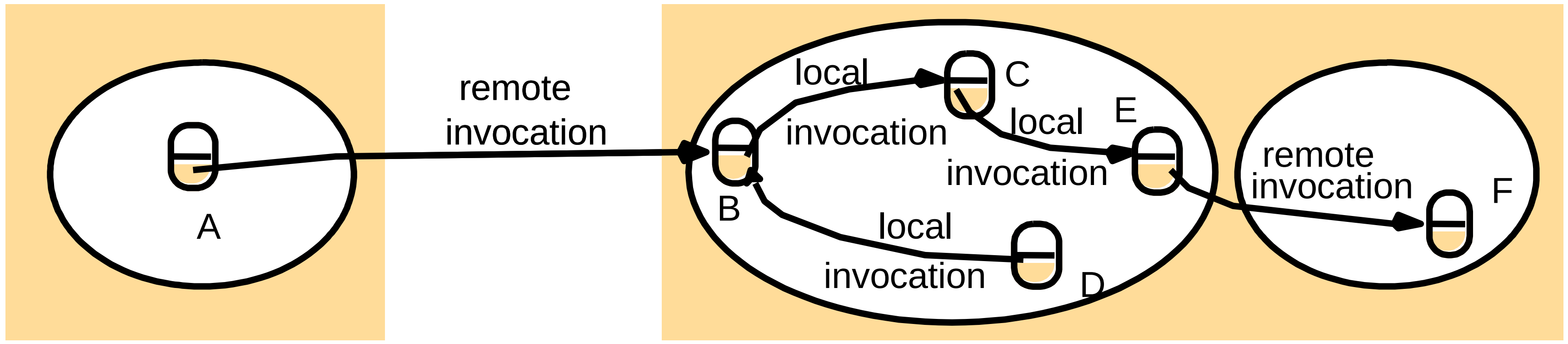
- Where any process includes remote objects, then it is equipped wlth both
  - Local garbage collector
  - Distributed garbage collector
- For any remote object O
  - O.holders is a list of all the processes that have a remote reference to that object i.e. got a stub for it
- When a client C receives a remote reference for O it makes an addRef call to O's garbage collector
  - resulting in its being added to O.holders
- When C's local garbage collector attempts to delete the stub object for O, it calls removeRef on O's garbage collector, resulting
  - it its being removed from O.holders
- When O.holders is empty, O can be deleted

Distributed Garbage Collection

- Possible difficulties
  - removeRef and addRef sent at same time from different processes
- Possibility that O would be deleted, even though a client *thinks* it has a reference
- Incorporate a delay / temporary reference to solve
  - addRef goes missing
- Client must detect, exception returned
  - removeRef message goes missing / not sent
- Time based leases are allocated for objects

The distributed object model: Remote and local method invocations

- Objects receiving remote invocations (service objects) are remote objects, e.g., B and F
- Object references are required for invocation, e.g., C must have E's reference or B must have A's reference
- B and F must have remote interfaces (of their accessible methods)

