Slides for Chapter 5:
Remote invocation

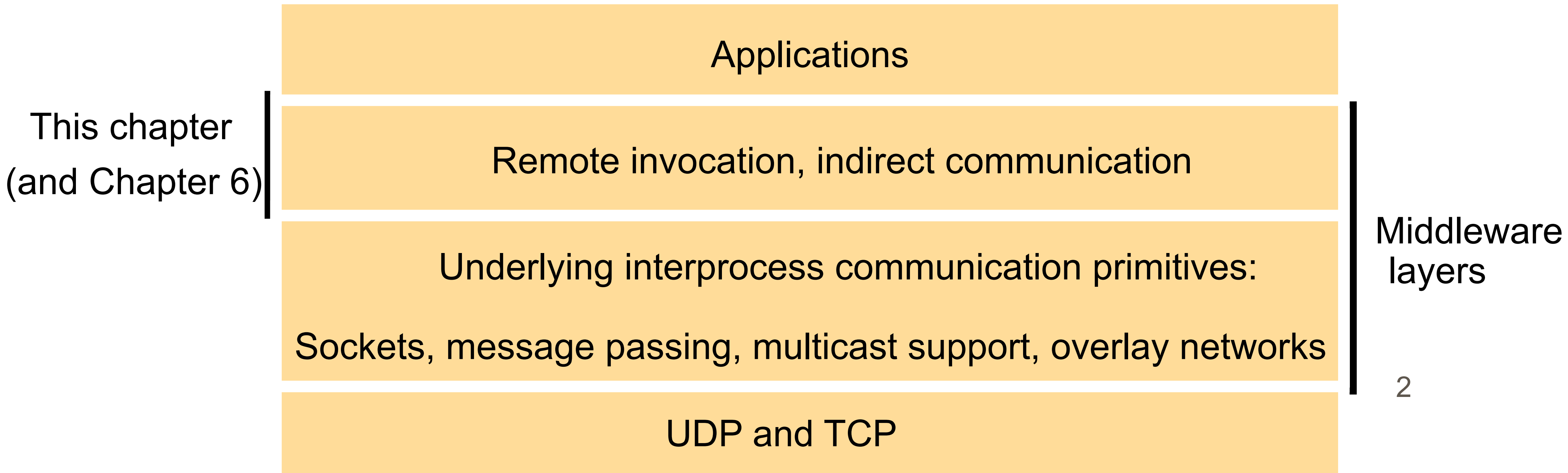
From **Coulouris, Dollimore and Kindberg**

Distributed Systems:

Concepts and Design

How two processes can communicate?

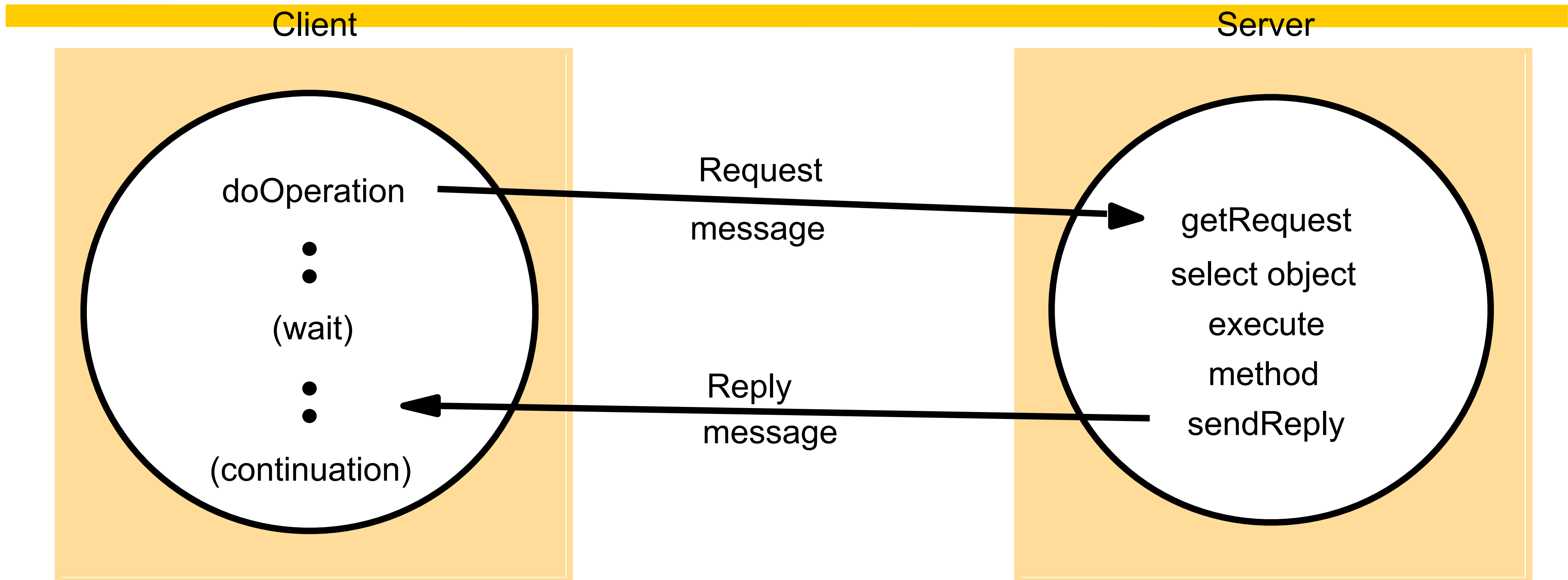
- Request-Reply protocols
  - simplest form
- Remote Procedure Calls
- Remote Method Invocation
  - same thing, but object oriented



