Distributed Systems
Direct Communication
PART I

Wouter Joosen

DistriNet, KU Leuven

October 03, 2023






KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

Overview of chapters

- Introduction
- Coordination models and languages: direct communication
  - Ch 4: Inter-process communication (only a small part)
  - Ch 5: Remote invocation
  - Ch 8: Distributed objects and components

(Assumes knowledge of computer networks: Chapter 3-4)

note:
CHAPTER 3 – HELICOPTER VIEW

1. Intro (concepts, terms)
2. Types of Network (LAN, WAN etc).
3. Network Principles: packet transmission, data streaming, switching, protocols, routing, congestion control, internetworking
4. Internet Protocols: IP addressing IP protocol, IP routing, IPv6, Mobile IP, TCP and UDP
5. Case studies: Ethernet, WiFi and Bluetooth
6. Summary

All assumed to be studied before.

note:
CHAPTER 4 – HELICOPTER VIEW

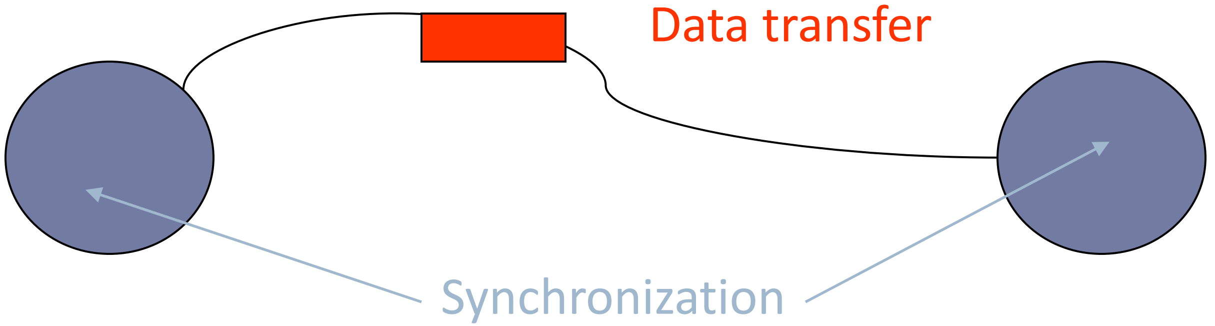
1. Intro
2. The API for the Internet protocols
   – Assumed to be studied before
3. External data representation and marshalling
   - Covered in this part (with chapter 5)

4. Multicast communication (skipped)
5. Network virtualization: Overlay networks (skipped)
6. Case study: MPI (skipped)
7. Summary

KATHOLIEKE UNIVERSITEIT LEUVEN DistriNet RESEARCH GROUP

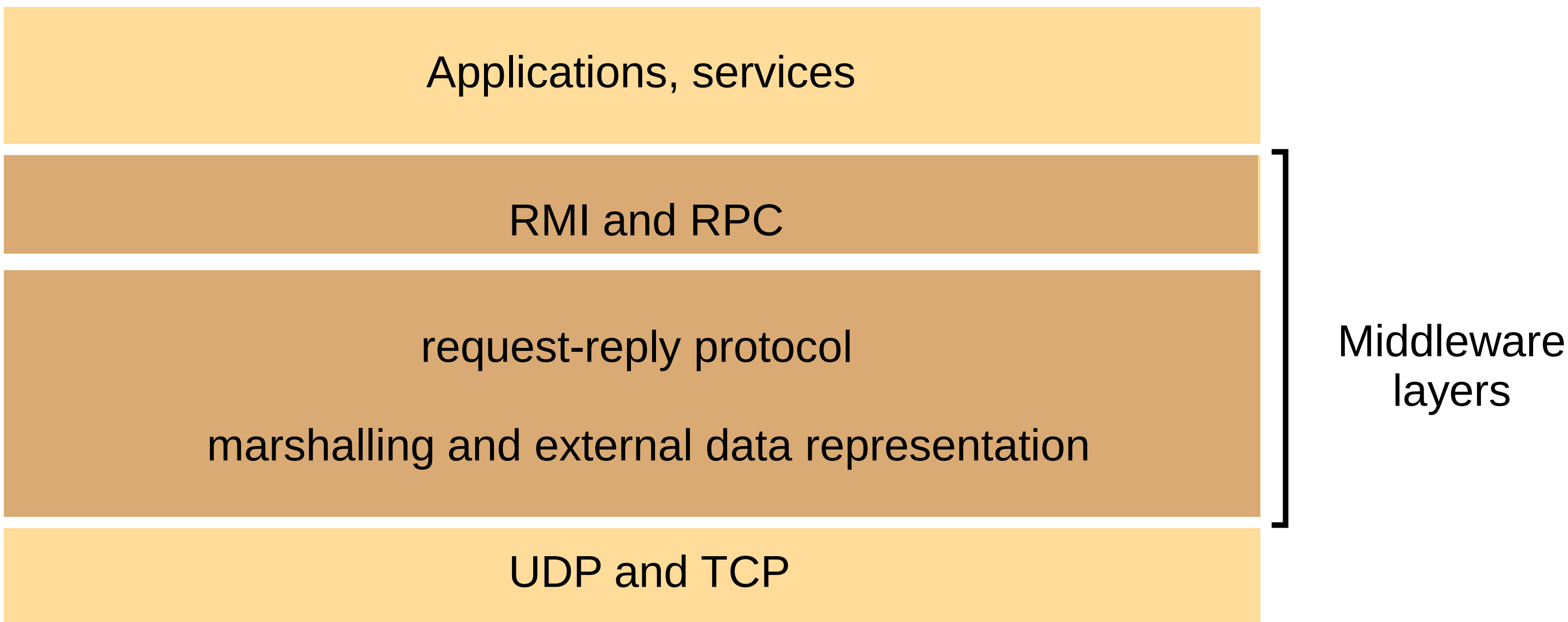
Introduction

- Communication
  - data representation
- Synchronization
  - how express cooperation?



Introduction

Distribution service in *__middleware__* *(subset of the larger middleware setting)*



Introduction

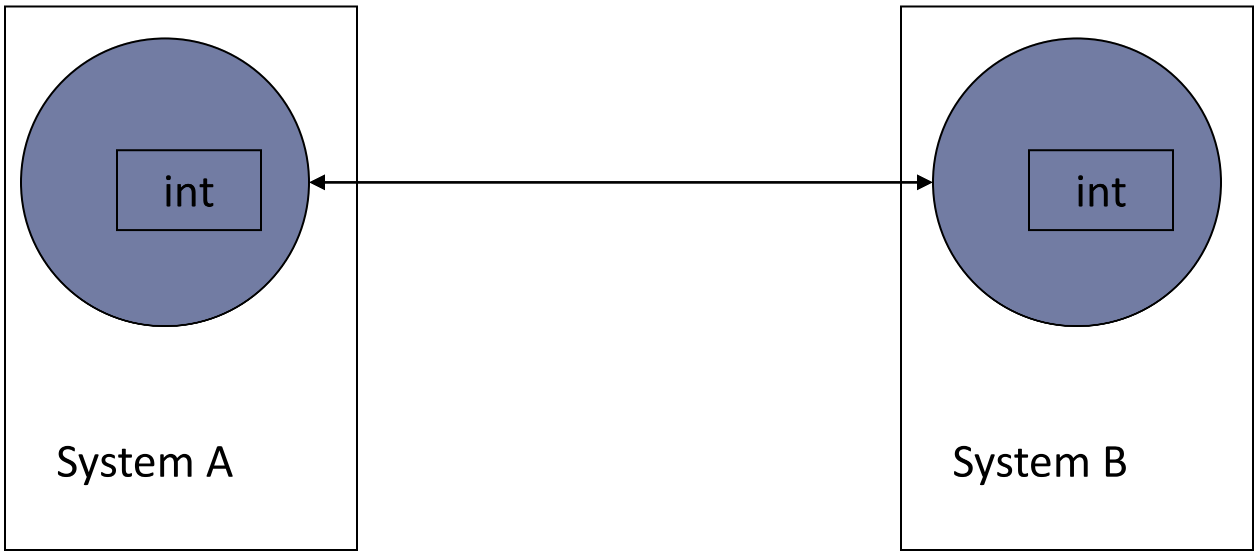
- Distribution service in middleware
  - shields developers of a distributed application from the complex distributed environment, e.g.
    - Low-level socket API
    - No transfer of structured data
    - ...
  - by offering programming abstraction layers on top of OS
    - Typical programming paradigms incorporated in distributed model (syntactically)
    - Abstraction of heterogeneity of systems

This lecture: overview

- Data representation
- Message passing
- Request-reply protocols
- Remote procedure calls

Data representation

- Problem



int = 2-complement, 32 bits

int = 1-complement, 40 bits

on the wire: int = ??

