Reconfiguration and Transient Recovery in State-Machine Architectures*

Submitted for publication

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Rushby@csl.sri.com

Phone: +1 (415) 859-5456 Fax: +1 (415) 859-2844

Abstract

We consider an architecture for ultra-dependable operation based on synchronized state machine replication, extended to provide transient recovery and reconfiguration in the presence of Byzantine faults.

The architecture allows processors suspected of being faulty to be placed on "probation." Processors in this status cannot disrupt other processors, but those that are nonfaulty or recovering from transient faults are able to remain synchronized with the other processors and with each other, can participate in interactively consistent exchange of data (i.e., Byzantine Agreement), and can restore damaged state data by loading majority-voted copies from other processors. The processors that are not on probation are able to coordinate membership of their group and to take processors on and off probation. These properties are achieved even if all the processors on probation and some of the others exhibit Byzantine faults.

Key elements of the architecture are modified treatments for the problems of interactive consistency, clock synchronization, and group membership. Classical algorithms for these problems that tolerate t Byzantine faults among n processors are extended to tolerate $t + p$ faults among $n + p$ processors, partitioned into n "core members" and p "probationers," provided no more then t faults occur among the core members.

Keywords: Reconfiguration, transient recovery, state-machine replication, Byzantine faults

*This work was partially supported by NASA Langley Research Center under Contract NAS1-18969

1 Introduction

There are two basic approaches to the design of systems that can tolerate multiple faults: the system must either have sufficient redundancy that it can mask multiple simultaneous faults, or it must diagnose faulty components[1] and reconfigure to exclude them from the system before additional faults arrive. The advantage of diagnosis and reconfiguration over multiple-fault masking is that it requires far less redundancy to tolerate the same number of faults. For example, a multiple-fault masking system requires $2t + 1$ components to tolerate t simultaneous faults in those elements of its architecture that use majority voting, and $3t + 1$ components in those elements that require clock synchronization or interactive consistency (under "Oral Messages" assumptions). A system based on reconfiguration could tolerate the same number of faults in the corresponding elements of its architecture with only $t + 2$, and $t + 3$ components, respectively. Even one that (as here) focuses only on increasing the resilience of clock synchronization and interactive consistency can reduce the number of components required from $3t + 1$ to $2t + 1$ for $t > 1$.

One serious disadvantage of the reconfiguration approach, however, is that it assumes that faults arrive sequentially and that a faulty component can be diagnosed and excluded before the next fault arrives: the approach will fail if several faults arrive in close proximity, or if diagnosis is inaccurate or incomplete. Unfortunately, it is provably impossible to achieve both accurate (no nonfaulty components are identified as faulty) and complete (all faulty components are identified) diagnosis in the presence of Byzantine faults [22].

Experience shows that transient faults are far more common than permanent ones. A transiently faulty component is one that malfunctions for a while and then spontaneously recovers. However, the processor in such a recovered component may have corrupted its state data while it was malfunctioning (or the actual fault may have been a bit-flip in memory—this is indistinguishable from the residue of a malfunction), and subsequent calculations based on these corrupted data will perpetuate and spread the contamination. This contamination can be repaired under a variation of the multiple-fault masking approach in which components periodically replace portions of their state data with majority-voted versions from other components.

[1] *Diagnosis* is used here to refer simply to the identification of faulty components; it does not imply identification of the type of fault that afflicts the component. A *component* is used here to refer to an active computing element that includes a processor and local memory.

This method of transient recovery depends on a recovering component being allowed to remain part of the system. In contrast, the standard reconfiguration approach would identify a transiently-faulty component as faulty and permanently exclude it from the system. This wastes a potentially useful resource and may require additional redundancy to compensate for its loss, thereby vitiating the main benefit of reconfiguration. A more elaborate reconfiguration scheme might allow an excluded component to be readmitted if it could pass certain tests, but such a scheme would have to provide some way to bring the component's state data up to date before it is readmitted. Such an architecture could become rather complicated, thereby increasing the likelihood of design faults in its own mechanisms for redundancy management.

In summary, multiple-fault masking can provide an attractive treatment for transient faults, but requires a lot of redundancy to mask multiple permanent faults, whereas reconfiguration provides good treatment for permanent faults, but is less attractive for transient faults. In this paper, we present a design that combines the attractive features of both these approaches with few of their disadvantages.

The paper is organized as follows. The traditional architecture for state machine replication with transient recovery is described in the next section. The proposed architecture with limited reconfiguration is described in Section 3 and its main algorithms, those for "extended" interactive consistency, are given in Section 4. The other algorithms required are outlined in Section 5 and conclusions presented in Section 6.

