Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001), Göteborg, Sweden, July 2001.

Synergistic Coordination between Software and Hardware Fault Tolerance Techniques*

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
ECE Department
University of Illinois
Urbana, IL 61801

Abstract

This paper describes an approach for enabling the synergistic coordination between two fault tolerance protocols to simultaneously tolerate software and hardware faults in a distributed computing environment. Specifically, our approach is based on a message-driven confidence-driven (MDCD) protocol that we have devised for tolerating software design faults, and a time-based (TB) checkpointing protocol that was developed by Neves and Fuchs for tolerating hardware faults. By carrying out algorithm modifications that are conducive to synergistic coordination between volatile-storage and stable-storage checkpoint establishments, we are able to circumvent the potential interference between the MDCD and TB protocols, and to allow them to effectively complement each other to extend a system's fault tolerance capability. Moreover, the protocol-coordination approach preserves and enhances the features and advantages of the individual protocols that participate in the coordination, keeping the performance cost low.

1 Introduction

To achieve high dependability in critical applications, software and hardware fault tolerance issues must be dealt with simultaneously. Often, software and hardware fault tolerance schemes are developed separately using different types of techniques, due to the distinctions in nature between software and hardware faults. For centralized computing applications, a simple integration of techniques from the two categories may effectively achieve the goal of simultaneous software and hardware fault tolerance. For example, using N-version programming in a system with N-modular redundancy could permit a software error to be masked by design diversity and a hardware fault to be tolerated via graceful redundancy degradation.

In the context of distributed computing, however, naively combining software and hardware fault tolerance techniques may lead to an invalid solution. In particular, potential incompatibilities between the techniques may make their combination inefficient, or may cause them to interfere with one another, resulting in a detrimental effect on system reliability. For example, to enable a distributed system to recover from a software error, it will be necessary to apply a software fault tolerance scheme that maintains consistency among the views on message/state validity from the standpoints of interacting processes; however, this consistency may not be preserved in the checkpoints established by a hardware fault tolerance protocol executing in the same system. Consequently, the system may become unable to recover from a software error that occurs after a rollback recovery caused by a hardware fault.

Despite its importance and difficulties, seamless integration of software and hardware fault tolerance techniques for distributed systems has not received enough attention. Several fault tolerance schemes have been proposed to deal with both software and hardware faults by mapping diverse software versions to redundant computer nodes (see [1, 2], for example). However, the scope of those efforts was limited to method development for utilizing redundant resources in a distributed system to protect the execution of diverse versions of a non-distributed application software against hardware failure. Simultaneous tolerance of software and hardware faults for distributed computing applications thus remains an important challenge.

In order to accomplish simultaneous software and hardware fault tolerance in a distributed computing environment, we propose an approach that will enable synergistic coordination between software and hardware fault tolerance techniques. In particular, we devise a scheme that allows the message-driven confidence-driven (MDCD) protocol, which we have developed for mitigating the effect of software design faults [3], to coordinate with a time-based (TB) checkpointing protocol that was developed by Neves and Fuchs for tolerating hardware faults [4]. Our approach emphasizes avoiding potential interference between software and hardware fault tolerance techniques and enabling them to be mutually supportive. Consequently, in the resulting protocol-coordination scheme, the concurrently run-

*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.

ning adapted MDCD and TB protocols are able to complement each other effectively to realize the goal of extending a system's fault tolerance capability. Specifically, the MDCD protocol is responsible for establishing a checkpoint in volatile storage, upon a message passing event that alters our confidence in a process state, to facilitate efficient software error recovery; while the duty of the TB protocol is to save checkpoints to stable storage, based on periodically resynchronized timers, for tolerating faults in the hardware that accommodates the interacting processes. Moreover, the establishment of a stable-storage checkpoint is conducted in a manner that adapts to the up-to-date knowledge provided by the MDCD protocol regarding potential process state contamination, thus preserving validity-concerned global state consistency and recoverability (see Section 2.1 for the definitions) which are crucial for software error recovery.

By carefully adapting the algorithms, we are able to preserve and enhance the features and advantages of the MDCD and TB protocols in this coordination scheme. In particular, just as neither of the protocols depends on costly message-exchange based coordination among processes, the protocol-coordination scheme itself does not require or involve message-exchange based coordination between the participating protocols. More importantly, this effort demonstrates that exploiting the synergy of differing fault tolerance techniques is an effective approach for enabling-technology integration.

The remainder of the paper is organized as follows. Section 2 reviews the MDCD and TB protocols. Section 3 describes the MDCD algorithm modifications, followed by Section 4 which presents the adapted TB checkpointing protocol and illustrates how it coordinates with the modified MDCD protocol in a synergistic fashion. The concluding remarks discuss the significance of this effort.

2 Background

2.1 MDCD Error Containment and Recovery Protocol

The development of the MDCD protocol was initially motivated by the challenge of guarding an embedded system against the adverse effects of design faults introduced by an onboard software upgrade [3]. 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 thus could become the reliability bottleneck or critical component of the system, and should 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, suggests that the MDCD protocol has the potential to become a general-purpose low-cost software fault tolerance technique for distributed systems. Accordingly, we have recently extended the MDCD approach by removing the architectural restrictions on the underlying system [5].

