Understanding Protocols for Byzantine Clock Synchronization[*]

Fred B. Schneider

87-859

August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

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

Understanding Protocols for Byzantine Clock Synchronization[*]

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

August 24, 1987

ABSTRACT

All published fault-tolerant clock synchronization protocols are shown to result from refining a single paradigm. This allows the different clock synchronization protocols to be compared and permits presentation of a single correctness analysis that holds for all. The paradigm is based on a reliable time source that periodically causes events; detection of such an event causes a processor to reset its clock. In a distributed system, the reliable time source can be approximated by combining the values of processor clocks using a generalization of a "fault-tolerant average", called a convergence function. The performance of a clock synchronization protocol based on our 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 that synchronized clocks be available to 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]) involve delays that are proportional to how closely clocks at participating sites are synchronized. And, real-time process control systems require that accurate timestamps be assigned to sensor values so that these values can be correctly interpreted.

Other applications further require that clocks advance at approximately the same rate as real time. To ensure that deadlines can be met in real-time process-control applications, tasks are usually broken into small computations and scheduled based on the processor clock. If a clock synchronization protocol suddenly sets that clock forward, thereby momentarily increasing its rate, the processor might not be able to handle in a timely manner all the tasks that become due. Also, clocks are sometimes used to assign timestamps to events so that it is possible to infer potential causality between events. For example, creation times of files are usually taken to define the order in which those files were created. A clock synchronization protocol that suddenly sets a clock back could destroy the consistency of time with respect to potential causality.

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 clocks periodically adjusted. 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 gives a single paradigm and correctness proof that can be used to understand all published[1] fault-tolerant protocols for keeping clocks in a distributed system synchronized despite faulty processors that can exhibit arbitrary behavior. The paradigm allows us to identify the different implementation choices made by each protocol in solving three subproblems it defines. This permits the

[1] E.g., [Babaoglu & Drummond 87], [Cristian *et al.* 86], [Halpern *et al.* 84], [Kopetz & Ochsenreiter 87], [Lamport & Melliar-Smith 84], [Lamport & Melliar-Smith 85], [Lundelius & Lynch 84], [Mahaney & Schneider 85], and [Srikanth & Toueg 85].

various Byzantine clock synchronization protocols[2] to be compared and the contributions of each to be isolated. Previously, clock synchronization algorithms were viewed in terms of three distinct classes: those based on convergence, those based on agreement, and those based on diffusion or flooding of messages. Our proof is interesting because it necessarily generalizes all of the correctness proofs that have appeared for the individual clock synchronization algorithms. Also, it is the first proof in which clocks are treated as advancing at discrete times ("ticks"). Previous proofs modeled clocks as monotonically increasing functions form real time to clock time.

The remainder of the paper is organized as follows. Our clock synchronization paradigm is described in section 2. Techniques for reading clocks across a computer-communications network are described in section 3. In section 4, we discuss properties of convergence functions, the central component of the paradigm, and give some examples of convergence functions. Section 5 discusses how agreement protocols can be used in implementing a convergence function. Conclusions and related work appear in section 6. Appendix 1 analyzes the performance of clock synchronization protocols derived from our paradigm and derives bounds for various parameters of such a protocol; appendix 2 contains a glossary of the notation used in the paper.

2. A Paradigm for Clock Synchronization

The hardware clock at a correct processor p can be viewed as implementing a function c_p. This function maps a real time t to a clock time $c_p(t)$, is non-decreasing in its argument, and is characterized by positive constants μ, ρ, and κ. Constant μ defines the range of initial values of the clock:

Hardware Initial Value: $0 \leq c_p(0) \leq \mu$. (2.1)

Constants κ and ρ restrict the rate that clock time increases as a function of real time. Physical clocks are counters that increase by 1 in response to periodically generated events called *ticks*. In a physical clock, the (real time) interval between ticks can vary and this can cause the clock value to advance at a different rate than real time. For our purposes, it is more convenient to model a hardware clock as having a fixed (real time) interval κ between ticks, but advancing by a varying real number value v, where $(1-\rho)\kappa \leq v \leq (1+\rho)\kappa$, in response to each tick. That is, we require

Hardware Rate: $$0 < 1-\rho \leq \frac{c_p(t+\kappa)-c_p(t)}{\kappa} \leq 1+\rho \quad \text{for } 0 \leq t, \qquad (2.2)$$

where κ is called the *tick width* and ρ the *drift rate* of the clock. Notice that (2.2) does not require the value of c_p to remain fixed between successive ticks although for most clocks this will be the case;

[2] In the distributed computing literature, arbitrary behavior in response to a failure is called *Byzantine* behavior. Clock synchronization protocols that can tolerate such failures are called Byzantine clock synchronization protocols.

Hardware Rate (2.2) merely ensures that rate at which the clock advances is within ρ of the rate at which real time passes.

We make no assumptions about the behavior of clocks at faulty processors—not even that they can be modeled by functions. A clock at 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. A virtual clock, like a hardware clock, is a function that maps real time t to clock time $\hat{c}_p(t)$, is non-decreasing in its argument, and is characterized by positive constants $\hat{\mu}$, $\hat{\rho}$, and $\hat{\kappa}$ such that for correct processors p and q

Virtual Synchronization: $$|\hat{c}_q(t)-\hat{c}_p(t)| \le \hat{\delta} \quad \text{for } 0 \le t, \tag{2.3}$$

Virtual Rate: $$0 < 1-\hat{\rho} \le \frac{\hat{c}_p(t+\hat{\kappa})-\hat{c}_p(t)}{\hat{\kappa}} \le 1+\hat{\rho} \quad \text{for } 0 \le t. \tag{2.4}$$

$\hat{\delta}$ characterizes how closely virtual clocks are synchronized with each other; $\hat{\rho}$ is the drift rate of virtual clocks; and $\hat{\kappa}$ specifies the (real-time) interval between virtual-clock ticks.

If a reliable time source is available, then satisfying (2.3) and (2.4) is simple. The reliable time source periodically distributes the correct time to all processors and, upon receipt of this correct time, a processor adjusts its virtual clock accordingly. Provided the time is distributed frequently enough, processor clocks will not drift too far apart in the interval between adjustments, so (2.3) will be maintained. And, provided no processor has to adjust its clock by too much, the adjustment can be spread over the interval that precedes the next resynchronization 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 just outlined. First, it periodically generates an event that when detected by a correct processor causes that processor to resynchronize its clock. This can be formalized in terms of constants r_{min}, r_{max}, and β as:

RTS1: A reliable time source generates a sequence of events at real times t^1_{RTS}, t^2_{RTS} ... such that

$$t^1_{RTS} = 0 \ \wedge \ (\forall\, i\colon\ 0<i\colon\ r_{min} \le t^{i+1}_{RTS} - t^i_{RTS} \le r_{max})$$

and the real time t^i_p at which a processor p detects the event produced at t^i_{RTS} satisfies

$$t^1_p = 0 \ \wedge \ (\forall\, i\colon\ 1 \le i\colon\ 0 \le t^i_p - t^i_{RTS} \le \beta).$$

(Choosing $t_{RTS}^1 = t_p^1 = 0$ models the fact that the protocol and clocks start at real time 0).

Second, in addition to causing events, the reliable time source facilitates clock synchronization by providing a value to each correct processor.

RTS2: At t_p^i, processor p obtains a value V_p^i that can be used in adjusting $\hat{c}_p$ to be consistent with (2.3) and (2.4).

There are two things to note about RTS1 and RTS2. First, these properties do not imply that the correct time is always available to processors—only that it is available periodically. Having a reliable time source from which the correct time is always available results in a different clock synchronization paradigm than the one just described.[3] Second, RTS2 does not stipulate that the same value be obtained by every processor—only that the value provided can be used to achieve (2.3) and (2.4). This permits values obtained by different processors to be compensated for known delays due to protocol execution and message delivery.

Although it is easy to satisfy RTS1 and RTS2 using a single clock, the resulting time source is only as fault tolerant as that clock is. A reliable time source that does not depend on correct operation of a single clock can be constructed by using approximately synchronized clocks in a distributed system. RTS1 is achieved by using the individual processor clocks to signal the periodic resynchronization events; RTS2 is achieved by having processors compute some type of fault-tolerant average of the values of the clocks at processors in the system.

To describe the implementation of RTS1 and RTS2 in a distributed system, it will be convenient to view resetting a virtual clock $\hat{c}_p$ as starting another virtual clock that runs concurrently with the old one. Thus, initially (i.e. at real time 0) p uses virtual clock $\hat{c}_p^1$ and p starts a new virtual clock $\hat{c}_p^{i+1}$ at real time t_p^{i+1}, when it detects the resynchronization event produced at time t_{RTS}^{i+1}. Using this convention, the value of $\hat{c}_p(t)$ is characterized by

$$t_p^i \le t < t_p^{i+1} \;\Rightarrow\; \hat{c}_p(t) = \hat{c}_p^i(t).$$

A virtual clock $\hat{c}_p^i$ is implemented at processor p using the hardware clock c_p at processor p and adding an adjustment value that is maintained by the clock synchronization protocol. Formally, this is given by

$$\hat{c}_p^i(t) \;\equiv\; c_p(t) + FIX_p^i(c_p(t)) \qquad (2.5)$$

[3] While we have been able to design clock synchronization protocols based on this other paradigm, we have so far been unable to develop a generic proof of correctness for them.

where $FIX_p^i(T)$ is a function from clock time at processor p to a correction for hardware clock c_p.[4] Thus, p reads $\hat{c}_p^i$ at real time t by reading c_p and then adding the appropriate adjustment value based on the current value of FIX_p^i.

So that a virtual clock does not violate Virtual Rate (2.4), the value of FIX_p^i must change gradually as a function of time. Therefore, $FIX_p^i(T)$ spreads any change in its correction to c_p over *adjustment interval AI* clock seconds, a parameter of the protocol. The following definition of $FIX_p^i(T)$ achieves this. In it, adj_p^{i-1} is the adjustment to c_p necessary to implement $\hat{c}_p^{i-1}$ and adj_p^i is the adjustment to implement $\hat{c}_p^i$, so $adj_p^i - adj_p^{i-1}$ is the additional amount that FIX_p^i must add to c_p over the AI clock-second interval starting from t_p^i in order to approach $\hat{c}_p^i$:

$$FIX_p^i(T) \equiv adj_p^{i-1} + \frac{(adj_p^i - adj_p^{i-1})(\min(T - c_p(t_p^i), AI))}{AI}.$$

The key to this definition is that $0 \le \min(T - c_p(t_p^i), AI) \le AI$, so that the change from adj_p^{i-1} to adj_p^i is gradually spread out over AI clock seconds.

