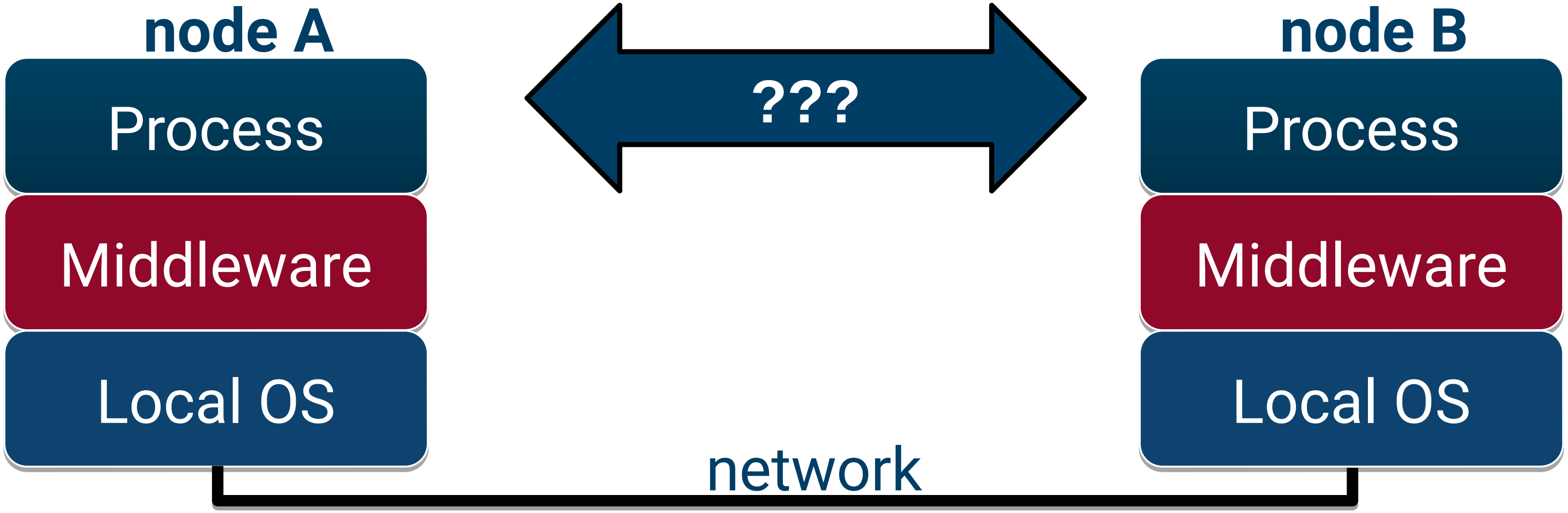
Distributed Systems

José Oramas

Middleware & Communication

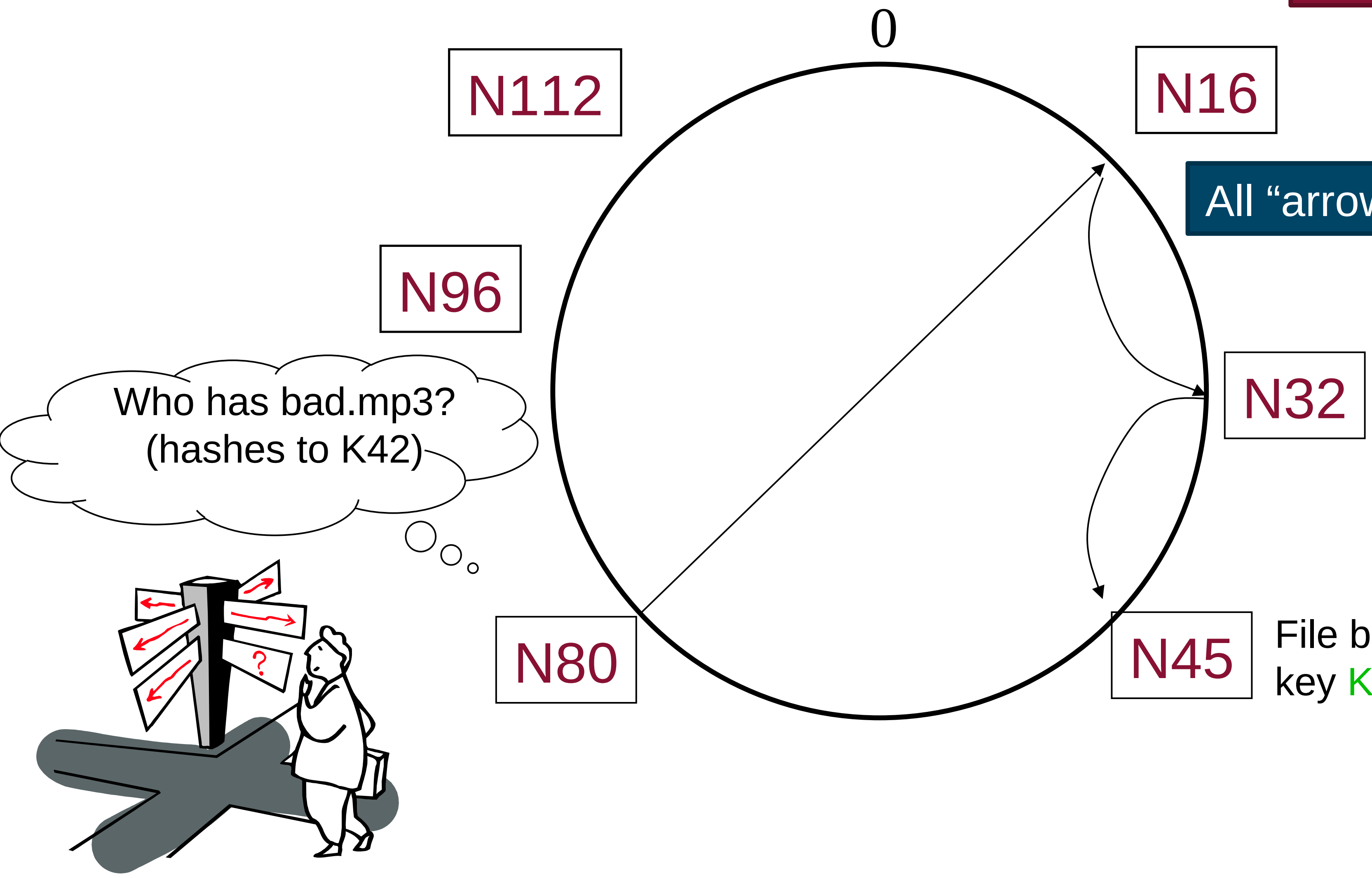
What we will see today



Middleware: the plumbing of distributed systems

1. It’s mostly invisible
2. It provides a standard way of doing things
3. It ties together parts of complex systems
4. It lets you focus on the core task

Search in *Chord*

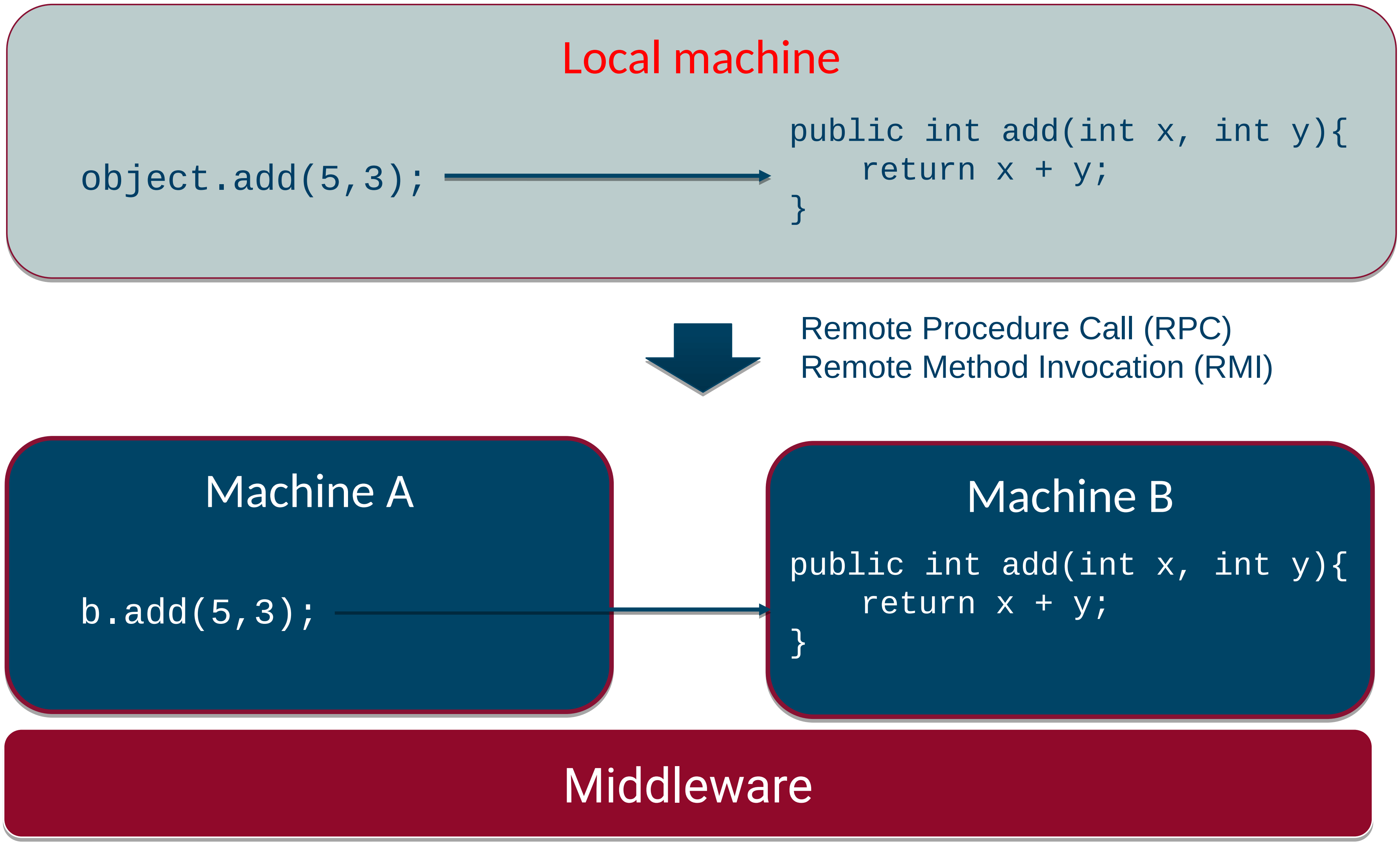


What are “*RPCs*”?

