Dr.-Ing. Michael Eichberg

TECHNISCHE UNIVERSITÄT DARMSTADT

Middleware (Technologies)

“Remoting” - RPC and RMI

Remoting? (A Definition)

Remoting allows a computer program to cause a **subroutine or procedure to execute in another address space** (commonly on another computer on a shared network) **without the programmer explicitly coding the details** for this remote interaction.

Middleware

What is Middleware?

What is Middleware?

Middleware is a *class of software technologies* designed to help (I) **manage the complexity** and (II) **heterogeneity inherent in distributed systems**.

A Simple Server Using Sockets

(Part I)

```
/* A simple server in the internet domain using TCP. The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg){perror(msg); exit(1);}

int main(int argc, char *argv[]){
   int sockfd, newsockfd, portno, clilen;
   char buffer[256]; int n;
   struct sockaddr_in serv_addr, cli_addr;

   sockfd = socket(AF_INET, SOCK_STREAM, 0); // socket() returns a socket descriptor
   if (sockfd < 0)
      error("ERROR opening socket");

   bzero((char *) &serv_addr, sizeof(serv_addr)); // bzero() sets all values in a buffer to zero.
   portno = atoi(argv[1]); // atoi() converts str into an integer
   ...
```

A Simple Server Using Sockets

(Part II)

```
...
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
   error("ERROR on binding");
listen(sockfd,5); // tells the socket to listen for connections
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0) error("ERROR on accept");

bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
n = write(newsockfd,"I got your message",18);

if (n < 0) error("ERROR writing to socket");
return 0;
}
```

Implementation of the "protocol".

A Simple Client Using Sockets

(Part I)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg){ perror(msg);exit(0);}

int main(int argc, char *argv[]){
   int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;
   char buffer[256];

   portno = atoi(argv[2]);

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0) error("ERROR opening socket");
   ...
```

A Simple Client Using Sockets

(Part II)

```
    server = gethostbyname(argv[1]);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(portno);

    if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
       error("ERROR connecting");

    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0) error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    printf("%s\n",buffer);

