A Decentralized High Performance Time Service Architecture

Danny Dolev* Rüdiger Reischuk†
Ray Strong‡ Ed Wimmers§

November 1995

Abstract

A time service is a means by which users, applications, etc. obtain values associated with some measurement of elapsed time from some agreed on event. In a relativistic world, time services are necessarily imperfect, there being no such thing as simultaneity. In this paper we discuss an architecture for delivering required time services to a collection of heterogeneous machines communicating over a wide area network consisting of multiply connected and overlapping local area networks with various communication media. The primary building block of our architecture is a *logical clock*. We explain how logical clock objects simplify the treatment and resolution of a number of problems associated with time services. Having used logical clock objects to decouple clock synchronization problems from problems associated with smoothness and conformity to political time, we provide a straightforward master-slave algorithm for clock synchronization and show how to transform it into an eavesdropping algorithm that takes advantage of the broadcast nature of most local area networks to dramatically reduce message overhead. Finally, we show how to integrate eavesdropping and regular clock synchronization algorithms into a fault tolerant network time service that tolerates and coordinates multiple master clocks or external sources of time.

*Institute of Computer Science, Hebrew University, Jerusalem, Israel This research was supported in part by GIF I-207-199.6/91.

†Institut für Theoretische Informatik, Med. Universität zu Lübeck, Wallstraše 40, 23560 Lübeck, Germany. This research was supported in part by GIF I-207-199.6/91.

‡IBM Research Division, Almaden Research Center, 650 Harry Rd., San Jose, CA 95120-6099

§IBM Research Division, Almaden Research Center, 650 Harry Rd., San Jose, CA 95120-6099

1 Introduction

A time service is a means by which users, applications, etc. obtain values associated with some measurement of elapsed time from some agreed on event. In a relativistic world, time services are necessarily imperfect, there being no such thing as simultaneity. However, in spite of relativity and various sources of physical uncertainty, many distributed applications depend on time services. In addition to systems that perform real time control, there are a number of applications involving scheduling, trading, or authentication, that have demanding requirements for time services. In this paper we discuss an architecture for delivering required time services to a collection of heterogeneous machines communicating over a wide area network consisting of multiply connected and overlapping local area networks with various communication media.

In the interest of best reflecting what is feasible on such a heterogeneous network, our design makes minimal assumptions about the hardware components of which it is composed. In the next section we collect those assumptions in the form of a network model and elaborate on our view of the general time service problem to be solved within that model. Next we describe the object that is the primary building block of our architecture, the *logical clock*. We explain how logical clock objects simplify the treatment and resolution of a number of problems associated with time services. Having used logical clock objects to decouple clock synchronization problems from problems associated with smoothness and conformity to political time, we provide a straightforward master-slave algorithm for clock synchronization and show how to transform it into an eavesdropping algorithm that takes advantage of the broadcast nature of most local area networks to dramatically reduce message overhead. Finally, we show how to integrate eavesdropping and regular clock synchronization algorithms into a fault tolerant network time service that tolerates and coordinates multiple master clocks or external sources of time.

2 Overview and Model

A time service is more than a method for synchronizing time-of-day clocks. In fact, following our design, the physical time-of-day clock is never actually adjusted or set. And yet our time service must provide a value for the time-of-day at each machine in such a way that applications that require some closeness of time-of-day values in a distributed system can be satisfied. Moreover, it must provide a value that is a measure of that closeness with associated guarantees that cannot be refuted by experiment. For example, an application might ask our time service for a time stamp and precision estimate and receive in response a political time value 08:45:33.425 on 9 October 1995 plus or minus 0.012 seconds. If the application received these values and then sent a message asking a peer at another machine on the network for similar values and, having received a response, obtained another local value set of 08:45:33.431 on 9 October 1995 plus or minus 0.012 seconds, then the remote value set should be consistent with the implied guarantee (the date and hour should be the same unless the corresponding time zones use different political times; but, if the remote "plus or minus" value was 0.013 seconds, then the minute and second values should be identical, and the remote millisecond value should be between 400 and 456).

