CMPSCI 677 Operating Systems **Spring 2017**

Lecture 17: April 4

Lecturer: Prashant Shenoy *Scribe:* ***Sheshera Mysore***

This lecture spoke about the implementation paradigms for consistency in replication and then moved to the next larger topic of fault tolerance.

17.1 Implementing Consistency Models

There are two methods to implement consistency mechanisms.

1. **Primary based protocols** These work by designating a primary replica which is responsible for propagating all updates (writes) to other replicas. Within primary based protocols there may be two variants:

   Remote write protocols: Here 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 write are only allowed through a primary the order in which writes were made is straightforward to determine which ensures sequential consistency.

   Local write protocols: Here 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.

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

2. **Replicated write 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. These are discussed next.

17.1.1 Gifford's Quorum-Based Protocol

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:

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

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

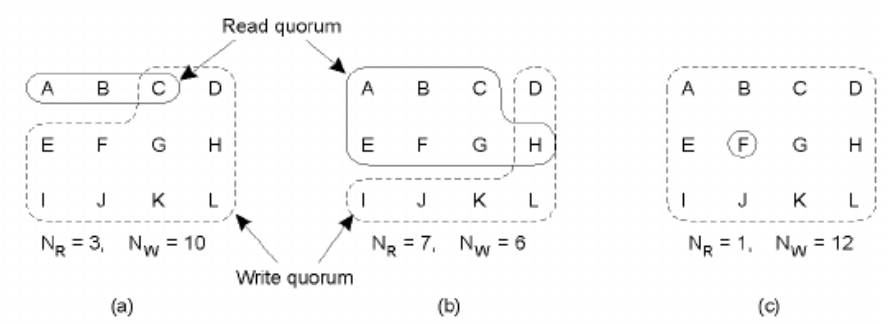

Figure 17.1: Different settings of N_R and N_W.

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. Different values of N_R and N_W are illustrated in Figure 17.1

Q: Should all write quorum nodes be up to date before a new write is made?
A: Here we assume that a write re-writes the entire file. In case parts of the file were being updated this would be necessary.
Q: Should all writes happen atomically?
A: This is an implementation detail, but yes. All the writes must acknowledge with the client before the write is committed.
Q: Can you read from the read quorum and just select the maximum version file?
A: Yes. But you want them to agree.

17.2 Replica Management

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

- 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?

Q: Is caching a form of client initiated replication?
A: 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.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 17.2: Failure models.

17.3 Fault Tolerance

Fault tolerant refers to ability of systems to work in spite of system failures. This is important in large distributed since a larger number of components imply a larger number of failures. Failures themselves could be hardware crashes or software bugs (which exist because most software systems are shipped when they are "good enough"). A system's *dependability* is evaluated based on:

- Availability: The percentage of time for which a system is available. Gold standard is that of the "five nines" i.e a system is available 99.999% of the time. This translates to a few minutes of down-time per year.
- Reliability: System must run continuously without failure.
- Safety: System failures should not compromise safety of client systems and lead to catastrophic failures.
- Maintainability: Systems failures should be easy to fix.

17.3.1 Failure Models

The different models of failure are shown in Figure 17.2. 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 tolerate against.

17.3.2 Redundancy

Fault tolerance may be achieved by means of redundant computations and per stage voting. The circuit shown in Figure 17.3 demonstrates this. Here each computation of the stages A, B and C is replicated and the results are aggregated by votes. This circuit is capable of tolerating one failure per stage of computation.

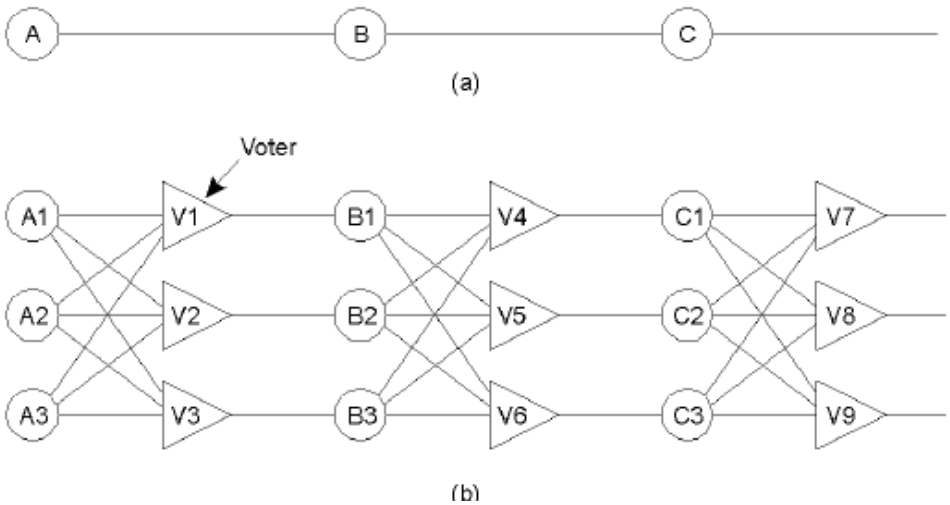

Figure 17.3: Redundancy for fault tolerance.

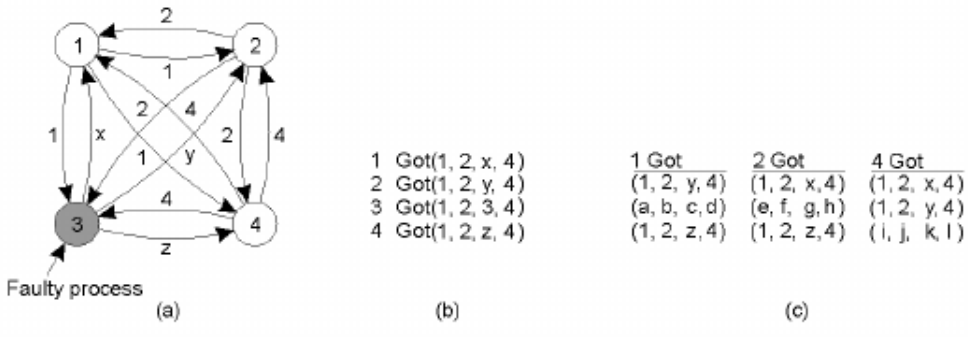

Figure 17.4: Recursive solution to Byzantine problem

17.3.3 Consensus and Byzantine Faults

Byzantine faults can be modeled as a consensus problem (Byzantine Generals' Problem) among nodes in presence of faulty processes assuming that a byzantine node will force the system to *not* reach consensus. A recursive solution to the problem is provided in figure 17.4. In this, each node collects information from all other nodes and sends it back to all others so that each node now can see the view of the world from the perspective of other nodes. By simple voting, each node can now either accept a single correct value or can identify a byzantine failure.

In a system with k such faults, $2k+1$ total nodes are needed to only detect that fault is present, while $3k+1$ total nodes are needed to reach agreement, despite the faults.