2 State Machine Replication with Transient Recovery

Our focus is the approach to fault-tolerant computing first implemented in SIFT [27] and subsequently refined by MAFT [12] and several other projects (e.g., [13]). This has become known as the *state machine* approach to fault-tolerant computing [21].[2]

A frame-synchronous architecture based on state-machine replication operates as follows. There are n replicated major components, generally called *channels*, that are electrically and physically isolated from each other so that their failures will, to the extent possible, be independent. Each channel has a processor, some memory, and access to sensors and actuators. The channels execute the same software and are synchronized so that they perform actions at approximately the same time. At

[2] Schneider's tutorial [21] describes the approach in its client-server form. Here, we use the original SIFT form (which is suited to control applications) where both clients and servers are combined into a single replicated application.

the beginning of each "frame" of activity, all channels sample their sensors and then exchange their sensor readings in a manner that ensures *interactive consistency* (also called "source congruence" or "Byzantine Agreement"), meaning that all nonfaulty channels are guaranteed to have identical sets of sensor readings. All channels then execute the same sensor conditioning and selection software to generate inputs for the control laws. All channels evaluate the same control laws and their outputs are then subjected to exact-match majority voting before being sent to the actuators. In order that voting should not become a single point of failure, each channel usually has its own voter, and the voted values from each channel are then further voted or averaged at the actuators through some form of "force-summing." For example, the outputs of different channels may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft [6, Figure 3.2–2].

There are three key elements to this architecture.

1. Clock synchronization ensures that all nonfaulty channels can coordinate their actions and communications using a common notion of time. Synchronization between nonfaulty channels must be achieved in the presence of disruption caused by faulty channels. The number of faulty channels that can be tolerated depends on the algorithm, assumptions, and fault model employed, but under the weakest assumptions it is fewer than one-third [10, 14].

2. Interactive Consistency ensures that all nonfaulty channels start their computations with identical sets of sensor readings. This also must be achieved in the presence of faults; the requirements are that all nonfaulty channels have a correct view of each others' sensor readings, and a common view of the sensor readings communicated by faulty channels.[3] As with clock synchronization, this can be achieved under the weakest assumptions only if fewer than one third of the channels are faulty. An important parameter to all algorithms for interactive consistency is the number of *rounds* of information exchange that they perform; no r-round algorithm can withstand more than r simultaneous Byzantine faults [5, 8]. Resource constraints on communication usually restrict the number of rounds to some small fixed number—generally one.

[3] Once the channels have exchanged sensor readings, they can compare different sensors (or the same sensor sampled by different channels or at different times) in order to compensate for inaccurate or faulty sensor readings. This aspect of fault tolerance addresses different issues than state-machine replication and is ignored here.

3. Exact-match majority voting of actuator outputs masks faults in execution of the main computational tasks. Because interactive consistency ensures that all nonfaulty channels start off with the same set of sensor readings, and because they all perform identical sensor conditioning and selection and the same control law calculations, all nonfaulty channels will produce identical outputs. Therefore exact-match majority voting will mask the outputs of faulty channels, provided that fewer than half the channels are faulty [27].

2.1 Multiple-Fault Tolerance

Notice the different number of faulty components tolerated by the different elements of this architecture: fewer than a third for clock synchronization and interactive consistency, and fewer than a half for the computational tasks. Thus, for example, a five-channel system can tolerate two faulty channels in the computational tasks, but only one in clock synchronization and interactive consistency. If we require a two-fault-tolerant system, then we must provide seven channels for clock synchronization and interactive consistency, and must use a two-round algorithm for the latter.

These calculations are based on worst-case assumptions about the behavior of faulty components (i.e., the *Byzantine* fault model). It is possible to tolerate more faulty components in clock synchronization and interactive consistency if some of the faults are "not too bad." *Hybrid* fault models and their associated algorithms consider both Byzantine and less serious kinds of faults occurring in combination [17, 20, 23]; *authenticated* algorithms use digital signatures to constrain the behavior of faulty components [5,15]. These approaches (which can be used in combination [9]) increase the total number of faults that can be tolerated in clock synchronization and interactive consistency, but Byzantine faults can never exceed one third of the components in clock synchronization, nor the number of rounds in interactive consistency [8, 10].

