Distributed Systems

Basics of Communication

Prof. Dr. Oliver Hahm

Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering
`oliver.hahm@fb2.fra-uas.de`
`https://teaching.dahahm.de`

20.04.2023

Goals

- Getting accustomed to a **generic message-oriented communication service** with a very high practical relevancy → the Internet and the TCP/IP protocol suite

Goals

- Getting accustomed to a **generic message-oriented communication service** with a very high practical relevancy → the Internet and the TCP/IP protocol suite
- Getting to know **sockets** as a common API for network programming

Goals

- Getting accustomed to a **generic message-oriented communication service** with a very high practical relevancy → the Internet and the TCP/IP protocol suite
- Getting to know **sockets** as a common API for network programming
- Communication services on higher layers (e.g., remote procedure calls (RPCs), web services) are based on these basic services

Goals

- Getting accustomed to a **generic message-oriented communication service** with a very high practical relevancy → the Internet and the TCP/IP protocol suite
- Getting to know **sockets** as a common API for network programming
- Communication services on higher layers (e.g., remote procedure calls (RPCs), web services) are based on these basic services

Layering

Higher layer communication services and middleware platforms offer a more abstract interface which is aligned with the corresponding cooperation paradigm. They are based internally on these fundamental concepts of the underlying communication system

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Agenda

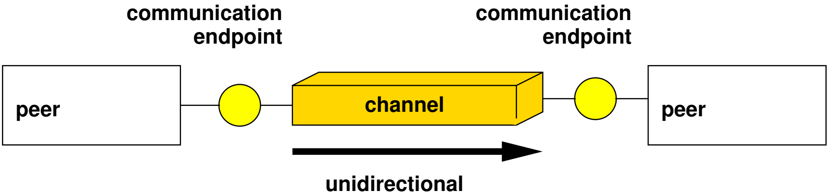
- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Basics of Communication

- All interaction between any participants requires an underlying communication capability
- Communication channel
  - The facility that allows for the connection/coupling of communication partners is called **communication channel** or simply **channel**

Basics of Communication

- All interaction between any participants requires an underlying communication capability
- Communication channel
  - The facility that allows for the connection/coupling of communication partners is called **communication channel** or simply **channel**
- Direction of the message flow of a channel
  - A channel is called **directed** or **unidirectional** if one process takes exclusively the sender role and the other process takes exclusively the receiver role
  - Otherwise the channel is called **undirected** or **bidirectional**



Aspects of Communication

1. The number of communication peers
2. Addressing
3. Buffering
4. Communication pattern
5. Message structure

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Number of Peers of a Channel

- Exactly two:
  - Most simple (and most common) case
- More than two:
  - For certain applications group communication may be appropriate
  - → **multicast** service
  - Special case: Broadcast

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Direct Addressing

- Each communication partner have a distinct, unambiguous (potentially globally unique) address
- Addressing can be explicit and symmetrical
  - → The sender must explicitly name the receiver – and vice versa

Example:

`SEND ( P, message )` - Send a message to process P

`RECEIVE ( Q, message )` - Receive a message from process Q

Direct Addressing

- Each communication partner have a distinct, unambiguous (potentially globally unique) address
- Addressing can be explicit and symmetrical
  - → The sender must explicitly name the receiver – and vice versa

Example:

`SEND ( P, message )` - Send a message to process P

`RECEIVE ( Q, message )` - Receive a message from process Q

- Asymmetrical variant (e.g., for server processes):
  - → Only the sender names the receiver, the receiver (server) gets to know the identity of the sender only on reception

Example:

```
SEND ( P, message )
RECEIVE ( sender_id , message )
```

Indirect Addressing

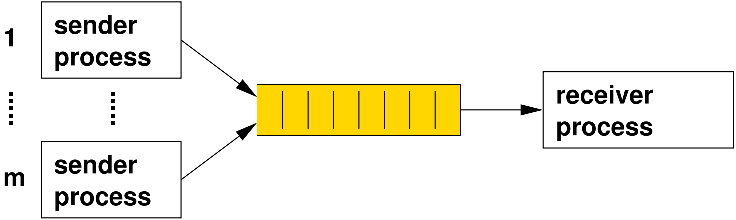
- Communication happens indirectly via intermediary instances
- Advantages:
  - Improved **modularity**
  - The number of communication partners can be restructured in a **transparent** manner, e.g., after a node failed
  - Extend options of group communication, like, for example, $m : 1$, $1 : n$, $m : n$
  - Intermediary instance may . . .
    - only forward
    - store and forward
    - transform/translate messages

Example for Indirect Addressing

Mailbox:

`SEND ( mbox, message )` - Send a message to a mailbox `mbox`.

