Practical Uses of Synchronized Clocks in Distributed Systems

Barbara Liskov
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Synchronized clocks are interesting because they can be used to improve performance of a distributed system by reducing communication. Since they have only recently become a reality in distributed systems, their use in distributed algorithms has received relatively little attention. This paper discusses a number of distributed algorithms that make use of synchronized clocks and analyzes how clocks are used in these algorithms.

1. Introduction

Synchronized clocks are quickly becoming a reality in distributed systems. For example, the network time protocol NTP [14] synchronizes clocks of nodes on geographically distributed networks. It does this at low cost and provides clocks that are synchronized to within a few milliseconds of one another. NTP is running on the internet today and is used to synchronize clocks of nodes throughout the United States, Canada, and various places in Europe.

Synchronized clocks are interesting because they can be used to improve the performance of distributed algorithms. They make it possible to replace communication with local computation. Instead of node N asking another node M whether some property holds, it can deduce the answer based on some information about M from the past together with the current time on N's clock.

Since the practical availability of synchronized clocks is a recent phenomenon, their use in distributed algorithms has not received much attention. This paper describes the role of synchronized clocks in several distributed algorithms. The focus is on practical algorithms that either are in use in systems today or that will be used in the near future. The algorithms differ in their synchronization requirements; some require clocks synchronized to within a few minutes of one another, while others require closer synchronization. All of them have much less stringent requirements on synchronization than current clock algorithms provide.

There is a considerable literature on clock synchronization algorithms [21]. It is not the goal of this paper to explain how clock synchronization works; instead the paper assumes the clocks exist and discusses how to use them. The ability of NTP to synchronize clocks in the internet with small clock skews and low cost is taken as evidence that relying on synchronized clocks in distributed algorithms is a reasonable thing to do, both in local area networks and geographically distributed networks.

Clock synchronization algorithms are based on probabilistic assumptions about clock rate and message delay. Therefore, clocks are only synchronized with some (very high) probability. Since clock synchronization can fail occasionally, it is most desirable for algorithms to depend on synchronization for performance but not for correctness. Depending on synchronization for performance is reasonable; since clocks will be synchronized most of the time, performance will only degrade rarely. Some of the algorithms to be discussed depend on synchronization only for performance, but others depend on it for correctness. Depending on synchronization for correctness is more problematic but it is sometimes appropriate. In practical systems, performance is very important. Furthermore, the correctness of an algorithm may depend on the non-occurrence of other low-probability events, so that having it also depend on synchronized clocks has little impact. Also there may be recovery mechanisms at a higher level to compensate for failures of the algorithm.

The remainder of the paper is organized as follows. It begins in Section 2 with a few remarks about synchronized clocks. Then it describes several distributed algorithms that use synchronized clocks. It concludes with a discussion of how to incorporate synchronized clocks into new algorithms.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-89-J-1988, and in part by the National Science Foundation under grant CCR-8822158.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-439-2/91/0007/0001 $1.50

2. Synchronized Clocks

Clock synchronization algorithms synchronize clocks with some skew ϵ: They guarantee that if c1 and c2 are the clocks at two nodes of a network, then at any instant the time at c1 differs from the time at c2 by no more than ϵ. As mentioned, the synchronization property cannot be provided absolutely, but only with some very high probability.

It is worth noting that practical clock synchronization algorithms must provide efficient engineering solutions to a number of problems. Some of these are technical problems, e.g., how to avoid being misled about the time when a message containing a time value is delayed in the network. The algorithms that exist today are robust in the face of problems such as network congestion and links with widely varying delays. They are less likely to be robust, however, in the face of operator and software errors. Most systems allow a manual override to set the clock but such an override clearly allows the clock to be set incorrectly, i.e., in a way that violates the clock skew assumption. Clock synchronization algorithms need to be thought of as part of a total system, and care must be taken to limit the damage caused by operator error. Furthermore, the algorithm must accomplish its task without consuming much of the bandwidth of the network and without requiring that every node be equipped with expensive devices.

A property closely related to synchronized clocks is synchronized clock rates. Several of the algorithms to be described depend only on clock rates being synchronized rather than on clocks being synchronized. This means that the clocks at the different nodes run at approximately the same rates although the times on these clocks may be different. The interest in clock rates dates from the time when clock synchronization was thought to be practically unattainable. Clock rates were assumed to be "naturally" synchronized so that no algorithm was required to keep them synchronized; in fact, to ensure that rates are truly synchronized requires algorithms similar to clock synchronization algorithms. The focus on algorithms that depend only on synchronized rates will likely diminish once synchronized clocks are available. Nevertheless, it is interesting to understand what enables an algorithm to depend on rates instead of time; this question is discussed in Section 8.