In the development of the protocol-coordination scheme described in the following sections, we retain the architectural assumptions used for the original development of the MDCD protocol for simplicity and clarity. That is, we assume that the underlying system consists of three computing nodes and two functionally different application software components, one of which has two versions, namely, a low-confidence version and a high-confidence version. Indeed, this assumption is applicable not only when the MDCD protocol is employed for guarded software upgrading, but also when the protocol is utilized to allow application of "primary-routine and secondary-routine" based software fault tolerance schemes, such as DRB (distributed recovery blocks [1]) and NSCP (N-self-checking programming [6]), to a selected software component in a distributed system. For the former case, the MDCD protocol lets an earlier, high-confidence version escort the execution of an upgraded version in which we have not yet established high confidence due to its insufficient onboard execution time. For the latter case, a better-performance less-reliable version is used as the primary routine running in the foreground, and a poorer-performance more-reliable version is used as the secondary routine running in the background to enable error recovery, as suggested by [1]. Accordingly, we use the following notation in the description of the MDCD protocol and the protocol-coordination scheme:

- 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 guarded operation, P_1^{act} actually influences the external world and interacts with process P_2, while the messages of P_1^{sdw} that convey its computation results to P_2 or external systems (e.g., devices) are suppressed. However, P_1^{sdw} receives the same incoming messages that the active process P_1^{act} does. Thus, P_1^{sdw} and P_1^{act} can perform the same computation based on identical input data. Should an error of P_1^{act} be detected, P_1^{sdw} will take over P_1^{act}'s active role. We call the messages sent by processes to external systems and the messages between processes *external messages* and *internal messages*, respectively. In order to keep performance

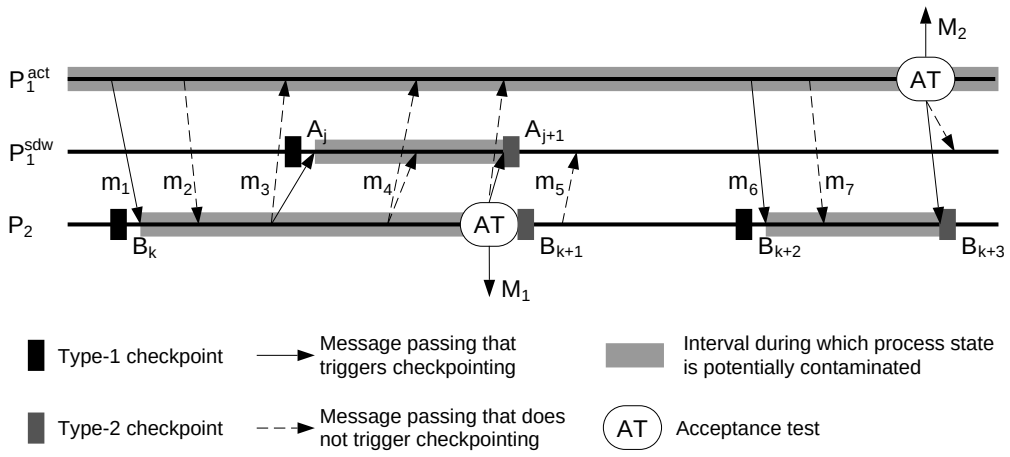

Figure 1: Message-Driven Confidence-Driven Checkpoint Establishment

overhead low, the correctness validation mechanism, *acceptance test* (AT), is only used to validate external messages from the active processes that are potentially contaminated (see below for the definition of *potentially contaminated process state*). Limiting the use of AT to external messages plays a dual role in performance cost reduction. Specifically, this strategy not only enables a process to perform AT less frequently, but also facilitates testing efficiency. This is because external messages often correspond to control commands/data that can usually be verified by simple logic checking or reasonableness tests, unlike internal messages, which convey intermediate computation results that are far more difficult to validate.

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.

Note that the above definitions are extended versions of the definitions of global state consistency and recoverability presented (in different forms) in [7, 8, 4]. For clarity of illustration, in the remainder of this paper, we use the terms "validity-concerned global state consistency and recoverability" and "basic global state consistency and recoverability" to refer to the extended and original versions, respectively, whenever it is necessary to distinguish between the two versions.

The 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, while an erroneous message received by an application software component will result in process state contamination [9]. Accordingly, 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 potentially contaminated state is validated as a non-contaminated state. By a "potentially contaminated process state," we mean i) the process state of P_1^{act} in which we have not yet established enough confidence, or ii) 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 1 illustrates the above concepts. 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. In the diagram, checkpoints B_k, A_j, and B_{k+2} are established immediately before a process state becomes potentially contaminated (we call them *Type-1* checkpoints), while B_{k+1}, A_{j+1}, and B_{k+3} are established right after a potentially contaminated process state gets validated as a non-contaminated state (we call them *Type-2* checkpoints).

Upon the detection of an error, P_1^{sdw} will take over P_1^{act}'s active role and prepare to resume normal computation with 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 roll-back or roll-forward action, P_1^{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 P_1^{act}'s messages.

