SAE INTERNATIONAL®

A Proposed Byzantine Fault-Tolerant Voting Architecture using Time-Triggered Ethernet

2017-01-2111

Published 09/19/2017

Andrew Loveless
NASA Johnson Space Center

Christian Fidi and Stefan Wernitznigg
TTTech

CITATION: Loveless, A., Fidi, C., and Wernitznigg, S., "A Proposed Byzantine Fault-Tolerant Voting Architecture using Time-Triggered Ethernet," SAE Technical Paper 2017-01-2111, 2017, doi:10.4271/2017-01-2111.

Abstract

Over the last couple decades, there has been a growing interest in incorporating commercial off-the-shelf (COTS) technologies and open standards in the design of human-rated spacecraft. This approach is intended to reduce development and upgrade costs, lower the need for new design work, eliminate reliance on individual suppliers, and minimize schedule risk. However, it has not traditionally been possible for COTS solutions to meet the high reliability and fault tolerance requirements of systems implementing critical spacecraft functions. Byzantine faults are considered particularly dangerous to such systems because of their ability to escape traditional means of fault containment and disrupt consensus between system components. In this paper, we discuss the design of a voting protocol using Time-Triggered Ethernet capable of achieving data integrity in the presence of a single Byzantine fault. Moreover, we explore how this capability can be combined with an exact-match voting strategy to realize a fault-tolerant computer system that can be used with different COTS processor boards, operating systems, and software frameworks.

Introduction

Following design trends in military and commercial aircraft over the last few decades, modern spacecraft architectures are moving towards adopting Integrated Modular Avionics (IMA) principles, as well as incorporating more commercial off-the-shelf (COTS) technologies and open standards (e.g. ARINC 653). The overall reduction in computing platforms, unique sparing, and harness mass in more integrated spacecraft avionics potentially offers significant cost and weight savings over federated design approaches [1]. The resulting increase in commonality, both among the interfaces and hardware platforms themselves, increases maintainability for long duration manned missions that cannot be supported or resupplied from earth (e.g. to cislunar space, or Mars). Moreover, since the Constellation program, NASA's strategy for the design of future spacecraft architectures (including launchers, landers, etc.) has heavily prioritized the use of COTS technologies for the purpose of reducing cost, minimizing the need for additional development, and removing schedule risk associated with the creation of custom hardware.

However, many spacecraft subsystems require a high degree of reliability and fault tolerance (e.g. 10^{-9} failures/hour) not traditionally achievable without the use of specially designed and proprietary solutions (e.g. self-checking computers). Studies conducted during the Orion program showed that the application of purely COTS designs to such systems would result in insufficient reliability and undue expense over the life of the program [2,3]. These problems are compounded by the IMA philosophy of leveraging the same hardware resources for both critical and non-critical functions, and the resulting need for robust time and space partitioning within both the computing platforms (e.g. memory space, computation time) and the data network (e.g. bus access) [4]. Nonetheless, advancements in COTS technologies (e.g. in radiation tolerance) continue to make their incorporation into architectures for human-rated spacecraft more feasible and attractive.

As a result, multiple NASA projects have explored the topic of developing safety-critical control systems which heavily utilize COTS components. The Scalable Processor-Independent Design for Extended Reliability (SPIDER) is a family of reconfigurable fault-tolerant architectures under development at NASA Langley Research Center (LaRC). SPIDER provides a generic host interface enabling it to use a variety of COTS components, including single-board computers (SBCs) and data concentrators [5]. Moreover, the Heavy Lift Vehicle (HLV) Avionics Flight Computing Architecture Study, also led from NASA LaRC, considered several fault-tolerant voting architectures for use in launcher applications - all of which could theoretically be realized with COTS hardware [6]. More recently, a design study for NASA's Evolvable Mars Campaign proposed a safety-critical lander architecture featuring four generic command and control computers networked using a COTS Time-Triggered Protocol (TTP/C) databus over a MIL-STD-1553 physical layer [7].

Most efforts pursue one Fault-Tolerant (1FT) architectures capable of continued operation in the presence of a single arbitrary fault (i.e. a fault with no restrictions on how it manifests). A particular class of arbitrary faults, called *Byzantine faults*, exhibit different behaviors to different components within the same system (e.g. conflicting data) [8]. The need for avionic systems to tolerate Byzantine faults is the result of the common requirement for *consensus* among multiple vehicle subsystems or components. There are several architectural approaches for realizing a Byzantine-resilient system (i.e. one capable of tolerating a Byzantine fault). The most common approach uses a *full exchange*, in which some mechanism is used to jointly reach agreement among all components in a set. Some designs instead use a *hierarchical* approach where agreement is established within smaller subsets of components, then among the subsets themselves [9,10]. This is the strategy used in the design of the Orion Vehicle Management Computers (VMCs).

In this paper we propose a 1-Byzantine fault-tolerant voting architecture based on the concept of full exchanges and realizable with a variety of COTS processor boards and real-time operating systems (RTOSs). It applies concepts from other NASA voting architectures, while also leveraging Time-Triggered Ethernet technology as a fundamental aspect of the fault-tolerance strategy. This approach simplifies the problem of data distribution between voting computers, lowers the complexity of implementing fault-tolerance mechanisms in software, and minimizes the demand for host computing resources. Specifically this paper makes the following contributions:

- We describe the physical system topology and show how it satisfies the requirements for Byzantine fault tolerance ("System Topology", "Fault Hypothesis", and "Satisfying Assumptions").
- We describe a simple method for achieving consensus among multiple processors over the Time-Triggered Ethernet network ("Ensuring Agreement").
- We demonstrate how this method can be combined with an exact-match voting approach in order to realize a triply-redundant fault-tolerant computer system ("Commanding").
- We have used the Semi-Markov Unreliability Range Evaluator (SURE) program to evaluate the reliability of such a system over varying mission durations ("Reliability Analysis").

General background regarding the topic of Byzantine fault tolerance is provided in the "Background" section. We discuss conclusions and tradeoffs in the "Summary/Conclusions" section.

Background

The problem of *Byzantine Agreement* was formulated during the development of the Software-Implemented Fault-Tolerance (SIFT) computer in the late 1970s, described in a watershed paper by Pease, Shostak, and Lamport in 1980 [11], and first characterized by name in 1982 [12]. The problem may be modeled as a system of n nodes, all of which can communicate directly with one another. Moreover, the following assumptions about the system are made:

A1. A fault in one node cannot cause a fault in another.
A2. Messages cannot be altered or lost in transit.
A3. The identity of the sender can always be determined.
A4. It is possible to detect the absence of a message.

One node n_i must share its initial value v_i with the other nodes in such a way that all non-faulty nodes (including n_i) agree on the same value v. Specifically, an algorithm is said to solve the problem of Byzantine Agreement if it is capable of guaranteeing the following properties in the presence of faults [13,14]:

- *Agreement:* All non-faulty nodes agree on the same value v.
- *Validity:* If n_i is non-faulty, then all non-faulty nodes agree on $v = v_i$.
- *Termination:* All non-faulty nodes eventually decide on a value.

If all nodes are non-faulty, then it is easy to fulfill the above requirements. In this case, the source node n_i may simply broadcast its initial value to the other nodes. Each receiver node can use the value it receives, and the source node n_i can directly use its own value. Even if nodes exhibit faulty behavior, it is possible to devise very simple algorithms that satisfy the conditions pairwise. For example:

- *Agreement and Validity:* All nodes delay making a decision indefinitely.
- *Agreement and Termination:* All nodes use the same predetermined value.
- *Termination and Validity:* All nodes use the value received from n_i directly.

It turns out, however, that it is nontrivial to formulate a solution that satisfies all three conditions in cases where the source node may be faulty. Above, the second solution attempts to sidestep this problem by using a predetermined result (e.g. zero). However, if the source n_i is non-faulty, the validity requirement is violated if the predetermined value does not happen to match the shared value v_i. Similarly, the third solution violates the agreement requirement in cases where the source n_i transmits different values to different nodes, or fails to transmit to some subset of receivers (i.e. typical Byzantine behavior).

