Protecting Distributed Software Upgrades that Involve Message-Passing Interface Changes

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

William H. Sanders
ECE Department
University of Illinois
Urbana, IL 61801

Abstract

We present in this paper an extension of the message-driven confidence-driven framework that we developed for onboard guarded software upgrading. The purpose of this work is to provide the framework with the capability of protecting distributed software upgrades that involve message-passing interface changes. To achieve this goal, we propose an approach to clustering the components involved in software upgrades and those involved in message-passing interface changes, such that from outside the cluster all those components can be perceived collectively as one virtual low-confidence component. Moreover, we develop a confidence-driven mechanism that enables combined use of sender- and receiver-side message logging for efficient, fine-grained error containment and recovery. The paper provides a detailed algorithm description.

1 Introduction

To protect distributed systems against failures caused by design fault introduced by software upgrades is an important and difficult problem. We have addressed this problem by proposing a guarded software upgrading (GSU) methodology and developing a message-driven confidence-driven (MDCD) error containment and recovery protocol [1, 2]. The framework was originally aimed at highly-available embedded distributed systems for mission-critical applications in which 1) software upgrades are carried out without interrupting mission operation (i.e., "onboard upgrades"), and 2) message-passing interface changes are prohibited because of complexity control and dependability concerns. Nonetheless, message-passing interface changes could be necessary in software upgrades for other distributed applications. Since interface updates, which may involve changes of message contents, ordering, and data types, may make it more difficult to devise dynamic error containment and recovery mechanisms and/or to use software components' old versions as the inherent, protective redundancy, guarded software upgrading will become a more challenging problem when a software upgrade involves message-passing pattern changes.

Structured (or "coordination-by-programmer") approaches (see [3], for example) and "plug-and-play" paradigms (see [4], for example) are viable ways to plan for future updates of component interfaces during software development. However, when a system we aim to upgrade contains legacy software components and/or an upgrade is intended to replace an old component with a commercial-off-the-shelf (COTS) product, it could be impossible for us to apply those approaches, because 1) we would be unable to have control over or access to the internal structure of legacy components or COTS programs, and 2) it is impractical to expect that COTS vendors would provide products that are structured in such a way that they can seamlessly and reliably interface with the existing structured components in our system.

Since incremental upgrades of legacy systems and use of COTS software products to update existing systems are becoming increasingly popular as ways to enhance a system's quality of service affordably, we require methods that impose no structural restrictions on either existing or new software and are dynamic in nature. Accordingly, through augmenting the MDCD framework, we develop algorithms that can protect distributed software upgrades involving message-passing interface changes.

The remainder of the paper is organized as follows. Section 2 reviews the MDCD approach. Section 3 describes how we augment the MDCD algorithms based on a clustering approach. The paper concludes with Section 4, which discusses the advantages of our approach.

2 MDCD Framework

The MDCD framework was initially motivated by the challenge of guarding a particular type of distributed embedded system (i.e., a system that consists of two functionally different interacting software components, one of which undergoes an onboard upgrade) against the adverse effects of design faults introduced by an onboard software upgrade [2]. In the MDCD protocol development, we introduced the "confidence-driven" notion to complement the message-driven (or "communication-induced")

approach employed by a number of existing checkpointing protocols for tolerating hardware faults [5, 6]. Specifically, we discriminate among the individual software components with respect to our confidence in their reliability; moreover, at execution time, we dynamically adjust our confidence in the processes corresponding to those software components, according to the knowledge about potential process state contamination caused by errors in a low-confidence component and message passing. Based on the dynamically adjusted confidence, each process is able to make decisions locally on whether checkpoint establishment, acceptance test, and rollback recovery should be carried out upon a message-passing event. The combined message-driven confidence-driven approach effectively improves system reliability while keeping performance cost low. Readers are referred to [2] for a detailed review of the original MDCD protocol.

