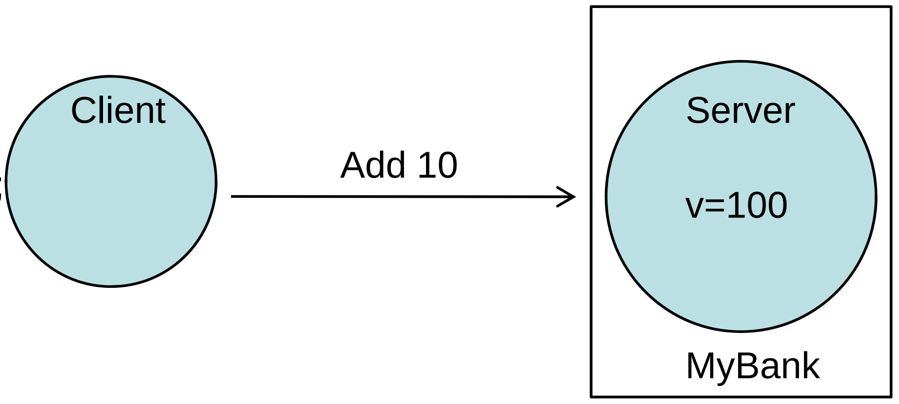
Programming models 1
Remote Object Invocation

Brian Nielsen

bnielsen@cs.aau.dk

Distributed Programming using TCP

```
socket= new Socket("MyBank");
sent_bytes=socket.send("Add 10",7);
 ...
socket.close();
```



- No application level acknowledgement
  - A successful send does not mean that the receiver (process, host) has received and processed the data
  - Means that TCP has buffered the data and will try to deliver it.
  - TCP only masks packet drops and (very) transient network failures
- No packetization
  - Byte stream abstraction: Returned value sent_bytes may be less than requested
  - Programmer must 1) packet length in message, 2) repeat sends
- Data-representation
  - How should “Add 10” be understood? 7-bit ascii?
- Detection and handling of crashes

Distributed programming

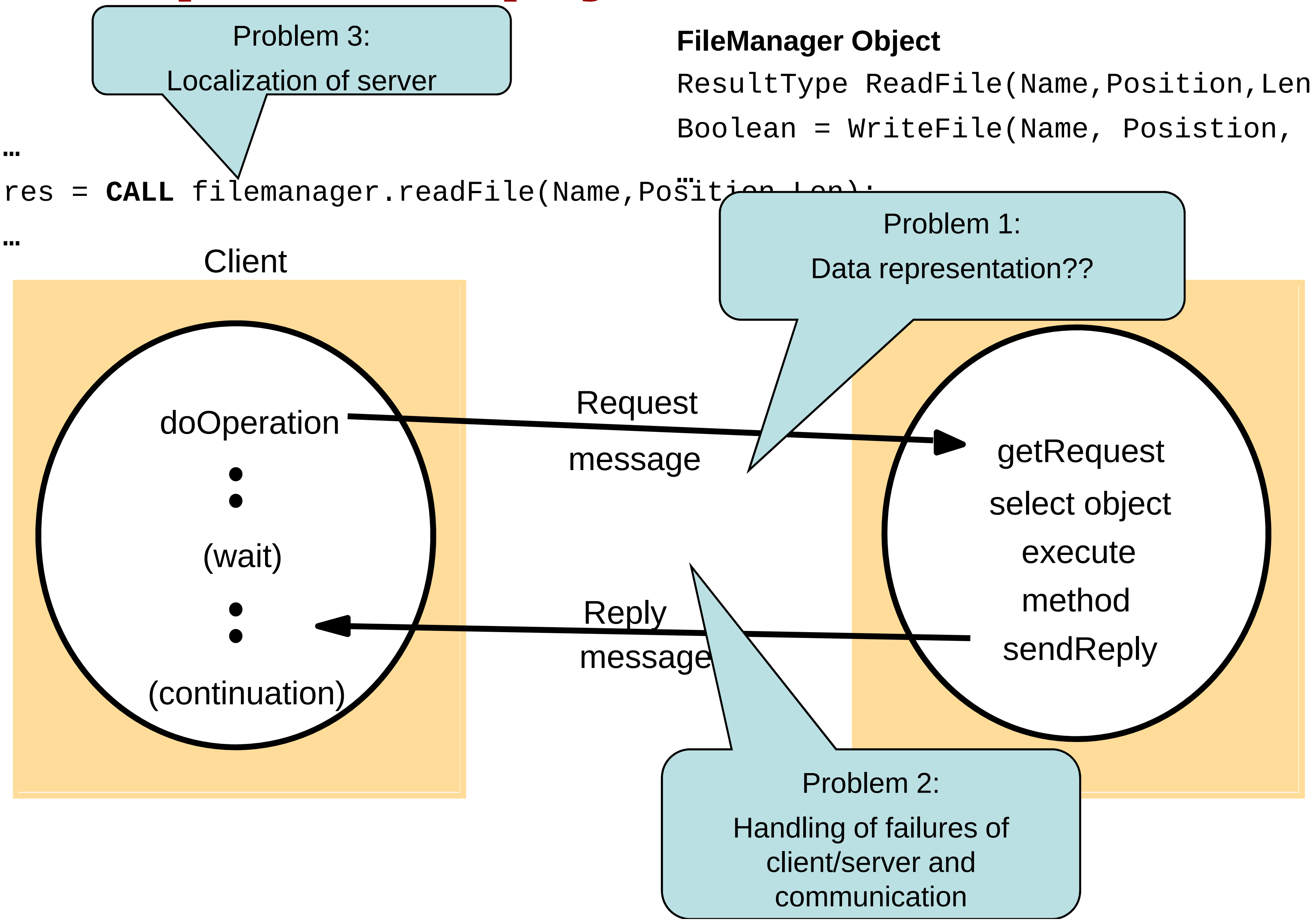
- Directly using the available network protocols
  - Socket API
- Extension of existing language primitives to support distributed programming
  - Remote Procedure Calls
  - Remote Method Invocation/Remoting
- Coordination Languages
  - Embed coordination language in sequential Programming language
  - Linda, Actors
  - Message Queues
- New distributed programming languages,
  - Emerald, Argus, ADA, Clouds, Arjuna, Salsa

Request-reply communication

FileManager Object

```
ResultType ReadFile(Name,Position,Len);
Boolean = WriteFile(Name, Posistion, Len);
…
```

```
…
res = CALL filemanager.readFile(Name,Position,Len);
…
```



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.