We have derived theorems showing that the rollback or roll-forward recovery decisions made locally by the individual processes guarantee the validity-concerned global state consistency and recoverability properties. The proofs of those theorems are given in [5].

2.2 Time-Based Checkpointing Protocol

The second of the two protocols that form the basis of our coordination scheme is the time-based checkpointing protocol proposed by Neves and Fuchs [4]. Time-based protocols allow processes to establish checkpoints using approximately synchronized and periodically resynchronized timers, thus eliminating the need for costly message-exchange based coordination among processes and reducing performance overhead [10, 4].

If the timers were exactly synchronized, all the processes would initiate their checkpoint establishments at exactly the same time, so that no consistency violation could happen. Further, if timer synchronization was perfect and message delivery took a negligible amount of time, recoverability would never be violated, because "in-transit" messages would not occur. However, in a distributed system, timers cannot be perfectly synchronized, and message delivery delay is often non-negligible. Figure 2(a) illustrates a scenario in which global state consistency and recoverability are violated. In the scenario, 1) after establishing its checkpoint, process P_a sends a message m_1 that is read before process P_b establishes a checkpoint, which destroys consistency; and 2) before establishing its checkpoint, P_b sends a message m_2 that is read by P_a after it completes its checkpoint establishment, so that m_2 becomes an in-transit message, which destroys recoverability. To circumvent those problems, time-based checkpointing protocols use strategies that ensure global state consistency and recoverability by blocking messages during some critical time periods before and/or after a process starts to save a checkpoint (see Figure 2(b)).

To reduce performance cost, the time-based checkpointing protocol developed by Neves and Fuchs and described in [4] uses a blocking period after a process starts to write a checkpoint to stable storage to ensure consistency but the protocol does not use blocking for recoverability. To ensure recoverability, this protocol saves all the messages for which it has not received acknowledge responses as part of the next checkpoint. Thus, when hardware error recovery is invoked, the protocol will be able to re-send all the unacknowledged messages. This protocol minimizes the performance overhead due to blocking by 1) avoiding the "blocking-for-recoverability" periods, and 2) retaining the "blocking-for-consistency" periods, as each of them overlaps with the time interval during which a stable-storage checkpoint establishment is in progress.

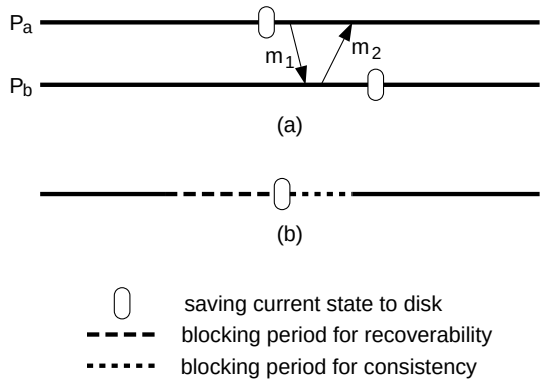

Figure 2: Global State Consistency and Recoverability

3 MDCD Protocol Modification

Since its goal is to mitigate the effects of software design faults, the MDCD protocol allows checkpoints to be saved in local volatile storage (RAM), instead of stable storage (disk), in order to keep performance overhead low. The protocol also has the potential to tolerate hardware faults, since it offers us the option to save checkpoints selectively in stable storage. Specifically, a broadcasted "passed AT" notification message would trigger each of the processes to establish a Type-2 checkpoint. Since the key assumption used for the MDCD algorithm derivation (see Section 2.1) implies that if an outgoing message is validated by AT, then the process state of the sending process and all the messages sent or received prior to performing the AT can be considered non-contaminated and valid, respectively, the resulting Type-2 checkpoints would constitute a consistent global state. Hence, it is possible to derive a variant MDCD protocol which lets each process, including P_1^{act}, save a (Type-2) checkpoint in stable storage upon the receipt of a "passed AT" notification message. When a hardware fault occurs, all the processes roll back to their stable-storage checkpoints for error recovery. Unfortunately, this approach (which we call the "write-through" approach) is unable to ensure low performance overhead and error recovery efficiency, because the frequency of and time between Type-2 checkpoint establishments depend on the external message sending rate; thus, a process may suffer an excessive rollback distance when a hardware fault occurs. In contrast, when the MDCD protocol is used for software fault tolerance, rollback distance of a process is minimized through its

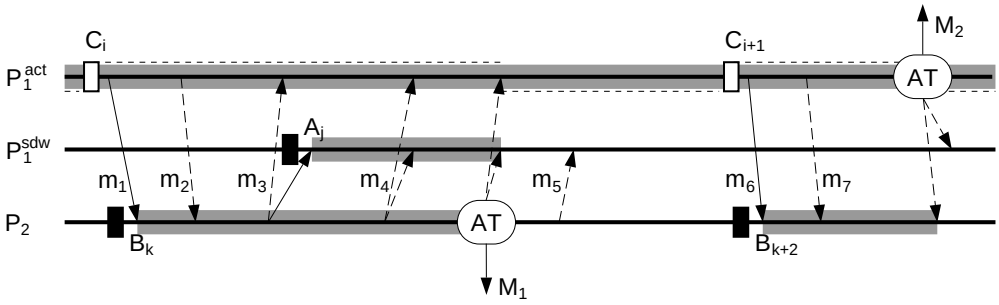

Figure 3: Modified MDCD Protocol

flexible, confidence-adaptive recovery behavior. That is, the process is able to choose between rollback and roll-forward, based on whether it is potentially contaminated.

