Programming of Distributed Systems

Topic II – Communication Models

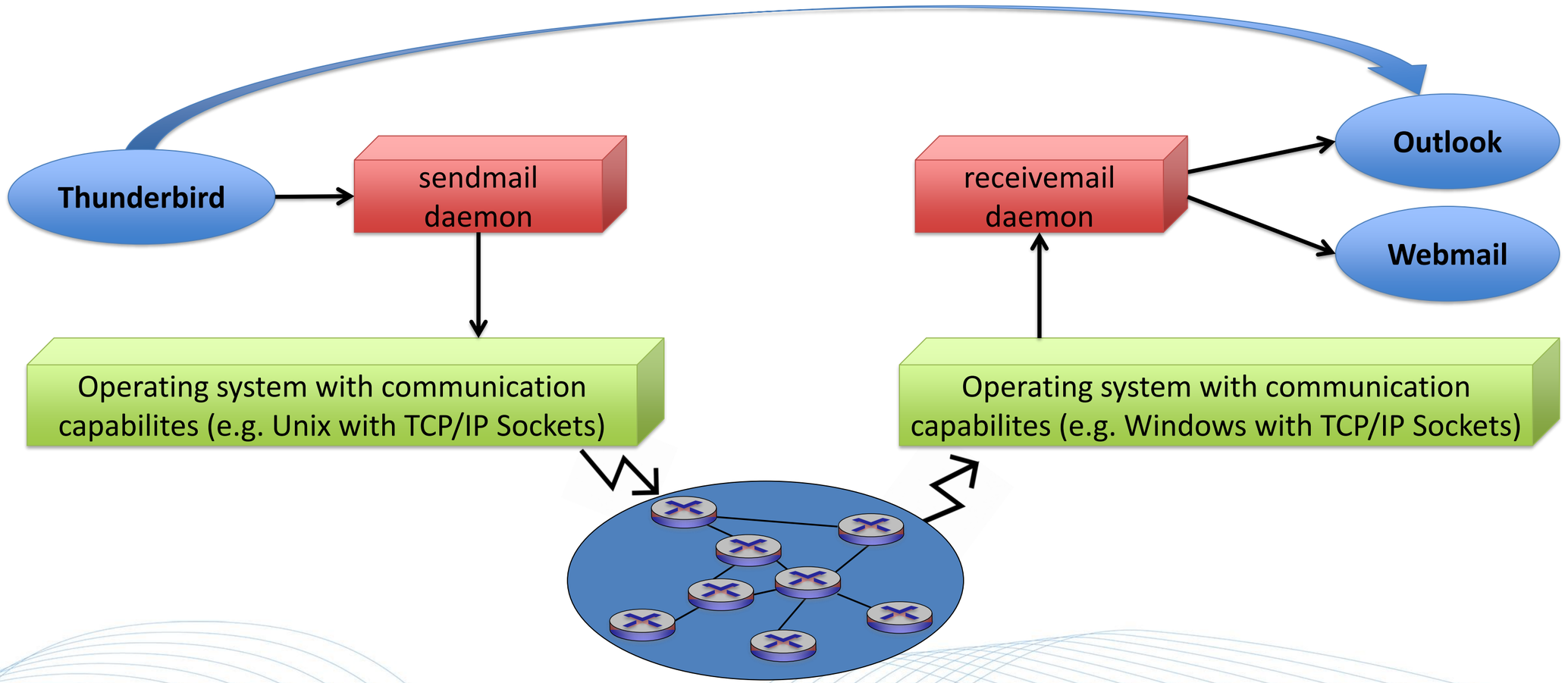
Dr.-Ing. Dipl.-Inf. Erik Schaffernicht

Reading Remarks

Reading Task:
Read chapter 2-4 of the course book!

Order and organization in the book differs from the presentation in the lecture significantly.

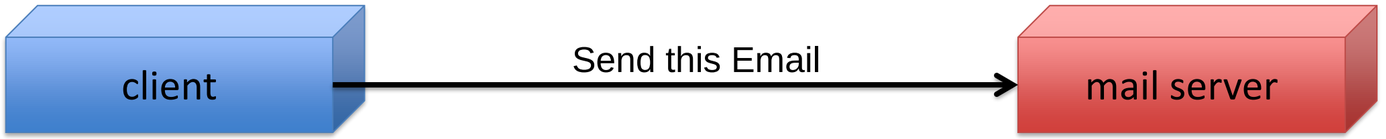
Communication via Email



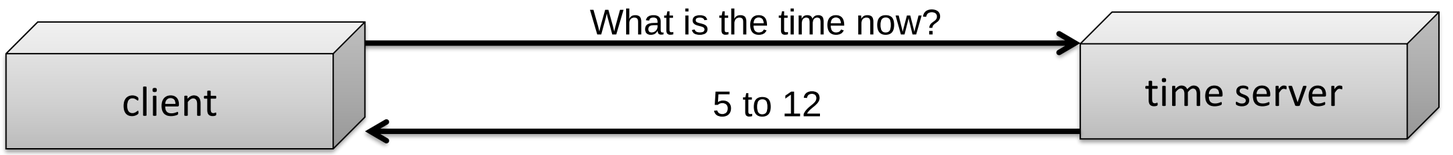
Communication via Email – Observations

Communication pattern

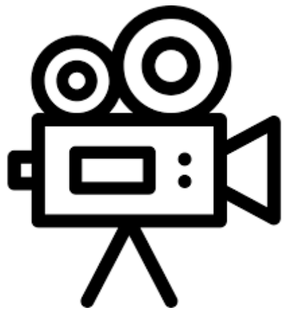
- one or more receivers → unicast / multicast
- arbitrary content
- information flow in only one direction → unidirectional
- sender does not wait for mail delivery → asynchronous
- delivery comes without guarantees → unreliable

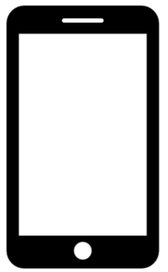


Other communication patterns

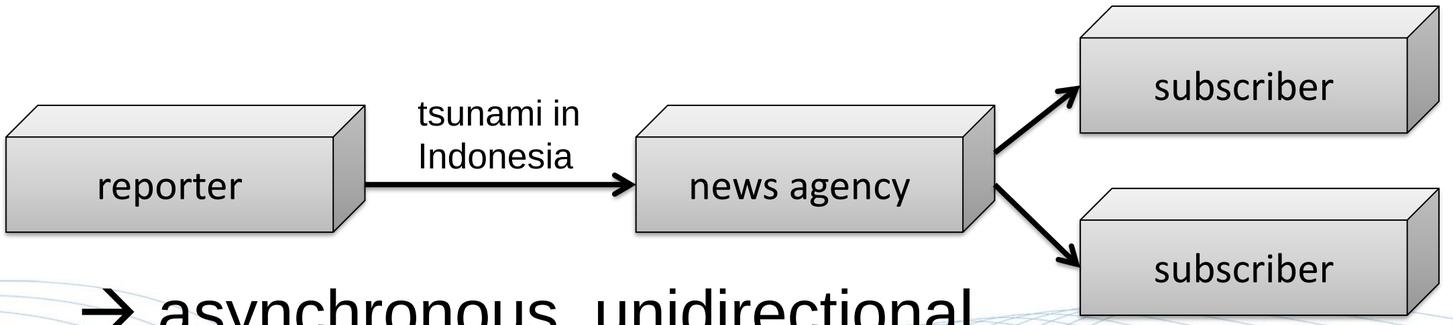


→ synchronous, bidirectional





→ synchronous, unidirectional



→ asynchronous, unidirectional

Further observations

Different roles of the communicating entities

- sender, receiver, stream source or sink, client, server, reporter, intermediary, recipient

Different types of communicated data

- messages, requests, results, assignments, data streams (audio, video, sensor), events

Different criteria for terminating the communication

- after sending an email, after recieving an answer, after the stream runs dry

Conclusion from the examples

- communication patterns are problem specific
- components of communication patterns need to be clear for all parties

➔ **common rule book** required: communication models

- definition of who is a potential communication partner
- definition of set of rules to achieve understanding between communication partners

Communication particpants

Software components on several abstraction levels

- processes/threads (IPC, RPCs)
- components of distributed applications, e.g.
  - web browser and web server
  - database clients and database server

which do not necessarily are in the same place

- no shared address space
- possible different IT systems

Four Parts of Communication Models

1. role schema
2. data model
3. failure semantics
4. termination semantics

Role Schema

Goal:

- patterns of action for communication partners

Method:

- define tasks and actions for each partner:

  Which actions are expected to be exectued when and by which partner?

Data Model

Goal:

- Have a common interpretation of the communicated data

Method:

- define rules to interprete the exchanged data
  stream structure, method signatures, packet formats,
  number and character representation, etc.

Failure Semantics

Goal:

- agreement over effects of communication failures

Method:

- define properties for communication actions with certain guarantees with failure of either the communication partner or the communication network

Termination Semantics

Goal:

- agreement over the end of a communication action

Method:

- define guarantees for the end of communication actions especially in the presence of failures

Message-based Communication

Base for all other considered communication models

- **Roles:** Sender / Receiver (→ Send/Receive model)
- **Data:** Messages
- Plenty of different termination and failure semantics possible

Available in almost every operating system (sockets, msg, ...)

Used to implement of more specific models (→ Middleware)

Two elementary operations

Sending a message to a receiver

***send*(IN receiver, IN message)**

Receiving a message from a sender

***receive*(OUT sender, OUT message)**

Variants of Send/Receive Models

Role Schema

- One or more receivers (unicast/multicast)
- Direct or indirect communication

Failure Semantics

- reliable or unreliable send

Termination Semantics

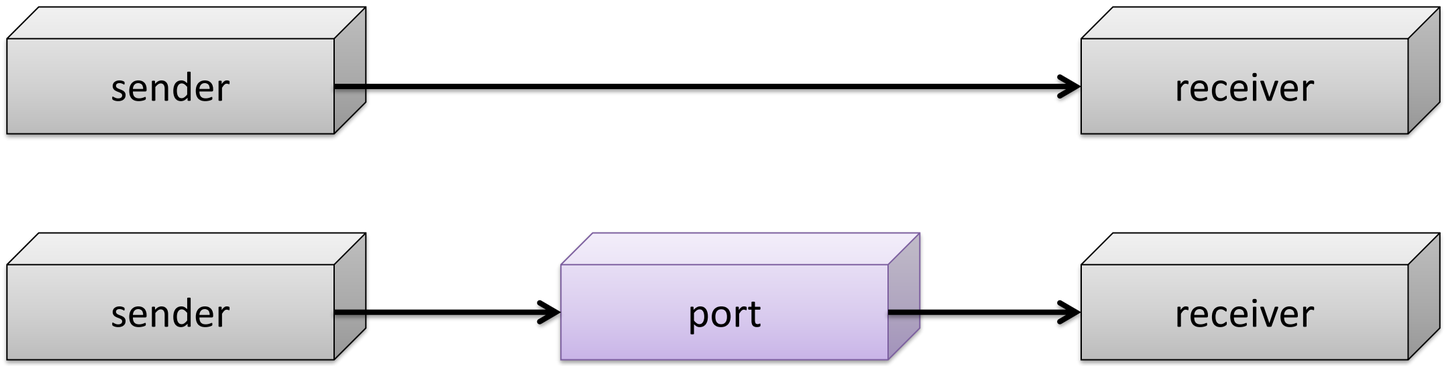
- Synchronous or asychronous send/receive

Unicast / Multicast

Application examples for multi-cast:

- Replicating fileserver
- Redundant calculation
- Multiplying datastreams

Direct vs. Indirect communiction



Port

- Object in the messaging system to send messages to and retrieve from with a name (123.123.123.123:**80**)

→ message storage & filtering, duplication, access control

Ports – Pros & Cons

Pros

- Persistence and location transparency of port names
- Queing and buffering → QoS
- Access control through interceptors → IT security

Cons

- Increased resources (time & memory) for creating message copies
- Increased amount of OS system calls

Reliable and Unreliable Send

Unreliable send

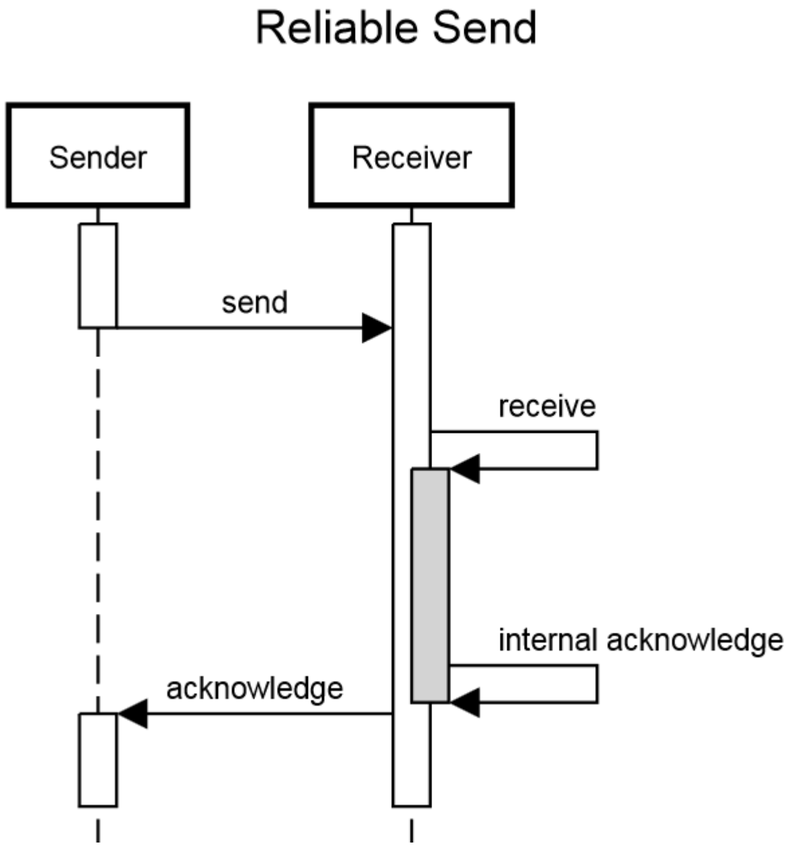
→ Sender does not know if the message has been received

Reliable send

→ Sender gets an acknowledgement of successful transmission

Doubled transport costs (time, data)

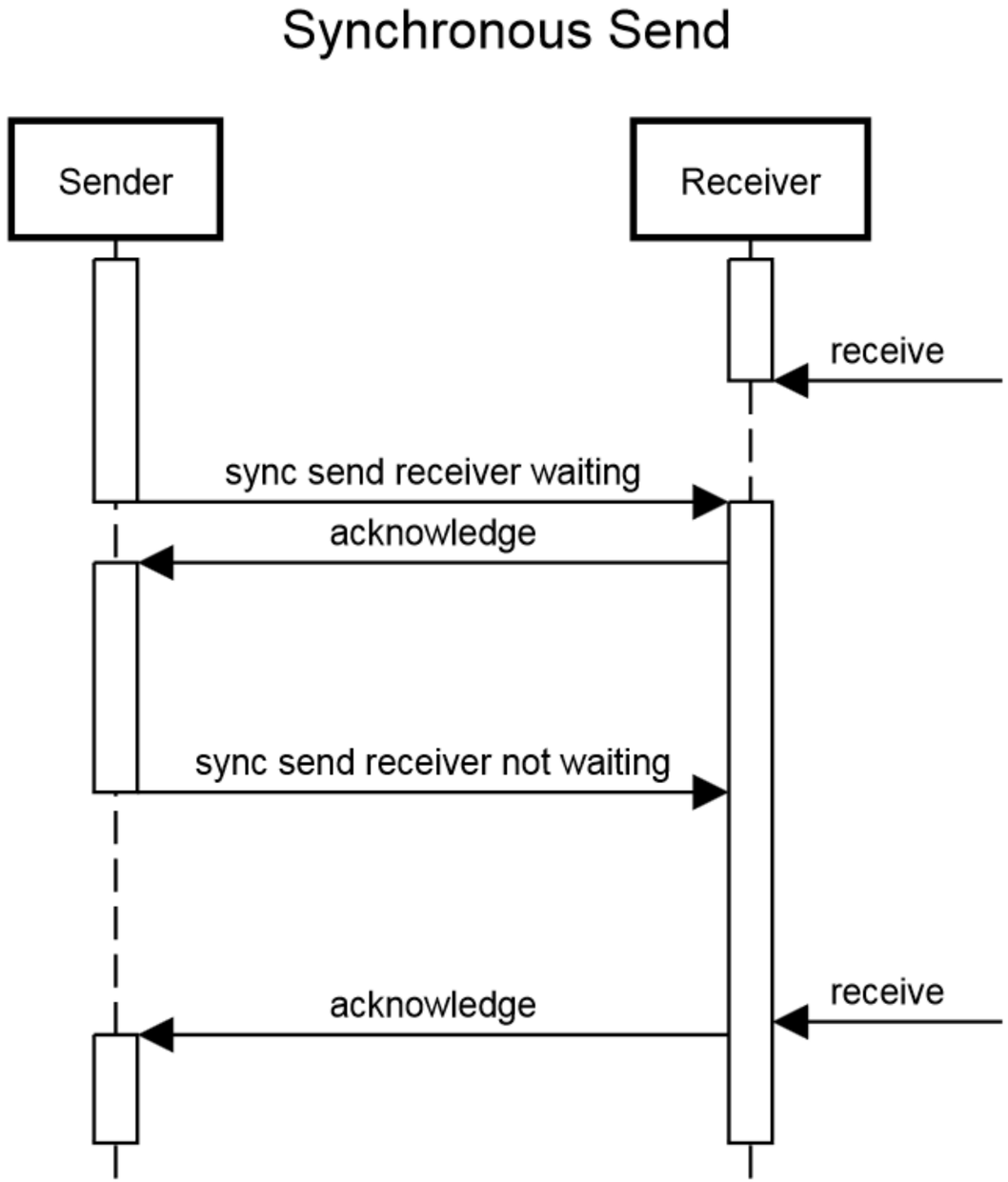
Timeouts (→ time management)



Synchronous Send

Sender waits for the receiver to accept the message

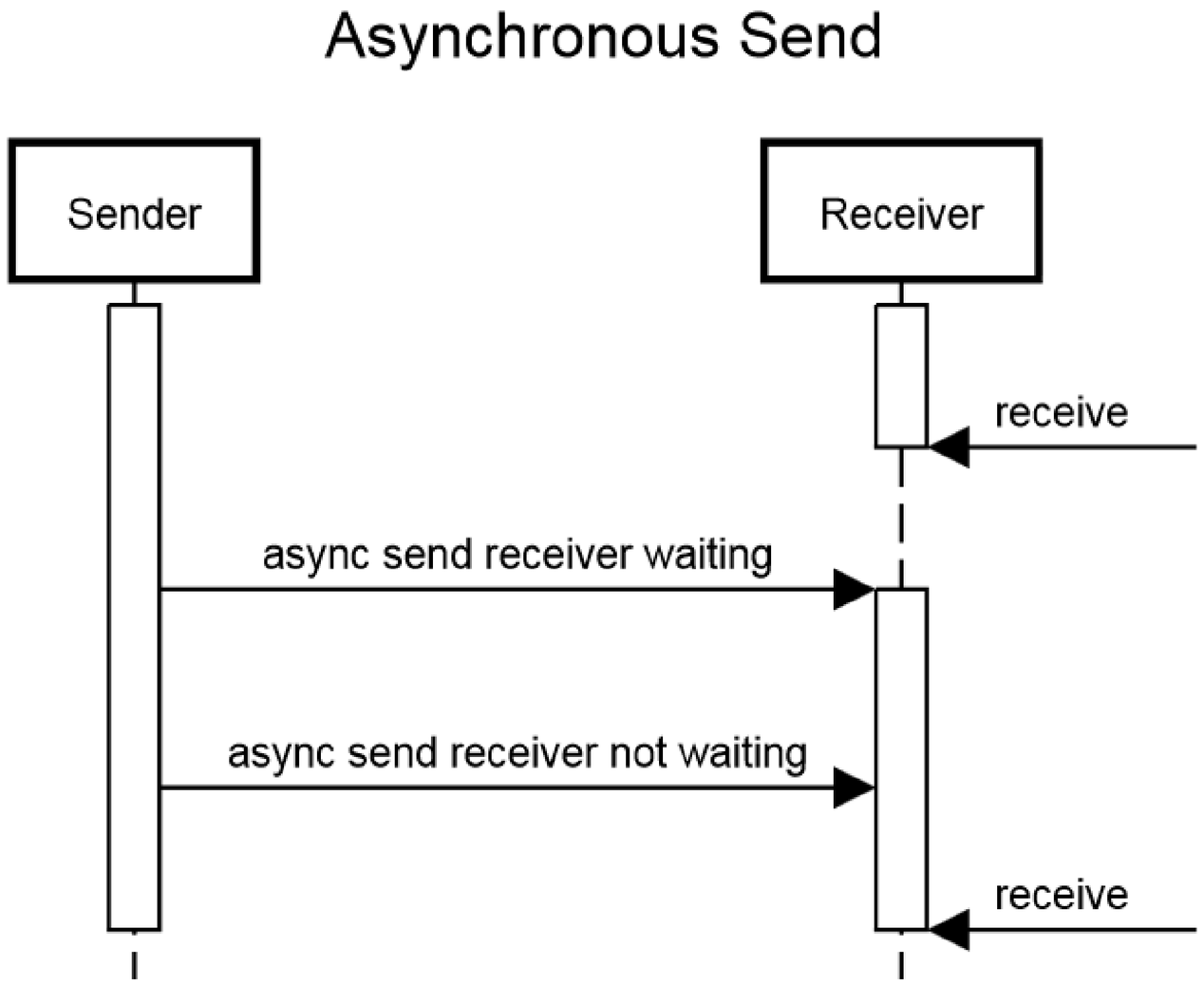
- may take time during which the sender is blocked
- guaranteed message arrival
- simple programming of sychronous activities



