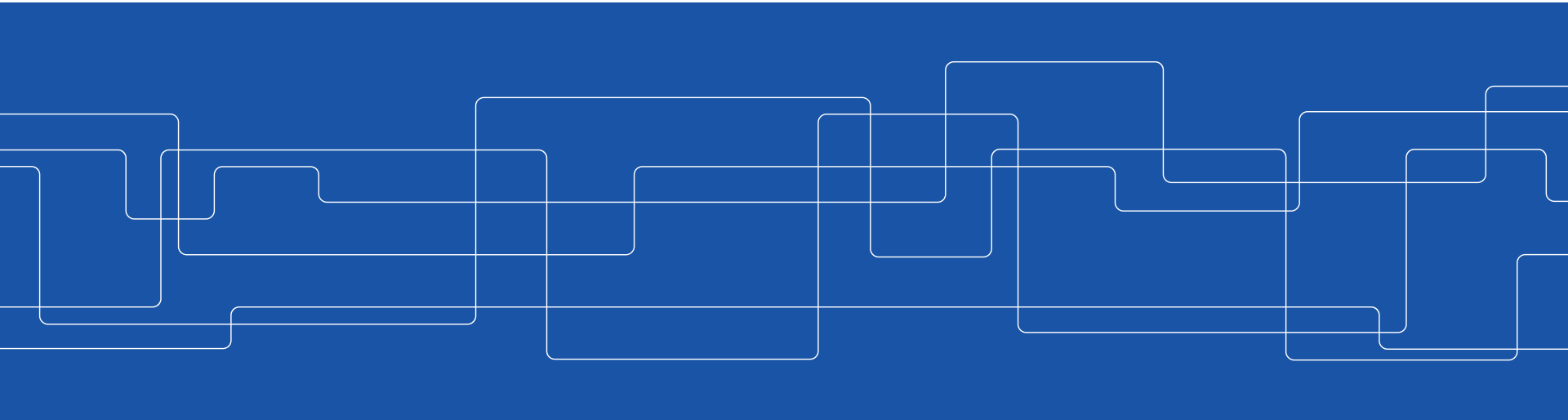




# Remote Invocation

Vladimir Vlassov and Johan Montelius





# Middleware

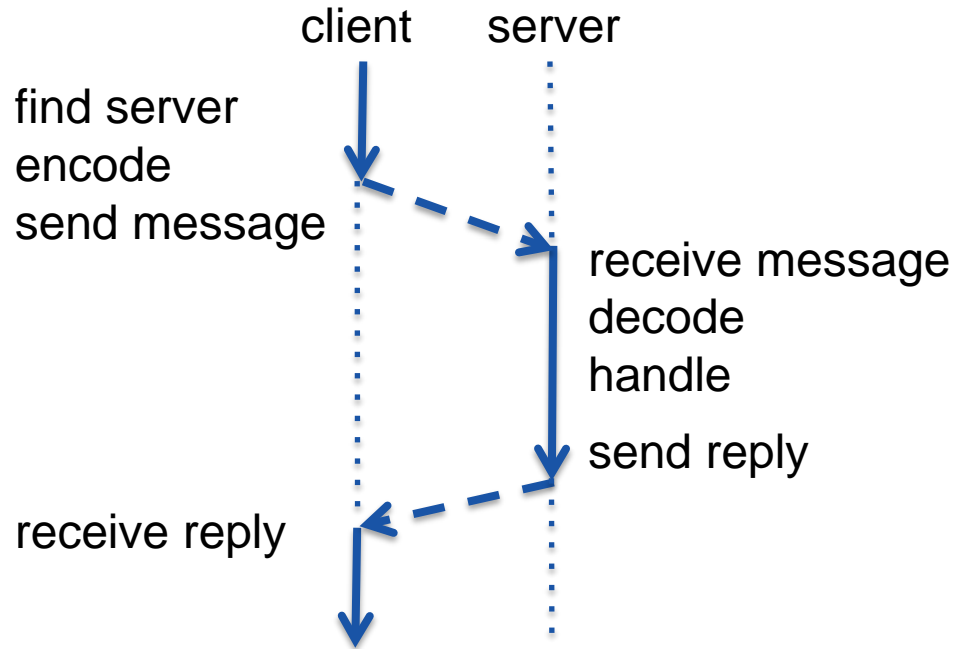
Application layer

Remote invocation / indirect communication

Socket layer

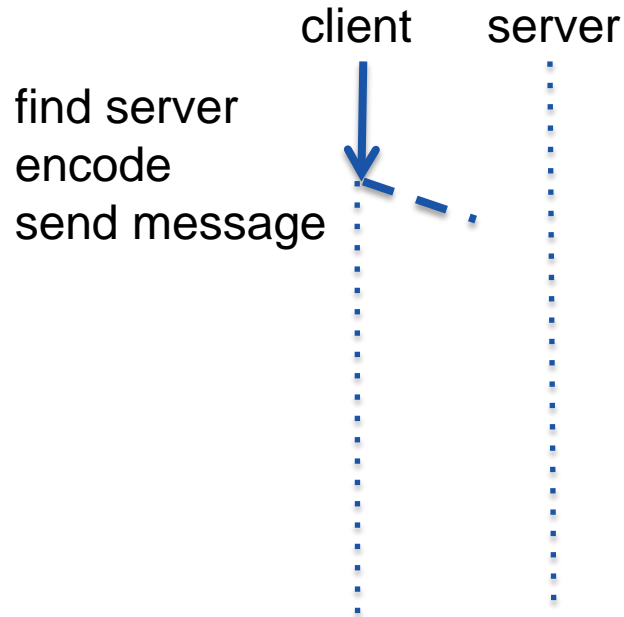
Network layer

# Request / Reply



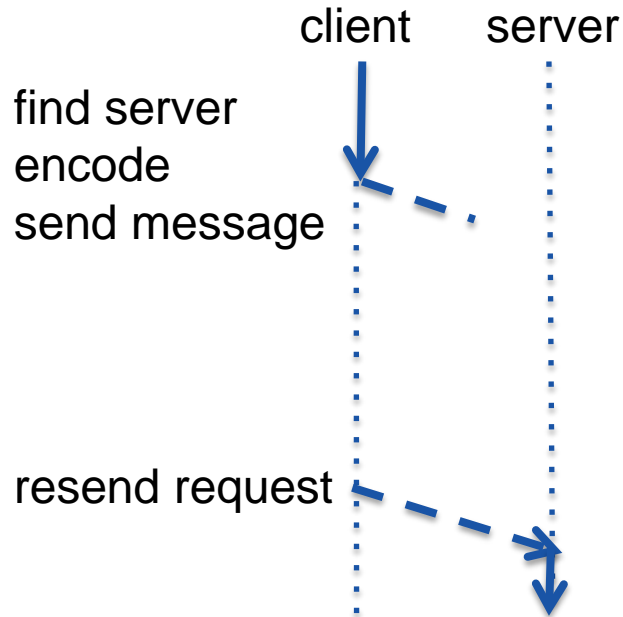
- identify and locate the server
- encode/decode the message
- send reply to the right client
- attach reply to request

