Distributed Recovery Units: An Approach for Hybrid and Adaptive Distributed Recovery

Nitin H. Vaidya
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
E-mail: vaidya@cs.tamu.edu

Technical Report 93-052
November 1993

Abstract

Traditionally, distributed recovery schemes have been designed for systems consisting of multiple recovery units. Each recovery unit (RU) resides on a single processor and it can fail and recover as a whole. This report introduces the "distributed recovery unit (DRU)" abstraction as an approach for design of "hybrid" and "adaptive" recovery schemes for distributed systems. The distributed system is viewed as a collection of DRUs, each DRU consisting of one or more RUs. This report presents a new recovery scheme based on the DRU abstraction. The proposed approach combines coordinated checkpointing with independent checkpointing and optimistic message logging to obtain a recovery scheme that can effectively trade the overhead during failure-free operation with the overhead during recovery.

1 Introduction

A distributed system is a collection of processes that communicate by sending messages. A number of failure recovery schemes have been designed to provide fault tolerance in distributed systems. Most of these recovery schemes can be divided into two categories: (a) coordinated checkpointing [1, 7, 14] and (b) message logging and independent checkpointing [5, 6, 12]. Coordinated checkpointing schemes require the distributed system to periodically record a consistent state on the stable storage. When a failure occurs, the system rolls back to the most recently recorded consistent state. Message logging schemes require the system to record (on stable storage) the messages sent and/or received during execution. This record of messages is used to recreate a consistent state whenever a failure occurs.

Recovery schemes proposed in the literature typically assume that the system consists of a number of recovery units. Each recovery unit can fail and recover as a whole. Each recovery unit is often simply a process or a single machine. Many recovery schemes often assume that each recovery unit is deterministic. The typical underlying assumption is that the recovery unit resides on a single processor [12]. This report presents a new approach that extends the concept of a recovery unit to a distributed environment and shows utility of the new approach in designing new recovery schemes.

For brevity, we use the term "process" to mean "recovery unit". The main contributions of this report are as follows:

- The report introduces the "distributed recovery unit (DRU)" abstraction as an approach for design of "hybrid" and "adaptive" recovery schemes for distributed systems. The distributed system is viewed as a collection of DRUs, each DRU consisting of one or more processes. (Section 2.3 motivates the proposed approach and Section 5 compares it with other relevant approaches.)
- A new distributed recovery scheme is presented based on the DRU abstraction (Section 3). The proposed scheme combines coordinated checkpointing with independent checkpointing and optimistic message logging. This recovery scheme facilitates a trade-

off between the amount of information logged on the stable storage and the overhead in performing recovery.

- The proposed DRU abstraction is shown to facilitate design of *adaptive* recovery schemes by allowing membership of the DRUs to change *dynamically* (Section 4). This capability is useful in automatically "fine-tuning" the recovery scheme if requirements of an application change over time.

The research presented here is related to previous work by Sistla and Welch [11], Lowry et al. [8] and Johnson [4]. Section 5 presents a comparison of the past work with the research presented here.

Note that the work presented here generalizes an approach presented in [13]. The proof of correctness of the algorithms presented in this report has been omitted here. An extended version of this technical report is expected to be available in the near future.

2 Distributed Recovery Units

A distributed recovery unit (DRU) is simply a collection of processes (possibly on different processors).[1] The distributed system consists of a collection of DRUs. In the extreme, each process may form a DRU by itself or all the processes may belong to the same DRU. There are many ways one may partition the system into DRUs. The actual approach used depends on the application requirements. Here are two possibilities:

- Processes may be assigned to DRUs such that the frequency of messages sent between processes in two different DRUs is small, while the frequency of messages sent between processes in the same DRU is large. When the message communication patterns change over time, a mechanism for dynamically changing the DRU membership can be used.

- Processes executing on some predetermined set of processors may form a DRU. When a process migrates from a processor in one set to another (for example, due to dynamic load balancing), the DRU memberships must be updated to reflect the new configuration.

[1]Recall that we use the term "process" in lieu of the term "recovery unit".

While it is not always necessary, for the recovery schemes presented in this report, it is convenient to have a DRU-leader for each DRU. In this report, we assume that a special process is created exclusively to perform the DRU-leader function for each DRU. Alternately, the function of a DRU-leader can be performed by one of the user processes in the DRU, or the DRU-leader functionality can be distributed to *all* processes within a DRU [13]. The choice made for implementing DRU-leader function affects the recovery scheme.

In this section, we present (i) a simple application of the DRU abstraction, (ii) the general approach for design of recovery schemes based on the DRU abstraction, and (iii) the motivation for the proposed approach.

2.1 A Simple Example

The simple example in this section leads to the design of a recovery scheme identical to the partitioning approach proposed by Sistla and Welch [11].

Consider a system consisting of multiple processes illustrated in Figure 1(a). The recovery scheme for this system requires that (i) each process takes a checkpoint just before sending a message to another process, and (ii) each process logs a received message before using it. This systems can recover from a failure without the domino-effect [9].

