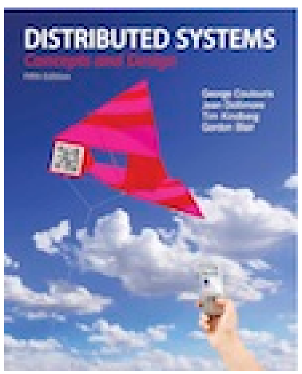
Slides for Chapter 5: Remote Invocation



From Coulouris, Dollimore, Kindberg and Blair

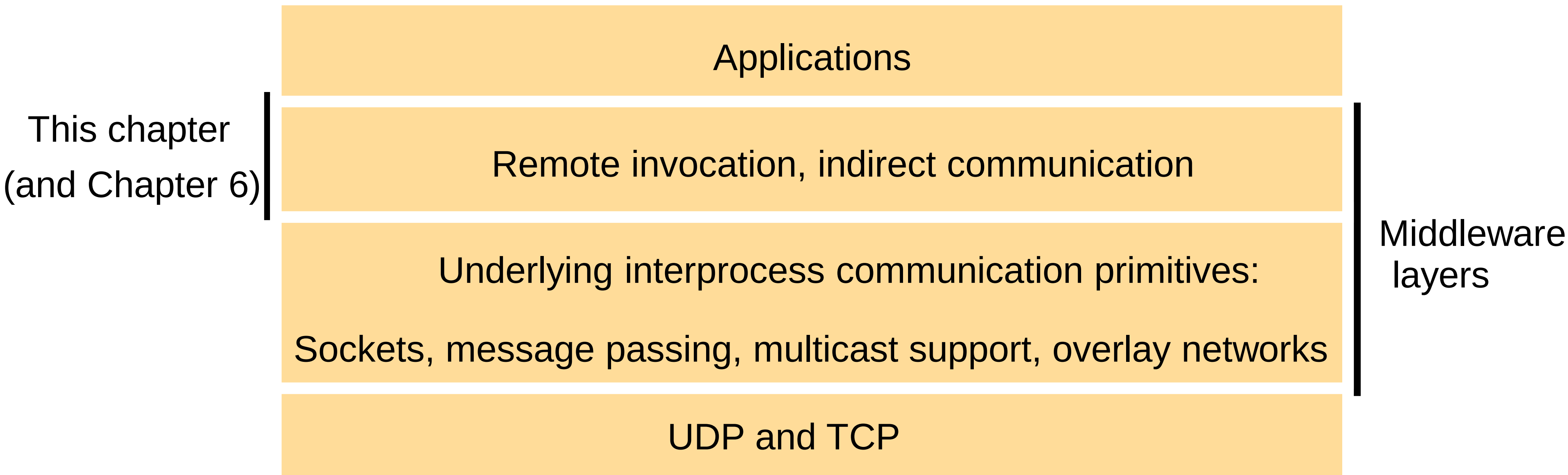
Distributed Systems:

Concepts and Design

Introduction [5.1]

- This chapter: how processes/objects/components/services communication via remote invocation (Chap 2)
- Request-reply
  - Small/thin pattern on top of message passing
  - Can use directly in app (“app protocols”), or build RPC/RMI on
- Remote Procedure Call (RPC)
  - Make a remote procedure look (almost) like a local one to call
- Remove Method Invocation (RMI)
  - Make a remote object look (almost) like a local one to invoke
  - Note: ‘RMI’ is generic category, Java RMI is a specific instance

Figure 5.1
Middleware layers



Request-reply protocols [5.2]

- Support low-level client-server interactions
  - Usually synchronous and reliable
- Built on top of `send` and `receive` operations from Chapter 4
  - Usually use UDP datagrams, could use TCP streams
- Three primitives
  - `doOperation`: client sends request message to server
  - `getRequest`: server receives request msg, selects+invokes oper.
  - `sendReply`: server sends reply message back to (blocked) client