What is instead required is an agreement protocol enabling receivers to work together to isolate the effects of a faulty (or even malicious) source node. The original papers proposed such a solution based on the concept of *Oral Messaging (OM)*, in which after a node broadcasts its initial value to all of its peers, the message is not used directly, but rather exchanged among the nodes once more (in the 1FT case) [11,12]. This additional step enables each node to retrieve the values originally sent to each of its peers. The expectation is that, given enough non-faulty nodes are present within the system, a receiver can mask the effects of a faulty node by considering the opinions of its peers in addition to its own received value. If the necessary requirements are met, agreement can be guaranteed by performing a majority vote over the redundant messages.

Figure 1 shows an example of an OM exchange between four nodes. In the first messaging round, node 4 broadcasts its initial value of 5 to all peers. However, node 4 is faulty and transmits an incorrect value to node 1. In the second round, each receiver node retransmits the original message to every other node (second round messages to node 4 are omitted from Figure 1 for clarity). After performing a majority vote, each non-faulty node agrees on the initial value of 5. Also note that if node 4 had initially transmitted different data to each of the

other nodes, then each node's majority vote would not yield a result. However, all non-faulty nodes still reach agreement that there is no value, and a default value can be used instead.

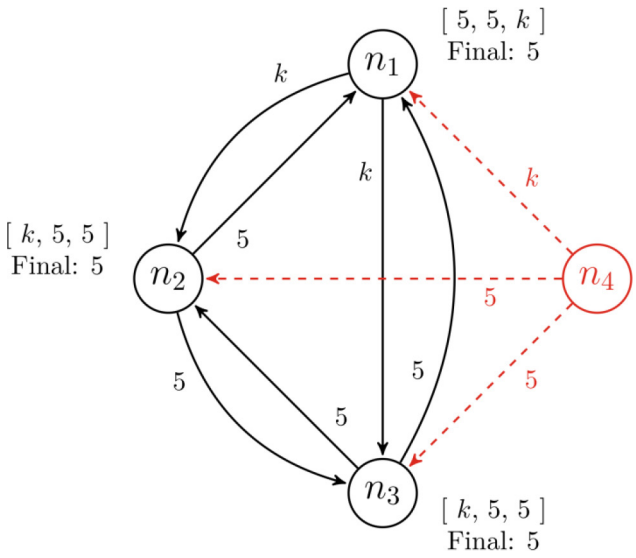

Figure 1. Example of message exchange in the OM(1) algorithm where node 4 is faulty. Second round messages sent to node 4 are not shown.

Another way of looking at the solution is that each node must receive the initial value from at least three different paths - otherwise the majority vote fails if the values do not match. Figure 2 demonstrates this concept from node 2's perspective, which is a "flattened" or "unrolled" version of the same algorithm. Of course, now that the value received from the source cannot be used directly, it is possible for a faulty receiver node to impact the exchange by retransmitting incorrect values. In this case, all the necessary conditions are still satisfied as long as the other three nodes are non-faulty. Moreover, the effect of two faulty receivers can be mitigated through the addition of two more non-faulty nodes. However, even if only one receiver is faulty, a faulty source could cause the four non-faulty receivers to reach agreement on different values.

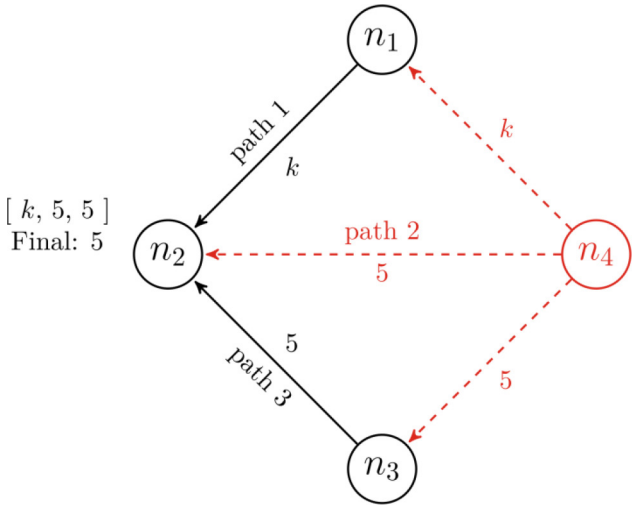

Figure 2. Example of message exchange in the unrolled OM(1) algorithm where node 4 is faulty. Only first and second round messages sent to node 2 are shown.

It is therefore apparent that some relationship exists between the total number of nodes, the number of redundant messages, and the number of faults which can be tolerated. In fact, it can be shown that any solution capable of tolerating f concurrently Byzantine faulty nodes must fulfill the following requirements [8,15]:

R1. There must be $\geq 3f + 1$ total nodes in the system.
R2. Nodes must receive messages via $\geq 2f + 1$ disjoint paths.
R3. Data must be exchanged $\geq f + 1$ times between nodes.

Solutions to the problem of Byzantine Agreement can be used to formulate solutions to the more complex agreement problems of *Interactive Consistency* and *Byzantine Consensus*. In Interactive Consistency, each node n_i must share its initial value v_i with the other nodes in such a way that all non-faulty nodes agree on the same vector of values V. Specifically, a solution to the problem must guarantee the following properties in the presence of faults [13,14]:

- *Agreement:* All non-faulty nodes agree on the same vector $V = [v_1, \ldots, v_n]$.
- *Validity:* If n_i is non-faulty, then all non-faulty nodes agree on $V[i] = v_i$.
- *Termination:* All non-faulty nodes eventually decide on a vector.

In Byzantine Consensus, each node n_i again shares its initial value v_i with the other nodes. Now, however, the goal is for all non-faulty nodes to agree on the same value v. A solution to this problem must guarantee the following properties in the presence of faults [13,14]:

- *Agreement:* All non-faulty nodes agree on the same value v.
- *Validity:* If all non-faulty nodes have the same initial value v_i, then all non-faulty nodes agree on $v = v_i$.
- *Termination:* All non-faulty nodes eventually decide on a value.

It is clear that the three agreement problems are closely related. The Interactive Consistency vector V can be generated by executing an instance of the Byzantine Agreement protocol on each of the n nodes [16]. This process results in all non-faulty nodes sharing the same view of the system. Additionally, if each node n_i performs the same operation on their vector V (e.g. majority vote, mid-value selection), then all non-faulty nodes will produce the same value v - thereby solving the Byzantine Consensus problem.

System Topology

The challenge of realizing a fault-tolerant computer system for safety-critical applications is that the overall reliability required (e.g. 10^{-9} failures/hour) is typically much higher than that of the system's individual components (e.g. 10^{-6}-10^{-7} failures/hour is typical) [17]. One fundamental strategy for the design of a highly-reliable embedded system is to use multiple redundant processors to perform the same computations in parallel. In such systems, comparison and voting operations are used to resolve commands produced by the collection of redundant processors to a single value. Many ultra-dependable systems in spaceflight applications use an exact-match voting approach, in which all processors are required to produce the same result in the absence of faults [18]. Examples include the Space Shuttle Data Processing System's (DPS) General Purpose Computers

(GPCs), the X-38 crew return vehicle's Flight Critical Computers (FCCs), and the Ares I launcher's upper stage command and control system [19, 20, 21]. However, exact match voting approaches are based on the assumption that all redundant processors are provided bitwise identical input data (e.g. sensor values, external interrupts) and maintain bitwise identical state information. Because these assumptions fundamentally rely on the ability to distribute single source data among the processors in the presence of faults, the system must be capable of solving the various Byzantine agreement problems.

For this purpose, we propose a switched voter architecture which uses Time-Triggered Ethernet (TTE) for data distribution and synchronization between processors. The system topology is depicted in Figure 3. Each host processor executes application-specific software and is locally connected to a network controller, which acts as its interface to the data network. The controller is responsible for executing the various required TTE protocol services (e.g. synchronization service). Collectively, the host and controller are referred to as an end system. Connectivity is provided through two or three redundant TTE network switches. The arrangement of switches and end systems forms a complete bipartite graph, in which each switch is directly connected to every end system (and vice versa) via point-to-point links in a physical star topology. All communication links and interfaces are bidirectional.

Communication between end systems occurs over the data network through all redundant switches simultaneously.

