

# Transactional File Systems Can Be Fast

Barbara Liskov and Rodrigo Rodrigues

*MIT Computer Science and Artificial Intelligence Laboratory*

## Abstract

Transactions ensure simple and correct handling of concurrency and failures but are often considered too expensive for use in file systems. This paper argues that performance is not a barrier to running transactions. It presents a simple mechanism that substantially lowers the cost of read-only transactions (which constitute the bulk of operations in a file system). The approach is inexpensive: it requires modest additional storage, but storage is cheap. It causes read-only transactions to run slightly in the past, but guarantees that they nevertheless see a consistent state.

## 1 Introduction

File systems (either local or distributed) typically provide weak semantics with respect to concurrent updates. For instance, most file systems allow two processes to concurrently write to the same file, and overwrite each other's changes. To avoid these problems, applications use different tricks like creating lock files, or creating temporary copies of a file and using atomic rename, to implement serial semantics. Requiring applications to resort to such tricks is not good: it increases the complexity of implementing applications that require concurrent access to shared files, and makes these programs more prone to errors, such as lack of proper synchronization or deadlock.

These problems can be avoided in a transactional file system. Transactions [6] guarantee that concurrency and failures are handled properly; as a result they can make the programming of concurrent access to shared data simpler. A transaction consists of one or more accesses to a file or set of files, e.g., to read, write, open, close. The application code ends a transaction by requesting a commit or abort. A commit request may fail (causing an abort); if it succeeds, the system guarantees that the transaction is serialized with respect to all other transactions and that all its modifications to the files are recorded reliably. If the transaction aborts, it is guaranteed to have no effect and all its modifications are discarded.

There have been several proposals for file systems with transactional semantics, e.g., QuickSilver [7] and Inversion [12]. These proposals point out the (obvious) advantages of transactions, but nevertheless transactional file systems have never gotten much traction. We believe that a major reason is performance (see [15] for a discussion of the

subject). Extra processing is required to do the concurrency control needed to ensure non-interference among concurrent accesses, e.g., by providing two-phase locking [6]. Also extra processing is required to process commits and aborts. These costs are especially pronounced when the file system is distributed at multiple servers. In this case a transaction may make use of data stored at more than one server, and therefore requires two-phase commit [5].

In this paper, we present a simple mechanism that will make transaction processing much faster. The technique allows us to avoid concurrency control and commit processing for read-only transactions. Not only does this greatly speed up execution of read-only transactions, but it can improve performance of read-write transactions since we no longer need to deal with contention between read-only and read-write transactions.

Our approach is designed for a system in which the file system is stored at servers and applications run at (remote) clients, on cached information. It works whether there is just one server, or many. It is particularly appropriate for a large-scale system in which files are stored at many servers that are widely distributed and may be very far away from clients, since it allows cached data to be exploited aggressively, and thus reduces communication between clients and servers.

Our approach works by running read-only transactions slightly in the past, using a somewhat dated, but nevertheless consistent, version of the system state. Read-only transactions are assigned a position in the serial order; this way we can guarantee that they see a consistent view. But by allowing them to run in the past, we avoid contention with current read-write transactions, and also we allow them to take better advantage of data already present in their local cache. Consistency is guaranteed even when there are multiple servers and the transaction makes use of objects stored at different servers.

We expect our technique to be extremely effective. This expectation is based on two observations. First, most transactions in a file system are read-only. This observation is supported by studies of file system workloads where reads largely dominate writes [17].

The second observation is that most applications do not mind reading data that is slightly stale, provided the data is consistent. This use of out-of-date state is obvious when you think about the semantics provided by transactions: they ensure that the state is consistent at the moment of commit, but that moment is a bit of an accident: the commit could have

happened a little earlier or a little later. And whenever it happens, the observed state can change immediately afterwards. The need for consistency is also obvious: without it, applications can encounter inconsistencies (e.g., violated invariants), which either cause them to fail in unexpected ways, or, to avoid this, require more complex code.

Therefore we expect that most transactions will take advantage of our new technique, since almost all transactions are read-only, and of those almost all will be able to use a slightly out-of-date state. We should point out that the state observed by one of these transactions might actually be up to date, but we don't promise that it is. We should also point out that some applications do not mind seeing a state that is significantly out-of-date (similarly to what happens in the world wide web). These applications can exploit our mechanism more aggressively by allowing increased staleness.