Asynchronous Send

Sender does not waits for the receiver to accept the message

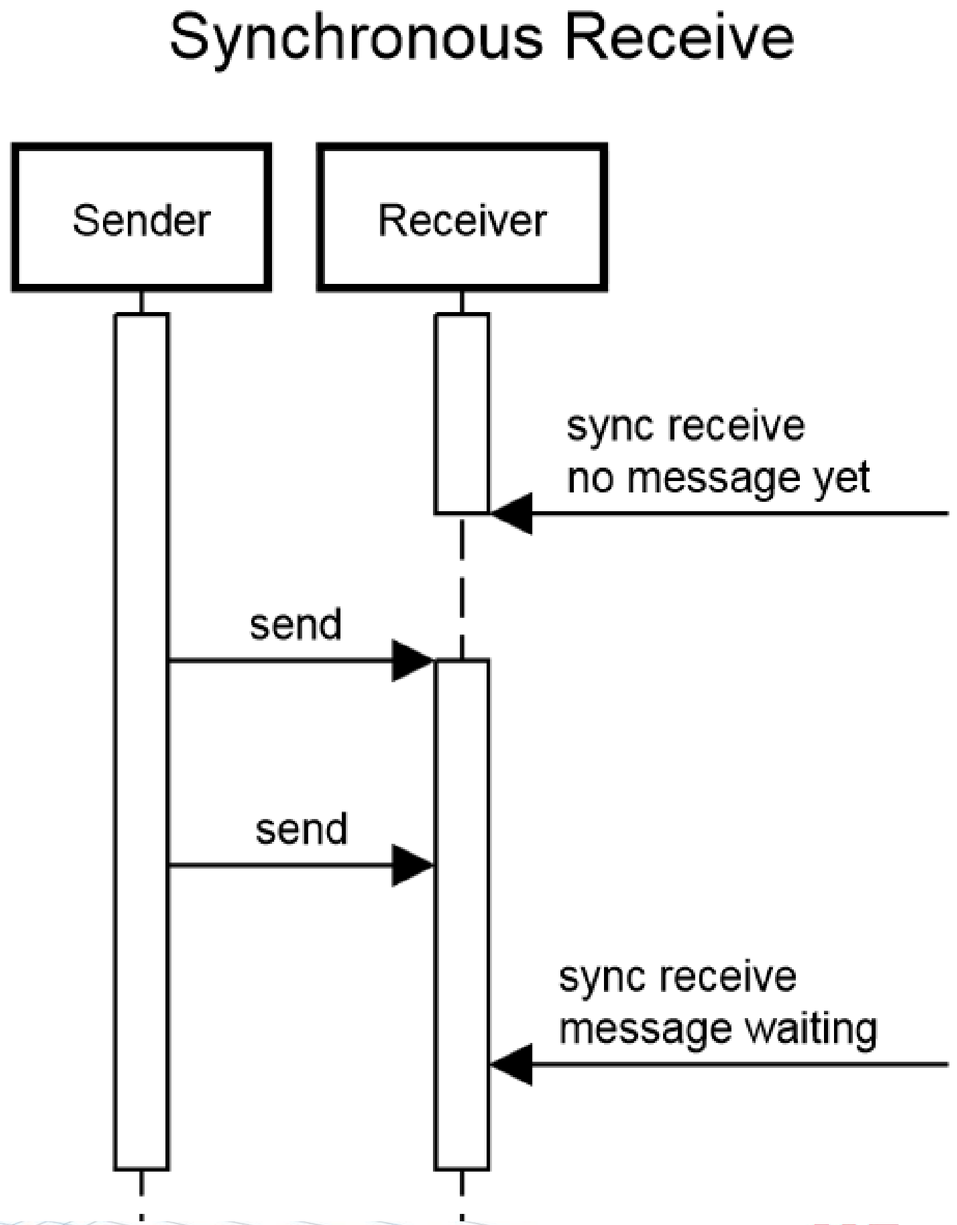
- Unclear if and when message is delivered
- Simple concurrent / parallel programming



Synchronous Receive

Receiver waits until message arrives

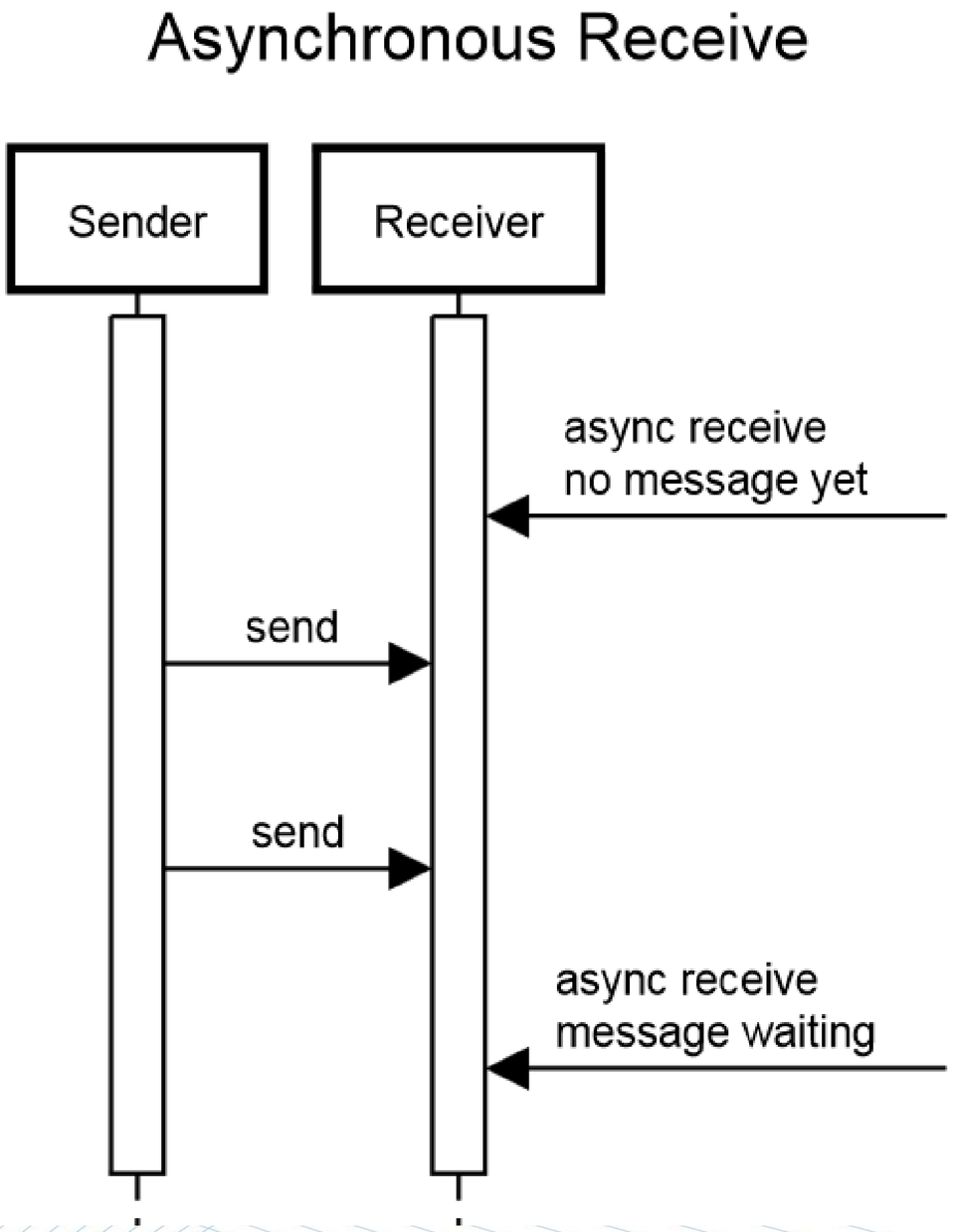
- may take time during which the receiver is blocked
- arrival time is exactly known
- simple programming of sychronous activities



Asynchronous Receive

Receiver continues if there is no message

- Unclear when messages arrive (internal messaging:polling, interrupt/events, message created LWP)
- Simple concurrent / parallel programming



Summary Termination Semantics

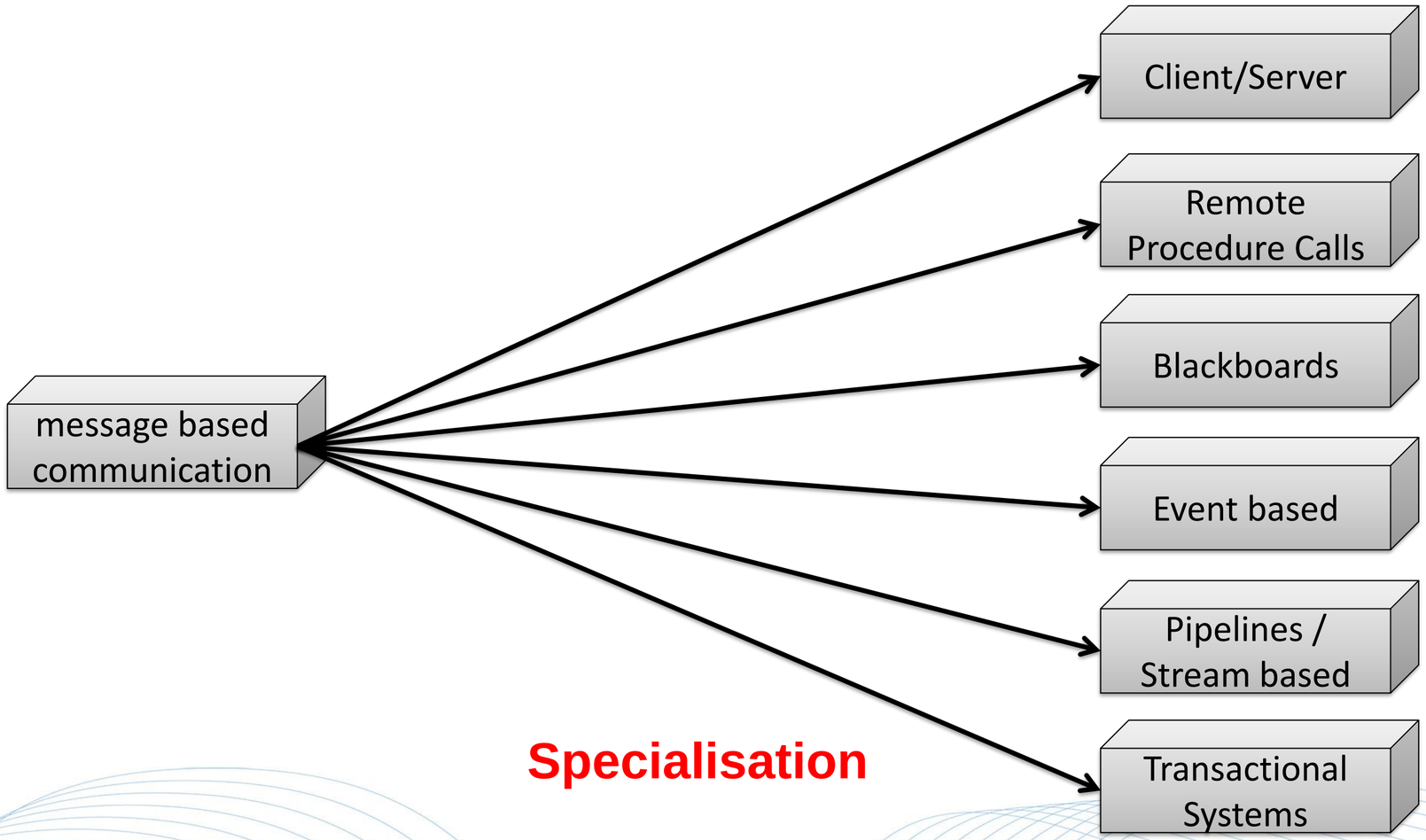
Synchronous Operations

- simple synchronisation
- parallel computing using threads
- DoS attacks by not answering

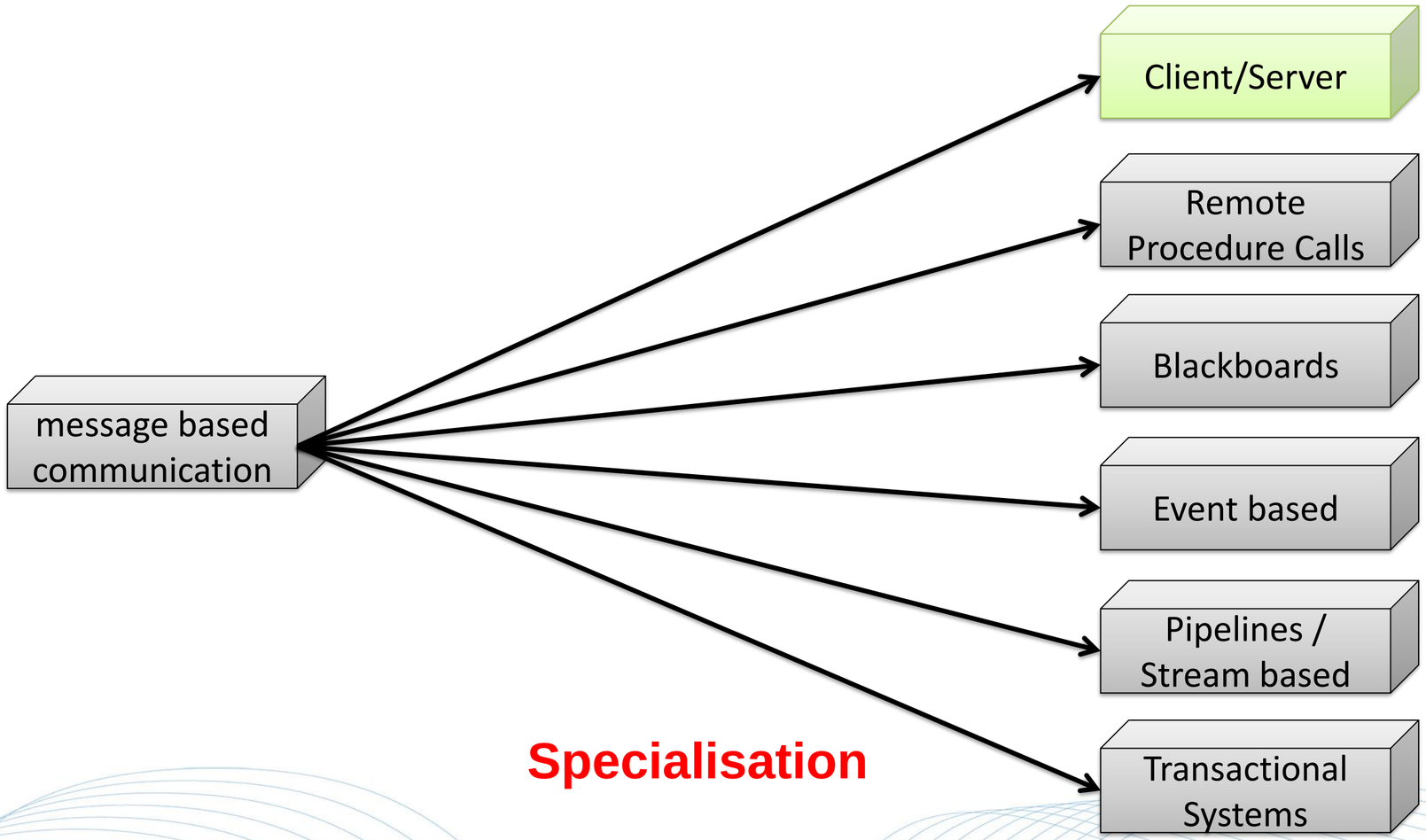
Asynchronous Operations

- complex sychronisation
- no threads required for parallel computing
- DoS attacks by flooding

Foundation for more complex models



Client / Server Models



Client / Server Models

Classic model for service oriented system designs

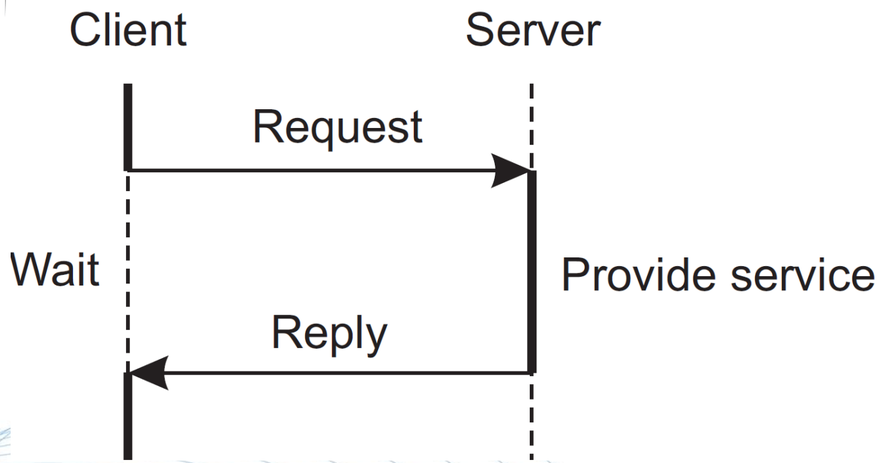
Application examples

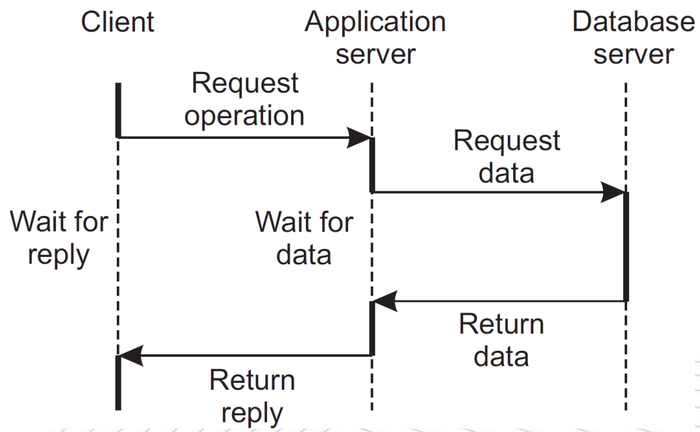
- Data base systems (client query / SQL server)
- DNS, file server, time server, authentification server
- web browser, web server
- Email

Client-Server

A client is a process that requests a service from a server.

A server is a process that implements a service.





An SMTP example

In general

Roles: Client/Server

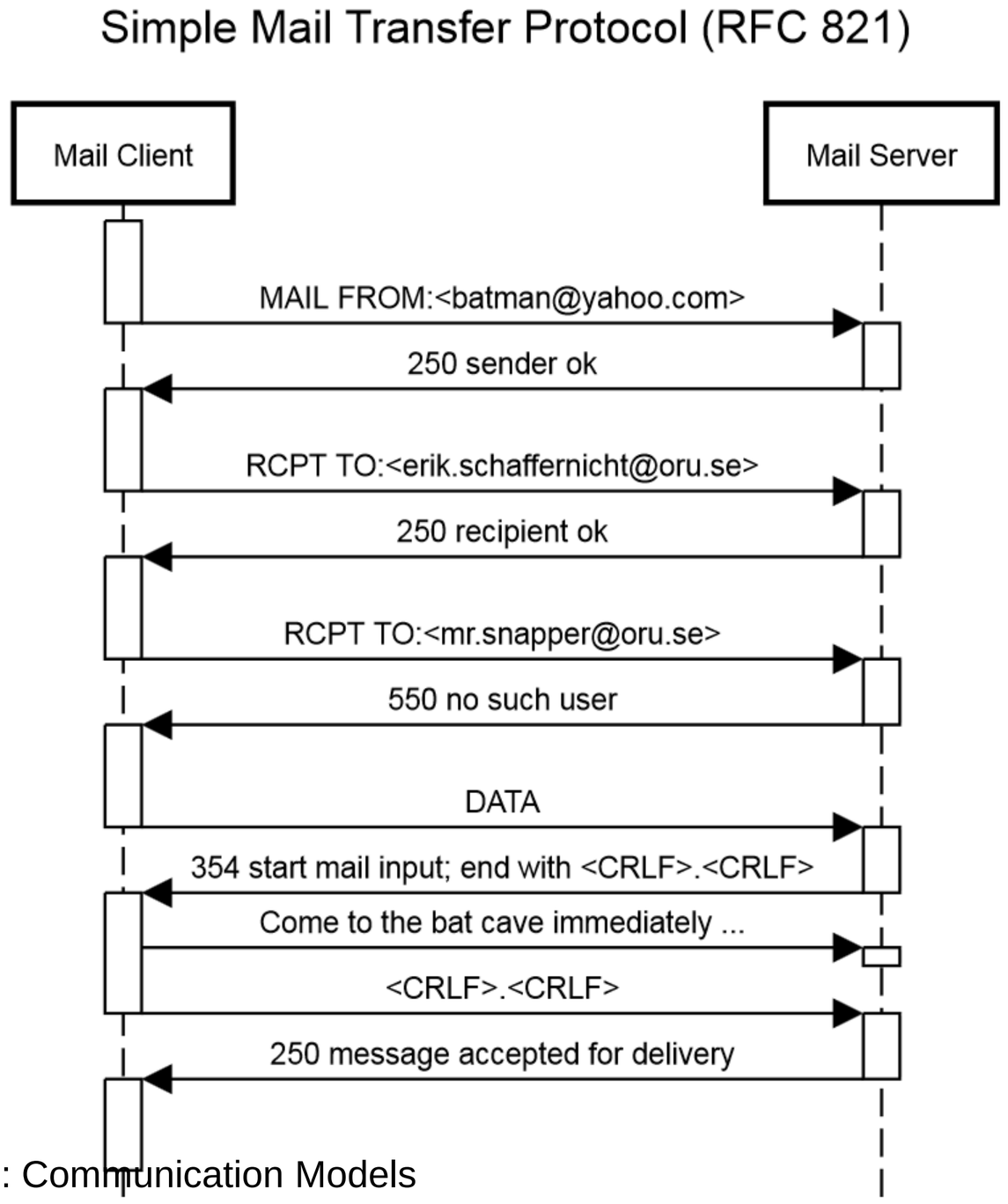
Data: messages with agreed structure (SMTP, SOAP, etc.)