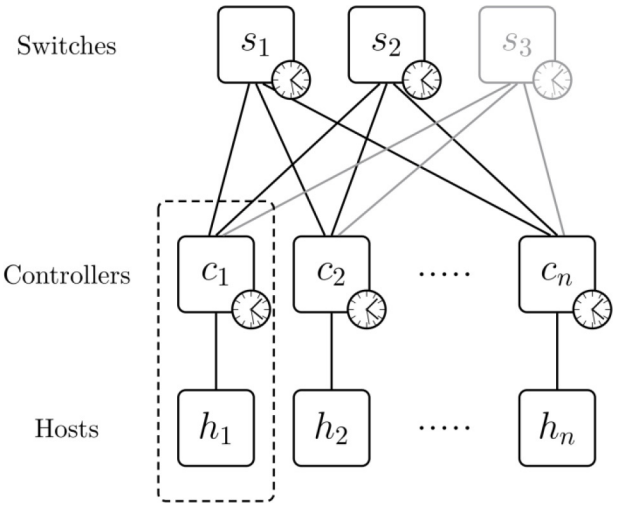

Figure 3. Physical topology of the TTE-based voting architecture. The dashed line outlines the components of an end system. The third switch is optional.

Because of the tight timing constraints associated with the TTE services, end system controllers and switches typically implement them in hardware by means of field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). For spaceflight applications, a radiation-tolerant TTE ASIC has been commercially developed by TTTech for use in both switches and end systems. Several options exist for the physical configuration of hosts and TTE hardware. An end system controller can be housed on a mezzanine card and mated directly to a COTS SBC in a stacking configuration. Additionally, a controller can be implemented on a peripheral card and connected to an industry standard backplane (e.g. CompactPCI, VPX). System on a chip (SoC) solutions are also possible, supporting both the end system host and controller functions within one device.

Fault Hypothesis

The design of a fault-tolerant computer system is based on assumptions regarding the types of faults each component may exhibit. This information is needed to form a *fault hypothesis* specifying which faults the system must tolerate without impacting the availability of critical system services [22]. Because the system is designed to tolerate only those faults which are covered by the fault hypothesis, it is important to demonstrate that assumptions limiting the severity of potential faults hold true in practice (i.e. with sufficiently high probability).

In the proposed architecture, end systems are assumed to have an *arbitrary* failure mode. This means that no assumptions are made regarding the manner in which an end system may fail. A faulty end system may transmit arbitrary messages of any contents, transmit messages at arbitrary points in time, or send different messages through different network switches. The behavior of switches is restricted to an *inconsistent omission* failure mode where value correctness of messages is guaranteed. Switches are assumed incapable of generating messages themselves and may not modify messages in order to produce new messages considered valid by the network protocol. However, switches may still drop or fail to receive an arbitrary number of messages and may relay messages asymmetrically (i.e. such that some receivers are not sent data). These more limited failure assumptions are required for the trusted role switches generally fill in the TTE synchronization protocol [23].

By constraining the failure modes of some components in the system, it is possible to partially relax the requirements necessary for reaching agreement. In the case of a device restricted to inconsistent omission, faulty behavior is limited to producing messages which are easily identified as invalid and dropped by non-faulty receivers (manifest fault), or failing to transmit messages at all (omission fault) [24] . It then follows that any message transmitted from the device which is successfully delivered can be trusted by the receiver. Azadmanesh and Kieckhafer showed that only $f + 1$ nodes are required to tolerate f strictly omissive asymmetric faults when missing or malformed messages are not included in the vote [25].

The failure behavior of TTE switches is constrained to the inconsistent omission failure mode through the use of a *high-integrity* design with self-checking pair electronics. Rather than transmitting separate signals over paired media, a high-coverage command and monitor (COM/MON) approach is used to enable error containment within the device [10]. A COM/MON design is constructed from two switch chip IP cores. One unit acts as a commander which drives the network media, while the other acts as a monitor responsible for detecting faulty messages produced by the commander. An agreement protocol is used to ensure that both the COM and MON operate on identical inputs. Each core performs the same operation and produces the same output in the absence of faults. The output generated by the COM is forwarded to the MON for comparison. If the output of the MON does not agree, then the MON intercepts transmission of the message from the COM [26]. This action results in the delivery of a truncated message fragment which is easily identified and discarded by the receiver. Because the COM and MON are designed to fail independently, fault propagation from a switch theoretically requires a very rare dual-correlated simultaneous fault (e.g. $10^{-6} \times 10^{-6} = 10^{-12}$ failures/hour).

Satisfying Assumptions

It is possible to leverage the combination of switches and end systems to implement an unrolled version of the classical Oral Messaging protocol. Of course, any fault-tolerant system can fail if it is based on assumptions which are not satisfied [27]. Generally, the difficulty of realizing a Byzantine-resilient system is not the implementation of the necessary algorithms (which are relatively simple), but rather the need for a communication system which satisfies the necessary system assumptions (A1-A4) [12].

Assumption 1

In Byzantine theory, the only way that nodes impact one another is through the messages they pass (A1). In real-life systems, however, common mode elements (e.g. a shared timing source) could potentially cause a fault in one device (say a chip or a board) to induce a fault in another seemingly unrelated component. In order to satisfy the assumption, it is necessary to partition the hardware components in the system into independent *fault containment regions* (FCRs). Each FCR generally requires independent power and clocking sources, and must be sufficiently dielectrically and physically isolated from the other FCRs [15]. If properly implemented, any arbitrary fault within one FCR (e.g. an electrical short) cannot cause the components in another FCR to fail. Likewise, the components within an FCR can operate correctly in spite of any arbitrary fault outside the FCR [8,18]. Because the failures of components within different FCRs are statistically independent events, FCRs can replace the concept of nodes in the generalized system model introduced in the "Background" section. In the proposed voting architecture, each end system forms a single FCR. Moreover, each switch contains two independent FCRs comprising a single error containment unit. The current TTE COM/MON design avoids the need for independent power and clocking between FCRs by using separate power and clock monitors to prevent common mode failures [26].

Previous examples assumed that agreement was required between all FCRs participating in the Oral Messaging exchange - which is often not the case. An *interstage* is a device which contributes FCRs and relays messages according to the algorithm, but which does not itself require agreement. As a result, interstages do not need to receive messages in the last round of exchange nor perform the final majority vote. The concept of an interstage was first leveraged in the Draper Fault-Tolerant Processor (FTP) architecture and is also used in the SPIDER Reliable Optical BUS (ROBUS) [8]. Here the TTE network switches act as interstages. Notionally, the lower hardware complexity of a switch in comparison to an onboard computer could result in increased reliability over equivalent voting architectures requiring all FCRs to contain full processors.

Preserving the necessary separation between the end system and switch FCRs requires sufficient electrical isolation across the communication links. This can be accomplished in a variety of different ways, including with passive galvanic isolators (if a copper physical layer is used), or by using optical fiber links.

If it is assumed that any end system may fail arbitrarily, then a 1FT design can be realized with a minimum of two high-integrity switches. This configuration provides more than the $\geq 3(1) + 1 = 4$ FCRs needed to ensure agreement between any two processors (R1). The connectivity requirement (R2) can be relaxed because all asymmetric switch faults are restricted to being omissive, but only if missing values are not included in the vote (i.e. a hybrid majority). The transmission of a message from an end system to the switches, then the reflection of the message back to the end systems, provides the required $\geq (1) + 1 = 2$ rounds of data exchange (R3). Because the end systems do not communicate directly, and because the switches provide the means by which to ensure agreement between any two processors, the total number of end systems is irrelevant and may be scaled according to the needs of the application.[1] For the purpose of the above calculations, all faults which occur within the boundary of one FCR are counted as a single fault.

