Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

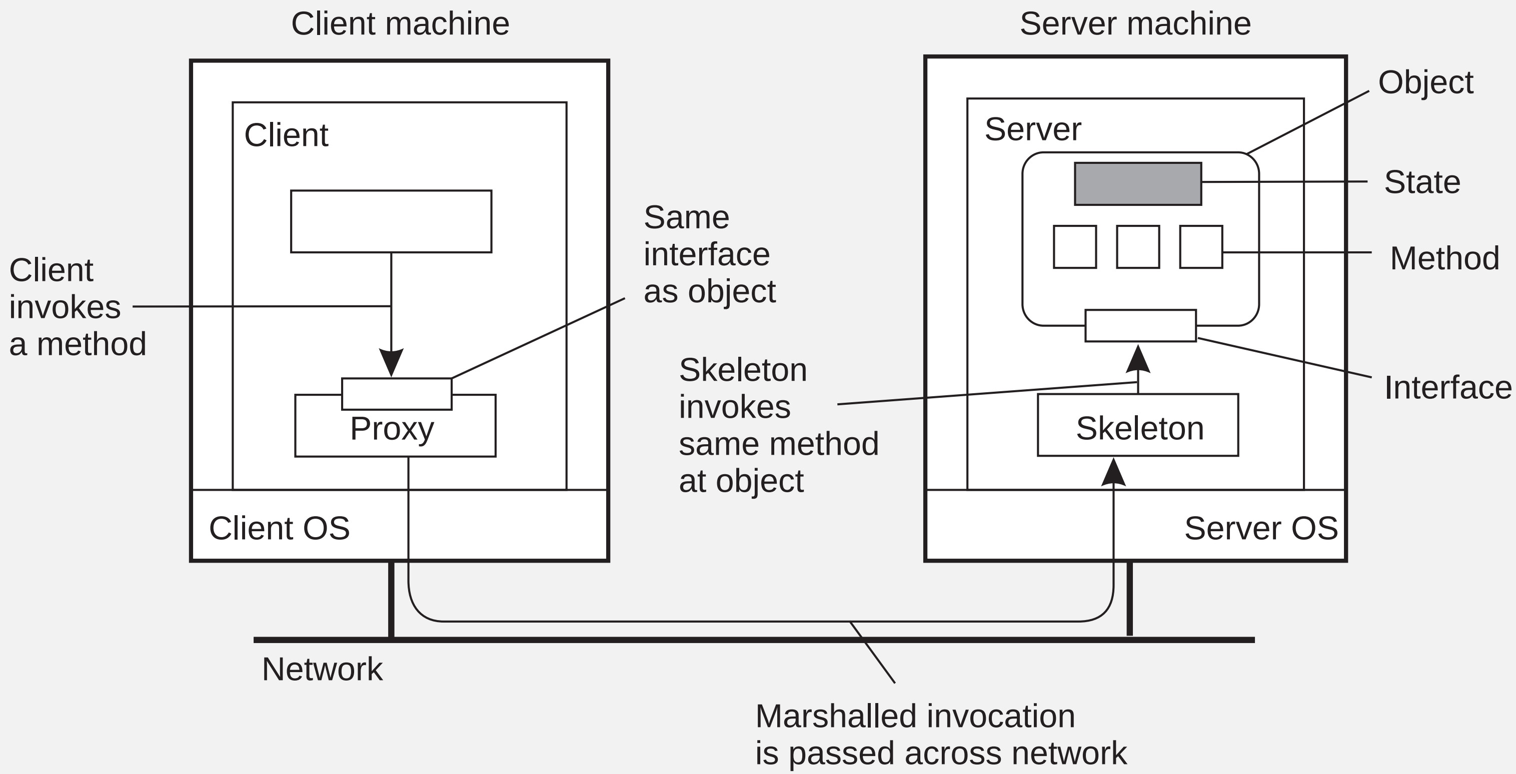
Chapter 10: Distributed Object-Based Systems

Version: December 10, 2012

vrije Universiteit amsterdam

Remote distributed objects

- Data and operations encapsulated in an object
- Operations implemented as methods grouped into interfaces
- Object offers only its interface to clients
- Object server is responsible for a collection of objects
- Client stub (proxy) implements interface
- Server skeleton handles (un)marshaling and object invocation



Remote distributed objects

Types of objects I

- Compile-time objects: Language-level objects, from which proxy and skeletons are automatically generated.
- Runtime objects: Can be implemented in any language, but require use of an object adapter that makes the implementation *appear* as an object.

Types of objects II

- Transient objects: live only by virtue of a server: if the server exits, so will the object.
- Persistent objects: live independently from a server: if a server exits, the object's state and code remain (passively) on disk.