2.2 Transient Recovery

As noted, transient faults are many times more common than permanent faults. When a channel whose processor has been afflicted with a transient fault recovers its ability to execute programs correctly, its overall fault status depends on the extent to which its state data has been corrupted and on the task being performed.[4] A task

[4] Programs are assumed to be stored in ROM and to be immune to corruption.

that uses only current sensor data (which are refreshed every frame) or uncorrupted state data will be performed correctly; one that uses corrupted state data will be performed incorrectly, and may then spread the corruption by storing results of the incorrect calculation in previously uncorrupted locations.

Clock synchronization and interactive consistency rely on little or no state data[5] and are therefore very likely to be performed correctly by a channel whose processor has recovered from a transient fault. The main computational tasks, however, will access state data (otherwise the data are not needed); those of them that reference corrupted data will be performed incorrectly and may spread the contamination. Thus, the contaminated state data that is the residue of a transient fault leads to *permanent* fault effects.

It follows that a number of sequential transient faults may be tolerated perfectly well by single-fault-tolerant mechanisms for clock synchronization and interactive consistency, while permanently contaminating the state data of several channels and thereby exhausting the multiple-fault-tolerance provided by majority voting of the main computational tasks.

This undesirable scenario can be overcome by causing all channels to exchange their private state data periodically; each channel then performs a majority vote on the exchanged data and replaces its private data by the majority value. Since the amount of state data may be large, only portions of it are generally exchanged and voted at each stage, so that it may take several frames to refresh the entire state. In this way, a channel that is recovering from a transient fault (i.e., one whose processor is calculating correctly, but that has corrupted state data) will be able gradually to repair its state. Provide that enough channels remain nonfaulty at all times, this approach provides self-stabilizing recovery from transient faults in the main computational tasks [4, 19].

2.3 Residual Weaknesses

State machine replication, augmented to provide transient recovery in the manner just described, is an attractively simple architecture that provides robust fault tolerance against transient Byzantine faults that arrive sequentially. However, practical

[5]Clock synchronization generally requires the local clock adjustment plus the frame counter. Oral Message algorithms for interactive consistency require nothing; authenticated algorithms can require a significant quantity of data to maintain crypto-synch and to foil spoofing attacks [9], which militates against their use in combination with transient recovery.

concerns limit the level of redundancy and the number of rounds that can be provided, so the architecture is vulnerable to the arrival of multiple faults in close proximity. For example, a five-channel, one-round system cannot tolerate more than one active Byzantine fault in its mechanisms for interactive consistency or clock synchronization. A permanent fault, or a transient fault that is slow to clear, exposes the system to total failure should another fault arrive. This vulnerability could be reduced if the system reconfigured to exclude the faulty component. (The resulting four-channel, one-round system can tolerate an additional Byzantine fault.) The next section introduces mechanisms to accomplish this.

3 Adding Reconfiguration: The Proposed Architecture

Our goal in extending the architecture described in the previous section is to retain most of its good features, while overcoming its weaknesses. One of its good features is the self-stabilizing approach to transient recovery: transiently faulty components remain in the system and are thereby able to repair their state data and to recover their health. However, this is also the source of the principal weakness in the architecture: because faulty components remain in the system, those with permanent faults, or transient faults that are slow to clear, seriously degrade its ability to mask additional faults in clock synchronization and interactive consistency.

We could attempt to overcome this weakness by diagnosing permanently faulty components and reconfiguring to exclude them. However, this approach would place great reliance on the accuracy and completeness of diagnosis, and on its ability to discriminate between transient and permanent faults, which is hard to justify in the presence of arbitrary (i.e., Byzantine) faults.

Our chosen approach takes a middle course between leaving faulty components in the system and reconfiguring to exclude them completely. Components suspected of being faulty can be placed on *probation*, which is a limbo state between full inclusion in the system and complete exclusion. Those components that are not on probation constitute the current *core* of the system. A requirement of the architecture is to ensure that the core comprises sufficiently many nonfaulty components that it can achieve interactive consistency and clock synchronization among its members; a desirable goal is to arrange matters so that the core can tolerate *additional* faults among its members. Core members suspected of being faulty can be moved to the probationary status. Faults among the probationers have no impact on the core, but those probationers that are nonfaulty are able to "shadow" the core so that they can establish and maintain clock and state data synchrony with the core. Furthermore, nonfaulty probationers contribute to the majority vote on actuator outputs and are able to contribute sensor data to the core in a reliable manner. Probationers that have recovered from a transient fault and appear to be working correctly can be readmitted to the core.

In more detail, the architecture that we propose is state-machine replication with transient recovery, extended as follows.

