THE CLASSIFICATION OF FAILURES
LOGICALLY SYNCHRONOUS REPLICAS
FAULT TOLERANT CLOCK SYNCHRONISATION

H R STRONG

Rapporteur: P Ezhilchelvan

FAULT TOLERANCE IN REAL TIME SYSTEMS

Ray Strong
IBM Research
Almaden Research Center
650 Harry Road
San Jose, CA 95120

ABSTRACT: This paper presents three lectures covering aspects of fault tolerance in real time distributed systems. The first lecture is on the classification of failures. It presents a classification of failures of components of distributed systems that is related to the complexity of algorithms required to tolerate them. There is a hierarchy of failure classes of relevance to real time systems including the classes of crash, omission, timing, clock, and Byzantine failures. One failure class is said to cover another if algorithms tolerant of the first class of failures are tolerant of the second. This notion is made precise and it is shown that, when "$\prec$" represents "is covered by," crash $\prec$ omission $\prec$ timing $\prec$ Byzantine, but clock and timing are incomparable. We will discuss attempts at the automatic conversion from algorithms tolerant of one class to those of another. Several open problems in the theory of failure classification are also covered.

The second lecture is on logically synchronous replicas. A method is described for using failure tolerant atomic broadcast protocols to maintain replicated data in a distributed system so that updates are performed at the same clock time at each site. This synchronous replicated data can be used to provide a single global state (including a global time) that facilitates such distributed applications as locking, leader election, load balancing, and common log. Real time requirements on components of a system that provides atomic broadcast are discussed. The increased complexity required of atomic broadcast algorithms that tolerate wider classes of failures is also covered.

The third lecture presents simple failure tolerant clock synchronization al-

gorithms and their real time systems requirements. In order to guarantee a given precision of clock synchronization, it is necessary to have a corresponding bound on the uncertainty of message transmission and processing time. In contrast, significantly better precision is obtainable, with high probability, when no absolute guarantee is required.

CLASSIFICATION OF FAILURES.

In this lecture we classify the faults or failures that occur in the components of a distributed system in order to compare the fault tolerance of distributed algorithms. We will attempt to use the term "fault" to refer to the cause of some behavior on the part of a component that does not meet the specification of the component. The bad behavior itself will be called a "failure."

Since we wish to describe and compare the fault tolerance of algorithms, we will restrict attention to the *input/output* behavior of components of a system. Note that any classification based on such input/output behavior is relative to a particular decomposition of the system into components. The relevant components of our system model are called *processors* and *links*.

To each component is associated a set of possible input events and a set of possible output actions. Included in the possible output actions for processors are the set of possible message transmissions. Included in the possible input events for processors are the set of possible message receipts. (Messages are not decomposed into constituent bits.) In addition to message receipt, the passage of a specific time duration also constitutes an input event for a processor, even if the processor is unable to detect the event.

Here we assume a Newtonian frame of reference including a time dimension called *real time*. We have found such an assumption of great benefit in communicating concepts related to time and timing failures. So, although the material described in these lectures could be handled within a relativistic reference frame, we choose the Newtonian frame because the relativistic frame hides the essence of the work in too many details concerned with relativity.

Input and output events for links are analogous to those for processors.

Each component is assumed to have an input-output *specification* describing its correct response (output) in relation to a history of previous inputs and outputs (v. [DSC]). If the specification is given in terms of state transitions for the component, then, since we are only concerned with input/output behavior, an incorrect state transition that leads to correct output will be considered correct input/output behavior and will not be considered an occurrence of a failure.

We will use the following example specification of a component throughout our definitions of classes of failures. We assume that it is possible for a component to receive two input messages: A and B. Also, there are two possible output messages: C and D. For simplicity, we assume that message A will be received once and only once in the lifetime of the component. If the receipt of message B follows the receipt of A within 10 ms, then the component is to send output message D at some time between 5 and 15 ms after the receipt of A. If the receipt of message B does not follow the receipt of message A within 10 ms, then the component is to send output message C between 10 and 15 ms after the receipt of A. Thus the sequences of events

$$A, 1, 2, 3, 4, B, 5, D, 6, 7, ...$$

and

$$A, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, C, 12, 13, 14, ...$$

represent correct input/output behavior on the part of the component, where a numeral represents the event corresponding to the passage of that number of ms of real time after the receipt of A. However, the sequence

$$A, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...$$

represents a failure called an omission failure.