We will call the two values that a time service must make available *time* and *precision*. The first value represents elapsed time in some agreed on *epoch* and the second value represents a guaranteed bound on observable discrepancies in the first value. Both values are measured in units of time. Whether the application interface calls for time to be delivered as two endpoints of an interval, a time value plus or minus a precision value, or a time value together with a yes or no answer to whether the precision is within an input value is irrelevant for the purposes of this paper. In any case, our time service must maintain both time and precision as coordinated values in a distributed system. If there is at least one source of universal time external to the system, then the precision value should also bound observable discrepancies with universal time.

A good time service delivers small values of precision and uses small amounts of system resources. The most important system resource for our purposes is network communication overhead. There is a potentially feasible solution to the time service problem in which all communication takes place outside the network because each machine in the network is equipped with its own independent means of receiving signals from a universal time source; but this solution has drawbacks including the inconvenience of retrofitting the receiving hardware to existing machines and the dependence on an uninterrupted time source (a single point of catastrophic failure).

For the rest of the paper we will assume that some number but not all of the machines of our network have access to a universal time source. We will also assume

- that each machine in the network has a physical clock that can be read by machine instruction;
- that these physical clocks have a known worst case observable *drift* rate against each other (measured in parts per million);
- that the physical clocks return monotone values and are never set forward or back for any reason while the machine is running; and
- that the time values presented by the physical clocks can represent time differences much smaller than the smallest precision that can be delivered by any time service.

3 Logical Clock Objects

A logical clock is an intermediary among applications that require timestamps, clock synchronization protocols, and real time or physical clocks. A logical clock obviates the need to tamper with, reset, or adjust the physical clock, allows clock synchronization protocols to use instantaneous positive or negative adjustments to time, and can provide a smooth time that is never set back and only adjusted at a user specified maximum rate of deviation from the time rate of the underlying physical clock.

The methods described in this section for accessing a logical clock include operations of acquiring a timestamp, adjusting the time by a specified amount at a specified rate of amortization (so that timestamps correspond to a piecewise linear monotone increasing function of the physical clock), changing the offset from the reading of the physical clock, setting a multiplier to change the rate at which time passes for the logical clock from that of the physical clock, and initializing the parameters of the logical clock, which include the original offset and rate as well as the rate of amortization to be applied to time adjustments.

Some attention is given to the problem of driving logical clocks from signals or interrupts that might otherwise interfere with an atomic reading of the logical clock.

The physical clock of a machine is used to time real time processes. If any real time processes might ever be running on a machine, it is a good idea not to set, change, or modify the physical clock. However, there are many reasons for modifying the time shown by a clock: (1) to give it an initial setting, (2) to adjust it to more accurately reflect the passage of some standard or real time, (3) to arrange for political time changes (daylight savings, leap seconds, etc.) [Mil]. Logical clock objects resolve the apparent conflict.

3.1 Purpose

The primary purpose of a logical clock is to be a modifiable function of the physical clock so that the physical clock need not be modified directly. Thus, when one wants the current political time, one should access a logical clock designed to return it rather than accessing the physical clock directly. In fact it would be good practice to access the physical clock only via a logical clock. Then the physical clock would never be set, changed or adjusted. When adjustment of time value is required, the adjustment can be made to a logical clock.

A secondary purpose of a logical clock is to provide certain smoothness properties to time values in spite of abrupt corrections both forward and backward that are called for by some clock synchronization protocol. In many cases it is desirable to provide a sequence of time values that is monotone nondecreasing and varies from the values corresponding to some standard or real time by at most a bounded rate [L, HMM]. However, for many clock synchronization protocols, it is more convenient to make corrections immediately and set clocks either forward or back by significant amounts. By using one logical clock directly in the synchronization protocol and reading time from another logical clock that amortizes the adjustments made by the first in order to smooth the changes and prevent time running backward, one can have the best of both worlds. This decoupling of clock synchronization from considerations of smoothness can simplify the consideration of smoothness of time given in many papers devoted to clock synchronization (v. [Cri, CS, L, HMM]).