Termination: often sychronous

Failure semantics:

What is the reason I did not get an answer?

→ More detailled look later

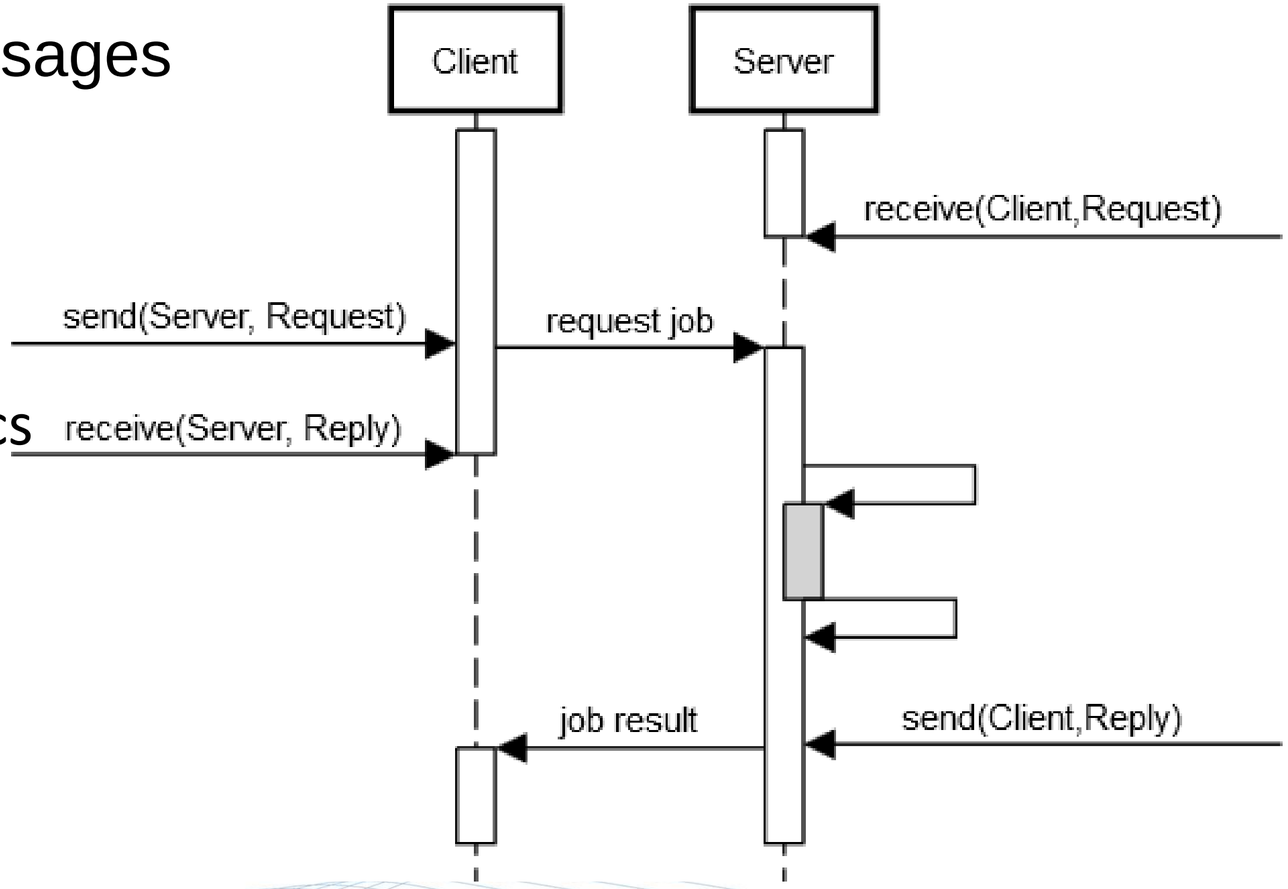


Realisation of Client/Server Model

Based on basic send/receive messages

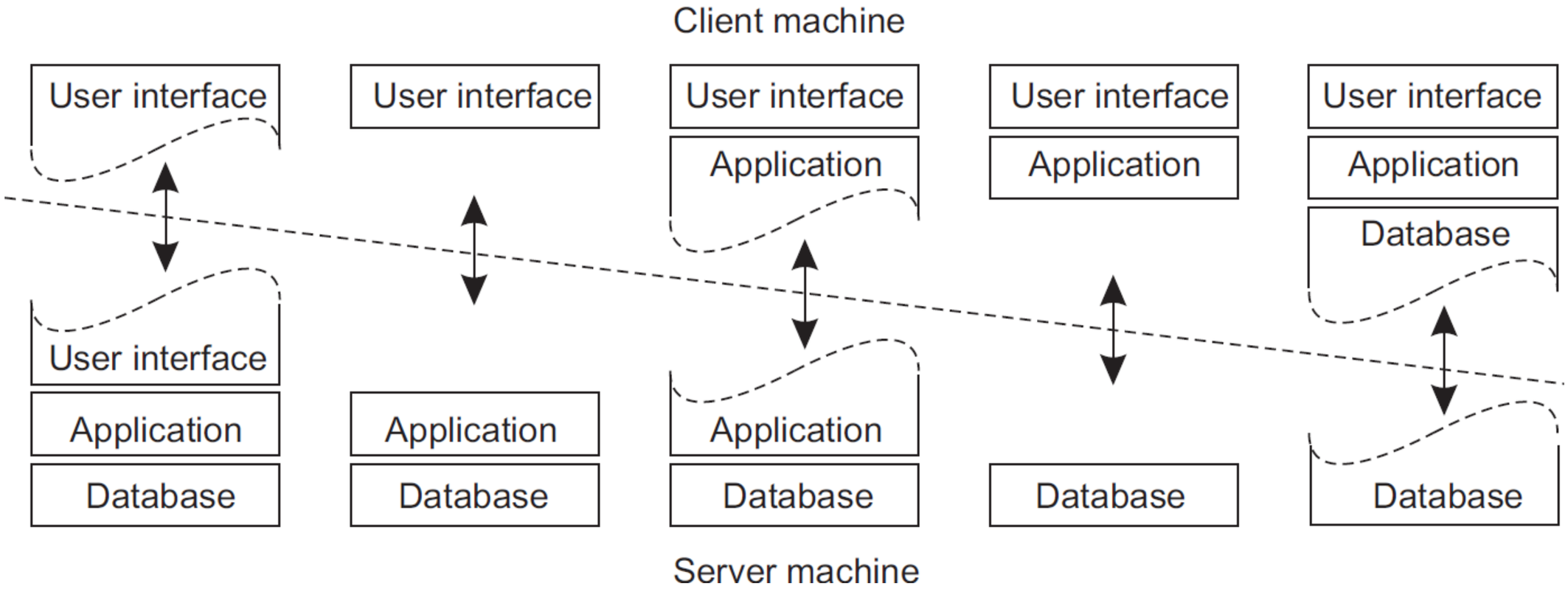
Job request (client)

- send/receive message pair
- request/awaitresult semantics



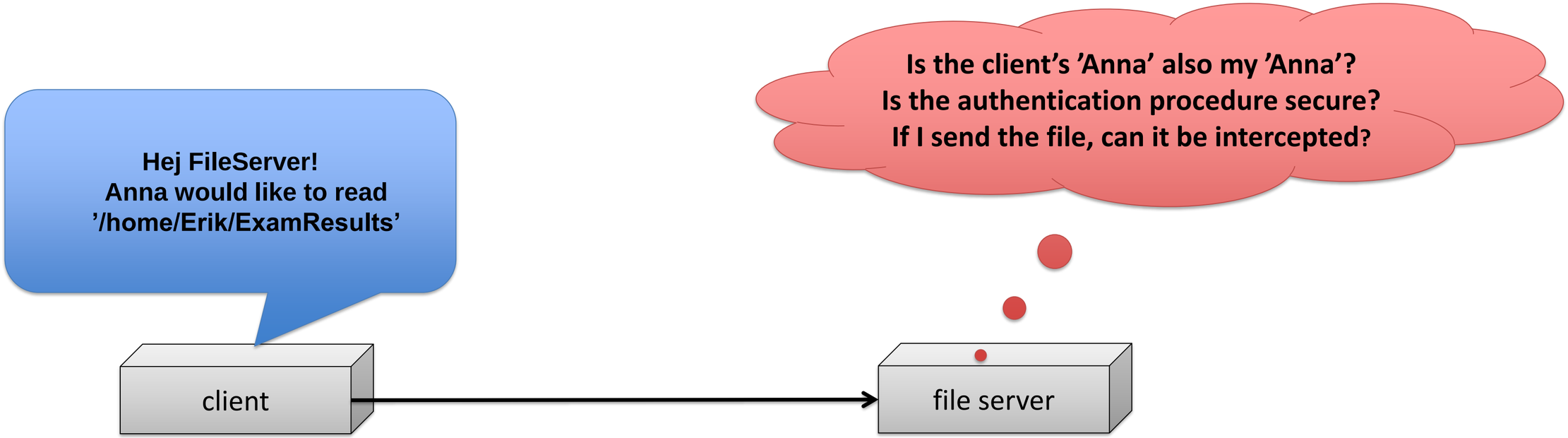
- Job accept (server)
  - receive/send message pair
  - accept/reply semantics

Client - Server



Thin clients vs. Fat clients

Security aspects of Client/Server models



Security aspects of Client/Server models

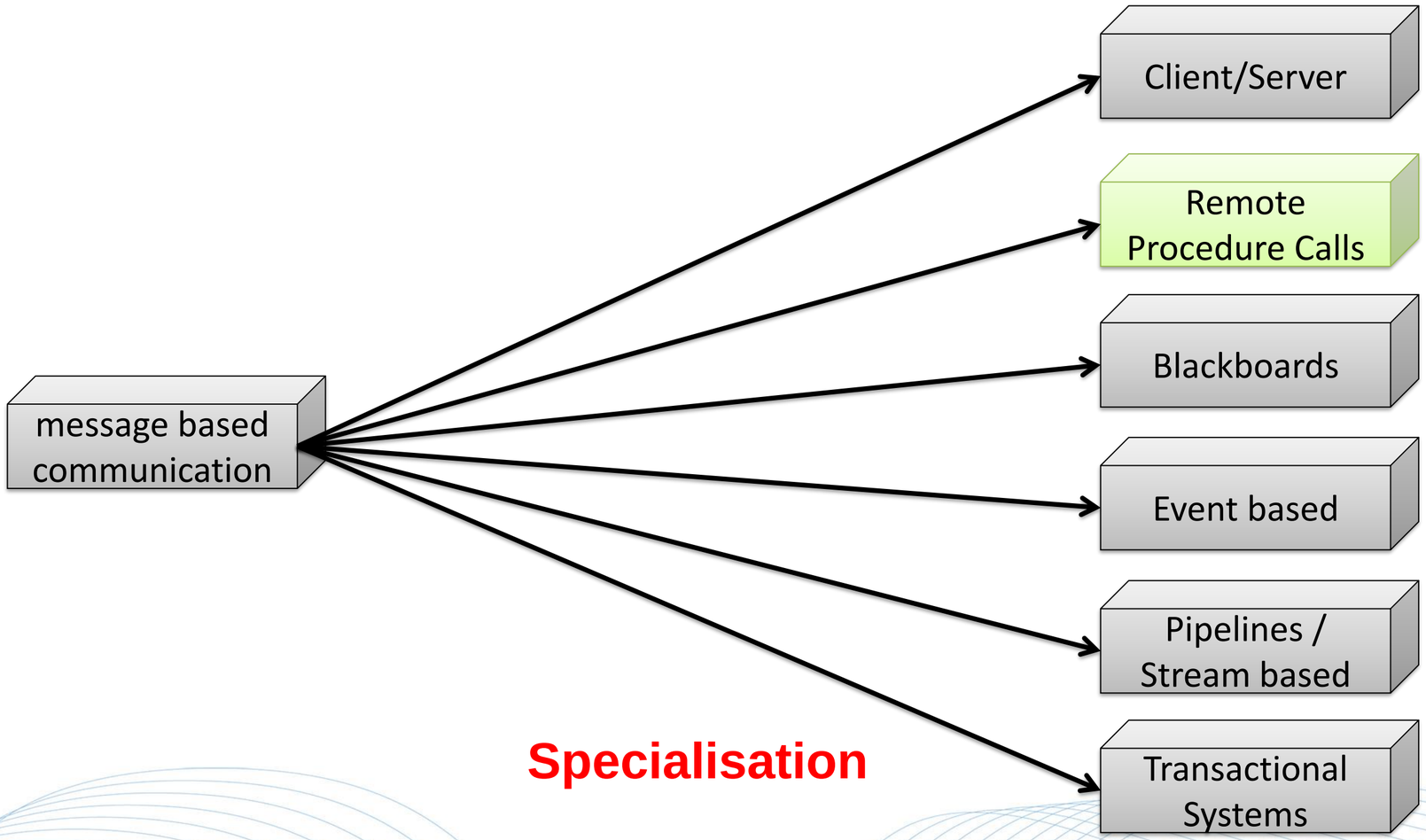
- **Access:**
  server needs to know who is the client → client authentication

- **Integrity & Confidentiality**:
  client needs to know who is the server → server authentification

- **Communication:**
  listing in by third parties, changing or disrupting communication
  → cryptography

➔ Client/Server models require quite some help

Remote Procedure Calls



Idea of Remote Procedure Calls

Usage / Adaptation of a

- application-oriented
- simple and
- wellknown

paradigm to deal with distributed systems

Examples:

Corba RPCs, Java RMIs, gRPC

RPC communication model

Roles: Function caller (Client) / Function excution (Server)

Data: procedure parameters, specified by signatures

Termination: mostly sychronous (blocking) behavior

Failure semantics: *maybe, exactly-once, at-least-once, at-most-once*

Typical RPC applications

Communication between OS components in microkernel OS

Communication between components in distributed applications

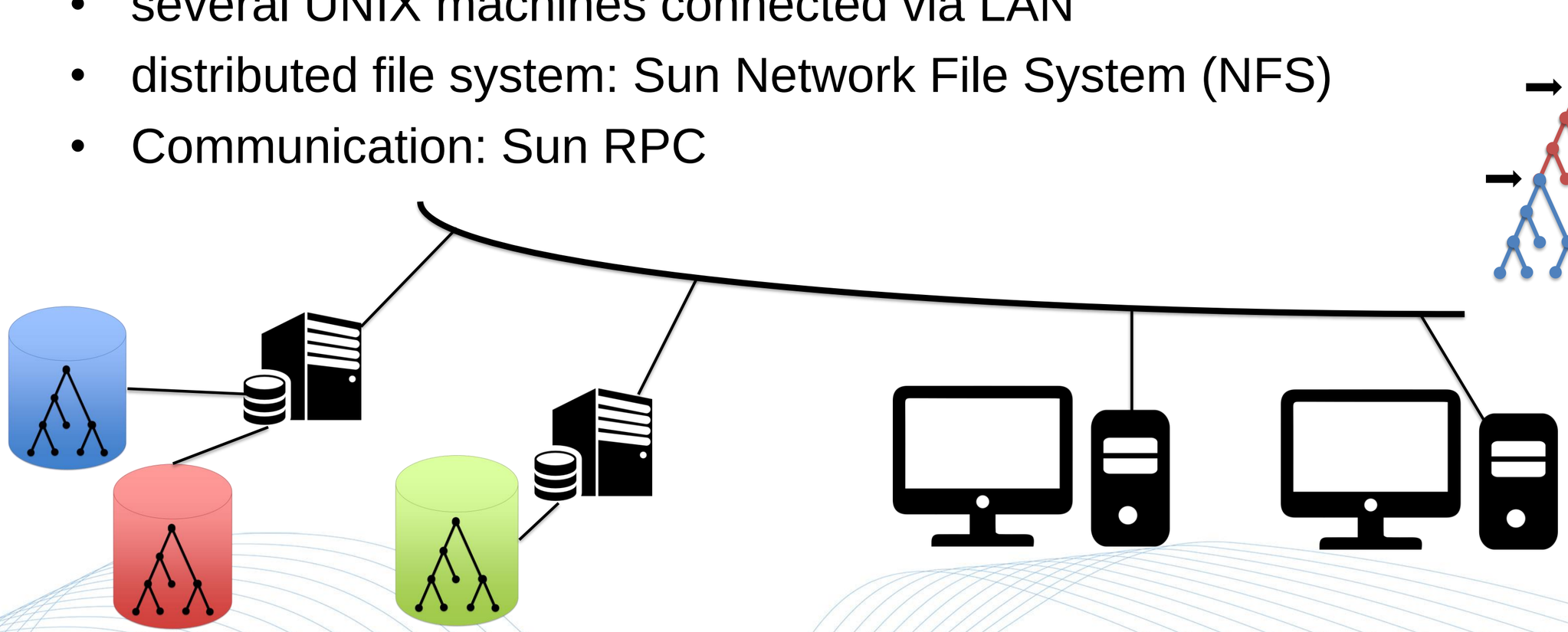
- Finance & Insurance (security infrastructures)
- Telemetry systems (industry, automobile, ...)
- Telecommunications systems (operator)
- Distributed real-time systems

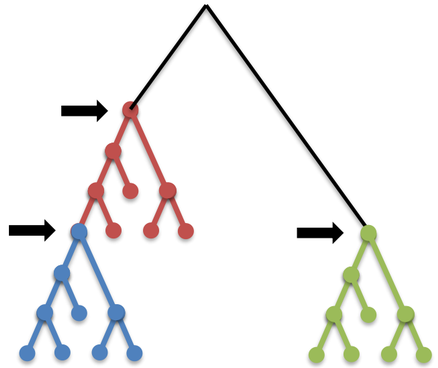
Communication between OS

Example – Network File Server

Setup:

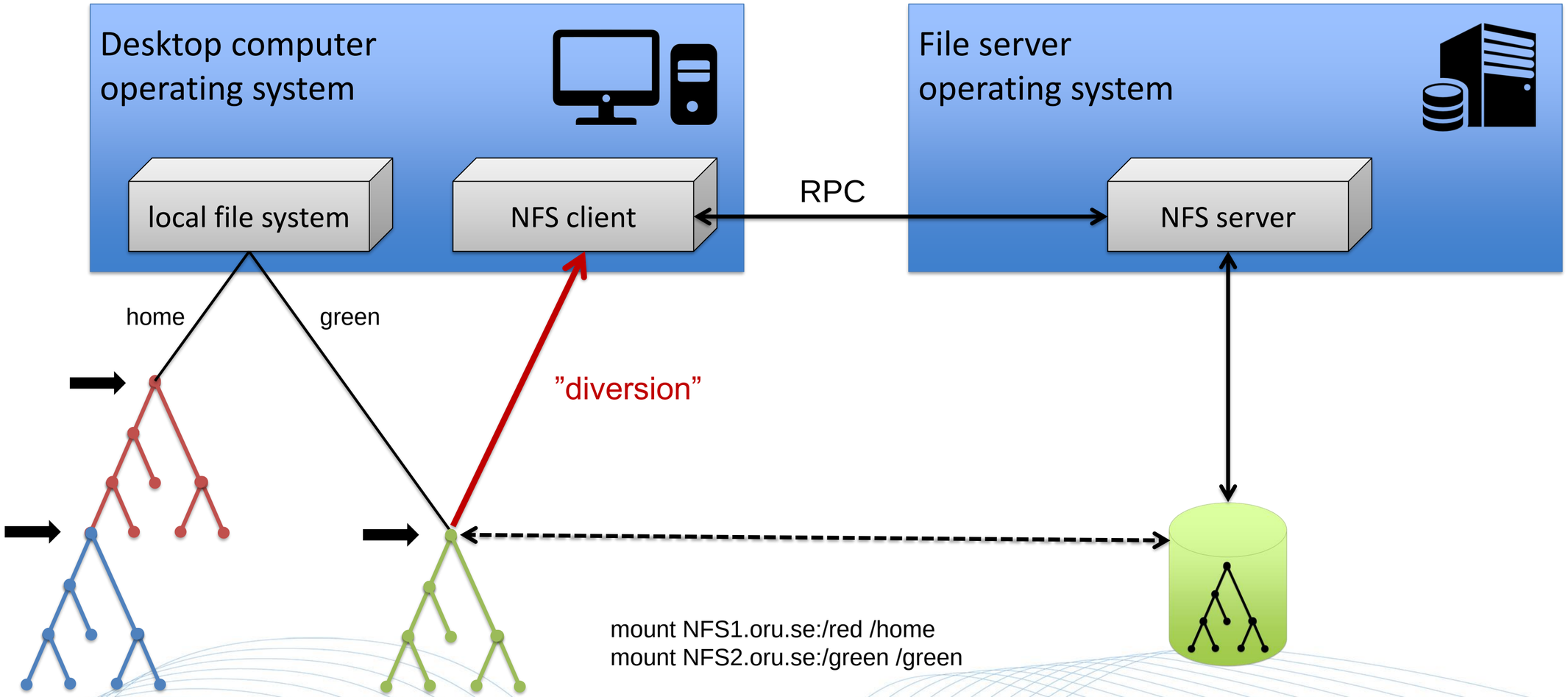
- several UNIX machines connected via LAN
- distributed file system: Sun Network File System (NFS)
- Communication: Sun RPC





