A Paradigm for Reliable Clock Synchronization*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

April 1986

ABSTRACT

Existing fault-tolerant clock synchronization protocols are shown to result from refining a single clock synchronization paradigm. In that paradigm, a reliable time source periodically issues messages that cause processors to resynchronize their clocks. The reliable time source is approximated by reading all clocks in the system and using a convergence function to compute a fault-tolerant average of the values read. The performance of a clock synchronization algorithm based on the paradigm can be quantified in terms of the two parameters that characterize the behavior of the convergence function used: accuracy and precision.

*This work is supported, in part, by NSF Grant DCR-8320274 and Office of Naval Research contract N00014-86-K-0092.

1. Introduction

Certain applications require synchronized clocks at the processors in a distributed system. For example, the accuracy of performance statistics computed in terms of elapsed time between events at different sites depends on how closely the clocks at participating sites are synchronized. Also, timeouts and other time-based synchronization schemes (such as the state-machine approach [Lamport 84]) often involve delays that are proportional to how closely the clocks at participating sites are synchronized.

Even if we could start all processor clocks at the same time, they probably would not remain synchronized for long. Crystal clocks found in today's processors run at rates that differ by as much as 10^{-6} seconds per second from real time and thus can drift apart by 1 second every 10 days; clocks based on power-line frequency can drift considerably more than this—when used as a time base, the power grid in the Northeastern United States typically drifts 4 to 6 seconds from real time over the course of an evening [Mills 85]. Keeping clocks in a distributed system synchronized without appealing to a single, centralized, time service requires that clock values be exchanged and adjusted periodically. If failures can result in faulty processors exhibiting arbitrary behavior, then the protocol has the additional burden of tolerating erroneous and inconsistent clock values.

This paper surveys fault-tolerant protocols for synchronizing clocks in a distributed system where faulty processors can exhibit arbitrary behavior. We show how existing clock synchronization protocols can be viewed as refinements of a single clock synchronization paradigm. That paradigm is described in section 2. In section 3, we discuss properties of convergence functions, the central component of a clock synchronization protocol. Techniques for reading clocks across a computer-communications network are described in section 4. Section 5 discusses how agreement protocols can improve the performance of a convergence function. Some conclusions and related work appear in section 6. An appendix derives bounds on the resynchronization interval.

2. Clock Synchronization

The clock at a correct processor p can be viewed as implementing in hardware a monotonically increasing, continuous[1] function c_p that maps a real time t to a clock time $c_p(t)$ that, for some positive constants μ and ρ, satisfies:

---

[1]Strictly speaking, c_p is not continuous because it advances in discrete ticks. However, if these ticks happen frequently enough, it is impossible for a program running on a processor p to identify two successive real times t and t' where $c_p(t) = c_p(t')$. Therefore, we can treat c_p as being continuous.

Initial Value: $0 \le c_p(0) \le \mu$ (2.1)

$$\textbf{Correct Rate:}\quad 1-\rho \le \frac{c_p(t_2)-c_p(t_1)}{t_2-t_1} \le 1+\rho \quad \text{for } t_1<t_2. \tag{2.2}$$

Condition (2.1) asserts that c_p is initially set to some value within μ of the real time; (2.2) asserts that the *drift rate* of c_p is within ρ of 1 clock-second per real-time second.

We make no assumptions about the behavior of clocks at faulty processors—not even that they can be modeled by functions. A clock on a faulty processor need not increase as real time passes and might give inaccurate or conflicting information when it is read.

A clock synchronization protocol implements a *virtual clock* $\hat{c}_p$ at each processor p. Virtual clocks at any correct processors p and q satisfy

$$\textbf{Synchronization:}\quad |\hat{c}_q(t)-\hat{c}_p(t)| \le \hat{\delta} \quad \text{for all } t, \tag{2.3}$$

$$\textbf{Rate:}\quad 1-\hat{\rho} \le \frac{\hat{c}_p(t_2)-\hat{c}_p(t_1)}{t_2-t_1} \le 1+\hat{\rho} \quad \text{for } t_1<t_2, \tag{2.4}$$

for given constants $\hat{\delta}$ and $\hat{\rho}$.

If a reliable time source is available, then clock synchronization is simple. The reliable time source periodically broadcasts the correct time and, upon receipt of such a broadcast, each processor adjusts its virtual clock accordingly. Provided the broadcast arrives at each processor at about the same time, all processors will adjust their clocks so that (2.3) is satisfied. Provided the broadcast is done frequently enough, processor clocks will not drift too far apart in the interval between broadcasts, so (2.3) will be maintained. Provided that no processor has to adjust its clock by too much upon receipt of a broadcast, the adjustment can be spread over the interval that follows and (2.4) will be maintained. We have only to implement the reliable time source.

The reliable time source serves two functions in the clock synchronization protocol outlined above.

RTS1: It periodically generates an event that causes every processor to resynchronize its clock at about the same time.

RTS2: It provides every processor with a time value that can be used in adjusting that processor's local clock. If each processor adjusts its clock based on the value it receives at the time that value is received, then (2.3) will hold.

Note that while one might desire that a reliable time source *always* be able to provide the correct time, RTS1 and RTS2 merely require that the correct time be made available periodically.

Although it is easy to satisfy RTS1 and RTS2 using a single clock, the resulting time source in not likely to be fault tolerant. Fortunately, a distributed reliable time source that satisfies RTS1 and RTS2 and is fault-tolerant can be constructed when approximately synchronized clocks are available. RTS1 is achieved by having each processor attempt resynchronization when virtual clocks in the system reach a certain value. RTS2 is achieved by having each processor independently read all the other clocks in the system and compute some type of fault-tolerant average of the values gathered.

Adjusting a virtual clock $\hat{c}_p$ can be viewed as simply starting another virtual clock that runs concurrently with the old one. Thus, after the i^{th} adjustment, p starts a new virtual clock $\hat{c}_p^i$. Define FIX_p^i to be the adjustment to c_p, processor p's (hardware) clock, that results in $\hat{c}_p^i$. That is,

$$\hat{c}_p^i(t) \equiv c_p(t) + FIX_p^i.$$

We refer to $\hat{c}_p^i$ as a *superscripted virtual clock* to distinguish it from $\hat{c}_p$.

We can now describe the clock synchronization protocol outlined above for a processor p in a distributed system consisting of N processors.