    return 0;
}
```

Issues When Using Sockets

What's lacking?

Clearly we need to take care of...

- … establishing a channel and of all errors that might happen during this process
- ... establishing a protocol
  (Who sends **what**, **when**, **in which order** and **what answer is expected.**)
- ... message formats
  (Transforming application level data to bytes that can be transmitted over the network.)

Issues When Using Sockets

What's lacking?

Clearly we

- … estab … during this process
- … estab … is **expecte**
- … mess …

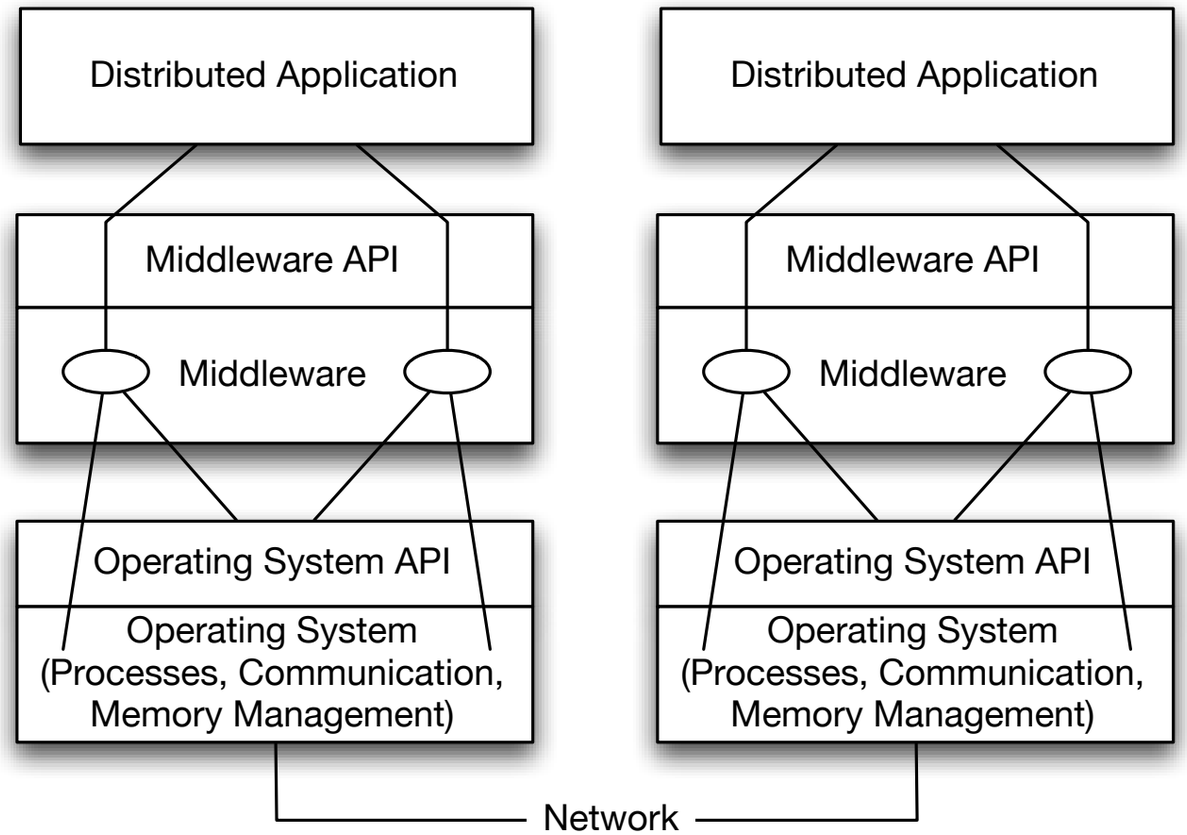
(Transforming application level data to bytes that can be transmitted over the network.)

We want:

- improved language integration, and
- generation of most of the networking code

Middleware as a Programming Abstraction

- A software layer **above the operating system** and **below the application program** that provides a common programming abstraction across a distributed system
- A higher-level building block than APIs provided by the OS (such as sockets)



Middleware as a Programming Abstraction

heterogeneity =dt.
Heterogenität
Ungleichartigkeit
Verschiedenartigkeit

Programming abstractions offered by **middleware mask some of the heterogeneity and handles some of the complexity programmers of a distributed application must deal with**:

- Middleware always mask heterogeneity of the underlying networks, hardware
- Most middleware mask heterogeneity of operating systems and/or programming languages
- Some middleware even mask heterogeneity among different vendor implementations of the same middleware standard

e.g., CORBA-IDL

Middleware as a Programming Abstraction

Middleware provides transparency with respect to one or more of the following dimensions:

- Location,
- Concurrency,
- Replication,
- Failures

Middleware as a Programming Abstraction

- An operating system (OS) is the software that makes the hardware useable
  (A bare computer without an OS could be programmed with great difficulty.)

- Middleware is the software that makes a distributed system (DS) programmable
  (Programming a DS is much more difficult without middleware.)

Middleware as Infrastructure

- Behind programming abstractions there is a complex infrastructure that implements those abstractions
  (Middleware plattforms are very complex software systems.)
- As programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional software layers
  - The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions

E.g. to handle errors we can have "transactional RPCs"

Middleware as Infrastructure

The infrastructure is also intended to *support additional functionality* that makes **development**, **maintenance**, and **monitoring** easier and less costly:

- logging,
- recovery,
- advanced transaction models (e.g. transactional RPC),
- language primitives for transactional demarcation,
- transactional file system,
- etc.

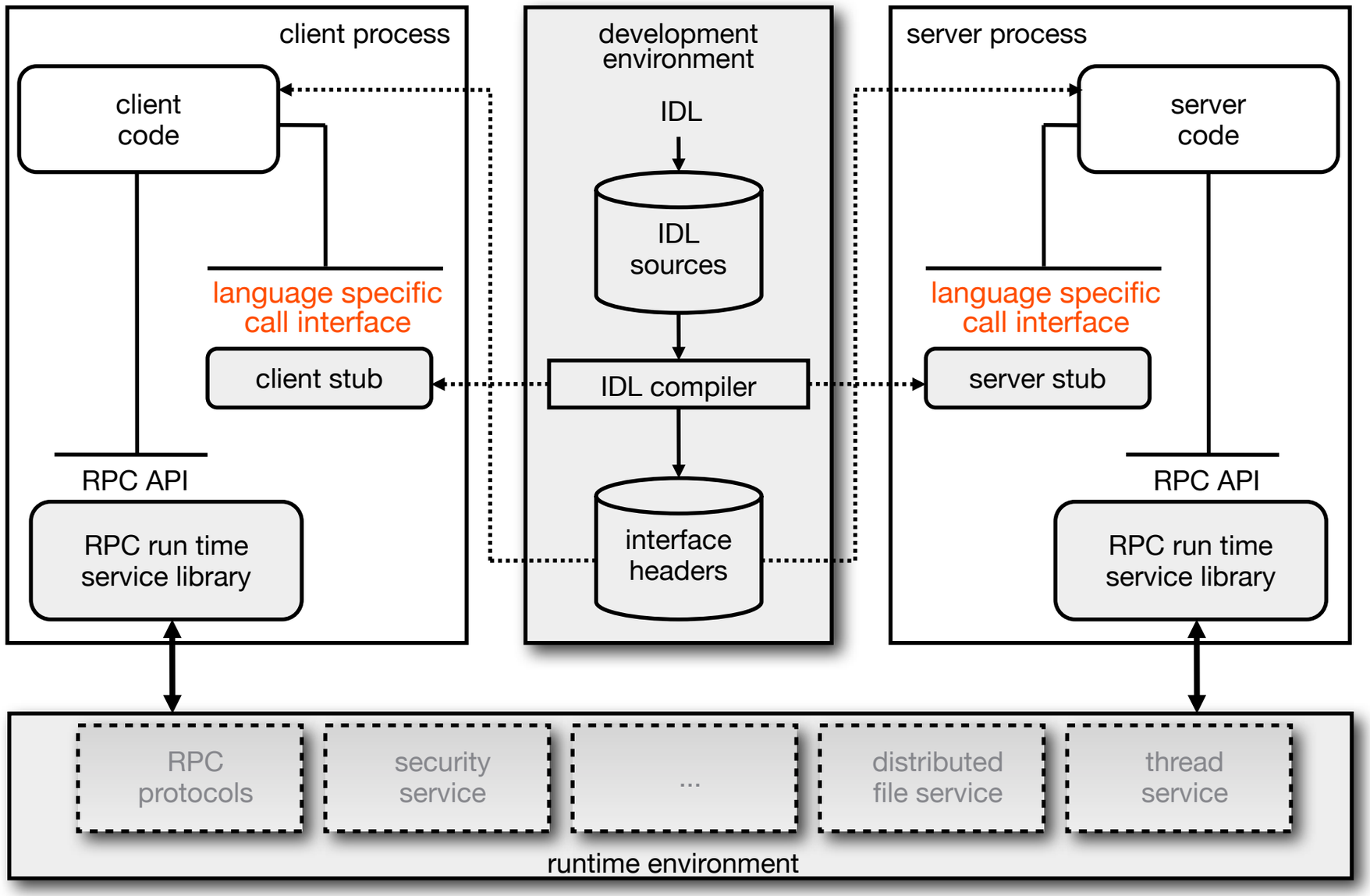
Middleware as Infrastructure

The **infrastructure also takes care of (all) the non-functional properties** typically ignored by data models, programming models, and programming languages:

- performance,
- availability,
- resource management,
- reliability,
- etc.

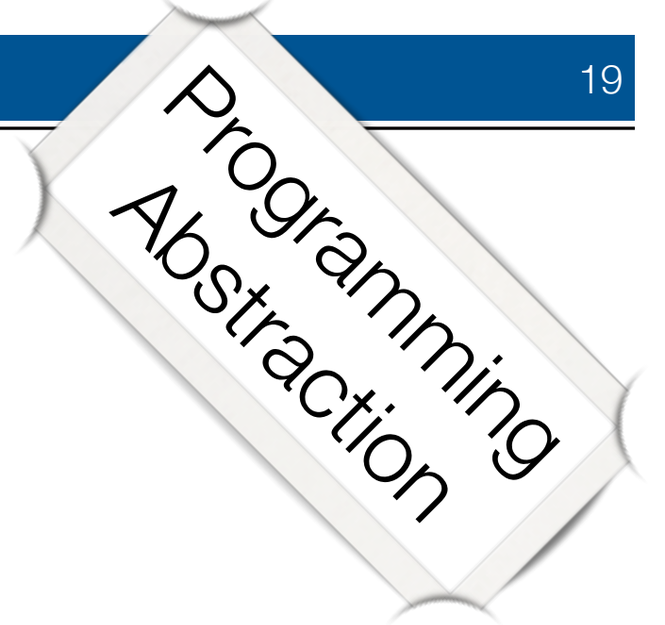
Middleware as Infrastructure

Conceptual Model



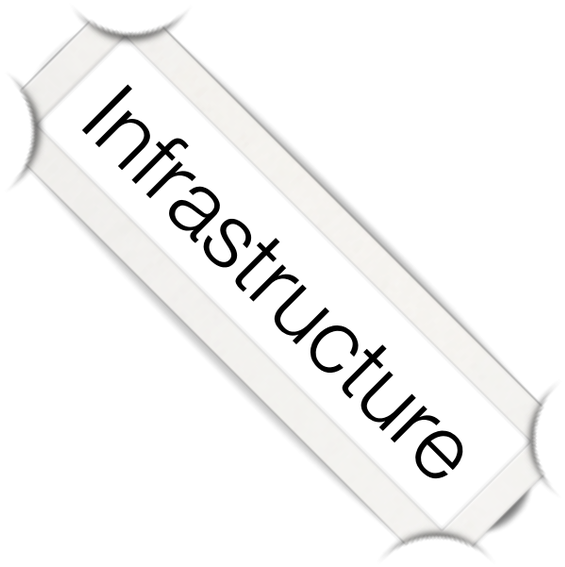
[Alonso; Web services: Concepts, Architectures and Applications; Springer Verlag, 2004]

Understanding Middleware



- Intended to hide low level details of hardware, networks, and distribution
- Continuing **trend towards increasingly more powerful primitives** that – without changing the basic concept of RPC – have additional properties or allow more flexibility in the use of the concept
- Evolution and appearance to the programmer is dictated by the trends in programming languages
  - RPC and C
  - CORBA and C++
  - RMI (Corba) and Java
  - “Classical” Web Services and XML
  - RESTful Web Services and JSON

Understanding Middleware



- Comprehensive platform for developing and running complex distributed systems
- Trend is towards **service oriented architectures** at a global scale **and standardization of interfaces**
- Another important trend is towards single vendor software stacks to minimize complexity and streamline interaction
- Evolution is towards integration of platforms and flexibility in the configuration (plus autonomic behavior)

Remote Procedure Call (RPC)

Programming Language Integration

Exchanging Data

Inner workings of RPC

Call Semantics

Remote Procedure Calls - Original Motivation

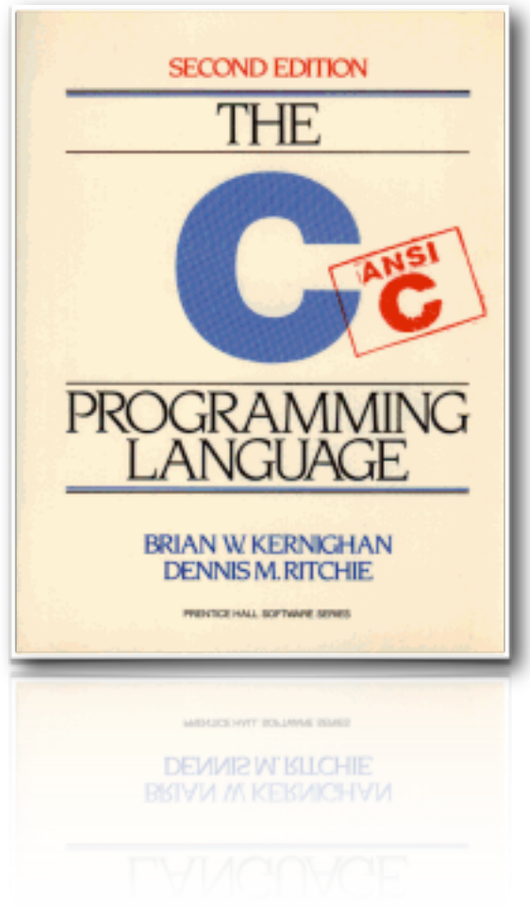
Birrell and Nelson (1984)

Emphasis is on **hiding network communication** by allowing a process to call a procedure of which the implementation is located on a remote machine:

- Removing the need for the distributed systems programmer to worry about all the details of network programming (i.e., no more sockets)
- Bridging the conceptual gap between *invoking local functionality via procedures* and *invoking remote functionality* via sockets

Remote Procedure Calls - Historic Context

- The notion of distributed service invocation became a reality at the beginning of 80's:
  - Procedural languages (mainly C) were dominant
  - In procedural languages, the basic module is the procedure
    (A procedure implements a particular function or service that can be used anywhere within the program.)
- It seemed natural to maintain this notion when talking about distribution:
  - The client makes a procedure call to a procedure that is implemented by the server
  - Since the client and server can be in different machines, the procedure is remote

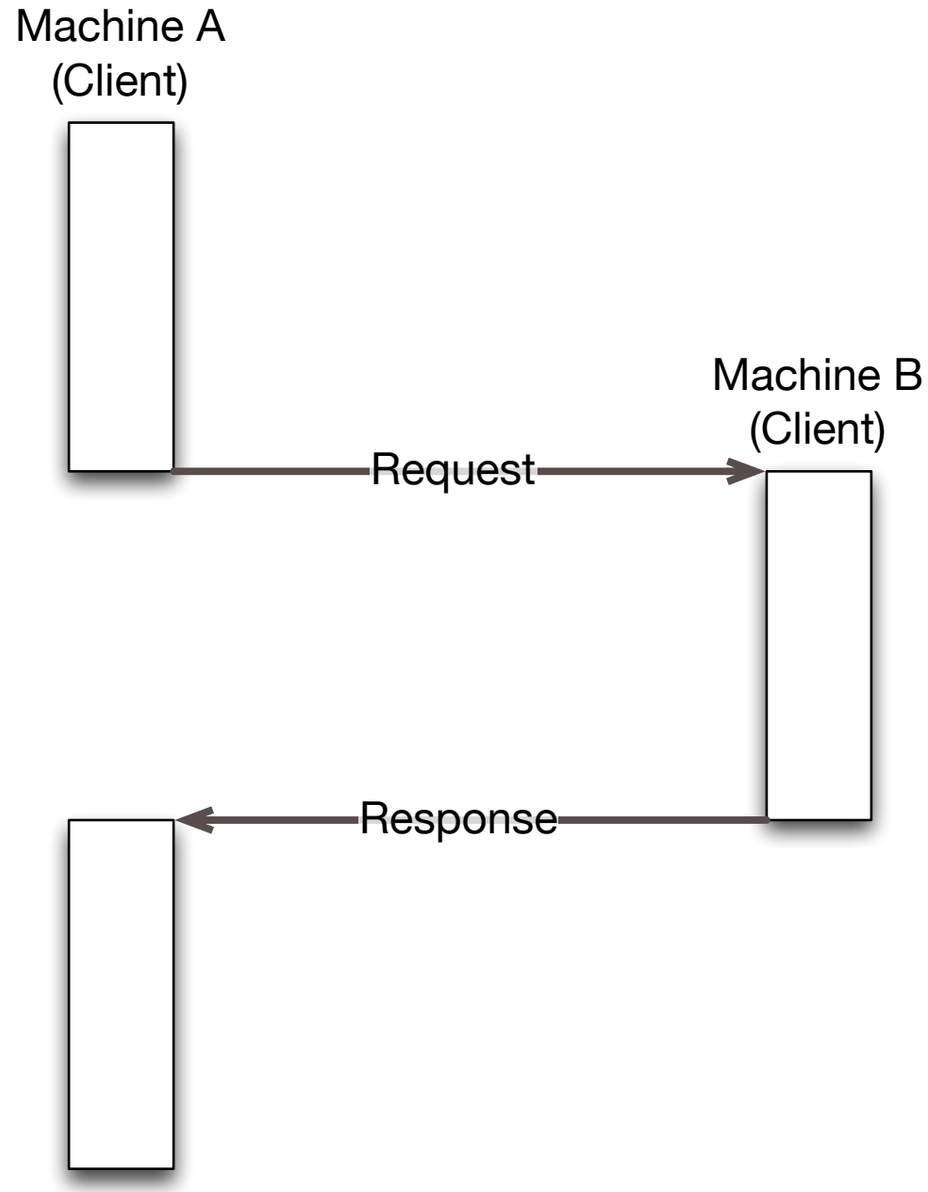


The Basics of Client/Server: Synchronous Communication

A server is a program that implements certain services.

Clients would like to invoke those services:

- Client & server reside on different computers and - may - run on different operating systems
- Communication is by sending messages (no shared memory, no shared disks, etc.)
- Some minimal guarantees are to be provided (handling of failures, call semantics, etc.)



We want communication to be conceptually as simple as this…

Problems to Solve: “Transparency of” RPCs

(When calling remote procedures.)

- How to make the service invocation part of the language in a more or less transparent manner?
- Debate:
  - **Transparent or**
  - **Not transparent?**

  A remote call is something completely different than a local call; should the programmer be aware of this?

Problems to Solve: Exchanging Data

(When calling remote procedures.)

- How to exchange data between machines that might use different representations for different data types?

- Different encodings of data types
  (character encoding, representations of floats, etc.)
- Different ways to order bytes:
  - Intel CPUs are “ little-endian”
  - Sun Sparc CPUs are “big-endian”

Problems to Solve: Exchanging Data

(When calling remote procedures.)

- Problem:
  Information in running programs is stored in high-level data structures;
  Information in messages passed over network consists of sequences of bytes
- Complex data types must be flattened:
  - **Marshalling** - the process of assembling data items into a form suitable for transmission in a message
  - **Unmarshalling** - the process of disassembling them on arrival at the destination to produce an equivalent collection of data items

Problems to Solve: Exchanging Data

(When calling remote procedures.)

- Problem:
  Information in running programs is stored in high-level data structures; Information in messages passed over network consists of sequences of bytes
- Two methods for exchanging data values:
  - **Values converted to an agreed external format** before transmission and converted to the local form on receipt
  - **Values transmitted in sender's format, together with an indication of the format used**; recipient converts values if necessary

Problems to Solve: Finding / Binding Services

(When calling remote procedures.)

- How to **find and bind the service** one actually wants among a potentially large collection of services and servers?
  The goal is that the client does not necessarily need to know where the server resides or even which server provides the service (location transparency).

Problems to Solve: Dealing with Errors.

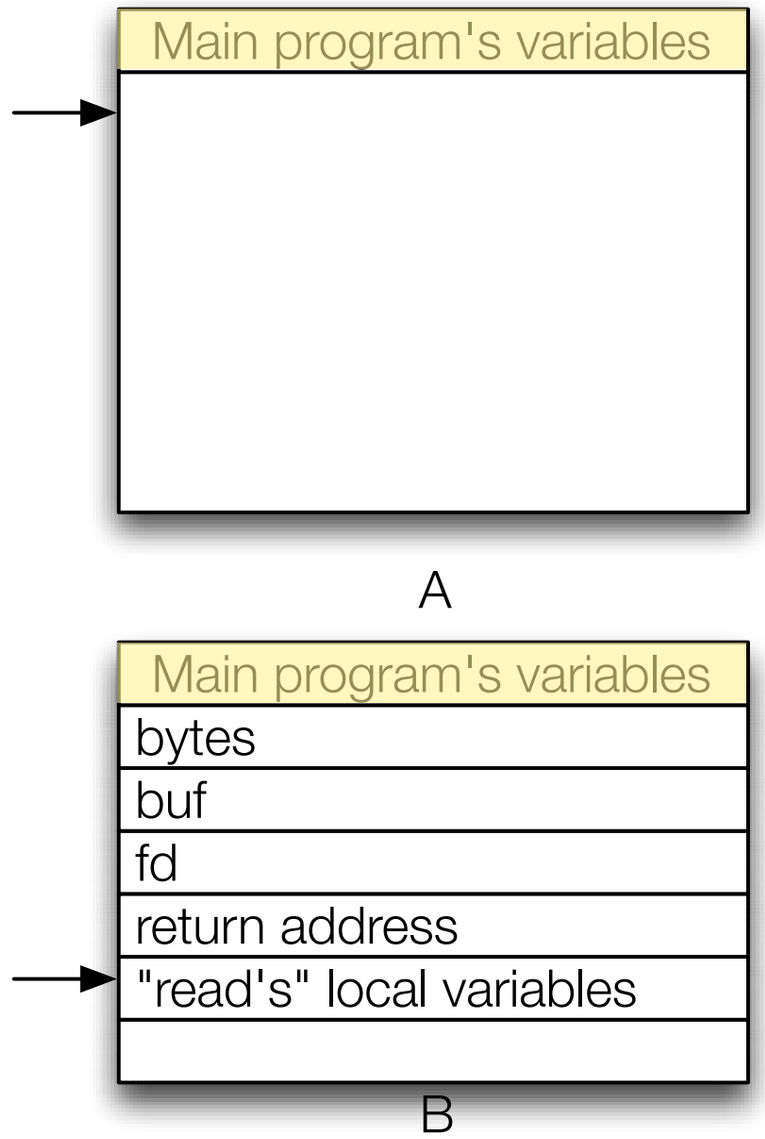
(When calling remote procedures.)

- How to **deal with errors** in the service invocation in a more or less elegant manner:
  - server is down,
  - communication is down,
  - server busy,
  - duplicated requests ...

Programming Language Integration?

Problems to Solve

- Conventional “local” procedure call (`read (fd,buf,bytes)`):
  - A. The stack before the call
  - B. The stack while the called procedure is active
- How to identify and address entities?
  - **Pointers cannot be passed as parameters**
  - Each machine has a **different address space**



How RPC Works

- As far as the programmer is concerned, a “remote” procedure call looks and works almost identically to a “local” procedure call – in this way, transparency is achieved
- To achieve transparency RPC introduced many concepts of middleware systems:
  - **Interface description language (IDL)**
  - Directory and naming services
  - Dynamic binding
  - Marshalling and unmarshalling
  - Opaque references

    Opaque references need to be used by the client to refer to the same data structure or entity at the server across different calls.
    (The server is responsible for providing these opaque references.)

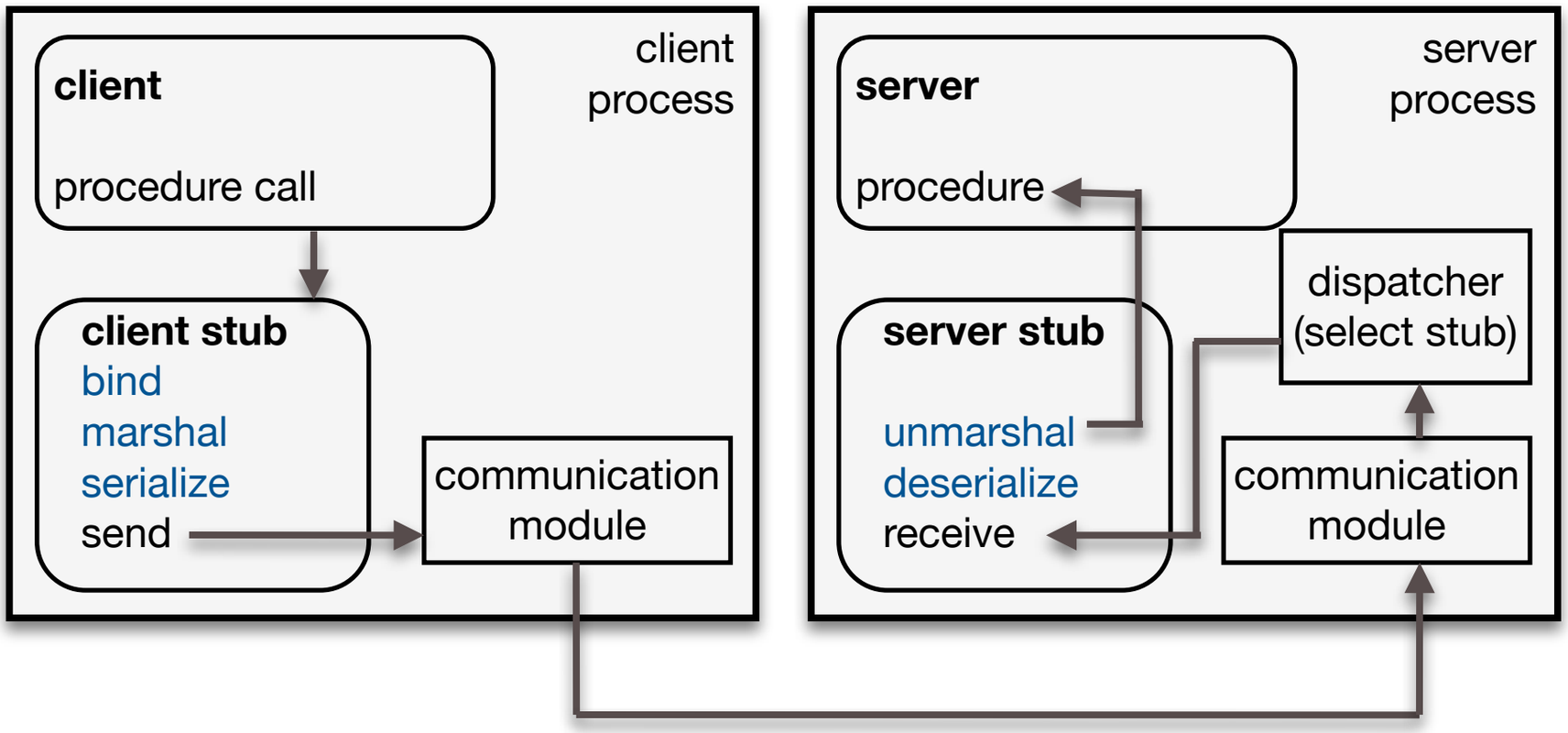
How RPC Works

Fundamentals

The procedure is “split” into two parts:

- **Client “stub”**
  Implementation of the interface on the local machine through which the remote functionality can be invoked
- **Server “stub”**
  Implementation of the interface on the server side passing control to the remote procedure that implements the actual functionality

Basics of RPC - Calling a Remote Procedure



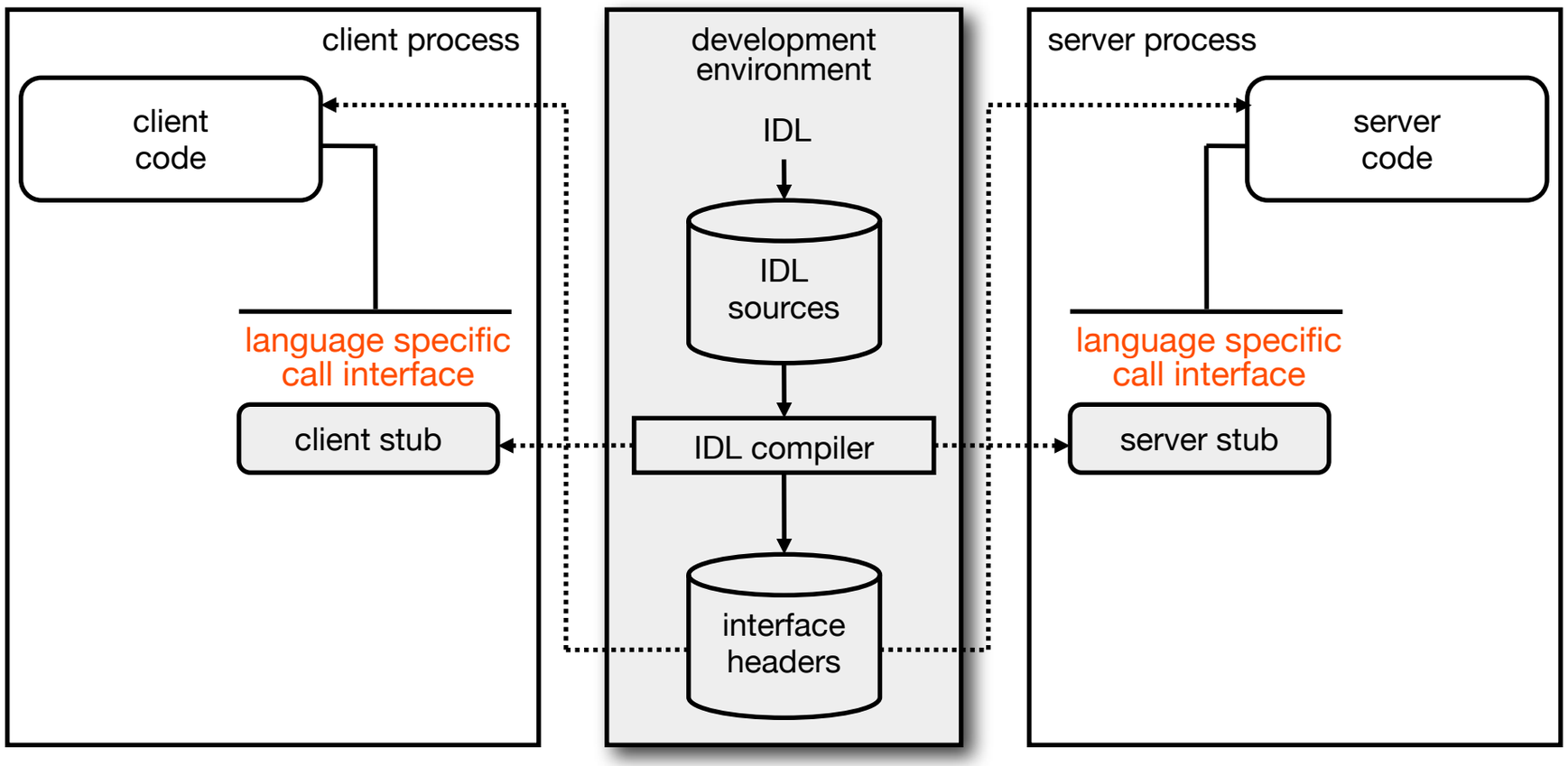
[Alonso; Web services: Concepts, Architectures and Applications; Springer Verlag, 2004]

Interface Definition Language (IDL)

Overview

- **Protocol interfaces** are described by means of an Interface Definition Language (IDL):
  - SUN RPC (see RFC 1831, 1014) uses XDR
  - CORBA's IDL is called IDL (ISO 14750)
  - JAXRPC uses XML
  - ...
- The interfaces are then compiled by an IDL compiler which generates the necessary stubs (and skeletons)

Developing Distributed Applications With RPC



[Alonso; Web services: Concepts, Architectures and Applications; Springer Verlag, 2004]

An Example Interface Definition (SUN RPC)

