Introduction to Distributed Systems

Prof. Dr. Peter Braun

15. Oktober 2024

1 Titlepage

Introduction to Distributed Systems

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Goals of this Unit

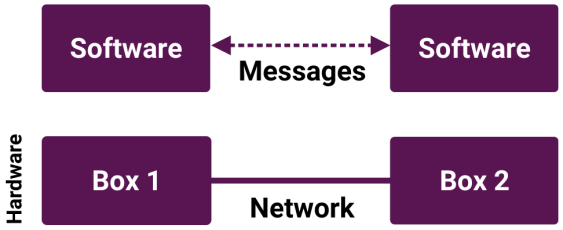
Goals of this Unit

- You know the architecture of client/server systems.
- You know the challenges of client/server systems.
- You know different types of communication.
- You know how to use TCP/IP sockets in Java.

Welcome to the first unit in this course about backend systems. We will start with an introduction to client-server systems, which is the kind of distributed system we will use in this course. We will first look at the architecture of client-server systems and consider the challenges we face when programming such systems. We will then practice programming such systems in the Java programming language. We will first look at the simple case of synchronous communication, and you will also be able to work out solutions yourself on some exercises. Finally, we will briefly look at failure semantics and asynchronous communication.

3 Distributed Systems

Distributed Systems

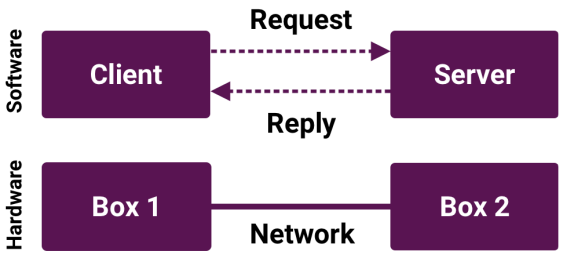


- Distributed system: hardware, network, and software
- For users, it appears to be a single system

What is a distributed system? Let's start at the lowest level. First, we deal with at least two physical or virtual computers. In the image next to me, we will name these two computers with the word "box."These two boxes are connected via a network. We still need software to make a system out of these two boxes, which only consist of some hardware and are connected via a network interface. Without the software, we only have two dump computers without the need to communicate and work together. The software defines the purpose of their working together. For example, they share resources. That would be a good use case. They could share processors and move processes between the two computers to perform some load balancing. Or they could share memory or disk space. Whatsoever. It is very important that without the software, that means, without a specific use case, there is no distributed system. On a very low level, the software can be the operating system itself.

4 Client/Server System (CS)

Client/Server System (CS)

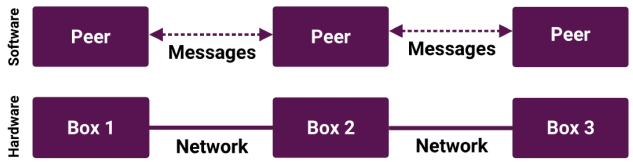


- Most popular pattern for distributed systems
- Roles are clearly distinguished between the nodes
- The client sends a request to a server
- The server answers with a reply message

Now, let's look at a special distributed system: the client/server system. We use client and server to describe the two computers' roles in this distributed system. Let's start with the server. The server, or the software running on the box, provides a service. Under a service, one can imagine everything. It can be a printer, which is physically connected to this computer and can be controlled by the software. However, it can also be a file system, and the software on the server can store, delete, and retrieve files. It can also be a service in the sense of an algorithm that accepts specific parameters and produces a particular output from these inputs. The computer that takes the role of the client is now the customer of this service. The software runs on this computer, which requires precisely the server's services for proper operation. From this role distribution, it is also clear that communication between client and server always starts from the client. The client sends a request to the server and waits for its response before continuing its operations. This distribution of roles between client and server sounds very simple at first, and this is precisely its great advantage. The distribution of roles is clear, and it does not change over the lifetime of the computers. Above all, the requirement to start a communication step on the client and never on the server makes programming such systems a bit easier. Nevertheless, programming is not trivial. In practice, we have many challenges in programming such a system, which must be considered.

