NASA/CR–2003–212161

A Comparison of Bus Architectures for Safety-Critical Embedded Systems

John Rushby
SRI International, Menlo Park, California

March 2003

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at (301) 621–0134
- Phone the NASA STI Help Desk at (301) 621–0390
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320

NASA/CR–2003–212161

A Comparison of Bus Architectures for Safety-Critical Embedded Systems

John Rushby
SRI International, Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681–2199

Prepared for Langley Research Center
under Cooperative Agreement NCC1-377

March 2003

Acknowledgments

I am grateful for helpful comments received from Bruno Dutertre, Kurt Liebel, Paul Miner, Ginger Shao, and Christian Tanzer.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076–1320
(301) 621–0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161–2171
(703) 605–6000

Abstract

Avionics and control systems for aircraft use distributed, fault-tolerant computer systems to provide safety-critical functions such as flight and engine control. These systems are becoming *modular*, meaning that they are based on standardized architectures and components, and *integrated*, meaning that some of the components are shared by different functions—of possibly different criticality levels.

The modular architectures that support these functions must provide mechanisms for coordinating the distributed components that provide a single function (e.g., distributing sensor readings and actuator commands appropriately, and assisting replicated components to perform the function in a fault-tolerant manner), while protecting functions from faults in each other. Such an architecture must tolerate hardware faults in its own components and must provide very strong guarantees on the correctness and reliability of its own mechanisms and services.

One of the essential services provided by this kind of modular architecture is communication of information from one distributed component to another, so a (physical or logical) communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. Consequently, these architectures are often referred to as *buses* (or *databuses*), although this term understates their complexity, sophistication, and criticality.

The capabilities once found in aircraft buses are becoming available in buses aimed at the automobile market, where the economies of scale ensure low prices. The low price of the automobile buses then renders them attractive to certain aircraft applications—provided they can achieve the safety required.

In this report, I describe and compare the architectures of two avionics and two automobile buses in the interest of deducing principles common to all of them, the main differences in their design choices, and the tradeoffs made. The avionics buses considered are the Honeywell SAFEbus (the backplane data bus used in the Boeing 777 Airplane Information Management System) and the NASA SPIDER (an architecture being developed as a demonstrator for certification under the new DO-254 guidelines); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA), recently adopted by Audi for automobile applications, and by Honeywell for avionics and aircraft control functions, and FlexRay, which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips.

I consider these buses from the perspective of their fault hypotheses, mechanisms, services, and assurance.

Contents

List of Figures

Chapter 1

Introduction

Embedded systems generally operate as closed-loop control systems: they repeatedly sample sensors, calculate appropriate control responses, and send those responses to actuators. In safety-critical applications, such as fly- and drive-by-wire (where there are no direct connections between the pilot and the aircraft control surfaces, nor between the driver and the car steering and brakes), requirements for ultra-high reliability demand fault tolerance and extensive redundancy. The embedded system then becomes a distributed one, and the basic control loop is complicated by mechanisms for synchronization, voting, and redundancy management.

Systems used in safety-critical applications have traditionally been *federated*, meaning that each "function" (e.g., autopilot or autothrottle in an aircraft, and brakes or suspension in a car) has its own fault-tolerant embedded control system with only minor interconnections to the systems of other functions. This provides a strong barrier to fault propagation: because the systems supporting different functions do not share resources, the failure of one function has little effect on the continued operation of others. The federated approach is expensive, however (because each function has its own replicated system), so recent applications are moving toward more integrated solutions in which some resources are shared across different functions. The new danger here is that faults may propagate from one function to another; *partitioning* is the problem of restoring to integrated systems the strong defenses against fault propagation that are naturally present in federated systems. A dual issue is that of *strong composability*: here we would like to take separately developed functions and have them run without interference on an integrated system platform with negligible integration effort.

The problems of fault tolerance, partitioning, and strong composability are challenging ones. If handled in an ad-hoc manner, their mechanisms can become the primary sources of faults and of *un*reliability in the resulting architecture [Mac88]. Fortunately, most aspects of these problems are independent of the particular functions concerned, and they can be handled in a principled and correct manner by generic mechanisms implemented as an architecture for distributed embedded systems.

One of the essential services provided by this kind of architecture is communication of information from one distributed component to another, so a (physical or logical) communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. Consequently, these architectures are often referred to as *buses* (or *databuses*), although this term understates their complexity, sophistication, and criticality. In truth, these architectures are the safety-critical core of the applications built above them, and the choice of services to provide to those applications, and the mechanisms of their implementation, are issues of major importance in the construction and certification of safety-critical embedded systems.

Capabilities and considerations once encountered only in buses for civil aircraft are now found in buses aimed at the automobile market, where the economies of scale ensure low prices. The low price of the automobile buses then renders them attractive to certain aircraft applications—provided they can achieve the safety required.

In this report, I describe and compare the architectures of two avionics and two automobile buses in the interest of deducing principles common to all of them, the main differences in their design choices, and the tradeoffs made. The avionics buses considered are the Honeywell SAFEbus [ARI93, HD92] (the backplane data bus used in the Boeing 777 Airplane Information Management System) and the NASA SPIDER [MMTP02] (an architecture being developed as a demonstrator for certification under the new DO-254 guidelines); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA) [TTT02, KG94], recently adopted by Audi for automobile applications, and by Honeywell for avionics and aircraft controls functions, and FlexRay [B^+01], which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips.

All four of the buses considered here are primarily *time triggered*; this is a fundamental design choice that influences many aspects of their architectures and mechanisms, and sets them apart from fundamentally *event-triggered* buses such as Controller Area Network (CAN), Byteflight, and LonWorks.

"Time triggered" means that all activities involving the bus, and often those involving components attached to the bus, are driven by the passage of time ("if it's 20 ms since the start of the frame, then read the sensor and broadcast its value"); this is distinguished from "event triggered," which means that activities are driven by the occurrence of events ("if the sensor reading changes, then broadcast its new value"). A time-triggered system interacts with the world according to an internal schedule, whereas an event-triggered system responds to stimuli that are outside its control.

The time-triggered and event-triggered approaches to systems design find favor in different application areas, and each has strong advocates. For integrated, safety-critical systems, however, the time-triggered approach is generally preferred. The reason is that an integrated system brings different applications ("functions" in avionics terms) together—whereas in a safety-critical system we usually prefer them to be kept apart! This is so that a failure in one application cannot propagate and cause failures in other applications; such protection against fault propagation is called *partitioning*, and it is most rigorously achieved (for reasons to be explained shortly) in time-triggered systems. Partitioning is a necessity in integrated safety-critical systems, but once achieved it also creates new opportunities. First, it simplifies the construction of fault-tolerant applications: such applications must be replicated across separate components with independent failure characteristics and no propagation of failures between them. Traditional "federated" architectures are typically hand-crafted to achieve these properties, but partitioning provides them automatically (indeed, they are the same as partitioning, reinterpreted to apply to the redundant components of a single application, rather than across different applications). Second, partitioning allows single applications to be "deconstructed" into smaller components that can be developed to different assurance levels: this can reduce costs and can also allow provision of new, safety-related capabilities. For example, an autopilot has to be developed to DO-178B assurance Level A [RTC92]; this is onerous and expensive, and a disincentive to introduction of desirable additional capabilities, such as extensive built-in self test (BIST). If the BIST could run in a separate partition, however, its assurance might be reduced to Level C, with corresponding reduction in its cost of development. Third, although the purpose of partitioning is to exclude fault propagation, it has the concomitant benefit that it promotes composability. A *composable* design is one in which individual applications are unaffected by the choice of the other applications with which they are integrated: an autothrottle, for example, could be developed, tested, and (in principle) certified, in isolation—in full confidence that it will perform identically when integrated into the same system as an autolander and a coffee maker.

Partitioning and composability concern the *predictability* of the resources and services perceived by the clients (i.e., applications and their subfunctions) of an architecture; predictability has two dimensions: *value* (i.e., logically correct behavior) and *time* (i.e., services are delivered at a predictable rate, and with predictable latency and jitter). It is temporal (time) predictability—especially in the presence of faults—that is difficult to achieve in event-triggered architectures, and thereby leaves time triggering as the only choice for safety-critical systems. The problem in event-driven buses is that events arriving at different nodes may cause them to contend for access to the bus, so some form of media access control (i.e., a distributed mutual exclusion algorithm) is needed to ensure that each node eventually is able to transmit without interruption. The important issue is how predictable is the access achieved by each node, and how strong is the assurance that the predictions remain true in the presence of faults.

Buses such as Ethernet resolve contention probabilistically and therefore can provide only probabilistic guarantees of timely access, and no assurance at all in the presence of faults. Buses for embedded systems such as CAN [ISO93], LonWorks [Ech99], or Profibus (Process Field Bus) [Deu95] use various priority, preassigned slot, or token schemes to resolve contention deterministically. In CAN, for example, the message with the lowest number always wins the arbitration and may therefore have to wait only for the current message to finish (though that message may be retransmitted in the case of transmission failure), while other messages also have to wait for any lower-numbered messages. Thus, although contention is resolved deterministically, latency increases with load and can be bounded with only probabilistic guarantees—and these can be quite weak in the presence of faults that cause some nodes to make excessive demands, thereby reducing the service available to others. Event-triggered buses for safety-critical applications add various mechanisms to limit such demands. ARINC 629 [ARI56] (an avionics data bus used in the Boeing 777), for example, uses a technique sometimes referred to as "minislotting" that requires each node to wait a certain period after sending a message before it can contend to send another. Even here, however, latency is a function of load, so the Byteflight protocol [Byt] developed by BMW extends this mechanism with guaranteed, preallocated slots for critical messages. At this point, however, we are close to a time-triggered bus, and if we were to add mechanisms to provide fault tolerance and to contain the effects of node failures, then we would arrive at a design similar to one of the time-triggered buses that is the focus of this comparison.

In a time-triggered bus, there is a static preallocation of communication bandwidth in the form of a global schedule: each node knows the schedule and knows the time, and therefore knows when it is allowed to send messages, and when it should expect to receive them. Thus, contention is resolved at design time (as the schedule is constructed), when all its consequences can be examined, rather than at runtime. Because all communication is time triggered by the global schedule, there is no need to attach source or destination addresses to messages sent over the bus: each node knows the sender and intended recipients of each message by virtue of the time at which it was sent. Elimination of the address fields not only reduces the size of each message, thereby greatly increasing the message bandwidth of the bus (messages are typically short in embedded control applications), but it also eliminates a potential source of serious faults: the possibility that a faulty node may send messages to the wrong recipients or, worse, may masquerade as a sender other than itself.

Time-triggered operation provides efficiency, determinism, and partitioning, but at the price of flexibility. To reduce this limitation, most time-triggered buses are able to switch among several schedules. The different schedules may be optimized for different missions, or phases of a mission (e.g., startup vs. cruise), for operating in a degraded mode (e.g., when some major function has failed), or for optional equipment (e.g., for cars with and without traction control). In addition, some make provision for event-triggered services, either "piggybacked" on time-triggered mechanisms, or "timesharing" between time- and event-triggered operation. Flexibility of operation is considered in more detail in Section 2.7.

Figure 1.1 portrays a generic bus architecture: application programs run in the *host* computers, while the *interconnect* medium provides broadcast communications; *interface* devices connect the hosts to the interconnect. All components inside the dashed box are considered part of the bus. Realizations of the interconnect may be a physical bus, as shown in Figure 1.2, or a centralized hub, as shown in Figure 1.3. The interfaces may be physically proximate to the hosts, or they may form part of a more complex central hub, as shown in Figure 1.4.

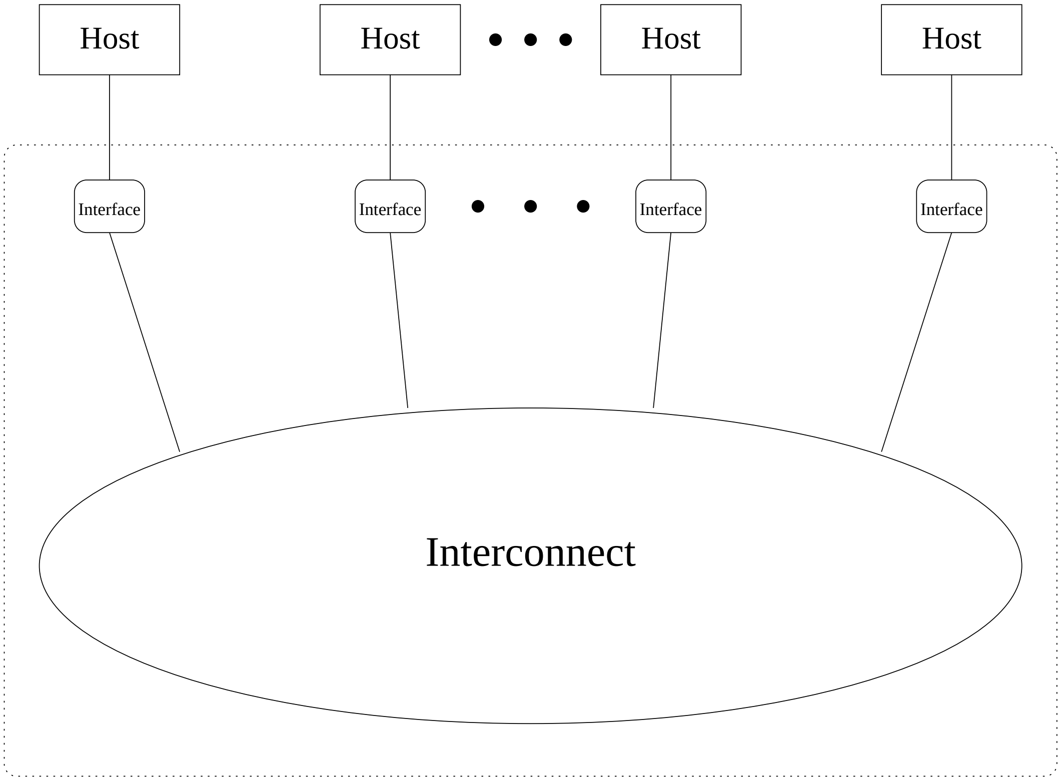

Figure 1.1: Generic Bus Configuration

Safety-critical aerospace functions (those at Level A of DO-178B) are generally required to have failure rates less than 10^{-9} per hour, and an architecture that is intended to support several such functions should provide assurance of failure rates better than 10^{-10} per hour.[1] Consumer-grade electronics devices have failure rates many orders of magnitude worse than this, so redundancy and fault tolerance are essential elements of a bus architecture. Redundancy may include replication of the entire bus, of the interconnect and/or the interfaces, or decomposition of those elements into smaller subcomponents that are then replicated. These topics are considered in more detail in Section 2.1.

