Scalable Byzantine-Fault-Quantifying Clock Synchronization

John Douceur

Jon Howell

October 15, 2003

Technical Report

MSR-TR-2003-67

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Scalable Byzantine-Fault-Quantifying Clock Synchronization

John Douceur and Jon Howell

Microsoft Research

October 15, 2003

1 Abstract

We present a scalable protocol for establishing bounds on clock synchronization in the presence of Byzantine faults. The worst a faulty participant can do to a correct host is cause the correct host to establish (arbitrarily) weak but correct bounds; because the correct hosts knows what those bounds are, we refer to the protocol as *Byzantine-fault quantifying*. Correct hosts can use the quantified bounds to inform path selection, enabling them to route around misbehaving hosts.

We describe how to employ the protocol in a practical environment that makes use of Byzantine-fault tolerant replicated state machines.

2 Introduction

This paper describes a scalable clock-synchronization protocol that withstands malicious failures. We developed this protocol in the context of Farsite, a distributed serverless filesystem. Farsite uses replication and encryption to produce reliable, private storage from resources supplied by incompletely trusted hosts. The hosts are desktop workstations, on the scale of 100,000 machines. By “incompletely trusted,” we mean that we expect only a small fraction of the population of machines to be Byzantine-faulty.

Farsite uses simple replication to ensure the availability of file data. To ensure the availability *and consistency* of directory metadata, upon which the consistency of the entire filesystem service depends, Farsite uses Byzantine-fault-tolerant replicated state machines. We call the set of hosts replicating a BFT state machine a “server group,” because it acts like a virtual trustworthy server.

A server group enforces consistency by doling out leases to client machines. A lease is a lock that expires: the lock is a promise that the host holding the lease can access the data it protects consistently; the expiration provides robustness against failure [4]. If a lease-holder vanishes (because of network failure, software crash or malicious failure) and fails to return the lease, the lease eventually expires so that it can be given out to another host.

Achieving consistency in the presence of expiration requires that the lessor and lessee agree on the expiration time, at least conservatively. The lessee cannot believe it holds the lease once the lessor believes the lease has expired. Clock synchronization is necessary to accomplish this agreement.

Note that when we say “synchronize clocks,” we don’t adjust the value of a (scalar) system clock, which then becomes input to future rounds of the algorithm, as does NTP [8]. Instead, we compute and maintain bounds (a pair) on the relationship between a local system clock and a global reference clock. The bounds let lessees behave conservatively: if a lease is set to expire at 3:00 PM, and based on its local clock the client can bound the global clock within (2:57,3:02), then the client must assume that the lease might have expired.

The purpose of the protocol presented in this paper is to enable each participant in Farsite to compute bounds relating its local clock to some global reference clock. In Section 3, we describe the protocol, treating the global reference clock and each participant as a separate “host.” In Section 4, we discuss how we implement a trustworthy reference clock in Farsite, and in

Section 5 how time bounds are communicated from individual hosts to server groups.

3 The Scalable Protocol

We present the scalable protocol in three steps. First, we assume that the participating hosts are arranged in a communication tree, and we describe how to make a single measurement that relates the local oscillator of each participating machine to the reference clock; the key idea is to make the measurement robust against adversarial participants. Next, we discuss how to schedule measurements to preserve scalability. Finally, we address path selection, the problem of establishing a communication tree.

3.1 Measurement

The first key idea is that we can enable every participant in the system to make a measurement of the global reference clock without trusting other participants and performing only constant work on any host, including the reference host.

The output of a measurement for a host *h* is a triple $\langle h_1, g_2, h_3 \rangle$. Values h_1 and h_3 are in units of the local oscillator on *h*, and value g_2 is in the units of the global oscillator. The measurement triple indicates that the global oscillator read g_2 at the same moment that the local oscillator read some value h_2, where $h_1 \le h_2 \le h_3$.

