6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 20: November 22, 1993

Lecturer: Bill Weihl *Scribe: Bienvenido Vélez-Rivera*

1 Lecture Plan

- Finish Talking about Distributed Transactions
- Start Talking about Data Replication

2 End of Distributed Transaction Processing

Figure 1 illustrates the basic idea behind distributed commit. During the first phase, the coordinator sends a prepare to commit message to every participant. Each participant in turn, prepares to commit the transaction by forcing its log to stable storage. If a participant cannot force its log, it sends the coordinator an Abort vote. Otherwise, the participant responds with a commit vote. If the coordinator receives commit votes from all participants, it starts running a consensus algorithm to reach agreement and chooses commit as the allowed value. If some participant responds with abort then the coordinator sends an abort outcome to every participant. The common way of implementing distributed commit is 2-

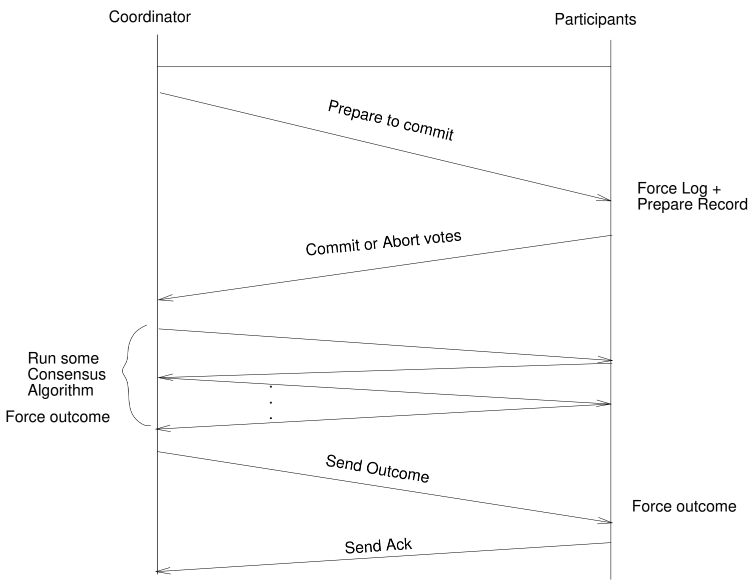


Figure 1:

Phase Locking (2PC). In 2PC no consensus algorithm is run. The coordinator simply records the outcome in stable storage and sends it to all participants.

2.1 Optimizing 2-Phase Commit

The problem with 2PC is that it requires the coordinator to force to stable storage enough information to be able to reconstruct the transaction database in the event of a crash. A naive implementation of 2PL also requires the participants to return Ack messages to the coordinator for it to be able to garbage collect the information kept in the transaction database.

The paper by B. Lampson and Lomet addresses the different optimizations that have been suggested to reduce the number of forced logs.

One of this optimizations is to presume that a transaction has aborted if its information cannot be found in the transaction database. The coordinator does not have to force recovery information unless the transaction commits. In the event of a coordinator crash, if some participant asks for the outcome then the coordinator (after recovering) will lookup the transaction in its database. It then finds out that no such information exists and responds with an abort outcome which is the correct answer.

2.2 Pressumed Abort

One can go one step further any remove information about all aborted transactions as soon as the abort outcome is known. The database will only keep information about active and committed transactions. Participants of an aborted transaction do not have to respond with Acks. Therefore, this optimization results in savings in both messages and forced log writes.

2.3 Pressumed Commit

The problem with the pressumed abort optimization is that it optimizes the rare case instead of the common one which is commit. Most transactions commit so we want to optimize this case.

Solution: Keep track of aborted transactions

Consider the following scenario:

- Coordinator sends prepare
- participants respond with commit
- coordinator crashed (transaction database is lost)
- coordinator looks up and finds no information
- coordinator responds with commit (since no info was found)

The outcome in the previous example is not correct because the coordinator crashed, and thus the transaction should have been aborted.

Solution: Force begin of transaction record *before* sending prepare to commit messages. This record must contain enough information for the coordinator to recover the transaction database in the vent of a crash in this phase of the algorithm. This introduces an additional forced log write.

Eventhough the pressumed commit optimization introduces an additional forced log write, its savings in terms of messages are significant since transactions that commit do not require the participants to respond with Acks.

2.4 New Pressumed Commit Optimization

In the paper by Lampson and Lomet handed out in class, the pressumed commit optimization described above is further optimized to avoid the additional forced log write. The new algorithm achieves this goal by keeping some information about each crash forever.

Problem set 10 is also related to this optimization.

2.5 What if a Participant Crashes before forcing its log

If the participant crashes before forcing its log, then there must be some way for the coordinator to know about this. Otherwise a participant can recover, do further work on behalf of the transaction and respond with a commit vote if asked to prepare to commit.

A simple solution to this problem is to keep a crash count at each node. Every time the node crashes the count is incremented and stored stably. The crash count must be sent together with all messages between coordinator and participants. The coordinator can tell that there has been a crash if it notices a change in some participant crash count.