Finally, logical clocks can be used to simulate timed events at any desired rate. If software for a timed process is to be simulated and the software gets its time from a logical clock. Then another logical clock running at a different rate can be substituted almost immediately with no other change to the program.

3.2 Definition

For the purposes of this paper, a *logical clock* is an object with a method providing for the return of a function of the time values produced by the physical clock. In a *smooth* logical clock, this function is piecewise linear, continuous, and monotone nondecreasing. Note that the undisturbed physical clock provides a monotone nondecreasing step function of real time. Recall that we assume that the time values provided by the physical clock are monotone nondecreasing. We will also assume that *time* is a data type inheriting the properties of real numbers. Typically time is of type *floating* or *double*. Here we treat it as an abstract type. Following are the program methods for managing a logical clock.

- **Time:** This function takes no argument and returns the current logical time value.
- **Adjust:** This function takes one argument of type time. Its effect is to add the (positive or negative) argument permanently to the function of physical time that is returned as **Time**, but to amortize this addition so that it takes place gradually over an amortization duration in such a way that a prespecified bound on the rate of deviation from physical time is respected.
- **Set:** This function takes one argument of type time. Its effect is to immediately set the current value of logical time to this argument, without amortization. The remainder of any unfinished amortization of an adjustment is ignored and the logical clock continues from this position at its current rate.
- **Change_offset:** This function takes one argument of type time. Its effect is to immediately increment the current offset by the argument. Any current amortization continues uninterrupted.
- **Change_rate:** This function takes one argument of type time. Its effect is to immediately multiply the current basic rate (first derivative of the logical clock with respect to physical time when no amortization is taking place) by the argument. However a corresponding Change_offset is applied so that the current reading of Time is unchanged. If there is no current amortization then the effect is exactly as if the logical clock function **Time** were multiplied by the argument, again with the corresponding Change_offset so that current Time remains fixed. If there is a current amortization, the portion remaining is treated as a new adjustment to be made to the logical clock after its rate has been multiplied by the argument.
- **LClock_init:** This function takes three arguments: *offset*, of type time; *amortization_rate*, of type time ≥ 0; and *rate*, of type time ≥ 0. Its effect is to create a logical clock with base rate *rate*, amortization taking place at rate *amortization_rate*, and with initial offset from physical time *offset*. Note that a clock

cannot be guaranteed monotone nondecreasing unless its *amortization_rate* α satisfies $0 < \alpha \leq 1$. If the *amortization_rate* is set to 0, then the method *Adjust* is treated as equivalent to the method *Change_offset*.

In a typical implementation of these functions, **LClock_init** might return a pointer to an allocated logical clock data structure and each of the other functions would then require such a pointer as an extra argument. There would also be one additional function that would destroy such a data structure and free its storage. We will suppress these and similar details in the interest of simplicity.

Change_offset might be used to accomodate leap seconds or daylight savings time changes: the change would be made to logical clocks intended to return political time; the change would not interfere with logical clocks being used in clock synchronization protocols because it would not be applied there (cf. [Mil]). It could also be used as the way to adjust logical clocks involved in clock synchronization where no amortization is needed. **Change_rate** might be used for logical clocks involved in clock synchronization to take advantage of known drift characteristics. Such an application is beyond the scope of this paper.

3.3 Implementation Specification

In this paper we omit a complete implementation. Rather we specify that the "state" inherent in a logical clock should be able to represent a piecewise linear function of physical clock time with at most one point of nonlinearity: the end of the current amortization period. In order to compute the current **Time**, one need only test whether one is in or after the current amortization interval, multiply by the appropriate rate and add the appropriate offset. Thus **Time** should be obtainable by (1) reading the physical clock and storing its value, (2) testing for amortization, (3) multiplying, and (4) adding, in the type *time*.

The exercise of each of the methods listed above should result in changing such a state to another such state. When one amortization is still underway and another is invoked, the implementation must resolve the changes left to be amortized to produce one new amortization interval at one new rate of amortization.