3.1.1 Measurement algebra

For the measurement to be useful in the future, we need some constraint on the relationship between the local and reference oscillators, or else we know nothing after the measurement is complete. We assume that oscillators *g* and *h* have rates differing by at most some bound λ:

$$|h-(g-h_0)| \le \lambda(g-h_0)$$

Or, solving for g:

$$\frac{1}{1+\lambda}h + h_0 \le g \le \frac{1}{1-\lambda}h + h_0$$

Given our measurement, we can bound h_0 in terms of $\langle h_1, g_2, h_3 \rangle$:

$$g_2 - \frac{1}{1-\lambda}h_3 \le h_0 \le g_2 - \frac{1}{1+\lambda}h_1$$

Into the future, at any time $h > h_3$, we can bound the values of g by substituting minimum and maximum values of h_0:

$$\frac{1}{1+\lambda}h + g_2 - \frac{1}{1-\lambda}h_3 \le g \le \frac{1}{1-\lambda}h + g_2 - \frac{1}{1+\lambda}h$$

3.1.2 Hierarchical synchronization

One way to make a scalable measurement of a reference clock, adopted by the NTP protocol, is to use *hierarchical synchronization*. Hosts in the topmost layer of the communication tree (*stratum 1*) directly measure the reference clock. Each host in stratum 2 measures a host in stratum 1, and computes information about the reference clock from that measurement and statistical information provided by the stratum 1 host. Hierarchical synchronization seems to require a participant to trust each ancestor on the path to the root to provide correct information about its synchronization measurements; we cannot afford that trust.

3.1.3 Hierarchical communication

To achieve scalability, however, hierarchy seems like an indispensable tool. To exploit hierarchy in the presence of adversaries, we remove from the protocol reliance on the purported computations of intermediate nodes. Instead, we use intermediate nodes only to relay communication to the global reference host; we can use cryptographic techniques to immediately verify whether an intermediate node has faithfully performed its communication duties.

A simple manifestation of this idea is to use signed communication between each participant and the global reference host. Concatenating signed messages on the way up the tree and demultiplexing messages on the way down clearly enables us to limit the number of packets seen by each host to a constant (the tree fanout). However, it leaves the problem that the number of bytes seen by hosts high in the tree is $O(n)$;

furthermore, the root host must verify $O(n)$ signatures and sign $O(n)$ outbound messages.

The problem is solved by observing that host *h* does not require that it receive a personal message from *g*, nor does it even require that *g* even be aware of *h*'s existence. To achieve the measurement property $h_1 \le h_2 \le h_3$, *h* only cares that for some message from *g* timestamped g_2, *h* can verify that g_2 occurred after some time h_1. Such a proof supports the left inequality; the right inequality is supported because of the causality of the universe and because *g* is trusted never to sign a timestamp that its oscillator has not yet produced.

The measurement protocol, then, works like this. Host *h* produces a nonce at local time h_1, and makes a note of that relationship. (We refer to the nonce itself by the local time h_1 that it was created.) Host *h* forwards the nonce to its parent in the communication tree.

That parent host *p* waits for other children to submit their nonces $i_1 \cdots k_1$, then it produces its own nonce p_1. Host *p* computes a cryptographic hash function to create a derivative hash $h = (h_1, i_1 \cdots k_1, p_1)$, and forwards that hash to its parent.

This process recurs up the tree, until the global reference host *g* collects a set of hashes from the nodes in the top layer. The global reference host reads its oscillator as value g_2, and signs a message containing g_2 and the top-level hashes. It relays that message to the top-level nodes.

The intermediate nodes relay the signed message from *g* down the tree, along with the Merkle-tree sibling data [7]: each node passes to its children the list of hashes that composed the input to its derivative hash.

When these data arrive at host *h*, *h* can verify that the message from *g* is indeed from *g* because of its signature. Furthermore, *h* can verify that g_2 must have occurred after h_1, for if it had not, then either an adversary managed to guess the value of nonce h_1, or an adversary managed to invert the cryptographic hash function to show how the top-level hash in g_2 can be computed as a function of input h_1.