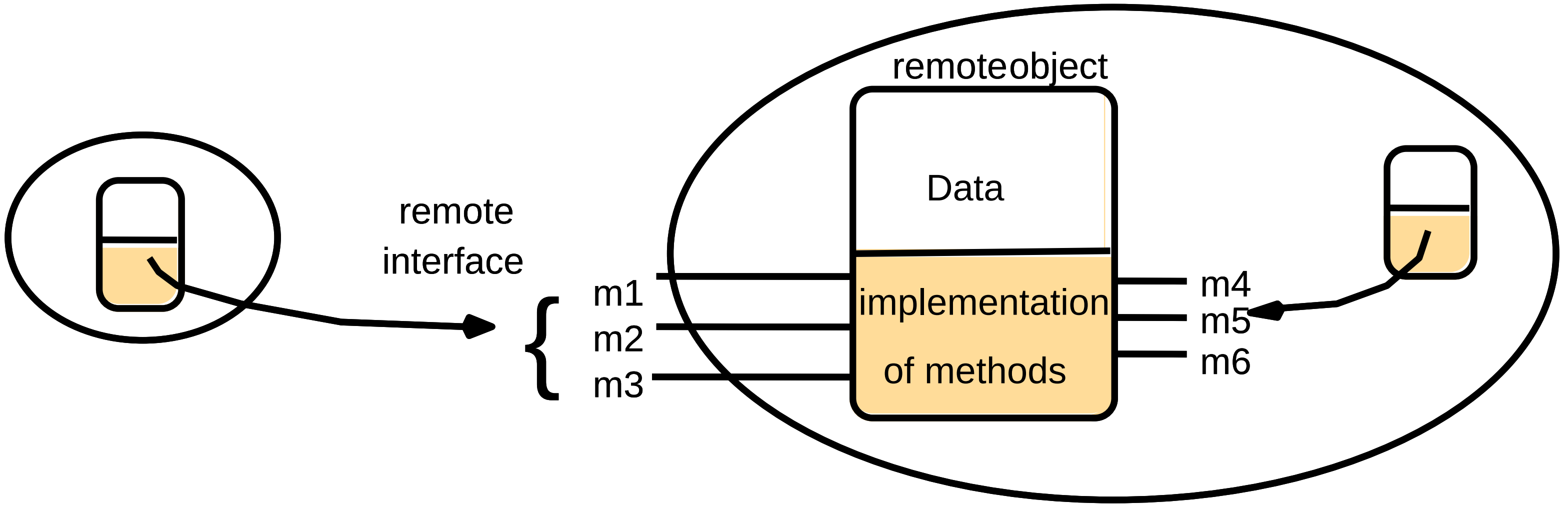


The distributed object model

- Remote object references
  - An unique identifier of a remote object, used throughout a distributed system
  - The remote object reference (including the ‘interface’ list of methods) can be passed as arguments or results in rmi.

The distributed object model:Remote interfaces

- Remote objects have a class that implement remote methods (as public).
- In Java, a remote interface class extends the `Remote` interface
- Local objects can access methods in an interface plus methods implemented by remote objects (Remote interfaces can't be constructed – no constructors)



Invocation Semantics

- Unreliable network
  - For all request –reply protocols, messages may get lost
- Solutions for lost / retransmitted messages
  - Retry request
  - Filter Duplicates
  - Retransmit results

Invocation Semantics

- In local OO system
  - all methods are invoked exactly once per request –guaranteed – unless whole process fails
- In distributed object system, we need to know what has happened if we do not hear result from remote object i.e. did the request go missing, did the response go missing
- 3 different types of guarantee (invocation semantics) may be provided
  - could be implemented in a middleware platform intended to support remote method invocations:
    - *Maybe*
    - *At-Least-Once*
    - *At-Most-Once*

Maybe Invocation Semantics

- If the invoker cannot tell whether a remote method has been invoked or not

- Very inexpensive, but only useful if the system can tolerate occasional failed invocations

At-Least-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked at least once
- Achieved by resending requests to mask omission failure
- Only useful if the operations are idempotent (x = 10, rather than x = x + 10)
- Inexpensive on server

At-Most-Once Invocation Semantics

- If the invoker receives a result, then it is guaranteed that the method was invoked only once
- If no result is received, then the method was executed either never or once
- Achieved by resending requests, and storing and resending responses
- More expensive on a server / remote object, which must maintain results and recognise duplicate messages

Invocation semantics: failure model

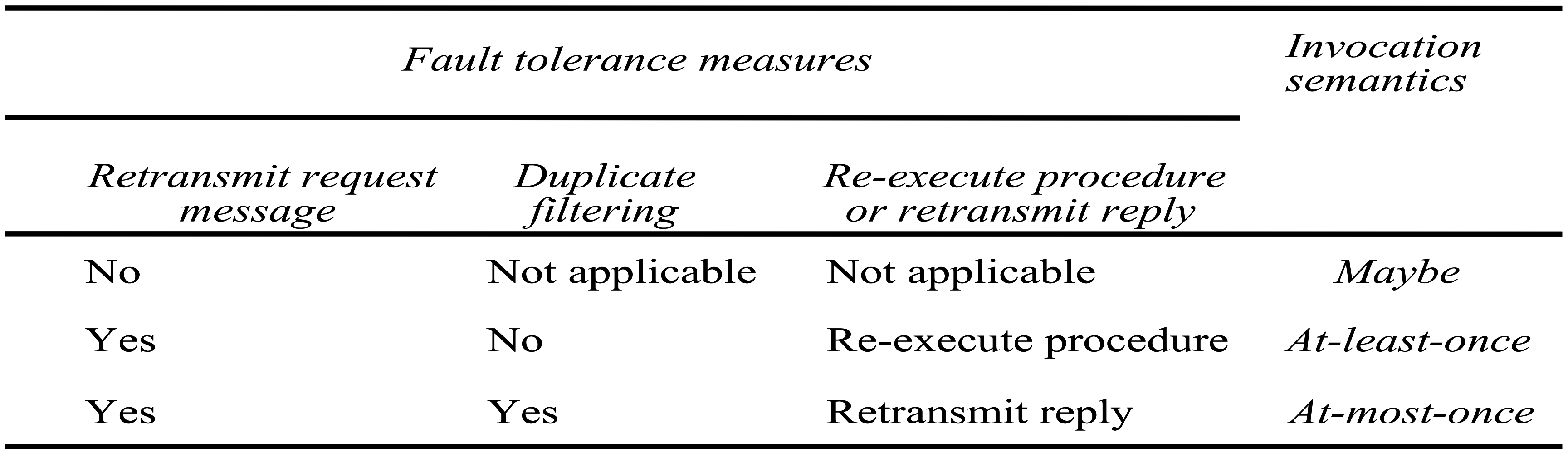
- Maybe, At-least-once and At-most-once
  - can suffer from crash failures when the server containing the remote object fails.