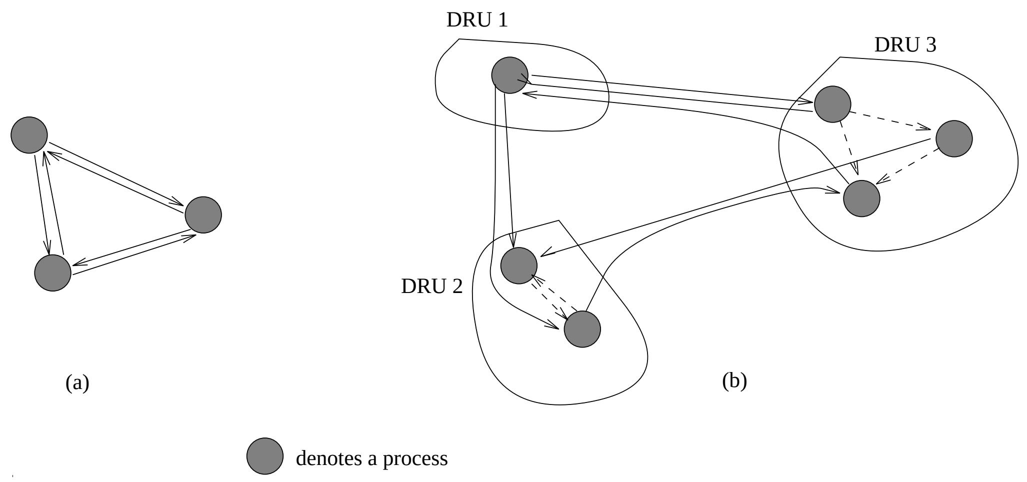

Figure 1: Application of DRU abstraction: A simple example

Now consider a system consisting of multiple DRUs illustrated in Figure 1(b). The recovery scheme for this system requires that (i) any message from a process in one DRU to a process in another DRU is not delivered until the sender DRU has taken a "logical" checkpoint, and (ii) any message received from a process in another DRU is logged before it is used. A *logical* checkpoint consists of a collection of states of the processes such that it can be guaranteed that, in spite of future failures, the state of a process will never roll back to an earlier state (or, this state can always be recreated if necessary). Such a state of a process is said to be *recoverable*. A message sent by a process while in a recoverable state is *committable*. Thus, above recovery scheme requires that a message from one DRU to another DRU is not delivered unless it is committable.

This scheme is identical to the partitioning approach proposed by Sistla and Welch [11] for large distributed system. However, our approach differs from [11] in that the DRU abstraction also facilitates design of recovery schemes that *do not* require messages between DRUs to be committable before they are delivered.

Observe that the system in Figure 1(b) can be obtained by starting with the system in Figure 1(a) and "replacing" a process in Figure 1(a) by a distributed recovery unit (DRU). To put it differently, a DRU in Figure 1(b) behaves *analogous* to a process in Figure 1(a). This example illustrates the basic idea behind the distributed recovery unit approach. The recovery schemes presented later fully exploit the advantages offered by the DRU abstraction.

2.2 The General Approach

The DRU abstraction is used here to design "hybrid" recovery schemes. The general approach may be divided into three steps.

Step 1: Choose two recovery schemes for distributed recovery – a *local* recovery scheme and a *global* recovery scheme. The local recovery scheme is primarily useful in recovering the state of a single distributed recovery unit, and the global recovery scheme is useful in recovering the state of the entire system. In the example above, the chosen global recovery scheme requires each DRU to take a "logical" checkpoint before sending a message to another DRU. The choice of local recovery scheme in that example is arbitrary.

Step 2: Modify the local and global recovery schemes (if necessary) to allow the two schemes to co-exist in the same system. Modifications become necessary whenever the local recovery scheme (without modification) is inadequate to allow a DRU to emulate the behavior of a process as expected by the global recovery scheme.

Step 3: This step is optional depending on whether the membership of processes to the DRUs changes dynamically or not. If the membership of the DRUs changes over time, the local and global recovery schemes may need further modifications. Additionally, protocols must be designed to facilitate DRU membership changes.

The modifications required in the local and global recovery schemes may or may not be minor depending on the choice of the two schemes. The example in the previous section does not require any modifications to the local and global recovery schemes, while the new recovery schemes presented in this report require some simple modifications.

2.3 Motivation

In the literature, a large number of distributed recovery schemes have been proposed (e.g. [2, 5, 6, 12]). None of the recovery schemes have been shown to be suitable for all applications. There are three parameters that may be used to evaluate the performance of a recovery scheme:

1. Overhead during normal (failure-free) operation.
2. Overhead during recovery after failure.
3. Delay in committing an output to the environment.

It is not possible to optimize all the above parameters simultaneously, and improvement in one parameter often results in degradation in another parameter. Therefore, no one (static) recovery scheme can be useful for all applications.

The report uses the distributed recovery units as a tool to design "hybrid" recovery schemes. The hybrid approach allows us to combine two recovery schemes and obtain advantages of both the schemes by choosing appropriate DRU sizes.

- Choosing *small*[2] DRUs results in performance comparable to the *global* scheme.
- Choosing *large* DRUs results in performance comparable to the *local* scheme.
- Intermediate DRU sizes result in "interpolation" between the local and global schemes.

In general, some DRUs may be small and some large.