- All channels maintain in their state a record of the current core members.
- The core members maintain clock synchronization among themselves using a standard algorithm. Nonfaulty probationers maintain synchronization by "listening in" to the execution of this algorithm (in a manner described in Section 5.1).
- The exchange of single-source data such as sensor samples and diagnostic information is performed using an algorithm for "extended" interactive consistency, as described in the next section.
- All channels contribute to the majority voting of actuator outputs and participate in the periodic exchange of state data and update their own state data with majority voted values.
- All channels periodically exchange diagnostic "syndrome" information in an interactively consistent manner and use this to modify their estimate of those channels that should constitute the core.

Let the number of faults that can be tolerated by the basic interactive consistency and clock synchronization algorithms be t_i and t_c among n_i and n_c channels, respectively. Then this architecture masks all faults provided no more than a minority of channels are faulty at any one time, and provided the core always contains $min(n_i, n_c)$ channels and no more than $max(t_i, t_c)$ faulty ones. For example, using standard algorithms for clock synchronization and one-round interactive consistency, we can tolerate one fault among four channels. A seven-channel implementation of the proposed architecture can therefore tolerate as many as three simultaneous faults, provided that the core always contains at least four channels and no more than one of them is faulty at any time.

This extended architecture depends on algorithms for *diagnosis* (to identify faulty and recovered channels) and *group membership* (to coordinate membership of the core), and on modified treatments of clock synchronization and interactive consistency (to allow probationers to participate without disrupting the core). It is the modified treatment of interactive consistency that is central to the development of the proposed architecture, so we introduce it first.

4 Extended Interactive Consistency

We begin be describing the classical problem of Interactive Consistency, and the Oral Messages algorithm. We then describe the extended problem and modified algorithms.

4.1 Classical Interactive Consistency

In the classical problem of *Interactive Consistency* [18], there are n processors, of which some number t may be faulty. Each processor p has some private value (e.g., from sampling a sensor) ν_p and the goal is to distribute these private values to all the other processors so that each processor q can form its estimate $V_q(p)$ of processor p's private value, satisfying the following conditions:

Agreement: If processors q and r are nonfaulty, then they agree on the value ascribed to any other processor p; that is: $V_q(p) = V_r(p)$.

Validity: If processors p and q are nonfaulty, then the value ascribed to p by q is indeed p's private value; that is, $V_q(p) = \nu_p$.

Interactive Consistency is the symmetric version of *Byzantine Agreement*: in the latter problem, there is a distinguished processor called the *transmitter* (it is called the "Commanding General" in the metaphor of the Byzantine Generals [15]) and the goal is to distribute its private value to all the other processors (which are called *receivers*) in a manner that ensures Agreement and Validity. An algorithm for Byzantine Agreement can be converted to one for Interactive Consistency by simply iterating it over all processors (each in turn taking the role of the transmitter). It is usually simpler to describe algorithms in their Byzantine Agreement form, and we will follow this practice here.

Algorithms for Interactive Consistency can be developed under various *fault models* and *message assumptions*. The most robust of these are the *Byzantine* fault model (there are no assumptions on the behavior of a faulty processor), and *Oral Messages* (faulty processors can manufacture messages purporting to come from other processors, and can change any messages that they forward from one processor to another). The r-round algorithm OM(r) can solve the problem under these assumptions, provided $r \geq t$ and $n > 3r$ [15].

The algorithm is described recursively; the base case is OM(0).

OM(0)

1. The transmitter sends its value to every receiver.

2. Each receiver uses the value obtained from the transmitter, or some arbitrary, but fixed, value if nothing is received.

Next, we describe the general case.

OM(r), $r > 0$

1. The transmitter sends its value to every receiver.

2. For each p, let v_p be the value receiver p obtains from the transmitter, or else be some arbitrary, but fixed, value if it obtains no value. Each receiver p acts as the transmitter in Algorithm OM($r-1$) to communicate this value v_p to each of the $n-2$ other receivers.

3. For each p, and each $q \neq p$, let v_q be the value receiver p obtained from receiver q in step (2) (using Algorithm OM($r-1$)), or else some arbitrary, but fixed, value if nothing was received. Each receiver p calculates the majority value among all values v_q it receives, and uses that as the transmitter's value; if no absolute majority exists, then p uses some value functionally determined by the set of v_q's.

This algorithm is optimal within its class: all algorithms based on oral message assumptions require at least $3t+1$ components and t rounds to tolerate t Byzantine faults. Other algorithms, based on different assumptions, can withstand more faults for the same number of components but none can do so in fewer rounds.