Data representation *(cont.)*

- Problem:
  - program data: typed, structured, object
  - message data: bit/byte stream
  - different representations of data in heterogeneous systems
- mapping to data items in messages:
  - flattened before transmission
  - rebuilt on arrival

Data representation *(cont.)*

- Conversion: different approaches
  - agreed form for transmission (implicit information)
    - e.g. int = 2-complement, 32 bits
    - both partners have full knowledge of transmitted data
    - e.g.: CORBA CDR, Sun XDR
  - full data description transmitted
    - *type, length, value* coding on the wire
    - interpretation at receiving site possible
    - e.g. ASN + BER, Java serialized form
  - Conversion to ASCII text
    - XML

Supporting
Sun NFS
(file system)

Data representation *(cont.)*

- Java serialized form
  - Handles: references to objects (within serialized form)
  - Primitive types: portable binary format

`Person p = new Person("Smith", "London", 1934);`

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

(The true serialized form contains additional type markers; h0 and h1 are handles)

Data representation *(cont.)*

- XML (e.g. used in Web Services for SOAP)
  - Markup language defined by World Wide Web consortium
  - Data items tagged with ‘markup’ strings
  - Users can define their own tags

```
<person id=‘123456789”>
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <…>
</person>
```

Data representation *(cont.)*

- Definitions:
  - marshalling = assembling a collection of data items into a form suitable for transmission
  - unmarshalling = disassembling a message on arrival to produce the equivalent collection of data items
- operations can be generated from specification
- In Java: serialization & deserialization

This chapter: overview

- Data representation
- Message passing
- Request reply protocols
- Remote procedure calls
- … Object request brokers (later)

Message passing

- Basic functionality
  - Procedure Send
    (p: PortId; m: Message);
  - Procedure Receive
    (p: PortId; VAR m: Message);

Message passing *(cont.)*

- **Semantics:**
  synchronous ⇔ asynchronous communication
  - synchronous = blocking
    - send: wait for corresponding receive
    - receive: wait for message arrival
  - asynchronous = no waiting for completion
    - send: no wait for message arrival
    - receive: announce willingness to accept
      or check for message arrival

Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous

type	Blocking Send	Blocking Receive	Language System
Syn	Yes	Yes	occam
Syn	Yes	No	-
Asyn	No	Yes	Mach Chorus
Asyn	No	No	Charlotte

Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous

Example: Occam style

Sender:

.....
Send(p, m);
{message is accepted!!!}
.....

Receiver:

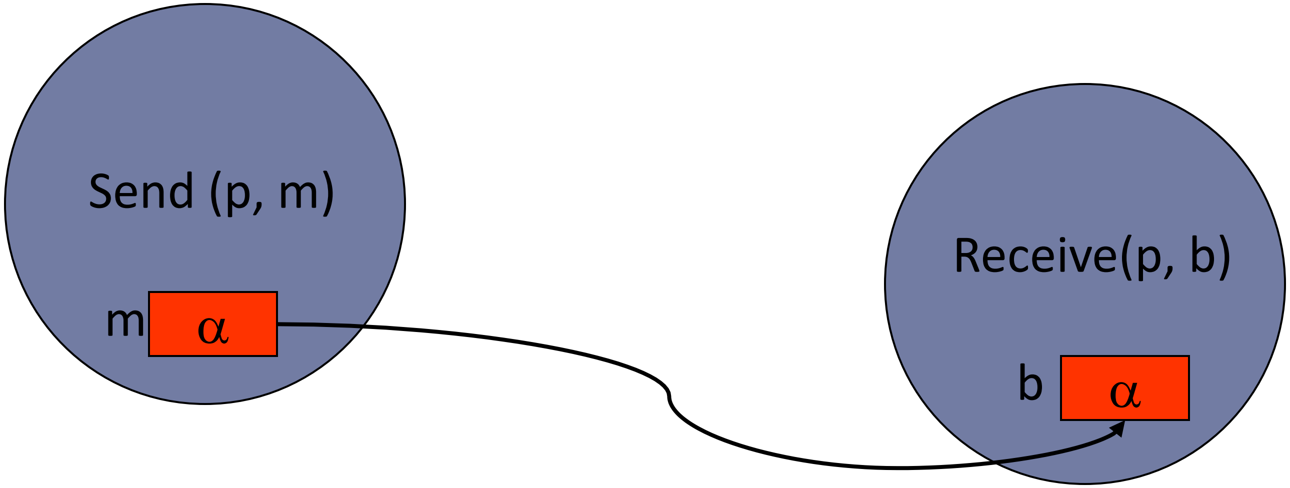
....
Receive(p, b);
{sender after send }
....

Communication ➔ Synchronisation point

b := m

Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous



Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous

Example: Mach style

Sender:

.....

send(....);

{message is in buffer!!!

arrival??}

Receiver:

....

receive(....);

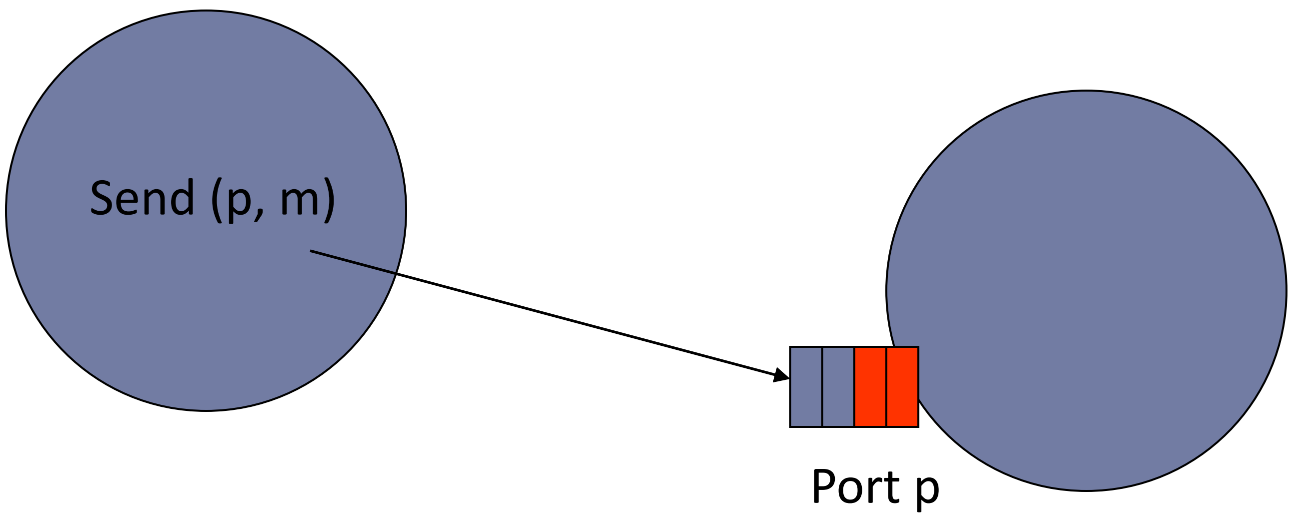
{message available }

....

