Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, April 2000.

On Low-Cost Error Containment and Recovery Methods for Guarded Software Upgrading[*]

Ann T. Tai Kam S. Tso
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

William H. Sanders
Elec. & Comp. Eng. Dept.
University of Illinois
Urbana, IL 61801

Abstract

To assure dependable onboard evolution, we have developed a methodology called guarded software upgrading (GSU). In this paper, we focus on a low-cost approach to error containment and recovery for GSU. To ensure low development cost, we exploit inherent system resource redundancies as the fault tolerance means. In order to mitigate the effect of residual software faults at low performance cost, 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 between the individual software components with respect to our confidence in their reliability, and keep track of changes of our confidence (due to knowledge about potential process state contamination) in particular processes. This, in turn, enables the individual processes in the spaceborne distributed system to make decisions locally, at run-time, on whether to establish a checkpoint upon message passing and whether to roll back or roll forward during error recovery. The resulting message-driven confidence-driven approach enables cost-effective checkpointing and cascading-rollback free recovery.

1 Introduction

New-generation spaceborne computing systems, such as NASA/JPL's X2000 for multiple deep-space missions, must have the ability to accomplish performance and dependability enhancement during a long-life mission [1]. This capability is referred to as *evolvability*. Concepts related to evolvability include hardware reconfigurability and software upgradability.

A challenge that arises from onboard software upgrade is that of guarding the system against performance loss caused by residual design faults introduced by the addition or modification of a spacecraft/science function. Unprotected software upgrades may cause severe damage to a mission. For example, NASA experienced a gap in fault protection on April 10, 1981, when a timely synchronization check was omitted after the addition of an alternate reentry program [2]. As a result, the first flight of the US space shuttle program was aborted 19 minutes before launch. The risk of unprotected software upgrade is further exemplified by MCI WorldCom's recent 10-day frame relay outage [3]. The outage began on August 5, 1999, four weeks after an upgrade to a new switching software to allow the network to handle increased traffic. The incident affected about 15% of MCI's network and 30% of its customers who rely on the high-speed frame relay.

To avoid the adverse impacts of unsuccessful upgrades, researchers have investigated into dependable system upgrade methods. For example, Sha *et al.* developed Simplex architecture which employed "analytic redundancy" to enable error recovery for upgraded software [4]. Powell *et al.* at LAAS-CNRS defined a Generic Upgradable Architecture for Real-Time Dependable Systems (GUARDS) [5]. Both Simplex and GUARDS were aimed at critical real-time applications and employed state-of-the-art fault tolerance techniques. Nonetheless, both architectures required special development effort for dedicated system resource redundancy. Furthermore, the problems associated with fault tolerance in distributed systems, such as error contamination caused by process interaction, were not of particular concern to those projects or the prior work in dynamic program modification [6]. In contrast to Simplex and GUARDS, the X2000 system has severe cost constraints imposed on it, precluding solutions which rely on dedicated resource redundancies such as multiple software versions that require expensive development effort. Moreover, the X2000 system has a distributed architecture for which error contamination among interacting processes is a major concern for fault tolerance.

To accommodate the requirements from the X2000 architecture and applications, we have developed a methodology called *guarded software upgrading* (*GSU*). Since application-specific techniques are an effective strategy of reducing fault tolerance cost [7], 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 software perspective, we make use of an earlier version, 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 mission operation; from 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 which is undergoing an upgrade. An informal discussion based on the initial version of the GSU methodology was presented in [8]. The central purpose of this paper is to elaborate and analyze the error containment and recovery methods for GSU in further detail and depth.

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

The error containment and recovery methods for GSU are devised based on novel adaptation and integration of the enabling techniques in the areas encompassing software fault tolerance, checkpointing and message logging based recovery, and distributed computing. 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. 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 of our confidence (due to knowledge about potential process state contamination caused by errors in the low-confidence component and message passing) in particular processes. 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.

The remainder of the paper is organized as follows. Section 2 provides an overview of the GSU framework, followed by Section 3 which elaborates the error containment and recovery algorithms for guarded software upgrading. Section 4 presents the formal proofs that verify the correctness of the algorithms. The concluding remarks highlight the significance of this effort and outline our plan for subsequent research.

