Proceedings of the 28th Annual Hawaii International Conference on System Sciences — 1995

Using Directed Graphs to Describe Entity Dependency in Stable Distributed Persistent Stores

Rasool Jalili and Frans A. Henskens

Basser Department of Computer Science
The University of Sydney, N.S.W., 2006, Australia
{rasool,frans}@cs.su.oz.au

Abstract

In a persistent object store, the acts of modifying data and reading modified data result in the creation of dependencies between the modifying process and the data. Dependencies may be represented using sets, and over time these may grow to encompass many objects and processes. Checkpoint and roll-back operations must propagate to all elements in such a set. This paper presents a new notation for representing dependencies, and shows that differentiating between the dependencies created by modifying data and reading modified data reduces the extent of propagation of checkpoint and roll-back operations.

Keywords: Stability, Dependency, Persistent Systems, Fault-tolerant Systems, Checkpoint, Roll-back.

1 Introduction

Persistent systems [1] provide uniform mechanisms for the manipulation of short-term and long-term data. Achieving such uniformity requires an abstract store often called persistent store [4]. According to [2, 5] such a store should appear to be of unbounded size and to be failure-free. The failure-free property of such stores is often referred to as store stability.

A Distributed Persistent Store (DPS) provides access to a shared network-wide persistent store for users of computers connected to a network. The secondary or backing storage, which typically occurs on disk, may be centralised (eg Casper [20]) or distributed across the networked nodes (eg Monads DSM [8]). Implementation of a DPS introduces issues not associated with single node persistent stores. Of our concern in this paper is the appearance of being failure-free (stability). Stability of a DPS can be achieved by stepping the store through a sequence of global consistent states [10, 13], requiring the implementation of distributed atomic updates.

From time to time computer systems unexpectedly fail, due either to hardware or software faults or loss of power. Such failures may result in loss of the contents of volatile memory (RAM), while the contents of the non-volatile memory typically remain unchanged. As persistent stores provide uniform management of objects, the transfer of data between volatile and non-volatile memory is transparent to the user. At any instant the state of the store is represented by the combination of the contents of the volatile and non-volatile memories. Since the contents of volatile memory are typically lost after system failure, a stable persistent store must be able to revert after failure to some consistent state described in non-volatile memory.

Techniques which make such reversion possible for persistent systems are typically based on regularly flushing the volatile system state to non-volatile storage (checkpointing or stabilising) and reverting to the most recent checkpoint state after failure (roll-back) [10]. For a DPS to be stable there must be at least one global consistent and non-volatile recorded state at any instant. It is desirable that this recorded state is close to the state at the time of failure, thus minimising the loss of modifications to the store in the event of roll-back.

Shadow paging has been used as a mechanism for implementing a roll-back recovery technique in single-node persistent stores (eg. [4, 8]). In the interval between checkpoint states two versions are maintained for all modified virtual pages; the stable or *shadow* version, and the modified or *current* version. During a checkpoint operation, all pages modified since the last checkpoint are flushed to backup store creating new shadow versions, and the disk blocks occupied by the previous shadow versions are de-allocated [18]. A checkpoint operation is assumed to be atomic and may be initiated as a periodic action, as part of orderly system shutdown, or as part of some higher level mechanism such as transaction commit.

During recovery from a failure, or on system restart, the system commences operation from the most recent stable state. In the case of recovery from a system failure, this is equivalent to a roll-back operation and results in the loss of modifications achieved since the last checkpoint. Reducing this loss may be achieved by increasing the frequency of checkpoints; this degrades

system performance because processing must be suspended for parts of the store undergoing a checkpoint. This degradation is particularly relevant if the entire store is stabilised as a single operation. In an attempt to reduce the impact of checkpoint operations on store useability, recent stability schemes based on shadow paging checkpoint parts of the store separately from each other [10].

This approach to stability requires consideration of the physical and logical dependencies which may be created in the interval between checkpoints [20]. Such dependencies may require parts of the store to be checkpointed together. While the roll-back operation has not been explicitly included in this and some subsequent discussion, its inclusion may be implied by the reader since checkpoint and roll-back operations are effectively inverses of each other.

Shadow paging has been used to provide stability in DPSs (eg. Casper [13] and Monads-DSM [8, 10]). In these systems consistency of the store is maintained in the case of both node and network failure; this is not as straightforward as for single-node stores because of the possibility that dependencies may involve multiple nodes.

In the following discussion we refer to a virtual page as *clean* if it is has not been modified since it was last checkpointed, otherwise it is called *dirty*. Objects are assumed to be paged entities equivalent to, for instance, files or programs in conventional systems. An object may also be referred to as clean or dirty; this is with respect to the status of the page currently being accessed. Thus one process may see an object as clean while simultaneously another process may see the same object as dirty. The term *entity* refers to an object or a process in this discussion.

