CAP Theorem

Prof. Dr. Peter Braun

4. Oktober 2024

1 Titlepage

CAP Theorem

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

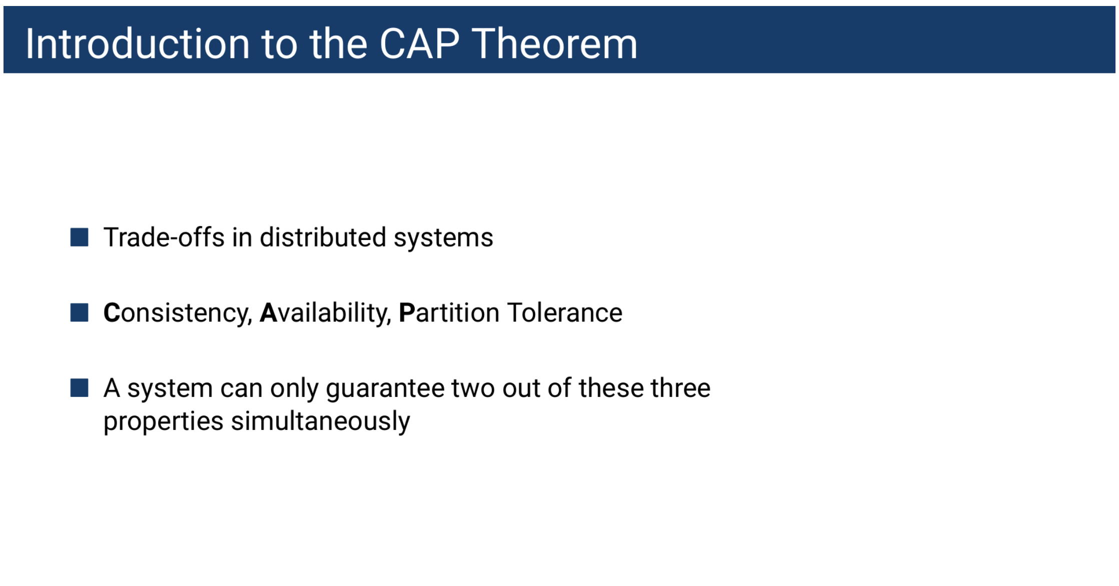
2 Learning Goals

Learning Goals

- Understand the trade-offs in distributed systems
- Grasp the key properties of the CAP Theorem
- Apply the CAP Theorem to real-world scenarios
- Analyze the consequences of choosing properties

Welcome to this unit about the CAP theorem. Today, we focus on an apparently theoretical aspect of distributed systems with tremendous practical implications. We study the boundaries of distributed systems. We will learn that we can only achieve some things in our list of requirements, and we have to make trade-offs. I will introduce the so-called CAP theorem. The three letters stand for the requirements we are studying today: Consistency, Availability, and Partition Tolerance. I will introduce these three requirements and discuss their real-world applications.

3 Introduction to the CAP Theorem



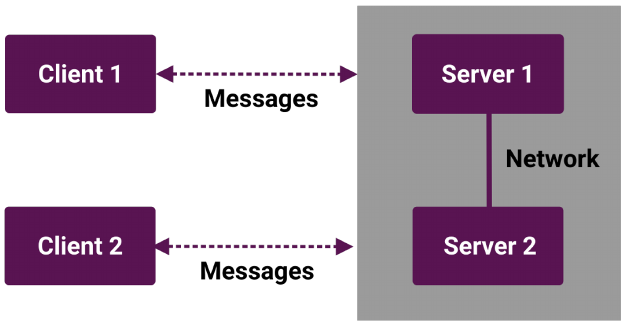
Introduction to the CAP Theorem

- Trade-offs in distributed systems
- **C**onsistency, **A**vailability, **P**artition Tolerance
- A system can only guarantee two out of these three properties simultaneously

Fundamentally, the CAP theorem is about understanding the trade-offs in distributed systems. At its core, it's about priorities and making decisions based on your application's needs. // The theorem is an acronym, C-A-P, and each letter represents a crucial aspect of distributed systems. The 'C' in CAP stands for Consistency. This means all nodes in a system see the same data at the same time. Next, we have 'A', which stands for Availability. This refers to ensuring every request gets a response, even if part of the system is down. The 'P' stands for Partition Tolerance. This attribute allows the system to continue functioning even when part of the system cannot communicate with the rest. All three aspects sound extremely important we would love to have a system that guarantees all three aspects, right? However, herein lies the catch, the CAP theorem states that a distributed system can only ever guarantee two out of these three properties at the same time. Let's create an example.