Clock synchronization algorithms typically synchronize clocks with "real" time, i.e., at any moment a node's clock differs from real time by no more than $\epsilon/2$. At the root of such algorithms is a dependence on devices that sample universal time; such devices are attached to time servers, and the algorithm spreads the information about the current time from the servers to other nodes in the network. Having clocks close to real time is obviously important; for example, the "time last modified" for files ought to be close to the time the modification actually occurred, or users may notice strange behavior. Also, the presence of an algorithm that synchronizes nodes' clocks to real time obviates the need for operators to set the clock manually.

None of the algorithms to be discussed depends on the clock times being close to real time. They do depend on clock rates being close to real clock rates, however. This is because each algorithm makes use of time intervals that have been chosen based on assumptions about how users of the system behave, e.g., what kinds of delays a user is willing to tolerate. If internal clock rates aren't close to real clock rates these assumptions will not be honored.

3. At-most-once Messages

The first example of the use of synchronized clocks is the SCMP protocol [12], which guarantees at-most-once delivery of messages. Many networks do not guarantee at-most-once delivery; instead they may duplicate messages and furthermore duplicates may arrive very late. In addition, since networks may lose messages, higher levels of a system re-send them, which may also lead to duplicates.

Implementing at-most-once semantics is typically done by having each message receiver maintain a table containing information about "active" senders that have communicated with the receiver recently. When a message arrives, if there is information about the sender in the table it is used to determine whether or not the message is a duplicate. If there is no information, there are two choices: either accept the message or reject it. If the message is accepted, there is a chance of accepting a duplicate. This chance can be made arbitrarily small by keeping information about senders long enough, but it is difficult to determine how long to keep this information in the presence of sender retransmission and networks with probabilistic delay.

The alternative of rejecting the message guarantees that no duplicates will ever be accepted. However, it gives rise to a problem. When a message is sent, we want to be reasonably certain that the receiver will accept it. Therefore we need to know that the receiver has information about the sender in its table. If it is unlikely to have such information, e.g., because this is the first time the sender has communicated with it in a while, then it is necessary to set up the information before sending the message. This can be done by means of a handshake in which a pair of messages is exchanged between the sender and receiver in advance of the at-most-once message. If the sender then sends many messages over the connection established by the handshake, the cost of the handshake is amortized across all of them. If there are only a few messages, the overhead is high relative to useful work. In the worst case, the sender transmits only one message per handshake. Yet this case may be quite common; it corresponds to a client that performs a single operation at each of many servers.

The SCMP protocol avoids the handshake between the sender and receiver by using synchronized clocks. The idea is that the receiver remembers all "recent" communications. If a message from a particular sender is "recent," the receiver will be able to compare it with the stored information and decide accurately whether the message is a duplicate. If the message from the sender is "old," it will be tagged as a duplicate even though it may not be, but this case is very unlikely. Thus the system will never accept a duplicate but it may occasionally reject a non-duplicate.

For the scheme to work, receivers need to know whether a message is "recent." When a node sends a message, it timestamps the message with the current time of its clock. When the message arrives at the receiver, it is considered recent if its timestamp is

later than the receiver's local time minus the *message lifetime interval* ρ; otherwise it is old. The message lifetime interval must be big enough (e.g., ten minutes) so that almost all messages will arrive within ρ time units of when they were sent; it is much larger than ε. The characteristics of ρ are discussed further in [12].

The protocol works as follows. Every module G has a *current time, G.time*; this is read from the clock belonging to its node. Every message m contains a timestamp, *m.ts*; this is G.time of the sending module at the time m is created. Even though a particular message may be duplicated either by the network or by the software that carries out a higher-level protocol, all copies of the message contain the same m.ts. Each message also contains a *connection identifier, m.conn*; this is selected by the sender without consultation with the receiver, unlike other protocols, e.g., TCP [19]. The connection id must be distinct from the ids used in other senders, e.g., at other nodes; in addition, if the sender has several outstanding messages to the same receiver, it should use a separate connection id for each.[1] Every message sent to a particular receiver and containing a particular connection id should contain a distinct timestamp; thus the connection id and the timestamp together constitute a unique message id with respect to that receiver.

Each receiver maintains a *connection table, G.CT,* that maps connection ids to connection information including the timestamp of the last message accepted on that connection. Not all connections have an entry in G.CT. G is free to remove an entry for connection C from its connection table provided $G.CT[C].ts \leq G.time - \rho - \varepsilon$; such an entry is considered to be "old." (Recall that ρ is the message lifetime interval.) A receiver also maintains an upper bound, *G.upper*, on the timestamps that have been removed from the table. Since only old timestamps are removed from the table, $G.upper \leq G.time - \rho - \varepsilon$.