5 Peer-to-Peer Systems (P2P)

Peer to Peer Systems (P2P)



- Tasks or data is partitioned between the peer nodes
- All peers perform the same tasks
- There is no need for central coordination

Another type of distributed system is the Peer-to-peer system. In contrast to client/server, where there is a hierarchy with different roles between the two types of nodes, all peers in a peer-to-peer system are equal or do more or less the same. Peer-to-peer systems are characterized by shared tasks or data between the nodes, the so-called peers. Each peer carries /part/ of the overall load, whether in the form of stored and managed data or computing tasks that need to be performed. This division results in a distributed structure where the entire network works as a unit to accomplish a specific task. Another characteristic feature of peer-to-peer systems is that all peers in the network perform the same tasks and provide and use services. For example, each peer can simultaneously download data from others and upload its data. This equality among peers makes the system very flexible and resilient, as each node can perform the same functions. A key aspect of P2P systems is the lack of central coordination. There is no central server to monitor or control the network, so that P2P networks can operate without a central authority. This increases reliability as the network functions even if individual peers fail. The peers coordinate independently and adapt dynamically to changes in the network, resulting in a robust and scalable system architecture.

6 Challenges of Client/Server Systems 1

Challenges of Client/Server Systems

- **Heterogeneity**: different hardware
- **Performance**: network communication needs time
- **Security**: confidentiality, integrity, availability
- **Failure** handling: detecting failures, masking failures
- **Scalability**: system handles high number of requests
- **Transparency**: concealment of the distribution

Let's take a closer look at some of the challenges. First, let's consider the challenge of heterogeneity in real-world systems. For example, imagine that the client is running on a Windows operating system, but the server is running on a Linux system. I'm sure you've all stumbled over the conventions for capitalizing file names or the characters that separate directories in a file path. Or imagine that the client and server are written in different programming languages. An integer data type in the C programming language doesn't necessarily mean the same as an integer data type in Python because the value ranges differ. Or consider different encodings for strings. In more hardware-oriented languages, strings are terminated with a null character, whereas in other languages, the length of the string is stored separately. Also, the character sets on the two computers may be different. When transferring data between client and server, these other data formats or encodings must be considered. A second challenge we have to deal with in client-server systems is the large performance area. Transferring data between client and server takes time, and even if we are using gigabit networks, it is still slower than sending data between two processes on the same computer. This challenge becomes even more significant when our system is distributed on different continents. In this case, sending a message can take several seconds. Therefore, when programming the client in particular, one must be prepared for such situations, where messages may arrive only after a long delay. Whenever we send messages over a network, security is, of course, a significant challenge. In the simplest case, we have a client and server connected in a local area network that we also have under complete control. But this is only a straightforward case. Suppose the client and server are organized in different networks and distributed worldwide. In that case, we cannot control what might happen to our data on the route between client and server. Therefore, we must protect messages against manipulation,

falsification, and simple eavesdropping.