Assumption 2

The assumption that messages cannot be altered or lost once transmitted (A2) is satisfied through a combination of different factors. Of course the total absence of communication errors cannot be guaranteed in any real-life system. In this architecture, 32-bit cyclic redundancy checks (CRCs) are used for detection of medium-induced errors. CRCs for received Ethernet frames are calculated by both switches and end systems, and invalid frames are dropped. Given a frame protected by the CRC is randomly corrupted (i.e. such that the original and resulting frame are totally uncorrelated) there is a $1/2^{32} = 2.3 \times 10^{-10}$chance the error is not detected [28]. These odds can be improved by assuming that individual bit errors are uncorrelated. The CRC-32 generator polynomial used for Ethernet is guaranteed to detect up to three uncorrelated bit errors in a maximally-sized 1514 byte frame (Hamming distance = 4) [29]. The maximum allowable bit error rate (BER), the ratio of the number of errors to transmitted bits, is specified as 10^{-10} by IEEE Std 802.3 for 1000BASE-T [30]. Given a message size of 1514 bytes, transmission rate of 100 Mbit/s, and using the calculation provided in [17], the undetected error rate over a given link can be approximated as:

$$(2.3 \times 10^{-10}) \times \left(10^{-10}\,\frac{error}{bit} \times 12{,}112\,\frac{bit}{msg}\right)^4 \times \left(8{,}256\,\frac{msg}{s} \times 3{,}600\,\frac{s}{hr}\right) = 1.5 \times 10^{-26}\,\frac{error}{hr}$$

The synchronous nature of time-triggered communication enables *temporal partitioning* that ensures the independence of separate critical dataflows. Message transmission and reception is scheduled to occur at specific instances in time according to a fault-tolerant global time base. Because the expected arrival times of all messages are known a priori, messaging faults in the time domain are detected and contained at the boundaries of each non-faulty FCR. Knowledge of the task scheduling, network scheduling, and message sizes can be used to calculate the size of switch and end system buffers and ensure that no messages are dropped due to overflow. Moreover, the time-triggered paradigm eliminates contention between frames and guarantees sufficient bandwidth for each dataflow. A babbling end system cannot monopolize the bandwidth between a switch and another end system. Babbling failures of the switches are precluded by their high-integrity design.

If A2 is assumed to also apply to the interface between host operating system (OS) partitions and the end system controller, then a mechanism for *spatial partitioning* is needed to preserve the

1. The TTE clock synchronization protocol requires a minimum of four synchronization masters (SMs) within the network to tolerate a single Byzantine fault without a degradation of network services. Typically, the roles of SMs are performed by end systems [23].

independence of OS partitions accessing the controller. To facilitate this, the communication ports on each TTE controller can be grouped into separate device partitions with independent address spaces. The partitioning scheme is specified during the offline scheduling process and stored within the configuration table loaded onto each controller. These device-level partitions may be directly mapped to the OS partitions (e.g. in ARINC 653). Applications in different OS partitions may access the network controller simultaneously for sending and receiving data without any interference.

Assumption 3

The third assumption states that the identity of a message's sender can always be determined (A3). In practice, this means that it must not be possible for a device to impersonate, or *masquerade* as, another device to which the receiver is also physically connected. In this architecture, communication between switches and end systems occurs over fixed lines, and the identity of the sender is implied by the physical port on which a message is received. Virtual links are defined offline and direct the flow of traffic between end systems. The virtual link associated with each time-triggered frame is denoted by a critical traffic identifier (CTID) occupying two bytes of the Ethernet header [31]. Each switch uses a static forwarding table to associate CTIDs with the corresponding physical input/output ports. If a faulty end system impersonates another by transmitting a frame with identical header information, even in the expected time slot, the frame will be verified against the switch configuration and subsequently dropped. Moreover, each end system can determine the identity of the switch from which it received a message. Because of the switches' high-integrity design, a faulty switch cannot alter a message such that its header information no longer properly identifies the end system which transmitted it.

Assumption 4

Lastly, the assumption that it is possible to detect the absence of a message (A4) is required to ensure that the majority vote is not delayed or prevented because a message was not received. Satisfying this assumption requires a synchronous mode of operation, which implies that the system has the following properties [12]:

1. There is a fixed maximum transmission delay for each message.
2. There is a fixed maximum time to generate each message.
3. Senders and receivers are synchronized to within a maximum known skew.

In fact, it can be shown that no solution to Byzantine Agreement exists in an asynchronous system if the source is faulty [14]. This is because in such cases an absent message is indistinguishable from an arbitrarily delayed one. Together, the above three properties can be used to calculate a window in which new messages are accepted and after which undelivered messages are considered absent. The first property is satisfied by the time-triggered communication method. The coordinated nature of messaging eliminates the possibility of contention between frames (e.g. for access to a switch egress port), and enables communication with fixed end-to-end latency (e.g. 15µs per switch) and very low jitter (e.g. <5µs). The second property is satisfied by 1) the determinism of the TTE switches, and 2) using a RTOS on each host processor, where the execution of tasks responsible for message generation are guaranteed to complete by a predetermined deadline. The third property is accomplished by using a network interrupt to drive the execution of flight software on the host processors. Interrupts are defined offline as part of the network scheduling process. At runtime, interrupts are triggered on each processor according to the value of its local clock. Because each end system controller is synchronized to the global timebase, the interrupts occur simultaneously across all processors.

Ensuring Agreement

The occurrence of a fault within an end system can result in the propagation of an error across its FCR boundary in the form of an erroneous message [32]. Moreover, the unrestricted failure mode of an end system enables the asymmetric transmission of such messages to different switches. Though the self-checking design of the switches ensures containment of internal errors, it cannot prohibit the flow of valid frames (i.e. with correct CRC and timing) containing erroneous data. This architecture uses a modified OM(1) algorithm to mask such errors, ensure data integrity, and achieve agreement between non-faulty end systems. The algorithm is adopted from the SPIDER Interactive Consistency (IC) protocol, which itself is an adaptation of the IC algorithm for the Draper FTP [4,24]. The steps are as follows:

1. One host h_i transfers its initial value v_i to its controller c_i.
2. Controller c_i broadcasts v_i to all s switches.
3. Each switch broadcasts the value to all controllers (including c_i).
4. Each controller collects the values received (v_1, … , v_s). Missing or malformed values are replaced with *NULL*.
5. Each controller performs a majority vote of non-*NULL* values.
6. If c_k finds a majority m_i, it is transferred to host h_k.

Manifest-faulty messages are discarded upon reception by both switches and end systems, and are therefore not included in the majority vote. Moreover, the vote is only performed out of the total number of messages received from the switches (i.e. not a fixed 2/2 or 2/3). This prevents asymmetric omissive faults from being changed into more troublesome Byzantine faults, and is necessary to tolerate one faulty switch in the two switch case. Unlike in SPIDER, a switch does not notify the controllers if it receives an invalid message [33]. There is therefore no distinction made at the receiving controllers between different causes of an absent message (e.g. dropped at the switch input, truncated at the switch output).

Figure 4 shows an example of the agreement algorithm in a network containing three end systems and three switches. Host processors connected to the controllers are not shown. Controller 1 attempts to broadcast the initial value of 5 to the switches. An arbitrary transmission error, however, causes an incorrect value k to be sent to switches 2 and 3. Each switch broadcasts the value received to all controllers, and each controller performs a majority vote of received values to agree on the incorrect value k. It is essential that controller 1 not use its initial value directly, but rather votes on the values returned by the switches. This may seem strange, since the value obtained from the vote was incorrect. However, this process is needed to ensure controller 1 agrees with the other non-faulty controllers. If instead two switches were used, and controller 1 sent different values to each switch, then all non-faulty controllers would agree that no majority was found. In cases where only switches are faulty, then

agreement between controllers is guaranteed as long as one non-faulty switch is present. This means the effects of two faulty switches are masked in a three switch configuration.

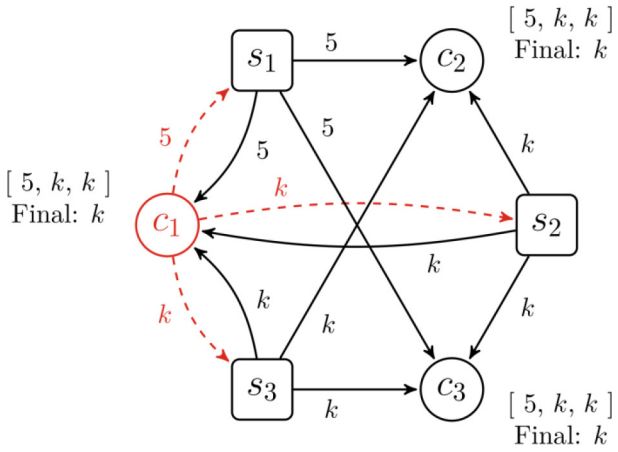

Figure 4. Example of message exchange in the agreement algorithm where controller 1 transmits asymmetrically.

This algorithm assumes that the originator of the message requires agreement with the other end systems to which it is transmitting. This is the case when, for example, one processor distributes data obtained from a locally-connected sensor to other processors in a redundant set. Of course, it may be necessary for the processors to agree on data originating from outside the set (e.g. from a remote interface unit (RIU) on the TTE network that is itself connected to various sensors). This scenario requires the same procedure to ensure an arbitrary transmission from the source cannot prevent the receivers from reaching agreement. However, it is no longer necessary for the switches to reflect messages back to the source. When reading data, messages are agreed upon before being made available to applications on the host processors. No additional exchanges are needed.