Fault tolerance takes two forms in these architectures: first is that which ensures that the bus itself does not fail, second is that which eases the construction of fault-tolerant ap-

[1] An explanation for this figure can be derived by considering a fleet of 100 aircraft, each flying 3,000 hours per year over a lifetime of 33 years (thereby accumulating about 10^7 flight-hours). The requirement is that no fault should lead to loss of an aircraft in the lifetime of the fleet [FAA88]. If hazard analysis reveals ten potentially catastrophic failure conditions in each of ten systems, then the "budget" for each is about 10^{-9} [LT82, page 37]. Similar calculations can be performed for cars—higher rates of loss are accepted, but there are vastly more of them. See the MISRA guidelines [MIS94]. Also note that failure includes malfunction as well as loss of function.

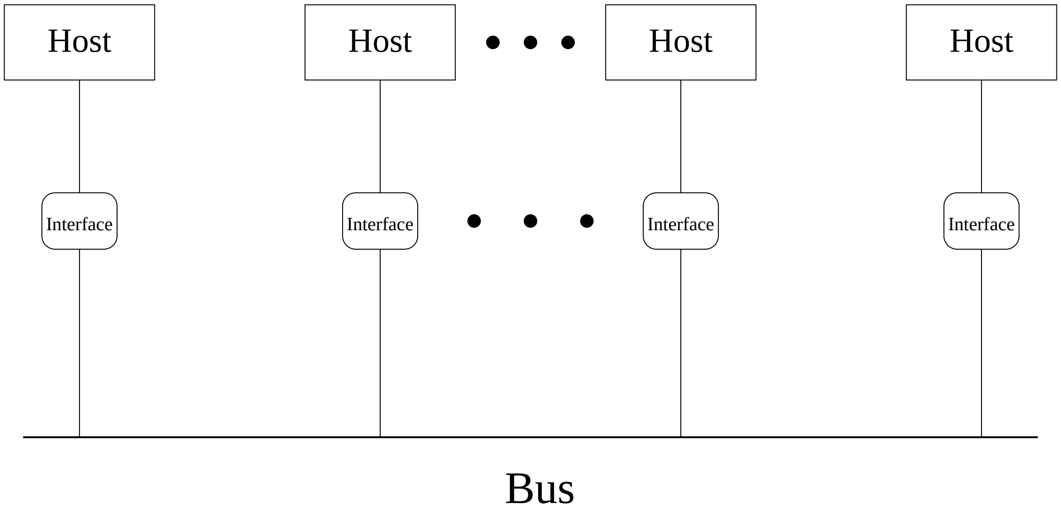

Figure 1.2: Bus Interconnect

plications. Each of these mechanisms must be constructed and validated against an explicit *fault hypothesis*, and must deliver specified *services* (that may be specified to degrade in acceptable ways in the presence of faults). The fault hypothesis must describe the *kinds* (or *modes*) of faults that are to be tolerated, and their maximum *number* and *arrival rate*. These topics are considered in more detail in Section 2.2.

Although the bus *as whole* must not fail, it may be acceptable for service to some specified number of hosts to fail in some specified manner (typically "fail silent," meaning no messages are transmitted to or from the host); also, some host computers may themselves fail. In these circumstances (when a host has failed, or when the bus is unable to provide service to a host), applications software in other hosts must tolerate the failure and continue to provide the function concerned. For example, three hosts might provide an autopilot function in a triple modularly redundant (TMR) fashion, or two might operate in a master/shadow manner. To coordinate their fault-tolerant operation, redundant hosts may need to maintain identical copies of relevant state data, or may need to be notified when one of their members becomes unavailable. The bus can assist this coordination by providing application-independent services such as interactively consistent message reception and group membership. These topics are considered in more detail in Section 2.6.

The global schedule of a time-triggered system determines when each node (i.e., host and corresponding interface) can access the interconnect medium. A global schedule requires a global clock and, for the reasons noted above, this clock must have a reliability of about 10^{-10}. It might seem feasible to locate a single hardware clock somewhere in the bus, then distribute it to the interfaces, and to achieve the required reliability by replicating the whole bus. The difficulty with this approach is that the clocks will inevitably drift apart over time, to the point where the two buses will be working on different parts of the

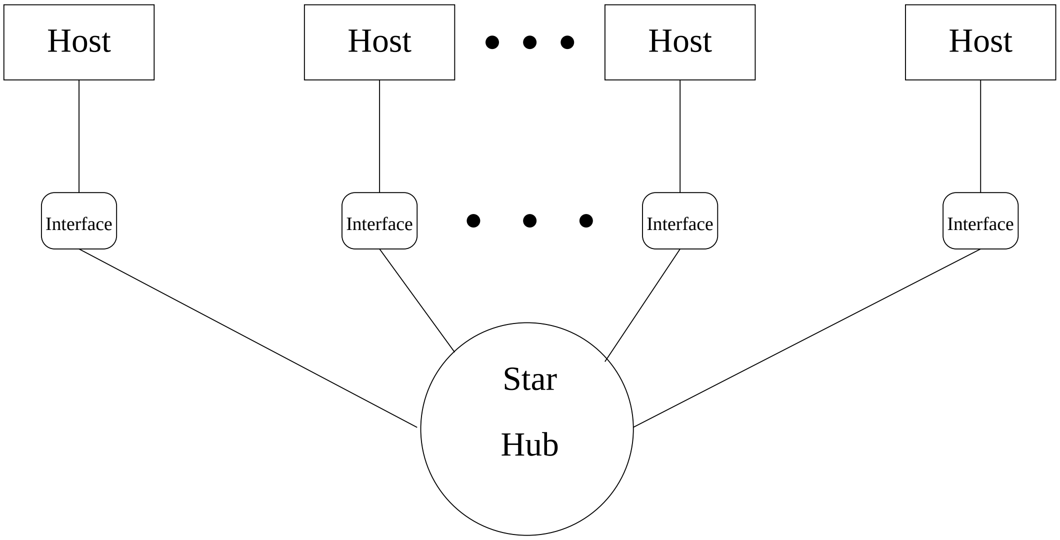

Figure 1.3: Star Interconnect

schedule. This would present an unacceptable interface to the hosts, so it is clear that the clocks need to be synchronized. Two clocks do not suffice for fault-tolerant clock synchronization (we cannot tell which is wrong): at least four are required for the most demanding fault models [LMS85] (although three may be enough in certain circumstances [Rus94]). Rather than synchronize multiple buses, each with a single clock, it is better to replicate clocks within a single bus. Now, the hosts are full computers—equipped with clocks—so it might seem that they could undertake the clock synchronization. The difficulty with this approach is that bus bandwidth is dependent on the quality of clock synchronization (message “frames” must be separated by a gap at least as long as the maximum clock skew), and clock synchronization is, in turn, dependent on the accuracy with which participants can estimate the differences between their clocks, or (for a different class of algorithms) on how quickly the participants can respond to events initiated by another clock. Specialized hardware is needed to achieve either of these with adequate performance, and this rules out synchronization by the hosts. Instead, the clocks and their synchronization mechanisms are made part of the bus; some designs locate the clocks within the interconnect, while others locate them within the interfaces. The topic of clock synchronization is considered in more detail in Section 2.3.

A fault-tolerant global clock is a key mechanism for coordinating multiple components according to a global schedule. The next point to be considered is where the global schedule should be stored, and how the time trigger should operate. In principle, the schedule could be held in the host computers, which would then determine when to send messages to their interfaces for transmission: in this case, the interfaces would perform only low-level (physical layer) protocol services. However, effective bus bandwidth depends on the global

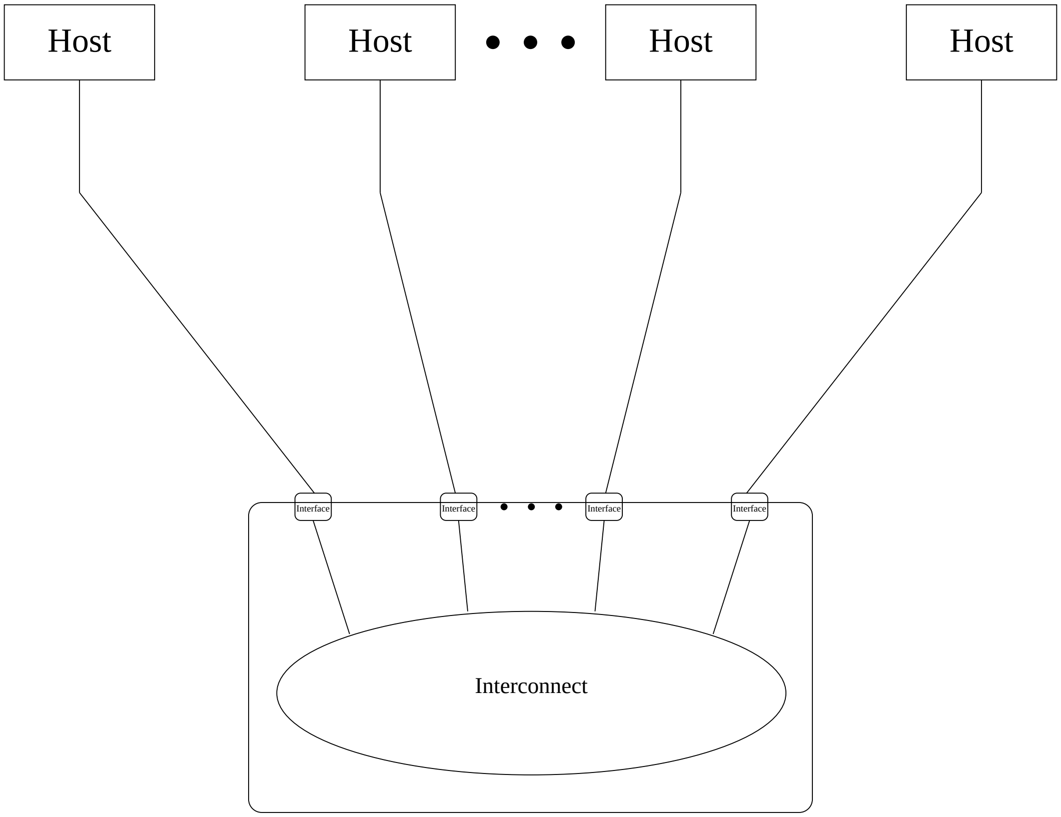

Figure 1.4: SPIDER Interconnect

schedule being tight, with little slack for protocol processing delays or interrupt latency. As with clock synchronization, this argues for hardware assistance in message timing and transmission. Hence, most of the buses considered here hold the schedule in the interface units, which then take on responsibility for most of the protocol services associated with the bus.

Although buses differ in respect to the fault hypotheses they consider, all those that place responsibility for scheduling in the interface units must consider the possibility that some of these may fail in a way that causes them to transmit on the interconnect at the wrong time, thereby excluding or damaging properly timed transmissions from other interfaces. The worst manifestation of this failure is the so-called “babbling idiot” failure where a faulty interface transmits constantly, thereby compromising the operation of the entire bus. To control this failure, it is necessary to introduce another component, called a *guardian* that restricts the ability of an interface to transmit on the interconnect. A guardian should fail independently of the interfaces, and have independent access to the schedule and to the global time. There are many ways to implement the guardian functionality. For example, we could duplicate each interface and arrange it so that the second instance acts as a check on the primary one (essentially, they are wired in series). The problem with this approach is the cost of providing a second duplicate for each interface. A lower-cost alternative reduces the functionality of the guardian at the expense of making it somewhat dependent on the primary interface. A third alternative locates the guardian functionality in the interconnect (specifically, in the hub of a star configuration) where its cost can be amortized over many interfaces, albeit at the cost of introducing a single point of failure (which is overcome by duplicating the entire hub). These topics are considered in more detail in Section 2.4.

It is nontrivial to start up a bus architecture that provides sophisticated services, especially if this must be performed in the presence of faults. Furthermore, it may be necessary to restart the system if faults outside its hypothesis cause it to fail: this must be done very quickly (within about 10 ms) or the overall system may go out of control. And it is necessary to allow individual hosts or interfaces that detect faults in their own operation to drop off the bus and later rejoin when they are restored to health. The topics of startup, restart, and rejoin are consider in Section 2.5.

Any bus architecture that is intended to support safety-critical applications, with its attendant requirement for a failure rate below 10^{-10}, must come with strong assurance that it is fit for the purpose. Assurance will include massive testing and fault injection of the actual implementation, and extensive reviews and analysis of its design and assumptions. Some of the analysis may employ formal methods, supported by mechanized tools such as model checkers and theorem provers [Rus95]. Industry or government guidelines for certification may apply in certain fields (e.g., [RTC92, RTC00] for airborne software and hardware, respectively, and [MIS94] for cars). These topics are considered in more detail in Section 2.8.

The comparison between the four bus architectures is described in the next chapter; conclusions are presented in Chapter 3.

Chapter 2

Comparison

We begin with a brief description of the topology and operation of each of the four bus architectures, and then consider each of them with respect to the issues introduced in the previous chapter. Within each section, the buses are considered in the order of their date of first publication: SAFEbus, TTA, SPIDER, FlexRay. Certain paragraphs are labeled in the margin by keywords that are intended to aid navigation.

2.1 The Four Buses

We describe the general characteristics of each of the bus architectures considered.

2.1.1 SAFEbus

SAFEbus$^{\text{TM}}$was developed by Honeywell (the principal designers are Kevin Driscoll and Ken Hoyme [HDHR91, HD92, HD93]) to serve as the core of the Boeing 777 Airplane Information Management System (AIMS) [SD95], which supports several critical functions, such as cockpit displays and airplane data gateways. The bus has been standardized as ARINC 659 [ARI93], and variations on Honeywell's implementation are being used or considered for other avionics and space applications.