Our technique is inexpensive, but it isn't totally without cost. The price to pay is that servers need to maintain old versions of the data. This price is small, however, since storage is cheap, and the size of disk storage has been steadily increasing. Furthermore, old versions needn't be retained for very long. Instead they can be discarded not long after they are superseded, e.g., when the current version has existed for fifteen minutes. The length of time to retain old versions is a system parameter and setting it depends on various trade-offs, in particular, how far in the past read-only transactions are allowed to run vs. how much storage to dedicate to old versions.

Various earlier works have proposed improving performance by running client computations in the past. For example, file systems allow the use of out of date cached information to improve performance and availability (e.g., Coda [14]), but these approaches do not provide consistency for computations running in the past. The idea of running computations in the past is exploited in versioning systems like the Elephant file system [13] or the TimeLine snapshot algorithm for persistent object stores [11]. Unlike our proposal, these systems keep old versions only for backup or time travel; furthermore, Elephant is a local file system and does not handle the consistency issues that arise in a distributed system.

In addition, the idea of running transactions in the past has been addressed in work on databases. There is a large body of work on multi-version schemes, e.g., [18, 19]. This work is primarily concerned with reducing conflicts by avoiding concurrency control for read-only transactions; most of it has not considered a distributed system, and none of it is concerned with taking advantage of information already in client caches. Other approaches provide weaker semantics. For example, the work described in [3, 16] allows read-only transactions to see an inconsistent state. The approach in [1] allows read-only transactions to make use of cached information and works in a distributed system but it does not have versions and as a result sometimes aborts read-only transactions or stalls their commits. Furthermore, al-

though it ensures that read-only transactions that commit see a consistent state, it does not serialize them: different read-only transactions can see different ordering of the read-write transactions. By contrast we provide full serializability for read-only transactions, we never abort them, and we need not delay their commits.

## 2 System Model and Requirements

We assume a persistent object store based on the client-server model. This store will serve as a storage substrate for a distributed file system we intend to build in the future.

Persistent objects are stored at servers; each object resides at a particular server, although objects can be moved from one server to another. We keep persistent objects at servers because it is important to provide continuous access to objects and this cannot be ensured when persistent storage is located at client machines (e.g., the machine's owner might turn it off). Each server may be replicated to provide highly-available access to persistent objects. There can be many servers; in a large system there might be tens of thousands of them. Furthermore, an application might need to use objects stored at many different servers.

Applications (like file systems) run at clients on cached copies of persistent objects. This architecture is desirable because it supports scalability: it reduces the load on servers by offloading work to clients, thus allowing servers to handle more clients.

Applications run as a sequence of atomic transactions. We distinguish two kinds of transactions, read-write, and read-only. Read-write transactions will require the use of a concurrency control mechanism, and communication with the servers will be needed to commit them, but processing of read-only transactions will avoid these costs. However, we still require that read-only computations behave like transactions: they must be serialized [6] with respect to all transactions, including all read-write transactions as well as other read-only transactions. For example, if a transaction T1 read  $z$  and modified  $x$ , and a later transaction T2 read  $x$  and modified  $y$ , a read-only transaction must not see a pre-T1 state of  $x$  and a post-T2 state of  $y$ . And this condition must hold even if the observed objects  $x$  and  $y$  are stored at different servers.

Our approach uses timestamps to provide proper behavior for read-only transactions. When a read-write transaction commits, it is assigned a timestamp that is consistent with the serial ordering of transactions. In other words, transactions are serialized in timestamp order, and a transaction will abort (or be retried with a different timestamp) if this is not possible.

Ensuring that read-write transactions are serialized in timestamp order requires concurrency control. We plan to use the optimistic scheme developed for the Thor persistent store because our earlier work [8] showed that it outperforms its competitors, including the best pessimistic scheme [4], in a client/server setting. But the approach could be used in

conjunction with a pessimistic scheme as well.