One way to implement the majority vote is in hardware on the end system controller. This approach avoids the overhead introduced by most software-based fault-tolerance techniques, which can commonly consume up to 30% of the available processing time. Redundant frames from the switches are individually read at the media access control (MAC) layer, checked for adherence to the network schedule, and discarded if determined invalid. With redundancy management disabled, all redundant Ethernet frames are passed to the higher layers. A voting module could be implemented above the User Datagram Protocol (UDP) layer and below the host interface. Depending on the port type, frames could be passed to either the voting module or the Internet Protocol (IP) layer. The voting module would be responsible for resolving redundant frames, IP packets, or UDP datagrams from the switches to individual messages transferred to the host. Voting can also be implemented in software either within or above the device driver, which retains the advantage of still performing the required two-round exchange in hardware. In this case, the precision of the network (which is determined a priori) can be used to form a temporal window which groups redundant messages from the same transmission.

The agreement algorithm outlined above can be used as the basis by which to achieve Byzantine Consensus between multiple end systems. It is often necessary for redundant processors to reach consensus on 1) internal state information, 2) input data, and 3) the diagnosis of system faults. It is clear that a consensus protocol is needed to tolerate the effect of a faulty processor. However, several scenarios can cause even non-faulty processors to disagree, and a consensus protocol to be necessary. Some examples include:

- If each processor has a dedicated sensor, and each sensor measures the same external variable, the values obtained from the sensors may differ.
- If a sensor is cross-strapped, synchronized processors will sample it at slightly different times (though within a known skew) and potentially obtain different values.
- A marginal signal transmitted from a cross-strapped input device could be interpreted differently by different processors.
- Depending on the skew between synchronized processors, a message may arrive before a deadline on some, but after a deadline on others.

The consensus algorithm can be summarized as follows:

1. All n hosts execute the agreement protocol (steps 1 - 6).
2. Each host h_i collects the majority values obtained $V_i = (m_1, \ldots, m_n)$. Missing values are replaced with *NULL*.
3. Each host executes the function *choice*(V_i) to produce v_{final}.

The algorithm works by transforming all messages sent by end systems in round 1 of the agreement protocol into symmetric messages (i.e. appearing the same for all non-faulty receivers). As a result, all non-faulty hosts are guaranteed to generate the same IC vector. The purpose of *choice*() is to resolve the vector of values (i.e. messages) to a single result [12]. Because *choice*() is performed on the same IC vector on each non-faulty host, all non-faulty hosts will produce the same final value. The selection of the *choice*() function is application-specific. Some possibilities include the average (for numeric values), median (for ordered values), or another majority. Moreover, *choice*() can apply different methods to different segments of the messages being resolved (i.e. average bits 0:31, vote bits 32:63).

Like in the agreement protocol, it may be necessary for the processors to reach consensus on data originating from other devices on the TTE network. In this case, messages generated from multiple input devices (e.g. inertial navigation units (INUs)) must be resolved to a single identical message on each host. Even if all devices are non-faulty, they are not guaranteed to (and will likely not) produce the same data. Assuming the possibility of arbitrary transmission errors, the agreement protocol is needed to ensure all non-faulty processors agree on the data produced by each device. However, instead of populating the IC vector with majorities corresponding to each processor, it is filled with majorities corresponding to each input device. The final step in the consensus algorithm is unchanged.

Commanding

If the input data for multiple processors is determined according to the agreement/consensus algorithms (and is therefore identical among non-faulty processors), and each processor performs the same computations, then all non-faulty processors are guaranteed to produce bitwise identical outputs. Because each output corresponds to a Boolean value (i.e. correct or incorrect), a majority vote can be used to reduce the outputs from all processors to a single value. If f

processors are faulty, a total of $2f + 1$ processors are needed to ensure the output is correct. Recalling the requirements discussed in "Assumption 1", a 1FT architecture therefore requires a minimum of two switches and 2(1) + 1 = 3 end systems.

Several system configurations would require the processors to interface with end effectors incapable of voting on redundant commands. Moreover, effectors may be unable to receive commands from multiple sources. One approach for accommodating such devices is to transmit all commands from one processor at a time (monitored by the other processors). Channelized approaches are also common, in which each processor is directly connected to a local databus (e.g. channels A, B, C). Each processor can only command effectors connected to its own bus (e.g. RIU1 on channel A, RIU2 on channel B). A channelized design was used in the X-38, and both Ares I and Delta IV launchers [6,20,21]. Alternatively, some effectors incapable of voting can still arbitrate between multiple sources (e.g. a force summing actuator). In this case, the processors may command separate servos whose combined force determines the response of the actuator. However, the impact of a faulty processor cannot be completely masked. When voting at the effector is not possible, it is necessary to instead determine the correct system output at each processor. If a processor experiences a fault which results in incorrect output, a voting algorithm enables it to correct itself before transmitting a command.

For this purpose, we first define a procedure for sharing the output from one processor with the others. The steps are below:

1. One host h_i transfers its initial value v_i to its controller c_i.
2. Controller c_i broadcasts v_i to all switches.
3. Each switch broadcasts the value to all controllers (except c_i).
4. Each controller c_k (where $c_k \neq c_i$) accepts the first valid value received.
5. If c_k accepts a value v_i, it is transferred to host h_k.

The voting algorithm can then be summarized as follows:

1. All n hosts execute the above sharing protocol (steps 1 - 5).
2. Each host h_i collects the values obtained, plus its initial value, in $V_i = (v_1, \ldots, v_n)$. Missing values are replaced with *NULL*.
3. Each host executes the function $maj(V_i)$ to produce v_{final}.

Like in the agreement protocol, manifest-faulty messages are discarded by both switches and end systems. However, voting is not performed over redundant messages transmitted from the same sender. Moreover, a majority vote is used in place of an application-specific *choice*() function. Unlike in the agreement protocol, the vote requires a fixed 2/3 majority to produce an output. There is no distinction made at each host between messages which were never sent and those absent due to system errors (e.g. failed end system, invalid CRC). This simplification is applicable in the 1FT case, since defeating the vote still requires two concurrent faults.

Unlike when reading input data, commanding does not require the processors to agree on each other's output values before executing the majority vote. Consider a system of three end systems and two switches. One host experiences a fault and calculates an incorrect output value of x. Also, during the exchange it transmits x to one switch but y to the other. Each receiver accepts the first valid value obtained from either switch. As a result, both non-faulty processors may possess a different vector of output values - e.g. (x, 5, 5) and (y, 5, 5). However, a majority vote on either vector produces the same correct result. If instead one switch is faulty, it may not forward messages to all recipients. Still, each processor can obtain all messages through the non-faulty switch. All processors will build the same vector of correct values and produce the correct output.

Of course, this voting scheme is sufficient only if the system can properly operate when merely a majority of the processors generate the correct output value. If this is not the case, it must be possible to silence the faulty processor. This can be accomplished using hardware interlocks. An interlock can be used at the output of each processor. If a majority of processors see one processor's output disagrees with the majority result, they can disable its communication path to the effectors [34]. Use of interlocks depends on the ability of good processors to diagnose that a processor is faulty. However, this requires all non-faulty processors to reach consensus on the diagnosis. Otherwise two good processors may disagree on whether another is faulty, and any mechanism to disable it will not be activated. To build consensus, the sharing protocol in the voting algorithm must be replaced with the agreement protocol introduced previously. Also, each processor's initial value cannot be directly included in its vector. The resulting algorithm is identical to the consensus algorithm, except the *choice*() function is replaced with a majority vote.

Some system configurations may enable the use of output devices capable of voting on redundant commands (e.g. an RIU controlling multiple actuators). Such devices may connect to the processors via the TTE network, another network technology, or point-to-point links. In this case, it is not necessary for the processors to vote on the system output prior to sending commands. As was the case when voting at the processors, such a system can mask the effect of one faulty processor without requiring redundant "effectors" to agree on each processor's output. Again, this capability requires absent values to be included in the vote. Though commands are voted at the output device, diagnosis of a faulty processor can still be accomplished by executing the modified consensus algorithm over the TTE network. If the effectors do not communicate using TTE, each processor must transmit its output twice - both to the effectors and to the TTE switches. A simplification can be made if the effectors are instead on the TTE network. In this case, commands transmitted to the switches can be forwarded to the effectors and reflected back at the processors simultaneously. Each effector builds a vector of values by accepting the first valid message received from each processor. The processors instead build the vector by voting the redundant messages according to the agreement algorithm. In both cases the vector is voted to obtain the final result. This process is illustrated in Figure 5.