4 Scenario

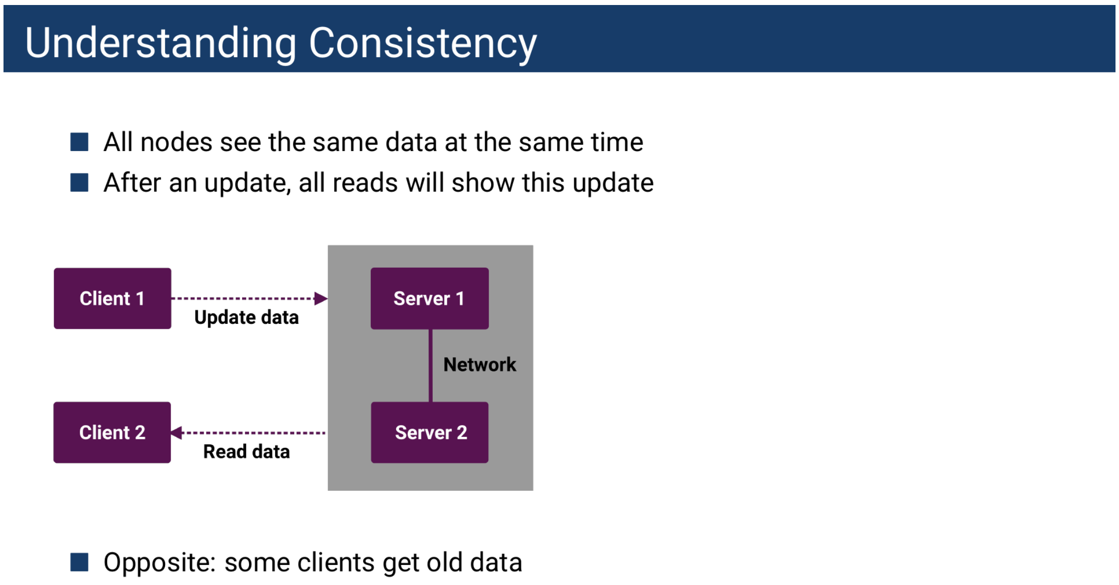
Scenario



- The two servers form a system (cluster)
- Clients communicate to the system

Let's examine a scenario to better understand the CAP Theorem. Imagine that we have two servers. Together, these servers form a system, also known as a cluster. These servers are interconnected and in constant communication with each other. This cluster of servers may be used for various purposes, including data storage, processing information, running applications, and so on. Now, some clients interact with this system. A client can be any entity that needs the services of our server cluster, such as a website user, a mobile app, or even another system. These clients communicate with our system, request information, send commands, and possibly even update data. We have a client/server scenario, as shown in the last two units. Let's now introduce the three requirements. We start with consistency.

5 Understanding Consistency


Understanding Consistency

- All nodes see the same data at the same time
- After an update, all reads will show this update

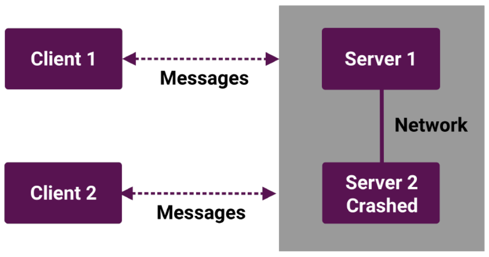
- Opposite: some clients get old data

When we talk about 'Consistency,' it means all of these nodes are seeing and presenting the same data at the same time. Imagine an update from client 1 - an additional row added to a table, a line of text deleted or modified. In a consistent system, all subsequent reads from any node will show this update immediately after an update. Every single node is updated simultaneously with the new information. When client 2 asks for exactly this data, it will receive the updated information. No matter if client one communicates with server one and client 2 communicates with server 2. However, this perfect 'Consistency' in the real world can sometimes be difficult to achieve. Due to delays in communication or other technical issues, certain nodes might lag in updating the data. This is the exact opposite of what we want from consistency. In such an event, some clients may still get the old data instead of the updated one. It may be only for a second, but it can cause issues, especially if decisions are made based on that data. To sum up, 'Consistency' in the context of the CAP Theorem is about ensuring that all nodes in the network see the same data at the same time. It's about every single read reflecting the most recent update. Contrary to this, it's when some clients get the old data.