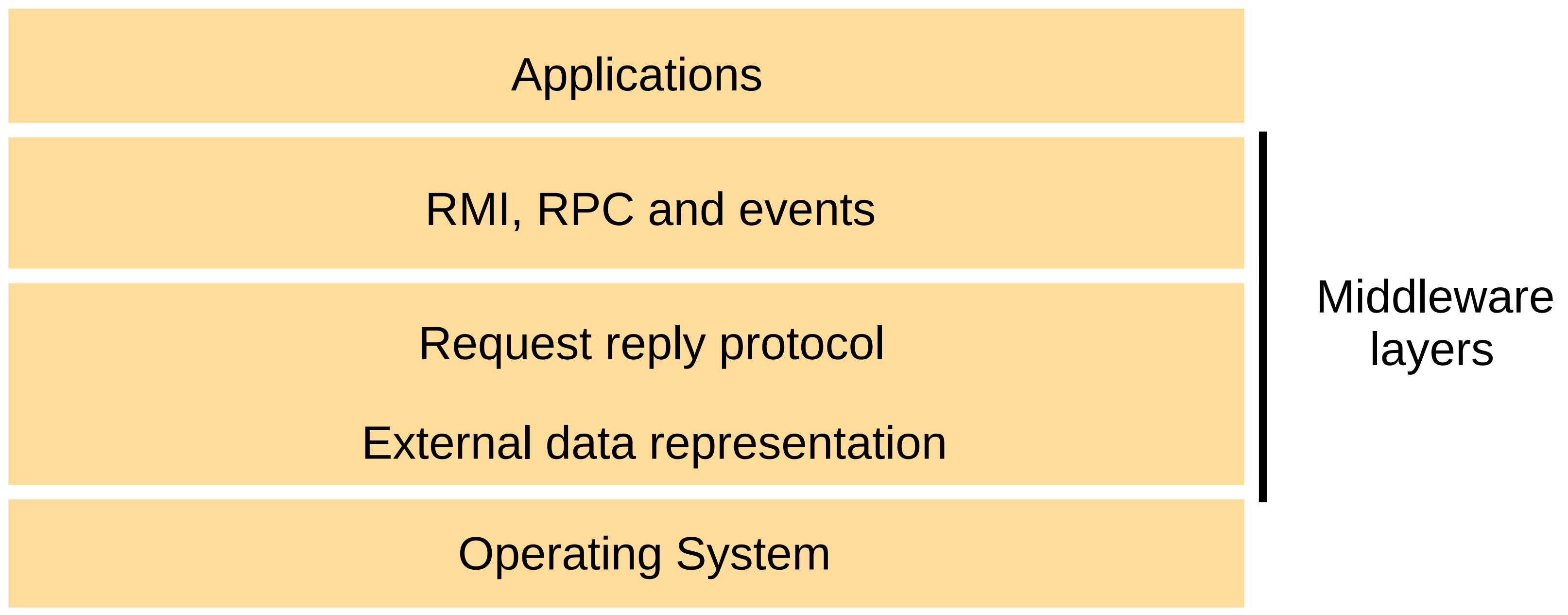
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

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

Request-reply message structure

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

Request-Reply Communication



External Data Representation

Heterogeneity

Hardware

- big or little endian?
- 16, 32, 64 bit integers ?
- ASCII characters vs. unicode
- floating point values, IEEE?
- C-strings vs. UTF-8
- Instruction-sets

Software

- Internal representation of data-structures (padding)

Marshalling

Marshall = Ceremonial-Master

- **Marshalling** is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- **Unmarshalling** it the process of disassembling them on arrival.
  - Values are converted to an agreed external format before transmission, and converted to the local format on receipt.
  - Values are transmitted in the sender's format together with an indication of the format used.
  - *Translate at sender side, receiver side, or both!*

Sender Representation

```
struct Person{
  string name;
  string place;
  long year;
} = {"Brian", "aau", 1969}
```

Transfer / "on wire" Representation

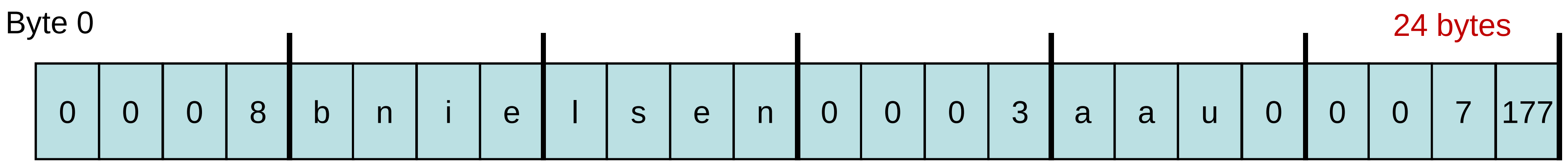
```
100101011111001111111111
```

Receiver Representation

```
class Person{
  string name;
  string place;
  long year;
} = {"Brian", "aau", 1969}
```

External data representation and marshalling

- CORBA common data representation (CDR).