All “arrows” are RPCs

File bad.mp3 with
key K42 stored here

What we are going to see today…



Local machine

```
object.add(5,3);
```

```
public int add(int x, int y){
    return x + y;
}
```

Remote Procedure Call (RPC)
Remote Method Invocation (RMI)

Machine A

```
b.add(5,3);
```

Machine B

```
public int add(int x, int y){
    return x + y;
}
```

Middleware

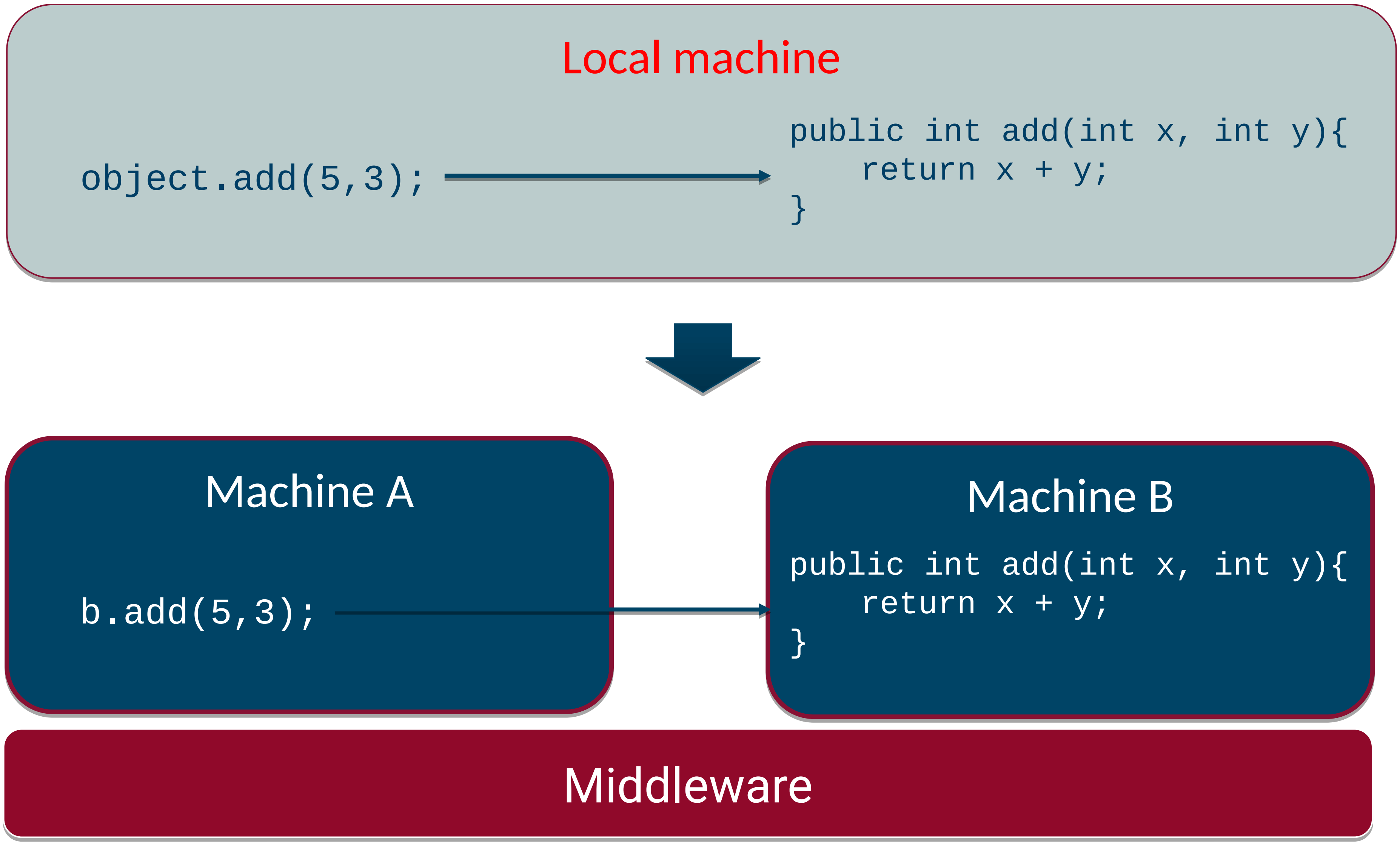
1 How to achieve basic communication

2 Why middleware?

3 The middleware principle

4 Middleware architecture

Let’s simplify things…



Local machine

```
object.add(5,3);
```

```
public int add(int x, int y){
    return x + y;
}
```

Machine A

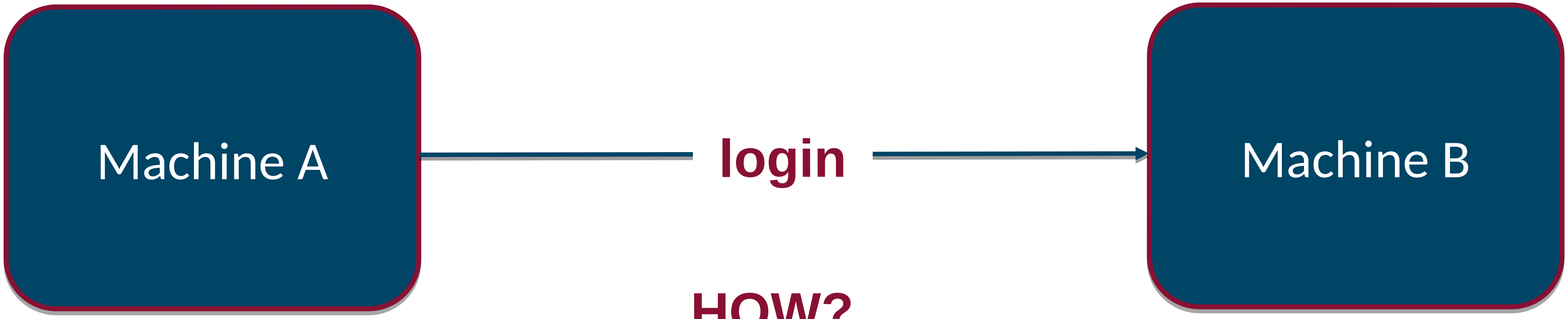
```
b.add(5,3);
```

Machine B

```
public int add(int x, int y){
    return x + y;
}
```

Middleware

First challenge:
Establishing basic communication



Machine A

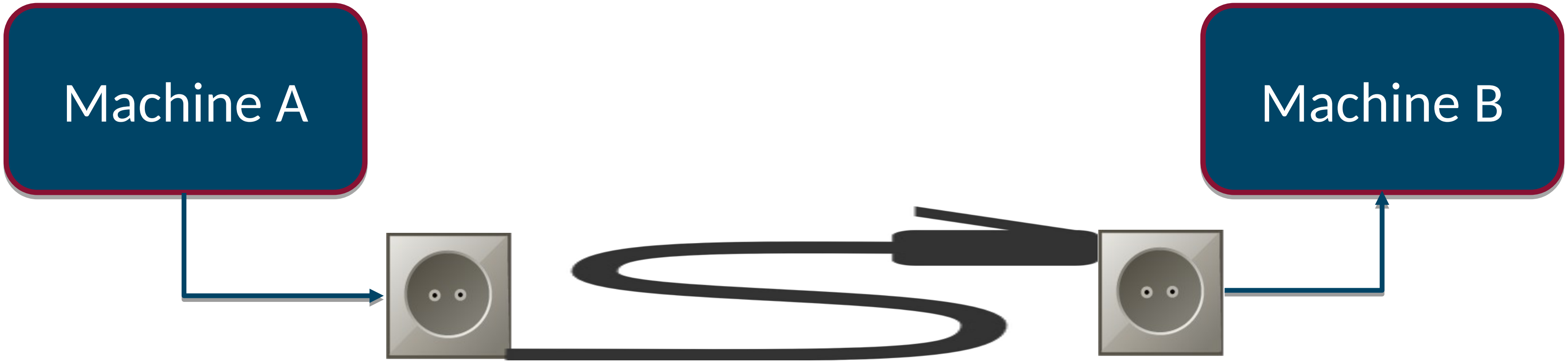
login

Machine B

HOW?

sockets

Abstraction for enabling inter-process communication
See it as a driver to put something “on the wire”
API Provided by all operating systems