When multiple high-confidence software components and/or multiple low-confidence components are present in a distributed system, individual processes may be potentially contaminated by different messages from a particular low-confidence component. Moreover, different processes in a system can be contaminated by errors in different low-confidence components. These factors collectively complicate the adjustment of confidence in individual processes, and make it more difficult to avoid cascading rollbacks. To address the problems, we introduced a *fine-grained* approach to adjusting confidence in a process state. This approach enables the development of the extended algorithms for the $(1, n)$-architecture (informally speaking, it is an architecture that contains a single low-confidence component and multiple high-confidence components; see [7] for the definition and a detailed review of the extended algorithms).

3 Augmenting the Framework

The MDCD framework described in Section 2 assumes that when a component undergoes an upgrade, it will not change the pattern of its communication with other components in the system. The algorithms presented in this section aim to remove this assumption and thus to further generalize the MDCD framework.

The major challenge in augmenting the MDCD framework to accommodate message-passing interface changes is that incompatibility between the new and old interprocess communication patterns would destroy recoverability [6]. To circumvent the difficulty, we devise a clustering mechanism that allows all the low-confidence components, including the upgraded components and those components involved in message-passing pattern changes to be "clustered" together and be collectively viewed as one virtual low-confidence component. The mechanism hides updated message-passing patterns inside the cluster and lets only unchanged message-passing activities remain visible to the world outside the cluster. In turn, this enables us to discover "embedded recovery points" and allows a distributed system that undergoes an upgrade involving message-passing interface changes to be considered as a type of system analogous to the $(1, n)$-architecture we dealt with in [7]. In addition, we adapt the message logging technique we developed in the original MDCD framework to enable confidence-driven, combined use of sender- and receiver-side message logging that guarantees recoverability.

3.1 Mechanism of Clustering

To explain why incompatibility between new and old interprocess communication patterns may destroy recoverability, we describe a scenario as follows. Assume that process P is an upgraded component and its message-passing interface with component P′ is also changed. If P′ establishes a checkpoint C upon the receipt of a message m from P (before passing m to the application), per the original MDCD framework, P′ will roll back to C if error recovery is invoked subsequently. However, after the rollback, we will have no way to determine whether and how the messages P′ received from P after the establishment of C should be restored (by an old version of P that takes over from P upon error recovery), because the message that triggers the establishment of C may reflect a communication pattern change.

To circumvent the difficulty, we devise a *clustering* mechanism that allows all the low-confidence components, including the upgraded components and those components involved in message-passing interface changes, to be "clustered" together and collectively viewed as one virtual low-confidence component. Thus, this mechanism hides changed message-passing patterns inside the cluster and lets only unchanged message-passing events remain visible to the world outside the cluster. In turn, this enables us to discover "embedded recovery points" and allows a system that undergoes an upgrade involving message-passing interface changes to be treated as a type of system analogous to the $(1, n)$-architecture we addressed in [7].

Suppose that process P is an upgraded component, and the upgrade changes the way P communicates with another process P′. Note that the communication pattern change means that P′ must also update the way it talks to P, regardless of whether P′ would otherwise undergo an upgrade. This in turn suggests that P′ should indeed also be regarded as a low-confidence component, based on the assumption that any modified code that has not yet had sufficient execution time has a greater likelihood of having design faults. More generally, a component upgrade that involves message-passing pattern changes will require some cooperating components to be updated as well, resulting in a cluster of low-confidence components.

It follows that when we group those low-confidence components into a cluster, all the messages coming out and going into the cluster will be those messages whose for-

mats (e.g., data type, ordering, source/destination) are not affected by the upgrade. This indeed suggests that if we view the cluster as a single low-confidence component, then we will be able to augment the algorithms devised for the $(1, n)$-architecture to protect distributed software upgrades that involve message-passing interface changes.

Figure 1 illustrates the idea of clustering. In the figure, we assume that P_1^{low} is an upgraded component (in which we have not yet had high confidence) and that its upgrade changes the way P_1^{low} talks to another component P_2. Since P_2 has to be updated as well in order to adapt to the changed communication pattern, we view the new version of P_2 also as a low-confidence component and call it P_2^{low}. On the other hand, we assume that other components are not going to be upgraded and that P_1^{low} does not change its way of communicating with them. Accordingly, we let the set of low-confidence components, call it G^{low}, be regarded as a single low-confidence component (see the dashed box), and let the old versions of those components be run in a shadow mode for error recovery purpose. We use G_{sdw}^{high} to denote the process set consisting of such components, and use G_{act}^{high} to denote the set of processes that are not involved in the upgrade and are actively in service.