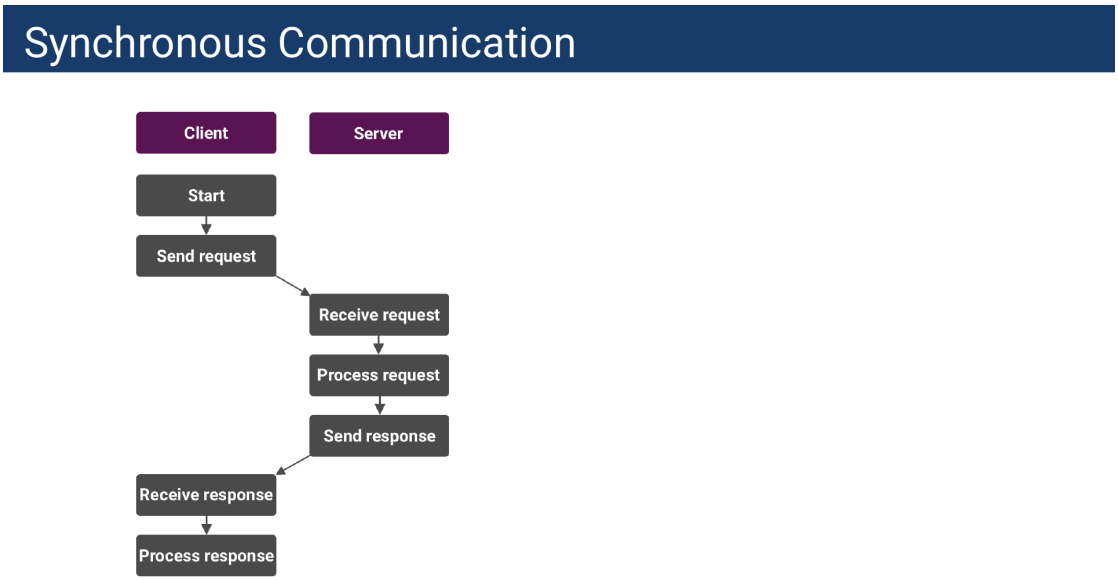
7 Challenges of Client/Server Systems 2

Challenges of Client/Server Systems

- **Heterogeneity**: different hardware
- **Performance**: network communication needs time
- **Security**: confidentiality, integrity, availability
- **Failure** handling: detecting failures, masking failures
- **Scalability**: system handles high number of requests
- **Transparency**: concealment of the distribution

We also have the challenge of many things failing in a client-server system. For example, the client may crash. This case is the most simple problem, where the server stops receiving messages from the client but is otherwise unaffected by the situation. More serious is when the network communication between client and server is disturbed or interrupted for longer. With this, the client-server system stops working because both parts depend on each other. The client cannot work without the server, and the server is relatively useless without requests from the client. Finally, the server can also crash, a serious problem because the client can no longer work reasonably without the server. In all of these three possible error cases, we as developers must first determine that there is currently an error and decide how to handle the error. This means that we have to define how much time a communication step between client and server may take and from which time we send an error message to the end user to report that either the network or the server is currently unavailable. It may also be helpful in certain error situations not to notify the end-user too quickly of a severe problem. We could consider what measures we can take to hide or overplay certain error cases. For example, we could compensate for a network connection that is interrupted only for a short time by having the end-user only work locally on the client for the time being. Later, after the network connection has been restored, the data is synchronized again with the server by the application itself, completely invisible to the end user. However, these are very complex algorithms that are necessary for such steps.

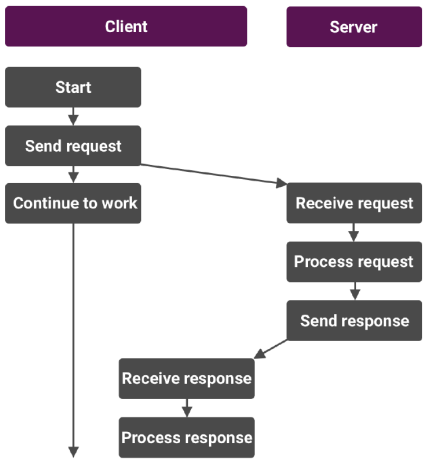
8 Synchronous Communication



we will look at the two most common client/server communication types. Let's start with the simplest case: synchronous communication. You can see in this graphic beside me the single processing steps that the client and server must go through. The client starts the application and tries to send a request to the server first. The arrow pointing from the box "Send request" to "Receive request" on the right side, indicates network communication. The arrow does not run horizontally but slightly downwards to symbolize the time delay, similar to a UML sequence diagram. The server then receives the message, the request encoded in the message is executed, and the server responds back to the client. While the server is busy processing the request, the client is waiting. You can easily imagine this as an active wait in which a loop constantly checks whether the response from the server has finally been received. Of course, this is not how it is solved in practice. On the operating system level, the process is put into a "blockedßtate, meaning it does not continue to work because it does not know what to execute. It waits for a signal that a response has arrived from the server. So, the process is sleeping during that time, but it still binds a thread or process on the client. Suppose the client has a graphical user interface where the end users can execute some events, such as pressing a button. In that case, it becomes clear that sleeping is not a good solution. The incoming events would not be processed while waiting, the graphical interface would not be updated, and the user would not get any feedback that pressing a button has any effect. So, synchronous communication should not be used with a graphical user interface for clients. But back to the client-server communication flow. When the response from the server arrives at the client, it is processed accordingly, and the communication step is finished.

9 Asynchronous Communication

Asynchronous Communication



Let's look at asynchronous communication to get around the problems with synchronous communication, especially in cases where the client has a graphical interface or interacts directly with the user in other ways. The communication step starts similarly. The client creates and sends a request to the server. But here comes the difference. The client is not put into a blocking state but can continue with other activities immediately. Waiting for the server's response takes place practically in parallel in a separate thread to further process the application on the client. Again, in the case of a graphical user interface, the client can now respond to clicks, and the end-user has the impression the application is still alive. In parallel, the request is processed on the server. At some point, the server sends the response to the client, and now the question is how the client processes the response. The solution is called a callback. When the client has sent the request to the server, it has already defined what function should be executed when the response arrives from the server. This function is now parallel with the application; the response is processed, and data must be updated on the graphical interface. When all this has been done, the processing of this function ends, and the thread is also finished.