➔ Mount points

Example contd – Network File Server

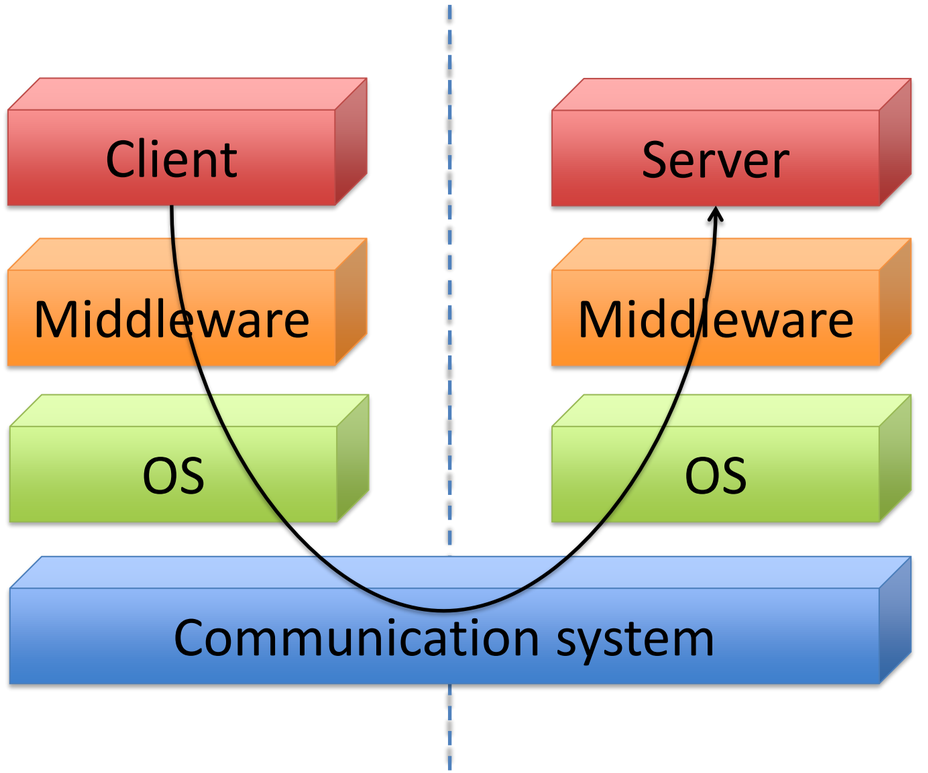


RPC Challenges

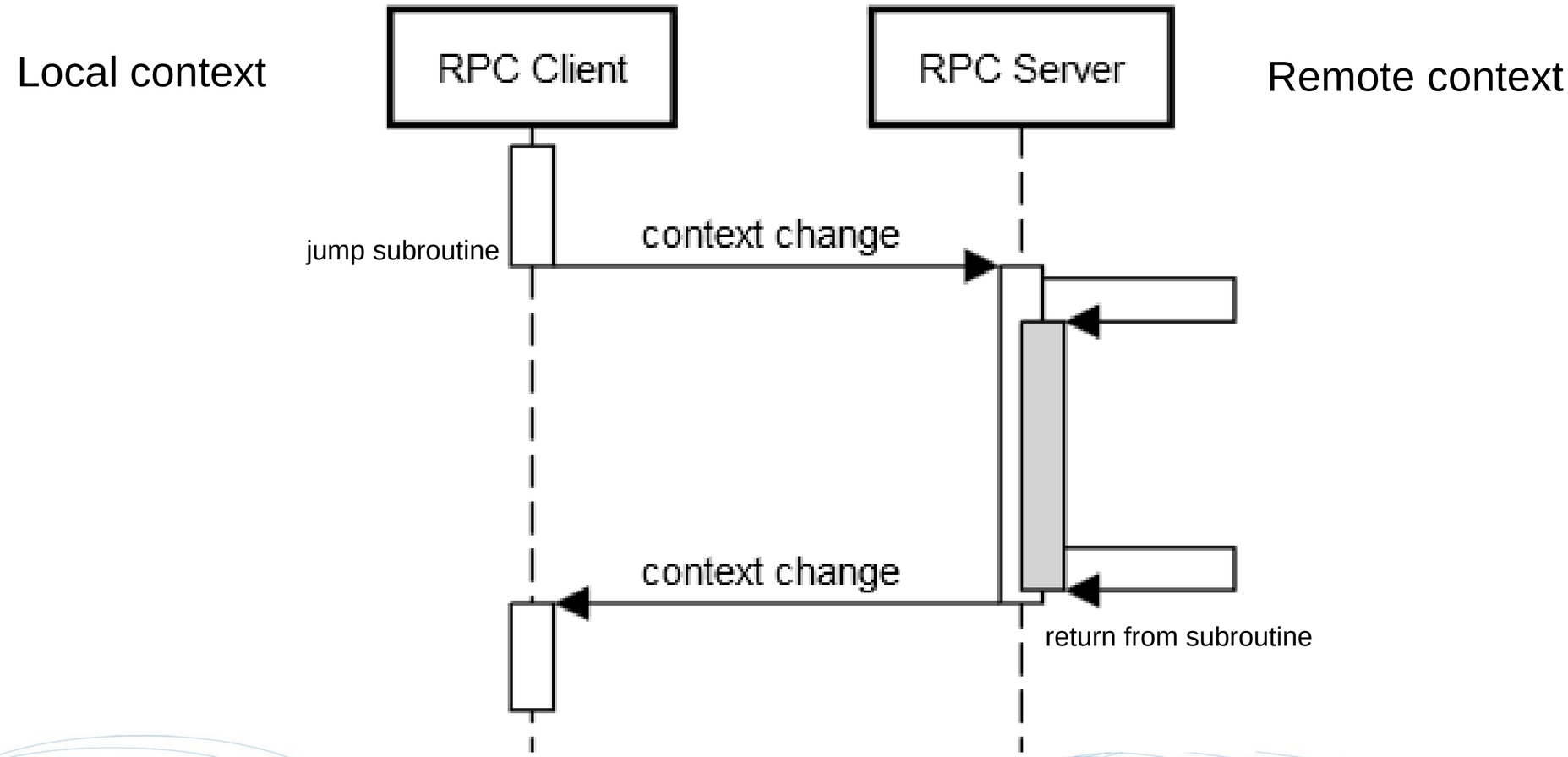
RPC → Generalization of Procedure Calls

Main difference is a change of context:

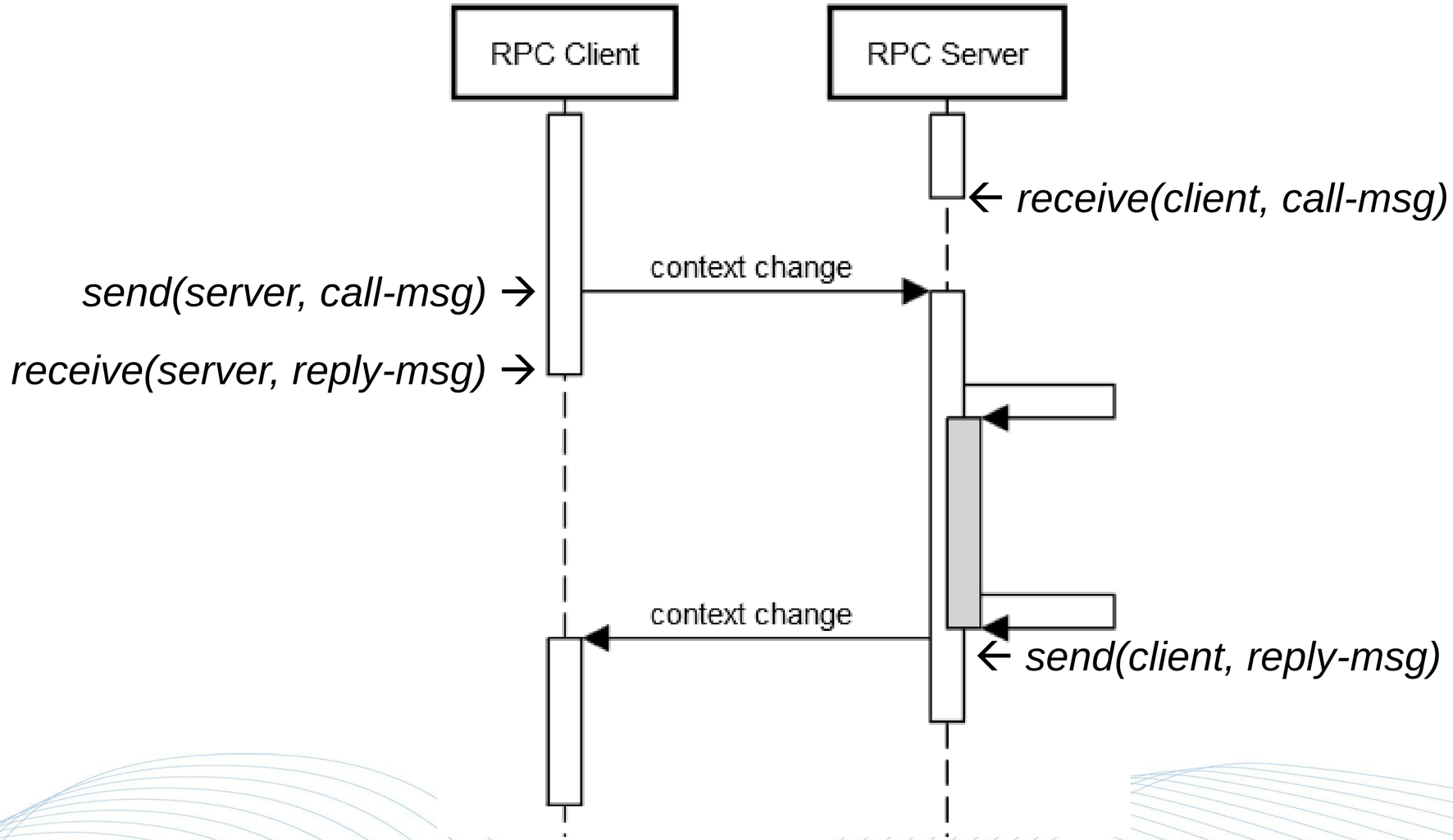
- Programming language context
- Name space context
- Adress space context
- Operating system context
- Hardware context



RPC communication pattern



Implemented as message-based system



Ideal Remote Procedure Call ...

Behavior is the same as a local function call
(in comparison to a request/reply model)

Call transparency

- Type checks of signatures
- No explicit management of the parameter exchange
- No explicit program sequence control

Location transparency

- No explicit adressing of the server (e.g via its ID or DNS)

Failure transparency

- No explicit failure management

7 challenges for the ideal RPC

1. Different programming languages
   - different PAR (Procedure Activation Record) format, stack & data type representations

2. Disjoined adress and name spaces
   - no direct PAR access for parameter handover & problems with global variables

7 challenges for the ideal RPC

3. Disjoined processes
    - no common program counter (instruction pointer)

Starting a procedure call

- explicitly stop caller execution, start procedure execution

Finishing a procedure call

- explicitly stop procedure execution, return caller execution

7 challenges for the ideal RPC

4. Different hardware
   - Different memory layouts of data in PARs
     character encoding, integer & float representations

5. Disjoined hardware (no common failure behavior)
   - Possible failures: communication network, client, server

7 challenges for the ideal RPC

6. Communication bottleneck
   - Performance reduction due to latency
     (e.g. image as parameter, local: pointer of 4 byte,
     RPC: several megabyte)

7. Open communication lines (→ security)
   - Observability of communication
   - Alteration of communication
   - Impediment of communication
   - Identification and authentification of client/server

Wanted: RPC communication model

Roles: Function caller (Client) / Function excution (Server)

Data: procedure parameters, specified by signatures

Termination: sychronous

Partial Failures: do not occur

In the presence of those seven challenges!

Not all challenges are unique to DS

1. Different programming languages (stack, data types)
2. Disjoined adress spaces (global variables, parameter handover)
3. Disjoined processes (no common sequence control)
4. Disjoined hardware (no common failure behavior)
5. Different hardware (no common data type representation)
6. Communication bottleneck (performance, timeliness)
7. Open communication (security)

Same issues for OS calls

Operating System Calls

1. Different programming languages

Solution

➔ Standardization of procedure activation record (PAR) structure and data type representation

- Defined by API specification and implemented through
  - libraries (application layer, e.g. libc)
  - API layer within the operating system

Operating System Calls

2. Different address and name spaces

Solutions

➔ Explicit access to caller's address space

- same physical memory
- operating system priviliges

➔ Proxy procedures with identical interface in a library

➔ Standardization of system call names using enumeration

Operating System Calls

3. Disjoint processes

Solutions

➔ Explicit syncronization (TRAP/RTI), possible due to operating system priviliges

Useful for RPCs

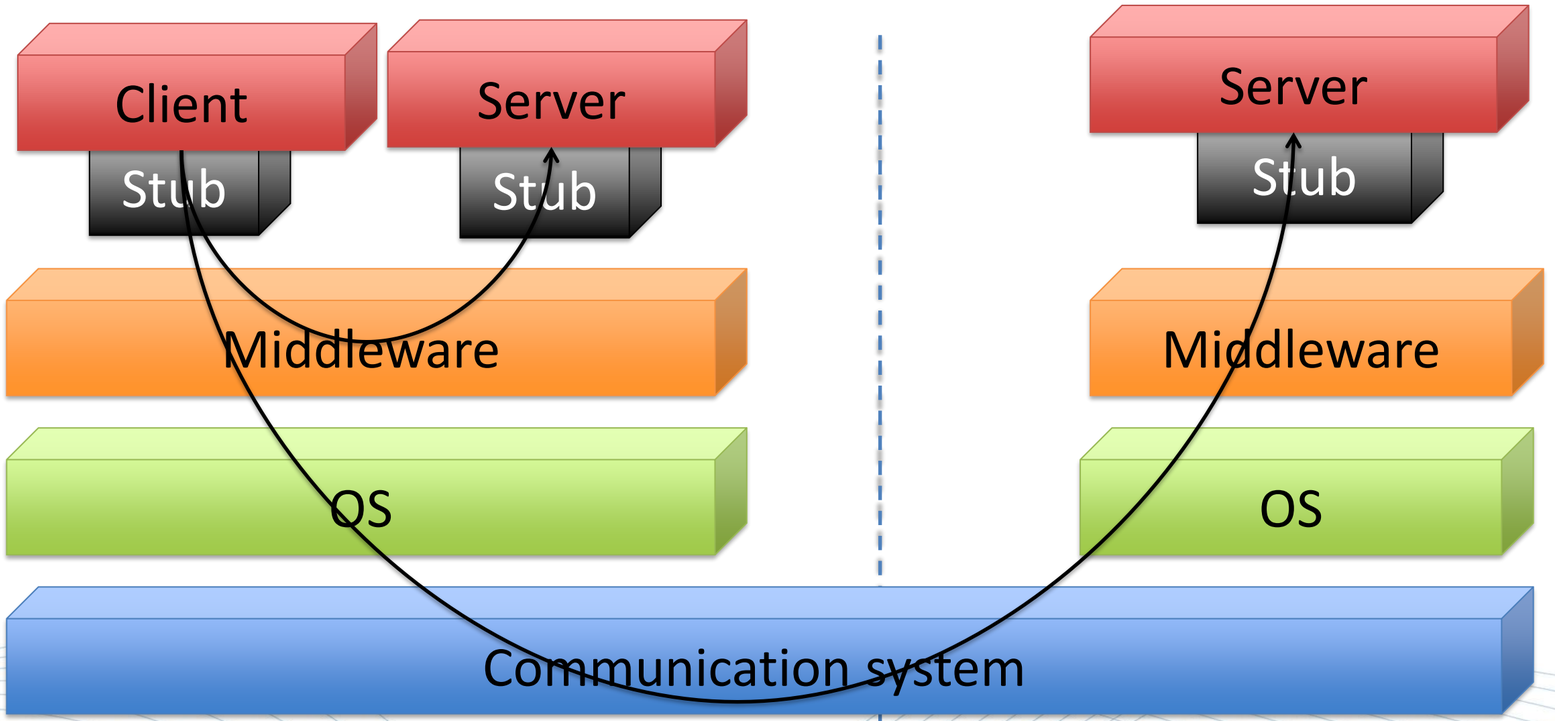
- Different languages and name spaces

Still problematic

- Different address spaces and sequence flow

Stubs

- Proxy of the called procedure on the client side
- Proxy of the caller on the server side



Stub – Client side

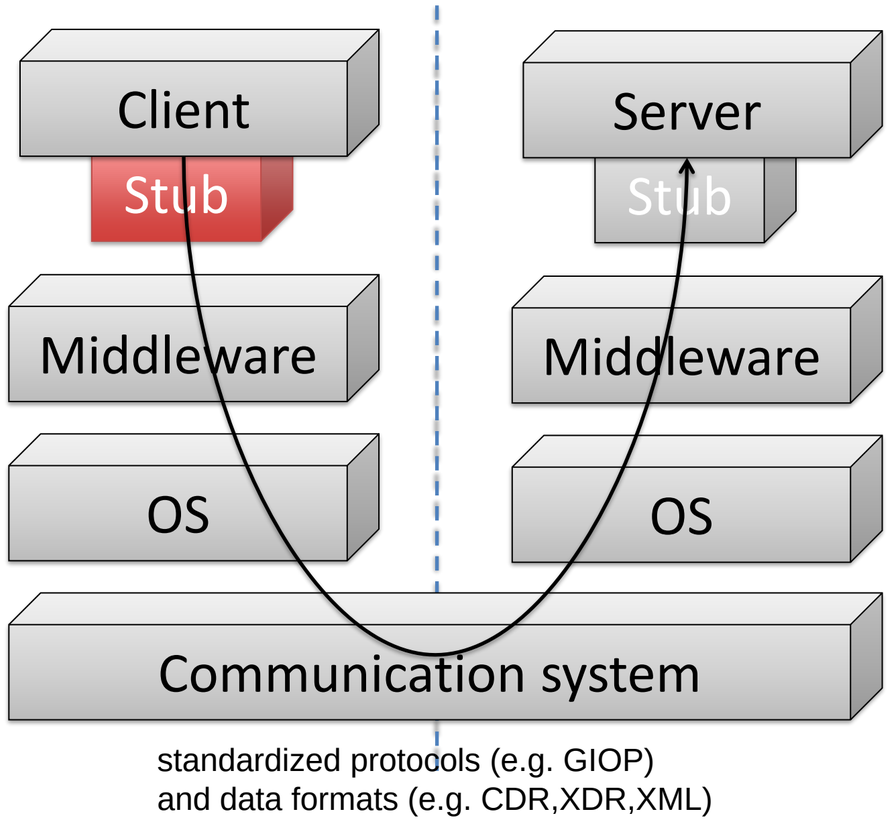
Proxy for the remote procedure

Role: represents the server

Data: provides signature for caller type check

Termination: implement procedure model

Failure semantics: *later*

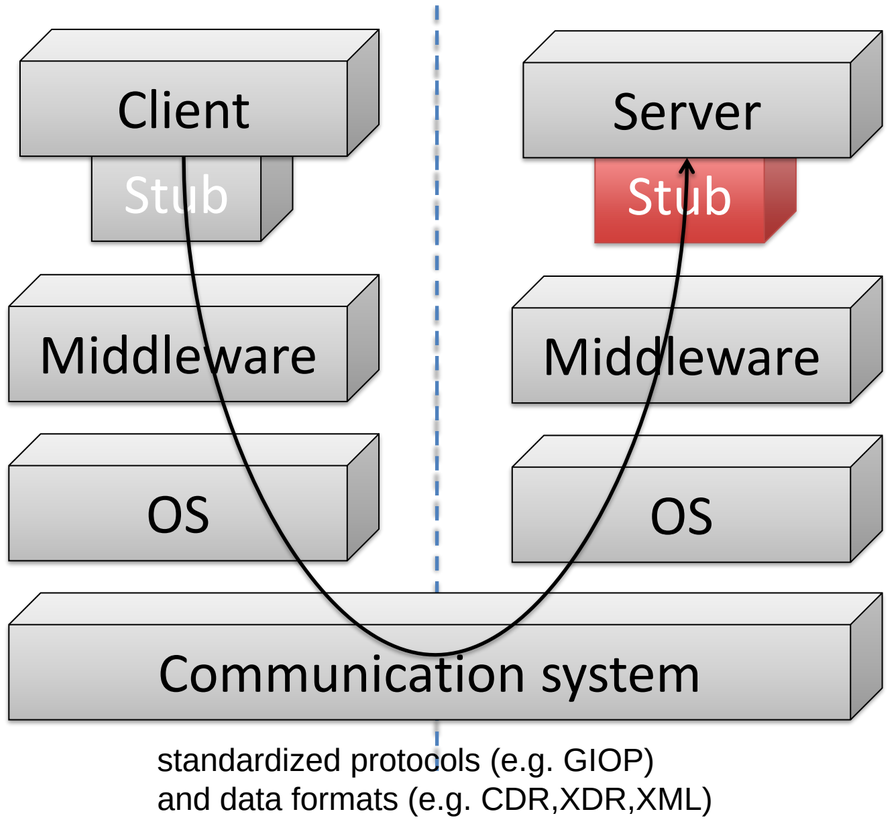


- language specific data type translations
- makes use of OS communication infrastructure (often sockets) to communication with the stub on the server side data type translations

Stub – server side (skeleton)

