NASA/CR–2010-216724

Monitoring Distributed Real-Time Systems: A Survey and Future Directions

Alwyn E. Goodloe
National Institute of Aerospace, Hampton, Virginia

Lee Pike
Galois, Inc., Portland, Oregon

July 2010

NASA STI Program . . . 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 plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and 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 also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, 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 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2010-216724

Monitoring Distributed Real-Time Systems: A Survey and Future Directions

Alwyn E. Goodloe
National Institute of Aerospace, Hampton, Virginia

Lee Pike
Galois, Inc., Portland, Oregon

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL08AD13T

July 2010

Acknowledgments

This work is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office. We thank Ben Di Vito and Paul Miner of the NASA Langley Research Center, Radu Siminiceanu of the National Institute of Aerospace, and Levent Erkök of Galois, Inc. for their comments on a draft of this report.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

Runtime monitors have been proposed as a means to increase the reliability of safety-critical systems. In particular, this report addresses runtime monitors for distributed hard real-time systems. This class of systems has had little attention from the monitoring community. The need for monitors is shown by discussing examples of avionic systems failure. We survey related work in the field of runtime monitoring. Several potential monitoring architectures for distributed real-time systems are presented along with a discussion of how they might be used to monitor properties of interest.

Contents

List of Figures

1 Introduction

Former U.S. President Ronald Reagan's signature phrase was the Russian proverb "Trust, but verify." That phrase is symbolic of the political environment during the U.S.-Soviet Cold War. For safety-critical systems, we must have a similar level of vigilance. The probability of a catastrophic failure occurring in ultra-critical digital systems—such as flight-critical commercial avionics—should be no greater than one in a billion per hour of operation despite the hostile environments in which they execute [2]. To achieve this order of reliability, a system must be designed to be fault-tolerant. However, unanticipated environmental conditions or logical design errors can significantly reduce a system's hypothesized reliability.

Testing is infeasible to demonstrate that a system exhibits "1 in a billion" reliability—the essential problem is that simply too many tests must be executed [3]. Formal verification—*i.e.*, rigorous mathematical proof—at the code level that a system exhibits ultra-reliability is also currently impractical for industrial designs, although "light-weight" methods continue to gain traction [4].

Because neither testing nor formal verification alone is sufficient to demonstrate the reliability of ultra-reliable systems, the idea of monitoring a system at runtime has been proposed. A *monitor* observes the behavior of a system and detects if it is consistent with a specification. We are particularly interested in *online* monitors, which check conformance to a specification at runtime (as opposed to *offline*, at a later time) and can therefore drive the system into a known good state if it is found to deviate from its specification. A monitor can provide additional confidence at runtime that the system satisfies its specifications.

Survey Contributions As we describe in Section 4, research in monitoring has mostly focused on monitors for software that is neither real-time nor distributed. Only a few studies have addressed monitoring real-time or distributed systems, which characterized safety-critical systems such as flight-critical systems for aircraft and spacecraft.

The open question—and the one for which this survey lays the groundwork for answering—is whether safety-critical embedded systems can be made more reliable by online monitoring.

In summary, the contributions of this survey are as follows:

- *Distributed real-time monitoring*: An investigation of a class of systems that is underrepresented in the monitoring literature and for which monitoring may improve its reliability.
- *System-level monitoring*: A focus on system-level properties, like fault-tolerance and distributed consensus, that cannot be monitored locally but are global properties of a distributed system.
- *Foundations for distributed real-time system monitoring*: A set of requirements that monitors for real-time distributed systems must satisfy as well as three abstract architectures for monitoring (see Section 6). We also present a set of properties for monitors to address in these systems (see Section 7).

Outline A brief outline of this document follows:

- Section 2: Describes three failures of safety-critical systems, two in commercial aviation and one in the Space Shuttle. We intermittently reference these scenarios throughout the document to ground our discussions.
- Section 3: Presents definitions, terminology, and basic concepts on distributed systems, fault-tolerance, and real-time systems.

- Section 4: Surveys recent research in monitoring broadly related to our investigation of monitoring distributed real-time systems.
- Section 5: Places monitoring in context given that our investigations are particularly motivated by safety-critical Space Shuttle and aircraft systems. In this context, monitoring is one aspect of an *integrated vehicle health management* approach to system reliability.
- Section 6: Describes how theoretical results in distributed systems (particularly pertaining to synchrony and faults) affect the ability to monitor them. We then present requirements that a monitoring architecture must satisfy to be beneficial to a system under observation. Finally, three conceptual architectures (bus monitor, single process monitor, and distributed processes monitor) for monitoring distributed real-time systems are presented.
- Section 7: Describes properties to be monitored in distributed real-time systems including various kinds of timing and data faults, point-to-point error detection, and fault-tolerant management software.
- Section 8: Summarizes the report and make concluding remarks.

2 The Need for Monitors: Real-World Failures

The design of highly-reliable systems is driven by the functionality it must deliver as well as the faults it must survive. Three case studies are presented in which safety-critical computer systems failed. Although none of these errors resulted in the loss of life or loss of the vehicle, the problems were sufficiently severe as they could have led to such a loss if they occurred in less favorable circumstances. First, is a presentation of a case study of a fault that recently occurred during the launch sequence of the shuttle orbiter. The next section covers an in-flight upset of a Boeing 777 due to a software design error. Finally, an in-flight upset of an Airbus A330 is presented.

2.1 Shuttle MDM Failure

The Space Shuttle's data processing system has four general purpose computers (GPC) that operate in a redundant set. There are also twenty-three multiplexer de-multiplexer (MDM) units aboard the orbiter, sixteen of which are directly connected to the GPCs via shared buses. Each of these MDMs receive commands from guidance navigation and control (GNC) running on the GPC and acquires requested data from sensors attached to it, which is then sent to the GPCs. In addition to their role in multiplexing/demultiplexing data, these MDM units perform analog/digital conversion. Data transfered between the GPC and MDMs is sent in the form of serial digital data.

The GPCs execute redundancy management algorithms that include a fault detection, isolation, and recovery (FDIR) function. During the launch of shuttle flight Space Transportation System 124 (STS-124), there was reportedly a pre-launch failure of the fault diagnosis software due to a "non-universal I/O error" in the Flight Aft (FA) MDM FA2 located in the orbiter's aft avionics bay [5].

According to [5, 6], the events unfolded as follows:

- A diode failed on the serial multiplexer interface adapter (SMIA) of the FA2 MDM.
- GPC 4 receives erroneous data from FA2. Each node votes and views GPC 4 as providing faulty data. Hence GPC 4 is voted out of the redundant set.
- Three seconds later GPC 2 also receives erroneous data from FA2. In this case, GPC 2 is voted out of the redundant set.
- In accordance with the Space Shuttle flight rules [7], GPC 2 and GPC 4 are powered down.
- The built in test equipment for FA 2 was then queried (port 1 status register for MDM FA 2). This caused GPC 3 to fall out of the redundant set with GPC 1. GPC 1 was left to run to gather more data, but engineers terminated the launch and the problem with FA2 was isolated and the unit replaced.

The above set of events sequentially removed good GPC nodes, but failed to detect and act on the faulty MDM. Based on the analysis reported in [6], it seems that the system does not tolerate single points of failure. Even though the nodes were connected to the MDM via a shared bus, conditions arose where different nodes obtained different values from MDM FA2 (Driscoll *et al.*, describe possible causes of Byzantine faults over a shared bus [1]). The observed behavior is consistent with the MDM FA2 exhibiting a Byzantine failure sending different values to the GPCs using the topology in Figure 1. The design engineers likely assumed that since the GPCs and MDMs all communicate via a shared bus, that each GPC would always obtain the same value from the MDM and consequently asymmetric faults would not occur.

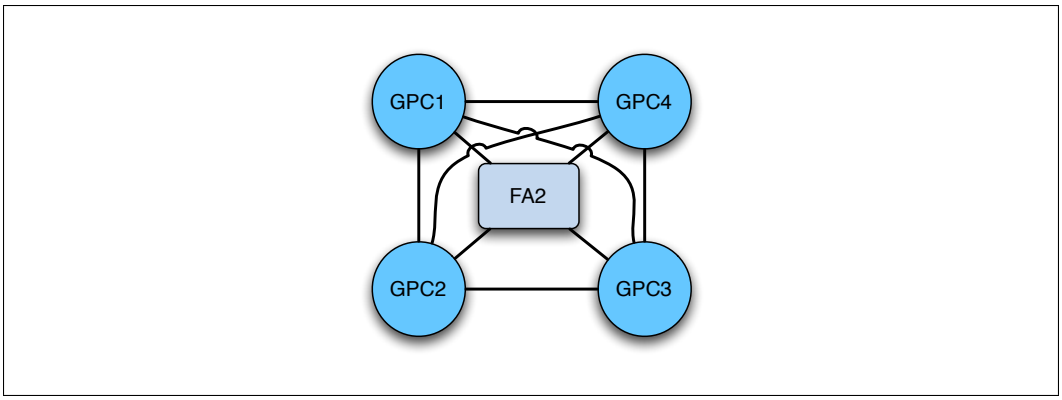

Figure 1. MDM, FA2, and GPCs

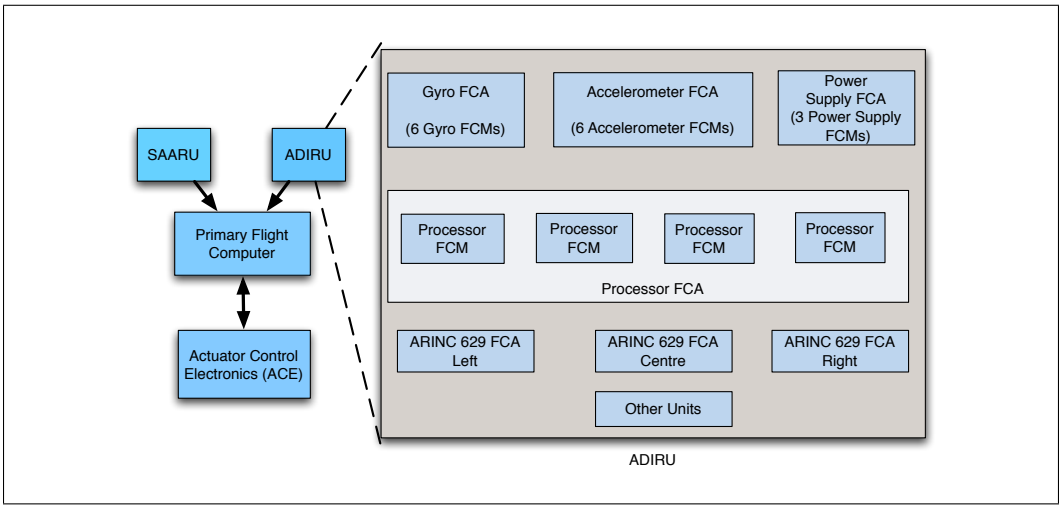

Figure 2. Boeing 777 Flight Computer

2.2 Boeing 777 In-Flight Upset

The primary flight computer on the Boeing 777 receives inertial data from the Air Data Inertial Reference Unit (ADIRU) [8], the Secondary Altitude and Air Data Reference Unit (SAARU), and Actuator Control Electronics (ACE) unit as depicted in Figure 2. These are all connected to the primary flight computer via Aeronautical Radio Inc. (ARINC) 629 units. The ADIRU and SAARU both accept inputs from a variety of sources including *Pitot probes*—devices used to measure air speed—and sensors indicating air temperature and aircraft angle of attack. The data from these sources is then used in computations, the results of which are fed to the flight computers. The two units differ in their design and construction ostensibly to provide fault-tolerance through design heterogeneity. Both units provide inertial data to the flight computer, which selects the median value of the provided data. The primary flight computer will compute aircraft control surface position commands based on input data and sends commands to the ACE, which perform analog/digital conversion, controls the variable feel actuators, and controls the aircraft control surfaces [9,10]. The Boeing 777 has three flight computers to provide triple redundancy.

The ADIRU depicted in Figure 2 is composed of seven fault containment Areas (FCA), each of which contains several fault containment modules (FCM) [8]. For instance, the Gyro FCA contains six ring-laser gyro FCMS and the Accelerometer FCA contains six accelerometer FCMs. The processor FCA is composed of four processor FCMs that execute redundancy management software that performs fault detection and isolation.

On August 1, 2005 a Boeing 777-120 operated as Malaysia Airlines Flight 124 departed Perth, Australia for Kuala Lumpur, Malaysia. Shortly after takeoff, the aircraft experienced an in-flight upset event as it was climbing to flight level 380.[1] According to [11], the events unfolded as follows:

- A low airspeed advisory was observed and simultaneously the slip/skid deflected to the full right indicating the aircraft was out of trim in the yaw axis.
- The primary flight display indicated that the aircraft was simultaneously approaching overspeed limit and the stall speed limit.
- The aircraft nose pitched up sending the aircraft climbing to flight level 410.
- The indicated speed decreased from 270 nautical miles per hour (kts) to 158kts and the stall warning and stick shaker devices activated.
- The pilot reported he disconnected the autopilot and lowered the nose of the aircraft.
- The aircraft autothrottle then executed an increase in thrust, which the pilot counteracted by moving the thrust levers into idle.
- The crew was then able to use radar assistance in order to return to Perth.