# Lost request



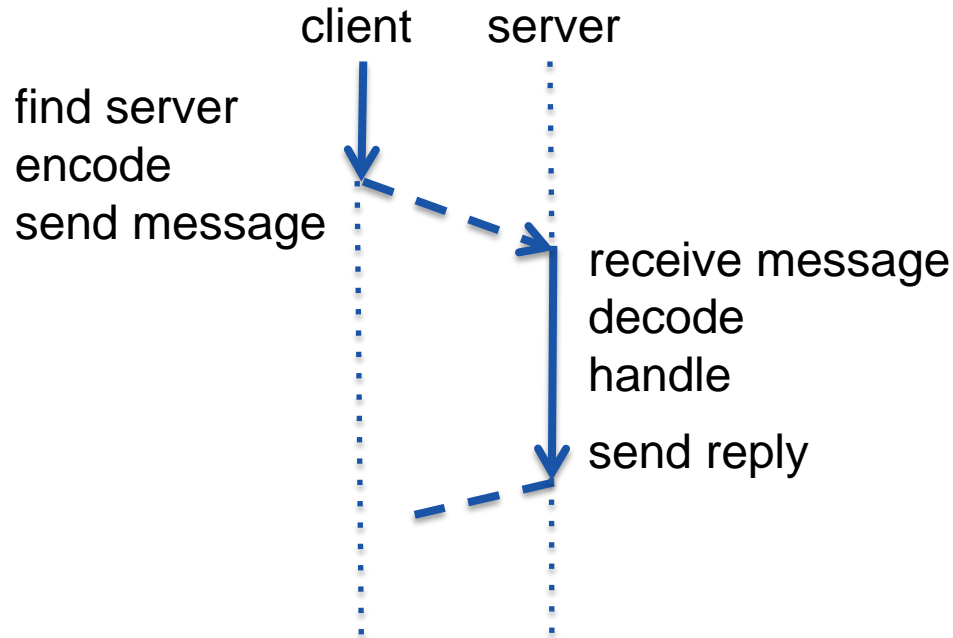
What do we do if **request is lost**?

# Resend request



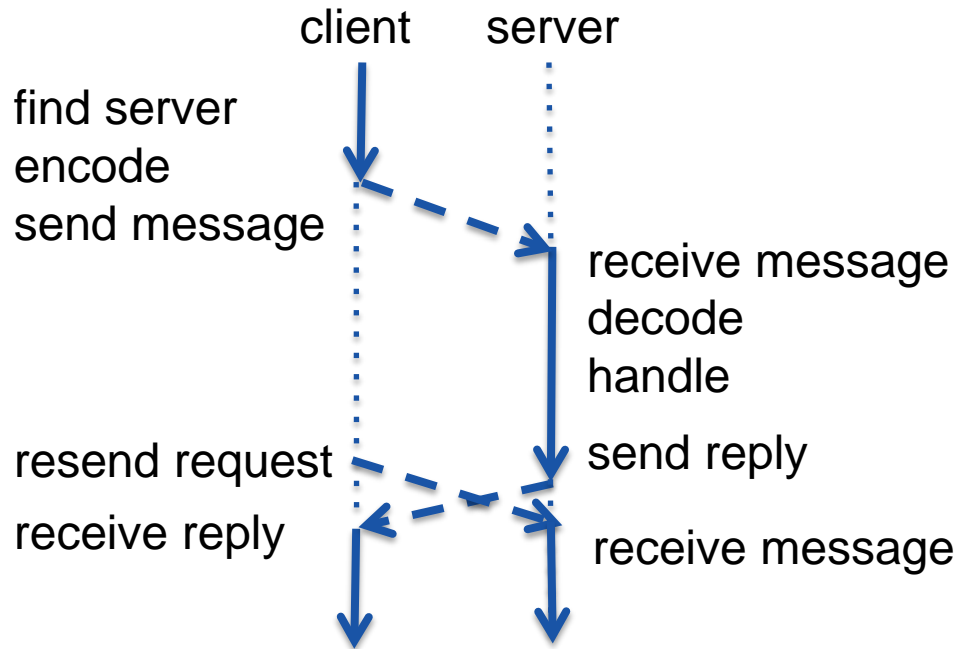
- need to detect that message is potentially lost
- wait for a **timeout** (how long) or error from underlying layer
- **resend** the request
- simple, problem solved

# Lost reply



- client will wait for **timeout** and **re-send request**
- not a problem

# Problem



- a problem
- server **might need a history of all previous request**
- ***might need***



# Idempotent operations

- add 100 euros to my account
- what is the status of my account
- Sweden scored yet another goal!
- The standing is now 2-1!





# History

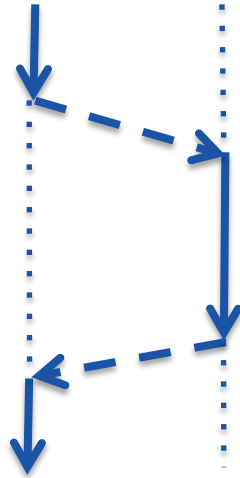
If operations are **not idempotent**, the server must make sure that the same request is **not executed twice**.

Keep a **history of all request and the replies**. If a request is resent the same reply can be sent without re-execution.

For how long do you keep the history?

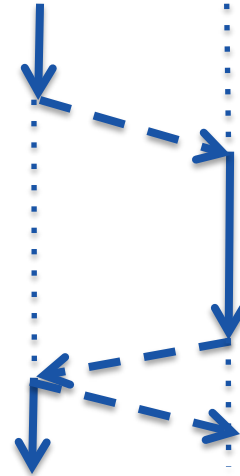
# Request-Reply-Acknowledge

client      server



Request-Reply (RR)

client      server



Request-Reply-Acknowledge (RRA)

# At-most-once or At-least-once

How about this:

If an operation **succeeds**, then..

**at-most-once**: the request has been **executed once**.

*Implemented using a history or simply not re-sending requests.*

**at-least-once**: the request has been **executed at least once**.

*No need for a history, simply resend requests until a reply is received.*



# At most or At least

How about **errors**:

*Even if we do resend messages we will have to give up at some time.*

If an operation **fails/is lost**, then..

***at-most-once:***

***at-least-once:***

# At most or At least

Pros and cons:

- *at-most-once without re-sending requests:*  
simple to implement, not fault-tolerant
- *at-most-once with history:*  
expensive to implement, fault-tolerant
- *at-least-once:*  
simple to implement, fault-tolerant

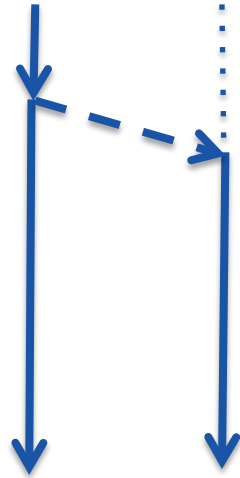
***Can you live with at-least-once semantics?***



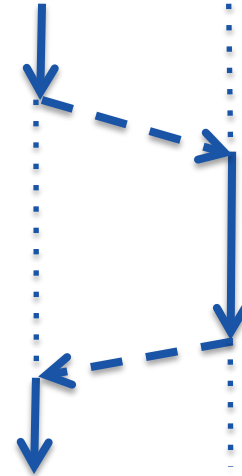
# UDP or TCP

Should we implement a request-reply protocol over UDP or TCP?