For FIX_p^i to work, AI must be long enough to avoid violating the drift rate bounds for $\hat{c}_p$ in Virtual Rate (2.4); however, it must not be too long or else Virtual Synchronization (2.3) could be violated or the next superscripted clock might be started before the full adjustment has been completed, leaving an even larger adjustment to be performed. The case where $AI \le \hat{\kappa}$ is called *instantaneous resynchronization*; otherwise *continuous resynchronization* occurs. Appendix 1 characterizes values for AI and other parameters of our paradigm that ensure Virtual Synchronization (2.3) and Virtual Rate (2.4) hold.

To implement RTS2, we use a function CF that essentially averages the values of the approximately synchronized clocks at correct processors in the system. In a system of N processors, V_p^{i+1} of RTS2 is defined by[5]

$$V_p^{i+1} = CF(p, \hat{c}_1^i(t_p^{i+1}), \ldots, \hat{c}_N^i(t_p^{i+1})),$$

where CF is called a *convergence function* because it brings clocks closer together. Given this definition of V_p^{i+1},

$$adj_p^{i+1} = CF(p, \hat{c}_1^i(t_p^{i+1}), \ldots, \hat{c}_N^i(t_p^{i+1})) - c_p(t_p^{i+1}) \qquad (2.6)$$

is the amount that $c_p(t_p^{i+1})$ differs from $\hat{c}_p(t_p^{i+1})$ and we can now give the clock synchronization

[4] In this paper, clock times are denoted by upper-case letters and real times by lower-case letters.

[5] Evaluating $CF(p, \hat{c}_1^i(t_p^{i+1}), \ldots, \hat{c}_N^i(t_p^{i+1}))$ seemingly requires that p be able to read at the same instant all the virtual clocks maintained by other processors. Section 3 explains how to get around this problem.

protocol for a processor p in a distributed system consisting of N processors. It appears in Figure 2.1.[6] Three important things about that protocol are unspecified. They are

- the implementation of "detect event generated at time t_{RTS}^{i+1}",
- how one processor reads the virtual clocks at other processors, and
- convergence function CF.

Different choices for these result in different clock synchronization protocols. In fact, the various choices permit viewing in terms of our paradigm all the published clock synchronization protocols that do not make use of an external time source.

Much of this paper is, therefore, devoted to different implementation choices for the unspecified aspects of Figure 2.1. The remainder of this section discusses different implementations of "detect event generated at time t_{RTS}^{i+1}". Section 3 discusses methods ways that one processor can read the virtual clocks at other processors. And, sections 4 and 5 give properties and examples of convergence functions.

```
$i := 1$;
$adj_p^0 := 0$;  $adj_p^1 := 0$;
**do forever**
    detect event generated at time $t_{RTS}^{i+1}$;
    $t_p^{i+1}$ := real time now;
    $adj_p^{i+1} := CF(p, \hat{c}_1^i(t_p^{i+1}), \ldots, \hat{c}_N^i(t_p^{i+1})) - c_p(t_p^{i+1})$;
    $i := i+1$
**od**
```

Figure 2.1. Clock synchronization protocol

[6] There and throughout, we use the notational device "t := real time now" as a way to talk about the value of a clock during execution. Variable t is not actually implemented and is not directly accessible to the program, although $c_p(t)$ is.

Detecting Resynchronization Events

An obvious approach to implementing "detect event generated at time t_{RTS}^{i+1}" uses the approximately synchronized virtual clocks. For some predefined value R, each processor p waits until $\hat{c}_p^{\,i}$ reads iR before starting $\hat{c}_p^{\,i+1}$. Either an interval timer or busy-waiting can be employed to implement this waiting.

In this scheme, t_{RTS}^{i} is the earliest real time that some correct processor's virtual clock has value iR. Since virtual clocks at correct processors can advance as quickly as $1+\hat{\rho}$ clock seconds per real second, $r_{min}=R/(1+\hat{\rho})$; and since they can advance as slowly as $1-\hat{\rho}$ clock seconds per real second, $r_{max}=R/(1-\hat{\rho})$. To compute β, note that at the time the fastest correct clock reads iR, due to (2.3) the slowest correct clock must read at least $iR-\hat{\delta}$. Thus, this (slow) correct clock might take as long as $\hat{\delta}/(1-\hat{\rho})$ real seconds until it reaches iR; so, $\beta=\hat{\delta}/(1-\hat{\rho})$.

Another implementation of "detect event generated at time t_{RTS}^{i+1}" is for each processor to broadcast a message when its virtual clock reaches some predefined value and to resynchronize when such a message has been received from a correct processor. Here, β is bounded by the variance in the (real-time) delay of performing the broadcast. The details of this scheme, which is based on a simple form of agreement, are given in section 5.

3. Reading Clocks from Afar

Processors have access to clock time, not real time. This means that in order for a processor p to obtain the arguments to CF needed to compute adj_p^{i+1} (see (2.6)), p must obtain $c_p(t_p^{i+1})$, $\hat{c}_1(t_p^{i+1})$, ..., $\hat{c}_N(t_p^{i+1})$, which requires that it read N clocks simultaneously. This is impossible for two reasons. First, without special hardware a processor can read only one clock at a time. Second, in a distributed system, processors do not necessarily have access to each others' clocks.

One solution to both of these problems is for each processor locally to implement approximations of the virtual clocks at other processors. Processor p maintains a collection of tables $\tau_p^i[1..N]$ that can be used to compute an approximation for $\hat{c}_q^{\,i}(t)$, and p approximates $\hat{c}_q^{\,i}(t)$ at real time t by $c_p(t)+\tau_p^i[q]$. Thus, p can approximate $\hat{c}_1(t_p^{i+1})$, ..., $\hat{c}_N(t_p^{i+1})$, simply by reading $c_p(t_p^{i+1})$ once and using it and τ_p^i to compute the N values needed.

In one technique to construct τ_p^i, first described in [Lamport & Melliar-Smith 84], processor p periodically communicates with the other processors in the system. Suppose the minimum and maximum delays (according to the clock at any correct processor) incurred in sending a message from one correct processor to another, receiving it, and processing it, are Γ_{min} and Γ_{max}. A processor p can compute $\tau_p^i[q]$ by executing

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

Processor q responds to a "i^{th} clock time?" request from p by sending back $\hat{c}_q(t_{rply})$, where t_{rply} is the real time the reply is sent.

Define *clock reading error* $\lambda_p^i(q)$ to be the error in p's approximation of $\hat{c}_q^{\,i}$. Let Λ be the maximum clock reading error for any pair of correct processors. That is,

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

In order to bound Λ, first note that p's approximation of q's clock can drift away from q's clock by at most $\hat{\rho}+\rho$ clock seconds per real second because the rate error of $\hat{c}_q$ is bounded by $\hat{\rho}$ and the rate error of c_p is bounded by ρ. 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 time request is accounted for in the calculation of $\tau_p^i[q]$. Thus, at (real) time t, $\lambda_p^i(q)$ satisfies

$$\lambda_p^i(q)\ \le\ \Gamma_{max}-\Gamma_{min}+(\rho+\hat{\rho})(t-lread_p(q))\ \le\ \Lambda \tag{3.1}$$

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 $\lambda_p^i(q)$ is a function of t, an upper bound on $t-lread_p(q)$ is usually known, and therefore Λ can be treated as a constant.

Error $\lambda_p^i(q)$ 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 adj_p^{i+1}, because this minimizes the clock reading error just before the clock values are actually needed. However, for reasonable intervals $t-lread_p(q)$, $(\hat{\rho}+\rho)(t-lread_p(q))\ll\Gamma_{max}-\Gamma_{min}$, so minimizing the uncertainty in the network delay is the key to reducing $\lambda_p^i(q)$. Uncertainty in network delay can be reduced by installing the clock reading protocol in the lowest level of the operating system. This is because a large part of the uncertainty in network delay can be attributed to uncertainty in program execution time due to interrupts and other forms of multiprogramming. The time it takes a message to traverse a wire connecting computers does not have a high variance. Even when messages are routed through intermediate sites, delays due to queuing in sites doing relaying can be measured and recorded in the message and therefore can be accounted for.

A variation on the clock reading scheme just given, used in the clock synchronization protocols of [Babaoglu & Drummond 87], [Cristian *et al.* 86], [Halpern *et al.* 84], [Lundelius & Lynch 84], and [Srikanth & Toueg 85], reduces the number of messages by half but can increase clock reading error.

Instead of requesting the time, each processor q periodically broadcasts its virtual clock value (including superscript i). Upon receipt of such a message, the receiver p updates $\tau_p^i[q]$ as follows.

```
receive $C$ from $q$
$t_{now}$ := real time now;
$\tau_p^i[q] := C - (c_p(t_{now}) - \Gamma_{min})$
```

The reduction in number of messages sent using this scheme is due to lack of explicit request messages—the passage of time, rather than an explicit request message, causes transmission of a clock value. However, in a point-to-point network, clock reading errors can increase when this scheme is used. This increase is because a processor p does not necessarily know what communications line it should monitor for the next clock message it will receive. Polling communications lines—even when done by processor microcode—increases Γ_{max}, since it is possible for a message to remain queued at the receiver for an entire polling cycle. Since polling does not increase Γ_{min}, the effect is to increase $\Gamma_{max} - \Gamma_{min}$, which, according to (3.1), increases $\lambda_p^i(q)$. Local area networks, which usually have a single connection between the processor and network, do not have this problem.

4. Convergence Functions

A convergence function CF for use in a system of N processors is a function of $N+1$ arguments that satisfies certain properties. The first argument identifies the processor evaluating CF; each of the following arguments x_q, $1 \le q \le N$, is a value from processor q. The properties required of convergence functions are given below. These properties are used in the proofs of Virtual Synchronization (2.3) and Virtual Rate (2.4) given in Appendix 1. Thus, this abstract characterization of convergence functions is what permits the single set of proofs of Appendix 1 to apply to a collection of clock synchronization protocols.

The first property required for a function CF to be a convergence function is that it be monotonically non-decreasing in its last N arguments.

Monotonicity: If $(\forall i\colon\ 1 \le i \le N\colon\ x_i \le y_i)$ then $CF(p, x_1, \ldots, x_N) \le CF(p, y_1, \ldots, y_N)$.

When CF is used for clock synchronization, arguments x_1 through x_N are time values, and this property states that the value of the Reliable Time Source does not decrease as time passes.

The next property asserts that the relative magnitudes of the virtual clock values—and not their absolute values—matter when they are combined to produce the value provided to p for RTS2. Thus, CF satisfies

Translation Invariance: $CF(p, x_1+v, \ldots, x_N+v) = CF(p, x_1, \ldots, x_N)+v$ for $0 \le v$.