Concurrency control in Thor works roughly as follows (the details are given in [2]). The client machine tracks objects read and written by an application transaction and sends this information to a server at commit time. This server selects a timestamp for the transaction. Then each server where a used object resides decides whether the transaction can commit locally. Two-phase commit is required if multiple servers are involved. A simpler protocol can be used if only one server is involved; this is a common case if data is partitioned across the servers appropriately.

To decide whether a transaction can commit, a server maintains a validation queue in which it stores information about reads and writes of recently committed or prepared transactions. It uses standard rules to determine whether commit is possible, e.g., it rejects the commit if this transaction has overwritten an object that has been read by a transaction with a later timestamp.

Assuming timestamps as just described, we can ensure a read-only transaction sees a consistent state by choosing a timestamp for it and ensuring that the data it reads is valid at that time.

A key issue in this system is the choice of the transaction timestamp: this choice should be made in such a way as to avoid unnecessary aborts. In particular, the selected value must be larger than the timestamps of transactions whose modifications were observed by this transaction, or commit is not possible. We use a simple scheme to ensure the value is large enough. Clients and servers include their local time in every message they send. Furthermore a node delays processing messages, such as fetch requests or replies, until the time of its own clock is later than the time in the message. (This is a slight variation on the logical time assignment proposed in [9].)

Thus the server that receives the commit request waits until the time on its local clock is recent enough. Then it obtains the timestamp by reading its clock. This timestamp is sure to be big enough. For example, in the above scenario, the timestamp for T2 will be greater than that of T1 because the client that ran T2 learned of a timestamp at least as big as T1's when it read x; it sent a still bigger timestamp in the commit request; and therefore the server that processed T2's commit request chose a still later timestamp.

This approach of using time to order commits is cheap to implement. It can lead to delays, e.g., the server that receives a commit request may need to wait before processing it, but delays are unlikely if clocks are loosely synchronized [10]. Furthermore, we rely on clock synchronization for performance, but not correctness: Clocks can drift arbitrarily without causing the system to malfunction. If clocks get out of synch, the system may not perform well. This could happen due to a malicious attack on a server or a denial of service attack. In this paper, we are not considering an adversarial model where nodes or the network may misbehave. Handling of Byzantine failures is left as future work.

### 3 Efficient Read-Only Transactions

This section sketches our approach for improving the performance of read-only transactions while ensuring a consistent view of the data.

As explained, we assume a client-server system where clients cache data and run transactions on cached information. Caching can be very aggressive; for example, the application might pre-load the cache.

If there is a miss in the client cache, the missing object will be fetched from its server. In this paper we assume that caches store whole objects and that when there is a miss an entire object is fetched. In fact objects may be very large, consisting of many pages and an individual page may be a more sensible granularity for caching and fetching. It is easy to extend our approach to work with pages rather than whole objects.

#### 3.1 Server Processing

To allow read-only transactions to run in the past, servers must maintain old versions of objects for a certain period of time (defined in a way that leads to modest storage requirements yet allows read-only transactions to run sufficiently far in the past). Thus servers represent objects as a sequence of versions. Each version has a timestamp; this is the timestamp of the transaction whose modifications caused that version to be created. Later versions have higher timestamps, and the latest version is the "current" version of the object.

The sequence of versions contains all intermediate versions between the oldest and the latest. Each version has a *validity interval*. For old versions, this is the period between when it was created and when it was superseded by the next version; for the current version, this is the time between when it was created and the current time of the server's clock. We say that a version is *valid at time t* if  $t$  falls into its validity interval:  $v.ts \leq t$  and either  $t$  is no greater than the time at the server if  $v$  is the current version, or  $v.t < v'.ts$ , where  $v'$  is the next version after  $v$ .

Maintaining old versions of objects can be done in several ways. For example, the current version might be stored at a particular disk location and old versions moved to a log using a copy-on-write scheme. Or, new versions might be kept in a log until older ones are discarded. The discussion of the best storage management schemes for maintaining old versions is left as future work.

We expect that there will be relatively few objects with more than one version, because versions can be discarded after a while, and modifications are typically rare, so that relatively few objects will be modified in the period during which versions are retained. For example, an old version might be discarded as soon as its timestamp is more than 15 minutes behind the current time at its server. Discarding old versions is reasonable because they are only being maintained to allow read-only transactions to commit at clients, and these transactions will commit close to current time.