Communication ➔ NO synchronisation point

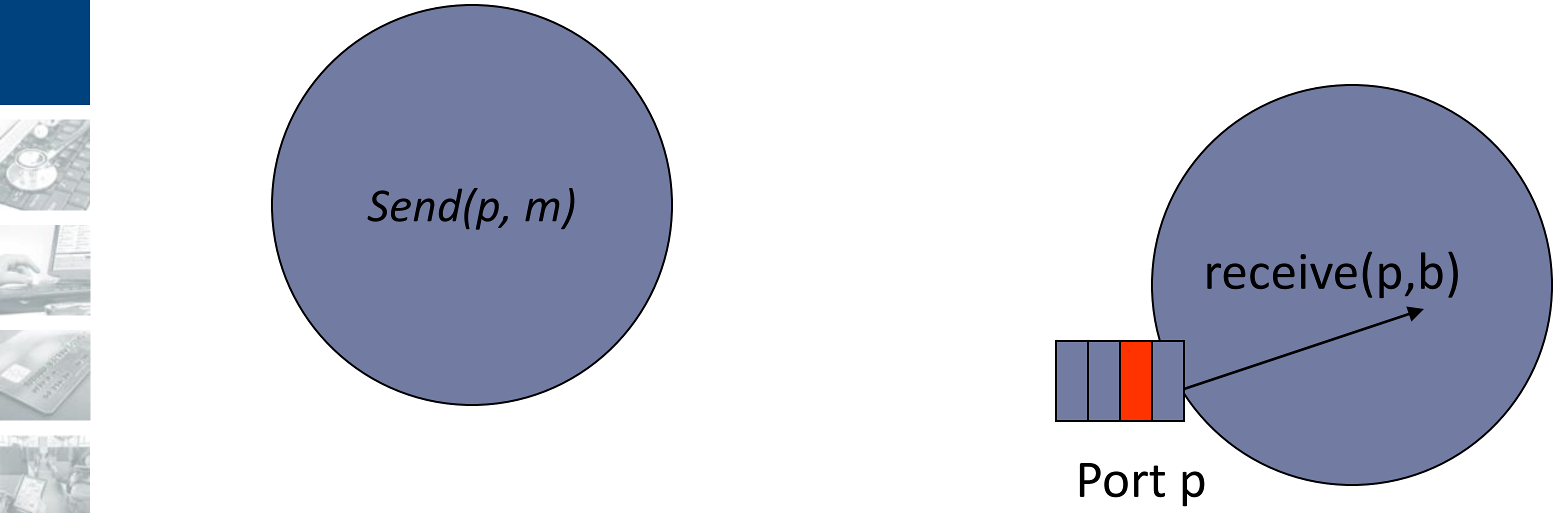
Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous



Message passing *(cont.)*

- **Semantics:** synchronous ⇔ asynchronous

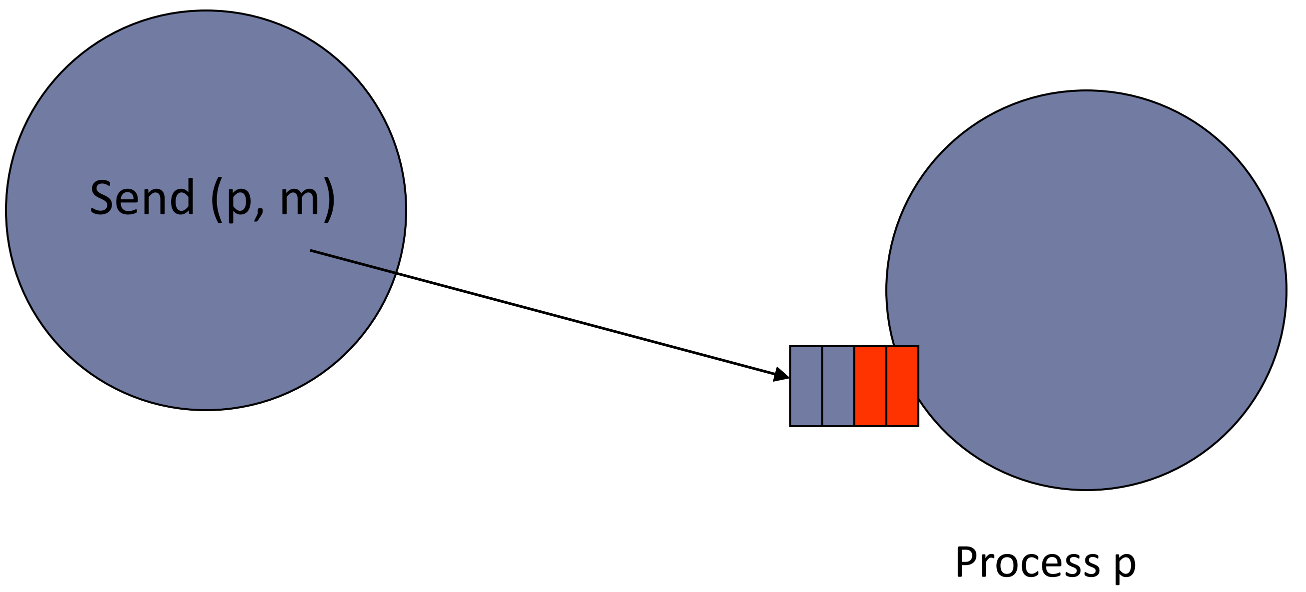


Message passing *(cont.)*

- **Semantics:** message destinations
  - message destination = communication identifier
    - preference for location independent identifiers
  - types of message destination:
    - process
    - port
    - mailbox

Message passing *(cont.)*

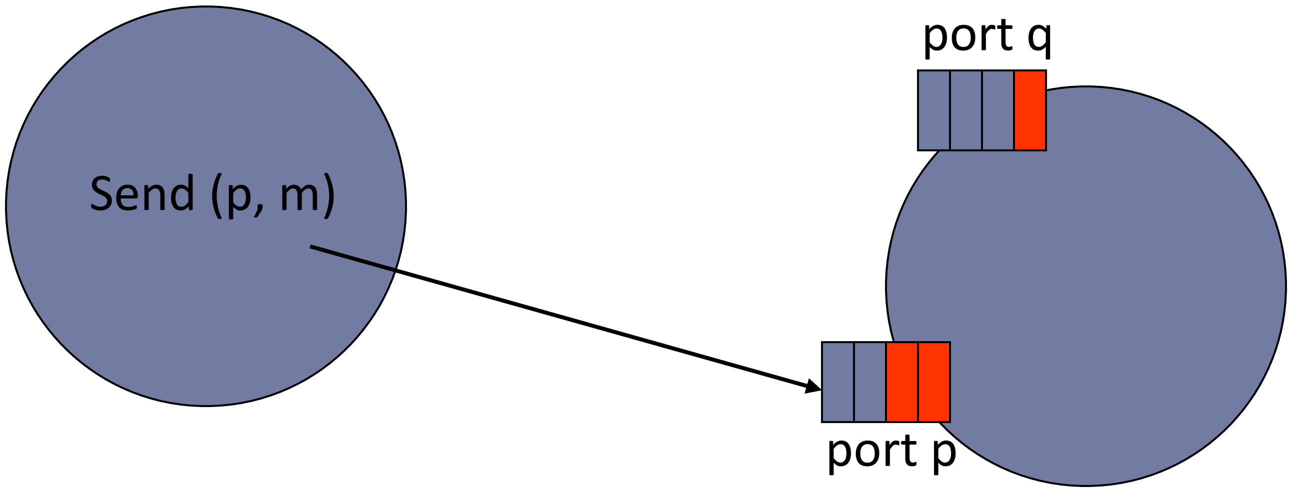
- **Semantics:** message destinations
  process