# Synchronous or Asynchronous

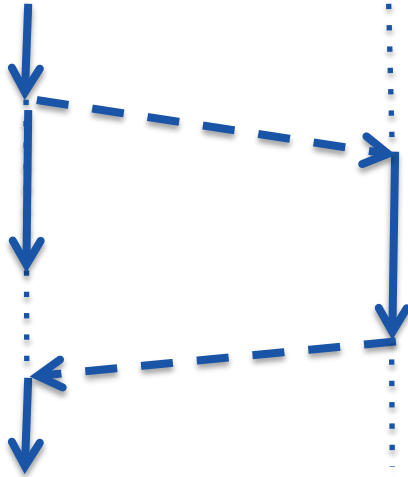


Asynchronous



Synchronous

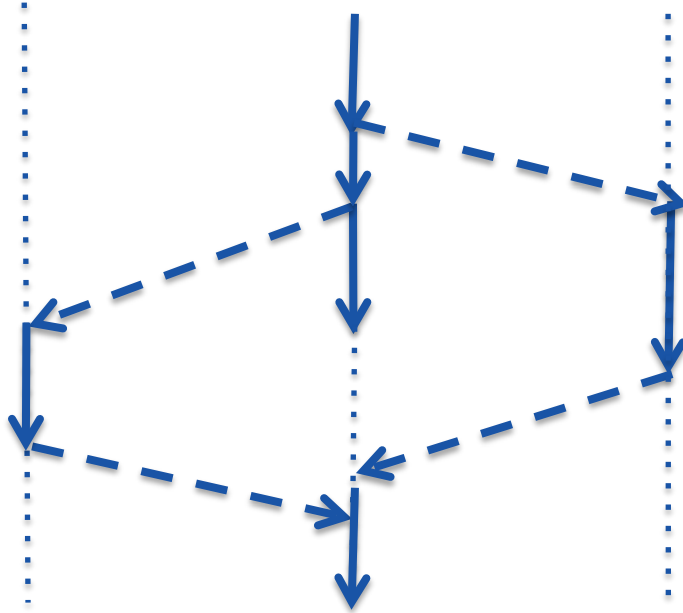
# RR over Asynchronous



- send request
- continue to execute
- suspend if not arrived
- read reply



# Hide the latency





# HTTP

A request reply protocol, described in RFC 2616.

Request = Request-Line \*(header CRLF) CRLF [ message-body ]

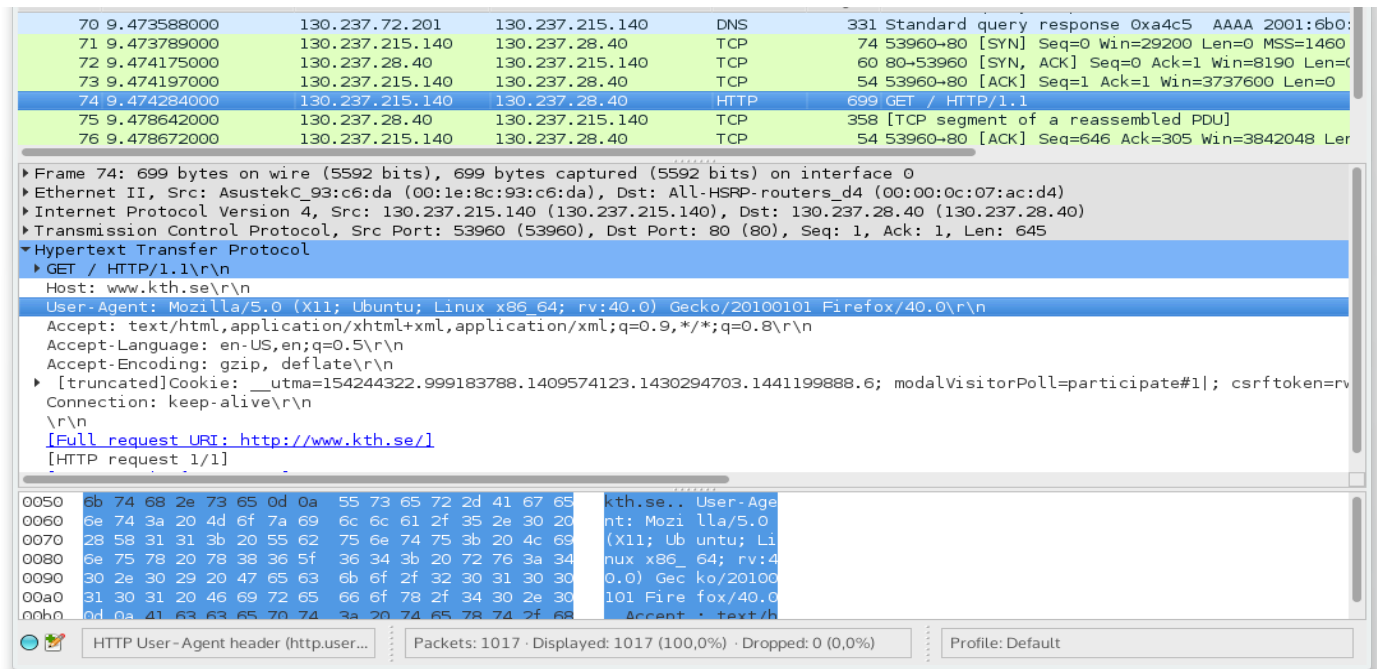
Request-Line = Method SP Request-URI SP HTTP-Version CRLF

GET /index.html HTTP/1.1\r\n foo 42 \r\n\r\nHello

# HTTP methods

- **GET**: request a resource, *should be idempotent*
- **HEAD**: request only header information
- **POST**: upload information to a resource, included in body, status of server could change
- **PUT**: add or replace a resource, idempotent
- **DELETE**: add or replace content, idempotent

# Wireshark



The image shows a Wireshark packet capture of an HTTP GET request. The top pane displays a list of packets, with packet 74 selected. The middle pane shows the details of packet 74, which is an HTTP GET request to <http://www.kth.se/>. The bottom pane shows the raw packet data in hexadecimal and ASCII.

**Packet List:**

No.	Time	Source	Destination	Protocol	Length	Info
70	9.473588000	130.237.72.201	130.237.215.140	DNS	331	Standard query response Oxa4c5 AAAA 2001:6b0:
71	9.473789000	130.237.215.140	130.237.28.40	TCP	74	53960→80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
72	9.474175000	130.237.28.40	130.237.215.140	TCP	60	80→53960 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0
73	9.474197000	130.237.215.140	130.237.28.40	TCP	54	53960→80 [ACK] Seq=1 Ack=1 Win=3737600 Len=0
74	9.474284000	130.237.215.140	130.237.28.40	HTTP	699	GET / HTTP/1.1
75	9.478642000	130.237.28.40	130.237.215.140	TCP	358	[TCP segment of a reassembled PDU]
76	9.478672000	130.237.215.140	130.237.28.40	TCP	54	53960→80 [ACK] Seq=646 Ack=305 Win=3842048 Len=0

