Checkpoint/Rollback in a Distributed System Using Coarse-Grained Dataflow*

David Cummings and *Leon Alkalaj*
Jet Propulsion Laboratory
California Institute of Technology

Abstract

The Common Spaceborne Multicomputer Operating System (COSMOS) is a spacecraft operating system for distributed memory multiprocessors, designed to meet the on-board computing requirements of long-life interplanetary missions. One of the main features of COSMOS is software-implemented fault-tolerance, including 2-way voting, 3-way voting, and checkpoint/rollback. This paper describes the COSMOS distributed checkpoint/rollback approach, which exploits the fact that a COSMOS application program is based on a coarse-grained dataflow programming paradigm and therefore most of the state of a distributed application program is contained in the data tokens. Furthermore, all computers maintain a consistent view of this dynamic state, which facilitates the implementation of a coordinated checkpoint.

1 Introduction

Computer systems on board interplanetary robotic spacecraft must operate reliably for many years in the presence of hostile environmental conditions such as temperature, radiation, and vibration stress that can cause both transient and permanent hardware failures. The Common Spaceborne Multicomputer Operating System (COSMOS) [4] has been developed jointly by NASA's Jet Propulsion Laboratory and Langley Research Center to address the needs of such reliable, real-time robotic missions.

The main features of COSMOS include: software-implemented fault-tolerance through process replication with 2-way or 3-way voting, and distributed program checkpoint and rollback; dynamic program patching without requiring system shut-down; support for multiprocessing, including load balancing and distributed clock synchronization; portability (COSMOS is written in ANSI C, and executes on top of commercially available real-time operating systems); and a software development environment that includes tools for the high-level specification and verification of flight software.

COSMOS was originally designed for execution on the MAX fault-tolerant distributed memory multicomputer which was built at JPL [13, 14]. The MAX machine consists of 10 processor/memory modules connected by two interconnection networks. The first network is a Byzantine resilient broadcast bus used for the reliable exchange of COSMOS synchronization and control messages. The second network is a point-to-point circuit-switched network used for the transfer of data among the computer modules. In this environment, COSMOS executes as a layer on top of the VRTX real-time operating system. Although it was originally designed for the MAX hardware, COSMOS can be ported to any distributed system containing a reliable, atomic broadcast capability that ensures that messages are delivered correctly and in the same order to all processors. COSMOS also assumes the presence of at least one stable storage device. For deep-space missions, stable storage would typically consist of a redundant pair of solid state recorders cross-strapped to two of the processors. COSMOS is currently being ported to several platforms including a commercially available shared memory multicomputer consisting of MC68040 processors running the VxWorks real-time operating system, and a network of UNIX workstations.

The focus of this paper will be a description of the distributed checkpoint/rollback approach used by COSMOS. A number of techniques for checkpoint/rollback in distributed systems, designed to avoid the well-known domino effect [12], have been proposed [6]. In *coordinated checkpointing*, all processes in the system, or an interacting subset of these processes, perform a checkpoint together [2,

*This work was supported by NASA's Office of Advanced Concepts and Technology (OACT) and the JPL Center for Space Microelectronics Technology (CSMT) Division for Spacecraft and Remote Sensing.

10, 16, 17]. *Loosely-synchronized checkpointing* reduces the coordination overhead through the use of loosely-synchronized clocks [18]. *Independent checkpointing* allows processes to be checkpointed independently, and uses message logging [3, 8, 9, 15] or message scheduling [19] to eliminate or reduce rollback propagation. The COSMOS approach to checkpoint/rollback is a form of coordinated checkpoint which takes advantage of the COSMOS data-driven execution model that requires all processors to maintain a consistent view of the state of the distributed application program throughout its execution. Rollback in COSMOS automatically adapts to any changes in the number of processors due to failures since the checkpoint was taken.

The outline of this paper is as follows. Section 2 contains an overview of the COSMOS data-driven execution environment. In Section 3, we describe the checkpoint/rollback algorithm and implementation in detail. Section 4 contains preliminary performance measurements of checkpoint overhead that were made using a software simulator of the MAX hardware. Although a MAX simulator was used, the checkpoint/rollback approach applies to general distributed memory configurations. In Section 5, we present concluding remarks.

2 COSMOS execution environment

2.1 Dataflow graph representation and distributed graph state

An application program in COSMOS is described by a graph in which the nodes represent functions that are executed as processes, and the arcs represent communication paths for the exchange of messages (tokens). Each node contains one or more inputs, and zero or more outputs. Arcs connect node outputs to node inputs. A node is said to *fire*, that is, a process instance is created and starts executing, when there is at least one token on each of its inputs. Multiple instances of a node can be fired in parallel if there are sufficient tokens on the inputs. In comparison to other dataflow programming paradigms, the COSMOS programming model can be described as coarse-grained [1]. A node represents a high-level process (as opposed to instruction-level dataflow). The application code for each node is written in a procedural language (C or Ada), whereas each node firing is applicative in that its results can depend only upon the current set of inputs, and not on any state retained from a previous firing. Because of this, the dynamic state of an executing graph is embodied, for the most part, in the current token configuration. If a node needs to retain state across firings, it must send the state information back to itself as a token.

