CMPSCI 677 Operating Systems **Spring 2025**

Lecture 17: April 07

Lecturer: Prashant Shenoy *Scribe:* ***Chang Jin (2025), Aishwarya Nair (2024)***

17.1 Overview

This lecture covers the following topics:

1. Primary-based protocols
2. Replicated writer protocols
3. Quorum-based protocols
4. Replica Management
5. Fault tolerance

17.2 Implementing Consistency Models

There are two methods to implement consistency mechanisms:

1. **Primary-based protocols** These work by designating a primary replica for each data item; these are coordinated by a coordinator. Different replicas could be primaries for different data items. The updates of a file are always sent to the primary first and the primary tracks the most recent version of the file. Then the primary propagates all updates (writes) to other replicas. Within primary-based protocols, there are two variants:

   Remote write protocols: All writes to a file must be forwarded to a fixed primary server responsible for the file. This primary in turn updates all replicas of the data and sends an acknowledgement to the blocked client process making the write. Since writes are only allowed through a primary the order in which writes were made is straightforward to determine which ensures sequential consistency. Since all nodes have a copy of the file, local reads are permitted.

   Local write protocols: A client wishing to write to a replicated file is allowed to do so by migrating the primary copy of a file to a replica server which is local to the client. Therefore, the coordinator for the data item will shift back and forth depending on the clients accessing the data. The client may therefore make many write operations locally while updates to other replicas are carried out asynchronously. There is only one primary at anytime.

   Both these variants of primary-based protocols are implemented in NFS.

2. **Replicated write protocols:** These are also called *leaderless protocols*. In this class of protocols, there is no single primary per file and any replica can be updated by a client. This framework makes it harder to achieve consistency. For the set of writes made by the client it becomes challenging to establish the correct order in which the writes were made. Replicated write protocols are implemented

most often by means of quorum-based protocols. These are a class of protocols which guarantee consistency by means of voting between clients. There are two types of replication:

Synchronous replication: When the coordinator server receives a Write request, the server is going to send the request to all other follower servers and waits for replication successful acknowledgements from them. Here, the speed of the replication is limited by the slowest replica in the system.

Asynchronous replication: In this, the write requests are sent to all the replicas, and the leader waits for the majority of the servers to say "replication is successful" before replying to the client that the request is completed. This makes it faster than synchronous replication. But the limitation is when we perform a read request on a subset of servers who has not yet completed the replication, we will get old data.

Question: In local-write protocols, how to determine when to move the primary?
Answer: That's a good question. The write request arrives at the local machine, which then sends a request; then, the old primary will move and transfer responsibility for item x to the new machine. The new machine will perform write and either reply to the client or send back an acknowledgment. In essence, you communicate with the server to request to become the new primary.

Question: Does the move of primary occur for single write or multiple writes?
Answer: If there is only one client accessing a data item, you can move the primary to the local machine, but if there are two clients accessing the same item, you cannot have two primaries, so you can either keep the remote machine or assign one of the two clients as the new primary

Question: In a replicated write protocol, is the read from a different client blocked?
Answer: It depends on the application layer, but for our purposes here, we assume read is not blocked.

Question: What happens if a replica blocks in the synchronous write?
Answer: The primary does not receive a response and tells the client the write failed.

Question: What happens if the replication fails for one of the replica?
Answer: Depends on the storage/file system, but you retry

Question: In primary-based protocols, is there a primary node for each data item?
Answer: Yes, there is a primary node for every data item. Each node can be chosen as primary for a subset of data items.

Question: In primary-based protocols, do we need to broadcast to all clients the fact that the primary has been moved?
Answer: Typically no. But it depends on the system. Ideally, we don't want to let the client know where the primary is. The system deals with it internally.

Question: Do the servers need to know who the primary is?
Answer: Yes.

Question: Do replicated write protocols always need a coordinator?
Answer: It is not necessary to have a coordinator if the client knows what machines are replicated in the system.

Question: How do we prevent clients from performing reads on a subset of servers who have not completed the replication yet?
Answer: From a consistency standpoint, asynchronous replication violates read-your-writes.

Question: How do you handle concurrent writes in Local-Write protocols?
Answer: When there are concurrent writes, there is no need to move the primary, as you can not determine

the primary in a concurrent write situation. In this case it will be a remote write.

Question: Are reads blocked in Synchronous Replication?
Answer: Reads are local, hence, they're not blocked.