SAFEbus uses a bus interconnect topology similar to that shown in Figure 1.2; the interfaces (called Bus Interface Units, or BIUs) are duplicated, and the interconnect bus is quad-redundant; in addition, the whole AIMS is duplicated. Most of the functionality of SAFEbus is implemented in the BIUs, which perform clock synchronization and message scheduling and transmission functions. Each BIU acts as its partner's bus guardian by controlling its access to the interconnect. Each BIU of a pair drives a different pair of interconnect buses but is able to read all four; the interconnect buses themselves each comprise two data lines and one clock line. The bus lines and their drivers have the electrical characteristics of OR gates (i.e., if several different BIUs drive the same line at the same time, the resulting signal is the OR of the separate inputs). Some of the protocols exploit this property.

Of the architectures considered here, SAFEbus is the most mature—it has been keeping Boeing 777s in the air for nearly a decade—but it is also the most expensive: the BIUs provide rich functionality and are fully duplicated at each node.

2.1.2 TTA

The Time-Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [KG93, KG94, MBSP02]. Commercial development of the architecture is undertaken by TTTech and it is being deployed for safety-critical applications in cars by Audi and Volkswagen, and for flight-critical functions in aircraft and aircraft engines by Honeywell (specifically, the FADECs of Honeywell's F124 engine for the Aermacchi M-346 fighter/trainer, and of General Electric's F110 engine for the F16 fighter) and Hamilton Sunstrand/Nord Micro (specifically, the cabin pressure control of the Airbus A380).

Current implementations of TTA use a bus interconnect topology similar to that shown in Figure 1.2; we will refer to this version as TTA-bus. The next generation of TTA implementations will use a star interconnect topology similar to that shown in Figure 1.3; we will refer to this version as TTA-star. The interfaces are essentially the same in both designs; they are called *controllers* and implement the TTP/C protocol [TTT02] that is at the heart of TTA, providing clock synchronization, and message sequencing and transmission functions. The interconnect is duplicated and each controller drives both copies. In TTA-bus, each controller drives the buses through a bus guardian; in TTA-star, the guardian functionality is implemented in the central hub. TTA-star can also be arranged in distributed configurations in which subsystems are connected by hub-to-hub links.

Of the architectures considered here, TTA is unique in being used for both automobile applications, where volume manufacture leads to very low prices, and aircraft, where a mature tradition of design and certification for flight-critical electronics provides strong scrutiny of arguments for safety.

2.1.3 SPIDER

A Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) is being developed by Paul Miner and colleagues at the NASA Langley Research Center [MMTP02] as a research platform to explore recovery strategies for radiation-induced high-intensity radiated fields/electromagnetic interference (HIRF/EMI) faults, and to serve as a case study to exercise the recent design assurance guidelines for airborne electronic hardware (DO-254) [RTC00].

The SPIDER interconnect is composed of active elements called Redundancy Management Units, or RMUs. Its topology can be organized either as shown in Figure 1.4, where the RMUs and interfaces (the BIUs) form part of a centralized hub, or as in Figure 1.3, where the RMUs form the hub, or similar to Figure 1.1, where the RMUs provide a distributed interconnect. The lines connecting hosts to their interfaces are optical fiber, and the whole system beyond the hosts (i.e., optical fibers and the RMUs and BIUs) is called the Reliable Optical Bus (ROBUS).

Clock synchronization and other services of SPIDER are achieved by distributed algorithms executed among the BIUs and RMUs [MMTP02]. The scheduling aspects of SPIDER are not well documented as yet, but the bus guardian functionality is handled in the RMUs following an approach due to Palumbo [Pal96].

SPIDER is an interesting design that uses a different topology and a different class of algorithms from the other buses considered here. However, its design and implementation are still in progress, and so it is omitted from some comparisons. I hope to increase coverage of SPIDER as more details become available.

2.1.4 FlexRay

FlexRay, which is being developed by a consortium including BMW, DaimlerChrysler, Motorola, and Philips, is intended for powertrain and chassis control in cars. It differs from the other buses considered here in that its operation is divided between time-triggered and event-triggered activities.

FlexRay can use either an "active" star topology similar to that shown in Figure 1.3, or a "passive" bus topology similar to that shown in Figure 1.2. In both cases, duplication of the interconnect is optional. Each interface (it is called a communication controller) drives the lines to its interconnects through separate bus guardians located with the interface. As with TTA-star, FlexRay can also be deployed in distributed configurations in which subsystems are connected by hub-to-hub links. Published descriptions of the FlexRay protocols and implementation are sketchy at present [B^{+}01, BMF02] (see also the Web site `www.flexray-group.com`).

FlexRay is interesting because of its mixture of time- and event-triggered operation, and potentially important because of the industrial clout of its developers. However, full details of its design are not available to the general public, so comparisons are based on the informal descriptions that have been published.

2.2 Fault Hypothesis and Fault Containment Units

Any fault-tolerant system must be designed and evaluated against an explicit *fault hypothesis* that describes the number, type, and arrival rate of the faults it is intended to tolerate. The fault hypothesis must also identify the different *fault containment units* (FCUs) in the design: these are the components that can *independently* be afflicted by faults. The division of an architecture into separate FCUs needs careful justification: there must be no propagation of faults from one FCU to another, and no "common mode failures" where a single physical event produces faults in multiple FCUs. We consider only physical faults (those caused by damage to, defects in, or aging of the devices employed, or by external disturbances such as cosmic rays, and electromagnetic interference): design faults must be excluded, and must be shown to be excluded by stringent assurance and certification processes.

It is a key assumption of all reliability calculations that failures of separate FCUs are statistically independent. Knowing the failure rate of each FCU, we can then use Markov or other stochastic modeling techniques [But92] to calculate the reliability of the overall architecture. Of course these calculations depend on claimed properties of the architecture (e.g., "this architecture can tolerate failures of any two FCUs"), and design assurance methods (e.g., formal verification) must be employed to justify these claims. The division of labor is that design assurance must justify a "theorem" of the form

enough nonfaulty components implies *correct operation*

and stochastic analysis must justify the antecedent to this theorem.

The assumption that failures of separate FCUs are independent must be ensured by careful design and assured by stringent analysis. True independence generally requires that different FCUs are served by different power supplies, and are physically and electrically isolated from each other. Providing this level of independence is expensive and it is generally undertaken only in aircraft applications. In cars, it is common to make some small compromises on independence: for example, the guardians may be fabricated on the same chip as the interface (but with their own clock oscillators), or the interface may be fabricated on the same chip as the host processor. It is necessary to examine these compromises carefully to ensure that the loss in independence applies only to fault modes that are benign, extremely rare, or tolerated by other mechanisms.

The fault *mode* mentioned above is one aspect of a fault hypothesis; the others are the total *number* of faults, and their *rate* of arrival. A fault mode describes the kind of behavior that a faulty FCU may exhibit. The same fault may exhibit different modes at different levels of a protocol hierarchy: for example, at the electrical level, the fault mode of a faulty line driver may be that it sends an intermediate voltage (one that is neither a digital 0 nor a digital 1), while at the message level the mode of the same fault may be "Byzantine," meaning that different receivers interpret the same message in different ways (because some see the intermediate voltage as a 0, and others as a 1). Some protocols can tolerate Byzantine faults, others cannot; for those that cannot, we must show that the fault mode is controlled at the underlying electrical level.

Spatial proximity fault

The basic dimensions that a fault can affect are value, time, and space. A *value* fault is one that causes an incorrect value to be computed, transmitted, or received (whether as a physical voltage, a logical message, or some other representation); a *timing* fault is one that causes a value to be computed, transmitted, or received at the wrong time (whether too early, too late, or not at all); a *spatial proximity* fault is one where all matter in some specified volume is destroyed (potentially afflicting multiple FCUs). Bus-based interconnects of the kind shown in Figure 1.2 are vulnerable to spatial proximity faults: all redundant buses necessarily come into close proximity at each node, and general destruction in that space could sever or disrupt them all. Interconnect topologies with a central hub are far more resilient in this regard: a spatial proximity fault that destroys one or more nodes does not disrupt communication among the others (the hub may need to isolate the lines to the destroyed nodes in case these are shorted), and destruction of a hub can be tolerated if there is a duplicate in another location.

Hybrid fault model

There are many ways to classify the effects of faults in any of the basic dimensions. One classification that has proved particularly effective in analysis of the types of algorithms that underlie the architectures considered here is the *hybrid* fault model of Thambidurai and Park [TP88]. In this classification, the effect of a fault may be *manifest*, meaning that it is reliably detected (e.g., a fault that causes an FCU to cease transmitting messages), *symmetric*, meaning that whatever the effect, it is the same for all observers (e.g., an off-by-1 error), or *arbitrary*, meaning that it is entirely unconstrained. In particular, an arbitrary fault may be *asymmetric* or *Byzantine*, meaning that its effect is perceived differently by different observers (as in the intermediate voltage example).

Arbitrary faults

The great advantage to designs that can tolerate arbitrary fault modes is that we do not have to justify assumptions about specific fault modes: a system is shown to tolerate (say) two arbitrary faults by proving that it works in the presence of two faulty FCUs with *no assumptions whatsoever* on the behavior of the faulty components. A system that can tolerate only specific fault modes may fail if confronted by a different fault mode, so it is necessary to provide assurance that such modes cannot occur. It is this *absence* of assumptions that is so attractive in safety-critical contexts about systems that can tolerate arbitrary faults. This point is often misunderstood and such systems are often derided as being focused on asymmetric or Byzantine faults, "which never arise in practice." Byzantine faults are just one manifestation of arbitrary behavior, and they certainly cannot be asserted not to occur (in fact, they have been observed in several systems that have been monitored sufficiently closely). One situation that is likely to provoke asymmetric manifestations is a *slightly out of specification* (SOS) fault, such as the intermediate electrical voltage mentioned earlier. SOS faults in the timing dimension include those that put a signal edge very close to a clock edge, or that have signals with very slow rise and fall times (i.e., weak edges). Depending on the timing of their own clock edges, some receivers may recognize and latch such a signal, others may not, resulting in asymmetric or Byzantine behavior.

SOS faults

Active/Passive faults

FCUs may be active (e.g., a processor) or passive (e.g., a bus); while an arbitrary-faulty active component can do anything, a passive component may change, lose, or delay data, but it cannot spontaneously create a new datum. Keyed checksums or cryptographic signatures can sometimes be used to reduce the fault modes of an active FCU to those of a passive one. (An arbitrary-faulty active FCU can always create its own messages, but it cannot create messages purporting to come from another FCU if it does not know the key of that FCU; signatures need to be managed carefully for this reduction in fault mode to be credible [GLR95].)

Maximum number of faults

Any fault-tolerant architecture will fail if subjected to too many faults; generally speaking, it requires more redundancy to tolerate an arbitrary fault than a symmetric one, which in turn requires more redundancy than a manifest fault. The most effective fault-tolerant algorithms make this tradeoff automatically between number and difficulty of faults tolerated. For example, the clock synchronization algorithm of [Rus94] can tolerate a arbitrary faults, s symmetric, and m manifest ones simultaneously provided n, the number of FCUs, satisfies $n > 3a + 2s + m$. It is provably impossible (i.e., it can be proven that no algorithm can exist) to tolerate a arbitrary faults in clock synchronization with fewer than $3a+1$ FCUs and $2a+1$ disjoint communication paths (or $a+1$ disjoint broadcast channels) [DHS86,FLM86] (unless digital signatures are employed—which is equivalent to reducing the severity of the arbitrary fault mode). Synchronization is approximate (i.e., the clocks of different FCUs need to be close together, not exactly the same); those problems that require exact agreement (e.g., group membership, consensus, diagnosis) cannot be solved in the presence of a arbitrary faults unless there are at least $3a + 1$ FCUs, $2a + 1$ disjoint communication paths (or $a + 1$ disjoint broadcast channels) between them, and $a + 1$ levels (or "rounds") of communication [Lyn96]. The number of FCUs and the number of disjoint paths required, but not the number of rounds, can be reduced by using digital signatures.

Self-checking

Because it is algorithmically much easier to tolerate simple failure modes, some architectures (e.g., SAFEbus) arrange FCUs (the BIUs in the case of SAFEbus) in self-checking pairs: if the members of a pair disagree, they go offline, ensuring that the effect of their failure is seen as a manifest fault (i.e., one that is easily tolerated). The controllers and bus guardians in TTA-bus operate in a similar way. Most architectures also employ substantial self-checking in each FCU; any FCU that detects a fault will shut down, thereby ensuring that its failure will be manifest. (This kind of operation is often called *fail silence*). Even with extensive self-checking and pairwise-checking, it may be possible for some fault modes to "escape," so it is generally necessary to show either that the mechanisms used have complete coverage (i.e., there will be no violation of fail silence), or to design the architecture so that it can tolerate the "escape" of at least one arbitrary fault.

Fail silence

Reconfiguration

Many architectures can tolerate only a single fault at a time, but can reconfigure to exclude faulty FCUs and are then able to tolerate additional faults. Prior to reconfiguration, it is necessary to identify the faulty component(s); algorithms to do this are based on distributed diagnosis and group membership. It is provably impossible to correctly identify an arbitrary-faulty FCU in some circumstances [WLS97], so there is tension between leaving a faulty FCU in the system and the risk of excluding a nonfaulty one (and still leaving the faulty one in operation). Most faults are *transient*, meaning they correct themselves given a little time, so there is also tension between the desire to exclude faulty FCUs quickly, and the hope that they may correct themselves if left in operation [Rus96]. A transient fault may contaminate state data in a way that leaves a permanent residue after the original fault has cleared, so mechanisms are needed to purge such effects [Rus93a].

Reconfig rate

Given decisions about the diagnosis and reconfiguration strategy, it is possible to calculate the time taken to perform these operations for given fault modes and, hence, to calculate the maximum rate at which diagnosis and reconfiguration can occur. The architectures considered here operate according to static schedules, which consist of “rounds” or “frames” that are executed repeatedly in a cyclic fashion. The *reconfiguration rate* is often then expressed in terms of faults per round (or the inverse). It is usually important that every node is scheduled to make at least one broadcast in every round, since this is how fault status is indicated (and hence how reconfiguration is triggered).