Processes in distributed systems may communicate through passing messages or accessing a global shared memory [16]. In the message passing model of process communication, the state of a process P_1, after receiving a message from another process P_2, depends on the state of P_2. Likewise, in the shared memory model of process communication, the state of a process P_1 accessing a dirty page modified by another process P_2, depends on the state of P_2. In this paper we concentrate on dependencies in shared memory systems, investigating the application of entity-based stabilisation using directed graphs to describe dependencies between entities. We show that the use of directed graphs allows separate description of checkpoint and roll-back dependencies, thus improving store efficiency. Further aspects of entity dependency including two phase checkpoint and roll-back are discussed.

2 Entity dependency in shared memory environments

Access by multiple processes to data objects typically results in inter-dependence between the processes and the objects. For example, when a process P_1 accesses a dirty data object D_1 previously modified by an other process P_2, its subsequent behaviour may be affected according to the modifications achieved by P_2. The states of P_1, D_1, and P_2 become inter-dependent as a result of the access performed by P_1. During normal operation on the store, sets of such dependent entities may be created. Such sets of dependent entities have been termed associations [20]. It is important for the logical integrity of the store that such dependent entities are checkpointed together. Checkpointing an entity belonging to an association necessitates the checkpointing of all entities in the association. The roll-back of an entity, likewise, necessitates the roll-back of all other entities belonging to the same association. Reducing the size of associations[1] improves store performance by curbing the propagation of checkpoint and roll-back operations from one entity to other dependent entities.

2.1 Implications of dependencies in single-node persistent stores

Dependency of objects in conventional systems is less critical than that in persistent systems due to the separation between management of main memory (RAM) and backup memory (file store). A process in a conventional system explicitly writes all its permanent modifications from memory back to the disk file store. This, if accompanied by consideration of shared objects, can produce a consistent stable state for the process. Whenever a user closes his files, he ensures that the data in the files is stable. In the case of a server failure which results in file corruption, the user may revert to the most recent backup taken by the system operator. After any failure users restart their processes, and with the exception of higher level software such as data base management systems the integrity of data at the object level is not critical to correct store operation.

In persistent systems, however, transparent transfer of data between volatile and non-volatile storage and also stability of the store is the responsibility of the system memory manager. Users in persistent systems view the store as stable and therefore expect all their update operations to remain durable. Such a view is achievable

[1] Such reduction must be sensible in terms of cost. Associations may be reduced to unit size by checkpointing after every update, however this would be as detrimental to system performance as allowing associations to grow too large.

to some extent through frequent checkpoints, as proposed for the Napier and Monads-PC persistent stores [5, 18].

Napier is a single-user persistent system constructed above a Unix-based computer. It involves an abstract machine (PAM) which provides the abstraction of the persistent store and also a programming language (Napier-88). Stability of the persistent store is provided through regular checkpointing and the implementation of shadow paging between subsequent checkpoints [4]. In terms of dependency, all entities in the store are assumed to become dependent on each other during operation, and therefore the whole store is stabilised atomically. In the case of any failure the whole store is rolled back to the last checkpoint state [4].

The Monads-PC is a purpose-built computer which supports persistence at the architectural level. The non-volatile backup for the persistent store consists of one or more volumes (disks), which form the granularity of stability [10]. Volumes are stabilised or rolled back without regard to the entities stored on them. Similarly to Napier, in uni-volume Monads-PC computers all entities are assumed to be dependent on each other and therefore no dependency information is maintained. However, in the case of multi-volume Monads computers any reference from an entity in one volume to an entity resident on another volume may result in dependency between the volumes. Thus, to date, dependencies in the Monads system have been maintained at the volume level.

2.2 Implications of dependencies in distributed persistent stores

A considerable body of research [3, 12, 14, 15, 19] has been carried out aimed at building recoverable distributed systems because of their higher probability of failure occurrence. The failure of a node or of the communication link (these failures result in network partitions) in distributed systems causes only a part of the system to be unavailable. To allow still-alive portions of the store to continue operation correctly after such a failure, their state should be made consistent with the recovery state of the failed (and temporarily inaccessible) portion.

Constructing a global consistent state in distributed systems is not as straightforward as for single-node systems. This is because of the logical dependencies described above, and the fact that such dependencies may traverse nodes. The difficulty is achieving atomicity of checkpoint for multiple nodes and in a situation where failure of a node or the inter-connecting medium can occur at any time. In the following paragraphs we review the implications of dependency for Casper and Monads-DSM, these being examples of stores implementing different distributed store control disciplines.

Casper employs the centralised server model to provide the abstraction of a DPS. It considers the world as a set of clients served by a central server which provides access to shared objects and maintains the stability and coherency of the paged persistent store. Checkpoint operations occur at the client level and may be cascaded to other clients. Clients which have seen the same dirty data are deemed to be dependent on each other and are grouped into dynamic sets called associations [20]. Each association is accompanied by a set of pages which have been modified by at least one member of the association since the last checkpoint. A page may belong to at most one such set. Whenever a client accesses a modified page, the association to which the client belongs is merged with the association defining other clients dependent on the page. If a client modifies a clean page, the page is added to the set of pages accompanying the association to which the client belongs.