In the distributed COSMOS environment, each processor/memory module contains a copy of the COSMOS kernel and a complete copy of the data structures that describe the static configuration of each graph in the system (number of nodes in the graph, the arcs connecting them, etc.). In addition, every copy of the kernel maintains a consistent up-to-date record of the dynamic state of each graph, including the number of tokens currently on each arc, and which processes are currently in execution on each processor. This dynamic state is updated through synchronization messages that are sent over the broadcast bus whenever a process completes execution. Such a *process completion* message contains information on the number of tokens that were produced on each output of the completing process. However, the token data are not contained in the message. Instead, the data remain on the originating processor and are transferred later over the data network, as needed, to any other processors requiring them for subsequent process firings.

In response to a process completion message, each copy of COSMOS performs two steps:

1. It *consumes* the input tokens used by the completing process. Token consumption consists of marking a token as no longer in use by a particular arc, and then deleting the token if it is no longer in use by any arcs.

2. It checks to see if any process is now fireable as a result of the availability of the tokens produced by the completing process. If so, all processors eligible to execute the process(es) compete for execution rights via the exchange of messages over the broadcast bus.

The deletion of input tokens and creation of output tokens is handled by COSMOS as one logically atomic operation. Therefore, if a process does not complete successfully due to a hardware fault, the input tokens are still available for a process retry on a different processor. (Other systems have taken similar advantage of applicative programming in their approaches to error recovery [5, 7, 11].) There are some error situations for which a process retry may not be adequate, for example, faults producing incorrect results which are not detected immediately, or faults affecting more than one processor. A distributed rollback can be used to recover from these more global error conditions.

The COSMOS distributed checkpoint/rollback algorithm is based on the consistent view of the dynamic graph state maintained on all processors, together with the fact that this dynamic state is largely embodied in the tokens. A checkpoint operation essentially consists of copying the current set of tokens in a graph to stable storage. [1] A rollback operation consists of re-introducing a saved set of tokens into the graph, and re-executing all processes as a result of the availability of these tokens as inputs. However, there are several features of COSMOS that force the checkpointing algorithm to be more complex than simply saving the current set of tokens. These features are: out-of-order process completions; process voting; and outputs that branch into multiple arcs, feeding the inputs of more than one node. These issues will be introduced in the following two sections, and their impact on checkpoint/rollback will be discussed in section 3.

2.2 Out-of-order process completion

Since several instances of the same node can execute concurrently, out-of-order process completion may occur due to the use of heterogeneous processors, processor load imbalance, clock drift, etc. COSMOS ensures the correct logical ordering of node firings in the presence of out-of-order completions through the use of *generation numbers*. Each firing of a node is assigned a sequential process generation number (*pgen* number). When a token is created as a result of a process completion, it is assigned a token generation number (*tgen* number) that is equal to the *pgen* number of the process firing that produced it. Each arc has a generation number (*agen* number) that identifies the next expected token generation number for that arc. If a token arrives on an arc and its *tgen* number does not match the arc's *agen* number, then the token is queued until all tokens with earlier generation numbers arrive first and are used for firings.

The data-driven execution of a COSMOS program is illustrated in Figures 1a through 1d using a COSMOS graph fragment consisting of three nodes, *N1*, *N2* and *N3*. These nodes take a, b, c, and x as inputs, and together compute the value $ax^2 + bx + c$. (In order to achieve reasonable performance, COSMOS nodes must in general be more coarse-grained than in this simple example.) Figure 1a shows a set of initial tokens with data values (in boxes) and generation numbers (above the boxes). For example, there are two tokens on the input arcs labeled x that are shared by nodes *N1* and *N2*. The first token, with $tgen = 1$, has the value 2, and the second token, with $tgen = 2$, has the value 5. Assume in this example that the *pgen* number to be assigned to the next firing of each node in the figure is 1, and the *agen* number for each arc is also 1. With the configuration shown in Figure 1a, one instance of *N1* (with $pgen = 1$) and two instances of *N2* (with $pgen = 1$ and $pgen = 2$) can start executing. In Figure 1b, *pgen* 1 of node *N1* has completed execution. It has consumed and deleted the input token on the arc labeled a, and has produced an output token with $tgen = 1$ and value 16. Note that the input token on the arc labeled x has not been deleted because it is still being used by *N2*. When *pgen* 1 of *N2* completes, COSMOS will recognize that that input token is no longer needed by any node, and the token will be deleted. In order to track a token's status for this purpose, COSMOS maintains a *consumption* bit mask for each token that indicates the set of arcs for which the token is still needed. When all bits in this mask are zero, the token is no longer needed on any arc and can be deleted.