The algorithm works by determining a per-connection bound that distinguishes "new" from "old," or potentially duplicate, messages, and comparing the timestamp of the newly arrived message with that bound. If the message's connection has an entry in the table G.CT, the bound is the timestamp of the most recent previously accepted message. If there is no table entry, the global bound G.upper is used. G.upper is an appropriate bound because if there is no information for the connection in G.CT, this means the last message on the connection (if any) contained a timestamp $t \leq G.upper$. Therefore, if a message arrives whose timestamp is later than this, it must be a new message. Since $G.upper \leq G.time - \rho - \varepsilon$ there is little chance of incorrectly flagging a message as a duplicate, provided ρ is large enough. Messages with timestamps less than the bound are discarded; if a message is accepted, its timestamp is stored in the G.CT entry for its connection.

Receivers that survive crashes need a way to determine whether a message that arrives after crash recovery is a duplicate of a message that arrived before the crash. SMTP uses time to solve this problem also. It maintains on stable storage [8] a timestamp *G.latest* that is larger than the timestamps of all messages accepted so far. A message that arrives too early (i.e., its timestamp is greater than G.latest) is discarded or delayed. After a crash, G.upper is initialized to G.latest. This will cause all potential duplicates to be rejected because only messages with timestamps less than this new G.upper could have been accepted before the crash. G.latest is maintained by writing $G.time + \beta$ to stable storage periodically; G.latest is the most recent value written to stable storage. β is some increment (e.g., a few seconds) that is large enough so that stable storage isn't written often, but small enough so that not many messages must be rejected after a crash. For many persistent servers, G.latest can simply be written to stable storage as part of the records that are being written there anyway to record information about the server's persistent state.

Synchronized clocks allow the protocol to establish a system-wide notion of "recent." Clocks are used to avoid communication (to establish a connection) and to save storage at receivers (only timestamps of recent messages need be saved). Timestamps identify messages that have already arrived. The identification is only approximate, since a single timestamp G.upper stands for all earlier messages, and therefore sometimes a message that is not a duplicate will be rejected.

If clocks get out of synch, there is no danger of a duplicate message being accepted, but recent messages may be flagged as duplicates. If a node's clock is slow, its messages are more likely to be flagged as duplicates by other modules; if its clock is fast, it is more likely to flag messages from other modules as duplicates. The algorithm does depend on the values stored for G.latest being monotonic, but this is easy to guarantee in software each time a new value for G.latest is written to stable storage.

4. Authentication Tickets in Kerberos

The next example is taken from the Kerberos system [22]. Kerberos provides a means for modules to communicate using secure, authenticated connections. It makes use of private keys using the DES encryption technique [15], and is based on the Needham and Schroeder authentication protocol [16]. Systems that use Kerberos make use of authenticated connections between every client-server pair. Kerberos uses synchronized clocks in two ways: to limit the use of particular keys, and to help servers detect replayed messages.

Every communication between a particular client C and server S is controlled by a *ticket*. The client obtains a ticket by applying to the ticket granting service, TGS. If the client is authorized to use S, the TGS gives it a ticket $T_{C,S}$ for S and also a secret "session" key, $K_{C,S}$, that can be used during future communication between C and S. The message from the TGS to the client containing $T_{C,S}$ and $K_{C,S}$ is encrypted using C's private key; this ensures both that the message really comes from the TGS (since only the TGS and C know C's private key) and also that the key cannot be stolen by an intruder that intercepts the message. When the client receives this message, it can decrypt it to obtain the key and the ticket.

C sends the ticket to S every time it communicates with S. The ticket $T_{C,S}$ identifies C and S and also contains the session key

[1] Thus in a system supporting lightweight threads within processes a connection id might be a triple <node-id, process-id, thread-id>, where the node-id is a unique name of the sender's node, the process-id identifies the process within the node, and the thread-id identifies the thread within the process.

$K_{C,S}$. The ticket is encrypted using S's private key (again to prevent forgery and theft); S can decrypt the ticket to obtain the key. Thus both (and only) C and S know the session key and therefore they can use it to exchange private information.

Since the ticket is what allows C to talk to S, its use must be controlled. For example, suppose C runs at a public workstation and the ticket was obtained on behalf of some person who was using that workstation. If that person leaves the workstation without logging out, someone else could obtain access to his or her tickets. Therefore, each ticket also contains an *expiration time*, E; a server S will only accept communications from C using ticket $T_{C,S}$ provided the expiration time it contains is less than the server's clock - ε.

This use of synchronized clocks saves a communication between S and TGS. Without synchronized clocks, S could ask the TGS whether a ticket were still valid and the TGS could determine validity by comparing the expiration time of the ticket with its local clock. In doing the test at S, we rely on the time at S's clock being a close approximation to the time at the TGS's clock.