Although no one was injured, the erratic information presented to the crew meant that they could not trust the instruments that they depend upon to safely fly the aircraft. The fact that the autopilot acted on the erratic information gave additional cause for concern.

An analysis performed by the Australian Transport Safety Bureau [11] reported that the problem stemmed from a bug in the ADIRU software that was exposed by a series of events that was unlikely to have been revealed during certification testing. On June 2001, accelerometer 5 failed, but rather than failing in a fail stop manner, it continued to output high voltage values. The failure is recorded in the on-board maintenance computer, but was not directly readable by the crew and since no in-flight warning had previously been sounded, there was no clear directive to replace the unit. The software was programmed to disregard output from accelerometer 5 and to use data produced by back-up accelerometers. The accelerometer failure was masked each time power was cycled in the ADIRU because even though the error was logged in the maintenance computer, the memory of this computer was not checked during the initialization process. Approximately one second before the upset event was recorded, accelerometer 6 failed. Due to a software bug, the fault-tolerant software used the erroneous data produced by accelerometer 5. The bug had been in previous releases of the ADIRU software, but had been masked by other code. In the version of the software used in the aircraft in question, the fact that only an error in accelerometer 6 had been detected since the unit was started meant that the software assumed there was no problem with accelerometer 5. A detailed critique from various perspectives can be found in Johnson and Holloway [12].

The designers clearly wanted to separate the maintenance computer's functionality from the operational flight functions, but a software bug resulted in errors detected by the maintenance computer not being transmitted to the flight systems and thus not realizing that accelerometer 5 was faulty.

[1] The "flight level" is a standard nominal altitude of an aircraft in hundreds of feet calculated from the world-wide average sea-level pressure.

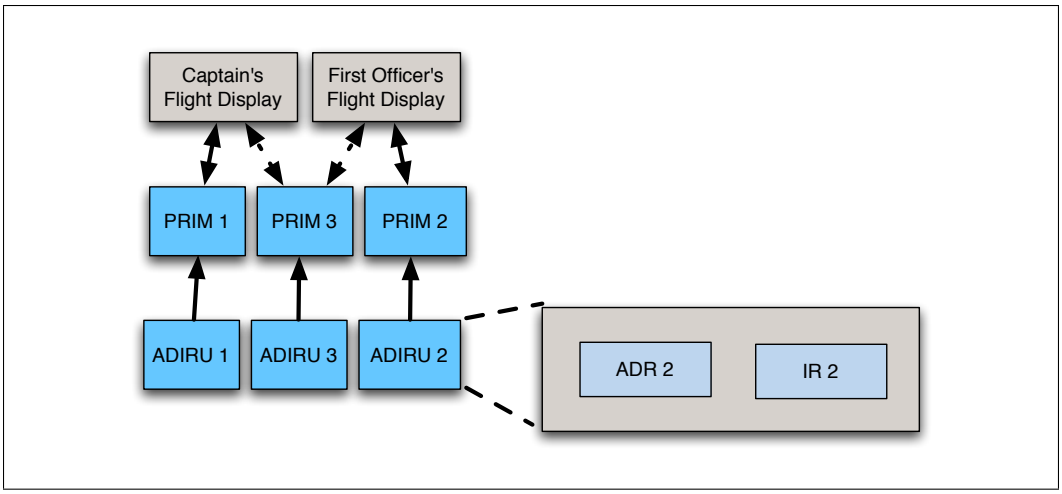

Figure 3. A330 Display/PRIM/Display Configuration

2.3 A330 In-Flight Upset

The Airbus A330 has three primary flight computers (known as PRIMs) and two secondary flight computers. One PRIM acts as the master sending orders to the other computers. The PRIMs are connected to three Air Data Inertial Reference Units (ADIRU). An ADIRU is composed of the following components:

- An air data reference (ADR) component that supplies barometric altitude, speed, angle of attack information, and other data. Air data modules convert pneumatic data from pitot and static probes into numeric data for the ADIRUs.
- An inertial reference (IR) component that supplies attitude, flight-path vector, track heading, and other data. This component has two independent GPS receivers.

ADIRU 1 is connected to PRIM 1 and ADIRU 2 is connected to PRIM 2 and ADIRU 3 is connected to PRIM 3. The pilot and copilot's displays are connected to PRIMs 1 and 2 respectively, but either display can switch to PRIM 3 if necessary. The architecture is illustrated in Figure 3.

The Air Data Inertial Reference System (ADIRS) control panel is located in the overhead panel of the flight deck. The panel provides local fault indications for the ADIRU IR and ADR components. A fault is indicated for a particular part (IR1, IR2, IR3, ADR1, ADR2, ADR3) by an amber fault light. The relevant part can be deactivated by pressing a button.

The flight computers execute a set of control laws. In 'normal law', the computer prevents the aircraft from exceeding a predefined safe flight envelope regardless of flight crew input. Note that many of the decisions are based on input from the ADIRUs. In an 'alternate law', there are either no protections offered or there are different types of protections used.

On October 7, 2008 an Airbus A330 operated as Qantas Flight QF72 from Singapore to Perth, Australia was cruising at flight level 370 with autopilot and auto-thrust engaged when an in-flight upset occurred. The events unfolded as follows [13, 14]:

- Autopilot disengaged and the *IR 1 failure* indication appeared on the ADIRS control panel.
- The flight display indicated that the aircraft was simultaneously approaching over-speed limit and the stall speed limit.

- The *ADR 1 failure* indication appeared on the ADIRS control panel.
- Two minutes into the incident, a high angle of attack was reported and the computers ordered a significant nose-down pitch and the plane descended 650ft.
- The crew switched the PRIM master from PRIM 1 to PRIM 2.
- PRIM 3 indicates a fault.
- The crew returned the aircraft to flight level 370.
- The captain switched his display to show data from ADIRU 3 instead of ADIRU 1.
- The computers ordered a nose down pitch and the plane descends 400ft.
- The captain applied back pressure to the sidestick.
- The crew switched the PRIM master from PRIM 2 to PRIM 1.
- The flight control law was manually changed from 'normal law' to 'alternate law' so that the computer was no longer enforcing predefined flight parameters.
- The crew made an emergency landing at Learmouth.

The incident resulted in injuries requiring fourteen people to be hospitalized.

The preliminary investigation uncovered that ADIRU 1 failed in a manner producing spurious spikes in data values [14]. Additional spikes in data from ADIRU 1 continued throughout the flight. Given that the computers were still reacting to corrupt data, it appears that the computer still accepted information from ADIRU 1 after the pilot switched the display to no longer show information from that unit. It was only after the pilot changed to the alternate control law did the aberrant behavior cease since the computers were no longer using algorithms that depended on the corrupt data. Although a final report is not expected for some time, it appears that ADIRU 1 failed while ADIRU 2 and ADIRU 3 appear to have operated normally, but the system design failed to identify and isolate the problem automatically. As long as the normal control law was engaged, the avionics system seems to have used the spurious data from ADIRU 1. The system designers likely assumed that such a problem would be detected and the bad unit disengaged before suffering any ill effects.

It is not clear from the literature exactly how fault tolerant the system was designed to be without human intervention. The system was not able to recover from a single point of failure of what seems to have been a babbling device without the crew taking action. Hence the architecture of the avionics was likely a contributing factor.

3 Preliminary Concepts

Having provided some motivating examples of real-world safety-critical system failures, this section introduces some preliminary concepts and terminology used throughout the remainder of this survey. Specifically, we introduce distributed systems, fault-tolerance, and real-time systems.

3.1 Distributed Systems

Introductory material on the foundations of distributed systems and algorithms can be found in Lynch's textbook [15]. A distributed system is modeled as a graph with directed edges. Vertices are called *nodes* or *processes*. Directed edges are called *communication channels* or *channels*. If channel c points from node p to node q, then p can send messages over c to q, and q can receive messages over c from p. In this context, p is the *sender* or *transmitter*, and q is the *receiver*.

The only distributed systems considered in this survey are those containing a fixed set of nodes and a fixed set of interconnects between nodes. Nodes or interconnects being introduced or removed from the system only happens at a conceptual level, resulting from faults removing nodes and interconnects, or nodes and interconnects reintegrating into the system after suffering a transient fault [16].

3.2 Fault-Tolerance

The terms 'failure', 'error', and 'fault' have technical meanings in the fault-tolerance literature. A *failure* occurs when a system is unable to provide its required functions. An *error* is "that part of the system state which is *liable to lead to subsequent failure*," while a *fault* is "the *adjudged or hypothesized cause* of an error" [17]. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to system failure.

In this report, we are primarily concerned with architectural-level fault-tolerance [18]. A *fault-tolerant system* is one that continues to provide its required functionality in the presence of faults (for the faults tolerated). A fault-tolerant system must not contain a *single point of failure* such that if the single subsystem fails, the entire system fails. Thus, fault-tolerant systems are often implemented as distributed collections of nodes such that a fault that affects one node or channel will not adversely affect the whole system's functionality.

A *fault-containment region* (FCR) is a region in a system designed to ensure faults do not propagate to other regions [19]. The easiest way to ensure this is to physically isolate one FCR from another. However, because FCRs may need to communicate, they share channels. Care must be taken to ensure faults cannot propagate over these channels. Generally, physical faults in separate FCRs are statistically independent, but under exceptional circumstances, simultaneous faults may be observed in FCRs. For example, widespread high-intensity radiation may affect multiple FCRs.

Here, we characterize the faults of a node in a distributed system based on the messages other nodes receive from it. Faults can be classified according to the *hybrid fault model* of Thambidurai and Park [20]. (The same characterization could be made of channels.) First, a node that exhibits the absence of faults is *non-faulty* or *good*. A node is called *benign* or *manifest* if it sends only *benign messages*. Benign messages abstract various sorts of misbehavior that is reliably detected by the transmitter-to-receiver fault-detection mechanisms implemented in the system. For example, a message that suffers a few bit errors may be caught by a cyclic redundancy check. In synchronized systems, nodes that send messages received at unexpected times are considered to be benign, too. A node is called *symmetric* if it sends every receiver the same message, but these messages may be

arbitrary. A node is called *asymmetric* or *Byzantine* if it sends different messages to different receivers, and at least one of the messages received is not detectably faulty [21]. (Note that the other messages may or may not be incorrect.)

The foregoing list of faults is not exhaustive. More elaborate fault models have been developed [22]. The benefit of more refined fault models is that a system designed to diagnose less severe faults can exhibit more reliable behavior in the presence of other faults as well. For example, a six-node distributed system can only tolerate one Byzantine fault, but it can tolerate up to one Byzantine fault and two benign faults [15]. Advanced fault-tolerant architectures, like NASA's SPIDER, are carefully designed under hybrid fault model for maximal fault-tolerance for the amount of redundant hardware [23, 24].

A *maximum fault assumption* (MFA) states the maximum kind, number, and arrival rate of faults for each FCR under which the system is hypothesized to operate correctly. If the MFA is violated, the system may behave arbitrarily. The satisfaction of the MFA itself is established by statistical models that take into account experimental data regarding the reliability of the hardware, the environment, and other relevant factors [25]. For example, for safety-critical systems designed for commercial aircraft, statistical analysis should ensure that the probability of their MFAs being violated is no greater than 10^{-9} per hour of operation [2]. Note that even if a system is proved to behave correctly under its MFA, but the probability of the MFA being violated is too high, the system will not reliably serve its intended function.

3.3 Real-Time Systems

Real-time systems are those that are subject to operational deadlines called "real-time" constraints. Consequently the correctness of such systems depends on both temporal and functional aspects. Real-time systems are generally classified as *soft real-time systems* or *hard real-time systems*. In soft real-time systems, missing a deadline degrades performance. For instance, dropping video frames while streaming a movie may inconvenience the viewer, but no permanent harm is done. In hard real-time systems, deadlines cannot be missed. For instance, a car engine control system is a hard-real time system since missing a deadline may cause the engine to fail. In such systems, deadlines must be kept even under worst-case scenarios.

4 Monitors: An Introduction and Brief Survey

A *monitor* observes the behavior of a system and detects if it is consistent with a given specification. The observed system may be a program, hardware, a network, or any combination thereof. We refer to the monitored system as the *system under observation* (SUO). If the SUO is observed to violate the specification, an alert is raised. Monitoring can be applied to nonfunctional aspects of a SUO such as performance, but historically, its focus has been on functional correctness. A variety of survey articles on monitoring have been published [26–29].