Machine A

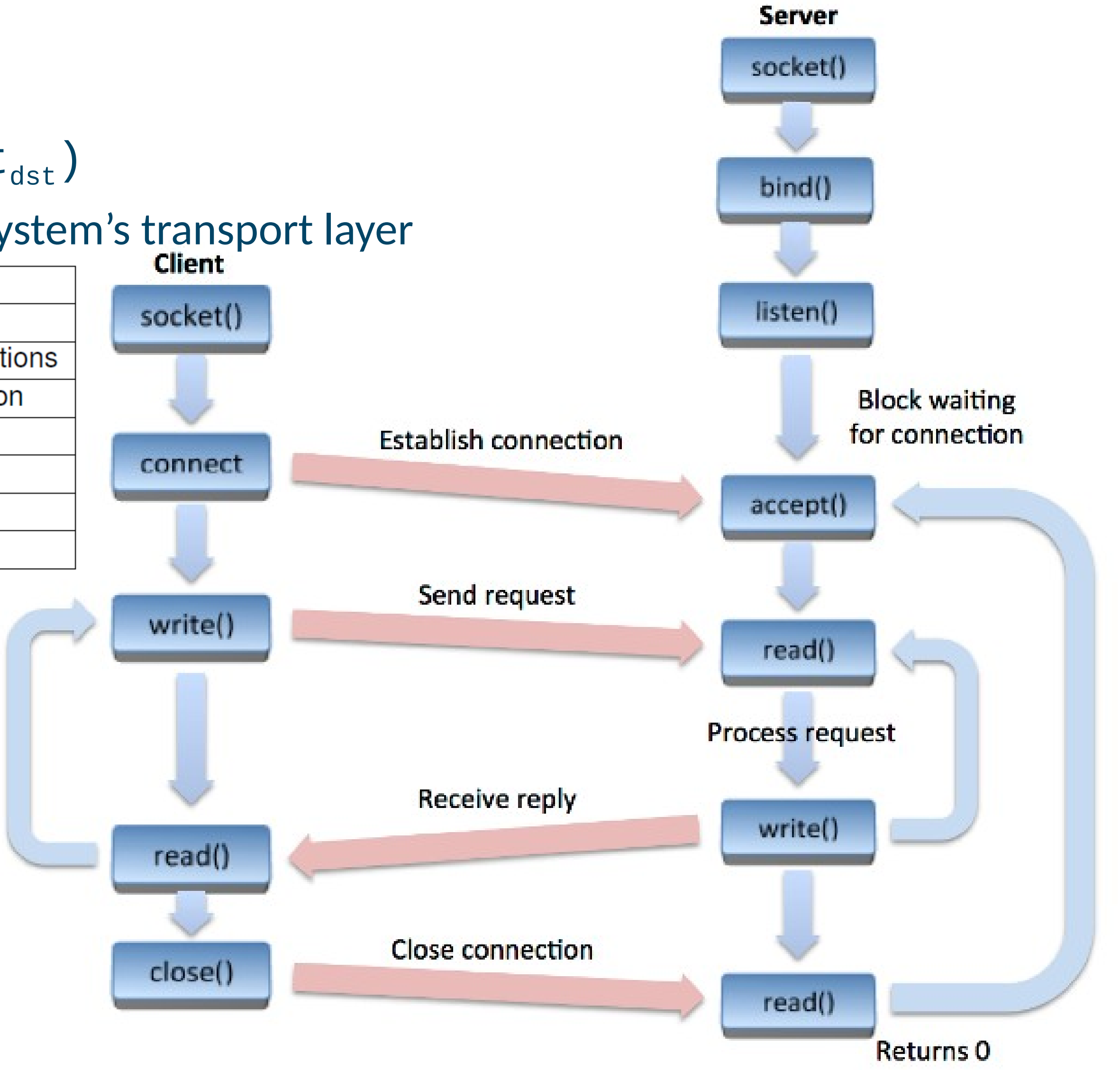
Machine B

Sockets

Berkeley sockets (Berkeley UNIX ’70)

- Socket pairs described by a 4-tuple
  $(\mathtt{ip}_{\mathtt{src}},\ \mathtt{port}_{\mathtt{src}},\ \mathtt{ip}_{\mathtt{dst}},\ \mathtt{port}_{\mathtt{dst}})$
- Standardized interface for using a system’s transport layer

SOCKET	Create a new communication endpoint
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept *N* connections
ACCEPT	Block until request to establish a connection
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
CLOSE	Release the connection



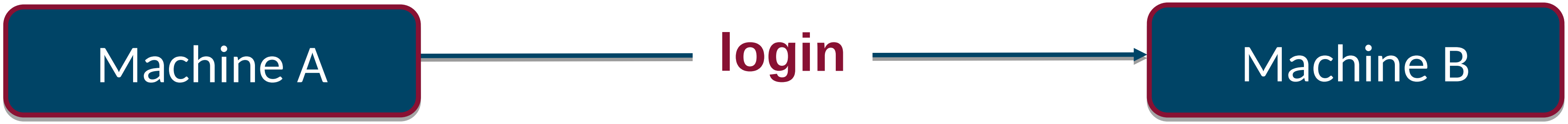
Socket programming in Java: the client

```java
package ds.sockets;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class UpperClientSocket {

      public static void main(String[] args){

            String ip = "localhost";
            int port = 4444;

            try {
                  Socket client = new Socket(ip,port);

                  //Create input and output stream
                  BufferedReader in = new BufferedReader(new InputStreamReader( client.getInputStream() ));
                  PrintWriter out = new PrintWriter(client.getOutputStream(),true);

                  String request = "Please convert this to uppercase";
                  out.println(request);   // <- request gets pushed to the server

                  System.out.println("Client requested: "+request);
                  System.out.println("Server responded: "+in.readLine());

                  client.close();
                  System.out.println("Client socket closed...");

            } catch (UnknownHostException e) {
                  System.err.println("Unknown host");

            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Socket programming in Java: the server

```java
package ds.sockets;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class UpperServerSocket {

      public static void main(String[] args){
            int portnumber = 4444;
            try {
                  //Setup a server socket
                  ServerSocket server = new ServerSocket(portnumber);
                  System.out.println("Accepting connections...");

                  //Wait for a connection, when one arrives, a client socket is returned
                  Socket client = server.accept();
                  System.out.println("Incoming client connection");

                  //Get data from client
                  BufferedReader input = new BufferedReader(new InputStreamReader( client.getInputStream() ));
                  PrintWriter out = new PrintWriter(client.getOutputStream(),true);

                  String line = input.readLine();
                  out.println(line.toUpperCase());
                  server.close();
                  System.out.println("Server socket closed...");

            } catch (IOException e) {
                  System.err.println("Could not listen on port "+portnumber);
                  e.printStackTrace();
            }

      }

}
```

That’s it? We are done?

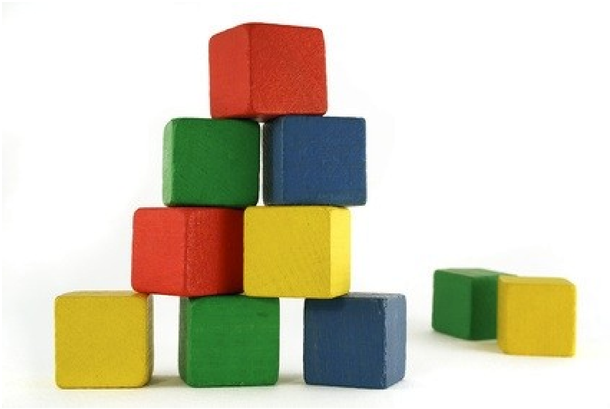


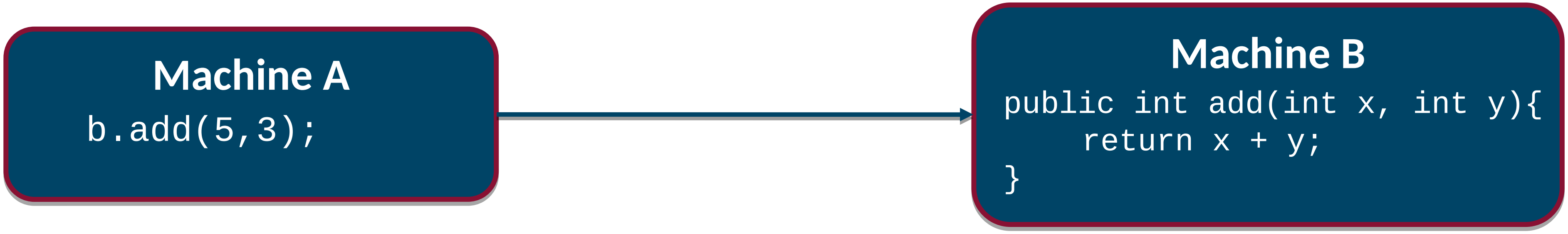
Machine A

login

Machine B

- You now know how to establish basic communication
- Use this as the building block to establish remote procedure calls





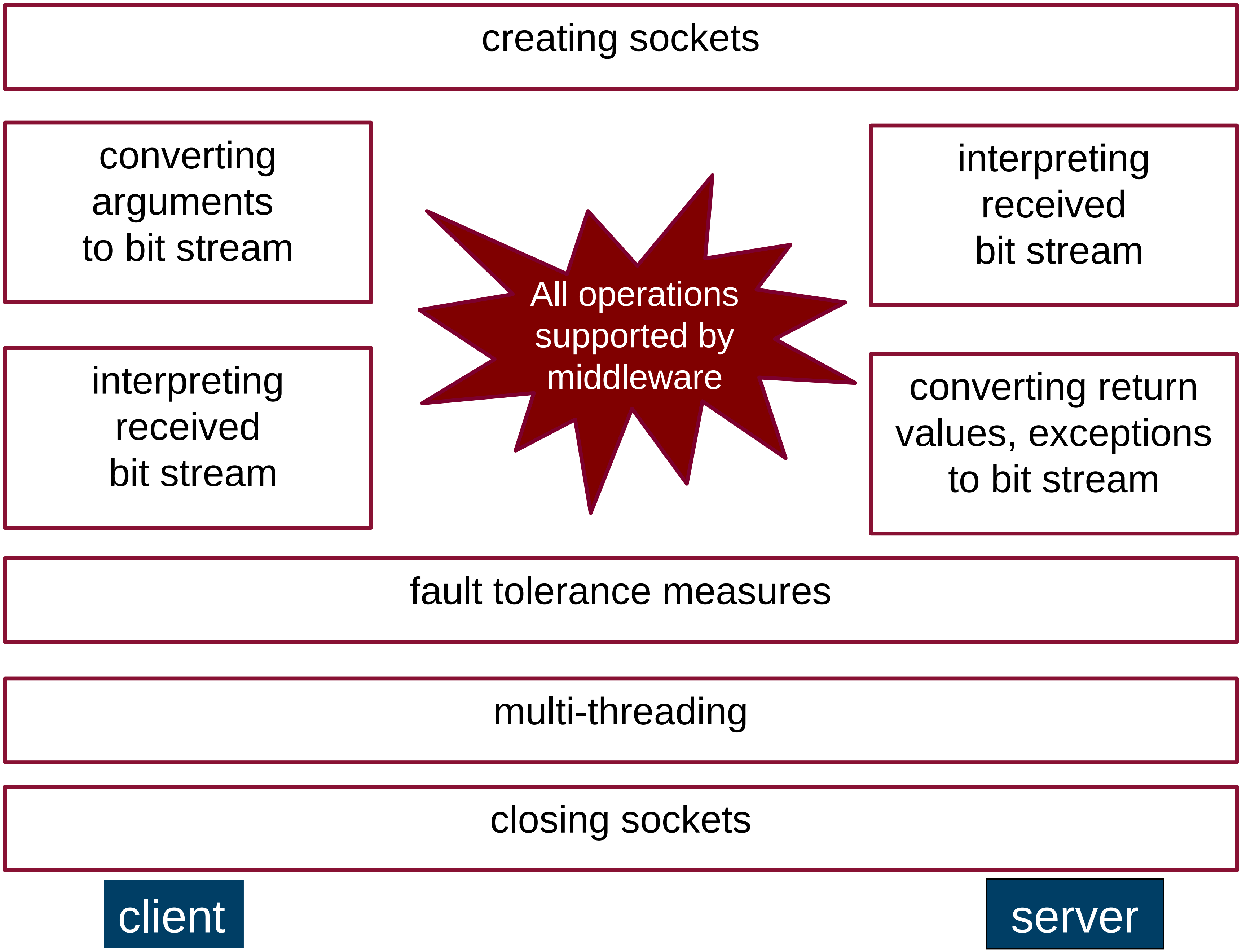
Machine A