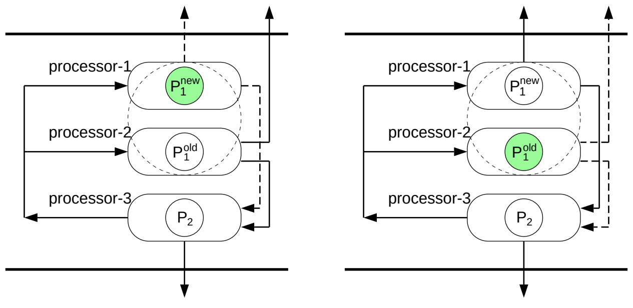
2 Motivation and Overall Framework

As an engineering model intended to service multiple deep-space missions, the X2000 architecture must accommodate a diversity of 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 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 [9]. 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 from 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 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^{new}	The process corresponding to the new version of an application software component.
P_1^{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 1 illustrates the two-stage approach. As shown in Figure 1(a), during onboard validation the outgoing messages of the shadow process P_1^{new} are suppressed but selectively logged (as shown by the dashed lines with arrows), while P_1^{new} receives the same incoming messages that the active process P_1^{old} does (as shown by the solid lines with arrows). Thus, P_1^{new} and P_1^{old} can perform the same computation based on identical input data. Note that each of the dashed circles that encapsulate P_1^{new} and P_1^{old} indicates that the two processes are created by two different versions of the same application software component.



(a) Onboard Validation Stage (b) Guarded Operation Stage

Figure 1: Two-Stage Approach to GSU

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^{new} to enter mission operation. If onboard validation completes successfully, then P_1^{new} and P_1^{old} switch their roles and enter the guarded operation stage. 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 1(b), during the guarded operation, P_1^{new} actually influences the external world and interacts with process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or devices are now suppressed and logged. Should an error of P_1^{new} be detected, P_1^{old} will take over P_1^{new}'s active role, and the system will resume its normal mode until the next upgrade attempt. The guarded operation is equipped with a set of low-cost error containment and recovery mechanisms which are elaborated in the next section.

3 Algorithms

3.1 Basics

A major difficulty in error recovery for embedded systems is that we are unable to roll back the effect of a computation error after it propagates to an external device. Since error propagation in a distributed system is, in general, caused by message passing, the invocations of the two major functions of the error containment and recovery protocol for GSU, namely, acceptance test (AT) and checkpointing, are all associated with the message sending or receiving actions. We call the messages sent by processes to devices and the messages between processes *external messages* and *internal messages*, respectively. In embedded systems, external messages are significantly more critical than internal messages because i) they directly influence the mission operation and functions, and ii) their adverse effects can not be reversed through rollback. Hence, in our low-cost error containment and recovery protocol, ATs are only invoked to validate the external messages from the processes that are potentially contaminated (see Section 3.2 for the definition of *potentially contaminated process state*). Further, P_1^{old} does not perform ATs because its external messages will not be released to devices during guarded operation. On the other hand, when P_1^{new} or P_2 passes an AT successfully, it sends a notification message to P_1^{old} to let it update its knowledge about the validity of process state and messages.

The following are the assumptions upon which we devise the error containment and recovery algorithms:

A1) The old version of a software component that has a sufficiently long onboard execution time can be considered significantly more reliable than the upgraded version newly installed through uploading.

A2) An erroneous state of a process is likely to affect the correctness of its outgoing messages, while an erroneous message received by an application software component will result in process state contamination.

A3) The error detection mechanism, an acceptance test (AT), has a high coverage (the conditional probability that the testing mechanism will reject a computation result given that the result is erroneous).

A1 implies that the likelihood that an error condition which occurs in the old version of an application software component can be considered negligible, suggesting that P_1^{old} and P_2 need not be treated by the protocol as possible sources of process state contamination. A2 implies that if an outgoing message is validated by AT, then the process state of the sender process and all the messages sent or received prior to performing the AT can be considered *non-contaminated* and *valid*, respectively. A3 suggests that the release of an erroneous command to an external device is unlikely to occur. Note that A1 is applicable not only to the upgrades for performance tuning and accuracy improvement but also to the *scheduled upgrades* aimed at fault removal [10]. The rationale is that the deep-space application software components which have sufficiently long onboard execution times are expected to be highly reliable, and that the scheduled onboard fault removal usually deals with the isolated faults that result in infrequent error conditions tolerable by the spaceborne system. On the other hand, the new version with a known fault removed may contain new undiscovered faults.

3.2 Error Containment

During guarded operation, P_1^{new} actually influences the external world and interacts with the process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or external subsystems are suppressed and logged (although P_1^{old} receives all the messages that P_1^{new} does and fully executes in the background). The algorithms rely on three key entities, namely,

1. A *dirty bit* (`dirty_bit`), which keeps track of potential process state contamination.
2. A *message count* (`msg_count`), which keeps track of the number of messages that are sent by a particular process (equivalent to the sequence number of its last message).
3. A *valid message register* ($\mathtt{VR}_1^{\mathtt{new}}$), which keeps track of the sequence number of the last message that has been sent by P_1^{new} and directly or indirectly validated by AT.

