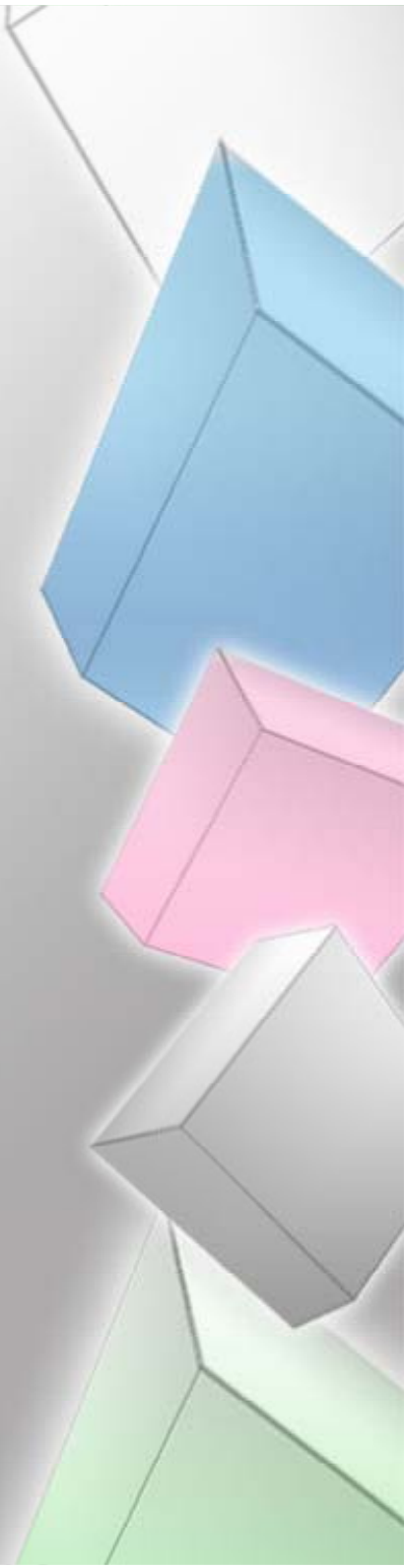
Advanced Topics in Operating Systems

MSc in Computer Science
UNYT-UoG

Dr. Marenglen Biba
8-9-10 January 2010



Lesson 10

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

Distributed Objects

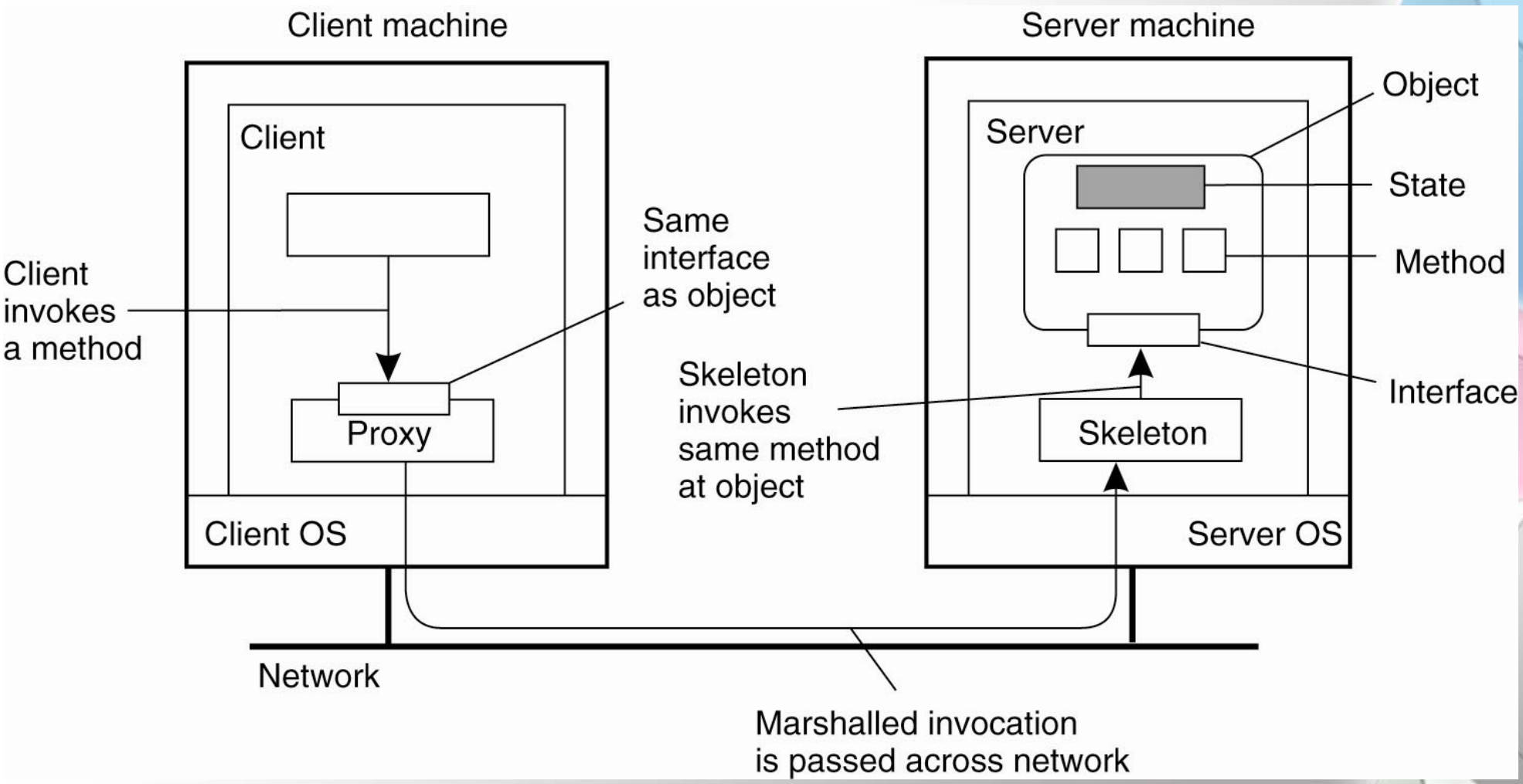


Figure 10-1. Common organization of a remote object with client-side proxy.

Compile-time Vs. Runtime Objects

- Language-level objects such as those supported by Java, C++, or other object-oriented languages, are referred to as compile-time objects.
  - Using compile-time objects in distributed systems often makes it much easier to build distributed applications
  - The obvious drawback of compile-time objects is the dependency on a particular programming language.
- Therefore, an alternative way of constructing distributed objects is to do this explicitly during runtime.
  - This approach is followed in many object-based distributed systems, as it is independent of the programming language in which distributed applications are written.
  - In particular, an application may be constructed from objects written in multiple languages.

Run-time objects

- When dealing with runtime objects, how objects are actually implemented is basically left open.
  - For example, a developer may choose to write a C library containing a number of functions that can all work on a common data file.
- The essence is how to let such an implementation appear to be an object whose methods can be invoked from a remote machine.
- A common approach is to use an object adapter, which acts as a *wrapper* around the implementation with the sole purpose to give it the appearance of an object.

Persistent Vs. Transient Objects

- A persistent object is one that continues to exist even if it is currently not contained in the address space of any server process.
  - In practice, this means that the server that is currently managing the persistent object, can store the object's state on secondary storage and then exit.
  - Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests.
- A transient object is an object that exists only as long as the server that is hosting the object.
  - As soon as that server exits, the object ceases to exist as well.
- To take the discussion away from middleware issues, most object-based distributed systems simply support both types.

Remote Objects

- A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is *not* distributed: it resides at a single machine.
- Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects.

Example: Enterprise Java Beans

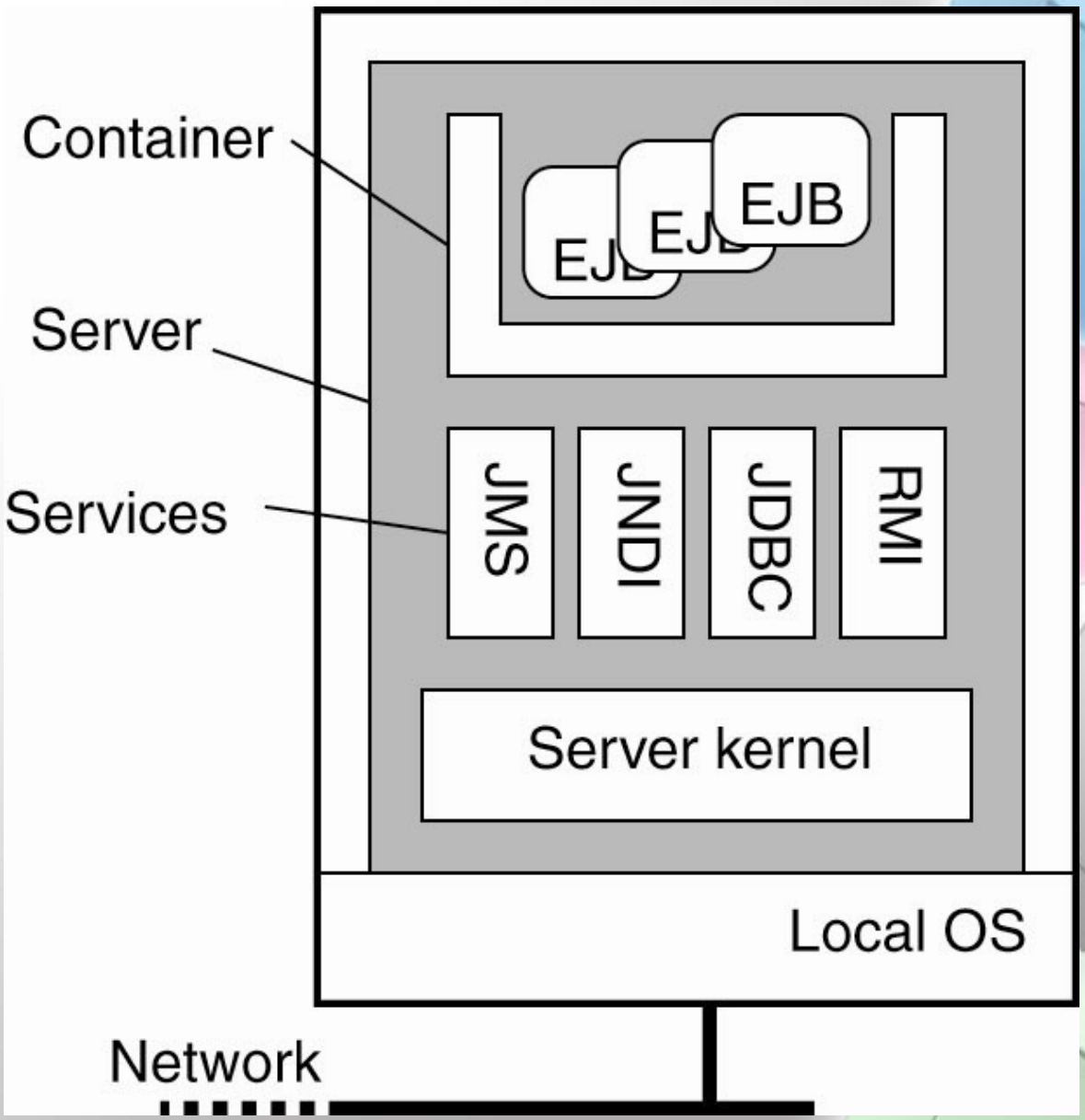


Figure 10-2. General architecture of an EJB server.

