Presented at EMSOFT 2001: First Workshop on Embedded Software, 8–10 October 2001, Lake Tahoe CA. Springer-Verlag Lecture Notes in Computer Science, Vol. 2211, pp. 306–323.

Bus Architectures For Safety-Critical Embedded Systems*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. Embedded systems for safety-critical applications often integrate multiple "functions" and must generally be fault-tolerant. These requirements lead to a need for mechanisms and services that provide protection against fault propagation and ease the construction of distributed fault-tolerant applications. A number of bus architectures have been developed to satisfy this need. This paper reviews the requirements on these architectures, the mechanisms employed, and the services provided. Four representative architectures (SAFEbus™, SPIDER, TTA, and FlexRay) are briefly described.

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

* This research was supported by the DARPA MOBIES and NEST programs through USAF Rome Laboratory contracts F33615-00-C-1700 and F33615-01-C-1908, and by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Incorporated.

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 [10]. 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.

In this paper, I survey some of the issues in the design of bus architectures for safety-critical embedded systems. I hope this will prove useful to potential users of these architectures and will alert others to the benefits of building on such well-considered foundations. My presentation is derived from a review of four representative architectures: two of these were primarily designed for aircraft applications and two for automobiles. The economies of scale make the automobile buses quite inexpensive—which then renders them attractive in certain aircraft applications. The aircraft buses considered are the Honeywell SAFEbus [1, 7] (the backplane dat bus used in the Boeing 777 Airplane Information Management System) and the NASA SPIDER [11] (an architecture being developed as a demonstrator for certification under the new DO254 guidelines [15]); the automobile buses considered are the TTTech Time-Triggered Architecture (TTA) [8, 24], recently adopted by Audi and Volkswagen for automobile applications, and by Honeywell for avionics and aircraft controls functions, and FlexRay [3], which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips. A detailed comparison of these four architectures, along with more extended discussion of the issues, is available as a technical report [17].

The paper is organized as follows: Section 2 examines general issues in time-triggered systems and bus architectures, Section 3 examines the fault hypotheses under which they operate, and Section 4 describes the services that they provide; Section 5 briefly describes the four representative architectures, and conclusions are provided in Section 6.

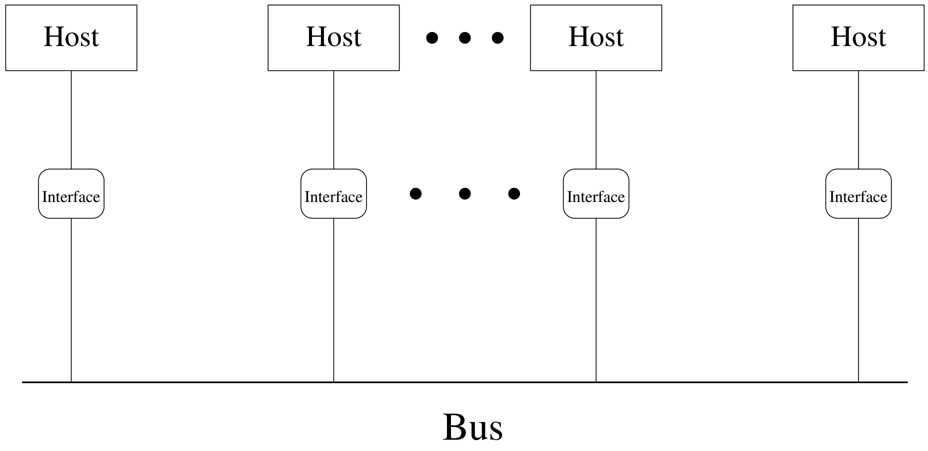

Fig. 1. Bus Interconnect

2 Time-Triggered Buses

The architectures considered here are called "buses" because multicast or broadcast communication is one of the services that they provide, and their implementations are based on a logical or physical bus. In a generic bus architecture, application programs run in *host* computers, and sensors and actuators are also connected to the hosts; an *interconnect* medium provides broadcast communications, and *interface* devices connect the hosts to the interconnect. The interfaces and interconnect comprise the bus; the combination of a host and its interface(s) is referred to as a *node*. Realizations of the interconnect may be a physical (passive) bus, as shown in Figure 1, or a centralized (active) hub, as shown in Figure 2. The interfaces may be physically proximate to the hosts, or they may form part of a more complex central hub. Many of the components will be replicated for fault tolerance.