Question: Can you improve consistency in asynchronous replication by sending additional messages?
Answer: The issue originally with asynchronous replication is in the case of crash failures of the local machines, when the write can not be propagated to the other replicas. Thus, sending additional messages is not fruitful in these cases.

Question: Can you do asynchronous replication without waiting for any of the operations to be successful?
Answer: Yes, you can do that by sending a message when the OS receives the write request.

17.3 Quorum-based Protocols

The idea in quorum-based protocols is for a client wishing to perform a read or a write to acquire the permission of multiple servers for either of those operations. In a system with N replicas of a file if a client wishes to read a file it can only read a file if N_R (the read quorum) of these replica nodes agree on the version of the file (in case of disagreement a different quorum must be assembled and a re-attempt must be made). To write a file, the client must do so by writing to at-least N_W (the write quorum) replicas. The system of voting can ensure consistency if the following constraints on the read and write quorums are established:

$$N_R + N_W > N \tag{17.1}$$

$$N_W > N/2 \tag{17.2}$$

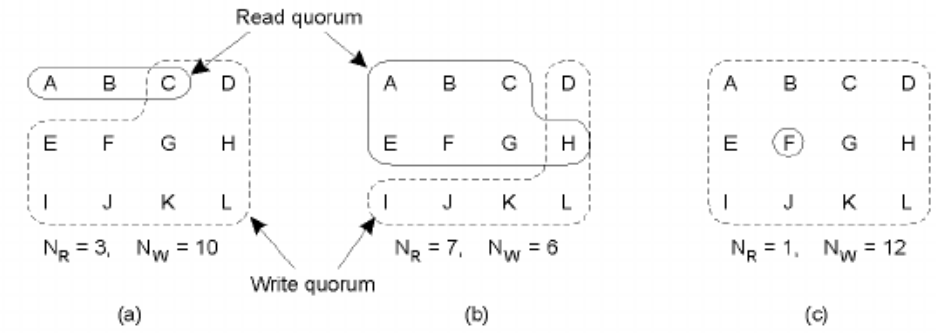

Figure 17.1: Different settings of N_R and N_W.

This set of constraints ensures that one write quorum node is always present in the read quorum. Therefore a read request would never be made to a subset of servers and yield the older version file since the one node common to both would disagree on the file version. The second constraint also makes sure that there is only one ongoing write made to a file at any given time. Different values of N_R and N_W are illustrated in Figure 17.1.

Question: Can you actually require N_R to be less than N_W?
Answer: Yes, else we will always pick one server that is never in the Write Quorum

Question: Will you check different combinations of N_R servers to get a successful read?
Answer: Consider $N_R = 3$. Pick 3 random servers and compare their version numbers as part of the voting phase. If they agree, then that is going to be the most recent version present and read is successful otherwise, the process needs to be repeated. If the number of servers is large and the write quorum is small, then you have to have multiple retries before you succeed. To avoid this make the write quorum as large as possible. If the write quorum is large, there is a high chance that read quorums will succeed faster.

Question: Why do we need them to agree on the version if we just do some reads and pick the one with higher quorum?
Answer: Consider that the version of file is four in the servers. Consider that you have performed two writes. In the first case, servers A, B, C, E, F, and G are picked and they update the version number to five. In the second case servers D, H, I, J, K, and L are picked and they have also updated the version number to five. In these scenarios, we will have a write-write conflict.

Question: Should all write quorum nodes be up to date before a new write is made?
Answer: Here we assume that a write re-writes the entire file. In case parts of the file were being updated this would be necessary.

Question: Should all writes happen atomically?
Answer: This is an implementation detail, but yes. All the writes must acknowledge with the client before the write is committed.

Question: How do write quorum nodes agree on an update?
Answer: The main focus of the agreement is to ensure that all nodes complete the writes and that all of them are using the same version number. Some of the ways of agreeing on a version is to use the current timestamp for the update version. The version could also be a simple integer which is incremented for an update.

Question: In examples (a) and (b) in Fig 17.1 where the data is not replicated to all nodes, will there ever be a case in which the read quorum never agree.
Answer: If the rules from equations 17.1 and 17.2 are followed, then your are guaranteed that the read quorum will always agree. However, if arbitrary values are chosen for N_R and N_W that don't follow these rules, the nodes might agree on reads and writes but the results may not be correct.

Question: In example (a), what happens if there's another write?
Answer: Good question, let's say another write occurs and your write quorum are A to J, now the file gets updated, there's a new version of the file known to A-J; K and L have the older version. When a read occurs, you simply pick three servers that agree, and it's simple to see that ABC can provide that.

