Proceedings of the 5th IEEE International Symposium on High Assurance Systems Engineering (HASE 2000),
Albuquerque, New Mexico, November 2000.

GSU Middleware Architecture Design*

Kam S. Tso Ann T. Tai
IA Tech, Inc.
Los Angeles, CA 90024

Leon Alkalai Savio N. Chau
Jet Propulsion Laboratory
Pasadena, CA 91109

William H. Sanders
University of Illinois
Urbana, IL 61801

(Extended Abstract)

1 Overview of GSU

NASA's future deep-space missions will require onboard software upgrade. A challenge that arises from this is that of guarding the system against performance loss caused by residual design faults in the new version of a spacecraft/science function. Accordingly, we have developed a methodology called "guarded software upgrading" (GSU) [1]. The GSU framework is based on the Baseline X2000 First Delivery Architecture, which comprises three high-performance computing nodes with local DRAMs and multiple subsystem microcontroller nodes that interface with a variety of devices. All nodes are connected by a high-speed fault-tolerant bus network that complies with the commercial interface standard IEEE 1394 [2].

Since application-specific techniques are an effective strategy for reducing fault tolerance cost, we exploit the characteristics of our target system and application. To ensure low development cost, we take advantage of inherent system resource redundancies as the means of fault tolerance. Specifically, from the software perspective, we make use of an earlier version of the software, in which we have high confidence due to its long onboard execution time, as a backup to protect the system when the new version enters into mission operations; from the hardware perspective, we make use of the processor that otherwise would be idle during a non-critical mission phase during which onboard software upgrade takes place, allowing concurrent execution of the new and old versions of the application software component that is undergoing an upgrade.

In order to mitigate the effect of residual faults in an upgraded software component, we take a crucial step in devising error containment and recovery methods by introducing the "confidence-driven" notion. This notion complements the message-driven (or "communication-induced") approach employed by a number of existing checkpointing protocols for tolerating hardware faults. In particular, we discriminate

i) Between internal and external messages in terms of their criticality to the mission, and

ii) Between the individual software components with respect to our confidence in their reliability.

The resulting protocol is thus both message-driven and confidence-driven (MDCD). Unlike traditional fault tolerance schemes for distributed systems, our error containment and recovery mechanisms do not involve process coordination or atomic action which usually results in significant performance overhead. Instead, we keep track of changes in our confidence in individual processes. These changes are due to knowledge about potential process state contamination caused by errors in the low-confidence component and message passing. This, in turn, permits the decisions on whether to take a checkpoint upon message passing, and whether to roll back or roll forward during recovery, to be made locally by individual processes, enabling cost-effective checkpointing and cascading-rollback free error recovery.

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.

2 Testbed

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 comply with the

*The work reported in this paper was supported in part by Small Business Innovation Research (SBIR) Contract NAS3-99125 from the Jet Propulsion Laboratory, National Aeronautics and Space Administration.

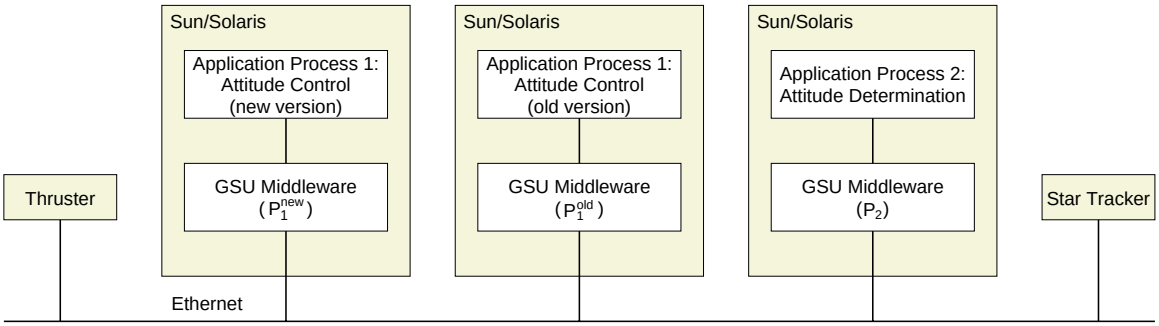

Figure 1: GSU Testbed

X2000 First Delivery avionics system architecture [2]. Specifically, the development is carried out on a network of Sun Ultra-10 workstations connected by a 10 Mbit/sec Ethernet. Each workstation is equipped with a 300 MHz UltraSPARC processor and 256 Megabytes of memory, and runs on the Solaris operating system. As shown in Figure 2, each workstation executes the GSU middleware that is instantiated from the generic code by a specific command-line argument, namely, $\mathtt{P}_1^{\mathtt{new}}$, $\mathtt{P}_1^{\mathtt{old}}$, or $\mathtt{P}_2$, 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 separate 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 cooperate through the middleware instances mentioned above, as shown in Figure 1. Specifically, the interactions between an application and middleware 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.