```
/* FileReadWrite service interface definition in file FileReadWrite.x */
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 SIMPLERW {
    void WRITE(writeargs)=1;
    Data READ(readargs)=2;
 }=2;
}=9999;
```

C Client of a File Server in SUN RPC

No more sockets!

```c
/* File S.c - server procedures for the FileReadWrite service */
#include <stdio.h>
#include <rpc/rpc.h>
#include"FileReadWrite.h"

void * write_2(writeargs * a) {
   /* do the writing to the file */
}

Data * read_2(readargs * a) {
   static Data result; /* must be static */
   result.buffer = ... /* do the reading from the file */
   result.length = ... /* amount read from the file */
   return &result;
}
```

C Client of a File Server in SUN RPC

No more sockets!

```c
/* File C.c - Simple client of the FileReadWrite service. */
#include <stdio.h>
#include <rpc/rpc.h>
#include "FileReadWrite .h„

main(int argc, char ** argv) {
   CLIENT *clientHandle; char *serverName = "coffee"; readargs a; Data *data;

   /* creates socket and a client handle */
   clientHandle= clnt_create(serverName, FILEREADWRITE, VERSION, "udp");

   if (clientHandle != NULL){
      a.f = 10; a.position = 100; a.length = 1000;

      /* call to remote read procedure */
      data = read_2(&a, clientHandle);

