Parallel and Distributed Computing

Exam Questions Example:

Question 1 (1 pt)
A colleague wants to give you an implementation of a concurrent object for integration with your highly parallel shared memory system. What is a good reason for requiring the object to be linearizable?

Group of Answer Choices

- Linearizability is the universally accepted notion of consistency for all concurrent objects.
- Linearizability ensures that the object's methods are executed atomically.
- **Linearizability means that you can relate the behavior of the concurrent object to an abstract sequential specification.**
- **Linearizability is compositional, so if your system is linearizable the system with the object added will be linearizable.**
- Linearizability ensures no thread has to wait for the object.

Question 2 (1 pt)
Of the following, which are good reasons to favor locks over a powerful RMW register such as CAS?

Group of Answer Choices

- Interference between threads when accessing memory is more likely when using CAS.
- **When used often, CAS can be slow due to increased memory bus traffic.**
- **Applications of CAS can be difficult to reason about compared to locks.**
- CAS and other RMW operations may unexpectedly get stuck.

Question 3 (1 pt)
You are the Chief Technical Officer of a bank that runs several data centers holding customer data spread across a large geographical area. Customers must be ensured timely access to their data. A software engineer suggests that you should use the Paxos algorithm to ensure that data is consistent across data centers. What is a reasonable reaction to the engineer's proposal?

Group of Answer Choices

- Using Paxos ensures strong consistency of data at all times, with no risk of lowering availability.
- **Using Paxos can lead to issues with customer availability if data centers become disconnected.**
- **Using Paxos may not be necessary, since strong consistency of data is not necessary for most customers.**

- Using Paxos means that consistency is guaranteed even when a majority of servers have crashed.

Question 4 (1 pt)
Reasoning about linearizability for a concurrent object requires a sequential specification of the object's behavior. For this course, what can be such a specification?

Group of Answer Choices

- **Enforceable method code contracts with preconditions and postconditions.**
- **A decision procedure that takes a sequence of object method calls and returns a boolean indicating whether the sequence is legitimate.**
- Code comments describing informally what a method is supposed to do.

Question 5 (1 pt)
You are the Chief Technical Officer of a bank that runs several private data centers holding customer data spread across a large geographical area. The data must be kept secure and consistent. A software engineer suggests that you should use the PBFT algorithm for this purpose. What is a reasonable reaction to the engineer's proposal?

Group of Answer Choices

- The assumptions made by PBFT on synchrony to ensure termination are unrealistic in a distributed data center environment.
- Occurrence of up to $f < n/3$ (for n the number of servers) Byzantine faults are likely in your data centers, so using PBFT is a good idea.
- **The message passing and resource overhead of PBFT are hard to justify in a controlled data center setting.**

Question 6 (1 pt)
What kind of question is the Java Memory Model designed to help us answer?

Group of Answer Choices

- **The question of which values can be read from a variable following (possibly concurrent) writes to the variable.**
- The question of whether two operations on memory are in the happens-before relation.
- The question of whether reading or writing a variable is thread-safe or not.
- The question of whether a Java method implements its specification.

Question 7 (1 pt)
What does the CAP theorem tell us about systems using Paxos?

Group of Answer Choices

- The CAP theorem says that for large systems, we can only provide eventual consistency, in contrast to Paxos.
- Nothing, the CAP theorem has no bearing on systems using Paxos.
- **The CAP theorem says that during network partitions, a system using Paxos must drop availability.**
- The CAP theorem says that using Paxos is a bad idea due to nontermination.
- The CAP theorem says that systems using Paxos can never provide availability.

Question 8 (1 pt)
Declaring a variable as volatile in Java means that

Group of Answer Choices

- The variable's value must be updated over time.
- The variable's value may change unexpectedly.
- **The variable's value becomes visible to all readers after a write operation is completed.**
- **The variable behaves like an atomic register.**

Question 9 (1 pt)
The consensus number of a register is

Group of Answer Choices

- **An indication of how difficult it is to implement the register on a personal computer such as a laptop.**
- **A measure of how well a register can be used to achieve synchronization for deterministic concurrent objects.**
- A measure of how good a register is for concurrent programming.
- **Useful when comparing it to other proposed registers for concurrency control.**
- A standard for classifying synchronization primitives used in industry.