- *Maybe*
  - if no reply, the client does not know if method was executed or not
    - omission failures - if the invocation or result message is lost

- *At-least-once*
  - the client gets a result (and the method was executed at least once) or an exception (no result)
    - arbitrary failures. If the invocation message is retransmitted, the remote object may execute the method more than once, possibly causing wrong values to be stored or returned.
    - if *idempotent* operations are used, arbitrary failures will not occur

- At-most-once
  - the client gets a result (and the method was executed exactly once) or an exception (instead of a result, in which case, the method was executed once or not at all)

Invocation semantics: failure model



Fault tolerance measures			*Invocation 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*

Design Issues of RMI

- Local invocations have
  - at-most-once, or
  - exactly-once semantics

- Distributed RMI, the alternative invocation semantics are:
  - *Retry request message* – retransmit until reply is received or on server failure – at-least-once semantics;

  - *Duplicate message filtering* – discard duplicates at server (using seq #s or ReqID);

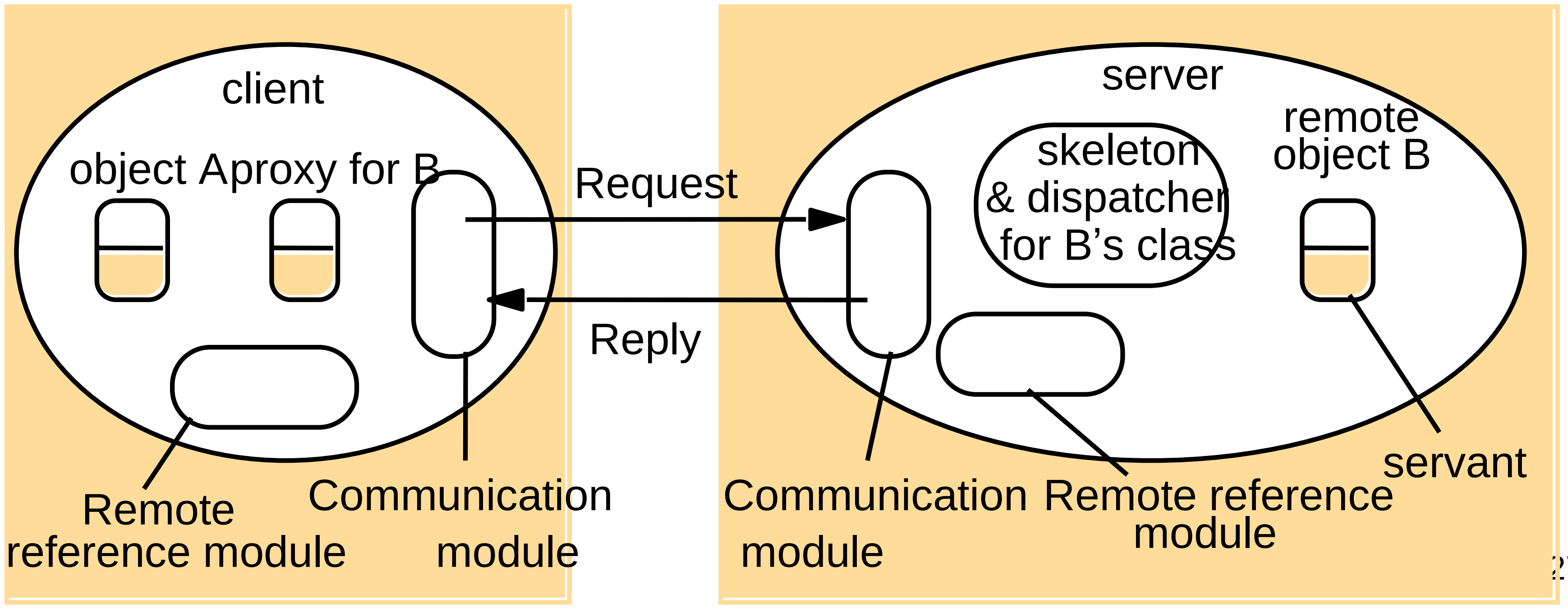
  - Buffer result messages at server for *retransmission* – avoids redo of requests (even for idempotent ops) – at-most-once semantics.
    - Idempotent operation – the one which can be performed repeatedly with the same effect as if it had been performed exactly once.