Processes: Object servers

Servant

The actual implementation of an object, sometimes containing only method implementations:

- Collection of C or COBOL functions, that act on structs, records, database tables, etc.
- Java or C++ classes

Skeleton

Server-side stub for handling network I/O:

- Unmarshalls incoming requests, and calls the appropriate servant code
- Marshalls results and sends reply message
- Generated from interface specifications

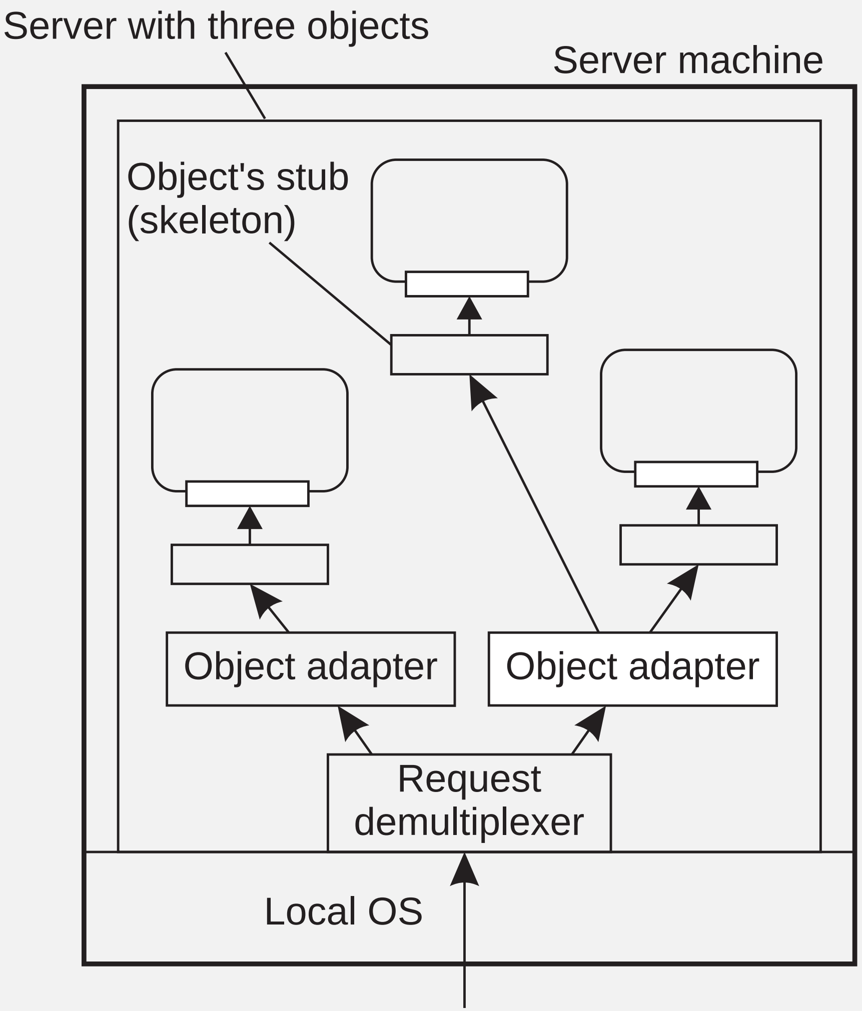
Processes: Object servers

Object adapter

The “manager” of a set of objects:

- Inspects (as first) incoming requests
- Ensures referenced object is activated (requires identification of servant)
- Passes request to appropriate skeleton, following specific activation policy
- Responsible for generating object references

Processes: Object servers



Observation

Object servers determine how their objects are constructed

Example: Ice

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

    adapter = ic->createObjectAdapterWithEndPoints
                ( "MyAdapter","tcp -p 10000");
    object  = new MyObject;

    adapter->add(object, objectID);
    adapter->activate();