`RECEIVE ( mbox, message )` - Receive a message from a mailbox `mbox`.



Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Buffering

- Capacity of a channel:
  The number of messages which can be **stored** temporarily in a channel to **decouple** sender and receiver in time
- The channel's capability for buffering messages is typically implemented by a (waiting) **queue**
- In distributed systems the waiting queue is typically realized on the receiver site (rendezvous site)
- Buffering can be used to restore the message order or to modify the sending order

No Buffering (Capacity Zero)

Source: https://de.toonpool.com/, Author: Fussel

- Unbuffered communication
- Sender and receiver are very closely coupled in time
- Also called **Rendezvous**
- Often considered to be too inflexible
- Example:
  - A sender is blocked when a `SEND` operation happens before a corresponding `RECEIVE` operation
  - As soon as the corresponding `RECEIVE` operation is executed the message is copied directly without any buffering from the sender process to the receiver process
  - If vice versa a `RECEIVE` operation happens at first, the receiver is blocked until the `SEND` operation is executed
- Example: Communication between threads in various microkernels such as RIOT or L4

Limited Capacity

- A channel can contain at any point of time a maximum of N messages (waiting queue with capacity N)
- `SEND` operation during a non-full waiting queue
  - The message is stored in the queue
  - The sender process resumes its normal operation
- Waiting queue is full (it contains N sent but not yet received messages):
  - The sender process blocks until free space in the queue is available again or the message is discarded
  - Analogously a receiver is blocked on a `RECEIVE` operation if the waiting queue is empty

Consequences

- Buffered communication enables **loose coupling** of the communication partners in terms of time
- Passing the message to the communication system does not imply that the receiver has received the message
- Typically the sender won't even know a maximum duration until a message is received
- If this knowledge is of importance for the sender an explicitly communication between sender and receiver is required:

```
Process P (Sender):                    Process Q (Receiver):
...                                    ...
send ( Q, message );           ⟶      receive ( P, message );
receive ( Q, reply);           ⟵      send ( P, "'acknowledgement"' );
...                                    ...
```

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Communication Pattern

One-Way

- Single message without response or acknowledgement

Request/Response

- Client role (consumer)
- Server role (producer)
- Often blocking on the client site (→ standard RPC)

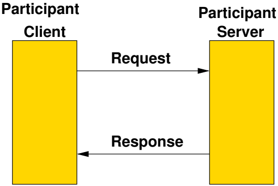
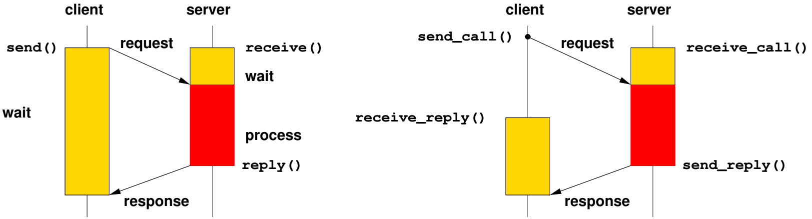



Photo by Mitchell Luo on Unsplash

Differing Synchronicity for Request/Response:

- Synchronous call: The sender process blocks until the end of the communication process (→ arrival of the response)
  ⇒ no parallelism
- Asynchronous call: Sender is only delayed for the initiation of the communication process (→ passing the message to the communication system)



(a) synchronous (b) asynchronous

Publisher/Subscriber Model

- Message classified by **topics** or event channels
- Receiver subscribe topics (subscriber)
- Sender publishes messages or events (publisher)
- Model allows for transparent sending of messages to multiple receivers!
- Examples: CORBA Notification Service, OMG DDS, MQTT

More Complex Communication Patterns

- Not very common in simple communication systems
- Exception: Three-way handshake between two participants for reliable connection establishment
- More complex patterns emerge by group communication
- Very common on the upper layers
- Example: business process

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Byte stream

- Passed messages of various `SEND` operations cannot be identified as individual units any more
  - ⇒ message borders get lost
- The receiver (and the communication system) observe only sequence of characters (byte stream)
- Example: UNIX pipes

Message container

- Messages can be identified by sender and receiver
- The messages have either a fixed length or the length can be derived on both sides

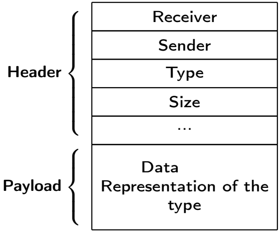
⇒ The message borders remain intact

- The correct interpretation of the internal structure of a message is the responsibility of the communication peers
- Example: UNIX message queues

Message Structure

Typed messages

- Messages have a typed structure
- The type is know to the sender and receiver and partly by the communication system
- The type is used as part of the operations
- Exemplary structure of a message:



Header
Receiver
Sender
Type
Size
…

Payload
Data
Representation of the type

- Message body may contain typed objects (→ object-orientation)

Message Serialization

Example

- Java object serialization transforms an object into a bytestream and vice versa (deserialization)
    - The header contains information about type, layout etc., the body contains the actual data
    - Java class implements the interface `java.io.Serializable`
    - All attributes of the class must be `serializable` themselves or marked as `transient`
    - Operations are `writeObject()`, `readObject()`

Messages of a Documental Nature

- Example: HTML over HTTP
- XML-Documents
  - Very popular today
  - Type description via scheme
- Example: SOAP (**Simple Object Access Protocol**)

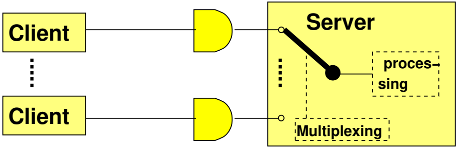
```
<soap-env:Envelope
xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap-env:Body>
        <tns:getFlaeche xmlns:tns="urn:tns:beispiel">
            <tns:seite1 xsi:type="xsd:double">8.0</tns:seite1>
            <tns:seite2 xsi:type="xsd:double">4.0</tns:seite2>
        </tns:getFlaeche>
    </soap-env:Body>
</soap-env:Envelope>

```

Agenda

- Basics of Communication
  - Number of Communication Peers
  - Addressing
  - Buffering
  - Communication Pattern
  - Semantics of Messages
- Server Architecture

Server Architecture

- Architectural principles for the internal structure of server processes
  - *Problem:* A server typically needs to communicate with multiple clients at once



Models

- Simple sequential server
- Sequential server as state machine
- Parallel server processes
- Multithreaded server

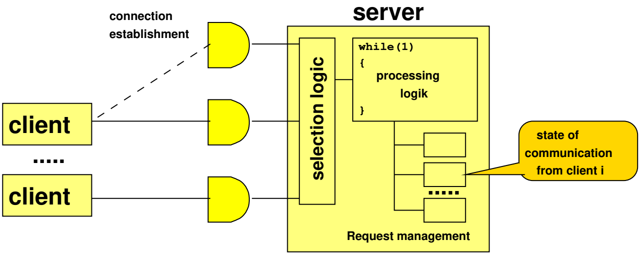
Simple Sequential Server

- *One* process handle the requests of all clients one after another

→ Problem if the server acts as a client towards another server while processing a request: ⇒ the whole server gets blocked!

- *Drawbacks:*
  - No concurrency in the server
  - No use of (a potentially) underlying multicore architecture by a single server process
- This approach is hardly acceptable for productive applications in the traditional Internet, but may be applicable for very constrained devices

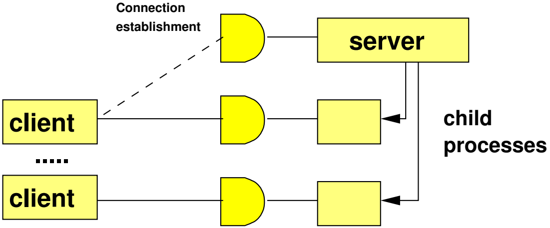
Sequential Server as State Machine



- No internal blocking:
  multiple requests can be handled in an overlapping manner
- Multiplexing "‘by hand"’ ⇒ complex to program
- Selection logic in UNIX:
  - non-blocking requests (Option `O_NDELAY`) and polling
  - `select()`

Parallel Server Processes

Architecture:



- Child processes preserve the current state of communication per remote peer in memory
- Advantage: Multicore architecture can be used
- Problem: Expensive process handling (→ context switches)

Multithreaded Server

- Automated resolution of the multiplexing problem
    - A thread is permanently assigned to each request at the start of processing
    - Each single thread of the server may block at any point of time without affecting the overall concurrency

  → Thread pool is required
- Applicable for all paradigms of distributed applications
- *Requires synchronisation!*

Current State of Multithreading

- All modern operating systems and runtime environments support threading
- Even many embedded operating systems (like RIOT) support multithreading by now
- Typical APIs
    - pthreads POSIX 1003.4 (C/C++)
    - Boost threads (C++)
    - Java Concurrency since SE 5: `java.util.concurrent`

Important takeaway messages of this chapter

- For all higher layer services in a distributed system an underlying communication system is required
- The facility that enables the communication between the peers is called channel
- Important characteristics of a communication system are
  - the number of participants
  - the addressing style
  - its capacity
  - the communication pattern
  - the semantics of the message
- Depending on the use case various architectures to design a server application are possible