Four Types of EJBs

- Stateless session beans
  - Bean implementing a service that lists top-books
- Stateful session beans
  - Shopping cart
- Entity beans
  - Long-lived persistent object: record customer information
- Message-driven beans
  - Publish-subscribe way of communication

Practical Session: EJBs

- What do you need?
  - JAVA SDK
    - JSDK will install also Glassfish Server
  - NetBeans (or Eclipse)

Practical Session: EJBs

- Examples in Netbeans
  - Converter
    - Java Servlets + EJB (stateless bean)
  - Cart
    - EJB (stateful bean)
  - Counter
    - Facelets + EJB (singleton bean)
  - HelloService
    - Web Service + EJB (stateless bean)
  - Timer

Project Proposal 2

- Implement a simple distributed object-based system with a three-tier architecture.
  - Web interface tier: servlet, xhtml
  - Processing tier: EJBs
  - Database tier: MySQL
- Task
  - Read data from a DB and report these in a webpage
- Requirements
  - Programming in Java
  - Knowledge of databases

Globe Distributed Shared Objects

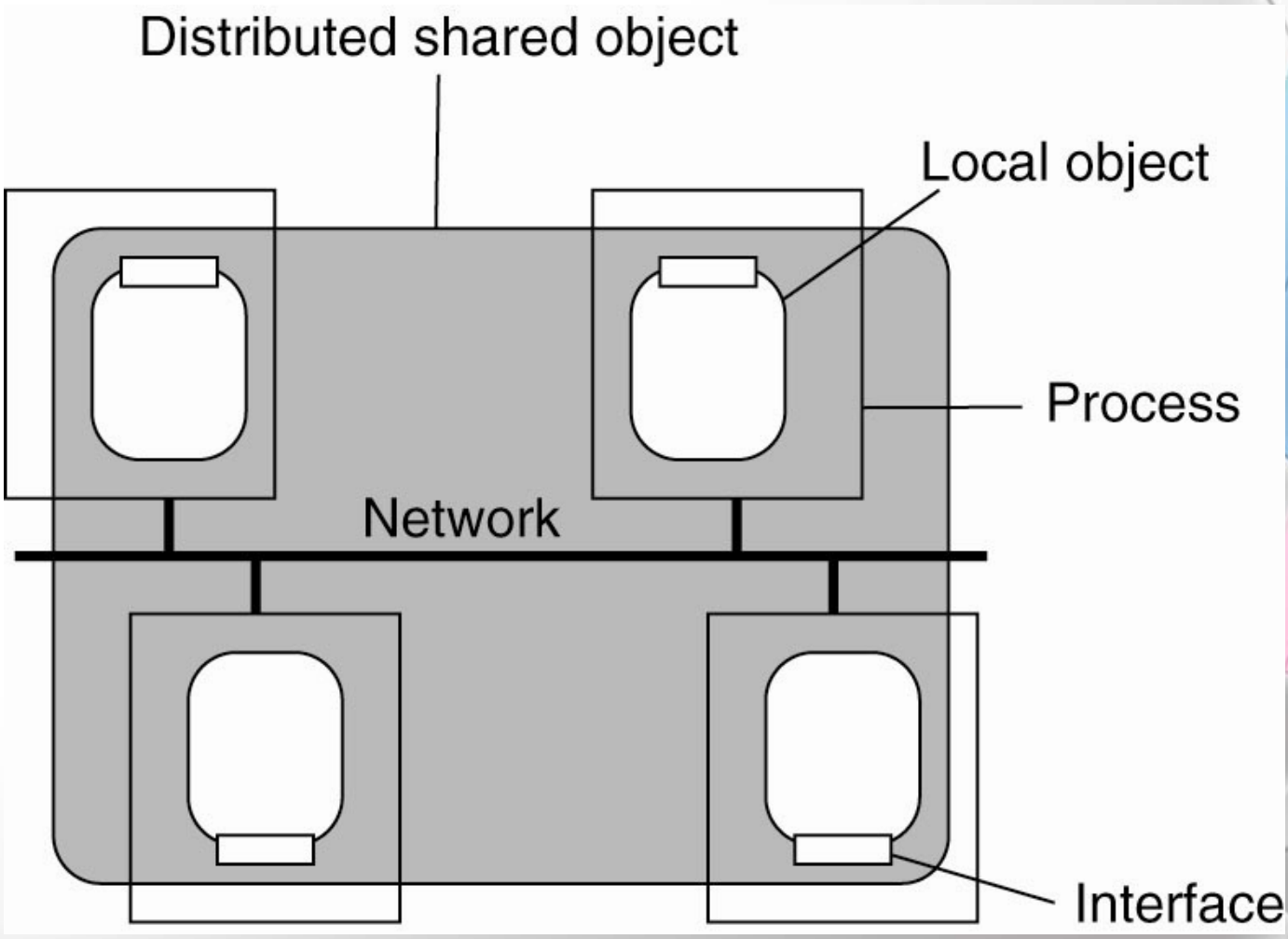


Figure 10-3. The organization of a Globe distributed shared object.

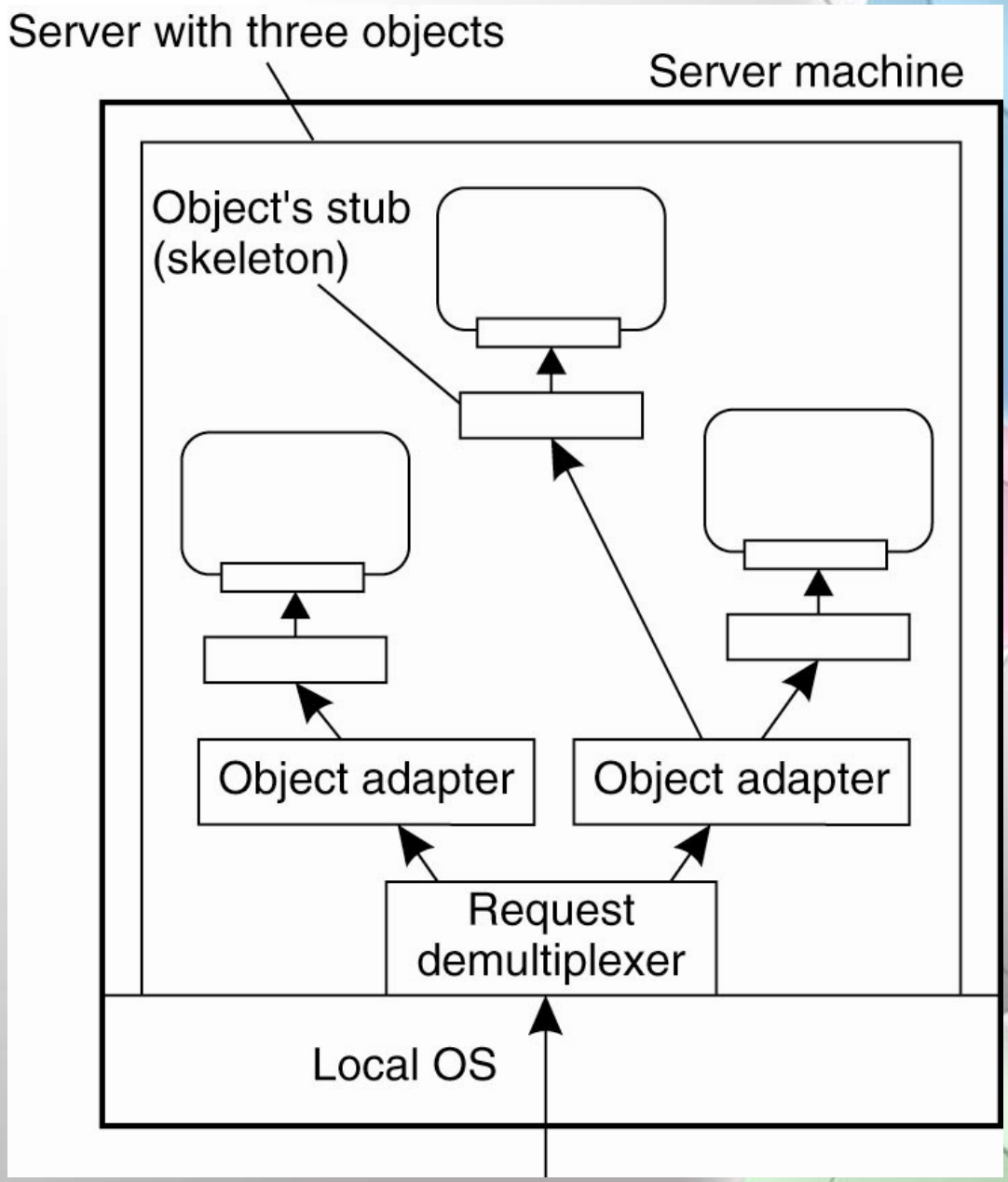
Object Servers

- An object server is a server tailored to support distributed objects.
- The important difference between a general object server and other (more traditional) servers is that an object server by itself does not provide a specific service.
  - Specific services are implemented by the objects that reside in the server.
- Essentially, the server provides only the means to invoke local objects, based on requests from remote clients.
  - As a consequence, it is relatively easy to change services by simply adding and removing objects.

- JBoss Server

Object Adapter

- Figure 10-5. Organization of an object server supporting different activation policies.