    ic->waitForShutdown();
}
```

Note

Activation policies can be changed by modifying the properties attribute of an adapter. Ice aims at simplicity, and achieves this partly by putting policies into the middleware.

Remote Method Invocation (RMI)

Basics

(Assume client stub and server skeleton are in place)

- Client invokes method at stub
- Stub marshals request and sends it to server
- Server ensures referenced object is active:
  - Create separate process to hold object
  - Load the object into server process
  - ...
- Request is unmarshaled by object's skeleton, and referenced method is invoked
- If request contained an object reference, invocation is applied recursively (i.e., server acts as client)
- Result is marshaled and passed back to client
- Client stub unmarshals reply and passes result to client application

RMI: Parameter passing

Object reference

Much easier than in the case of RPC:

- Server can simply bind to referenced object, and invoke methods
- Unbind when referenced object is no longer needed

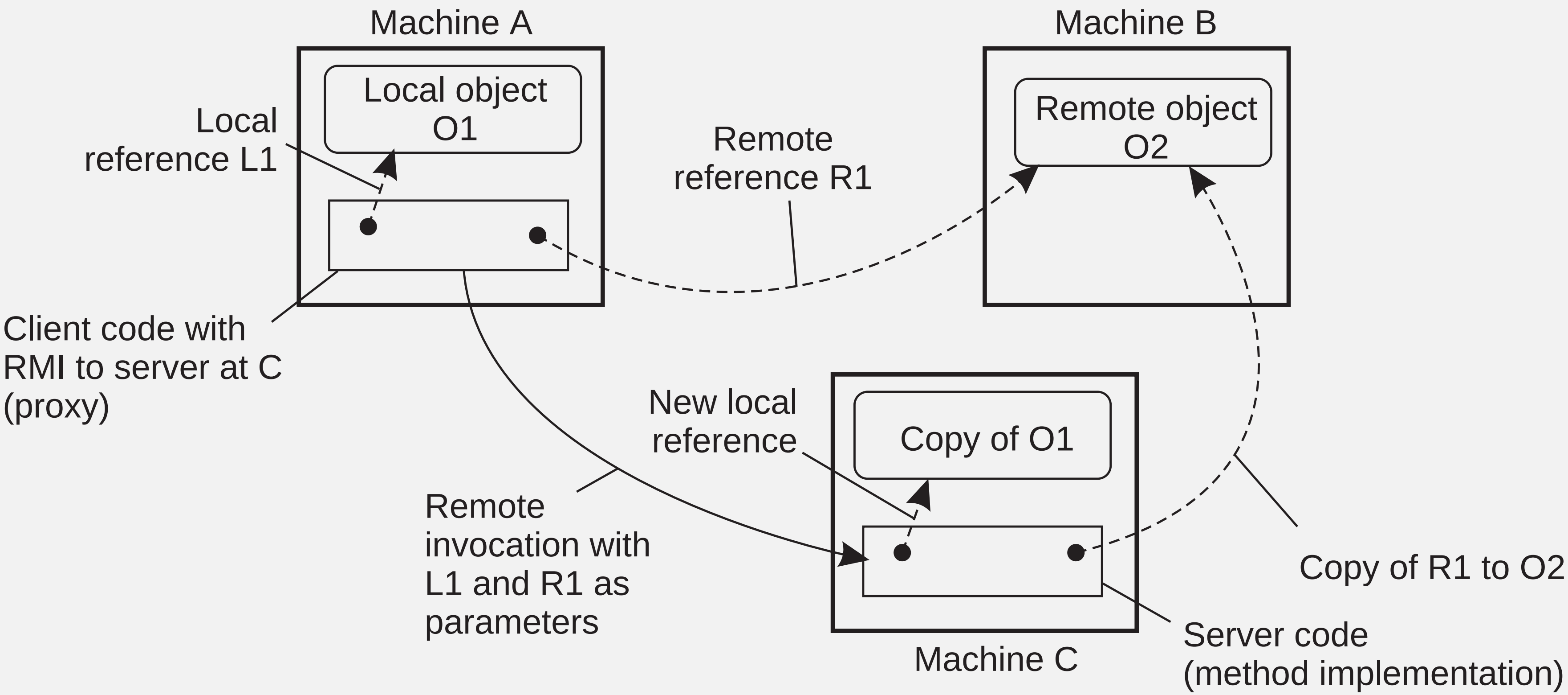
RMI: Parameter passing

Object-by-value

A client may also pass a complete object as parameter value:

- An object has to be marshaled:
  - Marshall its state
  - Marshall its methods, or give a reference to where an implementation can be found
- Server unmarshals object. Note that we have now created a copy of the original object.
- Object-by-value passing tends to introduce nasty problems

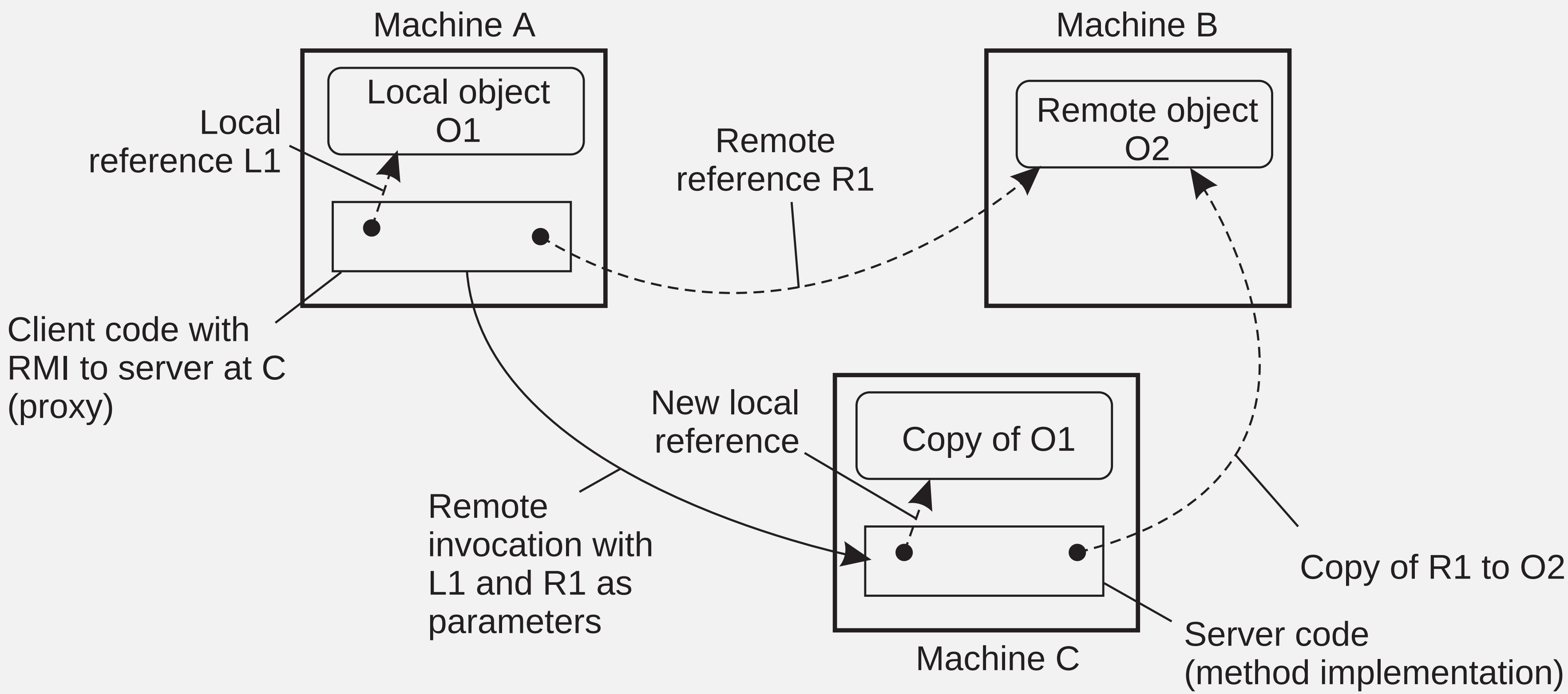