Early work on software monitoring focused on *off-line* monitoring, where data is collected and the analysis done off-line [30–32]. (Indeed, the term monitor was usually used to denote the act of collecting a program trace.) The focus of recent research has been *online* monitoring, where a specification is checked against an observed execution dynamically (although online monitoring may only be used during testing, if, for example, monitoring consumes too many additional resources). Online monitoring can be performed *in-line*, in which case the monitor is inserted into executing code as annotations. The Anna Consistency Checking System (Anna CCS) [33] is representative of a number of early monitor annotation systems. In Anna CCS, a user annotates Ada programs with properties written in the Anna specification notation and the system generates a function that acts as a monitor for this property. The functions are then called from the location where the annotation was placed. Recently, Java 5 [34] allows basic assertions to be inserted into programs. Online monitoring can also be *out-line*, where the monitor executes as a separate process [35, 36]. Examples of monitoring systems whose architectures combine aspects of both in-line and out-line monitoring are presented later. Historically, the focus has been on monitoring monolithic systems instead of distributed systems. A discussion of monitoring distributed systems is deferred until Section 4.1.

The research challenges of online monitoring include implementing efficient monitors (assuming the monitor shares resources with the observed system) that are synthesized from higher-level behavior specifications. In particular, efforts have focused on synthesizing safety properties (informally, properties stating that "nothing bad ever happens") from temporal logic specifications. Arafat, *et al.,* have developed an algorithm for generating efficient monitors for a timed variant of linear temporal logic (LTL) [37]. Since LTL's models are traditionally infinite traces [38] while a monitor typically has only a finite execution trace available, many monitoring systems use past-time linear temporal logic (PTLTL) as a specification language. PTLTL employs past-time temporal operators such as "previously" and "always in the past". PTLTL is as expressive as LTL, but more succinct than LTL [39]. Havelund and Rosu proposed a monitor synthesis algorithm for PTLTL [40] that created efficient monitors. This work extends PTLTL, allowing one to express properties such as "function $g()$ is always called from within function $f()$" [41].

The Eagle logic [42] is an attempt to build logics that would be powerful enough to subsume most existing specification logics. Eagle is a first order fixed-point, linear-time temporal logic with a chop operator modeling sequential composition. Although quite expressive, it does not yield efficient monitors. RuleR [43] attempts to address these inefficiencies. A monitor is expressed in RuleR as a collection of "rules" specified in propositional temporal logic, as a state machine, or context free grammar. A trace can be efficiently checked against the rules using a relatively simple algorithm.

The Monitoring and Checking (MaC) toolset is a sophisticated monitoring framework [44–48]. MaC is targeted at soft real-time applications written in Java. A distinguishing feature of the MaC project is that integration and monitoring concerns are divided into separate tasks. Requirements specifications in the form of safety properties are written in the Meta Event Definition Language (MEDL). MEDL is a propositional temporal logic of events and conditions interpreted over a trace of observations of a program execution. The logic has

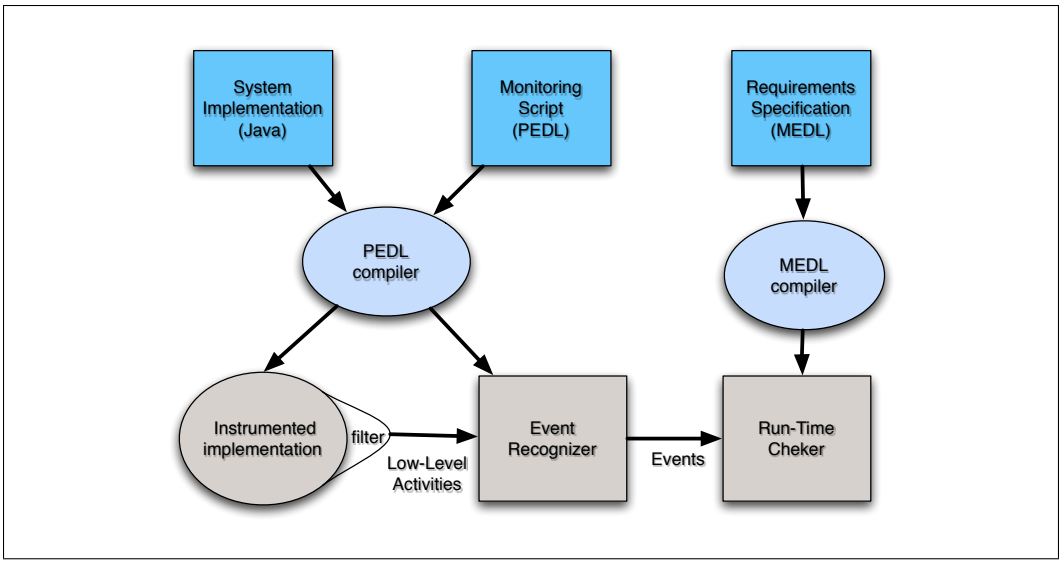

Figure 4. MaC Framework

been extended to handle dynamic indexing of properties [48]. The Primitive Event Definition Language (PEDL) is used to define program events to be monitored and gives a mapping from the program-level events to higher-level events in the abstract specification. A PEDL compiler takes as input the PEDL specification and a Java implementation and produces two artifacts:

- A sensor (called a filter) is generated that is inserted into the code (in this case bytecode) that keeps track of changes to monitored objects [44, 45].
- An event recognizer that processes information sent from the sensor and detects the events being monitored.

A MEDL compiler takes the MEDL specification and produces a verifier that checks that the trace of recorded events as provided by the event recognizer satisfies the safety properties given in the MEDL specification. The architecture is illustrated in Figure 4.

Another monitoring framework, also for Java, is the Java PathExplorer (PaX) [49, 50]. The basic architecture of PaX is similar to MaC in that it separates the integration and verification aspects of generating a monitor. PaX distinguishes itself in two areas. First, in addition to verifying logical properties, PaX performs error-pattern analysis by executing algorithms that identify error-prone programming practices. Secondly, the specification language is not fixed. Instead, users may define their own specification logics in Maude [51].

Monitor Oriented Programming (MOP) [52–54] can be seen as having evolved from PaX and is based on the idea that the specification and implementation together form a system. Users provide specifications in the form of code annotations that may be written in a variety of formalisms including extended regular expressions (ERE), Java modeling language (JML), and several variants of LTL. MOP takes annotated code and generates monitors as plain Java code. MOP monitors may be in-line or out-line. Monitors that enforce properties across many objects or threads are by necessity out-line monitors [54]. MOP includes an efficient mechanism for implementing such out-line monitors. A property that may hold in a number of objects may have an object parameter. MOP employs an efficient indexing scheme to look up instances of monitors when a monitored event occurs.

Much of the recent research on monitoring has focused on Java programs. A notable exception is the requirement monitoring and recovery (RMOR), which focuses on monitoring C programs [55]. In this work, monitors specifying both safety and bounded liveness properties are expressed as state machines observing events recorded in a trace. In a manner similar to PEDL/MEDL, the monitor-refinement specification, mapping high-level events to program-level events, is composed with the specification for the properties to be monitored. RMOR takes the behavioral and the refinement specifications as well as a C program and produces a new C program augmented with code to drive the monitor. The monitors created by RMOR run in constant-space (*i.e.,* no dynamic memory management) and hence are suitable for memory-constrained environments. Techniques from aspect-oriented programming are utilized in generating the monitors.

When a monitor detects that a property is violated, it may raise an alarm to alert the user of a potentially catastrophic problem or take action to compensate for the problem by *steering* the system into a safe mode of operation [47]. We do not specifically address the steering problem in this report.

4.1 Monitoring Distributed Systems

Most research in runtime monitoring has focused on monolithic software as opposed to distributed systems. That said, there has been some research in monitoring distributed systems; Mansouri-Samani and Sloman overview this research area (up to 1992) [56]. Bauer, *et al.*, describe a distributed system monitoring approach for local properties that require only a trace of the execution at the local node [57]. Each node checks that specific safety properties hold and if violated, sends a report to a centralized diagnosis engine that attempts to ascertain the source of the problem and to steer the distributed system to a safe state. The diagnosis engine, being globally situated, collects the verdict from observations of local traces and forms a global view of the system to render a diagnosis of the source of the error.

Bhargavan, *et al.*, [58, 59] focus on monitoring distributed protocols such as TCP. They employ *black-box* monitors that have no knowledge of the internal state of the executing software. Their monitors view traffic on the network, mimic the protocol actions in response to observed input, and compare the results to outputs observed on the network. A domain specific language, Network Event Recognition Language (NERL), is used to specify the events on the network that they wish to monitor and a specialized compiler generates a state machine to monitor for these events [59].

A more decentralized approach is taken by Sen, *et al.*, which grew out of the MOP project described above [60]. A major contribution of this work is the introduction of an epistemic temporal logic for distributed knowledge that allows the specification to reference different nodes. The task of monitoring a distributed computation is distributed among the nodes performing the computation with each node monitoring properties that refer to the state of other nodes. In order for a monitor at one node to access the state at another node, state vectors are passed around in a manner inspired by vector clocks. Since a node cannot know the current state of another node, the epistemic logic references the last known state of that node. When monitoring local properties, the monitors are similar to those in MOP. The communication overhead may be a concern in some application domains.

Chandra and Toueg [61] propose to extend the asynchronous model of computation (in which there may be unbounded delays on communication[2]) by adding distributed failure detectors, which are located at each node. Each detector maintains a set of nodes it suspects to have crashed. The detectors are unreliable in that they may erroneously add good nodes to the list of accused nodes and the list of suspects may differ at different nodes. At each step of computation, the failure detectors pass a list of processes currently suspected of

[2]The asynchronous model of computation is not generally applicable to hard real-time systems.

crashing. If the detectors satisfy certain conditions, the consensus problem can be solved. Several attempts have been made to extend this work [62–66]. Yet many of these attempts only handle what we would classify as benign faults rather than Byzantine faults. Kihlstrom *et al.* [65] uses statistical techniques and report that they can detect Byzantine faults.

4.2 Monitoring Hard Real-Time Systems

Real-time systems are a target for monitoring since they their temporal constraints make them hard to test and debug. Generally, research has targeted soft real-time systems (like the MaC work discussed earlier) or off-line monitoring. Early research in off-line monitoring was to debug scheduling errors [30–32,67]. This line of research usually focused on instrumenting applications with sensors to simply capture a time stamped trace of system calls, interrupts, context switches, and variables for the purposes of replay and analysis.

An early application of online monitoring to real-time systems focused on verifying that timing constraints are satisfied [67]. Telecom switches have real-time constraints, and approaches to monitoring these are investigated by Savor and Seviora [68]. The authors introduce a notation for specifying time intervals in the Specification and Description Language (SDL). Algorithms are developed for processing different interleavings of signals as well as to monitor that the specified signals occur within their designated timing intervals.

Many variants of temporal logics have been developed for specifying properties of real-time systems. A survey of these logics is given in Alur and Henzinger [69]. Recently, several efforts in the monitoring community have focused on monitoring metric temporal logic (MTL) specifications of real-time systems [70]. MTL can be used to reason about quantitative properties over time. For instance, one can specify the time elapsed between two events.

Before discussing monitoring based on MTL specifications, some of the distinguishing features of MTL are introduced. The semantics for an MTL expression is defined in terms of a timed state sequence $\rho(\sigma, \tau)$ composed of a finite sequence of states $\sigma = \sigma_1, \ldots, \sigma_n$ and a finite sequence of real numbers $\tau = \tau_1, \ldots, \tau_n$ of equal length. The pair of sequences are interpreted to read at time τ_i, the system is observed to be in state σ_i. MTL quantifiers can be defined over intervals. Let I be one of the following:

- An interval on the non-negative reals, the left endpoint of which is a natural number and the right endpoint of which is either a natural number or ∞.
- A congruence $=_d c$, for integers $d \geq 2$ and $c \geq 0$. The expression $y \in =_d c$ denotes $y = c \bmod d$.

The semantics for the temporal logic next operator is given as $(\rho, i) \models \bigcirc_I \phi$ iff $i < \mid \rho \mid$, $(\rho, i+1) \models \phi$ and $\tau_{i+1} \in \tau_i + I$. From the definition we see $(\rho \models \bigcirc_{[m,n]} \text{ true})$ holds if $\tau_{i+1} - \tau_i \in [m, n]$ and $(\rho \models \bigcirc_{=_d c} \mathit{true})$ holds if $\tau_{i+1} = c \bmod d$. The semantics for the until operator is defined as $\rho \models \phi_1 \; \mathcal{U} \; \phi_2$ iff $(\rho, j) \models \phi_2$ for some $j \geq i$ with $\tau_j \in \tau_i - I$ and $(\rho, k) \models \phi_1$ for all $i \leq k < j$. Thus $\phi \; \mathcal{U}_{[4,8]} \; \psi$ says that ψ holds between time units 4 and 8 and ϕ holds until then. The semantics for $\Box_I \; \phi$ and $\diamond_I \; \phi$ are similarly defined. Thati and Rosu [71] gave an algorithm for monitoring MTL formula and showed that even for a simple fragment of MTL, the algorithm is exponential in time and space. Drusinsky [72] uses a restricted fragment of MTL and represents MTL formulas as alternating finite automata to substantially reduce the space requirements. More recently, Basin, *et al.,* [73] propose a monitoring algorithm using finite structures to represent infinite structures that yields an online monitoring algorithm for a significant fragment of MTL that is polynomially bound in the size of the space consumed.