Message passing *(cont.)*

- **Semantics:** message destinations

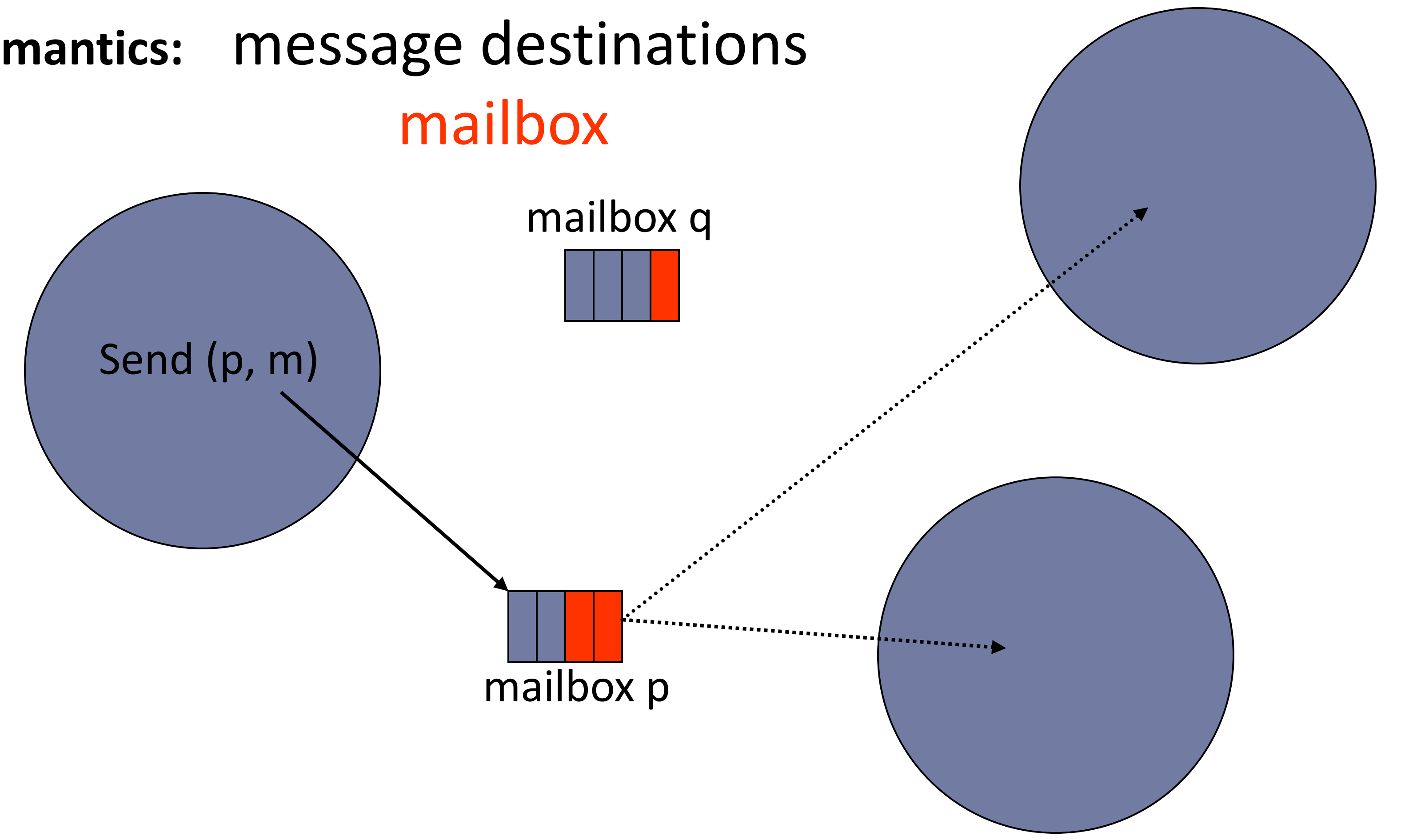
  port



Message passing *(cont.)*

- **Semantics:** message destinations

mailbox



Message passing *(cont.)*

- **Semantics:** message destinations
  - types of message destination:
    - process:
      - single entry point per process for all messages
    - port
      - one receiver, many senders
      - may have a message queue
      - many ports per process
    - mailbox
      - may have many receivers
      - message queue

Message passing *(cont.)*

- **Semantics:** reliability
  - possible failures
    - Corrupted messages
    - Duplicate messages
    - Omission: loss of messages
    - Messages out of order
    - Receiver process failure
  - Reliable communication
    - Delivered uncorrupted, in order, without duplicates
    - Despite a *reasonable* number of packets dropped or lost
    - Perfectly reliable communication can not often be guaranteed

Communication failure

Message passing *(cont.)*

- **How to implement reliable communication:**
  - Avoiding corruption
    - Include checksum in message
  - Avoids order mistakes and duplicates
    - Include a message number which identifies the message
  - Avoiding omission
    - Sender stores message in buffer, sends it and sets a time-out
    - Receiver replies with acknowledgement
    - Sender retransmits messages after timeout

Message passing *(cont.)*

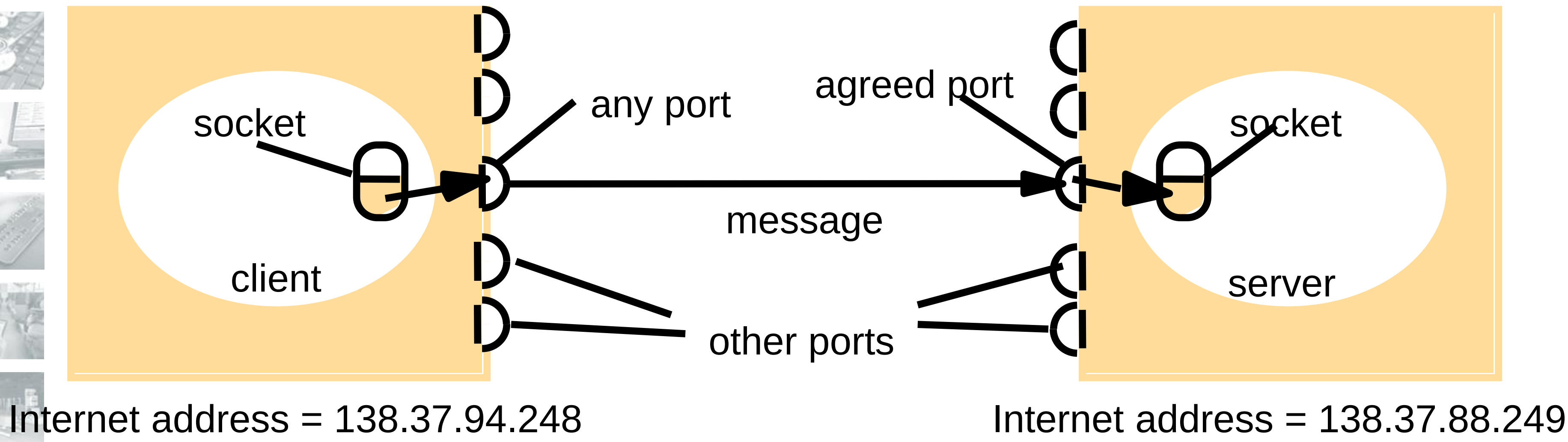
- **Semantics:** reliability
  - sender of message gets no reply?

    no distinction between

    - process failure
    - communication failure

Message passing *(cont.)*

- **Case study: UDP/TCP**
  - Sockets ⇔ ports
    - Socket bound to (TCP) port + Internet address



Message passing *(cont.)*

- **Case: UDP**
  - Messages:
    - Restricted packet size: < 2^{16} , < 2^{13} (8Kbytes), truncation
    - No conversion: bit transmission
  - Synchronization semantics:
    - Non-blocking send
    - Blocking receive
  - Timeouts: user can set timeout on receive operation
  - Receive from any: receive returns port + Internet address
    - But sockets can be bound to remote (IP address+port)
  - Unreliable message service
    - lost, out of order, duplicates
    - no message corruption: checksum

Message passing *(cont.)*

- **Case: TCP**
  - Stream communication: (⇔ message passing?)
    - Connect: create a communication channel through communicating sockets
    - Communication: read and write through channel
    - Close
  - Implementation:
    - TCP handles all communication
    - Uses buffers at sender and receiver side
    - No conversion: bit transmission
  - Synchronization semantics:
    - Non-blocking send, except for flow control (when buffers of sender or receiver are full)
    - Blocking receive

Message passing *(cont.)*

- **Case: TCP**
  - Setting up a client server connection:
    - Client sends request for communication to Server port
    - Server accepts client request
    - Typically, server creates new thread which handles communication with client
  - Reliable message service
    - Except broken connections
  - Overhead compared to UDP:
    - Buffering
    - Creating a connection: 2(?) extra messages
      - Sending a message, returning an acknowledgement
      - May create unacceptable overhead if goal is to send a single message.