Request-reply communication

- Designed to support typical client-server interactions
- Usually synchronous
- Can be implemented over UDP, TCP, others...
- When implemented over UDP is faster and more efficient, since
  - ACKs are redundant
  - establishing connections is useless
  - flow control is redundant if arguments and results are small
- On the other hand, TCP offers
  - unlimited message size
  - ordered delivery
  - simpler implementation (no need for acks)

Operations of the request-reply protocol



public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

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 message structure

Field	Type
messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

Client-server communication

- Failure model
  - UDP: could be out of order, lost...
  - process can fail...
- not getting a reply
  - timeout and retry
- duplicate request messages on the server
  - How does the server find out?
- idempotent operation: can be performed repeatedly with the same effect as performing once.
  - idempotent examples?
  - non-idempotent examples?
- history of replies
  - retransmission without re-execution
  - how far back if we assume the client only makes one request at a time?

Request-replay exchange protocols

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

Example: HTTP request-replay

Request:

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Replay:

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Remote Procedure Call

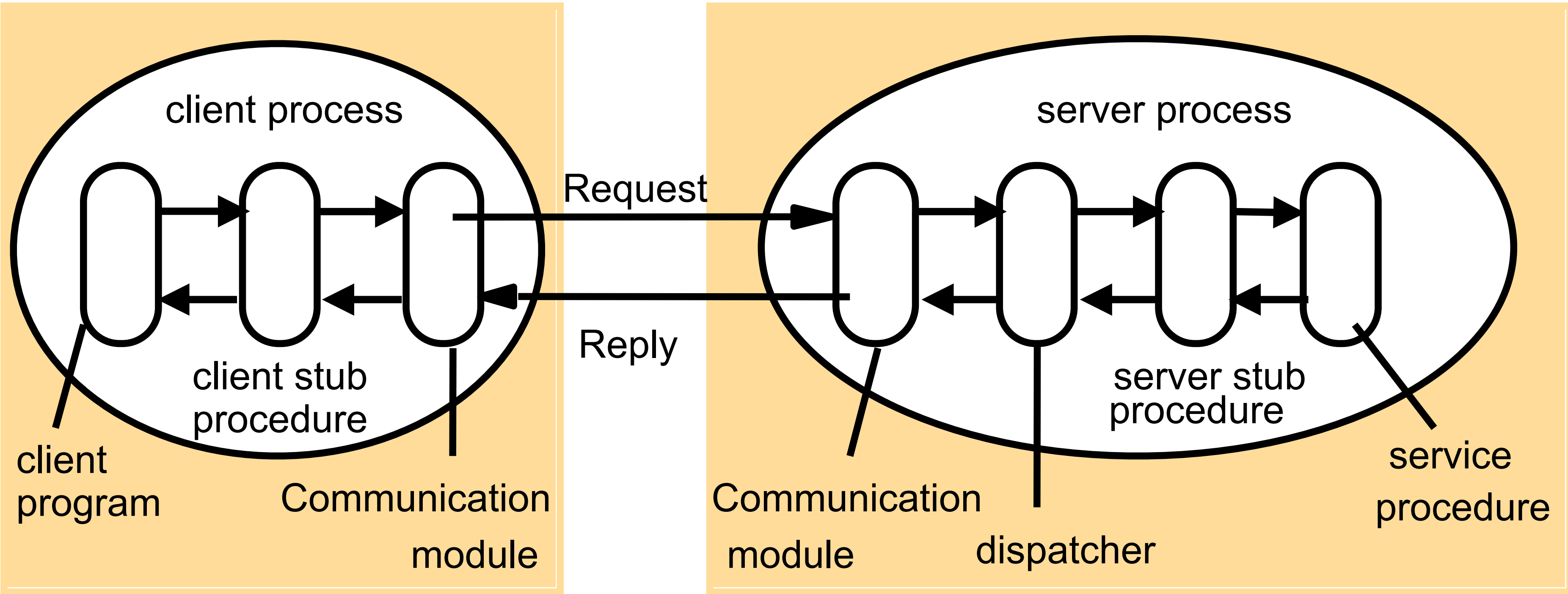
- major breakthrough in distributed computing
- aim: making the programming of distributed system similar, if not identical, to conventional programming
- RPC hides encoding/decoding of data, passing of messages, preservation of call semantics