The synchronous model of computation is used in computer hardware as well as hard real-time systems such as avionics. The synchronous paradigm follows a *stream* model

of computation. The stream values at an index are interpreted as being computed synchronously, and communication between streams is instantaneous. Synchronous languages like Lustre [74, 75] have been used by the embedded systems community for many years.

Although the synchronous paradigm is in common use in real-time systems, we know of only one effort focusing on specifying monitors for synchronous systems. LOLA [76] is a specification notation and framework for defining monitors for synchronous systems. LOLA is expressive enough to specify both correctness properties and statistical measures used to profile the systems producing input streams. A LOLA specification describes the computation of an output stream from a given set of input streams. A LOLA specification is a set of equations over typed stream variables of the form:

$$\begin{aligned} s_1 &= e_1(t_1, \ldots, t_m, s_1, \ldots, s_n) \\ \vdots &\quad \vdots \\ s_n &= e_n(t_1, \ldots, t_m, s_1, \ldots, s_n) \end{aligned}$$

where $t_1, \ldots, t_m$ is the set of input stream variables (also called independent variables) and $s_1, \ldots, s_n$ is the set of output stream variables (also called dependent variables) and $e_1, \ldots, e_n$ are stream expressions over the stream variables. Stream expressions are built from constants, stream variables, functions over stream expressions, boolean stream expressions (if-then-else), and expressions of the form $e[i, c]$, referring to offset i of stream expressions e, where c is a constant that acts as the default if the offset lands before the beginning of the stream or after the end of the stream. Four examples follow:

$$\begin{aligned} s_1 &= t_1 \vee t_2 \leq 1 \\ s_2 &= ((t_2)^2 + 7) \bmod 15 \\ s_3 &= \mathtt{ite}(s_1, s_2, s_2 + 1) \\ s_4 &= t_1[+1, false] \end{aligned}$$

Stream equations "lift" operators to the stream level, so s_1 says that at position i of the stream, $t_1(i) \vee t_2(i) \leq 1$, where t_1 is a stream of binary values and t_2 is a stream of integers. The next specification simply takes the integer stream t_2 and computes a new integer. The steam expression defining s_3 references the previous two expressions as inputs. If $s_1(i)$ is true, then $s_3(i)$ takes on the value $s_2(i)$, otherwise it takes the value $s_2(i) + 1$. The final example illustrates the offset operator. Here, each position i of stream s_4 corresponds to the values at position $i + 1$ of the input stream t_1 and to false if $i + 1$ is beyond the current end of the t_1 stream. The authors present a monitoring algorithm for converting the equational LOLA specifications into efficient executable monitors.

Mok and Liu developed the language Real-Time Logic (RTL) for expressing real-time properties for monitoring. A timing constraint specifies the minimum/maximum separation between a pair of events. A deadline constraint on an event E_1 and an event E_2 is violated if the event E_2 does not occur within the specified interval after event E_1. Given an event e and a natural number i, the occurrence time of the i-th instance of e is denoted as $@(e, i)$. Timing constraints in RTL often take the following form:

$$\begin{aligned} @(a, i) + 20 &\geq @(b, i) \geq @(a, i) + 5 \\ @(b, i) + 10 &\geq @(c, i) \geq @(b, i) + 2 \end{aligned}$$

which says that event b should not happen earlier than five time units after event a and no later than twenty time units after event a and that event c must not happen earlier than two time units after b or later than ten time units after b. To monitor RTL constraints, Mok *et al.* employ sensors that send timestamped events to the monitor, and an algorithm computes the satisfiability of the constraint [77, 78].

Any runtime monitoring of hard real-time systems cannot interfere with the execution of the application and particularly its timing constraints. Consequently, out-line monitors (*i.e.*, monitors executing as a separate process from the one being monitored) are preferred. Pellizzoni *et al.* [35, 36] have constructed monitors in FPGAs to verify properties of a PCI bus. The monitors observe data transfers on the bus and can verify if safety properties are satisfied.

5 Integrated Vehicle Health Management and Monitoring

This section places monitoring within the context of the larger goals of an integrated vehicle health management (IVHM) approach to system reliability. As overviewed by Ofsthun, IVHM is a program to increase the accuracy, adaptability, and management of subsystems in automobiles, aircraft, and space vehicles [79]. The goals of IVHM are to increase safety while decreasing maintenance costs (for example, by reducing false positives in built-in testing). IVHM broadly applies to the subsystems of a vehicle including propulsion systems, actuators, electrical systems, structural integrity, communication systems, and the focus here, avionics. Ofsthun notes that recent advances in IVHM add prognostics (*i.e.*, predictive diagnostics), which allow systems to determine the life or time span of proper operation of a component. In the case of components that tend to degrade due to wear and tear, predictive systems identify degraded performance and identify when preventive maintenance is needed.

IVHM research in aviation and space systems has typically focused on physical components such as actuators [80], engine components, and structures. Recent advances in sensors, signal processing, and computing allow accurate measurements of such components to be determined in real-time. Advances in engineering science have yielded improved models of the physical aircraft itself. IVHM systems monitor the physical system and compare it against the mathematical model of the physical system. As noted by a National Research Council report, this allows the "aircraft to trace back the system anomalies through a multitude of discrete state and mode changes to isolate aberrant behavior" [81]. These diagnostics allow the aircraft to detect unseen cracks in moving parts. Such systems can lower maintenance costs as well as improve safety.

Such IVHM measures make sense for systems that degrade over time, like materials or hardware. But software does not degrade over time, so it should be reliable on its first use, and it should remain reliable over time. Evidence for the reliability of critical software comes principally in one of three forms: certification, testing, and verification.

The Federal Aviation Authority (FAA) codifies high-assurance software development processes in standards such as DO-178B [82]. Currently, these standards rely heavily on a combination of rigorously-documented development processes and software testing. While certification provides evidence that good practices are followed in software development, it does not guarantee correctness. Recall from Section 2.2 the incident involving a Malaysian Air Boeing 777 where a software problem was undetected during DO-178B certification. Although we were not able to determine if the system was subject to level A certification, it nevertheless is disconcerting that the problem went undetected in certification. More or better testing is not a feasible solution for demonstrating reliability, at least for ultra-critical real-time software [3]. Consequently, The National Academies advocate the use of formal methods to augment testing using mathematical proof [83].

However, such proofs are not over the realizations themselves but over models of the realizations and the complete verification of a complex implementation remains elusive. Assuming that formal methods are applied to an abstraction of the system, some verification that the implementation complies to the specification is needed, especially when the specification has built-in assumptions about the implementation. Of course, as a model's fidelity to an implementation increases, so does the cost of formal verification.

Because of the respective disadvantages of certification, testing, and formal methods, a fourth form of evidence is being explored: the idea of runtime monitoring. Runtime monitoring can be seen as an aspect of IVHM [84–86] that can potentially detect, diagnose, and correct faults at runtime. Indeed, Rushby argues that runtime monitoring can be considered as a form of "runtime certification" [84, 85]. As Rushby notes, one motivation for runtime certification is that one source of software errors is a runtime violation of the assumptions

under which the software is designed (and certified). Runtime monitoring frameworks can refine formal specifications of requirements into software monitors. Furthermore, Rushby argues that monitors have the greatest potential payoff in monitoring system-level properties rather than unit-level requirements, which DO-178B practices catch well.

6 Monitor Architectures for Distributed Real-Time Systems

In this section, we explore various abstract monitoring architectures for fault-tolerant real-time systems, focusing on their tradeoffs. First, some theoretical aspects of monitoring distributed systems which frame our discussion are presented. Next, requirements relating to functionality, schedulability, and reliability relating to monitor architectures are considered. Finally, we present and compare three abstract monitor architectures for distributed systems.

6.1 The "Theory" of Distributed Monitoring

Recall that this paper examines the applicability of runtime monitoring to real-time distributed systems. Consequently, we wish to understand the limits of the ability of monitors to detect faults in these systems. While it may seem that monitoring distributed real-time systems requires a new theory, we argue that it does not. Rather, theoretical results in distributed systems subsume monitoring.

Our reasoning is based on the following uncontroversial thesis we propose:

> *Distributed-System Monitoring Thesis*: Monitors for a distributed system are other processes in the distributed system.

The thesis implies that there is no omniscient view of a distributed system. A monitor, whether it be a user or a process, somehow gathers information from the other processes in a distributed system, just like those other processes gather information from each other for distributed computation. A monitor may be privileged—for example, it might have a more accurate physical clock or it might have communication channels from every other node in the system—but these are differences of degree and not of kind: in principle, other processes could be augmented with more accurate clocks or more channels.

Some consequences follow from the thesis. The first consequence we have already mentioned: *Any theoretical result on distributed systems applies to distributed-system monitors.* The theory of distributed systems is well-developed, and has established fundamental limits on synchronization, fault-tolerance, fault-detection, and observation [15, 87, 88]. Any theoretical limitation on distributed systems applies to a distributed-system monitor. Therefore, let us review known theoretical limitations and consequences for monitoring distributed systems regarding synchronization and faults.

Synchronization and faults make distributed computing complex. Let us ignore the problem of faults for a moment and focus on synchronization. If a distributed system can synchronize, then computation can be abstracted as a state machine in which individual processors act in lock-step. Mechanisms have been developed to ensure that individual nodes agree on time, at two levels of abstraction [87, 88]. Simply agreeing on the order of events in the system is called *logical clock* synchronization. If processes must agree on the real time (or "wall-clock time") that events occurred at, then processes maintain and synchronize *physical clocks*. (Physical clocks can be implemented, for example, using crystal-controlled clocks common in digital hardware devices.)

The presence of faults complicates matters. A monitor detects faults in the system under observation (SUO). However, the monitor itself may suffer faults, and if it does, it may incorrectly attribute the fault to the SUO or take some other unanticipated action. A monitor must therefore be shown not to degrade the reliability of the SUO. For example, a monitor should be demonstrated not to become a "babbling monitor," continuously signaling an error even if there are no faults in the SUO. We expand on this issue in Section 6.3.

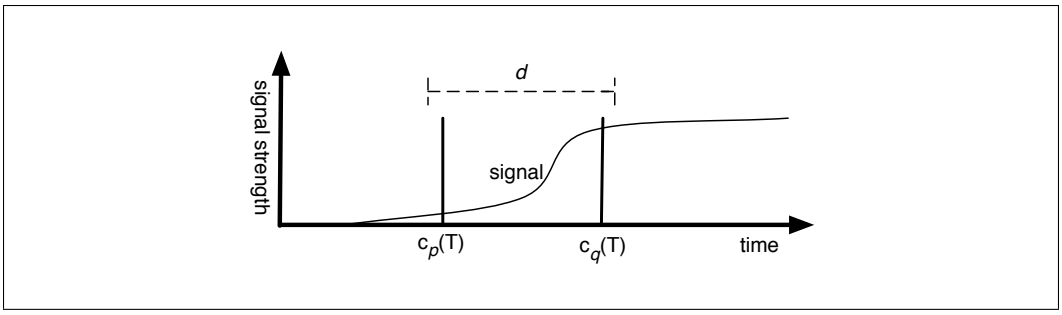

Figure 5. A Byzantine Interpretation of a Signal

But anomalous behavior is possible even if no process in the distributed system—including the monitor—is faulty. In particular, consider a distributed system which maintains synchronized clocks (as is the case in ultra-reliable fault-tolerant architectures [19]). Synchronized physical clocks do not maintain perfect synchrony. Because of the inherent imprecision of physical clocks, they can only be kept synchronized within some small real-time value d. The inherent asynchrony results in the potential for Byzantine behavior, which occurs when a single process appears to broadcast two different signals to two recipients [1,21]. As an example, consider Figure 5, which depicts two processes, p and q. For p, the function $c_p(T)$ takes a discrete clock time T from its own clock and returns the earliest real time when it reaches that clock time, and similarly for q. Note that these real times fall within d of each other, so we consider them to be synchronized. Suppose further they both monitor the same signal on a wire. For the signal to register with a process, it must be above some threshold. In the figure, p does not detect the signal while q does. Thus, q accuses the process transmitting the signal of being faulty while p does not, even though they are both synchronized.

As Fidge notes [88], such behaviors introduce nondeterminism: the same trace of actions by individual processes may lead to different global behaviors. Consequently, if a monitor observes violations of a property, and that property depends on some observable event meeting a real-time deadline, then *the monitor will return both false positives and false negatives*. Within some delta of real-time, it is indeterminate whether the property is satisfied.

6.2 Whither the Software-Hardware Distinction?

Monitoring approaches generally fall into ones for hardware and ones for software. The approaches we have surveyed in Section 4 primarily focus on software. However, for fault-tolerant real-time systems, the distinction is less useful from both the real-time and fault-tolerance aspects.

First, real-time guarantees are dependent upon the behavior of both software and hardware. Attempting to decompose a monitor to observe just the real-time behavior of the software or hardware in isolation is not practical (and perhaps not even feasible). Rather, if a monitor detects a missed deadline, a probabilistic analysis can be used to determine whether the failure is due to a systematic (*i.e.*, software) fault or a random (*i.e.*, hardware) fault. For example, if the deadline is missed by a large margin, then a logic error might be considered to be more probable than a random hardware fault. A similar conclusion might be reached if deadlines are missed too often. (Also see Section 7.1.2 for a more detailed discussion of these issues.)