Important

Mutual Exclusion: Only one process can enter the critical section
Deadlock-Free: One process will always progress to the critical section
Fairness (starvation-free): Each process eventually gets a turn
Livelock: When two different processes try to acquire a lock over and over but block each other

RMW Register allows one to read and modify the current value while being atomic. In short, it can coordinate shared data safely without locks; it utilizes compare and swap. When it tries to write a value, it writes first using a ? symbol. This means that other processes cannot write to it and know the value has yet to be determined.

Optimistic and lazy synchronization. Optimistic is compare and swap, checking if the data structure has changed if not modifying it, lazy synchronization is that we might logically delete something to avoid locking and then physically delete it later.

Relaxed memory models such as Total Store Ordering (TSO), Release consistency, and Weak ordering.

Consensus is achieved utilizing ***agreement*** between all processes, ***validity*** where all processes propose the same value, and lastly, ***termination*** where all processes must decide on a value.

Queens Algorithm works utilizing a 2 round phase algorithm. For each phase, a queen is selected. When each process for the first part of the phase sends out its value, if it receives n/2 + f of the same value that it chooses that value, if not in the second round when the queen sends out its value then that value is selected. f is the number of Byzantine faulty processes that the system can handle.
Kings Algorithm: 3 rounds per phase where in either middle round if it has received at least n-f processes will broadcast a proposed value, if a process receives a proposed message for some value that is more than f + 1 it adopts that value.

Safe Registers: Provide a correct value only if no write is ongoing
Regular Registers: Ensure that a read during a write may return either the old or the new value
Atomic Registers: Strict consistency where each read will reflect the latest write (the actions are linearizable)

Bivalent States: A state where the final consensus is undecided, for example, a value can either be a 0 or a 1.
Univalent States: A state where the outcome of a variable is determined but not yet known to the processors.

SpinLocks: We have TAS (test-and-set), bad due to high bus traffic (shared memory bus). We then have TTAS (test-test-and-set) which performs better due to the local cache usage. Also utilizing a backoff algorithm can improve this even more.

ALock: Array-based queue where spinlocks spin on their specific flag. It is space efficient doesn't overflow the bus, and has a fixed amount of locks.
MCS: Like Alock but utilize a linked list instead to handle a dynamic number of threads.

Hardware memory models: Sequential consistent & Relaxed Memory model. Examples of relaxed memory models are total store ordering, release consistency, and weak ordering.

Lecture 1: Introduction to Parallel Computing and Amdahl's Law

Amdahl's Law

Fundamental understanding of how speedup is the limit of parallel computing. Via the formula, we can specify which part of the program can be parallized for example 30% can't and 70% can be parallized.

$$S(n) = \frac{1}{s + \frac{p}{n}}$$, p = 1-s

Safety and Liveness Properties

Safety in a system guarantees that nothing bad will happen during execution while liveness ensures that something good will happen. For example, safety ensures that no processes will execute in a critical section at the same time while liveness makes sure that we are deadlock-free.

Lecture 2: Concurrency, Mutual Exclusion, and Synchronization

Concurrency and Synchronization Mechanisms

Threads and processes can be seen and modeled as a **state machine**. All events such as reading/writing have an asynchronous behavior to them. A key concept is that we might have unpredictable states in concurrent systems, which means that understanding the underlying system is crucial to ensure that the system is correct.

Critical Sections and Mutual Exclusion

In the introduction to locks, algorithms must be **deadlock-free** and **starvation-free mutual exclusion**. An Algorithm that achieves this is the **Bakery Algorithm**.

- **Deadlock-free:** Two processes don't wait for each other to finish, meaning we are not stuck in a continuous loop
- **Mutual exclusion:** An area where processes cannot access a shared resource at the same time
- **Starvation-free:** Making sure that no thread gets stuck waiting forever if, for example, new threads acquire the resource that it is waiting for. It is important to select the next processes to acquire a lock randomly or by a specific process.