Example: The Ice Runtime System

```
main(int argc, char* argv[]) {
      Ice::Communicator      ic;
      Ice::ObjectAdapter     adapter;
      Ice::Object            object;

      ic = Ice::initialize(argc, argv);
      adapter =
            ic->createObjectAdapterWithEnd Points( "MyAdapter","tcp -p 10000");
      object = new MyObject;
      adapter->add(object, objectID);
      adapter->activate();
      ic->waitForShutdown();
}
```

Figure 10-6. Example of creating an object server in Ice.

Communication

- RPC – Remote Procedure Call
- RMI – Remote Method Invocation

- RMI, is very similar to an RPC when it comes to issues such as marshaling and parameter passing.
- An essential difference between an RMI and an RPC is that RMIs generally support system-wide object references

Binding a Client to an Object

```
Distr_object* obj_ref;          // Declare a systemwide object reference
obj_ref = ...;                  // Initialize the reference to a distrib. obj.
obj_ref→do_something( );        // Implicitly bind and invoke a method
```

(a)

```
Distr_object obj_ref;           // Declare a systemwide object reference
Local_object* obj_ptr;          // Declare a pointer to local objects
obj_ref = ...;                  // Initialize the reference to a distrib. obj.
obj_ptr = bind(obj_ref);        // Explicitly bind and get ptr to local proxy
obj_ptr→do_something( );        // Invoke a method on the local proxy
```

(b)

Figure 10-7. (a) An example with implicit binding using only global references. (b) An example with explicit binding using global and local references.

Static and Dynamic RMI

- The usual way to support RMI is to specify the object's interfaces.
  - In Java, stub generation can be handled automatically.
  - The approach of using predefined interfaces is referred to as static invocation.
- Sometimes it is convenient to be able to compose a method invocation at runtime.
  - The application selects at run-time which method at a remote object it will invoke.
  - This approach of method invocation is called dynamic invocation.

Parameter Passing

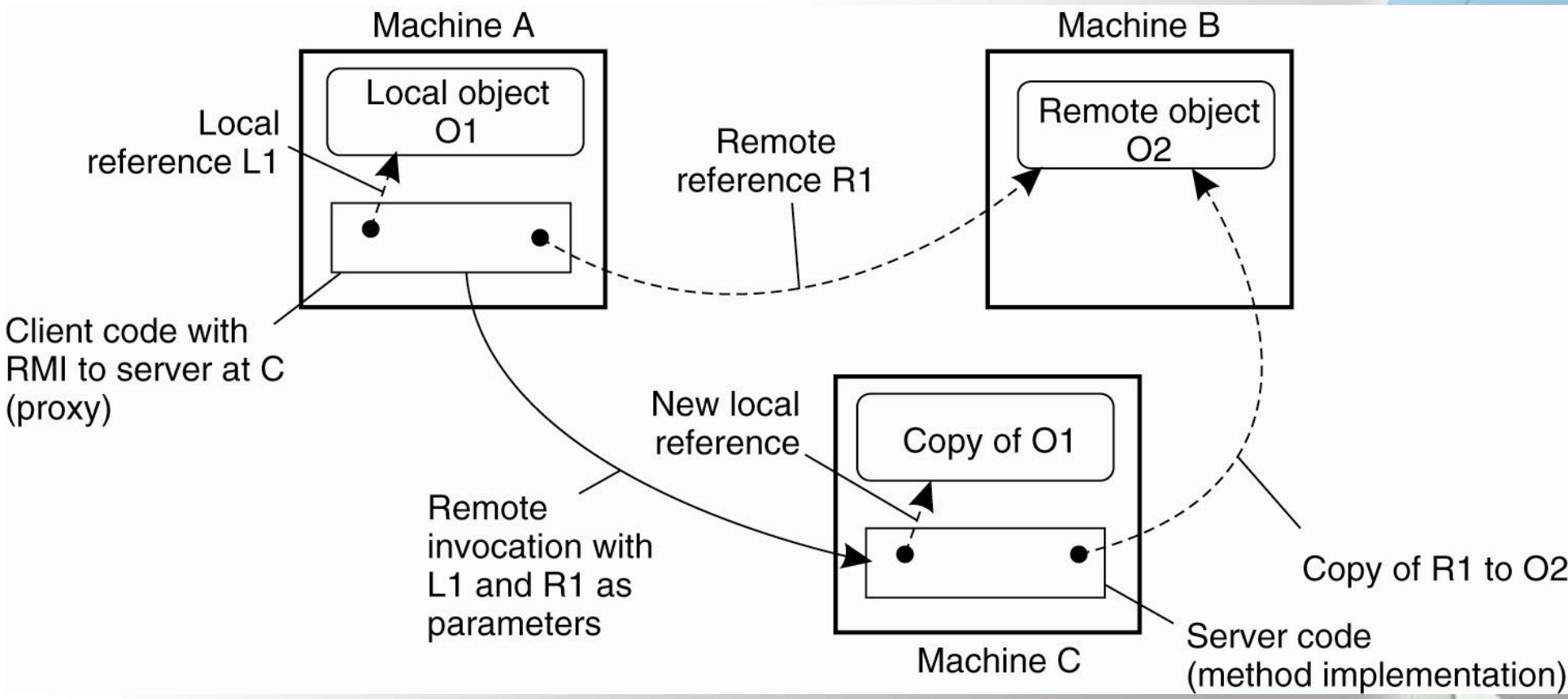


Figure 10-8. The situation when passing an object by reference or by value.

Java RMI

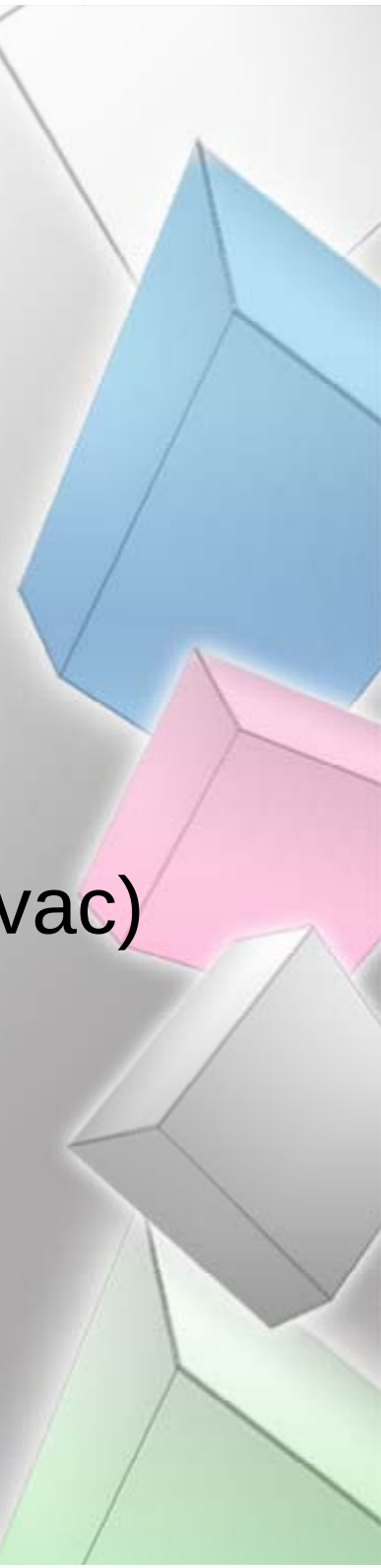
- Java adopts remote objects as the only form of distributed objects.
- Any primitive or object type can be passed as a parameter to an RMI, provided only that the type can be marshaled. In Java terminology, this means that it must be **serializable.**
- The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference.
- In an RMI, a local object is passed by making a copy of it, while a remote object is passed by means of a system-wide object reference.

RMI

- Server class
- Server side stub (or Skeleton)
- Client Class
- Client side stub (or Proxy)

Practical Session: RMI

- Client-Server
  - Examples in Eclipse using sockets
    - Server that tells the time
    - Chat
- RMI
  - Example in Netbeans (or simply with javac)
    - File Transfer Application



Project Proposal 3

- Implement a simple distributed object-based system with a three-tier architecture and RMI
  - 1st tier: Client.java
  - Processing tier: Server.java
  - Database tier: MySQL
- Task
  - Read data from a DB and report these back to the client
- Requirements
  - Programming in Java
  - Knowledge of databases

Object-based Messaging: CORBA

- CORBA is a well-known specification for distributed systems.
- CORBA messaging provides an interesting way of combining method invocation and message-oriented communication.
- What makes messaging in CORBA different from other systems is its inherent object-based approach to communication.
- In particular, the designers of the messaging service needed to retain the model that all communication takes place by invoking an object.
- In the case of messaging, this design constraint resulted in two forms of asynchronous method invocations (in addition to other forms that were provided by CORBA as well).