RMI: Parameter passing



Note

Systemwide object reference generally contains server address, port to which adapter listens, and local object ID. Extra: Information on protocol between client and server (TCP, UDP, SOAP, etc.)

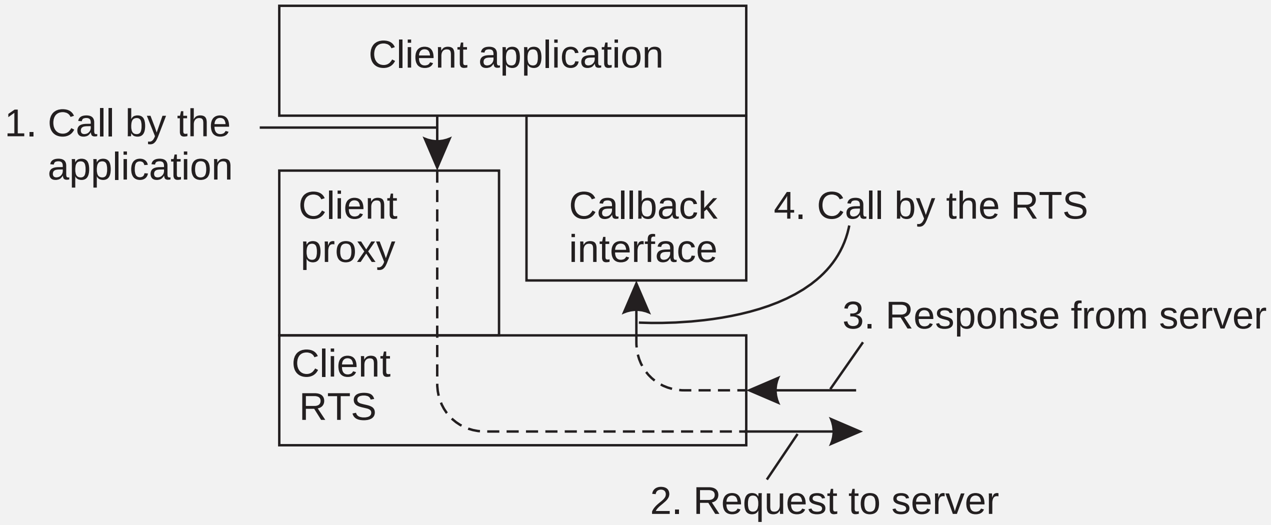
RMI: Parameter passing

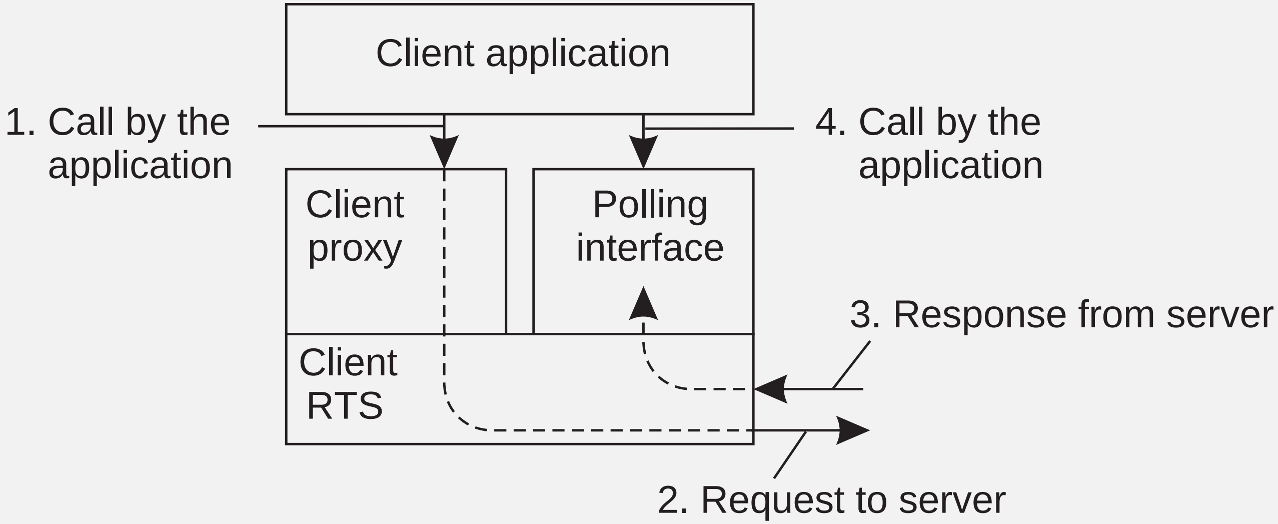


Question

What's an alternative implementation for a remote-object reference?

Object-based messaging

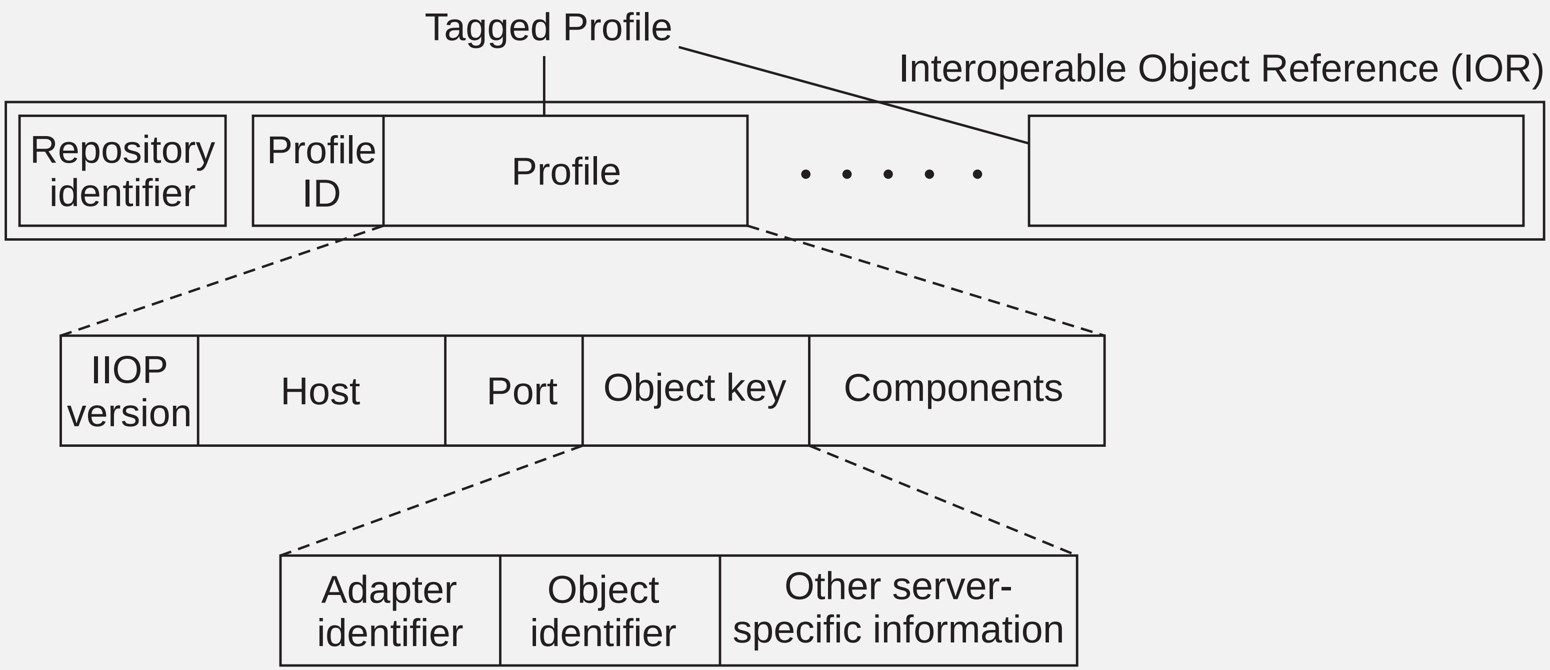




Object references

Observation

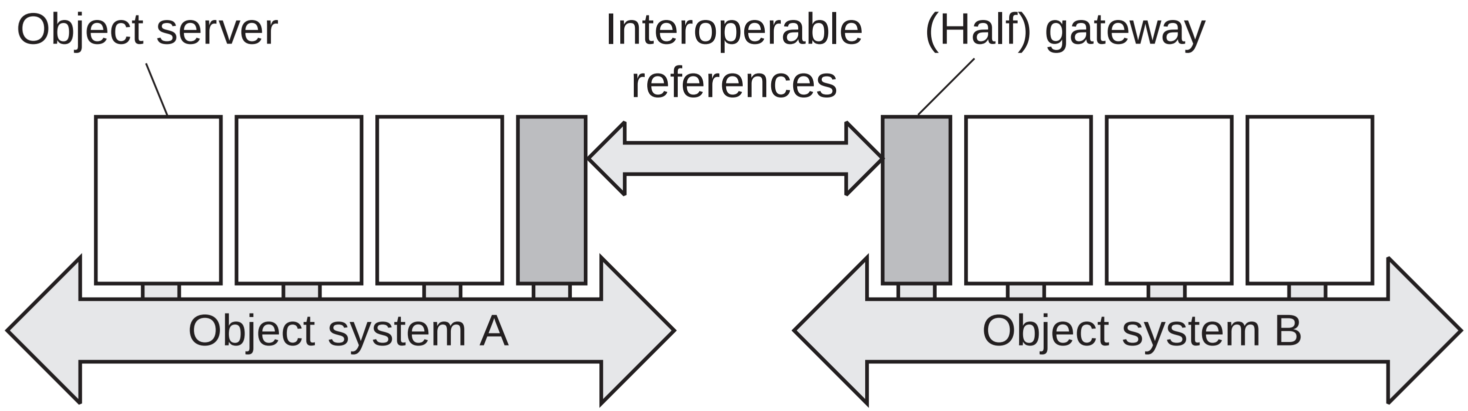
In order to invoke remote objects, we need a means to uniquely refer to them. Example: CORBA object references.



Object references

Observation

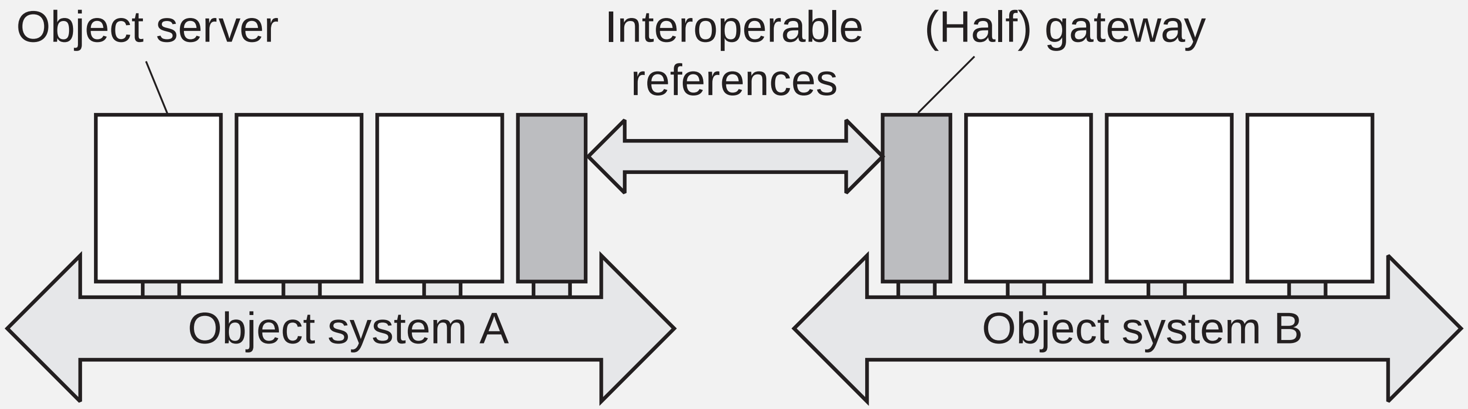