- Important aspects to consider in RPC:
  - the style of programming: programming with interfaces
  - call semantics associated with RPC
  - the transparency which is guaranteed by RPC

- RPC call semantics:
  - maybe
  - at-least-once
  - at-most-once
- Transparency
  - RPC must be as much like local procedure calls as possible, both in syntax and in semantics
  - However programmer must be aware that
    - latency is much higher
    - some data types cannot be used
    - higher failure rate

Implementation

- client: "stub"
  - local call, identical to original procedure
  - marshal arguments
  - communicate the request to the remote procedure
  - receive result
  - unmarshal result and return it
- server:
  - dispatcher: receives request and executes right procedure
  - "stub"
    - unmarshal arguments
    - execute code
    - marshal result
    - send it back

Role of client and server stub procedures in RPC in the context of a procedural language



Example: Sun RPC

- Client-server comm in the SUN NFS (network file system)
- Also called ONC (Open Network Computing) RPC
- In other unix OS as well
- Runs over UDP or TCP
- Interface Definition Language (IDL)
  - initially XDR is for data representation, extended to be IDL
  - less modern than CORBA IDL and Java
    - program numbers instead of interface names
    - procedure numbers instead of procedure names
    - single input parameter (structs)
- `rpcgen`: compiler for XDR: generates
  - client stub
  - server main procedure, dispatcher, and server stub
  - XDR marshalling, unmarshaling

Figure 5.9
Files interface in Sun XDR

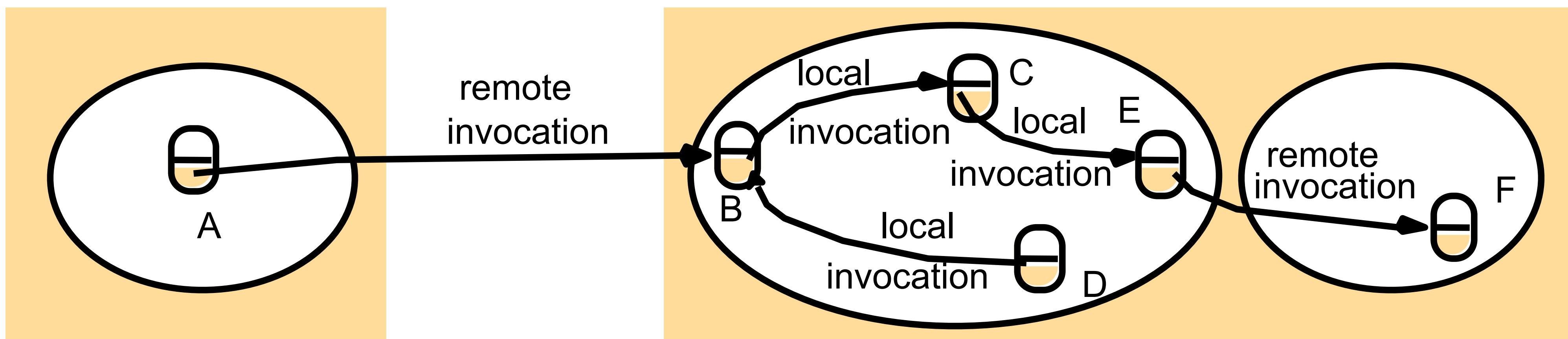
```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;        1
    Data READ(readargs)=2;          2
  }=2;
} = 9999;
```

