CMPSCI 677 Distributed and Operating Systems **Spring 2023**

Lecture 18: April 12

Lecturer: Prashant Shenoy
Scribe: Aishwarya Nair (2024), Yogeshwar Pullagurla (2023), Susmita Madineni (2022), Chen Qu (2019), *Sheshera Mysore (2017)*

18.1 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

18.2 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. 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. 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 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 OS received the write request.

18.3 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

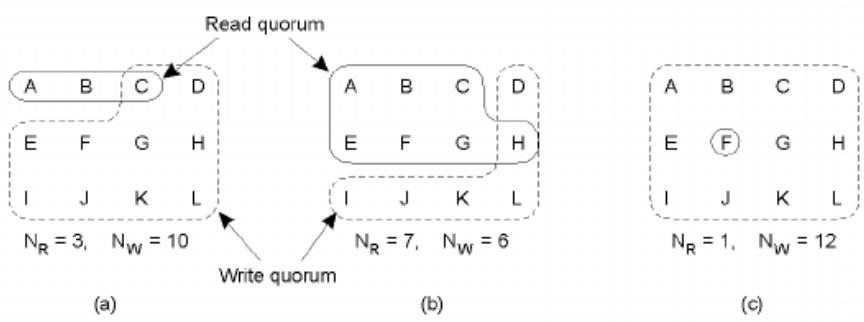

Figure 18.1: Different settings of N_R and N_W.

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:

$$N_R + N_W > N \tag{18.1}$$

$$N_W > N/2 \tag{18.2}$$

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 18.1.

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 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 a 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: Can you read from the read quorum and just select the maximum version file?
Answer: Yes. But you want them to agree.

Question: In examples (a) and (b) in Fig 18.1 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 18.1 and 18.2 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: 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 all 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.

18.4 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?
- Should replication be client-initiated or server-initiated?

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 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.

18.5 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 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.

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.