The hybrid recovery approach based on DRUs can be made "adaptive" by dynamically changing membership of processes to the DRUs. An adaptive recovery scheme can automatically "fine-tune" or adapt as an application's requirements change with time. For example, if at some time the local recovery scheme becomes more desirable then the DRU memberships can be dynamically changed to make each DRU large. The distributed recovery unit abstraction provides a mechanism to introduce adaptive behavior into the recovery schemes. To be able to adapt successfully, however, it is necessary to develop appropriate decision-making mechanisms (e.g. heuristics). Such heuristics have been developed and are a subject of current research [13]. It should be noted that recently Goldberg et al. [3] have also advocated the use of adaptive fault tolerance mechanisms.

To illustrate the general approach outlined above, we first present a recovery scheme assuming that the membership of processes to the DRUs does *not* change over time (Section 3). This recovery scheme is said to be *static* as the DRU membership is fixed. Next, we allow the DRU membership to change dynamically, and present the *dynamic* recovery scheme (Section 4).

3 A Static Recovery Scheme

The system model is as follows. The system consists of many processes communicating with each other via message passing. Each process is assumed to be deterministic, i.e., the final state of a process depends only on its initial state and on the content and order of messages it receives. Each process is assumed to be fail-stop [10]. The communication channels used for message passing are assumed to be reliable and first-in-first-out (FIFO).

[2]A *small* DRU contains few processes and a *large* DRU contains a large number of processes.

Step 1 of the general procedure outlined earlier requires us to choose local and global recovery schemes. We choose the local recovery scheme to be the coordinated checkpointing scheme by Chandy and Lamport [1] and the global recovery scheme to be an optimistic message logging scheme by Johnson and Zwaenepoel [5]. The optimistic logging scheme by Strom and Yemini [12] is also an excellent choice for the global recovery scheme. Our goal here is to demonstrate the utility of our approach in designing new and useful recovery schemes. We have chosen the scheme from [5] as an example for its ease of understanding.

The basic idea here is to implement a DRU using a (modified) coordinated checkpointing scheme such that each DRU would behave analogous to a *deterministic* process, as expected by the chosen global recovery scheme. The processes within different DRUs take checkpoints independently, as in the chosen global recovery scheme. However, the processes within a given DRU record their state in a consistent manner as governed by the chosen local recovery scheme. All messages in the system are treated “optimistically” (i.e. messages need not be committable before they are delivered).

Coordinated checkpointing schemes, in general, require a smaller stable storage as compared to message logging schemes. However, message logging schemes typically require a smaller re-execution overhead during recovery as compared to coordinated checkpointing schemes (with comparable failure-free overhead). Similarly, message logging schemes can also achieve smaller output-commit time with comparable failure-free overhead. By combining these two approaches, the proposed recovery scheme facilitates a trade-off between stable storage size, re-execution overhead during recovery and output commit delays.

As noted in step 2 in Section 2.2, the local and global recovery schemes need to be modified to allow them to co-exist. The following describes the static recovery scheme after the modifications are made. To begin with, we present some definitions and describe some data structures.

3.1 Preliminaries

The execution of each process can be divided into non-overlapping intervals. Each interval, called a *state interval* is initiated by the receipt of a message. Each state interval of a process can be identified by a unique *state interval index* [5], which is incremented by one each time a message is received by the process.

Each message is tagged by its *send sequence number* (SSN), the sender process identifier (sender-id), and identifier of the DRU (DRU-id) to which the sender process belongs. *Send sequence number* (SSN) of a message denotes the position of the message in the stream of messages sent by its sender process.

A message that is sent by a process in one DRU to a process in another DRU is said to be an *inter-DRU* message; any other message is said to be an *intra-DRU* message. When a process receives a message it can determine that the message is an inter-DRU (intra-DRU) message if its own DRU-id mismatches (matches) with the DRU-id tagged to the message. The solid and dotted arrows in Figure 1(b) denote inter-DRU and intra-DRU massages, respectively.

When DRU membership can change dynamically (Section 4), the DRU of each process may change over time. In this case, each process remembers DRU-id of each DRU it has been in. If a message is tagged with any of these DRU-id's, it is treated as an intra-DRU message. (Actually it is sufficient to remember identifiers of the DRUs the process has *recently* been in.)

For future reference, we define an *obsolete* state interval. A state interval β (or state during that interval) of a process is said to be *obsolete* if the state of the process is rolled back to a state interval prior to β **and** state interval β is not guaranteed to be reproduced subsequently. If state interval β_P of a process P depends on an obsolete state interval β_Q of process Q, then β_P is also obsolete.

3.2 Data Structures

Each user process maintains the following data structures. The DRU-leader needs to maintain fewer data structures (listed later).

- A *DRU-id*, which is the unique identifier of the DRU containing the process.
- A *DRU-leader* identifier, to enable a process to communicate with its DRU-leader.