This property allows values of CF computed by different processors at different times to be compared. If in the evaluation of CF by one processor, the values of arguments x_1 through x_N are shifted by the same amount (reflecting the passage of time) from the values used by the other, then the result computed by the first will be shifted by that amount from the result computed by the second.

Third, we require that the values of CF for two different processors p and q using similar values for at least $N-k$ corresponding arguments be closer than x_p and x_q were. This is the reason CF is called a "convergence function". The utility of a convergence function in this regard is characterized by a constant k called the *fault-tolerance degree* and a function π called the *precision*.[7] Fault-tolerance degree specifies the number of argument values that can differ significantly in the evaluation of CF by p and the evaluation of CF by q without greatly affecting the difference in the results; precision specifies how close together values obtained by these two evaluations must be. This is formalized by the

Precision Enhancement Property: $| CF(p, x_1, ..., x_N) - CF(q, y_1, ..., y_N) | \le \pi(\delta, \varepsilon)$ if

(a) at least $N-k$ of the x_i's are within δ of each other,

(b) the y_i's corresponding to those $N-k$ x_i's are within δ of each other, and

(c) for each of the $N-k$ argument pairs, $|y_i - x_i| \le \varepsilon$.

Conditions (a) and (b) define δ to be the width of the interval spanned by values from correct processors; when using CF to implement a reliable time source, this condition is satisfied if virtual clocks at correct processors 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, for small values of v, 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 must produce values that are close—at most $\pi(\delta, \varepsilon)$ apart—provided correct values are within δ, even though the values used for k of the arguments (presumably, from faulty processors) differ arbitrarily and each remaining pair of corresponding arguments differs by at most ε. Provided $\pi(\delta, \varepsilon) < \delta$, CF implements a time source that furnishes different processors with time values that are

[7] Our use of the term precision is based on its usual definition in connection with data and error analysis in the physical sciences [Bevington 69]. There, "precision" is a measure of how exactly a result is determined and, therefore, how reproducible that result is. When used in this sense, precision asserts nothing about whether the result is close to the quantity actually being measured—just that it is close to other results that measure that quantity. The term "accuracy" is reserved for characterizing how close a result is to the true value it measures.

closer than the least synchronized virtual clocks at correct processors.

The final property of a convergence function CF asserts that $CF(p, x_1, ..., x_N)$ is not more than $\alpha(\delta)$ away from any correct argument, where any argument found within a δ width interval containing $N-k$ or more arguments is considered correct.

Accuracy Preservation Property: Let X_{OK} be a subset of $x_1, ..., x_N$ whose members are within δ of $N-k-1$ of $x_1, ..., x_N$. Then,

$$(\forall p: x_p \in X_{OK}: \mid x_p - CF(p, x_1, ..., x_N) \mid \leq \alpha(\delta)).$$

An obvious consequence of this definition is

$$\alpha(\delta) \leq \delta. \qquad (4.1)$$

When CF is used as a reliable time source and correct clocks are synchronized to within δ, $\alpha(\delta)$ bounds the maximum amount by which virtual clock at a processor p must be adjusted. That is, for all correct processors p:

$$(\forall i: 0<i: \mid adj_p^{i+1} - adj_p^i \mid \leq \alpha(\delta)). \qquad (4.2)$$

Function α is called the *accuracy* of CF. This (in the sense of [Bevington 69]) is an apt name for two reasons. First, α bounds the rate change made to a virtual clock $\hat{c}_p$ (through FIX_p), thereby bounding the "accuracy" of the rate of that virtual clock. Second, insofar as the clock at any correct processor q approximates the real time and is therefore considered the true value of interest, α bounds the difference between the value of a newly reset virtual clock and that true value.

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

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

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

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

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

The fault-tolerance degree k, precision $\pi(\delta, \varepsilon)$ when there are f faulty processors, precision when $f=k$ as N goes to infinity, and accuracy $\alpha(\delta)$ for each of the above functions is given in Figure 4.1. The

Name	Fault-tolerance degree k	Precision $\pi(\delta, \varepsilon)$ (f faults)	Worst Precision (i.e. $N \rightarrow \infty$ $f=k$)	Accuracy $\alpha(\delta)$
CF_{EA}	$\frac{N-1}{3}$	$\frac{3f\delta}{N}+\varepsilon$	$\delta+\varepsilon$	$\frac{4\delta}{3}$
CF_{FCA}	$\frac{N-1}{3}$	$\frac{2f\delta}{N}+\varepsilon$	$\frac{2\delta}{3}+\varepsilon$	$\frac{4\delta}{3}$
CF_{Mid}	$\frac{N-1}{3}$	$\frac{\delta}{2}+\varepsilon$	$\frac{\delta}{2}+\varepsilon$	δ
CF_{Avg}	$\frac{N-1}{3}$	$\frac{f\delta}{N-2k}+\varepsilon$	$\delta+\varepsilon$	δ
CF_{CCA}	$\frac{N-1}{3}$	$\frac{f\delta}{N}+\varepsilon$	$\frac{\delta}{3}+\varepsilon$	$\frac{4\delta}{3}$
CF_{Byz}[8]	$\frac{N-1}{2}$	2Λ	2Λ	δ
CF_{FW} SE1[8]	$N-1$	$\frac{\Gamma_{max}(1+\hat{\rho})}{(1-\hat{\rho})}$	$\frac{\Gamma_{max}(1+\hat{\rho})}{(1-\hat{\rho})}$	$\Gamma_{max}+2(\delta-\Gamma_{min})$
CF_{FW} SE2[8]	$\frac{N-1}{2}$	$\frac{\Gamma_{max}(1+\hat{\rho})}{(1-\hat{\rho})}$	$\frac{\Gamma_{max}(1+\hat{\rho})}{(1-\hat{\rho})}$	$\Gamma_{max}+\delta-\Gamma_{min}$

Figure 4.1. Properties of Convergence Functions

other Convergence Functions mentioned in the figure are discussed in section 5. CF_{EA} was first presented and analyzed in [Lamport & Melliar-Smith 85] in connection with their *interactive*

[8] Assumes digital signatures.

convergence clock synchronization algorithm. CF_{FCA} was proposed in [Mahaney & Schneider 85]. CF_{Mid} and CF_{Avg} are given in [Dolev *et al.* 83]; CF_{Avg} is the basis for the clock synchronization protocol of [Lundelius & Lynch 84] and the (AMI S65C60) VLSI clock synchronization chip described by [Kopetz & Ochsenreiter 87]. Characterizing convergence functions in terms of precision and accuracy was first done by [Mahaney & Schneider 85]; most of the precision and accuracy functions given in Figure 4.1 were first reported there.

5. Using Agreement for Convergence

An *agreement protocol* allows correct processors in a distributed system to agree on an action or a set of values. This can help in two ways when implementing a Reliable Time Source. First, use of an agreement protocol to disseminate a signal that causes processors to resynchronize clocks can be used to satisfy RTS1. Second, use of an agreement protocol to disseminate each processor's clock can ensure that arguments in corresponding positions in evaluations of *CF* performed by different processors are equal, thereby enhancing the precision of *CF* and helping to satisfy RTS2.

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.[9] 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.[10]

Byzantine Agreement [Lamport *et al.* 82] is stronger (but more expensive to achieve) 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.

[9] If the transmitter is correct then the set of correct processors that "know" that the transmitter is faulty will be empty.

[10] 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.

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].

5.1. Agreement with Clocks

Protocols to implement Crusader's Agreement and Byzantine Agreement usually proceed as a series of rounds. In the first round, the transmitter sends its value to every other processor. In subsequent rounds, each processor sends a copy of every value it has received to every other processor. Eventually, each processor selects one from among the set of values it has received. The criteria for selection depend on the protocol—use of median or mode is not unusual. Relaying messages through different paths, although seemingly inefficient, is necessary because it prevents correct processors from being confounded by inconsistent values sent along different routes by faulty processors.

An agreement protocol intended for disseminating values must be modified for use in disseminating clocks. This is because, while operations like making copies of *values* and sending such copies through a network are simple, making copies of *clocks* and sending them through a network is not. The key to avoiding this problem is to compute and send clock differences rather than the clocks themselves [Lamport & Melliar-Smith 84].

To implement this scheme, $\hat{c}_x^i$ is encoded as a triple $\langle proc, i, offset \rangle$ that specifies $\hat{c}_x^i$ has difference *offset* from $\hat{c}_{proc}^i$. Thus, $\hat{c}_x^i(t)$ can be approximated by a processor p as $c_p(t)+\tau_p^i[proc]+offset$. This allows p to copy and send $\hat{c}_x^i$ to another processor q by executing

send $\langle proc, i, offset \rangle$ **to** q.

Processor q receives this copy by executing

receive $\langle proc', e, offset' \rangle$

and thereafter approximates $\hat{c}_x^i$ at time t by evaluating $c_q(t)+\tau_q^e[proc']+offset'$.

When a clock is approximated in this manner, error is introduced by passing that clock from p to q because $c_q(t)+\tau_q^e[proc]$ is only an approximation for $\hat{c}_{proc}^e(t)$. This means copies of $\hat{c}_x^i$ that traverse different routes and are received by a single processor might not be identical, even though they should be. Consequently, equality tests or selection of a clock based on the mode of a set of clocks received cannot be used when clocks are passed around the system in this fashion.

Two schemes have been devised for modifying an agreement protocol to avoid these problems with inequality of clock copies. The first is for the agreement protocol to be formulated in a way that avoids using equality tests to select one from among the different (clock) copies received. Lamport and Melliar-Smith use this technique in their Byzantine Agreement protocols for clocks, which are based on Byzantine Agreement protocols [Lamport *et al.* 82] that take the median of the set of values received, and hence do not use equality of values. The second way to avoid the inequality of clock copies problem is to consider a collection of clocks "equal" if all are within 2Λ of some clock value in that collection. (Recall, Λ is the maximum clock reading error between any pair of processes.) Mahaney and Schneider use this approach to modify the Crusaders Agreement protocol of [Dolev 82], which uses equality of values, to handle clocks [Mahaney & Schneider 85].

5.2. Obtaining Faster Convergence by Agreement

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}.

Crusader's Convergence: CF_{CCA} is:

(1) Each processor employs the Crusader's Agreement protocol to disseminate its clock.

(2) The value of CF_{CCA} at processor p is the result of p applying CF_{FCA} to the set of clocks received.

CF_{CCA} has half the precision of CF_{FCA} (i.e. convergence is twice as good) because due to CRU1 of Crusaders Agreement, it is not possible for correct processors p and q to use values for $\hat{c}_r(t)$ that differ by more than 2Λ unless one of p and q "knows" that r is faulty, in which case it can ignore $\hat{c}_r(t)$ completely. CF_{CCA} has the same accuracy and degree of fault tolerance as CF_{FCA}. It is interesting to note that when CF_{FCA} is iterated twice—which requires the same two rounds of message exchange as the Crusaders Agreement used in 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.

