Transactions in Distributed Systems

Sergio Caltagirone
University of Idaho
scaltagi@acm.org

1. Introduction

Everyone these days knows how convenient it is to have all of your files in one place. Even with the rise of the personal computer from the thin clients of the past, the need is still there. I, personally, have three personal computers, and it is very inconvenient not to have all of my files available on each system. However, in the past, a distributed file structure was more necessity rather than convenience. Because each client machine only had a small amount of RAM and disk space, the entire file system could not be kept on every machine. Additionally, the problem of keeping all of those machines consistent was an enormous, if not impossible task. If one client changed a file, every other client must also change the file in the same way so that each client has the most current version of the data. This made a distributed file system necessary.

However, the problems of a distributed storage system do not stop with consistency. Other problems emerged as systems such as Andrew, Coda, Quicksilver, and NFS were developed and researched, problems such as performance, and availability. As the storage systems grew away from the local machine, the problem of accessing and updating files in an efficient manner became apparent. Saving your file over the network is never as quick as saving the file to local disk.

Also, if the central file server is unavailable, or if your local client is no longer connected to the network (mobile environments), then how does the user continue to work and use the system? Plus, if we choose to solve consistency with locking the file your using, then what effect does that have on the availability to users who want to read it? These problems that were found in the 1980's, were seriously studied by researchers, and have evolved into the distributed systems we see today, from NFS to mobile phone networks.

This paper will discuss these issues with regard to distributed storage systems, and how research into transactions have helped alleviate some of these problems, while introducing some others. The storage systems we will be concerned with are distributed file systems, and support for database systems. Additionally, we will discuss how transaction research has evolved to meet the demands of new environments such as mobile computing, as well as the classic systems, ARGUS, Coda, and Quicksilver. In the end, the reader should be informed as to the problems of transactions, their limitations, how those limitations have been overcome, the evolution of transaction research, and its application to multiple environments.

2. Distributed File Systems

A distributed file system is a system where files dispersed to non-local systems, but look local to the user accessing them. An example of this would be that if my files were kept on another machine far away, but when I logged into my machine, they look and act as if they were right there.

In [1], Chow and Johnson identify two characteristics of a distributed file system (DFS). The two characteristics are dispersion and multiplicity. There must be multiple files in multiple locations, and multiple users accessing those files in multiple locations. Any system satisfying those characteristics would be considered a distributed file system. What is important is that given these characteristics, to the user it must be transparent; in other words, it must seem as though there is one user working on a file that is in one place.

The positive benefits of a distributed file system are that users can be on any client and access the files, fault tolerance can be easier to design for, easier to design security mechanisms to control file access, and multiple users can read/write a file at the same time (if designed correctly). With these benefits, it is easy to see how distributed file systems became popular. They provide a system the power to have many users with many locations access the same data. However, there are serious problems to consider when researching a distributed file system; some of the most serious are described next.

2.1. Architectures

Distributed file systems come in two flavors, client-server and peer-to-peer. Each architecture type has its own unique set of problems and solutions.

2.1.1. Client-Server

In a client-server architecture, a single machine, or set of machines act as designated servers, which hold the 'official' copy of all files. If a change to a file is made, then the changes are sent to the server, which then records those changes and updates the official copy so that any requests for that file receive the newest edition.

However, this architecture has many problems. First, if the server is unavailable, then the files are unavailable because no copies of those files exist in the system outside of the server. Additionally, if there are a lot of requests for a file, then the server can be overloaded causing it to neglect its duties to some of the clients.

The client-server architecture though has many merits. First, it is very simple to design and implement. Second, the need for infrastructure is minimized. It also helps solve many consistency problems (as described in the next section).

2.1.2. Peer-to-Peer

In a peer-to-peer architecture, each client machine keeps its own unique piece of the file structure. If another client needs a file that it doesn't have, then it simply contacts the client that does have it, and it is sent to them. However, this architecture requires one more component, a directory server. This server will tell each client where the parts of the file structure are stored; so that when they are needed a client looks up what client has that file and then goes and requests it.

