

Toward Monitoring Fault-Tolerant Embedded Systems (Extended Abstract)

Alwyn Goodloe and Lee Pike

1 INTRODUCTION

Flight-critical systems for aircraft and spacecraft must be ultra-reliable and so are designed to be fault-tolerant. For embedded control systems and data buses, a primary means for achieving fault-tolerance is hardware replication to provide redundancy so that the system can survive random hardware faults of individual components.

Nevertheless, a system may fail to meet its reliability requirements for one of three reasons: (1) the system suffers an unexpectedly high number of hardware faults, (2) hardware faults lead to unexpected system-level failures, or (3) software or design bugs result in systematic faults. Indeed, failures may result from a combination of these reasons (e.g., an unanticipated hardware faults triggers the execution of fault-management software that is incorrectly designed, leading to system failure).

A *monitor* [1], [2] is a runtime verification mechanism that observes the behavior of a system and detects if it is consistent with its specified correct behavior. Ultra-reliable systems stand to benefit from runtime monitors, if monitors can be constructed to increase their reliability. However, ultra-reliable systems, which are often distributed, real-time, fault-tolerant systems, have largely been ignored by the monitoring community. The challenges associated with monitoring this class of systems include the following:

- How to ensure the monitor does not interfere with the monitored system delivering its services and meeting its real-time deadlines.
- How to ensure the monitoring infrastructure does not reduce the reliability of the monitored system.
- How to monitor a fault-tolerant system, since fault-tolerant systems are distributed to provide replication.
- How to monitor for faults.

We propose these are all important open research questions.

In this extended abstract, we motivate the need for runtime monitoring for ultra-reliable systems. We motivate this need by first presenting an example of a

failure in the Space Shuttle's data processing system. The system was ostensibly designed with best-practices in mind yet still managed to suffer a failure. After presenting the motivating example, we touch on some of these open questions.

2 FAILURE IN THE SPACE SHUTTLE

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 demultiplexers (MDM) units aboard the orbiter, sixteen of which are directly connected to the GPCs via redundant shared busses. Each of these MDMs receives commands from guidance navigation and control (GNC) running on the GPCs and acquires the requested data from sensors attached to it, which is then sent back to the GPCs. In addition to their role in multiplexing/demultiplexing data, these MDM units perform analog/digital conversion.

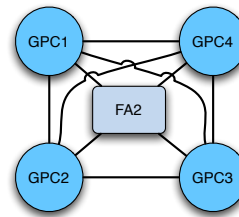


Fig. 1. Shuttle Data Processing System (GPCs and FA2)

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 a report of a pre-launch failure of the fault diagnosis software caused by a “non-universal I/O error” in the second flight aft (FA2) MDM [3], which is polled by the GPCs as shown in Figure 1. According to reports [3], [4], the events unfolded as follows:

- A diode failed on the serial multiplexer interface adapter 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 [5], GPC 2 and GPC 4 are powered down.
- GPC 3 then reads FA2's built-in test equipment and determines that it is faulty at which point it too is removed from redundancy set leaving only GPC 1 at which time engineers terminated the work 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.

3 BYZANTINE FAULTS

Based on the motivating example we have just provided, we discuss approaches to monitoring fault-tolerant systems.

3.1 Classifying Faults

Faults can be classified according to the hybrid fault model of Thambidurai and Park [6]. The classification is based on the observable behavior of a node, ranging from easy-to-detect faults, like fail-silent nodes that fail to respond to pings within a nominal time frame, to more nefarious faults. A particularly nefarious class of faults are *asymmetric* or *Byzantine* faults in which a node sends different messages to different receivers when the expectation is that the node should broadcast the same message to all receivers [7].

Byzantine faults are often transient rather than permanent, making them difficult to reproduce. For example, a transmitter failing to drive a signal sufficiently high or low can produce Byzantine faults in which different receivers observe a broadcasted message differently; timing errors in real-time systems can also produce Byzantine faults [8]. In part because of their transience, system engineers underestimate the probability of Byzantine faults.

3.2 Reconsidering the Space Shuttle Failure

A *maximum fault assumption* (MFA) for a system characterizes the maximum kind, number, and arrival rate of faults under which the system is hypothesized to operate correctly. If the MFA is violated, the system's assumptions about its environment are violated, and the system may behave arbitrarily.

The Space Shuttle incident is not isolated: the preliminary findings on the A330 in-flight upset exhibited an asymmetric fault [9]. In the case of both the Space Shuttle and the A330, it appears that the systems were designed to satisfy a MFA that did not accommodate asymmetric faults.

It is conceivable (we are speculating here) that the 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 occur [10]. 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. However, it has also been argued that Byzantine faults, while rare, are much more probable than generally believed [8]. Indeed, Driscoll *et al.*, describe possible causes of Byzantine faults over a shared bus [8], which are possible causes of the Space Shuttle failure.