To circumvent the inefficiencies associated with the write-through mechanism, we seek an approach that will enable the MDCD protocol to coordinate with some hardware fault tolerance protocol to achieve simultaneous software and hardware fault tolerance. A careful study led us to choose the time-based checkpointing protocol proposed by Neves and Fuchs in [4] (which is reviewed in Section 2.2). The rationale for this choice is that this particular time-based protocol is characterized by low performance cost [11] and shares the "no-direct-coordination" feature with the MDCD protocol.

In order to facilitate a synergistic coordination between the two protocols, we begin with making modifications for the MDCD error containment algorithms. The objective is to enable the MDCD protocol to ensure the readiness of the information that the TB protocol will require for establishing stable-storage checkpoints that satisfy validity-concerned consistency and recoverability properties. Appendix A provides the modified algorithms, and Figure 3 illustrates the behavior of the modified protocol.

Recall that with the original MDCD protocol, P_1^{sdw} will take over the active role of P_1^{act} when error recovery is invoked. Hence, P_1^{act} is exempted from performing checkpointing. Also, since P_1^{act} is created from a low-confidence software component, the process is invariably regarded as potentially contaminated during guarded operation, such that P_1^{act}'s `dirty_bit` has a constant value of one. But when hardware error recovery is concerned, P_1^{act} must be able, when a hardware fault occurs, to roll back to its stable-storage checkpoint and restart with other processes in the system to resume computation. In order to enable P_1^{act} to participate in the establishment of stable-storage checkpoints that satisfy the validity-concerned global state consistency and recoverability properties, we let P_1^{act} maintain a "pseudo dirty bit" (`pseudo_dirty_bit`), in addition to its "actual" `dirty_bit` (which has a constant value of one). The value of `pseudo_dirty_bit` is reset to zero when P_1^{act} passes an AT or receives a "passed AT" notification message, and is set to one immediately before P_1^{act} sends an internal application-purpose message, as indicated in Figure 3, respectively, by the dashed lines below and above the shaded line representing P_1^{act}'s potentially contaminated state. By examining the value of `pseudo_dirty_bit`, P_1^{act} will determine whether it should establish a (volatile-storage) checkpoint before it sends an internal application-purpose message. Specifically, if the value is zero, meaning that the internal message P_1^{act} intends to send is the first outgoing internal message since the last AT-based validation, P_1^{act} will establish a checkpoint before it sends the message. Thus, this checkpoint will be consistent with the checkpoint that will be established by the receiving process, after receiving the message and before passing it to the application. We call a checkpoint of P_1^{act} a "pseudo checkpoint" (represented by a hollow rectangle in Figure 3) due to its relationship with the value of `pseudo_dirty_bit`.

As shown in Figure 3, the modified protocol eliminates the Type-2 checkpoint establishment. This is because the coordination between the MDCD and TB protocols, as described in Section 4.2, allows error recovery to be independent of Type-2 checkpoints. However, the knowledge-updating actions (i.e., changing the value of `dirty_bit`, updating the valid message register $\texttt{VR}_1^{\texttt{act}}$, etc.) taken by a process when it passes an AT or receives a "passed AT" notification message are preserved in the modified algorithms.

Finally, since the adapted TB protocol (described in the next section) will enforce a blocking period when a process starts to save a checkpoint to stable storage upon the expiration of its checkpointing timer, the MDCD algorithms are accordingly modified such that during the blocking period:

1) An incoming application-purpose message will not be passed to the application, and

2) When a "passed AT" notification message arrives, the `dirty_bit` (or `pseudo_dirty_bit`) will be reset if and only if the piggybacked sequence number

of stable-storage checkpoint N_{dc} matches the value of the N_{dc} of the receiving process.

4 TB Protocol Modification

4.1 Necessity for Synergistic Coordination

Although the TB protocol guarantees basic global state consistency and recoverability, directly combining it with the MDCD protocol would not extend a system's fault tolerance capability, but rather may have a detrimental effect on system reliability. As shown in Figure 4(a), if P_2 and P_1^{sdw} follow the TB protocol to save their states when their checkpointing timers expire, then the checkpoint of P_2 will reflect a potentially contaminated process state, whereas the checkpoint of P_1^{sdw} will reflect a non-contaminated state. Consequently, when a hardware fault occurs in the node accommodating P_2, P_2 would have no choice but to roll back to a potentially contaminated state and become unable to restore a non-contaminated state if a software error is detected subsequently. This example illustrates a serious consequence of attempting to simply combine software and hardware fault tolerance techniques. In particular, when establishing the stable-storage checkpoint, P_2 ignores the knowledge about message/state validity that is made available by the MDCD protocol. This causes the loss of the most recent non-contaminated state of P_2 that is saved by the MDCD protocol in the volatile-storage checkpoint.