When any client belonging to an association initiates a stabilise operation, all clients in the association are forced to stabilise. Similarly, if any client in an association rolls back to its last stable state, all clients in the association must roll back. These requirements result in consistency of the persistent store.

The Monads-DSM provides a DPS using a distributed server model constructed over a network of Monads-PC computers. As described in section 2.1, the granularity of stability in Monads-PC computers is the volume. In a multi-volume Monads-PC or in the Monads-DSM, it is possible to have cross references between volumes. In order to ensure consistency, volumes containing cross references must be stabilised together. A dependency graph maintained at each node is used to describe dependencies between volumes. A two-phase commit protocol is used to perform a stabilise operation in which a volume and all its dependent volumes (according to the dependency graph) are stabilised together [10].

The problems with the Monads-DSM approach are not only the large granularity of stability (the volume which may contain many objects) but also the determination of dependency relationships between volumes regardless of the kind of access (read or write). Using volumes as the granularity of stability leads to the incidence of stabilise operations on non-essential data objects in a volume together with other objects which must be stabilised for consistency reasons. As we shall show, the lack of consideration of access type in defining the dependency of volumes results in larger dependency graphs than necessary. This larger than necessary dependency graph makes the stabilise operation in Monads-DSM inefficient. The issue of maintaining dependencies regardless of access type is also applicable in the case of Casper.

3 More efficient representation of dependencies

As mentioned above, the current methods of managing dependencies in DPSs consider dependency as a bilateral (undirected) relationship, and therefore reading modified data has the same effect as modifying data in terms of dependency. We show that different dependencies are created by modifying data and by accessing previously modified data, and that read operations create unilateral (as opposed to bilateral) dependencies. Because read operations typically outnumber write operations [7], we believe that the cost of cascaded operations will decrease dramatically by use of the proposed unilateral dependencies.

Reading from objects and writing to them are the two major operations which result in dependency. Other non-frequent operations such as create, open, close, and remove objects are normally reduced to write operations on the red-tape areas of objects or the system-related data. It is assumed that objects are paged, and that the unit of data transfer between main memory and secondary storage is the virtual page. We describe the management of dependency in terms of process' access to virtual pages comprising objects.

Without the ability to separately stabilise and roll-back entities (such as processes and objects) rather than collections of entities (such as volumes), the incidence of dependency between such entities may result in cascading of these operations to all entities. Therefore, we reduce the granularity of stability to the individual entity.

A process may perform one of the following operations on a virtual page.

1) It may read data from an unmodified page, which results in no dependency between the reading process and the object containing the page. This is because the data is stable at the time of the read operation, resulting in no change in the stability of the reading process.
2) It may read a modified page of an object, which results in a unilateral dependency between the reading process and the object containing the page (ie the object is not dependent on the process). This is because the data was unstable at the time of the read operation, and the unstable data caused the process to become unstable (if it was not already).
3) It may modify a page of an object, which results in a mutual dependency of the modifying process and the modified object. This is because the process caused the (possibly stable) data to become unstable.

Directed graphs may be used to represent directed dependencies between entities. We refer to such graphs as Directed Dependency Graphs (DDGs). Entities form vertices and dependencies between them form edges of DDGs. Each edge either describes a dependency between a process and an object or vice versa. Objects (processes) may depend on each other only through a vertex representing a process (object). We use $\rightarrow$ in order to specify the causal dependency between two entities. By $E_1 \rightarrow E_2$, we mean that E_1 depends (directly or indirectly) on E_2. Such relation ($\rightarrow$) is transitive, but not symmetric ie if E_1 depends on E_2 ($E_1 \rightarrow E_2$) then E_2 does not necessarily depend on E_1. However, the right hand side of a $\rightarrow$ relation may depend on the left hand side for one of the following reasons:

- through transitivity (a cycle in the directed dependency graph), or
- when the left hand side is a process which writes to an object.

In the case of a write operation which leads to a pair of dependencies (with opposite directions), instead of using two unilateral edges ($E_1 \rightarrow E_2$ and $E_2 \rightarrow E_1$), we use the notation $E_1 \leftrightarrow E_2$.

DDGs are constructed gradually as processes access objects. Initially the dependency graph for an entity contains the entity itself as the root and the only vertex. In terms of the possible operations performed by a process on an object, the graph grows according to the following criteria.

- When a process P_1 reads a modified page of an object O_1, the $P_1 \rightarrow O_1$ edge is inserted into the graph.
- When a process P_1 modifies a page of the object O_1, either the $P_1 \leftrightarrow O_1$ or $O_1 \leftrightarrow P_1$ edge is inserted into the graph.
- When a process in one graph reads a modified page or modifies a page of an object in another graph, the two graphs are merged to create a single larger graph.
- A graph shrinks when a collection of dependent entities is stabilised or revert to their last stable state.