Fault arrival rate

The reconfiguration rate determines how rapidly the architectures can recover from the effects of prior faults and so be ready to tolerate another. The *fault arrival rate* is the hypothesized rate at which faults (of different kinds) actually occur. Reliability modeling must analyse the fault arrival rate and mission time against the fault tolerance and reconfiguration rate of the architecture to determine if it can satisfy the mission goals.

Rejoin

An excluded FCU may perform a restart and self check. If successful, it may then apply to rejoin the system. This is a delicate operation for most architectures, because one FCU may be going faulty at the same time as another (nonfaulty) one is rejoining: this presents two simultaneous changes in the behavior of the system and may cause algorithms tolerant of only a single fault to fail.

Never give up

Historical experience and analysis must be used to show that the hypothesized modes, numbers, and arrival rate are realistic, and that the architecture can indeed operate correctly under those hypotheses for its intended mission time. But sometimes things go wrong: the system may experience many simultaneous faults (e.g., from unanticipated HIRF), or other violations of its fault hypothesis. We cannot guarantee correct operation in such cases (otherwise our fault hypothesis was too conservative), but safety-critical systems generally are constructed to a “never give up” philosophy and will attempt to continue operation in a degraded mode. Although it is difficult to provide assurance of correct operation during these events (otherwise we could revise the fault hypothesis), it may be possible to provide assurance that the system returns to normal operation once the faults cease (assuming they were transients) using the ideas of self-stabilization [Sch93].

The usual method of operation in “never give up” mode is that each node reverts to local control of its own actuators using the best information available (e.g., each brake node applies braking force proportional to pedal pressure if it is still receiving that input, and removes all braking force if not), while at the same time attempting to regain coordination with its peers.

2.2.1 SAFEbus

The FCUs of SAFEbus are the BIUs (two per node), the hosts, and the buses (there are two, each of which is a self-checking pair). In addition, two copies of the entire system are located in separate cabinets in different parts of the aircraft. ARINC 659 shows a single host attached to each pair of BIUs [ARI93, Attachment 2–1], but in the Honeywell implementation these are also paired: each host (called a Core Processing Module, or CPM) is attached to a single BIU, and forms a separate FCU.

The fault hypothesis of SAFEbus is the following.

Fault modes:

- Arbitrary active faults in BIUs and CPMs
- Arbitrary passive faults in the buses
- Spatial proximity faults that may take out an entire cabinet

Maximum faults: SAFEbus adopts a single-fault hypothesis: at most one component of any pair may fail. In more detail, the fault hypothesis of SAFEbus allows the following numbers of faults.

- At most one of the BIUs in any node (the entire node is then considered faulty)
- At most one of the CPMs in a node (the entire node is then considered faulty)
- At most one fault in either of the two buses
- Any number of nodes may fail, but for an application to be functional, at least one node that supports it must be nonfaulty

Fault arrival rate:

- Able to tolerate any rate

2.2.2 TTA

The FCUs of TTA depend on how the system is fabricated. It is anticipated that in high-volume applications, TTP controllers will be integrated on the same chip as the host, so these should be considered to belong to a single FCU. Current implementations of TTP-bus have the controller separate from the host, but the bus guardians are on the same chip as the controller. Guardians do, however, have their own clock oscillator, so they can be considered a separate FCU for time-partitioning purposes. TTA-star moves the guardians to the central hub, where they definitely form a separate FCU from the controllers. TTA-bus has two bus lines, which are separate FCUs; in TTA-star, the interconnect functionality is provided by the central hubs (so the bus guardian and interconnect are in the same FCU), which are duplicated.

The fault hypothesis of TTA is the following.

Fault modes:

- Arbitrary active faults in controllers and the hub of TTA-star
- Arbitrary passive faults in the guardians and buses of TTA-bus
- Spatial proximity faults that may take out nodes and a hub in TTA-star; spatial-proximity faults are not part of the fault hypothesis for TTA-bus

Maximum faults: TTA adopts a single-fault hypothesis. In more detail, the fault hypothesis of TTA assumes the following numbers of faults.

- For TTA-bus: in each node either the controller or the bus guardian may fail (but not both). One of the buses may fail. To retain single fault tolerance, at least four controllers and their bus guardians must be nonfaulty, and both buses must be nonfaulty. Provided at least one bus is nonfaulty, the system may be able to continue operation with fewer nonfaulty components.
- For TTA-star: to retain single fault tolerance, at least four controllers and both hubs must be nonfaulty. Provided at least one hub is nonfaulty, the system may be able to continue operation with fewer nonfaulty components.

Fault arrival rate:

- At most one fault every two rounds

2.2.3 SPIDER

The FCUs of SPIDER are the hosts, the BIUs, and the RMUs; a host and its BIU may form a single FCU in some implementations. The BIUs and RMUs may be distributed, or contained in the hub of the ROBUS.

The fault hypothesis of SPIDER is the following.

Fault modes:

- Arbitrary active faults in any FCU
- Spatial proximity faults that may take out nodes and RMUs (depending on the physical topology employed)

Maximum faults: If there are $n \geq 3$ BIUs and $m \geq 3$ RMUs, then SPIDER can tolerate an arbitrary fault in any one of these FCUs. SPIDER's maximum faults hypothesis is actually specified with respect to the hybrid fault model, and its constraints are $n > 2ba + 2bs + bm$, $m > 2ra + 2rs + rm$, and either $ba = 0$ or $ra = 0$, where *ba*, *bs*, and *bm* are the numbers of arbitrary-, symmetric- and manifest-faulty BIUs, and *ra*, *rs*, and *rm* are the corresponding numbers for the RMUs.

Fault arrival rate: Within the constraints presented above, SPIDER is able to tolerate multiple simultaneous faults. SPIDER's reconfiguration mechanism is graceful degradation: FCUs diagnosed as faulty may be ignored by nonfaulty FCUs. Its fault diagnosis algorithms, derived from those of [WLS97], satisfy strong properties: any FCU diagnosed as faulty is indeed faulty, and all maifest- and symmetric-faulty FCUs (and many abritrary-faulty ones) will be diagnosed [GM02, GM03]. The fault arrival rate hypothesis is a function of the amount of redundancy in the ROBUS, and can be adjusted within certain parameters by employing different numbers of BIUs and RMUs. Reliability modeling and analysis using the SURE program [But92] are described in [MMTP02].

2.2.4 FlexRay

Published diagrams of FlexRay indicate that a node consisting of a microcontroller host, a communication controller, and two bus guardians will be fabricated on a single chip. It appears that all four components will use separate clock oscillators, so that the controller and guardians can be considered as separate FCUs for time-partitioning purposes. The interconnects, whether passive buses or active stars, are separate FCUs.

The fault hypothesis of FlexRay is not stated explicitly; the following are inferences based on available documents.

Fault modes:

- Asymmetric (and presumably, therefore, also arbitrary) faults in controllers *for the purposes of clock synchronization*
- Fault modes for other services and components are not described
- Spatial proximity faults may take out nodes and an entire hub

Maximum faults:

- It appears that a single-fault hypothesis is intended: in each node, at most one bus guardian, or the controller, may be faulty. At most one of the interconnects may be faulty.
- For clock synchronization, fewer than a third of the nodes may be faulty.

Fault arrival rate: The fault arrival rate hypothesis is not described.

2.3 Clock Synchronization

Fault-tolerant clock synchronization is a fundamental requirement for a time-triggered bus architecture. Tightness of the bus schedule, and hence the throughput of the bus, is strongly related to the quality of global clock synchronization that can be achieved—and this is related to the quality of the clock oscillators local to each node, and to the algorithm used to synchronize them. There are two basic classes of algorithm for clock synchronization: those based on averaging and those based on events. Averaging works by each node measuring the skew between its clock and that of every other node, and then setting its clock to some "average" value. A simple average (e.g., the mean or median) over all clocks may be affected by wild readings from faulty clocks (which, under an arbitrary fault model, may provide different readings to different observers), so we need a "fault-tolerant average" that is largely insensitive to a certain number of readings from faulty clocks. Event-based algorithms rely on nodes being able to sense events directly on the interconnect: each node broadcasts a "ready" event when it is time to synchronize and sets its clock when it has seen a certain number of events from other nodes. Depending on the fault model, additional waves of “echo” or “accept” events may be needed to make this fault tolerant.

Schneider [Sch87] gives a general description that applies to all averaging clock synchronization algorithms; these algorithms differ only in their choice of “fault-tolerant average.” The Welch-Lynch algorithm [WL88] is a popular choice that is characterized by use of the “fault-tolerant midpoint” as its averaging function. We assume n clocks and the maximum number of simultaneous faults to be tolerated is t ($3t < n$); the fault-tolerant midpoint is the average of the $t + 1$’st and $n - t$’th clock reading, when these are arranged in order from smallest to largest. If there are at most t faulty clocks, then some reading from a nonfaulty clock must be at least as small as the $t+1$’st reading, and the reading from another nonfaulty clock must be at least as great as the $n - t$’th; hence, the average of these two readings should be close to the middle of the spread of readings from good clocks.

The most important event-based algorithm is that of Srikanth and Toueg [ST87]; it is attractive because it achieves optimal accuracy. Both averaging and event-based algorithms require at least $3a + 1$ nodes to tolerate a arbitrary faults.

2.3.1 SAFEbus

The SAFEbus bus is quad-redundant (a pair of self-checking pairs) and each of its four components comprises two data lines and a separate clock line. SAFEbus uses the clock lines for an event-triggered clock synchronization algorithm. The schedule loaded in each interface (BIU in SAFEbus terminology) indicates when a synchronization event should be performed, and these must be sufficiently frequent to maintain the paired BIUs of each node within two bit-times of each other.

In a clock synchronization event, each BIU asserts the clock lines of the two buses that it can write for four bit-times. The electrical characteristics of the SAFEbus cause it to act as an OR gate with the BIUs as its inputs. Thus, the near-simultaneous assertion of each clock line by multiple BIUs generates a pulse on each line that is the OR of its individual pulses. Each BIU synchronizes to the trailing edge of this composite pulse.

A faulty BIU could attempt to assert its clock lines for far longer than the specified four bit-times, thereby delaying the trailing edge that is the global synchronization event. The guardian function of its partner BIU will cut it off once the transmit window closes, and all receiving BIUs will count it out after some number of bit-times greater than four, but the synchronization event will still be delayed. However, this fault affects only the two buses driven by the faulty BIU. Each BIU reads all four buses (although it can write only two of them), detects the trailing edge of the composite synchronization pulse on each of them, and then combines these in a fault-tolerant manner [ARI93, Attachment 4–10] to yield the event to which it actually synchronizes.

SAFEbus also applies several other error detection and masking techniques to minimize the impact of faulty clocks, BIUs, and buses: for example, “pulses” are ignored from buses that have not changed state since the previous synchronization (to overcome stuck-at faults or failed power supplies).

There are several variants to the clock synchronization performed by SAFEbus: a “Short Resync” operates essentially as described above; a “Long Resync” is similar but provides additional information on the data lines to allow an unsynchronized BIU to rejoin the system; an “Initial Sync” is used at startup or following a disruption that requires a restart.

2.3.2 TTA

The TTA algorithm is basically the Welch-Lynch algorithm specialized for $t = 1$ (i.e., it tolerates a single fault): that is, clocks are set to the average of the 2nd and $n - 1$'st clock readings (i.e., the second-smallest and second-largest). This algorithm works and tolerates a single arbitrary fault whenever $n \geq 4$. TTA does not use dedicated wires or signaling to communicate clock readings among the nodes attached to the network; instead, it exploits the fact that communication is time triggered by a global schedule. When a node x receives a message from a node y, it notes the reading of its local clock and subtracts a fixed correction term to account for the network delay; the difference between this adjusted clock reading and the time for y's transmission that is indicated in the global schedule yields x's perception of the difference between clocks x and y.

Not all TTP nodes have accurate clock oscillators (because these are expensive); those that do have the `SYF` field set in the Message Descriptor List (MEDL—the global schedule known to all nodes) and the clocks used for synchronization are selected from those that have the `SYF` flag set.

For scalability, implementation of the Welch-Lynch algorithm should use data structures that are independent of the value of n—that is, it should not be necessary for each node to store the clock difference readings for all n clocks. Clearly, the t'th smallest clock difference reading can be determined with just t registers, and the t'th largest can be determined similarly, for a total of $2t$ registers per node. In TTA, with $t = 1$, this requires four registers. TTA does indeed use four registers, but not in quite this way. Each node maintains a queue of four clock-difference readings; whenever a message is received from a node that is in the current membership and that has the `SYF` field set, the clock difference reading is pushed on to the receiving node's queue (ejecting the oldest reading in the queue). When the current slot has the synchronization field (`CS`) set in the MEDL, each node runs the synchronization algorithm using the four clock readings stored in its queue.

This algorithm is able to tolerate a single arbitrary fault among the four clocks used in synchronization, but TTA is able to tolerate more than a single fault by reconfiguring to exclude nodes with faulty clocks. This is accomplished by the group membership service: any node with a clock that skews significantly from the global time will mistime its broadcast so that it occurs partially outside its assigned slot. This will cause its message to be truncated by its guardian, which will cause it to fail checksum and to be rejected by all nonfaulty nodes. The membership algorithm will then exclude this node. Only the clocks of current group members are eligible for use in synchronization, so the clock of the excluded node will not be placed in the clock-difference queue, and that of some other node having the SYF flag will be used instead.

2.3.3 SPIDER

SPIDER uses an event-based algorithm similar to that of Srikanth and Toueg, and also influenced by Davies and Wakerly [DW78] (this remarkably prescient paper anticipated many of the issues and solutions in Byzantine fault tolerance by several years) and Palumbo [Pal96].

The basic design of SPIDER is similar to that of the Draper FTP [Lal86] in that its active components are divided into two classes (BIUs and RMUs) that play slightly different rôles in each of its algorithms. In SPIDER, the RMUs play the part of the "interstages" in FTP. Its clock synchronization algorithm operates in three phases as follows.

1. Each RMU broadcasts a "ready" event to all BIUs when its own clock reaches a specified value.

2. Each BIU broadcasts an "accept" event to all RMUs as soon as it has received events from $t + 1$ RMUs (where t is the number of faults to be tolerated).