Figure 5.2
Request-reply communication

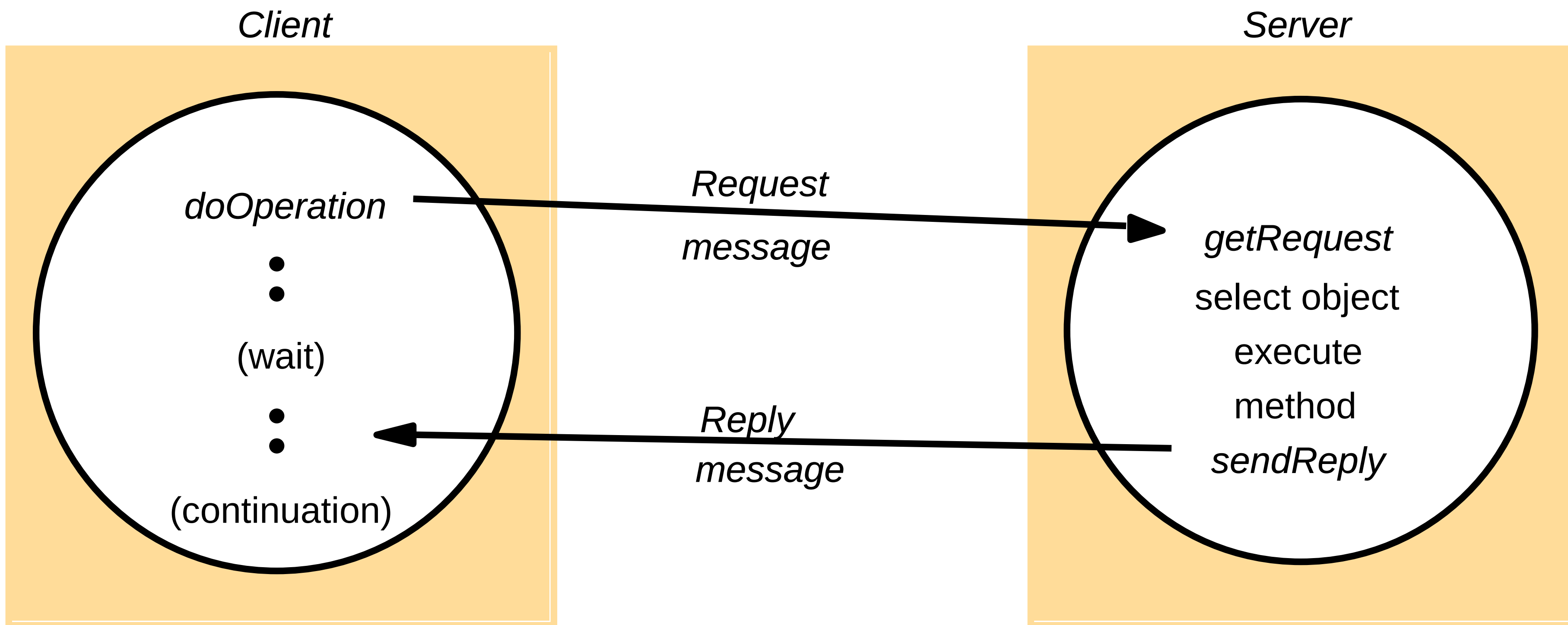


Figure 5.3
Operations of the request-reply protocol

***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.

Figure 5.4
Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

Request-reply protocols (cont.)

- Message identifiers: must identify request uniquely
  - `requestId`: usually a sequence counter (makes unique at client)
  - Client/sender identifier endpoint (with `requestId`, globally unique)
- Failure model
  - Over UDP: omission, misordering
  - Over UDP or TCP: server crash failure (later, Byzantine…)
- Timeouts: doOperation uses when blocked for reply
  - Options to use?
- Duplicate request msgs: server may get >1 times
  - how? problem?
  - Soln: server tracks what got from client (how?)

Request-reply protocols (cont.)

- Lost reply messages
  - Idempotent operation: just redo
  - Else store reply history (how many? How to use?)
- Q: should client and/or server ACK messages?

Figure 5.5
RPC exchange protocols

Name		*Messages sent by*	
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Using TPC streams to implement request-reply protocol

- Advantages
  - Never need multi-packet protocols
  - “Reliable”
- Disadvantages
  - More CPU intensive: scale

HTTP RR protocol SUMMARY (Read rest on in CDKB5 text)

- HTTP protocol specifies
  - Messages in RR exchange
  - Methods
  - Arguments
  - Results
  - Marshalling rules
  - Content negotiation
  - Authentication
- Implemented over TCP streams
  - Early versions: new connection for each request (later persistent)
  - Zinky(Akamki) ~2019: http3 will replace virtually all current TCP+UDP
- Request & reply msgs marshalled into ASCII
- Resource data can be represented as a byte sequence

Remote procedure call [5.3]

- Design issues
  - Style of programming promoted by RPC: using interfaces
  - Call semantics
  - Transparency