As a result of clustering, all the message-passing activities that reflect communication pattern changes are confined inside the cluster, which implies that communication between the processes in G^{low} and those in G_{act}^{high} is the same as the (suppressed) communication between the processes in G_{sdw}^{high} and those in G_{act}^{high}. In turn, this suggests that we can keep track of the outgoing messages from G^{low}, and extract the embedded recovery points that enable correct error recovery. More specifically, we view outgoing messages from G^{low} as the root cause of error contamination; and we let 1) all the processes in G_{act}^{high} and G_{sdw}^{high} asynchronously keep track of the sequence number of the latest received message from G^{low} and be aware of the sequence number of the latest message from G^{low} that is validated, and 2) processes in G^{low} maintain a shared message counter $\texttt{int\_mSN\_G}^{low}$ that indicates the sequence number of the last message sent by G^{low} to a process in G_{act}^{high}. Hence, processes in G_{act}^{high} and G_{sdw}^{high} can, based on fine-grained confidence adjustment, establish checkpoints in their checkpoint arrays, with a checkpointing rule similar to that for the $(1, n)$-architecture. Specifically, we require a process P_i to establish a checkpoint if and only if P_i is in one of the following situations:

S1) Immediately before its otherwise non-contaminated state becomes potentially contaminated, or

S2) Immediately before its already potentially contaminated state becomes further contaminated by a message (from a potentially contaminated process) whose piggybacked $\texttt{int\_mSN\_G}^{low}$ is greater than P_i's local variable $\texttt{int\_mSN\_G}^{low}$, given that 1) P_i has sent a message to a process in G_{act}^{high} after P_i's last checkpoint establishment, or 2) P_i's last checkpoint is established when P_i receives a message from a potentially contaminated process that is not in G^{low}.

The processes in G_{sdw}^{high} also maintain a shared message counter $\texttt{int\_mSN\_G}_{sdw}^{high}$ which keeps track of the sequence number of the last message that a process in G_{sdw}^{high} intends to send to a process in G_{act}^{high} and that is suppressed due to the shadow mode of the processes in G_{sdw}^{high}. In addition, each process in G_{sdw}^{high} saves its internal and external messages in two separate message logs. In both logs, the value of $\texttt{int\_mSN\_G}^{low}$ (taken at the point when the message in question is saved to the log) is used as the index of a saved message. As the checkpoints in the checkpoint array are also indexed by the value of $\texttt{int\_mSN\_G}^{low}$, a process in G_{sdw}^{high}, after rolling back to a checkpoint C, can re-send the messages that are saved in the logs and have indices greater than or equal to the index of C. Furthermore, when a process in G_{sdw}^{high} receives a "passed AT" notification, the process will delete from its message logs the (internal and external) messages with indices less than or equal to $\texttt{m.int\_mSN\_G}^{low}$ (which is piggybacked on the notification).

In addition to the sender-side message logging by the processes in G_{sdw}^{high}, all the processes in $\{G_{act}^{high}, G_{sdw}^{high}\}$ adaptively perform receiver-side message logging. Specifically, when a potentially contaminated process receives a message from a clean (i.e., not potentially contaminated) process, the receiving process logs the message. This is necessary because the potentially contaminated receiving process may roll back to a checkpoint that is established prior to the receipt of the message, but the sending process will not roll back to an earlier state in which the message has not yet been sent (as the sending process is clean when it sends the message). The confidence-driven receiver-side message logging scheme thus enables the receiving process to "re-receive" the logged message(s), and guarantees recoverability when combined with the sender-side logging performed by the processes in G_{sdw}^{high}.

3.2 Algorithms

Based on the clustering mechanism, we augment the algorithms for the $(1, n)$-architecture to make them capable of protecting upgrades that involve message-passing interface changes. To aid in the description, we use P_i^{low}, P_i^{ha}, and P_i^{hs} to denote processes in G^{low}, G_{act}^{high}, and G_{sdw}^{high}, respectively.