3.4 Atomicity of Time Reading

We assume that a logical clock may be read concurrently by many threads of computation; but it has only one modifier. Under this assumption it is easy to arrange wait-free execution of **Time**. One standard method is to keep an anchor pointer alternating between two copies of the "state" of the logical clock. The only code that can change this pointer is the modifier code. A **Time** reading consists of following the anchor pointer to the current state and then following steps (1) through (4) above. The modifier makes changes in the non-current copy of the "state" and then switches the pointer.

If more than one modifier is to be accomodated, then the modifications must be enqueued and driven to the logical clock by the one modifier with the power to switch the pointer. The typical delay involved in such an enqueueing should present an insignificant loss of precision (on the order of the product of the drift rate with the delay).

3.5 Time Service

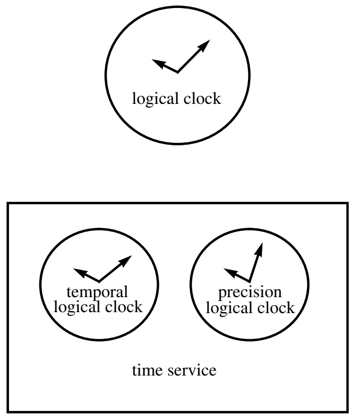

Figure 1: A Time Service consists of two Logical Clocks

A *time service* is a pair of logical clocks: one, called the *temporal* logical clock, is used to report the time; the other, called the *precision* logical clock, is used to report an upper bound on how far the reported time can be from some standard.

3.6 Synchronization

A typical clock synchronization protocol would maintain two time services: one for synchronization and one for political time. It would use a logical clock with no amortization for clock synchronization, a logical clock with rate set to d, the worst case rate of drift for the physical clock, to estimate drift since the last synchronization, a logical clock with an amortization rate of .01 to present political time to users, and a logical clock with rate set to d and amortization rate set to $\frac{.01}{d}$ to present a precision estimate to political time users. The clock synchronization process would drive all the logical clocks: when clock synchronization called for a change of x time units in logical time and a change of y time units in precision, the method *Change_offset* would be applied to each of the synchronization logical clocks with arguments x and y respectively, while the method *Adjust* would be applied to each of the political time logical clocks with the same arguments. Each would obtain physical clock time from the same physical clock, using the same code. No part of the clock synchronization protocol (or any other protocol) would ever adjust the physical clock. Alternatively, a process that depends on a very smooth time could use a temporal logical clock with an amortization rate of .001 driven by an external clock synchronization protocol coupled to it through the *Adjust* interface. If the amortization rate for a temporal logical clock is α, then its corresponding precision logical clock would have an amortization rate of $\frac{\alpha}{d}$.

4 Time Service Synchronization

4.1 Round Trip Synchronization

In this section we discuss well known methods for synchronizing one time service (called the *slave*) to another time service (called the *master*) by means of a pair of messages forming a *round trip* between slave and master. Such methods are discussed in [Cri, DRS, HMM, Mar, PR].

In spite of relativity, it is possible to temporally order some events that occur at distinct locations [L]. We assume that the causal relationship between message transmission and receipt orders transmission before receipt. Thus if the slave temporal clock reads A before transmission of a first message, if the master temporal clock reads B after receipt of the first message but before transmission of a second message, and if the slave temporal clock reads C after receipt of the second message, then the clock reading events are temporally ordered in the order A, B, C. Suppose at the time the master temporal clock read B it had a precision clock reading of V. Then the slave temporal clock could be adjusted by $B - \frac{A+C}{2}$ and, after the adjustment, the slave could associate a precision of $V + \frac{C-A}{2}$ with the time B. These changes would mean that, when the slave temporal clock reads $E > B$, its precision clock should read $V + \frac{C-A}{2} + d(E - B)$ where d is the known upper bound on the rate of drift.

This round trip synchronization is summarized in figure 2.