Suppose that, in the same scenario, instead of saving the current process state in stable storage when its checkpointing timer expires, the potentially contaminated process P_2 copies to stable storage its most recent volatile-storage checkpoint[1] C that is established prior to the timer expiration, while the non-contaminated process P_1^{sdw} saves its current state S to stable storage. As verified in [5], the most recent non-contaminated process states of P_2 and P_1^{sdw} (relative to a time instant t) are always globally consistent. It follows that the two stable-storage checkpoints, which are copies of C and S, will be globally consistent, because C and S reflect the most recent non-contaminated process states of P_2 and P_1^{sdw} (relative to any time instant t within the blocking period of P_1^{sdw}), respectively.

However, the above mechanism will fail to ensure recoverability in the scenario illustrated by Figure 4(b). In that scenario, the timer of P_1^{sdw} expires at t when both P_1^{sdw} and P_2 are potentially contaminated. Suppose that, when its timer expires, P_1^{sdw} copies to stable storage its most recent volatile-storage checkpoint (which reflects P_1^{sdw}'s most recent non-contaminated process state, relative to t). But after t and before P_2's timer expires, P_2 passes an AT, resulting in an in-transit "passed AT" notification message. It follows that m would 1) be reflected in P_2's stable-storage checkpoint as a valid message that has been sent and acknowledged (because P_1^{sdw} receives m before the blocking period starts), and 2) be excluded from P_1^{sdw}'s stable-storage checkpoint and not be restored in case of error recovery (because P_2 would only re-send unacknowledged messages).

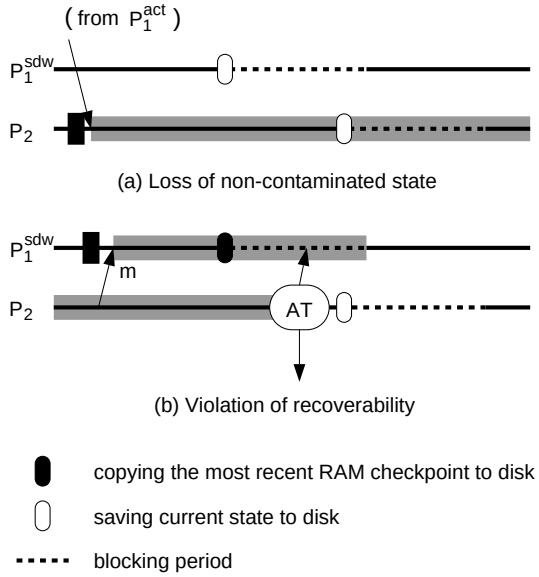

Figure 4: Consequence of Simple Combination

The above observations suggest that using knowledge about potential process state contamination for choosing the contents of a stable-storage checkpoint is necessary but not sufficient. To guarantee recoverability, an in-progress stable-storage checkpoint establishment must be responsive to the knowledge update that occurs during the blocking period. Accordingly, we modify the TB algorithm, as described in the next subsection.

4.2 Adapted TB Protocol

Figure 5 shows the adapted version of the TB checkpointing algorithm. Analogous to the original TB algorithm, the adapted version lets a process save a checkpoint in stable storage and undergo a blocking period when its checkpointing timer expires. And similar to the original TB algorithm, the adapted version lets a process save, as part of the next checkpoint, all unacknowledged messages. Thus, when hardware error recovery is invoked, the process will be able to re-send those unacknowledged messages.

To ensure validity-concerned global state consistency, the adapted TB algorithm first examines, upon the expiration of a process's checkpointing timer, whether the process is potentially contaminated. If its `dirty_bit` indicates that the current state is potentially contaminated, then the volatile-storage checkpoint will be copied to stable storage; otherwise, the current process state will be saved in stable storage, as illustrated in Figure 6(a). In either case, the pro-

[1] Recall that with the MDCD protocol, a process will not roll back any further than its most recent checkpoint; therefore, a process keeps only its most recent checkpoint in volatile storage.

```
createCKPT() {
  if (dirty_bit == 0)
    write_disk(current_state, 0, null);
  else
    write_disk(rCKPT, 1, current_state);
  N_dc++;
  dCKPT_time = dCKPT_time + θ;
  set_timer(createCKPT, dCKPT_time);
  if ((δ + 2ρ(N_dc − 1)θ + T_m(dirty_bit)) >
      (getTime() − (dCKPT_time − θ)))
    requestResyncTimers();
}
```

Figure 5: Adapted TB Checkpointing Algorithm

cess will begin to undergo a blocking period when it starts the checkpoint establishment. Note that, in addition to ensuring basic global state consistency, the blocking period prevents the following event from happening: after process P saves a checkpoint in stable storage and before the timer of another process P′ expires, P performs an AT and sends P′ a "passed AT" notification message. Therefore, the blocking period also ensures validity-concerned global state consistency.