Though not strictly necessary in a three processor configuration, it may be desirable to vote the redundant messages from each processor at the effector, thus ensuring redundant effectors all possess the same vector of commands before performing the final vote. This is necessary to ensure redundant effectors determine the same result in some two-fault scenarios.

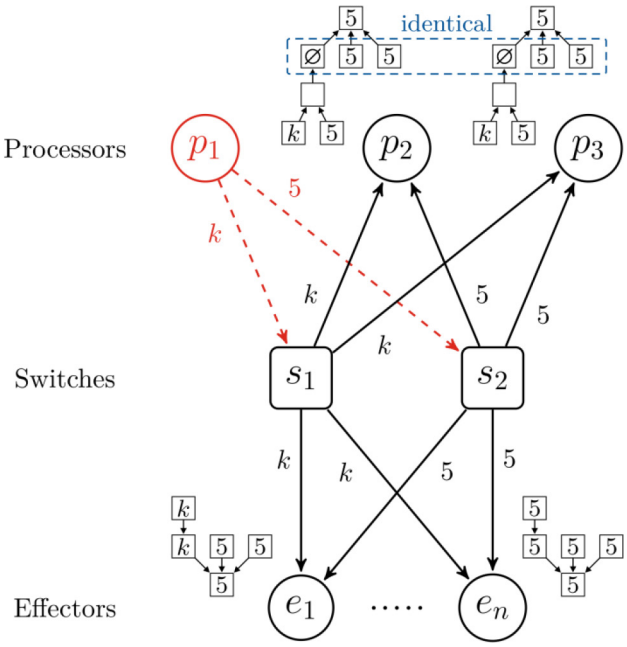

Figure 5. Example of commanding redundant effectors from triplex processors. Only message flows originating from processor 1 are shown. Still, messages from processors 2 and 3 are accounted for in each vote.

Reliability Analysis

It is generally infeasible to assess the reliability of a safety-critical system through conventional testing. Given a requirement of 10^{-9} failures/hour, validation through testing would require at least one billion hours of operation. Alternatively, software tools can be used to describe a fault-tolerant system using a mathematical model, and to calculate the probability of system failure under different conditions. The SURE tool was developed at NASA LaRC for computing upper and lower bounds on system reliability [35]. Unlike traditional fault-trees, the semi-Markov models used by SURE are capable of describing systems with transient faults, reconfiguration, and state-dependent failure rates [36]. The construction of a SURE model can be quite time-consuming and error prone, since models for even simple systems may contain hundreds of states and transitions. Rather than creating SURE models directly, the ASSIST (Abstract Semi-Markov Specification Interface to the SURE Tool) tool can be used to automatically generate models from scripts containing rules expressed in a high-level language [37]. We used ASSIST to generate semi-Markov models for various configurations of the TTE-based triplex voting system. We then used SURE to estimate the reliability of each configuration over varying mission durations. At a high level, the purpose of this work was to roughly assess the suitability of using the proposed architecture for long mission times. Moreover, we were interested in comparing the reliability of configurations 1) with two or three switches, and 2) with or without the ability to remove permanently faulty switches.

The fault model contains three standard-integrity end systems and either two or three high-integrity switches. The models assume that all end system faults are worst case (i.e. arbitrary with no restrictions). For simplicity, all switch faults are constrained to strictly omissive asymmetric behavior. Of course it is also possible that simultaneous faults in both of a switch's IPs cause the switch to fail arbitrarily. However, this possibility is not currently included in the models. Faults are modeled as either transient or permanent. Transient faults result in erroneous behavior for a finite duration of time before disappearing (e.g. due to system recovery or removal of an external factor). Permanent faults continuously impact the system until the offending component is removed. Intermittent faults, permanent faults which occur periodically, are not included.

In each model, end systems recover from transient faults by periodically exchanging and majority voting all internal state information. Each end system replaces its internal state with the voted result. In the case of a permanently faulty end system, the triplex processors degrade directly to a simplex. When using a majority vote to determine system output, both dual and simplex configurations can fail if a good end system becomes faulty. However, a simplex configuration lowers the system's overall fault arrival rate. Still, the reliability of the system could theoretically be improved by first degrading to a dual. For this configuration to make sense, it must be possible to determine which of two end systems failed with better than a 0.5 probability. This could be accomplished by designing the processors to increase the probability that failures are benign (e.g. they fail-stop), or using self-tests to help diagnose which is faulty [36]. Transient faults in switches are modeled as impacting messages stored in the input or output buffers. The impacts of these faults on the system persist until the switch forwards the next scheduled message. In two of the models, switches determined to be permanently faulty are removed. Of course, this behavior relies on the ability of the end systems to diagnose that a switch is faulty. The other two models allow permanently faulty switches to remain in the system. Also, all models account for the possibility that the system incorrectly degrades after interpreting a transient fault as permanent.

Modeling the system requires the specification of death states (i.e. absorbing states) representing combinations of faults which could result in system failure. The following rules were used by ASSIST to identify death states. Let EN denote the number of end systems in the configuration, SN the switches in the configuration, ET the transiently faulty end systems, ST the transiently faulty switches, EP the permanently faulty end systems, and SP the permanently faulty switches.

1. $(2 \times (ET + EP)) \geq EN$
2. $(ST + SP) = SN$
3. $((ET + EP) > 0)$ AND $((ST + SP) > 0)$

It is important to note that the maximum fault assumptions above do not account for the hardware required to maintain the underlying TTE services. It is assumed that enough end systems or switches are present elsewhere within the network for this purpose (e.g. to maintain synchronization). The SURE models calculate the probability that any of the death states are reached for various mission times. Table 1 contains some of the key parameters used for the reliability models. Let ES denote end system and SW switch.

Table 1. Key parameters used for generating the reliability models.

Mission time	0 – 8760 hr
ES initial number	3
ES permanent fault arrival rate	10^{-7}/hr
ES transient fault arrival rate	10^{-6}/hr
ES mean time to degrade	1.5 sec
ES std. dev. of time to degrade	$1/\sqrt{12}$ sec
ES state voting period	1 sec
ES mean time to recover	0.5 sec
ES std. dev. of time to recover	$0.5/\sqrt{3}$ sec
P(ES degrade from transient)	0.1
P(ES recover from transient)	0.9
SW initial number	2 or 3
SW permanent fault arrival rate	2×10^{-7}/hr
SW transient fault arrival rate	2×10^{-6}/hr
SW mean time to degrade	1.5 sec
SW std. dev. of time to degrade	$1/\sqrt{12}$ sec
Scheduled messaging period	1 msec
SW mean time to recover	0.5 msec
SW std. dev. of time to recover	$0.5/\sqrt{3}$ msec
P(SW degrade from transient)	0.1
P(SW recover from transient)	0.9

End system and switch faults are modeled using exponential transitions of the form $F(t) = 1 - e^{-\lambda t}$, where λ is the device failure rate and $F(t)$ is the probability of failure at time t (failure rates are constant). The permanent fault arrival rate for end systems is estimated based on the historical failure rates of CMOS semiconductor devices provided in [38]. The transient fault arrival rate is assumed to be ten times the permanent fault arrival rate [39]. Because of the switches' self-checking design (with two chip IPs), their fault arrival rates are approximated as two times those of the end systems. Degradation and recovery are modeled using non-exponential transitions expressed in terms of means and variances. For simplicity, both the time to degrade to a simplex configuration and the time to remove a faulty switch are assumed to be uniformly distributed over (1, 2) seconds. The period at which end systems replace their internal state information is estimated according to the transient fault recovery times used in [40]. The time from transient fault arrival to end system recovery is assumed to be uniformly distributed over this 1 second interval. The messaging period is calculated assuming a 4 Mbit/s transfer rate and 500 byte message size. The switch transient fault recovery time is again uniformly distributed (but based on the messaging period). Conservative estimates are used for the probabilities that a transiently faulty end system or switch is identified as permanently faulty.