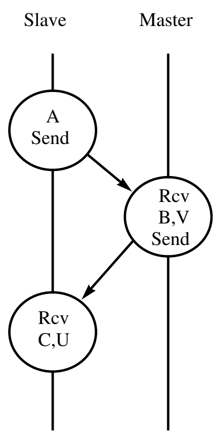

Figure 2: Round Trip Synchronization

Abstracting from this discussion of round trip synchronization, we use the term *synchronization experiment* for any protocol that results in a set of time values A, B, C, U, V such that A and C are readings of a logical slave clock with U the precision estimate corresponding to C, B is a reading of a logical master clock with corresponding precision estimate V, and the readings B, V can be temporally ordered between the reading A and the readings C, U. A synchronization experiment produces *tentative changes* for each time service it drives:

- **temporal:** $X = B - \frac{C+A}{2}$,
- **precision:** $Y = V - U + \frac{C-A}{2}$.

When a slave time service is initialized by a synchronization experiment, these changes are applied without testing. However, once the time service has been initialized, the changes should be applied only if the following predicates are true:

- **Master Sanity Check:** $|X| < U$,
- **Improvement Check:** $Y < 0$.

The slave time service is a synchronization time service so the changes should be applied to the two logical clocks by means of the *Change_offset* method. For any other dependent time services, the changes should be applied by means of the *Adjust* method.

4.2 Delayed Round Trip Synchronization

It often happens that the time between receipt of a synchronization request and its response it significant. To keep this time from counting against the round trip time, we introduce the following optimization.

Assume that after receipt of a synchronization request, a time D is obtained from the master synchronization logical clock. Then after some delay, a second pair of time values B, V is obtained from the master time service (as in round trip synchronization), and all values are sent in the master response. The values A, C, U are obtained at the slave as before. Then the optimized changes are as follows:

- **temporal:** $X = \frac{D+B-C-A}{2}$,
- **precision:** $Y = V - U + \frac{D-B+C-A}{2} + \frac{d(B-D)}{2}$.

4.3 Slave Synchronization Protocol

In figure 3 we present a general slave synchronization protocol built around synchronization experiments. The branch from **Sleep** to **Eavesdrop** represents an extension that will be discussed in section 5. In figure 3, the slave is initialized by constructing a time service and supplying a temporary precision requirement. The test **Need Synch** checks that the current precision clock reading is $\geq$ the precision requirement. The step **Set Timeout** sets a timer to expire when the precision clock reading will be first

$\geq$ the precision requirement. The step **Raise Req** increases the precision requirement by a small amount. The test **Large Improvement** checks whether the new precision clock reading will be smaller than the precision requirement by an amount much larger than the small amount used in **Raise Req**. Thus hysteresis is provided by **Lower Req**, which decreases the precision requirement by the amount used in **Large Improvement**.

If the step **Synch Exp** consists of a number of synchronization experiments iterated until the precision requirement is satisfied, then this protocol is a variant of Probabilistic Clock Synchronization [Cri]. If the changes X, Y are obtained by intersecting the implicit time intervals of master and slave using

$$X = \frac{min(B+V+\frac{C-A}{2}, C+U) + max(B-V-\frac{C-A}{2}, C-U)}{2} - \frac{C+A}{2}$$

and

$$Y = \frac{min(B+V+\frac{C-A}{2}, C+U) - max(B-V-\frac{C-A}{2}, C-U)}{2} - U,$$

then the protocol is a variant of the interval intersection protocol of [Mar].