To prevent the checkpoints from violating validity-concerned recoverability, the adapted TB algorithm makes the process that is engaged in stable-storage checkpoint establishment responsive to confidence change. Specifically, although none of the application-purpose messages sent to a process during its blocking period is read before the blocking period ends, the MDCD protocol lets the incoming "passed AT" notification messages be monitored and allows the process's `dirty_bit` to be updated during the blocking period (see Appendix A). This enables a potentially contaminated process that is engaged in stable-storage checkpoint establishment to adjust its behavior. More specifically, the process will begin its checkpoint establishment with copying its most recent volatile-storage checkpoint to stable storage, and will abort the copying action and replace the contents of the checkpoint with its current process state (equivalent to the state at the moment the blocking period starts) if the value of `dirty_bit` changes from one to zero within the blocking period, as shown in Figure 6(b).

Figures 6(c) and (d) further illustrate how protocol coordination facilitates the establishment of stable-storage checkpoints that satisfy validity-concerned consistency and recoverability, from the perspective of the interacting processes P_1^{act} and P_2. (Recall that there is no application-purpose interaction between P_1^{act} and P_1^{sdw}, and that all of the outgoing messages of P_1^{sdw} are suppressed.)

As shown in Figure 5, the adaptive checkpointing behavior described above is implemented by the function `write_disk`, which takes three arguments. The first argument specifies the contents of the checkpoint that a process should begin with writing to stable storage when its

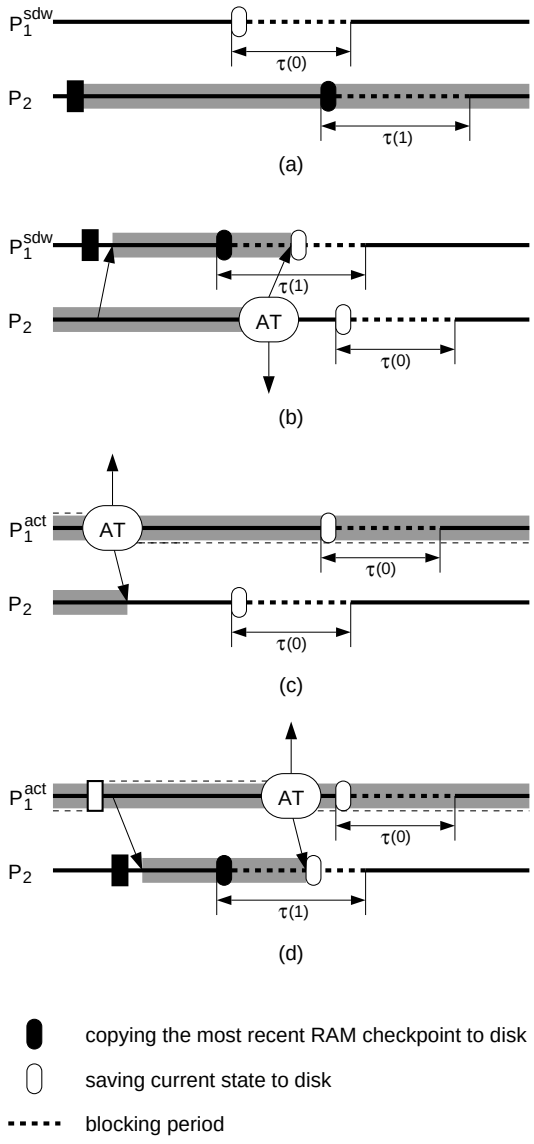

Figure 6: Stable-Storage Checkpoint Establishment based on Protocol Coordination

timer expires; the second argument specifies the value that the process's `dirty_bit`[2] must match, in order to continue writing the contents specified by the first argument to stable storage; and the third argument specifies the alternative checkpoint contents the process should use to overwrite the initial contents, if the value of `dirty_bit` becomes different from that specified by the second argument before the end of the blocking period.

When the adapted TB algorithm is contrasted with the modified MDCD algorithms shown in Appendix A, it is clear that the two entities, N_{dc} and `dirty_bit` (which are

[2]For process P_1^{act}, the second argument will instead specify the value of `pseudo_dirty_bit`.

Table 1: Comparison of Original and Adapted TB Protocols

Attribute	Original TB Protocol	Adapted TB Protocol
Blocking period	$\tau = \delta + 2\rho\phi - t_{min}$	$\tau(b) = \delta + 2\rho\phi + T_m(b)$, where $T_m(b) = b\, t_{max} - (1-b)t_{min}$
Checkpoint contents	Current state	Current state or the most recent volatile-storage checkpoint
Messages blocked	All	All but "passed AT" notification messages
Purpose of blocking	Consistency	Consistency and recoverability[a]

[a]Blocking (after a process is engaged in stable-storage checkpoint establishment) is necessary but not sufficient for validity-concerned recoverability; saving unacknowledged application-purpose messages and re-sending them during recovery (as the original TB protocol does) are also required.

maintained by the TB and MDCD protocols, respectively), play important roles in protocol coordination. Specifically, the TB protocol lets a process choose the contents of a stable-storage checkpoint based on the value of `dirty_bit`, and allows the process to adjust the contents of the checkpoint if the `dirty_bit` changes its value during the blocking period. Whereas the update of the `dirty_bit` during the blocking period is handled by the MDCD protocol in a cautious way. More succinctly, by comparing the stable-storage checkpoint sequence number N_{dc} piggybacked on a "passed AT" notification message with the local N_{dc}, a process decides whether to reset the local `dirty_bit`. (Therefore, it is not possible for a potentially contaminated process to wrongly adjust the checkpoint contents due to a "passed AT" notification message from a process that has already completed its stable-storage checkpoint establishment.)