  It is assumed that it is adequate to know the identifier of a process to be able to communicate with it. When this is not true, physical location of the process may be included in each process identifier.
- A *checkpoint number (CN)*, numbering each checkpoint taken by the process. Each message is tagged with the current checkpoint number (CN) of the sender process. (For the dynamic recovery scheme in Section 4, CN would denote the number of checkpoints the process has taken in its current DRU.)
- A *state interval index*, which counts the number of messages received by the process. The state interval index is incremented each time the process receives a message. Each message is tagged with the current state interval index of the sender process.
- A *dependency vector* [5] which records the largest index of any state interval of each other process on which this process directly depends.
- An *order information buffer* (in volatile memory) for intra-DRU messages received by the process. For each intra-DRU message, the order information includes identifier of the sender process, the SSN (send sequence number), sender's state interval index tagged to the message, and index of the state interval started by this message. Note that the order information does *not* include the message data. The order information is logged by writing this buffer to stable storage.
- A *message buffer* (in volatile memory) for inter-DRU messages received by the process. For each inter-DRU message, the message data *and* its order information are recorded in the message buffer. The messages are logged by writing this buffer to stable storage.

The DRU-leader maintains *DRU-id* and *checkpoint number (CN)* defined above, and a *DRU-set*, containing identifiers of all processes in its DRU.

The distributed recovery scheme consists of two parts: (a) failure-free operation and (b) failure recovery protocol. The following describes each of them.

3.3 Failure-free operation

Checkpointing: The procedure for checkpointing is based on the chosen *local* recovery scheme, namely, the coordinated checkpointing scheme by Chandy amd Lamport [1]. When a DRU-leader wants to initiate a checkpoint, it increments its own *checkpoint number (CN)*, takes a checkpoint and sends a *marker* message to each process in its DRU. A DRU-leader may decide to initiate a checkpoint at the request of a process in its DRU or based on some heuristic. For simplicity, it is assumed that the DRU-leader does not initiate a checkpoint unless all processes in its DRU have taken the previous checkpoint. (The DRU-leader can verify this condition by looking at the stable storage.)

When a process receives an intra-DRU message (*marker* message or otherwise) tagged by a checkpoint number larger than its own, the receiver process, before acting on the message, sets its own checkpoint number equal to that tagged with the message and takes a checkpoint. When a process receives a marker message tagged by a checkpoint number no larger than its own, it ignores the marker message.

A checkpoint of a process is considered *tentative* until all processes in its DRU have taken a corresponding checkpoint. A checkpoint of a process includes the state interval index during which the checkpoint is taken, as well as the dependency vector of the process at the time of checkpointing. A checkpoint of a DRU-leader includes all the data structures it maintains, namely, *DRU-set*, *DRU-id* and the *checkpoint number (CN)*.

Messages: Whenever a new message is received, the DRU-id tagged to the message is compared with the DRU-id of the receiver process to determine if the message is an inter-DRU message or an intra-DRU message. An inter-DRU message is saved in the *message buffer* of the receiving process. When an intra-DRU message is received, its order information is saved in the *order information buffer.* Message buffer and order information buffer are written to the stable storage periodically or when they get full.

Dependency vector update: On receiving each new message, the entry for the sender process in the dependency vector of the receiving process is set equal to the larger of its current value and the state interval index tagged to the message.

In addition to above, the dependency vector needs to be updated to add what we call "pseudo-dependencies" corresponding to each checkpoint taken by the process. However, these updates need not be performed during the checkpointing operation. These updates can be performed periodically or when a failure occurs. We describe the pseudo-dependency updates in Section 3.4.

3.4 Failure Recovery

In the event that a DRU-leader fails, its state is restored from its most recent checkpoint. If there are any additional failures, then procedure described below is invoked.

For correctness of the recovery procedure described below, we need to modify the dependency vectors as follows to add "pseudo-dependencies". Consider a DRU, say DRU1, whose processes have taken checkpoint number CN1. Let processes P and Q have taken checkpoint CN1 during state intervals σ_P and σ_Q, respectively. Then, update dependency vector of checkpoint CN1 of process P (i.e. dependency vector of σ_P) to reflect that state interval σ_P depends on state interval σ_Q of process Q. This in turn implies that each state interval of process P following σ_P also depends on σ_Q of Q. Therefore, these implied updates are also made to the dependency vectors. These updates are performed for each process pair in each DRU (for each checkpoint). In practice, these update may also be performed during normal operation. However, some of the updates may not be completed when a failure occurs, and the incomplete updates will have to be performed after the failure by looking at the checkpoints on the stable storage.

The consequence of the above update to the dependency vector is that if process P above rolls back to a state interval prior to σ_P, then Q must also roll back to a state interval prior to σ_Q. In other words, if checkpoint CN1 of process P is obsolete, then checkpoint CN1 of each process in its DRU (including process Q) is also obsolete.

To be able to recover from failures, it is necessary to first determine a recoverable system state. Johnson and Zwaenepoel [5] have presented algorithm FIND_REC to determine the "maximum recoverable system state" provided a list of "stable" state intervals is available. We use algorithm FIND_REC as well, except that the definition of "stable" state intervals in [5] needs to be modified to suit our implementation using DRUs. To define stable state intervals, we first need to define an *effective* checkpoint [5].

Definition 1 *The effective checkpoint for a state interval σ of some process P is the checkpoint on stable storage for process P with the largest state interval index ϵ such that $\epsilon \leq \sigma$ [5].*

Assuming the above meaning of ϵ, we present a definition of stable state intervals. The definition is recursive due to condition 2 below.