When a Byzantine Agreement is used to disseminate clocks, all correct processors agree within 2Λ on an approximation for the clock at each processor, due to BYZ1 and the error bounds in approximating clocks. Correct processors evaluating a convergence function will then differ by at most 2Λ in values in corresponding argument positions. Define Sel_g to be a function that returns its g^{th} largest argument. If we employ a Byzantine Agreement protocol that can tolerate k failures to disseminate arguments used in Sel_{k+1}, then we obtain a convergence function CF_{Byz} for clock synchronization:

Byzantine Convergence: CF_{Byz} is:

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

(2) The value of CF_{Byz} at processor p is the result of p applying Sel_{k+1} to the set of clocks received.

Provided there are k or fewer failures, Sel_{k+1} at a correct processor p selects a clock that is guaranteed to read within $\varepsilon=2\Lambda$ of the clock selected by every other. This means that the precision of CF_{Byz} is $\pi_{Byz}(\delta, \varepsilon)=2\Lambda$—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 is either a correct clock or 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 $\alpha_{Byz}(\delta)=\delta$.

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

5.3. Fireworks Agreement: An Optimization

When CF_{Byz} is used as a convergence function, only the largest $k+1$ clocks are actually needed. (Only the $k+1^{st}$ largest clock is returned, but to decide which clock is the $k+1^{st}$ largest, the $k+1$ largest clocks are needed.) Since performing a Byzantine Agreement can be costly—in both delay and number of messages exchanged—avoiding Byzantine Agreements on the other clocks is desirable. We, therefore, propose a somewhat weaker form of agreement to take the place of the Byzantine Agreements used in connection with CF_{Byz}. This new form of agreement, which we call a *Fireworks Agreement*, effectively allows correct processors to agree on the value of a single correct clock by causing all to terminate the protocol at approximately the same (real) time:

FW: All correct processors terminate with some *a priori* decided value v within β real seconds of each other.

The name Fireworks Agreement is in analogy with a public fireworks display, where 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 is related to message-delivery times.

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

A1: authenticate the sender of every message it receives and

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

These assumptions are satisfied if digital signatures are employed by the sender of a message or if fewer than 1/3 of the processors are faulty and the simulated authentication technique of [Srikanth & Toueg 84] is used to transmit messages. In either case (i) faulty processors are unable to masquerade as correct processors and (ii) faulty processors are unable to modify and then retransmit messages received from correct processors.

The following protocol implements a Fireworks Agreement for a message with value T. The protocol is specified for a processor p and described as two rules, each of which might be implemented as a separate process. The term "sufficient evidence" of rule (2) is defined below.

(1) When $\hat{c}_p(t)=T$, processor p signs and broadcasts $\langle T, p \rangle$ to all processors (including itself).

(2) Upon receiving "sufficient evidence", p broadcasts that evidence to all processors and terminates the protocol.

Two different schemes have been proposed for determining when there is "sufficient evidence" as required in rule (2). Before turning to the details of these, we show that any scheme satisfying the following properties leads to termination of the protocol by all correct processors within $\beta = \Gamma_{max}/(1-\hat{\rho})$ real seconds:

Achievement of Sufficient Evidence: Some correct processor eventually determines that there is "sufficient evidence".

Criterion for Sufficient Evidence: Evidence that is considered sufficient by a correct processor p and rebroadcast is considered sufficient by any correct processor receiving that broadcast.

According to Achievement of Sufficient Evidence, eventually some correct processor will determine that there is "sufficient evidence". Suppose p is the first to terminate and does so at real time t_{sat}. According to rule (2) above, it must have broadcast its "sufficient evidence" to all processors. In the worst case, there are no other undelivered messages in the network when p makes that broadcast. Thus, p's "sufficient evidence" can take as long as $\Gamma_{max}/(1-\hat{\rho})$ real seconds to be received by another correct processor q and therefore can be received as late as real time $t_{sat}+\Gamma_{max}/(1-\hat{\rho})$. According to Criterion for Sufficient Evidence, q must also consider this "sufficient evidence", and, according to rule (2), terminate the protocol. Thus, by $t_{sat}+\Gamma_{max}/(1-\hat{\rho})$ all correct processors have terminated the Fireworks Agreement and we conclude $\beta=\Gamma_{max}/(1-\hat{\rho})$.

Independent of the refinement of "sufficient evidence", Fireworks Agreement is used in constructing a convergence function CF_{FW} as follows. For the i^{th} Fireworks Agreement, we use $T=(i-1)R$ where (as in section 2)

$$r_{min} \le \frac{R}{(1+\hat{\rho})} \le \frac{R}{(1-\hat{\rho})} \le r_{max}.$$

And, for the value of $CF_{FW}(p, ...)$ associated with the i^{th} Fireworks Agreement we use:

$$CF_{FW}(p, \hat{c}_1(t_p^i)+v, \ldots, \hat{c}_N(t_p^i)+v) \equiv (i-1)R+\Gamma_{max}+v \quad \text{for } 0 \le v. \tag{5.1}$$

Note that Monotonicity and Translation Invariance hold for CF_{FW} by definition.

To bound precision $\pi_{FW}(\delta, \varepsilon)$ of CF_{FW}, substituting into the definition of precision, we get:

$$\pi_{FW}(\delta, \varepsilon) \ \ge \ |CF_{FW}(p, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) - CF_{FW}(q, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)|. \tag{5.2}$$

Without loss of generality, suppose $t_p^i < t_q^i$ so that due to Monotonicity (5.2) simplifies to

$$\pi_{FW}(\delta, \varepsilon) \ \ge \ CF_{FW}(p, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) - CF_{FW}(q, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)). \tag{5.3}$$

Using (5.1) with $v=(t_q^i-t_p^i)(1+\hat{\rho})$ we get:

$$CF_{FW}(p, \hat{c}_1(t_p^i)+(t_q^i-t_p^i)(1+\hat{\rho}), \ldots, \hat{c}_N(t_p^i)+(t_q^i-t_p^i)(1+\hat{\rho})) = (i-1)R+\Gamma_{max}+(t_q^i-t_p^i)(1+\hat{\rho}). \tag{5.4}$$

Equation (5.4) is now simplified as follows. First, because $\hat{c}_1(t_q^i) \le \hat{c}_1(t_p^i)+(t_q^i-t_p^i)(1+\hat{\rho})$ due to Virtual Rate (2.4), we conclude using Monotonicity that

$$CF_{FW}(p, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) \le CF_{FW}(p, \hat{c}_1(t_p^i)+(t_q^i-t_p^i)(1+\hat{\rho}), \ldots, \hat{c}_N(t_p^i)+(t_q^i-t_p^i)(1+\hat{\rho})).$$

Therefore, transitivity with (5.4) yields,

$$CF_{FW}(p, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) \le (i-1)R+\Gamma_{max}+(t_q^i-t_p^i)(1+\hat{\rho}).$$

By definition of β, $t_q^i-t_p^i \le \beta$. Making this substitution into the previous equation results in

$$CF_{FW}(p, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) \le (i-1)R+\Gamma_{max}+\beta(1+\hat{\rho}).$$

Substituting this into (5.3) gives a bound for $\pi_{FW}(\delta, \varepsilon)$:

$$\pi_{FW}(\delta, \varepsilon) \ \ge \ ((i-1)R+\Gamma_{max}+\beta(1+\hat{\rho})) - CF_{FW}(q, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)). \tag{5.5}$$

By definition (5.1), $CF_{FW}(q, \hat{c}_1(t_q^i), \ldots, \hat{c}_N(t_q^i)) \equiv (i-1)R+\Gamma_{max}$. We use this to simplify (5.5) further, obtaining

$$\begin{aligned}\pi_{FW}(\delta, \varepsilon) \ &\ge \ ((i-1)R+\Gamma_{max}+\beta(1+\hat{\rho})) - ((i-1)R+\Gamma_{max}) \\ &\ge \ \beta(1+\hat{\rho}) \\ &\ge \ \frac{\Gamma_{max}}{1-\hat{\rho}}(1+\hat{\rho}).\end{aligned}$$

Sufficient Evidence

One characterization of "sufficient evidence", which is the basis for the clock synchronization protocol of [Halpern *et al.* 84], exploits the fact that the clock at a correct processor must be within $\hat{\delta}$

of the clock at any other correct processor.[11]

SE1: Receipt of a message $m = \langle T, q \rangle$ by p is considered sufficient evidence iff m is correctly signed by $s \geq 1$ processors and received by p at real time t_{rcv} such that

$$T - s\,(\hat{\delta} + \Gamma_{min}) \leq \hat{c}_p(t_{rcv}) \leq T + s\,(\hat{\delta} + \Gamma_{max}). \qquad (5.6)$$

To show that SE1 satisfies the Criterion for Sufficient Evidence, suppose a Fireworks Agreement terminates at p at time $\hat{c}_p(t_{rcv})$ due to receipt of a message m. Thus, (5.6) holds. We must show that (5.6) will hold whenever m is forwarded to another correct processor q. Thus, we must show that

$$T - (s+1)\,(\hat{\delta} + \Gamma_{min}) \leq \hat{c}_q(t_{rcv2}) \leq T + (s+1)\,(\hat{\delta} + \Gamma_{max})$$

holds, where t_{rcv2} is the time that q received the copy of m forwarded by p. Since p and q are both correct, $|\hat{c}_p(t_{rcv}) - \hat{c}_q(t_{rcv})| \leq \hat{\delta}$. Therefore, we can rewrite (5.6) in terms of $\hat{c}_q(t_{rcv})$

$$T - s\,(\hat{\delta} + \Gamma_{min}) - \hat{\delta} \leq \hat{c}_q(t_{rcv}) \leq T + s\,(\hat{\delta} + \Gamma_{max}) + \hat{\delta}. \qquad (5.7)$$

Since at real time t_{rcv}, p forwarded the evidence to q, by the definitions of Γ_{min} and Γ_{max} we have

$$\hat{c}_q(t_{rcv}) + \Gamma_{min} \leq \hat{c}_q(t_{rcv2}) \leq \hat{c}_q(t_{rcv}) + \Gamma_{max}. \qquad (5.8)$$

We can now substitute in (5.8) for $\hat{c}_q(t_{rcv})$ using (5.7) and obtain

$$T - s\,(\hat{\delta} + \Gamma_{min}) - \hat{\delta} + \Gamma_{min} \leq \hat{c}_q(t_{rcv2}) \leq T + s\,(\hat{\delta} + \Gamma_{max}) + \hat{\delta} + \Gamma_{max},$$

which, since the copy of m forwarded to q by p contains one more signature, implies (5.6).