Message passing *(cont.)*

- Conclusion: UDP, TCP
  - general purpose communication protocols
  - primitive, low level operations:
    - Setting up a communication
    - No transfer of structured data
  - Difficult to use
  - efficient implementation
  - building blocks used for more complex interactions

Message passing *(cont.)*

- Conclusion: message passing
  - primitive, low level operations
  - difficult, hard to use
  - efficient implementation
  - building blocks used for more complex interactions

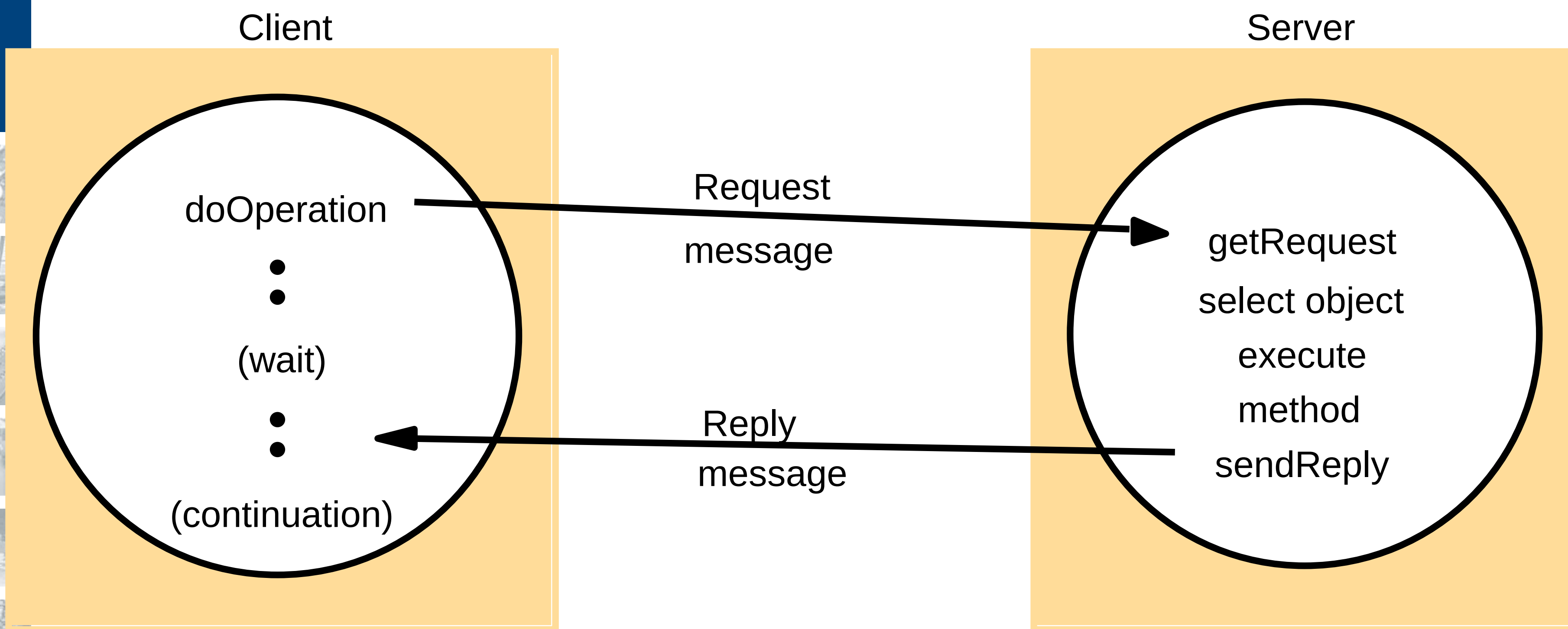
- From message passing
  - to Client-server (Request reply protocols)
  - to RPC, RMI

This lecture: overview

- Data representation
- Message passing
- Request-reply protocols
- Remote procedure calls

Request-Reply protocols

- Client-server communication



Request-Reply protocols

- Client-server messages/operations
  - Designed to support roles and message exchanges in typical client-server interactions
    - Acks redundant (replies are used)
    - Connections not necessary
    - No flow control
  - Basic operations:
    - Client: doOperation: sends request and returns answer to application program
    - Server: getRequest, sendReply

Request-Reply protocols *(cont.)*
Implementation options: on TCP and UDP

– Reliability measures of TCP are an overkill!!
  - Acknowledgement of receiver is redundant :
    the reply message is an acknowledgement!
  - Limited size of data packet transfer
  - One time communication

  => Making a connection is overhead

  => No stream needed, no flow control

⇒Use of TCP is (often) an overkill and may cause efficiency problems!

⇒UDP can be used for building more efficient client server communication.
  - What about reliability??

Request-Reply protocols *(cont.)*

Comparison: the case of implementing HTPP on top of TCP

- Case: Client-server communication: HTTP
  - Interactions:
    - Open connection
    - Client sends request
    - Server sends reply
    - Connection closed

    ✓ New since HTTP1.1: Persistent connections using TCP (for multiple requests at same server)
  - HTTP methods:
    - Get(URL) : request for the resource referred to by URL
    - Post(URL,data): replace or create resource at URL
    - ..

Message Passing and Request-Reply protocols: Conclusion

- So far rather low level implementations:
  - Setting up a connection
  - No transfer of structured data!!!
  - Concerns for synchronisation and failure model
  - No encryption of data
  - …
- Higher level Message Passing systems exist!!
  - MPI, Mach, ..

This chapter: overview

- Data representation
- Message passing
- Request-Reply protocols
- Remote procedure calls

This chapter: overview

- Data representation
- Message passing
- Request-Reply protocols
- Remote procedure calls

Remote procedure calls
Overview

- Basic principles
- Design issues
- Implementation aspects
- Asynchronous RPC

Remote procedure calls

Basic principles

- View on traditional application:
  - no OO yet!
  - application =

    main program

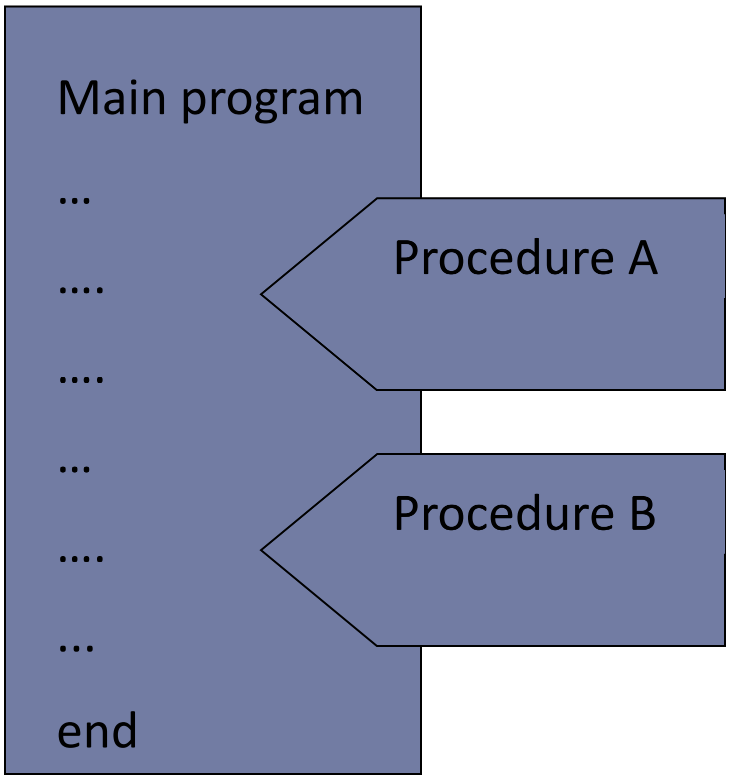
    \+ procedures (functions)

  ➔ familiar paradigm: procedure call

Remote procedure calls

Basic principles

- View on traditional application:



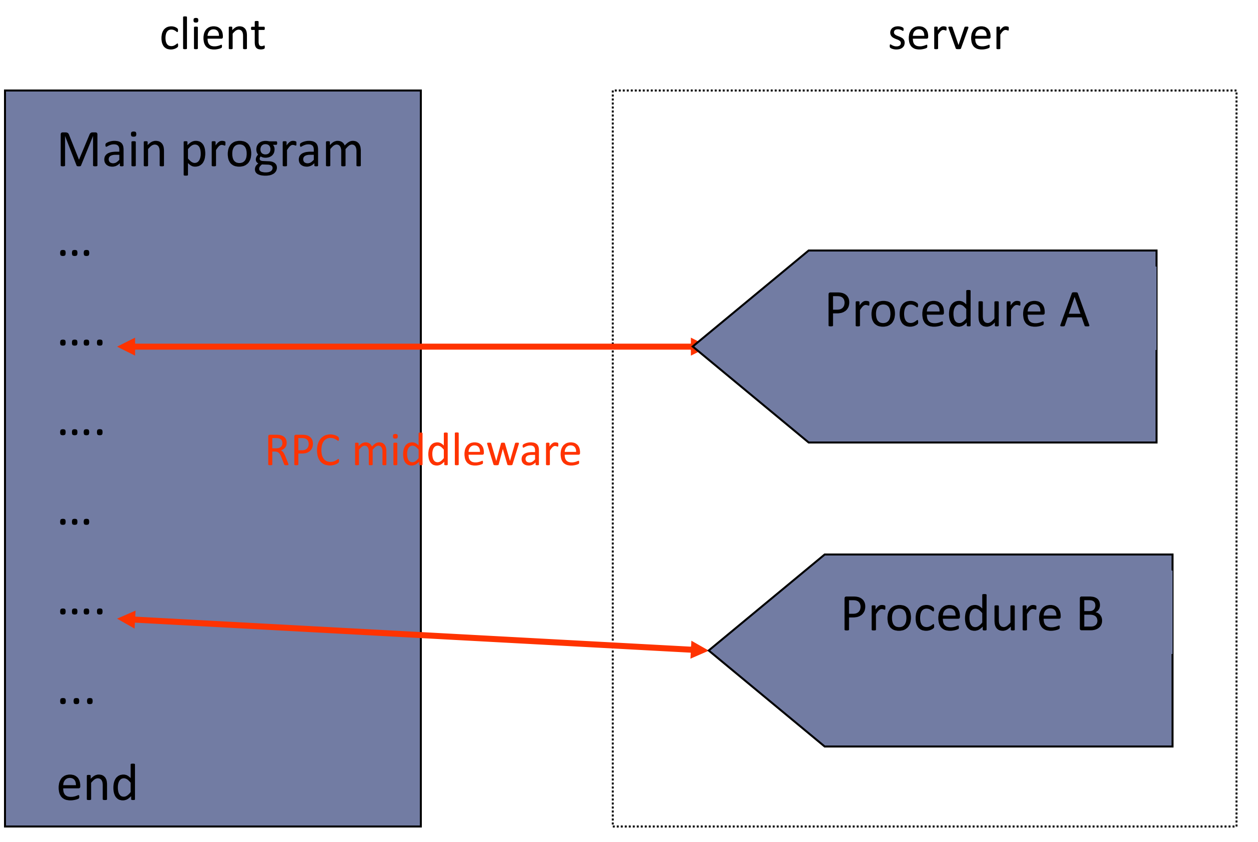
Remote procedure calls

Basic principles

- View on traditional application:
  - main program and procedures (functions)
  - familiar paradigm: procedure call
- approach in distributed systems:
  - group procedures into servers
  - main programs become clients
  - operations on server look like conventional procedure calls