$i := 1;\quad FIX_p^i := 0;$
do forever
    await Next Synchronization;
    Assume real time is now t_T;
    $FIX_p^{i+1} := CF(\hat{c}_p^i(t_T), \hat{c}_1^i(t_T), \ldots, \hat{c}_N^i(t_T)) - c_p(t_T);$
    $i := i+1$

CF, called a *convergence function*, implements the fault-tolerant average used to satisfy RTS2. In particular, $CF(\hat{c}_p^i(t_T), \hat{c}_1^i(t_T), \ldots, \hat{c}_N^i(t_T))$ provides the value of the reliable time source at real time t_T.

Three important things about this protocol remain unspecified. First, there is the implementation of "**await** Next Synchronization". An obvious approach uses $\hat{c}_p^i$:

$$\textbf{do } \hat{c}_p^i(t_T) \neq NextSynch \rightarrow \textbf{ skip od}$$

where *NextSynch* is a previously agreed on time. When virtual clocks at correct processors are synchronized to within $\hat{\delta}$, this scheme ensures that all processors resynchronize for the i^{th} time within $\hat{\delta}$ of each other. Another implementation of "**await** Next Synchronization" is for a processor to broadcast a message when $\hat{c}_p^i(now) = NextSynch$ and resynchronize when enough such messages have been received. The details of this scheme, which is based on a simple form of agreement, are given in section 5.

The second item that remains unspecified in our protocol is the convergence function CF. Properties and examples of convergence functions are the subject of sections 3 and 5. The final item to be specified in our protocol is how one processor reads the virtual clocks at other processors. Two techniques for this are discussed in section 4.

Different choices for the three unspecified items in the paradigm result in different clock synchronization protocols. The choices covered in sections 3, 4, and 5 permit all the published clock synchronization protocols we know of that do not make use of an external time source to be viewed in terms of our paradigm. Thus, the paradigm is quite general and provides a vehicle with which the clock synchronization literature can be surveyed.

3. Convergence Functions

In its most general form, a convergence function CF for use in a system of N processors is a function of $N+1$ arguments. The first argument is for the value owned by the processor invoking CF; each of the following N arguments is for a value from each of the N processors in the system. This means that when a processor p evaluates CF, the same value will appear in two argument positions—the first and the $p+1^{st}$.

For a function CF to be a convergence function, it must exhibit certain properties. First, because the relative distribution of the virtual clock values—and not their magnitudes—should matter when they are combined to implement a reliable time source, CF should satisfy

Translation Invariance: $CF(x_p+v,\ x_1+v,\ ...,\ x_N+v) = CF(x_p,\ x_1,\ ...,\ x_N)+v$.

Next, we require that when CF is evaluated by two different processors using similar values for $N-k$ corresponding arguments, it produces values that are closer together than its arguments. More specifically, for CF to be a convergence function there must exist a constant k called the *fault-tolerance degree* and a function π called the *precision*. The fault-tolerance degree specifies the number of faulty argument values that can be tolerated by CF; the precision specifies how close together values can be brought using CF. This is formalized by the

Precision Enhancement Property: If there exist values δ, ε, and indices a_1, ..., a_{N-k} such that $x_p = x_{a_i}$, $y_q = y_{a_j}$ and

(a) $(\max i, j: 1 \le i, j \le N-k: \ | x_{a_i} - x_{a_j} |) \le \delta$
(b) $(\max i, j: 1 \le i, j \le N-k: \ | y_{a_i} - y_{a_j} |) \le \delta$
(c) $(\forall i: 1 \le i \le N-k: \ | x_{a_i} - y_{a_i} | \le \varepsilon)$

then $| CF(x_p, x_1, \ldots, x_N) - CF(y_q, y_1, \ldots, y_N) | \le \pi(\delta, \varepsilon)$.

Conditions (a) and (b) define δ to be the width of the interval containing correct values; when using CF to implement a reliable time source, these conditions are satisfied if correct virtual clocks are synchronized to within δ when read by p and q. Condition (c) stipulates that corresponding (correct) arguments to CF are at most ε apart; for a reliable time source, this condition is satisfied if two values obtained by reading the same virtual clock v (real) seconds apart do not differ by more than $v+\varepsilon$ as a result of drift. The Precision Enhancement Property states that in order for CF to be a convergence function, two evaluations with arguments satisfying (a)−(c) must produce values that are close—at most $\pi(\delta, \varepsilon)$ apart—even though the values used for k of the arguments (presumably, from faulty processors) differ arbitrarily. Thus, provided $\pi(\delta, \varepsilon) < \delta$, CF implements a time source that furnishes different processors with time values that are closer than the least synchronized correct virtual clocks.

The final property of a convergence function CF requires that $CF(x_p, x_1, \ldots, x_N)$ is not too far from any of its arguments that are within δ of $N-k-1$ others.

Accuracy Preservation Property: For values $x_1, x_2, \ldots x_N$, and δ, and indices $a_1, \ldots, a_{N-k}$ such that $(\max i, j: 1 \le i, j \le N-k: \ | x_{a_i} - x_{a_j} |) \le \delta$,

$$(\max i, j: 1 \le i, j \le N-k: \ | x_{a_i} - CF(x_{a_j}, x_1, \ldots, x_N) |) \le \alpha(\delta).$$

Function α is called the *accuracy* of CF. When CF is used as a reliable time source, provided $\alpha(\delta) \le \hat{\delta}$, resynchronizing a clock when correct clocks are no more than δ apart leaves the new clock within $\hat{\delta}$ of all correct clocks.

Examples of Convergence Functions

Examples of functions that satisfy the three properties of convergence functions include:

Egocentric Average: $CF_{EA}(x_p, x_1, \ldots, x_N)$ is the average of all arguments x_1 through x_N that are no more than δ from x_p.

The degree k of fault tolerance for CF_{EA} is characterized by $3k+1 = N$. Precision π is bounded by $\pi(\delta, \varepsilon) = \frac{3f\,\delta}{N} + \varepsilon$ where f is the number of arguments that differ in the two function evaluations; in the worst case, this is slightly less than $1+\varepsilon$. Accuracy is bounded by

$\alpha(\delta)=4\delta/3$.

Fast Convergence Algorithm: $CF_{FCA}(x_p, x_1, ..., x_N)$ is the average of all arguments x_1 through x_N that are within δ of $N-k$ other arguments.