```
b.add(5,3);
```

Machine B

```
public int add(int x, int y){
    return x + y;
}
```

Remote invocation: requires …



creating sockets

converting arguments to bit stream

All operations supported by middleware

interpreting received bit stream

interpreting received bit stream

converting return values, exceptions to bit stream

fault tolerance measures

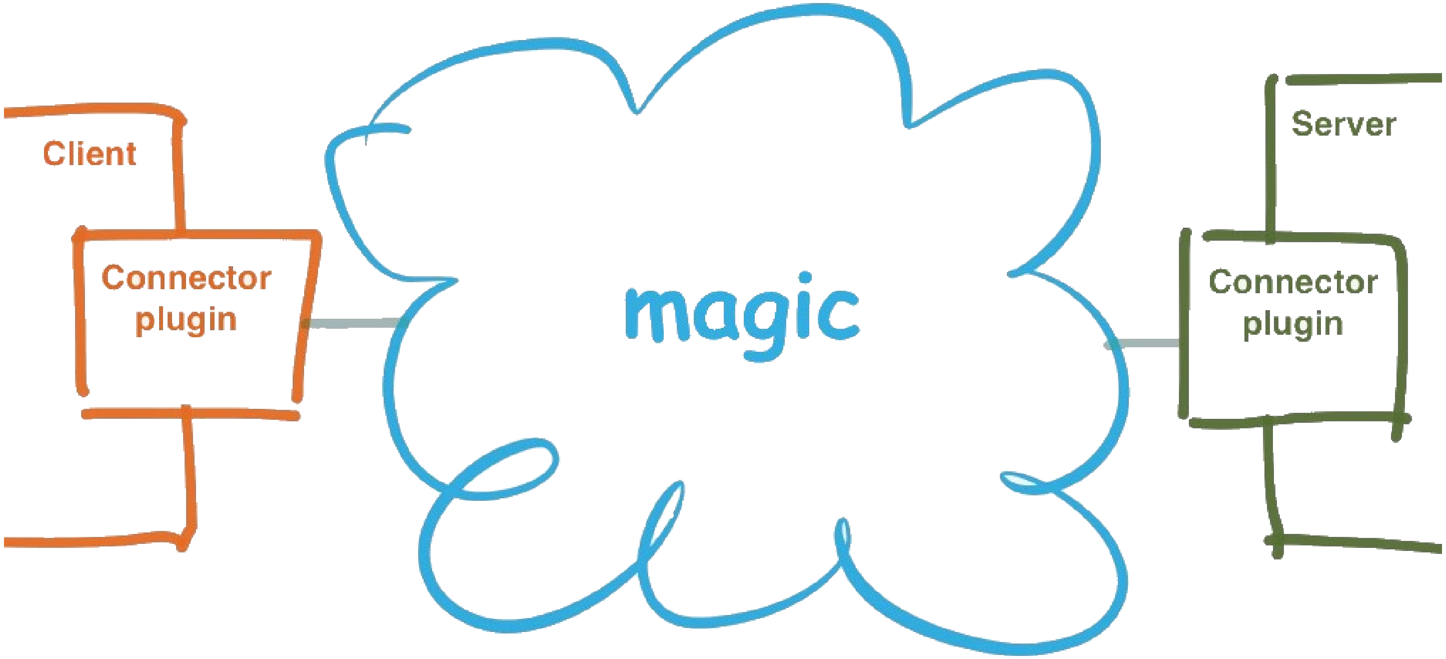
multi-threading

closing sockets

client

server

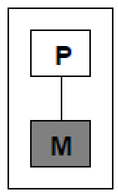
Why do we need middleware?



Hardware and OS architectures

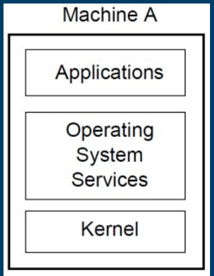
Uniprocessor

Hardware architecture



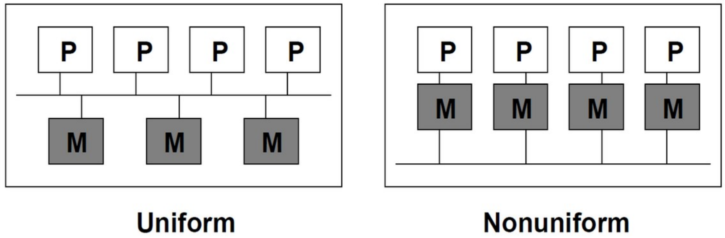
- Single processor
- Direct memory access

OS architecture



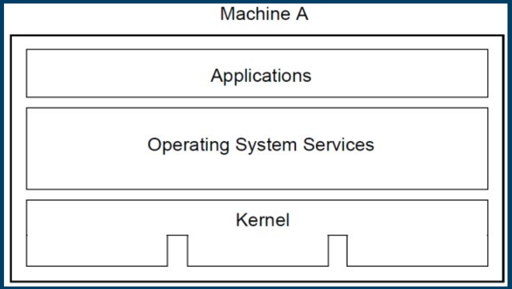
Multiprocessor

Hardware architecture



- Multiple processors
- Direct memory access
- Uniform or non-uniform (NUMA) memory access

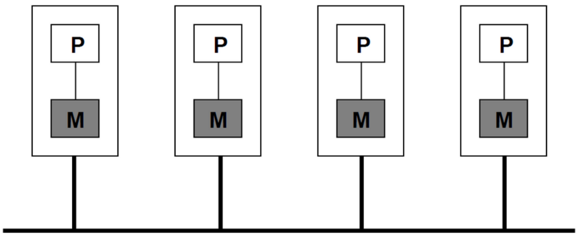
OS architecture



- Multi-CPU kernel design
- Transparent # cores
- Single System Image (SSI)
- Comm. primitives as in uniproc. OS

Multicomputer

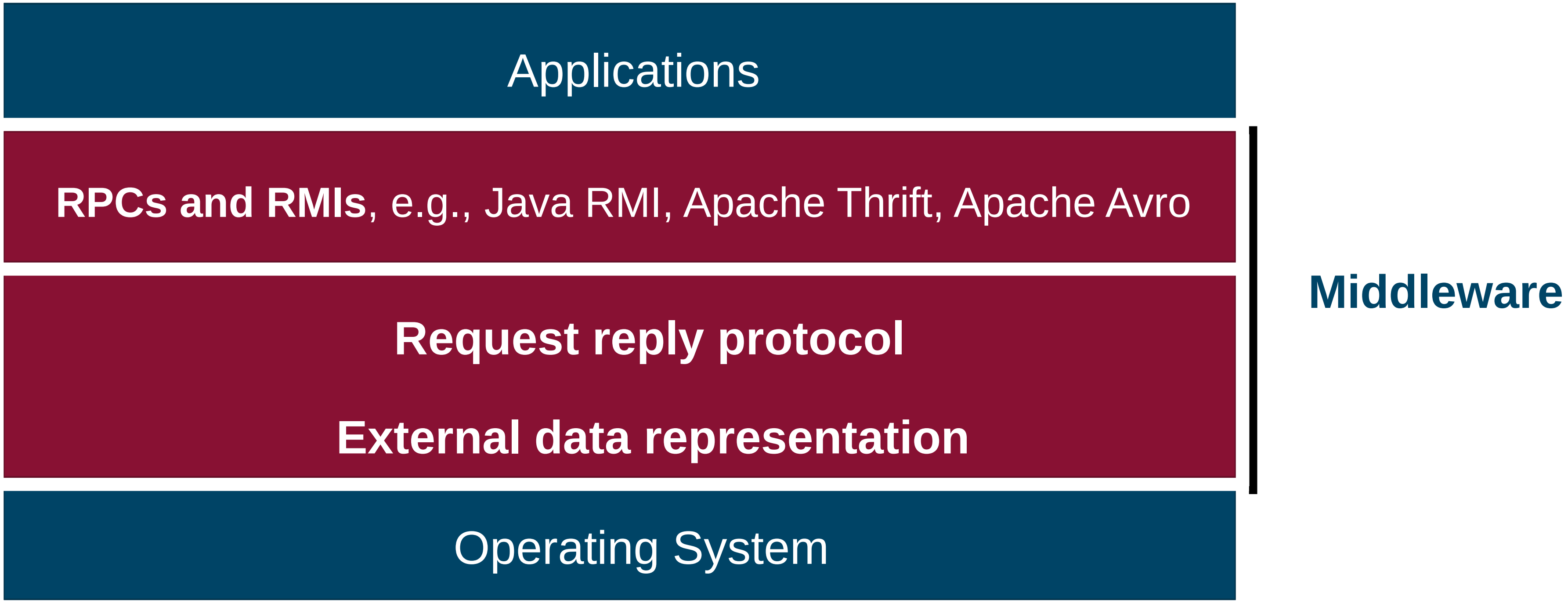
Hardware architecture



- Multiple computers
- No direct memory access
- Network
- Homogeneous or heterogeneous

OS architecture

Middleware Layers



Applications

RPCs and RMIs, e.g., Java RMI, Apache Thrift, Apache Avro

Request reply protocol

External data representation

Operating System

Middleware

Method invocation for local Objects

Within one process's address space

Object

- consists of a set of data and a set of methods.
- E.g., C++/Java object

Object reference

- an identifier via which objects can be accessed.
- i.e., a pointer (C++)

Interface

- Signatures of methods
    - Types of arguments, return values, exceptions
- No implementation
- E.g., hash table:
    - insert(key, value)
    - value = get(key)
    -

Remote Objects

May cross multiple process's address spaces

Remote objects

- Objects that can receive remote invocations.

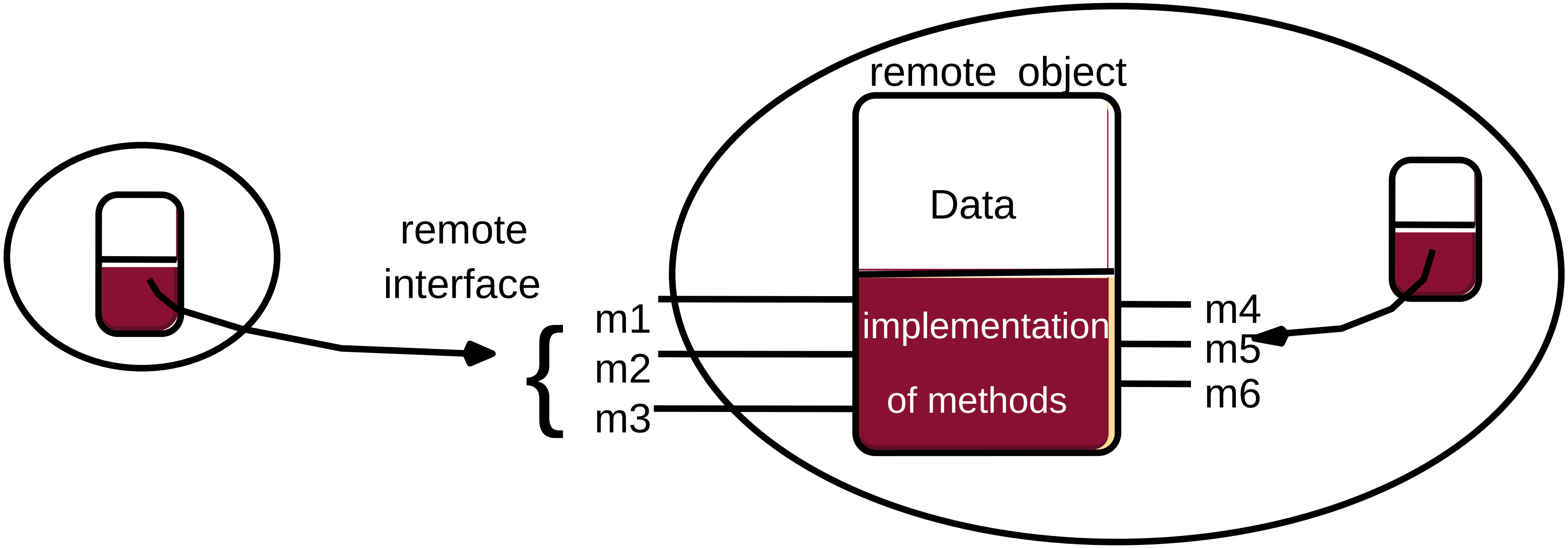
Remote object reference

- An identifier that can be used globally throughout a distributed system to refer to a particular unique remote object.

Remote interface

- Every remote object has a remote interface that specifies which of its methods can be invoked remotely
- Interface Definition Language

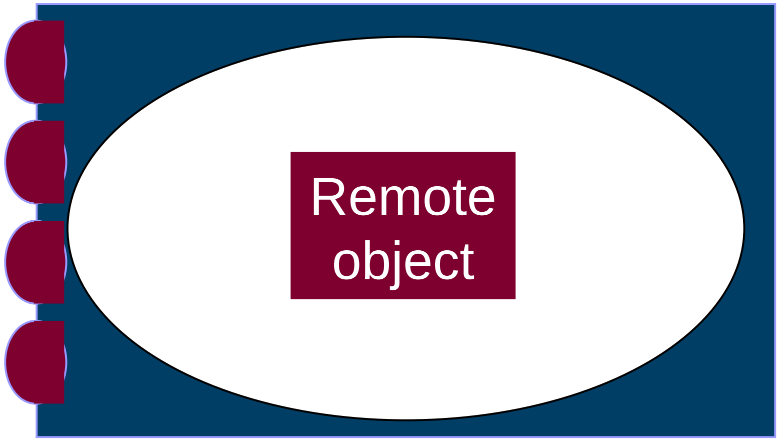
A Remote Object and Its Remote Interface