10 Basics of Network Communication

Basics of Network Communication

- Every computer has an IP address e.g. `192.168.1.1`.
- DNS resolves names to IP addresses.
- Ports are endpoints of communication channels.
- Two computers cannot create more than one channel using the same ports.

In the following section, we will first concentrate on synchronous communication and look at how such straightforward communication via TCP/IP could be implemented in Java. First, we have to repeat some basics from network communications. We assume that every computer has a more or less unique IP address. We neglect such cases in which the computers are occasionally given new IP addresses. We also assume that the computer may have a name that can be mapped to the computer's IP address via a directory service such as DNS. With such an IP address, a client can already address a server, but this would mean that only one service can be offered on each computer. To distinguish different services on the same computer, we need another concept called ports. This means that there can now be several software systems on the server that wait for incoming requests entirely independently of one another on different ports. On the other hand, a client can select the service it wants to communicate by specifying the port number. The concept of ports is used not only on the server but also on the client. A port is used for each outgoing connection from the client. So, we have a logical connection between the client and server and the port on the client with the respective port on the server. There cannot be two simultaneous connections between the same port numbers on the client and the same port on the server. However, it is possible that several connections to different servers can be opened from one client. It is also possible that a server can handle multiple connections from various clients on the same port.

11 Sockets in Java

Sockets in Java

- `Socket` for clients
- `ServerSocket` for servers
- `INetAddress` represents the name or IP address

To program a simple TCP/IP connection in Java, we now need the concept of sockets. A socket abstracts from the concrete physical connection and represents a logical endpoint consisting of the IP address and the port. Therefore, a connection between client and server requires a socket on the client and a socket on the server. Java provides classes for this purpose; on the client, the class is called Socket, and on the server, it is called ServerSocket. In Java, an IP address is not just a string; there is a class called INetAddress. Now, let's look at an example of realizing a simple TCP/IP connection via sockets between a client and a server in Java.

12 Screencast: Intro to Source Code

- todo

Okay, I would like to show you the source code of this very simple TCPIP socket communication between a client and a server. We start with a server. This is implemented in a class. We have a main method here. And the first thing that we do here is to open a server socket on this port here. And then we start an infinite while loop. So we want the server to run forever. The first statement in this while loop is this accept method here on the server socket. And this is a blocking or waiting statement. So at this point here, the server will wait for an incoming request from a client. The data, so as soon as there is a message or some data, this accept method returns with a socket. And then we can read from this input stream here the incoming message. We wrap this input stream here in an input stream reader and this in a buffered reader because we want to read text files in a more convenient way here by using this method read line. So just for information, we output here the incoming message on the screen. This here is the, let's say the function that our server offers. It transforms the incoming message to uppercase letters. Then we create an out stream by asking the socket for the output stream, wrapping this again to be able to write some text. We write the message here. Then we append the line feed symbol. We flush the stream. I think you should know that from your introduction to Java, when you write data into a stream, you have to flush it to mark that the stream is like full. And we close the output stream and the input stream here. Then we close the socket and then the loop begins again by waiting for the next message from the client. So let's switch to the client. Again this is a class. We have a main method here and we wrap the system in stream, so the keyboard input into a stream reader into a buffered reader. We open the socket on localhost on this port. That's the port that we used for the server to listen for incoming requests. And we have the output stream into the socket. And we here, in this line here, we ask for the input stream to later be able to read the response.

So here we wait for the input from the user. Then we send this as a message with a final backslash n here, line feed symbol. We flush the output stream. And then here we wait for the input stream. And this is here then printed to the output console. And then the method main terminates.

13 Get this Running

Get this Running

- Clone the code from GitLab
- Import the project as *Maven project* in your IDE

- Start method `main` in class `TcpServer` first.
- After that start method `main` in class `TcpClient`.

- You have to type in some characters on the console.
- Check the result from the server.

So, now it's your turn to get the first client-server system up and running. Please clone the GitLab repository, which you can reach by clicking the button next to me. Import the directory as a Maven project into your development environment and look at the two classes. It is exactly the source code we discussed in the last live demo. First, run the Main method from the TCPServer class and then run the main method from the TCP Client class. Then, type a few letters on the console and look at the result.