These entities are maintained by individual processes locally while the information kept is shared between certain processes through message piggybacking. To facilitate error containment and recovery efficiency, we enforce the following confidence-driven checkpointing rule (the necessary and sufficient condition for checkpointing):

Checkpointing Rule: We save the state of a process via checkpointing if and only if the process is at one of the following points: 1) immediately before its state becomes potentially contaminated, or 2) right after its state gets validated as a non-contaminated state.

By "a potentially contaminated process state," we mean 1) the process state of the low-confidence software component P_1^{new}, or 2) a process state that reflects the receipt of a not-yet-validated message that is sent by a process when its process state is potentially contaminated. Figure 2 illustrates the above concepts. The horizontal lines in Figure 2 represent the software executions along the time horizon. Each of the shaded regions represents the execution interval during which the state of the corresponding process is potentially contaminated. Note that 1) P_1^{new} is always considered potentially contaminated, 2) P_2 is treated as potentially contaminated after it receives an application-purpose message from P_1^{new} and before P_2 passes the subsequent AT (or receives a notification

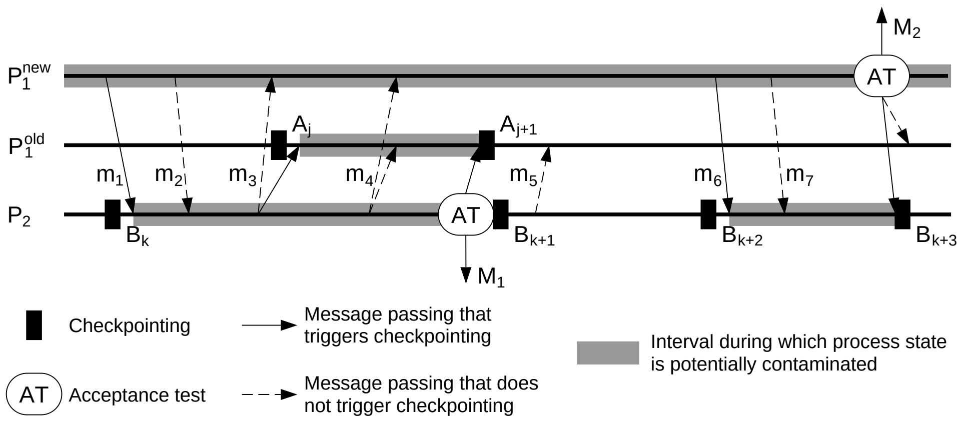

M2
P1new
AT
Aj
Aj+1
P1old
m1
m2
m3
m4
m5
m6
m7
P2
AT
Bk
Bk+1
Bk+2
Bk+3
M1
Checkpointing
Message passing that triggers checkpointing
Interval during which process state is potentially contaminated
AT
Acceptance test
Message passing that does not trigger checkpointing

Figure 2: Checkpoint Establishment

message reporting that P_1^{new} passes AT), and 3) P_1^{old} is regarded as potentially contaminated after it receives a message sent by P_2 when its process state is potentially contaminated and before P_1^{old} is notified that P_2 (or P_1^{new}) has passed the subsequent AT.

In the diagram, checkpoints B_k, A_j, and B_{k+2} are established immediately before a process state becomes potentially contaminated, while B_{k+1}, A_{j+1}, and B_{k+3} are established right after a process state is validated. While all these checkpoint establishments are triggered by the events of potential process state contamination and process state validation which change our confidence in a process, these triggering events themselves are induced by message passing. In other words, a message passing event will not trigger a process to establish a checkpoint unless the event alters our confidence in the process state(s) — to make a potentially contaminated process state become a validated state or vice versa. Therefore, our algorithms are both message-driven and confidence-driven. Consequently, as shown in Figure 2, the message passing events indicated by the dashed lines with arrows will not trigger checkpointing. And based on our assumption on the relationship between message and process state, m_1, m_2, m_3, and m_4 become *valid messages* after P_2 subsequently passes AT. Likewise, m_6 and m_7 become valid messages after P_1^{new} subsequently passes AT. On the other hand, the message m_5 sent by P_2 between checkpoints B_{k+1} and B_{k+2} is "inherently" considered a valid message because it is generated by P_2 when the process state of P_2 is not potentially contaminated.

The error containment algorithms for P_1^{new}, P_1^{old}, and P_2 are shown in Figures 3, 4, and 5, respectively. Recall that our key strategy for achieving error containment and recovery efficiency is to discriminate between the individual software components with respect to our confidence in their reliability. Accordingly, the algorithms are devised in a manner such that the three concurrent processes deal with the key entities in an asymmetric fashion, as described in the following.