There are several problems with peer-to-peer; the first is that if one client is unavailable, that section of the file system is similarly unavailable. Secondly, depending on network topology, the communication latency may be greater.

However, there are several positive issues that come alone with a peer-to-peer architecture. First is the issue of fault tolerance, if one machine is unavailable, only that section of the file system is lost and no others – so any user working on another file from another client is unaffected. Second, is that it is more difficult to overwhelm many clients than it is to overwhelm one server.

3. Problems with DFS

Although a distributed file system (DFS) provides system managers and users many benefit, its design is also fraught with many problems. These problems are complicated even further when it is discovered that to decrease one problem leads to an increase of at least one of the others. Therefore, each DFS must decide what are its primary (realistic) goals, and be designed to meet those goals.

3.1. Consistency

The problems of a distributed file system begin with the problem of consistency. Consistency attempts to say that every user will be working with the same data at all times. For example, if user A writes file X, then whenever user B reads file X will see the changes that A made. Additionally, if user A writes to file X, and user B writes to file X, both users will not write over each other, plus be able to see each other's changes.

However, that example is one from an ideal world. In the real world, it is very difficult to achieve that amount of consistency, at least without sacrificing performance or availability to a greater extent. To solve this problem, DFSs implement many mechanisms to improve consistency. Some of them are: locking, sharing, transactions, and message passing.

3.2. Performance

The problem of performance comes from many parts of a distributed file system. Part of the performance problem is latency. This latency is usually derived from a network. When a user requests a document, it takes time to transfer that document (or part of the document) across the network. This is because networks are very slow compared to the local system, and a poorly designed DFS can frustrate users and cause consistency issues. This is because a user may be reading a document before the changes of another user arrive via the network.

In addition to receiving a file through the network, another part of a DFS that causes performance issues is propagation of messages. As described before, if a user changes a file, then everywhere else that file is kept needs to know about those changes to update that file. However, it takes time for those messages to be sent, travel, be received, and changes executed. And depending on the system, if every change is sent over the network, then a user working on a document can be frustrated because the computer's resources are being spent on sending messages rather than editing the document.

3.3. Availability

Availability refers to a users ability to interact with the system at a given time. With distributed file systems, availability refers to the ability of a user to either read or write to a file when they want to. The problem of availability is strongly connected to the problem of consistency.

The issue of consistency is that it is difficult to support two users writing to a file at the same time (write-write), or one user reading and one user writing (read-write). To solve this, most DFSs attempt to limit the availability of a resource while a write is occurring. On the other hand, a DFS might instead abandon consistency in certain situations and accept that a user will be reading old data.

4. Transactions

Transactions were suggested as a method of solving these problems with distributed file systems. More specifically, transactions directly apply to the problem of consistency, but by doing so also speak to the problem of availability, which then allows a DFS to support better performance. All of which enhance the overall structure of a distributed file system.

4.1. Definition

Specifically, a transaction is a sequence of events that are treated as a fundamental unit of access. [1] Generally however, a transaction has three properties. [2] The first property is consistency. Consistency with regards to a transaction means that a transaction does what it is supposed to do, correctly. The second property is atomicity. Atomicity means that either the transaction is successful and executes correctly, or it doesn't – the transaction cannot only partially execute. The third property is durability. Durability is the property that guarantees that once a transaction is done, it cannot be undone; but the effects can be reversed by another transaction.

4.2. Gray's Model of a Transaction

In Jim Gray's seminal paper on transactions, he presented several important ideas of transactions; one of those was a model of what a transaction is to a computer system. [2] In the model, a system is nothing more than a sequence of states. A transaction, therefore, is nothing more than a tool that transforms the system from one consistent state to another. Since a transaction is atomic and durable, a transaction has only two outcomes: commit, where the transaction was successful and the system has changed states, or abort, where the transaction failed and the system has not changed states.