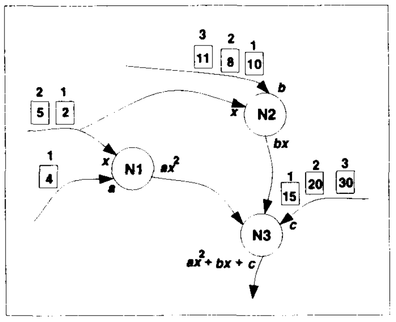
If *pgen* 1 of *N2* is the next process to finish executing, the state of the graph will be as shown in Figure 1c. At this point, there are sufficient inputs to fire one instance of *N3*, using $ax^2 = 16, bx = 20, c = 15$. However, if instead *pgen* 2 of *N2* were to complete before *pgen* 1 of *N2*, then we would have the situation shown in Figure 1d. In that case, even though there is a token on each input of *N3*, *N3* must *not* fire, because the token from *N2* is the wrong generation number (2 instead of 1). This is recognized by COSMOS, because the *agen* number for the arc bx is 1. The generation 2 token will not be used by *N3* until a generation 1 token becomes available and is used first.

2.3 Process voting

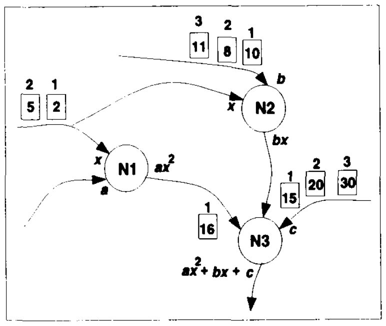
COSMOS supports 2-way and 3-way voting. The following discussion will be limited to 3-way voting. Figure 2a shows a fragment of a graph that includes three identical nodes, *V1*, *V2*, and *V3*, that execute on three different processors. (These are three nodes containing the same application code, as opposed to three concurrent *instances* of a single node.) Each of these three nodes receives inputs from *N1* and sends outputs to *N2*. (The second output on *N1* serves as a reminder that a node can have multiple outputs).

As shown in the figure, voting in COSMOS is performed at a node's input, which in this example is the input to *N2*. As soon as two of the three replicated processes have produced results, a comparison is performed by COSMOS. If the two results are in

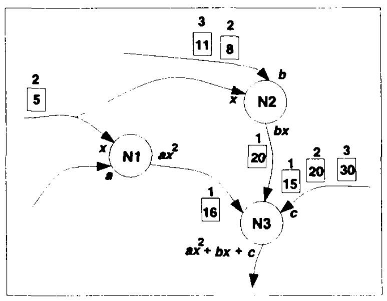
[1] Each graph represents an independent application program. Checkpoint and rollback are performed on a per-graph (i.e., per-application) basis.



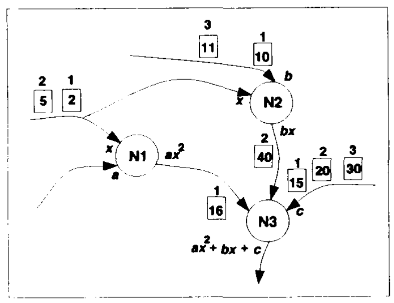
(a) COSMOS graph fragment



(b) Graph state after pgen 1 of N1 completes



(c) Graph state after pgen 1 of N2 completes



(d) Graph state after out-of-order process completion

Figure 1: COSMOS graph execution.

agreement, *N2* fires, even though the third input has not yet arrived. To support this performance feature, a *placeholder token* is created for the third node. In this example, a token from *N1* with generation number 1 has been produced, causing *V1*, *V2*, and *V3* to fire. *V1* and *V2* have completed and produced tokens which match, so a placeholder token (depicted by a dotted box) is placed on *V3*'s output. *N2* can now fire, since its input has been satisfied.

If *N2* then completes before *V3*, the situation will be as shown in Figure 2b. The two tokens from *V1* and *V2* have been deleted, because their consumption masks indicate they have been used on their respective arcs. However, since *V3* has still not finished executing, the token produced by *N1* cannot be deleted—its consumption mask indicates that it has been used on only two of the three arcs. Also, since the real token has not yet replaced the placeholder token on *V3*'s output, the placeholder token remains even though its two companion tokens on *V1*'s and *V2*'s outputs have been deleted. When the real token is finally produced by *V3*, COSMOS will detect that it is no longer needed and it will be deleted (after a checksum comparison with the placeholder token, which contains the checksums produced and agreed upon by the other two nodes). The placeholder token and the token on *N1*'s output will also be deleted at this time.