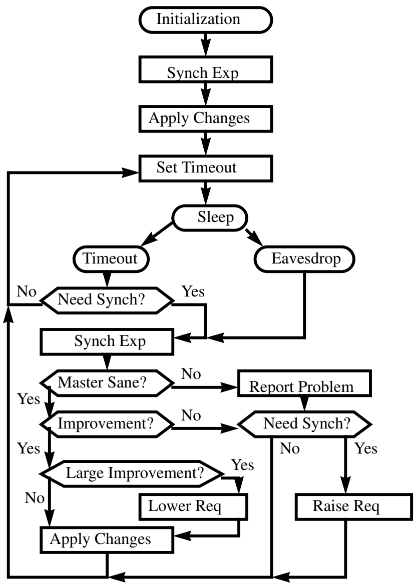

Figure 3: Slave Synchronization Protocol

5 Eavesdropping Clock Synchronization

The exercise by one slave of the protocol presented in figure 3 leads to a reasonably message efficient synchronization. If a synchronization experiment produces an expected new precision π, then the protocol will maintain an expected precision bound $\beta = \pi + \gamma$ where γ is a constant depending on the amounts by which *Req*, the required precision, is reduced in the step **Lower Req** and raised in the step **Raise Req**. The expected time between experiments will be $\frac{\beta-\pi}{d}$, provided there is only one slave. However, when many slaves independently synchronize to the same master, the message efficiency is reduced at least linearly with the number of slaves (queueing effects can make it much worse). Thus a protocol that makes insignificant band width demands for one slave, can make unsatisfiable band width demands for the hundreds of slaves that could share a local area network with a master.

In order to overcome this difficulty and produce a protocol that scales well, we designed eavesdropping synchronization experiments in which some slaves only listen to messages that travel between the master and other slaves. For example, if we require that each *active* synchronization experiment consist of two round trips in a broadcast medium (from which we take the better values), then other *eavesdropping* slaves can use values extracted from these messages to form equivalent synchronization experiments without sending messages. The diagram in figure 4 describes an eavesdropping synchronization experiment. We use **Bcst** as opposed to the **Send** of figure 2 to indicate that the message is sent via a broadcast medium and is available for any receiver. The step **Buf Empty** indicates that the eavesdropper determines there are no further broadcast messages waiting for it at this time (after it has obtained timestamp **A**). If the eavesdropper does not receive all the indicated messages, then the experiment fails with **No Improvement**.

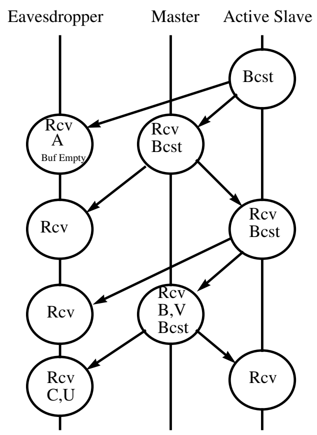

Figure 4: Eavesdropping Synchronization

If eavesdropping is added to the synchronization protocol as indicated in figure 3, the design works but generally makes no improvement. In order to improve message efficiency with eavesdropping, one must trade a little precision. Thus the test **Large Improvement?** and the step **Lower Req** must be modified. They are changed to maintain a certain ratio between the precision that triggers a timeout and that obtainable by active round trip synchronization. In addition to the values produced by any synchronization experiment, a Boolean value **Active** is produced. This value is true if the synchronization experiment was an active round trip; false, if it was the result of eavesdropping. The logic is modified so that the required precision is kept above a multiple of r times the new and improved precision produced by an active round trip. The value of r depends on the network; but values between 2 and 3 are typical. When the appropriate value of r is maintained, we can expect the message overhead to be between that required by 2 and 3 slaves, almost independent of the number of slaves participating (for numbers of slaves under 100).

6 Failure Tolerance

In our experiments with these protocols, we have noticed occasional message loss or very long delay; so we always put a one second timeout on any synchronization experiment. If the experiment has not completed within the timeout, it returns with **Master Sane** but **No Improvement**.

The more interesting problem is that of tolerating the occasional failure of a master. The most convenient solution to this problem is for each slave to subscribe to a *membership service* that sends a notification when the master fails. The response to such a notification is to promote a new master with access to the same source information as the old, if such is available; otherwise, we promote a new master from among the slaves to serve as an internal master until a better source is recovered. An *internal master* acts as source for a set of slaves.

If no membership service is available, then we have to find another method for failure detection and coordination of potential masters. In the next section we describe such a method, with the added benefit that it coordinates multiple sources in the network.

7 Coordinating Multiple Masters

This section presents a software method for decentralizing a master-slave clock synchronization protocol so that it allows clocks in a general network to be synchronized. Our method is directed toward networks of overlapping broadcast media (ethernets, token rings, etc.); but it applies as well to point-to-point networks where each (two-way) link is treated as an individual broadcast medium. We also show how to achieve stability of the network overhead required for clock synchronization in spite of clocks being at different logical distances from multiple external sources (the *logical distance* is the number of LAN hops necessary to get a message from one node to another).