      clnt_destroy(clientHandle); /* closes socket */
   }
}
```

Binding in RPC

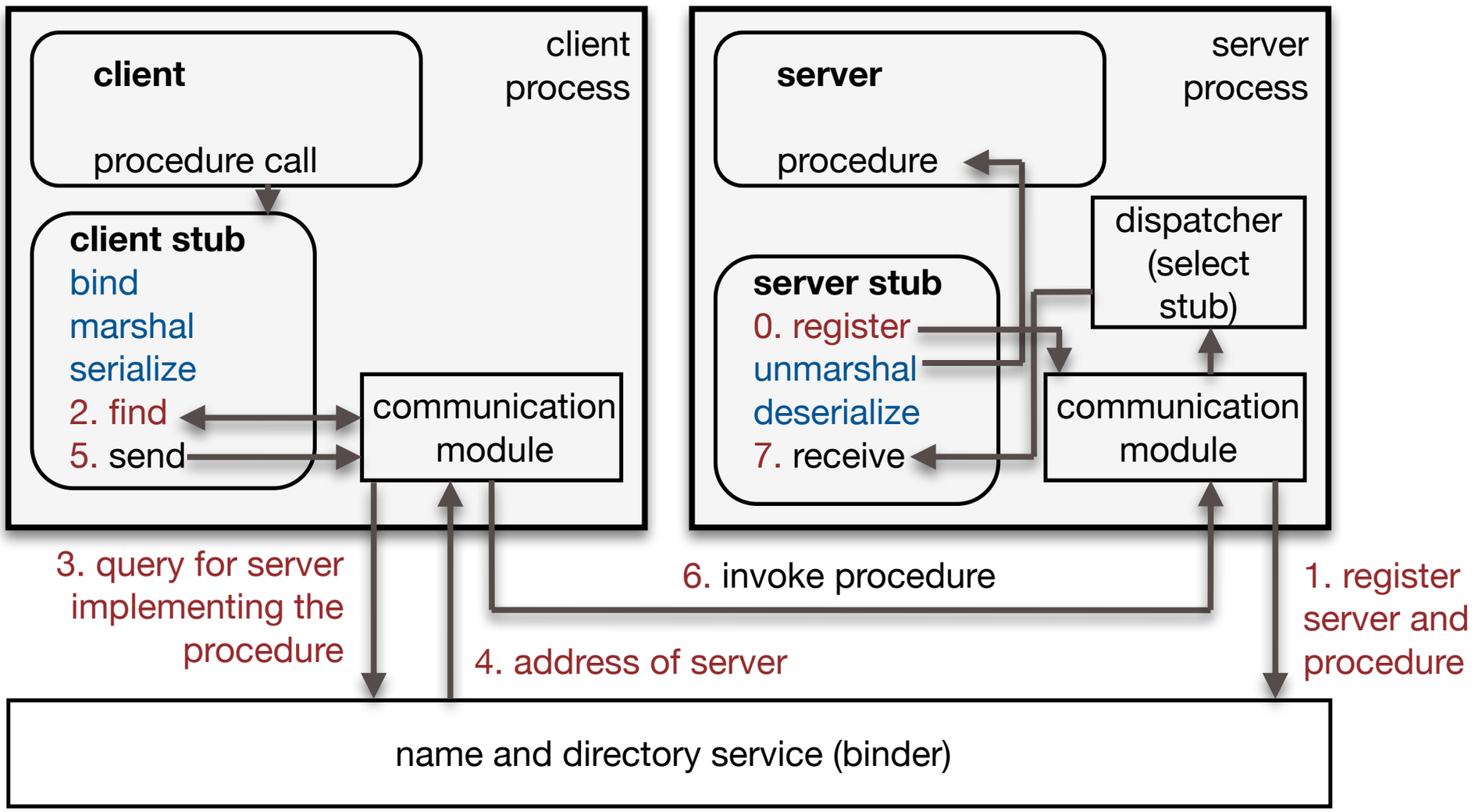
Overview

- Binding is the process whereby *a client creates a local association for (i.e., a handle to) a given server in order to invoke a remote procedure*
  (Form of handle depends on environment. Today, usually an IP address and port number. Alternatives: an Ethernet address, an X.500 address, etc.)
- **Static binding (local binding)**: the handle to the server where the procedure resides is hard-coded in client stub
- **Dynamic binding (distributed)**: clients use a specialized service to locate servers based on the signature of the procedure being invoked
  (Specialized service also called binder, or name service.)

Dynamic Binding Using a Service

- This service must be reachable by all participants; possible alternatives:
  - **predefined location**
  - **environment variables**
  - **broadcasting to all nodes looking for the binder**
- Uses IDL specifications to perform bindings

Dynamic Binding Using a Naming Service



[Alonso; Web services: Concepts, Architectures and Applications; Springer Verlag, 2004]

Static vs. Dynamic Binding

Static binding

Advantages

- Simple and efficient, no additional infrastructure beside client and server stub

Disadvantages

- Client and server tightly coupled
- If the server changes location, client must be changed
- Not possible to use redundant servers for load balancing

Dynamic binding

Advantages

- Enables what is missing with static binding

Disadvantages

- Adds indirection and infrastructure; name server, protocol for interacting with the directory server, primitives for registering with directory, etc.
- Hidden in the stubs

RPC - Call Semantics

Let's assume, a client makes an RPC to a service at a given server...

- After a time-out expires, the client may decide to resend the request
- If – after several tries – there is no success, what may have happened depends on the call semantics:
  - *maybe*: no guarantee
  - *at least once*
  - *at most once*
  - *exactly once*

RPC - Call Semantics

Let’s assume, a client makes an RPC to a service at a given server...

- **Maybe**: no guarantees. The procedure may have been executed (the response message(s) lost) or may have not been executed (the request message(s) lost).
  It is very difficult to write programs based on this type of semantics since the programmer has to take care of all possibilities.

RPC - Call Semantics

Let's assume, a client makes an RPC to a service at a given server...

- **At least-once**
  The procedure will be executed if the server does not fail, but it is possible that it is executed more than once
  - This may happen, for instance, if the client re-sends the request after a time-out
  - If the server is designed so that service calls are *idempotent* (each call results in the same outcome given the same input), this might be acceptable
- **At most-once**
  The procedure will be executed either once or not at all. Re-sending the request will not result in the procedure executing several times

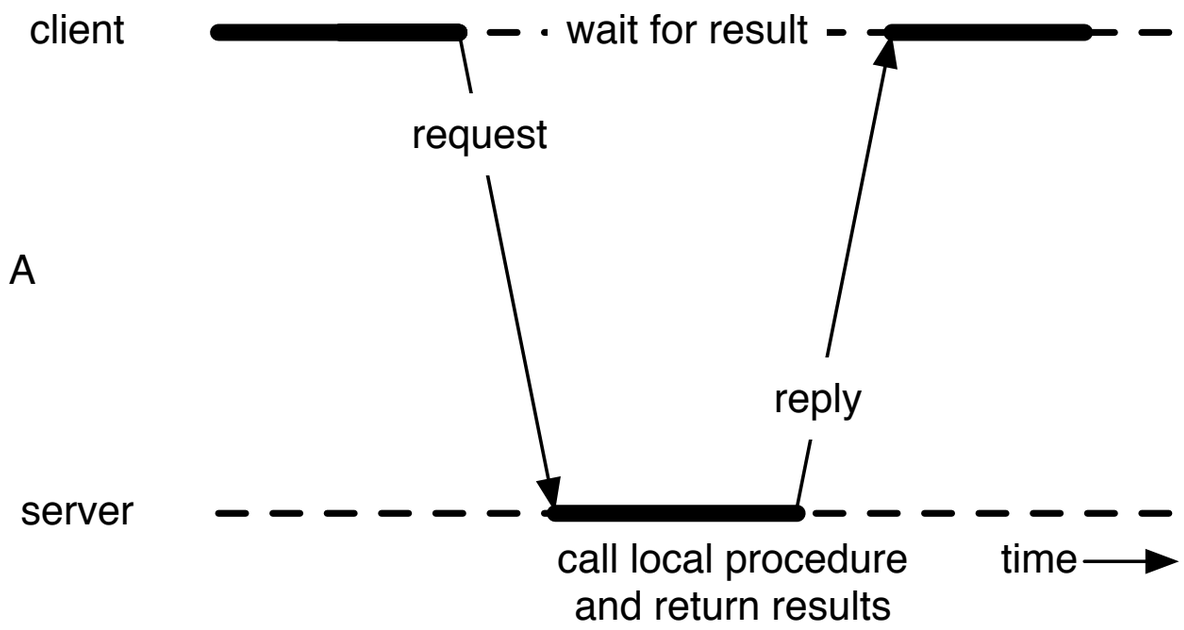
RPC - Call Semantics

Let’s assume, a client makes an RPC to a service at a given server...

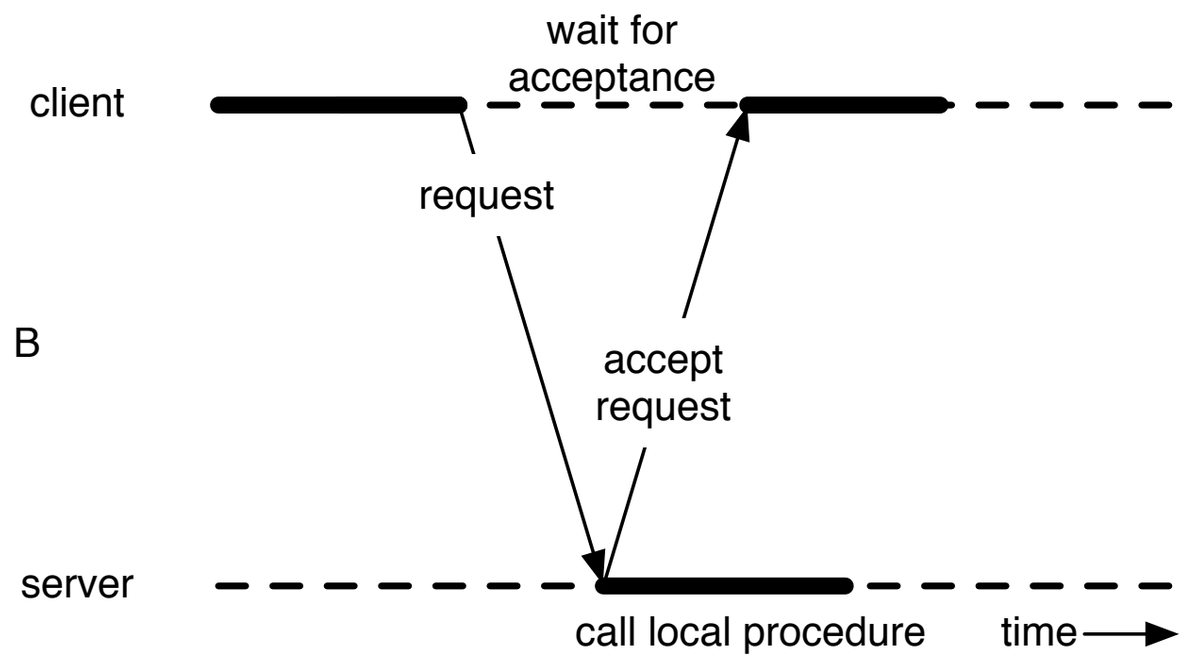
- **Exactly once**
  The system guarantees the local semantics assuming that a server machine that crashes will eventually restart.
  It keeps track of orphan calls, that is, calls on server machines that have crashed, and allows them to be later adopted by a new server.

Asynchronous RPC

A. The interconnection between client and server in a traditional RPC – note that **blocking** occurs – the client waits

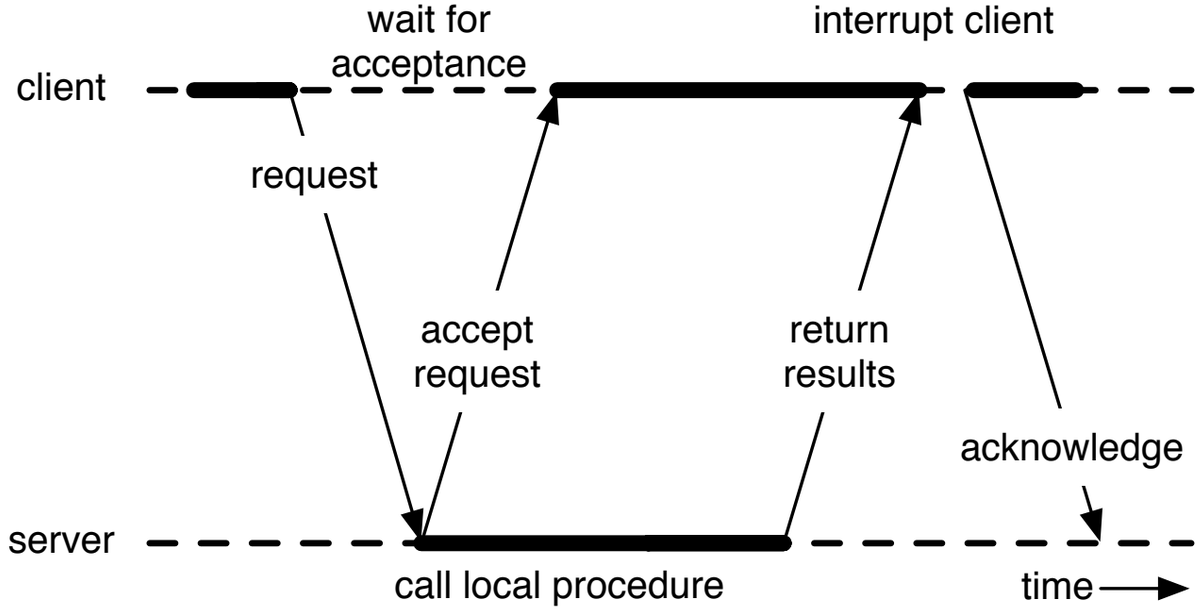


B. The interaction using asynchronous RPC – **no blocking**; useful when the client doesn't need or expect a result



Asynchronous RPC - Deferred

- A client and server interacting through two asynchronous RPCs – allows a client to perform other useful work - in particular sending further messages - while waiting for results



Stored Procedures

Based Upon RPCs

To add flexibility to their servers, (database) software vendors added to them the possibility of programming procedures that will run inside the server and that could be invoked through RPC.

Turned out to be very useful for databases:

- (Stored) procedures could be used to hide the schema and the SQL programming from the clients
- (Stored) procedures are a common mechanism found in all database systems (although differently implemented)

Distributed Environments

Based Upon RPCs

Common aspects: Security Services for Authentication / Authorization, Replication, Logging, Transaction Handling,...

- When designing distributed applications, there are a lot of crucial aspects common to all of them; RPC does not address many of these issues
- To support the design and deployment of distributed systems, programming and run time environments need to be created
  (These environments provide, on top of RPC, much of the functionality needed to build and run a distributed application.)
