TECHNISCHE
UNIVERSITÄT
DARMSTADT

Technical University Darmstadt

– Department of Computer Science –

Distributed Systems and Algorithms

A Summary of the Course Contents

Author:

Jonas Weßner

Teaching professor	:	Prof. Dr. Eberhard Max Mühlhäuser
Semester	:	Winter semester 2022/2023

CONTENTS

LIST OF ABBREVIATIONS

MPP Massively Parallel Programming
CN computer network
AS autonomous system
MIMD multiple instructions multiple data
GUI graphical user interface
OS operating system
IPC inter-process communicaiton
TCP Transmission Control Protocol
SCTP Stream Control Transmission Protocol
DCCP Datagram Congestion Control Protocold
CSS communication subsystem
QUIC Quick UDP Internet Connections
AMQP Advanced Message Queuing Protocol

1 INTRODUCTION TO DISTRIBUTED SYSTEMS

1.1 DEFINITION OF A DISTRIBUTED SYSTEM

A distributed system is a computer network (CN) providing some kind of abstraction/transparency provided by the transport layer and potentially by some kind of middleware. Distributed systems are looked at from the application layer of the TCP/IP model.

1.2 CHALLENGES

There are three basic challenges, which together form the *bermuda triangle* of distributed systems:

1. **No accessible global state:** The individual nodes might be in any state, which cannot be known at all other nodes at all times.
2. **No common time/clock:** As every system has its own clock and the transmission of messages requires time, the clocks of nodes cannot be synchronized perfectly.
3. **Indeterminism:** Due to race conditions and erroneous networks, the program execution for the whole system is non-deterministic.

1.3 REQUIREMENTS

We would like to require distributed systems to have certain properties. These properties differ from book to book, but we will define the following seven properties for this lecture:

1. **Heterogeneity Support:** Different hardware, operating systems and programming languages are supported.
2. **Openness:** Anyone might participate! Therefore open known standard protocols shall be used.
3. **Scalability:** Scalability concerning processors, memory, storage, and nodes.
4. **Security:** Secure the network, because malicious users may try to compromise it.

5. **Fault Tolerance:** We would like to be resilient to node (autonomous system (AS)) and network (communication subsystem (CSS)) failures.
6. **Concurrency:** Distributed Systems are inherently concurrent. We of course want to take advantage of this property. However, then we need to ensure consistency e.g. when concurrently accessing shared resources.
7. **Transparency:** We would like to abstract from different properties of the CN

1.4 ABSTRACTION LEVELS

There are several abstraction layers, from which we can look at distributed systems. In the following, we divide into 4 levels, which are ordered ascending by their level of abstraction. When talking about distributed systems, we would like to be at level 3 or 4.

1. **Physical configuration:** How are nodes connected physically i.e. using which typology and which transmission channels.
2. **Logical configuration:** Looking at the network from the *transport layer*.
3. **Process network:** Processes communicate using messages. The real location of the processes in the computer network is hidden. This is the abstraction layer of e.g. MPI.
4. **Distributed Algorithm:** This abstraction layer defines distributed algorithms regardless of the target environment, the target configuration (e.g. number of processors and RAM) and the programming language.

1.5 TRANSPARENCY

We would like distributed systems to provide transparency concerning certain properties. Transparency here means the concealment of the property. The properties are:

1. **Access:** The interface for accessing resources is identical regardless of the node, which is accessed.
2. **Location:** Resources are accessed without knowledge about their physical/network-level location.
3. **Concurrency:** Several processes act concurrently on shared resources, but the user experiences simulated single-user execution.
4. **Replication:** Multiple instances of e.g. compute nodes or data are used for performance or reliability reasons without the user noticing this.

5. **Failure:** Node and network failures are concealed.
6. **Mobility:** Resources might change their physical location during program execution without the user noticing it.
7. **Performance:** Local and remote accesses happen with similar performance. In practice, this is often not the case.
8. **Scaling:** Allows scalability without changes to the system structure or the user-defined algorithms.

1.6 CLASSIFICATION

According to Flynn's taxonomy, distributed systems can be classified as the multiple instructions multiple data (MIMD) model.

Furthermore, they can be classified based on their distributed property. In practice, usually more than one property is distributed:

1. **Function Federation:** The systems functionality is distributed over nodes.
2. **Load Federation:** Balancing load using multiple nodes.
3. **Data federation:** Distributing data on different nodes.
4. **Availability:** Increasing Availability through replication.