Programming with interfaces

- Explicit interface
  - Hide a lot of implementation details
  - Tell exactly how a client can access the server
- Keeping implementation separate from interface
  - Good idea? Why?
- Differences from local procedure interface
  - Can't access shared memory variables between client and server
  - Call by reference does not make sense for RPC
    - Parameters are **`in`**, **`out`**, or **`inout`**
  - Can't pass pointers
  - Anything else?
- IDL originally developed for RPC

Figure 5.8
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

- Choices for implementing `doOperation`
  - Retry request message
  - Duplicate request filtering at server
  - Retransmission of results: keep reply history, or re-execute procedure

Figure 5.9
Call semantics

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*

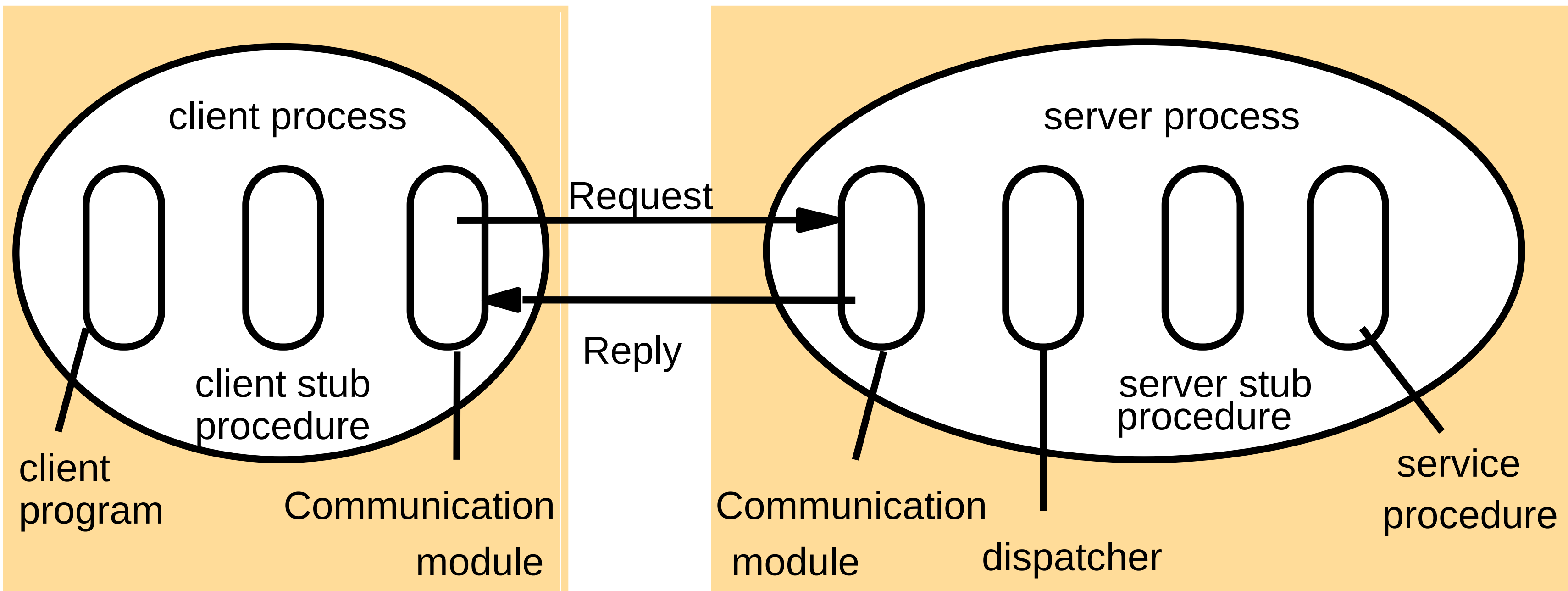
- How can each of these happen?
- What would you call local procedure call semantics?

Transparency

- RPC tries to offer at least location and access transparency
- Does client need to know call semantics?
- Implement RPC with stub/proxy over an RR protocol (Fig 5.10)

- Note: not covering Sun RPC (5.3.3), not testable

Figure 5.10
Role of client and server stub procedures in RPC



Remote method invocation [5.4]

- Fundamental difference between a procedure and an obj.?
- Similarities between RPC and RMI
  - Programming with interfaces
  - Both constructed on top of some RR protocol and have same choices in call semantics
  - Similar level of transparency
- Differences providing added expressiveness in RMI
  - Full expressive power of OO programming (not just a “fad”…)
  - Can cleanly pass object references as parameters

On Objects and QoS

"I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a Computer Scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented."