- The notion of distributed environment is what gave rise to middleware

RPC Summary - Advantages

- RPC provided a mechanism to implement distributed applications in a simple and efficient manner
- RPC followed the programming techniques of the time (procedural languages) and fitted quite well with the most typical programming languages (C), thereby facilitating its adoption by system designers

- RPC allowed the modular and hierarchical design of large distributed systems:
  - Client and server are separate entities
  - The server encapsulates and hides the details of the back end systems (such as databases)

RPC Summary - Disadvantages

- RPC is not a standard, it is an idea that has been implemented in many different ways (not necessarily compatible)
- RPC allows designers to build distributed systems but does not solve many of the problems distribution creates
  (In that regard, it is only a low-level construct.)
- Only supports “procedural” integration of application services
  (Does not provide object abstractions, e.g., polymorphism, inheritance of interfaces, etc.)

Remote Method Invocation (RMI)

Introduction

Problems to Solve

Protocol Stack

Example

Java Remote Method Invocation (RMI)

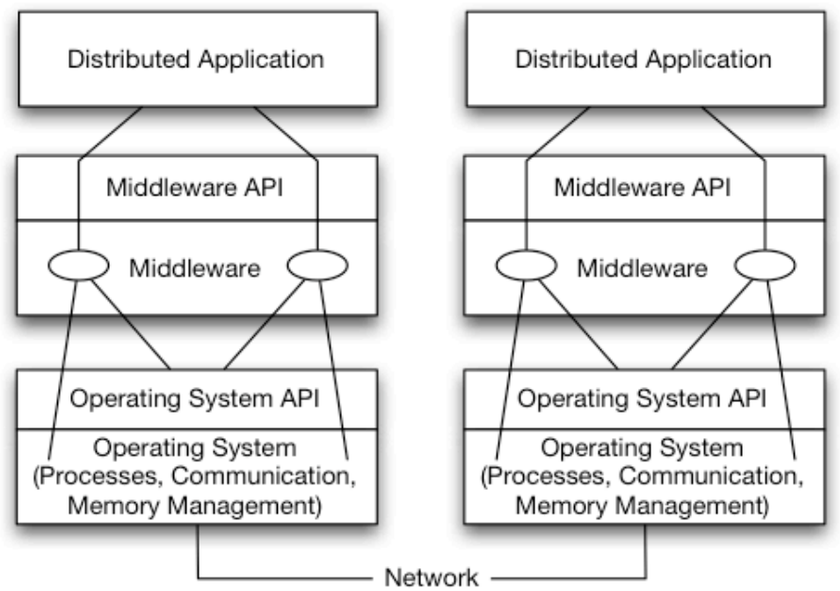
Allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM...

- Remote objects can be treated similarly to local objects
- Handles marshalling, transportation, and garbage collection of the remote objects
- Not an all-purpose Object-Request Broker (ORB) architecture like Webservices, CORBA and DCOM...
- Part of Java since JDK 1.1

RMI is Middleware

Middleware as a Programming Abstraction

- A software layer **above the operating system** and **below the application program** that provides a common programming abstraction across a distributed system.
- A higher-level building block than APIs provided by the OS . (such as sockets)



RMI is Middleware

Middleware

Middleware as Infrastructure

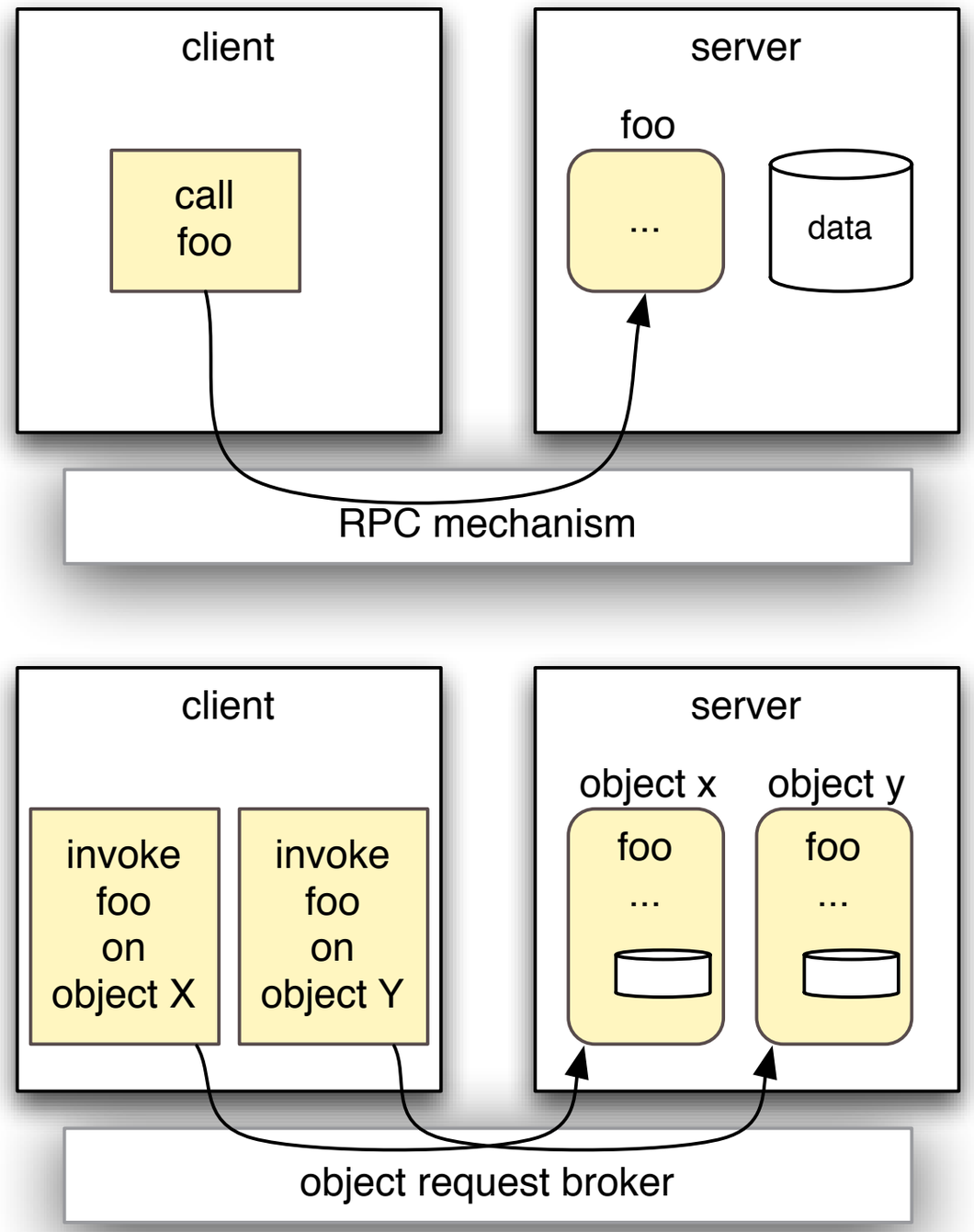
- Behind programming abstractions there is a complex infrastructure that implements those abstractions
  (Middleware plattforms are very complex software systems.)
- As programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional software layers
  - The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions

E.g. to handle errors we can have "transactional RPCs"

RMI vs. RPC

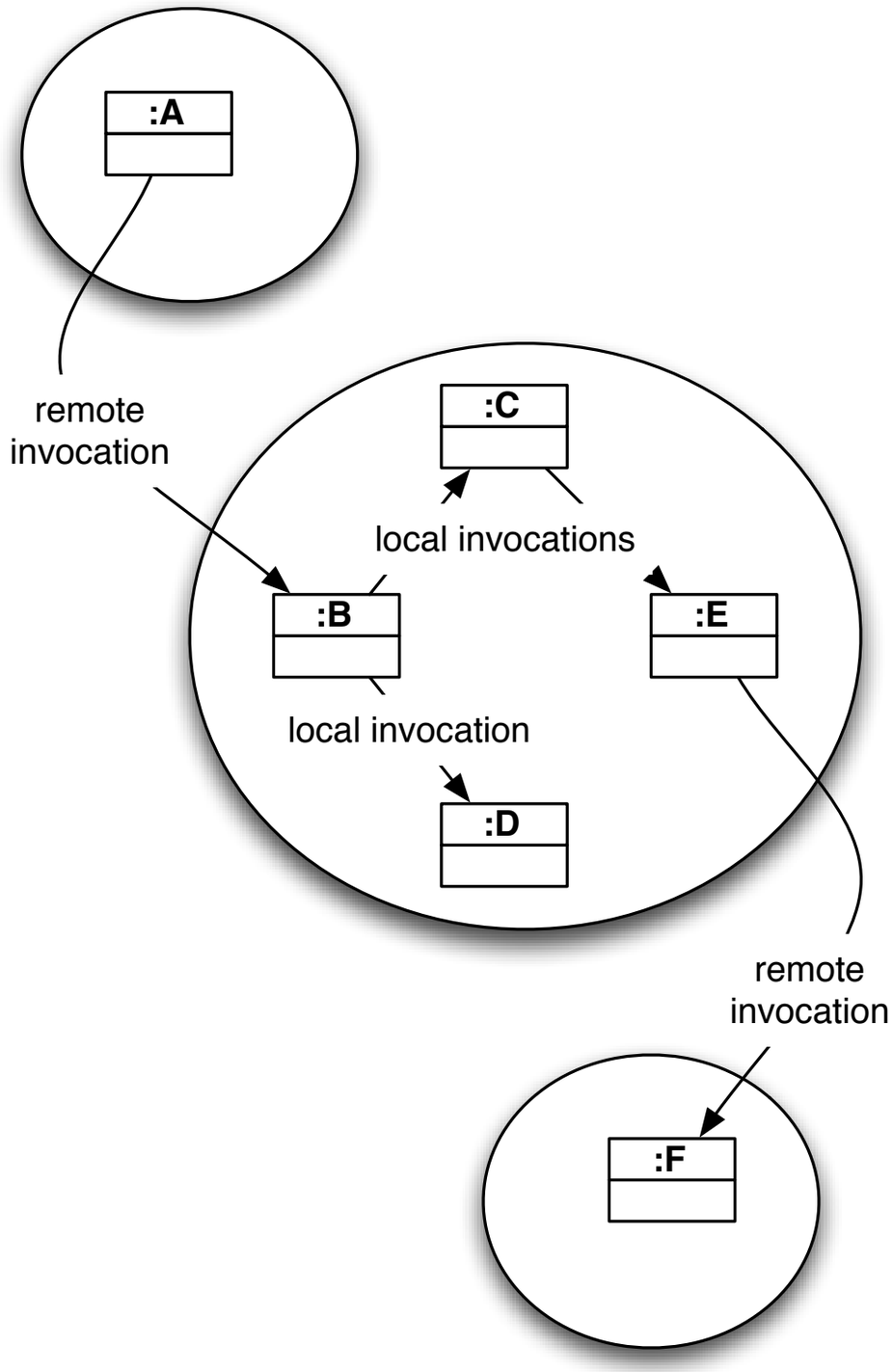
- Basically the *difference between*
  - procedure call

  and
  - operation invocation on an object



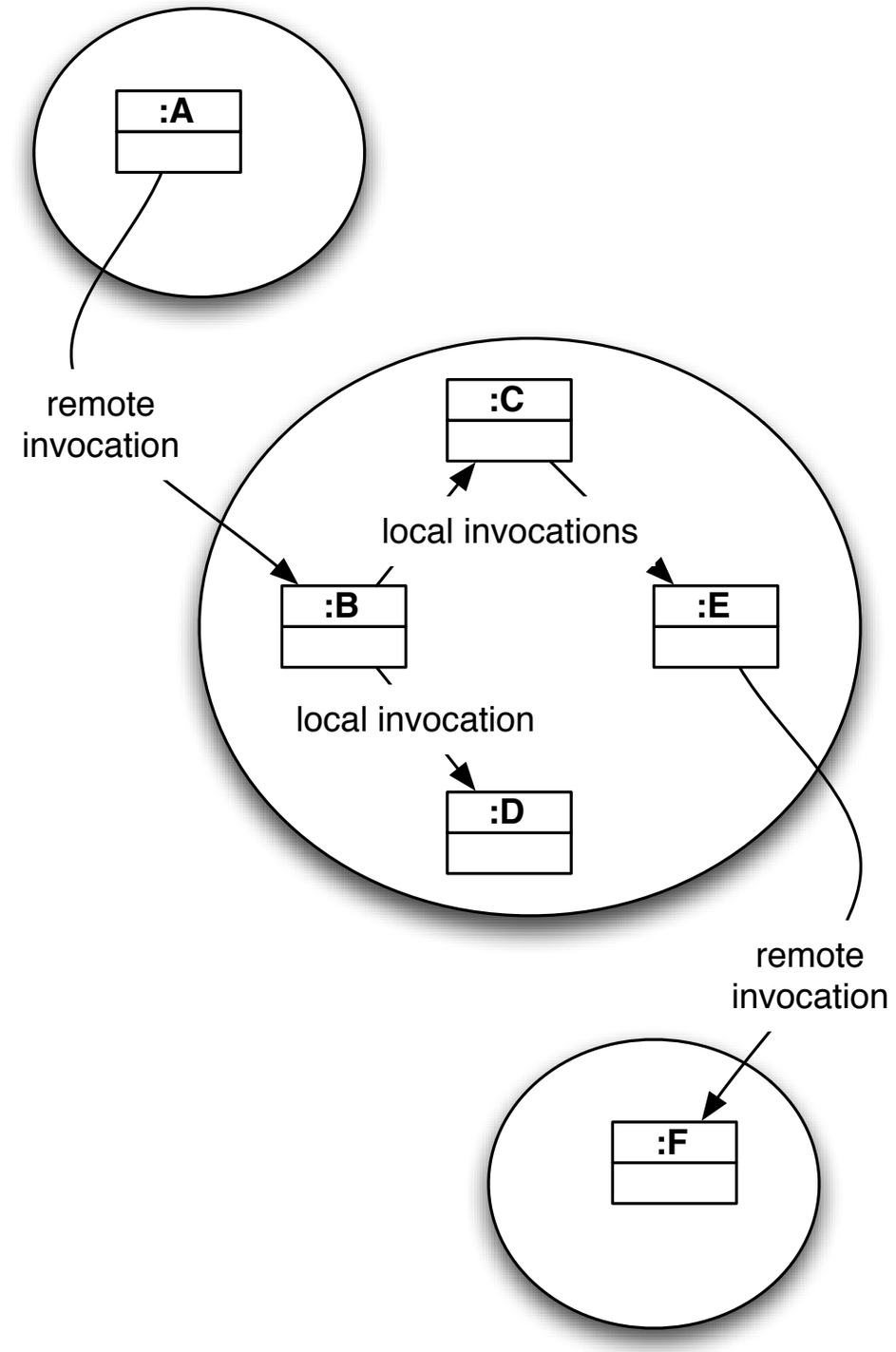
Distributed Object Model

- Each process contains objects, some of which can receive remote invocations, others only local invocations
  (Those that can receive remote invocations are called remote objects.)
- Objects need to know the remote object reference of an object in another process in order to invoke its methods
  (Remote Method Invocation; Remote Object References)



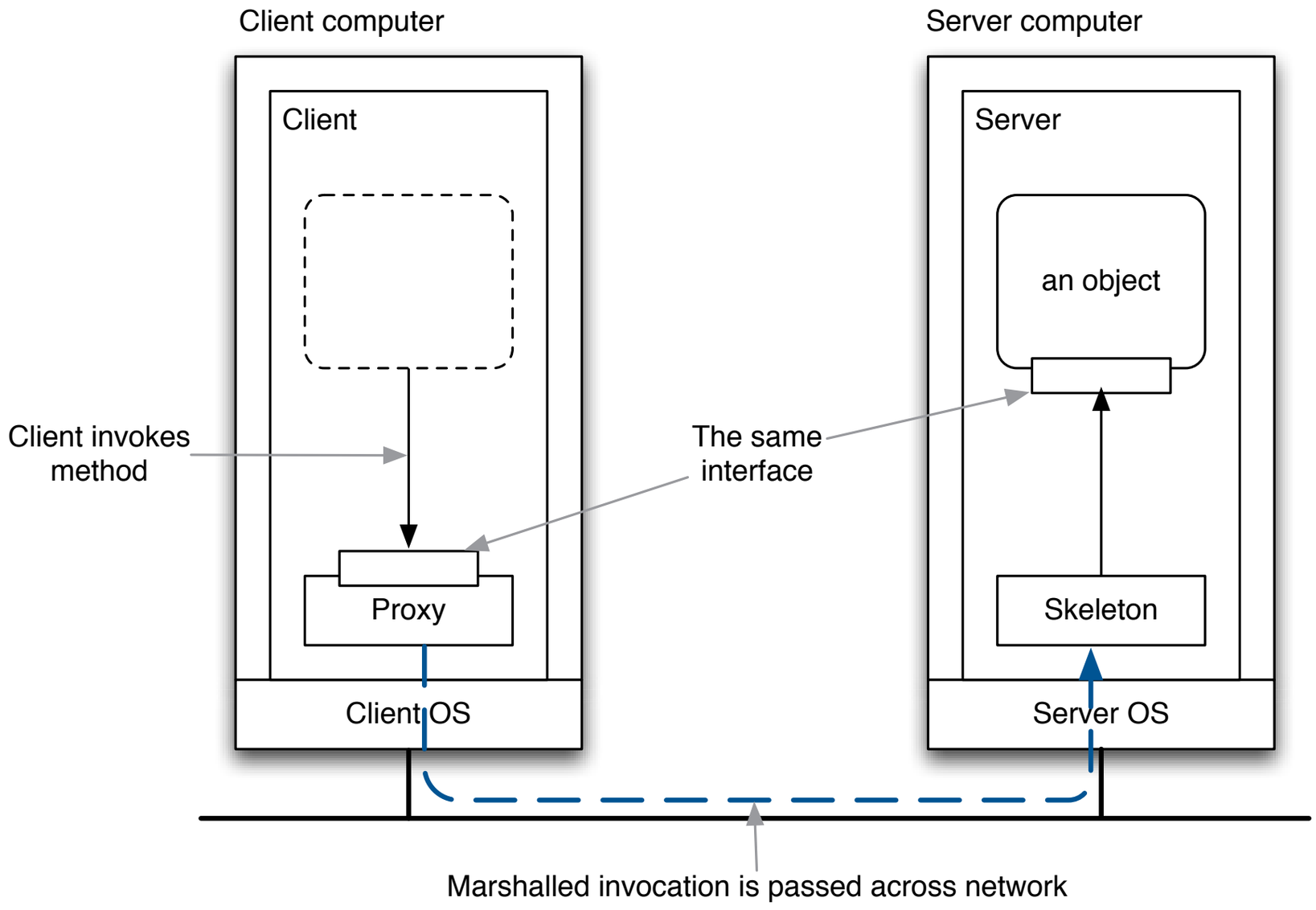
Distributed Object Model

- The remote interface specifies which methods can be invoked remotely
- Java Interfaces are used as the **Interface Description Language**



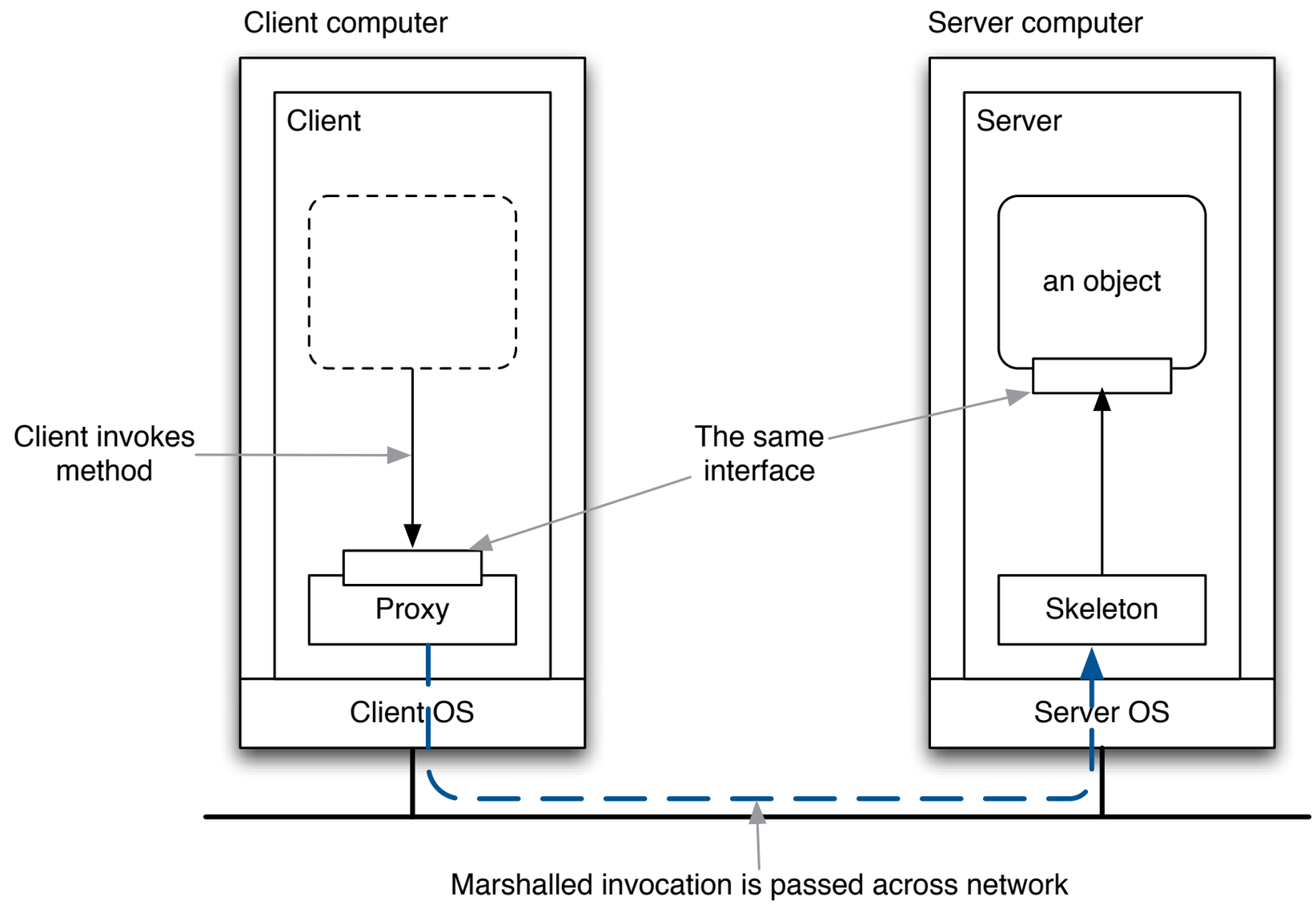
Anatomy of a Remote Method Invocation

- The **Proxy** ...
  - makes the RMI transparent to client
  - implements the remote interface
  - marshals requests and unmarshals results
  - forwards request



Anatomy of a Remote Method Invocation

- The **Skeleton** ...
  - implements the methods in remote interface
  - unmarshals requests and marshals results
  - invokes method in remote object



Remote Object References

Remote object references are system-wide unique:

- Can be passed freely between processes (e.g. as a parameter)
- The implementation of the remote object references is hidden by the middleware
  (Opaque references.)
- Improved distribution transparency compared to RPC

Interface Language

- No need for a dedicated language as RMI is targeted to a Java-only environment
- Uses Java interfaces extending `java.rmi.Remote`

Generation of Stubs and Binding

- RMI Compiler (**rmic**) generates stub and skeleton classes

No longer required to be called explicitly.

- Either static binding or dynamic binding via `Naming.lookup`
- Dynamic binding is supported by the **rmiregistry**
  - Starts a registry process for registering server objects
  - Enables binding of objects to names

Make sure the classpath is configured appropriately!

An Example Application

- Clients invoke a method `getTime()` on a remote object
- The server returns a `java.util.Date` instance representing the current server time
- Focus on JRMP only

RMI Development Process

1. Define the remote interface
2. Implement the RMI server
3. (Before Java 5) Generate RMI stubs (and if you use Java 1.1 skeletons)
4. Implement a server registrar
5. Implement an RMI client

The Time Server Interface

- All RMI interfaces must extend `java.rmi.Remote`
- All methods must throw `java.rmi.RemoteException`

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface Time extends Remote {

     public Date getTime() throws RemoteException;
}
```