Remote object references

Purpose : unique ID for objects in distributed system

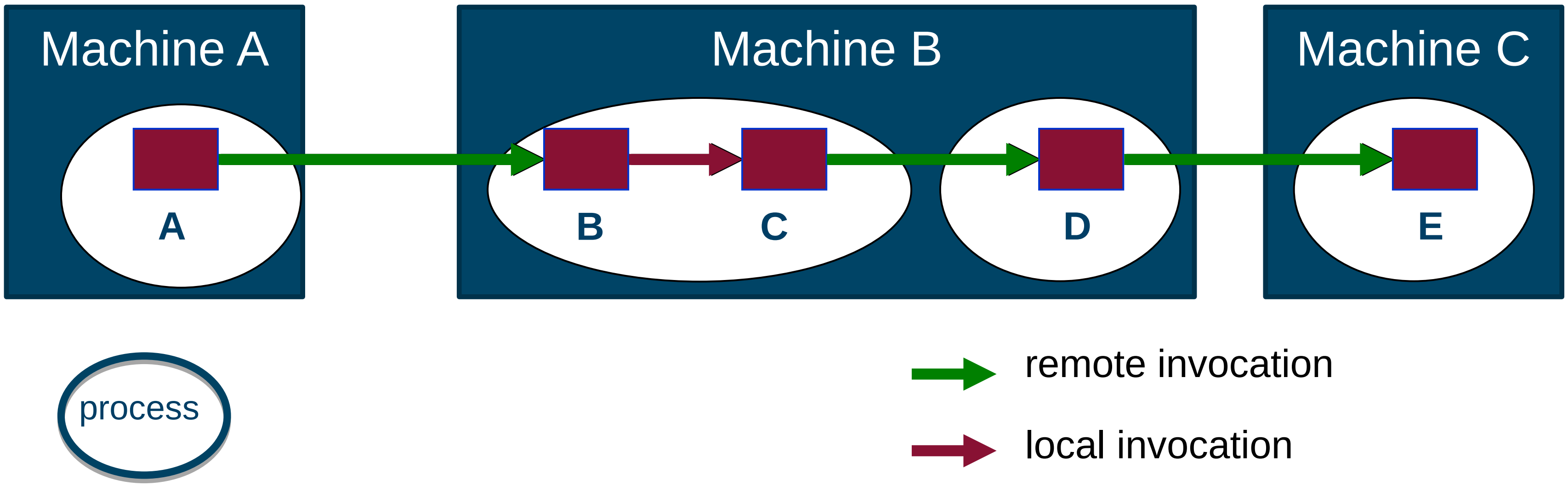
- uniqueness over time
- uniqueness over space



Host : Internet Address
Process : Port number + process creation time
Object : object number
Object type : remote interface

IP-address	Port number	Creation time	Object number	Interface
32	32	32	32	?

Local and remote invocations



Machine A

Machine B

Machine C

A

B

C

D

E

process

remote invocation

local invocation

Local invocation

Use Object Reference
Any public method
Invocation: exactly once

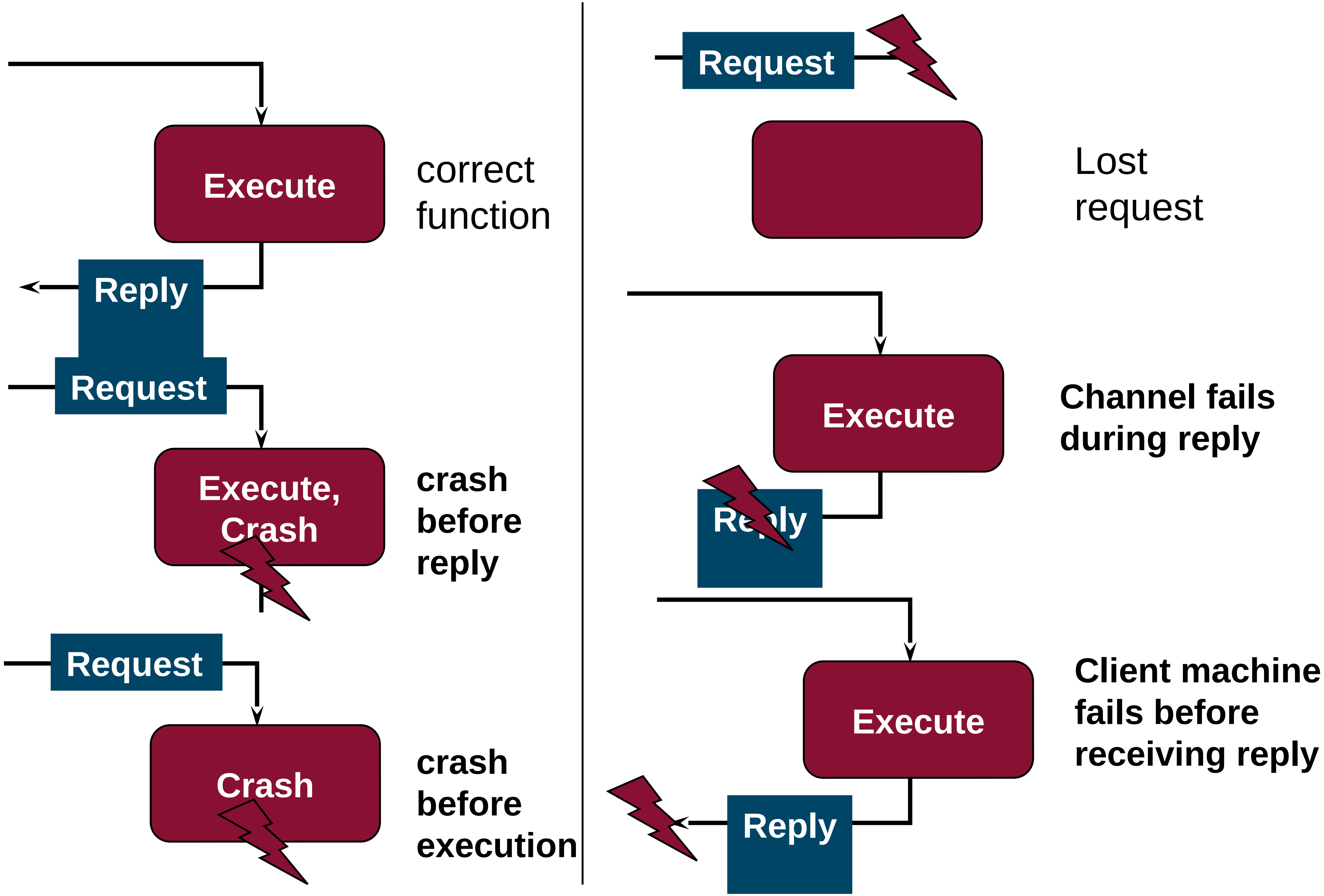
Remote invocation

Use Remote Object Reference
Limited access, remote interface
Invocation: ???

Break

See you in 15 mins.

Failure Modes of RMI/RPC



Execute

correct function

Reply

Request

Execute, Crash

crash before reply

Request

Crash

crash before execution

Request

Lost request

Execute

Channel fails during reply

Reply

Execute

Client machine fails before receiving reply

Reply

(and if request is received more than once?)

Fault tolerance

Techniques:

1. Retry-request message
2. Duplicate request filtering
3. Retransmission of results
    - 3.1 Re-execute call
    - 3.2 History table of results - Retransmit reply

determines

Invocation semantics:

1. Maybe
2. At-least-once
3. At-most-once

Invocation Semantics

What guarantees are given on the number of **executions** of remote method invocations?

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
			Maybe
			At-least-once
			At-most-once

Invocation Semantics

What guarantees are given on the number of **executions** of remote method invocations?

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

Role of the middleware…

Hide all these underlying complexity: provide transparency

Make invocation syntax similar to local invocation, hiding:

- Locate/contact remote object
- Marshalling: converting arguments to bitstream
- Fault tolerance measures
- Communication details (sockets)

But ...

Not everything should be hidden ...

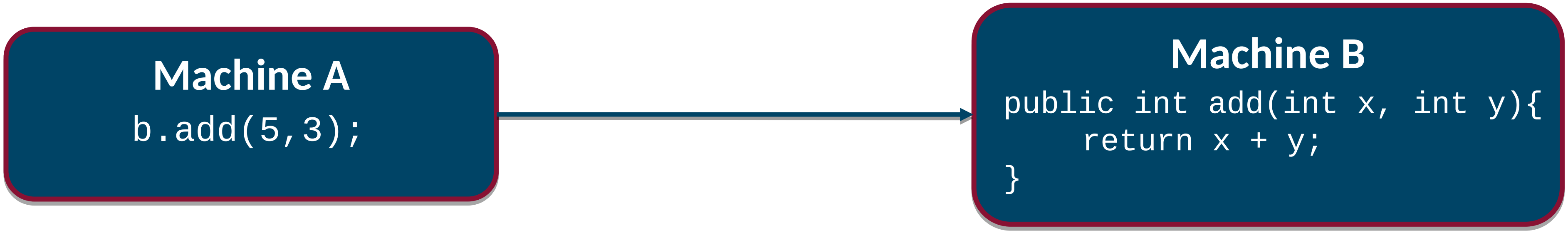
programmer should know object is remote
(network latency, potential invocation semantics, additional failures) **!**

Typical approach

- same invocation syntax (but catch exceptions ...)
- Use of remote interface reflects remoteness

Middleware Architecture

Proxy and Skeleton in Remote Method Invocation

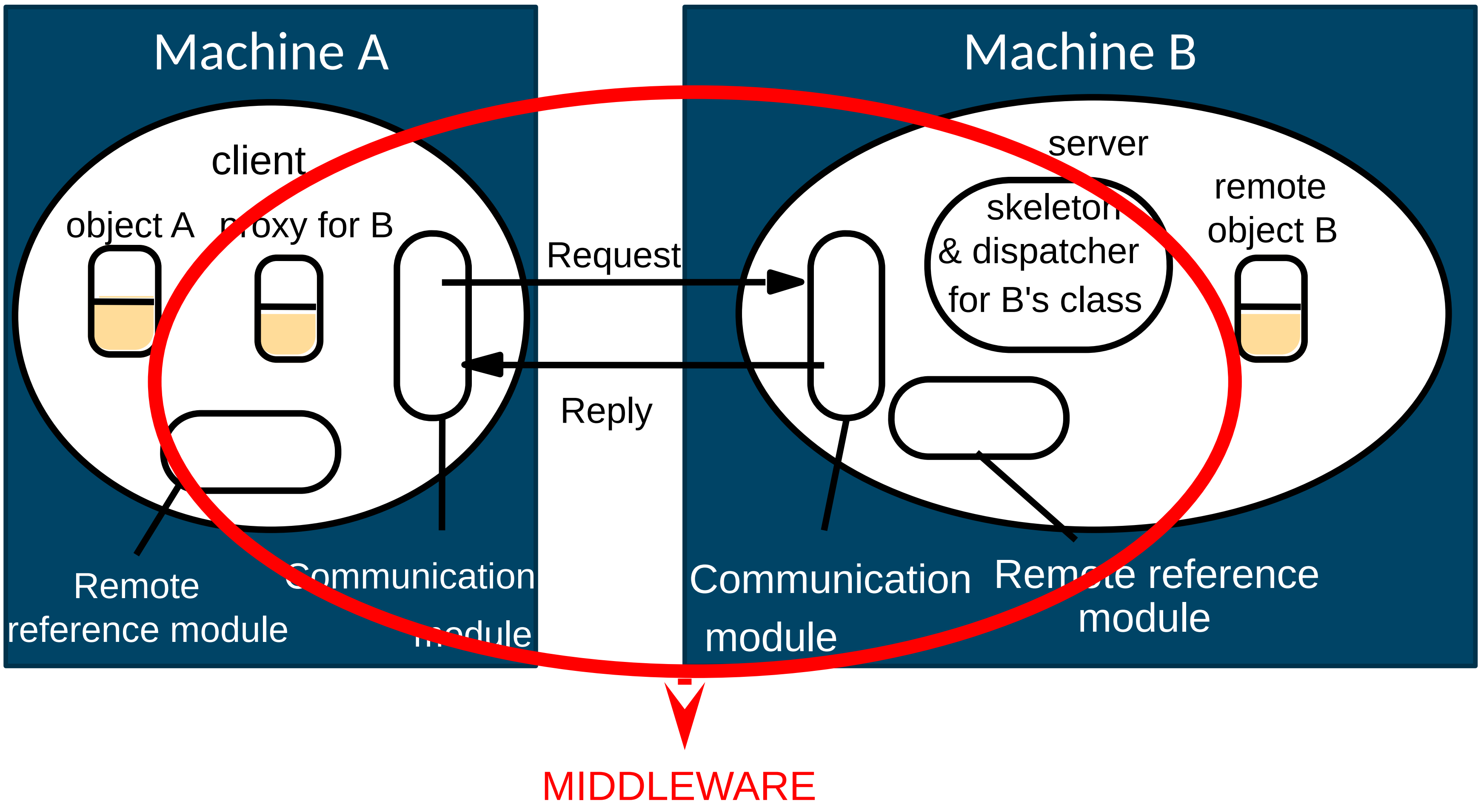


Machine A