As shown in Figure 3, P_1^{new} maintains its msg_count, which keeps track of the outgoing messages of P_1^{new} and is incremented when P_1^{new} sends an application-purpose message. To assure recoverability (see Section 4), P_1^{new} attaches the value of msg_count to its outgoing messages sent to other processes.

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (AT(m) == success) {
      // P1new maintains its msg count and
      // conveys it to P2 and P1old for
      // recovery purpose
      msg_count++;
      msg_send(m, null, device);
      // inform P1old and P2 that prior
      // messages are valid
      msg_send("passed_AT", msg_count, P1old);
      msg_send("passed_AT", msg_count, P2);
    } else {
      error_recovery(P1old, P2);
      exit(error);
    }
  } else { // m is an internal message
      msg_count++;
      msg_send(m, msg_count, P2);
  }
}
if (incoming_message_m_arrives) {
  application_msg_reception(m);
}
```

Figure 3: Error Containment Algorithm for P_1^{new}

P_1^{old} also maintains a msg_count (see Figure 4), which keeps track of outgoing messages of P_1^{old} and is incremented after P_1^{old} generates an outgoing message and before the message is logged. In addition, P_1^{old} has a valid message register VR_1^{new}, which keeps track of validated messages sent by P_1^{new}. VR_1^{new} gets updated when P_1^{old} receives a "Passed AT" notification message from P_1^{new} or P_2 using the attached information. P_1^{old} uses its dirty_bit to maintain knowledge about potential contamination of its own process state. Upon receiving a "Passed AT" message from P_1^{new} (or from P_2) or a message from P_2, which will turn the otherwise potentially contaminated state of P_1^{old} into a non-contaminated state or vice versa, dirty_bit is reset or set to 1, respectively.

P_2 maintains its msg_count and uses it to keep track of the outgoing messages of P_1^{new}, as shown in Figure 5. Upon receiving an application-purpose message or a "Passed AT" notification

```
if (outgoing_message_m_ready) {
  // msg_count keeps track of $P_1^{old}$'s own messages
  msg_count++;
  // suppress and log the outgoing message
  msg_log(m, msg_count);
}
if (incoming_message_m_arrives) {
  if (m.body == "passed_AT") {
    // $P_1^{new}$ or $P_2$ reports a successful AT
    $VR_1^{new}$ = m.msg_count; // last valid msg of $P_1^{new}$
    if (dirty_bit == 1) {
      dirty_bit = 0;
      checkpointing($P_1^{old}$);
    }
  } else { // application-purpose message from $P_2$
    // check the piggybacked dirty bit and
    // own process state
    if (m.dirty_bit == 1 && dirty_bit == 0) {
      checkpointing($P_1^{old}$);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 4: Error Containment Algorithm for P_1^{old}

message from P_1^{new}, msg_count is updated based on the parameter value attached to the incoming message. Upon passing its AT, P_2 attaches the value of msg_count to the "Passed AT" notification message to P_1^{old} (to enable P_1^{old} to update its VR_1^{new}). P_2 also has a dirty_bit which maintains knowledge about potential process state contamination. Upon passing AT or receiving a "Passed AT" message from P_1^{new}, P_2 resets dirty_bit; whereas upon receiving an application-purpose message from P_1^{new}, P_2 sets dirty_bit to 1. The value of dirty_bit is piggybacked to every application-purpose message P_2 sends to P_1^{old}, in order to let P_1^{old} be aware of potential process state contamination and make a correct decision on whether to roll back or roll forward upon the invocation of error recovery. We omit detailed description of how the processes perform checkpointing because the algorithms shown in Figures 4 and 5 are fairly self-explanatory in illustrating how the confidence-driven checkpointing rule described at the beginning of this subsection is implemented.

3.3 Error Recovery

Error recovery actions are also message-driven and confidence-driven in the sense that the AT-based error detection takes place when a potentially contaminated process (P_1^{new} or P_2) attempts to send an external message. Upon the detection of an error, P_1^{old} will take over P_1^{new}'s active role and resume computation with P_2. Since the error containment algorithms enable P_1^{old} and P_2 to maintain their knowledge about potential process state contamination and message validity, relatively simple error recovery algorithms can be devised for P_1^{old} and P_2, as shown in Figures 6 and 7, respectively. In particular, by checking their dirty bits locally, both P_1^{old} and P_2 are able to make their decisions on rollback or roll-forward in a straightforward manner. Accordingly, there are three possible scenarios during error recovery:

Scenario 1: Both P_1^{old} and P_2 roll back to their most recent checkpoints.