6 Understanding Availability

Understanding Availability

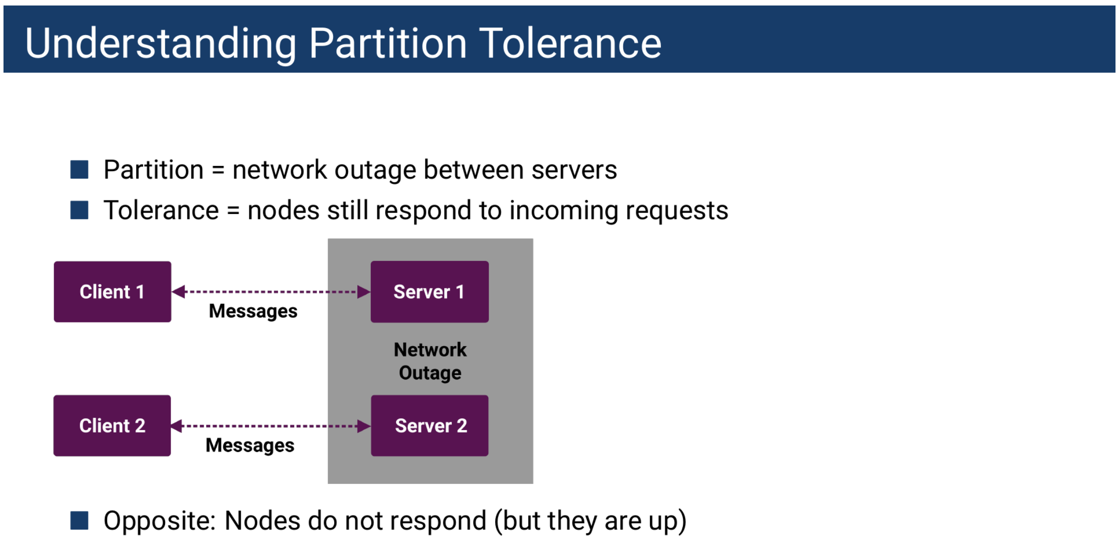
- The system is always up and responsive
- Every request receives a (non-error) response
- Even if some nodes don't work anymore



- Opposite: Clients don't get a response

Next, we will discuss "availability." In simple words, "availability" means that our system is always up and ready to work, no matter the time, the date, or the situation. The system is running for 24 hours straight. Now, in the lens of the CAP theorem, availability has a more specific interpretation. Here, availability means that the system delivers a response for every "request" that the system gets—every time, regardless of the internal status of the system. What if some nodes in our system stop working? With availability, the "system" will continue responding because another server can take over. In the picture beside me, server two crashes, but server one is still running and will deliver responses. This only works if enough servers are available for the expected number of requests. If the overall system load is too high, the system becomes unavailable. Let's look at the opposite for a moment. When clients do not get a response, the system is unavailable only if one server in the cluster crashes. We strive to avoid this scenario when we aim for high system availability.