How does the Bakery Algorithm work?
In the Bakery Algorithm, each process is assigned a number (a ticket) in a lexicographical order. Before entering the critical section, a process receives a ticket number, and the process with the smallest ticket number enters the critical section. If two processes receive the same ticket number, the process with the lower process ID is given priority.

Petersons Algorithm

Note: *Not very important due to busy waiting, so most likely will not show up.*

This algorithm ensures that we can achieve mutual exclusion between two threads and ensures that it is deadlock and starvation-free. It utilizes a flag array and a victim variable. There is no deadlock because if two threads are in the critical section at the same time only one of them will eventually succeed. It is starvation-free because it ensures that each thread will get its turn to enter the critical section. Keep in mind that the basic algorithm also only works on two threads.

Deadlock and Livelock Scenarios

Livelock is similar to deadlock but is a situation where processes **keep changing state** in response to each other and without making any progress. This for example happens if we have a situation where we might utilize a process of solving deadlocks but when doing so the processes continuously wait for each other.

Volatile Counters and Parallel Execution

Volatile counters mean that if multiple threads want to read/write to a shared variable it will fetch from the main memory rather than a cached value from the local memory.

Lecture 3: Linearizability and Quiescent Consistency in Concurrent Objects

Linearizability

Linearizability describes that method calls on concurrent objects (multi-threads) that must behave the same way as if executed sequentially. For example, on concurrent objects like a FIFO queue, methods like enqueue and dequeue need to have a determining specific point for linearization.

In short for example, if we have a situation where we have multiple threads accessing an object concurrently the output should be that if they were performed one after the other.

Compositionality of Quiescent Consistency

Compositionality of Quiescent consistency refers that for a concurrent object that is quiescent consistent after a period of activity. Meaning that the object behaves in a consistent way matching a valid sequential history.

FIFO Queue Implementation

This refers to two different implementations in a concurrent view where we either can have a lock-based to a wait-free imp, implementation. We might want to both in a way ensure consistency with atomic actions but we also don't want to have to wait for too long to perform actions. Therefore there are different transitions and specific use cases can be important.

```
AtomicInteger counter = new AtomicInteger(0);

void increment() {
    int oldValue, newValue;
    do {
        oldValue = counter.get(); // Step 1: Read the current value
        newValue = oldValue + 1;  // Compute the new value
    } while (!counter.compareAndSet(oldValue, newValue)); // Step 2-4:
Compare and set
}
```

Lecture 4: Atomicity and Consistency in Shared Registers

Register Consistency Models

There are multiple types of register types with different fundamentals on how consistency is managed.

- **Safe Registers:** Provide a correct value only if no write is ongoing
- **Regular Registers:** Ensure that a read during a write may return either the old or the new value
- **Atomic Registers:** Strict consistency where each read will reflect the latest write (the actions are linearizable)

Linearizability and Atomicity

Linearzilibty will ensure that operations made on a shared object may happen at the same time. An atomic register is linearizable, meaning its operations are invisible and will always return the latest committed value.

Snapshot Algorithm

A snapshot algorithm allows a process to take consistent snapshots of a shared memory. A snapshot algorithm is curricula and important when implementing a wait-free atomic snapshot distributed system.

Multi-reader, multi-writer (MRMW) Atomicity

This is an extension of atomic registers to multi-reader and multi-writer. The implementation of such registers involves timestamping to ensure that all processes can read consistent data. A conflict might happen if a write happens at the same time as a read, then the action will retry until it gets a clean snapshot. An issue with this is that if two writers write values a reader might either return the first or second value. This is a problem because it becomes inconsistent.

SRSW Atomic Register

Single Reader Singlöe Writer, Guarantees that read will always return the most recent write. This means only one process writes to the register and only one process reader from the register.

Lecture 5: Consensus in Distributed Systems: Impossibility and Critical States

Consensus Problem

The consensus problem resolves and ensures that multiple threads agree on a single data value. This is important when working with concurrent objects. Consensus is one of the most important concepts in fault tolerance systems.