1.7 TIER ARCHITECTURES

Applications can be distributed into multiple layers, which then result in Tier-architectures:

1. **2-Tier:** graphical user interface (GUI) on one tier, DBMS on other tier.
2. **3-Tier:** Client – Application Server – Database Server
3. **n-Tier:** Splitting the application further into n tiers, where each tier might consist of several nodes.

1.8 FUNDAMENTAL APPROACHES

There are 4 fundamental approaches for writing distributed software. The most widespread approach, which is also a topic of this lecture, is the distributed programming approach:

1. **Distributed Operating System:** This approach constructs an entire operating system (OS) built on top of a computer network, such that parallel programming and distributed storage are already supported

by the OS. However, because this approach is too overkill for specific use cases, where distribution is not required, it never became widespread. Today, the network OS, which adds network integration on OS-level (e.g. Mounting cloud storage, online authorization), is more common.

2. **Distributed Database:** Distributed databases allow for scalable and accessible storage of data. However, they do not provide much infrastructure when writing parallel algorithms. Distributed databases exist e.g. as cloud storage, but have never reached internet scale, such that any computer could be used to retrieve any data.
3. **Protocol Approach:** In this approach we define open protocols for communication to build distributed systems. This ensures openness and simplicity. On the counter side, only very limited functionality is supported and the distribution is usually only a client-server separation.
4. **Distributed Programming:** This approach can be structured into 3 possible models:

   0 The zero-support approach builds distributed programs on top of the transport layer using UDP or TCP directly.

   M The middleware approach uses a middleware to abstract from the transport layer. This is today's approach.

   D The distributed programming language and runtime system approach uses a parallel programming language and a distributed runtime. This results in compiler support for distributed programming. However, this is hard to realize and has seen no acceptance in the past 50 years. Maybe it could be the way to go in the future.

1.9 SYSTEM MODELS

Distributed Systems can be classified based on their system model:

1. **Client-Server:** The server is a fixed system participant, which provides functionality to clients.
2. **Peer-2-Peer:** All nodes have the same functionality.

1.10 DISTRIBUTED PROGRAMMING PARADIGMS

In figure 1.1 we see a taxonomy for distributed programming paradigms. We can structure them into pull- and push-paradigms.

TODO: clean this up (lecture slides 27.10.22)

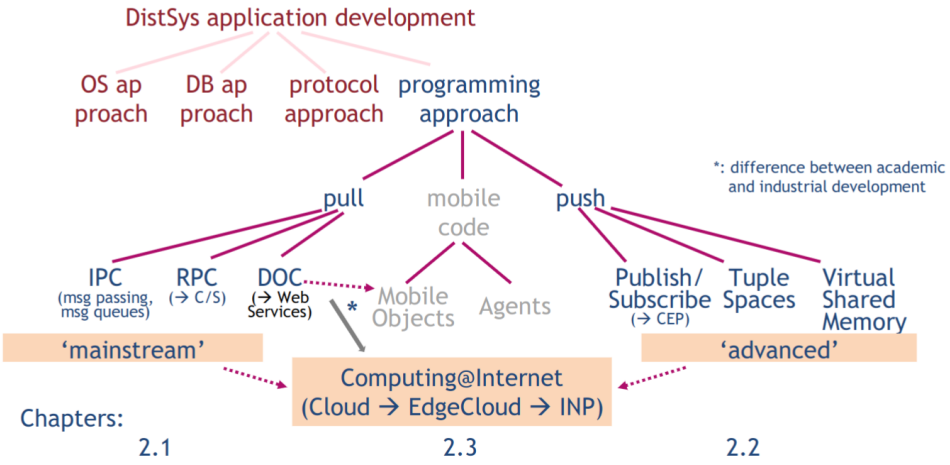

Figure 1.1: Pragmatic taxonomy of distributed programming paradigms

2

INTER-PROCESS COMMUNICATION (IPC)

Inter-process communication means exchanging data between processes. There are different definitions. For example, the definition on Wikipedia says that inter-process communicaiton (IPC) is only the exchange of data between processes on one system i.e. of processes with access to shared memory. However, in this course, the definition is broader and includes the exchange of data between processes in a distributed system. IPC is classified into three groups, which will be explained in detail in the following. Shared Memory, however, is less relevant for this course and will only be discussed briefly.

