

# Onboard Guarded Software Upgrading: Motivation and Framework<sup>\*†</sup>

Ann T. Tai  
IA Tech, Inc.  
10501 Kinnard Avenue  
Los Angeles, CA 90024

Leon Alkalai Savio N. Chau  
Jet Propulsion Laboratory  
California Institute of Technology  
Pasadena, CA 91109

*Abstract* — The goal of the guarded software upgrading (GSU) framework is to minimize mission performance loss due to onboard software upgrading activities and that due to system failure caused by residual faults in an upgraded version. We exploit inherent system resource redundancies as the means of fault tolerance to meet the development cost and onboard resource constraints. Furthermore, we devise a message-driven confidence-driven protocol to facilitate effective and efficient error containment and recovery.

## TABLE OF CONTENTS

1. MOTIVATION
2. GSU FRAMEWORK
3. GSU MIDDLEWARE
4. SUMMARY AND FUTURE WORK

## 1. MOTIVATION

A challenge that arises from flight software evolvability is to avoid system performance loss, with respect to spacecraft and science functions, during the version replacement for software upgrade [1]. To date, the practice of onboard deep-space software upgrade still requires to reboot the entire flight computer after uploading a partial modification and before starting to run the updated software. In the Mars Pathfinder mission, two small changes (in the flight software) were made as a result of Operational Readiness Test; it took two hours to complete the patch process during which all the normal functions of the flight computer were stopped [2]. The costly performance loss and potential mission-failure risk associated with the blackout period is apparently unacceptable for NASA's future missions.

Moreover, when a software upgrade is conducted in a complex computing environment that involves process communication and coordination, such as distributed sens-

ing and control, error containment becomes more difficult and unmitigated error conditions could lead to complete system failure. Although researchers have been investigating dependable system upgrade for critical applications (see [3, 4], for example, which describe two recent projects), the proposed solutions all require special effort for developing dedicated system resource redundancy and lack considerations of performance cost, onboard resource limitations (e.g., power, processors, etc.), weight, mass, and volume constraints, and error contamination among interacting processes in distributed embedded systems. Therefore, embedded systems for deep-space applications are unable to benefit from those solutions.

## 2. GSU FRAMEWORK

With the above motivation, we propose a methodology called *guarded software upgrading* (GSU). The goal of GSU is to avoid or minimize mission performance degradation due to software upgrading activities during a mission and that due to system failure caused by residual faults in an upgraded version. Further, the GSU infrastructure will assure:

- 1) Seamless uploading and version switching, and
- 2) Efficient software error containment and protection, meaning that the negative impact of an upgrade, such as system unavailability and message/data loss, will be eliminated or minimized.

The development of the GSU framework is

- Based on the notion of “message-driven and confidence-driven” (MDCD).
- Characterized by a two-stage approach.
- Focused on low development cost and low performance overhead.

<sup>\*</sup>0-7803-6599-2/01/\$10.00 ©2001 IEEE

<sup>†</sup>The work reported in this paper was supported in part by NASA Small Business Innovation Research (SBIR) Contract NAS3-99125.