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        dirty_bit = 0;
        // msg_count of $P_2$ keeps track of msg
        // sequence number of $P_1^{new}$
        msg_send(m, null, device);
        msg_send("passed_AT", msg_count, $P_1^{old}$);
        checkpointing($P_2$);
      } else {
        error_recovery($P_1^{old}$, $P_2$);
      }
    } else {
      // outgoing msg from a clean process state,
      // no check needed
      msg_send(m, null, device);
    }
  } else { // internal message
    msg_send(m, null, $P_1^{new}$);
    // piggybacking dirty_bit to msg to $P_1^{old}$ to
    // signal possible contamination
    m = append(m, dirty_bit);
    msg_send(m, null, $P_1^{old}$);
  }
}
if (incoming_message_m_arrives) {
  // must be from $P_1^{new}$
  msg_count = m.msg_count;
  if (m.body == "passed_AT") {
    if (dirty_bit == 1) {
      dirty_bit = 0;
      checkpointing($P_2$);
    }
  } else {
    if (dirty_bit == 0) {
      // checkpointing before getting "dirty"
      checkpointing($P_2$);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 5: Error Containment Algorithm for P_2

Scenario 2: Both P_1^{old} and P_2 roll forward.

Scenario 3: P_2 rolls back to its most recent checkpoint, while P_1^{old} rolls forward.

The scenario in which P_1^{old} rolls back and P_2 rolls forward can never happen. This is because P_1^{new} would never send application-purpose messages to P_1^{old}. Accordingly, the process state of P_1^{old} will not become potentially contaminated unless it receives a message sent by P_2 when the process state of P_2 is potentially contaminated, which results in Scenario 1.

After the rollback action or the roll-forward decision, P_1^{old} will compare the value of msg_count with VR_1^{new}. Depending upon whether the former is greater or less than the latter, P_1^{old} will send out the messages (with sequence numbers greater than the value of VR_1^{new}) in the message log or further suppress its messages, respectively, until the values of msg_count and VR_1^{new} match.

```
if (dirty_bit == 1) {
rollback(most_recent_ckpt);
}
// switch role with P_1^new and go forward
switch_to_active(VR_1^new, msg_count);
continue;
```

Figure 6: Error Recovery Algorithm for P_1^{old}

```
if (dirty_bit == 1) {
rollback(most_recent_ckpt);
}
// go forward
continue;
```

Figure 7: Error Recovery Algorithm for P_2

4 Discussion

Clearly, our error containment and recovery algorithms are based on the integration and adaptation of a number of existing enabling techniques. For example, we employ multiple software versions (which are inherently available to us) and acceptance tests as suggested by N-version programming (NVP) [11] and recovery blocks (RB) [12], respectively. But rather than being driven by program structure as suggested by these traditional software fault tolerance techniques, checkpoint establishment and acceptance test in our error containment algorithms are triggered by message passing events, which is a strategy adapted from the checkpointing techniques for hardware error recovery [13]. However, checkpointing techniques for hardware error recovery are concerned solely with the consistency between process states for assuring correct recovery from hardware faults. In contrast, since our objective is to mitigate the effect of residual faults in an upgraded software component, our concern is the consistency among the views of different processes on process state integrity, especially on the *valid messages* (see Sections 3.1 and 3.2) reflected in the process states. Thus, the notion of "confidence-driven" is again the key to our adaptation of enabling techniques.

Specifically, we adapt the terminologies and definitions in [13, 14] as follows: A *global state* includes the state of each process that is executing the application, and possibly messages between interacting processes and *information concerning their verified correctness*. A valid checkpointing mechanism must assure that it is always possible for the error recovery mechanism to bring the system into a global state that satisfies the following two properties:

Consistency If m is reflected in the global state as a valid message received by a process, then m must also be reflected in the global state as a valid message sent by the sender process.

Recoverability If m is reflected in the global state as a valid message sent by a process, then m must also be reflected in the global state as a valid message received by the receiving process(es) or the error recovery algorithm must be able to restore the message m.

When two or more process states (or checkpoints reflecting the process states) comprise a global state that satisfies the consistency property, we say that these process states are *globally consistent*, or that they comprise a *consistent global state*. It is worth noting that upon error recovery, P_1^{old} takes over P_1^{new}'s active role and thus becomes the "sender process" of the messages sent by P_1^{new} and reflected as valid messages in the global state. Before we proceed to explain why correct recovery can be achieved by the above algorithms, we introduce the following checkpoint classification based on the checkpointing rule described in Section 3.2:

Type-1 checkpoint: The checkpoint that is established, by a process that is otherwise considered not potentially contaminated, upon its receipt of a message from a process whose state is potentially contaminated.