The server responds to two kinds of calls from clients: fetch and commit. Read-only transactions do not require a commit call. Fetches occur for both read-only and read-write transactions. Read-write transactions require the latest version, but read-only transactions do not.

A read-only fetch request contains an object id and a timestamp. This timestamp identifies the version of that object to be returned in the response, namely the one that is valid at that time. The request will succeed unless the timestamp is far in the past and the required version was discarded. If the timestamp is greater than the time at the server's clock, the server will wait to process the request, as described in Section 2.

The reply to a fetch request contains the timestamp of that version, and either the timestamp of the next version if there is one, or an indication that this is the current version.

When a client wants to commit a read-write transaction it sends the commit request to a server. As part of processing the request, that server assigns a timestamp  $t$  to the transaction; it does this by reading its local clock, and then attaches its ID as the low-order bits, in order to ensure that the timestamp is globally unique. Then that server runs the commit protocol. This can be done locally if all objects used or modified by that transaction reside at that server. Otherwise the server runs two-phase commit, allowing all servers that store objects used or modified by that transaction to participate. Each such server must ensure that the transaction is serializable with respect to commits of other transactions that used objects it stores. If the transaction can be serialized at  $t$ , the server must create new versions for all objects modified by the transaction, and tag these versions with timestamp  $t$ . In the case of a transaction that spans multiple servers (two phase commit), there can be a period in which the outcome of the transaction is in doubt (it is prepared); we ignore this issue for now, and address it in Section 3.4.

Servers also maintain *invalidation lists*. These are sets of  $\langle o, t \rangle$  pairs, where  $o$  is an object identifier and  $t$  is a timestamp. The meaning of such a pair is that a new version was created for  $o$  with timestamp  $t$ . As part of committing a transaction, the server adds a pair for every object modified by that transaction to the list.

Invalidation lists are sent to clients piggybacked on other messages that are being exchanged, e.g., fetch responses and keep alives. The invalidation information includes the current time at the server. This time serves as a validity upper bound for all current versions in the client cache. The protocol guarantees that clients learn complete information: if the client receives invalidation information as of server time  $t$ , the system guarantees that it will hear all invalidations from that server (that might affect objects in its cache) that have occurred up to that point. Therefore a client has a consistent view about when later versions are created for objects in its cache.

## 3.2 Client Processing

As mentioned, clients cache objects aggressively to reduce the need to communicate with servers for read-only transactions. Our goal is to allow the current transaction to use information in the cache, but nevertheless insure that it sees a consistent state and is running at a "recent enough" time.

The client maintains an object table that records validity information for each version in its cache. Each entry in the table records the creation time  $t_1$  for that version, and also either a special mark indicating that that version is current (as far as the client knows), or a timestamp  $t_2$  indicating when that version was superseded by a later one. In addition the client has a table mapping each server to the most recent timestamp received from it. There is an entry in this table for each server with a current version in the cache. The server timestamp serves as the validity upper bound for all current versions from that server.

To start a read-only transaction, the client selects a timestamp  $t$ . Then it discards any objects from its cache for which  $t$  is not valid, and furthermore, if  $t$  is greater than  $s.ts$  for some server  $s$ , it also discards  $s$  from its table. Thus in running the transaction, the client will be able to use all cached versions because these versions are valid at  $t$ .

There are two conflicting goals when choosing this timestamp. First, we want to minimize the number of versions that must be discarded from the cache, i.e., we would like to choose a time that is valid for all or most versions. This way we will be able to avoid fetches when running the transaction. However, in addition we need to control how far in the past the transaction runs, so that the information it observes will not be too stale. What it means for a version to be stale is application-dependent. One possibility is to allow the application to provide a parameter that controls staleness when it starts the transaction, e.g., it might indicate that  $t$  should be one minute behind the current time at the client machine. Or an application might want to run very close to the present, and in a way that ensures local causality: later transactions must observe the effects of earlier ones that committed at that client. In this case the selected timestamp must be no less than that of the previous transaction, i.e., the timestamp used for the previous read-only transaction, or the timestamp assigned to the previous read-write transaction.