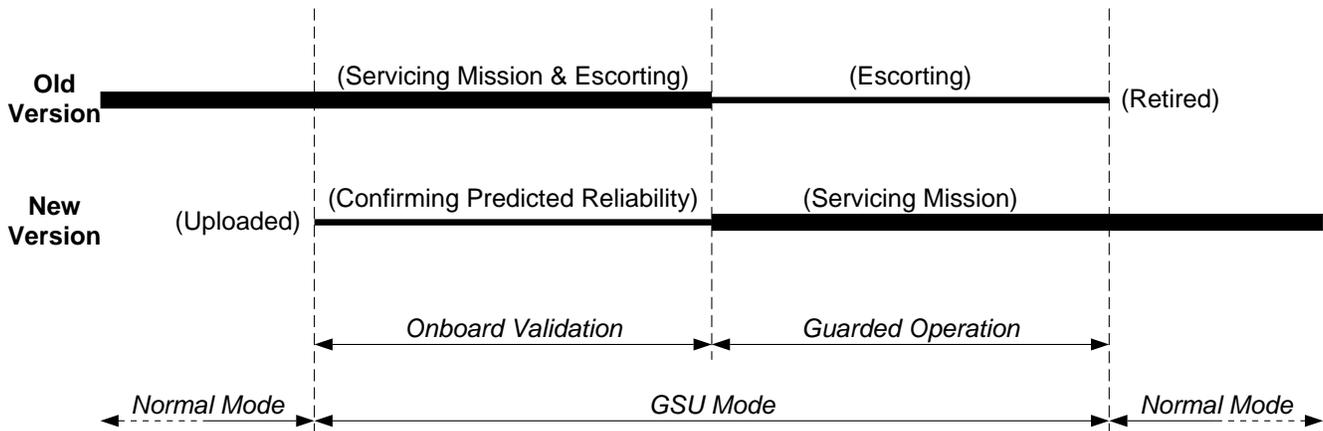


Figure 1: GSU Methodology

Specifically, we let an earlier version of the upgraded software component (i.e., inherent resource redundancy, which does not require additional development cost), which has long onboard execution time and deserves our high confidence, to escort the new version through *onboard validation* (stage 1) and *guarded operation* (stage 2), as illustrated in Figure 1.

X2000 is the Advanced Spacecraft Systems Development Program, initiated by NASA’s Office of Space Science in 1998, for the research and development of advanced spacecraft technologies for the future deep-space exploration missions [5]. As an engineering model intended to service multiple deep-space missions, the X2000 architecture must accommodate diverse requirements from different missions, which demand a computation power ranging from a single processor string to multiple strings, a throughput ranging from under 20 MIPS (million instructions per second) to over 100 MIPS, and a mass memory size ranging from 100 Mbytes to 1.5 Gbytes. Therefore, the X2000 architecture must be scalable and distributed in order to accommodate a broad spectrum of requirements. As a result, the Baseline X2000 First Delivery Architecture comprises three high-performance computing nodes (each of which has a 128-Mbyte local DRAM), the micro-controllers of subsystems, and a variety of devices, all connected by a fault-tolerant bus network that complies with the commercial interface standards IEEE 1394 and I2C [6, 7]. A useful feature of the X2000 distributed architecture is the I/O cross-strapping between the computing nodes and the IEEE 1394 and I2C buses. This feature permits the roles of the computing nodes to be interchangeable and the workload that comprises spacecraft and science functions to be shared by and migrated among processors in an efficient manner. As a result, the inherent resource redundancy in the distributed architecture can be employed by various onboard reliability enhancement activities, including guarded software upgrading. While the

distributed architecture facilitates the application of a variety of enabling technologies, it adds another dimension of challenge to onboard guarded software upgrading, which is that we must protect the system against failures caused by error propagation among interacting processes.

Since a software upgrade is normally conducted during a non-critical mission phase when the spacecraft and science functions do not require full computation power, only two processes corresponding to two different application software components are supposed to run concurrently and interact with each other. To exploit inherent system resource redundancies, we let the old version, in which we have high confidence due to its long onboard execution time, escort the new-version software component through two stages of GSU, namely, *onboard validation* and *guarded operation*. Further, we make use of the processor that otherwise would be idle (recall that the distributed X2000 architecture has three computing nodes) to enable the three processes (i.e., the two corresponding to the new and old versions, and the process corresponding to the second application software component) to execute concurrently. To aid in the description, we introduce the following notation:

- $P_1^{\text{new}}$  The process corresponding to the new version of an application software component.
- $P_1^{\text{old}}$  The process corresponding to the old version of the application software component.
- $P_2$  The process corresponding to another application software component (which is not undergoing upgrade).