An underlying objective of the above coordination strategy is to preserve validity-concerned recoverability by preventing a "passed AT" notification message (which would alter our confidence in a process) from becoming "in-transit" in nature. Since 1) a message will not become an in-transit message unless it is received after the timer of the receiving process expires and sent before the timer of the sending process expires, and 2) the adapted algorithm allows a process that is engaged in checkpoint establishment to adjust the contents of the checkpoint in response to a confidence change, it will be sufficient for us to achieve this objective if we can ensure that a potentially contaminated process will receive a "passed AT" notification message (given that it occurs) within the blocking period. Accordingly, we let the blocking period for a process that has a potentially contaminated state be the sum of the maximum clock drift and the maximum message-delivery delay, i.e., $\delta + 2\rho\phi + t_{max}$, where δ is the maximum initial clock deviation, ρ is the clock drift rate, ϕ is the elapsed time since the last clock resynchronization ($\phi = N_{\text{dc}}\theta$, where θ is the length of a checkpointing interval), and t_{max} is the maximum message-delivery delay.

On the other hand, if a process has a non-contaminated state when its checkpointing timer expires, the process will perform checkpointing just like the original TB protocol does. That is, the current process state will be saved to stable storage upon the timer expiration and a blocking period with a length of $\delta + 2\rho\phi - t_{min}$ will be enforced, where t_{min} is the minimum message-delivery delay.

Hence, with the adapted algorithm, the blocking period for a process will be $\delta + 2\rho\phi + t_{max}$ or $\delta + 2\rho\phi - t_{min}$ (the latter coincides with that used by the original TB algorithm), depending upon whether the process is potentially contaminated or not, respectively, at the moment of timer expiration. Accordingly, we formulate the length of a blocking period as $\tau(b) = \delta + 2\rho\phi + T_m(b)$, where b denotes the value of `dirty_bit`, and $T_m(b) = b\,t_{max} - (1-b)t_{min}$.

Table 1 contrasts the original TB protocol with the adapted version, in terms of the length of blocking periods, checkpoint contents, types of messages blocked during a blocking period, and purpose of blocking.

To evaluate the advantages of the protocol-coordination approach over straight extension of a software fault tolerance protocol for handling both software and hardware faults, we have conducted a model-based comparative study. The study focuses on the rollback distance, i.e., the amount of computation quantified in time units (seconds), that a process must undo due to a hardware fault. In particular, we contrast the mean rollback distance of a process when the protocol-coordination scheme is applied ($E[D_{\text{co}}]$) with that when the write-through approach is used ($E[D_{\text{wt}}]$). Figure 7 shows the quantitative results from one of the studies, which reveal that $E[D_{\text{co}}]$ is significantly less than $E[D_{\text{wt}}]$. The significant reduction is due to the fact that the protocol-coordination approach maximizes the likelihood that a process will roll back to its most recent non-contaminated state when a hardware fault occurs. In contrast, with the write-through approach, a process must roll back to the checkpoint that is equivalent to the process's most recent Type-2 checkpoint which does not reflect the

most recent non-contaminated state. Due to space limitations, we omit detailed discussion of the comparative study.

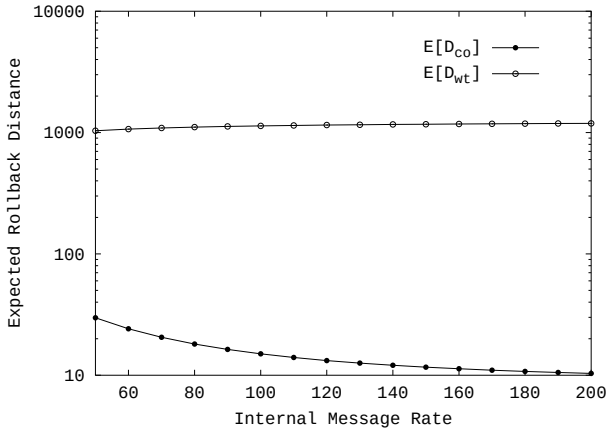

Figure 7: Improvement of Rollback Distance

It is worth noting that the coordination between the MDCD and TB protocols can be enabled or disabled in a seamless fashion. For example, when this approach is used for guarded software upgrading, after the successful completion of an onboard software upgrade, all the software components will be considered high-confidence components; accordingly, the MDCD protocol will go on leave, and each process's `dirty_bit` will have a constant value of zero. This, in turn, leads the adapted TB algorithm shown in Figure 5 to become equivalent to its original version [4].

5 Concluding Remarks

While research efforts often focus on one technical issue at a time, a critical application in real life usually involves multiple technical challenges in fault tolerance and calls for effective and efficient approaches that tackle various problems with a cohesive formulation. Among other enabling-technology integration issues, simultaneous software and hardware fault tolerance in a distributed computing environment poses a major challenge. The effort presented in this paper demonstrates that synergistic coordination between software and hardware fault tolerance techniques is a viable way to respond to this challenge.