An *omission* failure is a failure in which a component omits some specified output action. If a component omits all output actions after some point in time, the failure is called a *crash* failure. One cannot distinguish a crash failure from an omission failure or an omission failure from another more complicated type of failure called a late timing failure by looking at any finite history. For example,

$$A, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, C, 17, ...$$

represents a late timing failure. A *timing* failure is a failure in which a component performs some specified action, either too early (*early timing failure*) or too late (*late timing failure*). For example,

$$A, 1, B, 2, 3, 4, 5, 6, D, 7, ...$$

represents correct behavior; but

$$A, 1, B, 2, D, 3, 4, ...$$

represents an early timing failure. We use the term "Byzantine" to emphasize that failures may not belong to any of the restricted classes of failures that have been defined; but, formally, the class of *Byzantine failures* is the universal class of all possible incorrect input/output behavior [LSP].

Note that although it is not possible to distinguish omission from late timing failures by considering a finite history, it is possible to tell whether such a finite history is consistent with an omission only failure model. In other words, we can ask whether the finite history that includes a failure could be explained by an omission failure.

A system is said to *tolerate* a failure on the part of one of its components if the system meets its input/output specification in spite of the failure. The *fault pattern* of a history (execution) of a system is the set of components of the system that exhibit incorrect behavior in the history. The fault pattern of a history is said to belong to failure class K if the history is consistent with a failure model in which the only failures are from class K. A system (or algorithm) is said to *tolerate* a particular fault pattern from failure class K if the system input/output specifications are met in every history exhibiting that fault pattern from failure class K.

Failure class J is said to *cover* failure class K if, for any fault pattern p, any algorithm that tolerates p when it belongs to J also tolerates P when it belongs to K (cf. [DSC]). We use $J \succ K$ to represent the relation J covers K. It is easy to see that

$$Byzantine \succ timing \succ omission \succ crash.$$

In this lecture we will introduce one more class of failures that is relevant to a discussion of real time systems. The definition of this class requires a further decomposition of processor components into clock and other subcomponents. A *clock* failure is a failure of a processor that can be explained by an arbitrary failure of its clock subcomponent, with all other subcomponents working correctly [DSC]. For example the sequence

$$A, 1, C, 2, B, 3, ...$$

represents a clock failure that is not a timing failure, because the action C is not a correct action, simply performed too early.

Let "$\prec$" represent "is covered by." Then it can be shown that

$$\text{crash} \prec \text{omission} \prec \text{late timing} \prec \text{timing} \prec \text{clock or timing}$$

and that each of these coverages is strict, in the sense that none of the covered classes covers its coverer. With respect to this $\prec$ partial ordering, it can also be shown that clock and timing are independent and that early and late timing are independent.

A great deal of work has gone into attempts to provide compilers that will take algorithms tolerant of failure classes lower in this partial order and automatically produce algorithms tolerant of higher classes (the opposite direction being trivial). The recent work of Neiger and Toueg [NT] has been particularly successful in this regard. However, there seem to be time and other resource penalties that must be paid by the products of such transformations. Moreover, many open questions remain. For example, can any algorithm tolerant of any omission fault pattern involving at most k components be transformed into an algorithm tolerant of any timing fault pattern involving at most k components, without a time penalty?

LOGICALLY SYNCHRONOUS REPLICAS.

The purpose of this lecture is to outline a process by which a system consisting of a network of processors and links, each processor possessing its own subcomponent clock, can simulate a system in which a number of processors share the same clock and also share a fault tolerant memory. The simulated system is a useful abstraction whenever interprocessor coordination is required of a distributed system. Simulation will be accomplished by maintaining a replica of the simulated shared memory in the local memory of each processor and by synchronizing updates so that each update is made at the same local clock time at each processor.