Prof. Roger King, U. Colorado at Boulder, 1989

"My cat is CORBA-compliant".

Dr. John Nicol, GTE Labs, 1995

"My CORBA-compliant cat has great quality of service."

Dr. David Bakken, BBN, 1996

"The DCOM architecture is fundamentally ugly and unclean, at a profound and deeply-disturbing level."

Dr. David Bakken, BBN, 1998

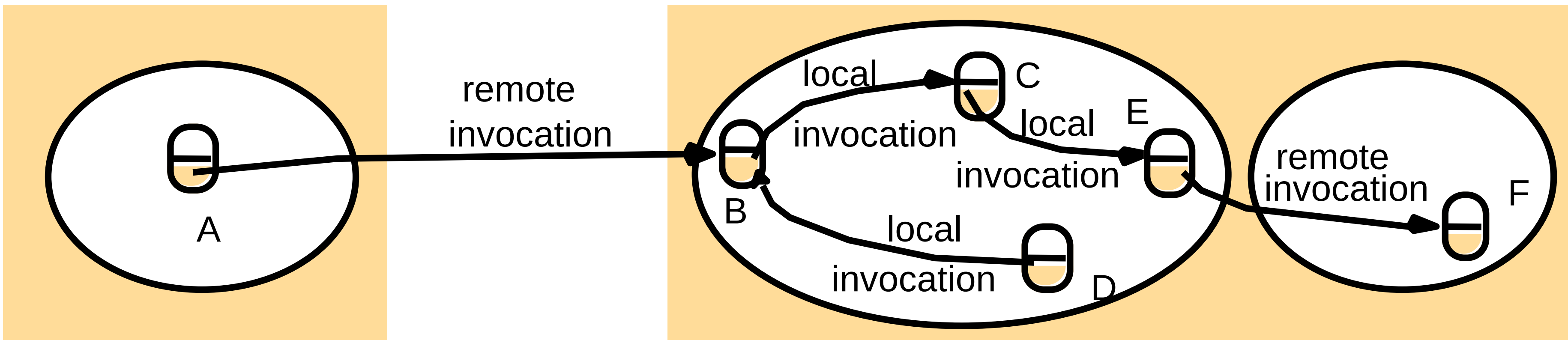
Design issues for RMI: object model!

- Local object model (C++, Java, …)
  - Collection/packaging of code and data
  - Communicate by invoking methods
  - Sometimes allowed to invoke instance variables directly
  - Object references are first-class values: assigned to variables, passed as parameters, …
  - Interfaces:impl sometimes 1:1 (C++), or many:1 (Java class can implement multiple interfaces)
  - Action: invocation can have side effects at invoked object: state changed, instantiate new object, invoked object invokes another…
  - Exceptions
  - Garbage collection (manual or automatic)

Distributed objects and distribited object models

- Most ways similar/identical to local object model
- Client-server architecture (encapsulation), with variations
  - Replication
  - Migration
- Distributed object model (Fig 5.12)
  - Process is a collection of objects (some remotely invoke-able)
  - Remote object references: need one to invoke a remote object
  - Remote interfaces: each object must have one