7 Understanding Partition Tolerance



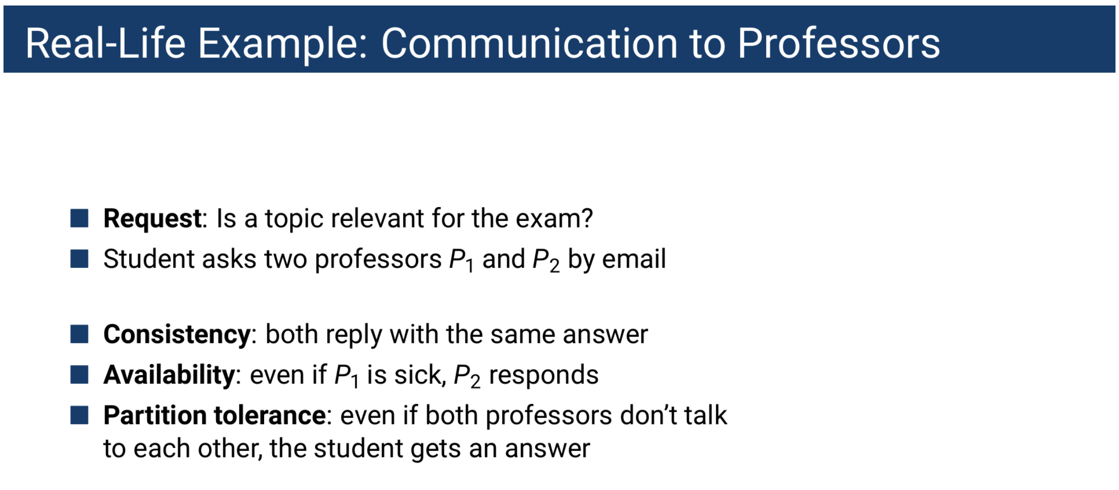
Understanding Partition Tolerance

- Partition = network outage between servers
- Tolerance = nodes still respond to incoming requests

- Opposite: Nodes do not respond (but they are up)

The last requirement is called "Partition Tolerance." Think of a network of servers where all the nodes, or individual servers, are active. They are up and running, ready to respond to any incoming tasks. It's an ideal functioning system. But what if suddenly there's a network outage? What if these servers can't talk to each other anymore? This is what we call a "Partition." The network breaks into single islands. Now, let's talk about "Tolerance". The word "tolerance" typically denotes endurance. In this context, tolerance is the system's ability to endure network splits. Despite a partition, every node stays active. This means the servers will continue to respond to incoming requests, even if they have become isolated due to the network split. Let's discuss the opposite again to make it clearer. In case of a network outage, which the servers can recognize, all servers individually decide to stop working. They are still up and running but deliberately choose not to answer any requests because the overall system has an error.

8 Real-Life Example: Communication to Professors



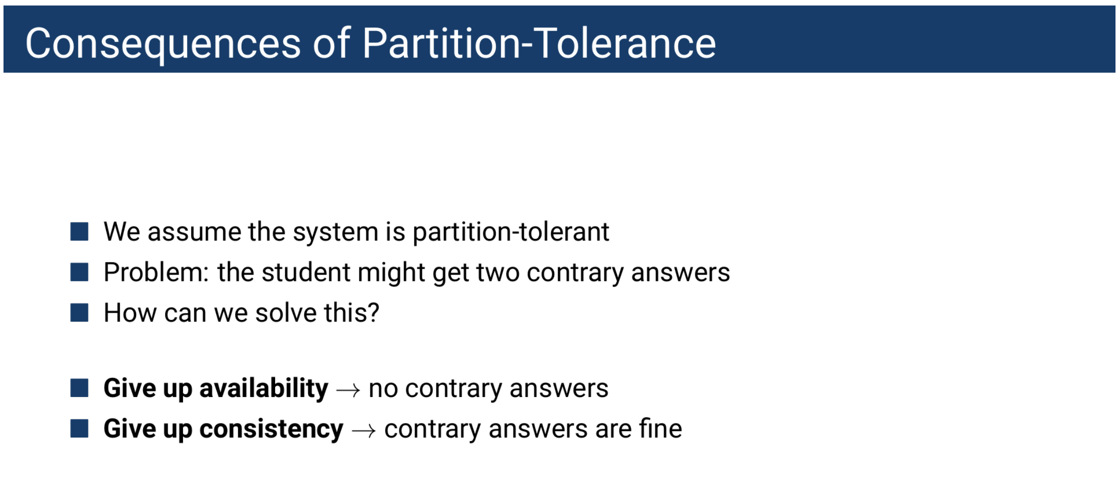
Real-Life Example: Communication to Professors

- **Request**: Is a topic relevant for the exam?
- Student asks two professors P_1 and P_2 by email

- **Consistency**: both reply with the same answer
- **Availability**: even if P_1 is sick, P_2 responds
- **Partition tolerance**: even if both professors don't talk to each other, the student gets an answer