Sun RPC

- binding (registry)
  - local binder--*portmapper*
  - server registers its program/version/port numbers with portmapper
  - client contacts the portmapper at a fixed port with program/version numbers to get the server port
  - different instances of the same service can be run on different computers--different ports
- authentication
  - request and reply have additional fields
  - unix style (uid, gid), shared key for signing, Kerberos

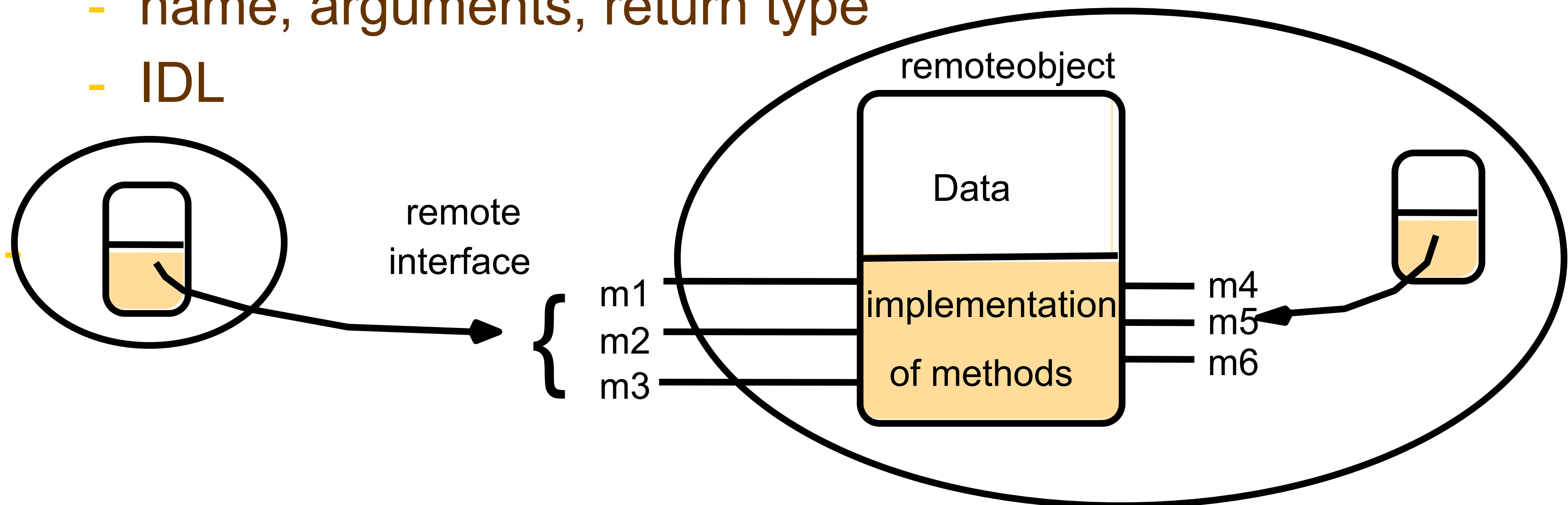
Object Communication

- Distributed objects
  - state: values of its instances variables
  - actions: accessed only by its own methods
- Local within the same process
- Remote: different processes (could be on different machines)