Clients update the object and server tables when they receive messages from servers. Each server response contains a timestamp, which is used to update that server's information in the server table. A fetch response causes the fetched object to be placed in the client cache and the object table to be updated to record information about that version. An invalidation message is processed by changing the status of any listed objects that are in the cache as current versions to become old versions, with the creation time of the newer version used as the validity bound for the object.

### 3.3 Correctness

In this section we sketch a correctness argument for our scheme. The original concurrency control algorithm [2] we presented in Section 2 guarantees that all transactions are serialized in timestamp order (i.e. they see the effects of all transactions with lower timestamps); here we only need to prove that read-only transactions that run in the past can also be serialized in the same timestamp order. This will ensure that they are serialized both with respect to read-write transactions and also with respect to read-only transactions.

The correctness argument must establish the following: if a client commits a read-only transaction with timestamp  $t$ , all versions read by the transaction are valid at time  $t$ , and will never become invalid (with respect to that timestamp). By establishing this invariant, we can ensure that read-only transactions see the correct versions of all the data they read, i.e., a version that reflects all modification from transactions with lower timestamps.

Let's consider the client first, and assume that at the start of a read-only transaction  $T$ , the information in the cache (about versions and their validity intervals) is accurate. Then the client works correctly because all the versions in its cache are valid for selected timestamp  $t$ :

- The condition holds when the transaction starts because the client discards all versions that aren't valid at  $t$ .
- The condition holds after a fetch request (assuming the server is correct) because the new version, which is added to the client cache, is valid at  $t$ .
- The condition holds after the client processes an invalidation because invalidations never cause the validity interval of any version in the cache to decrease. Only current versions are affected by an invalidation. If a current version is still considered to be current, all that happens is that its validity interval has been extended, because the new time for that object's server is greater than what was known before. If the version is no longer considered to be current, its interval nevertheless is larger than what it used to be, because the creation time for the next version of that object is larger than the last timestamp received from that server (we discuss how the servers enforce this condition next).

Therefore when the transaction is ready to commit, it can only have read versions that are valid at  $t$ .

Now we consider processing at the server. Assuming that the server serializes transactions correctly, all that is required is proper bookkeeping: the server must maintain versions with their creation times and it must add information to the invalidation list each time a read-write transaction commits. In addition it must respond correctly to fetch requests, so that the client receives versions with their validity intervals, and it must send correct invalidation information to clients.

However, there is one additional constraint on the server that may not be immediately apparent. The correctness argument for the client assumed that when the client received an invalidation, this never caused it to reduce the validity inter-

val of any version. The server must ensure that this assumption holds. In particular, it cannot create a new version for some object  $x$  if the transaction doing the modification has a timestamp less than the assumed validity interval for  $x$  in some client cache. It is trivial for a server to ensure this condition for transactions that do not require two-phase commit (because they commit at just one server). In this case, the server selects a timestamp for the transaction that is greater than any timestamp it has sent in messages to clients, and as a result the creation time for a new version of object  $x$  will be greater than the client-assumed validity interval for the previous version of  $x$ .

There is an issue with two-phase commit, however. In this case the timestamp is selected by the coordinator, and by the time the prepare request shows up at a participant, this timestamp might be older than what the server has sent in a message. One obvious way to respond is for the participant to reject the prepare, but this is undesirable. The alternative is to weaken the client-side interpretation of the validity interval it assumes for current versions. Rather than assuming these versions are good right up to the server timestamp, instead the client can assume they are good up to that time minus some delta. This weaker assumption allows the server to accept prepares with slightly old timestamps. In particular if delta is chosen to be equal to the maximum clock skew plus the maximum expected message delay when the network is working properly, the probability of a participant needing to abort a transaction because its timestamp is too small will be negligible. And furthermore the impact on when read-only transactions can run will also be small, since this only constrains the choice of the timestamp for a read-only transaction by a small amount.

The correctness argument above shows that read-only transactions are serialized with respect to read-write transactions. We also require that they are serialized with respect to one another. For example, it must not be possible for one read-only transaction to see that read-write transaction T1 has occurred but T2 has not while another one observes the opposite ordering. Such a situation cannot occur in our system, since timestamps determine the commit order. One of T1 and T2 has the smaller timestamp; suppose it is T1. Then it isn't possible for any transaction to observe a state where T2's modifications have occurred but T1's have not.