```
b.add(5,3);
```

Machine B

```
public int add(int x, int y){
    return x + y;
}
```



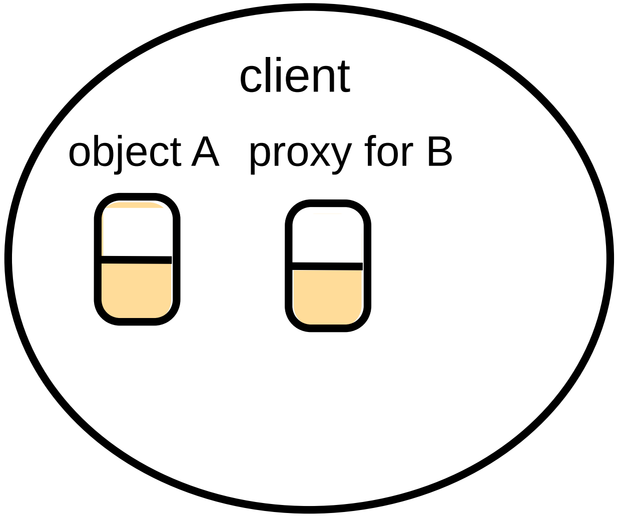
Proxy

Is responsible for making RMI transparent to clients by behaving like a local object to the invoker.

The proxy implements (Java term, not literally) the methods in the interface of the remote object that it represents. But,…

Instead of executing an invocation, the proxy forwards it to a remote object

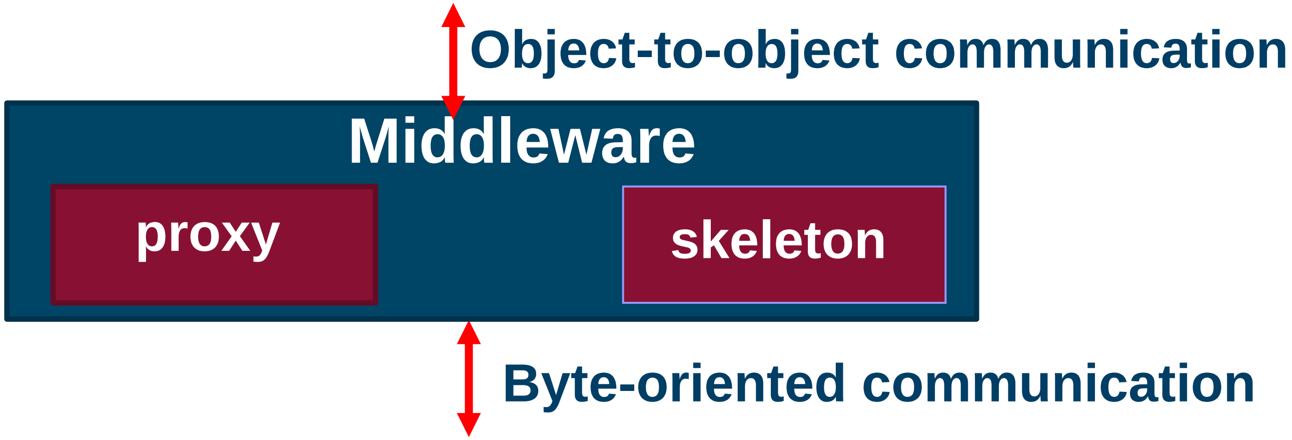
1. Marshals a request message
   - Target object reference
   - Method ID
   - Argument values
2. Sends request message
3. Unmarshals reply and returns to invoker



Skeleton = reverse of proxy at server side

Marshalling

= transform object in memory to bitstream

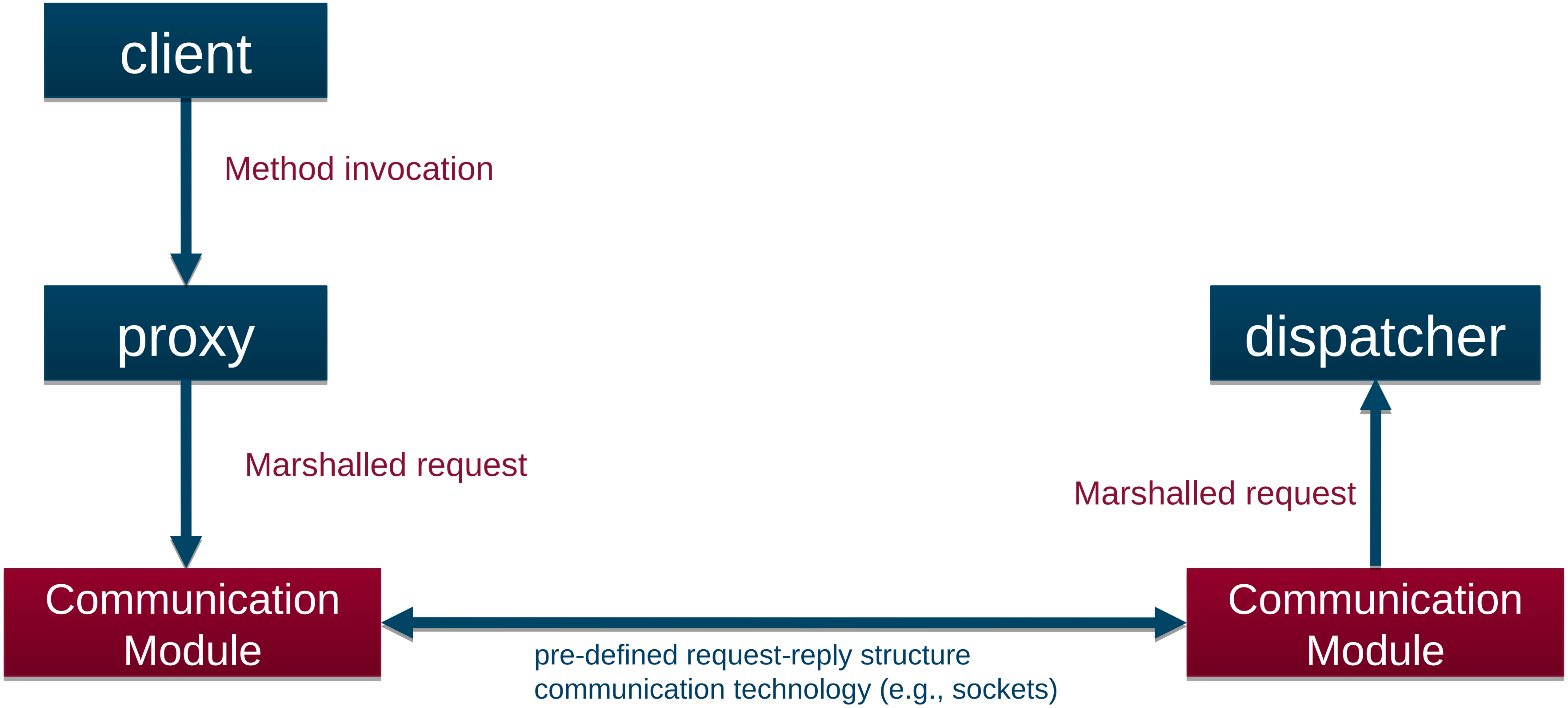


Options

- **marshalling using external standard**
  = External Data Representation
  e.g. CORBA's common data representation (CDR)
  Java serialization
  Google Protocol Buffer
- **marshalling using sender standard AND send this standard along**

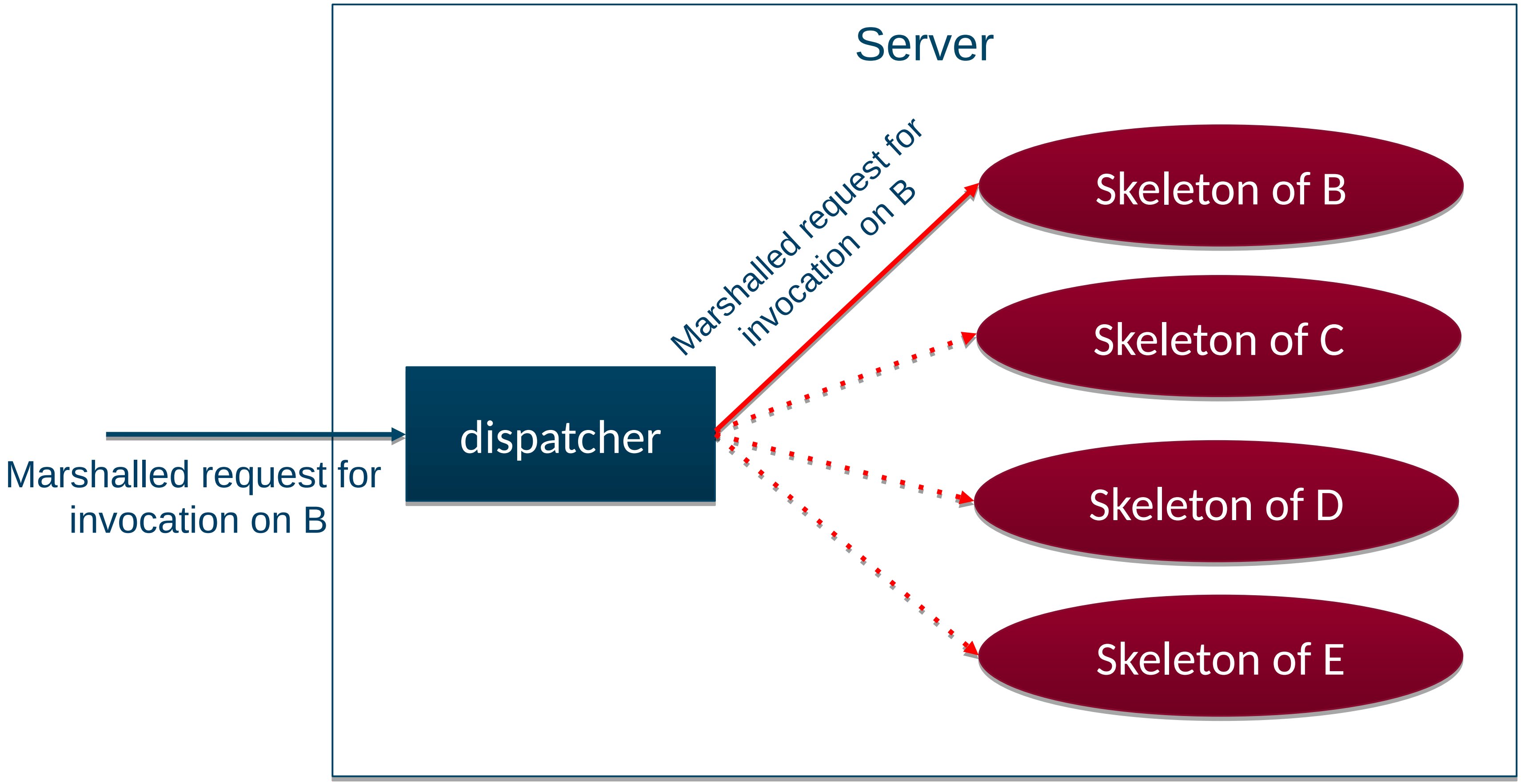
Communication Module

- Implements the invocation semantics
- Runs a request-reply protocol
