CS550-WA3

WeiboWang A2050351 wwang136@hawk.iit.edu

1.*Outline an efficient implementation of globally unique identifiers.*

An efficient implementation of globally unique identifiers (GUIDs) should meet the following requirements:

- **Uniqueness:** GUIDs must be unique across all systems and applications.
- **Efficiency:** GUIDs should be generated and validated efficiently.
- **Scalability:** The GUID generation and validation system should be able to handle a large number of requests.

One way to implement GUIDs efficiently is to use a centralized counter. This counter is maintained by a single server and is used to generate new GUIDs. When a client needs a new GUID, it requests one from the server. The server increments the counter and returns the new GUID to the client.

To ensure uniqueness, the counter must be synchronized across all servers. This can be done using a variety of methods, such as distributed consensus algorithms or replication.

To improve efficiency, the server can generate batches of GUIDs in advance. This way, the server can respond to client requests immediately without having to increment the counter each time.

The following is a high-level overview of an efficient implementation of GUIDs using a centralized counter:

1. **Initialize the counter.** This can be done by setting it to a random value or by using a timestamp.
2. **Generate a new GUID.** To do this, increment the counter and return the new value as a GUID.
3. **Validate a GUID.** To do this, check if the GUID is within the range of valid GUIDs.

This approach is scalable because the centralized server can be replicated to handle a large number of requests.

Another way to implement GUIDs efficiently is to use a distributed hash table (DHT). A DHT is a distributed data structure that can be used to store and retrieve data efficiently.

To generate a GUID using a DHT, the client generates a random hash value and then uses the DHT to lookup the corresponding GUID. If the GUID does not exist, the client creates a new GUID and stores it in the DHT.

To validate a GUID, the client looks up the GUID in the DHT. If the GUID exists, then the client knows that the GUID is valid.

This approach is also scalable because the DHT can be distributed across multiple servers.

Which approach is best for a particular application depends on a number of factors, such as the required level of scalability and the performance requirements.

2.Name at least three sources of delay that can be introduced between broadcasting

the time over the network and the processors in a distributed system setting their internal

clocks.

Here are three sources of delay that can be introduced between broadcasting the time over the network and the processors in a distributed system setting their internal clocks:

1. Network latency

Network latency is the time it takes for a packet to travel from one network node to another. This latency can be caused by a number of factors, such as the distance between the nodes, the type of network medium used, and the load on the network.

2. Clock drift

Clock drift is the difference between the rates at which two clocks tick. Even very accurate clocks will drift over time, and this drift can accumulate over long distances.

3. Processing delays

Processing delays are the time it takes for the processors in the distributed system to receive the time broadcast and to set their internal clocks. This delay can vary depending on the hardware and software used by the processors.

3.*Consider the behavior of two machines in a distributed system. Both have clocks that*

are supposed to tick 1000 times per millisecond. One of them actually does, but the other ticks

only 990 times per millisecond. If UTC updates come in once a minute, what is the maximum

clock skew that will occur?

If the two machines tick 1000 times per millisecond and 990 times per millisecond, respectively, then the slower machine will fall behind by 10 ticks per millisecond. This means that the slower machine will lose 600 ticks per minute.

If UTC updates come in once a minute, then the maximum clock skew that will occur is 600 ticks, or 600 milliseconds. This is because the slower machine can lose up to 600 ticks before the next UTC update comes in and corrects the skew.

In practice, the clock skew will likely be less than 600 milliseconds. This is because the UTC updates will help to keep the clocks synchronized, even if the clocks are ticking at different rates.

Here is an example of how the clock skew can accumulate:

```
Time | Faster Machine | Slower Machine
-------|----------------|---------------
00:00:00 | 0 | 0
00:00:01 | 1000 | 990
00:00:02 | 2000 | 1980
...
00:01:00 | 60000 | 59400
```

At 00:01:00, the slower machine is behind the faster machine by 600 ticks. This is the maximum clock skew that will occur.

To reduce the impact of clock skew, distributed systems often use clock synchronization algorithms. These algorithms work by exchanging clock information between the processors in the system and adjusting their internal clocks accordingly.

4.One of the modern devices that have (silently) crept into distributed systems are GPS receivers. Give examples of distributed applications that can make use of GPS information. Take one example and explain in detail how it uses the GPS information.