- Java object serialization / .NET object serialization
- XML `<person> <name> bnielsen </name> <place>aau</place> <year><1969/year> </person>`

  ~75 bytes
- ASN.1
  - BER (Basic Encoding Rules)
  - PER (Packed Encoding Rules)
- Issues
  - Speed
  - compactness (of messages and marshalling code)
  - self-containment (type info included)
  - hand-programming vs. interface-compilers
  - robustness

Request-Reply Communication

Failure model for request-reply protocols

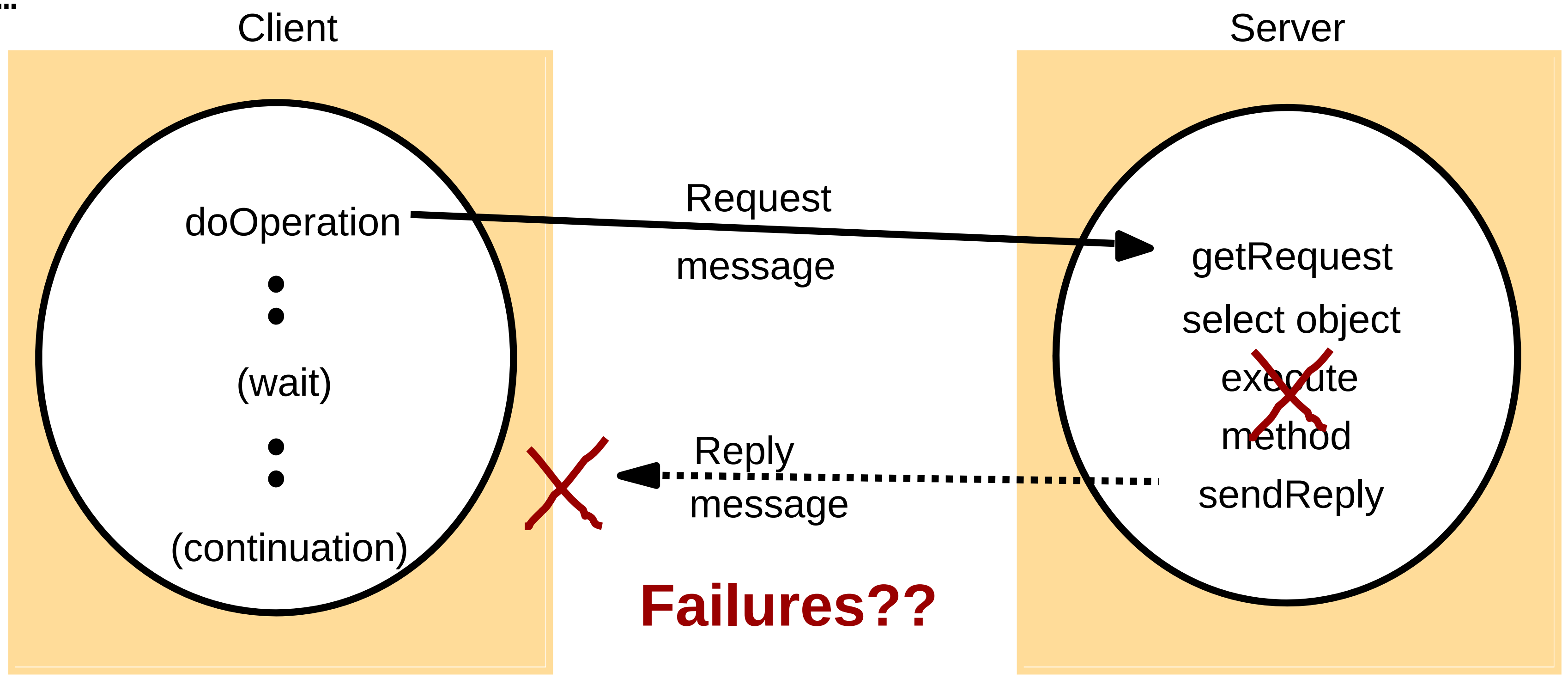
- Omission failures (lost request /reply)
- No ordering guaranteed (eg.UDP does not guarantee ordering)
- Clients and servers have crash faults

Request-reply communication

FileManager Object

```
ResultType ReadFile(Name,Position,Len);
Boolean = WriteFile(Name, Posistion, Len);
…
```

```
…
res = CALL filemanager.readFile(Name,Position,Len);
…
```