- Ensures communication between proxy & server-side



Dispatcher

- The dispatcher receives all request messages from the communication module.
- For the request message, it uses the method id to select the appropriate method in the appropriate skeleton, passing on the request message.

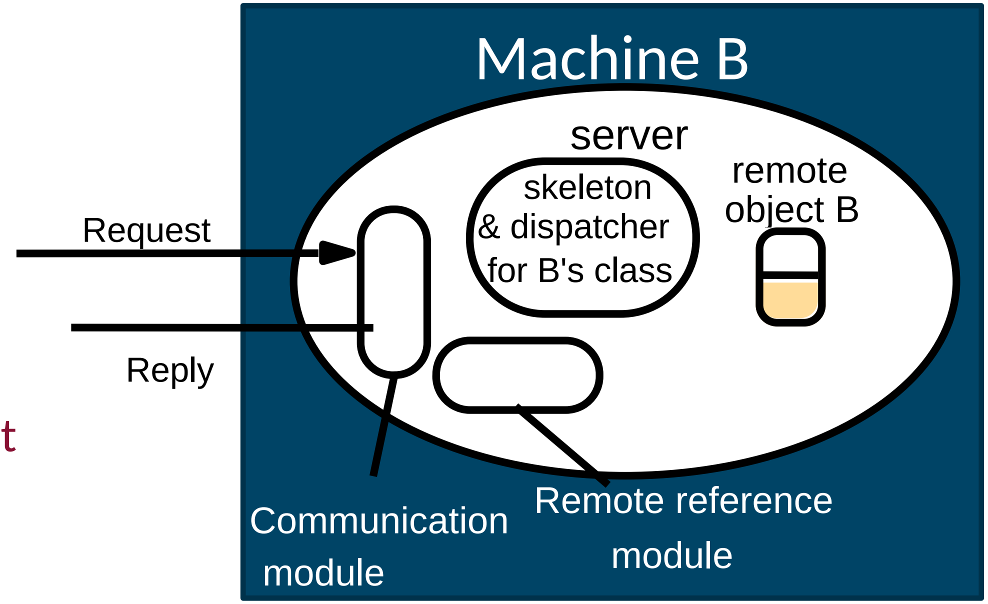

Server
Marshalled request for invocation on B
dispatcher
Marshalled request for invocation on B
Skeleton of B
Skeleton of C
Skeleton of D
Skeleton of E

Skeleton

Is responsible for making RMI transparent to **servers** by behaving like a **local invoker** to the **object**.

The **skeleton** …

1. Accepts a request message
2. Unmarshals the request
   - Target object reference
   - Method ID
   - Argument values
3. Invokes the method on the server object
4. Marshals reply and returns to proxy

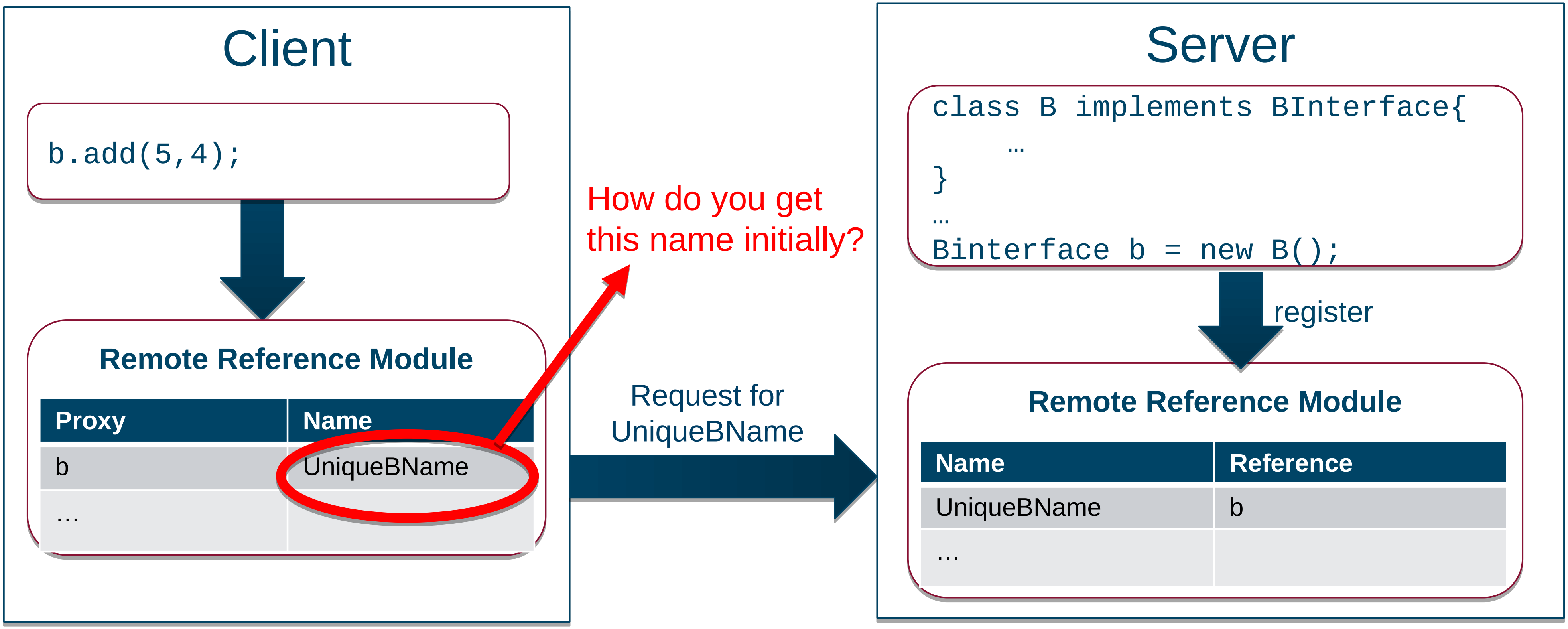


Remote Reference Module

Is responsible for translating between local and remote object references and for creating remote object references.

Has a remote object table

- **Server-side:** An entry for each remote object held by any process. E.g., B at host B.
- **Client-side:** An entry for each local proxy. E.g., proxy-B at host A



Client