This is the end of Distributed Transactions. For the rest of the lecture will talk about Replication and Availability.

3 Replication and Availability

3.1 Objective of Replication

We want to continue providing services and data availability even in the precense of crashes.

3.2 What causes failures?

One way to classify the causes of failures is

- Hardware failures—*e.g.*, some component fails
- Software failures—BUGS!

In a well engineering software system the statistic is 2-3 bugs per 1000 lines of code
Other sources of failures are:

- Operation failures—somebody pushes the wrong button
- Environment failures—machine room flods, earthquake
- Maintenance—repairman trips over the power cord

The biggest problem is *Software*

The book by J. Gray and A. Reuter "Transaction Processing" talks about software problems among other sources of failures.

One problem is that the process of fixing bugs introduces more bugs. Some bugs, however, occurr very infrequently. Some people concentrate fixing the frequently occurring bugs and simply document the rare ones. Sometimes, people even decide to stop fixing bugs at all. At some point the rate at which new bugs are introduced by fixing other bugs gets intolerable.

3.3 Dealing with Environmental Failures

The basic way to deal with this type of failure is redundancy. Sometimes though, some redundancy is not enough.

Last March, the bombing of the World Trade Center caused some company to move its computer operations to a backup site in New Jersey. The roof at the backup site collapsed leaving the system inaccessible for some time.

Redundancy helps but it has its limits. Its can be more effective if the redundant copies operate in different environments and under different conditions.

3.4 Dealing with Maintenance Failures

These types of failures were more common some years ago when some equipmennt required periodic maintenance. For example, some disk units required periodic replacement or maintenance. In 1990, after maintenance free disks became available, the mean time to failure for a disk increased from 8 thousand hours (about a year) to 100 thousand hours.

By eliminating maintenance and by looking at sources of failures, we can redesign systems that fail less frequently.

3.5 Dealing with Operational Failures

The basic way is to automize as much of the operation as possible to avoid human intervenction.

Question: Isn't the automation of operations simply moving the sources of failures to software?

The answer is yes to some extent. However, history have shown that mean time to failures can be increased by automation of operations. There is a positive net gain in automation of operations.

3.6 Dealing with software problems

If you can spend a lot of money, you can get better software:

- comercial code—expensive
- production code—more expensive
- space shuttle code—even more expensive

Can divide software faults in soft faults and hard faults:

- soft faults—restarting the system should fix things
- hard faults—if retry, system will fault again in the same way

Jim Gray calls soft bugs Heisenburg and hard bugs Bohr Bugs. Heisenburg bugs contitute 99% of all bugs. One way to deal with this type of bug is to restart the system to some state and retry the operation that failed. Bohr bugs are harder to deal with. You can spend money and reduce the number of these bugs, but can never eliminate them completely.

Another approach is called N-versioning of software and its derived from an analogy with hardware replication. The idea is to build N versions of the software independently and periodically check for agreement. The hope is that not all versions will fail simultaneously.

The assumption that makes N-version more effective is that failures are independent of one another. This is not necessarily true. Programmers tend to think in similar ways and to make similar interpretations or misinterpretations of specs. Thus in fact, systems developed independently may fail under the same circumstances.

Formal methods have not been successful in dealing with software bugs. They are usually too difficult and too expensive.

3.7 Dealing with Hardware Problems

The rest of the class we will talk about how to deal with hardware problems by means of replication.

3.8 Strongly Consistent Replication Schemes

In this type of scheme users are unaware that data or services are replicated. Their view of the system is that no replication is being done. For the remainder of the lecture we will concentrate on strongly consistent schemes.

There a fundamentally two approaches to strongly consistent replication:

Voting or Quorum Consensus We will talk about this later

Primary Copy or PrimaryBackup Designate one of the replicas as the primary. If the primary fails, one of the backups takes over. This has the advantage that recovery from failures is fast.

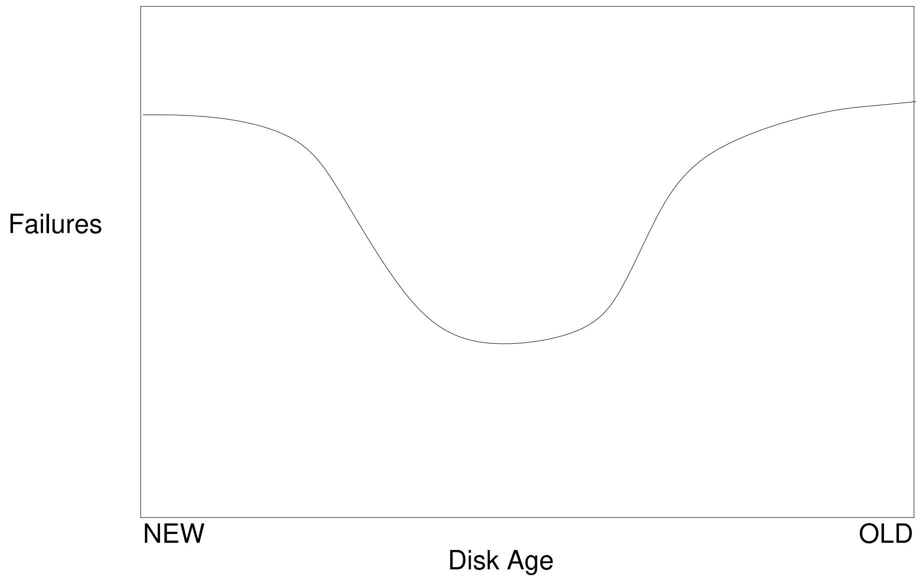


Figure 2:

3.9 Loosely Consistent Replication Schemes

In a loosely consistent scheme updates are not atomic. They propagate in the background. Meanwhile, the system remains in an insonsistent state. However, some applications can tolerate this level of inconsistency. Some examples of such applications are a routing process or a name server. Handout 47 discusses some techniques for implementing loosely consistent schemes.

3.10 How replication helps

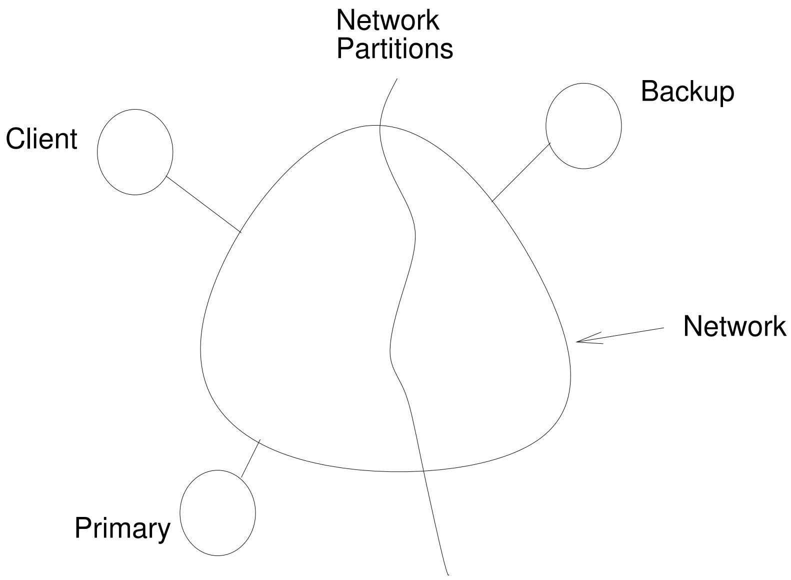
In 1987 for a single disk the Mean Time Between Failures (MTBF) was estimated to be about 6 years. If duplexed disks are used, then the new MTBF is the time it takes for one disk to fail before the second disk, which also failed, is fixed. The predicted MTBF for duplexed disks is about 60,000 years. However, the observed MTBF is really "just" 1300 years. This discrepacy can be attributed to the fact that failures are not completely independent and furthermore not uniformly distributed across the lifetime of the device.

Typically disks tend to fail more often when they are new and after they become old, as illustrated by the Figure 2. Duplexed disks can in fact have synchronized lives. It may be the case that both are new or old simultaneously.

Nevertheless, replication can do a good job at reducing MTBF's.

3.11 Primary Copy techniques

The spec we want for a strongly consistent replicated service is given at the top of handoout 47. The system has some state. Actions are allowed on bahalf of clients as long as all the



necessary state is available. Otherwise, the service can raise a fail exception or can execute the action at some later time.

The spec allows for actions to be completed without the client being able to tell. This can happen for example if some request os processed by a backup service but the network partitions. We can define a request to be done if the client eventually knows about it. The second specification in Handout 47 adds the notion of failures. Whenever a thread fails, an action can be lost or postponed.

Note: There is a typo in Handout 47. The body of the Fail thread should be inside a DO..OD loop.

What the spec says is "If the system doesn't fail, then if the client requests some action it gets done." It also specifies that requests are to be processed in the order in which they were requested by clients.

3.12 Implementations of Primary Copy Mechanisms

We must guarantee that all replicas see the same actions and in the same order. The natural thing to do is to use a consensus algorithm among replicas to agree on which sequence of actions to execute and in what order.

In the implementation given in Handout 47, the job of each replica is to fill an array containing the action to be executed in some order. Consensus must be reached on which array index to associate each action with. Once consensus is reached, each replica executes the actions in the order agreed starting from some initial state.

One invariant mantained by the implementation is that the index chosen for each action has the property that all participants have reached consensus in all previous indices.

One replica is designated as the primary and receives all the requests.

3.13 Optimizations to the Implementation

- Make fisrt round of paxos only once. Only do first phase when there is a failure that requires the primary to be changed. This optimization to be explained in the next lecture.
- If the primary has too many request, can do second round of Paxos in the background and respond to the client immediately. The price to be paid for this optimization is that some actions can be lost if the backups have not executed these actions at the time of a crash.