4.2 The Extended Interactive Consistency Problem

In the extended version of Interactive Consistency, there are again some number n of processors, but this time they are partitioned into p *probationers* and c *core members* (so that $n = p + c$); some number t of core members are faulty, and possibly all of the probationers. The goal is to exchange private values among all processors (both probationers and core members) in a manner that satisfies both Agreement and Validity. Thus the Extended problem is like classical Interactive Consistency, except that we have to tolerate as many as $t + p$ faults, given that no more that t of them are among the core members. The basic idea is that if core members are able to tolerate the t faults among themselves, then they can act as a fault-tolerant distribution mechanism to and from the probationers.

In the following subsections, we describe modifications to the classical OM algorithm that solve the extended interactive consistency problem; later we consider similar modifications to other algorithms.

4.2.1 Extended Oral Messages

The extended algorithms are described in their Byzantine Agreement formulation (i.e., where there is a distinguished transmitter). We begin with the case where the transmitter is one of the core members.

4.2.1.1 The Majority Vote Algorithm. In this algorithm, the core members run OM(r) among themselves. They will achieve Validity and Agreement provided $r \geq t$ and $c > 3r$. Each member of the core then sends its estimate of the transmitter's value to each probationer (the transmitter sends its own value). The inequalities above ensure that a strict majority of the core members are nonfaulty (i.e., $c > 2t$). Each nonfaulty probationer can therefore perform a majority vote on the values that it receives in order to decide on the same value used by the nonfaulty members of the core. □

A disadvantage of this algorithm is that it requires an additional round of broadcast messages (from each member of the core to each probationer) following those required for OM(r). Because message rounds are the critical resource in many implementations of state machine replication, this reduces the utility of the algorithm. Fortunately, it is possible to avoid this extra cost in the important case $r = 1$.

4.2.1.2 Optimization for $r = 1$. Here, the transmitter sends its value to core members only. Each member of the core then sends the value received to all other core members and to the probationers. Each receiver (both core members and probationers) then selects the majority value among those received. The key to this algorithm is that the probationers do not receive a value directly from the transmitter.

For core members, this algorithm is just OM(1), and it works as long as $1 \geq t$ and $c > 3$. For probationers, the argument goes as follows. If the transmitter is nonfaulty, each probationer gets at least $c - 2$ good values and at most one bad one (since there can be at most one faulty processor among the core). The good value wins the majority vote provided $c - 2 > 1$ (i.e., $c > 3$). If the transmitter is faulty, then all the other core members must be nonfaulty, and they will send the probationers the same values as they send each other. Thus both core members and probationers receive identical sets of values in the forwarding round. Although no majority need exist among these values, the selection algorithm is deterministic, so all receivers will decide on the same value. □

Next, we consider the case where a probationer is the transmitter. As with the previous case, we propose a straightforward extension to the existing algorithm.

4.2.1.3 The Extra-Round Algorithm. Assume the basic algorithm is OM(r). The transmitter (a probationer) sends its value to every core member. The core members then run OM(r) on those values to achieve Interactive Consistency on the set of values received, and each selects the majority value in that set (or some value functionally determined by the set if no majority exists). These values are then distributed to the other probationers as in the majority vote algorithm.

We prove that this algorithm achieves Agreement and Validity provided $r \geq t$ and $c > 3r$. Observe that the algorithm is actually the same as OM($r + 1$) on the $c + 1$ processors comprising the core plus the transmitter, followed by distribution to the probationers as in majority vote algorithm. There may be as many as t faults among the core, and the transmitter may be faulty, so to achieve Interactive Consistency among the core we need to withstand $t + 1$ faults with $r + 1$ rounds among $c + 1$ processors. We have $r \geq t$ and $c > 3r$, which ensures $r + 1 \geq t + 1$, but not $c + 1 > 3(t + 1)$ as required for OM($r + 1$). That is, we have enough rounds, but possibly insufficient processors, so we must undertake further analysis.

Notice that even with a nonfaulty transmitter, OM($r+1$) can fail in circumstances where OM(r) succeeds (consider $r = 1$ and four processors with a Byzantine fault among the receivers). This cannot happen in the circumstances of this algorithm, however. Provided $c > 3t$, there are sufficient processors among the core to reach interactive consistency on the value received from the transmitter by each member of the core. If the transmitter is nonfaulty, the nonfaulty core members will each receive the correct value and this will win the majority vote provided $c - t > 2t$ (there are at least $c - t$ nonfaulty core members and at most t faulty ones) and this follows from $c > 3t$. If the transmitter is faulty, we only require Agreement and this follows because the nonfaulty members of the core each have the same set of values and the selection process is functionally determined by this set.