We make the following assumptions about the simulating system.

1. The names of processors are distinct and totally ordered.
2. Each processor p possesses a clock C_p that is a monotone increasing function of real time.
3. No correct processor issues the same timestamp twice.

4. There is a constant e such that, for any correct processors p and q, and for any real time t,

$$|C_p(t) - C_q(t)| < e.$$

5. Let p, q, and r be correct processors and let p and q be connected by a correct link. There is a constant d such that, if correct p sends a message to correct q via a correct link at real time u and q receives and finishes processing the message at real time v, then for any correct processor r,

$$0 < C_r(v) - C_r(u) \leq d.$$

In figure 1, we give a program scheme for accomplishing the simulation. This scheme describes a communication protocol called *diffusion*. It is a program scheme rather than a program because most of its significant phrases are uninterpreted. Moreover, the interpretation we give to these phrases depends on the class of faults we wish to tolerate. Our specification for the system is that updates corresponding to requests generated at any correct processor, at any time, be performed in the same order and at the same local clock time at each correct processor, provided the network of correct processors remains connected by correct links. Moreover, we require that any update performed by any correct processor be performed by all correct processors and that the sequence of (update, local time of performance) pairs be the same for all correct processors provided they remain connected by correct links.

We will illustrate the interpretations of the scheme and argue for their correctness on an example system consisting of five processors and six links configured into a square and a triangle that have one link in common. For the failure class of omission failures and for the failure class of clock or timing failures, we will produce interpretations for the program scheme that tolerate any pattern consisting of two faulty components (processor or link). It is well known that a single Byzantine failure in this example network can cause the system to fail to meet its specifications because the network connectivity is not sufficiently high. However, there are cryptographic techniques that are beyond the scope of this lecture that allow the system to tolerate any pattern of failures with high probability (see [CASD] and [DS]).

```
1 Do forever
2       If learn update M with timestamp T
3           then do
4                   forward M to neighbors
5                   schedule M for T + D
6           od
7       Fi
8 Od
```

Figure 1: DIFFUSION BASED ATOMIC UPDATE

The uninterpreted phrases of the program scheme in Figure 1 are "learn update M with timestamp T," "forward M to neighbors," and "schedule M for $T + D$." We must also provide a constant D.

We first give interpretations for omission faults. There are two ways in which the event "learn update M with timestamp T" can occur at a processor: (1) an update request M originates at the processor at local time T, in which case the message (M, T) is prepared and a copy of the message is placed in a history H; or (2) a message (M, T) is received from another processor and is not found in H, in which case a copy of the message is placed in H. The phrase, "forward M to neighbors," is interpreted as the command to send a copy of (M, T) on all links. Finally, the phrase, "schedule M for $T + D$," means to place M in a lexicographically sorted list of update requests to be performed at local time $T + D$, and then at time $T + D$ to perform all updates corresponding to timestamp T in order and to remove all corresponding messages from H.

We leave as an exercise for the reader to show that, if an update M is learned by a correct processor then the corresponding message (M, T) has taken at most 4 hops. Suppose update request M is initiated by processor p at local time T corresponding to real time u. If correct processor r learns M at real time v, then we must have $C_r(v) \leq C_r(u) + 4d < T + 4d + e$. Thus it suffices to take $D \geq 4d + e$.

For clock or timing faults, things are slightly more complicated. We add to the message (M, T) a hop count k that can be used to test the timeliness of arriving messages. For this fault class, we interpret the first phrase as

before for the originating processor except that it produces the message $(M, T, 0)$ instead of (M, T), setting the local variable $k = 0$ in the process. If message (M, T, k) is received by a processor with (M, T) not in H, then, if the local time of receipt is between $T - ke$ and $T + k(d + e)$, the processor is considered to have learned M with timestamp T; otherwise, the message is ignored as *untimely*. If the processor learns M with timestamp T then it puts (M, T) in H as before.