4 MONITORING ULTRA-RELIABLE SYSTEMS

4.1 Monitors

A *monitor* [1], [2] is a runtime verification mechanism that observes the behavior of a system and detects if it is consistent with a given specification. A specification of a correctness property ϕ is typically expressed in some specification language and a monitor is constructed that accepts all traces satisfying ϕ . The 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. The user or other software is left to respond to the alert. The state-of-the-art in the field is represented by the Monitoring and Checking (MaC) [11] and Monitor Oriented Programming (MOP) [12] projects. The focus of most research in the area has been on monitoring Java applications such as detecting deadlocked threads. Two research efforts investigate monitoring distributed systems [13], [14], but fault-tolerant, hard real-time systems add additional complexities. In particular, the monitor itself must not interfere with the hard real-time deadlines.

4.2 Toward Monitoring Faults

A designer's formulation of a system's maximum fault assumption (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, the 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 in unanticipated environments). Because the MFA is chosen based on incomplete data, it may be too weak. Furthermore, a system's MFA is built on the *assumption* that there are no systematic software faults. Software faults can dramatically reduce the hypothesized reliability of a system.

In a fault-tolerant system, the relationship between software and hardware is subtle. Software that implements a system's services can be verified and validated

or monitored at runtime using known techniques. 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 that responds to its environment, where the environment introduces random hardware faults.

Thus, to know at runtime whether fault-management software is behaving correctly, one must also know the status of the hardware faults 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!

Rather than “reinventing” the fault-management software (which would be prone to the same software errors as the original) [15], we propose that monitors are well-suited for fault-detection *outside* of a system’s MFA. That is, monitors are suitable for monitoring faults that fall outside of the ones detected and corrected by the fault-management software—such as Byzantine faults in the case of the Shuttle’s data processing system and the A330 in-flight upset.

Constructing such a monitor does not require any knowledge or interaction with the fault-management software itself. For example, in the case of the Shuttle’s data processing system, a separate monitor might poll each of the GPCs (perhaps on a separate data bus) to ensure consensus among them. If less than three out of the four GPCs output different values, the monitor might note that some failure has occurred. The monitor does not need to identify the source of failure to diagnose a fault.

5 CONCLUSION

“Monitoring MFAs” is an open research topic that stands to improve the reliability of complex legacy systems. Specific research questions include those mentioned in the introduction. For example, a monitor should not interfere with a monitored system’s ability to meet real-time deadlines. So, special monitoring messages might be sent over a dedicated monitoring bus, for example. Likewise, monitors need to be prevented from reducing the overall reliability of the system. This includes preventing a monitor from generating false-positives (provided a raised alarm causes a system-wide reset, for example). If monitors are attached to the same data bus as the nodes being monitored, they must be outfitted with *bus guardians*, special hardware that ensures a node does not become a “babbling idiot” that monopolizes data bus bandwidth. Specific approaches to answering these questions should be explored and compared.

ACKNOWLEDGMENTS

This work is supported by NASA Contract NNL08AD13T from the Aviation Safety Program Office; Ben Di Vito of the NASA Langley Research

Center has provided general guidance on these efforts. We would like to thank the workshop reviewers who provided exceptionally detailed and helpful comments.

REFERENCES

- [1] N. Delgado, A. Gates, and S. Roach, “A taxonomy and catalog of runtime monitoring tools,” *IEEE Transactions of Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.
- [2] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *Journal of Logic and Algebraic Programming*, 2008, to Appear.
- [3] C. 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).
- [4] —, “Sts-126: Super smooth endeavor easing through the countdown to 1-1,” NASA Spaceflight.com, November 13 2008, available at <http://www.nasaspaceflight.com/2008/11/sts-126-endeavour-easing-through-countdown/>. (Downloaded Feb 3, 2009).
- [5] N. J. F. Center, “Space shuttle operational flight rules volume a, a7-104,” June 2002, available from <http://www.jsc.nasa.gov> (Downloaded Nov 28, 2008).
- [6] P. Thambidurai and Y.-K. Park, “Interactive consistency with multiple failure modes,” in *7th Reliable Distributed Systems Symposium*, October 1988, pp. 93–100.
- [7] Lamport, Shostak, and Pease, “The Byzantine generals problem,” *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.
- [8] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, “Byzantine fault tolerance, from theory to reality,” in *The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP*, ser. Lecture Notes in Computer Science. Springer, September 2003, pp. 235–248.
- [9] 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/air200806143.aspx.
- [10] D. Powell, “Failure mode assumptions and assumption coverage,” in *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*. IEEE Press, 1992, pp. 386–395.
- [11] 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*, 1999, pp. 114–122.
- [12] F. Chen and G. Roşu, “Mop: An efficient and generic runtime verification framework,” in *Object Oriented Programming, Systems, Languages, and Applications*, 2007, pp. 569–588.
- [13] 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: IEEE Computer Society, Apr. 2006.
- [14] K. Sen, A. Vardhan, G. Agha, and G. Rosu, “Efficient decentralized monitoring of safety in distributed systems,” in *6th International Conference on Software Engineering (ICSE’04)*, 2004, pp. 418–427.
- [15] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming,” *IEEE Transactions on Software Engineering*, vol. 12, pp. 96–109, 1986.