*Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001),* Hong Kong, China, November 2001.

# Low-Cost Flexible Software Fault Tolerance for Distributed Computing[*]

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

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

**Abstract**

*In this paper, we revisit the problem of software fault tolerance in distributed systems. In particular, we propose an extension of a message-driven confidence-driven (MDCD) protocol we have developed for error containment and recovery in a particular type of distributed embedded system. More specifically, we augment the original MDCD protocol by introducing the method of "fine-grained confidence adjustment," which enables us to remove the architectural restrictions. The dynamic nature of the MDCD approach gives it a number of desirable characteristics. First, this approach does not impose any restrictions on interactions among application software components or require costly message-exchange based process coordination/synchronization. Second, the algorithms allow redundancies to be applied only to low-confidence or critical interacting software components in a distributed system, permitting flexible realization of software fault tolerance. Finally, the dynamic error containment and recovery mechanisms are transparent to the application and ready to be implemented by generic middleware.*

## 1 Introduction

As network and interconnection technologies advance, distributed embedded computing is increasingly becoming a key technological component of complex systems in many areas, ranging from industry to transportation, military, and even space applications. For example, due to their perceived performance and dependability advantages, distributed architectures are increasingly desired by the designers of avionics systems for future long-life deep-space missions. This, coupled with the fact that complex system functions which are critical to an application are often implemented by software, makes software fault tolerance in distributed computing environments an important issue.

Researchers have investigated into this issue since 1970s. The proposed approaches can be classified into two categories, namely, "coordination-by-programmer" and "coordination-by-machine." Techniques in the first category suggested to avoid cascading rollbacks and simplify state restoration by pre-structuring the application (see [1, 2], for example). More specifically, "coordination-by-programmer" techniques let a pre-identified group of interacting processes be synchronized such that they will not proceed to communicate to other processes outside the group until all the processes within the group pass acceptance tests or other types of error detection mechanisms. The second category "coordination-by-machine" is also called "programmer-transparent-coordination." Unlike "coordination-by-programmer," the approaches of the second category attempt to use "monitors," or software mechanisms of a similar type, for establishment and deletion of recovery points (see [3, 4], for example). While it relieves the application programmer of the burden of coordinating the recovery point establishments of interacting processes, "coordination-by-machine" incurs additional performance costs (relative to "coordination-by-programmer") due to various types of coordination-purpose message exchange and data structure maintenance/search activities.

In this paper, we propose an approach that aims for low-cost flexible realization of software fault tolerance in distributed embedded computing environments. The message-driven confidence-driven (MDCD) nature of this approach makes it different from traditional software fault tolerance in several respects. First, in order to preserve performance advantages of distributed computing, the extended MDCD approach does not impose any restrictions on the interactions among application processes; further, the error containment and recovery algorithms require neither message-exchange based process coordination/synchronization nor a global algorithm to search for a set of consistent checkpoints when error recovery is invoked. Second, the algorithms make it possible to apply redundancies (diverse versions) only to low-confidence or critical interacting software components in a distributed system, while allowing them to communicate freely with other processes in the system. Finally, the dynamic message-driven confidence-driven mechanisms are transparent to the application, which enables a middleware implementation.

The development of the MDCD protocol was initially

[*]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.

motivated by the challenge of guarding a particular type of distributed embedded system (i.e., a system which consists of two functionally different interacting software components, one of which has two functionally similar versions) against the adverse effects of design faults introduced by an onboard software upgrade [5]. Indeed, there are a number of factors other than upgrading, such as complexity or test coverage, that may lead us to discriminate among interacting software components in a distributed system with respect to our confidence in their trustworthiness. Also, some software component in a distributed system may have a higher message-sending rate than others, which implies that an error of that component is more likely to propagate; such a component could become the critical component of the system in the sense that it dominates error contamination, and should thus be given priority for fault tolerance. Those factors, coupled with the MDCD protocol's ability to facilitate the application of software fault tolerance to selected interacting components, suggest that the MDCD protocol has the potential to become a general-purpose low-cost software fault tolerance technique for distributed systems. In particular, the MDCD approach enables the application of "primary-routine and secondary-routine" based software fault tolerance schemes to a selected software component in a distributed computing environment, while permitting the component to communicate to other components with no restrictions. With the above motivation, we extend the MDCD approach by removing the architectural restrictions on the underlying system.

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. 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 [6] while keeping performance cost low [7].

However, when a more general distributed computing environment is considered, the maintenance of knowledge about potential process state contamination becomes more challenging. In particular, 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 circumvent the problems, we introduce a *fine-grained* approach to adjusting confidence in a process state. By carefully adapting the checkpointing rule of the original MDCD protocol, we are able to permit a process to be validated partially and progressively, and keep the performance overhead for the fine-grained confidence adjustment low. Specifically, with the extended algorithms, the view of the most recent non-contaminated state of a potentially contaminated process P is kept updated based on fine-grained confidence adjustment, during the interval after the state of P becomes potentially contaminated and before P is validated as having a non-contaminated state or confirmed as being actually erroneous (see Section 2 for the definitions). This enables the interacting processes to roll back minimum distances to reach a consistent global state when error recovery is invoked. Furthermore, the carefully modified checkpointing rule not only facilitates fine-grained confidence adjustment, but also 1) prevents a process from establishing checkpoints that are "predictably useless," and 2) ensures timely removal of checkpoints that become useless after fine-grained confidence adjustment.

The remainder of the paper is organized as follows. Section 2 reviews the original MDCD protocol. Section 3 describes the extended MDCD algorithms in detail. The paper concludes with Section 4, which discusses the advantages of our approach.