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 Byteflight, CAN, Ethernet, LonWorks, or Profibus. 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. "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 is 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 prime contrast between these two approaches is their locus of control: a time-triggered system controls its own activity and interacts with the environment according to an internal schedule, whereas an event-triggered system is under the control of its environment and must respond to stimuli as they occur.

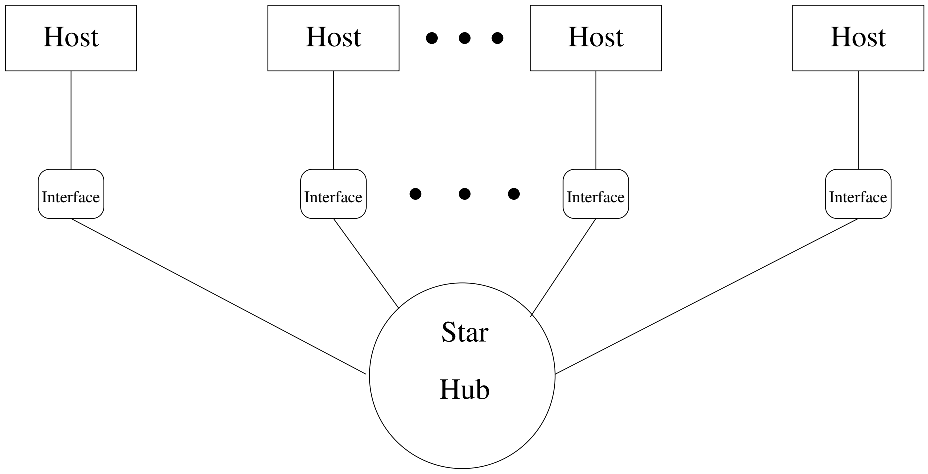

Fig. 2. Star Interconnect

Event-triggered systems allow flexible allocation of resources and this is attractive when demands are highly variable. However, in safety-critical applications it is necessary to guarantee some basic quality of service to all participants, even (or especially) in the presence of faults. Because the clients of the bus architecture are real-time embedded control systems, the required guarantees include predictable communications with low latency and low jitter (assured bandwidth is not enough). The problem with 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 non-safety-critical embedded systems such as CAN, LonWorks, or Profibus 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 therefore has to wait only for the current message to finish, while other messages must also 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 (e.g., the current message may be retransmitted in the case of transmission failure, thereby delaying the next message, even if this has higher priority). Furthermore, faulty nodes may not adhere to expected patterns of use and may make excessive demands for service, thereby reducing that available to others. Event-triggered buses for safety-critical applications add various mechanisms to limit such demands. ARINC 629 (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 automobile protocol developed by BMW extends this mechanism with guaranteed, preallocated slots for critical messages. But even preallocated slots provide no protection against a faulty node that fails to recognize them. The worst manifestation of this kind of fault is the so-called “babbling idiot” failure where a faulty node transmits constantly, thereby compromising the operation of the entire bus.

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 run time. A static schedule makes possible the control of the babbling idiot failure mode. This is achieved by interposing an independent component, called a bus *guardian*, that allows each node to transmit on the bus only when it is allowed to do so. The guardian must know when its node is allowed to access the bus, which is difficult to achieve in an event-triggered system but is conceptually simple in a time triggered system: the guardian has an independent clock and independent knowledge of the schedule and allows its node to broadcast only when indicated by the schedule.

Because all communication is triggered by the global schedule, there is no need to attach source or destination addresses to messages sent over a time-triggered bus: each node knows the sender and intended recipients of each message by virtue of the time at which it is 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: namely, the possibility that a faulty node may send messages to the wrong recipients or, worse, may masquerade as a sender other than itself.

Fault-tolerant clock synchronization is a fundamental requirement for a time-triggered bus architecture: the abstraction of a global clock is realized by each node having a local clock that is closely synchronized with the clocks of all other nodes. 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 each other node (e.g., by comparing the arrival time of each message with its expected value) 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 might provide different, or missing, readings to different observers), so we need a “fault-tolerant average” that is largely insensitive to a certain number of readings from faulty clocks. Schneider [18] 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 [25] is a popular choice that is characterized by use of the “fault-tolerant midpoint" as its averaging function. 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. The number of faulty nodes that can be tolerated, and the quality of synchronization that can be achieved, depend on the details of the algorithm, and on the fault hypothesis under which it operates. The event-based algorithm of Srikanth and Toueg [21] is particularly attractive because it achieves optimal accuracy.