GPS receivers are now commonly used in distributed systems in a variety of ways, including:

- **Fleet management:** GPS receivers can be used to track the location of vehicles in a fleet, such as delivery trucks or taxi cabs. This data can be used to optimize routing, dispatch vehicles, and provide customers with real-time updates on the location of their order or ride.
- **Asset tracking:** GPS receivers can be used to track the location of valuable assets, such as construction equipment, shipping containers, or medical devices. This data can be used to prevent theft, improve efficiency, and ensure compliance with regulations.
- **Personal safety:** GPS receivers can be used to track the location of people, such as children, employees working in remote areas, or hikers. This data can be used to provide emergency assistance if needed.

Here is a detailed example of how GPS information can be used in a distributed application:

Application: A ride-hailing application such as Uber or Lyft

How it uses GPS information:

1. When a user requests a ride, the application uses the user's GPS location to find nearby drivers.
2. The application then sends the user's location and destination to the drivers.
3. The drivers use their GPS receivers to navigate to the

user's location.
4. Once the driver arrives at the user's location, the user can start the ride.
5. The application uses the GPS locations of the driver and the user to track the progress of the ride.
6. When the ride is complete, the application uses the GPS locations of the driver and the user to calculate the fare.

5. (5 points) Consider a communication layer in which messages are delivered only in the order

that they were sent. Give an example in which even this ordering is unnecessarily restrictive.

n example of a communication layer in which ordering messages is unnecessarily restrictive is a distributed database. In a distributed database, multiple servers are used to store and retrieve data. When a client needs to write data to the database, it sends a write request to one of the servers. The server then writes the data to its local storage and replicates it to the other servers.

If the communication layer orders messages, then the client cannot know which server has received the write request first. This can lead to problems if the client tries to read the data before it has been replicated to all of the servers.

For example, suppose a client sends a write request to server A and then immediately reads the data from server B. If the communication layer orders messages, then server B will not have received the write request yet, and the client will read stale data.

To avoid this problem, distributed databases typically use a relaxed consistency model, such as eventual consistency. In eventual consistency, the database is guaranteed to eventually converge to a consistent state, but there may be a period of time during which different servers have different versions of the data.

This relaxed consistency model allows the communication layer to deliver messages out of order, which can improve performance and scalability.

Another example of a communication layer in which ordering messages is unnecessarily restrictive is a real-time chat application. In a real-time chat application, messages are sent and received in real time. If the communication layer orders messages, then the messages will be displayed to the users in the order that they were sent, even if the messages were not received in that order.

This can lead to a bad user experience. For example, suppose one user sends a message and then sends another message a few seconds later. If the communication layer orders messages, then the second message will be displayed to the user before the first message, even if the second message was received first.

To avoid this problem, real-time chat applications typically deliver messages to the users as soon as they are received, regardless of the order in which they were sent.

In general, ordering messages is only necessary in applications where the order of the messages is important. For example, in a financial trading system, it is important to order messages so that trades are executed in the correct order. However, in many distributed systems, ordering messages is unnecessarily restrictive and can lead to performance and scalability problems.

6.

Explain in your own words what the main reason is for considering weak consistency

models. It is often argued that weak consistency models impose an extra burden for

programmers. To what extent is this statement true?

The main reason for considering weak consistency models is to improve performance and scalability. Strong consistency models, such as sequential consistency, require all clients to see the same data in the same order, regardless of which server they are connected to. This can be difficult and expensive to achieve in distributed systems, especially large-scale ones.

Weak consistency models relax this requirement to some degree. For example, eventual consistency guarantees that all clients will eventually see the same data, but it does not guarantee that they will see it in the same order. This makes it easier to implement and scale distributed systems, but it also imposes some additional challenges on programmers.

One of the challenges of programming for weak consistency models is that programmers must be aware of the potential for data inconsistencies. For example, if a client reads data from a server before the server has had a chance to replicate the data to other servers, the client may read stale data.

Another challenge is that programmers must design their algorithms to be tolerant of data inconsistencies. For example, if a client is trying to update a piece of data, it must be able to handle the case where another client has already updated the data since the client read it.

To what extent is this statement true?

This statement is true to some extent. Programmers who are used to programming for strong consistency models will need to learn new concepts and techniques when programming for weak consistency models. However, the extra burden is not insurmountable, and there are many resources available to help programmers learn how to program for weak consistency models.

Overall, the benefits of weak consistency models outweigh the challenges. Weak consistency models allow distributed systems to be more performant and scalable, which is essential for many modern

applications.

7. (10 points) What kind of consistency would you use to implement an electronic stock market?

Explain your answer.

I would use **causal consistency** to implement an electronic stock market. Causal consistency is a weak consistency model that guarantees that causally related writes are seen by all processes in the same order. This means that if a write A causes a write B, then all processes will see write A before they see write B.

Causal consistency is a good fit for electronic stock markets because it ensures that all traders see the same sequence of events. This is important for preventing fraud and ensuring fairness in the market.