The requirements to achieve consensus are:

- **Termination:** Every process must eventually decide on a value
- **Agreement:** All processes must decide on the same value
- **Validity:** The agreed-upon value must be one of the initial values proposed by the processes.

Bivalent and Univalent States

- **Bivalent States:** A state where the final consensus is undecided, for example, a value can either be a 0 or a 1.
- **Univalent States:** A state where the outcome of a variable is determined but not yet known to the processors.

Proof of Consensus Impossibility

The Fischer-Lynch-Paterson theory states that a process cannot make a reliable assumption about the timing of a message and when it is delivered. A critical state in a certain action is a bivalent state where the next action cannot determine its value. Its state will then lead to a univalent state showing that the system is fragile in a consensus process. This means that even small variations in the timing of operations can lead to totally different outcomes in the result of a concurrent object.

The basis of this proof is that if a process is in the process of going from a bivalent state to a univalent state and crashes (crashes in its critical state) it creates uncertainty and the remaining processes cannot reliably move to a univalent state.

Faulty Consensus and Wait-Free Protocols

In a wait-free consensus, all processes can complete their action in a finite number of steps. In a broadcasting algorithm, we might have two different types. Broadcast An algorithm only broadcasts its value to the specific other process which will result in the other processes not reaching a consensus because they can't decide on a value. Type B broadcasting builds on the fundamental that it will always broadcast all its values to all processes where each process will then agree on a value and therefore reach a consensus. How it achieves consensus is with a bit more advanced algorithm such as majority-based if they receive multiple values (some might receive value v1 before v2 etc).

Consensus Number

Refers to the number of processes that can be operated on and can still solve consensus in a wait-free manner. Meaning n number of processes.

Lecture 6: Concurrent Data Structures and Synchronization Techniques

Fine-Grained Synchronization

Fine-grained synchronization involves dividing a data structure into smaller components each has its locks. Doing this will allow for better performance. An example of this could be that in a linked list we utilize a lock for each node etc.

Lock-free Synchronization

Lock-free synchronization involves utilizing atomic operations such as compareAndSet() which allows only one thread in an atomic way to set a value to a certain action without the use of locks.

Linearizability

Ensuring linearization is a correct concept in concurrent objects where operations such as add remove and contain will happen at some point in time. Meaning even if these actions happen at the same time they have to be able to be seen as if they happened linearly.

Optimistic and Lazy Synchronization

Optimistic synchronization involves a shared data structure without locks and assuming there will be no conflict. Once an operation is nearly done the thread locks the relevant parts of the data structure and then validates if the data structure has changed since it started. If it has it might need to retry the operation.

Lazy synchronization involves logical and physical actions such as removal. For example, we might mark a node as deleted while it physically is not deleted. This means that the deletion might happen later during a clean-up phase.

Lecture 7: Spin Locks

Spin locks and Test-and-Set (TAS) locks

Spin locks a simple locks that process continuously to check if they are free. These are useful when waiting times are short but can become a bottleneck in a system with longer delays.

Test-and-Set (TAS) Is a type of spinlock where a thread tries to compare and swap a value and if successful it acquires the lock and if not it retries. The issue with this is that it could lead to conflict.

Performance Issue and Contention

The performance of the test and set can be quite bad due to conflict and high bus traffic. For example, when a lock is quired by one thread it will lead to all waiting threads needing to invalidate their cache because the lock state has changed.

Improved Spin Lock Algorithms

Test-Test-and-Set (TTAS) is a lock that performs better than a normal TAS because it first checks if the lock is free using the local cache which will reduce the number of times it needs to communicate with the bus. It also utilizes an exponential backoff lock which also can help. Because it before test before the test and set it can via its won cache check the value before performing the normal TAS operation.

```
public class TTASLock {
  private AtomicBoolean state = new AtomicBoolean(false); // Shared lock
variable

  public void lock() {
    while (true) {
      // First test: spin locally until the lock looks free
      while (state.get()) {
        // Spin while the lock is held by another thread
      }
      // Second test: attempt to atomically set the lock
      if (!state.getAndSet(true)) {
        return; // Lock acquired
      }
    }
  }

  public void unlock() {
    state.set(false); // Release the lock
```

```
  }
}
```

Another algorithm is ALopck (Andersons Lock) which is a scalable array-based queue lock where each thread gets its position in a queue and spins on its local flag. It utilizes a FIFO queue system and is fair.