Object-Based Messaging

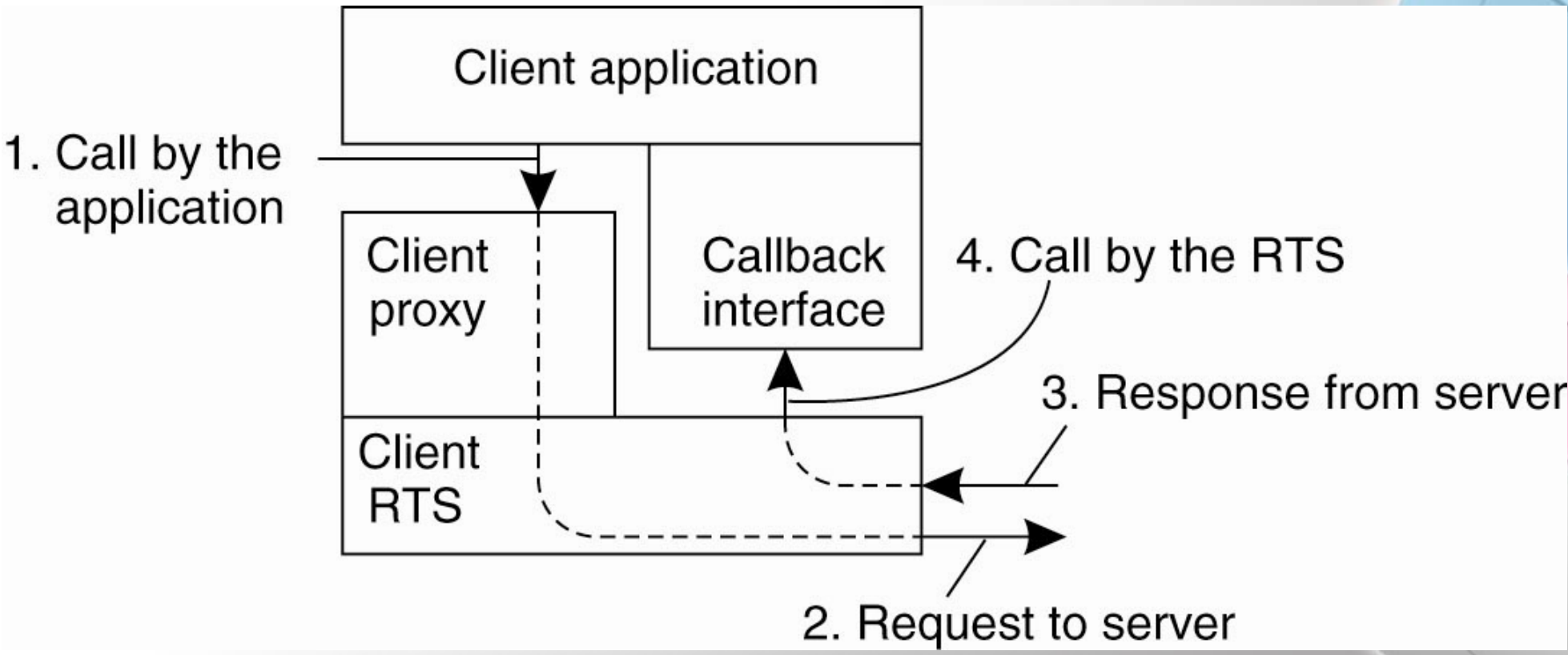


Figure 10-9. CORBA's callback model for asynchronous method invocation.

Object-Based Messaging (2)

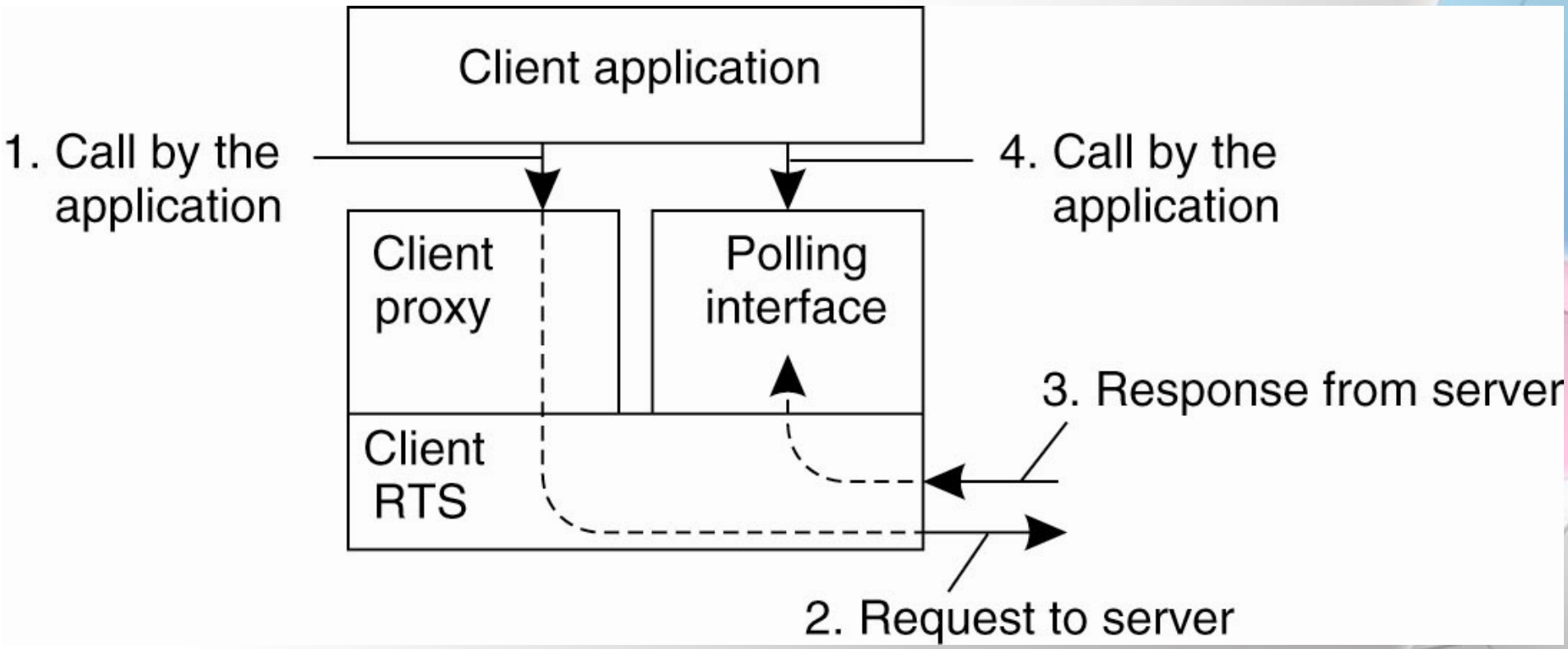
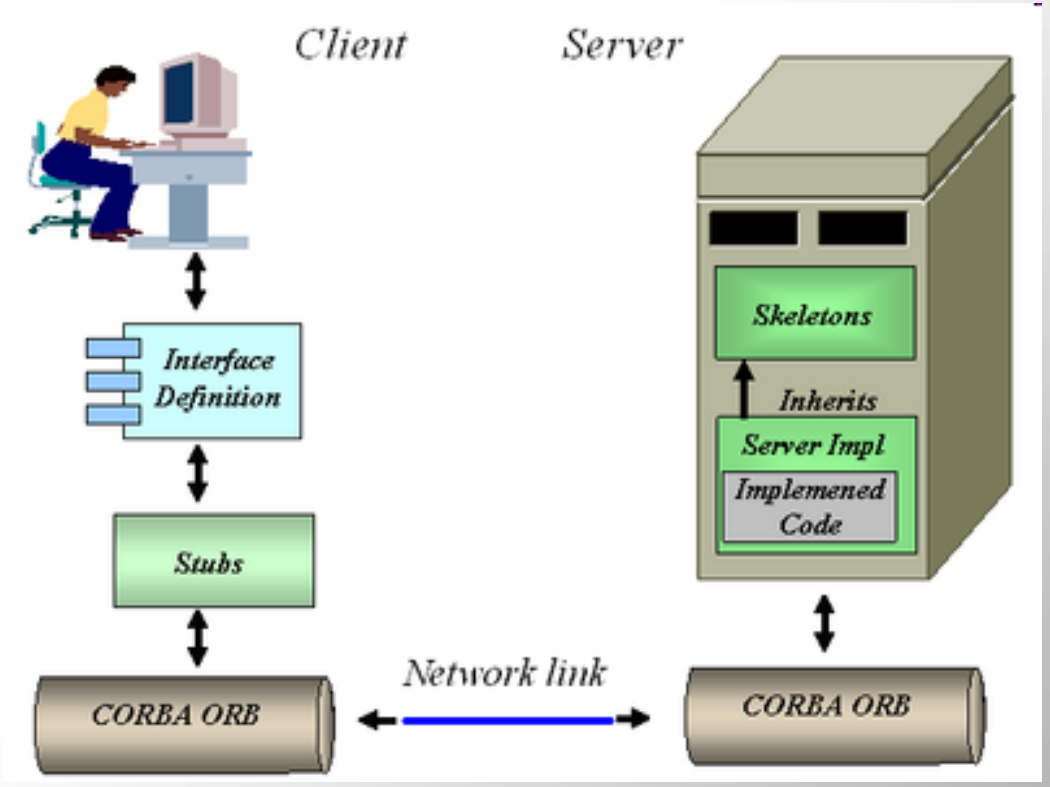
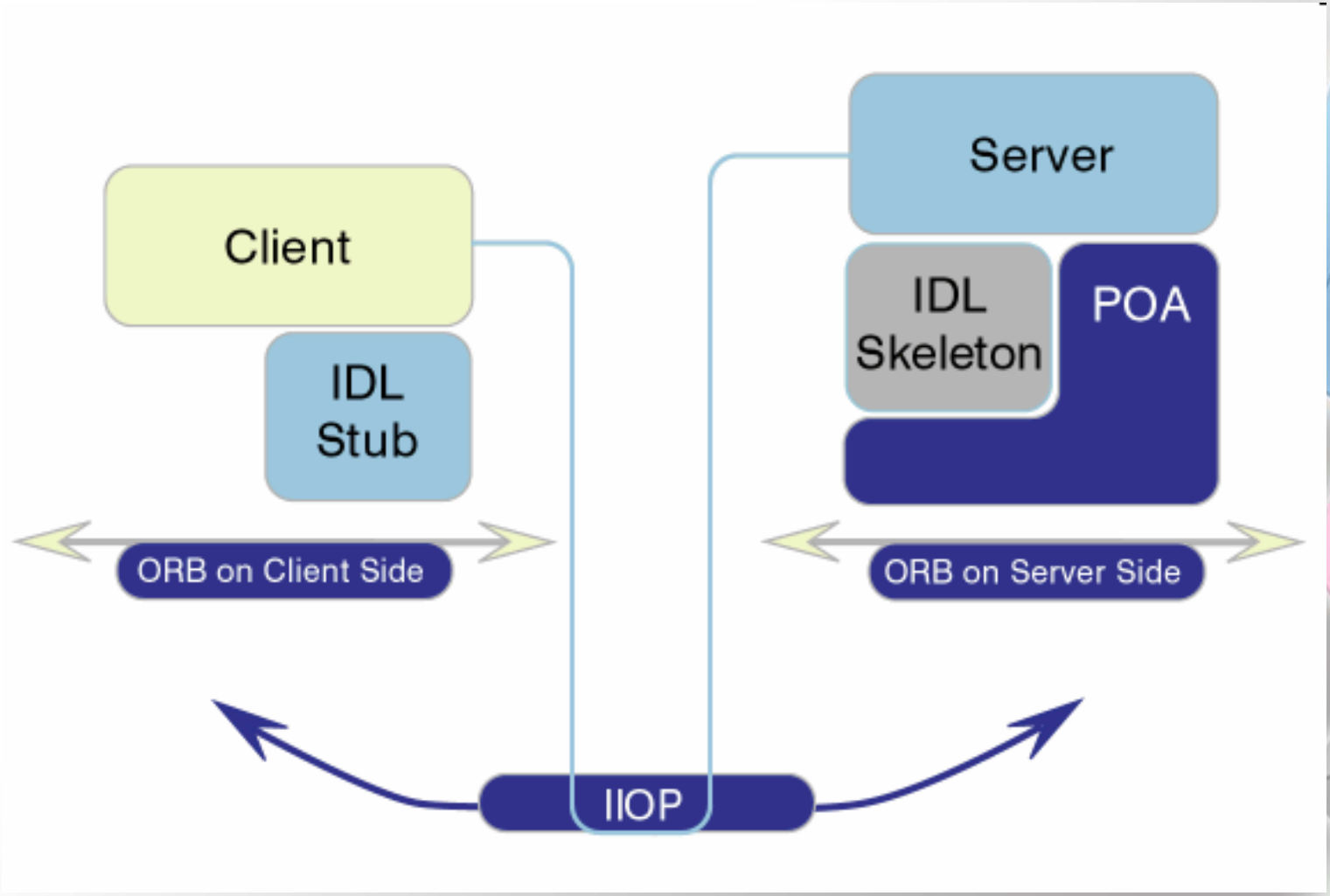


Figure 10-10. CORBA's polling model for asynchronous method invocation.