For example, suppose that a trader A places a buy order for a stock at a certain price. Then, trader B places a sell order for the same stock at the same price. If the communication layer delivers the messages to the exchange in any order, then it is possible that trader B's

sell order will be executed before trader A's buy order. This would be unfair to trader A, and it could also lead to market manipulation.

Causal consistency prevents this from happening by ensuring that the exchange sees trader A's buy order before it sees trader B's sell order. This ensures that trader A's buy order is executed first, and it prevents trader B from profiting from trader A's information.

Another advantage of causal consistency is that it is relatively easy to implement. There are a number of existing algorithms for implementing causal consistency, and these algorithms are not too expensive to run.

Overall, causal consistency is a good choice for electronic stock markets because it ensures fairness and is relatively easy to implement.

Here are some additional benefits of using causal consistency in an electronic stock market:

- **Reduced latency:** Causal consistency can help to reduce latency in the market by allowing the exchange to process orders as soon as they are received, without having to wait for all orders to be synchronized. This is important for ensuring that traders can get their orders

filled quickly.

- **Increased scalability:** Causal consistency can help to scale the market by allowing the exchange to distribute orders to multiple servers for processing. This can help to improve throughput and reduce the risk of a single server outage disrupting the market.
- **Improved robustness:** Causal consistency can help to improve the robustness of the market to errors and failures. For example, if a server fails, the other servers can continue to process orders without having to wait for the failed server to recover. This helps to ensure that the market remains operational even in the event of a failure.

8. (10 points) Consider a personal mailbox for a mobile user, implemented as part of a wide-area

distributed database. What kind of client-centric consistency would be most appropriate?

The most appropriate kind of client-centric consistency for a personal mailbox for a mobile user, implemented as part of a wide-area distributed database, is **monotonic read consistency**. Monotonic read consistency guarantees that

a client will always see the latest version of its data, in the order that it was written. This is important for ensuring that the user sees the most up-to-date emails in their mailbox, even if the mailbox is being replicated across multiple servers.

Other types of client-centric consistency, such as eventual consistency, would not be suitable for this application. Eventual consistency allows for the possibility that a client may see stale data, which would be unacceptable for a personal mailbox.

Monotonic read consistency can be implemented in a number of ways. One common approach is to use a version vector. A version vector is a vector of integers, one for each replica of the data. Each time the data is updated, the version vector is incremented for the replica that performed the update. When a client reads the data, it compares the version vector to the version vectors that it has stored locally. If the version vector is higher, then the client knows that the data has been updated and it fetches the latest version.

Another approach to implementing monotonic read consistency is to use a last-writer-wins policy. Under a last-writer-wins policy, the most recent write to the data is always the one that is visible to clients. This policy is simpler to implement than using version vectors, but it can lead to more conflicts if multiple clients are trying to write to the same data at the same time.

Overall, monotonic read consistency is the best type of client-centric consistency for a personal mailbox for a mobile user, implemented as part of a wide-area distributed database. It ensures that the user will always see the latest version of their data, in the order that it was written.

9. (10 points) We have stated that totally ordered multicasting using Lamport's logical clocks does

not scale. Explain why.

Lamport's logical clock algorithm is a simple and efficient way to order events in a distributed system. However, it does not scale well to large distributed systems.

One reason for this is that Lamport's algorithm requires all nodes in the system to communicate with each other in order to order events. This can be expensive and time-consuming in large systems.

Another reason is that Lamport's algorithm is not very fault-tolerant. If a node fails, the entire system can become unable to order events.

For these reasons, Lamport's logical clock algorithm is not typically used in large-scale distributed systems. Instead, more scalable and fault-tolerant algorithms are used, such as vector clocks and gossip protocols.

Here is a more detailed explanation of the scalability challenges of Lamport's logical clock algorithm:

- **Communication overhead:** Lamport's algorithm requires all nodes in the system to communicate with each other in order to order events. This can be expensive and time-consuming in large systems, especially if the network is slow or unreliable.
- **Centralized coordination:** Lamport's algorithm relies on a central coordinator to order events. If the coordinator fails, the entire system can become unable to order events.

- **Single point of failure:** The coordinator in Lamport's algorithm is a single point of failure. If the coordinator fails, the entire system can become unavailable.

10. (10 points) For active replication to work in general, it is necessary that all operations be carried

out in the same order at each replica. Is this ordering always necessary?

No, ordering all operations in the same order at each replica is not always necessary for active replication to work in general.