**Packet Details:**

- Frame 74: 699 bytes on wire (5592 bits), 699 bytes captured (5592 bits) on interface 0
- Ethernet II, Src: AsustekC\_93:c6:da (00:1e:8c:93:c6:da), Dst: All-HSRP-routers\_d4 (00:00:0c:07:ac:d4)
- Internet Protocol Version 4, Src: 130.237.215.140 (130.237.215.140), Dst: 130.237.28.40 (130.237.28.40)
- Transmission Control Protocol, Src Port: 53960 (53960), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 645
- Hypertext Transfer Protocol
  - GET / HTTP/1.1\r\n
  - Host: www.kth.se\r\n
  - User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:40.0) Gecko/20100101 Firefox/40.0\r\n
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8\r\n
  - Accept-Language: en-US,en;q=0.5\r\n
  - Accept-Encoding: gzip, deflate\r\n
  - [truncated]Cookie: \_\_utma=154244322.999183788.1409574123.1430294703.1441199888.6; modalVisitorPoll=participate#1; csrftoken=r...
  - Connection: keep-alive\r\n
  - \r\n
  - [Full request URI: <http://www.kth.se/>]
  - [HTTP request 1/1]

**Raw Data:**

Offset	Hex	ASCII
0050	6b 74 68 2e 73 65 0d 0a 55 73 65 72 2d 41 67 65	kth.se.. User-Age
0060	6e 74 3a 20 4d 6f 7a 69 6c 6c 61 2f 35 2e 30 20	nt: Mozilla/5.0
0070	28 58 31 31 3b 20 55 62 75 6e 74 75 3b 20 4c 69	(X11; Ubuntu; Li
0080	6e 75 78 20 78 38 36 5f 36 34 3b 20 72 76 3a 34	nux x86_64; rv:4
0090	30 2e 30 29 20 47 65 63 6b 6f 2f 32 30 31 30 30	0.0) Gecko/20100
00a0	31 30 31 20 46 69 72 65 66 6f 78 2f 34 30 2e 30	101 Firefox/40.0
00b0	0d 0a 41 63 63 65 70 74 3a 20 74 65 78 74 2f 68	Accept: text/b

**Status Bar:**

- HTTP User-Agent header (http.user...)
- Packets: 1017 · Displayed: 1017 (100,0%) · Dropped: 0 (0,0%)
- Profile: Default



# HTTP GET

GET / HTTP/1.1

Host: www.kth.se

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86\_64; rv:40.0) Gecko/20100101

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

Cookie: .....

Connection: keep-alive



# HTTP Response

HTTP/1.1 200 OK

Date: Tue, 08 Sep 2015 10:37:49 GMT

Server: Apache/2.2.15 (Red Hat)

X-UA-Compatible: IE=edge

Set-Cookie: JSESSIONID=CDC76A3;Path=/; Secure; HttpOnly

Content-Language: sv-SE

Content-Length: 59507

Connection: close

Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>

<html lang="sv">

<title>KTH | Valkommen till KTH</title>



# The web

On the web the resource is often a HTML document that is presented in a browser.

HTTP could be used as a general-purpose request-reply protocol.

# REST and SOAP

Request-reply protocols for Web-services:

- **REST (Representational State Transfer)**
  - content described in XML, JSON, . . .
  - light weight,
- **SOAP (Simple Object Access Protocol)**
  - over HTTP, SMTP . . .
  - content described in SOAP/XML
  - standardized, heavy weight





# HTTP over TCP

HTTP over TCP - a good idea?



# Masking a request-reply

Could we use a regular program construct to **hide** the fact that we do a **request-reply**?



# Masking a request-reply

Could we use a regular program construct to **hide** the fact that we do a **request-reply**?

- **RPC**: Remote Procedure Call
- **RMI**: Remote Method Invocation

# Motivation for RPC and RMI

*Message passing* is convenient for consumers-producers (filters) and P2P, but it is somewhat low level for client-server applications

- Client/server interactions are based on a request/response protocol;
- Client requests are typically mapped to procedures on server;
- A client waits for a response from the server.

Need for more convenient (easier to use) communication mechanisms for developing client/server applications



# Motivation for RPC and RMI

## Remote Procedure Call (RPC) and rendezvous

- Procedure interface; message passing implementation

## Remote Method Invocation (RMI)

- RMI is an object-oriented analog of RPC

RPC, rendezvous and RMI are implemented on top of message passing.



# Procedure calls

What is a procedure call:

- find the procedure
- give the procedure access to arguments
- pass control to the procedure
- collect the reply if any
- continue execution

How do we turn this into **a tool for distributed programming?**



# Operational semantics

```
int x, n;  
n = 5;  
proc(n);  
x = n;
```

```
int x, arr[3];  
arr[0] = 5;  
proc(arr);  
x = arr[0];
```