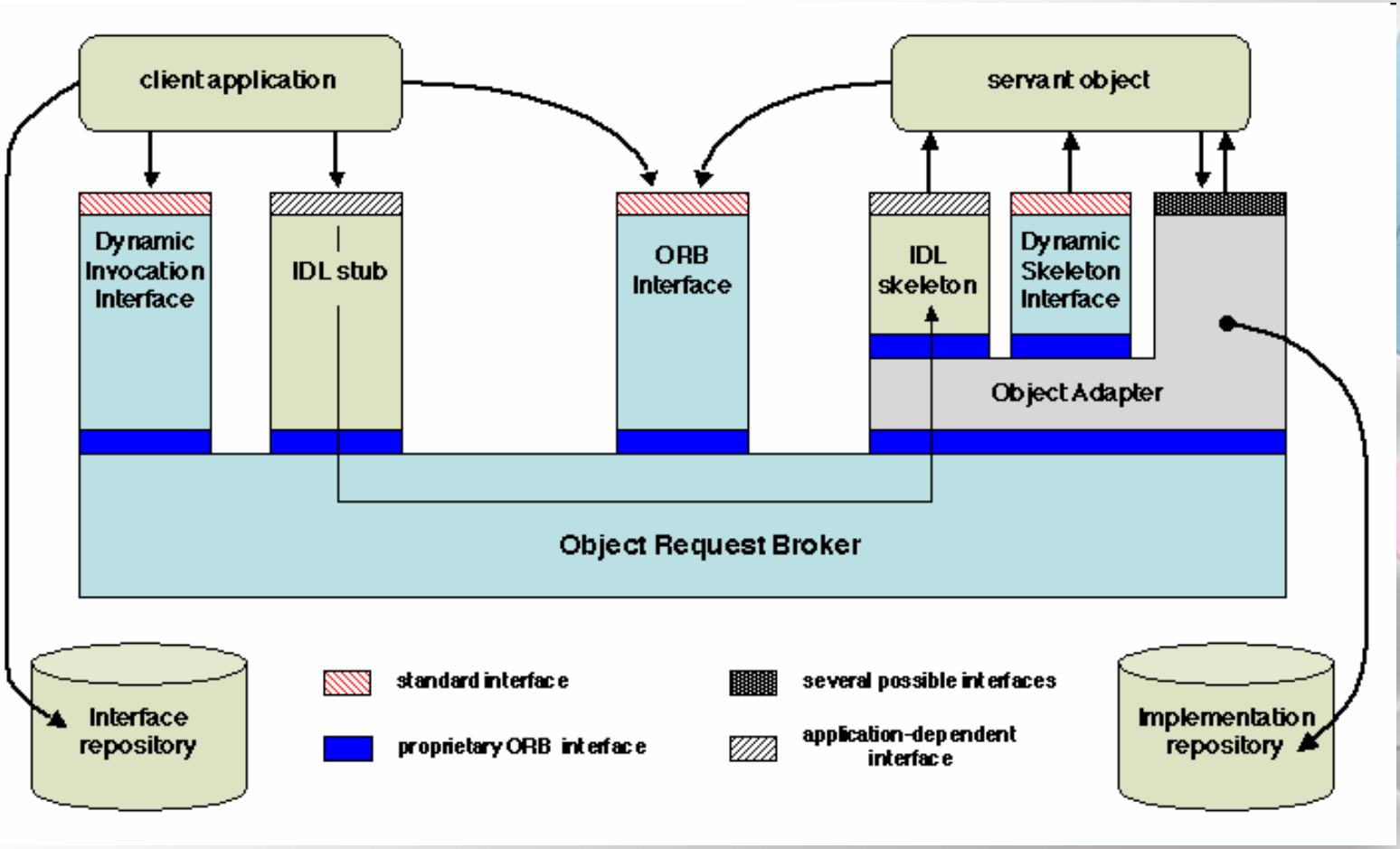
CORBA: simple schema



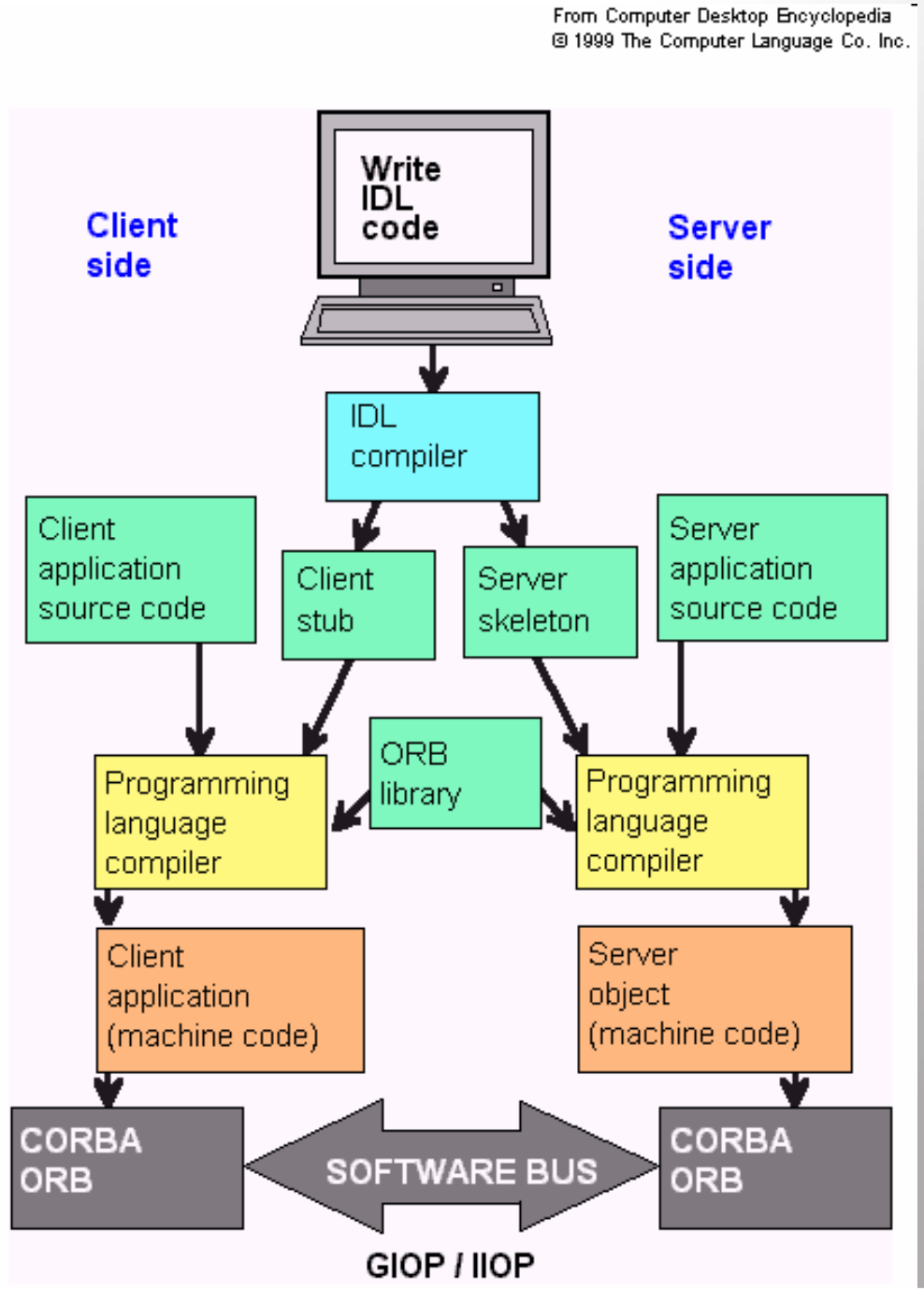
CORBA components



CORBA: general architecture



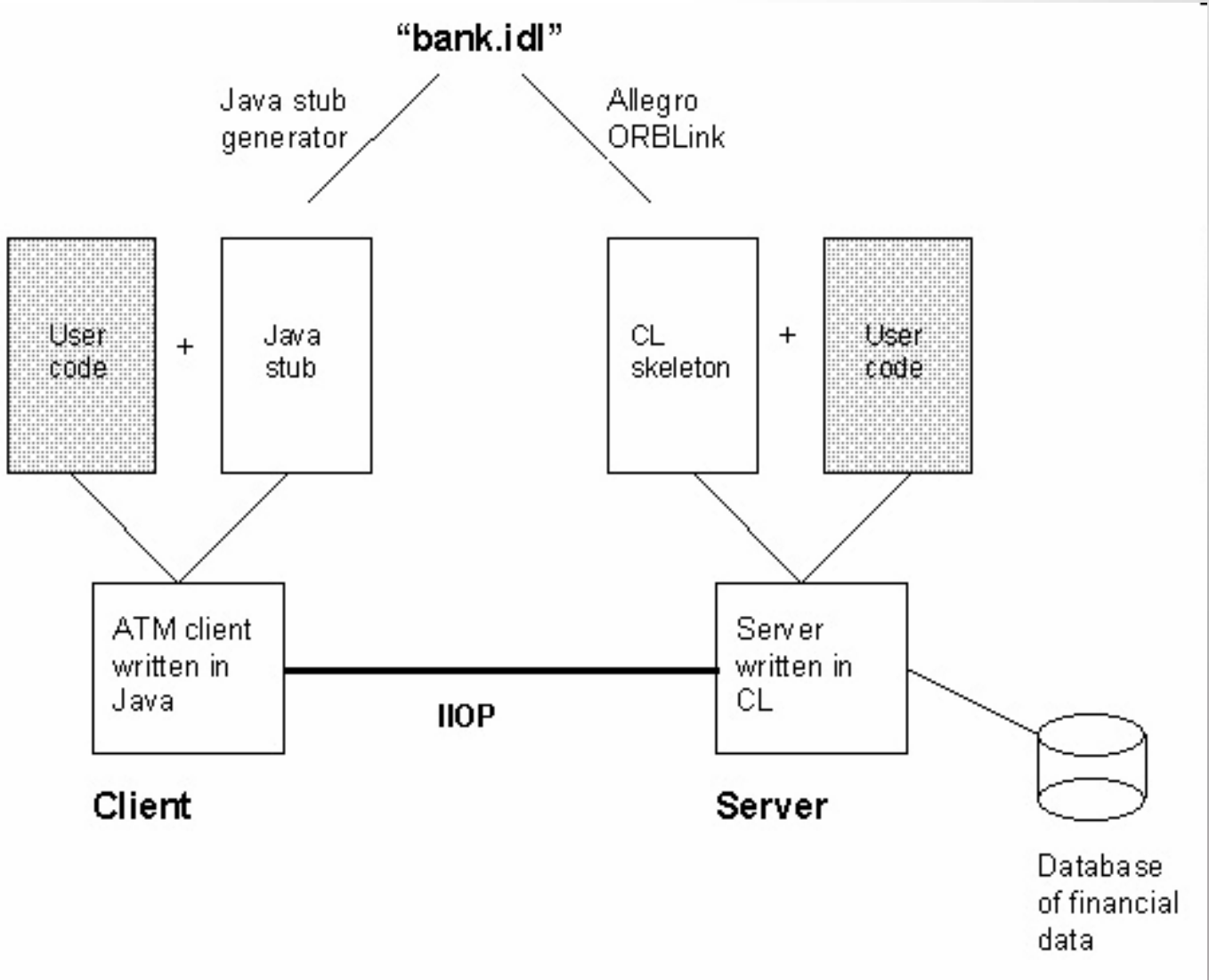
CORBA components



Practical Session: CORBA

- Applications
  - Helloworld
  - Multiplication of Arrays
  - Time teller

Project Proposal 4



Project Proposal 4

- Implement a simple CORBA distributed object-based system with a three-tier architecture
  - 1st tier: Client.java
  - Processing tier: Server.java
  - Database tier: MySQL
- Task
  - Read data from a DB and report these back to the client
- Requirements
  - Programming in Java
  - Knowledge of databases

JMS API

- Java has its own Java Messaging Service (JMS) which is again very similar to what we have seen here for CORBA.
- First you need to know Coordination-based systems.