3 Middleware

As shown in Figure 2, 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.

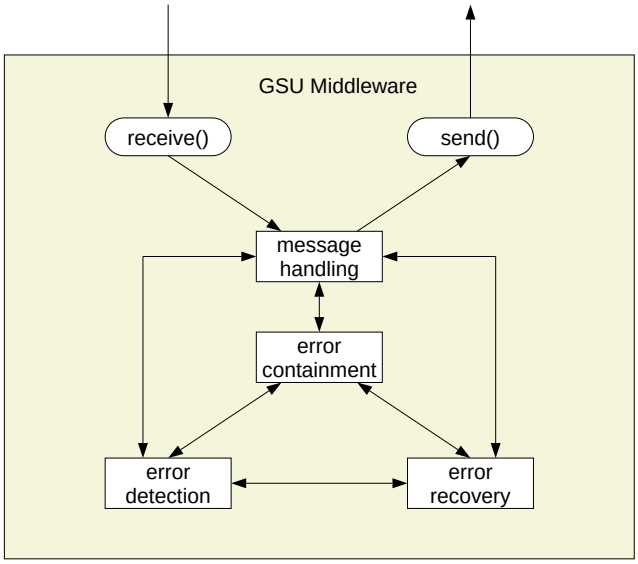

Figure 2: GSU Middleware Architecture

The confidence-adaptive requirements lead us to implement the invocable services as distributed objects of an overloading nature. For example, when a low-confidence process issues a request to send a message to a device, the invocable service `send()` that executes message-sending requests may make the requesting process undergo the following actions (with the support of the MDCD modules):

1) Perform an AT (acceptance test) and update the knowledge about state contamination of the sender process,

2) Send the application-purpose message to the device and the "passed AT" notification message to other processes, and

3) Establish a checkpoint.

A similar message-sending request from a high-confidence process may simply result in a message transmission. The overloading nature of the invocable services allows the MDCD protocol to be transparent to the programmer and allows the middleware itself to be generic, modifiable, and upgradable.

The MDCD software modules in the middleware are responsible for carrying out the message-driven confidence-driven protocol. The collection of modules includes:

Error-containment module: Responsible for monitoring message-passing events, signaling potential process state contamination, and making decisions on whether to establish a checkpoint upon a message-passing event.

Error-detection module: Responsible for making decisions on whether to perform an acceptance test on a process that issues a message-sending request, carrying out the acceptance test, and either marking the process as a high-confidence process upon a successful AT, or raising an error flag upon a failed AT.

Error-recovery module: Responsible for monitoring the error flags and making decisions on recovery mechanisms for individual processes upon a failed acceptance test, namely, decisions on whether to roll forward or roll back and whether it is necessary to re-send messages.

Message-handling module: Responsible for logging messages, updating message sequence numbers, piggybacking them to messages, and deleting the messages that are no longer needed in the message log of a process.

4 Demonstration

A synthetic spacecraft attitude determination and control application software [3] has been implemented as the demonstration system. The system has two application software components:

- The first application process performs the attitude control function by controlling the thrusters, and
- The second application process performs an attitude determination function based on the readings of the star tracker.

As shown in Figure 1, an upgraded version of the first application software components, P_1^{new}, that uses an improved algorithm, is executed on the first processor, while the old version of the first application software component (i.e., P_1^{old}) and the second application software component (i.e., P_2) are executed on the second and third processors of the testbed, respectively.

Figure 3 shows the graphical user interface (GUI) of the demonstration prototype. The current version of the GUI enables the user to visualize the MDCD protocol and the system behavior that is under the escort of the protocol. The subsequent version of the GUI, which is now being implemented, is aimed at facilitating measurement of the performance cost of the protocol and the performance penalty of error recovery.

The execution traces obtained from test runs of the preliminary version of the GSU Middleware show that the error containment and recovery protocol behaves as expected, and demonstrate that the middleware realization of the GSU infrastructure is feasible. On the other hand, we are aware that the preliminary version of the middleware has the following limitations: 1) Solaris OS is not sufficiently capable of supporting real-time computation with hard deadlines, and 2) hardware crashes cannot be handled by the protocol.