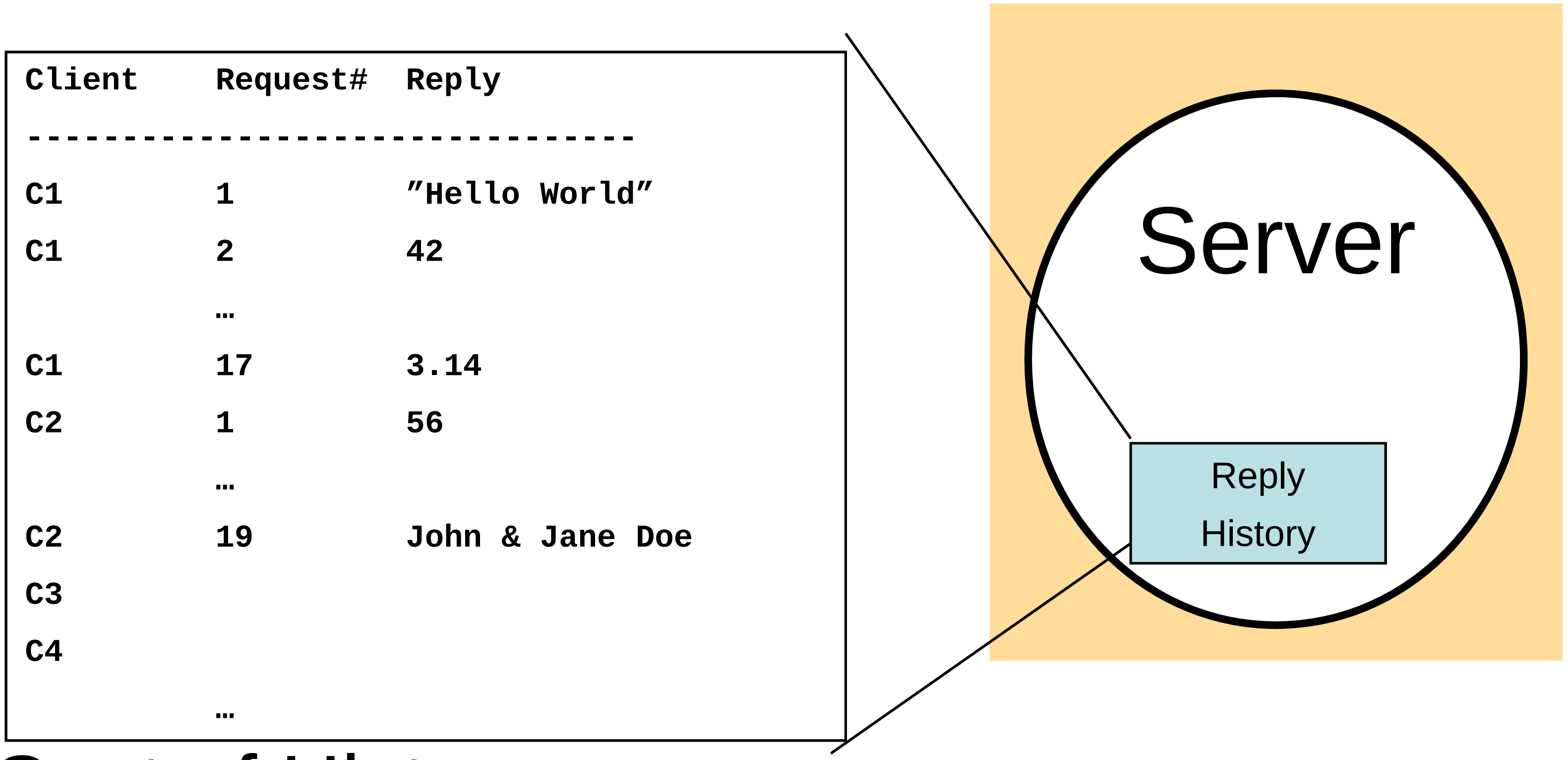


Failures??

Coping with failure

- Clients times out $\Rightarrow$ abort or retry
  - Lost Requests
    - Client: Times out and retransmits
    - Server discards duplicated request messages (seq nr)
  - Lost replies
    - Client: Times out and retransmits request
    - Server:
      - Rexecute, if **idempotent** operation
      - Use a **history** (buffer) of results and retransmit
  - Server Crash
    - Same as lost request or lost reply
- Client Crash

Server Reply History

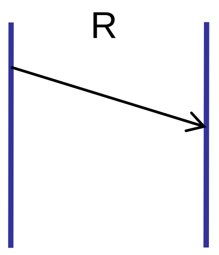


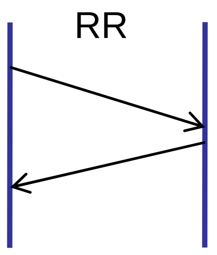
Client	Request#	Reply
C1	1	”Hello World”
C1	2	42
	…	
C1	17	3.14
C2	1	56
	…	
C2	19	John & Jane Doe
C3		
C4		
	…	

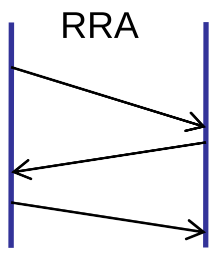
- Cost of History
  - Many clients, many requests
  - Large replies (File-server)
  - How do we garbage collect the history?

Request-reply exchange protocols

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







Distributed Objects
&
RMI

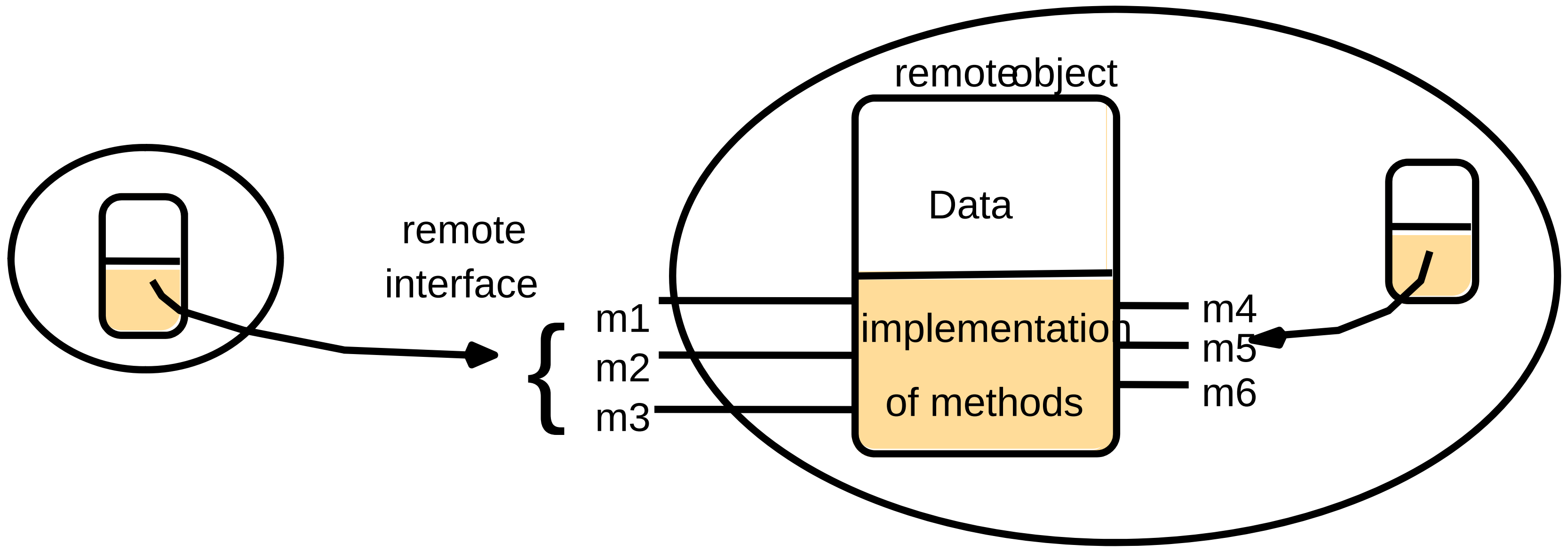
Objects for Dist. Sys?