Figure 2 illustrates the two-stage approach in further detail. As shown in Figure 2(a), during onboard validation the outgoing messages of the shadow process  $P_1^{\text{new}}$  are suppressed but selectively logged (as shown by the dashed lines with arrows), while  $P_1^{\text{new}}$  receives the same incoming

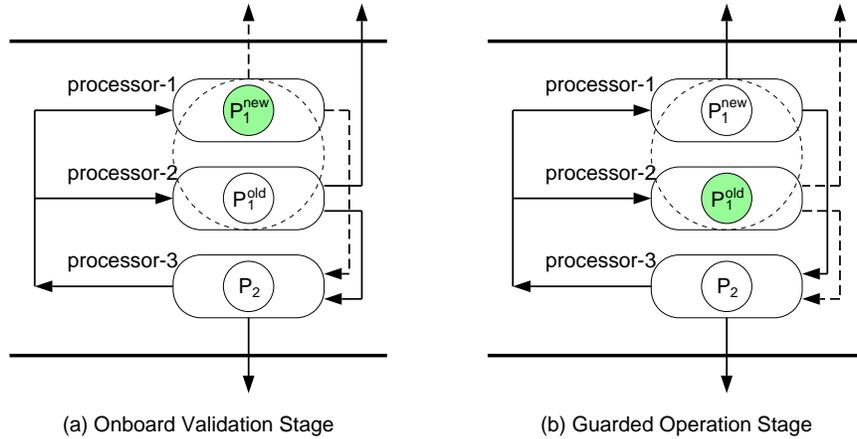


Figure 2: Two-Stage Approach to GSU

messages that the active process  $P_1^{\text{old}}$  does (as shown by the solid lines with arrows). Thus,  $P_1^{\text{new}}$  and  $P_1^{\text{old}}$  can perform the same computation based on identical input data. Note that each of the dashed circles that encapsulate  $P_1^{\text{new}}$  and  $P_1^{\text{old}}$  indicates that the two processes are created by two different versions of the same application software component.

By maintaining an onboard error log that can be downloaded to the ground to facilitate statistical modeling and heuristic trend analysis, onboard validation facilitates the decisions on whether and when to permit  $P_1^{\text{new}}$  to enter mission operation. If onboard validation completes successfully, then  $P_1^{\text{new}}$  and  $P_1^{\text{old}}$  switch their roles and enter the guarded operation stage. Since 1)  $P_1^{\text{new}}$  and  $P_1^{\text{old}}$  run concurrently and independently on two different processors prior to their role-switching, and 2)  $P_1^{\text{new}}$  maintains a message log and keeps track of the messages sent by  $P_1^{\text{old}}$ , the transition to guarded operation can be realized in a seamless fashion. In other words, the need for rebooting the computing nodes and the risk of state inconsistency or message missing are eliminated.

In order to minimize the impact on and risk to mission operation, onboard software upgrading is usually carried out in an incremental manner. In particular, most upgrades involve only a single software component at a time. As a result, the interaction patterns (message types and ordering) among the processes will remain the same after an upgrade. Accordingly, as indicated by Figure 2(b), during the guarded operation,  $P_1^{\text{new}}$  actually influences the external world and interacts with process  $P_2$ , while the messages of  $P_1^{\text{old}}$  that convey its computation results to  $P_2$  or devices are now suppressed and logged. Should an error of  $P_1^{\text{new}}$  be detected,  $P_1^{\text{old}}$  will take over  $P_1^{\text{new}}$ 's active role, and the system will resume its normal mode until the next upgrade attempt. The guarded operation is equipped with an error containment and recovery protocol, the MDCD protocol [8].