The error containment algorithm for P_i^{low} is shown in Figure 2. Note that after P_i^{low} performs an AT successfully, the "passed AT" notification message is broadcast to other processes in G_{act}^{high} and G_{sdw}^{high}. On the other hand, if the process fails on an AT, P_i^{low} will first try an error compensation algorithm. Normal computation will resume if error

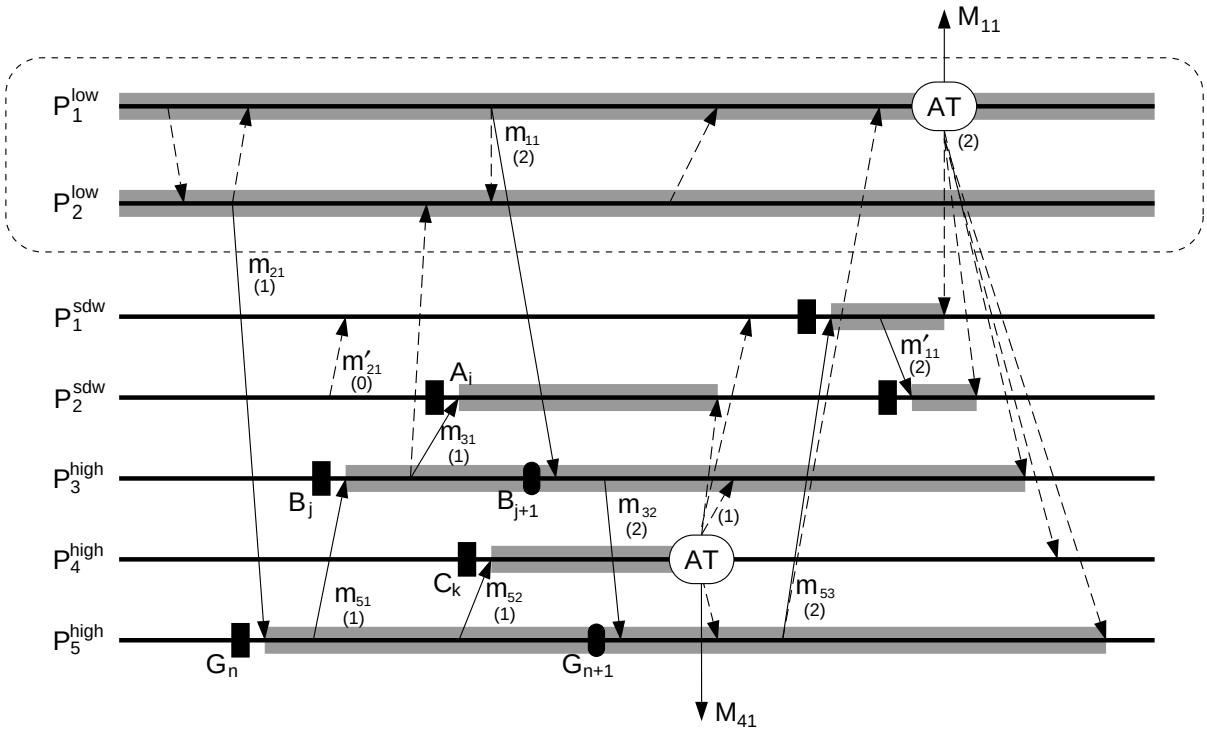

Figure 1: MDCD Approach Accommodating Message-Passing Interface Changes

compensation succeeds; otherwise error recovery will be invoked to let $G_{\text{sdw}}^{\text{high}}$ take over from G^{low}. When P_i^{low} sends a message to a process in $G_{\text{act}}^{\text{high}}$, P_i^{low} will first update (the shared) $\texttt{int\_mSN\_G}^{\text{low}}$ and broadcast it to all processes in G^{low}; then the message will piggyback the value of the updated $\texttt{int\_mSN\_G}^{\text{low}}$ and be sent to the designated receiving process in $G_{\text{act}}^{\text{high}}$.