As described, the protocol resists adversaries tampering with communication. But if an adversary merely mutes or delays communication, the protocol has two shortcomings. First, because intermediate nodes wait for all of their children to participate, any single failure would cause the protocol to halt. We address this problem in Section 3.2. Second, while a delay or drop in communication cannot cause a participant to compute incorrect bounds on the relationship between its clock and the global clock, it certainly prevents the protocol from tightening those bounds (or measuring initial bounds). We discuss how to circumvent this denial of service in Section 3.3.

3.1.4 Historical rate information

With only a bound on the rate deviation of the clocks, our knowledge of the synchronization of the clocks deteriorates (the bounds grow) as time passes from h_3, the moment of measurement. We bound rather than estimate the time, so simply estimating rate, which can improve time estimates, does not improve the resulting bounds. There are two ways we could consider using historical rate information to improve our algorithm.

3.1.4.1 Second-derivative assumption

We could consider an assumption on the deviation between the second derivative of the local and reference clocks. Such an assumption effectively enables us to use historical rate information to reduce deterioration in the bounds, since an observed rate would then be known not to suddenly swing.

Carrying a laptop from a cooled office to a park bench in the sunshine might suddenly change the rate of the oscillator in the laptop. Thus this assumption might be justified only if the thermal mass of each oscillator in the system is insulated from temperature changes in its environment. Furthermore, because the new assumption affects the squared term, its effectiveness at limiting bounds drift is quickly subsumed by the basic rate deviation assumption.

3.1.4.2 Mostly-constant rate assumption

Suppose that the rate deviation of any oscillator from a nominal rate is determined by the sum of two terms: one due to process limitations, which remains constant once the oscillator is put into service, and a second which varies due to thermal fluctuations. If the magnitude of the first term substantially dominates the second term, then we can measure the long-term rate, add conservative values reflecting the maximum deviation due to thermal effects, and have a dynamic bound on rate deviation that is much more precise than the static bound.

Typical "AT cut" crystal oscillators used in microprocessor circuits deviate in rate by a factor less than 5×10^{-6} across a temperature range of 0-70ºC. The total variation, accounting for both process deviation and temperature deviation, is typically listed as 10^{-4}. Thus, by measuring the constant term due to process variation, we stand to improve the deterioration rate of our bounds by more than an order of magnitude.

Maintaining this long-term rate information without loss requires maintaining a pair of convex hulls and the tangents between them. This task takes $O(1)$ amortized time per measurement. With adversarial input, it can consume linear space; that is, we may need to store every measurement ever made to ensure optimal results. However, a simple greedy algorithm (discarding minimum-angle points on the convex hulls) saves space with minimal impact on accuracy. In practice, since an adversary controls the bounds we measure, we would apply the greedy pruning algorithm to maintain a constant number of points on the convex hulls.

3.2 Scheduling

The basic description of the aggregated measurement protocol in the previous section had internal nodes in the communication tree blocking until all of their children report in. That description is clearly inadequate, because a single faulty host can stall the entire protocol.

Instead, let a host accept a hash from its child at any time, with each new hash displacing the last one from that child. According to some schedule, the host aggregates its children's hashes and its own nonce, and submits a new hash to its parent. This protocol ensures that any host submitting its hash "on time" will see its nonce included in the measurement, while hosts that abstain only harm themselves and their children in the tree.

The challenge, then, is to identify a suitable schedule for internal nodes to aggregate hashes, and a schedule for the root node to sign timestamps. First, the schedule should ensure that any host with a path to the root composed of non-faulty hosts succeeds in making enough measurements and with minimal delay so that it converges on "good" bounds. Second, a good schedule is parsimonious, to save network traffic and work by the aggregating hosts.