A transaction may contain several actions, which themselves are transactions. Therefore, a transaction may be simple or complex. A simple transaction is one that is simply a series of actions. A complex transaction is one that contains one or more transactions, and may contain concurrency (multiple running at once) and conditional logic (one may run dependent on the outcome of another). However, in the case of complex transactions, the internal transactions, although following the same principles as the primary transaction, are invisible to the system – it looks like a simple transaction to anyone but the transaction.

4.3. How to Build a Transaction System

Its nice that Gray provided such a nice model of what a transaction is. But how can a transaction system be built? Gray also provides some solutions to that question in the same paper. [2] Gray provides two example solutions, Time-Domain Addressing, and Logging and Locking. Each of these provides a different approach to implementing the transaction concept.

4.3.1. Time-Domain Addressing

The problem with most systems, as Gray describes, is that most systems overwrite the old object with the new object – thereby removing any opportunity to rollback and to document the changes to the system (necessary if a transaction fails). One solution is time-domain addressing, where an object is evolved by appending the new value to it (rather than replacing it), while "the old value continues to exist and may be addressed…"[2] This solution is somewhat equivalent to versioning the objects in the system – older versions continue to exist and are cataloged.

However, this system has some serious flaws. The probably is that not only do the old values need to be kept of an object, but the entire chain of dependencies, "so that if an error was discovered, the compensating transaction could be propagated to each transaction that depended on the erroneous data." [2] The information, because of this flaw, was found to grow exponentially – which is a problem for systems with limited storage space.

Another problem is that I/O activity is increased because each time an object is read, then a write has to be performed on the object documenting the write. However, these problems are performance issues and do not have any bearing on the actual effectiveness of a transaction.

4.3.2. Logging and Locking

Gray attributes the idea of logging to the Greeks. Gray notes that "the basic idea of logging is that every undoable action must not only do the action but must also leave behind a string…which allows the operation to be undone." [2] The same technique must also be applied to redoable actions. The logs should be kept in stable storage so that the system and the log have "independent failure modes."

Then to reconstruct a transaction, a new transaction is created which then reads the log and simply reconstructs the old state. The log does not have to keep the entire state of an object, but only the changed parts or fields. The problem with this solution is that is difficult to support concurrent transactions.

Therefore, Gray proposes a locking mechanism. Each object, as it is being accessed is locked. However, there are two types of locks: update locks, and reader locks. This allows a writer to lock an object for reading, but allows multiple readers to access the object concurrently.

4.4. Limitations of Transactions

Gray notes that although the concept of a transaction is very helpful to certain problem domains, such as airline reservations, banks, and car rentals, there are some limitations that exist. The three limitations that are discussed are nested transactions, transaction duration, and transaction programming.

4.4.1. Nested Transactions

Although a transaction may look like a single, atomic function which implements a sequence of actions, those actions, at the next lower level of abstraction may indeed be transactions. The problem is that undoing a transaction requires a compensating transaction, which in turn, must undo all of the underlying transaction, which were encapsulated in the one being undone.

4.4.2. Transaction Duration

Most transactions are assumed to last only a few seconds or less. But what of transactions which are expected to last days or longer? So far, we have sidestepped the problem of deadlock because it is very uncommon with transactions, even when there are hundreds of concurrent transactions every instant. However, as the lifetime of a transaction is increased, the probability of a deadlock occurring similarly increases. In fact, "the frequency of deadlock goes up with the square of the multiprogramming level and the fourth power of the transaction size." [2] Therefore, deadlock becomes a very important issue.

Gray suggests that the method of solving this problem involves decreasing the amount of consistency that is achieved by the transactions; and also decrease the number of locks, limiting locks to only active transactions – or ones that update an object. However, Gray notes that there is no basis to how well this "trick can be generalized." [2]

4.4.3. Transaction Programming

The third and last problem of transactions that Gray identifies is that of a separation of the concept of a transaction with a programming language. The problem is how do you implement the concept of a transaction in the system such that it is guaranteed that all of the properties hold? Here, Gray thinks that by including the language of BEGIN, SAVE, COMMIT, and ABORT, and also that objects generate undo and redo logs, programming languages can easily support transactions. [2]

In [3], Liskov and Scheifler discuss their implementation of a system, ARGUS, which was designed for the express purpose of supporting robust and distributed programs. More specifically, ARGUS was the first system to support atomicity at both the language and hardware level. Their system allowed for a fault resilient implementation, where if an atomic action failed, the system would automatically retain consistency – even with concurrent actions.