It is not important how object references are implemented per object-based system, as long as there is a standard to exchange them between systems.



Solution

Object references passed from one RTS to another are transformed by the bridge through which they pass (different transformation schemes can be implemented)

Object references



Observation

Passing an object reference *refA* from RTS A to RTS B circumventing the A-to-B bridge may be useless if RTS B doesn't understand *refA*

Globe object references: location independent

Stacked address

Stack of addresses representing the protocol to speak:

Field	Description
Protocol ID	Constant representing a (known) protocol
Protocol addr.	Protocol-specific address
Impl. handle	Reference to a file in a repository

Instance address

Contains all that is needed to talk in a propritary way to an object:

Field	Description
Impl. handle	Reference to a file in a repository
Initialization string	Used to initialize an implementation

Consistency and replication

Observation

Objects form a natural means for realizing entry consistency:

- Data are grouped into units, and protected by a synchronization variable (i.e., lock)
- Synchronization variables adhere to sequential consistency (i.e., values are set atomically)
- Operations of grouped data can be nicely grouped: object

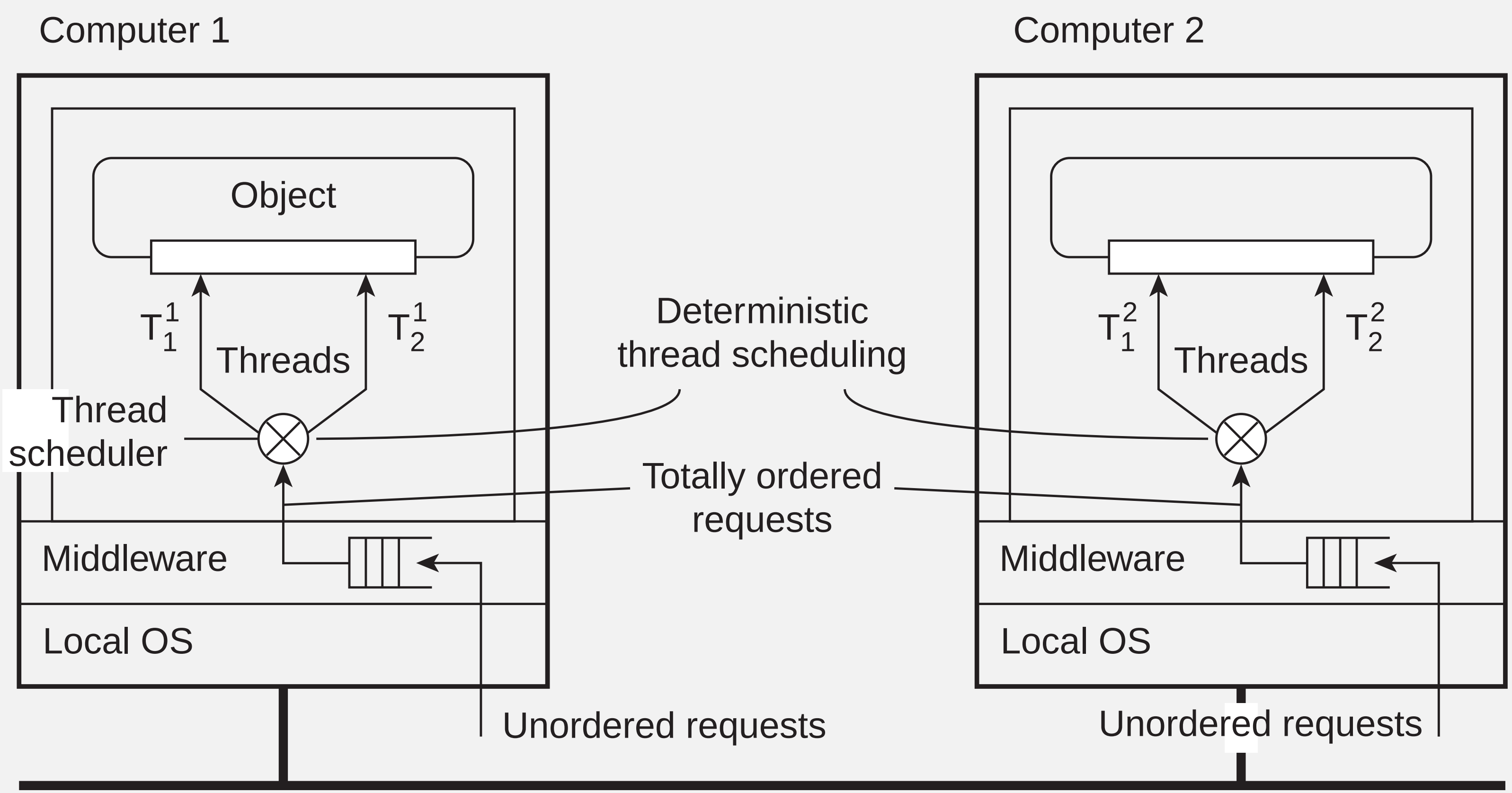
Problem

What happens when objects are replicated? One way or the other we need to ensure that operations on replicated objects are properly ordered.