3.2.1 Unsynchronized aggregation

A simple schedule is for each host to aggregate periodically, but without any particular synchronization. The justification for this approach is that relying on synchronization is difficult, since that is what the entire protocol is trying to achieve.

The tightness of the bounds computed by a participant is limited by the total delay between the construction of the nonce at h_1 and the receipt of the timestamp at h_3. If each host on the path from *h* to *g* aggregates periodically and without phase synchronization, then that delay will include waiting time proportional to the product of the aggregation period and the depth of *h* in the communication tree.

Suppose that hosts make measurements with period P. The clock-rate deviation λ requires that in the duration of one inter-measurement period, a host's bounds must grow by λP, so there is no point in performing a measurement with precision much better than λP.

So, we can relate measurement accuracy of m, measurement period P, aggregation period q, network delay μ, and tree depth d with the following two conditions:

$$\lambda P < m$$
$$d(q+\mu) < m$$

It is also clear that $\lambda P \approx d(q+\mu)$, for if one is much bigger than the other, then one source of error dominates the other, and we waste either measurements or aggregations.

The cost of a measurement is that of sampling the reference clock (which is expensive in our environment) and performing a digital signature (which is typically expensive). The cost of aggregation is the computation of a hash and the sending and receipt of a (hash-sized) packet, once per participant in the system.

For our application, we expect $\lambda = 0.01$, $\mu = 1\text{ms}$, and $d < 10$; we imagine $m = 1\text{s}$ will be quite sufficient. Those choices lead to $P = 100\text{s}$ and $q \approx 100\text{ms}$. The cost of P is a digital signature and BFT-state-machine operation every couple of minutes. The cost of q is that each host aggregates and sends ten packets per second, and hence receives some $10f$ packets per second, where f is the tree fanout.

We are reasonably satisfied with these values from an engineering perspective: for our system, we can tune the parameters and discover that the resulting costs are acceptable. Ideally, however, it would be nice if the system could self-tune, milking out ideal bounds (within a constant factor of $d\mu$). The present approach also wastes work proportional to $1/\lambda$: better clocks allow better synchronization, exploiting which requires more aggregations per measurement. It would be nice if the algorithm used resources constant in the quality of the clocks.

3.2.2 Self-synchronized aggregation

It would be nice to improve the aggregation schedule to not scale poorly with improved synchronization quality. Indeed, it seems that the aggregation tree should enable us to limit the number of packets received and the work performed by each host, including the reference host, to be proportional to the fanout f.

The basic idea is to give each host a budget of k hash submissions per measurement interval. The host tries to use its budget to minimize the time its hash spends waiting at the next level in the tree. For example, if the host believes that the next measurement occurs at 4:00, and its local clock bounds (from prior measurements) give it an accuracy of $\delta = 5\text{s}$, then the host can send a packet every $5/k$ seconds in that interval. If at least one arrives on time, then his measurement delay will be limited only by his parent's delay (unavoidable), network delay to and from his parent (unavoidable), and the maximum δ/k wait time. Since the latter quantity is smaller than the original δ, we can expect the host's bounds to converge to the sum of the unavoidable delay terms.

There are a couple of problems with this basic approach. The first is that the host cannot know when the parent's deadline is, since the parent must send its hash before the global measurement deadline to ensure it arrives on time. The second, related problem is that the host must anticipate the network delay, for if the global 4:00 occurs at the early end of its bounds, then a packet sent at exactly 4:00 will arrive too late to provide synchronization. But measuring network delay will not help, since an adversarial network can offer fast delivery for delay measurement and then poor performance when the timeliness matters.

We have tried several approaches to mitigate these problems, but each has its limitations. One is to send a series of submissions before the deadline with geometrically-dropping delays between them. This approach "discovers" the network delay to within a constant factor (the base of the exponential progression) at each layer in the tree. Unfortunately, it means that hosts at layer d in the tree can hope to converge on δ-bounds no better than exponential in d.