Naming: Object References

- One of the problems that early versions of CORBA had was that each implementation could decide on how it represented an object reference.
- Consequently, if process *A* wanted to pass a reference to process *B* as described above, this would generally succeed only if both processes were using the same CORBA implementation.
- Current CORBA systems all support the same **language-independent** representation of an object reference, which is called an **Interoperable Object Reference or IOR**.

CORBA Object References

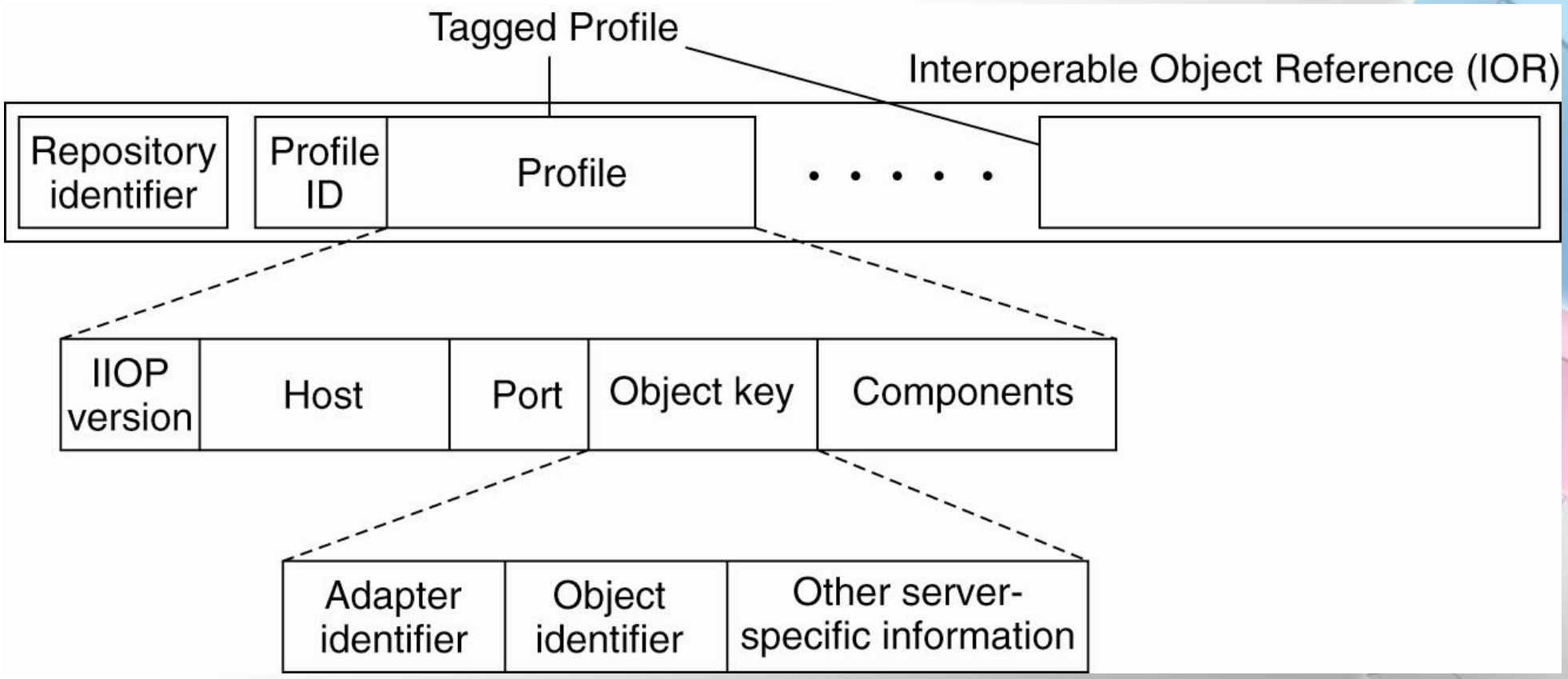
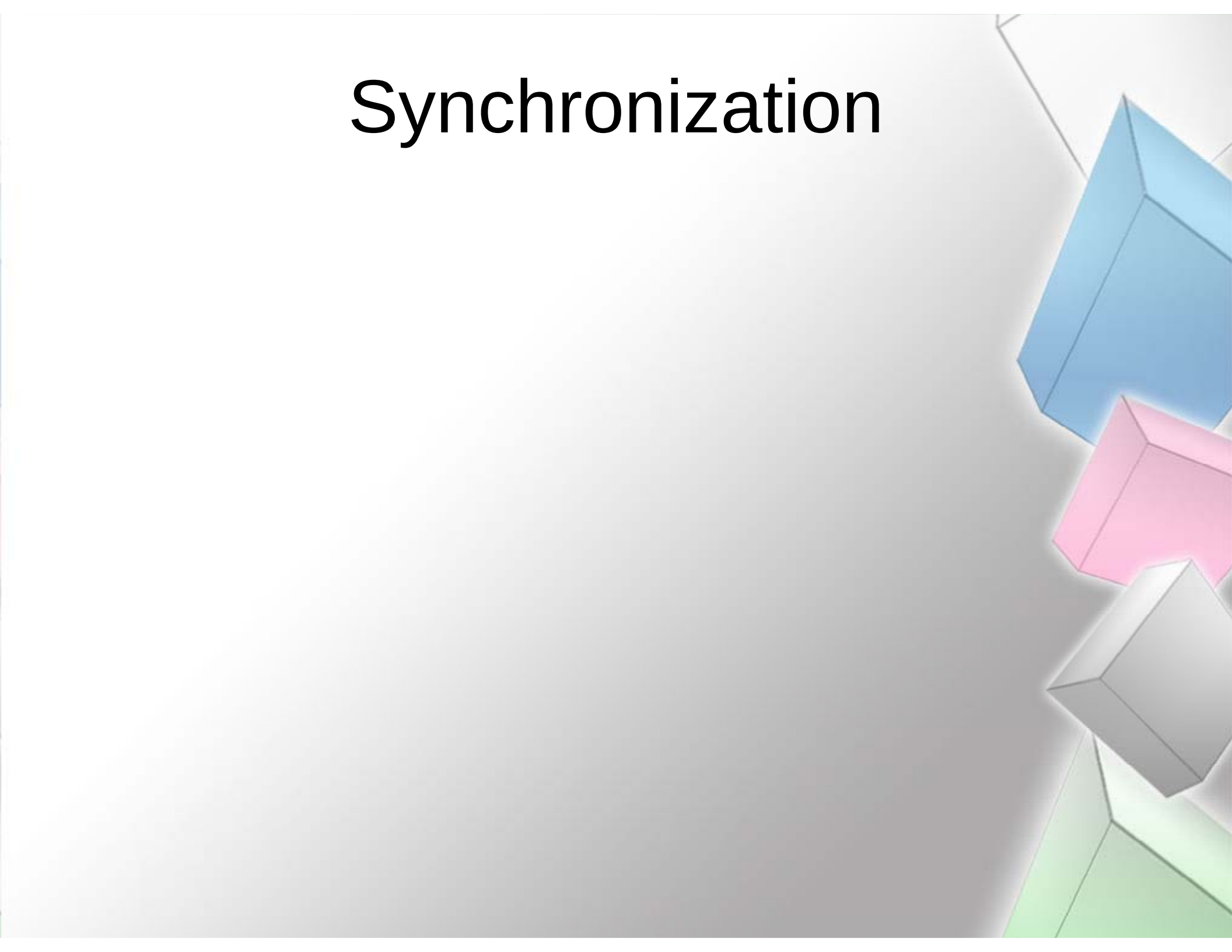


Figure 10-11. The organization of an IOR with specific information for Internet Inter-ORB Protocol (IIOP). Different *Tagged Profiles* for different protocols supported by the object server.