Question: If the writes are concurrent, should we write to all the servers?
Answer: You need a way to order the write internally.
Follow Up Question: How is this concurrent?
Answer: From the client's point of view, the writes are concurrent. The requests are sent, and all pending responses at the same time.

Question: Why is it okay to miss one of the older writes?
Answer: Depends on how the write is performed - if the entire file is replaced when write is performed, then it's okay

Question: Why is timestamp not used for this? What are we using instead?
Answer: You end up introducing more requirements: Machines have to be synchronous, timestamps have to be accurate, and we have clock synchronization problems. One example is to use version numbers(simple integers) to track the update of files.

Question: Why can't you just return the file with the highest version number?

Answer: It can be done; however, this is a voting based protocol. So we need to ensure that the files match at multiple replicas to ensure the correctness of the file returned. Thus, the protocol can be simplified by adding version numbers, but it is not necessary and the protocol will work regardless.

Question: Why is returning an older version allowed?
Answer: While a new write is in progress, the most consistent version of the file is the older file, hence it is allowed.

Question: Is this asynchronous and do all the replicas get the write eventually?
Answer: In quorum based protocol, the writes are only propagated to the write set. This is because quorum based protocols are good at handling failures.

Question: When are the quorums decided?
Answer: We only set the number of servers in the read and write quorums. We don't specify which servers form the read or write quorum. The client picks the servers at random.

17.4 Replica Management

Some of the design choices involved in deciding how to replicate resources are:

- Is the degree of replication fixed or variable?
- How many copies do we want? The degree of three can give reasonable guarantees. This degree depends on what we want to achieve.
- Where should we place the replicas? You want to place replicated resources closer to users.
- Can you cache content instead of replicating entire resources?
- Permanent replicas or temporary replicas?
- Should replication be client-initiated or server-initiated?
- How do we ensure consistency?

Question: Is caching a form of client-initiated replication?
Answer: Yes, but client-initiated could be broader than just caching of content; it could even be replication of computation. In the case of gaming applications, client demand for the game in a certain location may lead to the addition of servers closer to the clients. This would be client initiated replication as well.

17.5 Fault Tolerance

Fault tolerance refers to ability of systems to work in spite of system failures. Unlike centralized applications, where a program crash results in a complete application stoppage, a well-designed distributed system can continue to function even when one or more machines fail. Failures themselves could be hardware crashes or software bugs (which exist because most software systems are shipped when they are "good enough")

Fault tolerance is important in large distributed systems since a larger number of components implies a larger number of failures, which means the probability of at least one failure is high.

If a system has n nodes, and the probability that single one fails is p, then the probability that there is a failure in the system is given by:

$$p(f) = 1 - p^n$$

As n grows, this number probability converges to 1. In other words, there will almost always be a failed node in a large enough distributed system. This emphasizes the importance of fault tolerance.

Typically fault tolerance mechanisms are assumed to provide safety against crash failures. Arbitrary failures may also be thought to be Byzantine failures where different behavior is observed at different times. These faults are typically very expensive to provide tolerance against.

The idea is to build dependable systems. The requirements of dependable systems include availability, reliability, safety, and maintainability.

Question: How is availability guaranteed?
Answer: By employing replication and calculating statistically the probability of failure. This does not apply to correlated failure. You can also prove availability in practice by measuring downtime.

Question: Can availability be 100 percent?
Answer: No, because you cannot assume a component will never fail.

Question: What is the relationship between availability and reliability?
Answer: Higher reliability will grant higher availability

17.5.1 Types of Faults

- Transient Faults: Failure comes and goes. An example can be a bit flip in memory. Most of the time it refers to a fault that occurs once and is gone afterwards
- Intermittent Faults: Faults that recur from time to time but is not permanent.
- Permanent Faults: Failure has occurred and a component is dead.
- Crash Failure: Server halts, but works correctly until it halts.
- Omission Failure: Server did not respond to a request
- Timing Failure: The server takes a long time to respond.
- Response Failure: The response from the server is incorrect.
- Arbitrary Failure: Server produces arbitrary responses arbitrary times.

Failure masking by redundancy: We can replicate hard ware three times to mask a failure, i.e. when a component fails, the system continues to run. The disadvantage is the increase in hardware cost.

Question: Why do we need three copies?
Answer: If you're dealing with crash failure, you only need two copies, however, with arbitrary failure, you need three copies to vote, and you need the faulting copy to be outvoted.