The phrase, "forward M to neighbors," is now interpreted as, "if $k < 4$ then send $(M, T, k+1)$ on all links." The phrase, "schedule M for $T + D$," is interpreted exactly as before.

Now we do not know how to prove the correctness of our algorithm unless we use a $D \geq 4(d + e)$. Such a bound for D is always sufficient, but it is an open question whether it is necessary. (We can show that it is necessary to tolerate a pattern of three clock faults, but not two.)

FAILURE TOLERANT CLOCK SYNCHRONIZATION.

In the previous lecture, we assumed that clocks were synchronized to within a given maximum deviation e. In this lecture, we show how to accomplish such a synchronization so that faults can be tolerated, provided they do not disconnect the network of correct processors.

We will assume the model of the previous lecture, except for the clock synchronization. For this lecture, the assumed subcomponent clocks will be called hardware clocks and denoted H to distinguish them from the logical clocks denoted C, which we will synchronize. We also assume that there is a constant ρ bounding the drift rate of correct clocks as follows:

$$(1 + \rho)^{-1}(u - v) < H(u) - H(v) < (1 + \rho)(u - v).$$

Thus correct clocks proceed within a *linear envelope* of real time. Note that it is usually safe to take $\rho = 10^{-5}$.

The purpose of the program scheme presented in this lecture is to maintain *linear envelope clock synchronization* for logical clocks C so that

1. there are constants α and β with

$$(1 + \alpha)^{-1}(u - v) < C(u) - C(v) < (1 + \alpha)(u - v) + \beta$$

2. and there is a constant ϵ with

$$|C_p^{-1}(T) - C_q^{-1}(T)| < \epsilon.$$

Here $C^{-1}(T)$ is defined as the greatest lower bound of the set of real times t such that $C(t) \geq T$. The constant α is called the *accuracy* and the constant ϵ is called the *precision* of the clock synchronization. The constant β is called the *gap* since it represents the maximum amount by which a logical clock may be adjusted forward instantaneously.

The simplest way to maintain a logical clock is as an offset from the hardware clock. Alternatively, in order to reduce the gap to zero and provide a "continuous" logical clock, we can maintain C as a piecewise linear function of H and amortize the gap over some fixed duration (on H). See, for example, [GS] or [Cri]. For simplicity, we will assume that each processor has a register A and maintains C as $H + A$. Thus we will interpret the assignment $C \leftarrow X$ as $A \leftarrow X - H$. The unamortized gap provides an overestimate of the maximum deviation e between correct clocks as of any real time (the quantity used in the previous lecture). However, for most of each period between resynchronizations, correct clocks are actually within $(1 + \rho)\epsilon$ of each other. The only times when the deviation exceeds this term are times when one clock has been advanced to its ET and another has not.

Our clock synchronization algorithms require each processor to maintain a constant $PERIOD$ and a register ET in which it stores the next expected logical time to synchronize. We assume that the set of times at which processors begin executing the algorithm is contained in a real time interval as short as the maximum time required to diffuse information in the network. This time would be measured as at most $4d$ on the hardware clock of any processor in our example from the previous lecture. The program scheme for clock synchronization is given in figure 2. Note that it strongly resembles the program scheme of the previous lecture. This is no accident, since both schemes are instances of the diffusion communication protocol.

There are two uninterpreted phrases in the scheme: "learn Time after ET" and "forward Time after ET to neighbors." In the case of omission faults, there are two ways to "learn Time after ET." This happens either when $C = ET$ or on receipt of message (T) with $T = ET$. The phrase, "forward

```
1  C ← 0
2  ET ← PERIOD
3  Do forever
4        If learn Time after ET
5             then do
6                  forward Time after ET to neighbors
7                  C ← max(C, ET)
8                  ET ← ET + PERIOD
9             od
10       Fi
11 Od
```

Figure 2: DIFFUSION BASED CLOCK SYNCHRONIZATION

Time after ET to neighbors," is interpreted as send the message (ET) on all links. For omission faults the algorithm achieves an optimal accuracy of ρ and gap and precision of approximately $4d + 2\rho PERIOD$.