## 2 Background: The Original MDCD Protocol

The original development of the MDCD protocol assumes that the underlying system consists of three computing nodes and two functionally different application software components, one of which has two versions, namely, a better-performance less-reliable version running in the foreground and a poorer-performance more-reliable version running in the background to enable error recovery. The notation for the corresponding processes is as follows:

| | |
|---|---|
| $P_1^{act}$ | The active process corresponding to the low-confidence version of an application software component. |
| $P_1^{sdw}$ | The shadow process corresponding to the high-confidence version of the application software component. |
| $P_2$ | The (active) process corresponding to the second application software component in which we have high confidence. |

During concurrent execution, $\mathrm{P}_1^{\mathrm{act}}$ actually influences the external world and interacts with process $\mathrm{P}_2$, while the messages of $\mathrm{P}_1^{\mathrm{sdw}}$ that convey its computation results to $\mathrm{P}_2$ or external systems (e.g., devices) are suppressed. However, $\mathrm{P}_1^{\mathrm{sdw}}$ receives the same incoming messages that the active process $\mathrm{P}_1^{\mathrm{act}}$ does. Should an error of $\mathrm{P}_1^{\mathrm{act}}$ be detected, $\mathrm{P}_1^{\mathrm{sdw}}$ will take over $\mathrm{P}_1^{\mathrm{act}}$'s active role. We call the messages sent by processes to external systems and those between processes *external messages* and *internal messages*, respectively.

Because the objective of the MDCD protocol is to mitigate the effect of software design faults, we must ensure consistency among different processes' views on verified correctness (validity) of process states and messages. Accordingly, the MDCD algorithms aim to ensure that the error recovery mechanisms can bring the system into a global state that satisfies the following two properties:

**Consistency** If, in a global state $S$, $m$ is reflected as a message received by a process, then $m$ must also be reflected in $S$ as a message sent by the sending process, and the sending and receiving processes must have consistent views on the validity of $m$.

**Recoverability** If, in a global state $S$, $m$ is reflected as a message sent by a process, then $m$ must also be reflected in $S$ as a message received by the receiving process(es), and the sending and receiving processes must have consistent views on the validity of $m$, or the error recovery algorithm must be able to restore $m$.

A key assumption used in the derivation of the MDCD algorithms is that an erroneous state of a process is *likely* to affect the correctness of its outgoing messages (which convey the computation results of the process), while an erroneous message received by an application software component will result in process state contamination (because the data delivered by a message will normally become part of the state of the receiving process) [6]. Accordingly, the necessary and sufficient condition for a process to establish a checkpoint is that the process receives a message that will make its otherwise non-contaminated state become potentially contaminated. In order to keep performance overhead low, the correctness validation mechanism, *acceptance test* (AT), is only used to validate external messages from the active processes that are potentially contaminated. By a "potentially contaminated process state," we mean 1) the process state of $\mathrm{P}_1^{\mathrm{act}}$ that is created from a low-confidence software component, 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. Correspondingly, we use the term "a non-contaminated process state" to refer to a process state that is not potentially contaminated. We use the adverb "potentially" because i) whether a design fault in a software component will be manifested into an error in the state of the corresponding process is probabilistic rather than deterministic (depending upon the execution environment), and ii) an error in a process state is likely to be, but not necessarily, reflected in messages sent by the process. Accordingly, an AT is intended to either validate a potentially contaminated process as having a non-contaminated state, or confirm that the process is indeed erroneous.

The original version of the MDCD protocol also requires a process to establish a checkpoint right after its potentially contaminated state is validated by an AT as a non-contaminated state. Those checkpoints are called "Type-2" checkpoints, and are intended to be saved to stable storage for tolerating hardware faults. Recently, we have developed a scheme that enables synergistic coordination between the MDCD protocol and a time-based checkpointing protocol [8] for simultaneous tolerance of software and hardware faults [9]. Type-2 checkpoint establishment has thus been eliminated in the updated version of the MDCD protocol.

Figure 1 illustrates the MDCD approach. The horizontal lines in the figure represent the software executions along the time horizon. Each of the shaded regions represents an execution interval during which the state of the corresponding process is potentially contaminated. The symbols $m_{ij}$ and $M_{ik}$ denote, respectively, the $j$th internal message and $k$th external message sent by process $\mathrm{P}_i$.

Upon the detection of an error, $\mathrm{P}_1^{\mathrm{sdw}}$ will take over $\mathrm{P}_1^{\mathrm{act}}$'s active role and prepare to resume normal computation with $\mathrm{P}_2$. By locally checking its knowledge about whether its process state is contaminated or not (which is represented by a `dirty_bit`), a process will decide to roll back (to its most recent checkpoint) or roll forward (i.e., to continue its execution from the current state), respectively. After a rollback or roll-forward action, $\mathrm{P}_1^{\mathrm{sdw}}$ will "re-send" the messages in its message log or further suppress messages it intends to send (up to a certain point), based on the knowledge about the validity of $\mathrm{P}_1^{\mathrm{act}}$'s messages.

## 3 Extended MDCD Approach

### 3.1 Basic Concepts

As mentioned in Section 1, there are a number of factors that may result in differing levels of confidence in different software components in a system. For example, we may have low confidence in a software component with high complexity or poor testability. Software components in a distributed system may thus be categorized into two groups according to our confidence in their trustworthiness. Suppose that we have a more general distributed system architecture in which a subset of software components are low-confidence components. More formally, such an architecture can be defined by a pair $(l, n)$, where $1 \leq l < n$, and

- $l$ represents the number of functionally different, low-confidence software components;