Remote procedure calls
Basic principles

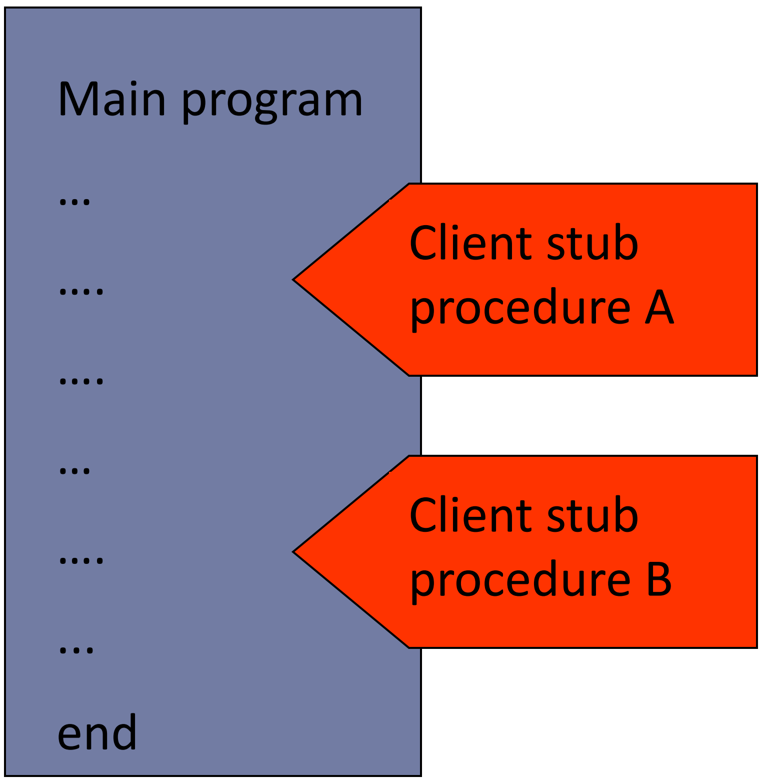
- Basic view on client-server model



Remote procedure calls

Basic principles

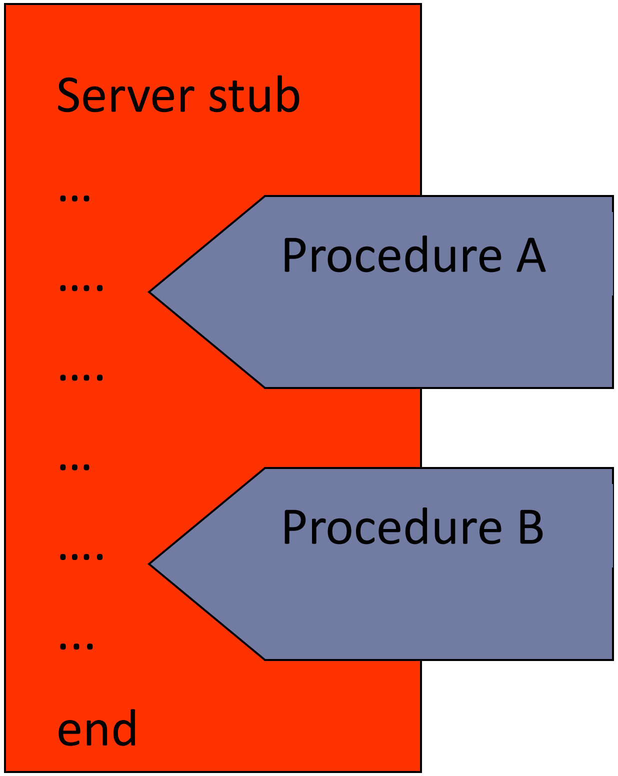
- RPC technology: client side



Remote procedure calls

Basic principles

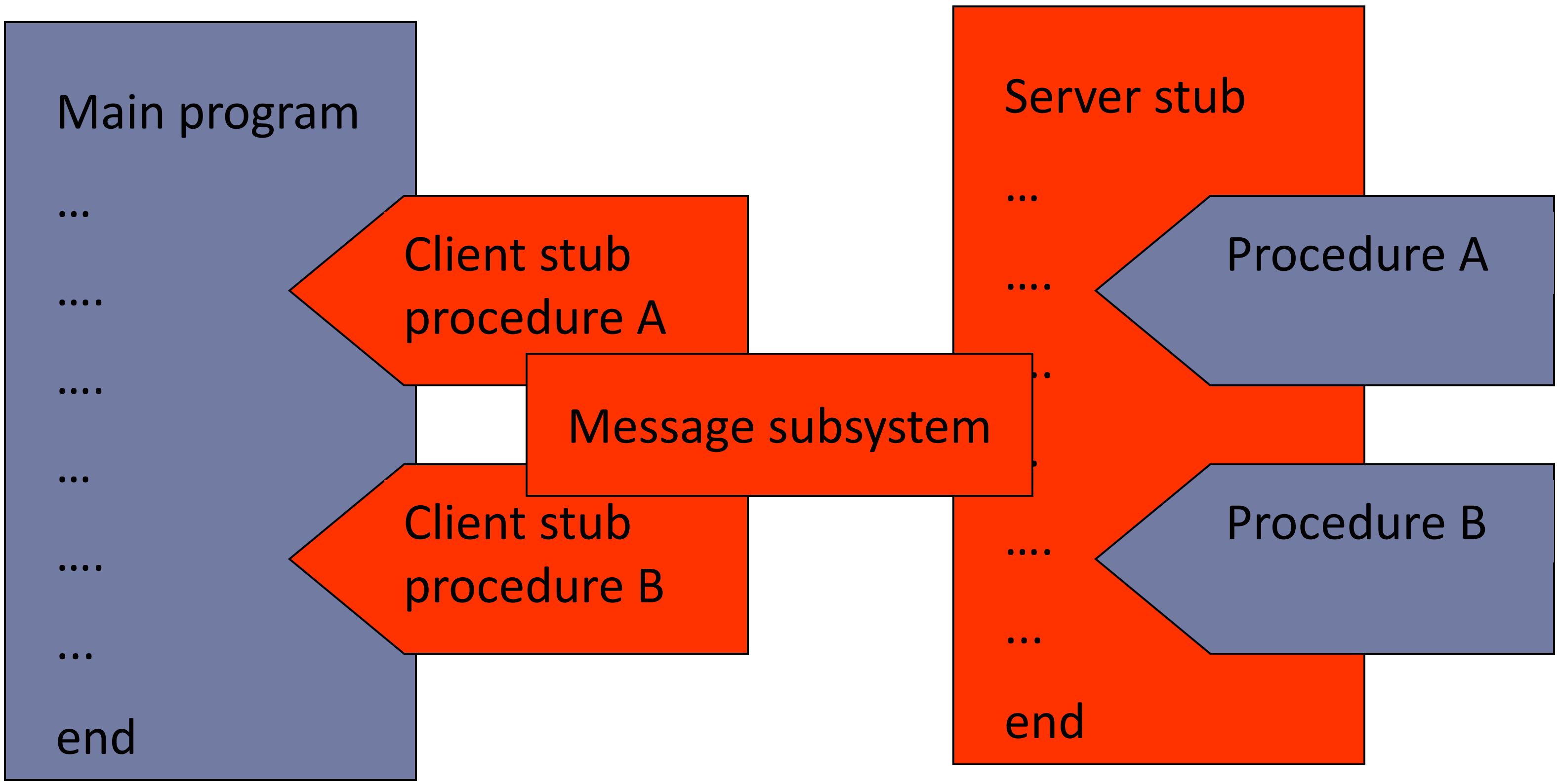
- RPC technology: server side



Remote procedure calls

Basic principles

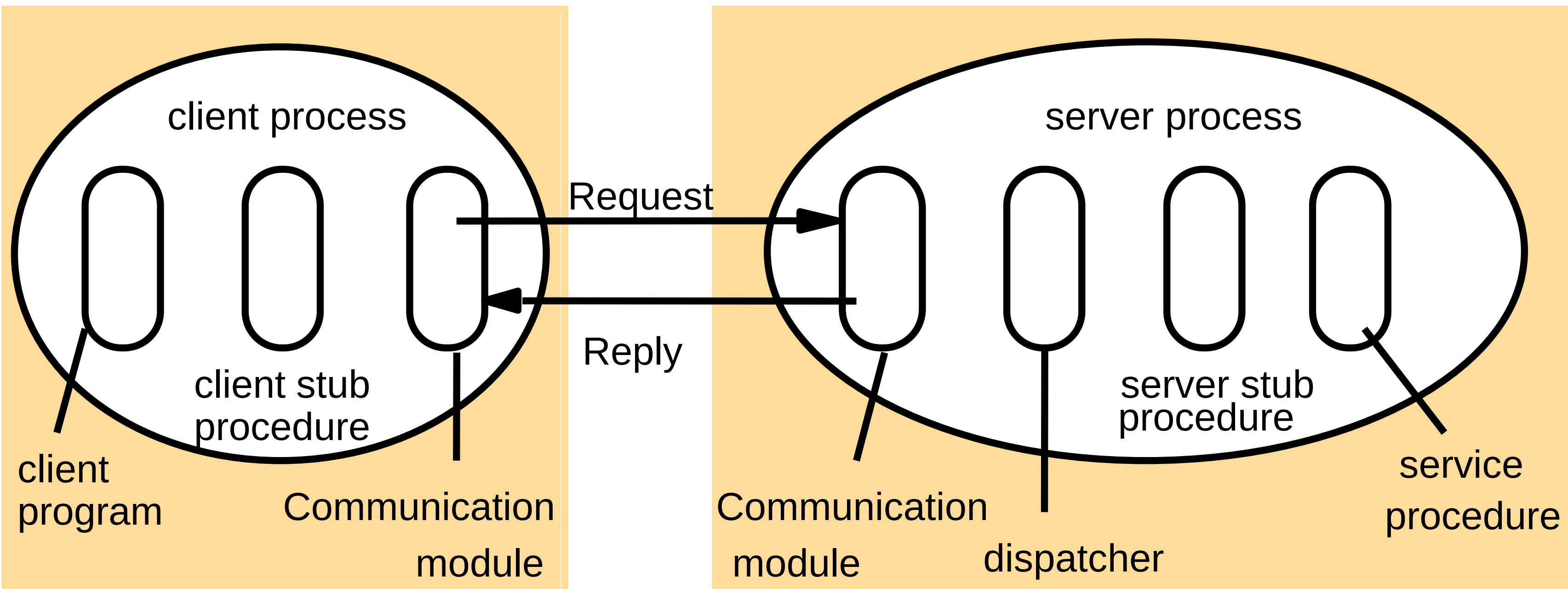
- RPC technology



Remote procedure calls

Basic principles

- RPC technology



Remote procedure calls

- Application program calls client stub procedure
- Client stub procedure marshalls parameters of call and gives it to communication module in client
- Communication module in client transmits a message with the marshalled RPC
- Communication module in server receives message and gives it to dispatcher
- Dispatcher determines which procedure is called and calls correct servers stub procedure with marshalled data
- Server Stub procedure unmarshalls data and calls the server procedure

Remote procedure calls

- Server procedure returns an answer to Server Stub procedure
- Server Stub procedure marshalls the answer and gives it to the communication module at server side
- Communication module at server side transmits the reply in a message
- Communication module at client side gives data to client stub procedure
- Who unmarshalls data and returns the answer to calling program

Remote procedure calls

- Primary characteristics:
  - code in client and server independent of communication system
  - familiar paradigm: procedure call
    - no message preparation
    - synchronous interaction
    - semantics?
  - IDL: Interface definition language
    - independent of language used for client or server
    - base for generation of stubs

Remote procedure calls

Design issues

- Classes of RPC systems
- Interface Definition Language (IDL)
- Exception handling
- Semantics of RPC
- Transparency

Remote procedure calls

Design issues

- Classes of RPC systems
  - RPC integrated within a particular programming language
    - e.g. Argus (with CLU), Arjuna (C++)
  - RPC based on a special IDL
    - e.g. Sun RPC, ANSA RPC, OSF/DCE

Remote procedure calls

Design issues

- Interface Definition Language
  - Describes operation signatures
  - Interface compilers for
    - Generating client and server stubs
    - In different languages (e.g. C, Pascal...)
    - ➔Abstraction of heterogeneity

Remote procedure calls

Design issues

- Exception handling
  - failures cannot be hidden!
    - Network ⇔ server failure?
    - Client cannot distinguish
  - approaches to support failures:
    - Language specific
    - using return codes of functions
    - extension provided by IDL

Remote procedure calls

Design issues

- Semantics of RPC
  - Maybe
  - At-least-once (e.g. Sun RPC)
  - At-most-once (e.g. ANSA RPC)
  - Exactly-once:
    - difficult or impossible given failures

Remote procedure calls

Delivery guarantees			RPC semantics
retry request	duplicate filtering	re-exec retrans reply	
no	not applic.	not.	Maybe
yes	no	re-exec	At-least-once
yes	yes	retrans reply	At-most-once

Remote procedure calls

Design issues

- Transparency
  - *"make RPC as much like local procedure calls as possible"*
  - RPC more vulnerable to failure
    - possibility of failure should not be hidden
  - calling instructions different
  - no shared memory between caller and callee

programming convenience

vs.

true transparency

Remote procedure calls

Implementation aspects

- Tasks for interface compiler: generate
  - client stub procedure
  - server stub procedure
  - marshalling and unmarshalling operations for each argument type
  - header for server procedure

Remote procedure calls
Implementation aspects

- Binding:
  linking client to server at execution time
- Binder interface

Procedure Register (serviceName: String;
serverPort: Port; version: integer);

Procedure Withdraw (…);

Procedure Lookup(serviceName: String;
version:…): Port;