For clock or timing faults, we again introduce a hop count variable k. The phrase, "learn Time after ET," is interpreted as either the event $C = ET$ with side effect $k \leftarrow 0$ or the receipt of message (T, k) when $T = ET$ and $C < T - k\epsilon$. The phrase, "forward Time after ET to neighbors," is interpreted as "if $k < 4$ then send message $(ET, k + 1)$ on all links" for our example network. The accuracy for clock or timing faults is no longer optimal:

$$\alpha = \rho + \frac{4\epsilon}{PERIOD}.$$

The gap is

$$\beta = \frac{4\epsilon}{PERIOD}.$$

The precision is the same as that for omission:

$$\epsilon \approx 4d + 2\rho PERIOD.$$

Note that the actual precision and the maximum deviation (for times when all correct clocks have the same value for ET) can be computed from the following exact formula:

$$(1 + \rho)\epsilon = 4d + (2 + \rho)\rho PERIOD.$$

Srikanth and Toueg [ST] show how to obtain optimal accuracy for a fault model that covers clock and timing faults at the cost of worse precision. It is an open question whether optimal precision and accuracy can both be obtained from the same algorithm.

In the rest of this lecture, we will investigate further the questions of optimal precision. For simplicity, we will assume that there are only two processors in our network and that the two clocks do not drift with respect to each other ($\rho = 0$). We will also assume that the constant δ represents the difference between the maximum and the minimum real times required for message transmission and processing between the two processors. This constant is called the *uncertainty*. In this case, it is easy to modify our algorithms so that the worst precision possible is $\delta/2$. There is a simple argument given by Dolev, Halpern, and Strong [DHS] that shows that any clock synchronization algorithm must have a worst case precision of at least $\delta/2$ in this case.

In this simple case, the problem is one of setting one clock by another. Since there is no drift, we can depend on the initial setting. We can accomplish this setting and achieve optimal precision ($\delta/2$) using a single "start" message. The "start" message is sent from the first processor to the second. The first processor sets its logical clock to 0 when it sends the message. The second processor sets its logical clock to $\delta/2$ when it receives the message. Unfortunately, the uncertainty is often much, much larger than the expected message delay; so our guaranteed precision is much worse than that achievable in practice, with high probability.

To take advantage of the probably faster message time, consider the following round trip experiment. The first processor places timestamp U on a message, using its hardware clock, and sends the message to the second processor. The second processor places timestamp T on the message, using its logical clock, and returns the message to the first processor. The first processor adds a second timestamp V to the returned message using its hardware clock. If the first processor now sets its logical clock to $T + (\delta/2)$ then the optimal worst case precision is achieved as before. However, if the first processor sets its logical clock to $T + ((V - U)/2)$, then a precision of $((V - U)/2$ is achieved (v. [Cri]). Moreover, we can have the best of both worlds by setting the logical clock of the first processor to

$T + min(((V-U)/2), (\delta/2))$, in which case the precision achieved is

$$\epsilon = min(\frac{V-U}{2}, \frac{\delta}{2}).$$

Cristian [Cri] suggests repeating this experiment several times to make the probability high that the best precision achieved is much less than $\delta/2$. Note that this approach does not even require a finite uncertainty, provided the expected message delay is finite.

Acknowledgements.

Most of the material of these three lectures is adapted from the paper by Cristian, Aghili, Strong, and Dolev [CASD]. The diffusion based clock synchronization algorithms were obtained by a straightforward application of the techniques of the above paper to the clock synchronization algorithm of Halpern, Simons, Strong, and Dolev [DHSS]. The approach of providing probabilistic guarantees of clock synchronization precision based on repeated round trip measurements was proposed by Cristian in [Cri]. The material has been modified in response to very helpful questions posed when these lectures were presented at the University of Newcastle upon Tyne. Special thanks are also due to Flaviu Cristian for helpful comments on an earlier version of this manuscript.