Type-2 checkpoint: The checkpoint that is established, by a process that is otherwise considered potentially contaminated, upon its passing an AT or receiving a "Passed AT" notification message.

Theorem 1 and Corollaries 1 and 2 (presented below) show that the recovery decisions (rollback or roll-forward) made locally by the individual processes satisfy the global state consistency property. As the error containment algorithms require a potentially contaminated process to perform AT for its outgoing external message and to keep other processes informed of successful external message sending, the information regarding external message passing reflected in a global state will never violate global state consistency. Therefore, the following theorem proofs are concerned solely with internal messages.

Theorem 1 *The most recent checkpoints of P_1^{old} and P_2 are always globally consistent.*

Proof. Per the necessary and sufficient condition used by our algorithms for checkpoint establishment (see Section 3.2), the process states of P_1^{old} and P_2 reflected in their most recent checkpoints are never potentially contaminated. If m_i (i is the sequence number) is reflected in the most recent checkpoint of P_2 as a valid message from P_1^{new}, then a successful AT must have been performed (by P_1^{new} or P_2) after the receipt of m_i. Upon receiving the corresponding "Passed AT" message (from P_1^{new} or P_2), P_1^{old} will update its valid message register VR_1^{new} using the information attached to the notification message and establish a checkpoint. Therefore, m_i must also be reflected in the most recent checkpoint of P_1^{old} as a valid message sent to P_2 (by P_1^{new}).

Conversely, if a message m_j is reflected in the most recent checkpoint of P_1^{old} as a valid message received from P_2, m_j must be sent by P_2 when its process state is not potentially contaminated. Then if the most recent checkpoint of P_1^{old} is Type-1, this checkpoint must be established upon the receipt of a message m_k ($k > j$) sent by P_2 when its process state is potentially contaminated. It follows that P_2 must have established a checkpoint upon receiving a message from P_1^{new} after sending m_j but before sending m_k. Hence, m_j will be reflected in the most recent checkpoint of P_2 as a valid message that has been sent. On the other hand, if the most recent checkpoint of P_1^{old} (in which m_j is reflected as a valid message from P_2) is Type-2, this checkpoint must be established upon the receipt of a "Passed AT" notification message from P_1^{new} or P_2. Per our error containment algorithms, the successful AT will also trigger P_2 to establish a checkpoint in which m_j is reflected as a valid message that has been sent. Then, the theorem follows from the definition of global state consistency. Q.E.D.

Corollary 1 *The process states of P_1^{old} and P_2 at time t that are not potentially contaminated are globally consistent.*

Proof. If $\mathrm{P}_1^{\mathrm{old}}$ and P_2 are not involved in any interprocess communication since their most recent checkpoints, the information about message sending and receiving reflected in the process states of the two processes at time t will remain the same as those reflected in their most recent checkpoints. Then the corollary follows from Theorem 1.

If $\mathrm{P}_1^{\mathrm{old}}$ and P_2 are involved in interprocess communication since their most recent checkpoints, then the interprocess communication will not involve any messages from $\mathrm{P}_1^{\mathrm{new}}$ because the process states of $\mathrm{P}_1^{\mathrm{old}}$ and P_2 at time t are not potentially contaminated. More precisely, the interprocess communication will involve only the message(s) sent by P_2 to $\mathrm{P}_1^{\mathrm{new}}$ and $\mathrm{P}_1^{\mathrm{old}}$. These messages are implicitly considered as valid messages sent and received by P_2 and $\mathrm{P}_1^{\mathrm{old}}$, respectively, because the dirty bit of each of the processes (which are not potentially contaminated) must remain zero since its most recent checkpoint. Then, the corollary follows from the definition of global state consistency. Q.E.D.

Corollary 2 *If at time t the process state of P_2 is potentially contaminated but that of P_1^{old} is not, then the process state of P_1^{old} at time t and the process state of P_2 reflected in its most recent checkpoint (relative to t) are globally consistent.*