Definition 2 *A state interval σ of process P is stable iff for $\epsilon < \alpha \leq \sigma$ following is true:*

1. *If the message that started state interval α is an inter-DRU message then it has been logged.*
2. *If the message that started state interval α is an intra-DRU message then: (a) its order information has been logged,* **and** *(b) the state interval of the sender process when this message was sent is stable.*

State interval 0 of each process is always stable. Each other state interval is initially marked *not stable*. These state intervals subsequently become stable. The above definition suggests recursive algorithm `is_stable` in Figure 2 to determine which state intervals are *stable*.

When a failure (single or multiple) is detected, the following steps are performed sequentially to recover from the failure:

```
**is_stable** ( $\sigma$, P )
{
if state interval $\sigma$ of process P is marked as *stable*
        return TRUE
if process P took a checkpoint during interval $\sigma$
        mark $\sigma$ as *stable* and return (TRUE)
if (**is_stable**($\sigma - 1$, P) = FALSE)
        return (FALSE)
if $\sigma$ is initiated by an inter-DRU message M
        if (message M is not logged on the stable storage)
               return FALSE
        else mark $\sigma$ as *stable* and return (TRUE)
else if $\sigma$ is initiated by an intra-DRU message M from process Q
        if order information for M is *not* logged
               return FALSE
        else if **is_stable**($\gamma$, Q), where $\gamma$ is the state interval of process Q when M is sent
               mark $\sigma$ as *stable* and return (TRUE)
        else return FALSE
}
```

Figure 2: Recursive procedure for determining *stable* state intervals

1. DRU-leader of each fault-free DRU (i.e. a DRU that does not contain a faulty process) takes a coordinated checkpoint of that DRU.

2. All tentative checkpoints are deleted. A checkpoint CN1 of process P is tentative if all processes in its DRU have not taken checkpoint CN1.

3. Dependency vectors are updated as described earlier to add the pseudo-dependencies.

4. Procedure `is_stable` in Figure 2 is used to determine which state intervals are stable. Subsequently, algorithm FIND_REC from [5] is performed at the stable storage to determine the maximum recoverable system state. For each process, algorithm FIND_REC determines the state interval of that process that is a part of the maximum recoverable system state. For process P, let $\mu(P)$ denote the state interval of process P that belongs to the maximum recoverable system state. Also, let $C(P)$ denote the effective checkpoint of state interval $\mu(P)$ of process P.

5. Due to the pseudo-dependencies, $C(P)$ must be identical for all processes P in a given DRU. To prove this, assume the contrary. Consider processes P and Q in a particular DRU. Let $C(P) \neq C(Q)$. Without loss of generality assume that $C(P) < C(Q)$. Let state intervals of P and Q when checkpoint number $C(P)$ of the DRU was taken be $\sigma_P^{C(P)}$ and $\sigma_Q^{C(P)}$. Similarly, let state intervals of P and Q when checkpoint number $C(Q)$ was taken be $\sigma_P^{C(Q)}$ and $\sigma_Q^{C(Q)}$. As $\mu(P)$ and $\mu(Q)$ are a part of the maximum recoverable state, $\sigma_P^{C(P)}$ and $\sigma_Q^{C(Q)}$ must also be recoverable. By the pseudo-dependency update procedure, $\sigma_Q^{C(Q)}$ depends on $\sigma_P^{C(Q)}$ and vice-versa. Also, $\sigma_P^{C(Q)} > \mu(P) \geq \sigma_P^{C(P)}$.[3] As $\sigma_P^{C(Q)}$ is not a part of the recoverable state, $\sigma_Q^{C(Q)}$ cannot be recoverable. This contradicts a previous statement. Therefore, this proves that $C(P) = C(Q)$.

   As the effective checkpoint (above) for all processes within a DRU is the same, let $C(DRU)$ denote the effective checkpoint for all processes in a given DRU.

[3]If $\sigma_P^{C(Q)} = \sigma_P^{C(P)}$, then it implies that P took checkpoints $C(P)$ and $C(Q)$ during the same state interval. In this case, we can pick $C(P) = C(Q)$ without resulting in an inconsistency.

6. For each DRU, the DRU-leader is restored to its state at checkpoint $C(DRU)$.

7. The state of all processes in a DRU can be recovered independent of other DRUs. Therefore, we consider any one DRU, say DRU1. Each process in DRU1 is rolled back to checkpoint number $C(DRU1)$ and reexecuted. For process P to be able to reach state $\mu(P)$, it must re-receive all the messages it had received previously since checkpoint $C(DRU1)$. In addition, the order in which messages were received must also be the same.

   The inter-DRU messages received by each process are obtained from the message log on the stable storage. Only order information for the intra-DRU messages is available in the stable storage. However, the intra-DRU messages will be reproduced as the processes in DRU1 execute from checkpoint $C(DRU1)$. This is guaranteed by the definition of a stable state interval. (Note: A recoverable state interval is always stable.) The order information for the intra-DRU messages is used to decide the order in which they are delivered when reproduced during recovery.

   Recovery is completed when each process P reaches state interval $\mu(P)$.

Any failures occurring during recovery can be handled by initiating the recovery again.