The correctness condition for the expiration time is: S must not use the ticket after it expires at the TGS. Therefore, correctness will fail if S's clock is slow (or the TGS's clock is fast). But, S's clock must be very slow for this to be a problem. The lifetime of a ticket is typically much larger than ε. If S's clock is just a little slow, it doesn't matter at all. Furthermore, the problems arising from using a ticket a little too long are less than those arising from other threats, such as tickets being stolen at unattended workstations.

One point that is interesting about this use of synchronized clocks is the following. A system like Kerberos is designed to permit secure communication in the face of various threats such as users trying to steal keys or trick the system into telling them about keys. To base a system like Kerberos on synchronized clocks requires that the clock synchronization algorithm be secure against similar threats. For example, if it were possible for a malicious user to "spoof" the synchronization algorithm in a way that causes S's clock to become very slow, tickets could be used long after they should have expired.

The second use of clocks in Kerberos, to avoid replays, makes use of *authenticators*. An authenticator is basically just a timestamp that has been encrypted. The timestamp is produced by the client reading its clock; the client then encrypts it by using the session key $K_{C,S}$. Since $K_{C,S}$ is known only to C and S, it is not possible (with very high probability) for an intruder to create a valid authenticator on its own; instead all an intruder can do is to re-use an authenticator. Messages containing old authenticators are discarded. If desired, a server can in addition retain the timestamps of all recent messages and discard any new messages that contain these timestamps.

The use of authenticators is similar to what occurs in the at-most-once message protocol. However, at-most-once delivery is not the point of the protocol. For example, if the client does not receive a reply to a message, it is free to send it again with a different authenticator.

If clocks get out of synch, no harm will occur if the server is maintaining a list of current messages; otherwise, if the server's clock is slow, it might accept a replay of an earlier message. As was the case with the at-most-once protocol, clocks are used to reduce storage at servers (only timestamps of current messages need be saved) and avoid communication (checking with a client about the status of a newly received message).

5. Cache Consistency

The next example concerns systems in which servers provide persistent storage for objects and programs that use those objects run at client workstations. To provide reasonable response time to clients, copies of persistent objects are cached at the workstations so that clients can use them locally when there is a cache hit.

As is the case in any system with cached copies, we need to be concerned with how to maintain cache consistency. One possibility is to use "leases" as discussed below. This idea first appeared in [5]; it is also used in the Echo system [13]. In either case the concept is used in a file system, so the objects in question are files. In the initial use of leases, the caches were write-through; in Echo, the caches are write-behind. This difference in cache behavior leads to a difference in how leases are used. Below I explain how these systems work given synchronized clocks. The systems in fact only require synchronized clock rates as discussed in Section 8.

In the case of the write-through cache, leases work as follows. Each client workstation obtains a lease for a file when the file is copied into its cache. The lease contains an expiration time E; when E has been reached, i.e., when E > time(client) - ε, the client stops using the file. The client can request that a lease can be renewed by asking the server for a new expiration time.

When a client modifies a file, the modification goes directly to the server (since this is a write-through cache). The server can do the modification immediately if there are no other outstanding leases on the file. Otherwise it communicates with the clients holding the leases, requesting them to give them up. The modification is done when all leases have been relinquished.

Of course, it is possible that a current holder of a lease might not respond, either because of a network problem, or because of a crash of its node. In this case, the system will wait until the expiration time of the lease, and then do the modification.

The idea of leases requires only a small extension to work with write-behind caches. Now there are two kinds of leases, read leases and write leases, and a client must use the file in accordance with its lease. Thus a client with a read lease can only read the file, while a client with a write lease can both read and write the file. There are the usual rules concerning readers and writers: Many clients can simultaneously have read leases for a file, but if a client has a write lease for some file, no other clients can have read or write leases for that file.

Each lease has an expiration time as discussed above. The only difference is that competition for leases now occurs when a client requests a lease (rather than when a file is written). If a client needs a lease that conflicts with leases held by other clients, the

server sends messages to the other clients requesting them to relinquish their leases. For example, if some client needs a write lease, the system will notify all holders of read leases to remove that file from their caches. The new lease is granted when all old leases are either relinquished or expired.

The invariant of interest in a system with leases is: each time a client uses a file, it has a valid lease for that file. Validity is determined by using the client's clock as an approximation of the time of the server's clock. If the client's clock is slow, or the server's clock is fast, the invariant will not hold. In this case, the client may continue to use the file after its lease has expired at the server.