### 3.4 Prepares

The algorithm just presented sends only one timestamp in the fetch request. However, it may be desirable to augment this by sending a choice of possible timestamps. Then the server can select a version that is valid at one of these choices. This gives it more flexibility, which can help to avoid stalls.

Stalls arise primarily because of two-phase commits. In this case there is a prepare phase followed by a commit phase. In the prepare phase, a server must get ready to commit and this includes creating tentative versions for all modified objects. These version tentatively supersede the current version.

If the transaction commits, they will become the new current versions, while if it aborts they will disappear.

If a server receives a fetch request for an object with a tentative version, and the timestamp it receives is valid for the tentative version, it cannot reply: it can't send the older version since if the transaction commits, that version won't be valid for that time, and it can't send the newer version since it will disappear if the transaction aborts. Therefore the server must stall. But if instead the client sends a set of timestamps, the server can select a committed version that is valid at one of these choices. This gives it more flexibility, which can help it to avoid stalls.

We can provide the extra flexibility by extending the fetch request to contain a timestamp interval  $[t_1, t_2]$ . Such an interval indicates that every timestamp between  $t_1$  and  $t_2$  is acceptable to the client. The server can then select the most recent committed version that is valid for a time in this interval. The client might choose the interval to be the intersection of all validity intervals of all the versions in its cache, or possibly of all versions used so far by that transaction.

The probability of a stall is related to the size of the interval provided in the fetch request. Providing an interval does not prevent stalls, but may make them less likely.

Note also that with the single timestamp scheme, choosing a timestamp a bit behind makes stalls less likely (since prepares usually resolve quickly). For example, if the timestamp were one minute behind the current time, the probability of a stall due to a prepare is very low. The advantage of the interval scheme is that it allow a very recent timestamp to be selected while avoiding stalls.

## 4 Conclusion

Transactions have been recognized as an important mechanism to provide simple and correct handling of concurrency and failures. However, transactions are often considered too expensive to be used in systems like file systems that require good performance.

In this paper, we argued that performance is not a barrier to running transactions. We did this by presenting a simple mechanism that substantially lowers the cost of running read-only transactions (which constitute the bulk of the operations in a file system).

The price to pay for using our new technique is small. There is an additional storage cost at the servers, but storage is cheap, disk space is increasing quickly, only a few versions are needed, and versions can be stored in a way that does not slow down commits of read-write transactions. Our technique makes read-only transactions run slightly in the past, but we argue that this is not problematic because applications still see a consistent view of the data and because slightly out of date data is usually perfectly acceptable.

In the future we intend to demonstrate our point in practice by building a file system that uses a persistent object store with the semantics provided by our technique.

## References

- [1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Mar. 1999. Also available as Technical Report MIT/LCS/TR-786.
- [2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, California, May 1995.
- [3] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, USA, Mar. 2003.
- [4] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, MN, June 1994.
- [5] J. N. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.
- [6] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.
- [7] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in QuickSilver. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, Austin, TX, Nov. 1987.
- [8] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, June 1999.
- [9] J. Lundelius. *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1988.
- [10] D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Network Working Report RFC 1305, Mar. 1992.
- [11] C.-H. Moh and B. Liskov. Timeline: A high performance archive for a distributed object store. In *First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.
- [12] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–217, San Diego, CA, USA, Jan. 1993.
- [13] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, Dec. 1999.
- [14] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. In *IEEE Computer*, May 1990.
- [15] M. Seltzer and M. Stonebraker. Transaction support in read optimized and write optimized file systems. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, 1990.
- [16] K. Shen, T. Yang, L. Chu, J. Holliday, D. Kuschner, and H. Zhu. Neptune: Scalable replica management and programming support for cluster-based network services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, Mar. 2001.
- [17] W. Vogels. File system usage in windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, Dec. 1999.
- [18] W. E. Weihl. Distributed version management for read-only actions. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, Minaki, Canada, Aug. 1985. ACM.
- [19] K.-L. Wu, P. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *Proc. of the 9th International Conference on Data Engineering*, 1993.