# Call by value/reference

## Call by value

- A procedure is given a copy of the datum

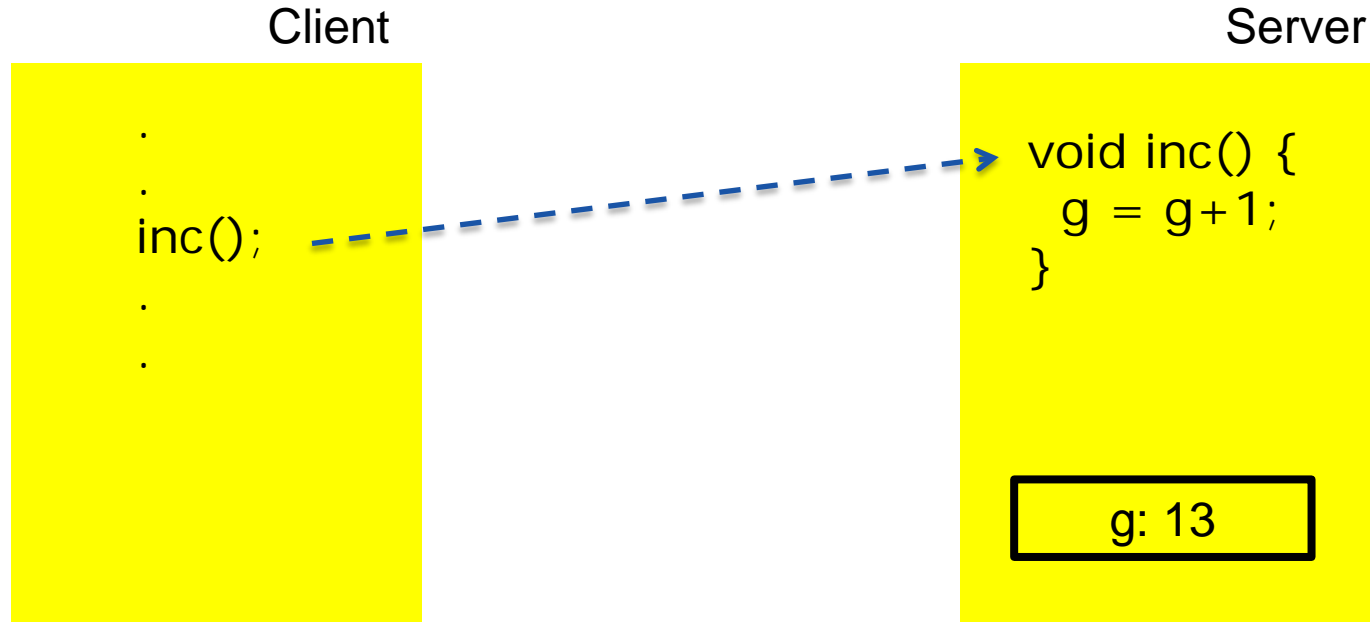
## Call by reference

- A procedure is given a reference to the datum

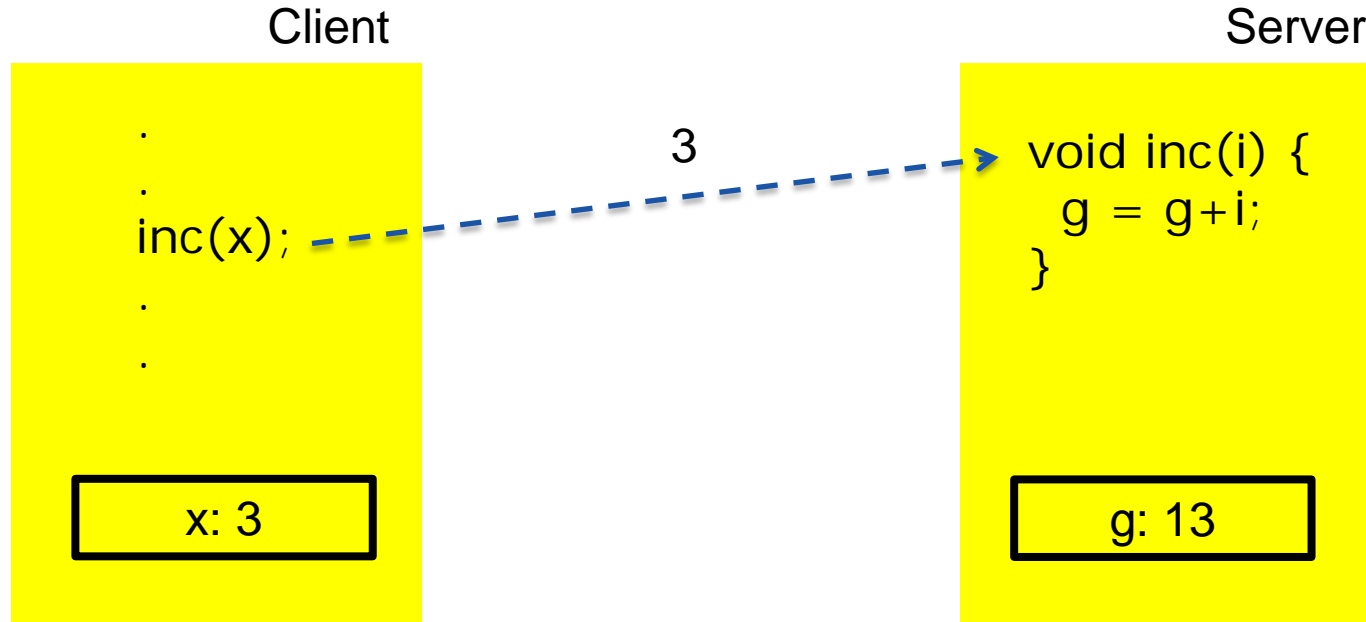
What if the datum is a reference and we pass a copy of the datum?  
Why is this important?



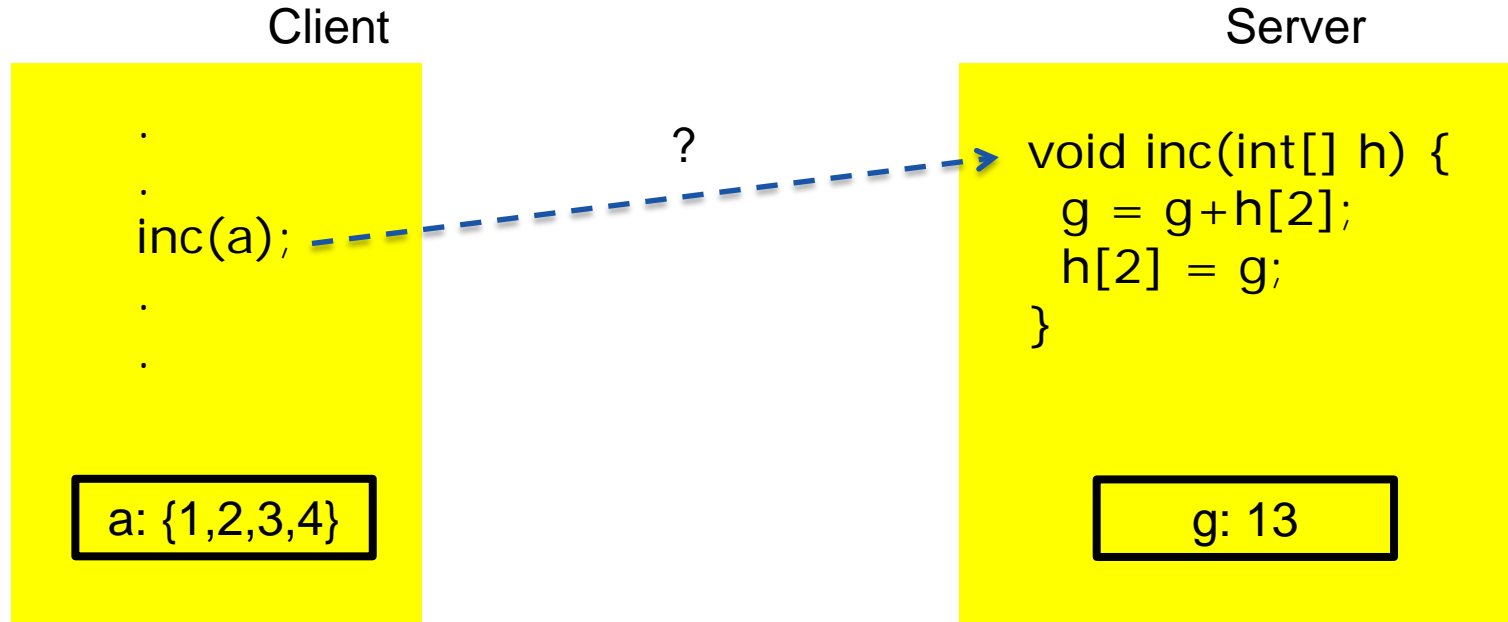
# RPC: Remote Procedure Call



# RPC: Remote Procedure Call



# RPC: Remote Procedure Call

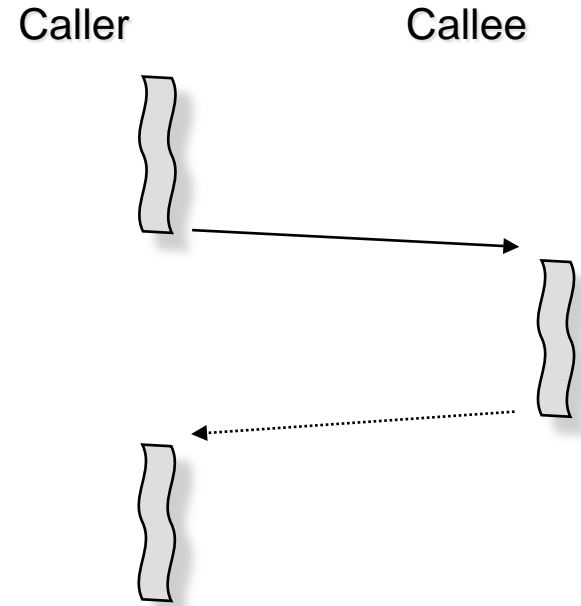


## RPC: Remote Procedure Call

**RPC** is a mechanism that allows a program running on one computer (VM) to cause a procedure to be executed on another computer (VM) without the programmer needing to explicitly code for this.