The degree k of fault tolerance for CF_{FCA} is characterized by $3k+1=N$. Precision π is bounded by $\pi(\delta, \varepsilon)=\frac{2f\,\delta}{N}+\varepsilon$ where f is the number of arguments that differ in the two function evaluations; in the worst case, this is $2\delta/3+\varepsilon$. The accuracy is bounded by $\alpha(\delta)=4\delta/3$.

Fault-tolerant Midpoint: $CF_{Mid}(x_p, x_1, ..., x_N)$ is the midpoint of arguments x_1 through x_N after the k highest and k lowest values have been discarded.

The degree k of fault tolerance for CF_{EA} is characterized by $3k+1=N$. Precision is bounded by $\pi(\delta, \varepsilon)=\delta/2+\varepsilon$; accuracy by $\alpha(\delta)=\delta$.

Fault-tolerant Average: $CF_{Avg}(x_p, x_1, ..., x_N)$ is the average of arguments x_1 through x_N after the k highest and k lowest values have been discarded.

The degree k of fault tolerance for CF_{Avg} is characterized by $3k+1=N$. Precision π is bounded by $\pi(\delta, \varepsilon)=\frac{f\,\delta}{N-2k}+\varepsilon$ where f is the number of arguments that differ in the two function evaluations; in the worst case, this is slightly less than $1+\varepsilon$. Accuracy is bounded by $\alpha(\delta)=\delta$.

CF_{EA} was first proposed and analyized in [Lamport & Milliar-Smith 85] in connection with a clock synchronization algorithm. CF_{FCA} is discussed in [Mahaney & Schneider 85], who were the first to view convergence functions (there, called inexact agreement protocols) in terms of accuracy and precision. CF_{Mid} and CF_{Avg} are given in [Dolev *et al.* 83]; CF_{Avg} is the basis for the clock synchronization protocol of [Lundulius & Lynch 84].

4. Reading Clocks from Afar

Processors have access to clock time—not real time. This means that in order for a processor p to read virtual clocks $\hat{c}_1$, ..., $\hat{c}_N$ at the same real time, p must read all N clocks simultaneously. This is impossible because a processor can do only one thing at a time. Moreover, message passing is the only way a processor can obtain a clock value from another in a distributed system. Message delivery times are typically non-trivial and unpredictable. Thus, it is impossible for a single processor in a distributed system to compute $CF(\hat{c}^i_p(t_T), \hat{c}^i_1(t_T), ..., \hat{c}^i_N(t_T)) - c_p(t_T)$ as required by the resynchronization protocol outlined in section 2.

A technique originally proposed in [Lamport & Milliar-Smith 85] allows one processor to compute an approximation for a virtual clock at another. Each processor p maintains a collection of tables $\tau_p^i[1..N]$ containing values that transform $c_p(t)$ into an approximation for $\hat{c}_q^i(t)$. Processor p approximates $\hat{c}_q^i(t)$, by $c_p(t)+\tau_p^i[q]$.

To construct τ_p^i, p periodically communicates with the other processors in the system. Suppose the minimum delay incurred in sending a message between any pair of correct processors is Γ_{min} and Γ_{max} is the maximum delay incurred. Thus, $\Gamma_{max}-\Gamma_{min}$ is the uncertainty in delivery time for a message. A processor p can compute $\tau_p^i[q]$ by executing

> **send** "i^{th} clock time?" **to** q;
> **receive** C **from** q **timeout after** $2\Gamma_{max}+\kappa$;
> **if** timed-out **then** $C := \infty$;
> $\tau_p^i[q] := c_p(t_{now})-\Gamma_{min}-C$

where κ is the maximum length of time (according to p's clock) it can take q to process the request made by p and t_{now} is the real time at which the statement assigning to $\tau_p^i[q]$ is executed.

Define the *clock reading error* $\lambda_p(q)$ to be the error in p's approximation of q's i^{th} virtual clock, and let Λ be the maximum clock reading error. That is,

$$| \hat{c}_q^i(t) - c_p(t)-\tau_p^i[q] | \le \lambda_p(q) \le \Lambda.$$

In order to compute a bound Λ on $\lambda_p(q)$, first note that p's approximation of q's clock drifts from q's clock by at most $2\hat{\rho}$ clock seconds per real second. Initially, $\tau_p^i[q]$ is in error by at most $\Gamma_{max}-\Gamma_{min}$ since only Γ_{min} of the message delay incurred by q's response to p's request for the time is accounted for in the calculation of $\tau_p^i[q]$. Thus, at time t, $\lambda_p(q)$ satisfies

$$\lambda_p(q) \le \Gamma_{max} - \Gamma_{min} + 2\hat{\rho}(t - Lread_p(q)) \le \Lambda$$

where $Lread_p(q)$ is the real time that p last executed an assignment to $\tau_p^i[q]$ in the clock reading protocol above. Although $\lambda_p(q)$ is a function of t, an upper bound on $t - Lread_p(q)$ is usually known, and therefore Λ is a constant.

$\lambda_p(q)$ can be kept small by recomputing $\tau_p^i[q]$ frequently, thereby keeping $t - Lread_p(q)$ small. In practice, it suffices to obtain clock values from all processors just before computing FIX_p^{i+1}, because this minimizes the clock reading error just before the clock values are actually needed. However, for reasonable intervals $t - Lread_p(q)$, $2\hat{\rho}(t - Lread_p(q))\, \Gamma_{max} - \Gamma_{min}$ so minimizing the uncertainty in the network delay is the key to reducing $\lambda_p(q)$.

A variation on this scheme [Lundulius & Lynch 84] reduces the number of messages by half but can increase the clock reading error. Instead of requesting the time, each processor q periodically broadcasts its virtual clock value (including the superscript). Upon receipt of such a message, the receiver p updates $\tau_p^i[q]$. The reduction in number of messages sent is due to lack of explicit request messages—the passage of time, rather than a request message, causes transmission of a clock value. However, in a point-to-point network, clock reading errors can increase when this variation is used. This increase is because a processor p does not necessarily know what communications line it should monitor for the next clock message. Polling the communications lines increases the uncertainty in message delivery delay since it is possible for a message to remain queued at the receiver for the polling cycle time. Most local area networks, however, have a single connection between the processor and the network and therefore do not have this problem.