Like real-time behavior, faults originate in either software or hardware and cannot gen-

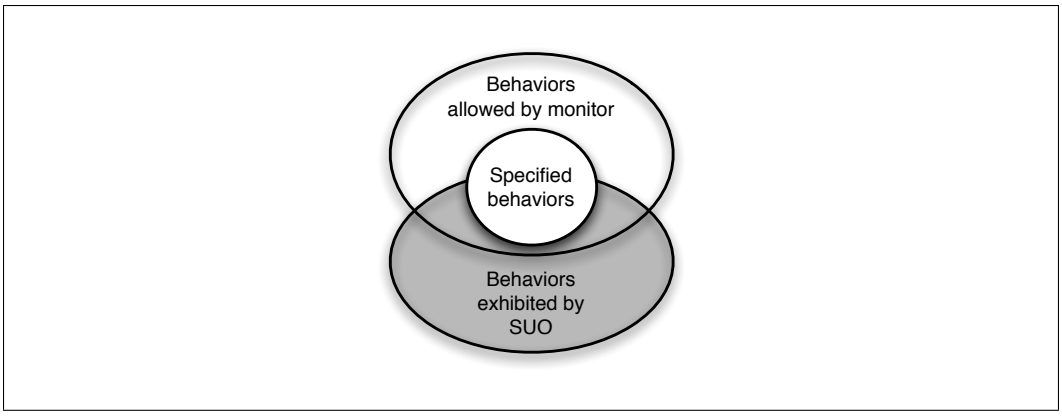

Figure 6. Relationship Between Behaviors Specified, Allowed, and Exhibited

erally be distinguished on the basis of a single observation. Rather, probabilistic methods can be used, based on knowledge of the environment and of the failure rate of the hardware.

6.3 Monitor Architecture Requirements

A *monitor architecture* is the integration of one or more monitors with the system under observation (SUO). The architecture includes all the additional hardware and interconnects required to carry out the monitoring. Monitoring fault-tolerant real-time systems places special constraints on potential architecture. We propose the following to be architectural constraints that must be met:

1. *Functionality*: the monitor does not change the functionality of the SUO unless the SUO violates its specification.

2. *Schedulability*: the monitor architecture does not cause the SUO to violate its hard real-time guarantees, unless the SUO violates its specification.

3. *Reliability*: the reliability of the SUO in the context of the monitor architecture is greater or equal to the reliability of the SUO alone.

4. *Certifiability*: the monitor architecture does not unduly require modifications to the source code or object code of the SUO.

If these criteria are met, we say that the monitor *benefits* the SUO. The functionality requirement is characteristic of monitoring systems and simply ensures that the monitor behaves like a monitor and does not modify the nominal functionality of the SUO. However, the monitor may modify the behavior of the SUO if the SUO is detected to violate its specification. We diagram this relationship in Figure 6; any portion of the behaviors exhibited by the SUO that are not allowed by the monitor may be modified by the monitor.

The schedulability requirement ensures that the monitor does not interfere with the timeliness of the services provided by the SUO. Timeliness violations can occur both due to additional constraints on the processes and the channels between processes that monitoring places upon the system. For example, software in the processes might be instrumented with additional code for monitoring purposes, which can affect the timeliness of services. Likewise, distributed monitors that communicate over the same channels that are used by the SUO can decrease the bandwidth available for the messages of the SUO.

When a monitor detects a runtime error, it may *steer* the SUO into a good state (although in general, steering is a separate concern from monitoring). The schedulability requirement should not be taken to imply that the SUO's schedule cannot be disrupted or modified during the steering phase.

The reliability criterion is somewhat subtle. The requirement is intuitive insofar as a monitor architecture should not decrease the reliability of the SUO. However, note that it is allowable for the reliability of the SUO in the context of the monitor architecture to be equal to the reliability of the SUO alone. For example, if the monitor only collects data for off-line analysis, then the monitor's reliability should not adversely affect the reliability of the SUO—even if the monitor is quite unreliable. Furthermore, this criterion does not mandate that faults in the monitor architecture must not propagate to a failure in the SUO. That is, suppose the reliability of a system $\mathcal{S}$ is $1 - 10^{-X}$ per hour of operation. Now suppose that $\mathcal{S}$ is composed with a monitor $\mathcal{M}$, resulting in $\mathcal{S} \cup \mathcal{M}$. Suppose in the composition, the monitor takes no corrective action, but faults from $\mathcal{M}$ can propagate to $\mathcal{S}$, and that the reliability of the composition is $1 - 10^{-Y}$ per hour, where $Y < X$; that is, the composition is less reliable than $\mathcal{S}$ alone. Now suppose that in the composition, the monitor does take corrective action if faults are detected in $\mathcal{S}$, and suppose that under this scenario, the reliability of the composition is $1 - 10^{-Z}$ per hour, where $Z > X$. Then the reliability criterion is satisfied, even though faults can propagate to the SUO from the monitor. Additionally, an implementation might refine the property to show, for example, that the monitor degrades gracefully in the presence of faults.

When we speak of the reliability of the SUO, we must keep in mind the difference between its nominal reliability, in which we suppose the only source of faults are environment-induced hardware failures, and its actual reliability, which takes into account both hardware failures as well as systematic design errors (in either hardware or software). Design errors may dramatically reduce the reliability of an ultra-reliable system [3]. If it is hypothesized that the reliability of the SUO is below its required reliability due to unknown design errors, then it may be quite easy for a monitor architecture to satisfy the reliability requirement. Indeed, each scenario described in the case-studies presented in Section 2 resulted from systematic errors rather than environment-induced random errors.

Finally, we propose the constraint of *certifiability*. If an SUO is intended to be evaluated under a safety-critical certification standard (e.g., DO-178B [82]), introducing a monitor should not make a re-evaluation of the SUO in the context of the monitor overly difficult. If the monitor satisfies the functionality, schedulability, and reliability constraints, then a certified SUO should be certifiable in the context of the monitor, but those constraints do not address the kind and level of evidence required to demonstrate that they are satisfied. An evaluation intended to provide this evidence could range from simply composing the independent evaluations of the SUO and the monitor to requiring a complete re-evaluation of the SUO in the context of the monitor. In practice, the evidence required will fall somewhere within that range: the certifiability constraint is one of degree. An "upper bound" is that adding a monitor would require fully repeating the verification and validation of the SUO in the context of the monitor.

We postulate that a monitor framework that requires rewriting or modifying the source code or object code of the SUO—like most monitoring frameworks do—generally makes re-evaluation too onerous, since it likely requires full re-evaluation of the SUO. Therefore, the constraint of certifiability is taken to mean that the monitor does not introduce "significant" source (or object) code modifications to the SUO.

A proven means to ease the difficulty of demonstrating that one software system does not inappropriately interact with another is to make the argument at the hardware-architecture level. For example, time-triggered architectures make this guarantee in the temporal dimension [24, 89]. The monitoring architectures described in Section 6.4 are intended to describe the hardware-architecture level approaches to monitoring distributed systems.

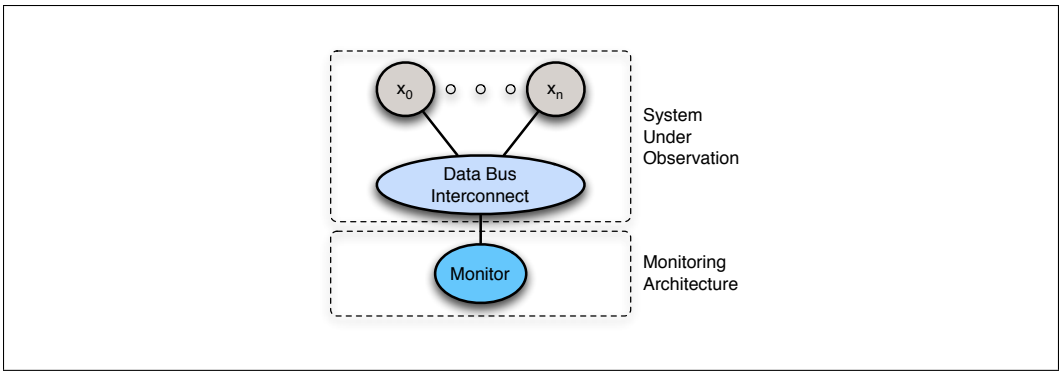

Figure 7. A Monitor Watching a Shared Bus

Finally, Rushby recently argues that runtime monitoring can provide a framework for *runtime certification*, in which some evidence of correctness is deferred to runtime [84] when monitors collect the evidence. The novel notion of runtime certification depends on a constraint like the certifiability constraint to hold as well as the monitors themselves being fault-free [90].

6.4 Monitor Architectures

We propose three abstract monitor architectures and discuss them in the context of the requirements stated above. We keep our presentation at a conceptual level given that we are primarily proposing architectures that may be investigated in future research. In the figures, we show a distributed SUO and an associated monitoring architecture. The SUO architecture is made up of a set of distributed processes x_0, x_1, ..., x_n together with an interconnect. We represent the interconnect abstractly. For example, the interconnect could be a simple serial bus, or it could be a graph of interconnects, such as a ring or star, between processes [19]. Recall from Section 3.1 that we are considering monitors for distributed systems containing a fixed set of nodes and a fixed set of interconnects between nodes that are modified only by faults in the systems.

Our presentation begins with the simplest architecture employing a single monitor and requiring no additional hardware, and proceeds in order of increasing complexity.

6.4.1 Bus-Monitor Architecture

The first architecture under consideration is depicted in Figure 7, where the monitor M observes traffic on the data bus of the SUO. In this architecture, the monitor receives messages over the bus just like any other process in the system. However, the monitor would likely be a silent participant that does additional error checks on in-bound messages or verify that protocol communication patterns are being followed. Only if it detects a catastrophic fault would it send messages to the other processes through the bus. As discussed in 4.2, BusMoP [35,36] adapts this very architecture to verify bus protocols. In the case of BusMoP, the monitor is implemented in an FPGA that is interposed between the peripheral hardware and the bus allowing it to sniff all activities on the bus. Bhargavan *et al.* similarly monitor activity over the wire to capture TCP packets [58, 59].

As compared to the other architectures being considered, the bus-monitor architecture requires the least additional hardware for monitoring. It is arguably the simplest monitoring architecture, although it must be ensured that the monitor does not become faulty and

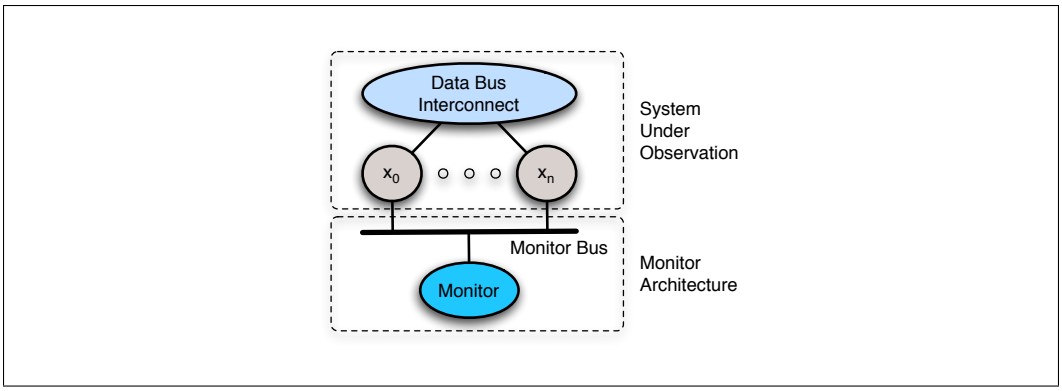

Figure 8. Single Monitor on a Dedicated Bus

consume unallocated bandwidth on the data-bus (provided it is not restricted from sending messages on the bus). The class of faults this monitoring architecture is able to capture, however, is limited: the monitor can only infer the health of a process from the messages processes passes over the bus. In particular, such an architecture is not amenable to performing fault detection beyond the level that the SUO itself could perform if it were so designed. This architecture, like BusMoP, is motivated by the use of commercial-off-the-shelf hardware that is not designed to be fault-tolerant.

6.4.2 Single Process-Monitor Architecture

Having observed that the SUO and monitor sharing a common bus can lead to problems, a natural alternative is for each to have its own dedicated bus, as depicted in Figure 8. Each process in the SUO is attached to the data bus as well as a single monitor process. In this architecture, each process is instrumented to send data to the monitor over the monitor bus, and the single monitor can compare the incoming data. As necessary, the monitor may signal the processes if a fault is detected. It would be relatively straightforward to adapt existing monitoring systems such as MaC [44–48] to this architecture. Rather than a single process sending data to single monitor, multiple processes send state information to a single monitor via a dedicated bus.