3 Fault Hypotheses and Fault Containment Units

Safety-critical aerospace functions are generally required to have failure rates less than 10^{-9} per hour [5], and an architecture that is intended to support several such functions should provide assurance of failure rates better than 10^{-10} per hour. Similar requirements apply to cars (although higher rates of loss are accepted for individual cars than aircraft, there are vastly more of them, so the required failure rates are similar). 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.

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 applications. 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 *modes* (i.e., kinds) of faults that are to be tolerated, and their maximum *number* and *arrival rate*. 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. 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) are considered in this analysis: design faults must be excluded, and must be shown to be so by stringent assurance and certification processes.

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.

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.

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 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.

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 [23]. 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).

The great advantage to designs that can tolerate arbitrary fault modes is that we do not have to justify assumptions about more 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 the attraction, in safety-critical contexts, of systems that can tolerate arbitrary faults. This point is often misunderstood and such systems are derided as being focused on asymmetric or Byzantine faults, “which never arise in practice.” Byzantine faults are just one manifestation of arbitrary behavior, and cannot simply 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.

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 digital 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.)

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 [16] 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 (unless digital signatures are employed—which is equivalent to reducing the severity of the arbitrary fault mode).

Because it is algorithmically much easier to tolerate simple failure modes, some architectures (e.g., SAFEbus) arrange FCUs (the "Bus Interface Units" 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). 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.

Some 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. In such cases, the *fault arrival rate* is important: faults must not arrive faster than the architecture can reconfigure. The architectures considered here operate according to static schedules, which consist of "rounds" or "frames" that are executed repeatedly in a cyclic fashion. The acceptable fault arrival 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).

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 high-intensity radiated fields (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. 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. Although it is difficult to provide assurance of correct operation during these upsets, 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 [20].

Restart during operation may be necessary if HIRF or other environmental influences lead to violation of the fault hypothesis and cause a 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 for only a few cycles—longer outages will lead to loss of control. For example, Heiner and Thurner estimate that the maximum transient outage time for a steer-by-wire automobile application is 50ms [6].

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 can send an uninterrupted "wake up" message) and, ideally, should not depend on collision detection (because this cannot be done reliably). Notice that it must be possible to perform startup and restart reliably even in the presence of faulty components.

4 Services

The essential basic purpose of these bus 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 from the configuration. A difficulty 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 [10]. 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.

The problem of distributing data consistently in the presence of faults is variously called *interactive consistency*, *consensus*, *atomic broadcast*, or *Byzantine agreement* [9, 12]. 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 between them, and $a + 1$ levels (or "rounds") of communication. The number of FCUs and the number of disjoint paths required, but not the number of rounds, 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.

The first choice is so unattractive that it vitiates the whole purpose of a fault-tolerant bus architecture. Most bus architectures therefore provide some type of interactively consistent message broadcast as a basic service. In addition, most architectures take steps to reduce the incidence 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 a central hub or "in series" with each interface, the bus guardians are 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).

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 [19], 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.

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 is 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).

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 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" [2].

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. Conversely, a membership and clique-avoidance service can assist the construction of an interactively consistent message service: simply exclude from the membership any node that receives a message different than the majority (TTA does this).

5 Practical Implementations

Here, we provide sketches of four bus architectures that provide concrete solutions to the requirements and design challenges outlined in the previous sections. More details are available in a companion report to this paper [17]. All four buses support the time-triggered model of computation, employ fault-tolerant distributed clock synchronization, and use bus guardians or some equivalent mechanism to protect against babbling idiot failure modes. They differ in their fault hypotheses, mechanisms employed, services provided, and in their assurance, performance, and cost.

SAFEbus. Honeywell developed SAFEbus™ (the principal designers are Kevin Driscoll and Ken Hoyme [7]) to serve as the core of the Boeing 777 Airplane Information Management System (AIMS) [22], which supports several critical functions, such as cockpit displays and airplane data gateways. The bus has been standardized as ARINC 659 [1] and variations on Honeywell's implementation are being used or considered for other avionics and space applications. It uses a bus interconnect similar to that shown in Figure 1; the interfaces (they are called Bus Interface Units, or BIUs) are duplicated, and the interconnect bus is quad-redundant. Most of the functionality of SAFEbus is implemented in the BIUs, which perform clock synchronization and message scheduling and transmission functions. Each BIU of a pair is a separate FCU and 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 and operate at 30MHz. 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; in particular, clock synchronization is achieved using an event-based algorithm.