Let's look at a real-life example illustrating how the CAP theorem works. Imagine you're a student preparing for an exam, and you need to be certain about the relevance of a certain topic. So, you decide to email your two professors, we'll call them Professor P1 and Professor P2. These two professors form the "system." Now, let's define the CAP theorem in this context. You make a "request,äsking whether this topic is relevant for the exam. Next comes "consistency". You're hoping to get the same answer from both professors, ensuring that the information you receive is consistent, regardless of the source. Then, there's ävailability.Ëven if Professor P1 is ill and unable to reply, you can still receive a response from Professor P2, indicating that the system is still available despite the failure of a single component. Lastly, we have "partition tolerance.Ïn an ideal scenario, the professors or the nodes in this network would communicate to ensure they both provide a consistent answer. However, even if both professors can't or don't communicate with each other, you can still get a response to your request. This is partition tolerance, the system's ability to function despite network link failures.

9 Consequences of Partition-Tolerance



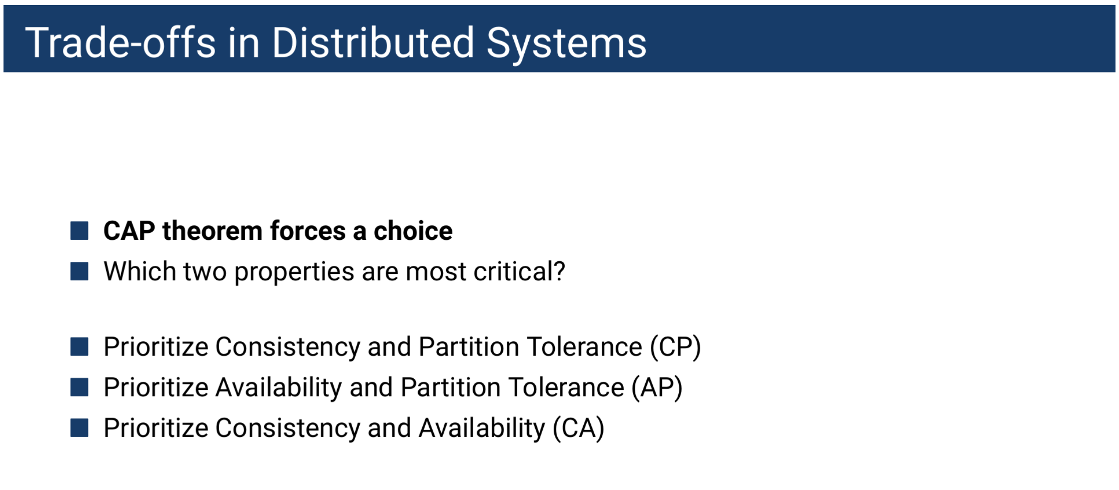
Consequences of Partition-Tolerance

- We assume the system is partition-tolerant
- Problem: the student might get two contrary answers
- How can we solve this?

- **Give up availability** → no contrary answers
- **Give up consistency** → contrary answers are fine

We will not discuss the implications of these requirements. We start with “Partition tolerance.” Let’s assume the system we’re working with is partition-tolerant. This means that clients will get answers even in case of a network outage. However, this causes a peculiar problem. You could get two differing responses with multiple nodes simultaneously serving your requests. Internally, the servers cannot coordinate because of the network outage. So, how do we solve this puzzle? There are two main paths we can follow. Both, however, require us to make concessions. It’s a question of compromise - what are we willing to give up to ensure a smoother operation? On the one hand, we can choose to ’give up availability.’ By sacrificing immediate response to your requests, we ensure that all answers will be consistent—no more contradictions. If the servers cannot coordinate, they stop working, and the system is no longer available. On the other hand, we can choose to ’give up consistency.’ Here, we are fine with possible contrary responses; our priority is maintaining quick responses. The system is still available but not consistent anymore. Please think for yourself about the other two scenarios. Assume that we persist on Consistency. We must give up either availability or partition tolerance. When we persist on Availability, we must give up consistency or partition tolerance.