Let us briefly analyze the abstract architecture with respect to the requirements listed in Section 6.3. In this architecture, the modification required to the SUO is ostensibly minor; each process needs only to be instrumented to broadcast select messages that it receives or sends over another bus, helping to ensure that the functionality of the SUO is not interfered with. On the other hand, all state information must be explicitly passed to the monitors via dedicated messages over the monitor's bus. By using a separate bus for monitoring messages they are not passed on the SUO interconnects, so there is less chance the monitor causes the SUO to violate its timeliness guarantees (additionally, faults in the data bus are independent of the monitor's bus). Whether this monitor architecture satisfies the reliability requirement depends on the specific implementation. If the monitor is simple compared to the processes in the SUO, the probability of design faults is lower. Furthermore, if the monitor architecture can be shown to only exhibit fail-silent (or other benign) failures, then its failure will not adversely affect the SUO.

In some cases, it might be possible to conceptually combine an architecture in which a monitor has a dedicated bus with the previous architecture described, in which a monitor watches a shared bus. For example, if the interconnect is TTTech's TTEthernet [91], monitor messages might be sent between distributed monitors over the network as conventional

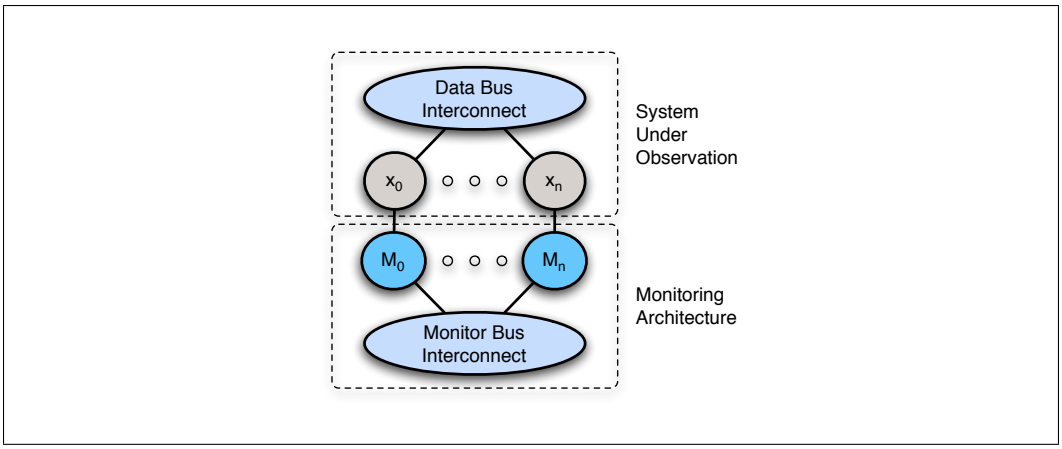

Figure 9. Distributed Monitors on a Dedicated Interconnect

Ethernet traffic, which is guaranteed by the TTEthernet not to interfere with safety-critical messages sent over the same network.

6.4.3 Distributed Process-Monitor Architecture

We show an architecture in Figure 9 where there are distributed monitors M_0, M_1, ..., M_n corresponding to each process x_0, x_1, ..., x_n in the SUO. Monitor M_i may be implemented on the same hardware as process x_i, or it could be in its own fault-containment unit; the former is cheaper (in terms of hardware resources), but the latter provides less chance of a monitor failing due to the same fault as its associated process. Note that the monitor's interconnect may be fault-tolerant even if the SUO's interconnect is not. Thus, you can "clamp on" fault tolerance. In this architecture, online distributive monitors need to communicate with each other in order to reach agreement on diagnoses. Thus, we abstractly represent some interconnect between the monitors. This is similar to the architectures implemented in [60] and [57], but in those cases the SUO and the monitors use a shared bus, which given the traffic between monitors may compromise hard real-time deadlines.

As compared to the single process-monitor architecture in Figure 8, this architecture has the advantage that it supports a potentially more reliable monitor since the monitors are distributed. So even if one or more M_i's fail, the monitor may assist the system.

A distributed monitor also has an additional benefit in that an individual monitor M_i can serve as a guardian to its corresponding process x_i. In the single process-monitor architecture, a "babbling" process could prevent other processes from sending messages to the monitor or prevent the monitor from signaling the processes in the case of a detected error.

On the other hand, a distributed monitor is more complicated, and that complexity may lead to less reliability in the monitor. Furthermore, the distributed process-monitor architecture may be nearly as expensive in terms of processes and interconnects as the SUO itself. A distributed architecture may not be feasible if the monitor has stringent cost, size, weight, or energy consumption constraints.

7 Monitoring Properties: What the Watchmen Watch

Thus far, we have discussed fault-tolerant distributed systems, monitoring in general, and potential monitor architectures. Our discussion leaves open the question of *what* to monitor in fault-tolerant real-time systems. Here, we attempt to classify relevant general properties and describe potential approaches to monitoring these classes of systems.

Again, we remind the reader that fault-tolerant real-time systems used in safety-critical contexts are typically engineered according to best-practices and are extensively tested. Thus, the kinds of properties described in this section are often taken into account in the architectural design. That said, the case studies in Section 2 illustrate situations in which deployed safety-critical systems have failed due to incorrect designs or architectures that make invalid assumptions about the environment. These systems are good candidates for monitoring.

In general, monitors can observe arbitrary *safety* properties in distributed real-time systems, up to the limits described in Section 6.1 and by Fidge [88]. Informally, safety properties state that "something bad does not happen." This is opposed to *liveness* properties, stating that "something good eventually happens" [92]. Liveness properties include important properties (*e.g.,* program termination), but in general, they cannot be monitored at runtime—intuitively, this is because there is no bound on when to declare the property to be falsified. However, as Rushby notes for real-time systems, all properties are effectively safety properties, since showing that "ϕ eventually happens *by a finite deadline*" is equivalent to showing that "not ϕ does not happen by the deadline" [84]. The classes of properties described in this section are safety properties.

This section is organized as follows. First, we consider using online monitoring to strengthen the fault model under which a system was originally designed—in particular, for tolerating Byzantine faults a system was not originally designed to tolerate. Online monitoring to strengthen point-to-point error detection (*e.g.,* cyclic redundancy checks) is then considered. Finally, we consider its use to check the correctness of software that manages fault-tolerance in a system.

For each class of properties, the applicability of the architectures proposed in Section 6 is considered.

7.1 Monitoring Fault-Model Violations

Fault-tolerant systems are designed to withstand a particular maximum fault assumption (MFA)—recall the review of fault-tolerance concepts from Section 3.1. A designer's formulation of a system's MFA is based on a variety of factors including knowledge about the operational environment, reliability of the individual components, expected duration a system is to be fielded, number of fielded systems, cost, and so on. Many of these factors are under-specified at design time (for example, systems are used well beyond their expected retirement dates or for unanticipated functionality). Because the MFA is chosen based on incomplete data, it may be inappropriate, and in particular, too weak. Furthermore, a system's MFA is built on the assumption that there are no systematic software faults, but software faults can dramatically reduce the hypothesized reliability of a system [3].

In a fault-tolerant system, the relationship between software and hardware is subtle. We can characterize two kinds of software in such a system: the software that implements a system's services, and the software that manages the system's response to faults. The behaviors of fault-management software are determined by the presence of random hardware failures caused by the environment—indeed, the software can be thought of as a reactive system responding to faults. Thus, to know at runtime whether fault-management software is behaving correctly, one must also know the status of the hardware failures in the system. So it would seem that for a monitor to determine the health of the fault-management

software, it must know which hardware faults are present. But detecting hardware faults is precisely the job of the fault-management software itself!

In Section 2, we saw concrete examples of system failures that were the result of an insufficient MFA (*i.e.*, unexpected faults). As mentioned before, Byzantine or asymmetric faults are particularly nefarious and often unanticipated. Consider the case of the aborted space shuttle launch described in Section 2.1. The failure of a single diode produced asymmetric faults that the system design failed to anticipate. It appears that the system was designed to satisfy a MFA that did not accommodate asymmetric faults. It is conceivable (we are speculating here) that designers chose a fault-model that excludes asymmetric faults because the designers judged that the probability of their occurrence to be so small that the additional complexity required in a system design intended to detect and mask such events was unwarranted. Indeed, in some cases, designs that handle rare faults can increase the probability that less rare faults do occur [93]. The intuition is that fault-tolerance requires redundancy, and redundancy means more hardware. The more hardware there is, the more likely some component in the system will fail.[3] However, it has also been argued that Byzantine faults, while rare, are much more probable than generally believed [1].

The fundamental property that establishes the absence of a Byzantine fault is a consensus property [21]. Consensus tells us that each receiver of a broadcasted message obtains the same value. If we presume that a system is not designed to tolerate Byzantine faults at the outset to ensure a consensus property holds, we are inclined to monitor consensus properties at runtime.

Below, we consider the use of the monitoring architectures we have described both to ensure consensus of distributed values and then to ensure timing assumptions made by the system are met.

7.1.1 Monitoring Consensus

To monitor consensus at runtime, the values receivers obtain must be compared. Consider *exact agreement*, where each receiver should obtain the exact same value, rather than *approximate agreement*, where each receiver should obtain data that agrees within some small δ, such as exchanged clock values [23]—we treat approximate agreement in the following section.

Consider a simple single-broadcast system in which one node, a transmitter, broadcasts a message to other nodes, the receivers. The transmitter, a receiver, or the interconnect may suffer a fault that causes a receiver to compute a received value that differs from those received by the other receivers. We will refer to the interconnect as the "bus" although it may be a virtual bus as described in Section 6.4.

In a Byzantine-agreement algorithm [21, 23], receivers exchange messages to compare their received values and ensure consensus. But if the system is not instrumented originally to be resilient to Byzantine faults using such algorithms, can any of the architectures proposed in Section 6 be adapted to verify consensus? Consider the three monitor architectures we have proposed:

- The bus-monitor architecture illustrated in Figure 7 of Section 6.4.1 will not reliably detect consensus violations by monitoring the broadcast messages sent on the bus, since the monitor acts just like another receiver.
- The process-monitor architecture depicted in Figure 8 of Section 6.4.2 can be applied to monitor this property as follows. Each receiver is instrumented to send a copy of (or a checksum of) its received message to the monitor. The monitor must ascertain whether

[3] Random hardware failure is statistically-independent: if the failure rate for a component is 10^{-5} per hour and there are 10 components, the failure rate is 10^{-4} overall, assuming each component fails independently.

or not the messages belong to the same round of communication. If so, the monitor then compares the values received to determine if they are the same. The solution is straightforward, but for the monitor to accurately make a diagnosis of consensus, neither the monitor itself nor the interconnects from the nodes to the monitor may suffer faults. In either case, false negatives could result—*i.e.*, the monitor mistakenly reports that consensus is violated, since even if the receivers obtain the same values from the transmitter, the monitor might receive or compute different values for each receiver. The chances of a false positive are arguably smaller. A false positive requires a scenario in which either (1) the monitor obtains the correct value from a receiver despite the receiver obtaining an incorrect value, or (2) the monitor itself confirms consensus even when the monitor receives inconsistent values.

- Finally, consider the distributed monitor architecture depicted in Figure 9 of Section 6.4.3, where each node has a dedicated monitor that samples receivers to determine the values they receive. For the monitors to determine whether consensus is reached, they must exchange values with one another. If our fault assumption allows for the monitors to exhibit Byzantine faults, then they must execute a Byzantine fault-tolerant algorithm to exchange values to determine whether consensus is reached, just as the original nodes would.

If our fault model allows for the possibility that monitors exhibit Byzantine faults, then the complexity introduced in order to handle Byzantine faults is relegated to the monitor architecture. Relegating the complexity may be redundant (*i.e.*, the SUO should be made fault-tolerant in the first place), but if the SUO is a legacy system, it may be possible to compositionally add fault tolerance to the system through the use of monitors.

7.1.2 Monitoring Timing Assumptions

One architectural approach to ensure hard real-time deadlines are met is the *time-triggered* approach [89]. In time-triggered systems, events are driven by a globally-known predetermined schedule, even in the presence of faults. Because the execution of a time-triggered system is driven by the passage of time, it is imperative that the distributed nodes in the system are periodically synchronized and that they agree with the real (*i.e.*, "wall clock") time. Time-triggered architectures are particularly prominent in fault-tolerant data buses including, for example, FlexRay (in some configurations), SAFEbus^{TM}, SPIDER, and TTA [19].

A sophisticated time-triggered data bus is the "backbone" that provides services such as communication, time reference, and fault management to various avionics functions. As such, the correctness of the bus is essential, and in particular, specific timing properties must hold to ensure the bus delivers the time-triggered abstraction.

Formulating and verifying the timing properties of a particular implementation is not easy (consider the appendices analyzing the timing properties of NASA's SPIDER [24]). To simplify and generalize matters, Rushby presents a mathematical theory called the *time-triggered model* [94] that presents a set of assumptions (or axioms) such that any time-triggered system satisfying them is guaranteed (*i.e.*, by mathematical proof) to behave synchronously. The axioms constrain system assumptions about the clock skew, clock drift, communication delay, as well as place constraints on the scheduling of communication and computation events in the system (schedules are functions from local clock-times to actions). Unfortunately, even the formulation of these high-level specifications is difficult. Rushby's original formulation was found to contain errors that were subsequently corrected by Pike [95, 96].