Correctness of the step that distributes the value from the core to the other probationers follows by the argument used for the majority vote algorithm. □ .

This algorithm has the disadvantage of requiring two extra rounds of messages: one to distribute the transmitter's value to the core, and one to distribute the chosen value from each core member to each probationer. There seems no way to eliminate the first of these because, in the worst case, we need to tolerate $t + 1$ faults (the transmitter plus t of the core members) and this cannot be done in fewer than $t + 1$ rounds. The extra rounds for the distribution to the probationers algorithm can be eliminated in when the basic algorithm is OM(1) using the optimization of Section 4.2.1.3. Further optimizations may be possible using techniques such as those of Echtle [7].

4.2.2 Other Interactive Consistency Algorithms

Interactive consistency algorithms derived from OM, such as OMH for the hybrid fault model [17], and ZA for the authenticated case [9], can be extended in exactly the same way as OM, since they are also based on majority voting (albeit in modified form). It is not clear whether algorithms such as the classical Signed Messages algorithm SM [15] that do not use majority voting can be extended in a similar manner.

4.3 Summary

Interactive consistency algorithms that use majority voting, such as OM, can be extended to tolerate an arbitrary number of faults, provided there no more than t faults among those components identified as the core, where t is the number of faults tolerated by the basic algorithm. It is necessary for the core to agree on its membership, but does no harm if other (faulty) channels "think" they are in the core.

In the case of one-round algorithms (where one fault can be tolerated among the core, provided this contains four or more members), there is no additional cost to the extended algorithm when the transmitter is a member of the core. If we are prepared to do without the values of sensors attached to probationers (which seems reasonable, since probationers are presumed faulty and are unlikely to provide useful data), then the only time that a probationer needs to be the transmitter in interactive consistency (thereby incurring the cost of an additional round of information exchange) is when it provides its diagnostic syndrome information (with a view to being readmitted to the core). This can probably be done far less often than every frame, thereby minimizing its cost.

5 Clock Synchronization, Diagnosis, and Group Membership

In this section, we briefly describe issues in these additional elements of the architecture.

5.1 Clock Synchronization

Unlike interactive consistency, where it is necessary for probationers to participate fully (i.e., as transmitters), clock synchronization can use an unequal arrangement. The idea is that the core members synchronize among themselves using an algorithm such as interactive convergence [14], ignoring inputs from probationers (in the case of interactive convergence, this essentially means treating their clock differences as if they are zero). Probationers, run the algorithm *as if* they were members of the core (in the case of interactive convergence, this means they ignore clock differences from other probationers and base their own corrections on a fault-tolerant average of their clock differences with the members of the core).

Provided the core contains enough channels to mask faults among its own members, this arrangement allows all nonfaulty processors to remain synchronized (though the synchrony may be less close between probationers than between core members). Full analysis is omitted here due to space limitations.

5.2 Diagnosis

Diagnosis of faulty components in state-machine architectures can be based on the observation that all nonfaulty channels perform identical calculations on identical data, and must therefore generate identical outputs. Majority voting is sufficient to mask faults among a minority of channels, and any channel whose output differs from the majority must be faulty. If the channels exchange their output values, any nonfaulty channel that receives a value different than its own from some other channel may correctly conclude that the other channel is faulty. This approach is complicated, however, by the need to achieve consensus (in order to decide membership of the core) and by the fact that Byzantine faulty channels may send correct values to some channels and incorrect ones to others.

Diagnosis becomes consistent (all nonfaulty channels agree on the fault status of all channels) and accurate (no nonfaulty channel is declared faulty) if an interactive consistency algorithm is used to distribute the output values (subject to number of faults being within the tolerance of the algorithm, number of channels, and number of rounds concerned). However, this method of diagnosis is not complete: not all faulty channels will be identified. It is impossible to achieve both accurate and complete diagnosis in the presence of Byzantine faults [22]. For example, if a faulty channel sends an incorrect value to one nonfaulty channel and the correct value to all others, then the receiving channel will report the incorrect value but the other channels will have insufficient evidence to decide whether this is the correct report of an incorrect value or the incorrect report of a correct one and will be unable to conclude which of the two channels is faulty.