It only remains to show that SE1 is eventually satisfied, hence Achievement of Sufficient Evidence holds. The argument is simple. A correct processor executing rule (1) of the protocol will receive a copy of the message it has broadcast. This copy will satisfy (5.6) because it will arrive between Γ_{min} and Γ_{max} clock seconds after it was sent.

Accuracy $\alpha_{SE1}(\delta)$ for SE1 is illustrated as follows. Suppose

p is the correct processor with the fastest clock,

q is a faulty (even faster) processor such that $\hat{c}_q(t) - \hat{c}_p(t) = \hat{\delta}$, and

r is the correct processor with the slowest clock and therefore $\hat{c}_p(t) - \hat{c}_r(t) = \hat{\delta}$.

Further, suppose q executes rule (1) at time $\hat{c}_q(t_{snd}) = T$ and broadcasts a message $m = \langle T, q \rangle$. By definition of p and q, $\hat{c}_p(t_{snd}) = T - \hat{\delta}$. The message will, therefore, be delivered to p by

[11] The protocol of [Cristian *et al.* 86] also uses a variant of this form of "sufficient evidence". However, the test used there is simpler than the one discussed here because their protocol tolerates only omission failures—not full Byzantine failures.

$T-\hat{\delta}+\Gamma_{min} \le \hat{c}_p(t_{rcv}) \le T-\hat{\delta}+\Gamma_{max}$ and p will find the message to be sufficient evidence, because it satisfies (5.6). By definition of p and r, we have

$$T-\hat{\delta}+\Gamma_{min}-\hat{\delta} \le \hat{c}_r(t_{rcv}) \le T-\hat{\delta}+\Gamma_{max}-\hat{\delta}.$$

Therefore, when r receives the copy of the message rebroadcast (according to rule (2)) by p, that time $\hat{c}_r(t_{rcv2})$ is given by

$$T-\hat{\delta}+\Gamma_{min}-\hat{\delta}+\Gamma_{min} \le \hat{c}_r(t_{rcv2}) \le T-\hat{\delta}+\Gamma_{max}-\hat{\delta}+\Gamma_{max}.$$

The message, therefore, satisfies (5.6) and is sufficient evidence for r to terminate. Moreover, since $T-\hat{\delta}+\Gamma_{min}-\hat{\delta}+\Gamma_{min} \le \hat{c}_r(t_{rcv2})$ and according to the protocol (i.e. (5.1)) r must set its clock ahead to $T+\Gamma_{max}$, r might therefore have to set it ahead by as much as

$$(T+\Gamma_{max}) - (T-\hat{\delta}+\Gamma_{min}-\hat{\delta}+\Gamma_{min}).$$

We conclude

$$\alpha_{SE1}(\delta) = \Gamma_{max}+2(\delta-\Gamma_{min}).$$

Accuracy $\alpha_{SE1}(\delta)$ reveals a problem with SE1: A faulty processor (i.e. q) with a fast clock can cause clocks at correct processors to reset so that they run faster than they should. (The consequences of this are quantified in the Appendix.) On the other hand, SE1 has fault-tolerance degree $N-1$ because it was not necessary to stipulate an upper bound on the number of faulty processors.

A second characterization of "sufficient evidence", first used in the clock synchronization protocol of [Srikanth & Toueg 84][12], is based on the fact that if every processor broadcasts a message when its clock reads T, then provided there are at most k faulty processors, the $k+1^{st}$ message received must be from a correct one or must follow a message from a correct one.

SE2: Receipt of $k+1$ messages originated by distinct processors is considered sufficient evidence.

It is easy to see that SE2 satisfies our Criterion for Sufficient Evidence—even after being forwarded to another processor, the $k+1$ messages used for sufficient evidence at one processor are still originated by $k+1$ distinct processors, so they will be considered sufficient evidence at another. Ensuring Achievement of Sufficient Evidence, requires making an assumption about the number of faulty processors. SE2 is guaranteed to hold only if $N \le 2k+1$ because then there are fewer than $k+1$ faulty processors and at least $k+1$ correct ones. Thus, fault-tolerance degree $k=(N-1)/2$.

[12] A similar scheme was later used in the protocol of [Babaoglu & Drummond 87].

The fact that when some processor receives sufficient evidence according to SE2 it must have received a message from a correct processor means that the accuracy of SE2 is better than that of SE1. A scenario that achieves worst-case accuracy with SE2 is given by the following. Suppose,

$p_1, p_2, \ldots, p_k$ are correct processors with fast clocks,

p_{k+1} is a faulty processor with a fast clock, and

r is the correct processor with the slowest clock, so $(\forall i\colon\ 1 \le i \le k+1\colon\ \hat{c}_{p_i}(t) - \hat{c}_r(t) = \hat{\delta})$.

Further, suppose each processor p_i, $1 \le i \le k+1$ broadcasts a message when $\hat{c}_{p_i}(t_{snd_i}) = (i-1)R = T$. Thus, these messages are sent at time $\hat{c}_r(t_{snd_i}) = T - \hat{\delta}$ and can be received by r as early as time $\hat{c}_r(t_{rcv}) = T - \hat{\delta} + \Gamma_{min}$. The set of $k+1$ messages broadcast by p_1 through p_{k+1} satisfy SE2, so r must advance its clock by as much as

$$CF_{FW}(r, \hat{c}_1^i(t_r^i), \ldots \hat{c}_N^i(t_r^i)) - (T - \hat{\delta} + \Gamma_{min})$$

$$= (T + \Gamma_{max}) - (T - \hat{\delta} + \Gamma_{min})$$

and we conclude

$$\alpha_{SE2}(\delta) = \Gamma_{max} + \delta - \Gamma_{min}.$$

Clearly, accuracy with SE2 is superior to that achieved with SE1. This is not without cost, however. SE2 requires that fewer than half the processors are faulty; SE1 makes no assumptions about the number of faulty processors.

Clock synchronization algorithms based on Fireworks Agreement are interesting because a processor cannot even 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 convergence functions discussed earlier provided an implementation of RTS2 only. On the other hand, inherent in Fireworks Agreement is that processor clocks are read in the less accurate of the two ways presented in section 3. Moreover, while it is possible to achieve precision of 2Λ using an agreement algorithm (i.e, CF_{Byz}), CF_{FW} does not come close. The precision of CF_{FW} depends on the maximum message delivery delay, while precision of CF_{Byz} is determined by the variance in message delivery delay.

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. Implementing the reliable time source involves solving three subproblems. Different solutions to these subproblems yield different protocols.

The first subproblem defined by our paradigm is to generate events that cause all processors to resynchronize. Any solution to this subproblem can be characterized in terms of three constants: r_{min} and r_{max} bound the real-time interval that can elapse between when the first correct processor to resynchronize for the i^{th} time does so and when the first correct processor to resynchronize for the $i+1^{th}$ time does so. β bounds the real time that can elapse between when the first correct processor resynchronizes for the i^{th} time and when the last correct processor resynchronizes for the i^{th} time.

The second subproblem defined by our paradigm is how a program being executed by one processor can read the clocks on another. A solution to this subproblem is characterized in terms of Λ, an upper bound on clock reading error.

The final subproblem defined by our paradigm is choice of a convergence function. Any function that satisfies the four properties given in §4—Monotonicity, Translation Invariance, Precision Enhancement, and Accuracy Preservation—will work. Such a function is characterized by its precision π, which bounds how closely it will bring values together, and its accuracy α, which bounds how far its result will be from its argument.

If processor clocks run close together but far from real time, clocks implemented by an algorithm based on our paradigm will remain synchronized with each other but will diverge from 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 60 KHz radio broadcasts 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 was 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.

Appendix 1: Proof of Clock Synchronization

This section gives sufficient conditions to ensure that the clock synchronization protocol of Figure 2.1 satisfies correctness conditions Virtual Synchronization (2.3) and Virtual Rate (2.4). We assume only the following about the solutions used for the three subproblems left open in that protocol.

> **Event Generation.** r_{min} and r_{max} are the lower and upper bounds for the real-time interval that can elapse between when the first correct processor to resynchronize for the i^{th} time does so and when the first correct processor to resynchronize for the $i+1^{th}$ time does so. β bounds the real time that can elapse between when the first and last correct processor resynchronizes for the i^{th} time.
>
> **Clock Reading.** Λ is an upper bound on the error associated with the value obtained when a program executing on one processor reads the clock on another.
>
> **Convergence Function.** *CF* has precision π, has accuracy α, and satisfies the Monotonicity, Translation Invariance, Precision Enhancement, and Accuracy Preservation Properties of §4.

To simplify the exposition that follows, p, q, r, and x are assumed to range over correct processors only.

Synchronization of Virtual Clocks

To prove that Virtual Synchronization (2.3) is satisfied, we start by establishing that all correct processors have started their i^{th} virtual clocks by the time the first correct processor starts its $i+1^{st}$ virtual clock. This is necessary in order to be able to execute the assignment to adj_p^{i+1} in the protocol.

Lemma 1: Let $t_q^{i+1}=(\min r\text{:}\ \ t_r^{i+1})$. If $\beta \le r_{min}$ then for any correct processor p, $t_p^i \le t_q^{i+1}$.

Proof: Let $t_x^i=(\min r\text{:}\ \ t_r^i)$. By the definition of r_{min} in RTS1, $r_{min} \le t_q^{i+1}-t_x^i$. Adding t_x^i to both sides, we get $r_{min}+t_x^i \le t_q^{i+1}$. The hypothesis that $\beta \le r_{min}$ implies $t_x^i+\beta \le r_{min}+t_x^i$, so by transitivity $t_x^i+\beta \le r_{min}+t_x^i \le t_q^{i+1}$. Moreover, from the definition of β in RTS1, $t_p^i \le t_x^i+\beta$, so again by transitivity $t_p^i \le t_x^i+\beta \le r_{min}+t_x^i \le t_q^{i+1}$. □

We now prove that virtual clocks that employ instantaneous resynchronization (i.e. $AI \le \hat{\kappa}$) satisfy Virtual Synchronization (2.3). Define

$$\bar{c}_p^i(t) \ \equiv \ c_p(t)+adj_p^i.$$

And, as before, let $\bar{c}_p(t)$ be the value of $\bar{c}_p^i(t)$ where i satisfies $t_p^i \le t < t_p^{i+1}$. The proof of Virtual

Synchronization (2.3) for $\bar{c}_p^i$ is in two steps. The first step (Lemma 2) shows that when the last correct processor to start its i^{th} virtual clock does so, the i^{th} virtual clocks at all correct processors will be close together; the second step (Lemma 3) extends this, showing that this implies that correct virtual clocks will remain close together.