5. Improved Convergence by Exploiting the Network

An *agreement protocol* allows correct processors in a distributed system to agree on an action to be taken or on a set of values. Use of an agreement protocol to disseminate a signal that causes processors to resynchronize clocks can ensure property RTS1 of a reliable time source. Use of an agreement protocol to disseminate each processor's clock can enhance the precision of a convergence function, hence help with RTS2, by ensuring that corresponding argument positions are equal in two evaluations of CF performed by different processors.

Agreement protocols are generally intended for use with values, not functions like clocks. The general structure of such a protocol is for a processor to send a copy of every value it receives to every other processor. After several rounds of this repeated message exchange, each processor selects one from among the set of values it has received. The criteria for selection depend on the agreement protocol—use of median or mode is not unusual. The relaying of messages through different paths, although seemingly inefficient, is a necessary and important part of most agreement protocols because it prevents correct processors from being confounded by inconsistent values sent by faulty processors.

It is not difficult to modify an agreement protocol intended for disseminating values to permit processors to agree on clocks: clock differences, which are relatively static, are exchanged. A superscripted virtual clock $\hat{c}_x^i$ is stored as a triple $proc, i, offset$ which specifies a clock with offset $offset$ from the virtual clock with superscript i at processor $proc$. (Note, $proc$ need not be the same as x.) Thus, $\hat{c}_x^i(t)$ is approximated by p as $\hat{c}_p^i(t)+\tau_p^i[proc]+offset$. Processor p can send $\hat{c}_x^i$ to another processor q by executing

$$\textbf{send } proc,\ i,\ offset \ \textbf{ to } q \tag{5.1}$$

and q can receive $\hat{c}_x^i$ by executing

$$\textbf{receive } proc,\ epoch,\ offset. \tag{5.2}$$

Subsequently, q approximates $\hat{c}_x^i$ by computing $\hat{c}_q^i(t)+\tau_q^i[proc]+offset$.

Because $\hat{c}_q^i+\tau_q^i[proc]$ is an approximation of $\hat{c}_{proc}^i$, an error is introduced when a clock is passed from one processor to another in this manner. Consequently, different copies of a clock received by a single processor might not be identical. Agreement protocols that test for equality of values must therefore be modified to handle clocks passed around the system in this fashion. The modification involves considering two values equal if they are approximately equal. Two values are approximately equal if they are within $\lambda_p(proc)+\lambda_q(proc)$ where p first converted $\hat{c}_x^i$ to a triple and q reconstructs $\hat{c}_x^i$. Values are therefore approximately equal if they are within 2Λ. (Recall, Λ is the maximum value of $\lambda_a(b)$ for any processors a and b.)

5.1. Crusader's Agreement

Crusader's Agreement [Dolev 82] allows a designated processor, called the *transmitter*, to disseminate a value in such a way that:

CRU1: All correct processors that do not "know" that the transmitter is faulty agree on the same value.

CRU2: If the transmitter is correct, then all correct processors agree on its value.

Thus, Crusader's Agreement potentially partitions processors into three classes: those that are faulty, those that are correct and "know" that the transmitter is faulty, and those that are correct and have agreed among themselves on a value from the ones sent by the transmitter. Crusader's Agreement is simple and inexpensive to implement in a distributed system where fewer than 1/3 of the processors are faulty and reliable communications is possible.[2] The following 2-round protocol for Crusader's Agreement allows clock values to be disseminated.

(1) The transmitter sends its clock to all other processors using (5.1).

(2) Each processor uses (5.1) to send the clock it has received using (5.2) from the transmitter to all processors (including itself).

---

[2] A communications failure can always be viewed as a failure of either the sending or receiving processor. Assuming reliable message delivery here is merely an expository convenience.

(3) Each processor sifts through the clocks it received in step (2) to identify a set of at most m suspicious processors that, if faulty, could account for differences among the values. If, after ignoring values received from suspicious processors, the differences in the values that remain are within 2Λ, then agree on the clock received in step (2); otherwise, decide that the transmitter is faulty.

The Crusader's Convergence Algorithm CF_{CCA} of [Mahaney & Schneider 85] is the result of employing Crusader's Agreement to disseminate values before applying CF_{FCA}. CF_{CCA} has half the precision of CF_{FCA} (i.e. convergence is twice as good) and the same accuracy and degree of fault tolerance. It is interesting to note that when CF_{FCA} is iterated twice—which requires the same two rounds of message exchange as CF_{CCA}—the worst case precision is $4\delta/9$, clearly inferior to the $\delta/3$ precision achieved when the two rounds of message exchange is used for a Crusader's Agreement. Employing Crusader's Agreement before CF_{EA}, CF_{Mid} and CF_{Avg} also results in precision improvements for those convergence functions.

5.2. Byzantine Agreement

Byzantine Agreement [Lamport *et al.* 82] is stronger than Crusader's Agreement—all correct processors agree on a value whether or not the transmitter is faulty:

BYZ1: All correct processors agree on the same value.

BYZ2: If the transmitter is correct then all correct processors agree on its value.

The literature contains numerous protocols for establishing Byzantine Agreement. An early survey of the area appears in [Fisher 83] and a tutorial in [Schneider 85]. One protocol especially suited for use in local area networks is described in [Babaoglu & Drummond 85]. See [Lamport & Milliar-Smith 84] for an example of one of the classic protocols in action.

For use in a convergence function, we can ignore details of implementing a Byzantine Agreement Protocol—it suffices to know what it achieves. When a Byzantine Agreement is used to disseminate clocks, it ensures that all correct processors agree within 2Λ on an approximation for the clock at each processor. Correct processors evaluating a convergence function will then differ by at most 2Λ in values in corresponding argument positions. Define CF_g to be a function that returns its g^{th} largest argument. If $k < g < N-k$ and we employ a Byzantine Agreement protocol that can tolerate k failures to disseminate the arguments used in CF_g, then we obtain a convergence function for clock synchronization:

(1) Each processor employs the Byzantine Agreement protocol to disseminate its clock.

(2) Each processor then uses CF_g to choose as its new clock the g^{th} fastest clock.