The paired BIUs at sender and receiver, and the quad-redundant buses, provide sufficient redundancy for SAFEbus to provide interactively consistent message broadcasts (in the Honeywell implementation) using an approach similar to that described by Davies and Wakerly [4] (this remarkably prescient paper anticipated many of the issues and solutions in Byzantine fault tolerance by several years). It also supports application-level fault tolerance (based on self-checking pairs) by providing automatic rapid rollover from masters to shadows.

Its fault hypothesis includes arbitrary faults, faults in several nodes (but only one per node), and a high rate of fault arrivals. It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed its fault hypothesis. It tolerates spatial proximity faults in the AIMS application by duplicating the entire system. SAFEbus is certified for use in passenger aircraft and has extensive field experience in the Boeing 777. The Honeywell implementation is supported by an in-house tool chain.

SAFEbus is the most mature of the four buses considered, and makes the fewest compromises. But 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. The Time Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [8]. 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.

Current implementations of TTA use a bus interconnect similar to that shown in Figure 1. The interfaces (they are called *controllers*) implement the TTP/C protocol [24] that is at the heart of TTA, providing clock synchronization, and message sequencing and transmission functions. The interconnect bus is duplicated and each controller drives both of them through partially independent bus guardians. TTA uses an averaging clock synchronization algorithm based on that of Lundelius and Lynch [25]. This algorithm is implemented in the controllers, but requires too many resources to be replicated in the bus guardians. The guardians, which have independent clocks, therefore rely on their controllers for a "start of frame" signal. This compromises their independence somewhat (they also share the power supply and some other resources with their controllers), so forthcoming implementations of TTA use a star interconnect similar to that shown in Figure 2. Here, the guardian functionality is implemented in the central hub which is fully independent of the controllers: the hubs and controllers comprise separate FCUs in this implementation. Hubs are duplicated for fault tolerance and located apart to withstand spatial proximity faults. They also perform signal reshaping to reduce the incidence of SOS faults.

TTA employs algorithms for group membership and clique avoidance [2]; these enable its clock synchronization algorithm to tolerate multiple faults (by reconfiguring to exclude faulty members) and combine with its use of checksums (which can be considered as digital signatures) to provide a form of interactively consistent message broadcasts. The membership service supports application-level fault tolerance based on master-backup or compensation. Proposed extensions provide state machine replication in a manner that is transparent to applications.

The fault hypothesis of TTA includes arbitrary faults, and faults in several nodes (but only one per node), provided these arrive at least two rounds apart (this allows the membership algorithm to exclude the faulty node). It never gives up and has a well-defined restart and recovery strategy from fault arrivals that exceed this hypothesis.

The prototype implementations of TTA have been subjected to extensive testing and fault injections, and deployed in experimental vehicles. Several of its algorithms have been formally verified [13, 14], 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 (e.g., Matlab/Simulink and Beacon). Current implementations provide 25 Mbit/s data rates; research projects are designing implementations for gigabit rates. TTA controllers and the star hub (which is basically a modified controller) are quite simple and cheap to produce in volume.

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.

SPIDER. A Scalable Processor-Independent Design for Electromagnetic Resilience (SPIDER) is being developed by Paul Miner and colleagues at the NASA Langley Research Center as a research platform to explore recovery strategies for radiation-induced (HIRF/EMI) faults, and to serve as a case study to exercise the recent design assurance guidelines for airborne electronic hardware (DO-254) [15].

SPIDER uses a star configuration similar to that shown in Figure 2, in which the interfaces (called BIUs) may be located either with their hosts or in the centralized hub, which also contains active elements called Redundancy Management Units, or RMUs.

Clock synchronization and other services of SPIDER are achieved by novel distributed algorithms executed among the BIUs and RMUs [11]. The services provided include interactively consistent message broadcasts, and identification of failed nodes (from which a membership service can easily be synthesized). SPIDER's fault hypothesis uses a hybrid fault model, which includes arbitrary faults, and allows some combinations of multiple faults. Its algorithms are novel and highly efficient and are being formally verified.

SPIDER is an interesting design that uses a different topology and a different class of algorithms from the other buses considered here. However, it is a research project whose design and implementation are still in progress and so it cannot be compared directly with the commercial products.

FlexRay. A consortium including BMW, DaimlerChrysler, Motorola, and Philips, is developing FlexRay 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. Published descriptions of the FlexRay protocols and implementation are sketchy at present [3] (see also the Web site `www.flexray-group.com`).