In the absence of the use of synchronized clocks there are two possibilities for maintaining cache consistency, neither of which is desirable. One alternative is for the client to check the validity of each file use. This alternative is not much better than not having caches at all. In particular, to read a file in its cache, a client would have to communicate with the server to determine if the cached copy is valid; if it is valid, the server need not send back a copy, so this part of the communication is saved, but in any case there would be long delays. This is effectively a system in which all leases have a lifetime of zero.

The other alternative is for the server to not invalidate a client's lease until it hears from the client. This is roughly what happens in cache consistency protocols in multi-processors. Such protocols are based on the assumption that nodes and communication never fail (or that all fail together); these assumptions ensure that the wait to release the lease will be very short. Such an assumption is less attractive in a distributed environment; here nodes and the network can fail independently, so that the wait can be long. In fact, requiring the server to wait for the client to give up leases is equivalent to having a lease with an infinite lifetime.

Thus we can see that the designer of such a system is presented with two unattractive choices: either depend on assumptions such as synchronized clocks that might fail causing inconsistencies, or sacrifice performance. Choosing to improve performance is a valid position, given the low probability of clocks getting out of synch.

6. Atomicity

Although the decision to use leases is justifiable, still it would be nice if there were a higher level mechanism to take care of cache consistency problems due to unsynchronized clocks. Transactions are such a mechanism. For example, in the Thor object-oriented database [9, 10], all accesses to objects occur within atomic transactions. Objects in Thor are not just files; instead they belong to various types, including user-defined types. Objects are stored at servers, which provide persistent storage for them. Clients run at workstations, and caching is used to reduce delay to the clients.

Optimistic concurrency control [7] is used to provide serialization of transactions. Each object has a version number that is copied into a client cache along with the object. As a transaction runs, it uses the objects cached at its workstation without communicating with the servers. When the transaction commits, the version numbers of all the objects it used are sent to the servers along with the new versions of all objects it modified. The servers compare these version numbers with those stored with the objects; if the numbers do not match, the transaction must abort.

To reduce the likelihood that transactions will abort due to stale data, servers notify clients when objects in their caches are modified by a committing transaction. Leases can be used to limit server responsibility for notifying clients about stale information and to delay commits of transactions that might cause transactions running at other clients to abort. Since Thor is a write-through system (writes are delayed until transactions commit, but really happen at this point), there is just one kind of lease. When a transaction that modified object x attempts to commit, the server can check with all other clients holding unexpired leases on x, asking them to give up their leases, and allow the transaction to commit only if all clients relinquish their leases, or when all leases expire.

The invariant in this system is: each time a client uses an object, it holds a valid lease for that object. If clocks get out of synch, the invariant might not be preserved. However, in Thor the worst that will happen is that some transaction may have to abort. No damage will have been done to the consistency of the persistent objects. Thus, the higher level mechanism (transactions) provides the safety that was missing in the lower level mechanism (leases).

7. Commit Windows

The final example concerns the use of synchronized clocks within a replication algorithm. The algorithm to be described is used in the Harp file system [11]; a similar technique is used in Echo [13].

Harp is a replicated Unix file system that provides highly reliable and highly available storage for files. It supports the virtual file system (VFS) [6] interface and is intended to be used within a file service in a distributed network, such as NFS [20, 23]. The idea is that clients continue to use the file service just as they always did. However, the server code of the file service calls Harp and achieves higher reliability and availability as a result. Harp runs each file operation as an independent atomic operation; as is usual in file systems, there is no mechanism to run transactions consisting of sequences of operations.

Harp uses the primary copy replication technique [1, 17, 18]. In a primary copy method, client requests are sent to just one server called the *primary*; the other servers are *backups*. The primary decides what to do and records any new information at a *sub-majority* of backups before committing the operation. A sub-majority is one less than a majority, e.g., if there are five servers, the operation would be recorded at two backups before it commits. Since the primary also knows about the operation, this means a committed operation is known to a majority of replicas.

The primary and backups always run within a *view*; a view is simply the group of replicas that are currently cooperating to provide service. When there is a failure, or a recovery from a failure, the system reconfigures itself by performing a *view change* [2, 3], which leads to the creation of a new view. A view always contains a majority of servers; this ensures that the new view intersects with the old one in at least one replica, which in turn can be used to ensure that the new view starts in a state that

reflects all committed operations from earlier views. The primary of the new view may be a different node than the primary of the old view.

In Harp, each replica maintains a *log* in which it records information about client modification operations.[2] To carry out a modification operation, the primary creates an event record that describes the modification, appends it to the log, and sends the logged information to the backups. As new log entries arrive in messages from the primary, a backup appends them to its log and sends an acknowledgement message to the primary. When acknowledgments have arrived from a sub-majority of backups, the primary commits the operation and responds to the client.