To see why this works, note that provided there are k or fewer failures, the Byzantine Agreement will ensure that each processor p obtains a vector $v_p[1]$ through $v_p[N]$ of the clocks at other processors. Due to BYZ2, if q is correct then $v_p[q](t)$ must be within 2Λ of $\hat{c}_q(t)$. Without loss of generality, assume that $v_p[1](t) > v_p[2](t) > \cdots > v_p[N](t)$. According to BYZ1, $|v_p[g](t) - v_q[g](t)| \leq 2\Lambda$ for all correct processors p and q. Thus, by selecting the g^{th} largest clock, we are guaranteed that the clock selected by each processor reads within $\varepsilon = 2\Lambda$ of the clock selected by every other. This means that the precision of the algorithm is $\pi(\delta, \varepsilon) = 2\Lambda$—the precision for the convergence function is independent of δ! To bound the accuracy, note that because $k < g < N - k$, the g^{th} largest clock lies between correct clocks. If correct clocks are within δ, then the new clock is no more than δ away from a correct clock, so we conclude that the accuracy of the algorithm is $\alpha(\delta) = 1$.

Clock synchronization algorithms based on Byzantine Agreement are described in [Lamport & Milliar-Smith 84] and analyzed in [Lamport & Milliar-Smith 85].

5.3. An Optimization

The convergence function in the preceding section involves Byzantine Agreements for values that are not needed: all the clocks are disseminated, but only the largest $g+1$ are used. (Only the $g+1^{st}$ largest clock is used for resynchronization, but to determine which clock is the $g+1^{st}$ largest, the g largest clocks are needed.) Since Byzantine Agreement protocols can be costly—in both time and number of messages exchanged—avoiding unnecessary Byzantine Agreements is prudent. We therefore propose a somewhat weaker form of agreement to take the place of the Byzantine Agreements used above and use it only for those clocks that are actually needed.

A *Fireworks Agreement* allows a collection of processors each with a value v to *accept* messages with that value at about the same time:

FW: All correct processors accept a message with value v within β real seconds of each other.

The thing being agreed on in a Fireworks Agreement is the real time that a value is accepted; not the value itself. The name Fireworks Agreement is in analogy with a public fireworks display—all participants agree on when the display is over. In a fireworks display, β is non-zero if observers are different distances from the pyrotechnics; in a distributed system, β will be related to the variance in message-delivery times.

In describing a protocol to implement Fireworks Agreement, we assume that it is possible for a correct processor to

A1: authenticate the origin of every message it receives and

A2: to determine whether a message it receives was modified by processors that relay the message.

These assumptions are satisfied if digital signatures are employed by the sender of a message or if there is a direct link between every pair or processors and the simulated authentication technique of [Srikanth & Toueg 84] is used to transmit messages. In either case, faulty processors are unable to masquerade as correct processors by sending messages and are unable to modify messages sent originally by correct processors before retransmitting them.

The following protocol implements a Fireworks Agreement with $\beta = \Gamma_{max} - \Gamma_{min}$ for use in clock synchronization in a system containing virtual clocks satisfying (2.3).[3] The agreement is for a message with value $T+a$, which will be the value virtual clocks have when the protocol terminates and is started by a processor when its clock reaches T, the *a priori* designated time for the next clock synchronization.

(1) When $\hat{c}_p^i(t) = T$, a processor p broadcasts $T+a, p$ to all other processors.

(2) Upon receiving $T+a, q$ directly from a processor q, a processor p relays $T+a, q$ to all processors.

(3) Upon receiving values $T+a, p_1$, ..., $T+a, p_{k+1}$ where $p_i \neq p_j$ for $i \neq j$: If the last message received, $T+a, p_{k+1}$, was received directly from p_{k+1}, then delay Γ_{min} and accept $T+a$. If the last message received, $T+a, p_{k+1}$, was not received directly from p_{k+1}, then accept $T+a$ immediately.

Assumptions A1 and A2 make it impossible for faulty processors to fool correct processors that are trying to determine the origin of a message or whether the message was relayed as required by steps (2) and (3) of the protocol. Steps (1) and (2) of the protocol together ensure that a value received by any correct processor is received by every correct processor. Step (3) ensures that a

[3] Recall, Γ_{min} is the minimum message delivery time and Γ_{max} the maximum message delivery time.

value is accepted by all processors within β real seconds of each other. Moreover, because there are at most k faulty processors, step (3) ensures that a value is accepted only after that value has been received from some correct processor.

When Fireworks Agreement is used in constructing a convergence function CF to implement a reliable time source, the value of CF is the time the $T+a$ message is accepted. Let t_p be the real time the message is accepted by processor p and let t_q be the real time the message is accepted by processor q. Due to FW, evaluations of the convergence function at correct processors p and q can differ by at most $\beta = \Gamma_{max} - \Gamma_{min}$. Thus, setting $\hat{c}_p^{i+1}(t_p) = T+a$ and $\hat{c}_q^{i+1}(t_q) = T+a$ satisfies the Precision Enhancement Property, with $\pi(\delta, \varepsilon) = (1+\hat{\rho})\beta$. Accuracy α of the Accuracy Preservation Property is given by $\alpha(\delta) = (\delta + 2\Gamma_{max})(1+\hat{\rho}) - a$ because in the worst case δ seconds can elapse between when the first correct processor reaches T and broadcasts its message and when the $k+1st$ processor broadcasts its message, followed by an additional $2\Gamma_{max}$ seconds for the protocol to complete.

Clock synchronization algorithms based on Fireworks Agreement are interesting because a processor cannot evaluate CF without causing every other correct processor to resynchronize its clock. Thus, the convergence function provides an implementation of both RTS1 and RTS2; the other convergence functions discussed in this paper provide an implementation of only RTS2.

The first clock synchronization protocol to be based on Fireworks Agreement is discussed in [Halpern *et al.* 84]. A more recent algorithm [Srikanth & Toueg 85] implements virtual clocks with rates much closer to the rate of the hardware clocks on which they are based.

6. Discussion and Conclusions

We have discussed clock synchronization protocols that can be viewed as refinements of a single paradigm. The paradigm is based on postulating a reliable time source that periodically issues messages to cause processors to synchronize their clocks. The reliable time source is implemented by evaluating a convergence function on the values of processor clocks. Thus, if processor clocks run close together but far from real time, clocks implemented by an algorithm based on this paradigm will remain synchronized with each other but will diverge from the real time.

In order to construct a clock synchronization algorithm that keeps clocks close to real time, the reliable time source must remain close to real time. Various international standards organizations maintain highly accurate synchronized clocks. In the United States, WWV radio broadcasts at 60 KHz provide a time signal accurate to a few milliseconds, as does the GEOS satellite.