Lemma 2: If (i) $\beta \le r_{min}$

(ii) $u \le \bar{\delta}_S$

(iii) $\pi(\bar{\delta}_S + 2\beta\rho,\ 2(\beta(1+\rho)+\Lambda)) \le \bar{\delta}_S$

then $(\forall i\!:\ 0<i\!:\ t_x^i = \max(t_p^i, t_q^i) \Rightarrow |\bar{c}_q(t_x^i) - \bar{c}_p(t_x^i)| \le \bar{\delta}_S)$.

Proof: By induction on i.

Base Case: $(\forall i\!:\ 0<i\le 1\!:\ t_x^i = \max(t_p^i, t_q^i) \Rightarrow |\bar{c}_q(t_x^i) - \bar{c}_p(t_x^i)| \le \bar{\delta}_S)$.

$0 \le c_p(0) \le u$ due to Hardware Initial Value (2.1).

$0 \le \bar{c}_p(0) \le u$ because $adj_p^0 = 0$ (see Figure 2.1).

$0 \le \bar{c}_q(0) \le u$ same argument for processor q.

$0 \le \bar{c}_p(t_x^1) \le u$ and $0 \le \bar{c}_q(t_x^1) \le u$ since $t_p^1 = t_q^1 = 0 = \max(t_p^1, t_q^1) = t_x^1$.

$|\bar{c}_q(t_x^1) - \bar{c}_p(t_x^1)| \le u$ substituting with previous line.

$|\bar{c}_q(t_x^1) - \bar{c}_p(t_x^1)| \le \bar{\delta}_S$ due to hypothesis (ii).

$(\forall i\!:\ 0<i\le 1\!:\ t_x^i = \max(t_p^i, t_q^i) \Rightarrow |\bar{c}_q(t_x^i) - \bar{c}_p(t_x^i)| \le \bar{\delta}_S)$.

Induction Case. As an Induction Hypothesis assume:

$$(\forall l\!:\ 0<l\le i\!:\ t_x^l = \max(t_p^l, t_q^l) \Rightarrow |\bar{c}_q(t_x^l) - \bar{c}_p(t_x^l)| \le \bar{\delta}_S). \tag{A1}$$

According to the protocol of Figure 2.1, the definition of $\bar{c}_p^{i+1}$, and the fact that reading the clock at another processor has an associated error, we have:

$$\bar{c}_p^{i+1}(t_p^{i+1}) = CF(p, \bar{c}_1^i(t_p^{i+1}) + \lambda_p(1), \ldots, \bar{c}_N^i(t_p^{i+1}) + \lambda_p(N)) \tag{A2}$$

$$\bar{c}_q^{i+1}(t_q^{i+1}) = CF(q, \bar{c}_1^i(t_q^{i+1}) + \lambda_q(1), \ldots, \bar{c}_N^i(t_q^{i+1}) + \lambda_q(N)) \tag{A3}$$

The arguments to CF in both (A2) and (A3) are defined (and therefore can be computed) due to Lemma 1 and hypothesis (i). Without loss of generality, assume $t_q^{i+1} \le t_p^{i+1}$. For correct processors p and q, we conclude

$$\bar{c}_q^{i+1}(t_p^{i+1}) \le \bar{c}_q^{i+1}(t_q^{i+1}) + (t_p^{i+1} - t_q^{i+1})(1+\rho)$$

due to Hardware Rate (2.2). Using Monotonicity of CF, we substitute for $\bar{c}_q^{i+1}(t_q^{i+1})$ in this formula based on (A3) and obtain

$$\bar{c}_q^{\,i+1}(t_p^{i+1}) \le CF(q, \bar{c}_1^{\,i}(t_q^{i+1})+\lambda_q(1), \dots, \bar{c}_N^{\,i}(t_q^{i+1})+\lambda_q(N)) + (t_p^{i+1}-t_q^{i+1})(1+\rho). \tag{A4}$$

Since $t_p^{i+1}-t_q^{i+1} \le \beta$ due to RTS1 (§2), (A4) can be simplified to

$$\bar{c}_q^{\,i+1}(t_p^{i+1}) \le CF(q, \bar{c}_1^{\,i}(t_q^{i+1})+\lambda_q(1), \dots, \bar{c}_N^{\,i}(t_q^{i+1})+\lambda_q(N)) + \beta(1+\rho).$$

Translation Invariance allows the $\beta(1+\rho)$ term to be moved inside CF, resulting in

$$\bar{c}_q^{\,i+1}(t_p^{i+1}) \le CF(q, \bar{c}_1^{\,i}(t_q^{i+1})+\beta(1+\rho)+\lambda_q(1), \dots, \bar{c}_N^{\,i}(t_q^{i+1})+\beta(1+\rho)+\lambda_q(N)). \tag{A5}$$

We can now use the Precision Enhancement Property for CF to show that $t_x^{i+1}=\max(t_p^{i+1},t_q^{i+1}) \Rightarrow |\bar{c}_q^{\,i+1}(t_x^{i+1})-\bar{c}_p^{\,i+1}(t_x^{i+1})| \le \bar{\delta}_S$, as required to establish the Induction Case. By assumption, $t_q^{i+1} \le t_p^{i+1}$ so it suffices to prove $|\bar{c}_q^{\,i+1}(t_p^{i+1})-\bar{c}_p^{\,i+1}(t_p^{i+1})| \le \bar{\delta}_S$ to establish the Induction Case. To do so, we first determine constants ε and δ for the Precision Enhancement Property.

To characterize ε, note that due to Hardware Rate (2.2) and the fact that $t_p^{i+1}-t_q^{i+1} \le \beta$, for each correct processor a,

$$\beta(1-\rho) \le (t_p^{i+1}-t_q^{i+1})(1-\rho) \le \bar{c}_a^{\,i}(t_p^{i+1})-\bar{c}_a^{\,i}(t_q^{i+1}) \le (t_p^{i+1}-t_q^{i+1})(1+\rho) \le \beta(1+\rho).$$

Also, from the definition of Λ,

$$(\forall b\colon\ |\lambda_a(b)| \le \Lambda).$$

Therefore, the difference between the value in equation (A5) of the r^{th} argument to CF and in equation (A2) for any correct processor a can be at most $\varepsilon=2(\beta(1+\rho)+\Lambda)$.

To characterize δ of the Precision Enhancement Property, note that by Induction Hypothesis (A1) we have for correct processors a and b

$$t_x^i=\max(t_a^i,t_b^i) \Rightarrow |\bar{c}_a(t_x^i)-\bar{c}_b(t_x^i)| \le \bar{\delta}_S. \tag{A6}$$

Without loss of generality, assume $\bar{c}_a^{\,i}(t_x^i) \le \bar{c}_b^{\,i}(t_x^i)$. Thus for the a^{th} and b^{th} arguments to CF in (A2):

$$\begin{aligned}
\delta &= |\bar{c}_a^{\,i}(t_p^{i+1})-\bar{c}_b^{\,i}(t_p^{i+1})| \\
&= \bar{c}_b^{\,i}(t_p^{i+1})-\bar{c}_a^{\,i}(t_p^{i+1}) \\
&\le \bar{c}_b^{\,i}(t_x^i)+(t_p^{i+1}-t_x^i)(1+\rho) - (\bar{c}_a^{\,i}(t_x^i)+(t_p^{i+1}-t_x^i)(1-\rho)) && \text{due to Hardware Rate (2.2) since } t_x^i \le t_p^{i+1} \text{ by Lemma 1.} \\
&\le \bar{c}_b^{\,i}(t_x^i)-\bar{c}_a^{\,i}(t_x^i)+2\beta\rho && \text{algebra and the definition of } \beta. \\
&\le \bar{\delta}_S+2\beta\rho && \text{due to (A6).}
\end{aligned}$$

Using (A5) to characterize $\bar{c}_q^{\,i+1}(t_p^{i+1})$ and (A2) to characterize $\bar{c}_p^{\,i+1}(t_p^{i+1})$, we get

$$|\bar{c}_q^{\,i+1}(t_p^{i+1})-\bar{c}_p^{\,i+1}(t_p^{i+1})| \ \le\ CF(q,\bar{c}_1^{\,i}(t_q^{i+1})+\beta(1+\rho)+\lambda_q(1), \dots, \bar{c}_N^{\,i}(t_q^{i+1})+\beta(1+\rho)+\lambda_q(N))$$

$$
\begin{aligned}
&- \; CF(p, \bar{c}_1^{\,i}(t_p^{i+1})+\lambda_p(1), \ldots, \bar{c}_N^{\,i}(t_p^{i+1})+\lambda_p(N)) \\
&\leq \; \pi(\bar{\delta}_S+2\beta\rho,\, 2(\beta(1+\rho)+\Lambda)) \qquad \text{by Precision Enhancement Property} \\
&\leq \; \bar{\delta}_S \qquad \text{by Hypothesis (iii).}
\end{aligned}
$$

This completes the Induction Case. □

Lemma 3: If (i) $\beta \leq r_{min}$

(ii) $u \leq \bar{\delta}_S$

(iii) $\pi(\bar{\delta}_S+2\beta\rho,\, 2(\beta(1+\rho)+\Lambda)) \leq \bar{\delta}_S$

(iv) $\bar{\delta}_S+2\rho(r_{max}+\beta) \leq \bar{\delta}$

(v) $\alpha(\bar{\delta}_S+2\rho(r_{max}+\beta))+2\rho\beta \leq \bar{\delta}$

then $(\forall t\colon\ 0 \leq t\colon\ |\bar{c}_q(t)-\bar{c}_p(t)| \leq \bar{\delta})$.

Proof: The conclusion of the lemma is equivalent to

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \max(t_p^i,t_q^i) \leq t < \min(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{\,i}(t)-\bar{c}_q^{\,i}(t)| \leq \bar{\delta})) \quad \wedge \tag{A7}$$

$$\begin{aligned}(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \min(t_p^{i+1},t_q^{i+1}) \leq t < \max(t_p^{i+1},t_q^{i+1})\colon\ & t_q^{i+1} \leq t_p^{i+1} \Rightarrow |\bar{c}_q^{\,i+1}(t)-\bar{c}_p^{\,i}(t)| \leq \bar{\delta} \wedge \\ & t_p^{i+1} \leq t_q^{i+1} \Rightarrow |\bar{c}_q^{\,i}(t)-\bar{c}_p^{\,i+1}(t)| \leq \bar{\delta}))\end{aligned} \tag{A8}$$

We first prove (A7). Due to hypothesis (i) – (iii) we can use Lemma 2 to conclude:

$$(\forall i\colon\ 0<i\colon\ t_x^i=\max(t_p^i,t_q^i) \Rightarrow |\bar{c}_q(t_x^i)-\bar{c}_p(t_x^i)| \leq \bar{\delta}_S).$$

According to Hardware Rate (2.2), correct clocks drift apart no more than 2ρ clock seconds/real second, and therefore

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \max(t_p^i,t_q^i) \leq t\colon\ |\bar{c}_q^{\,i}(t)-\bar{c}_p^{\,i}(t)| \leq \bar{\delta}_S+2\rho(t-\max(t_p^i,\, t_q^i)))).$$

This implies

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \max(t_p^i,t_q^i) \leq t \leq \min(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{\,i}(t)-\bar{c}_p^{\,i}(t)| \leq \bar{\delta}_S+2\rho(r_{max}+\beta))). \tag{A9}$$

because $r_{min}-\beta \leq \min(t_p^{i+1},t_q^{i+1})-\max(t_p^i,t_q^i) \leq r_{max}+\beta$ due to RTS1 (§2). Using Hypothesis (iv), (A9) can be simplified to

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \max(t_p^i,t_q^i) \leq t \leq \min(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{\,i}(t)-\bar{c}_p^{\,i}(t)| \leq \bar{\delta})),$$

which implies (A7) as desired.