In particular, by carrying out algorithm modifications that are conducive to synergistic coordination between volatile-storage and stable-storage checkpoint establishments, we are able to circumvent potential interference between the MDCD and TB protocols, and allow them to effectively complement each other to extend a system's fault tolerance capability. Moreover, the protocol-coordination approach preserves and enhances the features and advantages of the individual protocols that participate in the coordination, keeping the performance cost low.

Another important contribution of this effort is that it fosters the utilization of various state-of-the-art fault tolerance techniques that are available to us for building affordable, highly dependable distributed systems. Our current work is directed toward formally validating the protocol-coordination approach, quantifying its benefits with respect to both dependability enhancement and performance cost reduction, and investigating general guidelines for synergistic coordination between differing fault tolerance techniques.

Recently, we have completed the first version of the middleware (GSU Middleware) that implements the prototype MDCD protocol. We plan to incorporate the protocol-coordination scheme described in this paper into the GSU Middleware, to experimentally assess the effectiveness of the approach.

References

[1] 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.

[2] M. Hecht, J. Agron, H. Hecht, and K. H. Kim, "A distributed fault tolerant architecture for nuclear reactor and other critical process control applications," in *Digest of the 21st Annual International Symposium on Fault-Tolerant Computing*, (Montreal, Canada), pp. 3–9, June 1991.

[3] 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.

[4] 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.

[5] 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," (submitted for publication).

[6] 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.

[7] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[8] Y.-M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Trans. Computers*, vol. 46, pp. 456–468, Apr. 1997.

[9] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "On the effectiveness of a message-driven confidence-driven protocol for guarded software upgrading," *Performance Evaluation*, vol. 44, pp. 211–236, Apr. 2001.

[10] N. Neves and W. K. Fuchs, "Using time to improve the performance of coordinated checkpointing," in *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, (Urbana-Champaign, IL), pp. 282–291, Sept. 1996.

[11] G. P. Kavanaugh and W. H. Sanders, "Performance analysis of two time-based coordinated checkpointing protocols," in *Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, (Taipei, Taiwan), pp. 194–201, Dec. 1997.

A Modified MDCD Algorithms

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (AT(m) == success) {
      pseudo_dirty_bit = 0;
      // maintaining message SN
      msg_SN_P1^act++;
      msg_sending(m, null, null, device);
      // prior messages are valid
      msg_sending("passed_AT", msg_SN_P1^act, Ndc,
                  {P1^sdw, P2});
    } else {
      error_recovery(P1^sdw, P2);
      exit(error);
    }
  } else { // m is an internal message
    // P1^act's dirty bit always equals 1
    m = append(m, dirty_bit);
    msg_SN_P1^act++;
    if (pseudo_dirty_bit == 0) {
      checkpointing(P1^act);
      pseudo_dirty_bit = 1;
    }
    msg_sending(m, msg_SN_P1^act, Ndc, P2);
  }
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT")
    if (m.Ndc == Ndc)
      pseudo_dirty_bit = 0;
  else if (not_in_blocking_period)
    application_msg_reception(m);
}
```

Figure 8: Modified Error Containment Algorithm for P_1^{act}

```
if (outgoing_message_m_ready) {
  // maintaining message SN
  msg_SN_P1^sdw++;
  // suppress and log the outgoing message
  msg_logging(m, msg_SN_P1^sdw, msg_log);
}
if (incoming_message_queue_nonempty) {
  // P1^act or P2 reports a successful AT
  if (m.body == "passed_AT") {
    if (m.Ndc == Ndc) {
      VR1^act = m.msg_SN; // last valid msg of P1^act
      memory_reclamation(msg_log);
      dirty_bit = 0;
    }
  } else if (not_in_blocking_period) {
    // application-purpose message from P2
    if (m.dirty_bit == 1 && dirty_bit == 0) {
      checkpointing(P1^sdw);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 9: Modified Error Containment Algorithm for P_1^{sdw}

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        dirty_bit = 0;
        msg_sending(m, null, null, device);
        msg_sending("passed_AT", msg_SN_P1^act, Ndc,
                    {P1^act, P1^sdw});
      } else
        error_recovery(P1^sdw, P2);
    } else {
      // outgoing msg from a clean state
      msg_sending(m, null, device);
    }
  } else {
    // piggyback dirty_bit on internal message
    m = append(m, dirty_bit);
    msg_sending(m, null, Ndc, {P1^act, P1^sdw});
  }
}
if (incoming_message_queue_nonempty) {
  // must be from P1^act
  if (m.body == "passed_AT") {
    if (m.Ndc == Ndc) {
      msg_SN_P1^act = m.msg_SN;
      dirty_bit = 0;
    }
  } else if (not_in_blocking_period) {
    msg_SN_P1^act = m.msg_SN;
    if (dirty_bit == 0) {
      checkpointing(P2);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
}
```

Figure 10: Modified Error Containment Algorithm for P_2