3 Checkpoint/rollback design and implementation

The dynamic state of a distributed application program in COSMOS can be partitioned into three sets: the current set of tokens; the set of dynamic state variables associated with nodes and arcs (e.g., *pgen* numbers and *agen* numbers); and the set of currently-executing processes.

The checkpoint/rollback approach that was chosen saves a minimum amount of state information in the checkpoint file, and reconstructs the rest during rollback. In particular, only the set of tokens and an essential subset of the dynamic state variables are saved. During rollback, the tokens are re-introduced as if they

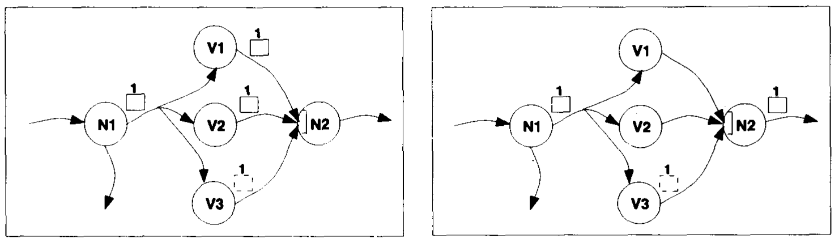

V1
N1
V2
N2
V3
1
(a) Graph fragment with voting
(b) Graph state if N2 completes before V3

Figure 2: 3-way voting and placeholder tokens.

were being announced for the first time due to process completions, and the normal COSMOS process firing logic is used to recreate the set of executing processes that existed at checkpoint time. This approach to rollback automatically adapts to changes in hardware configuration (processors that may have been added or removed) since the checkpoint.

Checkpoint and rollback are both initiated by application software, via simple calls to system routines. The COSMOS kernel makes no assumptions about checkpoint frequencies or about when a rollback should occur. Once a checkpoint or rollback is requested, the operation itself is transparent to the application.

3.1 Checkpoint overview

A checkpoint operation is divided into two phases. Phase 1 is the *token snapshot phase*, and phase 2 is the *checkpoint file write phase*. Phase 1 is started when an application program requests a checkpoint, which causes a synchronization message to be broadcast to all processors. In response to this message, all copies of COSMOS simultaneously mark the current set of tokens. (Even though each processor does not have a copy of all token data, each processor has the exact same view of the control information for each token. This includes which output produced the token, and the token's *tgen* number and arc consumption mask.) The token marking serves two purposes: it identifies which tokens should be copied to stable storage during phase 2, and it also prevents the tokens from being deleted until the checkpoint operation has completed. While this marking is occurring, all application processes are suspended from execution, to ensure a consistent snapshot of the current token state. In addition to marking the tokens, the processor that has physical access to the checkpoint storage device records information about each graph node that will be used during phase 2. This information includes a derived *pgen* and *agen* number for each node.

Once phase 1 has completed, the processor with physical access to the checkpoint device begins phase 2, copying the tokens and associated control information to a checkpoint file on stable storage. During phase 2, normal application process execution proceeds in parallel with the checkpoint operation. A system checkpoint process executes as a VRTX task, alongside user processes executing as other VRTX tasks. The checkpoint process requests token data from other processors as needed, much the same way as input token data are requested from other processors at the start of normal process execution. If an error occurs during the checkpoint, then a rollback to the previous version of the checkpoint file can be performed. (Recovery from errors during checkpoint has only been partially implemented to date.)

After the checkpoint process has finished the copy to stable storage, it broadcasts a synchronization message to all processors informing them that the checkpoint operation is complete. In response to this message, each copy of COSMOS removes the checkpoint marking from every marked token, and deletes those tokens whose arc consumption masks are now zero.

3.2 Rollback overview

A rollback operation is also divided into two phases. Phase 1 is the *graph cleanup phase*, and phase 2 is the *token reannouncement phase*. Phase 1 is started when an application program requests a rollback, which causes a synchronization message to be broadcast to all processors. In response to this message, all copies of COSMOS simultaneously remove all traces of the dynamic state of the graph, namely, all tokens and process executions. Once the cleanup operation is finished, phase 2 of rollback begins.