Cache Coherence

Cache coherence is about issues in multi-core systems where multiple processors cache shared data which means that when it updates it could lead to inconsistencies.

MCS (Mellor-Crummey and Scott) lock

Queue-based lock where each thread inquiries itself and waits for its turn in the queue. Pretty much like ALock besides it utilizes a linked list instead of an array. In short, when it is time that it is in front and ready to be dequeued it will tell the next in line that they are first etc.

Overall

Spinlocks vs normal locks is that they are busy waiting but have no overhead of suspending and waking up threads which means that it is very fast for short critical sections. This means fast in low contention (locks are available soon).

Lecture 8: Hardware Memory Models

Sequential Consistency (SC)

Sequential consistency is a memory model that ensures that all results of any execution are the same as if the operations of all processors were executed in some sequential order. This means that sequential consistency ensures that operations happen in the order of the program. **The drawback** of this is that it is sometimes a bit too strict in high-performance systems where optimizations such as instruction reordering can be utilized.

Relaxed Memory Models

Is a memory model that is more flexible in which operations are ordered to improve performance. Examples of such models are **total store ordering**, **release consistency**, and **weak ordering**. Modern processes do not implement strict SC but rather use relaxed models.

Memory Barriers (Fences)

Certain actions can be instructed to be a memory barrier which means that it prevents certain types of reordering and therefore ensures that the memory operation is executed before other starts. This is extremely important if we have certain actions in a relaxed memory model that need to have a strict level of consistency.

Real-world Architecture Examples (x86, ARM, RISC-V)

Each architecture employs its relaxed memory model.

Types of reordering:

1. **Write → Read Reordering (Store-Load)**: A write to a variable might be delayed, and a read to another variable may happen first.
    - **Example**: Writing `x = 1` might be delayed, and `y` is read before `x` is written.
2. **Read → Write Reordering (Load-Store)**: A read from a variable may happen before a previous write completes.
    - **Example**: Reading `y` happens before writing `x = 1`.
3. **Write → Write Reordering (Store-Store)**: Two write operations may be swapped, so the later write occurs before the earlier one.
    - **Example**: `x = 1` might happen after `y = 1`, even though `x` was meant to happen first.
4. **Read → Read Reordering (Load-Load)**: Reads from two different variables may happen out of order.

Lecture 9: Work Stealing and Fine-Grained Parallelism

Work Stealing Model

A working stealing model involves that each processor has its queue of tasks. When a certain process finishes all of its tasks it steals other processes' tasks. It is a simple enqueue and dequeue list.

Fine-Grained Parallelism

In short, it means a division of computational problems into a large number of smaller tasks that can be executed in parallel.

Java Memory Model

JMM provides rules to manage the visibility and ordering of memory-accessed shared variables.

Lecture 10: Java Memory Model and Synchronization

Java Memory Model

As mentioned in the last chapter a memory model defines the way how read and write operations on a shared memory are ordered. JMM ensures that operations on memory behave consistently across different JVM.

Sequential Consistency (SC) and Total Store Order (TSO)

Sequential consistency is a strict memory model where all operations are executed in the same order as they appear in the program. **Total Store Order** is more flexible allowing reordering of read/write operations. Synchronization order also called SO is the action that includes acquiring and releasing locks, volatile reads, and writes.

Data Race Freedom (DRF) and Synchronization

A data race is the process where two threads access the same variable and at least one of them writes to it without proper synchronization.