At any given time each entity belongs to one and only one dependency graph. To find the entities dependent on an entity, it is sufficient to find the location of the entity in its graph and then traverse the directed graph from the entity. Dependencies between local and remote entities are also managed by DDGs. Such dependencies result in distributed DDGs encompassing entities resident on multiple nodes.

4 Implication of DDGs on checkpoint and roll-back operations

Dependency graphs are used to identify entities to which a stabilise or roll-back operation should cascade. However, the path traversed for cascading roll-back may be different from that for cascading checkpoint operations.

We use a single dependency graph and then apply separate stabilise and roll-back algorithms to achieve the required operation, two more symbols are introduced which specify the dependency between two entities in terms of stabilisation or roll-back. By $E_1 \xrightarrow{S} E_2$, we mean that when E_1 stabilises, E_2 also should be stabilised and therefore E_1 depends on E_2 in terms of stability. Likewise, $E_1 \xrightarrow{R} E_2$ means that E_1 depends on E_2 in terms of roll-back. Figure 1 shows the relationship between the edges forming a dependency graph and their meanings in checkpoint and roll-back graphs. By convention, we are able to use a single dependency graph to indicate separate stability and roll-back relationships.

Dependency Graph	Stabilising Graph	Roll-back Graph
$\rightarrow$	$\xrightarrow{S}$	$\xleftarrow{R}$
$\leftarrow$	$\xleftarrow{S}$	$\xrightarrow{R}$
$\leftrightarrow$	$\xleftarrow{S}$ and $\xrightarrow{S}$	$\xrightarrow{R}$ and $\xleftarrow{R}$

Figure 1 The relationship between Dependency Graph, Stabilising Graph, and Roll-back Graph.

To clarify the distinction between the implication of directed dependency on stabilise and roll-back operations, consider a scenario in which a process P_1 reads a modified page of the object O_1. Until commencement of the next stabilise operation for process P_1, all subsequent actions taken by P_1 are unstable because they may depend on the modified O_1 data. During this period of instability, either of O_1 or P_1 may be alternately stabilised or rolled back.

1) If O_1 is checkpointed, P_1 is not required to be checkpointed. The worst case is that after O_1 is checkpointed, P_1 rolls back to its last stable state. This results in no inconsistency and P_1 may read the stable data from O_1 again without problems.
2) If P_1 is checkpointed, O_1 must also be checkpointed. Otherwise, in the case of the roll-back of O_1 there is an inconsistent state in which orphan (no longer existent) data has been read by P_1 and which may have affected its subsequent behaviour.
3) If O_1 rolls back, P_1 has to roll-back because the roll-back results in P_1 having read orphan data.
4) If P_1 rolls back, O_1 does not have to roll-back because P_1, after reversion, can simply redo the read operation.

4.1 Stability and roll-back of dependent entities

Checkpointing (rolling-back) an entity requires the atomic checkpoint (roll-back) of the entity itself, all of its dependents, and the system-related data structures. The effect of using DDGs on the propagation of checkpoint and roll-back is demonstrated in figure 2. Using the method of associations and dependency graphs which have been used in Casper and Monads-DSM, all entities would be checkpointed (rolled back) as a result of the checkpoint (roll-back) of any entity in the graph. Figure 2(a) shows a DDG formed according to the cross-access of three processes and four objects according to the rules described in section 3. Shaded vertices depict entities with unstable states and non-shaded vertices depict entities with stable states. Figure 2(b) shows the effect of stabilising P_1; this operation propagates only to O_1 and O_2. Figure 2(c) shows the effect of rolling back P_3 which only propagates to O_4.

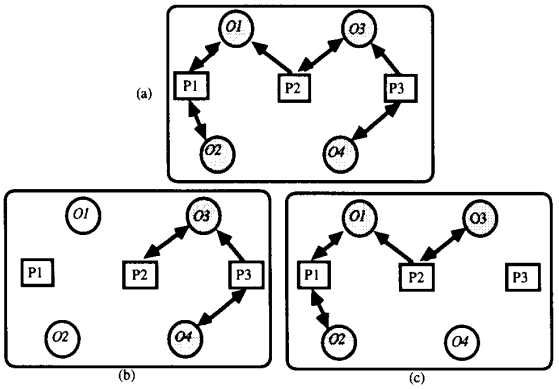

Figure 2 a) A sample directed dependency graph. b) The effect of checkpointing P_1. c) The effect of the roll-back of P_3.

5 Achievement of stability using DDGs

The application of DDGs to provide a stable DPS requires the following facilities:

1) The system kernel must support the detection of operations which create inter-entity dependencies.
2) Dependency graphs must be independent in terms of checkpoint and roll-back operations. Checkpoint (roll-back) of an entity belonging to a DDG must not lead to checkpoint (roll-back) in any other DDG.
3) Checkpoint operations must be atomic, and must ensure the existence of at least one global consistent stable state. For a distributed store, this can be achieved through the application of a two-phase protocol.

5.1 Kernel support for dependency detection

As implied in section 3, processes cannot become directly dependent on other processes. Objects also cannot be directly dependent on other objects.