Read operations (e.g., to determine file status) could be handled similarly to modification operations by making entries in the log and waiting for the ack from the backup, but this seems unnecessary because read operations don't change anything. Therefore read operations are performed entirely at the primary. This can lead to a problem if the network partitions. For example, suppose a partition separates the primary from the backups, and the backups form a new view with a new primary. If the old primary processes a read operation at this point, the result returned might not reflect a write operation that has already committed in the new view. Such a situation does not compromise the state of the file system, but it can lead to a loss of external consistency [4]. (A violation of external consistency occurs when the ordering of operations inside a system does not agree with the order a user expects.)

Synchronized clocks can be used to reduce the probability of having a violation of external consistency. Essentially the primary holds leases, but the object in question is the entire replica group. Each message sent by a backup to the primary gives the primary a lease. The primary can do a read operation unilaterally if it holds unexpired leases from a sub-majority of backups. When a new view starts, its new primary cannot reply to any client requests until all leases given to the old primary by replicas in the new view have expired. Leases are short, e.g., to create the expiration time in a lease a backup might add one second to its current time; therefore a new view is unlikely to be delayed since by the time the view change has finished, the old leases will all have expired.

The invariant in this system is: whenever a primary performs a read it holds valid leases from a sub-majority of backups. This invariant will not be preserved if clocks get out of synch. However, the impact of violating the invariant is small. At worse there will be a violation of external consistency, but in fact this is unlikely. For the violation to happen, there must be two clients C1 and C2, with C1 doing a read at the old primary, and C2 doing a write at the new one. Here is the scenario:

1. C2 performs a write at the new primary.
2. C2 informs C1 (e.g., by performing a remote procedure call to C1) about the update.
3. C1 reads the modified file (at the old primary) and does not see the update.

It is unlikely that clocks would become unsynchronized enough for this to happen.

A similar technique could be used to avoid two-phase commit for read-only transactions. The technique works as follows. Suppose a transaction starts at some node and visits other nodes by making remote procedure calls (RPCs). Each node that it visits contains objects that the transaction can read or modify. The reply to an RPC contains an indication of whether the transaction read or modified objects during that call and a time during which the node promises to not release any read locks. When the transaction attempts to commit, if it is read-only (all RPC replies indicated that it only read objects), and if the time at the coordinator is less than the promised release time (minus ε) for all read locks, the coordinator can commit the transaction without communicating with any other nodes. Note that as in the commit window scheme, a node must not provide service when it recovers from a failure until it is certain that all promises have expired.

If clocks get out of synch, this algorithm may result in non-serializable behavior. For example, if one of the participants releases a transaction T1's read locks early because its clock is fast, this may allow some other transaction T2 to modify objects at P after T1 read them, and to modify objects at some other node Q before T1 reads them. As was the case in the Harp file system, a situation in which this occurs is highly unlikely. Furthermore, out-of-synch clocks affect the serial order only of read-only transactions; all transactions that modify objects would be serialized properly with respect to one another. Although it is probably not a good idea to adopt such transactions as the only choice, they might be a useful option. Certain applications might be willing to settle for a slight danger of seeing an inconsistent state in a read-only transaction to gain the improved performance that would result.

8. Synchronized Rates

Several of the algorithms discussed above can be implemented using synchronized clock rates rather than synchronized clocks. This section discusses how rates are used instead of clocks, and when rates are adequate.

Suppose we want to use the cache-consistency lease mechanism, but based on synchronized rates instead of synchronized clocks. Then instead of the message from the server containing an expiration time, it would contain a *time to expiration*, i.e., an interval such as "five seconds." A client always receives a lease in response to some message it sent; it simply adds the expiration interval of the lease to the time of its clock when it sent the request for the lease, obtaining a local expiration time T_C. The server does the same thing, but its local expiration time T_S depends on the time of its clock when it sent the response. Provided the clock rate differences are bounded by some skew, and that this skew is used at the client to determine when the leases expire, we can be sure that the lease will expire at the client no later than it expires at the server. This is true because the response at the server must have occurred after the request was sent by the client.

[2]In Harp, the log is kept in volatile memory, which is backed-up by an uninterruptible power supply. The power supply allows the server enough time to copy the log to disk in the event of a power failure.

The use of leases illustrates the kind of situation in which synchronized clock rates can be used instead of synchronized clocks. In all these algorithms, there is some *event of interest*, e, such as the expiration of a lease at the server or the expiration of a ticket at the TGS. Call the node where this event happens the *owner*, O, of the event. The event e occurs at time T_e; here T_e is the real time at which e occurs rather than a time of some node's clock. Some other node (or nodes) depends on event e and must make a conservative judgment about when it occurs. Call this other node the *dependent node*, D. The dependent node makes the approximation by means of an event d of its own; d happens at absolute time T_d, and we require that $T_d \leq T_e$. For example, the expiration of a lease at the client is such an event d and it must happen no later than the expiration of that lease at the server.