Active replication is a technique for fault tolerance in distributed systems. It works by replicating the state of the system across multiple servers, called replicas. Each replica maintains a copy of the system's state and processes requests independently. When a client requests an operation, it sends the request to all of the replicas. The replicas then perform the operation and return the results to the client.

In some cases, it is not necessary for all of the replicas to perform the operation in the same order. For example, if a client requests an operation that does not affect the state of the system, such as a read-only operation, then the replicas can perform the operation in any order.

However, in other cases, it is necessary for all of the replicas to perform the operation in the same order. For example, if a client requests an operation that updates the state of the system, such as a write operation, then all of the replicas must perform the operation in the same order to ensure that the system remains in a consistent state.

Whether or not it is necessary to order all operations in the same order at each replica depends on the specific application.

Here are some examples of cases where it is not necessary to order all operations in the same order at each replica:

- **Read-only operations:** Read-only operations do not affect the state of the system, so they can be performed in any order.
- **Independent operations:** Independent operations are operations that do not affect the same data. Independent operations can be performed in any order.

- **Commutative operations:** Commutative operations are operations that produce the same result regardless of the order in which they are performed. Commutative operations can be performed in any order.

11. (10 points) Define what a hashing algorithm is. What is the difference between a hashing

algorithm and a cryptographic hashing algorithm? Give the pseudocode of a simple hashing

algorithm. What is the time complexity of your algorithm?

Hashing algorithm: A hashing algorithm is a function that takes an input of any size and produces a fixed-size output, called a hash value. Hashing algorithms are used to create unique identifiers for data, to detect changes to data, and to quickly search for data in large datasets.

Cryptographic hashing algorithm: A cryptographic hashing algorithm is a special type of hashing algorithm that is designed to be resistant to collisions and preimages. Collisions occur when two different inputs produce the same hash value. Preimages occur when it is

possible to find an input that produces a given hash value. Cryptographic hashing algorithms are used in security applications, such as digital signatures and password hashing.

Difference between hashing algorithms and cryptographic hashing algorithms:

Characteristic	Hashing algorithm	Cryptographic hashing algorithm
Purpose	Create unique identifiers for data, detect changes to data, quickly search for data	Secure data and verify its integrity
Resistance	May not be resistant to collisions or preimages	Resistant to collisions and preimages
Applications	Data indexing, caching, file integrity checking	Digital signatures, password hashing, message authentication codes

12. (10 points) Explain in your own words what a blockchain is. What are the two main problems

blockchains solve? Explain how these problems are solved in blockchain technologies.

What is a blockchain?

A blockchain is a distributed database that is shared among the nodes of a computer network. As a database, a blockchain stores information electronically in digital format. Blockchains are best known for their crucial role in cryptocurrency systems, such as Bitcoin, for maintaining a secure and decentralized record of transactions.The innovation with a blockchain is that it guarantees the fidelity and security of a record of data and generates trust without the need for a trusted third party.

Two main problems blockchains solve:

1. **Double-spending:** Double-spending is the act of spending the same digital asset twice. This is a problem with digital assets because they can be easily copied and transmitted. Blockchains solve this problem by

creating a tamper-proof record of all transactions.

2. **Data integrity:** Data integrity is the assurance that data has not been tampered with. This is a problem in many distributed systems because it can be difficult to track changes to data. Blockchains solve this problem by using cryptography to ensure that data cannot be altered without detection.

How blockchain technologies solve these problems:

- **Double-spending:** Blockchains solve the double-spending problem by using a decentralized network of computers to validate transactions. When a transaction is initiated, it is broadcast to the network. The network then verifies that the transaction is valid and that the sender has sufficient funds to complete the transaction. If the transaction is valid, it is added to a block. The block is then added to the blockchain. Once a transaction is added to the blockchain, it cannot be reversed.
- **Data integrity:** Blockchains solve the data integrity problem by using cryptography to secure the blockchain. Each block in the blockchain is linked to the previous block using a cryptographic hash function. This means that if any data in a block is changed, the hash of the block will also change. This makes it very difficult to

tamper with data on a blockchain without detection.

13. (10 points) There are 3 main consensus algorithms in blockchain technologies. Proof of Work,

Proof of Space, and Proof of Stake. Define each one, and discuss 2 advantages and 2

disadvantages of each approach (compared to other approaches).

below is the Advantages, **Disadvantages**

Consensus algorithm	**Advantages**	**Disadvantages**
Proof of Work (PoW)	Secure, decentralized	Energy-intensive, slow, and expensive
Proof of Space (PoS)	Energy-efficient, scalable	Not as secure as PoW, can be centralized
Delegated Proof of Stake (DPoS)	Fastest, most scalable, energy-efficient	Most centralized, not as secure as PoW