To support the construction of DDGs the operating system kernel for each node must [11] maintain information describing:

(1) Which virtual pages (resident in main memory or not) have been modified since the objects containing the pages were checkpointed.
(2) Which modified virtual pages have been accessed in this time-slice by the currently executing process.
(3) Which virtual pages have been modified in this time-slice by the currently executing process.

Detection of whether a non-resident page is modified may be achieved by recording which virtual pages have been modified since the last checkpoint. When handling a page fault, the kernel checks if the page (V_1) has been previously modified, and if so it is mapped into main memory as modified/read-only and a dependency is recorded. Otherwise, it is mapped in as read-only. If the executing process attempts to write on the page an exception condition is created; the kernel grants write access after modifying the corresponding DDG.

In a multi-tasking system a normal process-switch may subsequently result in the activation of another process P_2, which may then access V_1. This access must be detected by the stability mechanism because it may cause modification of the dependency graph information. It is not sufficient to rely on access fault management to detect the access, however, because the page is currently in main memory in writeable form; in short the access would not cause an access fault. To detect such accesses and thus permit dependency information to be updated, the kernel (or the architecture) must support identification of modified memory pages inherited by the next process. TLB-based architectures support this facility at the hardware level [20]. Architectures such as Monads [17], however, which do not provide this facility, must emulate it at the kernel level to support the DDG scheme.

5.2 Critical entities in DDGs

Each normal entity belongs to one and only one DDG. Thus DDGs are autonomous in terms of stabilisation ie checkpointing the entities belonging to a DDG does not result in checkpointing of entities belonging to another DDG. However, there are special system entities which logically belong to more than one DDG. We discuss such entities in this section.

The kernel in each node can be considered as an object including data structures used in the node management. Each kernel function typically results in modifications to these data structures, making the current user process dependent on the kernel and vice versa. This recursively results in dependency between all entities and the kernels. For simplicity we will refer to each node kernel as a single entity which belongs to all DDGs on that node.

Due to the lack of a one-to-one correspondence between virtual address space and physical disks address space[2], object identifiers must be mapped into their physical addresses. We assume that a mapping table exists per disk to perform such mapping for all objects on the disk. We refer to this mapping table as the *disk directory* and allocate a special object with well-known identifier per disk (root object) which contains all the information required for management of the disk (including the disk directory and disk free-list). Each access to objects located on a disk requires access to and possibly modification of the root object. For example, any disk page allocation requires the modification of the disk free-list.

If shadow paging is the method of stability, each attempt to modify a 'clean' page results in dependency between the object containing the page and the root object. Thus each modified object necessarily depends on the root object of its storage disk and vice versa. Therefore, each network-wide DDG may contain one or more kernel entities and also one or more disk root objects.

Disk root objects and kernel entities are critical entities; their inclusion in DDGs can act as a focus to cause cascade of stabilise operations through whole the distributed store if they are considered similarly to the other normal entities. Accordingly we do not consider critical entities as normal entities and do not include them in DDGs. They are considered as permanent entities of each dependency graph and are restricted in terms of the propagation of operations through them; the dependencies between them and DDGs are referred to as *implicit* dependencies. In fact, critical entities act as obstacles to the propagation of checkpoint and roll-back operations.

Figure 3(a) shows the system state in a node with three dependency graphs spread over two disks D_1 and D_2. All dependency graphs on the node depend on the kernel entity and all dependency graphs with entities belonging to a disk depend on the disk root object. We refer to such dependencies as *strong dependencies*. DDGs on a node strongly depend on the node critical entities. Figure 3(b) shows individual views of each dependency graph in terms of critical entities. The root object is assumed to have a virtual instance per dependency graph, helping us to improve our stabilisation protocol. We must however, somehow stabilise the root object to guarantee the

[2]Note that even in computer systems which provide single-level stores (virtual stores as long as their disk address space), mapping is typically required. This is due to the gradual allocation of disk blocks to virtual pages.

possibility of roll-back of other existing non-stable dependency graphs.