Transparency

- Remote invocations should be made transparent
  - the syntax of a remote invocation is the same as that of a local invocation, but
  - the difference between local and remote objects should be expressed in their interfaces.
- Java RMI:
  - Remote objects implement `Remote` interface and throw *RemoteExceptions*
- Remote object should be able to keep its state consistent in the presence of concurrent accesses from multiple clients.

Implementation of RMI

- Several separate object and modules
- An application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference.



RMI: Communication module

- Two modules
  - Transmits *request* and *reply* messages between client and server;
- Responsible for providing a specified invocation semantics, e.g. at-most-once;
- The communication module in the server select the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the *request* message.

RMI: Remote reference module

- Responsible for:
  - translating between local and remote object references;
  - creating remote object references.
- Each module has a *remote object table* that records the correspondence between local object references in that process and remote object references (which are system-wide).
- Actions:
  - Create a remote object reference for a first invocation of remote object;
  - Holds local object reference (which may refer either to a proxy or to a remote object)

RMI: Servants

- An instance of a class which provides the body of a remote object.
- Handle the remote requests passed on by the corresponding skeleton.
- Created when remote objects are instantiated and remain in use until they are no longer needed.

The RMI software

- Consists of a layer of software between the application-level objects and the communication and remote reference modules:
  - *Proxy*
  - *Dispatcher*
  - Skeleton

Proxy

- *Proxy:*
  - makes remote method invocation transparent to clients by behaving like a local object to the invoker;
  - instead of executing an invocation, forwards it in a message to a remote object;
  - hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client;
  - one proxy for each remote object for which a process holds a remote object reference.
  - The proxy is ***static***

The RMI software: *Dispatcher*

- A server has one dispatcher for each class representing a remote object

- Receives the *request* message from the communication module and uses the *methodId* to select the appropriate method in the skeleton, passing on the request message.

- The dispatcher and proxy use the same allocation of `methodIds` to the methods of the remote interface.

The RMI software: *Skeleton*

- The class of a remote object has a skeleton, which implements the methods in the remote interface.

- Unmarshalls the arguments in the *request* message and invokes the corresponding method in the servant;

- It waits for the invocation to complete and then marshals the result, together with any exceptions, in a *reply* message to the sending proxy’s method.

Representation of a remote object reference

- a remote object reference must be unique in the distributed system and over time. It should not be reused after the object is deleted.

Why not?

- the first two fields locate the object unless migration or re-activation in a new process can happen
- the fourth field identifies the object within the process
- its interface tells the receiver what methods it has (e.g. class *Method*)
- a remote object reference is created by a remote reference module when a reference is passed as argument or result to another process
  - it will be stored in the corresponding proxy
  - it will be passed in request messages to identify the remote object whose method is to be invoked

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

Server and client programs

- Server contains:
  - The classes for the dispatchers and skeletons; together with the implementations of the classes of all the servants that it supports;
  - *initialization* section – responsible for creating and initializing at least one of the servants to be hosted by the server.

- Client program contains:
  - The classes of the proxies for all the remote objects that it will invoke.
  - It can use a binder to look up remote object references,

Distributed garbage collection

Ensures

- if a local or remote reference to an object is still held anywhere in a set of distributed objects
  - then the object itself will continue to exist,
- As soon as no object holds a reference to it
  - the object is collected and the memory it uses recovered.

Java distributed garbage collection algorithm

Based on reference counting. Works with the local garbage collector as follows:

1. Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects. E.g. *B.holders* is a list of client processes referencing a remote object B.

2. When the client first receives a remote reference for B it makes an *addRef(B)* invocation on the server. The server adds C to *B.holders*.

3. When C’s garbage collector notices that the proxy for B is no longer reachable it makes a *removeRef(B)* invocation on the server and deletes the proxy. The server deletes C from *B.holders.*

4. When *B.holders* is empty, the server’s local garbage collector will reclaim the space occupied by B unless there are any local holders.

Java distributed garbage collection algorithm

- Failure handling:
  - Needs to be time delay between removal of last reference and garbage collection; duplication of messages, missing messages
  - *addRef* and *removeRef* are idempotent: *if an addRef(B)* call returns an exception, the client will not create a proxy but will make a *removeRef(B)* call. The effect of *removeRef* is correct whether or not the *addRef* succeeded.

- Failures of client processes:
  - Servers *lease* their objects to clients for a limited period of time (starts with *addRef* and ends when time period elapses or when the client calls *removeRef).*

Summary

RMI

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.

- Local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once

- Middleware components - proxies, skeletons and dispatchers
  – hide details of marshalling, message passing and object location from programmers.