References.

[CASD] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: from simple message diffusion to Byzantine agreement," Digest of Papers, FTCS-15, 200-206, Ann Arbor, 1985, see also IBM Research Report RJ5244, July 30, 1986.

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

[DHS] D. Dolev, J. Halpern, and R. Strong, "On the Possibility and Impossibility of Clock Synchronization," *JCSS* 32, pp. 230-250, 1986.

[DHSS] D. Dolev, J. Halpern, B. Simons, and R. Strong, "Dynamic Fault-Tolerant Clock Synchronization," IBM Research Report RJ6722,

March 3, 1989. See also "Dynamic Fault-Tolerant Clock Synchronization," PODC-3, pp. 89-102, Vancouver, 1984.

[DS] D. Dolev, and R. Strong, "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal of Computing*, vol. 12, no. 4, pp. 656-666, 1983.

[DSC] D. Dolev, R. Strong, and F. Cristian, "Distinguishing Timing Failures From Clock Failures," Manuscript in Progress, 1989.

[ES] P. Ezhilchelvan and S. Shrivastava, "A Classification of Faults in Systems," Technical Report, University of Newcastle Upon Tyne, 1989.

[GS] A. Griefer, and R. Strong, DCF: Distributed Communication with Fault-tolerance, PODC-7, pp. 18-27, Toronto, 1988.

[LSP] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382-401, July 1982.

[NT] G. Neiger and S. Toueg, "Automatically Increasing the Fault-Tolerance of Distributed Systems," Proceedings, PODC-7, pp.248-262, Toronto, 1988.

[ST] T. Srikanth and S. Toueg, "Optimal Clock Synchronization," *JACM* 34, 626-645, 1987.

DISCUSSION

Rapporteur: Paul Ezhilchelvan

First Lecture

On the causes of system failure, Professor Randell pointed out that a system can fail without any of its components failing (implying that a system failure can also be caused by a design fault). The speaker answered to Professor Anderson by saying that the term "Byzantine" is used to refer to any failure not just two-facing failures. Professor Kopetz asked about the components that can potentially suffer clock failures; the speaker pointed out that when he defines a clock fault for a component, he assumes clock to be a sub-component in that component. Professor Anderson wondered why "value" faults are not considered in the classification presented. Value faults, in the speaker's opinion are not so relevant to real-time issues as the types of faults he talked about. When Professor Randell passed a remark that semiconductor memory faults can cause a processor to produce erroneous values, the speaker replied that such occurrences are less likely in IBM machines which are in his mind when he presented the fault classification.

Second Lecture

Professor Ercoli asked the speaker to highlight on his definition of clock-precision which, according to the speaker, turned out to be the real-time interval within which any two good clocks will read the same value. Professor Turski wondered whether there is any mechanism by which processors in the system know of failures in the system. The speaker denied assuming any such mechanism and reiterated that any number of failures can occur and that non-faulty processors must remain connected. At this juncture, Professor Randell expressed his opinion, and the speaker agreed, that the problem of network partitioning is being considered as a seperate problem. In reply to Professor Kopetz's question, the speaker mentioned that the clock precision he was able to achieve was in the order of tens of milliseconds and that this precision could be improved. Professor Nehmer remarked that the shared memory system the speaker is attempting to construct has been abandoned thirty years ago by IBM itself and the reply was such a notion is not completely true. Professor Shrivastava expressed his opinion that the replicas are indeed synchronous as opposed to logically synchronous.

Third Lecture

Professors Wells and Kopetz asked whether the clock increments will always be positive and the speaker replied that the required increment can turn out to be either way and there are algorithms designed to make increments to be only positive. The speaker replied to Dr. Chris Holt by saying that the clock drift factor has to be estimated through a series of experiments. Professor Randell got it clarified that the speaker refers to a processor failure that can be explained in terms of the processor's faulty clock as a clock failure.