Nonetheless, those inadequacies are resolvable, because

- The GSU Middleware is implemented in such a way that it can be readily ported to the real-time OS VxWorks, which spaceborne computing systems employ, and
- The MDCD protocol is capable of cooperating with another checkpointing protocol to tolerate hardware faults. Development effort on an integrated protocol that is capable of tolerating both software and hardware faults is under way.

References

[1] 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.

[2] 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.

[3] E. Shalom, "The input output unit for the attitude and articulation control subsystem on the Cassini spacecraft," in *Proceedings of the 14th Digital Avionics Systems Conference*, (Cambridge, MA), pp. 347–352, Nov. 1995.

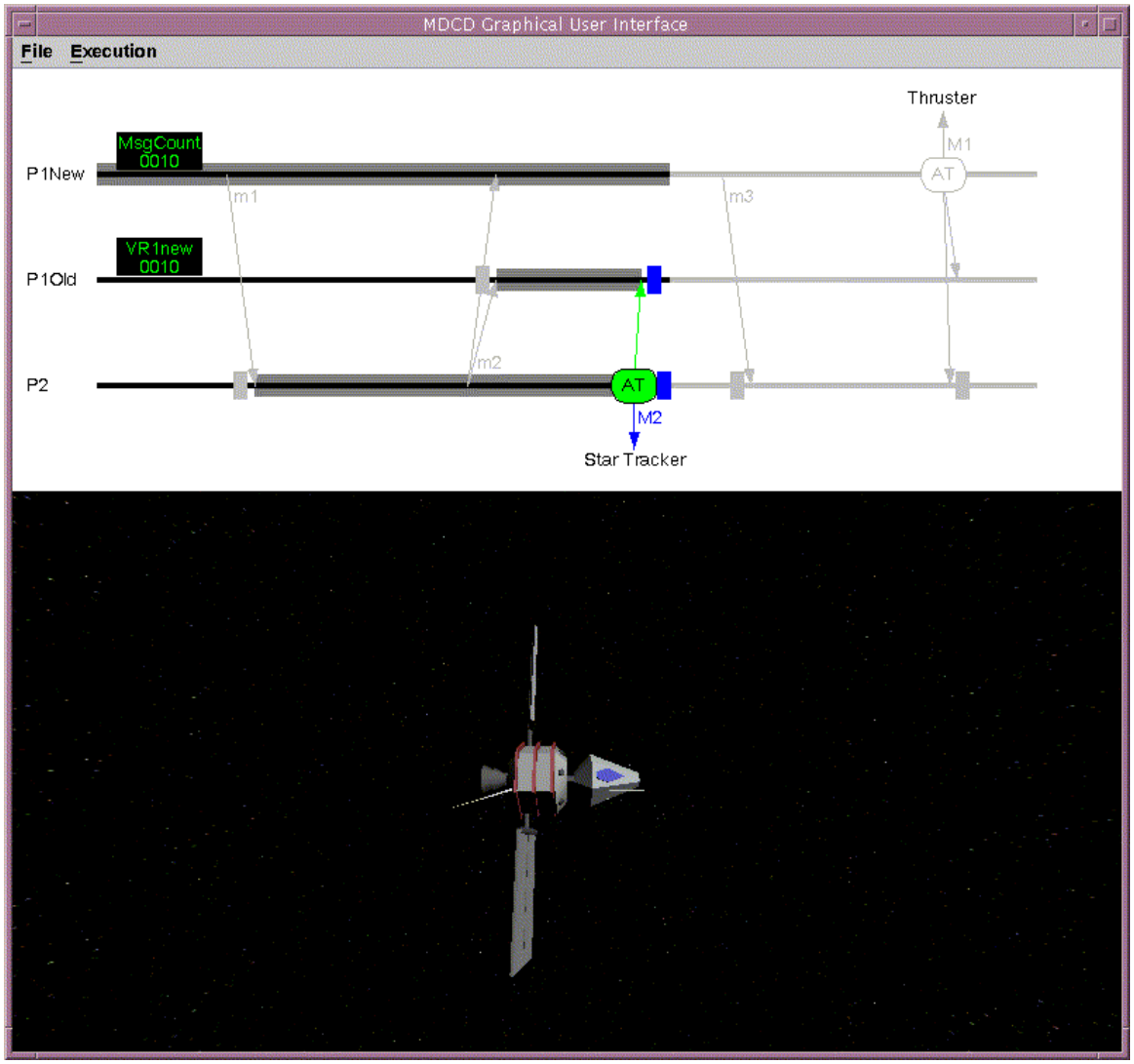

Figure 3: Demonstration Application: Spacecraft Attitude Determination and Control