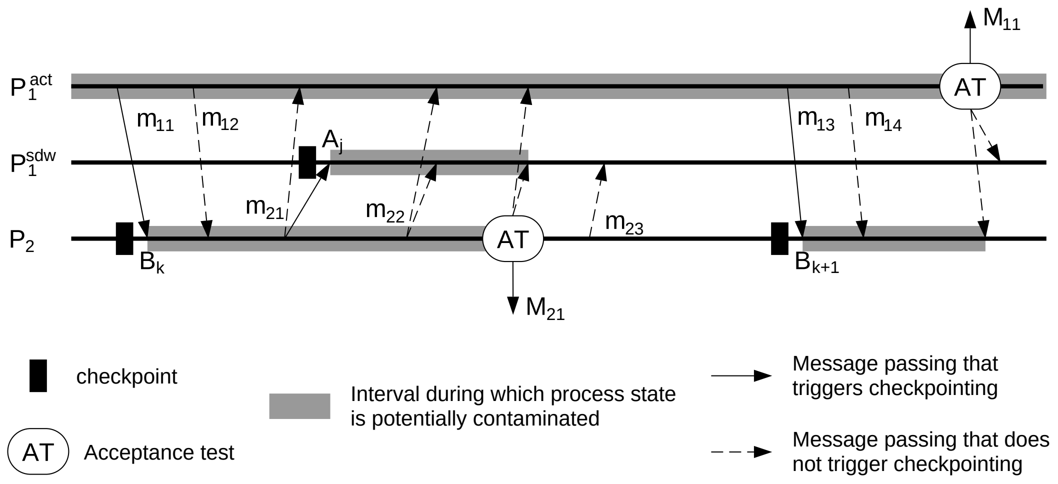
M11
P1 act
AT
m11
m12
m13
m14
P1 sdw
Aj
m21
m22
m23
P2
AT
Bk
Bk+1
M21
checkpoint
Interval during which process state is potentially contaminated
Message passing that triggers checkpointing
AT
Acceptance test
Message passing that does not trigger checkpointing

Figure 1: Message-Driven Confidence-Driven Approach

- $n$ represents the total number of functionally different software components in the system.

Note that $1 \leq l < n$ implies that there exist one or more low-confidence software components in the system which are a proper subset of $n$. For simplicity of illustration, we call the architecture type defined above an "$(l, n)$-architecture." For each of the $l$ low-confidence components in an $(l, n)$-architecture, the system accommodates a functionally similar, high-confidence component executing in the background to facilitate recovery. Hence, an $(l, n)$-architecture has a total of $(n + l)$ software components, including $l$ pairs of functionally similar, confidence-differing components. It follows that the distributed system considered for the original MDCD protocol development is a special case of the $(l, n)$-architecture in which $l = 1$ and $n = 2$, that is, a $(1, 2)$-architecture.

For a general $(l, n)$-architecture, it is possible that a process state will reflect multiple potential contaminations due to the receipt of more than one message from a potentially contaminated process, or from different potentially contaminated processes. On the other hand, the error contamination sources of individual potentially contaminated processes may partially overlap. Our assumption (see Section 2) on the relationship between the correctness of a process state and correctness of the messages (generated or received by the process) suggests that, if a message generated by a process P at at time $t$ passes an AT, then the AT not only verifies that the state of P is non-contaminated up to $t$, but also implies that any messages received or sent by P prior to $t$ are correct[1]. Accordingly, if the set of those messages is denoted by $M$, then the AT also implies that any process (other than P) that is potentially contaminated exclusively by the set (or a subset of) $M$ can be regarded as non-contaminated.

[1]In [6], we conducted a model-based analysis that demonstrated the effectiveness of the MDCD protocol with respect to reliability improvement when the protocol is executed in the environments in which this is not always the case.

The above observation leads us to introduce the approach of fine-grained confidence adjustment as follows. If at time $t$, an already potentially contaminated process P$'$ receives a message $m$ that may further contaminate P$'$, and the local knowledge indicates that the state $S'$ of P$'$ at $t$ (before $m$ is passed to the application) has a chance to be validated before the subsequent state (which reflects the receipt of $m$ by the application) gets validated, then we require P$'$ to establish another checkpoint C$'$ to save $S'$. In that manner, when a successful AT that is performed at $t'$ ($t' > t$) by a process other than P$'$ implies that P$'$ is non-contaminated up to $S'$, all the earlier checkpoints (relative to C$'$) established by P$'$ can be deleted, since P$'$, which has been partially validated by the AT, will roll back no further than C$'$, if an error is detected in the system after $t'$. These concepts are the basis for the MDCD algorithm extension, which is described in the following subsections.

## 3.2 Extended Algorithms

The algorithm extension effort consists of two parts. The first part yields a set of extended algorithms for the $(1, n)$-architecture, enabling the MDCD approach to be utilized in the systems which have multiple functionally different software components. In the second part, we further extend the algorithms for the $(l, n)$-architecture, such that the MDCD approach can be applied to the systems that have one or more low-confidence components. Our technical objective is to deal with the increased complexity in error containment and recovery in such systems with minimal performance cost.

### 3.2.1 For $(1,n)$-Architecture

We begin with extending the algorithms by removing the constraint on $n$, the total number of functionally different software components in the system, while letting the number of low-confidence components $l$ remain 1. In contrast to the $(1,2)$-architecture, the system now has multiple high-confidence software components $P_2, \ldots, P_n$, which are functionally different from $P_1^{act}$ and $P_1^{sdw}$; this necessitates modifications of the error containment and recovery algorithms, as described below.