- *Objects* are units of data with the following properties:
  - *typed and self-contained*
    - Each object is an instance of a *type* that defines a set of *methods* (interface) that can be invoked to operate on the object.
    - The separation between interfaces and the objects implementation
    - Invocation is syntactically and (semantically) independent of an object's location or implementation.
  - *encapsulated*
    - The only way to operate on an object is through its methods; the internal representation/implementation is hidden from view.
    - State only accessible via message passing / RMI
    - Already logically partitioned $\Rightarrow$ physical distribution
    - Unit for persistence, caching, location, replication, and/or access control.
  - *dynamically allocated/destroyed/binding*
    - Objects are created as needed and destroyed when no longer needed; not bound to specific program scope
    - Garbage collection: even more necessary in DS
    - Client dynamically locates and binds to servers,
  - *uniquely referenced*
    - Each object is uniquely identified during its existence by reference that can be held/passed/stored/shared.
    - For Distr sys: add mapping between id and (current) location

Distributed Objects in the Marketplace

1. Java Remote Method Invocation (*JAVA-RMI*)
   - API and architecture for distributed Java objects
2. Microsoft *Remoting*
   - Distributed objects for .NET
3. Microsoft *Component Object Model* (*COM/DCOM*)
   - binary standard for distributed objects for Windows platforms
   - e.g., clients generated with Visual Basic, servers in C++
   - extends OSF DCE standard for RPC
4. CORBA (Common Object Request Broker Architecture)
   - OMG consortium formed in 1989
   - multi-vendor, multi-language, multi-platform standard
5. Enterprise Java Beans (EJB) [1998]
   - CORBA-compliant distributed objects for Java, built using RMI
6. Web services and SOAP

A remote object and its remote interface



Service Interface:

•specification of remotely callable procedures offered by server

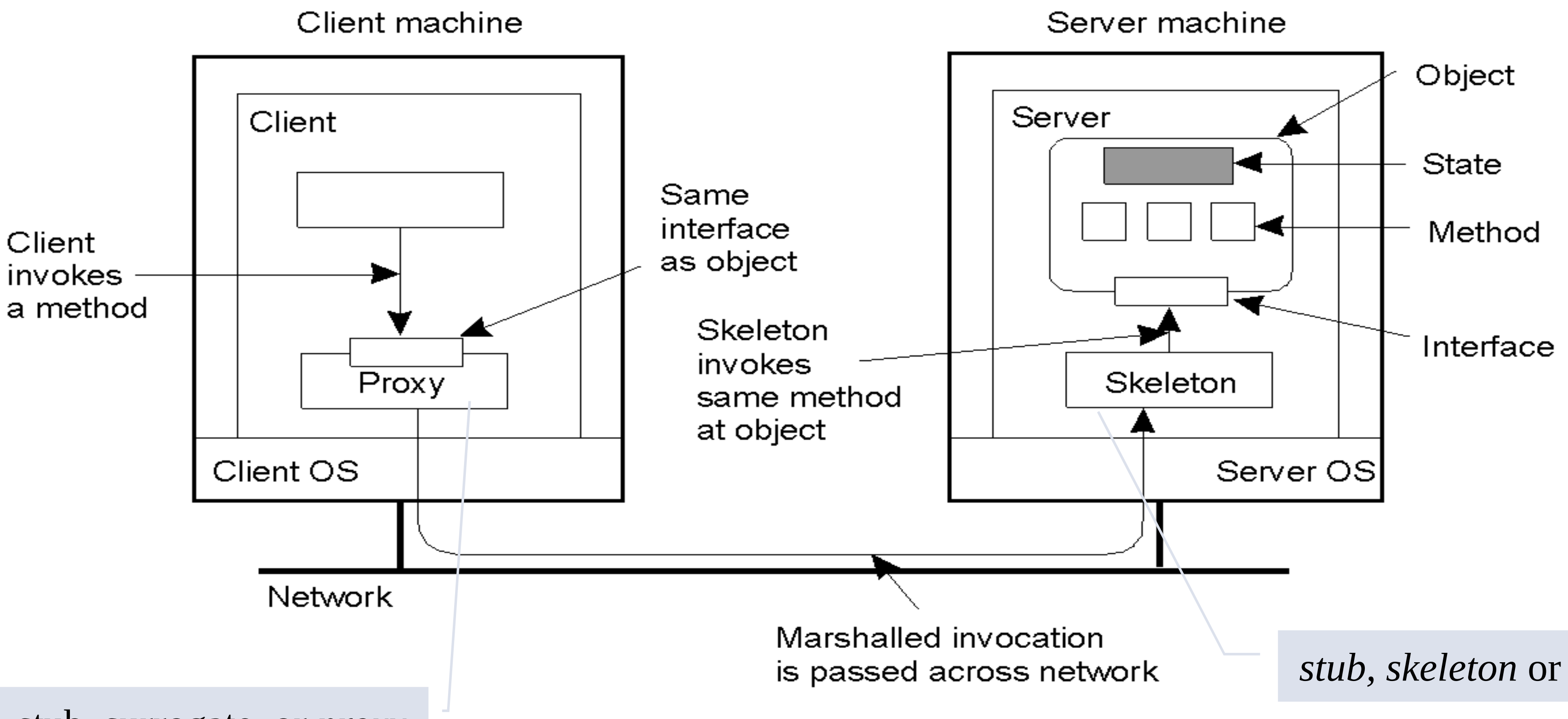
•method signatures (name, input/out parameters and types)

•=remote interface

•Interface Definition Language (IDL): Allows for language heterogeneity

Distributed Objects

Creating the illusion of "procedure call" **o_1=remoteObj.m(o_2, o_3,o_4);**



stub, surrogate, or *proxy*

stub, skeleton or *guard*

Parameter Passing

- **o_1=remoteObj.m(o_2, o_3,o_4);**
- Should parameters (IN and OUT / return) be transferred by value or reference**???**
  - Normally, anything is by reference, except primitive or valueTypes
- By reference: an remoteObjectRef is transfered
  - Access to by-reference-parameters will be yet another expensive RMI
    - IN parameters at server: o_2.m()
    - Return parameters at client: o_1.m()
- Call-By-Value: a copy created at receiver
  - potentially expensive marshalling and communication of large objects state+code
- System objects cannot be marshalled eg. open files,threads,)