Remote object and its remote interface

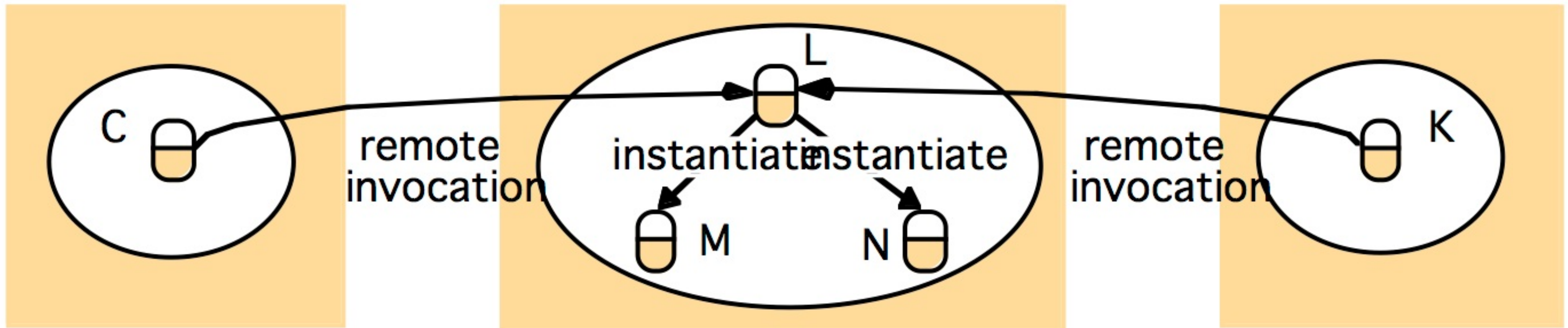
- Remote object reference
  - accessing the remote object identifier throughout a distributed system
  - can be passed as arguments
- Remote interface
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - IDL