Synchronization



Synchronization

- The fact that, in DOBS, implementation details are hidden behind interfaces may cause problems: when a process invokes a (remote) object, it has no knowledge whether that invocation will lead to invoking other objects.
- As a consequence, if an object is protected against concurrent accesses, we may have a cascading set of locks that the invoking process is unaware of.

Synchronization

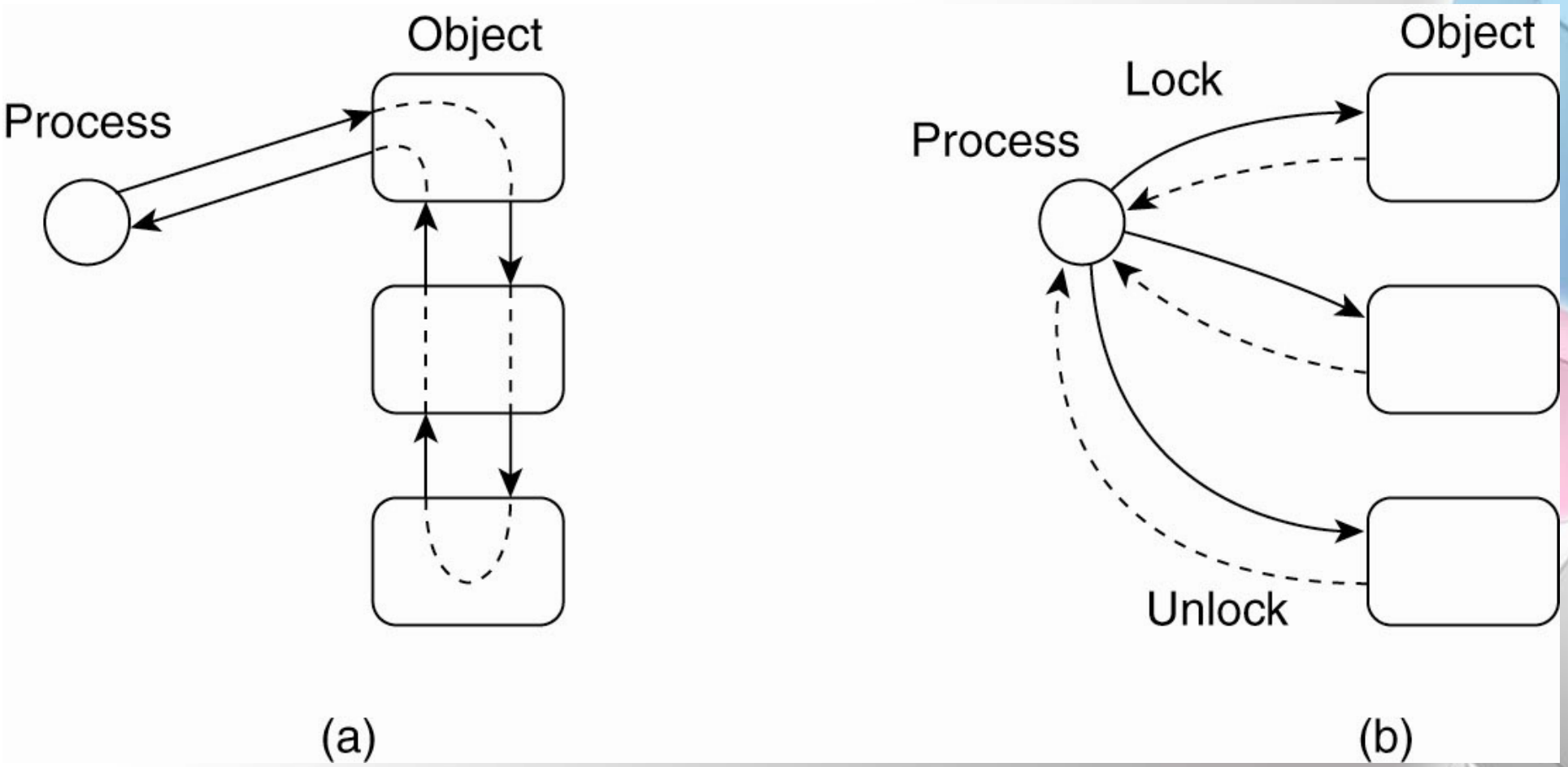


Figure 10-14. Differences in control flow for locking objects. b) the process exerts more control: can give up locks when it believes deadlock has occurred.

Synchronization at object-servers

- In object-based distributed systems it is therefore important to know where and when synchronization takes place.
- An obvious location for synchronization is at the object server.
  - If multiple invocation requests for the same object arrive, the server can decide to serialize those requests (and possibly keep a lock on an object when it needs to do a remote invocation itself).

- However, letting the object server maintain locks complicates matters in the case that invoking clients crash. For this reason, locking can also be done at the client side, an approach that has been adopted in Java.

Fault Tolerance

Example: Fault-Tolerant CORBA

The basic approach for dealing with failures in CORBA is to replicate objects into **object groups.**

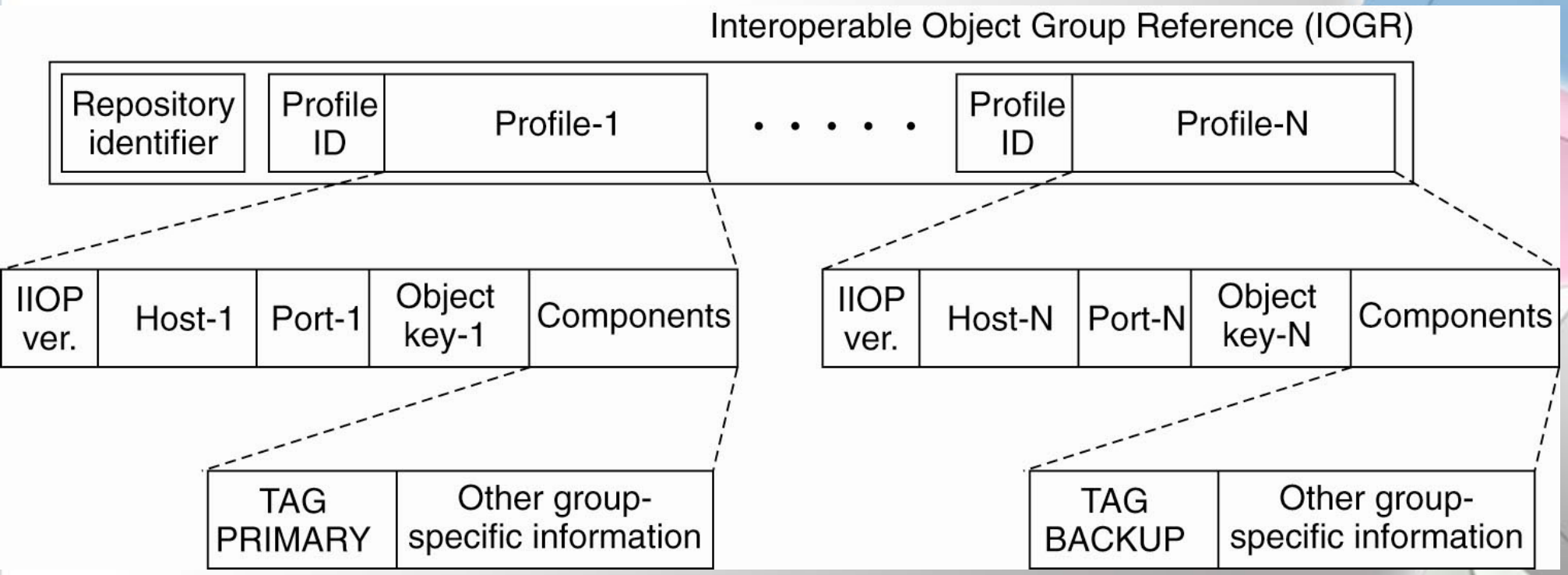


Figure 10-19. A possible organization of an IOGR for an object group having a primary and backups.