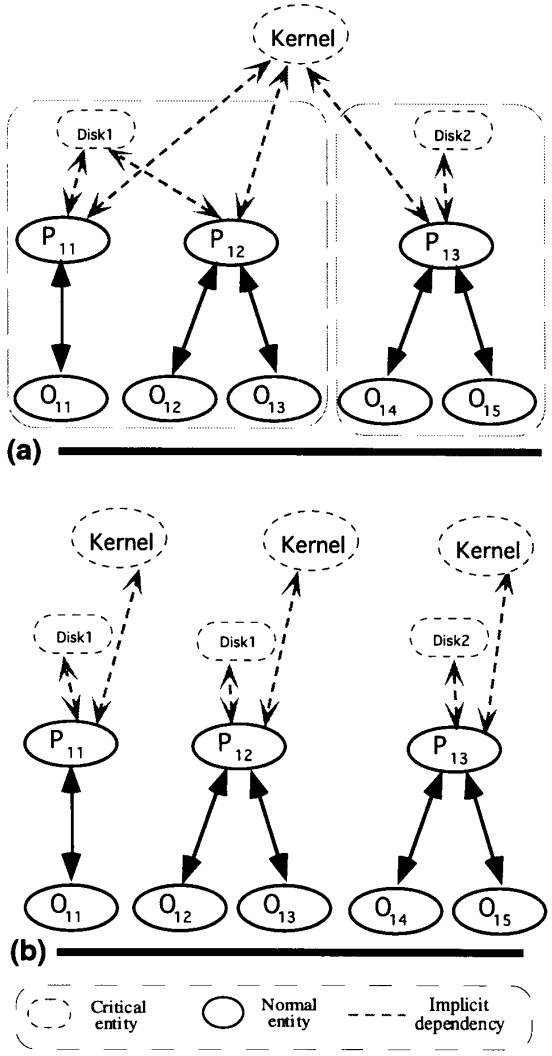

Figure 3 (a) Dependency of DDGs on critical entities. (b) Critical entities act as obstacles in operation propagation.

5.3 Achievement of atomic checkpoint operation

Checkpoint operations advance the stable system state and release disk blocks occupied by the previous stable state. For dependent entities distributed over multiple nodes it is crucial, for the sake of consistency, that the effects of a checkpoint operation appear to be atomic. This means that either all dependent entities are checkpointed resulting in a new stable state, or none of them are checkpointed. Otherwise, for instance, the occurrence of a disk crash while a checkpoint is in progress may result in an inconsistent system state.

To provide atomicity, checkpoint of a DDG is achieved through a two phase protocol. The most important issue in such a protocol is the possibility of reversion to the most recent stable state for each participant which has failed to achieve its checkpoint or become inaccessible due to communication failure. As described in the next section, it is important that cycles in a dependency graph do not cause dead-locks during checkpoint operations. A tagging pre-phase is used to avoid such dead-locks, followed by a two phase checkpoint. The protocol guarantees atomicity of operations with the possibility of roll-back operation as soon as a participating node detects the failure (inaccessibility) of another node involved in the operation.

Cyclic dependency graphs: As mentioned above, the root object in each disk depends on all modified objects resident on the disk and vice versa. Checkpoint of each modified object necessitates the stability of its corresponding root object. Furthermore checkpointing an object requires modification to the root object. If the root object was checkpointed when further dependent entities belonging to the same disk existed in the current dependency graph, the root object would have to be checkpointed more than once during the single atomic operation. Multiple checkpoints of the root object in a single atomic operation make it impossible to revert to the most recent state if failures occur during the checkpoint. This is because of the disappearance of the stable states older than the last one during the achievement of the disk root object checkpoint. To prevent such inconsistencies and also to improve the efficiency of checkpoint operation, all entities on a disk belonging to the same dependency graph form a group. Members of such a group are checkpointed together followed by a single checkpoint of their corresponding disk root object. The need for a tagging pre-phase is demonstrated by the following example which assumes no tagging pre-phase, and nodes each supporting a single disk.

Consider the checkpoint of O_{11} in a dependency graph distributed over nodes N_1, N_2, and N_3 in figure 4. Starting from O_{11} and checkpointing entities during traversal of the graph, N_2 becomes aware that O_{21} belongs to the same graph as O_{11} when it is informed to checkpoint its part of the graph. N_2 checkpoints O_{21} as well as its corresponding disk root object (note that according to the current knowledge of N_2, no other entity residing on the same disk as O_{21} belongs to the same dependency graph), and sends a message to N_3 requesting the checkpoint of P_{31}. While N_2 waits for the completion

of the checkpoint in N_3, it will receive a request from N_3 requesting the checkpoint of O_{22}. As discussed above, N_2 can not checkpoint O_{22} while the checkpoint of O_{21} has not completed or aborted. N_3 cannot respond to the initial request from N_2 until it receives a response to its request of N_2. This scenario results in either roll-back (due to the non-response between N_2 and N_3) or in a deadlock situation.

The tagging pre-phase prevents such a deadlock situation. During the pre-phase, the DDG is traversed and an identical tag is attached to all entities belonging to the same DDG. This allows the checkpoint algorithm to properly group all entities belonging to the same disk.

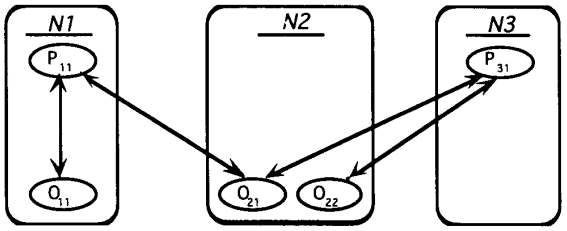

Figure 4 Entities on the same disk which indirectly depend on each other.