10 Trade-offs in Distributed Systems



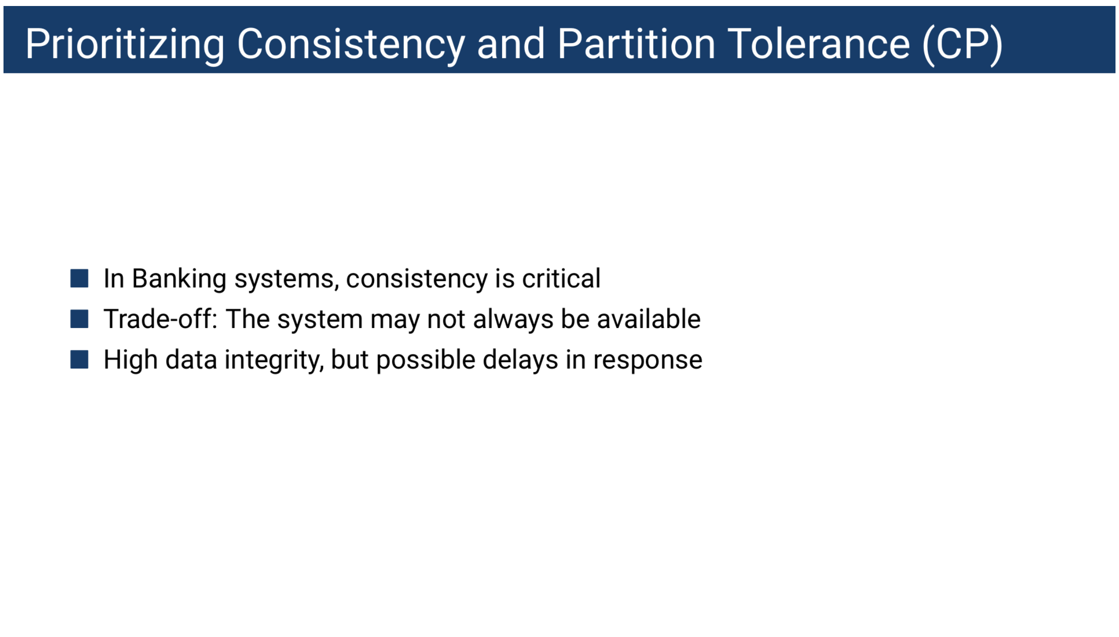
Trade-offs in Distributed Systems

- **CAP theorem forces a choice**
- Which two properties are most critical?

- Prioritize Consistency and Partition Tolerance (CP)
- Prioritize Availability and Partition Tolerance (AP)
- Prioritize Consistency and Availability (CA)

Regarding distributed systems, choices have to be made due to the CAP theorem. The CAP theorem states that a distributed data system can only simultaneously provide up to two of the three requirements: Consistency, Availability, and Partition tolerance. Now, the question arises: which two properties are most crucial to prioritize? We name the three possible options: consistency and Partition tolerance (CP), availability and partition tolerance (AP), and Consistency and Availability (CA). Let's now look at some practical examples.