Replicated objects

Problem

We need to make sure that requests are ordered correctly at the servers and that threads are deterministically sheduled



Replicated objects

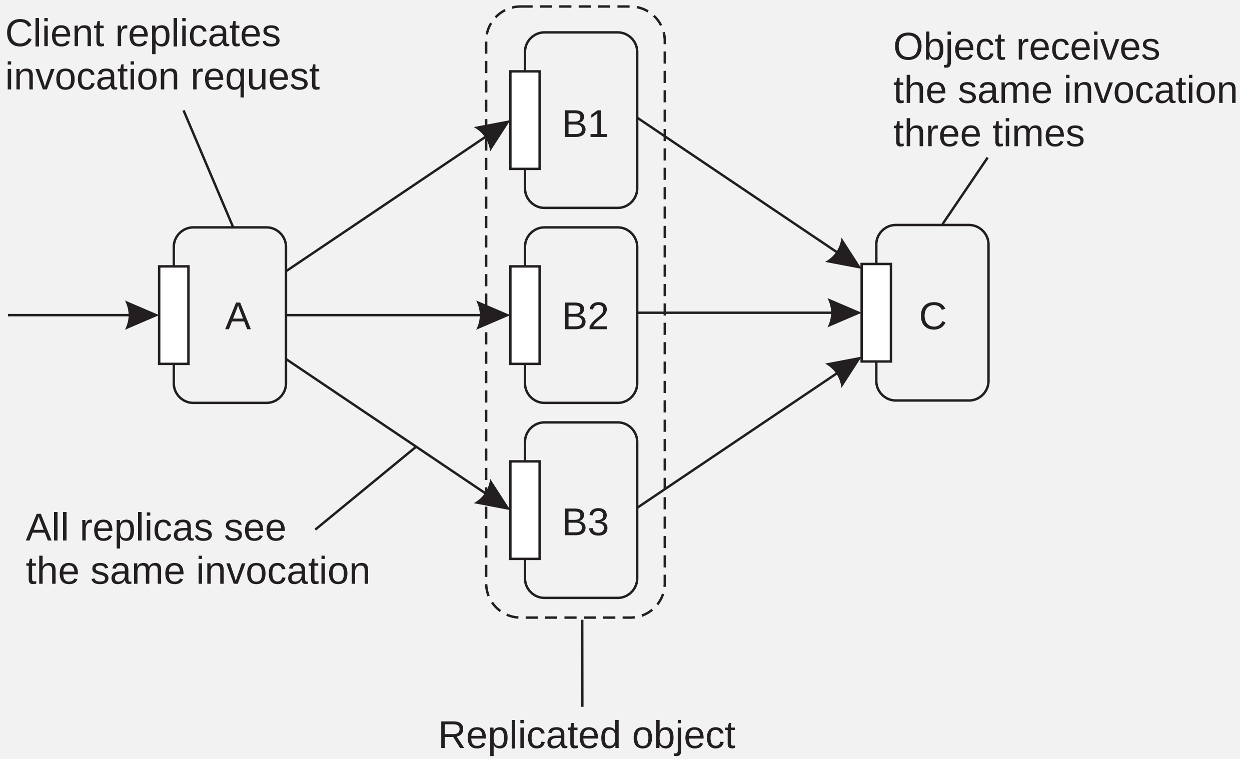
Observation

We are dealing with nasty issues here. Simplicity may dictate completely serialized (i.e., single-threaded) executions at the server.

Replicated invocations

Active replication

Updates are forwarded to multiple replicas, where they are carried out. There are some problems to deal with in the face of replicated invocations



Replicated invocations

Solution

Assign a coordinator on each side (client and server), which ensures that only one invocation, and one reply is sent