The error containment algorithm for $P_1^{act}$ requires minimal modification for the extension. With the extended algorithm, as shown in Figure 2, after $P_1^{act}$ performs an AT successfully, the "passed AT" notification message is broadcast to all other processes, not just to $P_2$ and $P_1^{sdw}$ as in the error containment algorithm for the $(1,2)$-architecture.

```
// P_1^act's dirty bit has a constant value of 1
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (AT(m) == success) {
      // P_1^act maintains its msg SN and conveys
      // it to P_2, ..., P_n, and P_1^sdw
      msg_SN_P_1^act++;
      msg_sending(m, null, null, device);
      msg_sending("passed_AT", null, msg_SN_P_1^act,
                  {P_1^sdw, P_2, ..., P_n});
    } else {
      error_recovery({P_1^sdw, P_2, ..., P_n});
      exit(error);
    }
  } else {
    // m is an internal message to P_i,
    // i ∈ {2, ..., n}
    msg_SN_P_1^act++;
    msg_sending(m, dirty_bit, msg_SN_P_1^act, P_i);
  }
}
if (incoming_message_queue_nonempty) {
  application_msg_reception(m);
}
```

Figure 2: Error Containment Algorithm for $P_1^{act}$ for $(1,n)$-Architecture

The extension of the error containment algorithm for $\{P_i \mid 2 \leq i \leq n\}$ requires significantly more effort. Recall that in the $(1,2)$-architecture, $P_2$ is the only high-confidence component that interacts directly with $P_1^{act}$ and sends external messages (which may require AT-based validation). Thus $P_2$ is able to keep track of the sequence numbers of the messages sent by $P_1^{act}$ just using the information piggybacked on messages and a local variable msg_SN_$P_1^{act}$. In contrast, a $(1,n)$-architecture has multiple high-confidence processes, each of which may 1) directly communicate with $P_1^{act}$, and 2) send external messages and perform AT. Thus, the values of the local variables msg_SN_$P_1^{act}$ maintained by $P_2, \ldots, P_n$ may differ. As a result, dynamic confidence adjustment becomes more difficult. To see this, suppose that $P_i$ passes an AT and broadcasts a "passed AT" notification message, which piggybacks $k$, the value of $P_i$'s local variable msg_SN_$P_1^{act}$. If $k$ is the sequence number of the latest message (relative to the time of the AT-based validation) $P_1^{act}$ sends to $P_i$, $k$ is not necessary to be the sequence number of the latest message sent by $P_1^{act}$. This is because $P_1^{act}$ may have sent additional messages to other processes after it sends $P_i$ the message whose sequence number is $k$ and before $P_i$ performs the AT-based validation.

Recall that with the algorithms for the $(1,2)$-architecture, a successful AT performed by process $P_1^{act}$ or $P_2$ at time $t$ will convince us that 1) the process states of $P_1^{old}$ and $P_2$ at $t$ are non-contaminated, and 2) all the messages sent or received prior to $t$ are valid. In contrast, with the $(1,n)$-architecture, a successful AT performed by process $P_i$ at time $t$ can validate a process $P_k$, $k \neq i$, only if the following condition is satisfied: $P_k$'s state at $t$ is *not* influenced directly or indirectly by any message that is sent by $P_1^{act}$ and has a sequence number greater than the local variable msg_SN_$P_1^{act}$ of $P_i$ (which is piggybacked on the "passed AT" notification message). On the other hand, if at $t$, $P_k$'s state violates the above condition, the AT may validate an earlier state of $P_k$ if this earlier state satisfies the above condition. This, in turn, suggests that for the $(1,n)$-architecture, it will be feasible to allow fine-grained adjustment of confidence in a process. The advantage of fine-grained confidence adjustment is that it 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.

In order to realize fine-grained confidence adjustment, with the modified checkpointing rule for $\{P_i \mid 2 \leq i \leq n\}$, $P_i$ will establish a checkpoint if and only if the process is under 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 msg_SN_$P_1^{act}$ is greater than $P_i$'s local variable msg_SN_$P_1^{act}$, given that 1) $P_i$ has sent a message to a process other than $P_1^{act}$ 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 other than $P_1^{act}$.

Accordingly, with the extended algorithm for the $(1,n)$-architecture (see Figure 3), $P_i$ maintains an array of checkpoints indexed by the value of the msg_SN_$P_1^{act}$ field. This is unlike the error containment algorithm for the $(1,2)$-architecture, with which checkpoint $A_{i-1}$ can always be overwritten by A$_i$. As shown in Figure 3, when P$_i$ receives a "passed AT" notification message, the process will delete from the checkpoint array (`ckpt_array`) all the checkpoints with index values smaller than the value of `msg_SN_P`$_1^{act}$ piggybacked on the notification message, if such checkpoints exist in the array. Thus, when error recovery is invoked, P$_i$ will roll back to the checkpoint with the smallest index value in the checkpoint array, or roll forward, if the local `dirty_bit` is one or zero, respectively. Note that in the latter case, the checkpoint array must be empty.