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();
};
```

Implementation of remote objects



- Needs remote object reference
- Calling of methods of objects in another process/host
- Remote objects might have methods for instantiation (hence remote instantiation)

- Garbage collection
  - local garbage collector, with additional module to coordinate
- Exceptions
  - unexpected events or errors
  - more and different exceptions from local methods

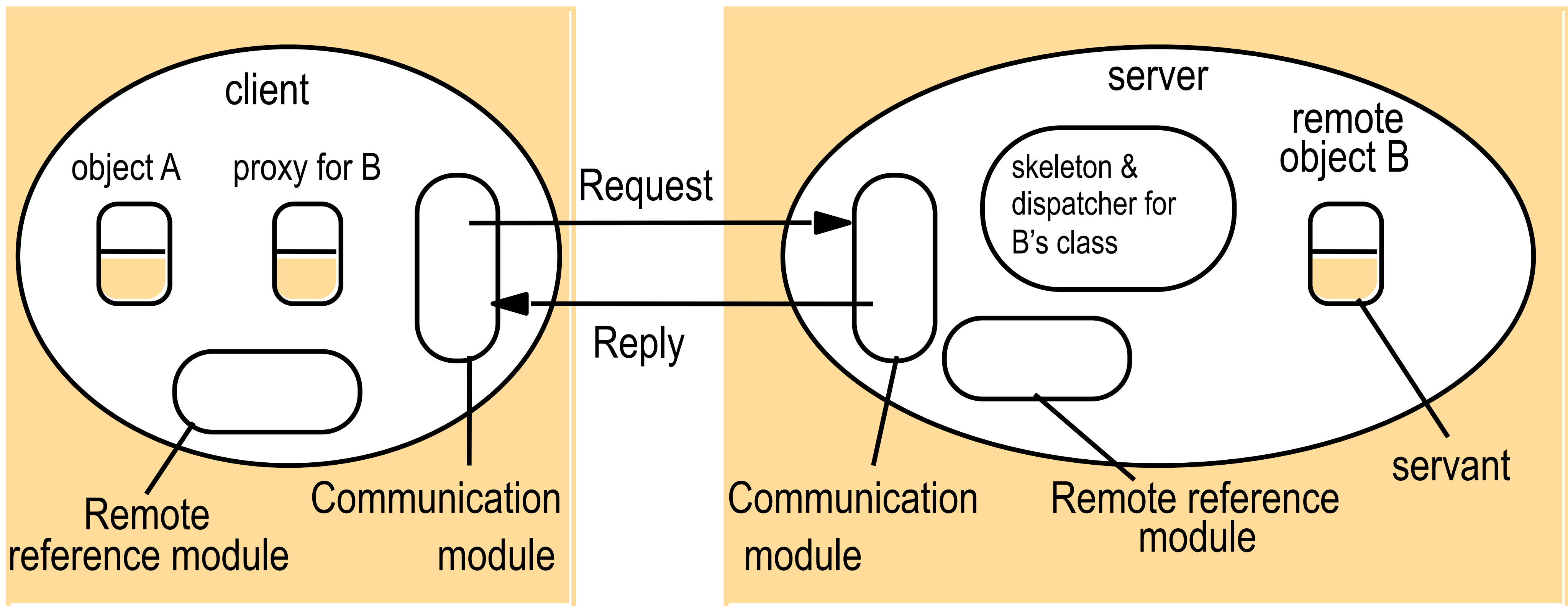
RMI Invocation semantics

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*

RMI transparency

- like a local call
  - marshalling/unmarshalling
  - locating remote objects
  - accessing/syntax
- latency
- more likely to fail
- errors/exceptions: failure of the network? server? hard to tell
- consistency on the remote machine
  - Argus (1982): incomplete transactions, abort, restore states [as if the call was never made]

RMI implementation:
The role of proxy and skeleton in remote method invocation



- Proxy: behaves like a local object, but represents the remote object
- Dispatcher: look at the methodID and call the corresponding method in the skeleton
- Skeleton: implements the method
- Generation of proxies, dispatchers and skeletons: IDL (RMI) compiler

- Communication module
  - request-reply: message type, requestID, remote object reference
  - implements specific invocation semantics
  - selects the dispatcher, passes on local reference from remote reference module, return request
- Remote reference module
  - translating between local and remote object references
  - remote object table
    - remote objects held by the process (B on server)
    - local proxy (B on client)
  - remote object (first time): add to the table, create proxy

RMI implementation: server, activation of objects

- Server initialization
  - Server creates the first object for remote access
  - Usually clients are not allowed to create servers
- Server threads
  - concurrency
- activation of remote objects
  - many server objects, all running?
  - active and passive status
    - active: available for invocation in a running process
    - passive: not running, state is stored on disk
    - activation: create an active object from a passive object, register the new active object
- Java RMI—objects can be activatable (similar to inetd)

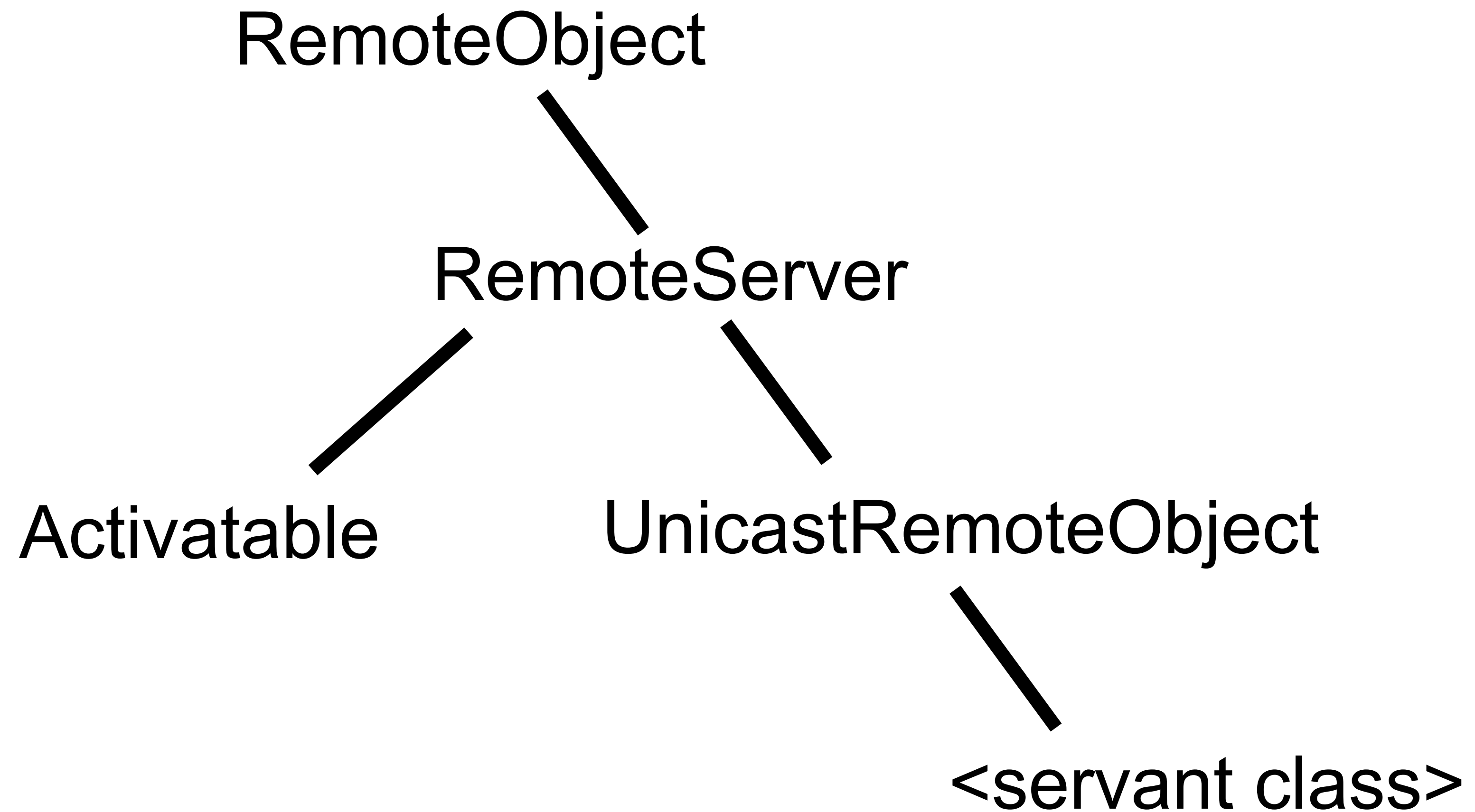
RMI implementation: persistent and migrable objects

- persistent object stores