Proof. If at time t the process state of P_2 is potentially contaminated, then the most recent checkpoint (relative to t) of P_2 must be Type-1. Because the process states of P_2 and $\mathrm{P}_1^{\mathrm{old}}$ immediately before P_2 establishes the Type-1 checkpoint are not potentially contaminated, they are globally consistent per Corollary 1. If a message m_i is reflected in the process state of $\mathrm{P}_1^{\mathrm{old}}$ at time t as a valid received message from P_2, m_i must also be reflected in the most recent checkpoint of P_2 as a valid outgoing message (sent to $\mathrm{P}_1^{\mathrm{old}}$ and $\mathrm{P}_1^{\mathrm{new}}$). This is because m_i must be sent by P_2 before it establishes its most recent checkpoint; otherwise the process state of $\mathrm{P}_1^{\mathrm{old}}$ at time t would have been potentially contaminated, which is a contradiction. Conversely, if the most recent checkpoint of P_2 reflects a valid message m_j that is received from $\mathrm{P}_1^{\mathrm{new}}$, then m_j must be reflected (through the value of $\mathsf{VR}_1^{\mathsf{new}}$) in the process state of $\mathrm{P}_1^{\mathrm{old}}$ at time t as a valid message sent by $\mathrm{P}_1^{\mathrm{new}}$. This is because m_j must have been validated by AT (performed by P_2 or $\mathrm{P}_1^{\mathrm{new}}$) right before the establishment of an earlier (Type-2) checkpoint of P_2; also, after the successful AT, $\mathrm{P}_1^{\mathrm{old}}$ must receive the sequence number of the last validated message of $\mathrm{P}_1^{\mathrm{new}}$ that is attached to the corresponding notification message, and update its $\mathsf{VR}_1^{\mathsf{new}}$ accordingly. Hence, the corollary. Q.E.D.

Recoverability is assured by 1) the confidence-driven "rollback or roll-forward" decisions by $\mathrm{P}_1^{\mathrm{old}}$ and P_2, and 2) the message log of $\mathrm{P}_1^{\mathrm{old}}$, and the two key entities, namely, `msg_count` and $\mathsf{VR}_1^{\mathsf{new}}$. Recall that our algorithms yield the following system behavior: i) upon a successful AT performed by $\mathrm{P}_1^{\mathrm{new}}$ or P_2, $\mathrm{P}_1^{\mathrm{old}}$ will update its valid message register $\mathsf{VR}_1^{\mathsf{new}}$ (using the information attached to the notification message), and subsequently establish a checkpoint; and ii) during error recovery, a process will at most rollback to its most recent checkpoint. Accordingly, the value of $\mathsf{VR}_1^{\mathsf{new}}$ identifies the valid messages which are sent by $\mathrm{P}_1^{\mathrm{new}}$ and "permanently received" by another process or an external subsystem (i.e., they will never be "unreceived" through rollback recovery). On the other hand, `msg_count` of $\mathrm{P}_1^{\mathrm{old}}$ may be decremented or remain the same after recovery, depending upon whether $\mathrm{P}_1^{\mathrm{old}}$ has to roll back. As explained earlier, $\mathrm{P}_1^{\mathrm{old}}$ will "re-send" the messages in its message log or suppress the messages it intends to send after recovery, if the value of `msg_count` is greater or less than that of $\mathsf{VR}_1^{\mathsf{new}}$, respectively, until the two values match. Therefore, our algorithms guarantee recoverability.

In addition to the correctness verification of the algorithms, we have also carried out a model-based study to assess the performance cost of the protocol quantitatively [15]. Specifically, in terms of mean rollback distance and task completion time, the model-based study contrasts the performance cost of our protocol to that resulting from the traditional software fault tolerance techniques which involve process coordination and pre-structured checkpointing and rollback actions. The quantitative results show that our protocol leads to significant performance cost reduction due to the dynamic nature of its error detection, containment, and rollback/roll-forward recovery mechanisms. Because of space limitations, we omit the detailed discussion.

It is also important to validate the low-cost protocol's ability to enhance reliability for onboard software upgrading. Recall that the derivation of our protocol is based on three basic assumptions, namely, A1, A2, and A3 (Section 3.1). These assumptions are consistent with those related hypotheses made and used by researchers and practitioners in the areas of software engineering, software fault tolerance, and distributed systems. Nonetheless, in order to validate the effectiveness of the 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 assumptions [16]. The quantitative results confirm that the message-driven confidence-driven approach is indeed effective.

5 Concluding Remarks

With the goal of avoiding or minimizing mission performance degradation due to system failure caused by residual faults in an upgraded software component, we have proposed a low-cost error containment and recovery protocol. This effort makes two important contributions. First, in order to mitigate the effect of residual faults in an upgraded software component, we introduce the notion of "confidence-driven," which complements the message-driven approach employed by a number of existing checkpointing protocols for tolerating hardware faults. In particular, we discriminate between the individual software components with respect to our confidence in their reliability, and keep track of changes of our confidence (due to knowledge about potential process state contamination caused by errors in the low-confidence component and message passing) in particular processes. As a result, the combined message-driven confidence-driven approach eliminates the needs for 1) process coordination or atomic action, and 2) prestructuring checkpoint establishment and recovery actions in the application programs. In other words, the confidence-driven nature of the protocol 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, facilitating cost-effective checkpointing and cascading-rollback free error recovery.