3. Each RMU resets its clock as soon as it has received events from $t + 1$ BIUs.

This synchronizes the RMUs; the BIUs can be synchronized by one more wave of events from the RMUs.

2.3.4 FlexRay

FlexRay uses the standard Welch-Lynch algorithm, with clock differences presumably determined in a manner similar to TTA. However, published descriptions of FlexRay indicate that it has no membership service and no mechanisms for detecting faulty nodes, nor for reconfiguring to exclude them. To tolerate two arbitrary faults (as claimed in published descriptions), FlexRay must therefore employ at least seven nodes with five disjoint communication paths or three broadcast channels ($3t + 1$, $2t + 1$, and $t + 1$, respectively, for $t = 2$), whereas TTA can do this with five nodes (providing the faults arrive sequentially)—and with seven nodes it can tolerate four sequential faults.

2.4 Bus Guardians

Some kind of bus guardian functionality is necessary to prevent faulty nodes usurping the scheduled time slots of other nodes or even—in the case of the "babbling idiot" fault mode—destroying all legitimate communication. Bus guardianship depends on message transmission by an interface to an interconnect being mediated by a separate FCU that has an independent copy of the schedule, and independent knowledge of the global time. Such a fully independent guardian is likely to be expensive, however, and equivalent almost to a second interface (as it is in SAFEbus). Most architectures, therefore, seek to reduce the cost of bus guardianship; they do so in different ways, and incur different penalties.

2.4.1 SAFEbus

SAFEbus makes no compromises: its BIUs are paired and each member of a pair acts as a guardian for the other. Each BIU performs its own clock synchronization and has its own copy of the schedule. As a result, SAFEbus is expensive: its nodes cost a few hundred dollars each.

2.4.2 TTA

In TTA-bus, the guardians have their own clock oscillators and independent copy of the schedule, but they are not able to synchronize independently, and they share the same power supply and physical environment as their controllers. Most of the functionality of a bus guardian is shared across both bus lines.

TTA-bus guardians are synchronized by a start-of-round signal received from their controller. If this signal is given at the wrong time, then the guardian will open its window at the wrong time and will allow its (presumably faulty) controller to transmit at that wrong time. However, its transmission will either collide with a legitimate transmission, resulting in garbage, or will hit the slot of an already excluded node, or some other unused part of the frame. In neither case will it be acknowledged, so the errant controller will shut down if it is "not too faulty" to follow the TTP/C protocol; in any case, the other nodes of the system will exclude both the errant node (since it will have failed to broadcast in its own slot) and the node (if any) whose slot it usurped, and will thereafter proceed unhindered. The errant node will not be able to repeat its trick because a guardian places tight limits on how far the start-of-round signal can move (which is enough to reduce this scenario to extremely low probability in the first place).

In TTA-star, guardian functionality is moved to the central hub. Since there is now only one guardian per interconnect, rather than one per node, more resources can be expended in its construction. The guardian in a hub is able to perform independent clock synchronization and is therefore a fully independent FCU, and provides full coverage against clock synchronization faults and babbling fault modes in all controllers. The penalty is that a central hub is a single point of failure, and as an active entity, it is probably less reliable than a passive bus. This penalty is overcome by duplication, which has the concomitant benefit of providing tolerance for spatial proximity faults.

2.4.3 SPIDER

The scheduling aspects of SPIDER are not well documented as yet, but the bus guardian functionality is handled in the RMUs following an approach due to Palumbo [Pal96].

2.4.4 FlexRay

Operation of the bus guardians of FlexRay is not described in any detail in the available documents. Published diagrams show two guardians per node sharing the same chip and power supply as the controller. These guardians presumably operate in a manner similar to those of TTA-bus, with similar vulnerabilities, though at greater cost (since two guardians per node presumably require two oscillators).

2.5 Startup and Restart

It obviously is necessary to be able to start up the bus and its associated components from cold—preferably with no outside assistance. Modern aircraft systems generally have enough redundancy to operate for several days with some components failed or faulty [Hop88]. This allows repairs to be deferred until the aircraft's schedule brings it to a major maintenance site. Car owners are notorious for deferring maintenance and for operating their vehicles with faults present. These characteristics make it necessary that startup can be performed correctly in the presence of faults.

Restart during operation may be necessary if HIRF or other environmental influences lead to violation of the fault hypothesis and thereby cause complete failure of the bus. Notice that this failure must be detected by the bus, and the restart must be automatic and very fast: most control systems can tolerate loss of control inputs (e.g., by reverting to some form of local control and either releasing the actuators, or freezing them in the previously commanded position) for only a few cycles: longer outages will lead to loss of control. For example, Heiner and Thurner of DaimlerChrysler estimate that the maximum transient outage time for a steer-by-wire automobile application is 50 ms [HT98]. Given that other (e.g., host-level) activities may need to be performed on restart, this suggests 10 ms as a reasonable goal for bus restart. The presence of faulty components could complicate, or even prevent restart (e.g., if multiple faults are present, but some of the algorithms can tolerate only single faults), so it is desirable that previous reconfigurations (e.g., that excluded those faulty components) should be recorded in a way that makes this information available during restart.

Restart is usually initiated when an interface detects no activity on any bus line for some interval; that interface will then transmit some "wake up" message on all lines. Of course, it is possible that the interface in question is faulty (and there was bus activity all along but that interface did not detect it), or that two interfaces decide simultaneously to send the "wake up" call. The first possibility must be avoided by careful checking, preferably by independent units (e.g., both interfaces of a pair, or an interface and its guardian); the second requires some form of collision detection and resolution: this should be deterministic to guarantee an upper bound on the time to reach resolution (that will allow a single interface to send an uninterrupted “wake up” message) and, ideally, should not depend on reliable collision detection (because there is no such thing).

Components that detect faults in their own operation, or that are notified of their faulty operation by others (e.g., through failed comparison with a paired component, or by exclusion from the group in a system that employs group membership) may drop off the bus and undergo local restart and self-test. If the test is successful (i.e., the fault was transient), then the component will attempt to reintegrate itself into the ongoing operations of the bus. This is a delicate operation because the sudden arrival of a new participant in the bus traffic can present symptoms rather like a fault—and can be particularly challenging to handle if a real fault is manifested simultaneously, or if another component rejoins at the same time.

Restart of the whole bus, and reintegration of individual components, can be interpreted as self-stabilizing services: self-stabilizing algorithms are those that converge to a stable state from any initial state [Dij74, Sch93]. Such algorithms generally assume that no faults are present (or that transient faults have ceased) once stabilization begins; in the circumstances considered here, however, it is possible that some residual faults remain during stabilization. Thus, the algorithms employed for restart and reintegration in bus architectures do not make explicit reference to self-stabilization, although this may provide an attractive framework for their formal analysis. Frameworks that integrate self-stabilization with fault tolerance have been proposed [AG93, GP93] that may provide a useful foundation for this endeavor.

2.5.1 SAFEbus

A SAFEbus node in the “out-of-sync” state listens on the bus for a Long Resync; if it finds one, it uses the information in that message to integrate itself into the ongoing activity of the bus. If a Long Resync is not detected for a certain length of time, the node transmits an Initial Sync message on all buses (note that both BIUs in the node must agree on this action). Due to the OR gate character of the SAFEbus lines, and the coding used for the Initial Sync message, it does no harm if several nodes attempt to send this message nearly simultaneously. After sending or receiving an Initial Sync message, a node waits a specified amount of time and then sends a Long Resync message; all nodes should be reintegrated and the bus restarted at this point.

SAFEbus uses the same mechanisms for cold start and restart; these are very fast, as nodes will send Initial Sync messages after a timeout that is little longer than a single round of the cyclic schedule, and the bus will be synchronized and operational in the round after that. Reintegration is even faster, as the reintegrating node need wait only for a Long Resync to be sent, and each node initiates at least one of these per round. The SAFEbus mechanisms are fully decentralized and very robust (e.g., they do not depend on collision detection).

2.5.2 TTA

TTA's mechanisms for cold start, restart, and reintegration are conceptually similar to those of SAFEbus, but cannot use the electrical properties of the bus (because TTA operates above this level, and can be used with any transmission technology) and are therefore rather more complex. In addition, TTA uses distributed group membership information, and it is necessary to initialize or update this consistently.

A TTA controller that has been excluded from the membership, or that has recently restarted, listens to activity on the bus until it recognizes an "I-Frame" message. These are broadcast periodically, and contain sufficient information to initialize the clock and membership vector of the controller. The controller then observes the bus activity for a further round to check that its "C-State" (i.e., the controller state information that is encoded in the CRCs attached to each message) is consistent with the other controllers on the bus, and thereafter resumes normal operation.

If no I-Frame is detected, the controller will transmit one itself after a certain interval, but on only one of the two buses (presumably this limitation to a single bus is intended to limit the harm caused by a faulty controller that attempts to cold start an already-functioning bus). The membership indicated in this I-Frame will contain only the controller that sends it. Other controllers that receive this I-Frame will synchronize to it and start executing the schedule indicated in their MEDLs. During the subsequent round of messages, each controller will add others to its membership when it observes their broadcasts. All nodes should be fully integrated by the end of the first round following the cold start I-Frame. Some ambiguities in the description of the state-machine that specifies these transitions [TTT02, Section 9.3] were identified in simulation by Bradbury [Bra00].

It is possible that two controllers could send a cold start I-Frame at the same time, resulting in a collision on the bus. This should cause no recognizable message to be received; the two initiating controllers will have different timeouts, so their subsequent attempts will not collide, and one of them will succeed in starting the bus (modulo the possibility of collisions with other controllers, which are resolved in the same way). The danger is that the colliding messages may not be received the same everywhere (they will be traveling down the bus from different sources, at finite speed), and that some nodes will receive one or other of the messages, while others receive an invalid message. One proposed solution to this danger is for nodes sending cold start I-Frames always to act as if a collision had occurred. A related problem can arise because the cold start I-Frame is sent on only a single bus, where SOS or other faults may cause asymmetric reception. Here, as in the case of asymmetric collision detection, it is possible for some controllers to synchronize to a cold start I-Frame, while others do not—the latter may subsequently synchronize to a different initial I-Frame, resulting in two coexisting cliques. A proposed solution for this case is to send cold start I-Frames on both buses, and to deal with faulty transmission of cold start frames in the bus guardian of the hub.

Another problem can arise because there are no checks on the content of the cold start I-Frame. A faulty controller could provide bad data (e.g., an undefined mode number or MEDL position) and thereby cause good receivers to detect errors and shut down. Proposed solutions to this problem include more checking by the bus guardian of a central hub, or a more truly distributed start/restart algorithm.

Recent analysis has exposed the problems in TTA startup and restart described above. These can arise only in highly unusual circumstances, and are being addressed in the design of the new TTA-star configuration.

2.5.3 SPIDER

These aspects of SPIDER are not documented as yet.

2.5.4 FlexRay

As described below in Section 2.7.4, FlexRay differs from SAFEbus and TTA in that the full schedule for the system is not installed in each node during construction. Instead, each node is initialized only with its own schedule, and learns the full configuration of the system by observing message traffic during startup. This seems vulnerable to masquerading faults.

The method for initial synchronization of clocks in FlexRay is not described. It is difficult to initialize the Welch-Lynch algorithm if faults are present at startup: [Min93] describes scenarios that lead to independent cliques. It seems that TTA's clique-avoidance protocol will rescue it from these scenarios, but in the absence of such a mechanism, it is not clear how FlexRay can do so. There are clock synchronization algorithms that self-stabilize in the presence of faults (e.g., [DW95]), but these are complex, or rely on randomization. Randomization is generally considered unacceptable in a safety-critical system because correct operation is only probabilistically guaranteed, but it may be acceptable during startup (though recall the failure of the first attempt to launch the Space Shuttle [SG84]) or as part of a "never give up" strategy.

2.6 Services

The essential basic purpose of these architectures is to make it *possible* to build reliable distributed applications; a desirable purpose is to make it *straightforward* to build such applications. The basic services provided by the bus architectures considered here comprise clock synchronization, time-triggered activation, and reliable message delivery. Some of the architectures provide additional services; their purpose is to assist straightforward construction of reliable distributed applications by providing these services in an application-independent manner, thereby relieving the applications of the need to implement these capabilities themselves. Not only does this simplify the construction of application software, it is sometimes possible to provide *better* services when these are implemented at the architecture level, and it is also possible to provide strong assurance that they are implemented correctly.

Applications that perform safety-critical functions must generally be replicated for fault tolerance. There are many ways to organize fault-tolerant replicated computations, but a basic distinction is between those that use *exact* agreement, and those that use *approximate* agreement. Systems that use approximate agreement generally run several copies of the application in different nodes, each using its own sensors, with little coordination across the different nodes. The motivation for this is a "folk belief" that it promotes fault tolerance: coordination is believed to introduce the potential for common mode failures. Because different sensors cannot be expected to deliver exactly the same readings, the outputs (i.e., actuator commands) computed in the different nodes will also differ. Thus, the only way to detect faulty outputs is by looking for values that differ by "a lot" from the others. Hence, these systems use some form of selection or threshold voting to select a good value to send to the actuators, and similar techniques to identify faulty nodes that should be excluded. Brilliant, Knight and Leveson describe some of the difficulties with this approach in the context of N-version programming [BKL89]. The most troublesome of these for applications of the kind considered here is that hosts accumulate state that diverges from that of others over time (e.g., velocity and position as a result of integrating acceleration), and they execute mode switches that are discrete decisions based on local sensor values (e.g., change the gain schedule in the control laws if the altitude, or temperature, is above a specific value). Thus, small differences in sensor readings can lead to major differences in outputs and this can mislead the approximate selection or voting mechanisms into choosing a faulty value, or excluding a nonfaulty node. The fix to these problems is to attempt to coordinate discrete mode switches and periodically to bring state data into convergence. But these fixes are highly application specific, and they are contrary to the original philosophy that motivated the choice of approximate agreement—hence, there is a good chance of doing them wrong. There are numerous examples that justify this concern; several that were discovered in flight tests are documented by Mackall and colleagues [IRM84, Mac85, Mac88] and summarized in [Rus93b, Section 3.3]. The essential points of Mackall's data is that all the failures observed in flight test were due to bugs in the design of the fault tolerance mechanisms themselves, and all these bugs could be traced to difficulties in organizing and coordinating systems based on approximate agreement.