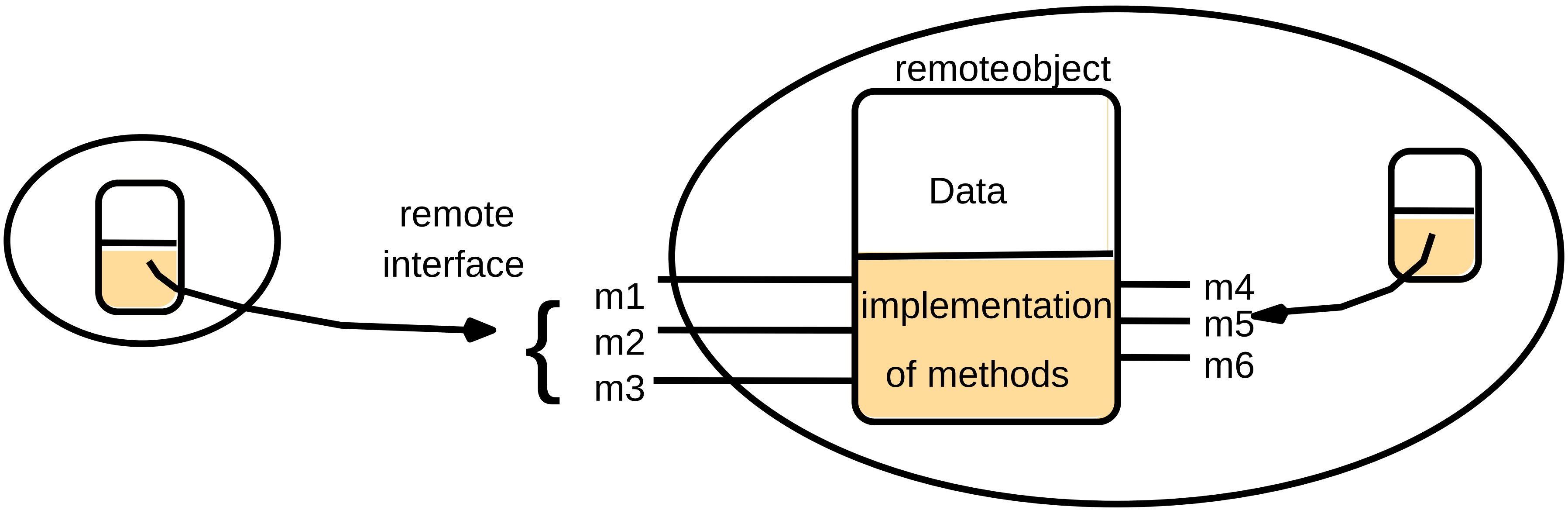
Figure 5.12
Remote and local method invocations



Remote object references and remote interfaces

- Remote object reference
  - ID that can be used throughout a DS
  - Strongly analogous to local object references
- Remote interfaces
  - A class implements one or more remote interfaces (Fig 5.13)
  - CORBA: see previous, uses IDL
  - Java RMI: just like any other Java interface (extends it)
  - Multiple inheritance of interfaces in both CORBA and Java

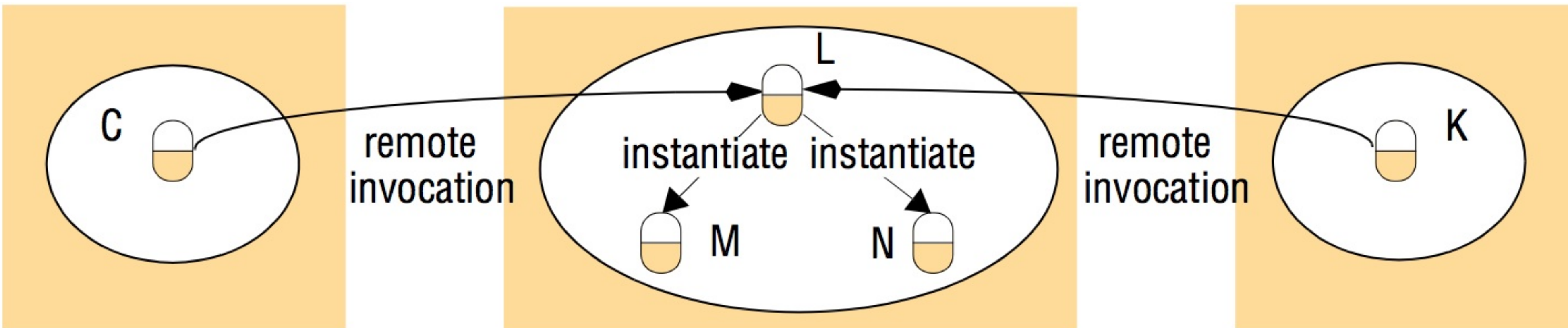
Figure 5.13
A remote object and its remote interface



Actions in a distributed object system

- Can result in chain of invocations across computers
- Can instantiate new objects
  - Usually local
  - Or via a **factory** interface
- Garbage collection
  - Harder than local garbage collection (why?)
  - Local GC and distributed GC module cooperate (using ref. counts)
- Exceptions:
  - Similar to local
  - But more for remote problems
  - Also can have app-level exceptions (e.g., CORBA cross-language)