In order to validate the effectiveness of the MDCD protocol with respect to the reliability improvement it provides us with under realistic, non-ideal conditions, we have conducted probabilistic modeling using the parameter values that are appreciably less than perfect with regard to the design assumptions [9]. The quantitative results confirm that the MDCD approach is effective with respect to reliability improvement as surmised. Figure 3 illustrates the results of a quantitative comparative study based on stochastic activity network models [10]. In this particular evaluation, reliability is evaluated as a function of the fault manifestation rate of an upgraded software component  $\mu_{\text{new}}$ . As shown by the curves, the MDCD protocol provides significant reliability improvement over the baseline system where no error containment and recovery mechanisms are provided.

### 3. GSU MIDDLEWARE

Unlike the traditional software fault tolerance schemes in which recovery-point establishment and/or rollback patterns need to be pre-structured in application software, the dynamic nature of the MDCD protocol allows the error containment and recovery mechanisms to be transparent to the programmer, and facilitates a middleware implementation.

We prototyped the MDCD protocol in a middleware architecture that implements the GSU methodology, called the GSU Middleware. The design of the middleware testbed is intended to be consistent with the X2000 First Delivery Avionics System Architecture. Specifically, the development is carried out on a network of Sun Ultra-10 workstations connected by a 100 Mbit/sec Ethernet. Each workstation is equipped with a 300 MHz UltraSPARC processor and 256 Megabytes of memory, and runs the Solaris operating system. Each workstation executes the GSU mid-

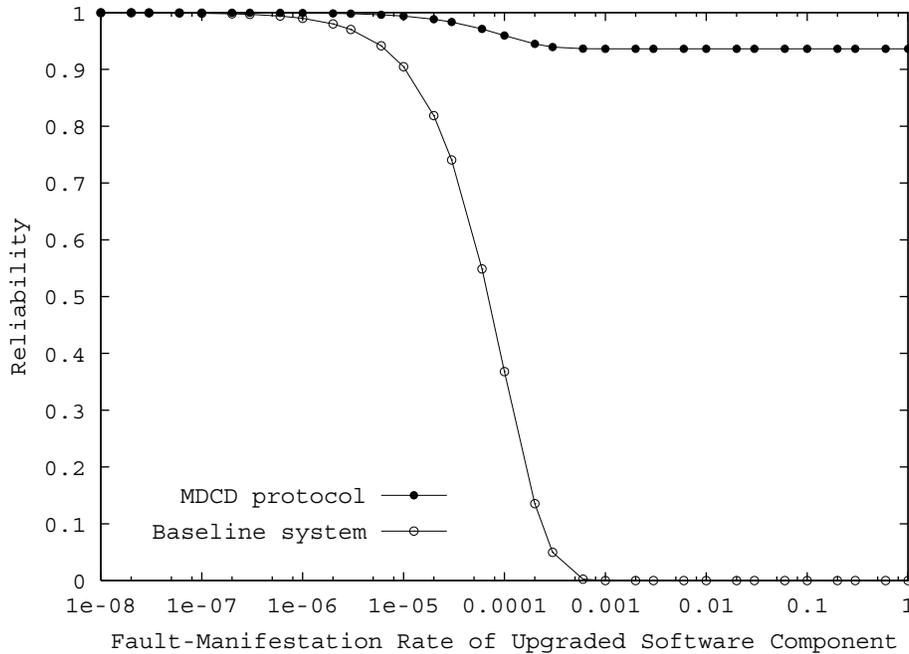


Figure 3: Reliability as a Function of  $\mu_{new}$

middleware which is an instance of the generic code created according to a specific command-line argument, namely,  $P_1^{new}$ ,  $P_1^{old}$ , or  $P_2$ , identifying the application process to be escorted.

To minimize the burden on flight software programmers and to enable the middleware itself to be onboard upgradable, the GSU middleware is implemented as a program that is entirely separated from the application software and is executed as an independent Unix process. At run time, the applications are not linked to each other, nor do they directly communicate with each other. Rather, the distributed application processes interact through the middleware instances mentioned above, as shown in Figure 4. Specifically, the interactions between an application and a middleware instance are realized using a TCP/IP socket.