Because the timing constraints are complex, yet it is essential that they hold, we are motivated to see whether they can be monitored at runtime. The question to ask then is how can these properties be monitored?

Unfortunately, the timing constraints cannot be monitored directly. The constraints essentially relate the values of local hardware clocks to one another and to the "real" (or "wall-clock") time. As a concrete example, consider the (revised) *clock drift-rate* constraint [95] that bounds the clock-time of a local clock to drift no more than a linear rate of real-time:

> Let $t_1, t_2 \in \mathbb{R}$ be real (or wall-clock) times (modeled as real numbers) such that $t_1 \geq t_2$. Let $\rho \in \mathbb{R}$ be a constant such that $0 < \rho < 1$. Suppose that C models a digital clock, by mapping a real-time value to the current clock-time. Then $\lfloor (1-\rho) \cdot (t_1 - t_2) \rfloor \leq C(t_1) - C(t_2) \leq \lceil (1+\rho) \cdot (t_1 - t_2) \rceil$.

The property cannot be monitored directly—a monitor does not have access to the "true" real time any more than the monitored node does (assuming each of their hardware clocks are equally accurate).

Rather than monitoring timing constraints directly, one can only monitor for the presence of faults. Both random hardware failures (*e.g.,* caused by radiation) and systematic faults (*e.g.,* caused by incorrect timing assumptions) may result in the same observations. In general, a monitor cannot distinguish whether an observed event results from one or the other faults. However, for a given set of environmental conditions, the rate of observed faults due to random hardware failures may differ from the fault-arrival rate due to systematic timing constraint violations. In this case, probabilistic modeling methods may be useful for characterizing the probable cause of an observed fault [97–99]. The soundness of diagnosing the cause of an observed fault depends on the fidelity of the fault model stating the probability of random faults caused by hardware failure and the environment. In general, there is a paucity of reliable fault-arrival data [100]. It may be possible to apply recent advances in monitoring probabilistic properties [101, 102] to monitor for timing faults.

7.2 Point-to-Point Error-Checking Codes

CRCs Error-checking codes are standard practice in point-to-point communication for detecting data corruption. One popular class of error-checking code is *cyclic redundancy checksums* (CRC) [103,104]. CRCs are functions that compute a remainder using polynomial division modulo 2 over a bitstream (we refer to the remainder as a CRC and let context distinguish the uses). In general, the way a CRC is used to detect faults in point-to-point communication is for the transmitter to compute a CRC over a data word (a fixed-width bitstream) and then to send both the word and the CRC to the receiver. The receiver then computes a CRC over the received word, and the result should match the received CRC.

CRCs are popular error-checking codes because they can be implemented efficiently in hardware, and they are good at detecting random errors. In particular, CRCs using a polynomial of degree n will detect any single *burst error* of length n or less—that is, it can detect any subsequence $b_1, b_2, \ldots, b_n$ of bits in a message where each bit b_i is arbitrarily a 0 or 1. In addition to detecting burst errors, CRCs can detect errors that are arbitrarily-spaced. The *Hamming Distance* (HD) of a CRC for a word is the smallest number of arbitrary errors that go undetected by the CRC. The HD of a CRC depends on both the specific CRC and the size of the data word. For typical CRCs and data bitstream sizes, HDs are less than 10; as the size of a bit stream increases, HDs decrease [104].

CRCs are widely-used in telecommunications, Ethernet, and embedded systems. In particular, CRCs are used in ultra-reliable systems—*i.e.*, systems designed so that the probability of their failure is no greater than 10^{-9} per hour of operation [2]. Paulitsch *et al.* describe the use of CRCs in ultra-reliable systems and highlight a variety of flawed assumptions about system design and CRC usage that may reduce the ability of CRCs to capture transmission errors [103]. Indeed, some flawed assumptions may reduce the overall reliability of the system by an order of magnitude or more.

	11-Bit Message	USB-5
Receiver A	1 1 1 1 1 1 0 1 1 0 1	1 0 0 0 1
Transmitter	1 1 1 1 1 1 0 1 1 0 $\frac{1}{2}$	1 $\frac{1}{2}$ 0 $\frac{1}{2}$ 1
Receiver B	1 1 1 1 1 1 0 1 1 0 0	1 1 0 1 1

Figure 10. Driscoll *et al.*'s Schrödinger CRC [1]

Schrödoinger CRCs For example, Driscoll *et al.* describe what they call "Schrödinger's CRCs" [1]. A Schrödinger's CRC is a bitstream and CRC that is broadcast by a transmitter to two different receivers such that the original bitstream and the CRC is corrupted in a way so that bitstreams received differ, but they pass the respective CRC checks ("Schrödinger's CRC" pays homage to the famous Schrödinger's Cat thought experiment from Quantum Mechanics). To illustrate the phenomenon, consider Figure 10. We illustrate a transmitter broadcasting an 11-bit message to two receivers listening on a data bus. We use the USB-5 CRC ($x^5 + x^2 + 1$), a CRC generally used to check Universal Serial Bus (USB) token packets [104].

Now, suppose the transmitter has suffered some fault such as a "stuck-at-1/2" fault so that periodically, the transmitter fails to drive the signal on the bus sufficiently high or low. A receiver may interpret an intermediate signal as either a 0 or 1. In the figure, we show the transmitter sending three stuck-at-1/2 signals, one in the 11-bit message, and two in the CRC. USB-5 catches a one-bit error in the message, so if the receivers interpret the bit differently, one of their CRCs should fail. However, if there is a bit-error in the CRC received from the transmitter also, then each receiver computes a correct CRC even though they receive different messages!

Effects like these (as well as others discussed by Paulitsch *et al.* [103]) can reduce the effectiveness of CRCs and more generally reduce the reliability of a system. Moreover, the example demonstrates that CRCs are insufficient for preventing Byzantine faults from occurring in a distributed system.

Monitoring CRCs Because there are known faults that can cause point-to-point error-checking codes to fail in unexpected ways, we might ask if online monitoring can help. Here we speculate on monitoring approaches.

Examples like Schrödiger's CRC are instances of the more general Byzantine fault problem, in which case a system should be built to tolerate Byzantine faults by executing fault-tolerant consensus algorithms [21]. If the system cannot be (re)designed to tolerate Byzantine faults, then the monitoring approaches we described in Section 7.1 for detecting consensus violations are generally applicable.

Indeed, an "optimization" is that monitors themselves can use CRCs to check consensus. One advantage in doing so is that CRCs require less bandwidth than the message itself. This is of interest in monitoring architectures in which nodes of the SUO send special monitoring messages to monitors, monitors exchange messages with each other, or the monitors communicate over the same bus used by the SUO. Furthermore, a shorter message has less chance of being corrupted itself in transit to the monitor. However, by using CRCs, there is a danger that consensus is violated, but CRCs sent to or exchanged by monitors *agree* (*i.e.*, a false negative).

7.3 Monitoring Fault-Tolerant Management Software

Thus far we have focused on directly monitoring fault-tolerance at the architectural level rather than the functional correctness of the software that manages fault tolerance. The failure of the fault-tolerant management software affects the functional correctness of a system, and in its worst manifestation, it can result in a safety violation. Consequently, monitoring for an error in the fault-tolerant management software reduces to traditional monitoring of safety properties.

7.3.1 Safety-Critical Systems

Consider the canonical steam-boiler example [105], which has features that is representative of safety-critical control applications. The following description is taken directly from [105]:

> The physical plant consists of a steam boiler. Conceptually, the boiler is heated and the water in the boiler evaporates into steam and escapes the boiler to drive a generator. The amount of heat and, therefore, the amount of steam changes without any considered control. Nevertheless, the safety depends on a bounded water level q in the boiler and steam rate v at its exit. A set of four equal pumps supply water for steam that leaves the boiler. These four pumps can be activated or stopped by the controller system.The controller reacts to the information of two sensors, the water level sensor and the steam rate sensor, and both may fail.
>
> The water level has two safety limits, one upper (M_u) and one lower (M_l). If the water level reaches either limit, there is just time enough to shut down the system before the probability of a catastrophe gets unacceptably high. The steam rate has an upper limit of W and, again, if reached the boiler must be stopped immediately.

A monitor would need to verify that $v \leq W$ and $M_l \leq q \leq M_u$ possibly shutting the system down if the bounds are violated.

As noted above, we assume that the SUO is a fault-tolerant system and we are verifying that a fault in the fault-tolerant management system does not interfere with the functional correctness by monitoring the steam rate and water levels. In considering the most appropriate monitoring architecture for this application, we assume that the steam rate sensor and water level sensor communicate to the computers via a shared data bus. Having the monitor attached to the data bus as depicted in Figure 6.4.1 of Section 7 is not appropriate since this architecture cannot detect Byzantine faults. The single monitor architecture of Figure 8 in Section 6.4.2 would require each of the nodes to be instrumented to report to the monitor a copy of the received message. CRC checks can be used to mitigate errors in transit. Implementing this architecture is straightforward, but we must assume that there are no faults in the monitor, which is a reasonable assumption, given its simplicity. A more sophisticated approach may be a fault-tolerant monitoring system using the architecture in Figure 9 in Section 6.4.3, where each node's monitor checks the measured values received, data is exchanged among the monitors and votes are taken. This would add an additional level of fault-tolerance at the expense of additional complexity.

7.3.2 Traffic Patterns

One can monitor a protocol's behavior by observing internal state, but this requires some instrumentation of the protocol in order to insert sensors that transmit the state information to the monitor. In many cases, this is impractical because the manufacturer will not allow a third-party to access their product, or because the protocol is implemented in silicon as a FPGA or a customized chip. Many protocol properties may be verified by observing

the communication traffic and verifying that the messages obey the pattern defined by the protocol specification that the message fields have the expected value. Although there are significant challenges in recognizing the protocol events of interest in a huge stream of data and verifying properties over that event trace, there has been some progress in this area that can be applied to critical systems. For instance, Bhargavan *et al.* [58, 59] observe TCP traffic and verify that a trace of messages between two nodes satisfy properties such as:

- An acknowledgement is generated for at least every other TCP message received.
- Acknowledgement sequence numbers are non-decreasing.

We also discussed BusMoP [35, 36] in Section 4.2 that monitored data transfers on a PCI bus. An example is a property that says any modification to a particular bus control register only happens when not in use. It may be possible to apply similar techniques to monitor communication patterns for fault-tolerant buses such as TTEthernet.

The most straightforward monitor architecture to monitor bus-traffic patterns is depicted in Figure 6.4.1 in Section 7. As noted above, this architecture cannot be used to detect Byzantine faults, but can be employed to monitor messages if there are no Byzantine faults. This may seem a relatively easy solution to engineer, but Bhargavan et.al. [58] show that simply viewing the traffic from the point of an external observer is quite difficult since events may be buffered or lost between the monitor and the SUO. The single monitor architecture of Figure 8 in Section 6.4.2 would require each of the nodes to be instrumented to report to the monitor a copy of the received message. The monitor then checks that the message is correctly formed and obeys the protocol's communication pattern. This choice has the disadvantage that nodes have to be instrumented to send the message to the monitors and a failure on the monitor link could mean that the message received by the monitor differs from the message actually received, but CRCs could be employed to mitigate that issue. The distributed monitor in Figure 9 in Section 6.4.3 would use the monitor at each node to check the messages received at that node. The monitors may have to communicate, which as we saw above can be a drawback. The latter two architectural design choices should not require as much buffering of the data stream since the nodes would be instrumented to only send the packets of interest to the monitor, but this does not mean that messages still cannot be lost under the right conditions. Regardless of the architecture chosen, data corrupted on the interface to the monitor could lead to both false positives and false negatives.

8 Conclusions

Online monitoring is a promising technique for making safety-critical real-time distributed systems more reliable. Section 4 surveyed a number of past and recent efforts in software monitoring. Several research groups have had ongoing efforts in the area for over a decade and have produced impressive tool sets to both create monitors and to instrument the SUO. Consequently, there is a solid foundation of research in online monitoring of conventional software systems upon which to build. Yet little research to date has focused on hard real-time critical systems, where monitoring can arguably have the most impact in preventing costly and possibly fatal system failures.

Section 5 introduced the IVHM philosophy of monitoring a vehicle while in motion for wear and tear on the physical systems such as cracks on aircraft wings. We argued that software monitoring fits in with the IVHM philosophy by monitoring for violations of specific safety properties, namely, consensus properties.

Sections 6 and 7 explored monitoring frameworks for distributed real-time systems. We have described potential monitoring architectures and have discussed their tradeoffs. We have also described classes of properties to monitor. We believe this work will provide a foundation for future directed research.

References

1. Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP*, Lecture Notes in Computer Science, pages 235–248. Springer, September 2003.

2. John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report CR–4551, NASA, December 1993.

3. Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19:3–12, 1993.

4. Steven D. Johnson. Formal methods in embedded design. *Computer*, 36:104–106, November 2003.

5. Chris Bergin. Faulty MDM removed. NASA Spaceflight.com, May 18 2008. Available at http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues-faulty-mdm-removed/. (Downloaded Nov 28, 2008).