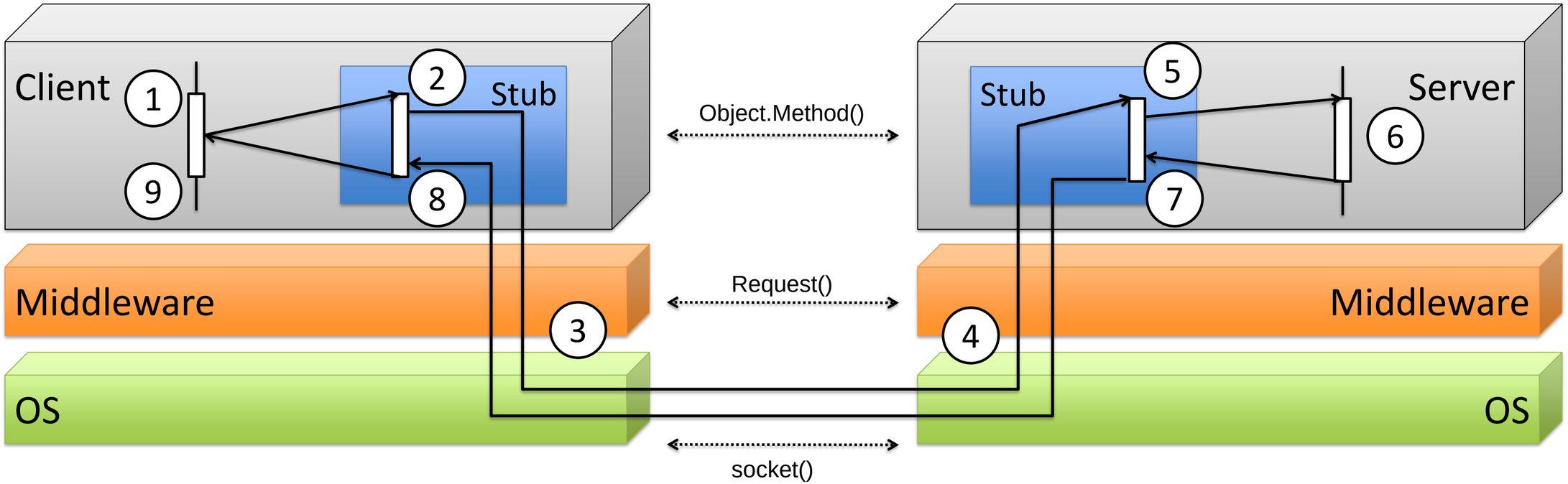
Proxy for the procedure caller

Role: represents the client
Data: generates signature for procedure call
Termination: implement procedure model
Failure semantics: *later*



- language specific data type translations
- receives messages from the communication system
- calls the server side proecdure

RPC example



Stub generation

➔ Compiler needs to allow this concept

Automatical generation requires:

- Interface definition (specific languages availiable, e.g. IDL)
- Knowledge of the implementation language of the client/server
- Communciation model of the middleware platform including localization capabilities

Back to the challenges

4. Different hardware

Problem

Computer architecture specific representation of data

- characters and strings encoding (ASCII, UTF-8, ISO-8859-15, EBCDIC, etc.)
- Float number representation (length and order of mantissa and exponent)
- Integers (little endian vs. big endian, 1's vs 2's complement, 16/32/64 bit)

Back to the challenges

4. Different hardware

Solutions

➔ Standardization of data type representation within messages (CDR, XDR,XML – but inefficient double convertings possible)

➔ Include datatype description in each message (larger messages, more costly de-marshalling)

Implementation

Inside the stubs as part of the parameter (de-)marshalling

Partial failures

5. Disjoined hardware

Goal

Maintain the RPC = LPC illusion in the presence of communication failures and node failures

Problem

- Multiple components with seperate hardware
  → partial and independent failures
- Highly distributed systems, changing availibility, heterogenity → failures are commonplace

Failure models

1. ***Fail-stop* failures:**
   - **Nodes:** failed nodes are silent and remain inoperable
   - **Communication:** messages are lost
2. ***Fail-stop-return* failures:**
   - **Nodes:** failed nodes are silent, but might resume operation
   - **Communication:** messages might be delayed
3. ***Byzantine* failures:**
   - **Nodes:** failed nodes behave chaotic or malicious
   - **Communication:** message might be lost, delayed or changed

Partial failures

Failure transparency is a **consensus** problem

➔ Client and server have to agree in the presence of failures on whether a procedure was or was not executed

> Consensus problems in distributed systems are only solvable, if the number of failures is limited.

Limited amount of failures

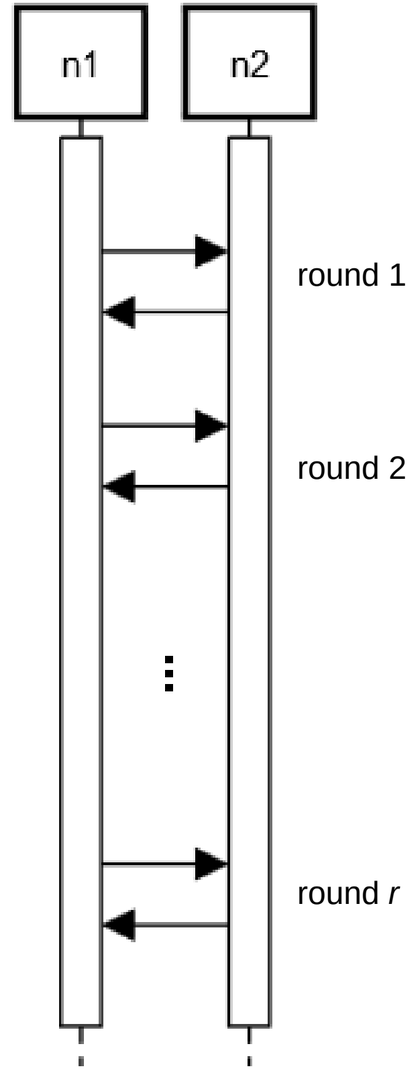
Assumption: Algorithm *a* creates consensus between node n_1 and n_2 after *r* rounds of exchanging message pairs

Messages get lost in round *r*

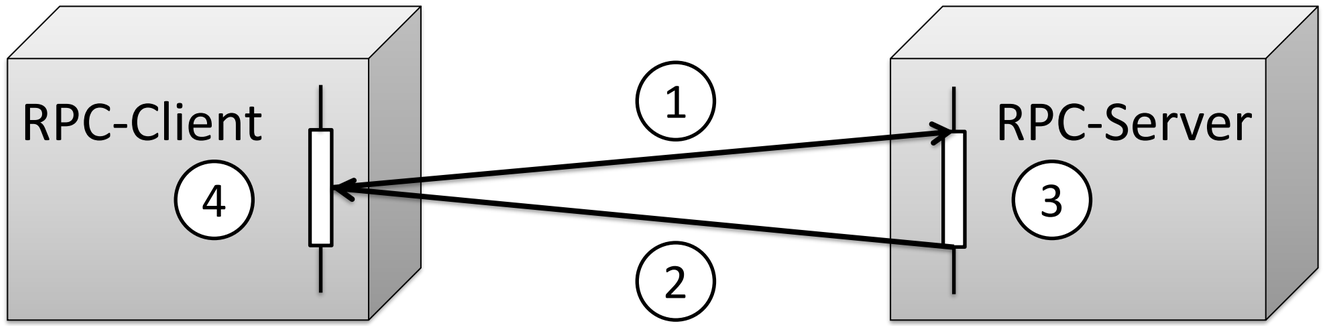
→ Algorithm required that creates consensus in *r-1* rounds

Messages get lost in round *r = 1*

→ No communication, no consensus



Four failure scenarios for RPCs



Failure of the communication system

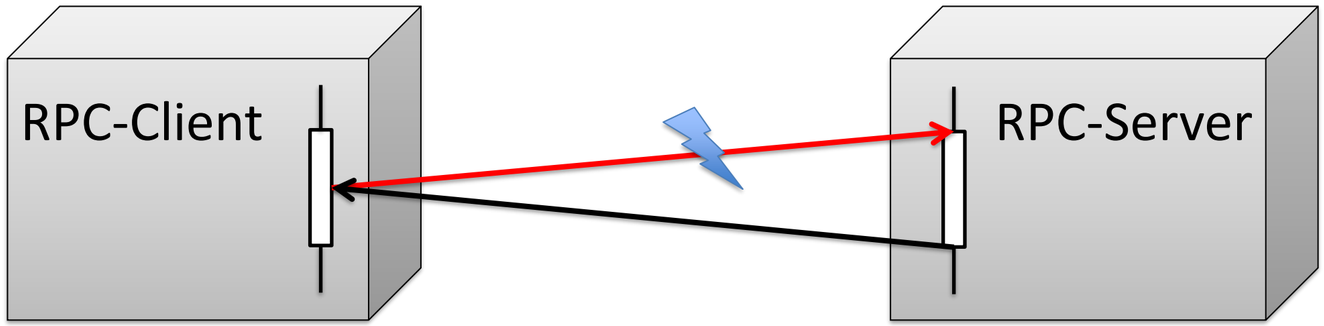
1. Loss (fail-stop) of RPC request message
2. Loss (fail-stop) of RPC reply message

Failure of nodes

3. Server (fail-stop) failure
4. Client (fail-stop) failure

Reading Task:
JMP - Chapter 8.3 of the course book!

1. Loss of RPC requests

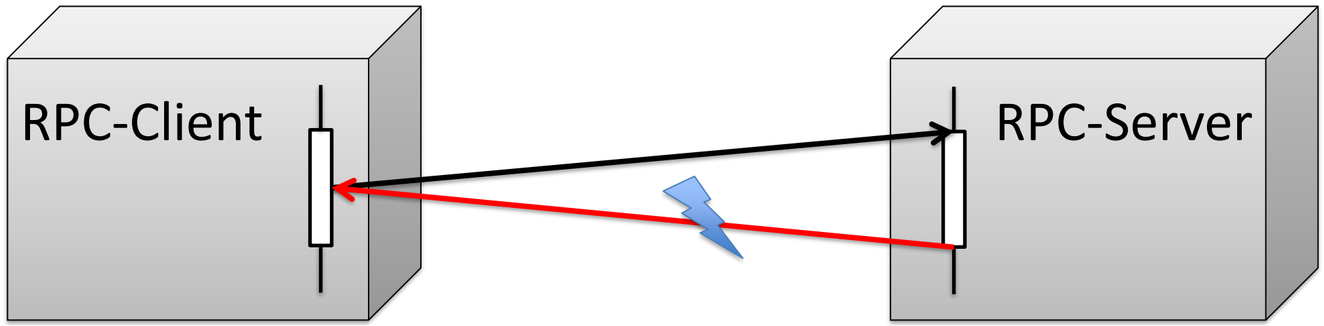


Symptom: server does not answer

Effect: server does not execute procedure

Remedy: resend the RPC request after a time out

2. Loss of RPC replies



Symptom: server does not answer (looks the same for client)

Effect: server does execute procedure (different from 1.)

Remedy: resend the RPC request after a time out
→ repeated call of the same procedure

Idea 1: Idempotent server operations

> A idempotent procedure generates the same effect independently of the number of times it is executed.

→ Simple failure handling: Client can repeat RPC arbitrary often

Examples

- reading an account balance
- deleting a file

But not a general solution

- charging an account
- attaching data to files

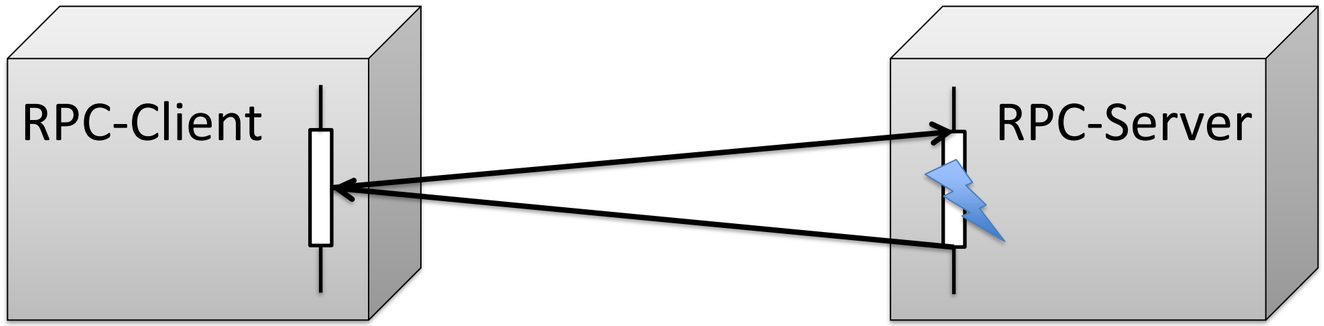
Idea 2: Detection of repetitions

Brute-force approach (costly, but simple)

- Client enumerates RPC requests
  → repetition gets same number

- Server checks sequence numbers and filters repetitions & resends results
  → server needs to save sequence numbers and results

3. Server failure



Symptom: server does not answer (again the same)

Effect:

1) server does not execute

2) server does execute partially

3) server does execute completely

Fail-stop here

```
forever do {
    receive(client, rpcRequest)
    call(rpcRequest.op,
         rpcRequest.params,
         rpcReply);
    send(client, rpcReply);
}
```

Reading Task:
Pay attention to Note 8.9!

3. Server failure

Remedy: fail-stop model → server is gone for ever

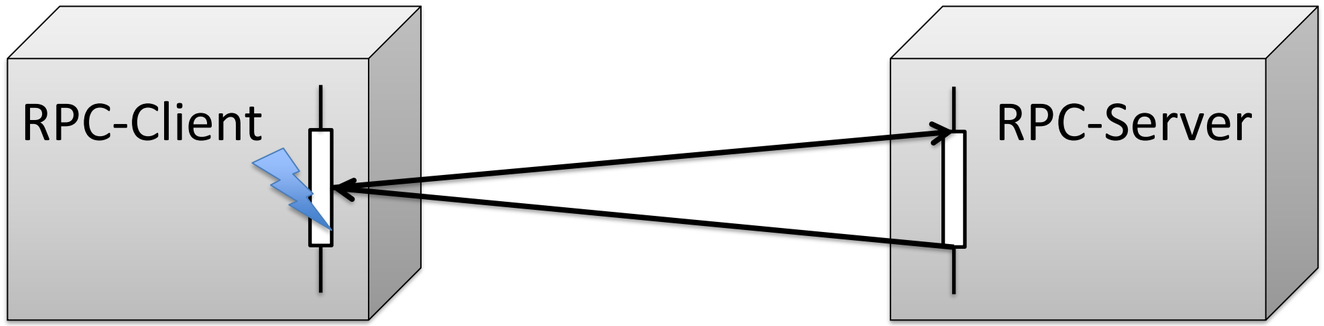
fail-stop-return model → it is getting complicated

Server behavior after return to service needs to address server state at the time of failure

Server requires a persistent log of all operations

Recovery procedure to restart all not completed RPC operations

4. Client failure



Symptom: ...

Effect: nobody is waiting/receiving the results

→ orphaned procedures

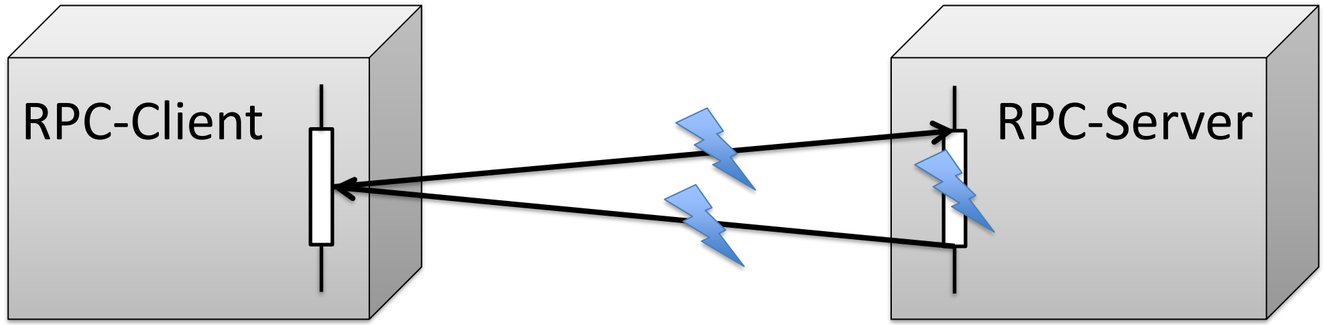
- allocated resources and potential locks on the server
- might interfere with restarting clients (fail-stop-return)

4. Client failure

Remedy:

- Ignoring the issue
- **Orphan extermination:** persistent client logs for RPC activities
  → recovery procedure (e.g. sending *kill* messages)
- **Reincarnation:** current epoch counter is increased and sent to servers with each system restart
  → server terminate calculations of older epochs
- **Expiration:** no RPCs from a restarted client until all orphans termininated (finished computation, or timed out)

Failure semantics



1. Maybe

- no guarantees
- simple and fast implementation

Failure semantics

2. Exactly-Once

- in case of sucessful reply: exactly one execution
- in case of a failure: not executed

→ RPC = LPC

→ costly (all-or-nothing property of transactions)

→ not achievable (consensus problem)

Requirements

- known upper limit of failures
- expensive transaction management (consensus algorithms)

Failure semantics

3. At-least-once

- in case of sucessful reply: at least one execution
- in case of a failure: no guarantees (no execution, partial execution, one or more executions)

→ Simple failure reaction: resend RPC request

Requirements

- idempotent server operation

Failure semantics

4. At-most-once

- in case of sucessful reply: exactly one execution
- in case of a failure: no multiple executions (no execution, partial execution or one execution)

→ More complex failure reaction: resend with identical ID

Implementation

- Client: manage RPC IDs
- Server: duplicate call filtering + storage of results

Communication bottleneck

Conflict between **convenience** and **efficiency**

- location transparency
- access transparency
- failure transparency

- performance (communication bandwidth and latency)
- real-time capabilities (latency fluctuations)

Communication bottleneck

Communication costs considerations during **system design**

Limiting the used bandwidth and latency effects

- object design, interface design
  - RPC granularity (many small RPCs vs. one huge RPC) and object granularity (amount of object interactions)
  - parameter volume
- ’close’ co-location of communication intensive objects
  - static placements vs. dynamic placements (e.g. gravitation models with dynamic objects)

Communication bottleneck

Communication costs considerations during **implementation**

- exploiting locality: distinction in stubs between
  - local communication (using shared memory)
  - remote communication (using messages)
- problem specific communication models
- consideration of technical infrastructure (latency, bandwidth, reliability, MTU size)
- use of professional implementations

IT security

All aspects of the client/server model are equally relevant

- mutual authentification of calling and executing instance
- authorisation of RPCs
- observing, changing or disrupting communication by third parties

Payday

Remote Procedure Calls are not for free

- stub costs: marshalling, failure semantics, program sequence control
- communication management: localization of objects
- message system: buffering, copying and transport of messages, synchronisation
- IT security

RPC Conclusion

RPC = LPC

- Location and access transparency
- Heterogenity
- Sequence control: synchronous messages

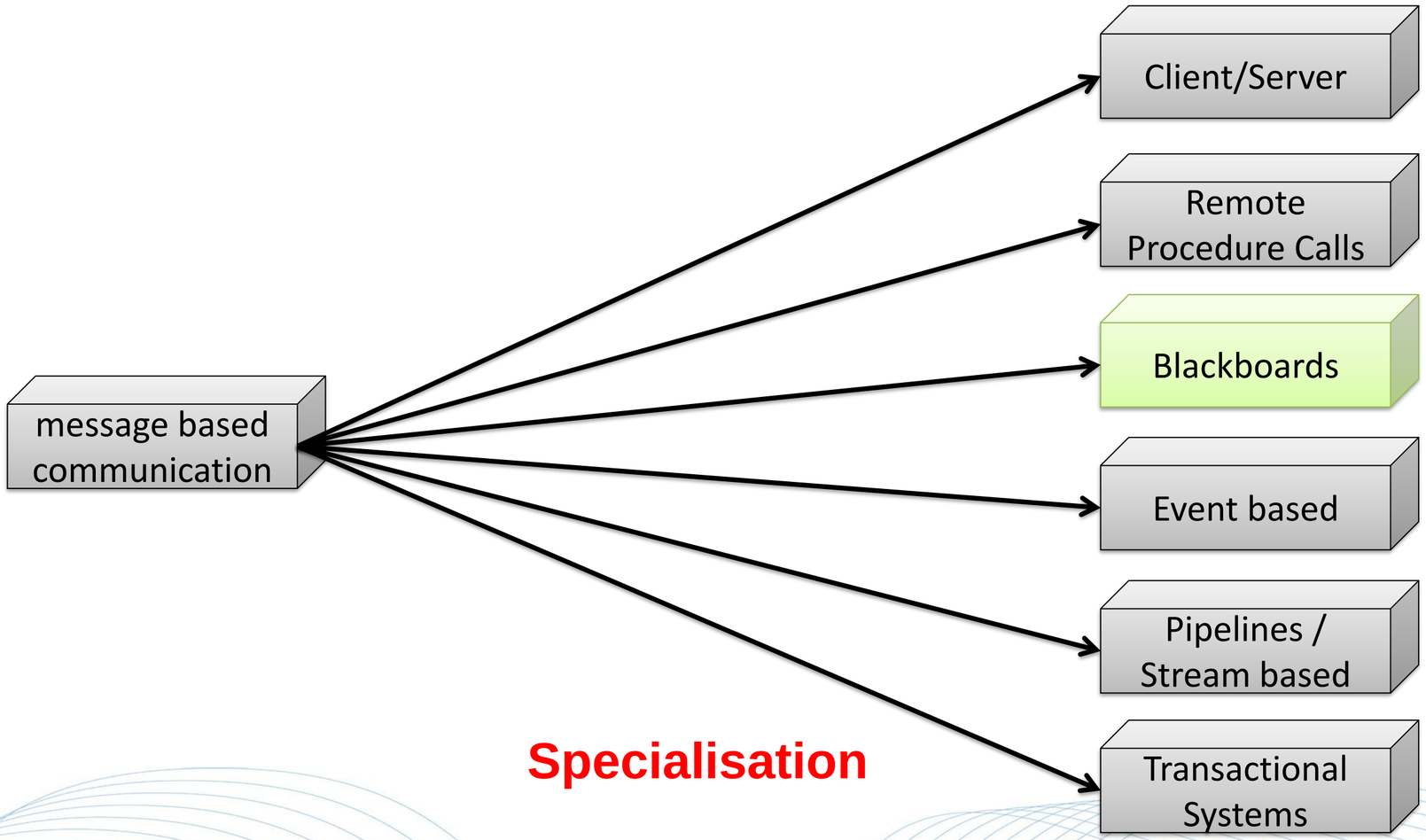
RPC != LPC

- IT security (typically not addressed in stubs)
- failure transparency (typically not addressed in stubs)
- performance

} application specific

Reading Task:
RPCs with examples in chapter 4.2, take particular note of the parameter passing discussion.