(WWV broadcasts at 5, 10, and 15 MHz are accurate to only 100 milliseconds, due to uncertainty in propagation delays.) Employing radio receivers to inject such correct real times into a distributed system is one way to provide the needed source of time. Algorithms for clock synchronization when an external source of time is available are described in [Marzullo & Owicki 83], [Marzullo 84], and [Lamport 85].

The fact that so many clock synchronization algorithms can be viewed in terms of a single paradigm came as a bit of a surprise. Previously, clock synchronization algorithms were viewed in terms of three classes: those based on convergence, those based on agreement, and those in the style of [Halpern *et al.* 84]. It was pleasing to discover that all the published algorithms can, in fact, be viewed in terms of a single paradigm based on convergence functions. In addition, viewing algorithms as refinements of a single paradigm allows their performance to be compared. Performance of a clock synchronization algorithm based on convergence functions is characterized by π, α, and the cost of computing the underlying convergence function. Thus, by defining the notion of a convergence function and giving a framework in which its performance can be quantified, we have made it possible to compare existing algorithms as well as given insight into the construction of new algorithms.

Acknowledgments

Discussions with Ozalp Babaoglu, Steve Mahaney, Leslie Lamport, and Sam Toueg have been helpful. In addition, I am grateful to Ozalp Babaoglu, David Gries and Jacob Aizikowitz for useful comments on an early version of this paper. The diagram in the appendix was promptly and expertly prepared by Lori Dyess. The notions of accuracy and precision were developed jointly with Steve Mahaney under a consulting agreement with AT&T Bell Laboratories.

References

[Babaoglu & Drummond 85] Babaoglu, O. and R. Drummond. Streets of Byzantium: Network architectures for fast reliable broadcasts. *IEEE Trans. on Software Engineering SE-11*, 6 (June 1985), 546-554.

[Dolev 82] Dolev, D. The Byzantine Generals strike again. *Journal of Algorithms 3* (1982), 14-30.

[Dolev *et al.* 83] Dolev, D., N.A. Lynch, S.S. Pinter, E.W. Stark, and W.E. Weihl. Reaching approximate agreement in the presence of faults. *Proc. Third Symposium on Reliability in Distributed Software and Database Systems*, Oct. 1983, IEEE Computer Society, 145-154.

[Fisher 83] Fischer, M. The consensus problem in unreliable distributed systems (a brief survey). *Proc. International Conference on Foundations of Computation Theory*, Borgholm, Sweden, August 1983.

[Halpern *et al.* 84] Halpern, J., B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. *Proc. of the*

Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1984, 89-102.

[Lamport 84] Lamport, L. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS 6*, 2 (April 1984), 254-280.

[Lamport 85] Lamport, L. Notes on a time service. Preliminary Report, DECSRC, Palo Alto, CA, Nov. 1985.

[Lamport & Milliar-Smith 84] Lamport, L and P.M. Milliar-Smith. Byzantine clock synchronization. *Proc. of the Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984, 68-74.

[Lamport & Milliar-Smith 85] Lamport, L. and P.M. Milliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM 32*, 1 (Jan. 1985), 52-78.

[Lamport *et al.* 82] Lamport, L., R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS 4*, 3 (July 1982), 382-401.

[Lundelius & Lynch 84] Lundelius, J. and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Proc. of the Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984, 75-88.

[Mahaney & Schneider 85] Mahaney, S.R. and F.B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. *Proc. of the Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, August 1985, 237-249.

[Marzullo & Owicki 83] Marzullo, K. and S.S. Owicki. Maintaining the time in a distributed system. *Proc. of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, August 1983, 295-305.

[Marzullo 84] Marzullo, K. Maintaining the time in a distributed system. An example of a loosely-coupled distributed service. Ph.D. Thesis, Department of Electrical Engineering, Stanford University.

[Mills 85] Mills, D.L. Experiments in network clock synchronization. ARPANet RFC957, Sept 1985.

[Schneider 85] Schneider, F.B. Paradigms for distributed programs. In *Distributed Systems. Methods and Tools for Specification*, M. Paul and H.J. Siegert, eds. Lecture Notes in Computer Science, Vol. 190, Springer-Verlag, Berlin, 1985, 432-443.

[Srikanth & Toueg 84] Srikanth, T.K. and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Technical Report TR 84-623, Department of Computer Science, Cornell University, Ithaca, New York, July 1984.

[Srikanth & Toueg 85] Srikanth, T.K. and S. Toueg. Optimal clock synchronization. *Proc. of the Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, August 1985, 71-86.

Appendix: Resynchronization Interval

The maximum interval that can elapse before starting a new virtual clock depends on the maximum rate at which virtual clocks drift apart, how closely virtual clocks are synchronized, and the precision and accuracy of the convergence function being used. In this appendix, we give the precise relationship between these parameters.

Notice that in the clock synchronization protocol of section 2, $\hat{c}_q^{i+1}$ is computed using virtual clocks $\hat{c}_p^i$ for all processors q. Thus, we require

Concurrent Clocks Property: $\hat{c}_p^i$ must have been started at every processor p if $\hat{c}_q^{i+1}$ has been started at any processor q.

Let RC_{min} be the minimum clock time that can elapse between successive clock resynchronizations by any processor. If virtual clocks are synchronized to within $\hat{\delta}$, then, provided

$$\hat{\delta} < RC_{min}, \tag{7.1}$$

the Concurrent Clocks Property will hold.