During phase 2, a processor with physical access to the checkpoint storage device reads information out of the checkpoint file, and announces the tokens in much the same way as output tokens are announced when a process completes execution. However, before the token reannouncement can occur, certain graph state variables must be updated in order for COSMOS to be able to correctly process the tokens that will be reannounced. The *pgen* number for each node must be reset to its value at checkpoint time, and the *agen* number for each arc must also be reset. This information is stored in the checkpoint file, and is read in by the rollback processor and broadcast over the bus. In response to these broadcast messages, each copy of COSMOS updates the graph state variables in its local copy of the graph. After all of this information has been read in and distributed, the rollback processor reads the token information out of the file. In the file, tokens are grouped according to the graph node that produced them. Within a single node's token set, the tokens are further partitioned into subsets according to their *tgen* numbers. This allows the rollback software to reannounce the tokens using broadcast messages that are very similar to process completion messages. First, all tokens for node 1, generation k are announced, as if *pgen* k of that node had just completed execution. Then, all tokens for node 1, generation $k + 1$ are announced. This continues for all generations of node 1, then node 2, etc. As these tokens are reannounced, the copy of COSMOS on each processor assesses the fireability of nodes due to the availability of the tokens. Process instances are readied for execution as usual. However, the actual process executions are deferred until all tokens have been reannounced, for reasons that will be discussed in section 3.4.

When the end of the checkpoint file is reached, the rollback software broadcasts a synchronization message indicating "end of rollback". At this point, the copy of COSMOS on each processor enables process executions. This causes the set of processes that was in execution at the time of checkpoint to start executing again, using the same set of inputs as existed at checkpoint time. However, the specific processors on which these processes execute will in general be different than at checkpoint time.

It should be noted that at least two versions of the checkpoint file must be retained (and exactly two versions are retained in the current implementation), for the following reason. Because of the way voting is implemented, there could be a latency period between the time a fault occurs and the time it is detected by the voting logic. This could result in bad tokens being written to the checkpoint file. The rollback logic must ensure that it uses a clean version of the checkpoint file. This requires that the maximum fault latency be known, and that the rollback logic use the earlier version of the checkpoint file if the time since the last checkpoint is less than this maximum fault latency. For example, if a rollback is triggered due to a 3-way vote mismatch, and the time since the last checkpoint is less than the maximum fault latency, then the most recent checkpoint file could contain the two or three bad tokens that caused the 3-way mismatch. (Determination of maximum fault latency is application-specific. Basically, it is the maximum time a token can be queued on an arc feeding a voted node. If a queueing analysis cannot be performed to determine this maximum delay, then a COSMOS tool called the *dataflow animator* can be used to estimate this value.)

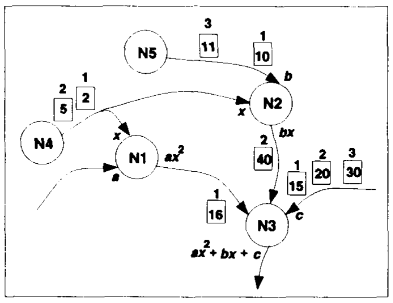
3.3 Checkpoint file description

For each graph node, the following information is stored in the checkpoint file: next *pgen* number for the node, *agen* number for all arcs emanating from the node, and a list of the tokens produced by the node that existed at checkpoint time. The tokens are grouped by *tgen* number. For each token, the token data and the token's arc consumption mask are included.

Each graph node can have multiple outputs, and each output can have multiple arcs emanating from it. This means that there can be many *agen* numbers to save for each node. In order to reduce the size of the checkpoint file, only a single *agen* number is stored for each node. This *agen* number is obtained during phase 1 of checkpoint, by examining the *agen* numbers of all output arcs and the *tgen* numbers of all output tokens on that node. The *agen* number that is stored is the minimum of these values. At the beginning of phase 2 of rollback, the *agen* number of each arc emanating from the node is set to this value. Then, as tokens are reannounced, the arc consumption masks of the tokens are used to increment the *agen* numbers accordingly.

In the checkpoint file we must also account for *null tokens*, which result from generations of a node that have completed execution out of order but have not produced any tokens on one or more outputs. [2] For this purpose, null token indicators are included in the file. A null token indicator is actually used for two purposes: to represent a generation that completed

[2]Null tokens are simply a convenient bookkeeping device for handling this situation. They were introduced into COSMOS for reasons unrelated to checkpoint/rollback, but they must be handled properly by the checkpoint/rollback logic.



(a) COSMOS graph fragment for checkpoint

N1 Next pgen = 2, agen = 1
tgen = 1, Data = 16, Mask = 1
N2 Next pgen = 1, agen = 1
tgen = 2, Data = 40, Mask = 1
N3 Next pgen = 1, agen = 1
N4 Next pgen = 3, agen = 1
tgen = 1, Data = 2, Mask = 2
tgen = 2, Data = 5, Mask = 1
N5 Next pgen = 4, agen = 1
tgen = 1, Data = 10, Mask = 1
tgen = 2, NULL TOKEN
tgen = 3, Data = 11, Mask = 1

(b) Checkpoint file contents

Figure 3: COSMOS checkpoint example.

but did not produce tokens on an output, and to represent a generation that completed and produced tokens that have already been used and deleted. However, if a token is missing at checkpoint time because it has not yet been produced, even though tokens with higher *tgen* numbers have already been produced (out-of-order completion), we *don't* store a null token indicator in the file. During phase 2 of checkpoint, these cases must be distinguished so the software can determine whether or not to insert null token indicators into the file. Placeholder tokens add some complexity in making this distinction.