Remote procedure calls

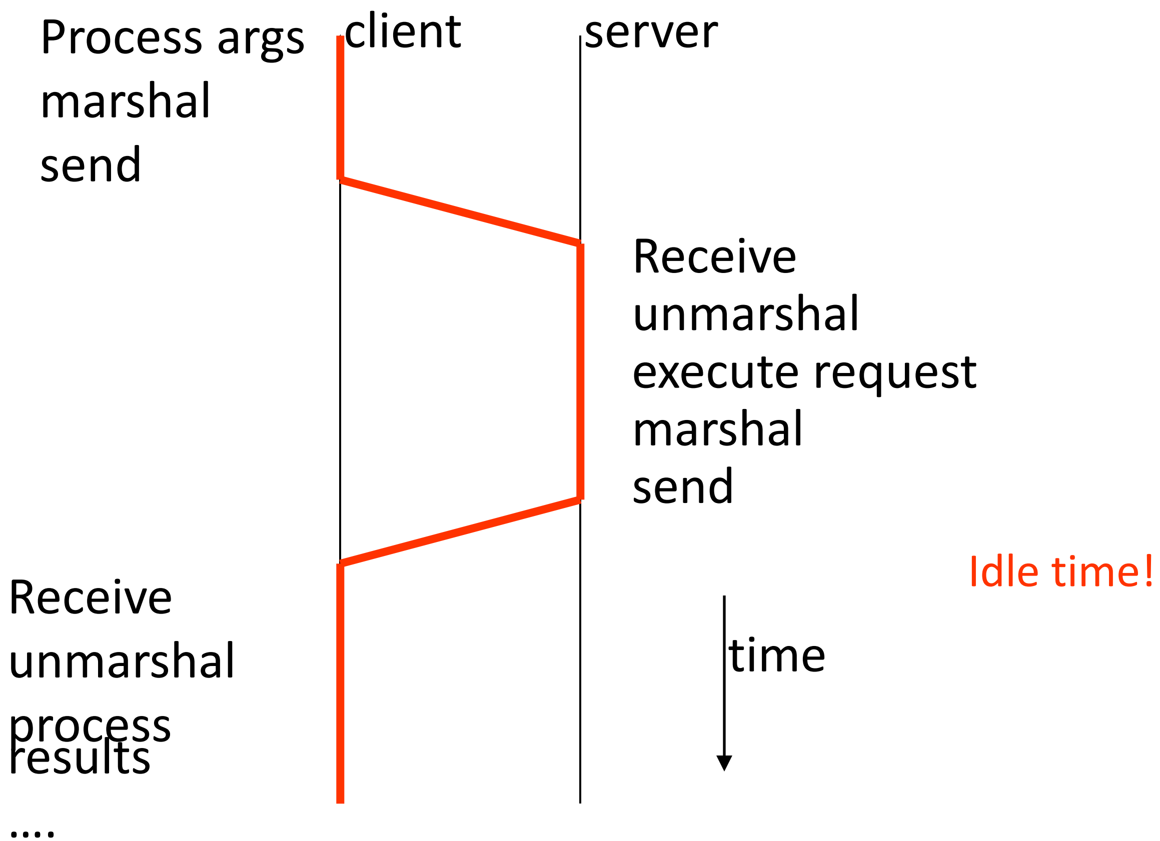
Implementation aspects

- Binding:
  - server will register service at binder
  - client will lookup the service
- Locating the binder?
  - well known host address
  - responsibility of OS
  - broadcast message by client

Remote procedure calls

Asynchronous RPC

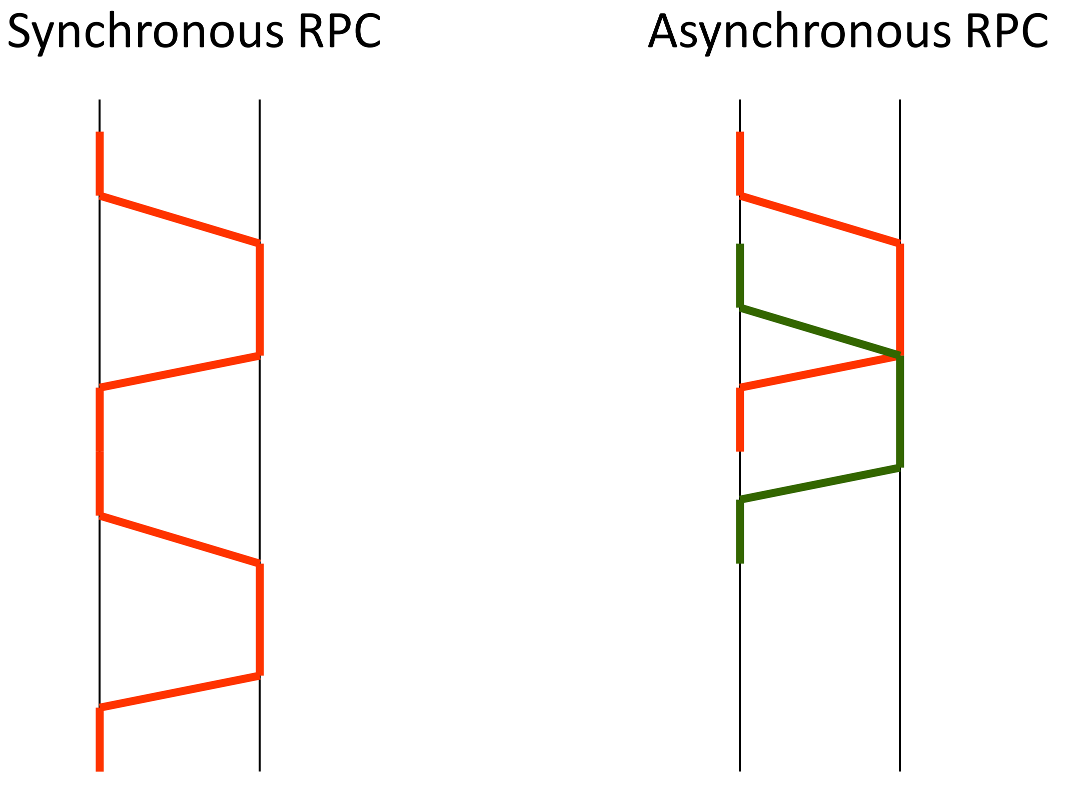
- Problem: throughput RPC?



Remote procedure calls

Asynchronous RPC

- Timing:



Remote procedure calls

Asynchronous RPC

- When to use?
  - Many request, small amount of information, limited processing: e.g. windows system
  - parallel requests to several servers
- Additional optimisations:
  - buffer request at client until …
  - proceed without waiting when no reply is required

Remote procedure calls

Asynchronous RPC

- Extensions:
  - call streams:
    - mix of synchronous and asynchronous calls
    - message ordering preserved
    - connection oriented
    - connection breaks when semantics cannot be guaranteed

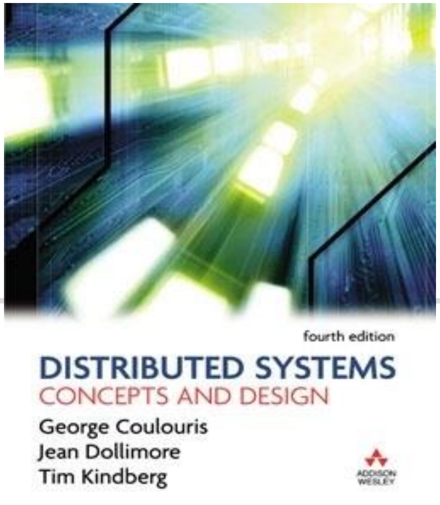
Remote procedure calls

Asynchronous RPC

- Extensions:
  - Promises:
    - allow clients to continue with (other work) and retrieve result of call later
    - created at the time of call
    - store results of call (object with same type)
    - operations:
      - get, await result
      - result available?
    - Alternative names: futures, tickets, continuations

Remote procedure calls

- Conclusion:
  - familiar paradigm
  - has been basic primitive for distributed programming for many applications and systems…
  - limitations
    - Failures to be handled by clients (hard)
    - No transaction support
    - Only one-to-one communication



KATHOLIEKE UNIVERSITEIT LEUVEN

Distributed Systems:
Direct Communication – Part I
Questions?