Blackboards



Blackboards

Blackboard model is designed for the cooperation of **autonomous** and **anonymous** components

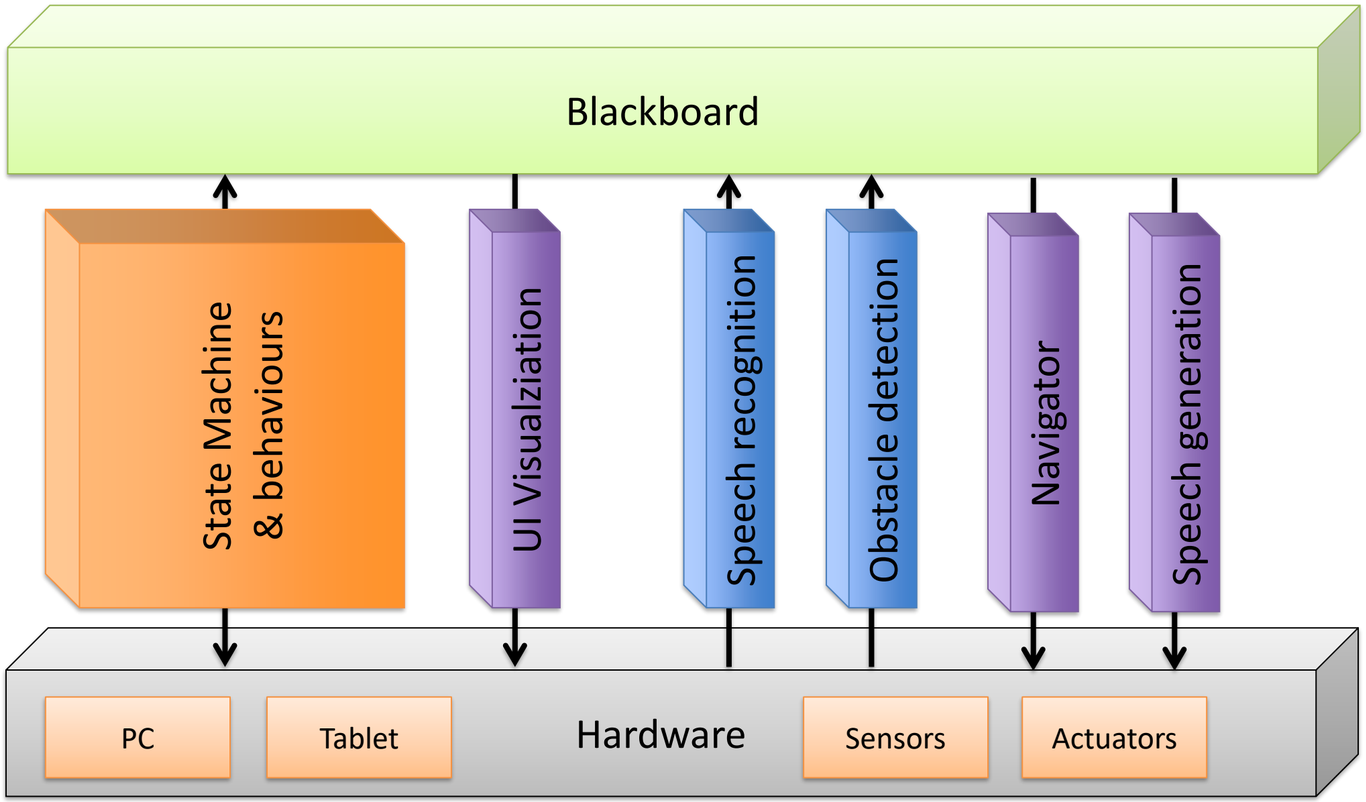
→ spatial and temporal decoupling

Idea

communication by posting messages via a virtual, global memory

(bulletin board, *anslagstavla*, schwarzes Brett)

Blackboard example



Blackboard role schema

Specialist

- autonomous blackboard user
- specialist for a concrete task
  (e.g. speech recognition, navigator)
- communicates with other specialist exclusively via the blackboard

Blackboard role schema

Moderator (optional)

- control component
- coordinates the specialist's work and/or delegates work to specialists
- observes the blackboard
  (e.g. for garbage collection)

Blackboard data model

- central persistent memory for all communication
- at a know location
- problem-specific data;
  in general: tuples <`type, name, value`> ➔ tuple space
- Methods for creating, reading, updating and deleting of tuples

```
<tuple>
  <type>SensorData</type>
  <name>LIDAR</name>
  <data>
     <array>
       <value>0</value>
       <value>12</value>
       <value>44</value>
     </array>
  </data>
</tuple>
```

```
<SensorData, LIDAR, [0 12 44 ...]>
```

Blackboard communication model

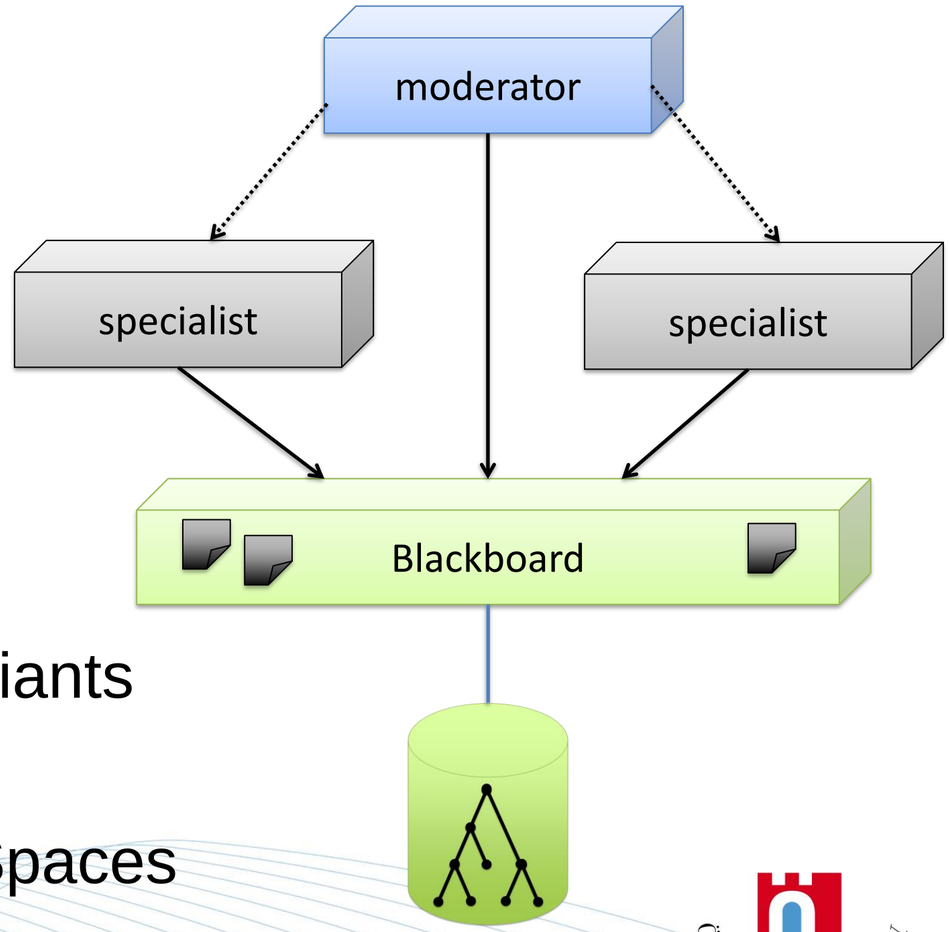
Roles: Specialist [/ Moderator] / Blackboard

Data: tuples

Termination: mostly asynchronous behavior

Failure semantics: reliable and unreliable variants

***Implementations**:* Linda Tuple Spaces, JavaSpaces



Blackboards

Pros

- Openness
  - functional scaling by adding new specialists
- Size Scalability
  - dynamic amount of specialists
- Interoperability
  - joint tuple definition
- IT security
  - anonymous, communication control by the board, isolation of specialists

Blackboards

Pros

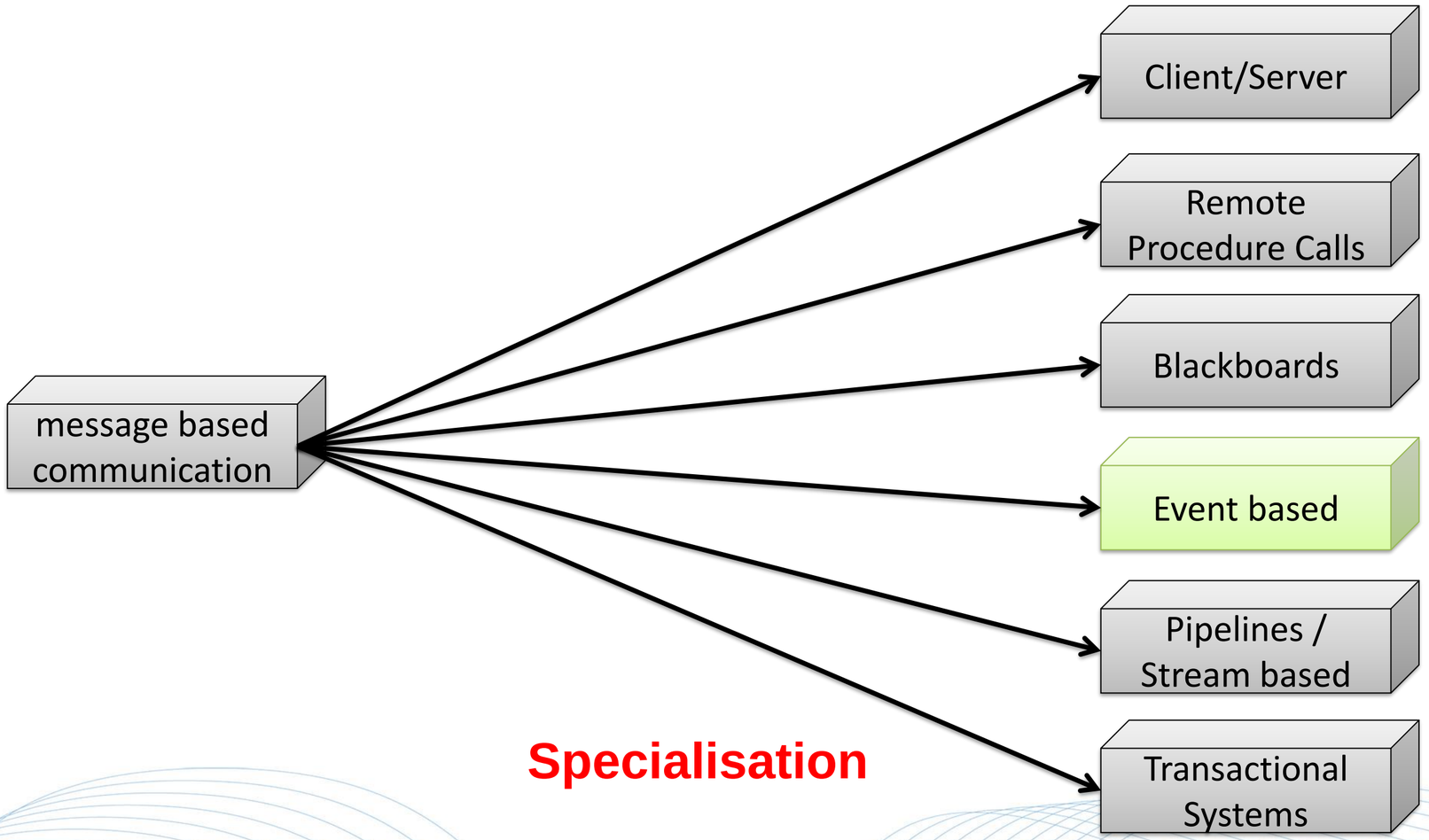
- Failure transparency
  - redundant specialists, isolation of specialists
- Efficiency
  - Specialists perform parallel computations
- Reusability
  - seperation of specific algorithms, data structures and coordination

Blackboards

Cons

- Efficiency
  - board and moderator are possible bottlenecks
  - sychronization of write operations on the board
- Availibility
  - board and moderator are single-point-of-failures
- Design & Implementation
  - Often only empirical moderator strategies
  - complex interactions, asychronism, non-determinism
    → hard to design test cases

Event based models



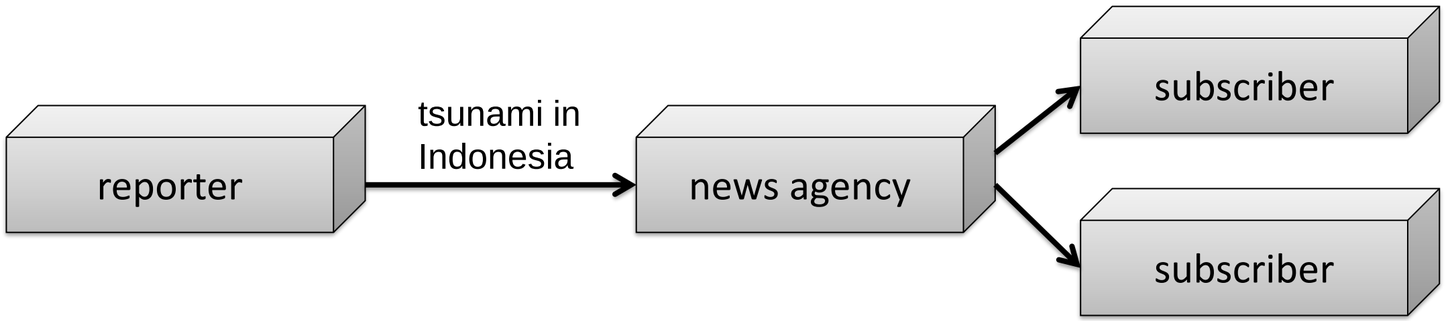
Event based models

Similar to blackboards cooperation between autonomous and anonymous components (→spatial and temporal decoupling)

Additionally **asychronous change notifications**

Application scenarios

- Trading systems
- News agencies
- Survaillance systems
- Flexible replication systems



Role schema

Publisher

- Registers subscribers and their interests
- Sends message to subscribers in case of events
- Advertises offered event repertoire

Subscriber

- Subscribes to event messages from publishers
- Receives event messages

Role schema

Event Service / Agency (optional)

- Subscription management
- Anonymisation
- Event mediation / service directory
- spatial and temporal decoupling between subscribers and publishers, e.g. buffering of events for inactive subscribers

➔ Kind of the blackboard of this communication model

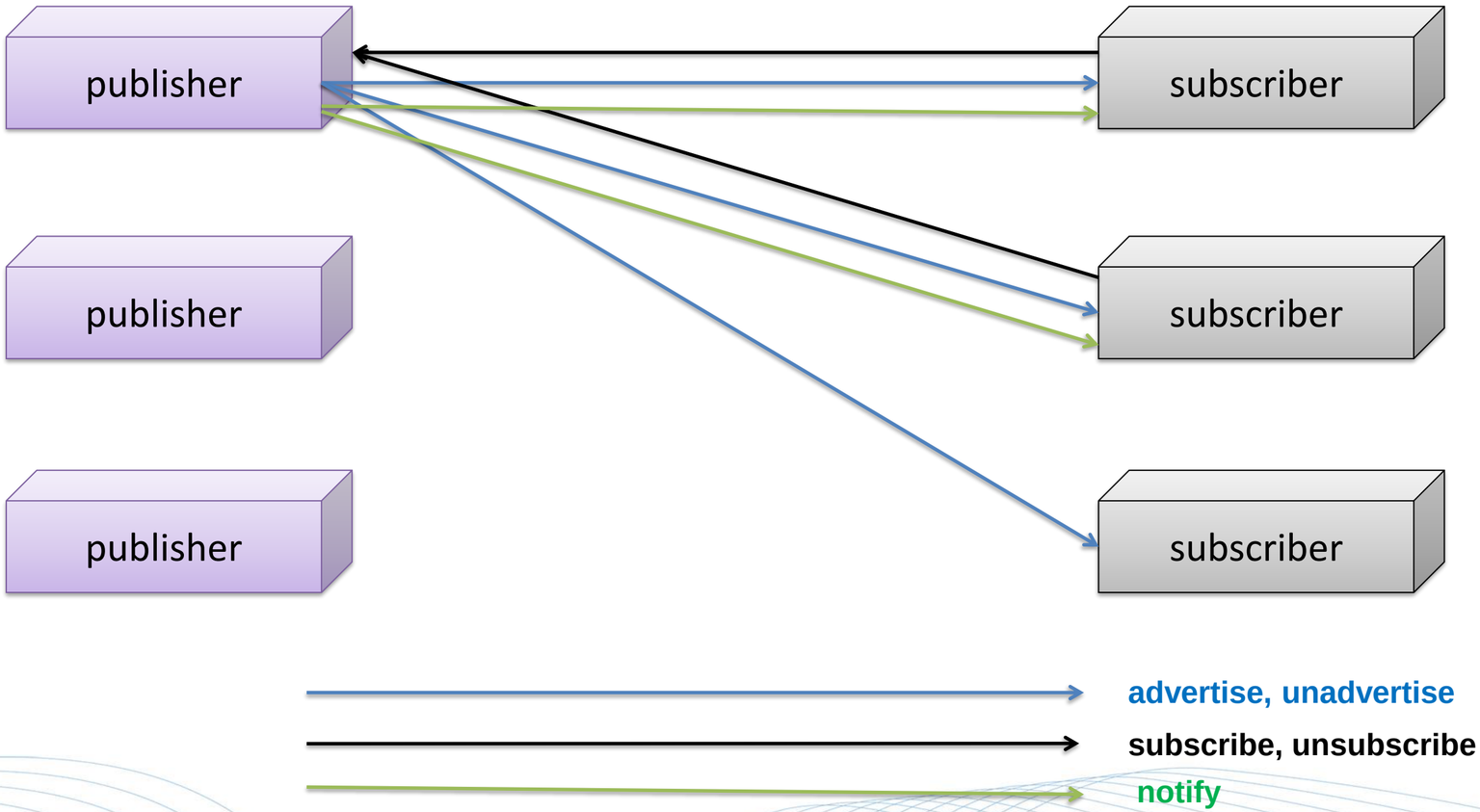
Data model

problem-specific data:
  in general: tuples <`event type, name, value`>

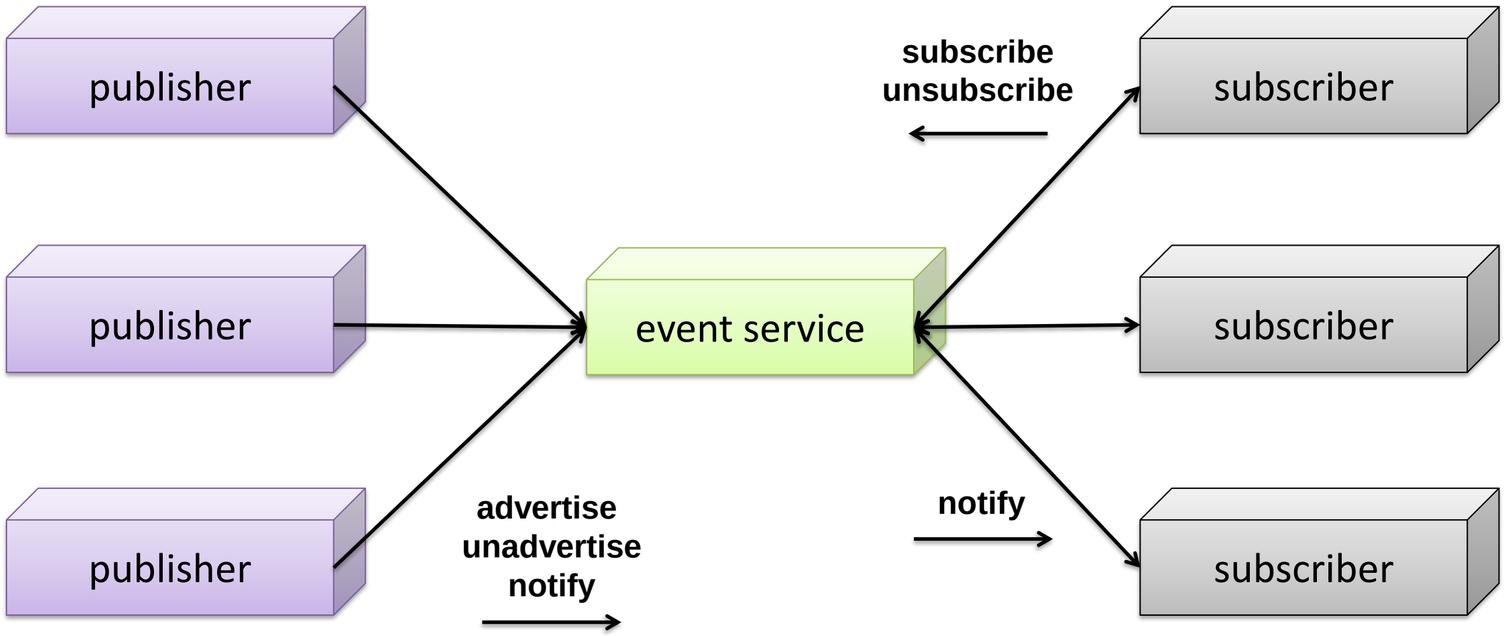
Methods for

- Reporting events (`notify`)
- Subscribing to event messages (`subscribe, unsubscribe`)
- Announcing event repertoire (`advertise, unadvertise`)