Two Phase Checkpoint Using DDGs: In order to provide atomicity of a network-wide checkpoint, it is achieved in two phases, *flush* and *commit*. The flush phase is initiated after completion of the tag pre-phase in which the entire DDG is traversed and the same Dependency Graph Number (DGN)[3] is associated with each entity. All page flushes and root object modifications are achieved in this phase but remain non-completed (by advancing to the new state of the disk root objects) until commencement of the commit phase. The first phase is started by the checkpoint initiator node with results as described below.

First, local entities with the same DGN are classified in groups per disk according to the pre-phase tags. Then entity flushing is achieved for all such groups. For each group, this includes flushing of all group members followed by flushing of the disk root object. Flushing of an individual entity consists of writing all its modified pages currently held in main memory back to disk and modifying the disk root object such that the object appears clean. Flushing of the disk root object must also include declaration of itself as clean. During the period after flushing the root object and before the commitment of the checkpoint, the root object has an uncertain state and any access which results in the modification of the root object is prohibited.

In parallel a 'flush' message is sent to each node which stores entities belonging to the current DDG requesting them to perform the flush phase for all entities with the same DGN (this occurs recursively to encompass all remote entities stored on remote nodes). The initiator (coordinator) then waits for participants' replies.

On receipt of the 'flush' message, each participant achieves the flushing phase in the same way as the coordinator. The node may propagate the operation to other nodes. This may result in the receipt of multiple checkpoint requests for the same DDG by a node, due to the possible existence of cycles in the graph. In such cases only the first message is processed and later messages are acknowledged. If a participant propagates its received checkpoint request to other nodes, it must wait for their responses before itself appropriately replying to the checkpoint request.

Upon receipt of a completion message, each node checks if all successors have replied and if so, it either

- communicates the completion of its 'flush' to its predecessor if it is not the coordinator, or
- initiates the commit phase if it is the coordinator.

If a node detects the inaccessibility of its successor in the checkpoint propagation graph (for instance through time-out or guardian techniques), it autonomously initiates an abort of the checkpoint, commencing the second phase. This is discussed in more detail in section 6.

In the case of completion of the flush phase, the coordinator decides to commit and propagates this decision through the graph. The commit phase is also achieved per disk, and achieves atomic transfer from the previous stable state to the current flushed state. This is achieved by atomically updating the pointers to each disk root object so that these pointers now reference the new root objects created by the checkpoint. It is crucial to somehow guarantee the existence of at least one stable state despite the possible occurrence of failures during the transition from the previous stable state to the next one.

The operation which completes a disk checkpoint is similar to the commit operation for database updates. We use Challis' algorithm [6] to achieve the atomic update of the pointer to each disk root object. Two pointers are allocated to each disk to represent two subsequent stable states of the system. If a failure (crash) occurs while a pointer is being stored as the final stage of a checkpoint operation then the pointer is potentially incorrect. After recovery from the crash, the system is able to identify the consistent pointer and therefore the most recent stable state of the disk.

We add to Challis' algorithm in order to fulfil the requirements of two phase checkpoint atomicity. We

[3] We also refer to DGN as checkpoint-id.

assume that the store is spread over a multi-node system and each node is capable of mounting more than one disk. The new algorithm detects:

- the most up-to-date correct root object pointer for each disk,
- failure of a node after achieving the first phase of a checkpoint operation for one of its local disks, and
- failure of a node after achieving the second phase of a checkpoint operation for one of its local disks and before all local disks containing entities in the dependency graph have completed their second phase.

A message protocol allows a recovering node to decide which disk root object pointer to use if an incomplete checkpoint is detected during failure recovery. According to this protocol the node either completes the second phase of the incomplete checkpoint or performs a roll-back operation.

6 Implication of using DDGs on checkpoint abort

As soon as a node detects inaccessibility of its successor in the checkpoint propagation graph, it assumes the receipt of an abort or roll-back request through its edge to the inaccessible node. Whereas systems using associations to describe dependent entities require such an abort to be propagated to all entities in the association, systems using DDGs allow some flushed entities to commit their checkpoint while other entities revert to their previous stable states. As shown in the following example, the use of DDGs thus results in less loss of data modifications after failure.

Consider the scenario depicted in figure 5. The process P_{11} on node N_1 initiated a checkpoint which propagated to O_{11}, and hence to N_2 and through N_2, to N_3 and N_4. Suppose that N_2 detects inaccessibility of N_3 and therefore acts as if it has received an abort request through the edge between P_{21} and O_{31}. Although node N_4 remains accessible, O_{41} and P_{41} are unaffected by the abort because of the nature of the directed edge linking P_{21} and O_{41}. The entities resident on node N_4 can thus continue to commit their checkpoint. Because of the nature of the edges linking the involved entities resident on nodes N_1 and N_2, however, the roll-back operation cascades to those entities.