Systems based on exact agreement face up to the fact that coordination among replicated computations is necessary, and they take the necessary steps to do it right. If we are to use exact agreement, then every replica must perform the same computation on the same data: any disagreement on the outputs then indicates a fault; comparison can be used to detect those faults, and majority voting to mask them. A vital element in this approach to fault tolerance is that replicated components must work on the same data: thus, if one node reads a sensor, it must distribute that reading to all the redundant copies of the application running in other nodes. Now a fault in that distribution mechanism could result in one node getting one value and another a different one (or no value at all). This would abrogate the requirement that all replicas obtain identical inputs, so we need to employ mechanisms to overcome this behavior.

Interactive consistency

The problem of distributing data consistently in the presence of faults is variously called *interactive consistency*, *consensus*, *atomic broadcast*, or *Byzantine agreement* [PSL80, LSP82]. When a node transmits a message to several receivers, interactive consistency requires the following two properties to hold.

Agreement: All nonfaulty receivers obtain the same message (even if the transmitting node is faulty).

Validity: If the transmitter is nonfaulty, then nonfaulty receivers obtain the message actually sent.

Algorithms for achieving these requirements in the presence of arbitrary faults necessarily involve more than a single data exchange (basically, each receiver must compare the value it received against those received by others). It is provably impossible to achieve interactive consistency in the presence of a arbitrary faults unless there are at least $3a + 1$ FCUs, $2a + 1$ disjoint communication paths (or $a + 1$ disjoint broadcast channels) between them, and $a + 1$ levels (or "rounds") of communication. Some of the parameters, but not the number of rounds required, can be reduced by using digital signatures.

The problem might seem moot in architectures that employ a physical bus, since a bus surely cannot deliver values inconsistently (so the agreement property is achieved trivially). Unfortunately, it can—though it is likely to be a very rare event. The scenarios involving SOS faults presented earlier exemplify some possibilities.

Dealing properly with very rare events is one of the attributes that distinguishes a design that is fit for safety-critical systems from one that is not. It follows that either the application software must perform interactive consistency for itself (incurring the cost of n^2 messages to establish consistency across n nodes in the presence of a single arbitrary fault), or the bus architecture must do it, or the bus architecture must eliminate the fault modes that necessitate multiple rounds of information exchange (so that consistency is achieved by a simple broadcast).

Signal reshaping

The first choice is so unattractive that it vitiates the whole purpose of a fault-tolerant bus architecture, and the second is described below in separate subsections for those architectures that provide it. The third choice hinges on elimination of asymmetric transmissions (i.e., those that appear as one value to some receivers, and as different values, or the absence of values, to others). As noted, SOS faults are among the most plausible sources of asymmetric transmissions. SOS faults that cause asymmetric transmissions can arise in either the value or time domains (e.g., intermediate voltages, or weak edges, respectively). In those architectures that employ a bus guardian "in series" with an interface, the bus guardian is a possible point of intervention for the control of SOS faults: a suitable guardian can reshape, in both value and time domains, the signal sent to it by the controller. Of course, the guardian could be faulty and may make matters worse—so this approach makes sense only when there are independent guardians on each of two (or more) replicated interconnects. Observe that for credible signal reshaping, the guardian must have a power supply that is independent of that of the controller (faults in power supply are the most likely cause of intermediate voltages and weak edges).

Voting

Interactively consistent message broadcast provides the foundation for fault tolerance based on exact agreement. There are several ways to use this foundation. One arrangement, confusingly called the *state machine* approach [Sch90], is based on majority voting: application replicas run on a number of different nodes, exchange their output values, and deliver a majority vote to the actuators.[1] This approach was first developed by the SIFT project at SRI [WLG^{+}78]. Usually, selected intermediate state values are voted as well as outputs (this promotes recovery from transients [Rus93a]), and the architecture can assist these activities by providing services that make the distribution and selection of voted values transparent to the application programs.

Master/shadow

Another arrangement is based on self-checking (either by individuals or pairs) so that faults result in fail-silence. This will be detected by other nodes, and some backup application running in those other nodes can take over. The architecture can assist this master/shadow arrangement by providing services that support the rollover from one node to another. One such service automatically substitutes a backup node for a failed master (both the master and the backup occupy the same slot in the schedule, but the backup is inhibited from transmitting unless the master has failed). A variant has both master and backup operating in different slots, but the backup inhibits itself unless it is informed that the master has failed. A further variation, called *compensation*, applies when different nodes have access to different actuators: none is a direct backup to any other, but each changes its operation when informed that others have failed (an example is car braking: separate nodes controlling the braking force at each wheel will redistribute the force when informed that one of their number has failed).

Compensation

Group membership

The variations on master/shadow described above all depend on a "failure notification," or equivalently a "membership" service. The crucial requirement on such a service is that it must produce *consistent* knowledge: that is, if one nonfaulty node thinks that a particular node has failed, then all other nonfaulty nodes must hold the same opinion—otherwise, the system will lose coordination, with potentially catastrophic results (e.g., if the nodes controlling braking at different wheels make different adjustments to their braking force based on different assessments of which others have failed). Notice that this must also apply to a node's knowledge of its *own* status: a naïve view might assume that a node that is receiving messages and seeing no problems in its own operation should assume it is in the membership. But if this node is unable to transmit, all other nodes will have removed it from their memberships and will be making suitable compensation on the assumption

[1] There is often another round of voting performed directly by the actuators, through some form of physical "force-summing." For example, outputs of different nodes may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft.

that this node has entered its “blackout” mode (and is, for example, applying no force to its brake). It could be catastrophic if this node does not adopt the consensus view and continues operation (e.g., applying force to its brake) based on its local assessment of its own health.

A membership service operates as follows. Each node maintains a private *membership* list, which is intended to comprise all and only the nonfaulty nodes. Since it can take a while to diagnose a faulty node, we have to allow the common membership to contain at most one faulty node. Thus, a membership service must satisfy the following two requirements.

Agreement: The membership lists of all nonfaulty nodes are the same.

Validity: The membership lists of all nonfaulty nodes contain all nonfaulty nodes and at most one faulty node.

These requirements can be achieved only under benign fault hypotheses (it is provably impossible to diagnose an arbitrary-faulty node with certainty). When unable to maintain accurate membership, the best recourse is to maintain agreement, but sacrifice validity (nonfaulty nodes that are not in the membership can then attempt to rejoin). This weakened requirement is called “clique avoidance.” Note that it is quite simple to achieve consistent membership on top of an interactively consistent message service: each node broadcasts its own membership list to every other node, and each node runs a deterministic resolution algorithm on the (identical, by interactive consistency) lists received. It is much more difficult to achieve consistent membership in the absence of an interactively consistent message service (and will require multiple rounds of message exchange).

2.6.1 SAFEbus

SAFEbus provides two important services: interactively consistent message transmission, and automatic master/shadow rollover.

Messages from one host to another traverse two BIUs on their way to the four separate buses, and two more BIUs on their way to a receiving host. Although not required by the ARINC 659 standard, Honeywell’s implementation of SAFEbus has an additional path for cross-comparison of messages between the receiving BIUs. This additional cross-comparison is crucial to SAFEbus’s ability to provide interactive consistency.

SAFEbus allows as many as four different nodes to occupy a single slot managed as a master and as many as three shadows. If the master fails to send an expected message, then its first shadow will take over the slot within a few bit times, and so on down to the third shadow. A faulty BIU in a shadow cannot usurp its master’s slot inappropriately because it will be inhibited by the guardian function of its partner BIU. The interactively consistent message transmission provided by SAFEbus ensures that the master and shadows will all have seen an identical history of messages and can therefore provide seamless transfer of function (masters and shadows may not have the same state, since some shadows may be specified to provide degraded functionality). SAFEbus provides a special class of messages that allow masters to communicate specifically with their shadows.

In Honeywell's implementation, each node has a pair of hosts (CPMs) that cross-compare and fail silent on disagreement; the BIUs do the same (in any implementation), so the whole node is fail-silent. In this context, master/shadow rollover is an effective and straightforward way to provide high availability and is preferred to other fault-masking methods such as majority voting.

SAFEbus does not employ membership, but nodes read their own transmissions on the bus just as other receivers do and are therefore quickly able to detect if they have suffered a transmission fault, and can make suitable compensation, if necessary. Most SAFEbus applications rely on self-checking, fail-silence, and master/shadow rollover to provide fault tolerance, and do not require a membership service. It is, however, quite straightforward to implement such a service, if required, at the application level—given the underlying support for interactively consistent message transmission.

2.6.2 TTA

Interactive consistency requires that any transmission results in identical reception at all receivers. Unlike SAFEbus, TTA does not have enough independent signal paths and rounds of voting to achieve this property directly, but it achieves it indirectly in a very clever way.

TTA uses high-grade checksums that can be considered equivalent to digital signatures. This eliminates the possibility that different recipients obtain *different* values from a single broadcast, leaving only a residual asymmetry between those that do receive a value and those that do not. This "weak consistency" is preferable to asymmetric reception, but is inadequate for the construction of consistent replicas and backups. Simple acknowledgments are insufficient to fix the difficulty, because they may be lost or received asymmetrically also. TTA, however, provides a property called "clique avoidance" as part of its membership service, and this is equivalent to a consistent acknowledgment: only those nodes that receive a message, or those that did not (whichever is in the majority) will remain in the membership [BP00, KBP01]. In common cases (where just one newly faulty node fails to receive, or a newly faulty sender fails to transmit), clique avoidance is achieved by the standard membership function and excludes exactly the faulty node. In rare cases, where there are multiple faults, or an asymmetric fault, the behavior of the clique-avoidance protocol is sound (the survivors have consistent state), but may be Draconian, in that nonfaulty processors may be removed from the membership (though they may rejoin at the next round).

The combination of checksummed transmissions and clique avoidance provides a form of interactive consistency related to "Crusader Agreement" [Dol82] and to "Weak Byzantine Agreement" [Lam83]. Crusader agreement is similar to Byzantine agreement (i.e., interactive consistency) except that when the transmitter is faulty, it is acceptable for some receivers not to agree with others on the value transmitted, provided they "explicitly know" that the transmitter is faulty. In the "Draconian" agreement achieved by TTA, this "explicit knowledge" is achieved through the clique-avoidance protocol and is associated with exclusion from the group. In weak Byzantine agreement, receivers may agree on *any* value whenever there is a fault (not just a fault in the transmitter). In the agreement achieved by TTA, the surviving clique may agree in (incorrectly) ascribing “no value received” to a nonfaulty transmitter (and hence excluding it from the membership) when there are multiple faults within two rounds.

The clique-avoidance component of TTA’s group membership protocol (and hence the Draconian form of agreement) engages only when there are multiple faults within two rounds, or an asymmetric transmission; in all other circumstances, TTA’s combination of checksummed transmissions and group membership provides interactive consistency of the classical form. The Draconian behavior is acceptable (and is the price paid for using fewer communication paths and messages than required for full interactive consistency) but undesirable, and its occurrences should be minimized. We cannot do much about multiple fault arrivals in a short interval, but asymmetric transmissions are most plausibly the consequences of SOS faults, and so it is these that we should seek to minimize. As noted earlier, it is difficult to control these faults with bus guardians that are integrated with the controllers, so this is a reason for preferring the TTA-star architecture to TTA-bus.

The TTA membership and clique-avoidance service not only supports a form of interactive consistency, but the membership information can be used to organize various master/shadow or compensation strategies for fault tolerance at the application level. TTA provides explicit support for shadow nodes, which occupy the same slots in the schedule as their master and can read all bus traffic, but cannot transmit until the master has failed. The membership service is also exploited internally by TTA, to allow it to operate in the presence of multiple faulty clocks (synchronization is performed only over nodes that are in the membership).

A proposed extension to TTA is a service that supports state machine replication in a transparent way [KB00]. The idea is to identify some of the state variables of an application as ones that should be voted. Exchange and voting of those variables is then managed by the TTA controllers in a way that is transparent to the application. This is accomplished by locating the voted variables in that area of memory that the host shares with its controller (hosts and controllers interface through dual-ported RAM). The application reads and writes those variables in the usual way; behind the scenes, multiple instances of the application will be running on different hosts; their controllers broadcast the values of voted variables to each other (exploiting the interactively consistent broadcasts provided by TTA), and replace their local copies by majority-voted versions. The attraction of this service is that it is truly transparent to the application: neither its function nor its timing is changed by the decision to make it fault tolerant using state machine replication.

2.6.3 SPIDER

As described in Section 2.3.3, the arrangement of BIUs and RMUs in the hub of SPIDER’s ROBUS is similar to that of hosts and interstages in the Draper FTP. The motivation for the architecture of FTP (and possibly of SPIDER) was the desire to achieve interactive consistency with only three full processors—since this is all that is required to tolerate (using TMR) the failure of a host processor running the actual application. The problem, of course, is that interactive consistency requires at least four participants to tolerate a single arbitrary fault. The architecture adopted in FTP overcomes this limitation by adding three impoverished processors (these are the "interstages") that act rather like mirrors. The processors and interstages comprise six FCUs, so interactive consistency is feasible in theory—and it is achieved in practice by a very clever algorithm whose correctness has been formally verified [LR94]. The interactive consistency algorithm of SPIDER is similar to that of FTP (with RMUs taking the part of the interstages). The algorithm operates as follows.

- A host sends its value to its BIU.
- The BIU broadcasts this value to all RMUs.
- RMUs broadcast the value received to all BIUs.
- Each BIU performs a hybrid-majority vote on the values received and forwards the winner to its host. (A hybrid-majority vote is one from which manifestly bad values are excluded.)

This differs from the FTP algorithm in that the broadcast to all RMUs in the second step replaces (what would be in SPIDER terminology) a BIU-to-BIU broadcast and a BIU-to-own-RMU transfer.

2.6.4 FlexRay

In contrast to the other architectures considered, FlexRay provides no services beyond clock synchronization and reliable (best efforts) message delivery. In particular, FlexRay does not provide interactively consistent message transmission (nor even weakly consistent), provides no membership or failure notification service, and contains no mechanisms to control SOS faults.