11 Prioritizing Consistency and Partition Tolerance (CP)

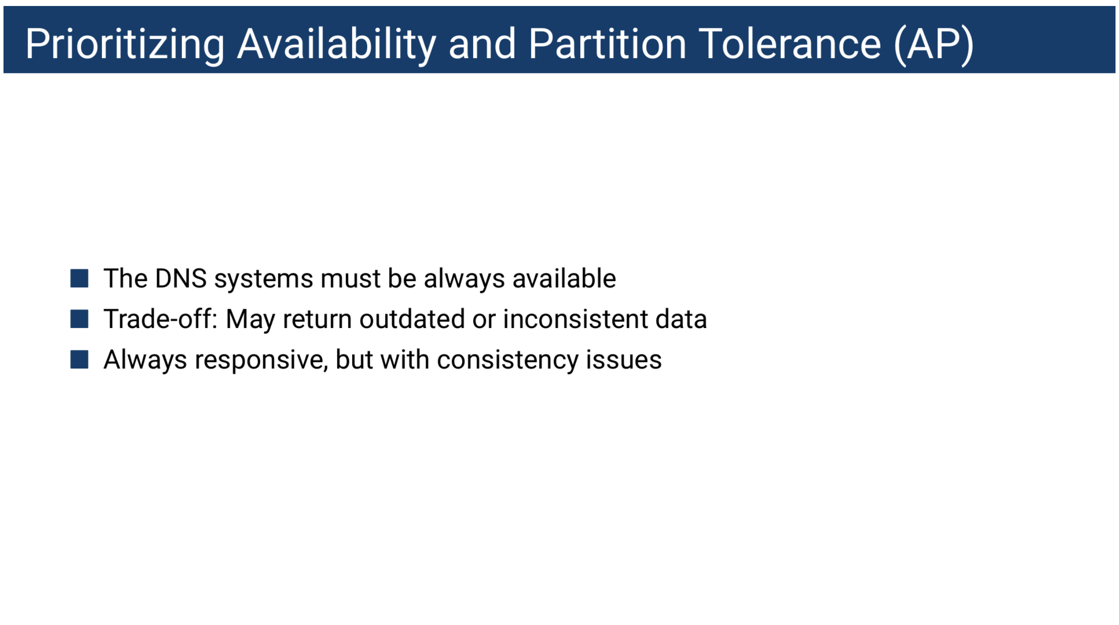


Prioritizing Consistency and Partition Tolerance (CP)

- In Banking systems, consistency is critical
- Trade-off: The system may not always be available
- High data integrity, but possible delays in response

We're focusing on when a system prioritizes Consistency and Partition Tolerance, often abbreviated as CP. Let's depict this concept with a practical example. In banking systems, consistency is critical. All transactions must be recorded accurately. For instance, if you transfer money from one account to another, the amount must decrease from one account and increase in the other. It would be a catastrophe if anything goes wrong in the transaction. However, achieving this impeccable level of consistency comes with a trade-off. In a CP system, the system is partition tolerant. This means, and this is important now, that as long as consistency is not violated, the system will continue to work. Consistency is not an issue when two people ask for their bank balance, for example. However, the system is not 100

12 Prioritizing Availability and Partition Tolerance (AP)

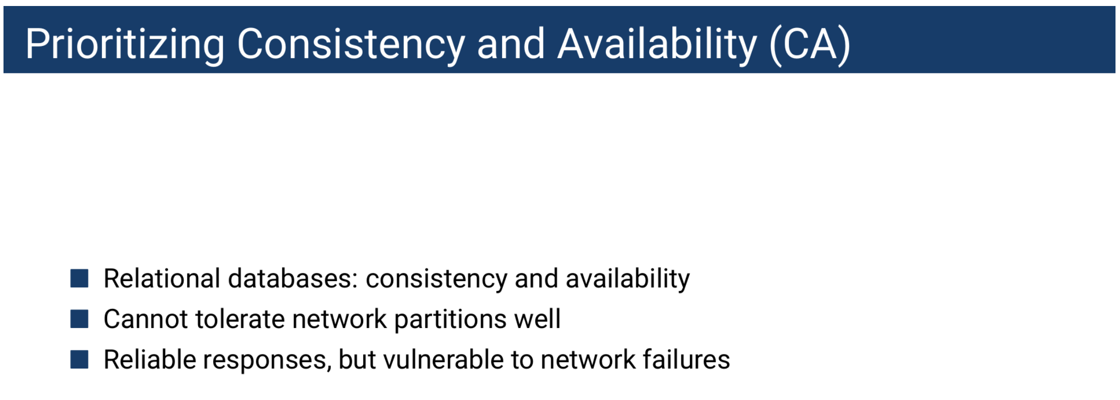


Prioritizing Availability and Partition Tolerance (AP)

- The DNS systems must be always available
- Trade-off: May return outdated or inconsistent data
- Always responsive, but with consistency issues

Next, we discuss an example of an AP system. The Domain Name System DNS converts human-readable addresses into IP addresses. Because of its critical role, DNS systems need to be always available. Picture trying to reach your favorite website but getting an error because the name could not be resolved. This situation must not occur. However, this comes with a trade-off. In a partitioned scenario, when the system is splitting into separate areas, it may return outdated or inconsistent data because it prioritizes constant availability. For example, an update to the name mapping to IP address might be available on one DNS server but only on some.