The Time Server Implementation

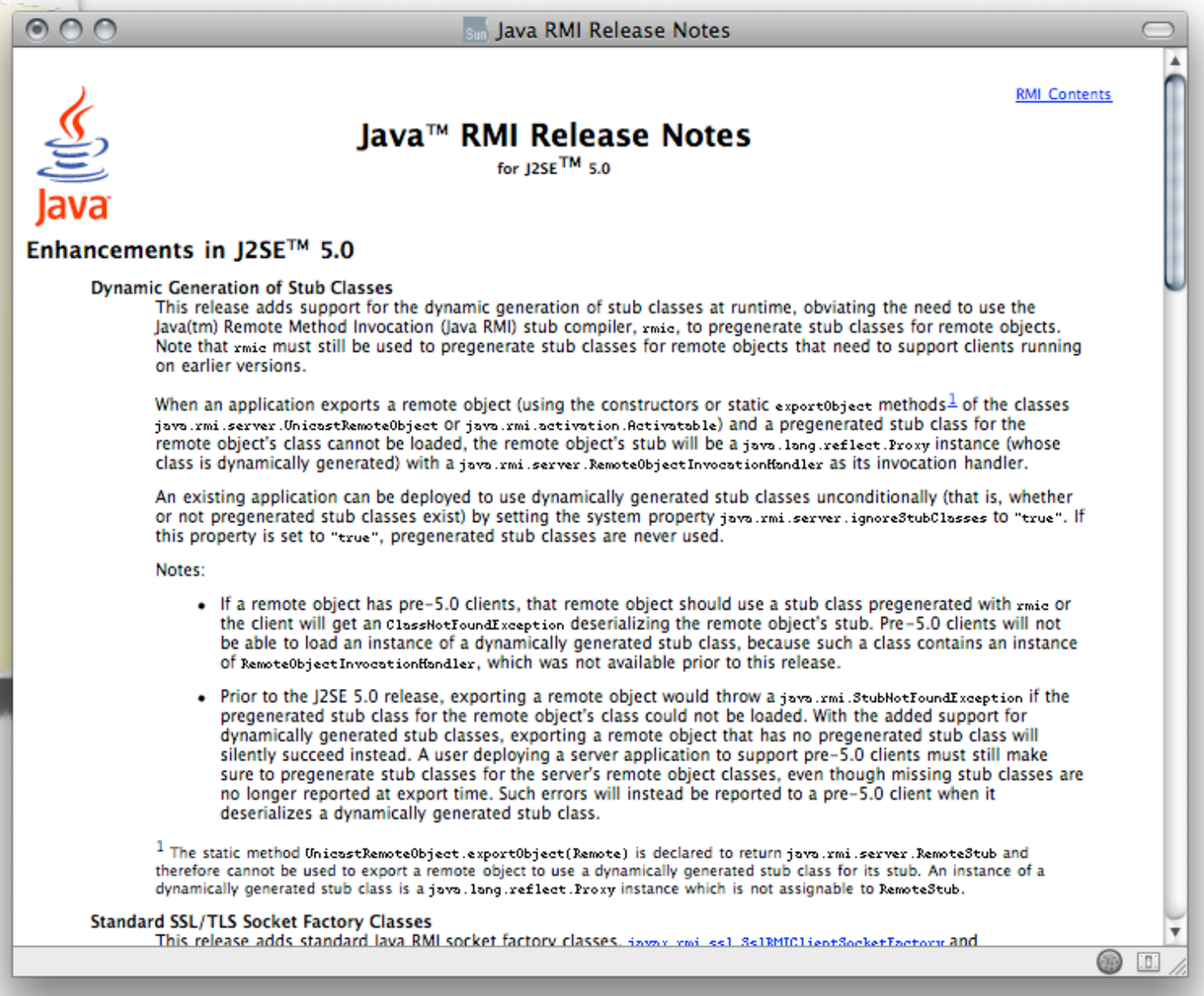
- Active JRMP servers...
  - must extend `java.rmi.UnicastRemoteObject`
  - must implement the interface

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class TimeServer extends UnicastRemoteObject implements Time {

     public TimeServer() throws RemoteException { super(); }

     public Date getTime() throws RemoteException {
          return new Date(System.currentTimeMillis());
}    }
```

Generating Stubs for the Time Server

- Stubs (and skeletons) are generated from compiled code by invoking rmic

```
> ls rmitest/server
TimeServer.class  TimeServer.java

> rmic rmitest.server.TimeServer
> ls rmitest/server
TimeServer.class  TimeServer_Skel.class
TimeServer.java   TimeServer_Stub.class
```



The Time Server: Registrar

- Responsible for binding a *server object* to a name.
- Server class may be the registrar itself (main method).
- Here: dedicated registrar class.