To prove (A8), without loss of generality we assume that $t_q^{i+1} \leq t_p^{i+1}$. Thus, (A8) is equivalent to

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \min(t_p^{i+1},t_q^{i+1}) \leq t < \max(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{\,i+1}(t)-\bar{c}_p^{\,i}(t)| \leq \bar{\delta})) \tag{A10}$$

and it suffices to prove that. To do this, we infer from (A9)

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ t=\min(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^i(t)-\bar{c}_p^i(t)| \le \bar{\delta}_S+2\rho(r_{max}+\beta))). \tag{A11}$$

Therefore, we can take δ in the definition of accuracy α to be $\delta=\bar{\delta}_S+2\rho(r_{max}+\beta)$ and using the Accuracy Preservation Property obtain a bound for how $\bar{c}_q^{i+1}(t_q^{i+1})$ differs from any argument to CF used in calculating $\bar{c}_q^{i+1}(t_q^{i+1})$. Since $\bar{c}_p^i(t_q^{i+1})$) must have been such an argument:

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ t=\min(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{i+1}(t)-\bar{c}_p^i(t)| \le \alpha(\bar{\delta}_S+2\rho(r_{max}+\beta)))).$$

This implies that

$$\begin{aligned}(\forall i\colon\ 0<i\colon\ (\forall t\colon\ &\min(t_p^{i+1},t_q^{i+1})\le t\colon \\ &|\bar{c}_q^{i+1}(t)-\bar{c}_p^i(t)| \le \alpha(\bar{\delta}_S+2\rho(r_{max}+\beta))+2\rho(t-\min(t_p^{i+1},t_q^{i+1}))))\end{aligned} \tag{A12}$$

due to Hardware Rate (2.2). From the definition of β in RTS1, $0\le\max(t_p^{i+1},t_q^{i+1})-\min(t_p^{i+1},t_q^{i+1})\le\beta$, so equation (A12) implies

$$\begin{aligned}(\forall i\colon\ 0<i\colon\ (\forall t\colon\ &\min(t_p^{i+1},t_q^{i+1})\le t<\max(t_p^{i+1},t_q^{i+1})\colon \\ &|\bar{c}_q^{i+1}(t)-\bar{c}_p^i(t)| \le\alpha(\bar{\delta}_S+2\rho(r_{max}+\beta))+2\rho\beta)).\end{aligned}$$

Substituting for $\alpha(\bar{\delta}_S+2\rho(r_{max}+\beta))+2\rho\beta)$ according to Hypothesis (v) yields

$$(\forall i\colon\ 0<i\colon\ (\forall t\colon\ \min(t_p^{i+1},t_q^{i+1})\le t<\max(t_p^{i+1},t_q^{i+1})\colon\ |\bar{c}_q^{i+1}(t)-\bar{c}_p^i(t)| \le\bar{\delta}))$$

as was required (i.e. (A10)) in order to prove (A8). □

The previous lemma established that virtual clocks using instantaneous resynchronization satisfy Virtual Synchronization (2.3). We now prove that virtual clocks using continuous resynchronization also satisfy Virtual Synchronization (2.3). Define ai to be the maximum number of real seconds it takes for adjustment interval AI to elapse at any correct processor. Thus, $ai=AI/(1-\hat{\rho})$. Further, define a *fixed clock* $\ddot{c}_p^i$ to be a function from real time to clock time satisfying[13]

FC1: $(\forall t\colon\ t_p^{i+1}+ai<t\colon\ \ddot{c}_p^{i+1}(t)=\bar{c}_p^{i+1}(t))$ and

FC2: $(\forall t\colon\ t_p^{i+1}\le t\le t_p^{i+1}+ai\colon\ \ddot{c}_p^{i+1}(t)\in[\bar{c}_p^i(t),\bar{c}_p^{i+1}(t)])$.

Thus, outside of its adjustment interval, the value of $\ddot{c}_p^{i+1}(t)$ is the same as $\bar{c}_p^{i+1}(t)$; and during its adjustment interval, the value of $\ddot{c}_p^{i+1}(t)$ is guaranteed to lie between the value of $\bar{c}_p^i(t)$ and $\bar{c}_p^{i+1}(t)$.

From FC1 and FC2, we conclude that in order to prove for any given D $(\forall t\colon\ 0\le t\colon\ |\ddot{c}_p(t)-\ddot{c}_q(t)|\le D)$, we must establish

[13] We use the notation $x\in[a,b]$ to denote $min(a,b)\le x\le max(a,b)$.

$$(\forall t\colon\ t_p^{i+1} \le t < t_p^{i+2} \wedge t_q^{j+1} \le t < t_p^{j+2}\colon\ |\bar{c}_p^{\,i+1}(t) - \bar{c}_q^{\,j+1}(t)| \le D) \text{ and} \tag{A13}$$

$$\begin{aligned}(\forall t\colon\ t_p^{i+1} \le t \le t_p^{i+1} + ai \wedge t_q^{j+1} \le t \le t_q^{j+1} + ai\colon\ & |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j}(t)| \le D \wedge \\ & |\bar{c}_p^{\,i+1}(t) - \bar{c}_q^{\,j}(t)| \le D \wedge \\ & |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j+1}(t)| \le D).\end{aligned} \tag{A14}$$

Since according to definition (2.5), a virtual clock $\hat{c}_p$ satisfies the definition of a fixed clock, by choosing $\hat{\delta} \ge \alpha(\bar{\delta}+2\rho(ai))$, the following theorem proves that Virtual Synchronization (2.3) holds for virtual clocks that use *FIX* to implement continuous resynchronization.

Theorem 4: If $(\forall t\colon\ 0 \le t\colon\ |\bar{c}_p(t) - \bar{c}_q(t)| \le \bar{\delta})$

then $(\forall t\colon\ 0 \le t\ \ |\ddot{c}_p(t) - \ddot{c}_q(t)| \le \alpha(\bar{\delta}+2\rho(ai)))$.

Proof: The result follows if we prove (A13) and (A14) for $D = \alpha(\bar{\delta}+2\rho(ai)))$. Using the definition of $\bar{c}_p$, we rewrite the hypothesis of the theorem as:

$$(\forall t\colon\ t_p^{i} \le t < t_p^{i+1} \wedge t_q^{j} \le t < t_q^{j+1}\colon\ |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j}(t)| \le \bar{\delta}). \tag{A15}$$

This implies (A13) if $\bar{\delta} \le \alpha(\bar{\delta}+2\rho(ai))$. To see that $\bar{\delta} \le \alpha(\bar{\delta}+2\rho(ai))$, first note that $\bar{\delta} \le \bar{\delta}+2\rho(ai)$ since $0 \le \rho$ and $0 \le ai$. The result then follows because $\delta \le \alpha(\delta)$ due to (4.1).

All that remains is to prove (A14). Due to Hardware Rate (2.2), $\bar{c}_p^{\,i}$ and $\bar{c}_q^{\,j}$ can drift apart by at most 2ρ seconds per second. Therefore, we can extend the range of (A15) as follows:

$$(\forall t,\colon\ t_p^{i} \le t < t_p^{i+1} + ai \wedge t_q^{j} \le t < t_q^{j+1} + ai\colon\ |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j}(t)| \le \bar{\delta}+2\rho(ai)). \tag{A16}$$

By definition, $t_p^{i} < t_p^{i+1} < t_p^{i+1} + ai$ and $t_q^{j} < t_q^{j+1} < t_q^{j+1} + ai$. So, from (A16) we conclude

$$(\forall t\colon\ t_p^{i+1} \le t < t_p^{i+1} + ai \wedge t_q^{j+1} \le t < t_q^{j+1} + ai\colon\ |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j}(t)| \le \bar{\delta}+2\rho(ai)). \tag{A17}$$

From property (4.1) of α, $\bar{\delta}+2\rho(ai) \le \alpha(\bar{\delta}+2\rho(ai))$, so

$$(\forall t\colon\ t_p^{i+1} \le t < t_p^{i+1} + ai \wedge t_q^{j+1} \le t < t_q^{j+1} + ai\colon\ |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j}(t)| \le \alpha(\bar{\delta}+2\rho(ai))). \tag{A18}$$

According to the Accuracy Preservation Property using $\delta = \bar{\delta}+2\rho(ai)$ due to (A16), and using the same argument as was used to change the range of (A15) to get (A17), we conclude:

$$(\forall t\colon\ t_p^{i+1} \le t < t_p^{i+1} + ai \wedge t_q^{j+1} \le t < t_q^{j+1} + ai\colon\ |\bar{c}_p^{\,i+1}(t) - \bar{c}_q^{\,j}(t)| \le \alpha(\bar{\delta}+2\rho(ai))) \tag{A19}$$

$$(\forall t\colon\ t_p^{i+1} \le t < t_p^{i+1} + ai \wedge t_q^{j+1} \le t < t_q^{j+1} + ai\colon\ |\bar{c}_p^{\,i}(t) - \bar{c}_q^{\,j+1}(t)| \le \alpha(\bar{\delta}+2\rho(ai))) \tag{A20}$$

We can now combine (A18), (A19), and (A20) obtaining (A14) with $\alpha(\bar{\delta}+2\rho(ai)) = D$. This, then, completes the proof. □

Rates of Virtual Clocks

To prove that virtual clocks satisfy Virtual Rate (2.4), we first require the following technical lemma.

Lemma 5: If $\varepsilon \geq 0$ then $(\max x,y,z\colon \min((x+\varepsilon)-y,z) - \min(x-y,z)) \leq \varepsilon$.