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        memory_reclamation(msg_SN_P$_1^{act}$,
                           ckpt_array, null);
        dirty_bit = (size(ckpt_array) $\geq$ 1);
        msg_sending(m, null, null, device);
        msg_sending("passed_AT", null, msg_SN_P$_1^{act}$,
                    {P$_1^{sdw}$, P$_2$, ..., P$_n$});
      } else {
        error_recovery({P$_1^{sdw}$, P$_2$, ..., P$_n$});
      }
    } else { // external msg from a clean state
      msg_sending(m, null, null, device);
    }
  } else { // internal msg to P$_j$
    if (P$_j$ $\neq$ {P$_1^{act}$, P$_1^{sdw}$}) {
      // "propagating out" to P$_j$, 2 $\leq$ j $\leq$ n
      prgt_bit = dirty_bit;
    }
    msg_sending(m, dirty_bit, msg_SN_P$_1^{act}$, P$_j$);
  }
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT") {
    memory_reclamation(m.msg_SN_P$_1^{act}$,
                       ckpt_array, null);
    dirty_bit = (size(ckpt_array) $\geq$ 1);
  } else {
    if ((m.dirty_bit == 1 &&
         m.msg_SN_P$_1^{act}$ > msg_SN_P$_1^{act}$) &&
       (dirty_bit == 0 || prgt_bit == 1)) {
      checkpointing(P$_i$, ckpt_array);
      dirty_bit = 1;
      // "propagating in" via P$_j$, j $\neq$ i
      prgt_bit = (m.sender $\neq$ P$_1^{act}$);
    }
    application_msg_reception(m);
  }
  msg_SN_P$_1^{act}$ = max(m.msg_SN_P$_1^{act}$, msg_SN_P$_1^{act}$);
}
```

Figure 3: Error Containment Algorithm for P$_i$ for $(1, n)$-Architecture

Figure 4 illustrates the behavior of the extended MDCD protocol. While the other notation used in the figure follows the convention defined in Figure 1 (Section 2), an oval represents a checkpoint established by a process when its state has already been potentially contaminated, and the numbers in the parentheses underneath $m_{ij}$ and beside the dashed lines that represent the broadcast of a "passed AT" notification message indicate the value of the piggybacked `msg_SN_P`$_1^{act}$. In the scenario shown by the diagram, checkpoints A$_i$, B$_j$, C$_k$, and G$_n$ are established under the situations that match S1; whereas B$_{j+1}$ and G$_{n+1}$ are established under the situations that match S2. For example, P$_4$ establishes G$_{n+1}$ upon receiving message $m_{22}$ from P$_2$, since 1) the piggybacked `msg_SN_P`$_1^{act}$ is greater than the local `msg_SN_P`$_1^{act}$, and 2) P$_4$'s error-propagation bit (`prgt_bit`) is 1 (because P$_4$ sends messages to P$_2$ and P$_3$ after being potentially contaminated by $m_{11}$). Thus, when P$_3$ successfully passes the AT for its external message $M_{31}$, the state of P$_4$ saved in G$_{n+1}$ is validated and the previous checkpoint G$_n$ is deleted. In this manner, if P$_4$ fails the AT for message $M_{41}$, P$_4$ will roll back to G$_{n+1}$ (instead of G$_n$), which reflects P$_4$'s most recent non-contaminated state.

Note that there is an important difference between the original and extended MDCD error containment algorithms. Specifically, the former is intended to ensure that the most recent non-contaminated state is kept available in a checkpoint for a potentially contaminated process, so that the process will be able to roll back to that state in case of error recovery; whereas the latter attempts to maintain, for a potentially contaminated process, a series of states which may be validated in sequence, so that the process will be allowed to roll back a minimum distance when error recovery is invoked. This algorithmic difference implies that, although both the original and extended algorithms save the current state $S$ of a process P immediately before it becomes potentially contaminated, the mechanisms with which they maintain the view of P's most recent non-contaminated state differ. In particular, with the original algorithm, $S$ is persistently viewed as the most recent non-contaminated state until P is validated directly or indirectly by an AT; with the extended algorithm, the view of the most recent non-contaminated state of P may be updated, based on fine-grained confidence adjustment, during the interval between the time when the state of P becomes potentially contaminated and the time P is validated as having a non-contaminated state or confirmed as being actually erroneous.

To reduce the performance cost for fine-grained confidence adjustment, the extended checkpointing mechanism prevents P$_i$ from establishing checkpoints that are "predictably useless" (a "useless checkpoint" is a checkpoint that reflects a process state that will never become part of a consistent global state [10]). Note that the extended checkpointing mechanism implies the following: An already contaminated process P$_i$ will *not* establish another checkpoint C when P$_i$ receives a message that is from another potentially contaminated process and piggybacks a `msg_SN_P`$_1^{act}$ greater than P$_i$'s local `msg_SN_P`$_1^{act}$, unless 1) P$_i$ has sent a

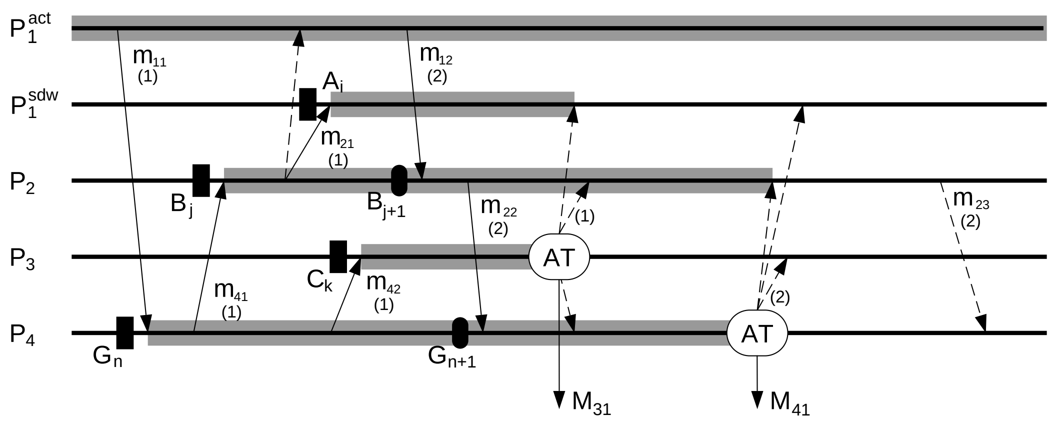

Figure 4: MDCD Approach for $(1, n)$-Architecture

message to a process other than $P_1^{act}$ since $P_i$'s last checkpoint establishment, or 2) the establishment of $P_i$'s last checkpoint is triggered by a message from a process other than $P_1^{act}$. To examine the last two conditions is necessary for preventing a process from taking useless checkpoints. This is because if neither of the two conditions holds, $P_i$ will not have a chance to be partially validated just up to the state saved in C; this implies that C will be useless. Consider again the scenario shown in Figure 4. Suppose that after being potentially contaminated by $m_{11}$, $P_4$ did not send any message to $P_2$ or $P_3$; checkpoint $G_{n+1}$ then would not have been established per the algorithm, because it is impossible for $P_4$ to be partially validated just up to the state saved in $G_{n+1}$, and thus it would have been a useless checkpoint.

The extended error containment and recovery algorithms for $P_1^{sdw}$ incorporate changes similar to those described above, as shown in Figure 5. In addition to maintaining the register that keeps track of the last valid message sent by $P_1^{act}$ (i.e., $\mathtt{VR}_1^{act}$), $P_1^{sdw}$ also maintains its local variable $\mathtt{msg\_SN\_P}_1^{act}$, which is updated according to the sequence number of $P_1^{act}$ piggybacked on the incoming messages.

Unlike other processes, $P_1^{sdw}$ runs in the background and suppresses all of its internal and external messages. Accordingly, $P_1^{sdw}$ itself does not perform AT and always relies on the ATs performed by other processes for confidence adjustment. This, coupled with the fact that $P_1^{sdw}$ will not be directly influenced by messages from $P_1^{act}$ (since there is no interaction between the two functionally equivalent processes), allows the checkpointing mechanism for $P_1^{sdw}$ to be simpler, relative to that for $\{P_i \mid 2 \leq i \leq n\}$. In particular, when $P_1^{sdw}$ receives an application-purpose message from a potentially contaminated process, comparing the piggybacked $\mathtt{msg\_SN\_P}_1^{act}$ with the local variable $\mathtt{msg\_SN\_P}_1^{act}$ will be sufficient for $P_1^{sdw}$ to avoid taking predictably useless checkpoints.