14 Screencast 1

So I hope it worked on your computer. Let's briefly demonstrate how it should work. We start the server. And next we start the client. And now the client here waits for an input. We just type in hello and the server response is hello in capital letters. So this is how it should work.

15 Exercise 1: Send many messages

Exercise 1: Send many messages

- Let's refactor the code
- Create a new method for sending messages
- Send at least two messages by calling this method

In the first exercise, we want to improve the code so that it sends not only one message but many. We move the code to send a message to a method we can call multiple times from the client's Main method. Please try to implement this. When you are done, please click the "Continue" button, and we will discuss the solution together.

16 Screencast 2

So, I want to show you the solution for this exercise. We are here in class TCP Client. The main method only contains two method calls here of this new method sendMessage. The parameter is the text of the message that we want to send to the server. And we immediately output here the return value, the response message. The method sendMessage now is new. That was the former main method here. And we have a parameter message. The rest of the code is identical, except of the return statement here. Instead of just printing out the response to the console, we return the message here. And the server did not change at all. When we execute this, we first have to start the server. And then we start the client. And you can see here the response from the server.

17 Exercise 2: Add a second parameter to the message

Exercise 2: Add a second parameter to the message

- So far, a `String` is sent as the message payload
- Change the source of the client and server to send/receive a `String` and an `Integer` value.

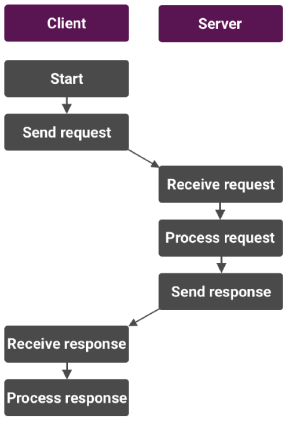
In the second task, we want to try to send a message consisting of two parameters: a single String parameter and a second integer parameter. Please think about what needs to be changed in the source code to send a message with two parameters. Click on the "Continue" button to see the solution.

18 Screencast 3

So your task was to add a second parameter, so the message that the client sends to the server should not only contain a text part but also a number, an integer value. And we separate this very easily here, so the message that we sent to the server contains two parts and we separate the text part and the integer part by using this hash symbol here. And that's the key point. So whenever we send more than one element in our message, we must inform the receiver, the server in this case, about where does the second parameter start. And we go for the hash symbol, we could have used any other symbol as well. The important point of course is that this is unique, so the hash symbol, we expect the hash symbol to be unique. If we had a problem, if the hash symbol would be part of the text file here or the text string here. So and the server looks like this. The server receives here the input, then we split the input using the hash symbol as the separator character, and then we interpret the number, we pass this as an integer value, and then we return the text file in the response message as many times as often as the number was given here. So let's start the server. And we start the client. And you can see the result here. I open the source code of the client here. So we send James Bond 3. That means we want to have this name here three times. So 1, 2, 3, and Hans Blufeldt five times 1, 2, 3, 4, 5. So that's it.

19 Introduction to Failure Semantics

Introduction to Failure Semantics



The next topic we must talk about is failure semantics. This means we must think about the possible failure situations and how to deal with them. Failures can happen; we already mentioned this earlier in this unit. It is one of the challenges in distributed systems. Failures can be less or more severe depending on the use case or the application. Failure Semantics refers to the rules and expectations defining a system's behavior when its components fail. It outlines the possible ways in which failures can occur, how they are detected, and the actions the system should take in response. In distributed systems, where multiple components interact across networks, failures are not just possible but inevitable due to the environment's complexity and unpredictability. These failures can range from simple crashes to message loss. In the first step, please check the picture beside me again and identify where an error can occur. What can go wrong? Please press the button if you want to continue.