The event of interest e is always preceded by some other event f that leads to it and that also occurs at e's owner. For example, the TGS granting a ticket is such an event f. Note that the dependent node will find out about f via a message that arrives from the owner (possibly sent via intermediate nodes); for example, this is how the client finds out about the granting of the lease. Rates can be used instead of clocks if there is some still earlier event g that happens at the dependent node D and that leads to f. For example, a lease is granted at the server because a client requested it. The situation is illustrated in Figure 8-1.

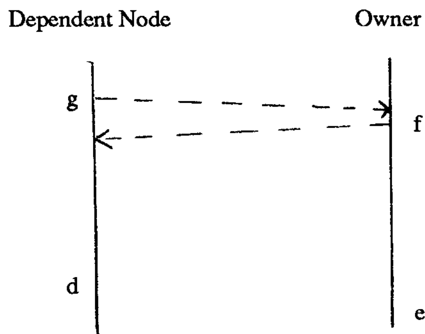

Figure 8-1: Using rates. Time increases going down.

Rates work because this communication pattern enables D to make the necessary conservative judgment. The event d will happen at time $T_g + \lambda - \varepsilon$. Here T_g is the absolute time at which event g occurred, λ is the expiration interval, and ε is the appropriate bound on the skew based on how rates can vary over an interval λ. Furthermore, e will happen at time $T_f + \lambda$. Thus we have:

$$T_d = T_g + \lambda - \varepsilon$$
$$T_e = T_f + \lambda$$

Since $T_g \leq T_f$ and the skew in the rates of the clocks at D and O during the interval λ is bounded by ε, we know that $T_d \leq T_e$

Thus rates can be used when there is a communication already happening that allows the dependent node to estimate approximately when the event of interest happens. Clocks are needed when there is no such communication; in fact, clocks permit the communication to be avoided. For example, in the at-most-once protocol the owner O is the sender of the message; the receiver is the dependent node D. Since the message is sent autonomously by the sender, without a prior communication from the receiver, there is no way to use rates. In the case of Kerberos tickets, the request of a client for a ticket might appear to be event g, but the client cannot be trusted to abide by any rules, so that the server is the one that enforces the expiration time. Thus the server is actually the dependent node D. If the TGS communicated with the server before granting the ticket, it would be possible to use rates: The server's response in this communication would be the event g that leads to event f (the TGS's granting of the ticket). However, this communication does not in fact take place, which speeds up the ticket-granting process.

The focus on algorithms that depend only on rates is likely to diminish now that synchronized clocks exist. Note that synchronized clocks are more powerful than synchronized rates; they support all algorithms that depend on rates, and some other algorithms besides.

9. Discussion

Earlier sections of this paper have looked at how synchronized clocks are used in a number of distributed algorithms. In each case, clocks were used to provide improved performance by avoiding communication. In some algorithms, communication could also have been avoided by retaining state (e.g., in the use of authenticators in Kerberos, or in the at-most-once message protocol); for these, the use of clocks can also be thought of as a way of using garbage collection of "old" information to reduce storage requirements.

The algorithms differ in the consequences of clocks getting out of synch. There are the following possibilities:

1. No effect on correctness. This is the case with the at-most-once message protocol and also with the authenticators in Kerberos provided the server keeps track of the timestamps of all recent messages.

2. Compensation. Even when clock synchronization gives rise to errors, there may be some other part of the system that compensates. The use of atomic transactions is an example of how a problem at one level of a system may be resolved at a higher level.

3. Domination by other failures. In some systems, failures that arise because clocks are out of synch are dominated by other possible failures. This is what happens with Kerberos tickets. Tickets might be used too long if servers' clocks are slow. However, using a ticket that was supposed to last for several hours for an extra few minutes is not a serious matter, especially compared to other difficulties such as stolen tickets.

4. Trade-off for performance. Finally, it is reasonable to choose a mechanism that works improperly when clocks fail when the alternatives are unacceptable. For example, the alternatives to leases are either high overhead on each read, or very long periods in which certain files cannot be used.

Although none of the algorithms depends on clocks approximating "real" time, all require that clock rates approximate real time passing. For example, a Kerberos ticket is supposed to have a lifetime that approximates a real time interval; a ticket that is intended to last for one hour should last for about that long. The algorithms rely on real clock rates because they all depend on lifetime intervals that are chosen based on expectations about user requirements. Thus, the lifetime of a ticket is chosen based on an analysis of the likelihood that it will be stolen within that time and the seriousness of the consequences of such an event.