Invocation semantic

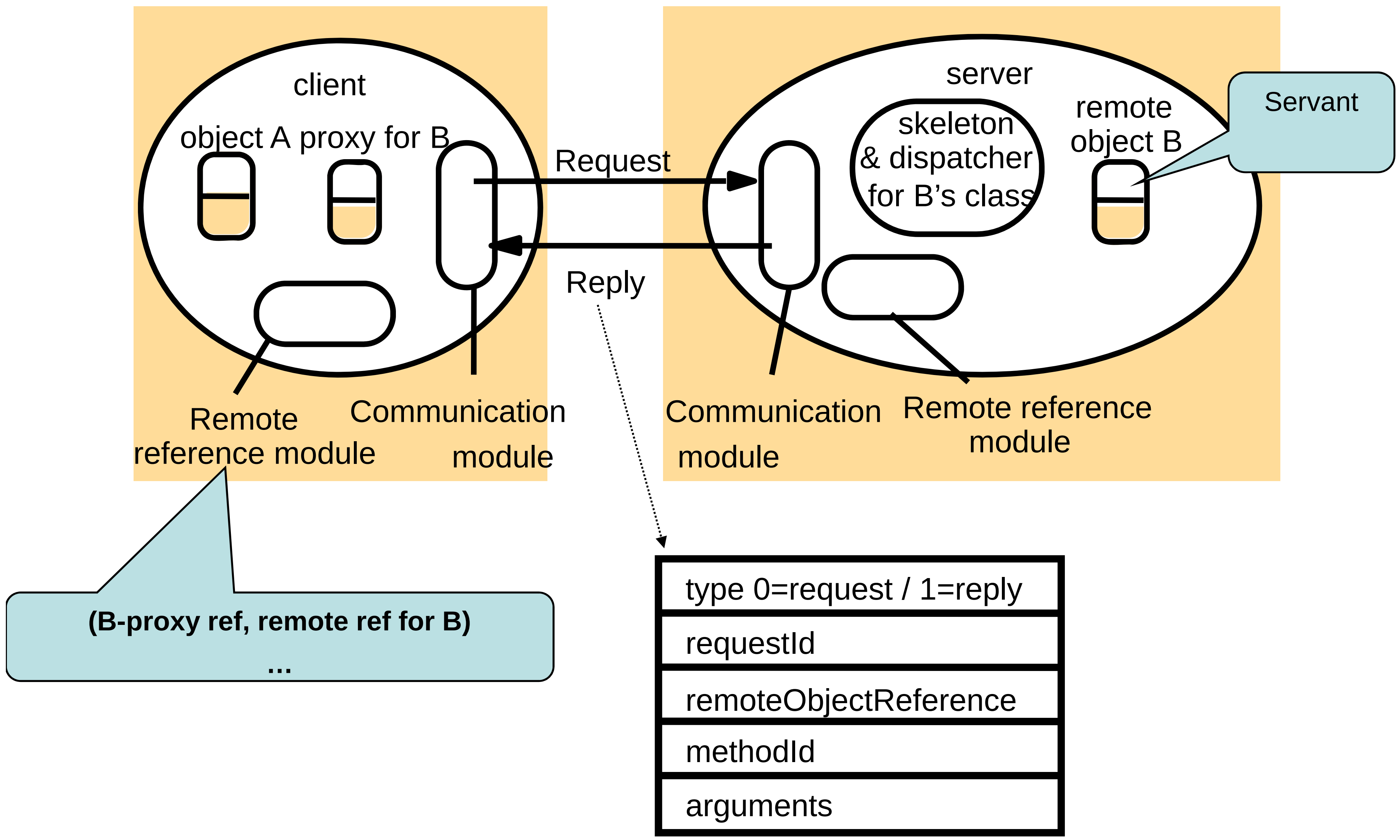
Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Maybe: delivery of video frame

At-least once: Idempotent operations

At-least once: most general, most costly

Implementing RMI



Representation of a remote object reference

Remote Object Reference: uniquely identifies an object system-wide

Fx:

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

Problem: What if objects may migrate?

Implementation of RMI

Proxy:

•Make RMI transparent to calling object

•Hide remote object reference

•Marshall/Unmarshall

Communication Module:

•Implements request/reply protocol

Remote Reference Module

•remote object table

•Translates local and remote references

•Updated dynamically

•(B-proxy-ref, B remote ref)

Dispatcher:

•Inspects request and calls requested method in skeleton

Skeleton:

•Implements methods of remote interface

•unmarshalls, invokes servant, marshalls

Implementing RMI

1. client calls proxy
2. proxy obtains remote reference, updates remote references
3. proxy marshalls parameters
4. proxy forwards request to clients communication module
5. clients communication module sends request to server
6. server communication module at server receives request
7. server communication module forwards request to dispatcher for the requested class
8. dispatcher calls requested method in skeleton
9. skeleton unmarshals parameters, updates remote reference module,
10. skeleton calls servant
11. skeleton marshalls results, and updates remote reference module
12. skeleton forwards reply to server communication module
13. server communication module sends reply
14. clients communication module receives request, forwards it to proxy object
15. proxy unmarshalls
16. proxy updates remote reference module
17. proxy returns to result to client

Distributed Garbage Collection

- Reclaim object when no object/node in the system can reference the object
- GCProtocol, v. 1.0: Reference Counting

client C

1. When creating a new proxy for object o: call ***server.addRef(o)***
2. When destroying a stub, call ***server.removeRef(o)***

server o

1. On ***addref(o)***, increment ***o.count***.
2. On ***removeRef(o),*** decrement ***o.count***
3. Reclaim object when:

   no local references remain

   AND

   o.count is 0

Garbage Collection: Complications

1. Cyclic datastructures
2. What if a client fails without releasing object references?
   1. ***If*** we can detect client-failure: decrement counts, but we must associate counts with unique *clientIDs.*
3. What if an object is reclaimed prematurely due to a transient network failure that heals?
   1. must guarantee that the server detects the dangling reference
   2. requires unique *objectIDs*
4. What if ***addRef*** and ***removeRef*** messages from a given client are delivered out of order?
   1. tag messages with increasing *sequence-numbers*
5. What about races if a last reference is in transit??

Reliable GC: *Client*

•**Garbage Collection Protocol, version 2.0: holders+leasing**

1. When creating a proxy for object o, call ***server.addRef(o,C)***

   Always await acknowledgement for ***addRef*** call before acknowledging receipt of the reference.

2. When destroying a proxy, send ***server.removeRef(o,C)***

   Never destroy a stub until all transmitted references have been acknowledged by their recipients.