Two processes involved:

- **Caller (RPC client)** is a **calling process** that initiates an RPC to a server.
- **Callee (RPC server)** is a **called process** that accepts the call.

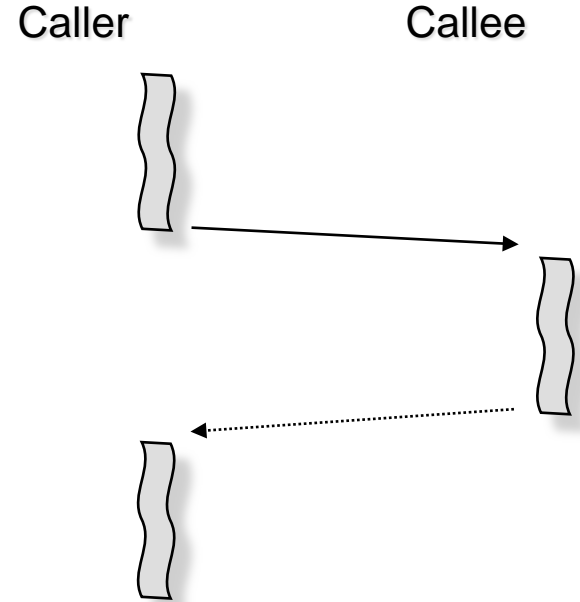


## RPC: Remote Procedure Call (cont'd)

Each RPC is executed in a **separate process (thread)** on the server side

**An RPC is a synchronous operation.**

- The caller is suspended until the results of the remote procedure are returned.
- Like a regular or local procedure call.
- Guess why?



# Identifying a Remote Procedure

Each RPC procedure is uniquely identified by

- A program number
  - identifies a group of related remote procedures
- A version number
- A procedure number

An RPC call message has three unsigned fields:

- Remote program number
- Remote program version number
- Remote procedure number

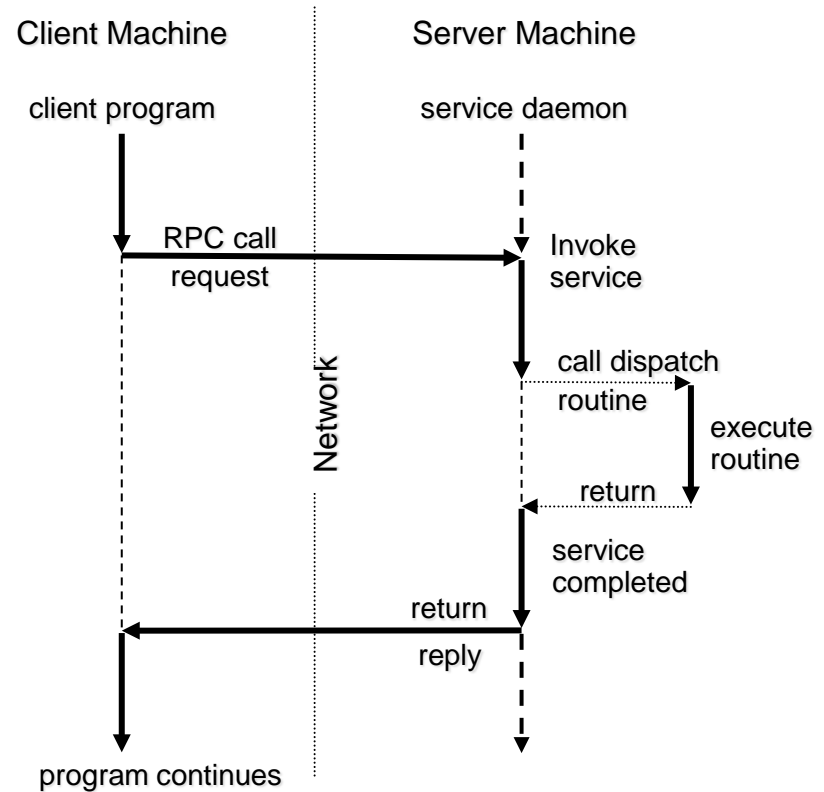
The three fields uniquely identify the procedure to be called.

# Executing RPC

On each RPC the server starts a **new process** to execute the call.

- The new process terminates when the procedure returns and results are sent to the caller.
- Calls from the same caller and calls from different callers are serviced by **different concurrent processes** on server.

Concurrent invocations might interfere with each other when accessing shared objects – might need **synchronization**





# Open Network Computing (ONC) RPC (SunRPC)

- targeting intranet, file servers etc
- **at-least-once** call semantics
- procedures described in **Interface Definition Language (IDL)**
- XDR (eXternal Data Representation) specifies message structure
- used UDP as transport protocol (TCP also available)





# Java RMI (Remote Method Invocation)

- similar to RPC but:
  - we now **invoke methods of remote objects**
  - **at-most-once** semantics
- Objects can be passed as arguments, how should this be done?
  - **by value**
  - **by reference**



# Java RMI

We can do either:

A *remote object* is passed as a reference (*by reference*) i.e. it remains as at the original place where it was created.

A *serializable object* is passed as a copy (*by value*) i.e. the object is duplicated.



# Finding the procedure/object

How do we locate a remote procedure/object/process?

Network address that specifies the location or..

a known “binder” process that keeps track of registered resources.

# Remote Method Invocation (RMI)

**Remote method invocation (RMI)** is a mechanism to invoke a method on remote object, i.e. object in another computer or virtual machine.

RMI is the object-oriented analog of RPC in an distributed OO environment, e.g. OMG CORBA, Java RMI, .NET

- RPC allows calling procedures over a network
- RMI invokes objects' methods over a network

**Location transparency:** invoke a method on a stub like on a local object

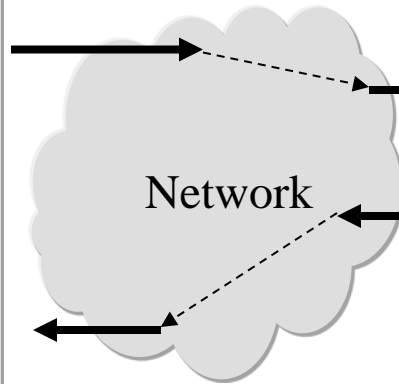
**Location awareness:** the stub makes remote call across a network and returns a results via stack

# Remote Method Invocation

↪ `r = a.m(x);`

```
a { // stub
  m(x) {
    1. Marshal x
    2. Send Msg w/ a, m, x

    8. Receive Msg w/ result
    9. Unmarshal result
    10. Return result
  }
}
```



```
a {
  m(x) {
    return x*5
  }
}

a_skeleton { // skeleton
  m( ) {
    3. Receive Msg
    4. Unmarshal x
    5. result = a.m(x)
    6. Marshal result
    7. Send Msg w/ result
  }
}
```

Up call

# Locating Objects

How does a caller get a reference to a remote object, i.e. stub?