It is possible to improve the completeness of diagnosis by accumulating evidence over time. For example, if channel a reports that channel x sent an incorrect value in one frame, and channel b that it sent an incorrect value in another, then we have two accusations against x, which is sufficient to accurately diagnose it as faulty under a single-fault assumption. Several authors develop several sophisticated diagnosis algorithms of this type [1, 25, 26]; the accuracy of some of these has been formally verified by Lincoln [16].

This approach can be *in*accurate in the presence of transient faults, however. For example, in the scenario just given, a and b may be transiently faulty in the (different) frames when they accuse x of sending an incorrect value. Although no more than a single fault is active at any one time, the nonfaulty channel will be inaccurately diagnosed as faulty. In an architecture where reconfiguration permanently excludes channels diagnosed as faulty, inaccurate diagnosis can be very harmful (since it leaves a faulty channel operating while removing a nonfaulty one).

In the architecture proposed here, the consequences of inaccurate diagnosis are less severe, because a channel diagnosed as faulty is merely placed on probation and can be readmitted to the core if it subsequently appears nonfaulty. In these circumstances, it may be preferable to trade accuracy for completeness and to quickly place a merely suspect channel on probation, where it can be made to undergo self-tests or a reboot, rather than to favor accuracy and to leave it in the core with the hope of eventually obtaining unequivocal evidence of its faultiness. Lincoln describes a diagnosis algorithm called UD that favors completeness over accuracy [16].

In general, however, it seems that diagnosis algorithms should consider the use to be made of their results, rather than strive for abstract accuracy or completeness [3]. For example, in the architecture proposed here, diagnosis will be used in group membership, and the crucial factor in deciding whether a channel should remain in the core is its ability to contribute to interactive consistency (which is dependent on correct operation of its processor, but not on correctness of its state data). For example, in a five channel system, we are better off leaving a channel with a correct processor but faulty state data in the core rather than placing it on probation. (One-round interactive consistency algorithms can withstand a single Byzantine fault among either four or five processors, but five processors using hybrid or authenticated algorithms can withstand larger numbers of nonByzantine faults than can four processors; state-data faults are masked by majority voting and with one faulty channel among five, we can still withstand an additional fault.) If the channel has a faulty processor, however, we are better off placing it on probation, because we will otherwise be vulnerable to failure if a second fault arrives (for example, a one-round hybrid fault tolerant interactive consistency algorithm can withstand two simultaneous faults among five processors only if both faults are of the essentially trivial “manifest” kind [17]).

Diagnosis based on comparison of output values does not distinguish errors due to a faulty processor from those due to bad state data. A plausible way to distinguish these is to require each processor to perform a complex computation on some fresh item of data[6] that is common to all channels. A suitable item might be the value of frame counter: this value should be among those that the channels exchange and subject to majority voting in each frame, and the diagnostic computation should be performed on the majority-voted value.

In summary, the goal of the diagnosis step should be to collect *syndrome* data that can be used in deciding group membership. Suitable components of the syndrome include the following.

[6]It should be fresh in order to detect a processor that merely reuses the result from an earlier frame.

- The results of a self-test (identifies gross processor faults).
- The results of a computation performed on recent majority-voted data, such as the frame counter (helps identify processor and communication faults).
- The output values sent to the actuators (helps identify processor and state data faults).

Each channel (whether in the core or on probation) collects these components of its syndrome, and distributes them to all other channels in a manner that ensures interactive consistency. Algorithms for achieving interactive consistency in this architecture were described in Section 4. If the fault-tolerance of the basic interactive consistency algorithm employed is t, and if the core contains t or fewer faulty channels, then all nonfaulty channels will obtain each others' syndromes correctly, and will agree on the syndromes ascribed to faulty channels.

5.3 Group Membership

The purpose of collecting syndrome information is to decide which channels should be in the core and which should be placed on probation. This is an instance of the *group membership* problem for synchronous systems first introduced by Cristian [2]. The present instance is simpler than the general problem because all nonfaulty channels are performing the same computations in lockstep. Thus, all that is necessary is for each processor to run an identical, deterministic, selection algorithm on the interactively consistent syndrome information that was distributed in the diagnosis phase.