Our general plan calls for slaves with the most precise synchronization to an external source to act as local masters for other slaves. There will be times when other slaves take over temporarily because of changing network conditions. If no external source is available, then eventually the slave with the unique dominant id in the network will synchronize the entire network, until an external source can be reestablished. The novelty of our decentralization scheme consists of five elements:

1. the explicit use of broadcast communication for slave communication of synchronization requests,

2. the introduction of a random wait to fill an unfilled request,

3. the introduction of a factor of weighted distance from source that is applied to the threshold that indicates a need for synchronization,

4. the introduction of new thresholds used to indicate that the external source has been lost, and

5. the idea of becoming an explicitly internal source when the external source has been lost.

We will assume that all processors in the network are capable of performing as either masters or slaves (if this is not the case, the modifications will be obvious). We will also assume that some processors are attached to identified external sources of time: these processors will be said to have *distance* 1 from their attached external sources. They will also be assumed to have an upper bound on the precision of their sources, which will be treated as Precision in the master data structure below.

Each potential master will maintain a data structure with the following fields:

time	precision	source	distance

.

This data structure will be sent with every synchronization message that the process sends as a master. The time field will be the reading of a synchronization temporal logical clock; the precision field will be the corresponding reading of the synchronization precision logical clock. The result of every completed synchronization experiment will be a pair of change values $< X, Y >$, a link value L indicating the communication link (LAN) over which the synchronization experiment took place, and a quadruple $< T, P, S, D >$ corresponding to the data structure from the master.

Each potential master also maintains the following variables and functions:

Req	a parameter for computing when synchronization is needed
W	a long timeout value
Z	a parameter for computing when synchronization has been lost
$SourceLink$	link over which last synchronized
UnfRqst(L)	true if unfilled synch request on link L
Master(L)	true if current master on link L

In figure 5 and figure 6 we present a simplified version of the protocol followed by a potential master. This is an expansion and generalization of figure 3. We have suppressed the details of *Req* manipulation from figure 3.

Our decentralized protocol consists of the prescribed responses for four events. The protocol can be tuned by adjusting parameters W and Z in addition to the tuning discussed in connection with figure 3. The four events are as follows:

1. **PT Timeout:** a time to check whether Precision has crossed some threshold,

2. **UR Timeout:** a time to check whether a Synchronization Request has been satified,

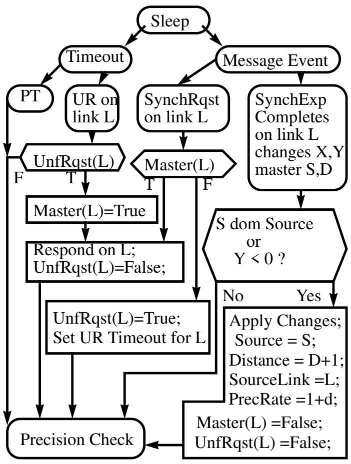

Figure 5: Master/Slave Synchronization Protocol Part I

3. **SynchRqst on** L**:** the receipt of a message from another protocol participant requesting synchronization via link L,

4. **SynchExp on** L **with changes** X, Y **and master** S, D**:** the receipt of a message that completes a synchronization experiment on link L and produces changes X, Y from a master data structure indicating source S and distance D.

After some processing for each event control is passed to the precision checking part of the protocol represented by step **Precision Check**, which is expanded in figure 6. Eventually, control returns to the step **Sleep** to await the next of the four events.

After a **PT Timeout** control passes immediately to **Precision Check**.

After a **UR Timeout** we check $UnfRqst(L)$. If there is an unfilled synchronization request, then the local slave becomes master for the link and sends a master response. Note that we optimize here so that the time spent waiting for the timeout does not count against the round trip time. Details of the optimization appear in section 4.2. Then $UnfRqst(L)$ is set to $False$ and control passes to **Precision Check**.