They supported this functionality using a simple locking model. In the model, the usual locking rules applied: "multiple readers are allowed, but readers exclude writers, and a writer excludes readers and other writers." Additionally, with the occasion of every lock, a version of the object is made previous to the lock for the purpose of recoverability. However, if the action was successful the stored version of the object was discarded, with a log kept of the change for later reversal if necessary. The system also supported actions with multiple, concurrent, nested actions.

The actual language implementation relied on two concepts, guardians and handlers. Guardians are dynamically created processes that run at each node and control the access to resources for processes at that node. The guardians execute the handlers and synchronize them regarding access to resources. If the system crashes, the guardian is recreated which then reinitializes the objects and resumes computation.

The handlers are wrappers for transactions. They allow the system to gracefully handle exceptions and errors in actions. They also abstract transaction actions away from the guardians. By doing this, nested actions and concurrency is easily supported. ARGUS is a good example of how transaction support can be implemented in both hardware and software to build a fault resilient, concurrent, and consistent system.

5. Database Systems

The concept of a transaction first came from the database research community. It is relevant, therefore, that this

paper at least discuss some of the issues that were presented in that forum. Specifically, how a distributed system can support a transaction based database system.

Two of the most important aspects for a database management system, with regard to the operating system support are crash recovery and consistency. [4] Since the database relies heavily on the underlying operating system to provide a file system to store the database data, then there is, of course, a strong need by a database to utilize the operating system for protection of those files against crashes and inconsistencies. Support for these protections are usually implemented using transactions.

5.1. Crash Recovery

The purpose of recovery is "used to restore data in a system to a usable state." [5] Verhofstad provides seven techniques for recovery, but only three will be discussed: audit trail, differential files, and backup/current versions. An audit trail is a record of the sequences of actions taken on the system. A differential file records all of the alterations made to objects in the system. The backup/current version technique is a method of containing previous values of objects so that they can be later recovered. [5]

Notice how audit trail and differential files mechanism, which is the logging of all actions taken on the system compares to the locking and logging concept of transaction implementation given by Gray (discussed earlier). Also notice that the idea of keeping backup versions is similar to the time-domain addressing also discussed by Gray. This implies that transactions can be used implicitly to support recovery of a database.

Additionally, Stonebreaker in [4] describes crash recovery as applying to the abortion of a transaction in mid action – causing the system to revert to a state previous to the start of the transaction (since transactions are all or nothing). According to Stonebreaker, a database management system accomplishes this through the use of an intentions list. An intentions list is a record of all of the updates performed on a database by the transactions. Transactions make this system work because after a transaction has accessed and updated the database, its last step is to maintain the intentions list by including the actions it performed. At this point the commit flag is set, indicating the transaction was successful, and the intentions list is forced to disk so as to be protected in the event of a crash.

During recovery, the database management system examines the intentions list and processes the actions to return the system to the pre-crash consistent state. However, if the transaction was not complete, the transaction did not set the commit flag, and therefore those actions are not performed – thereby backing out of the transaction.

5.2. Consistency

Stonebreaker argues that transaction support for crash recovery and consistency should completely reside in the operating system because it would negatively impact buffer management. For consistency control, this is a problem because buffer management is what determines the level of consistency in the operating system. The argument is that when a transaction completes, it is usually acting within user space buffers, a commit signal then needs to be sent to the kernel so that these buffers can be sent to disk. However, therefore a buffer manager must have knowledge of transactions, and the functionality of transactions is duplicated in both the database management system and the operating system; which is unnecessary and forces the database to do some of its own buffering. [4]

However, most operating systems are not fast enough to provide consistency control for the database, and therefore until operating systems improve for database environments, the best choice is for databases to provide their own control. [4]

6. Mobile Computing

A mobile computing environment is unique from almost all others. It is unique because it faces several challenges that originate from both its environment and goals. One of those problems is disconnected operation, most mobile systems continually move in and out of the network, while their users expect that at least most of the functionality (including modifying distributed files) remain unaffected. Therefore, the goals of the system are too not only make its distributed nature transparent from the user, but also make network connectivity transparent.