```
if (outgoing_message_m_ready) {
  // msg_SN_P$_1^{sdw}$ keeps track of P$_1^{sdw}$'s own messages
  msg_SN_P$_1^{sdw}$++;
  // suppress and log the outgoing message
  msg_logging(m, msg_SN_P$_1^{sdw}$, msg_log);
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT") {
    // P$_1^{act}$ or P$_i$, i $\in$ {2, ..., n},
    // reports a successful AT
    memory_reclamation(m.msg_SN_P$_1^{act}$,
                       ckpt_array, msg_log);
    dirty_bit = (size(ckpt_array) $\geq$ 1);
    // last valid message of P$_1^{act}$
    VR$_1^{act}$ = max(m.msg_SN_P$_1^{act}$, VR$_1^{act}$);
  } else {
    // application-purpose message
    // from P$_i$, i $\in$ {2, ..., n}
    if (m.dirty_bit == 1 &&
        m.msg_SN_P$_1^{act}$ > msg_SN_P$_1^{act}$) {
      checkpointing(P$_1^{sdw}$, ckpt_array);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
  msg_SN_P$_1^{act}$ = max(m.msg_SN_P$_1^{act}$, msg_SN_P$_1^{act}$);
}
```

Figure 5: Error Containment Algorithm for $P_1^{sdw}$ for $(1, n)$-Architecture

It can be shown that when $n = 2$, the extended algorithms for the $(1, n)$-architecture will result in system recovery behavior consistent with that under the original MDCD protocol (for the $(1, 2)$-architecture). For example, with the original MDCD protocol, a potentially contaminated process will always roll back to its most recent checkpoint when error recovery is invoked. Since the original MDCD algorithms always let checkpoint $A_{i-1}$ be overwritten by $A_i$ and thus a process never maintains multiple checkpoints, the above recovery behavior can be viewed as the scenario in which a contaminated process rolls back to the check-

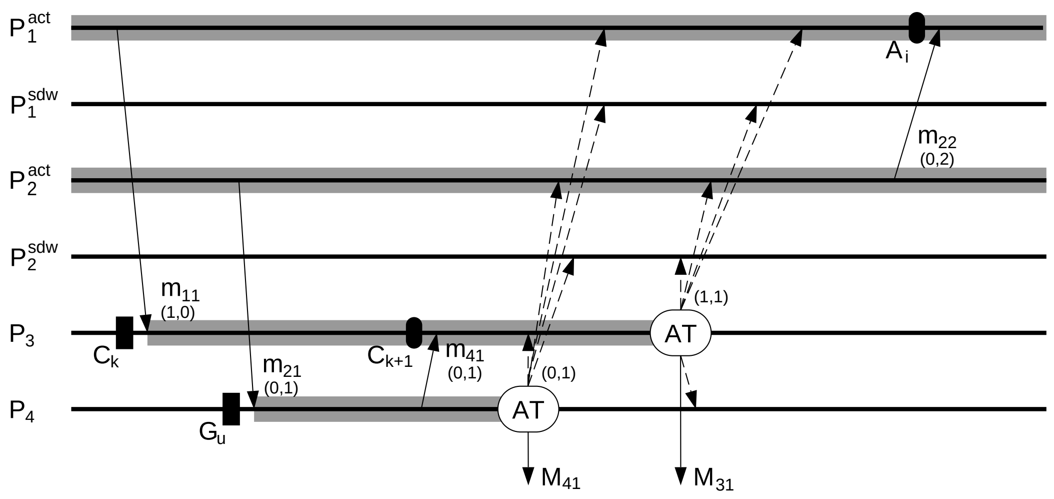

Figure 6: MDCD Approach for $(l, n)$-Architecture

point having the smallest $\texttt{msg\_SN\_P}_1^{\texttt{act}}$ among those in the checkpoint array, complying with the extended algorithms for the $(1, n)$-architecture.

### 3.2.2 For $(l, n)$-Architecture

The MDCD error containment and recovery algorithms can be further extended for the general $(l, n)$-architecture, where $1 \leq l < n$, to permit the MDCD approach to be utilized in a system that has one or more low-confidence software components. A major difficulty for this extension stems from the fact that there may exist multiple sources of error contamination in an $(l, n)$-architecture. One approach is to treat all the low-confidence software components equivalently in case of error recovery. In other words, the error recovery algorithms will let $P_1^{sdw}$, $P_2^{sdw}$, ..., $P_l^{sdw}$ take over the active roles of $P_1^{act}$, $P_2^{act}$, ..., $P_l^{act}$, respectively, after $P_1^{sdw}$, $P_2^{sdw}$, ..., $P_l^{sdw}$ and $P_{l+1}$, ..., $P_n$ complete their rollback or roll-forward recovery actions. While the advantage of low performance cost is preserved in this approach, a serious drawback is the inefficient resource-redundancy utilization.

Therefore, we devise a solution that lets $P_i^{sdw}$ take over from $P_i^{act}$ if and only if the process that performs the failed AT is $P_i^{act}$ itself or is contaminated by $P_i^{act}$ (through its message). This implies that, except for those to be replaced, the rest of the processes among $\{P_i^{act} \mid 1 \leq i \leq l\}$ must participate in error recovery. Accordingly, with the extended algorithm, $P_i^{act}$ will be required to take checkpoints based on a checkpointing mechanism similar to that for $\{P_i \mid l < i \leq n\}$, as shown in Figure 6.

To ensure global state consistency and recoverability, each of the processes corresponding to high-confidence components keeps track of the messages from $P_1^{act}$, $P_2^{act}$, ..., $P_l^{act}$ by maintaining a vector consisting of local variables $\{\texttt{msg\_SN\_P}_\texttt{i}^{\texttt{act}} \mid 1 \leq i \leq l\}$. Likewise, each $P_1^{sdw}$, $P_2^{sdw}$, ..., $P_l^{sdw}$ maintains knowledge about message validity based on a set of valid message registers $\{\texttt{VR}_\texttt{i}^{\texttt{act}} \mid 1 \leq i \leq l\}$. Much as in the algorithms for the $(1, n)$-architecture, each process maintains an array of checkpoints. However, the checkpoints are indexed on the vector $\langle\texttt{msg\_SN\_P}_1^{\texttt{act}}, \ldots, \texttt{msg\_SN\_P}_l^{\texttt{act}}\rangle$. Further, the mechanisms that the error containment algorithms for the $(1, n)$-architecture use for updating $\texttt{msg\_SN\_P}_1^{\texttt{act}}$ and $\texttt{VR}_1^{\texttt{act}}$ are adapted for updating $\{\texttt{msg\_SN\_P}_\texttt{i}^{\texttt{act}} \mid 1 \leq i \leq l\}$ and $\{\texttt{VR}_\texttt{i}^{\texttt{act}} \mid 1 \leq i \leq l\}$, respectively.