```
import java.rmi.Naming;

public class TimeRegistrar {

     /** @param args args[0] has to specify the hostname. */
     public static void main(String[] args) throws Exception {
          String host = args[0];
          TimeServer timeServer = new TimeServer();
          Naming.rebind("rmi://" + host + "/ServerTime", timeServer);
     }
}
```

The Time Server: Client

```
import java.rmi.Naming;

public class TimeClient {

    public static void main(String[] args) throws Exception {

        String host = args[0];
        Time time = (Time) Naming.lookup("rmi://" + host + "/ServerTime");
        System.err.println("Server reports: " + time.getTime().toString());
    }
}
```

Starting the Example

1. Start **rmiregistry** tool on server
2. Start `TimeRegistrar` on server
3. Start `TimeClient` on client
4. “The” `Time` interface need to be available on both server and client
   (Code downloading is - by default - not enabled.)

Garbage Collection (GC)

- RMI uses a reference-counting garbage collection algorithm
- When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference
  (Allows the JVM's GC to discard the object (if no other local references exist.)
- Network Problems may lead to premature GC; calls then result in Exceptions

Failure Handling

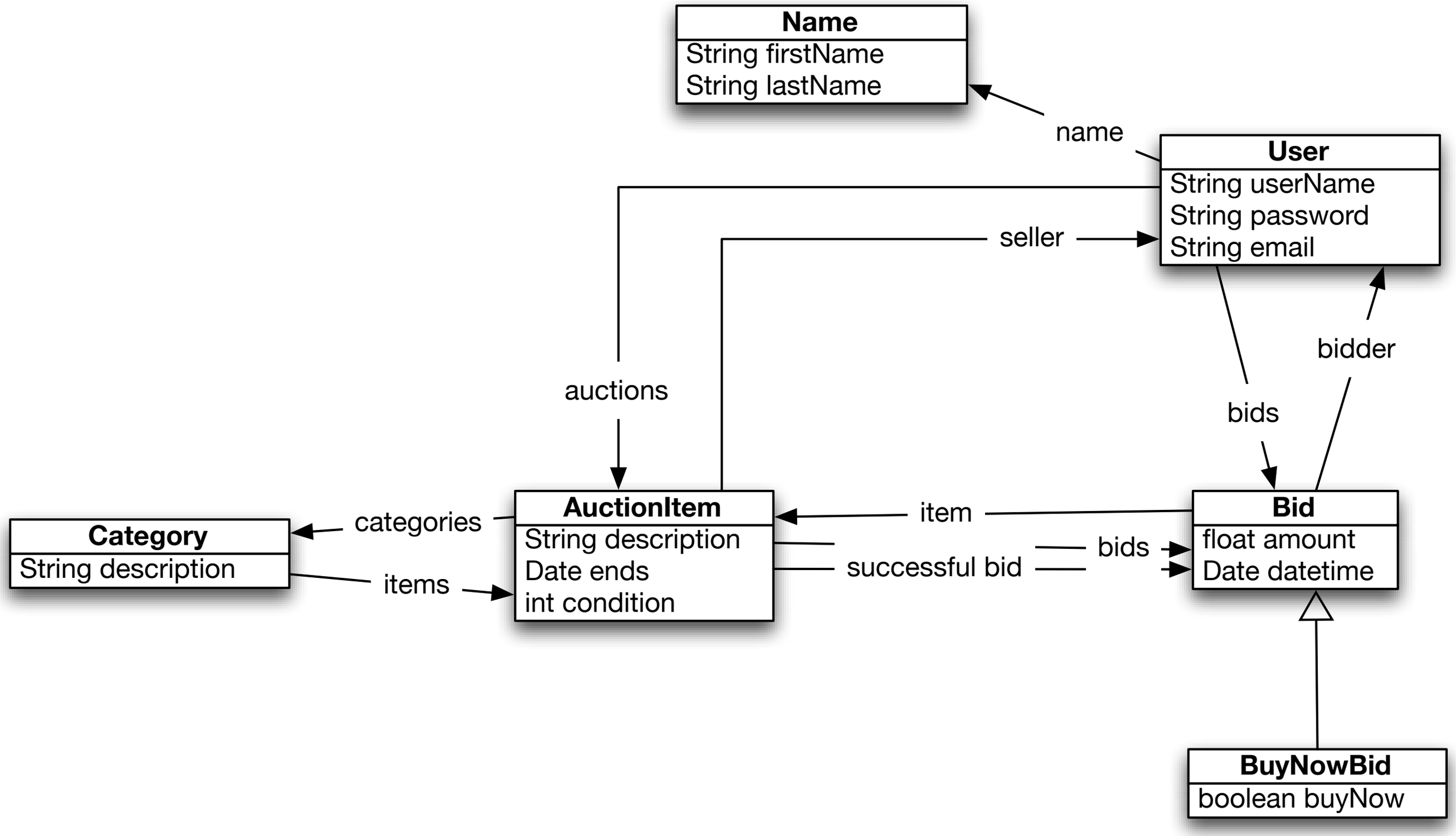
- Network Problems via Java Exceptions extending `java.rmi.RemoteException`
  (All methods in the remote interface have to declare that they may throw this exception.)
- Call Semantics: “at most once” semantics for remote calls

How is the data transfered between the client and the server?

(I.e., what happens when we sent a Date object from the server to the client?)

“Domain Model” of a simple Auction Application

Example used for the discussion in the following.



Java Serialization

„Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for developers to handle object serialization.“

[The Java API Documentation]

Java Serialization

- The API defines a default protocol, which can also be customized...
- by implementing
  `readObject(ObjectInputStream in)`
  and
  `writeObject(ObjectOutputStream out)`,
  you can do additional steps before and after default serialization.
- implementing the `Externalizable` Interface gives you full control over the protocol

Serializing an `AuctionItem`

Java Serialization - Example

- Serializable classes have to „implement“ the marker interface `java.io.Serializable` (marker interfaces do not define methods)

```
public class AuctionItem implements Serializable {
    // persistent fields
    private String description;
    private Date ends;
    private int condition;
    private List<Category> categories;
    private List<Bid> bids;
    private Bid successful_bid;
    private User seller;

    // non-persistent fields
    private transient SimpleDateFormat nonPersistentField;
    ...
}
```

Serializing an `AuctionItem`

Java Serialization - Example

```
public class SerializationTest {

    private AuctionItem item;

    void saveItem() throws IOException {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("AuctionItem.ser");
            ObjectOutputStream out = new ObjectOutputStream(fos);
            out.writeObject(item);
            out.flush();
        } finally {
            if (fos != null)
                fos.close();
        }
    }
    ...
}
```

*De*serializing an `AuctionItem`

Java Serialization - Example

```
public class SerializationTest {
    ...

    AuctionItem loadItem() throws IOException, ClassNotFoundException {

        FileInputStream fin = null;
        try {
            fin = new FileInputStream("AuctionItem.ser");
            ObjectInputStream in = new ObjectInputStream(fin);
            item = (AuctionItem) in.readObject();
        } finally {
            if (fin != null)
                fin.close();
        }
        return item;
    }

    ...
}
```

Java Serialization

Summary

Pros:

- Built into the Java language → no libraries necessary
- Ability to persist graph of objects, not just single ones
- Can also handle reference circles automatically
- Low implementation effort:
  - just use the marker interface `Serializable`
  - mark non-persistent fields as `transient`
- Default protocol can be overwritten (customized)

Java Serialization

Summary

Cons:

- Usually, stored in binary format → other (non-Java) applications cannot access the data; not recommended for long term storage (if a class evolves the file format will break)
- Only access of the whole object-graph possible; *there is no mean to query or update a single object independently*
  (This property makes a naïve use in the context of a distributed application problematic!)
- Customization requires you to write persistence code

Java Serialization

Summary

Conclusions:

- Serialization can do a good job for persisting data only needed by a single application which always needs the full data set and where the data will not be stored for long
- For large-scale applications serialization lacks crucial features like partial loading / updates, queries, storage in a DB, etc.

Java Serialization and RMI

A Short Example

Scala

```scala
package de.tud.cs.stg.ctfda.rmi

import _root_.java.rmi.server.UnicastRemoteObject

// The business entity.
case class Customer(val id: Long, var name: String, var familyName: String)
// this class is serializable; it (implicitly) inherits from Serializable

// The interface of the business object.
@remote // … after that all methods declare to throw a java.rmi.RemoteException
trait CustomerAccess {
  def customers(): List[Customer]
}

// The remote object.
class CustomerAccessImpl extends UnicastRemoteObject with CustomerAccess {
  val customers = Customer(1l, "M.", "Eichberg") :: Nil
}
```

Java Serialization and RMI

A Short Example - The Server


```scala
object CustomersServer extends App {

  import java.rmi.Naming

  java.rmi.registry.LocateRegistry.createRegistry(1099);

  val host = "localhost" // args(0);
  val service = "rmi://" + host + "/Customers"
  val remoteObject = new CustomerAccessImpl
  Naming.rebind(service, remoteObject);
  println("Server is up and running; press any key to shut down.");
  System.in.read

  …

  Naming.unbind(service)
  UnicastRemoteObject.unexportObject(remoteObject, false) // TODO: in loop
  println("Server terminated")
}
```

Java Serialization and RMI

A Short Example - The Client

Scala