Figure 3 shows the error containment algorithm for P_i^{ha}. Note that there are multiple high-confidence processes in $G_{\text{act}}^{\text{high}}$, each of which may 1) directly communicate with the processes in G^{low}, and 2) perform AT and send external messages. Therefore, the values of the local variables $\texttt{int\_mSN\_G}^{\text{low}}$ maintained by the processes in $G_{\text{act}}^{\text{high}}$ may differ. Moreover, as with the $(1, n)$-architecture, a successful AT performed by process P_i^{ha} at time t can validate a (potentially contaminated) process P_k^{ha} ($k \neq i$) only if the following condition is satisfied: P_k^{ha}'s state at t is not influenced directly or indirectly by any message that is from G^{low} and that has a sequence number greater than the local variable $\texttt{int\_mSN\_G}^{\text{low}}$ of P_i^{ha} (the value of this local variable is piggybacked on the "passed AT" notification message from P_i^{ha}). On the other hand, if at t, P_k^{ha}'s state violates the above condition, the AT may validate an earlier state of P_k^{ha} if this earlier state satisfies the above condition. Accordingly, based on the clustering mechanism and the checkpointing rule explained in Section 3.1, the algorithm shown

```
// P_i^low's dirty bit has a constant value of 1
if (outgoing_message_m_ready) {
  if (receiver(m) == device) {
    if (AT(m) == success) {
      msg_sending(m, device);
      // piggybacking int_mSN_G^low
      msg_sending("passed_AT", int_mSN_G^low,
                  {G_act^high, G_sdw^high});
    } else {
      if (error_compensation(m) == fail) {
        // compensation is not feasible
        error_recovery({G_act^high, G_sdw^high, G^low});
        exit(error);
      }
    }
  } else {
    // internal message sending
    if (receiver(m) ∈ G_act^high) {
      int_mSN_G^low++;
      msg_sending(int_mSN_G^low, {G^low - P_i^low});
    }
    msg_sending(m, dirty_bit, int_mSN_G^low,
                receiver(m));
  }
}
if (incoming_message_queue_nonempty) {
  application_msg_reception(m);
}
```

Figure 2: Error Containment Algorithm for P_i^{low}

in Figure 3 implements fine-grained adjustment of confidence in processes, which enables us to eliminate the risk of cascading rollback and minimize the rollback distance of a process when an error is detected in the system.

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        memory_reclamation(int_mSN_G^low, ckpt_array,
                           msg_log);
        dirty_bit = (size(ckpt_array) ≥ 1);
        msg_sending(m, device);
        msg_sending("passed_AT", int_mSN_G^low,
                    {{G_act^high - P_i^ha}, G_sdw^high});
      } else {
        if (error_compensation(m) == fail)
          // compensation is not feasible
          error_recovery({G_act^high, G_sdw^high, G^low});
      }
    } else { // external msg from a clean state
      msg_sending(m, device);
    }
  } else { // internal msg to P_i^ha
    if (receiver(m) ∈ {G_act^high}) {
      // "propagating out"
      prgt_bit = dirty_bit;
    }
    msg_sending(m, dirty_bit, int_mSN_G^low,
                receiver(m));
  }
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT") {
    memory_reclamation(m.int_mSN_G^low,
                       ckpt_array, msg_log);
    dirty_bit = (size(ckpt_array) ≥ 1);
  } else {
    if ((m.dirty_bit == 1 &&
        m.int_mSN_G^low > int_mSN_G^low) &&
        (dirty_bit == 0 || prgt_bit == 1)) {
      checkpointing(P_i^ha, ckpt_array);
      dirty_bit = 1;
      // "propagating in"
      prgt_bit = (sender(m) ∉ G^low);
    } else {
    if (m.dirty_bit == 0 && dirty_bit == 1) {
      msg_logging(m, int_mSN_G^low, int_msg_log);
    }
    application_msg_reception(m);
  }
  int_mSN_G^low = max(m.int_mSN_G^low, int_mSN_G^low);
}
```

Figure 3: Error Containment Algorithm for P_i^{ha}

Note that the algorithm also implements confidence-driven receiver-side logging. Specifically, upon the receipt of a message, P_i^{ha} will examine `m.dirty_bit` (which is piggybacked on the message) and its own `dirty_bit`. If the value of `m.dirty_bit` is zero and P_i^{ha}'s own `dirty_bit` has a value of 1, P_i^{ha} will save the message to its message log using the current value of the local variable $\texttt{int\_mSN\_G}^{low}$ as the index of the entry, before passing the message to the application. Thus when P_i^{ha} passes an AT, P_i^{ha} will delete all the entries of its checkpoint array and message log; likewise, when P_i^{ha} receives a "passed AT" notification, it will delete all the entries in its checkpoint array and message log, except those whose index values are greater than $\texttt{m.int\_mSN\_G}^{low}$ (which is piggybacked on the notification).

The error containment algorithm for processes in G_{sdw}^{high} is shown in Figure 4. Since all the processes in G_{sdw}^{high} run in the background and suppress both internal and external messages, they do not themselves perform AT, but rely on the ATs performed by other processes for confidence adjustment.