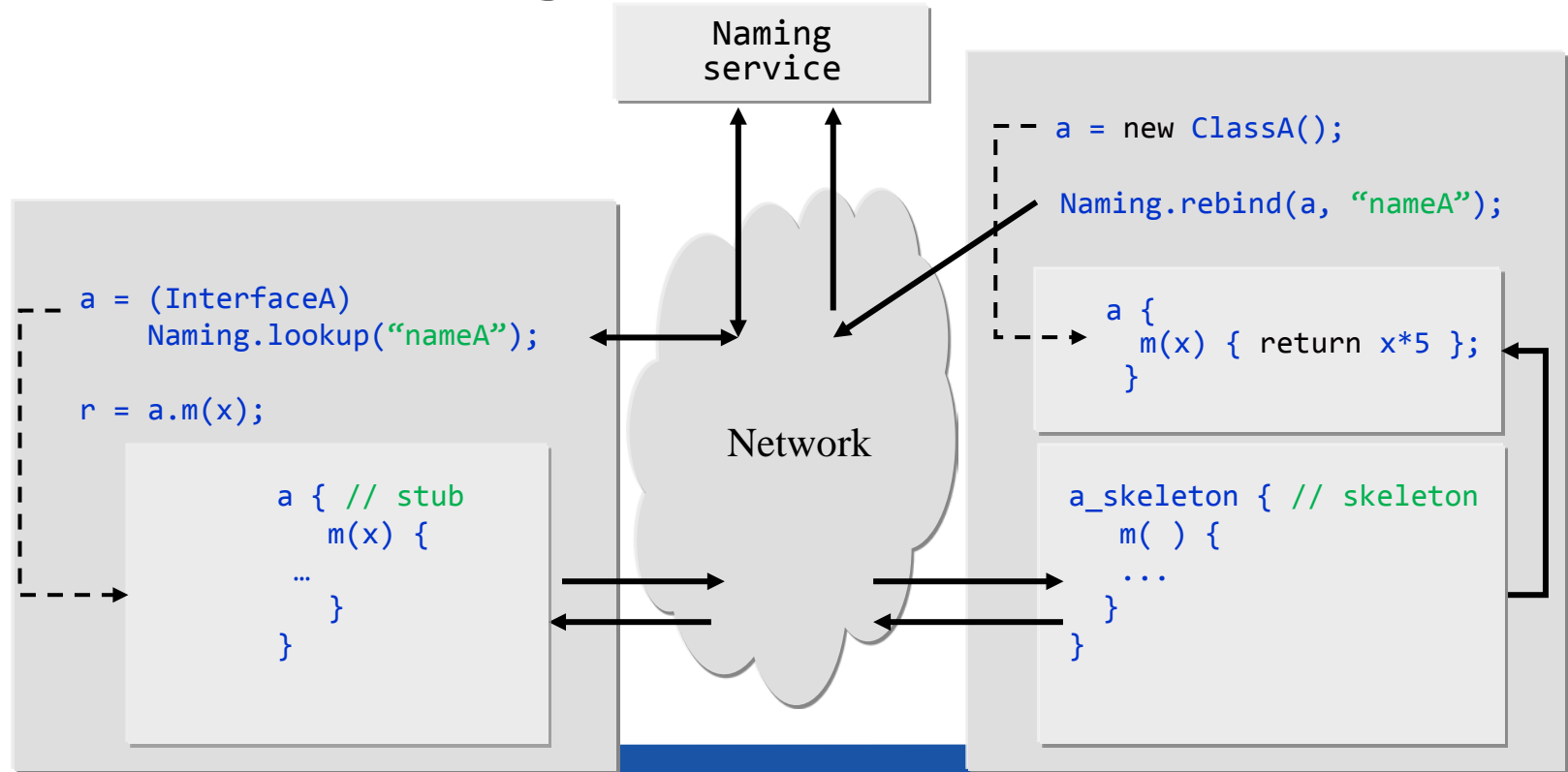
One approach is to use a ***Naming Service***:

- Associate a unique name with an object.
- Bind the name to the object at the Naming Service.
  - The record typically includes name, class name, object reference (i.e. location information) and other information to create a stub.
- The client looks up the object by name in the Naming Service.

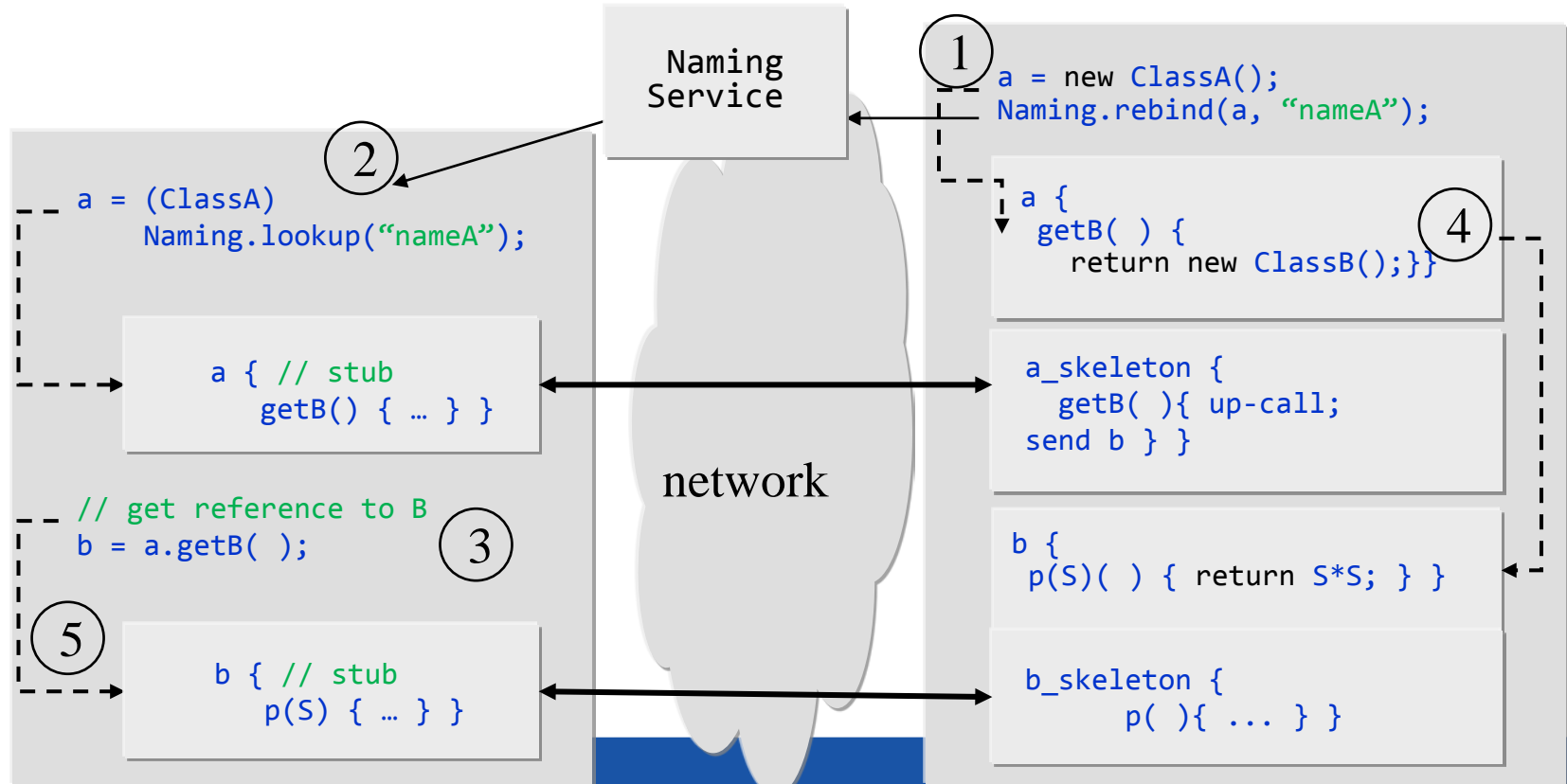
***The primary reference problem***: How to locate the Naming Service?