From the Concurrent Clocks Property, we conclude that the number of virtual clocks that have been started by each processor can differ at most by 1 at any real time t. This allows synchronization requirement on virtual clocks (2.3) to be reformulated in terms of superscripted virtual clocks. Let t_p^i be the real time that superscripted virtual clock $\hat{c}_p^i$ is started by processor p. Then we have,

$$|\,\hat{c}_q^i(t) - \hat{c}_p^i(t)\,| \;\le\; \hat{\delta} \quad \text{for} \quad \max(t_p^i,\, t_q^i) \;\le\; t \;<\; \max(t_p^{i+1},\, t_q^{i+1}) \tag{7.2}$$

$$|\,\hat{c}_q^{i+1}(t) - \hat{c}_p^i(t)\,| \;\le\; \hat{\delta} \quad \text{for} \quad \max(t_q^{i+1},\, t_p^i) \;\le\; t \;<\; t_p^{i+1} \tag{7.3}$$

Let R_{max} be the maximum real time that can elapse before clock resynchronization is necessary to preserve (7.2) and (7.3). Consider a processor p with a virtual clock implemented by $\hat{c}_p^{i-1}$ that is running (slow) at $1-\hat{\rho}$ clock seconds per second and a processor q with a virtual clock implemented by $\hat{c}_q^{i-1}$ that is running (fast) at $1+\hat{\rho}$ clock seconds per second. (See Figure 1.) Now, suppose p is the last processor to start its i^{th} clock, q is the first, and that at real time t_p^i, when p starts $\hat{c}_p^i$,

$$\hat{c}_q^i(t_p^i) - \hat{c}_p^i(t_p^i) \;=\; \delta_p^i.$$

Due to the definition of R_{max}, p will start $\hat{c}_p^{i+1}$ at real time $t_p^{i+1} \le t_p^i + R_{max}$ and q will start $\hat{c}_q^{i+1}$ at real time

$$t_q^{i+1} \;\le\; t_p^i + \frac{(1-\hat{\rho})R_{max} - \delta_p^i}{1+\hat{\rho}} \tag{7.4}$$

because

Figure 1. Clock Resynchronization Scenario

$$\hat{c}_p^i(t_p^i) + \delta_p^i + (t_q^{i+1} - t_p^i)(1+\hat{\rho}) \;\le\; \hat{c}_p^i(t_p^i + R_{max})$$

$$= \;\hat{c}_p^i(t_p^i) + \delta_p^i + (t_q^{i+1} - t_p^i)(1+\hat{\rho}) \;\le\; \hat{c}_p^i(t_p^i) + R_{max}(1-\hat{\rho})$$

If at time t_q^{i+1} correct virtual clocks (with superscript i) are in a δ_q^{i+1} wide interval, then due to the Accuracy Preservation Property, starting $\hat{c}_q^{i+1}$ results in a virtual clock that can be as much as $\alpha(\delta_q^{i+1} + \Lambda)$ from any correct virtual clock with superscript i, because we (pessimistically) assume that q approximates all clocks high by maximum clock reading error Λ. In the worst case, $\hat{c}_q^{i+1}$ will continue to run as fast as possible, so by t_p^{i+1} it could be as much as $\alpha(\delta_q^{i+1} + \Lambda) + 2\hat{\rho}(t_p^{i+1} - t_q^{i+1})$ away from $\hat{c}_p^i$. Therefore, to satisfy requirement (7.3), we must have

$$\alpha(\delta_q^{i+1} + \Lambda) + 2\hat{\rho}(t_p^{i+1} - t_q^{i+1}) \;\le\; \hat{\delta}. \tag{7.5}$$

And, to satisfy requirement (7.2), we must have

$$\delta_p^i + 2\hat{\rho} R_{max} \leq \hat{\delta}. \tag{7.6}$$

All that remains is to determine δ_p^i and δ_q^{i+1}. Since at real time t_p^i the i^{th} virtual clocks at correct processors are δ_p^i apart, by t_q^{i+1}, they can be as much as $2\hat{\rho}(t_q^{i+1} - t_p^i) + \delta_p^i$ apart. Substituting for t_q^{i+1} based on (7.4), we get

$$\delta_q^{i+1} \leq 2\hat{\rho} \left[\frac{(1-\hat{\rho})R_{max} - \delta_p^i}{1+\hat{\rho}} \right] + \delta_p^i.$$

Finally, δ_p^i is defined inductively as follows. For the first epoch, which starts at real time 0 and implements virtual clocks superscripted by 1, we have $\delta_p^1 \leq \mu$, due to initial value condition (2.1) and the definition of FIX_p^0 in the protocol of section 2. For the $i+1^{st}$ epoch, we have $\delta_p^{i+1} \leq | \hat{c}_q^{i+1}(t_p^{i+1}) - \hat{c}_p^{i+1}(t_p^{i+1}) |$. We now consider two cases, depending on how RTS1 is achieved.

The first case we consider is where processors resynchronize their clocks when the previous superscripted virtual clock reaches some given value. The worst case is when $\hat{c}_q^{i+1}$, once started, continues to run as fast as possible, in which case[4]

$$\hat{c}_q^{i+1}(t_p^{i+1}) = CF(\hat{c}_q^i(t_q^{i+1}), \hat{c}_1^i(t_q^{i+1}) + \lambda_q(1), \ldots, \hat{c}_N^i(t_q^{i+1}) + \lambda_q(N)) + (t_p^{i+1} - t_q^{i+1})(1+\hat{\rho}).$$

By definition,

$$\hat{c}_p^{i+1}(t_p^{i+1}) = CF(\hat{c}_p^i(t_p^{i+1}), \hat{c}_1^i(t_p^{i+1}) + \lambda_p(1), \ldots, \hat{c}_N^i(t_p^{i+1}) + \lambda_p(N))$$

Thus, we have

$$\begin{aligned}
\delta_p^{i+1} &= | CF(\hat{c}_q^i(t_q^{i+1}), \hat{c}_1^i(t_q^{i+1}) + \lambda_q(1), \ldots, \hat{c}_N^i(t_q^{i+1}) + \lambda_q(N)) + (t_p^{i+1} - t_q^{i+1})(1+\hat{\rho}) \\
&\quad - CF(\hat{c}_p^i(t_p^{i+1}), \hat{c}_1^i(t_p^{i+1}) + \lambda_p(1), \ldots, \hat{c}_N^i(t_p^{i+1}) + \lambda_p(N)) | \\
&= | CF(\hat{c}_q^i(t_q^{i+1}) + (t_p^{i+1} - t_q^{i+1})(1+\hat{\rho}), \hat{c}_1^i(t_q^{i+1}) + \lambda_q(1) + (t_p^{i+1} - t_q^{i+1})(1+\hat{\rho}), \ldots, \\
&\qquad \hat{c}_N^i(t_q^{i+1}) + \lambda_q(N) + (t_p^{i+1} - t_q^{i+1})(1+\hat{\rho})) \\
&\quad - CF(\hat{c}_p^i(t_p^{i+1}), \hat{c}_1^i(t_p^{i+1}) + \lambda_p(1), \ldots, \hat{c}_N^i(t_p^{i+1}) + \lambda_p(N)) |
\end{aligned}$$

due to translation invariance. Since the clock reading error for correct processors is bounded by Λ, the value of each argument (in the second evaluation of CF) $\hat{c}_a^i(t_p^{i+1}) + \lambda_p(a)$ for any correct processor a satisfies the following inequality:

[4]Recall, $\lambda_q(v)$ is the error associated with processor q reading the clock at processor v.

$$\hat{c}_a^i(t_q^{i+1})+(t_p^{i+1}-t_q^{i+1})(1-\hat{\rho}) \;\le\; \hat{c}_a^i(t_p^{i+1})+\lambda_p(a) \;\le\; \hat{c}_a^i(t_q^{i+1})+(t_p^{i+1}-t_q^{i+1})(1+\hat{\rho})+\Lambda$$

Thus, the difference between an argument in the second evaluation of CF and the corresponding argument $\hat{c}_a^i(t_q^{i+1})+\lambda_q(j)+(t_p^{i+1}-t_q^{i+1})(1+\hat{\rho})$ in the first evaluation of CF is bounded by $2\hat{\rho}(t_p^{i+1}-t_q^{i+1})+\Lambda$. Provided CF has sufficient fault tolerance degree to cope with faulty processors, we can use the Precision Enhancement Property of CF with $\delta=\hat{\delta}$ due to (7.5) and $\varepsilon=2\hat{\rho}(t_p^{i+1}-t_q^{i+1})+\Lambda$ to conclude that

$$\delta_p^{i+1} \;\le\; \pi(\hat{\delta},\, 2\hat{\rho}(t_p^{i+1}-t_q^{i+1})+\Lambda).$$

The second case we consider is where all processors resynchronize their clocks within β real seconds and all start their new clocks at a given value $T+a$. This case corresponds to the use of a Fireworks Agreement and is much simpler than the previous one. By definition, $|\,t_p^{i+1}-t_q^{i+1}\,|=\beta$. Because q can run $\hat{c}_q^{i+1}$ as long as β seconds before the new clock at p $\hat{c}_p^{i+1}$ is started, $\hat{c}_q^{i+1}(t_p^{i+1})$ can be as large as $T+a+(1+\hat{\rho})\beta$. Thus, we have $\delta_p^{i+1}\le(1+\hat{\rho})\beta$ because both $\hat{c}_p^{i+1}$ and $\hat{c}_q^{i+1}$ start with value $T+a$.

Putting this all together, the interval R in real seconds between clock resynchronizations must satisfy $R\le R_{max}$ where R_{max} satisfies (7.5) and (7.6). Since virtual clocks do not necessarily run at 1 clock second per second, the resynchronization interval RI in clock seconds used by every processor must satisfy $RI/(1+\hat{\rho})\le R_{max}$ so that the fastest processor does not exceed the R_{max} bound. Combining this with the lower bound for RI given by (7.1), we get

$$\hat{\delta} \;<\; \frac{RI}{(1+\hat{\rho})} \;<\; R_{max}. \tag{7.7}$$

Virtual Clock Rates

Simply setting a clock ahead or back in order to maintain synchronization with other clocks can cause problems. In real-time process-control applications, tasks are broken into small computations and scheduled based on clock readings to ensure that real-time deadlines can be met. If a clock synchronization protocol suddenly sets a clock forward, the processor might not be able to handle all the tasks that have become due. In other applications, clock times are used to infer possible causality between events. For example, creation times for files are usually taken to define the order in which the files were created. Suddenly setting a clock back can destroy the consistency of time with potential causality. Finally, when clocks are used to obtain performance measurements, a sudden shift in the clock value can introduce errors by the amount of the shift.

For these reasons, a clock synchronization protocol must satisfy a rate restriction like (2.4), which prevents the value of the clock from changing by too large an amount over too short an interval. One way to satisfy (2.4) is to include as part of a time value the superscript of the virtual clock that furnished that value and choose $\hat{\rho}$ such that $\rho \le \hat{\rho}$. According to (2.2), clocks at correct processors run at a rate between $1-\rho$ and $1+\rho$. Thus, clock values with the same superscript can be compared and manipulated because they were obtained from a set of clocks satisfying (2.4). Clock values with different superscripts, however, do not have this property. These values are incomparable because of the discontinuity when a new virtual clock is started. This is an obvious limitation of the scheme, since time values that are far apart are likely to have come from virtual clocks with different superscripts.

A second way to satisfy (2.4) is by evenly spreading any change between FIX_p^{i-1} and FIX_p^i over the entire i^{th} epoch. Instead of making an instantaneous shift in the value of $\hat{c}_p$ when $\hat{c}_p^i$ is started, the clock drift rate is modified to compensate for the change. According to (7.7), an epoch lasts at most RI clock seconds. Thus, we implement $\hat{c}_p^i$ by incrementing $\hat{c}_p^{i-1}$ by $tick_p^i$ whenever c_p is incremented.

$$tick_p^i \equiv 1 + \frac{(FIX_p^i - FIX_p^{i-1})}{RI}$$

The drift of $\hat{c}_p$ due to this compensation can be computed as follows. According to the Accuracy Preservation Property, a clock value can be shifted by at most $\alpha(\delta)$ when it is resynchronized provided correct processors lie within an interval of width δ. Since (2.3) ensures that any two correct clocks are within $\hat{\delta}$, we conclude that a correct clock at processor p can be shifted by at most $\alpha(\hat{\delta})$, and therefore

$$0 < \; | \, tick_p^i \, | \; \le \frac{\alpha(\hat{\delta})}{RI}$$

According to (2.2), the rate of a correct processor (hardware) clock is between $1-\rho$ and $1+\rho$. Adding the compensation due to $tick_p^i$, we find that the rate of $\hat{c}_p$ must be between $1-\rho-\frac{\alpha(\hat{\delta})}{RI}$ and $1+\rho+\frac{\alpha(\hat{\delta})}{RI}$. Thus, if $\hat{\rho}$ satisfies

$$1-\hat{\rho} \;\le\; 1-\rho-\frac{\alpha(\hat{\delta})}{RI} \;\le\; 1+\rho+\frac{\alpha(\hat{\delta})}{RI} \;\le\; 1+\hat{\rho}$$

then (2.4) will hold.