Figure 3a shows a COSMOS graph fragment that can be used to illustrate some of these ideas. This figure is a copy of Figure 1d, with two additional nodes shown. We assume that the same processes are in execution, with the same *pgen* and *agen* numbers, as in the earlier example. Figure 3b shows the contents of the checkpoint file for these 5 nodes.

The checkpoint file contains 5 entries, one per node. For *N1*, the next firing will be assigned *pgen* 2, and the *agen* number for its arc is 1. There is a single token for *N1* in the file, and its arc consumption mask has a single bit set. This indicates that there is one arc on which the token has not yet been consumed. *N4* has two tokens, and their different arc consumption masks reflect the fact that each has been consumed on a different arc. The *tgen* 1 token has been consumed on the arc feeding *N1*, but at checkpoint time it was still in use by an execution of *N2*. The *tgen* 2 token has been consumed on the arc feeding *N2*, but has not yet been assigned to a firing of *N1*. *N5* has a null token indicator for *tgen* 2, because that token has already been deleted.

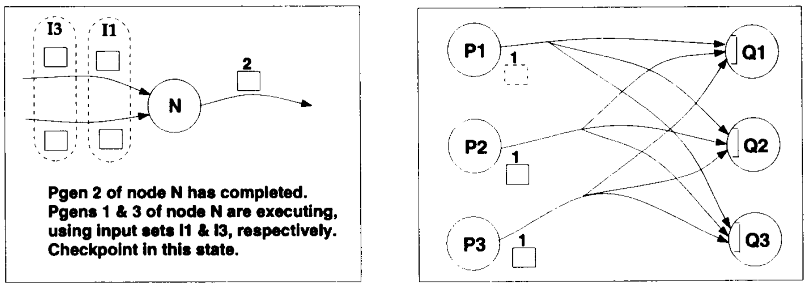
3.4 Out-of-order process completions

Earlier, it was mentioned that process executions are deferred until rollback has completed. This is a consequence of out-of-order process completions, as can be seen in the following example. Consider a node with three process instances in execution, *pgen* 1, *pgen* 2 and *pgen* 3. Assume *pgen* 2 completes, producing a *tgen* 2 token. At this time, a checkpoint is taken. This is shown in Figure 4a.

The checkpoint file will contain the *tgen* 2 token. It will also contain the two sets of input tokens that caused the firings of *pgen* 1 and *pgen* 3. Call these sets of input tokens *I1* and *I3*, respectively. At the beginning of phase 2 of rollback, the *pgen* number for the node will be reset to 1, and sometime later the tokens will be reannounced. There are several possible orderings of token reannouncements during rollback. Consider the following ordering:

1. The tokens in the set *I1* are reannounced. This causes *pgen* 1 of the process to be readied for execution. The *pgen* number associated with the node is incremented to 2, in preparation for the next firing of the process.

2. The tokens in the set *I3* are reannounced. This causes *pgen* 2 of the process to be readied for execution. The *pgen* number associated with the node is incremented to 3, in preparation for the


I3
I1
2
N
Pgen 2 of node N has completed.
Pgens 1 & 3 of node N are executing, using input sets I1 & I3, respectively.
Checkpoint in this state.
P1
P2
P3
1
1
1
Q1
Q2
Q3

(a) Checkpoint with out-of-order completions (b) Checkpoint with placeholder token

Figure 4: More checkpoint examples.

next firing of the process. However, because process executions are not allowed to commence, the record describing the *pgen* 2 execution is guaranteed to remain in the graph until after rollback completes. That is, *pgen* 2 will not complete and announce output tokens until after the rollback is finished.

3. The *tgen* 2 token is reannounced. The rollback logic detects that there is a *pgen* 2 version of the process readied for execution, i.e., that there is a generation number conflict. The *pgen* number for that version of the process is incremented to 3. Also, the *pgen* number associated with the node (which will be used for the next firing) is incremented from 3 to 4.

Note that if process executions were allowed to commence before rollback completed, it is possible for the *pgen* 2 execution, which is readied in response to the *I3* tokens in step 2, to complete and announce output tokens before the *tgen* 2 token is reannounced in step 3. These output tokens would be tagged with *tgen* 2 instead of *tgen* 3, which would be incorrect.

3.5 Voting and branching arcs

Even if a graph contains triplicated (3-way voted) nodes which can mask a single fault, checkpoint/rollback could still be necessary for that graph if recovery from multiple faults is required. Placeholder tokens must be handled specially by the checkpoint/rollback code. This can be illustrated by the following example.