Notice that it is not necessary to restrict this computation to those channels that are in the core, nor to require interactive consistency on the results of the computation. Provided the core maintains enough redundancy to mask faults among its own members, all nonfaulty channels (including those among the probationers and no matter how many faulty probationers there are) will achieve interactive consistency on the syndrome data for all channels. By running a deterministic computation, all nonfaulty channels will then arrive at a common view of the membership of the core. Notice that it does no harm if a faulty channel acts as if it were in the core. We do not consider here the actions that should be taken when a channel is first placed on probation. It may be desirable to force it to reboot and run diagnostics: this can be achieved by having all nonfaulty channels send a "reboot" signal to the channel concerned. A majority voter on each processor's "reboot" line will ensure this is only performed when necessary (since the nonfaulty processors must always be a majority).

In order to mask faults in actuator output, and in order for damaged state data to be repaired by loading majority-voted values, we must require that some majority of channels is nonfaulty at all times. This requirement can be relaxed if we move to a more drastic form of reconfiguration where the outputs of probationers can be disabled, and where only values from the core are used in refreshing state data. In this case, the requirement is for a majority of the core to be nonfaulty (this could be a strict minority of all components), but the majority voters at the actuators become rather more complicated (they need to maintain state in order to track membership of the core) and we do not pursue this design here.

6 Conclusions

The interactive consistency and clock synchronization elements of architectures based on state machine replication can generally tolerate fewer faults than the majority voting used to mask faults at the actuators. Reconfiguring to permanently exclude faulty components allows these elements to tolerate more faults, but places great demands on accuracy and completeness of the diagnosis of faulty components, and provides poor treatment for transient faults. We have proposed an architecture that allows components diagnosed as faulty to continue operation "on probation," so that transiently faulty components can repair their state, but in such a way that they do not degrade the fault tolerance of clock synchronization and interactive consistency.

This architecture adds very little complexity to the basic approach of state-machine replication with transient recovery, but can provide useful additional resilience. If no components are ever placed on probation, the proposed architecture is equivalent to the basic one. If diagnosis is done in a cautious way (so that only indisputably faulty components are placed on probation), it is sure to provide increased fault tolerance. The tradeoffs in using less cautious diagnosis can be evaluated experimentally or with reliability modeling techniques. In future, we hope to formally verify properties of the proposed architecture.

References

[1] Richard W. Buskens and Ronald P. Bianchini, Jr. Distributed on-line diagnosis in the presence of arbitrary faults. In *Fault Tolerant Computing Symposium 23* [11], pages 470–479.

[2] Flaviu Cristian. Agreeing who is present and who is absent in a synchronous distributed system. In *Fault Tolerant Computing Symposium 18*, pages 206–211, Tokyo, Japan, June 1988. IEEE Computer Society.

[3] Judith Crow and John Rushby. Model-based reconfiguration: Toward an integration with diagnosis. In *Proceedings, AAAI-91 (Volume 2)*, pages 836–841, Anaheim, CA, July 1991.

[4] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 163–188. Springer-Verlag, Vienna, Austria, September 1992.

[5] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

[6] Carl S. Droste and James E. Walker. *The General Dynamics Case Study on the F16 Fly-by-Wire Flight Control System*. AIAA Professional Study Series. American Institute of Aeronautics and Astronautics. Undated.

[7] Klaus Echtle. Fault masking with reduced redundant communication. In *Fault Tolerant Computing Symposium 16*, pages 178–183, Vienna, Austria, July 1986. IEEE Computer Society.

[8] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186, 1982.

[9] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications—5*, pages 79–90, Champaign, IL, September 1995. IFIP WG 10.4, preliminary proceedings.

[10] J. Y. Halpern, B. B. Simons, H. R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, B.C., Canada, August 1984. Association for Computing Machinery.

[11] *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993. IEEE Computer Society.

[12] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[13] Jaynarayan H. Lala and Richard E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, January 1994.

[14] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[16] Patrick Lincoln. Formally verified algorithms for diagnosis of manifest, symmetric, link, and byzantine faults. Technical Report SRI-CSL-95-14, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1995.

[17] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23* [11], pages 402–411.

[18] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[19] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [24, pp. 237–257].

[20] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[21] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[22] Kang C. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Fault Tolerant Computing Symposium 17*, pages 55–60, Pittsburgh, PA, July 1987. IEEE Computer Society.

[23] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.

[24] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

[25] C. J. Walter, N. Suri, and M. M. Hugue. Continual on-line diagnosis of hybrid faults. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications—4*, volume 9 of *Dependable Computing and Fault-Tolerant Systems*, pages 233–249. Springer-Verlag, Vienna, Austria, January 1994.

[26] Chris J. Walter. Identifying the cause of detected errors. In *Fault Tolerant Computing Symposium 20*, pages 48–55, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[27] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.