To convert a distributed algorithm to one that uses synchronized clocks, there are two places to look. By examining the messages that are being exchanged, it may be possible to identify some that could be avoided by using timestamps. Or, in the case where message exchange is already reduced by maintaining state, it may be possible to find a way to save storage by using timestamps as a garbage collection technique. After finding a place to use timestamps, the next step is to analyze the consequences of using synchronized clocks, both on normal behavior (when clocks are in synch) and during clock failures. During this analysis the time interval that will be used is selected (all algorithms have such an interval, e.g., the message lifetime interval ρ in the at-most-once protocol). An algorithm based on timestamps is a good idea if ultimately the worst case behavior is sufficiently unlikely or sufficiently benign so as to represent a good tradeoff for improved performance.

Acknowledgments

The author gratefully acknowledges the help given by readers of earlier drafts of this paper, including Dorothy Curtis, Robert Gruber, John Guttag, Paul Johnson, Andrew Myers, Liuba Shrira, Raymie Stata, Greg Troxel, and John Wroclawski.

10. References

1. Alsberg, P., and Day, J. A Principle for Resilient Sharing of Distributed Resources. Proc. of the 2nd International Conference on Software Engineering, October, 1976, pp. 627-644. Also available in unpublished form as CAC Document number 202 Center for Advanced Computation University of Illinois, Urbana-Champaign, Illinois 61801 by Alsberg, Benford, Day, and Grapa..

2. El-Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the Fifth Symposium on Principles of Database Systems, ACM, 1986, pp. 240-251.

3. El-Abbadi, A., Skeen, D., and Cristian, F. An Efficient Fault-tolerant Protocol for Replicated Data Management. Proc. of the Fourth Symposium on Principles of Database Systems, ACM, 1985, pp. 215-229.

4. Gifford, D.K. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Corporation, March, 1983.

5. Gray, C., and Cheriton, D. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. Proc. of the Twelfth ACM Symposium on Operating Systems Principles, 1989, pp. 202-210.

6. Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. USENIX summer '86 Conference Proceedings, 1986, pp. 238-247.

7. Kung, H. T., and Robinson, J. T. "On Optimistic Methods for Concurrency Control". *ACM Trans. on Database Systems 6*, 2 (June 1981), 213-226.

8. Lampson, B. W., and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. Xerox Research Center, Palo Alto, Ca., 1979.

9. Liskov, B., et al. Preliminary Design of the Thor Object-Oriented Database System. In preparation.

10. Liskov, B., Gruber, R., Johnson, P., and Shrira, L. A Highly Available Object Repository for use in a Heterogeneous Distributed System. Proc. of the Fourth International Workshop on Persistent Object Systems Design, Implementation and Use, Martha's Vineyard, MA, September, 1990.

11. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. Replication in the Harp File System. Submitted for publication.

12. Liskov, B., Shrira, L., and Wroclawski, J. Efficient At-Most-Once Messages Based on Synchronized Clocks. To appear in ACM Trans. on Computers.

13. Mann, T., Hisgen, A., and Swart, G. An Algorithm for Data Replication. Report 46, DEC Systems Research Center, Palo Alto, CA, June, 1989.

14. Mills, D.L. Network Time Protocol (Version 1) Specification and Implementation. DARPA-Internet Report RFC-1059. July 1988.

15. National Bureau of Standards. Data Encryption Standard. Government Printing Office, Washington, DC, 1977.

16. Needham, R., and Schroeder, M. "Using Encryption for Authentication in Large Networks of Computers". *Comm. of the ACM 21*, 12 (December 1978), 993-999.

17. Oki, B. M., and Liskov, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. Proc. of the 7th ACM Symposium on Principles of Distributed Computing, ACM, August, 1988.

18. Oki, B. M. Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT/LCS/TR-423, M.I.T. Laboratory for Computer Science, Cambridge, MA, August, 1988.

19. Postel, J. DoD Standard Transmition Control Protocol. DARPA-Internet RFC-793. September, 1981.

20. Sandberg, R., et al. Design and Implementation of the Sun Network Filesystem. Proc. of the Summer 1985 USENIX Conference, June, 1985, pp. 119-130.

21. Simons, B., Welch, J., and Lynch, N. An Overview of Clock Synchronization. Research Report RJ 6505, IBM Almaden Research Center, 1988.

22. Steiner, J.G., Neuman, C., Schiller, J.I. Kerberos: An Authentication Service for Open Network Systems. Project Athena, MIT, Cambridge, MA, March, 1988.

23. Sun Microsystems, Inc. NFS: Network File System Protocol Specification. Tech. Rept. RFC 1094, Network Information Center, SRI International, March, 1989.