Table 2 shows the probability of system failure, or system unreliability, of the four configurations for select mission durations. Only the lower bounds on unreliability are given (i.e. estimates are conservative). These values should be read with the understanding that the models do not reflect the reliability of a full avionics system - which would be lower due to the inclusion of many other devices (INUs, RIUs, etc.).

All four configurations meet the traditional reliability requirement of 10^{-9} failures/hour. For short mission times, the difference in reliability between all four configurations is minimal. As the mission time increases, the reliability of the three switch configuration, where faulty switches can be diagnosed and removed, improves in comparison to the other cases. This trend is made more apparent by plotting the probability of failure for each configuration over time (see Figure 6).

Table 2. Lower bound on system unreliability calculated by SURE for select mission durations.

SW count initial	3	2	3	2
SW degradation	Yes	Yes	No	No
P_f(1 hour)	3.38E-13	1.22E-12	1.35E-12	1.46E-12
P_f(24 hours)	1.89E-10	6.94E-10	7.58E-10	8.21E-10
P_f(1 months)	1.76E-7	6.46E-7	7.05E-7	7.63E-7
P_f(3 months)	1.57E-6	5.75E-6	6.27E-6	6.78E-6
P_f(6 months)	6.32E-6	2.30E-5	2.51E-5	2.71E-5
P_f(9 months)	1.44E-5	5.22E-5	5.68E-5	6.13E-5
P_f(1 year)	2.54E-5	9.21E-5	1.00E-4	1.08E-4

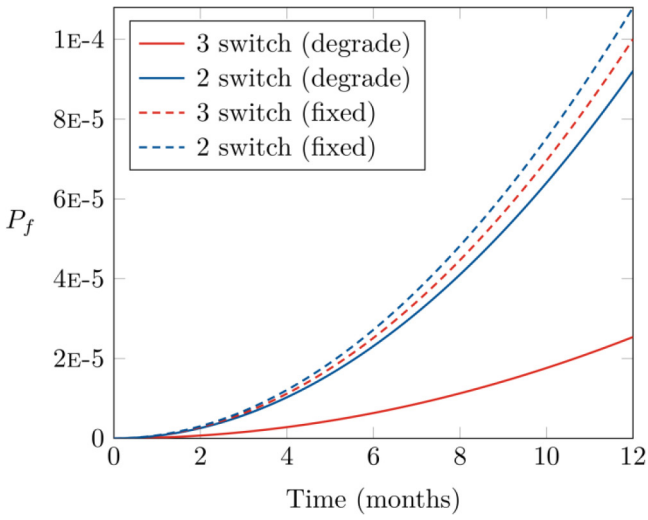

Figure 6. Failure probabilities of the four configurations as a function of mission time.

Interestingly, the reliability of the degradable two switch configuration was calculated to be better than that of the non-degradable three switch configuration (though only marginally). These results suggest that the ability to diagnose and remove a faulty switch may have a larger impact on system reliability than the initial number of switches. This is because, by not removing a permanently faulty switch from the configuration, the system becomes vulnerable to failure from a concurrently faulty end system (death state rule #3). The system's susceptibility to this type of failure can be better understood by examining the probability of entering each death state. Table 3 shows the relative probabilities of reaching the six most likely death states for both three switch cases over a one year mission. The death states are expressed by the tuple (EN, ET, EP, SN, ST, SP).

Table 3. The relative probabilities that system failure results from entering specific death states.

3 Switch (degrade)		3 Switch (fixed)	
Death state	Rel. prob.	Death state	Rel. prob.
(1,1,0,3,0,0)	0.894	(3,1,0,3,0,1)	0.680
(1,0,1,3,0,0)	0.089	(1,1,0,3,0,0)	0.228
(3,0,0,1,1,0)	0.008	(3,0,1,3,0,1)	0.068
(1,1,0,2,0,0)	0.006	(1,0,1,3,0,0)	0.023
(3,0,0,1,0,1)	0.001	(3,1,0,3,0,2)	0.001
(1,0,1,2,0,0)	0.001	(3,0,0,3,1,2)	0.001

When permanently faulty switches can be removed, the dominant cause of system failure is a fault in the remaining end system after already having degraded to a simplex configuration (accounting for approximately 99% of the total failure probability). On the other hand, when switches cannot be removed, there is roughly a 75%

chance that system failure results from a simultaneously faulty end system and switch. By enabling switch degradation, even if diagnosis and removal takes a relatively long time, the window during which the system is vulnerable to this type of failure can be greatly reduced.

As expected, solutions to the models are extremely sensitive to changes in the permanent and transient fault arrival rates. Using the fault arrival rates in Table 1, varying the degradation and recovery times for end systems and switches (e.g. from 10^{-9} to 1 hours) was found to have minimal impact on the predicted system reliability. However, at higher fault arrival rates (e.g. 10^{-4} /hr), the probability of system failure increases for large degradation and recovery times. The probabilities of misdiagnosing transient faults were found to significantly impact system reliability. However, because these values were initially estimated and would be highly implementation-dependent, no further sensitivity analysis was performed.

Summary/Conclusions

A major challenge in the incorporation of COTS hardware in spacecraft architectures is the need to fulfill the high reliability requirements of systems implementing critical functions (e.g. 10^{-9} failures/hour). This issue is compounded by the adoption of IMA principles and the need to prevent faults from propagating through shared resources and negatively impacting critical system services. To address this, we proposed an approach for realizing a Byzantine-resilient voting system that can be implemented using currently available COTS technologies. TTE is used for data exchange and synchronization between devices, eliminating the need for a dedicated cross-channel data link (CCDL) and external timing equipment. Though intended to meet the requirements of future manned spacecraft, this architecture could also be suitable for other types of ultra-dependable systems (e.g. commercial aircraft). Moreover, the underlying protocols (e.g. agreement, consensus) are applicable to any TTE network and can be scaled to accommodate any number of devices.

Though a 1FT design is possible with only two switches, the addition of a third switch minimizes the number of two-fault combinations resulting in system failure. Due to its use of switches in the agreement algorithm, the architecture is limited to performing two rounds of data exchange (to/from the switches) and therefore cannot be extended to tolerate two simultaneous Byzantine faults (even with sufficient FCRs). Preliminary reliability analysis comparing four configurations of the architecture suggest all are susceptible to failure for very long mission durations. However, the fault tolerance could be improved with sparing (which returns a degraded system to full strength) or an independent backup (if the system fails). A more accurate reliability analysis could be performed under a hybrid fault model, which considers different types of faults and their individual fault arrival rates. Still, a model is only capable of calculating the probability that the maximum fault assumptions are not violated. Formal proofs are needed to verify the correctness of the fault assumptions and proposed voting algorithms.

The fault hypothesis for this architecture assumes that end system controllers can fail arbitrarily. If their failure behavior can instead be restricted to the inconsistent omission failure mode (e.g. with a COM/MON design), then it is possible to ensure agreement of input data without voting on copies of data sent from multiple switches. However, voting would still be needed to determine the output of the triplex and mask the impact of a faulty host. If the hosts can also be restricted to this reduced failure model (e.g. via self-checking), then it is possible to implement simpler fault tolerance schemes that eliminate voting altogether (e.g. hot standby). However, satisfying this assumption generally requires specially designed hardware and limits the range of available processors which can be used.

