CMPSCI 677 Operating Systems **Spring 2014**

Lecture 17: March 31

Lecturer: Prashant Shenoy *Scribe: Nicholas Monath*

17.1 Replication & Consistency Continued

17.1.1 Primary-based Protocols

Primary-based protocols assume that there is a coordinator/primary replica for each data item, which is in charge of implementing the consistency method.

17.1.2 Replicated-Write Protocols

Here, there is no notion of a coordinator/primary replica, and a write can take place at any replica. All replicas will take part in the consistency method. Any replica is allowed to update and so consistency is more complicated to achieve.

17.1.3 Remote-Write Protocols

This is a primary-based protocol. Each data item has one node assigned to be the primary replica for that data item. Clients send write requests for a particular data item to their local replicas, which forward the request to the node which is the coordinator for that data item. The coordinator then enforces consistency, using any of the previously discussed techniques. Reads, assuming caching is not used, are implemented in the same way as writes. File Systems use this approach often.

17.1.4 Local-Write Protocols

This is another primary-based protocol. The read or write requests of clients are sent to a local replica. That replica now becomes the primary replica for the data item of the request. It is essentially a performance optimization of Remote-Write Protocols. Despite the performance optimization of subsequent requests taking place on a near-by replica, we have the limitation of needing to keep track of the primary replica for each data item and the primary replica might change often.

17.1.5 Quorum-Based Protocol

A replicated-write protocol that uses *voting* to request and acquire permissions from replicas. We define two quorums, a *read quorum*, notated N_r, and a *write quorum*, notated N_w, where N is the number of servers. The quorums represent the number of nodes that agree on the value for a read/write request. We make two constraints on the quorums:

1. $N_r + N_w > N$
2. $N_w > \frac{N}{2}$

A read or write request can only be processed if the above constraints are satisfied. This is also known as **Giffrord's Quorum-Based Protocol**.

A special case of this protocol is known as *read once, write all* (ROWA) where $N_w = N$.

17.1.6 Replica Management

Issues such as Geo-locations of the replicas, number of replicas etc. have to be taken care of. This will be discussed in further lectures.

17.2 Fault Tolerance

There are many reasons for a system or service to crash. In a distributed system if a machine fails, other machines in the system can share the responsibility of the crashed machine. Some nodes can fail without the whole system going down. Fault tolerance deals with the semantics and the mechanisms of dealing with failed machines. As the number of nodes in the system increases (approaches infinity), the probability of a failure approaches 1. Fault tolerance system should provide services despite faults, such as transient faults, intermittent faults, and permanent faults.

17.2.1 A Perspective

Systems are inherently unreliable. OS crashes frequently, buggy software, unreliable hardware, software/hardware incompatibilities. With more "novice" users of computer system, we want to make computing systems more reliable. We will do so by adding an additional layer of reliability.

17.2.2 Basic Concepts

Requirements for dependable systems:

- Availability: A system as a whole should be available for use at any given time. Measured by the percentage of time a system is available. 99.999% ("five 9's") is desirable.
- Reliability: System should run continuously without failure
- Safety: temporary failures should not result in catastrophic failures.

- Maintainability: A failed system should be easy to repair

There are three kinds of fault tolerance:

- Transient faults: A "one time" fault, appears once and never appears again. No noticeable reason – a bit flipped in RAM etc.
- Intermittent fault: Appears every once and a while and then goes away. Difficult to reproduce or track
- Permanent fault: Faults such as system crashes.

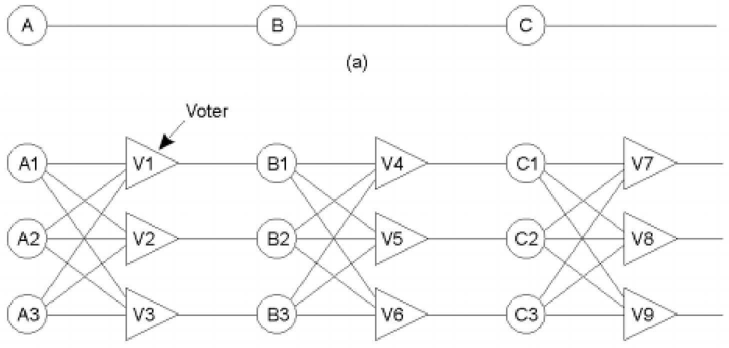
17.2.3 Failure Models

There are four types of failures:

- Crash failure: A server halts but is working correctly before the halt
- Omission failure (Receive/Send Omission): Server fails to respond to incoming requests or fails to send back a response
- Timing failure: The latency of server request causes a problem, i.e. it causes other processes to timeout
- Response failure (value failure, state transition failure): Server sends a response but the value of the response is incorrect or the server deviates from the correct flow of control.
- Arbitrary failure: A server may produce arbitrary responses at arbitrary times. Sometimes it works as expected, other times it does not. No way to predict when it is behaving correctly. Related to Byzantine faults. The most difficult kind of failure to deal with.

17.2.4 Failure Masking by Redundancy

Triple modular redundancy (shown below) is a way to design hardware that is resilient to crash failure and arbitrary failure. The top circuit in the diagram is not resilient to either kind of failure. In the lower diagram, the voter nodes pass along the majority of the responses of the incoming nodes. The voter is replicated as it too might be faulty.



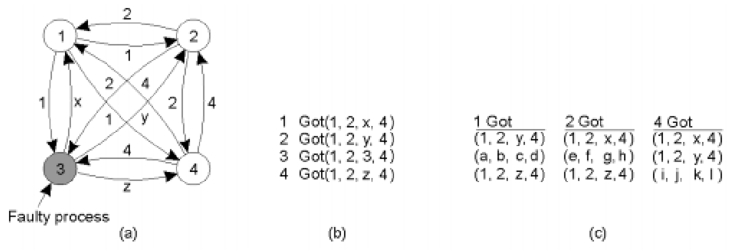
17.2.5 Agreement in Faulty Systems

Agreement can be achieved in the presence of crash failures by having replicated versions of the servers; specifically you need one more replica than the number of crashes expected. Agreement is much harder to achieve in the presence of arbitrary or byzantine failures. Most practical systems do not protect against byzantine faults.

17.2.6 Byzantine Fault

Byzantine faults of a network are often described in the simplified scenario where two perfect processes communicate over an unreliable channel. The processes need to reach agreement on a 1-bit message, but messages sent over the network might go undelivered or be changed during transmission. The scenario is described as the *two army problem*, two generals trying to agree on whether or not to attack a common enemy. The generals communicate with a messenger traveling through hostile territory. The message he delivers might not be the same as what was original sent. Additionally, the messenger might be captured and the message might go undelivered. The generals can never reach agreement in this scenario.

Byzantine faults of processes are described as the *Byzantine generals problem.* Can N general reach agreement with a perfect channel, given that M of the N may be traitors? Lamport provides a recursive algorithm to discover which of the generals are traitors. In the first step of the algorithm, the generals announce their value for a message to the other generals. Then the generals assemble vectors of the values they receive (step (b)). The generals then announce the vectors they assembled to the other processes, shown in (c). The traitors are said to be those generals who returned values other than the value returned by the majority of the generals.



```
1  Got(1, 2, x, 4)
2  Got(1, 2, y, 4)
3  Got(1, 2, 3, 4)
4  Got(1, 2, z, 4)
```

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The algorithm gives us the following results:

- To detect a k faulty processes we need $2k + 1$ replicas.
- To reach agreement between processes with k faults, $3k + 1$ processes are needed.

17.2.7 Reaching Agreement

These methods for reaching agreement in the presence of Byzantine faults are *expensive.* There have been many different strategies to reduce the cost. Papers for these will be posted.

17.3 Reliable One-One Communication

In lecture 3, we discussed RPC communication in the presence of crash failures. Now, we discuss how to achieve reliable communication when packets can be dropped by the network. To do so nodes send acknowledgements back to the sender of the message. However this does not scale well when multicasting a message to many nodes; the sender becomes a bottleneck. Negative acknowledgements can be used in multicasting. Each packet is given a sequence number. A node can determine when it has missed a packet if it sees a gap in the sequence numbers that it receives.