```scala
object CustomersClient extends App {

  import java.rmi.Naming.lookup

  val host = "localhost" // args(0)
  val ca = lookup("rmi://" + host + "/Customers").asInstanceOf[CustomerAccess];
  println("customers: " + ca.customers)
}
```

Marshalling via Java Serialization

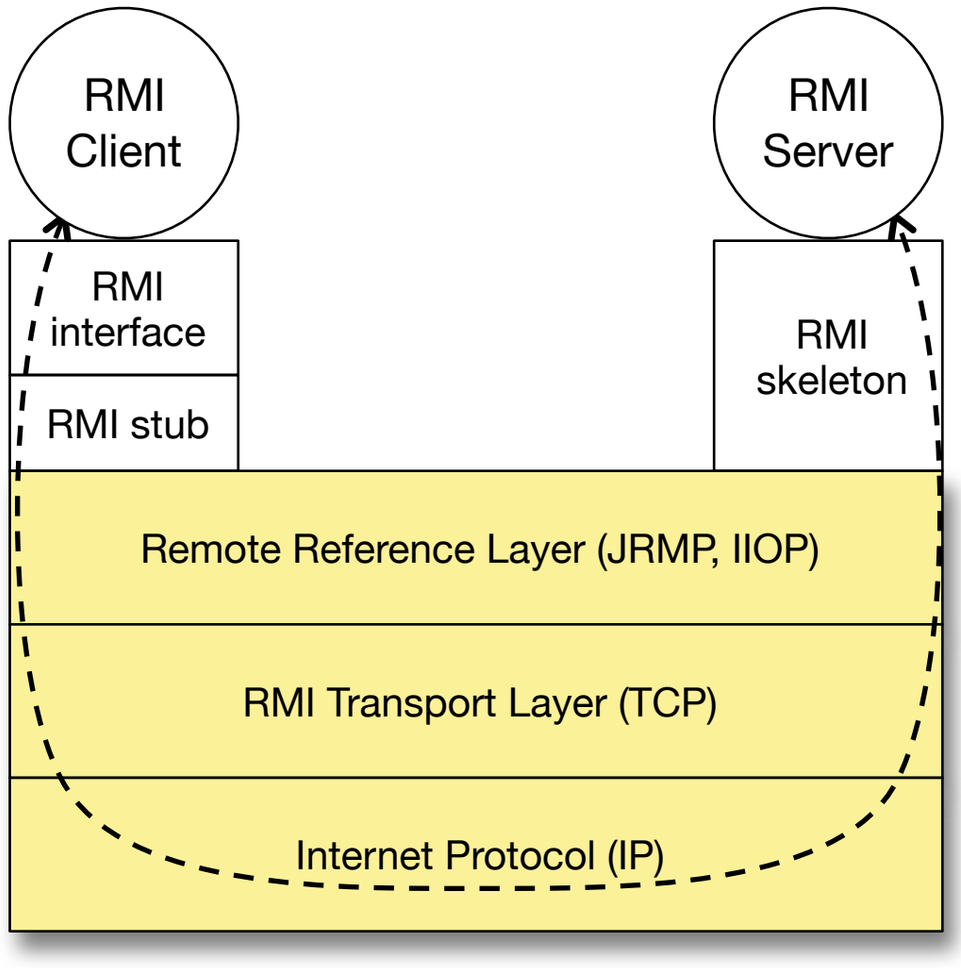
- **Serialization refers to the activity of flattening an object or a connected set of objects into a serial form** that is suitable for transmitting in a message
  - Both primitive data types and objects can be serialized
  - The receiver has no prior knowledge of the types of the objects in the serialized form
  - Supports reflection (the ability to enquire about the properties of a class)
- Objects are prepared for transport with Java serialization using `java.io.Serializable`
  - No problems with conversions - advantage of being Java-only
  - Overhead: accounts for ~25%-50% of the time for a remote invocation

Marshalling of Objects

- **Referential integrity is maintained in a remote call**
  If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM.
- **Referential integrity is not maintained across remote calls**

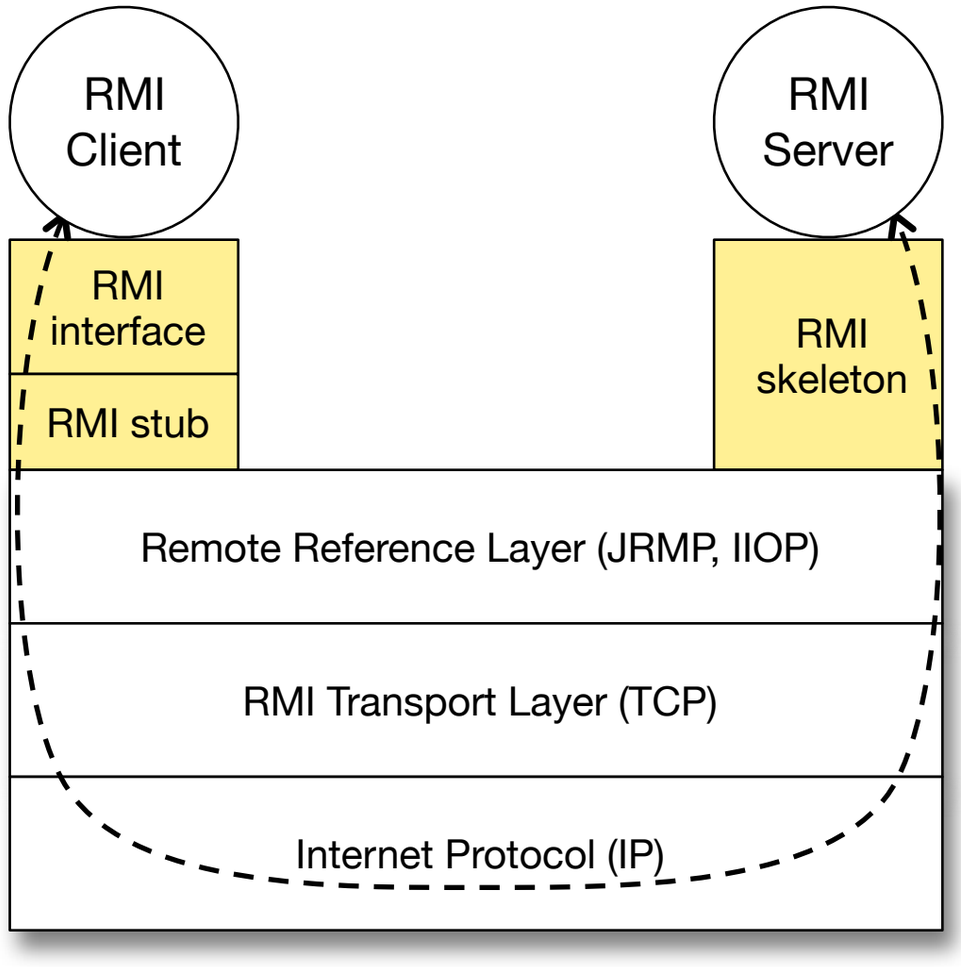
Anatomy of an RMI-based Architecture

- **Remote Reference Layer**
  RMI-specific communication atop TCP/IP; connection initialization, server location, …; handles serialized data
- **RMI Transport Layer (TCP)**
  Connection management; provision of reliable data transfer between end points
- **Internet Protocol**
  Data transfer contained in IP packets (lowest level)



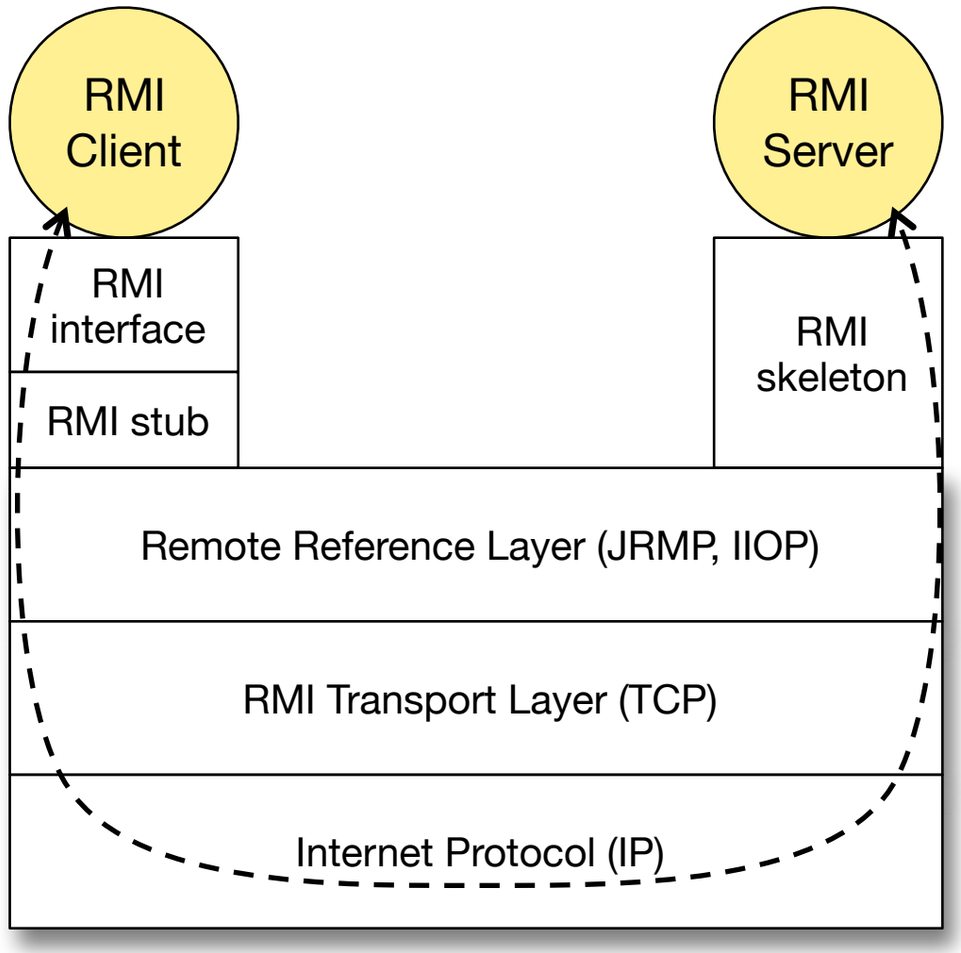
Anatomy of an RMI-based Architecture

- **RMI Interface**
  Defines what operations are made available to clients; the only class the client must know
- **RMI Stub**
  Implements the interface; takes calls from the client; performs marshaling / unmarshaling
- **RMI Skeleton**
  Calls the server methods; performs unmarshaling / marshaling



Anatomy of an RMI-based Architecture

- **RMI Server**
  Implements the interface; actually provides defined functionality
- **RMI Client**
  Some application relying on the interface; only needs to have the interface in its class path



RMI Capabilities

- **Parameter passing <u>either by reference or by value</u>:**
  - **Primitive values and serializable objects are passed by value**
  - **Remote objects are passed by reference**
- Dynamic loading of unknown classes
  A serialized object is automatically annotated with a descriptor (e.g., URL) where the byte code of its class can be downloaded.
- Tunneling through HTTP in case of firewall presence possible

RMI Packages

- `java.rmi`
  Core package
- `java.rmi.server`
  Core package for server-side classes and interfaces
- `java.rmi.registry`
  For interfacing with the RMI lookup service
- `java.rmi.activation`
  Classes and interfaces for implementing server objects that can be activated upon client request
- `java.rmi.dgc`
  Distributed garbage collection

RMI - Recommended Article

Understanding Java RMI Internals
By Ahamed Aslam.K
January 6, 2005

www.developer.com/java/ent/article.php/3455311

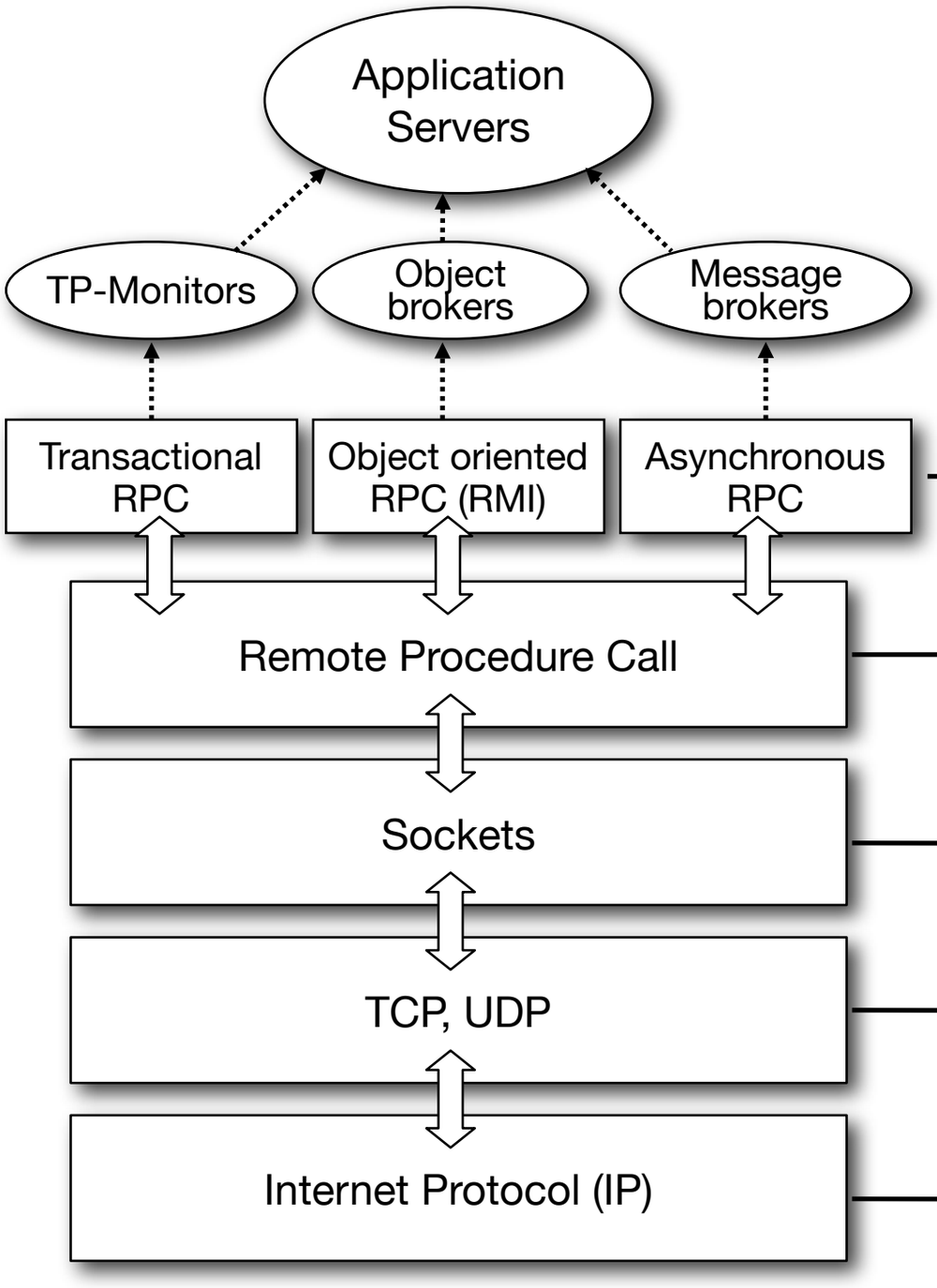
Developing Distributed Enterprise Applications

Middleware, RPC, RMI - a Summary

RMI *vs. RPC*

- RMI is object-oriented:
  - Whole objects can be passed
    („normal“ objects by-value, remote objects by-reference)
  - Subtype polymorphism can be used to deal with implementation variants
- With RPC, basically only primitive types can be passed
- RMI enables behavior mobility
  (Both data and code can be transferred from client to server and vice versa.)
- RMI supports automatic distributed garbage collection
  (In RPC the programmer has to take care of garbage collection.)
- RMI has a built-in security concept
  (In RPC an additional security layer is required.)
- RPC can be used in many different languages and environments
  (With RMI, everything needs to be Java (although …))

The Genealogy of Middleware



Specialized RPC, typically with additional functionality or properties but almost always running on RPC platforms.

Remote Procedure Call:
hides communication details behind a procedure call and helps bridge heterogeneous platforms.

Sockets:
operating system level interface to the underlying communication protocols

TCP, UDP:
User Datagram Protocol (UDP) transports data packets without guarantees.
Transmission Control Protocol (TCP) verifies correct delivery of data streams.

Internet Protocol (IP):
moves a packet of data from one node to another

Historic Development