3.5 Garbage Collection

A checkpoint of process P can be deleted when a subsequent checkpoint becomes the effective checkpoint for the state interval of process P in a recoverable state. Any messages or order information logged prior to a deleted checkpoint can also be deleted.

4 A Dynamic Recovery Scheme

The recovery scheme presented above assumes that the membership of processes to DRUs is statically decided. However, it is possible to adapt the recovery scheme to time-varying needs of an application if the membership of the processes to DRUs can change over time.

This section presents *DRU-fork* and *DRU-merge* protocols for allowing DRUs to partition and merge, as well as the checkpointing and recovery procedures.

The DRU-leader initiates each of the *DRU-fork*, *DRU-merge* and checkpointing procedures described below. The DRU-leader initiates these procedures in a mutually exclusive manner. That is, the DRU-leader *does not* initiate any of these procedures while any one of them is in progress. Additionally, none of these procedures are initiated while recovery is in progress. These restrictions can be relaxed at the cost of a more complex recovery procedure.

4.1 Additional Data Structures

For the dynamic recovery scheme, new data structures are needed in addition to those presented earlier in Section 3.2. The following additional data structures are maintained by each process.

- A *version-number*. The version number is initialized to one. It *may* sometimes be incremented during a DRU-fork or DRU-merge operation. Note that the *version-number* is different from *incarnation* numbers used in many algorithms [12].

- A *DRU-id-list*, containing DRU identifiers of all DRUs that this process has been in. (As time progresses, it is possible to delete older entries in the *DRU-id-list*. We omit the algorithm here.)

  Version-number and *DRU-id-list* are both saved with the process checkpoint.

- Temporary variable to be used during the DRU-fork and DRU-merge protocols. These variables are introduced below as and when needed.

4.2 Maintaining Unique DRU Identifiers

When DRU membership can change dynamically, it is necessary to design a protocol for maintaining unique DRU identifiers. Many different mechanisms can be used for maintaining unique DRU identifiers. This section presents one such mechanism and [13] presents another.

Identifiers of the processes are assumed to form a linear order. Given this assumption, identifier of a DRU is given by a tuple *(proc-id , vers-num)* where *proc-id* is the largest identifier of any process in that DRU and *vers-num* is the *version-number* of that process. Because DRUs can merge and fork, the DRU identifiers are subject to change. The *DRU-fork* and *DRU-merge* protocols are designed to appropriately update DRU identifiers when DRU memberships change.

4.3 *DRU-merge* Protocol

DRU-merge protocol is used to merge two different DRUs into one DRU. The protocol requires active participation of the DRU-leaders of the merging DRUs. We assume that based on some decision-making mechanism (e.g. heuristics) the two DRU-leaders have decided to merge the two DRUs.

The DRU-leaders first determine the DRU-id of the merged DRU. If the DRU-id's for the two DRUs are DRU1 = (pid1, version1) and DRU2 = (pid2, version2) then the following procedure is used to determine the new DRU-id = (pid, version).

```
If (pid1 > pid2)
     pid = pid1
     version = 1 + version1
     new-DRU-leader <-- leader of DRU1
else
     pid = pid2
     version = 1 + version2
     new-DRU-leader <-- leader of DRU2
```

The above procedure also determines which DRU's leader will become the DRU-leader for the merged DRU. The new-DRU-leader then determines membership of the merged DRU by asking the other DRU-leader for its DRU-set. The other DRU-leader terminates itself after performing these steps.

Next, the new-DRU-leader sends *new-dru-id (pid, version, new-DRU-leader)* messages to all processes in their respective DRUs. When a process receives a *new-dru-id (pid, version, new-DRU-leader)* message, it (a) updates its *next-DRU-id* to equal (pid, version), (b) if its process identifier is *pid* then it increments its own version number, (c) updates its *next-DRU-leader* to equal *new-DRU-leader*, (d) sends a *ack-new-dru-id* message back to *next-DRU-leader*, and (e) adds the new DRU-id to the DRU-id-list. Until the new-DRU-leader has received *ack-new-dru-id* messages from all the processes in the new DRU, the new-DRU-leader cannot initiate a checkpoint for its DRU. When all the *ack-new-dru-id* messages have been received, the new-DRU-leader for the merged DRU can initiate a coordinated checkpoint of the merged DRU. The merged DRU is considered to have come into existence starting from this checkpoint. The checkpointing procedure described later is used to checkpoint the new DRU.

4.4 *DRU-fork* Protocol

A DRU-leader may decide using some decision-making mechanism (e.g. heuristics) to partition the DRU into two DRUs. First, the DRU-leader determines the DRU-ids for the two partitions (partitions 1 and 2) using the following procedure.

```
pid1 = largest of the process identifiers in partition 1.
version1 = 1 + version number of process pid1.
DRU-id for partition 1 will be (pid1, version1).

pid2 = largest of the process identifiers in partition 2.
version2 = 1 + version number of process pid2.
DRU-id for partition 2 will be (pid2, version2).
```