```
if (outgoing_message_m_ready) {
  if (internal(m) && receiver(m) ∈ G_act^high) {
    int_mSN_G_sdw^high++;
    msg_logging(m, int_mSN_G_sdw^high, int_msg_log);
    msg_sending(int_mSN_G_sdw^high, {G_sdw^high - P_i^hs});
  } else {
    // save the external msg to the log,
    // using int_mSN_G_sdw^high as an index
    msg_logging(m, int_mSN_G_sdw^high, ext_msg_log);
    msg_sending(int_mSN_G_sdw^high, {G_sdw^high - P_i^hs});
  }
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT") {
    memory_reclamation({m.ext_mSN_G^low,
                        m.int_mSN_G^low},
                        ckpt_array, msg_log);
    dirty_bit = (size(ckpt_array) ≥ 1);
    // last valid message of G^low
    iVR^low = max(m.int_mSN_G^low, iVR^low);
  } else {
    // application-purpose message from G_act^high
    if (m.dirty_bit == 1 &&
        m.int_mSN_G^low > int_mSN_G^low) {
      checkpointing(P_i^hs, ckpt_array);
      dirty_bit = 1;
    } else {
      if (m.dirty_bit == 0 && dirty_bit == 1)
        msg_logging(m, int_mSN_G_sdw^high, int_msg_log);
    }
    application_msg_reception(m);
    int_mSN_G^low = max(m.int_mSN_G^low, int_mSN_G^low);
  }
}
```

Figure 4: Error Containment Algorithm for P_i^{hs}

In addition to maintaining its local variable $\texttt{int\_mSN\_G}^{low}$ which is updated according to the sequence number piggybacked on the incoming messages from G^{low}, each process in G_{sdw}^{high} (i.e., P_i^{hs}) maintains a valid message register $\texttt{iVR}^{low}$ that indicates the sequence number of the last valid (internal) message sent by processes in G^{low} and is updated every time P_i^{hs} receives a "passed AT" notification message. Much like the processes in G^{low}, the processes in G_{sdw}^{high} maintain a shared variable

$\texttt{int\_mSN\_G}_{\text{sdw}}^{\text{high}}$ to keep track of the sequence number of the last (suppressed) message that a process in $G_{\text{sdw}}^{\text{high}}$ intends to send to a process in $G_{\text{act}}^{\text{high}}$. Recall that with the clustering scheme, the message-passing events between G^{low} and $G_{\text{act}}^{\text{high}}$ and those between $G_{\text{sdw}}^{\text{high}}$ and $G_{\text{act}}^{\text{high}}$ are identical. Accordingly, when P_i^{hs} saves an external message to `ext_msg_log`, the message will be saved in an entry indexed by the value of $\texttt{int\_mSN\_G}_{\text{sdw}}^{\text{high}}$, which enables not only memory reclamation but also message re-sending after error recovery.

During recovery, P_i^{ha} (a process in $G_{\text{act}}^{\text{high}}$) will roll back to a checkpoint with the smallest index in its checkpoint array, if `dirty_bit` of P_i^{ha} has a value of 1 when error recovery is invoked. After the rollback, the process will resume normal computation and replay the message(s) saved in the `int_msg_log`, if the log is nonempty.

After error recovery, P_i^{hs} (a process in $G_{\text{sdw}}^{\text{high}}$) will take over from P_i^{low}. P_i^{hs} will re-send or further suppress its outgoing messages (to the processes in $G_{\text{act}}^{\text{high}}$) if $\texttt{iVR}^{\text{low}}$ is less or greater than $\texttt{int\_mSN\_G}_{\text{sdw}}^{\text{high}}$, respectively. More specifically, in the former case, P_i^{hs} will re-send all the messages in its message logs `int_msg_log` and `ext_msg_log`; in the latter case, P_i^{hs} (possibly together with other processes in $G_{\text{sdw}}^{\text{high}}$) will suppress all its internal and external messages that are generated during the interval from error recovery invocation to the point when the counter $\texttt{int\_mSN\_G}_{\text{sdw}}^{\text{high}}$ is increased to a value that matches the value of $\texttt{iVR}^{\text{low}}$. And since each message saved in `ext_msg_log` is indexed by the value of $\texttt{int\_mSN\_G}_{\text{sdw}}^{\text{high}}$ (taken at the point the external message is generated), the messages saved in the logs will be "re-sent" in the right order.