The middleware is implemented in C++ because 1) C++ executes efficiently, and 2) C++ can be easily ported to the C++/VxWorks platform used for the JPL X2000 flight software development. The current version of the GSU Middleware includes:

- 1) A set of *invocable services* that execute a message-sending or -receiving request from an application process, in a manner adaptive to the confidence in the sender and/or receiving processes, and
- 2) A collection of the *MDCD modules* responsible for adjusting confidence in a process based on message-passing events and for making decisions on whether to take a checkpoint upon a message-passing event and

whether to roll back or roll forward during error recovery.

#### 4. SUMMARY AND FUTURE WORK

Unprotected onboard software upgrading may cause severe performance degradation or unavailability of mission-critical embedded systems. The current practice of embedded software upgrading for deep-space missions requires a full system reboot, disrupting mission operation and causing risks. Therefore, we have developed the GSU framework. The framework and resulting onboard GSU infrastructure (middleware) will enable

- Enhancement of system responsiveness to the evolving mission requirements or space environment conditions unanticipated prior to mission launch, through updating the embedded software in a timely and cost-effective manner.
- Improvement of system dependability by safeguarding an onboard software upgrade and preventing the mission from failure caused by faults in a new version or inconsistency between the new and old versions.
- Reduction of system operation and maintenance costs, via the exploitation of inherent and/or nondedicated resource redundancy and minimization of the performance overhead of onboard error containment and recovery mechanisms.

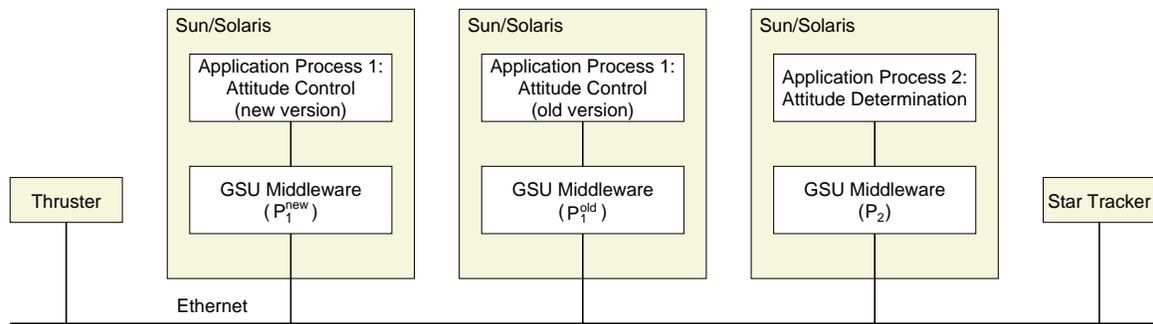


Figure 4: GSU Testbed

The GSU framework will also benefit a wide variety of commercial application domains, in addition to NASA applications. In particular, 1) the guarded software upgrading techniques will be useful for many other applications which are subject to frequent software upgrading and require high availability and/or safety, such as Internet services, transportation systems, airline reservation systems, telephone systems and medical systems, and 2) the methods of utilizing inherent, non-dedicated resource redundancies and our confidence-driven error containment and recovery algorithms will promote and accelerate the transfer of state-of-the-art fault tolerance techniques from research domain to real applications.

The following is our plan for the subsequent efforts:

1. Development of onboard validation techniques:
  - 1) Methods of onboard error log analysis.
  - 2) Statistical evaluation methods to validate the predicated reliability of the new version.
  - 3) Techniques that enable “non-stop validation” (i.e., after a detected failure of the new version, we let it recover by copying the state of the old version and re-starting execution from there).
  - 4) Decision algorithms for whether and when to enter the next GSU stage.
2. Development of onboard algorithms for determining when the old version can be taken offline (completely retires).
3. Development and implementation of fault-injection mechanisms for facilitating experimental assessment of the GSU methodology and middleware.