```
b.add(5,4);
```

Remote Reference Module

Proxy	Name
b	UniqueBName
…	

How do you get this name initially?

Request for UniqueBName

Server

```
class B implements BInterface{
    …
}
…
Binterface b = new B();
```

register

Remote Reference Module

Name	Reference
UniqueBName	b
…	

Binding Service

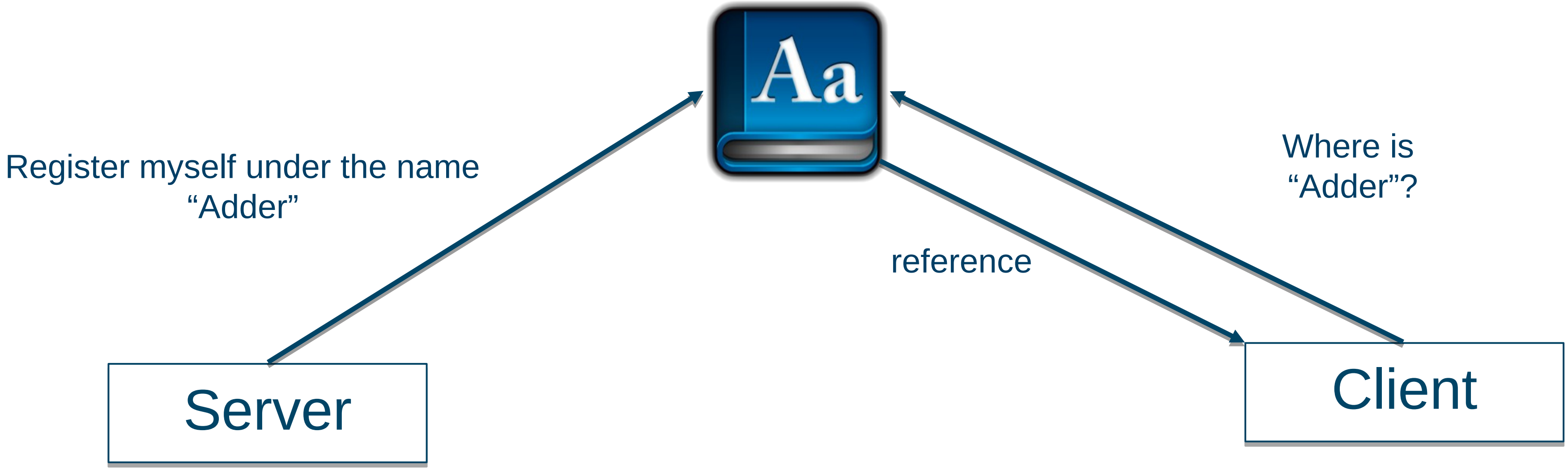
How do you get the initial reference?

Option 1) Contact the server

- Go to the server and ask for the name
- What if multiple processes are running of the same class?

Option 2) Contact a Binding Service

- Dictionary service for Middleware
- Or DNS system for RPC



Register myself under the name "Adder"

Where is "Adder"?

reference

Server

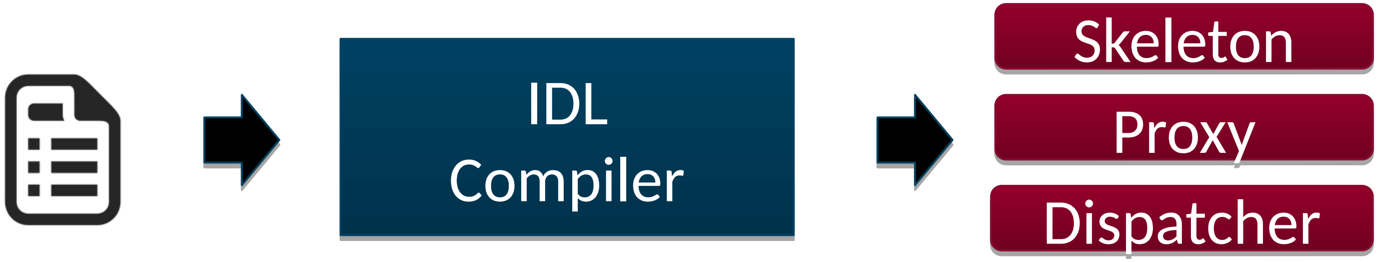
Client

Do I need to write all this code?

NO!

Generated by middleware

… based on IDL



IDL Compiler

Skeleton

Proxy

Dispatcher

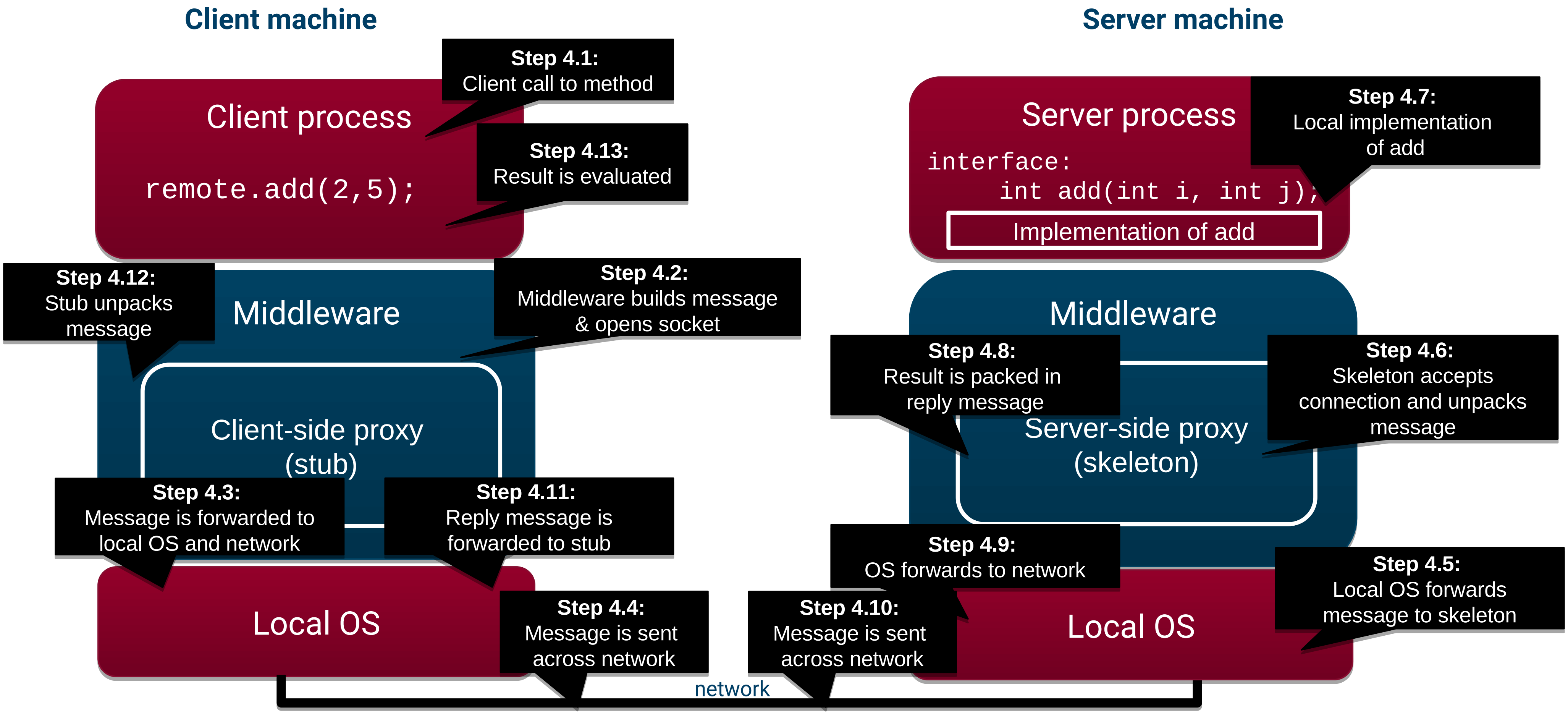
Putting it all together

Example: remote method that adds two numbers

STEP 1: Define remote interface, using Interface Definition Language

STEP 2: Implement remote methods

STEP 3: Middleware automatically generates proxies



```
remote.add(2,5);
```

```
interface:
     int add(int i, int j);
```

Step 4.1: Client call to method

Step 4.2: Middleware builds message & opens socket

Step 4.3: Message is forwarded to local OS and network

Step 4.4: Message is sent across network

Step 4.5: Local OS forwards message to skeleton

Step 4.6: Skeleton accepts connection and unpacks message

Step 4.7: Local implementation of add

Step 4.8: Result is packed in reply message

Step 4.9: OS forwards to network

Step 4.10: Message is sent across network

Step 4.11: Reply message is forwarded to stub

Step 4.12: Stub unpacks message

Step 4.13: Result is evaluated

Questions ?

Middleware & Communication