Figure 4b shows a fragment of a COSMOS graph containing two sets of triplicated nodes. The first set, {*P1*, *P2*, *P3*}, supplies voted inputs to the second set, {*Q1*, *Q2*, *Q3*}. There is a *tgen* 1 token on *P2*'s and *P3*'s output, but *pgen* 1 of *P1* is still executing. A placeholder token with *tgen* = 1 has been created for *P1*'s output. Because the voted inputs for *Q1*, *Q2* and *Q3* are satisfied, a process instance for each of these nodes is created and begins execution. Assume that *Q2* completes and consumes the three tokens (including the placeholder token) on the three arcs feeding it. *P1*, *Q1* and *Q3* are still executing, and a checkpoint is performed.

If a rollback takes place later using this checkpoint, then the question of how to handle the placeholder token arises. If we just save the two real *tgen* 1 tokens in the checkpoint file, we can recreate the placeholder token during rollback based on the availability of these other two tokens, with essentially the same logic that was used to create the placeholder prior to checkpoint. However, when it is created during rollback, its arc consumption mask must reflect the fact that the token has already been consumed on the arc from *P1* to *Q2*. In addition, the *agen* number of the arc from *P1* to *Q2* must be changed from 1 to 2 during rollback, to reflect the fact that the next expected token on this arc will have a *tgen* number of 2. After experimenting with several ways of handling these problems, it became clear that the cleanest solution was to store explicit placeholder tokens in the checkpoint file and "reannounce" them during rollback, even though placeholder tokens are never announced during normal process completions.

4 Performance measurements

Due to unavailability of the MAX hardware, the performance measurements presented here were made using a software simulator of the MAX hardware and VRTX. This simulator, together with the COSMOS code, executes as a single UNIX process. (The same COSMOS code that runs on the MAX hardware is linked in with the simulator, which provides the underlying services normally provided by MAX or VRTX.) These tests were run on a SUN SPARCstation 10.

We measured average CPU utilization per token for a checkpoint operation. Response times were not measured, because the simulator does not faithfully reproduce the timing characteristics of the MAX hardware. (The simulator was originally intended as a COSMOS software development platform, not as a performance measurement tool.) The total number of tokens in the graph at the time of checkpoint varied from 27 to 2322. Four different token sizes were used: 8 bytes, 80 bytes, 400 bytes, and 800 bytes. Due to the fact that a simulator was used, the measurements will not necessarily reflect the overhead that would be incurred in a real target environment. However, they should provide a reasonable ballpark estimate of the CPU overhead for a checkpoint operation on a SPARC 10-class processor. (In fact, these measurements are probably conservative, because some processing that would execute in parallel on a real target, such as token marking, executes serially in the simulator.)

The hardware environment that was simulated consisted of three processor/memory modules. Roughly one-third of the token data were resident on the processor performing the checkpoint, and the other two-thirds had to be copied over the network. This data copying represents a significant portion of the checkpoint overhead. As the number of processors increases, there will be a corresponding increase in checkpoint overhead due to this copying. If there are n processors, then on average the fraction of tokens that must be copied for a checkpoint is $(n-1)/n$. For large n, this value approaches one, which is 50% greater than the 2/3 value that applies to our 3-processor simulation. Therefore, for large n the average token copying overhead should be approximately 50% greater than in our tests.

The results of the tests are plotted in Figure 5. For small tokens, the CPU overhead per token is in the 1-2 millisecond range on a SPARCstation 10. The checkpoint frequency is application-dependent, as are the average number of tokens in the graph and the average size of the tokens. If a graph contained 1000 small tokens and a checkpoint were taken every minute, the CPU overhead incurred would be approximately 3%. If instead the graph contained 1000 800-byte tokens, the CPU overhead incurred for 1-minute checkpoints would be approximately 11%. (We suspect that the steep initial slope of the curve for 800-byte tokens is

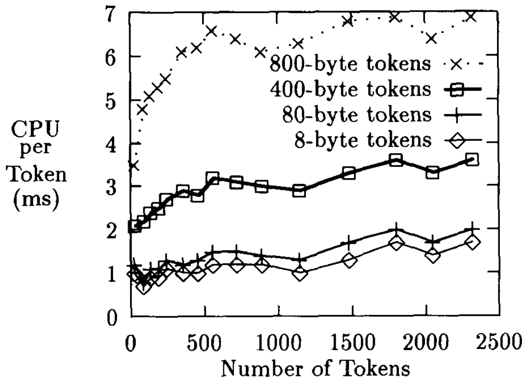


Figure 5: CPU utilization per token for checkpoint.

an artifact of the simulator. There are plans to investigate this further using a real target, once the COSMOS port to VxWorks is completed.)