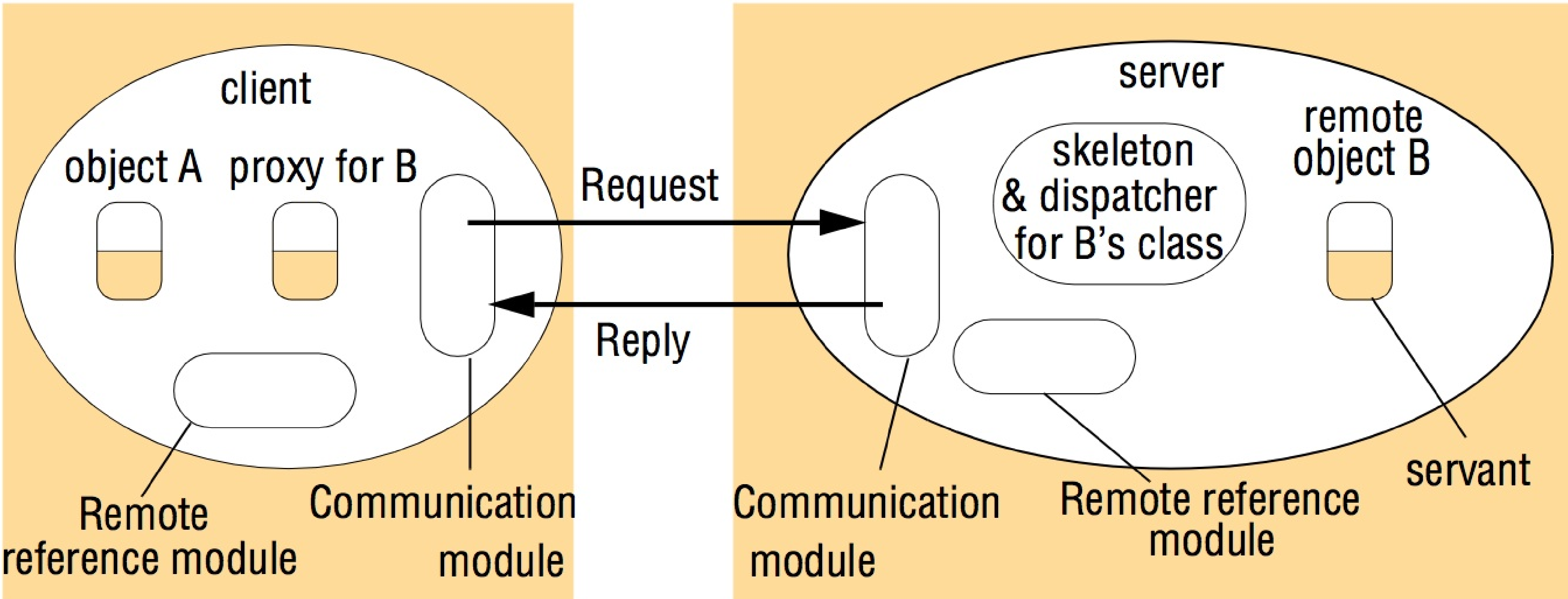
Figure 5.14
Instantiation of remote objects



Implementation of RMI

- (See Fig 5.15)
- Communication modules: cooperate to implement the call semantics
- Remove reference modeue
  - Translate between local and remote object references
  - Create remote object references
- Servant: instance of a class, body of remote object
- RMI software
  - Proxy: provide transparency
  - Dispatcher & Skeleton: one per class of a remote object

Figure 5.15
The role of proxy and skeleton in remote method invocation



Implementation of RMI (cont.)

- Dynamic invocation
  - Don't use a compiler-generatated proxy (`doOperation` body), program one!
  - Useful when IDL not available when compiling program
    - CORBA Interface Repository
  - Examples: debugger, class browser, shared whiteboard
  - Dynamic skeletons: server side analogue
- Binder: mapping from text names to remote obj. refs
- Activator: manages object activation and passivation
  - Registers passive objects available for activation
  - Start named server processes (incl. remote object in them)
  - Keep track of servers for activated remote objects

Implementation of RMI (cont.)

- Persistent object stores
  - **Persistent object**: one guaranteed to live between activations
  - Managed by a persistent object store
  - Marshalled state in file or database
- Object location
  - Objects can migrate!
  - **Location service**: maps from object references to probable current locations

Distributed garbage collection

- Job: recycle objects no longer “pointed to” by a reference
- Typical scheme
  - Use reference counting
  - Local garbage collector
  - Distributed garbage collector (cooperates with locals)
- Algorithm
  - Each server tracks names of processes that have references to its remote object
  - If local GC notices proxy not reachable, lets GC on object host know
  - When no references to object, recycle it
- Complications: ref in msg

Leases

- Used in Java, Jini
- Client has “lease” of object for fixed time
  - Has to renew it before expiration
  - Way of removing un-freed refs
  - Avoids the complicated distributed GC algorithm
- Note: not covering Section 5.5 (Case Study: Java RMI)