A second strategy sends submissions for a δ-wide window *before* the first time the host believes could be the deadline. The idea is that if δ is bigger than the maximum network delay over an interval, then the window will accommodate the network delay, providing a measurement; if δ is smaller, then the host already has good synchronization and does not mind missing a measurement opportunity. Unfortunately, hosts below the well-synchronized host also miss out on a measurement, even if they need one. With our best variation on this scheme, we could ensure that all hosts in the tree converge, but to δ-bounds no better than quadratic in d.

3.2.3 Bad ideas for aggregation scheduling

Suppose we use an unsynchronized schedule with a variable aggregation period. Smaller periods occur near the beginning of the measurement period *P* as measured on the global reference timeline, and larger ones fill out the later times. Poorly synchronized hosts will find their hashes waiting a long time during the infrequent period, but that time is still short enough that the host improves its synchronization so that future submissions are better-aligned with the high-accuracy (low-delay) phase. This "optimization" offers both small aggregation-induced delays and reduced total message traffic versus the periodic approach. However, that reduced traffic is arranged in a (deliberately) clumpy fashion that only reduces long-term average resource usage, but not peak load, making the optimization of questionable value.

Suppose we execute an auxiliary protocol in which each parent informs its children when (in local time) it plans to aggregate, and the children use that local information to deliver hashes in a timely fashion. This approach requires coming up with *another* (local) synchronization scheme, and then reasoning about why it works reliably. Indeed, the problem is harder than one might guess: as described in Section 3.2.2, an adversary makes measurements of network behavior difficult to use. We suspect that this approach is doomed.

3.3 Path selection

In prior sections, we established a measurement protocol and a scheduling algorithm that keeps the protocol scalable with a small constant. We still have to address the problem that the path from a host to the global reference host involves some faulty host.

The good news is that the faulty host cannot cause any host to compute incorrect bounds; it can only prevent a host from computing very tight bounds, perhaps no tighter than $(-\infty, \infty)$. Because the victim host knows that its bounds are poor, however, it can attempt the protocol along an alternate path to the root. If its bounds improve, it can abandon the path that (because of a faulty participant) produced lousy bounds.

Obviously, the protocol doesn't scale if every host attempts to use every other host as a parent on each measurement, to maximize the likelihood of discovering a non-faulty path. A simple fix: let each correct host try two or three paths at any time, periodically discarding the worst-performing parent to test some other potential parent. Faulty hosts will observe no such restraint, of course; we expand this point shortly.

Even a set of polite, restrained hosts can get into trouble using the present path-selection algorithm, however. Because each host is greedy, each will eventually discover that the shortest path to the root is to contact the root directly, forming a degenerate tree with only one layer. To preserve scalability, the protocol must enforce a limit on fanout.

Perhaps we can enforce a tree by limiting the number of children each hosts accepts. Such behavior could cause our tree of correct hosts to self-organize into a tree with acceptable fanout. Unfortunately, it also presents a golden opportunity to faulty hosts: if the set of faulty hosts can clog all of the fanout slots at the root, they can impede access to the global reference clock by any of the correct hosts.

We have not yet discovered a simple approach that can impose a well-shaped tree on a fully-connected graph in the presence of faulty hosts. Instead, we rely on the application of the algorithm to provide an adversary-resistant scalable subgraph from which the greedy algorithm samples edges to construct correct paths.

In Farsite, the subgraph follows from the natural structure of the file service. First, there is a hierarchy of BFT replicated-state-machine groups. The topmost group is the global reference host (see Section 4). If a group *A* delegates work to a group *B*, then every host comprising *B* is entitled to contact every host in group *A*. After all, the correct members of group *A* must trust that most members of group *B* are correct. Finally, if a host *h* is a client accessing resources managed by a group *B*, then *h* is allowed to contact members of *B*. If *B* is willing to provide file services to *h*, it should at least be willing to give *h* the time of day.