6. Chris Bergin. STS-126: Super smooth endeavor easing through the countdown to L-1. NASA Spaceflight.com, November 13 2008. Available at http://www.nasaspaceflight.com/2008/11/sts-126-endeavour-easing-through-countdown/. (Downloaded Feb 3, 2009).

7. NASA - Johnson Flight Center. Space shuttle operational flight rules volume A, A7-104, June 2002. Available from http://www.jsc.nasa.gov (Downloaded Nov 28, 2008).

8. M. Sheffels. A fault-tolerant air data/inertial reference unit. *IEEE AES Systems Magazine*, March 1993.

9. Y.C. Yeah. Triple-triple redundant 777 primary flight computer. In *IEEE Aerospace Applications Conference*, pages 293–307. IEEE Press, 1996.

10. G. Bartley. *Boeing B-777: Fly-By-Wire Flight Controls*, chapter 11. CRC Press, 2001.

11. Australian Transport Safety Bureau. In-flight upset event 240 Km North-West of Perth, WA Boeing Company 777-200, 9M-MRG 1 August 2005. ATSB Transport Safety Investigation Report, 2007. Aviation Occurrace Report - 200503722.

12. C. Johnson and C. Holloway. The dangers of failure masking in fault-tolerant software: Aspects of a recent in-flight upset event. In *Instituition of Engineering and Technology International Conference on System Safety*, pages 60–65, 2007.

13. Kerryn Macaulay. ATSB preliminary factual report, in-flight upset, Qantas Airbus A330, 154 Km West of Learmonth, WA, 7 October 2008. Australian Transport Safety Bureau Media Release, November 14 2008. Available at http://www.atsb.gov.au/newsroom/2008/release/2008_45.aspx.

14. Australian Government. In-flight upset 154 Km West of Learmonth 7 October 2008 VH-QPA Airbus A330-303. Australian Transport Safety Bureau Aviation Occurrence Investigation AO-2008-70, 2008. Available at http://www.atsb.gov.au/publications/investigation_reports/2008/AAIR/aair200806143.aspx.

15. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

16. Lee Pike and Steven D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press. Available at http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html.

17. Jean-Claude Laprie. Dependability—its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictability Dependable Computing Systems*, ESPRIT Basic Research Series, pages 3–24. Springer, 1995.

18. Ricky W. Butler. A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center, 2008.

19. John Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

20. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.

21. Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.

22. M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

23. Paul Miner, Alfons Geser, Lee Pike, and Jeffery Maddalon. A unified fault-tolerance protocol. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004. Available at http://www.cs.indiana.edu/~lepike/pub_pages/unified.html.

24. Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center, 2005.

25. Elizabeth Ann Latronico. *Reliability Validation of Group Membership Services for X-by-Wire Protocols*. PhD thesis, Carnegie Mellon University, May 2005. Available at http://www.ece.cmu.edu/~Ekoopman/thesis/latronico.pdf.

26. B. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, 1995.

27. Severine Colin and Leonardo Mariani. *Model-based Testing of Reactive Systems*, volume 3472 of *LNCS*, chapter Run-time Verification, pages 525–556. Springer Verlag, 2005.

28. N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime monitoring tools. *IEEE Transactions of Software Engineering*, 30(12):859–872, 2004.

29. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2008. To Appear.

30. J. Tsai, K. Fang, H. Chen, and Y. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, 1990.

31. P. Dodd and C. Ravishankar. Monitoring and debugging distributed real-time programs. *Software - Practice and Experience*, 22(10):863–877, 1992.

32. S. Chodrow, F. Jahanian, and M. Donner. Runtime monitoring of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 74–83, 1991.

33. S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specificed programs. *IEEE Computer*, 26(3):32–41, 1993.

34. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 2005. Third Edition.

35. R.Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *RTSS'08: Proceedings of the 29th IEEE Real-Time System Symposium*, pages 481–491, 2008.

36. Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 481–491, Washington, DC, USA, 2008. IEEE Computer Society.

37. Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification revisited. Technical Report TUM-I05, Technical University of Munich, October 2005. Available at http://users.rsise.anu.edu.au/~baueran/publications/tum-i0518.ps.gz.

38. Z.Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

39. F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *LICS02: Proceeding of Logic in Computer Science 2002*, pages 383–392. IEEE Computer Society Press, 2002.

40. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

41. G. Rosu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns. In *RV'08: Proceedings of Runtime Verification*, 2008. To Appear.

42. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, number 2937 in LNCS. Springer-Verlag, 2004.

43. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From eagle to ruler. In *RV07: Proceedings of Runtime Verification 2007*, number 4839 in Lecture Notes in Computer Science, pages 111–125. Springer-Verlag, 2007.

44. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems*, pages 114–122, 1999.

45. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 279–287, 1999.

46. K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, 2002.

47. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002. Proceeding of 2nd International Conference on Runtime Verification.

48. O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144(4):91–108, 2005. Proceeding of 5th International Conference on Runtime Verification.

49. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

50. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.

51. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narcisco Martí-Oliet, José Meseguer, and José Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

52. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-base framewrok for software development analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, number 3308 in Lecture Notes on Computer Science, pages 357–373. Springer-Verlag, 2004.

53. F. Chen and G. Roşu. Java-MOP: a monitoring oriented programming environment for Java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer-Verlag, 2005.

54. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 569–588, 2007.

55. K. Havelund. Runtime verification of C programs. In *TestCom/FATES*, number 5047 in Lecture Notes in Computer Science. Springer-Verlag, 2008.

56. M. Mansouri-Samani and M. Sloman. Monitoring distributed systems (a survey). Imperial College Research Report DOC92/23, Imperial College of Science and Technology and Medicine, 1992.

57. A. Bauer, M. Leucker, and C. Schallhart. Model-based runtime analysis of distributed reactive systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC)*, Sydney, Australia, April 2006. IEEE Computer Society.

58. K. Bhargavan, S. Chandra, P. McCann, and C.A. Gunter. What packets may come: Automata for network monitoring. *SIGPLAN Notices*, 35(3):209–219, 2001.

59. K. Bhargavan and C. A. Gunter. Requirement for a practical network event recognation language. *Electronic Notes in Theoretical Computer Science*, 70(4):1–20, 2002.

60. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *ICSE'04: Proceedings of 6th International Conference on Software Engineering*, pages 418–427, 2004.

61. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1997.

62. Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *CSF'97: Proceedings of the 10th IEEE Computer Security Foundations Workshop 1997*, pages 116–124, 1997.

63. A. Doudou and A. Schiper. Muteness detectors for consensus with Byzantine processes. In *Proceedings of the 17th ACM Symposium on Principle of Distributed Computing*, pages 31–5, 1997.

64. L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. Fault detection for Byzantine quoroum systems. *IEEE Transaction on Parallel and Distributed Systems*, 12(9):996–1007, 2001.

65. K. Kihlstrom, L. Moser, and P. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

66. A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proceedings of the 2nd conference on Hot Topics in System Dependability*, 2006.

67. F. Jahanian, R. Rajkumar, and S. Raju. Run-time monitoring of timing constraints in distributed real-time systems. *Real-Time Systems Journal*, 7(2):247–273, 1994.

68. T. Savor and R. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136–147, 1997.

69. R. Alur and T. Henzinger. Logics and models of real time: A survey. In *Real Time Theory and Practice*, volume Lecture Notes in Computer Science 600. Springer-Verlag, 1992.

70. R. Alur and T. Henzinger. Real time logics: complexity and expressiveness. In *Fifth Annual Symposium on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, 1990.

71. P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes Theoretical Computer Science*, 113, 2004.

72. Drusinsky D. On-line monitoring of metric temporal logic wit time-series constraints using alternating finite automata. *Journal of Universal Computer Science*, 12(5):482–498, 2006.

73. D. Basin, F. Klaedtke, S. Muller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, Leibniz International Proceedings in Informatics, pages 49–60. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany., 2008.

74. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

75. P. Caspi, D. Pialiud, N. Halbwachs, and J. Plaice. LUSTRE: a declratative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, 1987.

76. B. D'Angelo, S. Sankaranarayanan, C. Snchez, W. Robinson, Zohar Manna, B. Finkbeiner, H. Spima, and S. Mehrotra. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation adn Reasoning*, pages 166–174. IEEE Computer Society Press, 2005.

77. A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, pages 252–262, 1997.

78. C. Lee. *Monitoring and Timing Constraints and Streaming Events with Temporal Uncertainties*. PhD thesis, University of Texas, 2005.

79. S. Ofsthun. Integrated vehicle health management for aerospace platforms. *IEEE Instrumentation and Measurement Magazine*, pages 21–24, September 2002.

80. M. Schwabacher, J. Samuels, and L. Brownston. The NASA integrated vehicle health management technology experiment for the X-37. In *Proceedings of the SPIE AeroSense 2002 Symposium*. Society of Photo-Optical Instrumentation Engineers, 2002.

81. National Research Council. Decadal survey of civil aeronautics. National Academies Press, 2006.

82. RTCA Inc. Software considerations in airborne systems and equipment certification, 1992. RCTA/DO-178B.

83. Daniel Jackson, Martyn Thomas, and Lynette I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence*. National Academies Press, 2007.

84. John Rushby. Runtime certification. In *RV'08: Proceedings of Runtime Verification, Budapest, Hungary, March 30, 2008. Selected Papers*, pages 21–35. Springer-Verlag, 2008.

85. John Rushby. Just-in-time certification. In *12th IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS)*, pages 15–24, Auckland, New Zealand, July 2007. IEEE Computer Society. Available at http://www.csl.sri.com/~rushby/abstracts/iceccs07.

86. Gabor Karsai, Gautam Biswas, Sherif Abdelwahed, Nag Mahadevan, and Eric Manders. Model-based software tools for integrated vehicle health management. In *SMC-IT '06: Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology*, pages 435–442. IEEE Computer Society, 2006.

87. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

88. Colin Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, 1996.

89. Hermann Kopetz. *Real-Time Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

90. John Rushby. Software verification and system assurance. In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–10, Hanoi, Vietnam, November 2009. IEEE Computer Society.

91. Hermann Kopetz. The rationale for time-triggered ethernet. In *RTSS'08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 3–11. IEEE Computer Society, 2008.

92. Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

93. D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 386–395. IEEE Press, 1992.

94. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

95. Lee Pike. A note on inconsistent axioms in Rushby's "systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348, May 2006. Available at http://www.cs.indiana.edu/~lepike/pub_pages/time_triggered.html.

96. Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *FMCAD'07: Proceedings of Formal Methods in Computer Aided Design*, pages 231–238. IEEE, 2007. Available at http://www.cs.indiana.edu/~lepike/pub_pages/fmcad.html. Best Paper Award.

97. V. Rosset, P. Souto, P. Portugal, and F. Vasques. A reliability evaluation of a group membership protocol. In *Proceedings of SAFECOMP 2007*, volume 4680 of *LNCS*, pages 397–410. Springer, 2007.

98. Elizabeth Latronico and Philip Koopman. Design time reliability analysis of distributed fault tolerance algorithms. *International Conference on Dependable Systems and Networks*, pages 486–495, 2005.

99. Elizabeth Latronico, Paul Miner, and Philip Koopman. Quantifying the reliability of proven SPIDER group membership service guarantees. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 275, Washington, DC, USA, 2004. IEEE Computer Society.

100. Theresa C. Maxino and Philip J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1):59–72, 2009.

101. U. Sammapun, I. Lee, and O. Sokolsky. RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 147–153, 2005.

102. U. Sammapun, I. Lee, O. Sokolsky, and J. Regehr. Statistical runtime checking of probabilistic properties. In *RV'07: Proceedings of Runtime Verification*, Lecture Notes in Computer Science 4839, pages 164–175, 2007.

103. Michael Paulitsch, Jennifer Morris, Brendan Hall, Kevin Driscoll, Elizabeth Latronico, and Philip Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 346–355. IEEE Computer Society, 2005.

104. Philip Koopman and Tridib Chakravarty. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 145. IEEE Computer Society, 2004.

105. J.-R. Abrial, E. Borger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 07 - 2010	Contractor Report	

4. TITLE AND SUBTITLE
Monitoring Distributed Real-Time Systems:
A Survey and Future Directions

5a. CONTRACT NUMBER
NNL08AA19B

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Goodloe, Alwyn E.; Pike, Lee

5d. PROJECT NUMBER

5e. TASK NUMBER
NNL08AD13T

5f. WORK UNIT NUMBER
645846.02.07.07.07.15.02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

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-2010-216724

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 64
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Ben Di Vito

14. ABSTRACT

Runtime monitors have been proposed as a means to increase the reliability of safety-critical systems. In particular, this report addresses runtime monitors for distributed hard real-time systems. This class of systems has had little attention from the monitoring community. The need for monitors is shown by discussing examples of avionic systems failure. We survey related work in the field of runtime monitoring. Several potential monitoring architectures for distributed real-time systems are presented along with a discussion of how they might be used to monitor properties of interest.

15. SUBJECT TERMS
Distributed systems; Fault tolerance; Real-time systems; Reliability; Run-time monitoring

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)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	49	(443) 757-5802