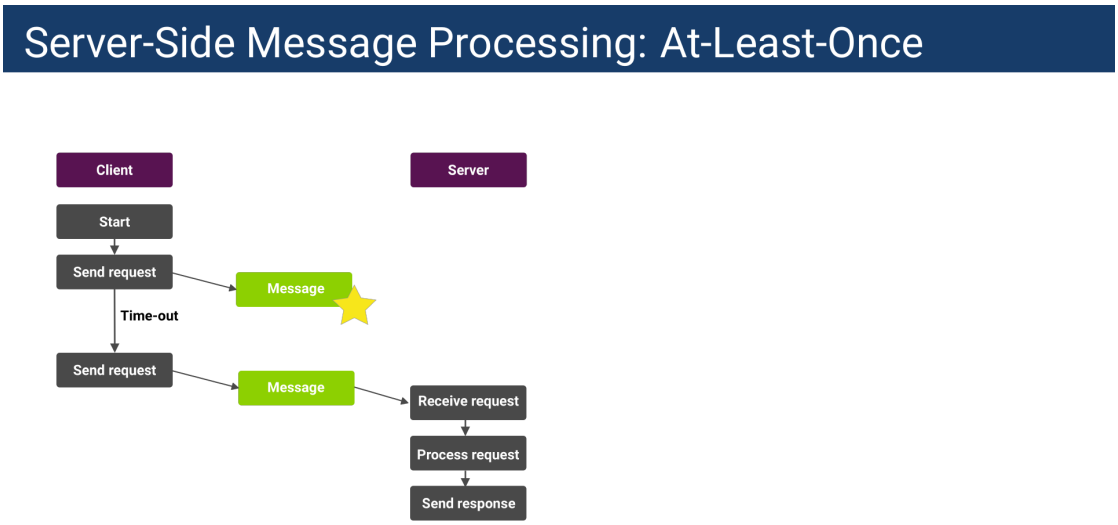
20 All Possible Failure Situations

All Possible Failure Situations

- Server crashes before the request is received
- Request message is lost
- Server crashes during processing of the request
- Reply message is lost
- Client crashes before the response is received

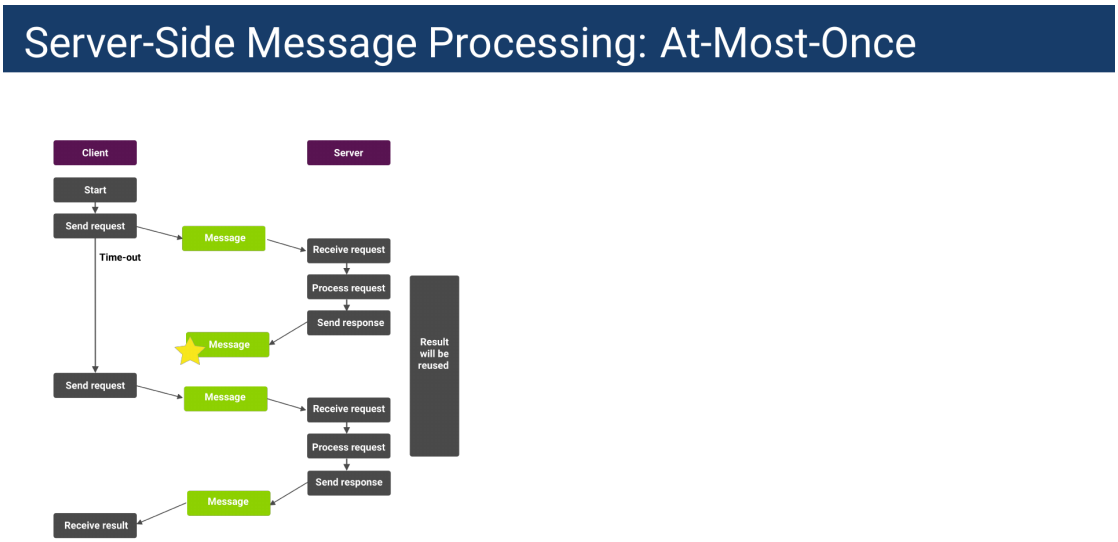
Here are five common failure scenarios: The server crashes before receiving the request. The server is unavailable in this scenario before processing a client's request. From the client's perspective, it might appear as though the server is unresponsive. Depending on the system's design, the client may retry the request or failover to another server. The second one is "Request message is lost." Network issues or misconfigurations can lead to the loss of a request message before it reaches the server. The client might not receive any acknowledgment, leading it to believe the request was ignored. To mitigate this, systems often implement timeouts and retransmission strategies, where the client resends the request after a certain period. The third one is "Server crashes during processing of the request." This situation is particularly challenging as the server might have partially completed the request before crashing. Without proper handling, this can lead to inconsistencies, such as partially updated data. Implementing transactional mechanisms or idempotent operations helps ensure the request can be safely retried or rolled back without adverse effects. The next one is "Reply message is lost." Even after successful processing, the server's response may be lost in transit back to the client. The client might then incorrectly assume that the server failed to process the request. As with lost requests, retries and acknowledgment mechanisms are commonly employed to handle this failure scenario. The last one is "Client crashes before the response is received," If the client crashes before receiving the server's reply, the server might complete the task, but the client is unaware of the result. When the client restarts, it may repeat the request, potentially leading to duplicate operations. Designing idempotent requests ensures that repeated requests don't