References

1. Rushby, J., “Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance,” NASA Langley Technical Report Server, 1999.
2. McCabe, M. and Baggerman, C., “Avionics architecture interface considerations between constellation vehicles,” 2009 IEEE/AIAA 28th Digital Avionics Systems Conference, 1.E.2-1-1.E.2-10, 2009, doi:10.1109/DASC.2009.5347562.
3. Fletcher, M., “Progression of an Open Architecture: from Orion to Altair and LSS,” Proc. Fault-Tolerant Spaceborne Computing Employing New Technologies, Albuquerque, NM, 2009.
4. Rushby, J., “A Comparison of Bus Architectures for Safety-Critical Embedded Systems,” Computer Science Laboratory, SRI International, Menlo Park, CA, 2001.
5. Miner, P.S., Malekpour, M.R., and Torres-Pomales, W., Design of the Protocol Processor for the ROBUS-2 Communication System - NASA/TM-2005-213934, 2005.
6. Hodson, R.F., Chen, Y., Morgan, D.R., Butler, A.M., Sdhuh, J.M., Petelle, J.K., Gwaltney, D.A., Coe, L.D., Koelbl, T.G., and Nguyen, H.D., “Heavy Lift Vehicle (HLV) Avionics Flight Computing Architecture Study,” 2011.
7. Polsgrove, T., Chapman, J., Sutherlin, S., Taylor, B., Fabisinski, L., Collins, T., Dwyer Cianciolo, A., Samareh, J., Robertson, E., Studak, B., Vitalpur, S., Lee, A., and Rakow, G., “Human Mars Lander Design for NASA’s Evolvable Mars Campaign,” 2016.
8. Butler, R.W., “A Primer on Architectural Level Fault Tolerance,” 2008.
9. Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P., “Byzantine Fault Tolerance, from Theory to Reality,” in: Anderson, S., Felici, M., and Littlewood, B., eds., *Computer Safety, Reliability, and Security*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-20126-7: 235-248, 2003.
10. Driscoll, K.R., Hall, B., and Schweiker, K., “Application Agreement and Integration Services,” 2013.
11. Pease, M., Shostak, R., and Lamport, L., “Reaching Agreement in the Presence of Faults,” *J. ACM JACM* 27(2):228-234, 1980, doi:10.1145/322186.322188.
12. Lamport, L., Shostak, R., and Pease, M., “The Byzantine Generals Problem,” *ACM Trans Program Lang Syst* 4(3):382-401, 1982, doi:10.1145/357172.357176.
13. Singhal, M. and Shivaratri, N.G., “Advanced Concepts in Operating Systems,” McGraw-Hill, Inc., New York, NY, USA, ISBN 978-0-07-057572-1, 1994.
14. Diamantopoulos, P., Maneas, S., Patsonakis, C., Chondros, N., and Roussopoulos, M., “Interactive Consistency in practical, mostly-asynchronous systems,” *ArXiv14107256 Cs*, 2014.
15. Stine, D., “Digital Signatures for a Byzantine Resilient Computer System,” Masters, Massachusetts Institute of Technology, 1995.

16. Fischer, M.J., "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)," Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory, Springer-Verlag, London, UK, UK, ISBN 978-3-540-12689-8: 127-140, 1983.
17. Paulitsch, M., Morris, J., Hall, B., Driscoll, K., Latronico, E., and Koopman, P., "Coverage and the Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," Proceedings of the International Conference on Dependable Systems and Networks, IEEE, ISBN 978-0-7695-2282-1: 346-355, 2005, doi:10.1109/DSN.2005.31.
18. Lala, J.H. and Harper, R.E., "Architectural principles for safety-critical real-time applications," *Proc. IEEE* 82(1):25-40, 1994, doi:10.1109/5.259424.
19. Hanaway, J.F. and Moorehead, R.W., "Space shuttle avionics system," 1989.
20. Kouba, C., Buscher, D., and Busa, J.L., The X-38 Spacecraft Fault-Tolerant Avionics System, 2003.
21. Marchant, C.C., "Ares I Avionics Introduction," Seattle, WA, United States, 2009.
22. Kopetz, H., "The Fault Hypothesis for The Time-Triggered Architecture," *Building the Information Society*, Springer, Boston, MA: 221-233, 2004, doi:10.1007/978-1-4020-8157-6_20.
23. SAE International Aerospace Standard, "Time-Triggered Ethernet," SAE Standard AS6802, Rea. Nov. 2016.
24. Lincoln, P. and Rushby, J., "Formal Verification of an Interactive Consistency Algorithm for the Draper FTP Architecture Under a Hybrid Fault Model," Proceedings of the Ninth Annual Conference on Computer Assurance, Gaithersburg, MD: 107-120, 1994.
25. Azadmanesh, M.H. and Kieckhafer, R.M., "Exploiting Omissive Faults in Synchronous Approximate Agreement," *IEEE Trans. Comput.* 49(10):1031-1042, 2000, doi:10.1109/12.888039.
26. Obermaisser, R., "Time-Triggered Communication," CRC Press, ISBN 978-1-4398-4661-2, 19.
27. Powell, D., "Failure Mode Assumptions and Assumption Coverage," 1995.
28. Koopman, P., Driscoll, K., and Hall, B., "Selection of Cyclic Redundancy Code and Checksum Algorithms to Ensure Critical Data Integrity," *Dep. Electr. Comput. Eng.*, 2015.
29. Koopman, P., "32-bit cyclic redundancy codes for Internet applications," IEEE Comput. Soc, ISBN 978-0-7695-1597-7: 459-468, 2002, doi:10.1109/DSN.2002.1028931.
30. IEEE Standard for Ethernet, IEEE Std 802.3-2012, 2012.
31. Loveless, A.T., "On TTEthernet for Integrated Fault-Tolerant Spacecraft Networks," American Institute of Aeronautics and Astronautics, ISBN 978-1-62410-334-6, 2015, doi:10.2514/6.2015-4526.
32. Kopetz, H., From a Federated to an Integrated Architecture for Dependable Embedded Systems, 2004.
33. Miner, P.S., Malekpour, M., and Torres, W., "A conceptual design for a Reliable Optical Bus (ROBUS)," Proceedings. The 21st Digital Avionics Systems Conference, 13D3-1-13D3-11 vol.2, 2002, doi:10.1109/DASC.2002.1053014.
34. Hitt, E. and Mulcare, D., "Fault-Tolerant Avionics," 2001.
35. Butler, R.W. and White, A.L., "SURE Reliability Analysis: Program and Mathematics," 1988.
36. Butler, R.W. and Johnson, S.C., "Techniques for Modeling the Reliability of Fault-Tolerant Systems with the Markov State-Space Approach," 1995.
37. Johnson, S.C. and Boerschlein, D.P., "ASSIST User Manual," 1995.
38. Constantinescu, C., "Trends and Challenges in VLSI Circuit Reliability," *IEEE Micro* 23(4):14-19, 2003, doi:10.1109/MM.2003.1225959.
39. Siewiorek, D.P., "The Theory and Practice of Reliable System Design," Digital Press, Bedford, MA, ISBN 978-0-932376-13-8, 1982.
40. Butler, R.W. and Elks, C.R., "A Preliminary Transient-Fault Experiment on the SIFT Computer System," 1987.

Contact Information

Andrew Loveless
Command and Data Handling Branch (EV2)
NASA Johnson Space Center
andrew.loveless@nasa.gov

Acknowledgments

This work is conducted as part of the Avionics and Software (A&S) project, which is funded by NASA's Advanced Exploration Systems (AES) program.

The authors would like to thank Lisa Erickson (NASA/JSC EC6) for her assistance in reviewing this paper.

Definitions/Abbreviations

1FT - One Fault-Tolerant

ARINC - Aeronautical Radio, Incorporated

ASIC - Application-Specific Integrated Circuit

ASSIST - Abstract Semi-Markov Specification Interface to the SURE Tool

BER - Bit Error Rate

CCDL - Cross-Channel Data Link

CMOS - Complementary Metal-Oxide-Semiconductor

COM/MON - Command/Monitor

COTS - Commercial Off-The-Shelf

CRC - Cyclic Redundancy Check

CTID - Critical Traffic Identifier

DPS - Data Processing System

FCC - Flight Critical Computer

FCR - Fault Containment Region

FPGA - Field-Programmable Gate Array

FTP - Fault-Tolerant Processor

GPC - General Purpose Computer

HLV - Heavy Lift Vehicle

IC - Interactive Consistency

IMA - Integrated Modular Avionics

INU - Inertial Navigation Unit

IP - Internet Protocol

LaRC - Langley Research Center

MAC - Media Access Control

NASA - National Aeronautics and Space Administration

OM - Oral Messaging

OS - Operating System

RIU - Remote Interface Unit

ROBUS - Reliable Optical Bus

RTOS - Real-Time Operating System

SBC - Single-Board Computer

SIFT - Software-Implemented Fault-Tolerance

SM - Synchronization Master

SoC - System on a Chip

SPIDER - Scalable Processor-Independent Design for Extended Reliability

SURE - Semi-Markov Unreliability Range Evaluator

TTE - Time-Triggered Ethernet

TTP/C - Time-Triggered Protocol

UDP - User Datagram Protocol

VMC - Vehicle Management Computer

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. This process requires a minimum of three (3) reviews by industry experts.

ISSN 0148-7191

http://papers.sae.org/2017-01-2111