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

1 INTRODUCTION TO DISTRIBUTED SYSTEMS

1.1 DEFINITION OF A DISTRIBUTED SYSTEM

A distributed system is a **CN!** (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 (**AS!** (AS!)) and network (**CSS!** (CSS!)) failures.

6. **Concurrency:** Distributed Networks 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 **MIMD!** (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: GUI!** (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 **OS!** (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 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 **??** 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) TODO: create chapters, like done in the lecture slides

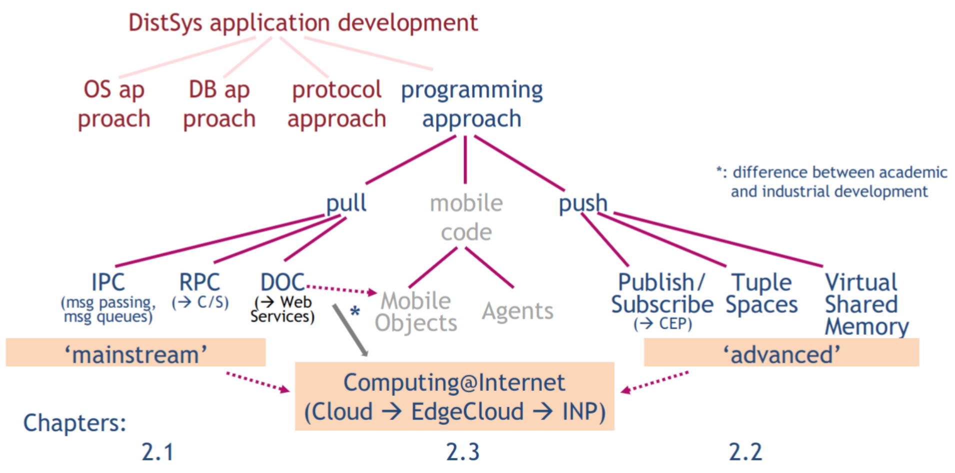

Figure 1.1: Pragmatic taxonomy of distributed programming paradigms

2 INTERPROCESS COMMUNICATION

IPC! (**IPC!**) can be split into three types, which will be explained in more detail in the following.

2.0.1 *Shared Memory*

NOTES: Implicit communication about shared memory. Synchronization using locks Widespread in OS and MPP DIfficult in DistSystems, where no shared memory exists -> emulate it if required

Less relevant for this lecture

2.0.2 *Message Passing*

- Explicit communication through messages - synchronization through blocking receive and send operations - exactly what happens in distSyst, because we send messages over the transport layer

TCP Sockets usage: connection cycle: (insert a picture from slide 15) Server: socket (creates port association for accepting multiple remote connections) -> bind -> listen -> accept -> read/write -> close Client: socket(create port association) -> connect -> write/read -> close - stream semantics - Server has many incoming connections on the same port. Therefore it must distinguish between the incoming connections. Therefore, one listener socket is used and then another transmission socket is created for each connected client. This way we then can read and write from different distinct sockets and also close them independently. This is somewhat a dirty pattern, however, it is the currently used pattern.

UDP socket usage: Server: create socket -> loopreceive -> close Client: create socket -> send to address+port -> end or send new message. - no connection - integrity: somehow yes. UDP will throw errors based on checksum, but not correct the error or resubmit the message - validity not provided - -> reliability not provided - UPD is the basis for sophisticated transport protocols, which might be better than TCP for some use cases. - message semantics, not stream semantics

Other transport layers than UPD and TCP of today's age: - TCP is still by far the most common one - e.g. QUIC is another very fast transport layer protocol - IEFT Working Group TAPS works on a generic interface to transport layer. Such that programmers can use any transport protocol without changing their program code (accept maybe one variable determining the protocol)

SCTP (slide 25) DCCP (slide 26) - separate header and data checksum -> if only data checksum is corrupted. Therefore, we can error handle it depending on the intact header. Otherwise (like in TCP) we must just throw corrupted packets away because e.g. the sequence number might not be correct. QUIC - implemented in chrome, chromium and some other apps - Contrary to other proposed protocols, this sees more acceptance due to Google's big influence. - less overhead on connection establishment (fewer handshake messages) - Is faster, because linking directly into TSL - Intertwines with HTTP2 - On the contrary, if TSL or HTTP should be replaced, QUIC breaks, because it breaks the rules of layer separation.

2.0.3 *Message Queues*

2.1 DISTRIBUTED PROGRAMMING LANGUAGES

- not accepted in the past, maybe coming up in the future - **DRTS!** (**DRTS!**)

2.2 GLOSSARY

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

2.2.1 *Massively Parallel Programming*

MPP! (**MPP!**) is a programming model which makes enormous use of multi-threading 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.

2.2.2 *Computer Network*

A **CN!** is a set of **AS!**s 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.

2.2.3 *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!** (**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.