The presence of multiple error sources makes cascading-rollback avoidance a more difficult issue. To understand this, consider the scenario illustrated in Figure 6. Suppose that $P_4$ passes an AT when it attempts to send external message $M_{41}$, but subsequently $P_3$ fails the AT for $M_{31}$. As a result, both $P_3$ and $P_4$ have to roll back to their first checkpoints, because $P_3$ is contaminated by message $m_{11}$ and subsequently receives $m_{41}$ from $P_4$ before the error detection. We resolve this problem by *selectively* logging messages at the receiving side. More specifically, when a potentially contaminated process P receives a message $m$, P will log the message if there is a component $\texttt{msg\_SN\_P}_\texttt{i}^{\texttt{act}}$ in the piggybacked vector $\langle\texttt{msg\_SN\_P}_1^{\texttt{act}}, \ldots, \texttt{msg\_SN\_P}_l^{\texttt{act}}\rangle$ whose value is smaller than that of $\texttt{msg\_SN\_P}_\texttt{i}^{\texttt{act}}$ in the local vector. More succinctly, the condition for activating receiving-side logging is:

$$\exists\ \texttt{i},\ \texttt{m.msg\_SN\_P}_\texttt{i}^{\texttt{act}} < \texttt{msg\_SN\_P}_\texttt{i}^{\texttt{act}}$$

When P rolls back to recover from a detected error, the rollback of the process that sends $m$ will not be required if it is not potentially contaminated at the time of error detection. Therefore, in the scenario shown in Figure 6, $P_3$ will log message $m_{41}$ (since $\texttt{m.msg\_SN\_P}_1^{\texttt{act}} < \texttt{msg\_SN\_P}_1^{\texttt{act}}$). Then, $P_4$ will be allowed to go forward when

$P_3$ fails the AT for $M_{31}$, because $P_4$'s message to $P_3$ ($m_{41}$) that is logged by $P_3$ will be replayed by $P_3$ when it resumes execution after rollback. On the other hand, message logging will not be required if the above condition is not satisfied, i.e.,

$$\forall\ \mathtt{i},\ \mathtt{m.msg\_SN\_P_i^{act}} \geq \mathtt{msg\_SN\_P_i^{act}}$$

Under that circumstance, the process that sends message $m$ must also be contaminated and thus must also roll back if error recovery (invoked after the message-passing event) requires the receiving process to roll back, implying that $m$ will be re-sent when the sending process resumes execution.

Aside from the selective message logging mechanism described above, the error containment and recovery algorithms for the general $(l, n)$-architecture are devised in a fashion analogous to that in which we devise the algorithms for the $(1, n)$-architecture, although more conditions need to be checked to determine whether it is necessary for a process to establish a checkpoint upon receiving an application-purpose message and whether a process should delete checkpoints from its checkpoint array upon receiving a "passed AT" notification message. Due to space limitations, we do not present those algorithms.

## 4 Discussion and Concluding Remarks

In order to mitigate the effects of software design faults in a distributed computing environment with low performance cost and low development cost, we have adapted the communication-induced checkpointing strategy that was devised for hardware error recovery and complemented the strategy by introducing the confidence-driven notion. By devising the method of fine-grained confidence adjustment, we are able to extend the original MDCD protocol such that the MDCD approach can be utilized as a general-purpose software fault tolerance technique for distributed systems.

A few software fault tolerance projects have also addressed confidence-related issues. For example, in an NVP project, it was proposed that the decision algorithm could employ a "gold version" (i.e., a version that deserves higher confidence relative to other versions of the same program) as a reference for detecting erroneous voting results caused by related faults [11]. In [12], it was suggested that the secondary (backup) routine in the RB (recovery block) or DRB (distributed recovery block) scheme could use a version that is less efficient, but more reliable, than the primary (active) routine.

In contrast with those proposed techniques, the MDCD approach enables us to adjust our confidence in a process dynamically at runtime. This, coupled with the algorithm extensions described in this paper, facilitates more flexible, cost-effective software fault tolerance. The extended MDCD algorithms allow us to provide fault tolerance to critical software components only, while letting other components be protected against the effect of error propagation. The MDCD error containment mechanisms do not impose any restrictions on interactions among the application software components and are transparent to the application programmer, enabling a middleware implementation [13]. Further, the dynamic confidence-driven checkpointing and AT-based validation mechanisms allow individual processes to determine whether and when to establish a checkpoint at runtime, and let them choose error recovery actions based on their local knowledge about process state contamination. As a result, the MDCD error containment and recovery mechanisms require neither direct message-exchange based process coordination/synchronization, nor a global algorithm to search for a set of consistent checkpoints when error recovery is invoked.