This poses a gigantic problem for consistency. If a user is updating an object without contact to a network, what of other users attempting to read the object? Obviously, if there is no connectivity, the readers will not be able to see the changes being made to the object outside of the network. Additionally, what if another user is updating an object on the network, while the user is changing the object off the network? What happens when the user reenters the network – which changes over-ride?

6.1. Isolation-Only Transactions

The Coda file system, based out of Carnegie Mellon University, has already addressed these problems in an

ingenious and very efficient manner. Their goal was to provide disconnected operation to a user through "a special form of client disk caching…" [6] In this way, when the client is disconnected, all services to the file are based on its local cache, but when the client is reconnected to the network, the updates are reintegrated into the distributed file system.

To implement this reintegration, the designers of Coda turned to transactions, specifically isolation-only transactions (IOT). Isolation-only transactions are a sequence of file access operations that guarantee certain properties that are relevant to mobile environments. However, IOTs do not guarantee atomicity and only conditionally guarantee durability (permanence). Therefore, if a transaction fails, it may leave the system in an inconsistent state, and that a transaction, once executed, may not be permanent.

An IOT works by executing the entire transaction on the client's local disk cache, where the transaction does not access any remote files and the actions are logged. If the client is connected, then the transaction is committed to the network, otherwise the transaction is said to be *pending* until the client connects the network. To other processes on the client, the pending transaction is treated as committed.

However, to commit a transaction, a validation is necessary. For a transaction to be considered valid, and therefore committed, it must satisfy the isolation property – where the results of the interleaved execution are equivalent to the serial execution of the same actions on just the client's local cache. Additionally, it must be globally serializable, meaning that the set of actions that were executed on the client's cache can be added to the global (network) set of actions and the system and still be considered serializable. If the transaction satisfies these properties, then the updated files are transferred onto the network overwriting the previous copies. [6]

If a transaction is not found to satisfy these properties, then a series of resolution options exist. Lu and Satyanaranyanan present four options. [7] The first is re-executing the transaction on the up-to-date files, if no inconsistencies can be found to exist. The second option is to invoke the transaction's application specific resolver. This option attempts to request that the application that created the files resolve the conflict using application specific knowledge about the files. The third option is to abort the transaction(s) that are not compliant. The fourth, and last, option includes notifying the user so that they can manually resolve the inconsistencies.

The problem is that although these options attempt to exhaust all ways to resolve consistencies, it cannot guarantee that even though the user executed the transaction, the transaction will eventually be accepted and committed globally. Additionally, if a transaction is executed on the local disk cache, when reconnected to the global system, it may cause inconsistencies. To solve this, we attempt to resolve the inconsistencies, but if they are irresolvable, then the transaction is discarded.

Although these problems with consistency are serious, the problem of disconnected operation is very difficult, and this may potentially be an excellent solution to the problem. Overall, the system seems to work well and can solve most problems, especially if the transaction granularity is small enough – the probability of a conflict is greatly reduced and the value of the system is increased. [6]

7. Quicksilver

It is sometimes helpful to get the prospective of additional case studies on a subject, this is the intention of this section. In 1988, IBM Almaden Research Center built a distributed operating system called QuickSilver, and utilized atomic transactions as their chosen failure recovery mechanism. They documented their research in [8]. What is unique about QuickSilver is that their transaction protocols and logs were exposed to clients and servers, allowing them to "tailor their recovery techniques to their specific needs." They defend that this choice was made to "allow servers to make their own choices to balance simplicity, efficiency, and recoverability." [8] As seen earlier, most systems hide these in either a programming language or the operating system.