2.1 SHARED MEMORY

IPC using shared memory is less relevant for this course and will therefore only be discussed briefly.

When the processes have access to shared memory, they can communicate by writing and reading from that memory. This is often used in multithreaded applications, such as in OS development or Massively Parallel Programming (MPP) but is difficult to achieve in distributed systems. The communication is therefore implicit (i.e. there is no dedicated operation for sending messages). To synchronize processes and avoid race conditions, mutex locks or semaphores are used.

2.2 MESSAGE PASSING

Message Passing is an explicit communication through dedicated send and receive operations. Here, the synchronization between processes happens through blocking send or receive operations.

2.2.1 *Sockets*

Sockets define communication endpoints for processes through a socket address, which consists of the IP address and a port number. Different well-known port numbers are used for specific services e.g. HTTP servers listen on port 80 and ssh agents on port 22. In practice, there are two types of sockets for the two main transport layer protocols – **UDP!** (UDP!) and Transmission Control Protocol (TCP). The corresponding sockets, therefore,

provide the functionality, which is provided by the respective transport layer protocol.

Usually, many processes can send messages to the same endpoint of another process (e.g. many clients send messages to the same server socket address). In order to be able to respond to all processes separately, usually servers keep one socket for incoming requests and then open another socket for responding, such that there are $n + 1$ sockets, given the number of clients is equal to n.

2.2.1.1 *User Datagram Protocol (UDP)*

Characteristics of communication via UDP sockets are the following:

- Connection-less communication
- Messages are sent without acknowledgment, retries, ordering, congestion control and flow control.
- Message size is limited to 65 MB
- Very performant, because of limited comfort features and connection-less design.
- Sends are non-blocking
- If messages are sent before receiving socket is bound to this address, the messages are discarded at the destination machine
- Datagram header and payload are protected by a checksum, which ensures only valid messages are received. However, messages will not be corrected based on that, but rather be discarded.
- Usage examples: DNS, Voice over IP, Sensor Data, Video Streaming

Usually, usage of UPD-sockets follows the following scheme for the **sender**:

1. Create a socket
2. Send something over the socket to a destination socket address
3. Close socket

The workflow for the **receiving** side of a UPD-socket communication is usually the following:

1. Bind a socket to the local server address
2. Receive a message from the socket
3. Close socket

2.2.1.2 *Transmission Control Protocol (TCP)*

Characteristics for communication over TCP-sockets are the following:

- Connection-oriented communication
- The message size is arbitrary but will be delivered as a byte stream. Therefore, a data format on top must ensure that the end of messages can be identified.
- Reliability: validity and integrity ensured by using checksums and retransmission.
- Flow control: Fast writer is blocked until a *slow receiver* is ready for new bytes.
- Congestion control: Fast writer will slow down for *slow networks*.
- Example Applications protocols: HTTP, FTP, SMTP

The TCP-socket interfaces on POSIX and Windows systems follow the so-called Barkley socket interface, which is shown in figure 2.1.

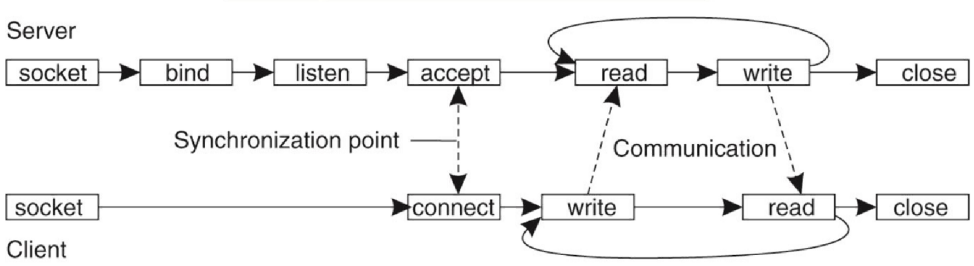

Figure 2.1: Barkley Socket Interface

One challenge for TCP implementations is when to send messages in the local write buffer. One strategy would be to send them as soon as the buffer is full. But this is only useful when a large amount of data is written, otherwise, it would delay the send operations of small messages. Another approach, which was implemented in the past is to combine the previous approach with a timeout (e.g. 0.5 seconds), after which the buffer is guaranteed to be flushed. Nowadays, Nagle's algorithm is used. This flushes the buffer if it is full or if all of the previously sent packages have already been acknowledged.