Second, this effort demonstrates the appropriateness of utilizing the pertinent features of a target system and application in devising error containment and recovery methods, for the objective of reducing development and performance costs. With regard to the application addressed in this paper, the following pertinent features have enabled us to pursue a low-cost approach: 1) inherent system resource redundancies are available, 2) software upgrade is done in an incremental fashion, 3) internal and external messages have differing criticalities to the mission, and 4) among the interacting processes in the distributed system, a subset of them deserves our high confidence.

Furthermore, the error containment and recovery methods described in this paper can be rather easily extended and generalized. In particular, we plan to extend and apply the methodology to general distributed systems in which we can discriminate between interacting software components with respect to their reliability. Indeed, a number of factors other than upgrading may result in differing levels of confidence in different software components in a system. For example, we may have better confidence in a software component with lower complexity or higher testability. Software components in a distributed application may thus be categorized into two groups according to our confidence in their reliability. In a manner analogous to the GSU methodology, the high-confidence group can be exploited to enhance the efficiency of error containment and recovery. Specifically, if the size of the low-confidence group m remains 1 (only one low-confidence software component among the cooperating application software components), the error containment algorithm for P_2 described in Section 3.2 can be adapted with only minor modifications, such that P_2 and the additional application software components $P_3, \ldots, P_n$ can apply the same modified algorithm. The modified algorithm will also be applicable to the case in which we have multiple low-confidence software components ($m > 1$) if we treat these components as a single group during error recovery. On the other hand, for more efficient onboard guarded software upgrading, it is desirable, upon error detection, to isolate and configure out the faulty software component from the low-confidence group. To realize this requires further modifications to the error containment algorithms and the addition of a diagnosis procedure for fault isolation. One way is to use the "colored dirty bits," which would enable us to distinguish between different sources of process state contamination and to preserve the simplicity and low performance cost of the protocol.

It is also worth noting that the dynamic nature of the protocol allows the error containment and recovery mechanisms to be transparent to the programmer and facilitates a middleware implementation. Currently, we are prototyping the protocol in a middleware architecture that implements the GSU methodology. After the prototyping effort, performance and reliability benchmarking will be conducted through fault injection; the results will be used to further analyze and improve the algorithms and methodology.

References

[1] L. Alkalai and A. T. Tai, "Long-life deep-space applications," *IEEE Computer*, vol. 31, pp. 37–38, Apr. 1998.

[2] A. Avižienis, "Towards systematic design of fault-tolerant systems," *IEEE Computer*, vol. 30, pp. 51–58, Apr. 1997.

[3] J. Rendleman, "MCI WorldCom blames Lucent software for outage," in *PC Week*, Ziff-Davis, August 16, 1999.

[4] 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, Apr. 1998.

[5] 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.

[6] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53–65, Mar. 1993.

[7] J. A. Abraham, "The myth of fault tolerance in complex systems," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, (Hong Kong, China), Dec. 1999.

[8] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "On-board guarded software upgrading for space missions," in *Proceedings of the 18th Digital Avionics Systems Conference*, vol. 2, (St. Louis, MO), pp. 7.B.4–1–7.B.4–8, Oct. 1999.

[9] 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.

[10] C. T. Baker, "Effects of field service on software reliability," *IEEE Trans. Software Engineering*, vol. 14, pp. 254–258, Feb. 1988.

[11] A. Avižienis, "The N-Version approach to fault-tolerant software," *IEEE Trans. Software Engineering*, vol. SE-11, pp. 1491–1501, Dec. 1985.

[12] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[13] E. N. Elnozahy, D. B. Johnson, and Y.-M. Wang, "A survey of rollback-recovery protocols in message-passing systems," Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Oct. 1996.

[14] N. Neves and W. K. Fuchs, "Coordinated checkpointing without direct coordination," in *Proceedings of the 3rd IEEE International Computer Performance and Dependability Symposium*, (Durham, NC), pp. 23–31, Sept. 1998.

[15] A. T. Tai and K. S. Tso, "Verification and validation of the algorithms for guarded software upgrading," Phase-II Interim Technical Progress Report for Contract NAS3-99125, IA Tech, Inc., Los Angeles, CA, Sept. 1999.

[16] 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 International Computer Performance and Dependability Symposium (IPDS 2000)*, (Schaumburg, IL), Mar. 2000.