2.7 Flexibility

The static schedule of a time-triggered bus is rather inflexible, so some bus architectures make provision for switching between different schedules at startup or during operation. The different schedules may be optimized for different missions, or phases of a mission (e.g., startup vs. cruise), for operating in a degraded mode (e.g., when some major function has failed), or to accommodate optional equipment (e.g., for cars with and without traction control). It is necessary to protect against inappropriate schedule switches (or switches initiated by faulty nodes), so some kind of voting is usually employed.

The physical wires or optical cables routed around an aircraft or car represent significant costs (e.g., in material, installation, maintenance, and weight) and there is strong interest in minimizing the number that are used. Some of the purposes typically performed by an event-driven bus such as CAN can be taken over by a time-triggered bus in a straightforward way; for other purposes, however, the flexible resource allocation of an event-driven bus is considered a necessity, so some way must be found to provide this capability within a time-triggered bus if this is to completely subsume existing event-triggered buses. The different buses approach the issue of flexibility in very different ways.

2.7.1 SAFEbus

A SAFEbus schedule (called a *table*) may be comprised of several *frames*; each frame is a self-contained description of the allocation of messages to time slots. Only one frame may be active at any one time. Slots allocated to "Long Resync" messages may be marked as allowing a frame change. In this case, the BIU that sends the Long Resync message in that slot indicates the new frame that is to be used. The old and new frames begin with different patterns of Short and Long Resync messages so that any receivers that enter the wrong frame (e.g., due to an asymmetric transmission fault) will fail to synchronize and drop off the bus (this is rather like the clique-avoidance protocol of TTA). Frames can exist in several versions: the version of the new frame is also indicated in the Long Resync message, and any node whose table memory contains a different version will drop off the bus.

Multiple frames provide some flexibility to choose among different modes of behavior, but it seems that the capability is used only at a very coarse level: for example, a table may have frames for hardware initialization, software initialization, self-test, and flight. The flight frame will contain no frame change commands: once entered, it cannot be left.

SAFEbus provides no mechanisms for event-triggered behavior, but its BIUs are connected to an IEEE 1149.5 bus for test and maintenance purposes.

2.7.2 TTA

TTA schedules are precomputed and loaded into a data structure called the Message Descriptor List (MEDL) present in each controller. A limited form of the same data is present in each bus guardian. There is considerable flexibility in selection of the number and length of messages each node may transmit in each cycle, but the selection is fixed once it is loaded into the MEDL. TTA checks that all nodes have the same MEDL version during startup.

The MEDL may allow certain nodes to request mode changes at certain points. A requested mode change may be either immediate or deferred; if the latter, nodes that transmit later in the cycle have the opportunity to override the mode change request, which occurs at the end of the cycle. All modes are based on the same schedule (so the bus guardians are not affected by mode changes): all that changes are the recipients and intended interpretation of the messages that are sent.

Space for diagnostic and other event data may be set aside in each message and used in some application-specific manner. This allows each node a fixed bandwidth for event data.

A variation is for each node to interpret this data as a simulation of the traffic on a shared event-triggered bus. It is proposed to use this approach to provide a simulation of CAN; later versions of TTA are so fast that it is calculated that the simulation can go faster than a real CAN bus while absorbing only a small fraction of the TTA bandwidth.

Observe that this approach brings all the safety attributes of TTA to the simulated event-triggered bus: if one node manifests faulty behavior on the simulated bus, the other nodes can remove it from their membership—this ability to detach a faulty node from the simulated bus is beyond the capability of any real event-triggered bus.

Although it appears that TTA can perform the rôle of a CAN bus in addition to its own, a full car or aircraft system is likely to need additional buses for secondary control functions (to say nothing of those for entertainment). For example, a car door contains motors and primitive controllers for the window, lock, and mirror; even a CAN bus provides excessive functionality and costs too much to connect each of these devices separately. Consequently, ultra-low-cost buses are emerging that can connect smart sensors and actuators to a gateway on a more muscular bus. These low-cost buses must operate with extremely primitive controllers that lack even a clock oscillator. TTA incorporates this kind of service through the TTP/A protocol [KHE00].

2.7.3 SPIDER

These elements of SPIDER are not yet developed.

2.7.4 FlexRay

FlexRay aims to be more flexible than the other buses considered here, and this seems to be reflected in the choice of its name.

FlexRay partitions each time cycle into a “static” time-triggered portion, and a “dynamic” event-triggered portion. The division between the two portions is set at design time and loaded into the controllers and bus guardians. Nodes communicate using the Byteflight protocol during the event-driven portion of the cycle. A similar consortium to FlexRay has developed the Local Interconnect Network (LIN) protocol [LIN00] and this is presumably used to provide a low-cost sensor bus in association with FlexRay (similar to TTP/A for TTA).

Unlike SAFEbus and TTA, FlexRay does not install the full schedule for the time-triggered portion in each controller. Instead, this portion of the cycle is divided into a number of slots of fixed size, and each controller and its bus guardians are informed of which slots are allocated to their transmissions. Nodes requiring greater bandwidth are assigned more slots than those that require less. Each controller learns the full schedule only when the bus starts up. Each node includes its identity in the messages that it sends; during startup, nodes use these identifiers to label their input buffers as the schedule reveals itself (e.g., if the messages that arrive in slots 1 and 7 carry identifier 3, then all nodes will thereafter deliver the contents of buffers 1 and 7 to the task that deals with input from node 3). There is an obvious vulnerability here: a faulty node could masquerade as another (i.e., send a message with the wrong identifier) during startup and thereby violate partitioning for the remainder of the mission. It is not clear how this fault mode is countered. Neither is it clear how configuration errors, in which two nodes are allocated to the same slot, are detected during startup. (Presumably the message received in that slot will be garbled by the collision and will fail checksum, but then what?)

2.8 Assurance

Safety-critical systems must be furnished with strong assurance that they are fit for their purpose. Regulation and certification establish requirements for assurance in some application areas (e.g., [RTC92,RTC00] for airborne software and hardware, respectively, and [MIS94] for cars). Assurance is generally achieved by a combination of testing the actual artifact, analysis and review of its design, and scrutiny of its design process. Since safety-critical systems, such as the bus architectures considered here, must be fault tolerant, some of the testing will involve fault injection. However, testing and fault injection can provide direct assurance only to failure rates of about 10^{-4} or 10^{-5} per hour, which are far short of those required for safety-critical applications. The remaining assurance must be derived by analysis of the system's design. Formal methods can assist in this process.

In formal methods, a mathematical model is constructed of key elements of the system's operation (e.g., its clock synchronization algorithm), and mechanized calculation is used to demonstrate that it meets its requirements, under its specified assumptions. The appropriate branch of applied mathematics for modeling discrete algorithms (whether they are destined for software or hardware implementation) is formal logic, and calculation is performed by the methods of automated deduction, such as theorem proving or model checking. One attribute that renders formal methods particularly attractive in this domain is that it allows *all* behaviors of fault-tolerant algorithms to be examined through logical case analysis; this is especially powerful when considering arbitrary fault modes, because unlike explicit testing or simulation, we do not have to particularize the notion of "arbitrary" but can leave it totally unconstrained.

2.8.1 SAFEbus

SAFEbus was approved by the FAA as part of the certification for the Boeing 777 (the FAA certifies only complete aircraft, not components), and is a flight-critical part of every 777, whose commercial deliveries began in May 1995. Details of the assurance processes used have not been published, but must have been extensive, and are now supported by substantial field experience, with no failures recorded.

2.8.2 TTA

TTA implementations have been subjected to extensive fault-injection experiments in the context of the FIT project of the European Union, and evaluated in full-scale experimental applications developed by DaimlerChrysler and several other automobile companies and their suppliers. Aircraft engine controllers and cockpit automation systems under development by Honeywell and cabin pressurization systems under development by Hamilton Sundstrand/Nord Micro will be certified under FAA and JAA requirements.[2]

The basic Welch-Lynch clock synchronization protocol employed in TTA has been formally verified by Miner [Min93] and by Schwier and von Henke [SvH98]. The actual TTA protocol has been formally verified by Pfeifer, Schwier, and von Henke [PSvH99]. A new verification is planned (by me) that will extend the analysis beyond the standard fault hypothesis of TTA using a hybrid fault model developed by Schmid [Sch00]. The membership and clique-avoidance protocol of TTA has been formally verified by Pfeifer [Pfe00], but only under the standard fault hypothesis of TTA. Formal verification of its properties in the presence of asymmetric transmissions, and fault numbers and arrival rates beyond those of the fault hypothesis, is in progress. Some of these properties have already been verified by traditional (i.e., not mechanically checked) mathematical proofs [BP00, Mer01] and recent work by Bouajjani and Merceron formally verifies aspects of the clique-avoidance algorithm [BM02]. Among the simpler properties of TTA, the timing rules for controllers and bus guardians have been formally verified [Rus01], and mutual exclusion on the bus has been examined by model checking [MMP99]. The main remaining challenges are formally to verify the properties of startup, restart, and reintegration, and to compose the many separate analyses into a single verification of the integrated TTP/C protocol. A survey of many of these formal verifications for aspects of TTA is available [Rus02].

2.8.3 SPIDER

The interactive consistency algorithm of SPIDER has been formally verified [MMTP02] (it is similar to that previously performed for the Draper FTP architecture [LR94]). Its diagnosis algorithm also has been formally verified [GM02]; it is similar to the algorithms developed for MAFT [KWFT88] whose verification is described by Walter, Lincoln, and Suri [WLS97]. Formal verification of the SPIDER diagnosis algorithm led to insights that have produced an improved algorithm, which has also been formally verified [GM03]. Formal verification of the SPIDER clock synchronization algorithm is in progress.

One of the main goals of the SPIDER project is to serve as a demonstration study for certification under the DO-254 guidelines for airborne hardware [RTC00].

[2] Although the current applications for Honeywell's TTA-based engine controllers are military, their assurance processes closely follow those required by the FAA.

2.8.4 FlexRay

FlexRay is still under development. The only assurance technique so far described is a simulation model being developed by Motorola. As noted above, the basic Welch-Lynch clock synchronization algorithm has been formally verified. However, FlexRay documents speak of the system being operational as soon as two clocks synchronize. This is outside the parameters of the formal analyses, which would need to be revisited and extended to cover the cases $n = 2$ and $n = 3$. Furthermore, there are known pathologies in initialization of the Welch-Lynch algorithm in the presence of faults that can lead to clique formation [Min93]. It is not described how FlexRay avoids these, and verification of its startup mechanisms could be very challenging. The other protocols of FlexRay are not described in sufficient detail to assess the feasibility of their formal verification.

Chapter 3

Conclusion

The four buses considered here provide different solutions to very similar sets of requirements. All provide fault-tolerant, distributed clock synchronization, and support the time-triggered model of computation. They differ in their fault hypotheses, mechanisms, services, assurance, performance, and cost.

SAFEbus is the most mature of the four, and makes the fewest compromises. It employs paired bus interface units, with each member of a pair acting as a bus guardian for the other, and paired, self-checking buses. Its fault hypothesis includes arbitrary faults, faults in several nodes (but only one per node), and a high rate of fault arrivals—and it never gives up. It tolerates spatial proximity faults by duplicating the entire system. It provides interactively consistent message broadcasts (in the Honeywell implementation), and supports application-level fault tolerance (based on self-checking pairs) by providing automatic rapid rollover from masters to shadows. It is certified for use in passenger aircraft and has extensive field experience as the backbone for the integrated avionics on the Boeing 777. The Honeywell implementation is supported by an in-house tool chain. The raw bus operates at 30 MHz, is two bits wide, and achieves high utilization. Because each of its major components is paired (and its bus requires separate lines for clock and data), it is the most expensive of those available for commercial use (typically, a few hundred dollars per node).

TTA is the next-most mature. In its TTA-bus configuration, it is vulnerable to spatial proximity faults, and its bus guardians are not fully independent of its interface controllers, so the TTA-star configuration is generally to be preferred. Its fault hypothesis includes arbitrary faults, and faults in several nodes (but only one per node), provided these arrive at least two rounds apart. It never gives up and has a well-defined recovery strategy from fault arrivals that exceed this hypothesis. It provides a form of interactively consistent message broadcasts and a consistent membership service. Proposed extensions provide state machine replication in a manner that is transparent to applications. Other proposed extensions provide a fully protected event-based service within the time-triggered framework. Its prototype implementations have been subjected to extensive testing and fault injections, and deployed in experimental vehicles. Several of its algorithms have been formally verified, and aircraft applications under development are planned to lead to FAA certification. It is supported by an extensive tool suite that interfaces to standard CAD environments (Matlab/Simulink). Current implementations provide 25 Mbit/s data rates; research projects are designing implementations for gigabit rates. TTA controllers and the star coupler (which is basically a modified controller) are quite simple and cheap to produce in volume.

SPIDER is a research project and it is unfair to compare it directly with the commercial products. Its fault hypothesis uses a hybrid fault model, which includes arbitrary faults, and allows some combinations of multiple faults. It provides interactively consistent message broadcasts. Its algorithms are novel and highly efficient and are being formally verified. It is planned to be used as a demonstration study for certification under the DO-254 guidelines for airborne hardware. SPIDER is interesting because it uses very strong algorithms and can use different topologies from the other buses.

FlexRay is still under development. It has no stated fault hypothesis, and appears to have no mechanisms to counter certain fault modes (e.g., SOS faults or other sources of asymmetric broadcasts, and masquerading on startup). Its bus guardians are not fully independent of their controllers. Its clock synchronization can tolerate faults in no more than a third of its nodes, and its initialization in the presence of faults is not described. A never-give-up strategy is not described. It provides no services to its applications beyond best-efforts message delivery. Event-based services share the same bus; bus guardians protect only the time-triggered section of the bus cycle. No systematic or formal approaches to assurance or certification are described. It is the slowest of the commercial buses, with a claimed data rate of no more than 10 Mbit/s. It is asserted to be cheap to produce in volume, but this is questionable as each node requires three clock oscillators (one for the controller and one for each of the bus guardians). Some of the deficiencies of FlexRay may be overcome as its development proceeds, and more recent descriptions of its design may correct or contradict some of the assertions above (which were written prior to September 2001).[1] However, the decision to provide no services to support fault-tolerant applications seems a deliberate and irreversible design choice. This means that all mechanisms for fault-tolerant applications must be provided by the application programs themselves. Thus, application programmers, who may have little experience in the subtleties of fault-tolerant systems, become responsible for the design, implementation, and assurance of very delicate mechanisms with no support from the underlying bus architecture. Not only does this increase the cost and difficulty of making sure that things are done right, it also increases their computational cost and latency. For example, in the absence of an interactively consistent message service provided by the architecture, application programs must explicitly transmit the multiple rounds of cross-comparisons that are needed to implement this service at a higher level, thereby substantially increasing the message load. Such a cost will invite inexperienced developers to seek less expensive ways to achieve fault tolerance—in

[1] For another comparison, written at about the same time, see [Kop01].

probable ignorance of the impossibility results in the theoretical literature, and the history of intractable “Heisenbugs” (rare, unrepeatable, failures) encountered by practitioners who pushed for 10^{-9} with inadequate foundations.