After **SynchRqst on L** we check $Master(L)$. If it is true then we proceed to respond as above. Otherwise, we set $UnfRqst(L)$ to $True$ and set a UR Timeout for a time $Z \times Distance + R$, where R is a small random time. Note that this duration should be much less than W. Then control passes to **Precision Check**.

After the completion of a synchronization experiment on link L that produces potential changes X, Y and involves a master with source S and distance D, we check for improvement as in figure 3; but this time, in addition to checking whether $Y < 0$, we have to check whether S dominates the current $Source$. This test is indicated by **S dom Source** in figure 5. Here the relation *dominates* must totally order all host ids. All ids must dominate "NULL" and any external source must dominate all network host ids. This allows external sources to take over sychronization of a network

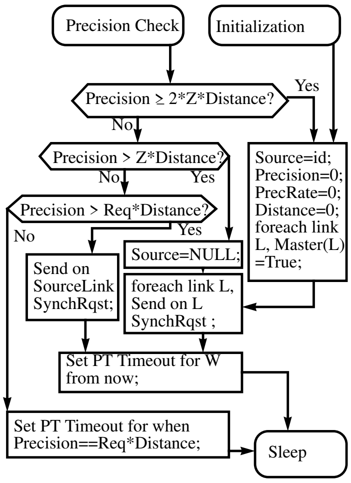

Figure 6: Master/Slave Synchronization Protocol Part II

when links to them are established. When there are no available external sources, then the protocol is designed to synchronize all slaves in the network to the master with the unique dominant id. After appropriate changes are applied and appropriate management of *Req* has taken place (not shown), control passes to **Precision Check**.

At **Precision Check**, Precision is compared against a number of thresholds to decide whether synchronization is needed, whether synchronization to the current source has been lost, and whether the local slave should try to take over as master of the network. Note that each of the thresholds has *Distance* as a multiplicative factor. This stabilizes the protocol. Without this factor, slaves at larger distances tend to increase the rate at which they make synchronization requests so that communication overhead increases with time. When a slave has installed itself as master, it periodically tries to find a better source by sending synchronization requests on its links.

8 Conclusion

We have presented a design for general network time services. Elsewhere we have reported on preliminary experiments involving logical clocks and clock synchronization protocols [DSW]. Logical clock objects and time services based on round trip synchronization and eavesdropping have been implemented in C++. Operating from user space, our eavesdropping clock synchronization scheme easily maintains thirty millisecond precision on a local area network with insignificant communication overhead. We have not yet implemented the general scheme on a wide area network; but we believe it provides a simple solution to failure and change problems in time service networks.

References

[Cri] F. Cristian, Probabilistic Clock Synchronization, *Distributed Computing* 3, 146-158, 1989.

[DRS] D. Dolev, R. Reischuk, and R. Strong, Observable Clock Synchronization, *Proc. 13th Ann ACM Symp. on Principles of Distributed Computing*, 284-293, 1994.

[DSW] D. Dolev, R. Strong, and E. Wimmers, Experience with RAPID Prototypes, *Proc. 5th Int Workshop on Rapid Systems Prototyping*, 62-72, 1994.

[HMM] J. Halpern, N. Megiddo, and A. Munshi, Optimal Precision in the Presence of Uncertainty, *J. Complexity* 1, 170-196, 1985.

[L] L Lamport, Time, Clocks and the Ordering of Events in Distributed Systems, *C. ACM* 21, 558-565, 1978.

[Mar] K. Marzullo, *Maintaining time in a distributed system: An example of a loosely coupled distributed service*, Ph.D. dissertation, Stanford University, 1984.

[Mil] D. L. Mills, Measured performance of the Network Time Protocol in the Internet system, *ACM Computer Communication Review* 20:1, 65-75, 1990.

[PR] B. Patt-Shamir and S. Rajsbaum, A Theory of Clock Synchronization, *Proc. 26th ACM Symp. on Theory of Computation*, 1994.

[CS] F. Cristian and F. Schmuck, Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm, *Proc. 9th Ann. ACM Symp. on Principles of Distributed Computing*, 133-143, 1990.