Why is this subgraph sufficient to ensure effective clock synchronization in Farsite? If host *h* needs to synchronize with respect to *B*, then either it has some correct path to the root, if by no other means, then through a host in *B*, or else every member of some BFT group in the path from *B* to the root group is corrupt. If that is the case, then *B* is effectively faulty anyway. If *B* is faulty, then it is not important that *h* synchronize with it.

4 The Global Reference Clock

The Byzantine-fault-quantifying protocol can withstand faulty hosts in the system, but requires a trustworthy global reference clock. In our application and others, the reason for withstanding faulty hosts is that no particular host in the system is trustworthy.

Hence we produce a trustworthy global clock from the set of untrusted hosts composing the root server group. To do so, we arrange for the BFT state machine to serve in the role of the reference host.

First, the replicas composing the machine execute a Byzantine-fault-tolerant clock synchronization algorithm [6,9,2] to synchronize the correct hosts on a scalar time value consistent with real time. Then the state machine collects hashes, timestamps them, and signs its timestamp message so that it can be propagated back down the tree.

Each individual host in the root server group belongs to the top layer of the communication tree. When each host performs an aggregation, it submits its hash into the BFT state machine by acting as a client to the same server group in which it is a server replica; the machine's state thus collects the submitted hashes.

To timestamp the hashes, the server group must execute an operation with an agreed-upon time value. Our BFT replication algorithm [1] timestamps operations as follows: when the primary replica proposes a request as the next operation to be performed by the state machine, it also proposes a timestamp value. Each correct replica vets the proposal, and a replica vets the proposed timestamp by ensuring that it agrees to within some established bound ε of the replica's notion of time. Such a bound must account for both precision limitations of the Byzantine clock synchronization algorithm and the message delay in delivering the proposal. If the proposal does not pass muster, correct replicas clamor for a view-change, to elect a new primary replica. Hence if ε is tighter than the actual synchronization of the group, then the progress of the state machine will stall until the group's synchronization converges.

Now that we have a mechanism for timestamping BFT state machine requests, we can declare what the state machine does with the submitted hashes: it simply adds a $\langle \text{time}, \text{hash} \rangle$ record to its replicated state, and returns no value.

As each replica decides that the timestamp deadline has passed, it gathers the set of $\langle \text{time}, \text{hash} \rangle$ records stored in its view of the replicated state machine, signs them, and sends the signature out to each top-layer host in the communication tree. Each top-layer host collects $f+1$ signatures of its hash to form a "group-signed timestamp:" effectively, a document signed by the BFT state machine. That signed timestamp is the message propagated back down the Merkle tree in the reply phase of the measurement protocol.

The meaning of a group-signed $\langle \text{time}, \text{hash} \rangle$ record must be interpreted carefully. Because the time value was only known within a bound of ε, the value g_2 from Section 3.1.1 is actually a time bound: $(g-\varepsilon, g+\varepsilon)$.

5 Injecting bounds into server groups

The previous section detailed how we use (conventional, unscalable) BFT clock synchronization and BFT state machines to produce a trustworthy global reference clock. In so doing, the root server group acquires a notion of time: it is in fact the reference for all time in the system. In this section, we address how to make time-bound measurements available to server groups other than the root group.

The present task is to take the measurements computed by the possibly-faulty replicas that compose a server group, and aggregate them to produce a correct time-bound for each operation performed by the group. Once we have achieved the task, then operations that need to evaluate the validity of leases granted by or to the group have available as an input bounds on the current time.