A safety-critical bus architecture provides certain properties and services that assist in construction of safety-critical systems. As with any system framework or middleware package, these buses offer a tradeoff to system developers: they provide a coherent collection of services, with strong properties and highly assured implementations, but developers must sacrifice some design freedom to gain the full benefit of these services. For example, all these buses use a time-triggered model of computation, and system developers must build their applications within that framework. In return, the buses are able to guarantee strong partitioning: faults in individual components or applications (“functions” in avionics terms) cannot propagate to others, nor can they bring down the entire bus (within the constraints of the fault hypothesis). Partitioning is the minimum requirement, however. It ensures that one failed function will not drag down others, but in most safety-critical systems the failure of even a single function can be catastrophic, so the individual functions must themselves be made fault tolerant. Accordingly, most of the buses provide mechanisms to assist the development of fault-tolerant applications. The key requirement here is interactively consistent message transfer: this ensures that all masters and shadows (or masters and monitors), or all members of a voting pool, maintain consistent state. Three of the buses considered here provide this basic service; some of them do so in association with other services, such as master/shadow rollover or group membership, that can be provided with much reduced latency when implemented at a low level. FlexRay, alone, provides none of these services.

It is unlikely that any single bus architecture will satisfy all needs and markets, and it is to be expected that new or modified designs will emerge to satisfy new requirements. I hope that the comparison provided here will help potential users to select the existing bus best suited to their needs, and that it will help designers of new buses to learn from and build on the design choices made by their predecessors.

Bibliography

[AG93] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993.

[ARI93] Aeronautical Radio, Inc., Annapolis, MD. *ARINC Specification 659: Backplane Data Bus*, December 1993. Prepared by the Airlines Electronic Engineering Committee.

[ARI56] Aeronautical Radio, Inc., Annapolis, MD. *ARINC Specification 629: Multi-Transmitter Data Bus; Part 1, Technical Description (with five supplements); Part 2, Application Guide (with one supplement)*, December 1995/6. Prepared by the Airlines Electronic Engineering Committee.

[B$^+$01] Joef Berwanger et al. FlexRay–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Detroit, MI, April 2001. Society of Automotive Engineers. Paper number 2001-01-0676.

[BKL89] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. The consistent comparison problem in N-Version software. *IEEE Transactions on Software Engineering*, 15(11):1481–1485, November 1989.

[BM02] Ahmed Bouajjani and Agathe Merceron. Parametric verification of a group membership algorithm. In Damm and Olderog [DO02], pages 311–330.

[BMF02] Florian Bogenberger, Bernd Müller, and Thomas Führer. Protocol overview. In *FlexRay International Workshop: The Communication System for Advanced Automotive Control Applications*, Munich, Germany, April 2002. Available from `http://www.flexray.com/htm/infws402.htm`.

[BP00] Günther Bauer and Michael Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *19th Symposium on Reliable Distributed Systems*, Nuremberg, Germany, October 2000.

[Bra00] David Bradbury. *Simulation of a Time Triggered Protocol*. Honours Thesis, Basser Department of Computer Science, Sydney University, Australia, 2000. Available from `http://www.cs.usyd.edu.au/˜agathe/pub/Thesis091100.pdf`.

[But92] Ricky W. Butler. The SURE approach to reliability analysis. *IEEE Transactions on Reliability*, 41(2):210–218, June 1992.

[Byt] *Byteflight Specification*. Available at `http://www.byteflight.com`.

[Deu95] Deutsche Industrie Norm, Berlin, Germany. *Profibus Standard: DIN 19245*, 1995. Two volumes; see also `http://www.profibus.com`.

[DHS86] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, April 1986.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[DO02] Werner Damm and Ernst-Rüdiger Olderog, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, Oldenburg, Germany, September 2002. Springer-Verlag.

[Dol82] Danny Dolev. The Byzantine Generals strike again. *Journal of Algorithms*, 3(1):14–30, March 1982.

[DW78] Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978.

[DW95] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization with Byzantine faults. In *Fourteenth ACM Symposium on Principles of Distributed Computing*, page 256, Ottawa, Ontario, Canada, August 1995. Association for Computing Machinery.

[Ech99] Echelon Corporation, Palo Alto, CA. *Introduction to the LonWorks System*, 1999. Available at `http://osa.echelon.com/Program/LonWorksIntroPDF.htm`.

[FAA88] Federal Aviation Administration. *System Design and Analysis*, June 21, 1988. Advisory Circular 25.1309-1A.

[FLM86] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, September 1995. IEEE Computer Society.

[GM02] Alfons Geser and Paul S. Miner. A formal correctness proof of the SPIDER Diagnosis Protocol. In *Theorem Proving in Higher Order Logics*, Track B Proceedings, pages 71–86, August 2002. NASA Conference Publication NASA/CP-2002-211736.

[GM03] Alfons Geser and Paul Miner. A new on-line diagnosis protocol for the SPIDER family of Byzantine fault tolerant architectures. To appear as a NASA Technical Memorandum, 2003.

[GP93] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance. In *Twelfth ACM Symposium on Principles of Distributed Computing*, pages 195–206, Ithaca, NY, August 1993. Association for Computing Machinery.

[HD92] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 68–73, Seattle, WA, October 1992.

[HD93] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):34–39, March 1993.

[HDHR91] Kenneth Hoyme, Kevin Driscoll, Jack Herrlin, and Kathie Radke. ARINC 629 and SAFEbus™: Data buses for commercial aircraft. *Scientific Honeyweller*, pages 57–70, Fall 1991.

[Hop88] Harry Hopkins. Fit and forget fly-by-wire. *Flight International*, pages 89–92, December 3, 1988.

[HT98] Günter Heiner and Thomas Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Fault Tolerant Computing Symposium 28*, pages 402–407, Munich, Germany, June 1998. IEEE Computer Society.

[IRM84] Stephen D. Ishmael, Victoria A. Regenie, and Dale A. Mackall. Design implications from AFTI/F16 flight test. NASA Technical Memorandum 86026, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1984.

[ISO93] International Standards Organization, Switzerland. *ISO Standard 11898: Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*, November 1993.

[KB00] Hermann Kopetz and Günther Bauer. Transparent redundancy in the time-triggered architecture. In *The International Conference on Dependable Systems and Networks*, pages 5–13, New York, NY, June 2000. IEEE Computer Society.

[KBP01] Hermann Kopetz, Günther Bauer, and Stefan Poledna. Tolerating arbitrary node failures in the Time-Triggered Architecture. In *SAE 2001 World Congress*, Detroit, MI, March 2001. Society of Automotive Engineers. SAE paper number 2001-01-0677.

[KG93] H. Kopetz and G. Grünsteidl. TTP—a time-triggered protocol for fault-tolerant real-time systems. In *Fault Tolerant Computing Symposium 23*, pages 524–533, Toulouse, France, June 1993. IEEE Computer Society.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[KHE00] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. In *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport Beach, CA, March 2000. IEEE Computer Society.

[Kop01] Hermann Kopetz. A comparison of TTP/C and FlexRay. Available from `http://www.tttech.com/technology/docs/protocol_comparisons/HK_2001-05-C%omparison_FlexRay-TTP-C.pdf`, 2001.

[KWFT88] 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.

[Lal86] Jaynarayan H. Lala. A Byzantine resilient fault tolerant computer for nuclear power application. In *Fault Tolerant Computing Symposium 16*, pages 338–343, Vienna, Austria, July 1986. IEEE Computer Society.

[Lam83] Leslie Lamport. The weak Byzantine Generals problem. *Journal of the ACM*, 30(3):668–676, July 1983.

[LIN00] Local interconnect network (LIN). See `http://www.lin-subbus.org/`, 2000.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LR94] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT82] E. Lloyd and W. Tye. *Systematic Safety: Safety Assessment of Aircraft Systems*. Civil Aviation Authority, London, England, 1982. Reprinted 1992.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996.

[Mac85] Dale A. Mackall. Qualification needs for advanced integrated aircraft. NASA Technical Memorandum 86731, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1985.

[Mac88] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[MBSP02] Reinhard Maier, Günther Bauer, Georg Stöger, and Stefan Poledna. Time-triggered architecture: A consistent computing platform. *IEEE Micro*, 22(4):36–45, July/August 2002.

[Mer01] Agathe Merceron. Proving "no cliques" in a protocol. In *24th Australasian Computer Science Conference*, Gold Coast, Queensland, Australia, January/February 2001. IEEE Computer Society. Available from `http://www.cs.usyd.edu.au/~agathe/pub/goldCoastR.pdf`.

[Min93] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, November 1993.

[MIS94] The Motor Industry Software Reliability Association, Nuneaton, UK. *Development Guidelines for Vehicle Based Software*, PDF version 1.1, January 2001 edition, November 1994. Available at `http://www.misra.org.uk`.

[MMP99] Agathe Merceron, Monika Muellerburg, and G. Michele Pinna. “No collisions” in a protocol with n stations: A comparative study of formal proofs. In *5th International Workshop on Formal Methods for Industrial Critical Systems, Part of the Federated Logic Conference*, Trento, Italy, July 1999.

[MMTP02] Paul S. Miner, Mahyar Malekpour, and Wilfredo Torres-Pomales. Conceptual design of a Reliable Optical BUS (ROBUS). In *21st AIAA/IEEE Digital Avionics Systems Conference*, Irvine, CA, October 2002.

[Pal96] Daniel L. Palumbo. Fault-tolerant processing system. United States Patent 5,533,188, July 2, 1996.

[Pfe00] Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000. Kluwer Academic Publishers.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, editors, *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, January 1999. IEEE Computer Society.

[RTC92] Requirements and Technical Concepts for Aviation, Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992. This document is known as EUROCAE ED-12B in Europe.

[RTC00] Requirements and Technical Concepts for Aviation, Washington, DC. *DO254: Design Assurance Guidelines for Airborne Electronic Hardware*, April 2000.

[Rus93a] 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.

[Rus93b] John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also available as NASA Contractor Report 4551, December 1993.

[Rus94] 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. Also available as NASA Contractor Report 198289.

[Rus95] John Rushby. Formal methods and their role in the certification of critical systems. In Roger Shaw, editor, *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, pages 1–42, Bruges, Belgium, September 1995. Springer. Also issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification).

[Rus96] John Rushby. Reconfiguration and transient recovery in state-machine architectures. In *Fault Tolerant Computing Symposium 26*, pages 6–15, Sendai, Japan, June 1996. IEEE Computer Society.

[Rus01] John Rushby. Formal verification of transmission window timing for the time-triggered architecture. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.

[Rus02] John Rushby. An overview of formal verification for the time-triggered architecture. In Damm and Olderog [DO02], pages 83–105.

[Sch87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch93] Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[Sch00] Ulrich Schmid. How to model link failures: A perception-based fault model. Technical Report 183/1-108, Technical University of Vienna, Department of Automation, October 2000. (To be presented at DSN 2001.).

[SD95] William Sweet and Dave Dooling. Boeing's seventh wonder. *IEEE Spectrum*, 32(10):20–23, October 1995.

[SG84] Alfred Spector and David Gifford. Case study: The space shuttle primary computer system. *Communications of the ACM*, 27(9):872–900, September 1984.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[SvH98] D. Schwier and F. von Henke. Mechanical verification of clock synchronization algorithms. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 262–271, Lyngby, Denmark, September 1998. Springer-Verlag.

[TP88] 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.

[TTT02] Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria. *Time-Triggered Protocol TTP/C: High-Level Specification Document*, July 2002. Specification edition 1.0.0, document number D-032-S-10-028.

[WL88] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

[WLG$^+$78] 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.

[WLS97] Chris J. Walter, Patrick Lincoln, and Neeraj Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-03-2003	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE A Comparison of Bus Architectures for Safety-Critical Embedded Systems	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) John Rushby	**5d. PROJECT NUMBER** NCC1-377
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 728-30-10-01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Ave Menlo Park, California 94025	**8. PERFORMING ORGANIZATION REPORT NUMBER** Project 11003, Deliverable 25b
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR’S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR’S REPORT NUMBER(S) NASA/CR–2003–212161

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390 Distribution: Standard

13. SUPPLEMENTARY NOTES

An electronic version can be found at http://techreports.larc.nasa.gov/ltrs/ or http://techreports.larc.nasa.gov/cgi-bin/NTRS.
Langley Technical Monitor: Paul S. Miner. Report prepared by SRI International under NASA Cooperative agreement NCC1-377 with Honeywell International

14. ABSTRACT

We describe and compare the architectures of four fault-tolerant, safety-critical buses with a view to deducing principles common to all of them, the main differences in their design choices, and the tradeoffs made. Two of the buses come from an avionics heritage, and two from automobiles, though all four strive for similar levels of reliability and assurance.
The avionics buses considered are the Honeywell SAFEbus (the backplane data bus used in the Boeing 777 Airplane Information Management System) and the NASA SPIDER (an architecture being developed as a demonstrator for certification under the new DO-254 guidelines); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA), recently adopted by Audi for automobile applications, and by Honeywell for avionics and aircraft control functions, and FlexRay, which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips.

15. SUBJECT TERMS

Certification, Integrated Modular Avionics, fault tolerance

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: `help@sti.nasa.gov`)
Unclassified	Unclassified	Unclassified	Unlimited	63	**19b. TELEPHONE NUMBER** ***(Include area code)*** (301) 621-0390