An Example Architecture

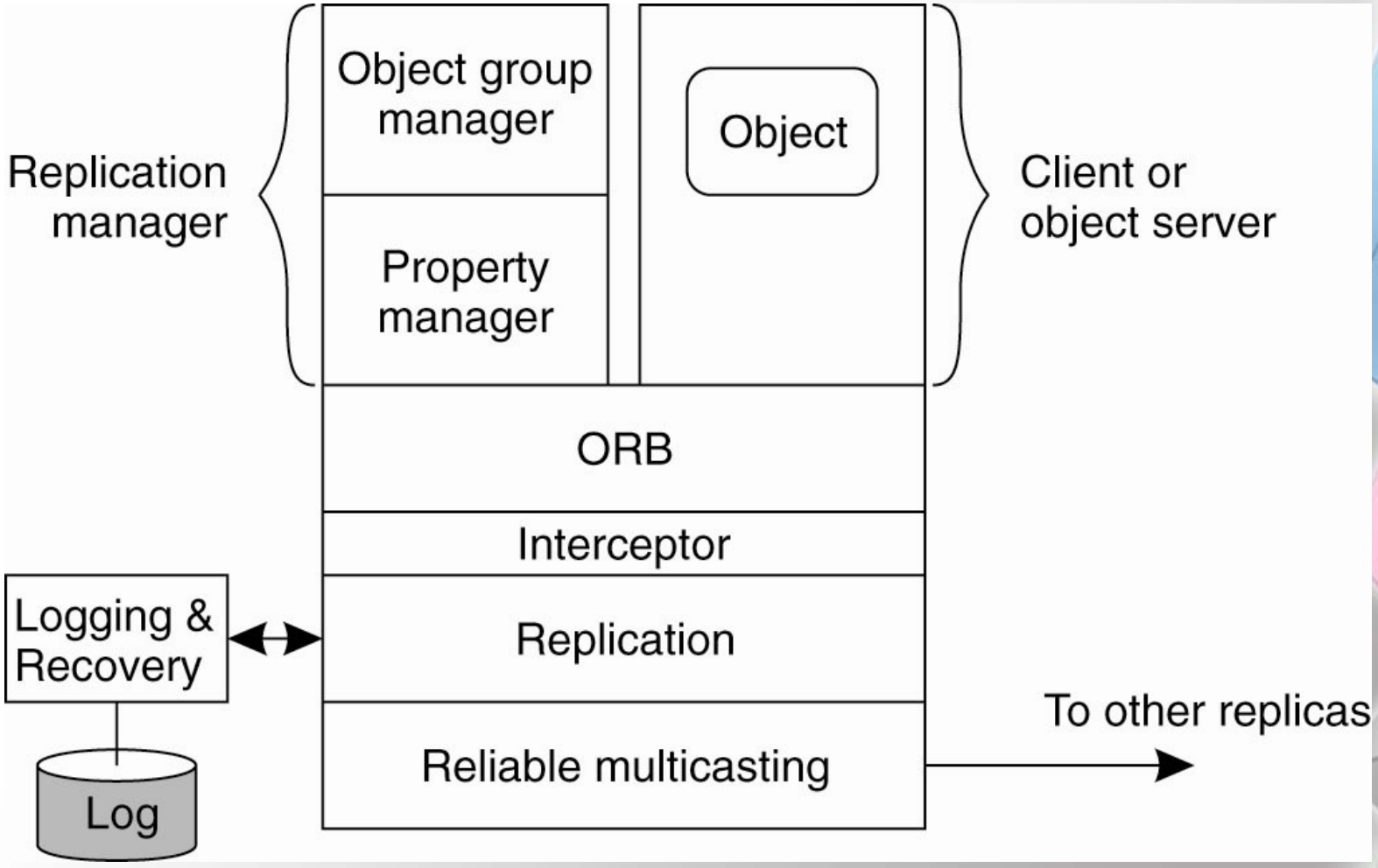


Figure 10-20. An example architecture of a fault-tolerant CORBA system.

Fault-Tolerant Java

Ensure that the Java virtual machine can be used for active replication.

- Active replication essentially dictates that the replica servers execute as deterministic finite-state machines

Causes for nondeterministic behavior of JVM

1. JVM can execute native code, that is, code that is external to the JVM and provided to the latter through an interface.
2. Input data may be subject to nondeterminism.
3. In the presence of failures, different JVMs will produce different output revealing that the machines have been replicated.

Solution: Let servers run according to a primary-backup scheme

Security

- When we consider the general case of invoking a method on a remote object, there are at least two issues that are important from a security perspective:

(1) is the caller invoking the correct object and

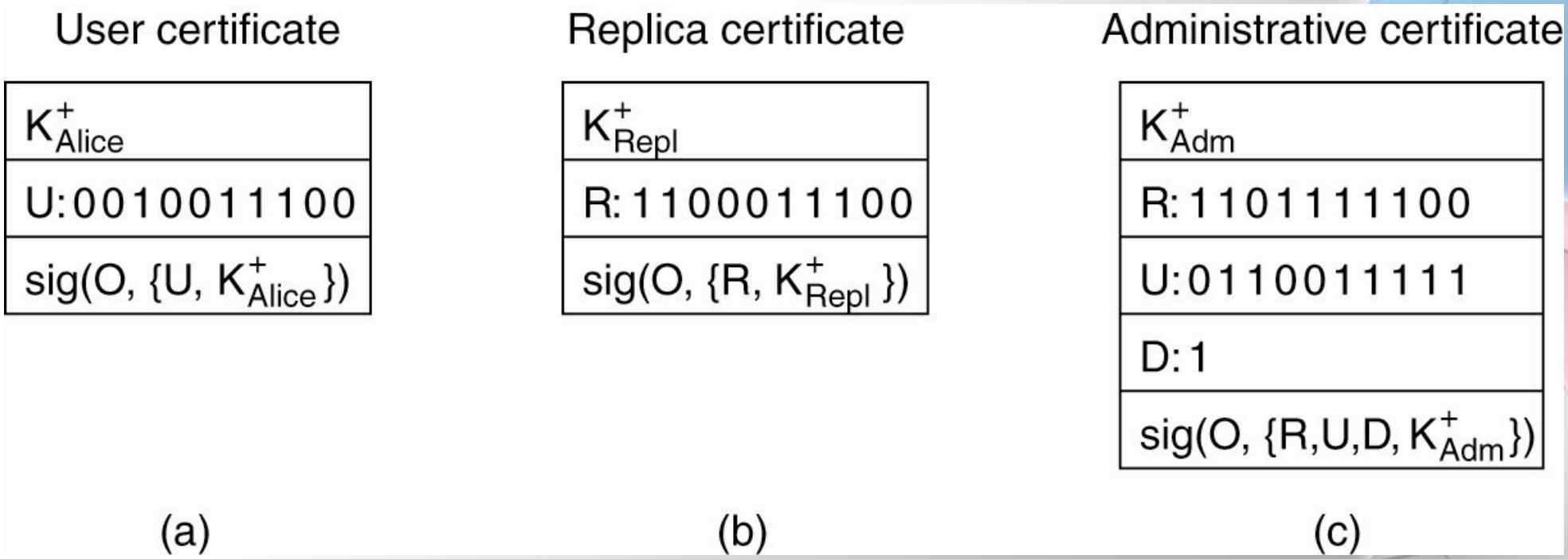
(2) is the caller allowed to invoke that method.

We refer to these two issues as:

secure object binding => authentication

secure method invocation => authorization

Overview of Globe Security



- Figure 10-21. Certificates in Globe: (a) a user certificate, (b) a replica certificate, (c) an administrative certificate.

Secure Method Invocation (1)

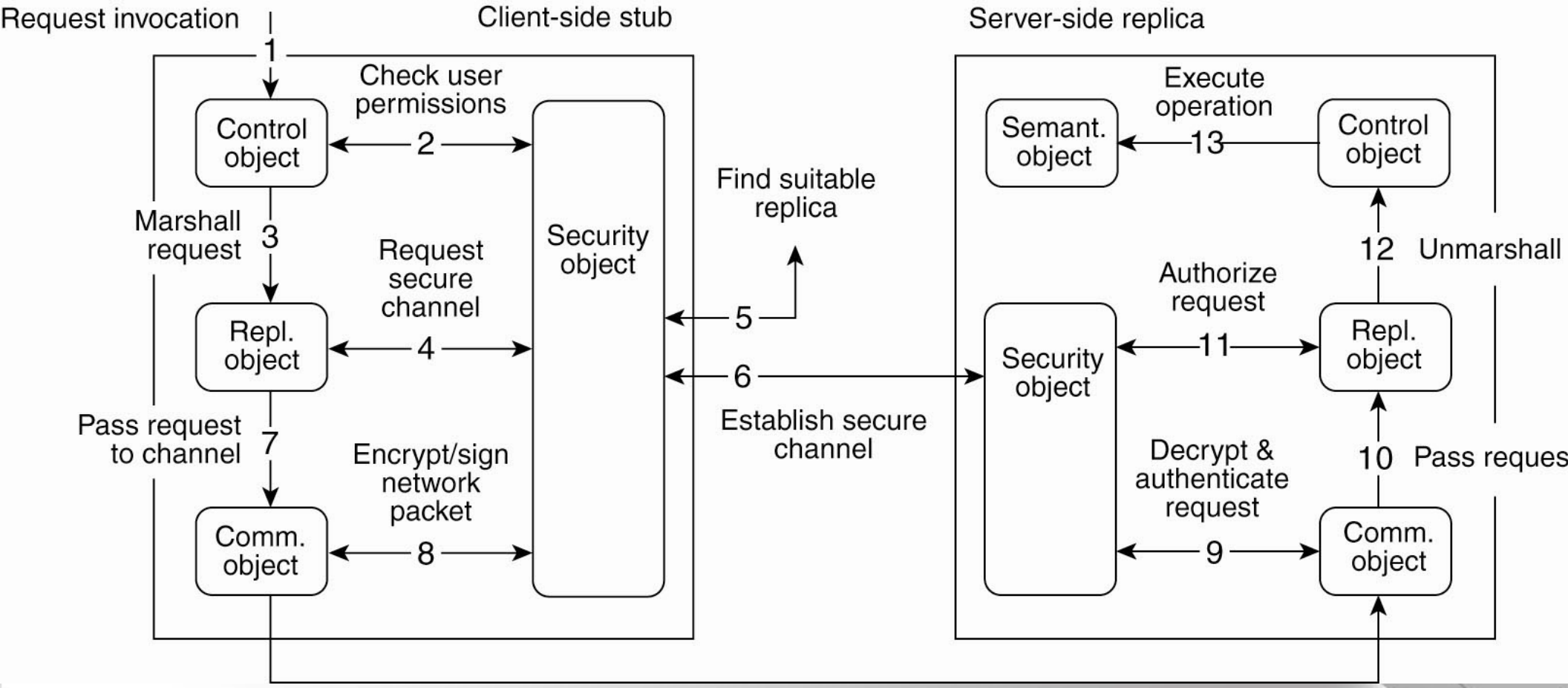


Figure 10-22. Secure method invocation in Globe.

Secure Method Invocation (2)

Steps for securely invoking a method of a Globe object:

1. Application issues a invocation request by locally calling the associated method
2. Control subobject checks the user permissions with the information stored in the local security object.
3. Request is marshaled and passed on.
4. Replication subobject requests the middleware to set up a secure channel to a suitable replica.

Secure Method Invocation (3)

5. Security object first initiates a replica lookup.
6. Once a suitable replica has been found, security subobject can set up a secure channel with its peer, after which control is returned to the replication subobject.
7. Request is now passed on to the communication subobject.
8. Subobject encrypts and signs the request so that it can pass through the channel.

Secure Method Invocation (4)

9. After its receipt, the request is decrypted and authenticated.
10. Request then passed on to the server-side replication subobject.
11. Authorization takes place:
12. Request is then unmarshaled.
13. Finally, the operation can be executed.

Remote objects: Authentication Proxy

- To authenticate the client, a separate authenticator is used.
- When a client is looking up the remote object, it will be directed to this authenticator from which it downloads an **authentication proxy**.
  - This is a special proxy that offers an interface by which the client can have itself authenticated by the remote object.
  - If this authentication succeeds, then the remote object (or actually, its object server) will pass on the actual proxy to the client.

End of Lesson 10

- Readings
  - Distributed Systems, Chapter 10, Except 10.4.2, 10.6,