Sample architecture without event service



Sample architecture with event service



Failure & Termination Semantics

Roles: Subscriber / Publisher / Event Service

Data: tuples

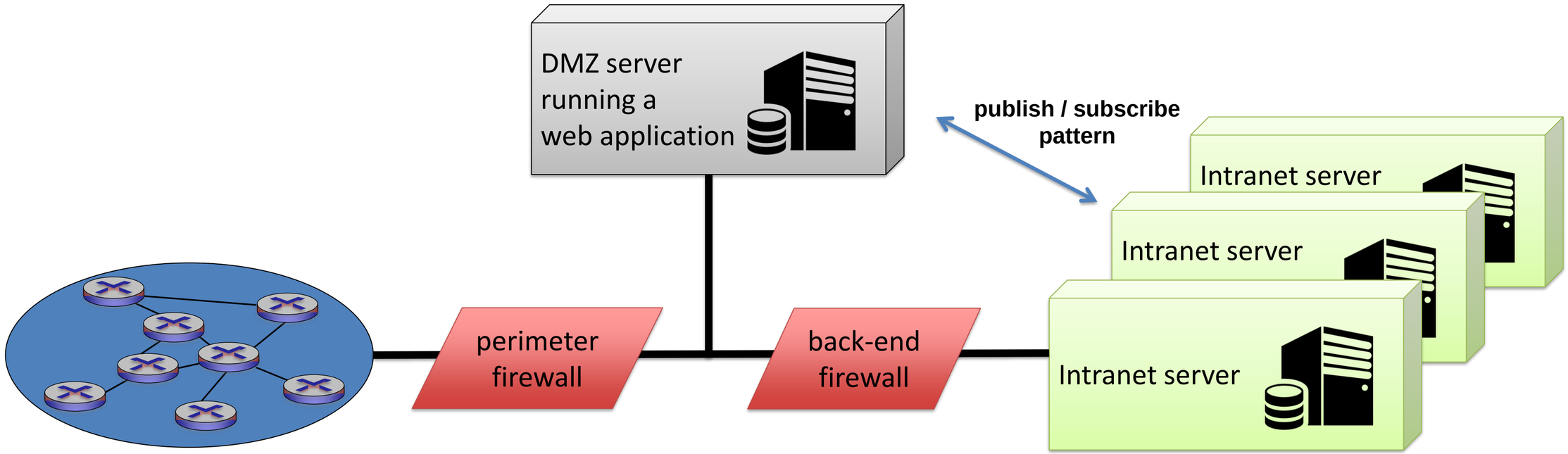
Termination:

- (un-)subscribe/(un-)advertise: in general sychronous
- notify: in general asychronous

Failures:

- (un-)subscribe/(un-)advertise: reliable
- notify: unreliable

Example: Replications



Event based systems

Pros

- Openness, interoperability, size scalabilty, efficiency
  - Same as for the blackboard model
- IT security, reusabilty
  - Same as blackboard for the event service variant

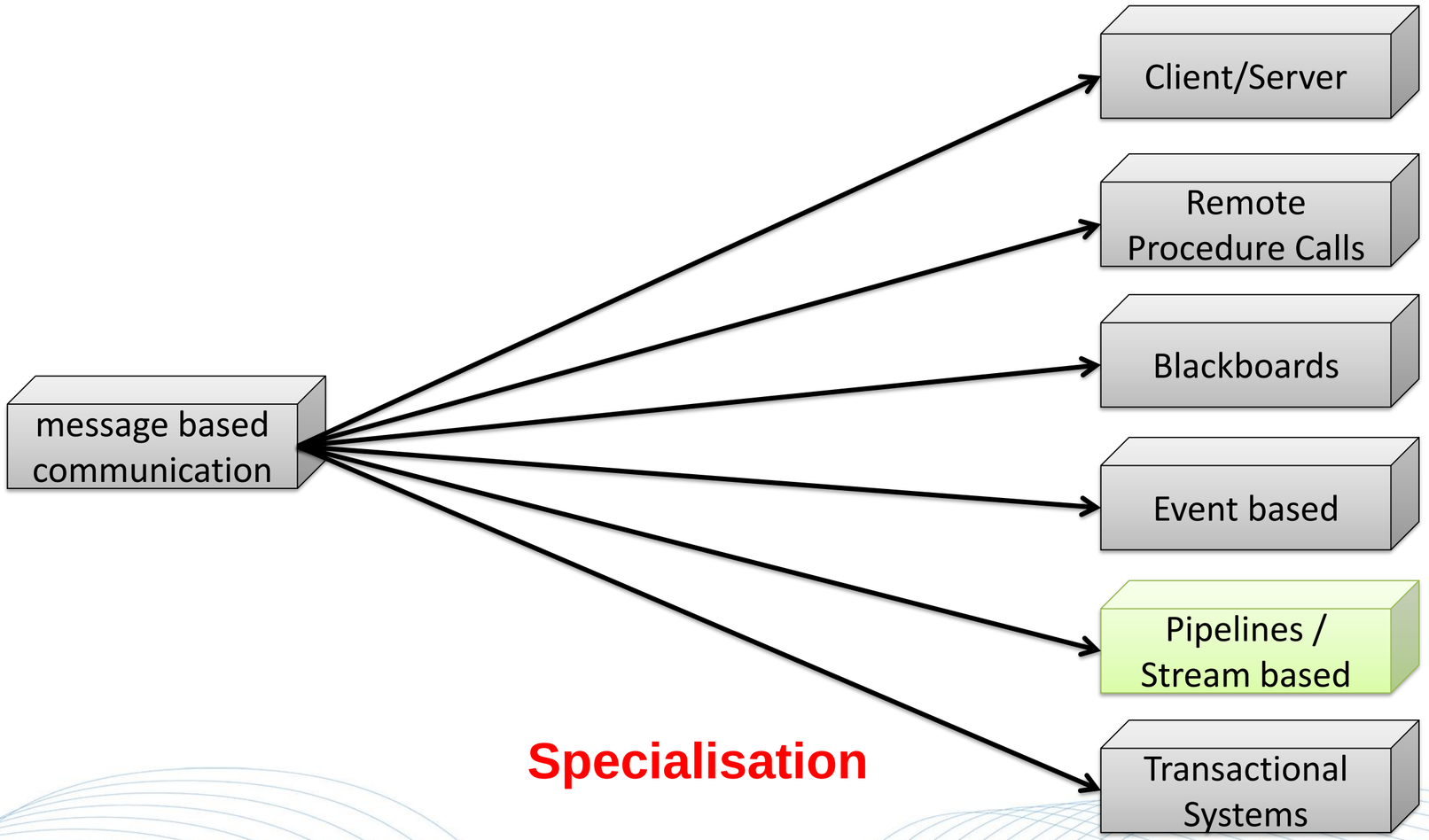
- Timeliness
  - Major improvement compared to blackboards

Event based systems

Cons

- Efficiency
  - Management effort of the event service:
    - book keeping of publisher repertoire and subscriptions (n*m)
    - filtering and storing of events
- Availibility
  - Event service is single-point-of-failure
- Design & Implementation
  - Same as blackboard

Pipelines



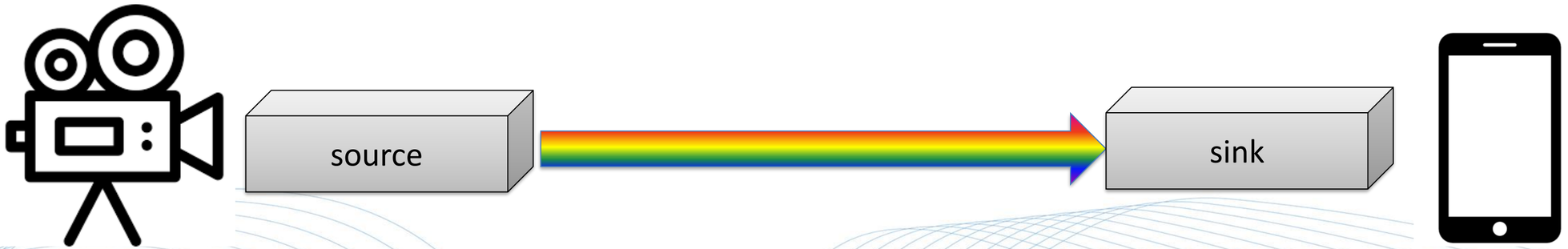
Pipelines / Stream based communication

Goal

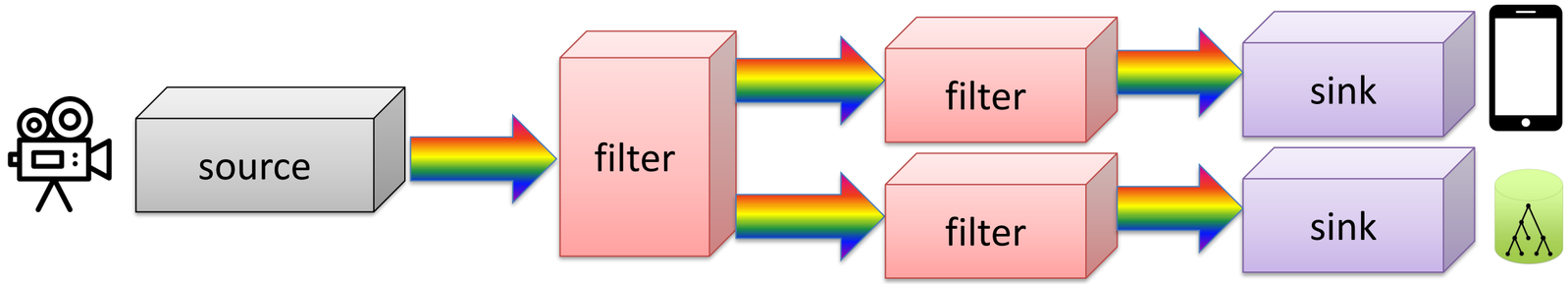
- Communication of high volume data streams in real time

Examples

- Distributed multi media systems: audio / video streams
- Distributed control systems: sensor data streams



Role schema



Source

- data stream producer (camera, database, sensor, software)

Filter

- decoding/encoding, mixing, replicating data streams

Sink

- data stream consumer (display, database, software)

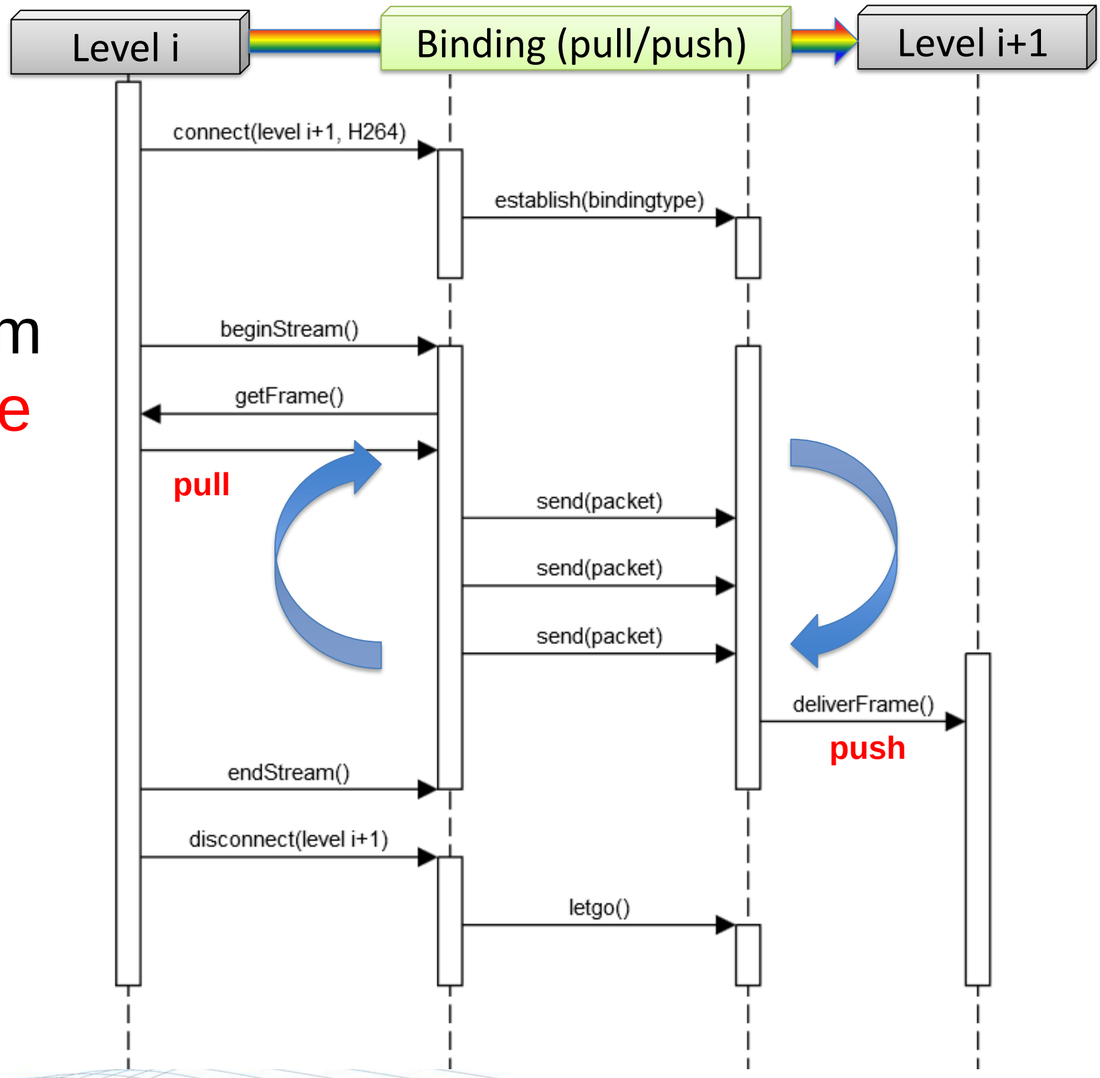
Bindings

- connections between sources, filters and sinks

Role schema - Bindings

Active connection elements between stream processing units that are stream type aware (H.264, H.265, PCM, ...)

- type checks when establishing a connection → correctness
- type dependent data model, failure and termination semantics → efficiency
- wrapping of protocol aspects like sychronization, buffering, segmentation → abstraction



Data model

(possibly infinite) sequence of bytes

Structure defined by stream types and stream format

- audio data: e.g. CD digital audio (PCM)
  - 2 channels of 44100 16 bit values each second
  - Grouping in frames of 33 byte
- video data: e.g. MPEG-2
  - 25 frames (I-B-P) as group-of-pictures (IBBPBBPBBPBBI)
- sensor data

Failure and Termination Semantics

High data volume
→ efficiency is important
→ no universal failure model

Efficient approaches exploit application knowledge (time constraints, data model properties)

- Any guarantees / losses are acceptable?
- Recovery possible?

Pipelines / Stream based communication

Performance, efficieny

- Data model specific bindings, application specific failure semantics

Interoperability, heterogenity

- location specific bindings adapt to local programming language, operating system & hardware (compare Stubs)

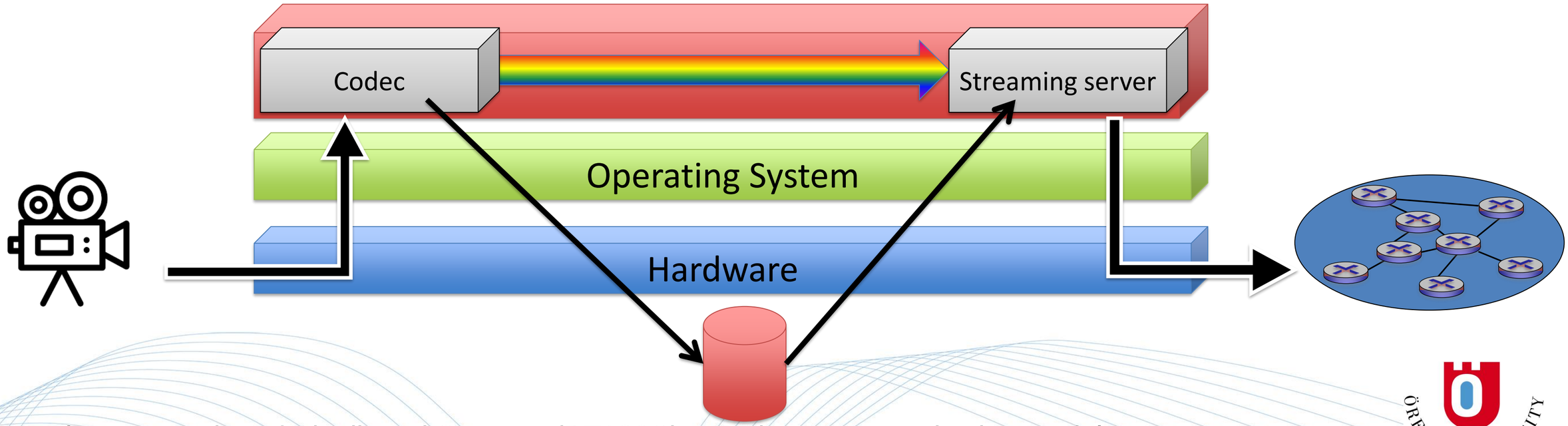
IT security

- bindings isolated from the application, which have the control over the communication

First foray into communication and sychronization

Context change: live streaming

→ local computer with IPC using the shared memory (organized in memory pages)



First foray into communication and sychronization

Exchange of data between processes
→ **communication** (IPC)

Different speeds of source and sink
→ **sychronization**

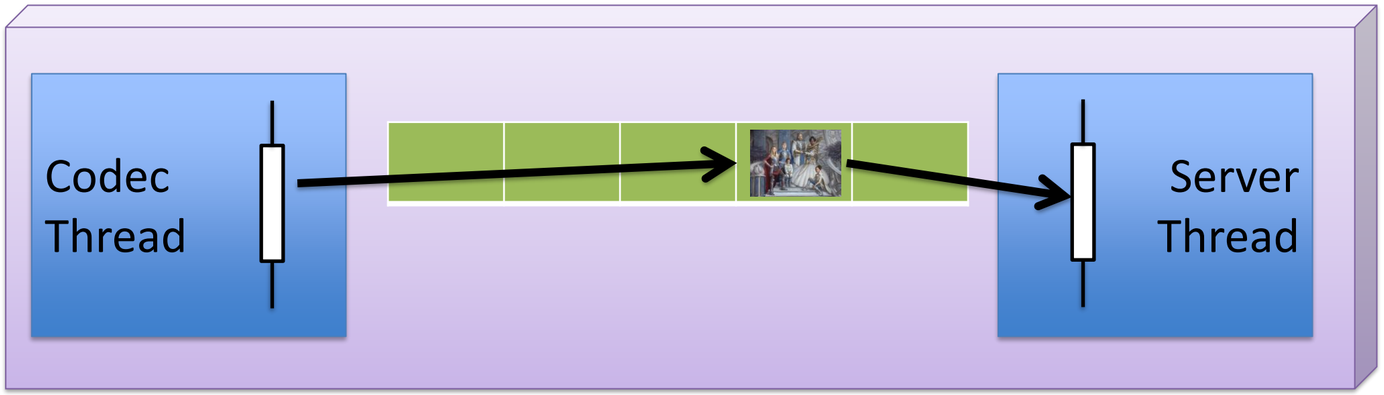
Let us add a buffer


```
forever do {
      bufferQ.end <- Image
}
```

```
forever do {
      Image <- bufferQ.front
}
```

Problem 1: Concurrent read and write

How to guarantee the consistency of the read image?

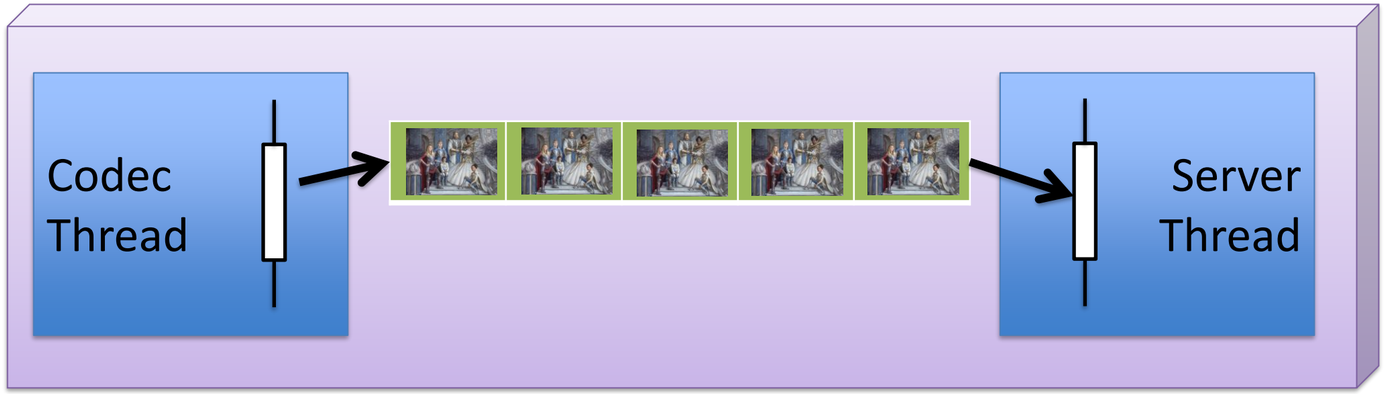


operations on shared variables (buffer elements) are **critical sections**

→ requires synchronization (e.g. via mutual exclusion)

Problem 2: Different Relative Speed

Different speeds by source and sink: buffer empty or buffer overflow



→ requires again synchronization

Reminder

A phase in which a thread is accessing exclusively a resource is called **critical section**. These sections require **mutual exclusion** of concurrent threads.