4 Concluding Remarks

In order to protect distributed software upgrades that include message-passing interface changes, we have augmented the MDCD framework that was developed for guarding onboard software upgrades for distributed embedded systems. By introducing the method of clustering, we are able to preserve the dynamic nature and low-cost advantage of the framework, while extending its capability. In order to verify the correctness of the algorithms, we have also conducted formal proofs which are omitted in this paper due to space limitations.

We prefer dynamic error containment and recovery techniques over pre-structured approaches, especially for distributed systems in which processes coordinate through message passing, because 1) COTS software components are increasingly used in system upgrades and it is impractical to impose structural requirements on COTS products; 2) even if new components are strictly structured for interface updates and/or error recovery, systems that demand upgrades often contain legacy components that will likely be unable to accommodate structured or "plug-and-play" approaches; and 3) structured approaches are often application-specific and programmer-aware, which reduces flexibility and adds development and maintenance costs, while dynamic error containment and recovery techniques can be implemented by reusable middleware.

The enhanced MDCD framework can also be applied to provide fault tolerance to distributed systems that contain groups of software components that lack trustworthiness because of their vulnerability to malicious attacks. More specifically, to prevent malicious-code migration, we can isolate those components from the rest of the system (which is composed of trustable components) by clustering them and building a line of defense at the boundary between the cluster and the rest of the system. In addition to checkpoint establishment upon the receipt of a message from the cluster, we can perform malicious code detection before passing the message to the application and apply assertion-triggered exception handling.

Accordingly, we plan to evaluate the potential of the MDCD framework in intrusion tolerance. We also intend to investigate a hierarchical realization of the framework in which fault/intrusion tolerance can be adaptively and recursively activated to deal with system components that have different levels of trustworthiness.

References

[1] 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," *Performance Evaluation*, vol. 44, pp. 211–236, Apr. 2001.

[2] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "Low-cost error containment and recovery for onboard guarded software upgrading and beyond," *IEEE Trans. Computers*, vol. 51, pp. 121–137, Feb. 2002.

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

[4] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. Software Engineering*, vol. SE-23, pp. 759–776, Dec. 1997.

[5] Y. M. Wang *et al.*, "Checkpointing and its applications," in *Digest of the 25th Annual International Symposium on Fault-Tolerant Computing*, (Pasadena, CA), pp. 22–31, June 1995.

[6] 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.

[7] A. T. Tai, K. S. Tso, W. H. Sanders, L. Alkalai, and S. N. Chau, "Low-cost flexible software fault tolerance for distributed computing," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001)*, (Hong Kong, China), pp. 148–157, Nov. 2001.