The DRU-leader spawns two DRU-leaders, one for each DRU to be formed by the DRU-fork protocol. Let their identifiers be *leader1* and *leader2*, respectively. The DRU-leader sends leader1 and leader2 messages containing their respective DRU-sets and DRU-ids. The new leaders initialize their checkpoint number (CN) to 0. The DRU-leader then sends *new-dru-id(pid1, version1, leader1)* message to all processes in partition 1 and *new-dru-id(pid2,* *version2, leader2)* message to all processes in partition 2, and then terminates itself. When a process receives a *new-dru-id (pid, version, new-DRU-leader)* message, it (a) updates its *next-DRU-id* to equal (pid, version), (b) if its process identifier is *pid* then it increments its own version number, (c) updates its *next-DRU-leader* to equal *new-DRU-leader*, (d) sends a *ack-new-dru-id* message back to *next-DRU-leader*, and (e) adds the new DRU-id to the DRU-id-list. Until a new-DRU-leader (i.e. leader1 or leader2) has received *ack-new-dru-id* messages from all the processes in its DRU, it cannot initiate a checkpoint. When all *ack-new-dru-id* messages have been received, the new-DRU-leader for a forked DRU can initiate a coordinated checkpoint of its DRU. A new DRU (i.e. partition 1 or 2) is considered to have come into existence starting from this checkpoint. The checkpointing procedure described later is used to checkpoint the new DRUs.

Observe that the procedures for *DRU-fork* and *DRU-merge* are very similar. They differ primarily in the way the new DRU identifiers are determined.

4.5 Failure-free operation

Checkpointing: The checkpointing algorithm presented earlier needs to be modified to accommodate the *DRU-fork* and *DRU-merge* protocols presented earlier. As before, when a DRU-leader wants to initiate a checkpoint, it increments its own *checkpoint number (CN)*, takes a checkpoint and sends a *marker* message to each process in its DRU.

When a process receives a message (*marker* or otherwise) tagged by a DRU-id identical to its current DRU-id and tagged by a checkpoint number larger than its own, it sets its own checkpoint number equal to that tagged with the message and takes a checkpoint before using the message.

When a process receives a message (*marker* or otherwise) tagged by DRU-id equal to next-DRU-id, the receiver process, before acting on the message, (a) sets its own checkpoint number equal to the CN tagged to the message, (b) sets its DRU-id equal to next-DRU-id, (c) sets DRU-leader equal to next-DRU-leader, (d) sets its next-DRU-id equal to NULL, and (e) takes a checkpoint.

When a process receives a marker message not covered by above two cases, it ignores the marker message.

Messages: Messages and order information is logged identically to the static recovery scheme. The only difference here is that a message is considered to be an intra-DRU message if the tagged DRU-id is identical to *any* DRU identifier in the DRU-id-list. The reason is illustrated using Figure 3. Figure 3 shows DRU1 consisting of processes P, Q, R, S and T. DRU1 performs

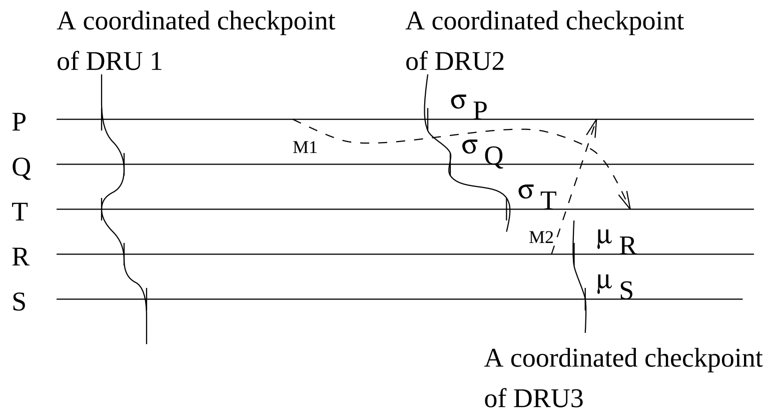
A coordinated checkpoint of DRU 1
A coordinated checkpoint of DRU2
A coordinated checkpoint of DRU3
P
Q
T
R
S
M1
M2
σ_P
σ_Q
σ_T
μ_R
μ_S

Figure 3: Example of the dependency vector update procedure

DRU-fork and as a result, DRU2 and DRU3 are formed. DRU2 consists of processes P, Q and T, while DRU3 consists of R and S. For future reference, let the processes (P, Q, T) in DRU2 take the first checkpoint during state intervals σ_P, σ_Q and σ_T respectively. Similarly, let the processes (R, S) in DRU3 take the first checkpoint in DRU3 during state intervals μ_R and μ_S respectively.

As the DRU-id tagged to the message is found in the DRU-id-list, messages M1 and M2 in Figure 3 are both treated as intra-DRU messages by their receivers. As elaborated in the next subsection, if M1 or M2 need to be reproduced for failure recovery, the sender and receiver processes will rollback to a checkpoint taken while in DRU1.

Dependency vector update: Dependency vector update during failure-free operation is identical to the static recovery scheme. However, the pseudo-dependency update procedure needs to be modified. The modified procedure is described below.

4.6 Failure Recovery

After a failure is detected, dependency vectors of the processes are updated to add pseudo-dependencies. These updates ensure that if a process P rolls back to a checkpoint CN1, then *all* processes in the DRU, say DRU1, of process P when it took checkpoint CN1 also roll back to checkpoint CN1. Note that process P may no longer be in DRU1. In fact, DRU1 may not even exist when the failure is detected, as it may have forked or joined another DRU. The pseudo-dependency update procedure has two components:

- The first component updates the dependency vectors as described at the beginning of Section 3.4. For Figure 3, the first component would add pairwise dependencies between state intervals σ_P, σ_Q and σ_T (and similarly add dependencies between μ_R and μ_S).