5 Concluding remarks

The COSMOS distributed checkpoint/rollback software takes advantage of the COSMOS data-driven execution environment, in which all processors maintain a consistent view of the dynamic state of the distributed application. Only the tokens, which represent the messages between application processes, are checkpointed. The rest of the application process state is recreated during rollback. There is no need to save internal process state such as stack contents. A potential advantage of this approach is that it tends to minimize the amount of information required for a checkpoint. On the other hand, there tend to be more messages in a COSMOS system than in a conventional distributed system, because any internal process state that must be retained across firings is "looped back" in the form of tokens. Also, the relatively low coordination overhead at checkpoint time is made possible because of the background level of coordination overhead that is sustained continually in order to maintain a consistent view of the graph state on all processors. This background overhead is related to the granularity of the application—the more coarse-grained the application, the lower the background synchronization overhead. In addition, there are a number of complexities due to out-of-order process completions, voting, and branching arcs that make the checkpoint/rollback software error-prone. Extensive testing and analyses were required in order to gain a high degree of confidence in the correctness of the implementation. One unexpected result of this work was that the checkpoint/rollback capability proved to be a very useful tool for testing COSMOS itself. By performing thousands of checkpoint/rollback sequences in rapid succession, COSMOS was forced into states that had not previously been observed. A number of bugs in the baseline COSMOS system, unrelated to checkpoint/rollback, were uncovered and fixed as a result.

Acknowledgements

Blair Lewis was responsible for major portions of the checkpoint/rollback design and implementation. Douglas Blough and Loring Craymer contributed many valuable suggestions, and Julie Wang provided very helpful user feedback.

References

[1] R. G. Babb, " Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer*, pp. 55–61, July 1984.

[2] G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," *Proc. of 13th International Conference on Fault-Tolerant Computing*, pp. 48–55, June 1983.

[3] A. Borg, J. Baumbach and S. Glazer, "A Message System Supporting Fault Tolerance," *Proc. of 9th ACM Symposium on Operating Systems Principles*, pp. 90–99, October 1983.

[4] L. G. Craymer, *et al.*, "Common Spaceborne Multicomputer Operating System and Development Environment Technical Specification," *JPL Technical Document D-11525*, Caltech, JPL, February 1994.

[5] R. Finkel and U. Manber, "DIB—A Distributed Implementation of Backtracking," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 2, pp. 235-256, April 1987.

[6] T. M. Frazier and Y. Tamir, "Application-Transparent Error-Recovery Techniques for Multicomputers," *Proc. of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 103–108, March 1989.

[7] D. H. Grit, "Towards Fault Tolerance in a Distributed Applicative Multiprocessor," *Fourteenth International Conference on Fault-Tolerant Computing: Digest of Papers*, pp. 272–277, June 1984.

[8] D. R. Jefferson, "Virtual Time," *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, July 1985.

[9] D. B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pp. 14–19, June 1987.

[10] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, pp. 23–31, January 1987.

[11] F. C. H. Lin and R. M. Keller, "Distributed Recovery in Applicative Systems," *Proc. of the 1986 International Conference on Parallel Processing*, pp. 405–412, August 1986.

[12] B. Randell, P. A. Lee and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* Vol. 10, No. 2, pp. 123–165, June 1978.

[13] R. D. Rasmussen, G. S. Bolotin, N. J. Dimopoulos, B. F. Lewis and R. M. Manning, "Advanced General Purpose Multicomputer For Space Applications," *International Conference on Parallel Processing*, pp. 54–57, August 1987.

[14] R. D. Rasmussen, N. J. Dimopoulos, G. S. Bolotin, B. F. Lewis and R. M. Manning, "MAX: Advanced General Purpose Real-Time Multicomputer For Space Applications," *Real-Time Systems Symposium*, pp. 70–78, December 1987.

[15] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems*, Vol. 3, No. 3, pp. 204–226, August 1985.

[16] Y. Tamir and C. H. Sequin, "Error Recovery in Multicomputers using Global Checkpoints," *Proc. of 13th International Conference on Parallel Processing*, pp. 32–41, August 1984.

[17] Y. Tamir and T. M. Frazier, "Application-Transparent Process-Level Error Recovery for Multicomputers," *Hawaii International Conference on System Sciences-22*, Vol. I, pp. 296–305, January 1989.

[18] Z. Tong, R. Y. Kain and W. T. Tsai, "Rollback Recovery in Distributed Systems Using Loosely Synchronized Clocks," *IEEE Trans. On Parallel and Distributed Systems*, pp. 246–251, Vol. 3, No. 2, March 1992.

[19] Y.-M. Wang and W. K. Fuchs, "Scheduling Message Processing for Reducing Rollback Propagation," *Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, pp. 204–211, June 1992.