3. Resend ***server.addRef(o,C)*** every ***lease interval***.

4. Tag each garbage collection message with:

   (i) a strictly increasing *sequence-number*

   (ii) a clientID ***C*** guaranteed unique across all clients.

Reliable GC: *Server*

•**Garbage Collection Protocol, version 2.0: holders+leasing**

1. On ***addRef(o,C)*** add C to ***o.holder***s

    o.holders shows (*clientID, add-time, sequence#)*

    add-time is the server's time when it received the ***addRef*** request

    sequence# is the client's *sequence-number* recorded in the ***addRef*** request

2. On ***removeRef(o,C)***, remove C from ***o.holders***

    discard ***removeRef*** messages with *sequence-number* < *sequence#* in record

3. Periodically scan ***o.holders***

    if C's *add-time* is older than *lease interval*

        remove *C* from ***o.holders***

4. Reclaim object when ***o.holders*** is empty and no local references exist

JavaRMI

Case study: Java RMI

- Extends the Java object model providing support for distributed objects.
  - Same syntax as for local method invocation
  - Different
    - call semantics
    - parameter passing semantics
    - remote exceptions
- Classes can be downloaded dynamically

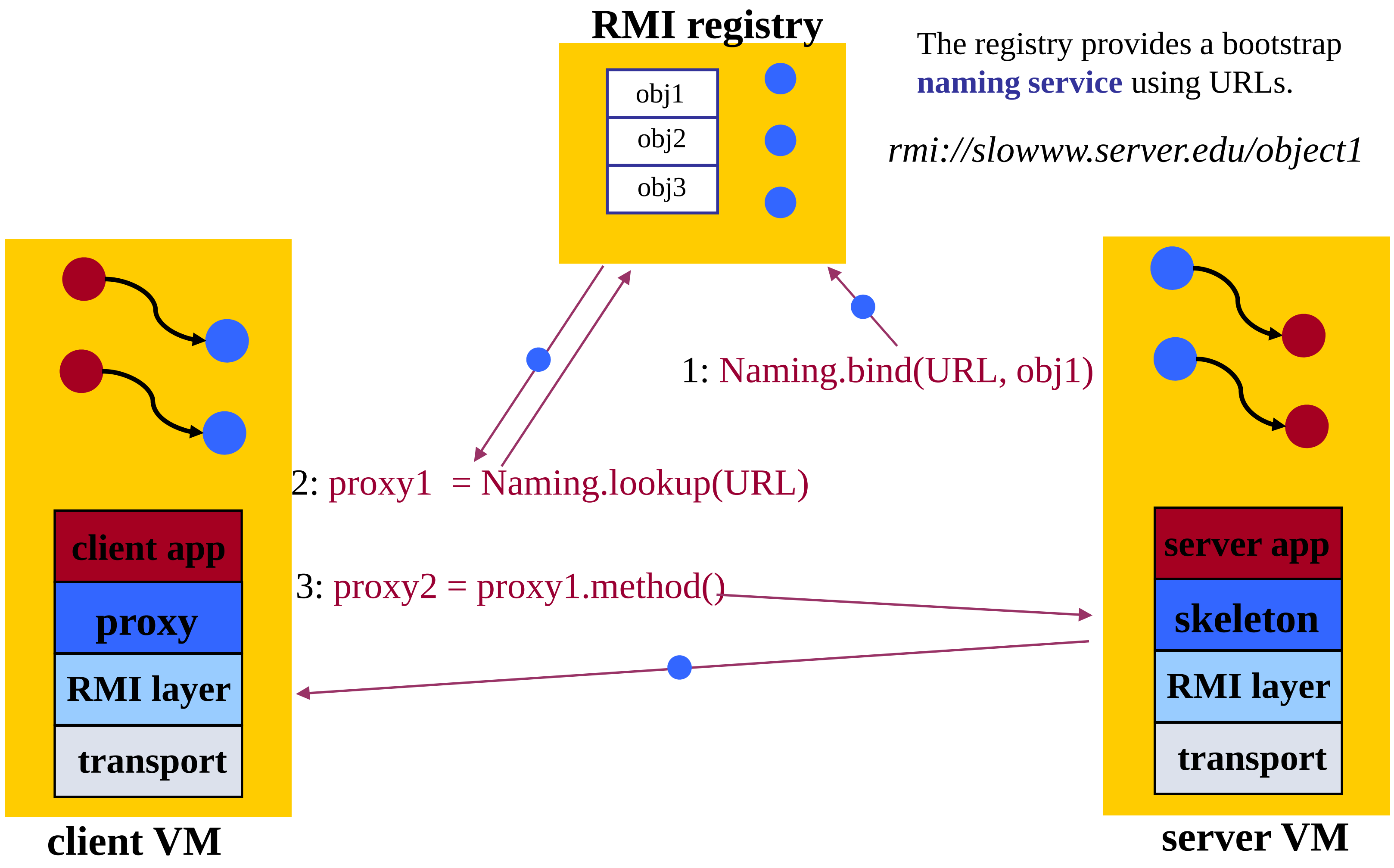
Remote Objects

- Remote interfaces defined by extending the *`Remote`* interface.
- “Remote objects” (servants) =def implements `Remote` interface
- All methods must throw **`RemoteException`**
- Corollary: because the visible parts of a remote object are defined through a Java interface, constructors, static methods and non-constant fields are *not* remotely accessible (because Java interfaces can’t contain such things).
- the **rmic** compiler (<java 1.6) generates stub-code for classes that implement remote interfaces.

Registry = NameServer

The registry provides a bootstrap **naming service** using URLs.

rmi://slowww.server.edu/object1



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.

Byte Code Instructions for Stubs?