Lecture 11: Fault-Tolerant Consensus in Synchronous Systems

Synchronous Systems Consensus Model

In a synchronous system, all communication happens in rounds known as bounded delays. Each round involves receiving messages, processing them, and sending out a certain result. The model is built to be able to handle that certain processes might crash and fail meaning messages can be lost. The goal of a synchronous system is to ensure that all nonfaulty processes agree on a single value meaning we reach a consensus. Synchronous vs Asynchronous means that for synchronous systems all actions will happen in rounds while in asynchronous everything can happen whenever.

f-Resilient Consensus Algorithms

An algorithm called FloodSet Algorithm works as follows:

1. Each process broadcasts its value to the others in every round
2. After receiving a new value the process broadcasts these new values in the subsequent rounds.
3. After f+1 rounds where f is the number of failures, each process decides on the **minimum value** it has received. {1,2,3,4} 1 is picked

Utilizing this algorithm it can handle up to f failures and f+1 rounds are required to guarantee that consensus is reached.

Lower Bound for Consensus with Crash Failures

Any type of f-resilient consensus algorithm requires at least f+1 rounds to ensure that all nonfaulty processes agree on a decision.

Byzantine Failures and Consensus

Byzantine Failures specify that if a process fails it doesn't just die but starts behaving in a malicious way sending conflicting information. There is proof stating that a nonconsensus algorithm, can tolerate f Byzantine failure if f >= n/3 where n is the number of processes. In short, it means that we need at least n > 3f processes to reach a consensus where f is a byzantine failure.

hej

Lecture 12: Byzantine Consensus and Randomized Algorithms

Byzantine Fault Tolerance (BFT)

The fault tolerance of byzantine refers to the ability of a system to achieve consensus when nodes might act maliciously. This means in situations where nodes might send inconsistent information to other nodes.

Few different algorithms that do this:

- **Queen Algorithm:** works with n > 4f, it involves selecting a leader called queen in each phase. Each round the queen will broadcast a value that others follow in one phase. It runs in f+1 phases where each phase is two rounds. Based on the received value the processors can decide based on the majority of messages received, and if no value is supported they can then follow the queen's broadcast.
  - Set own value to the value that was
    If own value appears > $n/2+f$ times
      support this value
    else
      do not support any value
- **Ben-Or Algorithm:** This algorithm is utilized in asynchronous systems and can tolerate n > 8f failures but can be adjusted to tolerate n > 4f.
- **Kings Algorithm:** 3 rounds per phase where in either middle round if it has received at least n-f processes will broadcast a proposed value, if a process receives a proposed message for some value that is more than f + 1 it adopts that value.
  - Round 2:
    If some value x appears ≥ $n-f$ times
      Broadcast "Propose x"
    If some proposal received > f times
      Set own value to this proposal

Randomized Consensus Algorithms

This type of algorithm achieves consensus in distributed systems where processes randomly decide between different options. They are used when deterministic algorithms are unable to guarantee consensus due to faults. This algorithm utilizes randomness to make sure that eventually, all processes will agree on the same value. In short, randomness is utilized to break deadlocks in situations where processes might get stuck in conflicting states where likely a deterministic algorithm has failed.

Ben-Or Algorithm is one of the earliest randomized consensus algorithms that operate under asynchronous conditions. It works by that each process proposes a value and exchanges proposals. If the processes see enough proposals for a certain value (n-2f processes) adopt

and broadcast that value. Otherwise, a value is chosen at random. It does this over and over until the processes have agreed on a value.

Byzantine Agreement with Authentication

Same as the other algorithms run for f + 1 iterations. It utilizes a digital signing method to make sure and detect if byzantine nodes are detected.

Ben-or Algorithm

Pretty simple, like a normal fault tolerance message system but accounts for that messages can be delayed.

1. Each process broadcasts its value to all other processes
2. Collect proposals for at least n-f processes.
3. If a process receives the same value from at least n-2f processes it adopts and decides on it If processes receive at least n-4f to n-2f values of the same it proposes this value for the next round

Lecture 13: Fault Tolerance and Consensus in Distributed Systems

Fault Tolerance in Distributed Systems

Fault tolerance in distributed computing refers to a system that can continue functioning after a partial failure. For example, we might utilize data replication to make sure that we stay consistent.