2.2.2 *Transport Layer Alternatives*

TCP is by far the most used transport protocol for applications, which requires reliability. At the same time, it is old and not optimal. Therefore, there are other more modern transport protocols, which, however, are yet to be widely accepted.

The Stream Control Transmission Protocol (SCTP) offers message-oriented semantics (like TCP) but otherwise behaves like TCP to ensure reliability and its other useful features. Furthermore, reliability, ordering and fragemenütation of messages can be toggled in order to provide a more flexible interface for different use cases. Moreover, multi-streaming (multiple independent data streams for one connection e.g. one stream for messages, one stream for videos) and multi-homing (endpoints can have multiple IP addresses for failover) are supported.

The Datagram Congestion Control Protocold (DCCP) provides fast and unreliable transmission of messages (like UDP) with additional congestion control. This protocol has not been accepted by the community and will probably die out.

Quick UDP Internet Connections (QUIC) is a protocol designed by Google and is currently in the standardization process. It is implemented in Chrome, Chromium and other Google Apps. It intertwines with security (TSL) and application layer (HTTP) in order to increase its performance. This, however, violates the principle of distribution of concerns.

2.2.3 *Distributed Programming Languages*

Although distributed programming is of high interest, distributed programming languages have not been established yet. The reason is, that programmers do not like to learn new languages, especially if these are not widespread yet. Furthermore, one distributed programming language typically only supports one programming paradigm (OOP, functional programming ...), such that one distributed programming language can not make everybody happy. Moreover, the comfort of distributed programming languages can also partly be achieved through middleware. Nevertheless, research in this area is being done. Some examples of distributed programming languages are CSP, ARGUS, Linda, Emerald, Erlang and SCALA.

2.3 MESSAGE QUEUES

Message queues provide asynchronous persistent communication through a middleware. A message queue is an (optionally persistent) buffer at communication servers – the so-called brokers. A powerful feature that is provided by message queues is that the sender and receiver must not be present at the same time as long as the message broker is online. For example, a sensor can send something to a queue and it might be picked up by another node later in time.

The communicating processes have access to the following primitive operations to work with the message queue:

- **Put:** Upload a message to a specific queue.
- **Get:** Block until there is a message in a specific queue and retrieve it.
- **Poll:** Check if a message is in a specific queue and, if there is one, retrieve it. This is a non-blocking operation.
- **Notify:** Register a handler, which is called as soon as a message arrives in a specific queue.

Specific implementations are for example Advanced Message Queuing Protocol (AMQP) or MQTT. They, however, also have features of public subscribe models and, therefore, can be classified as message queuing communication as well as publish-subscribe communication.

AMQP, as an example of an implementation of message queuing, is quite bloated and uses rather complicated terms and semantics: The components of the system (e.g. clients, broker, persistence layer) are called *containers*, within which one or more *nodes* must be created. A *connection* is a bidirectional transport stream between containers. In order to get to a *session*, two unidirectional *channels* must be combined. The possible number of *sessions* is limited to the number *channels*. Nevertheless, AMQP is a popular framework.

3

REMOTE PROCEDURE CALLS

4

GLOSSARY

In this chapter, basic terms will be defined and explained.

4.1 MASSIVELY PARALLEL PROGRAMMING

Massively Parallel Programming (MPP) is a programming model which makes enormous use of multithreading and therefore of multi-core parallelization. As opposed to in distributed systems, in MPP all processes or threads share common main memory, such that messages/information travel almost instantaneously from one execution thread to another. Also, communication is much less error-prone, because it is done in main memory as opposed to via a network.

4.2 COMPUTER NETWORK

A CN is a set of ASs connected by a CSS. Here an AS is not an AS in the sense of autonomous computer networks, but instead a single node of a computer network i.e. some CPU with some memory.

4.3 RELIABILITY

Reliability in the context of IPC is the conjunctive presence of validity and integrity. Validity means that all messages in a sending process's outgoing buffer will be delivered to the receiving process's incoming buffer at some point in time. Integrity means that any message received is identical to the sent message, meaning no things like bit flips happened during transmission. Reliability is provided by the TCP. However, when using TCP, messages are not received as they have been sent due to the byte stream semantics. So depending on how tight one looks at the definition of integrity, TCP might not fully provide integrity.