- Configuration problem: URL of the naming service

# Use of Naming Service



# Remote Reference in Return







# Remote invocation design decisions

- failure handling: maybe / at-most-once / at-least-once
- call-by-value / call-by-reference
- message specification and encoding
- specification of resource
- procedure binder – naming service

# Examples

- **SunRPC**: call-by-value, at-least-once, IDL, XDR, binder
- **JavaRMI**: call-by-value/reference, at-most-once, interface, JRMP (Java Remote Method Protocol), rmiregistry
- **Erlang**: message passing, maybe, no, ETF (External Term Format), local registry only
- **CORBA** (Common Object Request Broker Architecture): call-by-reference, IDL, ORB (Object Request Broker), tnameserv
- **Web Services**: WSDL (Web Services Description Language), UDDI (Universal Description, Discovery, and Integration)



# Java RMI (Remote Method Invocation)

**Java RMI** is a mechanism that allows a thread in one JVM to invoke a method on an object located in another JVM.

- Provides **Java native ORB (Object Request Broker)**

The Java RMI facility allows applications or applets running on different JVMs, to interact with each other by invoking remote methods:

- Remote reference (stub) is treated as local object.
- Method invocation on the reference causes the method to be executed on the remote JVM.
- Serialized arguments and return values are passed over network connections.
- Uses **Object streams** to pass objects “by value”.



# RMI Classes and Interfaces

## `java.rmi.Remote`

- Interface that indicates interfaces whose methods may be invoked from a non-local JVM -- remote interfaces.

## `java.rmi.Naming`

- The RMI Naming Service client that is used to bind a name to an object and to lookup an object by name.

## `java.rmi.RemoteException`

- The common superclass for a number of communication-related RMI exceptions.

## `java.rmi.server.UnicastRemoteObject`

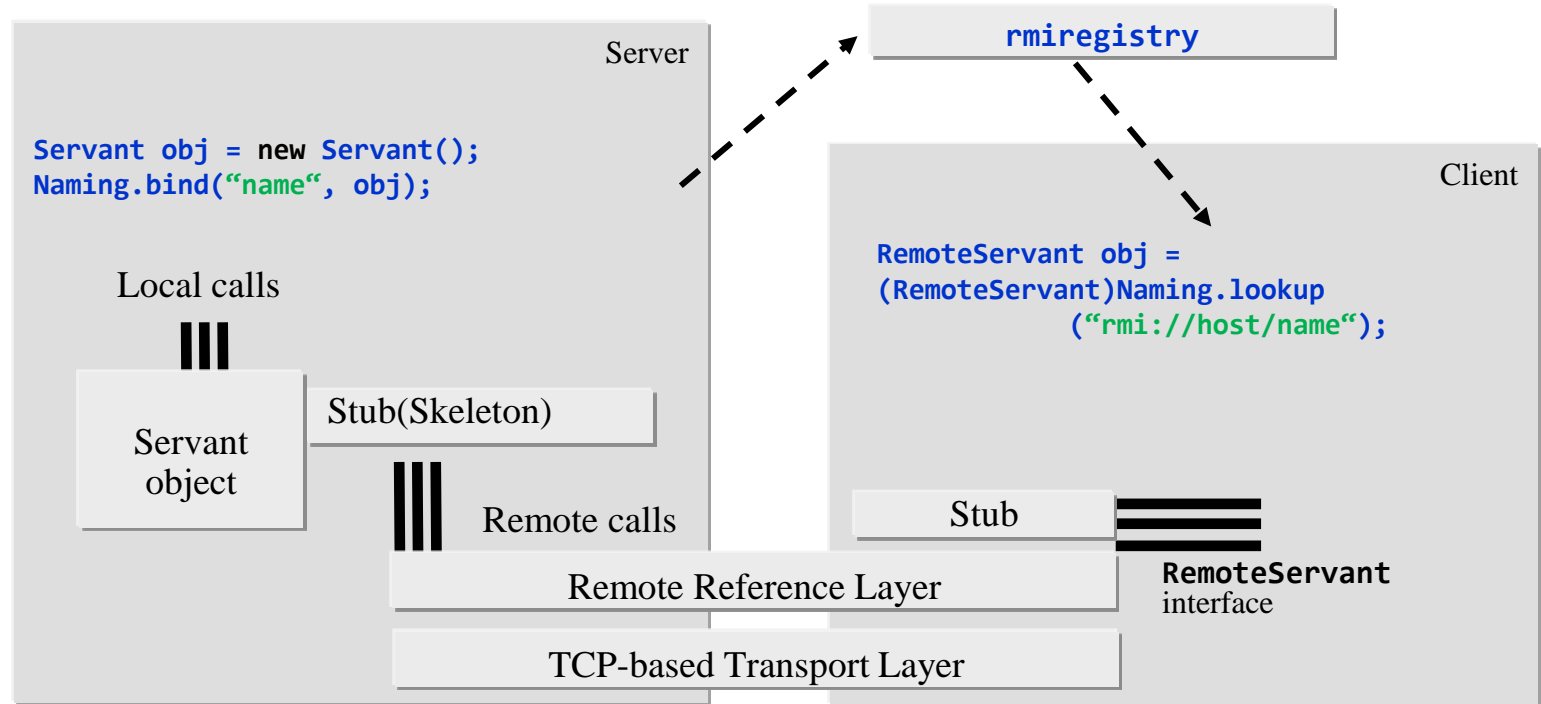
- A class that indicates a non-replicated remote object.



# Developing and Executing a Distributed Application with Java RMI

1. Define a remote interface(s) that extends **java.rmi.Remote**.
2. Develop a class (a.k.a. servant class) that implements the interface.
3. Develop a server class that provides a container for servants, i.e. creates the servants and registers them at the Naming Service.
4. Develop a client class that gets a reference to a remote object(s) and calls its remote methods.
5. Compile all classes and interfaces using **javac**.
6. Start the Naming service **rmiregistry**
7. Start the server on a server host, and run the client on a client host.

# Architecture of a Client-Server Application with Java RMI





# Summary

Implementations of remote invocations: procedures, methods, messages to processes,  
have fundamental problems that needs to be solved.

Try to see similarities between different implementations.

When they differ, is it fundamentally different or just implementation details.