The problem QuickSilver faced was that they were built on the client-server architecture. Therefore, the server was required to document the state of the clients; which caused problems because each needed to be aware of the failure states of the other. If a server crashes, it must therefore recover quickly and cleanly for obvious reasons.

QuickSilver utilizes transactions to recover from many types of failures, global memory, client crashes, server crashes, and file system failures; all which rely on the transactions to rely on the file system to recover their state. The QuickSilver implementation includes three components, a transaction manager which manages commit coordination between the servers, a log manager which keeps the common recovery log, and the deadlock detector which detects deadlocks and aborts the transactions causing the deadlock.

How the system works, is that clients and servers utilize IPC message passing. Every IPC message belongs to a transaction, which has its own unique id, and that id is communicated within the message. When a server

accepts a message and then executes the action on behalf of that transaction. The transaction manager places the message in the protocol, which is then logged using the log manager. The deadlock detector is always running in the process, monitoring transactions to make sure they are always making progress and never waiting in a deadlock scenario.

There are several protocols that QuickSilver can manage, and each server communicates its own commit protocol. The two basic models of commit protocols are the one-phase and the two-phase commit. The one-phase protocol is very simple; a server transmits a transaction, and is notified that the transaction is complete. The two-phase protocol includes two steps, one locking the resource, and two, unlocking the resource when it is done. [1] The two-phase protocol is must more reliable with regard to consistency control.

If the system experiences a failure, then the node that experienced the failure contacts its Log Manager who then furnishes the log which the transaction manager can utilize to reconstruct the last consistent state on the node. Notice that each node drives its own recover, which the authors admit can decrease performance; but this performance hit can managed by using replicated logs, and backpointers, which allow a server to modify data in place and reply their aborted transactions. [8]

In the end, the researchers concluded that the recovery manager mechanism was efficient enough for most systems, while also minimizing the number of communication messages and CPU overhead. The QuickSilver system also acted as a proof-of-concept for the server-driven recovery process and algorithms. Overall, the system was very well designed and performed well in the bench tests provided in the paper. However, they note that future work needed to be done with nested transactions and object managers. [8]

8. Conclusion

This paper has attempted to give the reader an overview of the seminal research accomplished with regard transactions in distributed systems, while focusing on storage systems. It began with an overview of what a distributed file system is, and the problems associated with that model. It then described Gray's original work with transaction models while noting some of the limitations of transactions. These limitations, as seen in the paper, with further research, have been able to be overcome – and successful implementations have been developed.

The second part of the paper described the original work in transactions in the database system research community. To contrast this, the next section described some more contemporary work done with transactions in mobile environments. And finally, the paper presented one of the early, and successful, implementations of transactions with regard to distributed operating systems as applied to recovery.

Overall, the reader should now better understand the evolution of the transaction research, the problems that were discovered, and the methods and implementations that were used to solve them. Additionally, the reader should also see how transactions are used to solve the problems of a distributed file system, both classic and in a mobile environment. Lastly the reader should understand the utility of the transaction model, and that while limitations exist, they can be successfully overcome.

9. References

[1] R. Chow and T. Johnson, *Distributed Operating Systems and Algorithms*. Reading, Massachusetts: Addison-Wesley, 1997.

[2] J. Gray, "The Transaction Concept: Virtues and Limitations," presented at 7th International Conference of Very Large Databases, 1981.

[3] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 381-404, 1983.

[4] M. Stonebraker, "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, pp. 412-418, 1981.

[5] J. S. M. Verhofstad, "Recovery Techniques for Database Systems," *Computing Surveys*, vol. 10, pp. 167-195, 1978.

[6] Q. Lu and M. Satyanaranyanan, "Isolation-Only Transactions for Mobile Computing," *Operating Systems Review*, vol. 28, pp. 81-87, 1994.

[7] Q. Lu and M. Satyanarayanan, "Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions," presented at 5th IEEE HotOS Topics Workshop, Orcas Island, WA, 1995.

[8] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan, "Recovery Management in Quicksilver," *ACM transactions on Computer Systems*, vol. 6, pp. 82-108, 1988.