21 Server-Side Message Processing: At-Least-Once



Now, that we know about all possible failure situations, we must consider handling them. Let's first think about the optimal solution before we come to the one shown in the picture beside me. The optimal case would be that every operation is processed exactly once, ensuring no duplicate executions or missed requests. This would eliminate issues such as duplicated transactions or missed operations. But is this possible to achieve? Given the complexities of distributed systems, we often settle for ät-least-onceßemantics, where the system guarantees that each operation will be executed at least once. In this model, the client may resend a request if it does not receive a timely response, assuming the initial request might have been lost or the server crashed before processing. You can see this case in the figure beside me. The first request is lost. The client gets a timeout and resends the message as often as possible. While this approach ensures that no request is lost, it does come with the risk of processing the same request on the server multiple times, especially if the system does not have mechanisms to detect and discard duplicates.

22 Server-Side Message Processing: At-Most-Once



At-most-once semantics is a delivery guarantee in distributed systems that ensures an operation is executed either once or not at all. This approach is particularly valuable when it is crucial to avoid the potential side effects of duplicate operations, such as double payments or repeated updates. The server must store the results of previously processed requests to implement at-most-once semantics effectively. When a request arrives, the server checks if it has already processed it. This is typically done using a unique identifier associated with each request. If the request has been processed before, the server reuses the stored result and returns it to the client, avoiding re-executing the operation. If the request is new, the server processes it, stores the result, and returns the outcome to the client. One point we leave open: please think about this yourself: When does the server delete the stored results?

23 Failure Semantics

Failure Semantic

Time-outs are used to detect failure situations

- *Maybe* – no guarantee at all
- *At-least-once* – but possibly more
- *At-most-once* – but possibly none at all
- *Exactly-once* – in general impossible to achieve

There is a difference between local and remote execution!

In distributed systems, timeouts are a crucial mechanism for detecting failure situations. When a client sends a request to a server or another component, it expects a response within a certain timeframe. A timeout occurs if no response is received within this time, indicating a potential failure. However, timeouts can only signal that something might have gone wrong without providing certainty about the nature of the failure. Different delivery guarantees can be affected by time-outs. “Maybe”: In this scenario, a request is not guaranteed to be delivered or processed. The client is still determining if the server even received the request or if the operation was executed. This is the least reliable delivery model and offers minimal assurance. “At-least-once”: Here, the system guarantees that the operation will be performed at least once, but there’s a risk of being executed multiple times due to retries after a time-out. This can lead to issues like duplicated transactions if the operation is not idempotent. “At-most-once”: The operation is guaranteed to be executed either once or not at all. If a time-out occurs, the operation might have been performed or failed silently, leading to uncertainty. “Exactly-once”: Achieving exactly-once delivery is generally considered impossible in distributed systems, especially in the presence of failures. While it is an ideal scenario, the complexity of ensuring an operation is performed exactly once without duplication or omission makes it unfeasible in most practical systems. The reason can be put: The server can crash. We learn from this that there is a significant difference between local and remote execution. Local operations, performed within the same machine or tightly coupled environment, generally have lower latency and higher reliability. In contrast, remote execution involves communication over a network, introducing potential delays, packet loss, and other failures. This makes time-outs and delivery guarantees far more critical and challenging to manage. We can make an effort to overcome this distribution through new programming paradigms,

but we cannot completely hide it.

24 Summary

Summary

- You know the architecture of client/server systems.
- You learned two types of communication.
- You learned how to use TCP/IP sockets in Java.

That's it for today. I appreciate your interest in the introduction to programming a simple client-server system in Java. Today, you have learned about two types of communication: synchronous and asynchronous. You have implemented the first examples of sending messages over a socket from a client to a server and back again. This is indeed how such distributed systems were implemented in the very beginning. But not for long, as you can imagine; it's too complex and low-level. We are looking for more sophisticated techniques that can be used in more complicated situations. Please also remember what we learned about failure semantics. In the next unit, we will look at methods of making client-server programming easier for us developers.