Assumption

- Concurrent threads (CT) are asynchronous (e.g. infinite loop)
- CT enters and leaves the critical sections at some point

Solution: Semaphores / Monitors

Two operations and a state

- wait() / P()
- signal() / V()

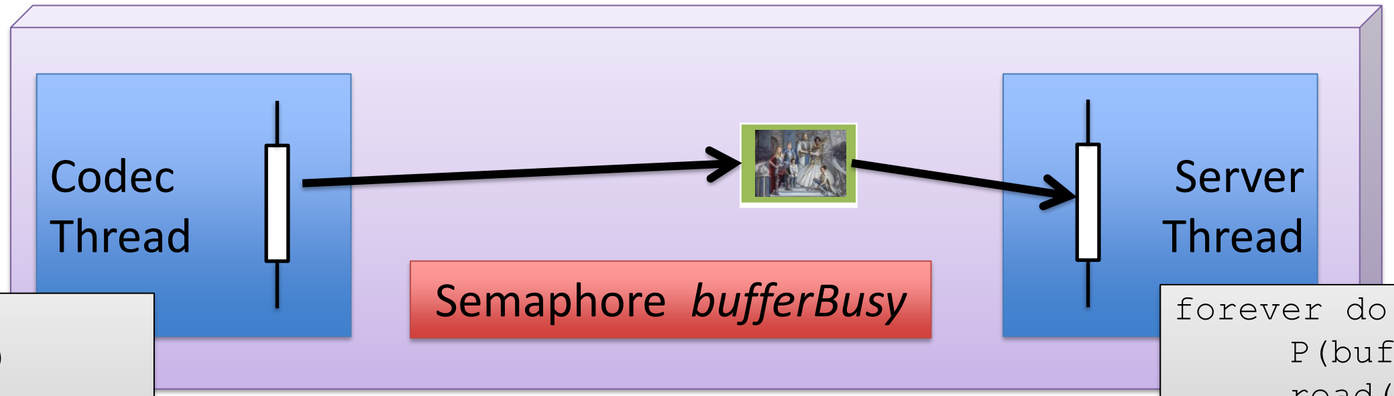
Mutex semaphore (boolean as state)

- Single access to a resource with MUTual EXclusion

Counting semaphore (counter as state)

- Resource with multiple available units

Problem 1: Concurrent read and write



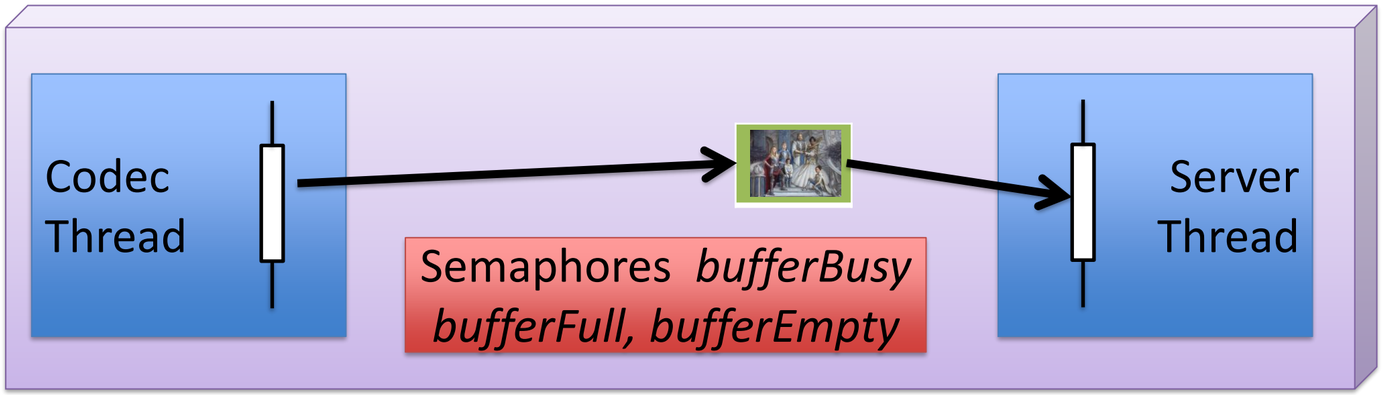

```
forever do {
      encodeCam(img)
      P(bufferBusy)
      write(buffer,img)
      V(bufferBusy)
}
```

```
forever do {
      P(bufferBusy)
      read(buffer,img)
      V(bufferBusy)
      sendToClient(img)
}
```

Solved mutual exclusion

Unsolved different speeds

Problem 2: Different Relative Speed




```
forever do {
      encodeCam(img)
      P(bufferFull)
      P(bufferBusy)
      write(buffer,img)
      V(bufferBusy)
      V(bufferEmpty)
}
```

```
forever do {
      P(bufferEmpty)
      P(bufferBusy)
      read(buffer,img)
      V(bufferBusy)
      V(bufferFull)
      sendToClient(img)
}
```

Solution introduces a new problem

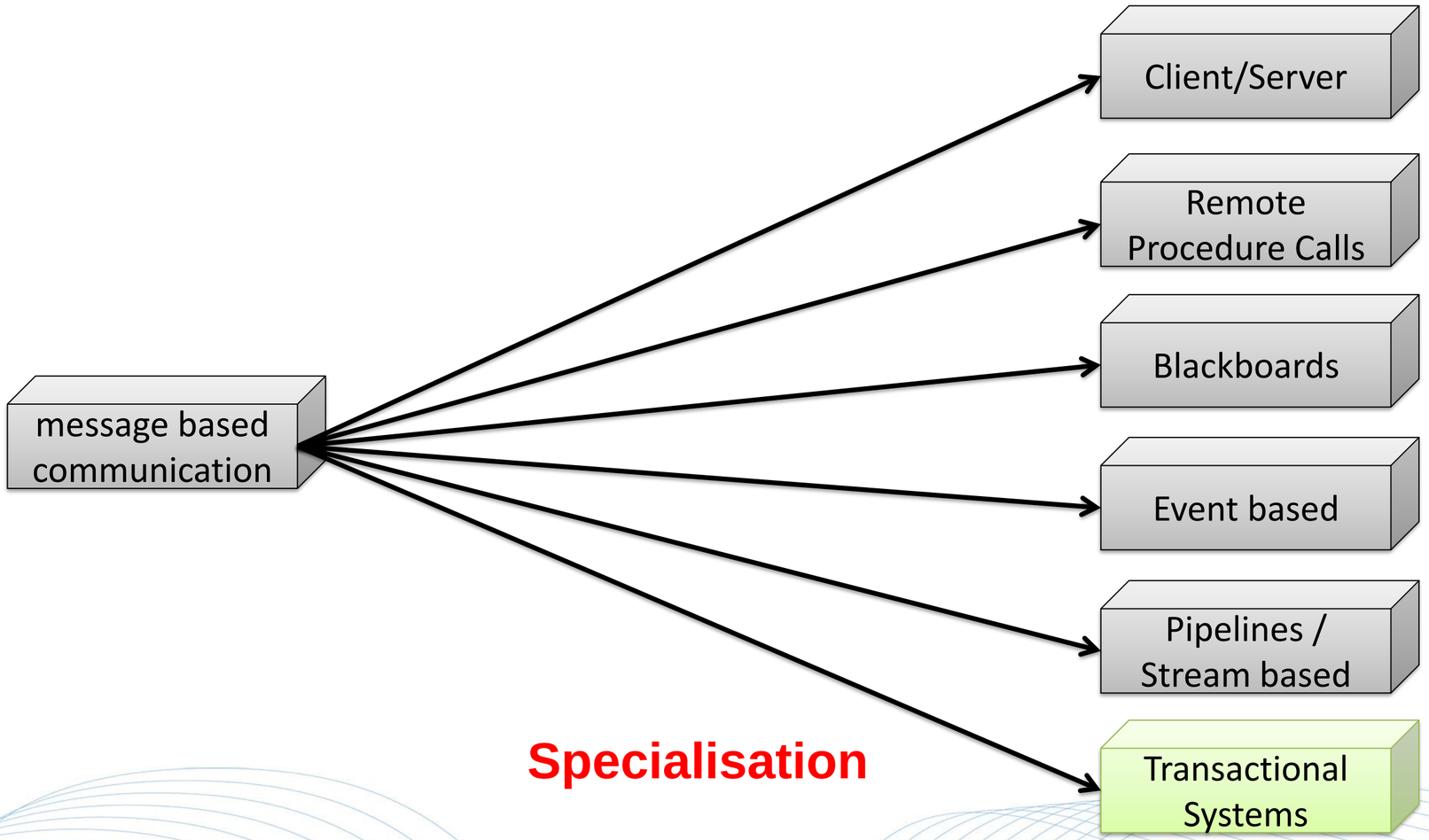
Semaphores achieve mutual exclusion by blocking (blocking synchronization)

Prevents parallel processing ➔ performance issues where masked by hardware development progress

→ Multicore architectures

- parallel / distributed algorithms
- Non-blocking sychronization
- Transactional memory

Transactional Memory



Transactions

A transaction is a collection of operations that adhere to the **ACID** properties.

ACID

- Atomicity
- Consistency
- Isolation
- Durability

ACID properties

Atomicity

- Transactions are indivisible
  → all-or-nothing, also in the presence of failures

Consistency

- Transactions start and end in a consistent state
  → i.e. they do not violate system invariants, e.g. patrial results are not visible to the outside

ACID properties

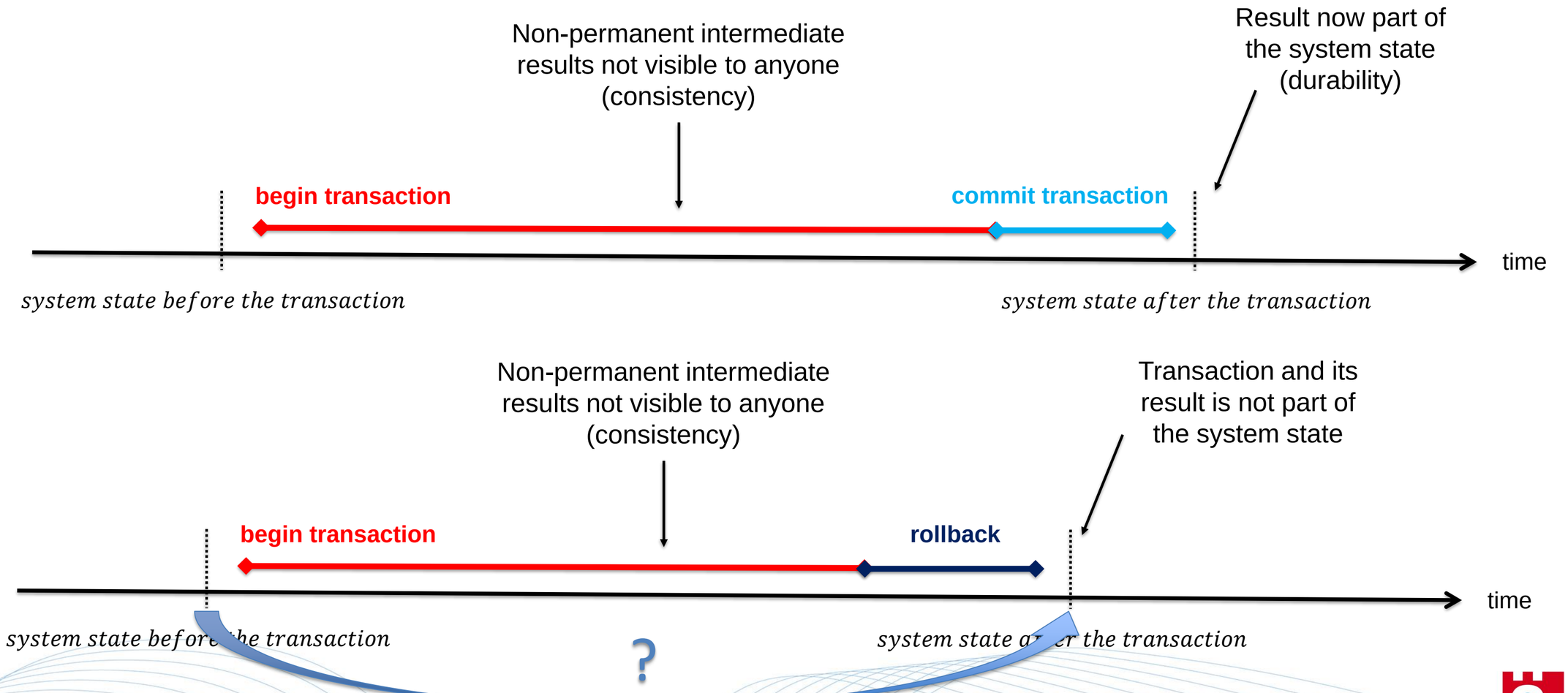
Isolation

- concurrent transactions do not influence each other
  → result of parallel transactions is equivalent of any sequential execution order

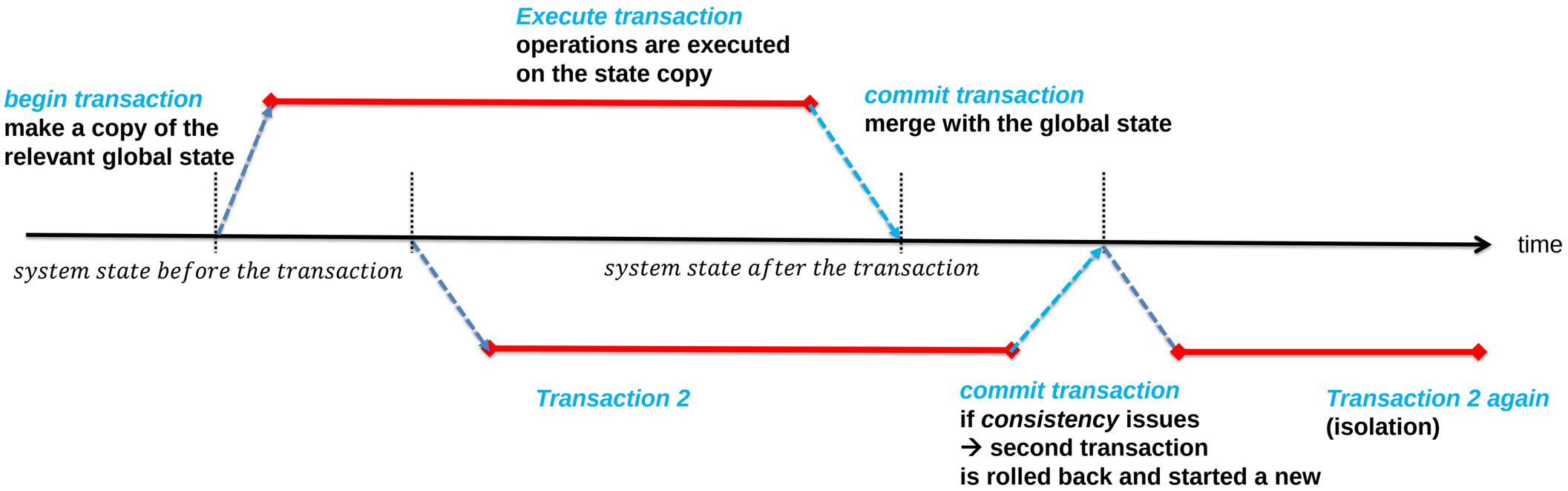
Durability

- The result of a finished transaction is permanent
  → even in the presence of failures

Commit vs. Rollback



Transactional Memory



More on **Consistency** towards the end of the course (topic 6)

Transactional Memory

Semaphores, Monitors, Leases

- Programmer identifies critical sections and deals with correctness, dead-locks, fairness, ...

Transactional Memory

- Programmer identifies critical sections and the TM paradigm does the rest

Idea

- Communication with a shared (virtual) memory
- Synchronization using transactions

Failure & Termination Semantics

Roles: nodes executes a group of sychronous or asynchronous operations that are atomic and isolated

Data: memory objects (e.g. memory page on OS level, program variables on application level)

Termination & Failures:

- ACID

Transactional Memory

Alternative paradigm to coordinate parallelism

- Locks and mutexes ➔ pessimistic CS access control
- Transactions ➔ optimistic CS access control

State of the Art research & development

- Software: Pypy & gcc 4.7+
- Hardware: TSX Transactional Sychronization Extensions (Intel Haswell), Blue Gene & Power 8/9 (IBM)

Summary

Communication Model Components

Role schema

- Who is going to say what and when?

Data model

- The used language of the communication

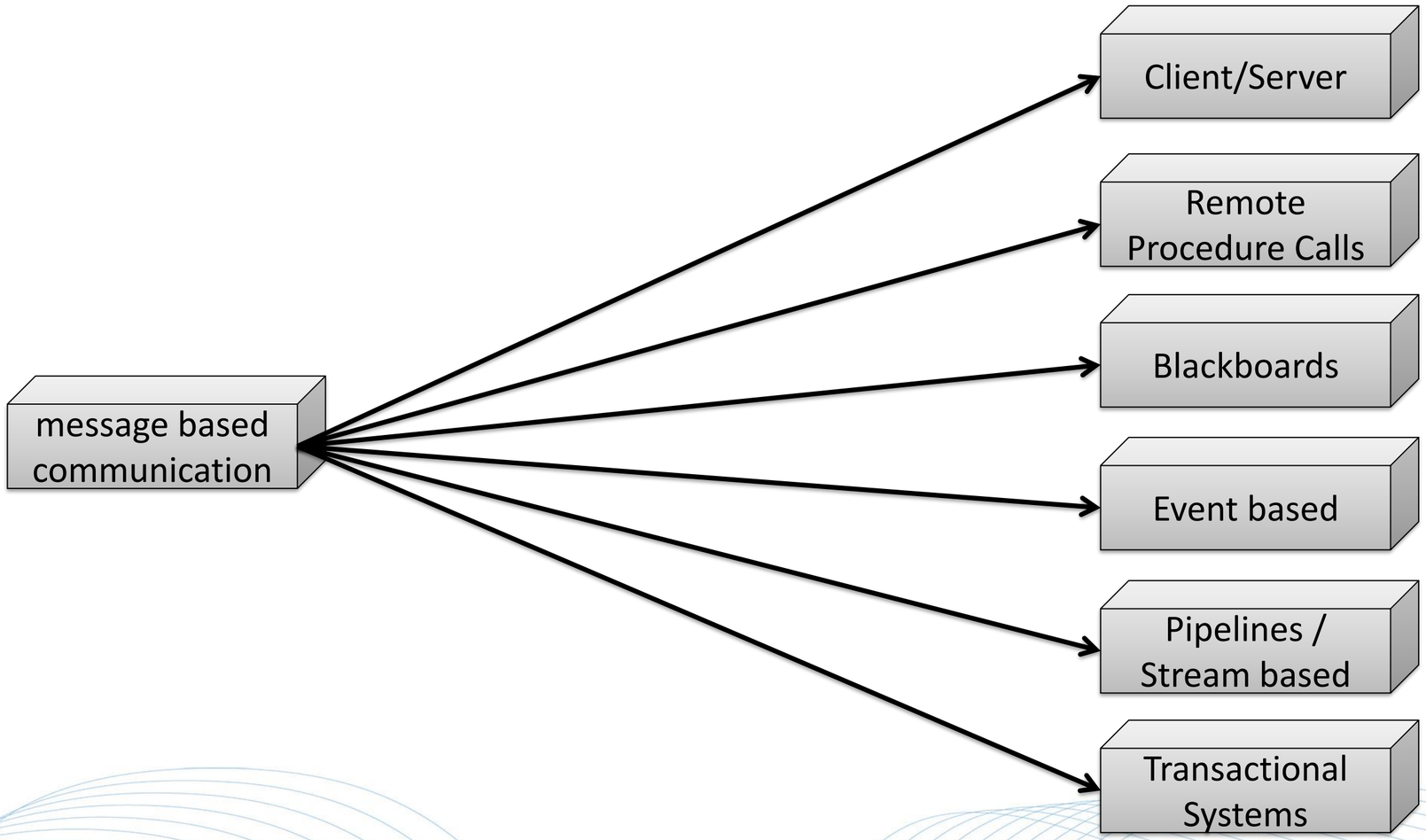
Failure semantics

- What to do if something goes wrong?

Termination semantics

- When is it over?

Summary (2)



Example: Communication Patterns in ROS2

ROS

Robot Operating System
Version 2

NOT an operating system, but an open-source robotics middleware

Topics

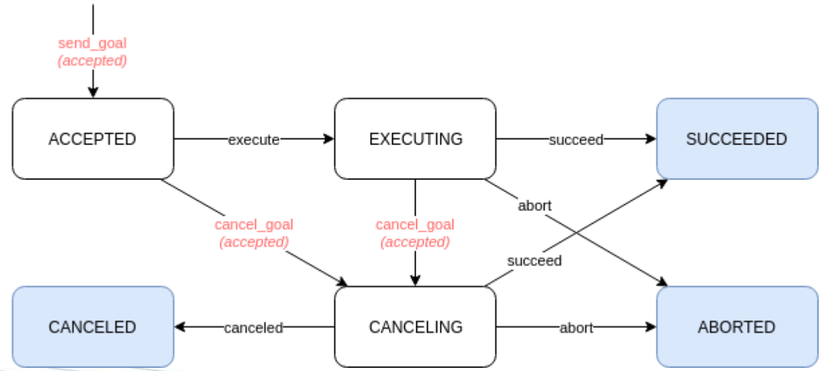
- multi-cast one-way publish subscribe pattern
  often streams from sensors e.g. camera images

Services

- client server RPC pattern
  example: setting some parameter

Actions

- non-blocking client server RPC pattern
  example: go to location command for the robot



ÖREBRO UNIVERSITY

Example (contd): Communication Patterns in ROS2

Actions: Implemented as a mix of services & a topic (actions streams progress report via a topic)

Communication abilities in ROS2 are provided by the DDS (Data Distribution Service) middleware

DDS uses a publish subscribe pattern for communication

> "ROS 2 could either implement services and actions on top of publish-subscribe […] or it could use the DDS RPC specification […] for services and then build actions on top, again like it is in ROS 1. […] it may be the case that services just become a degenerate case of actions."
>
> Source: https://design.ros2.org/articles/ros_on_dds.html