## REFERENCES

- [1] L. Alkalai and A. T. Tai, “Long-life deep-space applications,” *IEEE Computer*, vol. 31, pp. 37–38, Apr. 1998.
- [2] R. Baalke, Office of the Flight Operations Manager, “Mars Pathfinder update,” Mars Pathfinder Weekly Status Report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, June 1997.
- [3] L. Sha, J. B. Goodenough, and B. Pollak, “Simplex architecture: Meeting the challenges of using COTS in high-reliability systems,” *CrossTalk: The Journal of Defense Software Engineering*, vol. 11, pp. 7–10, Apr. 1998.
- [4] D. Powell *et al.*, “GUARDS: A generic upgradable architecture for real-time dependable systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 10, pp. 580–599, June 1999.
- [5] L. Alkalai, “NASA Center for Integrated Space Microsystems,” in *Proceedings of Advanced Deep Space System Development Program Workshop on Advanced Spacecraft Technologies*, (Pasadena, CA), June 1997.
- [6] S. N. Chau, L. Alkalai, A. T. Tai, and J. B. Burt, “Design of a fault-tolerant COTS-based bus architecture,” *IEEE Trans. Reliability*, vol. 48, pp. 351–359, Dec. 1999.
- [7] S. N. Chau *et al.*, “The implementation of a COTS based fault tolerant avionics bus architecture,” in *Proceedings of IEEE Aerospace Conference*, (Big Sky, MT), Mar. 2000.
- [8] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, “On low-cost error containment and recovery methods for guarded software upgrading,” in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, (Taipei, Taiwan), pp. 548–555, Apr. 2000.
- [9] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, “On the effectiveness of a message-driven confidence-driven protocol for guarded software upgrading,” in *Proceedings of the 4th IEEE In-*

*ternational Computer Performance and Dependability Symposium (IPDS 2000)*, (Chicago, IL), pp. 59–68, Mar. 2000.

- [10] J. F. Meyer, A. Movaghar, and W. H. Sanders, “Stochastic activity networks: Structure, behavior, and application,” in *Proc. Int’l Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.



Ann T. Tai received her Ph.D. in computer science from the University of California, Los Angeles. She is the President and a Sr. Scientist of IA Tech, Inc., Los Angeles, CA. Prior to 1997, she was associated with SoHaR Incorporated as a Sr. Research Engineer. She was an Assistant Professor at the University of Texas at Dallas during 1993. Her current research interests concern the design, development and evaluation of dependable, affordable and evolvable computing systems, and low-cost error containment and recovery methods. She authored the book, *Software Performability: From Concepts to Applications*, published by Kluwer Academic Publishers.



Leon Alkalai is the Center Director for the Center for Integrated Space Microsystems, a Center of Excellence at the Jet Propulsion Laboratory, California Institute of Technology. The main focus of the center is the development of advanced microelectronics, micro-avionics, and advanced computing technologies for future deep-space highly miniaturized, autonomous, and intelligent robotic missions.

He joined JPL in 1989 after receiving his Ph.D. in computer science from the University of California, Los Angeles. Since then, he has worked on numerous technology development tasks including advanced microelectronics miniaturization, advanced microelectronics packaging, reliable and fault-tolerant architectures. He was also one of the NASA appointed co-leads on the New Millennium Program Integrated Product Development Teams for Microelectronics Systems, a consortium of government, industry, and academia to validate technologies for future NASA missions in the 21st century.



Savio N. Chau is a system engineer at the Jet Propulsion Laboratory. He is currently developing scalable multi-mission avionics system architectures for the X2000 Program. He has been investigating techniques to apply low-cost commercial bus standards and off-the-shelf products in highly reliable systems such as long-life spacecraft. His research areas include scalable distributed system architecture, fault tolerance, and design-for-testability. He received his Ph.D. in computer science from the University of California, Los Angeles. He is a member of Tau Beta Pi and Eta Kappa Nu.