- The second component is required to update dependency vectors of the processes in DRUs that have forked into new DRUs. Instead of a precise procedure, we illustrate the procedure with the example of Figure 3. The second component of the pseudo-dependency update procedure would add pairwise dependencies between σ_P, σ_Q, σ_T, and μ_R, μ_S. For example, σ_P would depend on μ_S, and vice-versa. As before, the dependency vectors for all state intervals subsequent to σ_P, σ_Q, σ_T, μ_R and μ_S are updated to take into account the above updates.

The rest of the recovery protocol is identical to that described earlier in Section 3 with one addition. After the recovery is completed, all processes (including the DRU-leaders) clean up all the state information regarding the DRU-merge and DRU-join protocols. In other words, if a DRU-merge (DRU-fork) was in progress when a failure occurred, the protocol must be re-initiated after recovery.

5 Comparison with other relevant schemes

Sistla and Welch [11] propose an approach that partitions a system into multiple clusters. The messages passing between the clusters are treated as input-output messages. As shown in Section 2, this scheme can be obtained using our approach as well. However, in general, our approach does *not* require messages between DRUs to be treated as input-output.

Lowry et al. [8] suggested an optimistic scheme for systems partitioned into clusters. This is an improvement over Sistla and Welch [11] where all messages across the clusters are treated pessimistically. Lowry et al. design interfaces between clusters that allow (i) each cluster to use different recovery schemes and (ii) the messages between two clusters to be logged optimistically. Our work differs from [8] in its goals and the approach used. Our goal is to design a *single* recovery scheme for a distributed system that can adapt to the requirements of an application. The distributed recovery units abstraction is used as a means for achieving this goal. The goal in [8] is to build large distributed systems such that each partition can use its recovery scheme without any modifications.

Johnson [4] proposed an output-driven optimistic message logging and checkpointing scheme, in which recording of needed recovery information on stable storage is driven by the need to commit output to the outside world. Johnson allows each process to individually choose between messages logging and checkpointing, or only checkpointing. This flexibility can be used to adapt the performance of the recovery scheme to the needs of an application. Our approach differs from Johnson's approach in that we achieve adaptability by means of the DRU abstraction. We believe that our approach as well as Johnson's approach are both good candidates for the design of adaptive schemes. Suitability of either approach in practice needs to be evaluated further.

6 Conclusions

This report describes a novel approach for designing distributed recovery schemes. The proposed approach is based on the *distributed recovery unit* (DRU) abstraction. With this abstraction, the system is viewed as a collection of DRUs, each DRU consisting of one or more processes.

A *hybrid* recovery scheme for such a system is designed using two other schemes, named *local* and *global* recovery schemes. The performance of the hybrid recovery scheme is comparable with the local (global) recovery scheme when the number of processes in each DRU is large (small). In general, the hybrid recovery scheme "interpolates" between the local and global recovery schemes. This feature of the hybrid recovery scheme allows the hybrid scheme to be made *adaptive* by dynamically changing the membership of various DRUs. The changes in DRU membership can be made using the dynamic recovery scheme presented in the report. Depending on the requirements of the application, the DRU sizes may adaptively chosen to be small or large (or, some DRUs may be small, and others large).

Design and evaluation of good heuristics to exploit the adaptive capability of the proposed approach is a subject of ongoing research.

References

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 63–75, February 1985.

[2] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Computers*, vol. 41, May 1992.

[3] J. Goldberg, I. Goldberg, and T. F. Lawrence, "Adaptive fault tolerance," in *IEEE Workshop on Advances in Parallel and Distributed Systems*, pp. 127–132, October 1993.

[4] D. B. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," in *Symposium on Reliable Distributed Systems*, pp. 86–95, October 1993.

[5] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *Journal of Algorithms*, vol. 11, pp. 462–491, September 1990.

[6] T. Juang and S. Venkatesan, "Crash recovery with little overhead," in *International Conf. Distributed Computing Systems*, pp. 454–461, 1991.

[7] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 13, pp. 23–31, January 1987.

[8] A. Lowry, J. R. Russell, and A. P. Goldberg, "Optimistic failure recovery for very large networks," in *Symposium on Reliable Distributed Systems*, pp. 66–75, 1991.

[9] D. L. Russell, "State restoration in systems of communicating processes," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 183–194, March 1980.

[10] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comp. Syst.*, vol. 1, pp. 222–238, August 1983.

[11] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," in *Proc. ACM Symp. on Principles of Distributed Computing*, pp. 223–238, August 1989.

[12] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, pp. 204–226, August 1985.

[13] N. H. Vaidya, "Dynamic cluster-based recovery: Pessimistic and optimistic schemes (preliminary version)," Tech. Rep. 93-027, Comp. Sc. Dept., Texas A&M Univ., May 1993.

[14] Y. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Symposium on Reliable Distroibuted Systems*, pp. 78–85, October 1993.