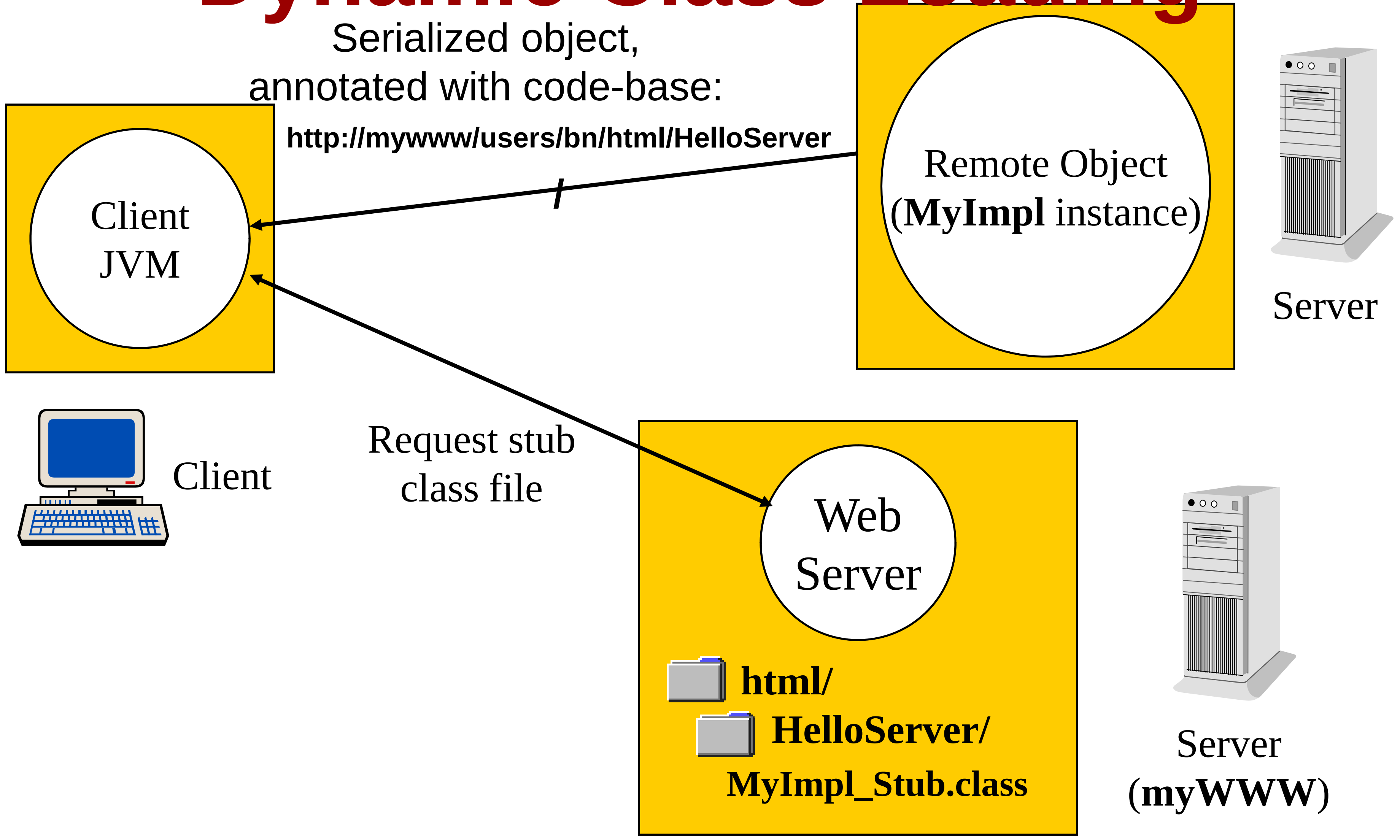
- A client (server) receives a (serialized) object passed by RMI.
- It wants to call a method on the received object
- BUT serialized objects *do not* contain is the actual JVM instructions (the byte codes), that implement *methods* of the received object.
- ⇒ When an object is unserialized, the client JVM must have some way of loading a class file that *does* contain the code
  - If no suitable class file is found it throws **java.lang.ClassNotFoundException**
- **⇒ Dynamic loading of code**

Distributing Class Files

1. Manually copy all class files to (all) client and servers CLASSPATH eg. by ftp
2. Put in shared directory in Network File System (LANs only)
3. JVM can be instructed to automatically fetch code through http
   - publish code at a web-server
   - serialized object contains URL
   - set the *property* **java.rmi.server.codebase** in the JVM where the serialized object originates, eg.

```
java  –Djava.rmi.server.codebase=
        http://mywww/users/bn/html/HelloServer
```

Dynamic Class Loading



Security Managers

- Dynamically loaded code from remote clients (perhaps programmed by other people) cannot / should not be trusted
- Anybody that knows the interface can access the remote object!
- Before a Java application is allowed to download code dynamically, a suitable ***security manager*** and ***security policy*** must be set.
- If no security manager is set, stubs and classes can only be loaded from the local **CLASSPATH**.

1. This command at the start of the program enables dynamic loading

   System.setSecurityManager(new RMISecurityManager()) ;
2. Define the the **java.security.policy** property
   1. **java –Djava.security.policy=policy.all HelloClient**
   2. (or use System.setProperty() in the program)
3. **policy.all is a** text file containing our security policy

Defining a Security Policy

- text file with contents:

```
grant {
    permission  java.security.AllPermission  “”, “” ;
 } ;
```

- This policy allows downloaded code to do essentially *anything* the current user has privileges to do:
  - Read, write and delete arbitrary files; open, read and write to arbitrary Internet sockets; execute arbitrary UNIX/Windows commands on the local machine, etc.
  - It is a dangerous policy if there is any chance you may download code from untrustworthy sources (e.g. the Web).
  - For now you can use this policy, but please avoid dynamically loading code you cannot trust!

RMIC (<java 1.6)

- RMIC stub compiler creates proxy and skeleton code
- Use `RMIC –keep` if you want to see these

```
fire2 [~]:javac examples/RMIShape/ShapeListClient.java
fire2 [~]:javac examples/RMIShape/ShapeListServer.java
fire2 [~]:rmic –keep examples.RMIShape.ShapeListServant
fire2 [~]:rmic –keep examples.RMIShape.ShapeServant
//Stub code contained in ShapeListServant_Stub.java and
//ShapeListServant_Skel.java

//Start Server
borg [~]:rmiregistry&
borg [~]:java -Djava.security.policy=Grant.java examples/RMIShape/ShapeListServer

//start Client
fire2 [~]:java -Djava.security.policy=Grant.java examples/RMIShape/ShapeListClient
Write Rectangle
```

END