It is worthy noting that the MDCD concepts and approach can also be utilized to enable traditional software fault tolerance schemes to be applied in distributed computing environments in a flexible, cost-effective fashion. In particular, schemes that are characterized by the concurrent execution of primary and secondary routines, such as NSCP (N-self-checking programming [14]), can be readily accommodated by the MDCD approach. To see this, consider a $(1, n)$-architecture. With the extended algorithms, the NSCP scheme can be applied to a critical component only, while allowing that component to interact freely (even prior to a results comparison or acceptance test) with the rest of the components in the system.

We have conducted model-based analyses to validate the effectiveness of the extended MDCD approach when it is applied to provide software fault tolerance to a critical component in a distributed system. (By a "critical component," we mean a software component that dominates error contamination in the system, due to its poor reliability and/or high message-sending rate.) In particular, we use the software tool *UltraSAN* [15] to carry out reliability analyses. The study demonstrates that the extended MDCD approach is as effective as the original MDCD protocol with respect to reliability improvement. In addition, the study illustrates another important capability of the MDCD approach that we have postulated earlier. That is, as shown in Figure 7, when a high message-sending rate of a software component makes it dominate error propagation in the system, the error containment and recovery mechanisms provided by the extended MDCD protocol are capable of effectively mitigating the error-propagation effect from this critical component and significantly improving system reliability.

Our current work is directed toward 1) quantitatively assessing the performance cost of the fine-grained confidence adjustment approach, and 2) investigating the feasibility of incorporating retry-upon-error-detection mechanisms and exception-handling techniques into the MDCD

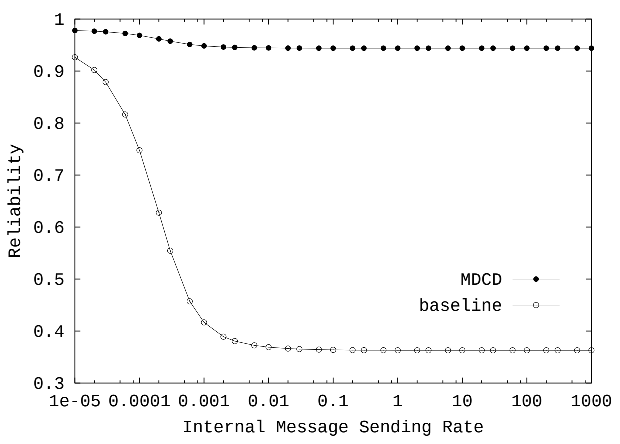

Figure 7: Reliability as a Function of Internal Message Sending Rate

protocol. We plan also to investigate the size-reducing technique called "incremental checkpointing" [16], which can be applied to augment our useless-checkpoint avoidance strategy for further performance overhead reduction.

## References

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

[2] J. Xu, A. Romanovsky, and B. Randell, "Coordinated exception handling in distributed object systems: from model to system implementation," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, (Amsterdam, The Netherlands), pp. 12–21, May 1998.

[3] G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points," in *Digest of the 13th Annual International Symposium on Fault-Tolerant Computing*, (Milano, Italy), pp. 48–55, June 1983.

[4] K. H. Kim, "Programmer transparent coordination of recovering concurrent processes: Philosophy and rules of efficient implementation," *IEEE Trans. Software Engineering*, vol. SE-14, pp. 810–821, June 1988.

[5] 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.

[6] 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.

[7] 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*, 2002.

[8] 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.

[9] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "Synergistic coordination between software and hardware fault tolerance techniques," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, (Göteborg, Sweden), pp. 369–378, July 2001.

[10] L. Alvisi, E. Elnozahy, S. A. Husain, and A. De Mel, "An analysis of communication induced checkpointing," in *Digest of the 29th Annual International Symposium on Fault-Tolerant Computing*, (Madison, WI), pp. 242–249, June 1999.

[11] A. Avižienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer*, vol. 17, pp. 67–80, Aug. 1984.

[12] K. H. Kim, "The distributed recovery block scheme," in *Software Fault Tolerance* (M. R. Lyu, ed.), pp. 189–209, West Sussex, England: John Wiley & Sons, 1995.

[13] K. S. Tso, A. T. Tai, L. Alkalai, S. N. Chau, and W. H. Sanders, "GSU middleware architecture design," in *Proceedings of the 5th IEEE International Symposium on High Assurance Systems Engineering*, (Albuquerque, NM), pp. 212–215, Nov. 2000.

[14] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun, "Definition and analysis of hardware-and-software fault-tolerant architectures," *IEEE Computer*, vol. 23, pp. 39–51, July 1990.

[15] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, 1995.

[16] J. S. Plank, "An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance," Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, TN, July 1997.