Proof: First, suppose $z \leq x-y$. This implies that $\min(x-y,z)=z$ and that $z \leq (x+\varepsilon)-y$. Consequently, $\min((x+\varepsilon)-y,z)=z$. This means that when $z \leq x-y$,

$$\min((x+\varepsilon)-y,z) - \min(x-y,z) = z-z = 0.$$

Next, suppose $z > x-y$. This implies that $\min(x-y,z)=x-y$. Therefore,

$$\begin{aligned}
&\min((x+\varepsilon)-y,z)-\min(x-y,z)\\
=&\min((x+\varepsilon)-y,z)-(x-y)\\
=&\min((x+\varepsilon)-y-(x-y),\ z-(x-y))\\
=&\min(\varepsilon, z-x+y)\\
\leq&\varepsilon
\end{aligned}$$

Since when $v \leq \varepsilon$ then $\max(0,v) \leq \varepsilon$, the lemma follows. □

We are now able to prove that virtual clocks have rates that satisfy Virtual Rate (2.4).

Theorem 6: If (i) $0 < \kappa \leq \hat{\kappa}$

(ii) $|\hat{c}_q(t) - \hat{c}_p(t)| \leq \hat{\delta}$

(iii) $(1+\rho)\left[1+\dfrac{\alpha(\hat{\delta})}{AI}\right] \leq (1+\hat{\rho})$

(iv) $(1-\hat{\rho}) \leq (1-\rho)\left[1-\dfrac{\alpha(\hat{\delta})}{AI}\right]$

then $0 < 1-\hat{\rho} \leq \dfrac{\hat{c}_p(t+\hat{\kappa})-\hat{c}_p(t)}{\hat{\kappa}} \leq 1+\hat{\rho}$ for $0 \leq t$.

Proof: Let i satisfy $t_p^i \leq t < t_p^{i+1}$. Consequently, $\hat{c}_p(t)=\hat{c}_p^{\,i}(t)$. Observe that $t+\hat{\kappa} \leq t_p^{i+1}$ because otherwise we would have

$$t_p^i \leq t < t_p^{i+1} < t+\hat{\kappa},$$

which would imply that p started $\hat{c}_p^{\,i+1}$ between two virtual clock ticks, contrary to the protocol of §2.

We are interested in bounding

$$\frac{\hat{c}_p^i(t+\hat{\kappa})-\hat{c}_p^i(t)}{\hat{\kappa}}$$

$$= \frac{\left[c_p(t+\hat{\kappa})+FIX_p^i(c_p(t+\hat{\kappa}))\right] - \left[c_p(t)+FIX_p^i(c_p(t))\right]}{\hat{\kappa}}$$

$$= \frac{c_p(t+\hat{\kappa})-c_p(t)+FIX_p^i(c_p(t+\hat{\kappa}))-FIX_p^i(c_p(t))}{\hat{\kappa}} \qquad \text{(A21)}$$

First, we derive an upper bound for (A21). For a correct processor p, we have

$$c_p(t+\hat{\kappa})-c_p(t) \leq (1+\rho)\hat{\kappa} \qquad \text{(A22)}$$

from Hardware Rate (2.2). According to the definition of FIX_p^i (in §2), we have

$$FIX_p^i(c_p(t+\hat{\kappa})) \;=\; adj_p^{i-1}+\frac{(adj_p^i - adj_p^{i-1})(\min(c_p(t+\hat{\kappa})-c_p(t_p^i), AI\,))}{AI}$$

$$FIX_p^i(c_p(t)) \;=\; adj_p^{i-1}+\frac{(adj_p^i - adj_p^{i-1})(\min(c_p(t)-c_p(t_p^i), AI\,))}{AI}$$

Therefore,

$$FIX_p^i(c_p(t+\hat{\kappa}))-FIX_p^i((c_p(t)) =$$
$$\frac{(adj_p^i-adj_p^{i-1})(\min(c_p(t+\hat{\kappa})-c_p(t_p^i), AI) - \min(c_p(t)-c_p(t_p^i), AI))}{AI}. \qquad \text{(A23)}$$

Letting

$$x+\varepsilon = c_p(t+\hat{\kappa})$$
$$x = c_p(t)$$
$$y = c_p(t_p^i)$$
$$z = AI$$

due to Hypothesis (i) and the fact that hardware clocks are non-decreasing, we have $\varepsilon \geq 0$. Thus, we can apply Lemma 5 to infer from (A23):

$$FIX_p^i(c_p(t+\hat{\kappa}))-FIX_p^i((c_p(t)) \;\leq\; \frac{(adj_p^i-adj_p^{i-1})(\varepsilon)}{AI}$$

$$\leq\; \frac{(adj_p^i-adj_p^{i-1})(c_p(t+\hat{\kappa})-c_p(t))}{AI}$$

$$\leq \frac{(adj_p^i - adj_p^{i-1})((1+\rho)\hat{\kappa})}{AI} \tag{A24}$$

Substituting into (A21), using (A22) for $c_p(t+\hat{\kappa})-c_p(t)$ and (A24) for $FIX_p^i(c_p(t+\hat{\kappa}))-FIX_p^i(c_p(t))$ we get

$$\frac{\hat{c}_p^i(t+\hat{\kappa})-\hat{c}_p^i(t)}{\hat{\kappa}} \leq (1+\rho)\left[1+\frac{adj_p^i-adj_p^{i-1}}{AI}\right].$$

According to (4.2),

$$-\alpha(\hat{\delta}) \leq | adj_p^i - adj_p^{i-1} | \leq \alpha(\hat{\delta}) \tag{A25}$$

since Hypothesis (ii) stipulates that virtual clocks are synchronized to within $\hat{\delta}$. Therefore,

$$\frac{\hat{c}_p^i(t+\hat{\kappa})-\hat{c}_p^i(t)}{\hat{\kappa}} \leq (1+\rho)\left[1+\frac{\alpha(\hat{\delta})}{AI}\right].$$

Thus, using Hypothesis (iii) and transitivity, we get

$$\frac{\hat{c}_p^i((t+\hat{\kappa})-\hat{c}_p^i(t)}{\hat{\kappa}} \leq (1+\hat{\rho})$$

as desired.

Next, we derive a lower bound for (A21). According to Hardware Rate (2.2), we conclude

$$(1-\rho)\hat{\kappa} \leq c_p(t+\hat{\kappa})-c_p(t). \tag{A26}$$

Using the same argument as above with $-\alpha(\hat{\delta})$ as the lower bound for the value of $adj_p^i-adj_p^{i-1}$ (due to (A25)), we get

$$(1-\rho)\left[1-\frac{\alpha(\hat{\delta})}{AI}\right] \leq \frac{\hat{c}_p^i(t+\hat{\kappa})-\hat{c}_p^i(t)}{\hat{\kappa}}.$$

Thus, using Hypothesis (iv) we get

$$(1-\hat{\rho}) \leq \frac{\hat{c}_p^i((t_1+1)\hat{\kappa})-\hat{c}_p^i(t_1\hat{\kappa})}{\hat{\kappa}}$$

as desired. □

Appendix 2: Glossary of Notation

The section in which each term is defined appears in parenthesis at the end of the entry for that term.

ai Maximum number of real seconds in adjustment interval over which resynchronization is spread (Appendix 1).

AI Number of clock seconds in adjustment interval over which resynchronization is spread (§2).

adj_p^i $\hat{c}_p^i(t) = c_p(t)+adj_p^i$ except during the adjustment interval (§2).

c_p Hardware clock at processor p (§2).

$\hat{c}_p$ Virtual clock at processor p (§2).

$\hat{c}_p^i$ i^{th} virtual clock at processor p (§2).

$\bar{c}_p$ Virtual clock at processor p using instantaneous resynchronization (Appendix 1).

$\ddot{c}_p$ Fixed clock at processor p (Appendix 1).

FIX_p^i Correction factor to spread $adj_p^{i-1} - adj_p^i$ over AI and transform c_p into $\hat{c}_p^i$ (§2).

r_{min} Minimum real time between successive events produced by the reliable time source (§2).

r_{max} Maximum real time between successive events produced by the reliable time source (§2).

R Resynchronization interval in clock seconds (§2).

t_p^i Real time p starts $\hat{c}_p^i$ (§2).

t_{RTS}^i Real time the reliable time source generates the i^{th} event (§2).

V_p^i Value provided by reliable time source to p for starting $\hat{c}_p^i$ (§2).

$\alpha_X(\delta)$ Accuracy of convergence function CF_X (§4).

β Maximum real time delay between generation of an event by the reliable time source and detection of that even by a correct processor (§2).

Γ_{max} Maximum delay according to the clock at any correct processor to send a message from one processor to another (§3).

Γ_{min} Minimum delay according to the clock at any correct processor to send a message from one processor to another (§3).

$\hat{\delta}$ Difference between any two correct virtual clocks (§2).

$\bar{\dot{\delta}}_S$ Bound—at the instant both have been started—on the difference between any two identically superscripted correct virtual clocks that use instantaneous resynchronization (§7).

κ Real time tick width for c_p (§2).

$\hat{\kappa}$ Real time width of a tick by $\hat{c}_p$ (§2).

$\lambda_p^i(q)$ Bound on error in p's approximation of $\hat{c}_q^i$ (§3).

Λ Maximum clock reading error for any pair of processors (§3).

μ Upper bound on $c_p(0)$ (§2).

$\pi_X(\delta, \varepsilon)$ Precision of convergence function CF_X (§4).

ρ Upper bound on c_p drift rate (§2).

$\hat{\rho}$ Upper bound on $\hat{c}_p$ drift rate (§2).

$\tau_p^i[q]$ Approximation of $\hat{c}_q^i(t_p^{i+1}) - c_p(t)$ (§3).

Acknowledgments

Discussions with Ozalp Babaoglu, Steve Mahaney, Leslie Lamport, and Sam Toueg have been helpful. In addition, I am grateful to Jacob Aizikowitz, Ozalp Babaoglu, David Gries, Keith Marzullo Mike Merritt, and Rick Schlichting for useful comments on an earlier versions of this paper. The notions of accuracy and precision were developed jointly with Steve Mahaney under a consulting agreement with AT&T Bell Laboratories.

References

[Babaoglu & Drummond 87] Babaoglu, O. and R. Drummond. (Almost) no cost clock synchronization. *Proc. Seventeenth International Symposium on Fault-tolerant Computing*, Pittsburgh, Penn., July 1987, IEEE Computer Society, 42-47.

[Bevington 69] Bevington, Philip R. *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill Book Company, New York, 1969, p. 3.

[Cristian *et al.* 86] Cristian, F., H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance faults, and processor joins. *Proc. Sixteenth International Symposium on Fault-tolerant Computing*, Vienna, Austria., July 1986, IEEE Computer Society, 218-223.

[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.

[Kopetz & Ochsenreiter 87] Kopetz, H. and W. Ochsenreiter. Clock synchronization in distributed real time systems. *IEEE Transactions on Computers C-36*, 8 (August 1987), 933-940.

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

[Lamport & Melliar-Smith 85] Lamport, L. and P.M. Melliar-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.