Commit Protocols (2PC, 3PC)

These commit proctors are used in distributed systems to ensure that all nodes involved in a transaction either commit to a transaction or abort it to ensure consistency.

- **2PC:** For a 2PC structure, we have a coordinator who sends a prepared message to all participants if they are ready to commit or not. Each participant checks if they are ready. For the 2nd phase, the coordinator collects the responses, and if all participants are ready the coordinates commit and send out a reply to all participants.
- **3PC:** Similar to 2PC but solves a block issue. Same as the first phase in 2PC, but in phase 1 we have a can-commit phase where all participants reply if they are ready to commit. In phase 2 the coordinator sends out a pre-commit message to all participants stating that they should be prepared to commit. Each process then needs to acknowledge this and then in phase 3, the actions get committed.

Paxos Algorithm

The Paxos algorithm is used as a consensus algorithm that tolerates node failure and is used for a system to agree on a single value. There are three different roles, proposers, acceptors, and learner. The Paxors algorithm works in four phases;

- **Prepare Phase**: A proposer generates a proposal number n and sends a prepared message to a majority of the acceptors. When acceptors receive a prepared message, they check if the proposal number n is greater than any previous proposals. The acceptors send out a promise to not accept any proposals less than n. The acceptor will also send back the most recent proposal
- **Proposal Phase**: If a proposer receives a majority of the acceptors it moves on to the next phase. The proposer looks at the value returned from the acceptors. If any acceptor has already accepted a proposal the proposer adopts the value with the highest proposal number among the responses. If no accepter has acceåt anmyu proposal the prosper can choose a new initial value. The proper then sends an acknowledgment message to the acceptors.
- **Acceptance Phase:** When a competitor receives the ack, it accepts the value if it has not already promised to ignore a proposal greater than that number (newer proposals). The acceptor then responds with an acknowledgment.
- **Commit:** If a prospers receives the majority of the acknowledgment it can then consider the value can be chosen.

Practical Byzantine Fault Tolerance (PBFT)

In distributed systems, some nodes may behave maliciously or provide incorrect information. Utilizing 2PC and Paxor assumes nodes fail by crashing but do not handle byzantine failures. A protocol to fix this is PBFT, by allows nodes to communicate and vote on the correct state with the assumption that the faulty nodes are less than ⅓ of the total. It works by a leader-based algorithm where we have one node that coordinates actions and the other nodes vote to ensure the correct value is committed. If the primary is faulty another node can take over. It works by the following: It tolerates $f < n/3$

- **Pre-prepare phase:** The client sends a requires to the leader that it wants to operate. The primary node sends out unique requests to all repliaces.
- **Prepare Phase:** The replacements verify the message and then broadcast a prepared message to all other replacements. Once it receives $2f + 1$ prepare messages from other repliaces it moves over to the commit phase.
- **Commit phase:** Each replica then sends out a commit message. Once a replica receives $2f + 1$ commit messages it knows that all nodes agree on the operation. After this, enough nodes agree and the operation can be considered to have reached a consensus. The operation gets executed after the final commit message.

Lecture 14: Distributed Storage and CAP

Theorem CAP Theorem

CAP theorem says that a system can't achieve consistency, availability, and partition tolerance

ACID vs BASE Models

ACID: Atomicity, Consistency, Isolation, and Durability for reliable database transactions.
BASE “Basically Available”, Soft State, and Eventually consistent which is a model focusing on availability and scalability and that data might eventually become consistent.

Distributed Hash Tables (DHTs)

DHTs are a decentralized system that distributes data across nodes in a network using hash functions.

P2P Architectures and Handling Churn

Peer-to-peer p2p architecture distributes tasks among different nodes without a centralized coordinator.

Consistent Hashing and LinearHashing

Consistent hashing is a method that maps both data and machines into a hash ring where nodes can either join or leave. Linear hashing is a dynamically updateable hashing scheme.