13 Prioritizing Consistency and Availability (CA)

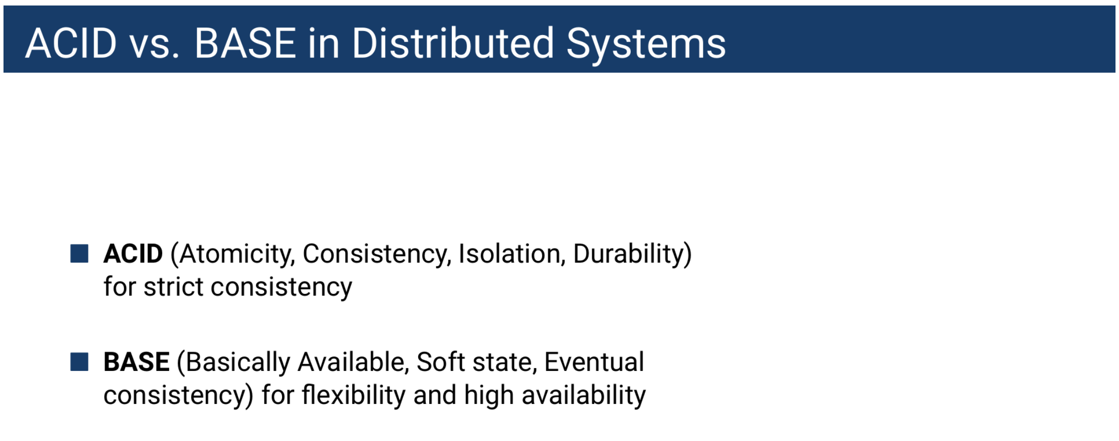


Prioritizing Consistency and Availability (CA)

- Relational databases: consistency and availability
- Cannot tolerate network partitions well
- Reliable responses, but vulnerable to network failures

The last combination is Consistency and Availability, which largely applies to relational databases. In these databases, emphasis is placed on ensuring both consistency and availability. In a network outage in the cluster, the system does not guarantee that it will work correctly. In other words, the system requires the network to work and is not prepared for a network outage.

14 ACID vs. BASE in Distributed Systems



ACID vs. BASE in Distributed Systems

- **ACID** (Atomicity, Consistency, Isolation, Durability) for strict consistency
- **BASE** (Basically Available, Soft state, Eventual consistency) for flexibility and high availability

Finally, I want to connect the CAP theorem to a topic you are aware of from database management systems: the ACID principle. ACID refers to Ato'micity, Consistency, Isolation, and Durability. This is largely about strict consistency. 'Atomicity' means that operations within a transaction are treated as one indivisible unit. Either all are executed, or none of them are. There's no in-between. 'Consistency' ensures that a transaction brings the database from one valid state to another. 'Isolation' ensures that simultaneous transactions don't impact each other's execution. And finally, 'Durability' guarantees that once a transaction is committed, it will stay so, remaining unaffected by software or system failures. The ACID principle is extremely important for many kinds of data, such as bank accounts and money transfers. ACID comes at some cost. As we have learned from the CAP theorem, when consistency is key, the system can only be available or partition-tolerant, but not both. With the rise of extremely large backend systems at Facebook or Google, the question of whether consistency is always necessary or if it is possible to balance consistency against other requirements came up. BASE stands for 'Basically Available, Soft state, Eventual consistency', a model designed for flexibility and high availability. It must be seen in contrast to the ACID principle. Eventually, consistency means that an update will be visible for all clients at some point in the future, but not immediately. For example, you can observe this eventual consistency when comparing comments to social media posts with two devices from two different users. One user might see more comments than the other because both requests were answered by different servers in the backend.

15 Summary

Summary

- The CAP theorem is a fundamental concept
- Always consider the specific requirements

- **No one-size-fits-all solution**

That's it for today. Let's wrap things up and summarize what we've covered around the CAP theorem. This theorem is a foundational concept for designing distributed systems. It offers a framework to understand the key trade-offs involved. The theorem gives us three important aspects - consistency, availability, and partition tolerance. But it also comes with the stark reality that simultaneously satisfying all three is impossible. We have to pick two out of three based on our project needs, which leads to three possible combinations. Next, it's important to highlight that we must always consider the specific requirements and the context of the system. You can't just apply the CAP theorem blindly. Think about what your system is meant to do. If it's a banking system, we might favor consistency over availability. We might favor availability over consistency if it's a social media platform. Keep the context firmly in mind when deciding on your trade-offs. Lastly, remember that there is no universal 'one-size-fits-all' solution. The decisions you make while designing your distributed system will involve trade-offs. The real challenge is balancing these trade-offs for the best outcome.