FlexRay can use either an "active" star interconnect similar to that shown in Figure 2, or a "passive" bus similar to that shown in Figure 1. In both cases, duplication of the interconnect is optional. The star configuration of FlexRay (and also that of TTA) can also be deployed in distributed configurations where subsystems are connected by hub-to-hub links. Each FlexRay interface (it is called a communication controller) drives the lines to its interconnects through separate bus guardians located with the interface. (This means that with two buses, each node has three clocks: one for the controller and one for each of the two guardians; this differs from the bus configuration of TTA where there is one clock for the controller and both guardians share a second clock.) Like the bus configuration of TTA, the guardians of FlexRay are not fully independent of their controllers.

FlexRay aims to be more flexible than the other buses considered here, and this seems to be reflected in the choice of its name. As noted, one manifestation of this flexibility is its combination of time- and event-triggered operation. 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. Communication during the event-driven portion of the cycle uses the Byteflight protocol. 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 only of those slots allocated to their transmissions (nodes requiring greater bandwidth are assigned more slots than those that require less). Controllers learn 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 appears to be a 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.

Like TTA, FlexRay uses the Lundelius-Welch clock synchronization algorithm but, unlike TTA, it does not use a membership algorithm to exclude faulty nodes. FlexRay provides no services to its applications beyond best-efforts message delivery; in particular, it does not provide interactively consistent message broadcasts. This means that all mechanisms for fault-tolerant applications must be provided by the applications programs themselves. Published descriptions of FlexRay do not specify its fault hypothesis, and it appears to have no mechanisms to counter certain fault modes (e.g., SOS faults or other sources of asymmetric broadcasts, and masquerading on startup). A never-give-up strategy has not been described, nor have systematic or formal approaches to assurance and certification.

FlexRay is interesting because of its mixture of time- and event-triggered operation, and potentially important because of the industrial clout of its developers. Currently, it is the slowest of the commercial buses, with a claimed data rate of no more than 10 Mbit/s.

6 Summary and Conclusion

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 its fault hypothesis).

Partitioning is the minimum requirement, however. It ensures that one failed function will not drag down others, but in many 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 master/shadow rollover or group membership, that can be provided with much increased efficiency and much reduced latency when implemented at a low level. FlexRay, alone, provides none of these services. In their absence, all mechanisms for fault-tolerance must be implemented in applications programs. 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, applications 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 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.

It is unlikely that any single bus architecture will satisfy all needs and markets, so it is possible that FlexRay's lack of application-level fault-tolerant services will find favor in some areas. It is also to be expected that new or modified architectures will emerge to satisfy new markets and requirements. (For example, it is proposed that TTA could match FlexRay's ability to support event-triggered as well as time-triggered communications by allocating certain time slots to a simulation of CAN; the simulation is actually faster than a real CAN bus, while retaining all the safety attributes of TTA.) I hope that the description provided here will help potential users to evaluate existing architectures against their own needs, and that it will help designers of new architectures to learn from and build on the design choices made by their predecessors.

References

1. *ARINC Specification 659: Backplane Data Bus*. Aeronautical Radio, Inc, Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.
2. 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.
3. Joef Berwanger et al. FlexRay–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Society of Automotive Engineers, Detroit, MI, April 2001. Paper number 2001-01-0676.
4. Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978.
5. *System Design and Analysis*. Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

6. 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, IEEE Computer Society, Munich, Germany, June 1998.
7. Kenneth Hoyme and Kevin Driscoll. SAFEbus™. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 68–73, Seattle, WA, October 1992.
8. Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.
9. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
10. 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.
11. Paul S. Miner. Analysis of the SPIDER fault-tolerance protocols. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, NASA Langley Research Center, Hampton, VA, June 2000. Slides available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Presentations/lfm2000-spider/`.
12. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
13. Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, eds., *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, Oct. 2000.
14. Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Charles B. Weinstock and John Rushby, eds., *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 207–226, San Jose, CA, Jan. 1999.
15. *DO254: Design Assurance Guidelines for Airborne Electronic Hardware*. Requirements and Technical Concepts for Aviation, Washington, DC, April 2000.
16. 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, Association for Computing Machinery, Los Angeles, CA, August 1994. Also available as NASA Contractor Report 198289.
17. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 2001. Available at `http://www.csl.sri.com/~rushby/papers/buscompare.pdf`.
18. Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1987.
19. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
20. Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
21. T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
22. William Sweet and Dave Dooling. Boeing's seventh wonder. *IEEE Spectrum*, 32(10):20–23, October 1995.
23. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, IEEE Computer Society, Columbus, OH, October 1988.
24. *Specification of the TTP/C Protocol*. Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria, July 1999.
25. J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.