  - stored in marshaled form on disk for retrieval
  - saved those that were modified
  - persistent or not:
    - persistent root: any descendent objects (reachable from the root) are persistent (eg. persistent Java, PerDiS)
    - certain classes are declared persistent (eg. Arjuna system)
- migrable objects
  - marshaled and moved among location
  - location service for migratable objects:map remote object references to their probable current locations (Clouds and Emerald systems)
  - Cache/broadcast scheme (similar to ARP)
    - Cache locations, if not in cache, broadcast to find it
  - Improvement: forwarding (similar to Mobile IP)

RMI implementation: remote garbage collection

- Reference count
- Java's approach
  - the server of an object (B) keeps track of proxies
  - addRef(B) is called when a proxy is created for a remote object; tells the server to add an entry
  - when the local host's garbage collector removes the proxy, cremoveRef(B) tells the server to remove the entry
  - when no entries for object B, the object on server is deallocated
  - Race condition
    - removeRef(B) from client X
    - addRef(B) from client Y
  - lease times for dealing with client failures

Java classes supporting Java RMI



Example Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;            1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                1
                Naming.rebind("Shape List", aShapeList );                 2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;          // contains the list of Shapes              1
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {        2
        version++;
            Shape s = new ShapeServant( g, version);                         3
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;          1
          Vector sList = aShapeList.allShapes();                                  2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Java RMI Callbacks

- Callbacks are an extension of standard RMI
- server notifying the clients of events
  - Initiated by server [opposite to initiated by client]
- why?
  - polling from clients increases overhead on server
  - no need to update the clients before informing users
- how
  - remote object (*callback object*) on client for server to call
  - client tells the server about the callback object
  - server put the client on a list
  - server call methods on the callback object when events occur
- client might forget to remove itself from the list
  - lease--client expire