The solution is parallel to the proposal and acceptance of scalar time in the root group. A correct primary replica with time bounds (b_1, b_2) will propose bounds $(b_1 - \varepsilon, b_2 + \varepsilon)$. A correct replica with bounds (b_3, b_4) will accept the proposed bounds if:

$$b_1 - \varepsilon \le b_3 \le b_1 + \varepsilon$$
$$b_2 - \varepsilon \le b_4 \le b_2 + \varepsilon$$

To ensure liveness, we must ensure that replicas can synchronize to within ε. In the scalar proposal mechanism from the previous section, the choice of ε accounts for the network delay and the relative synchronization of the replicas, which in that scenario were linked directly to one another by their participation in a BFT clock-synchronization protocol. Here, ε must accommodate bounds that vary due to delays in the BFQ clock-synchronization tree, those convergence limits described in Section 3.2. Thus, to remain truly scalable, ε must vary with the depth of the members in the BFQ communication tree (which happens to correspond to the BFT server group tree in our application), or be otherwise automatically tuned. For our application, where we assume total system size $n \le 10^5$, asymptotic scalability is not essential, and we can afford to select a very conservative ε.

6 Related Work

Haber and Stornetta's time-stamping schemes exploit hashes to produce non-forgeable timestamps that scale to many clients [3]. The measurement step in our protocol acquires a timestamp, although the requirements on it are different: it is not important, for example, that a client be able to prove to a third party when its nonce was stamped.

Lamport and Mann propose a scalable protocol for distributing time from a trusted time source using hierarchical synchronization. Their protocol propagates path information, and hosts use path information and computed time intervals to discard invalid data from faulty or corrupt intermediaries [5].

7 Summary

We have presented a scalable protocol and algorithms for synchronizing clocks in a distributed system that produces as output conservative bounds on the accuracy of the synchronization. The protocol is *Byzantine-fault quantifying* in that the results of malicious behavior are apparent in the bounds output by the protocol, and those bounds may be used to prevent safety violations.

The three central components of the protocol are the measurement protocol, the schedule that exploits aggregation, and the path-selection algorithm that helps correct hosts "route around" malicious behavior. We explored how to use various assumptions on oscillator behavior to improve the quality of the bounds.

We presented a technique for building a virtual global reference clock from untrusted hosts using conventional, non-scalable Byzantine-fault-tolerant clock synchronization. We presented a technique for gathering clock bounds computed by member replicas of a BFT state machine to produce a deterministic bounds pair useful inside the state machine.

We discussed how this protocol fits into the context and engineering assumptions of the Farsite scalable filesystem. Farsite's control structure provides an implicit graph for path selection. Non-critical timing in Farsite tolerates the modest quality bounds produced by a simple aggregation schedule. Farsite's distrust of all hosts necessitates a virtual global reference clock and a mechanism for injecting bounds into a server group.

Acknowledgements

Our thanks to the Farsite team, of whose project this protocol is only a sliver. Thanks to Mike Sinclair and Tom Blank for sharing their experiences with real-world oscillators (Section 3.1.4.2). Thanks to John Dunagan for discussion about maintaining long-term rate information (Section 3.1.4.2). Thanks to Dan Simon for discussions about aggregation scheduling (Section 3.2).

References

[1] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", In Proc. 3rd OSDI, USENIX, February 1999.

[2] Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. "Dynamic Fault-Tolerant Clock Synchronization." J. ACM 42(1) pp. 143–185, January 1995.

[3] S. Haber and W. S. Stornetta. "How to timestamp a digital document." *J. of Cryptology*, 3(2), 1991.

[4] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, 6(1):51--81, February 1988.

[5] Leslie Lamport and Timothy Mann. "Marching to Many Distant Drummers." Unpublished manuscript available at http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-drummers, 1997.

[6] Leslie Lamport and P. M. Melliar-Smith. "Byzantine Clock Synchronization." PODC '84, 1984.

[7] R. Merkle, "Protocols for Public Key Cryptosystems." IEEE Symposium on Security and Privacy, 1980.

[8] David L. Mills. "Internet Time Synchronization: the Network Time Protocol." In Zhonghua Yang and T. Anthony Marsland (Eds.), *Global States and Time in Distributed Systems*, IEEE Computer Society Press, 1991.

[9] Fred B. Schneider, "Understanding Protocols for Byzantine Clock Synchronization." Cornell Department of Computer Science TR 87-859, 1987.