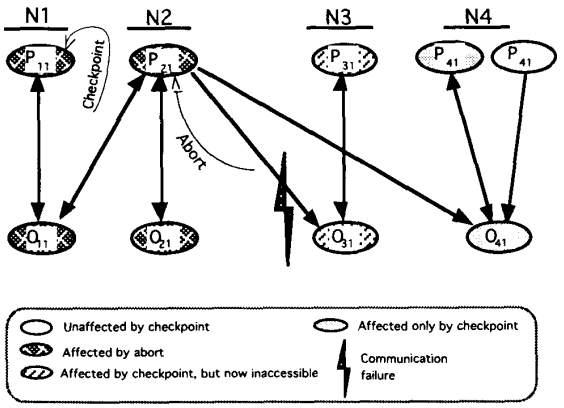

Figure 5 Implication of DDGs on checkpoint abortion

Conclusion

In this paper we demonstrated shortcomings of the current techniques used to describe dependency relationships between processes and objects in a distributed persistent store. In particular we showed that the cascade effect of stabilise operations resulted in larger stabilise operations than absolutely necessary. We also showed that the cascade effect of roll-back operations could result in unnecessary loss of store modification.

We presented an alternate method for describing entity inter-relationships. This alternate representation uses directed graphs. The directed edges in such graphs allows different interpretation of the graph during stabilise and roll-back operations. We showed that, because of the extra flexibility afforded by the DDG representation, it is possible to significantly reduce the cascade effects of store stability operations. In particular, checkpoint operations are improved in terms of efficiency, and the extent of roll-back operations is reduced. This is a significant achievement, because only those modifications which it is absolutely necessary to reverse are lost as a result of roll-back operations.

The techniques described in this paper have been evaluated by simulation and shown to result in significant stability-related performance improvements. Subsequently a new version of the Monads architecture which incorporates hardware support for the construction of directed dependency graphs has been designed and is currently being implemented [9].

Acknowledgment

This research has been supported by the Ministry of Culture and Higher Educations, the Government of the I. R. of Iran.

References

[1] Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26(4), pp. 360-365, 1983.

[2] Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "CMS - A Chunk Management System", *Software Practice and Experience*, 13(3), pp. 259-272, 1983.

[3] Bhargava, B. and Lian, S. "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems- An Optimistic Approach", *Proceedings of the 7th Symposium on Reliable Distributed Systems*, Columbus, OH, pp. 3-12, 1988.

[4] Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.

[5] Brown, A. L., Dearle, A., Morrison, R., Munro, D. and Rosenberg, J. "A Layered Persistent Architecture for Napier88", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 155-172, 1990.

[6] Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Useability and Responsiveness*, Academic Press, pp. 245-270, 1978.

[7] Cvetanovic, Z. and Bhandarkar, D. "Characterization of Alpha AXP Performance Using TP and Spec Workloads", *IEEE Computer Architecture News*, 22(2), pp. 60-70, 1994.

[8] Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", Basser Department of Computer Science, The University of Sydney, N.S.W., Australia, Technical Report 462, ISBN 0 86758 668 0, 1991.

[9] Henskens, F. A., Koch, D. M., Jalili, R. and Rosenberg, J. "Hardware Support for Stability in a Persistent Architecture", *Proceedings of the 6th International Workshop on Persistent Object Stores*, to appear, France, 1994.

[10] Henskens, F. A., Rosenberg, J. and Hannaford, M. R. "Stability in a Network of MONADS-PC Computers", *Proceedings of the International Workshop on Computer Architectures to support Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 246-256, 1990.

[11] Jalili, R., Henskens, F. A., Koch, D. and Rosenberg, J. "Operating System Support for Object Dependencies in Persistent Object Stores", *Proceedings of the Workshop on Object-oriented Real-time Dependable Systems (WORDS'94)*, to appear, IEEE Computer Society Press, California, 1994.

[12] Johnson, D. B. and Zwaenepoel, W. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", *Proceedings of the 7th Symposium on Principles of Distributed Computing*, ACM, pp. 171-181, 1988.

[13] Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings of the 4th International Workshop on Persistent Object Systems*, pp. 99-109, 1990.

[14] Lin, L. and Ahamad, M. "Checkpointing and Rollback-Recovery in Distributed Object Based Systems", *Proceedings of the 20th International Symposium on fault Tolerant Computing*, IEEE Computer Society, pp. 97-104, 1990.

[15] Lowry, A., Russell, J. R. and Goldberg, A. P. "Optimistic Failure Recovery for Very Large Networks", *Proceedings of the 10th Symposium on Reliability in Distributed Software and Database Systems*, IEEE, pp. 66-75, 1991.

[16] Nitzberg, B. and Lo, V. "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, 24(8), pp. 52-60, 1991.

[17] Rosenberg, J. "The MONADS Architecture - A Layered View", *Proceedings of the 4th International Workshop on Persistent Object Systems*, Morgan-Kaufmann, pp. 215-225, 1990.

[18] Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for Security and Persistence of Information*, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

[19] Storm, R. E. and Yemini, S. A. "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems*, 3(3), pp. 204-226, 1985.

[20] Vaughan, F., Basso, T. L., Dearle, A., Marlin, C. and Barter, C. "Casper: a Cached Architecture Supporting Persistence", *Computing Systems*, 5(3), pp. 337-359, 1992.