# Instant Snapshots in a Federated Array of Bricks

Minwen Ji
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-15
January 28, 2005*

snapshots, checkpoints, storage, distributed systems, clusters

Snapshot has become a fundamental requirement on mid- to high- end storage systems. Its applications include archiving, recovery, report generation, decision making tools and remote mirroring. State-of-the-art snapshot techniques on existing storage systems typically work on a single (fault-tolerant) controller, and need to pause the applications or change the operation mode of the file systems or databases when a snapshot is taken. In a federated array of bricks (FAB), a snapshot may involve tens to thousands of independent controllers or processors, and may be taken at a high frequency, e.g., once every 30 seconds for atomic updates in remote mirroring. Therefore, an efficient distributed snapshot algorithm that can make the snapshot operations transparent to applications is needed in FAB.

In this paper, we propose such an algorithm, which avoids pausing or aborting write requests by the novel use of a tentative data structure during the two phase commit of a snapshot creation. The snapshot operations are serializable with data operations (i.e., reads and writes), hence ensure consistency of the snapshots. Read-only operations on snapshots are optimized in common cases, only requiring communications to a small subset of the bricks, in particular, a single replica set or three bricks in FAB. The algorithm has been prototyped in FAB and has been tested with trace-based experiments.

* Internal Accession Date Only Approved for External Publication
© Copyright Hewlett-Packard Company 2005

# Instant Snapshots in a Federated Array of Bricks

Minwen Ji

**Abstract**

Snapshot has become a fundamental requirement on mid- to high- end storage systems. Its applications include archiving, recovery, report generation, decision making tools and remote mirroring. State-of-the-art snapshot techniques on existing storage systems typically work on a single (fault-tolerant) controller, and need to pause the applications or change the operation mode of the file systems or databases when a snapshot is taken. In a federated array of bricks (FAB), a snapshot may involve tens to thousands of independent controllers or processors, and may be taken at a high frequency, e.g., once every 30 seconds for atomic updates in remote mirroring. Therefore, an efficient distributed snapshot algorithm that can make the snapshot operations transparent to applications is needed in FAB.

In this paper, we propose such an algorithm, which avoids pausing or aborting write requests by the novel use of a tentative data structure during the two phase commit of a snapshot creation. The snapshot operations are serializable with data operations (i.e., reads and writes), hence ensure consistency of the snapshots. Read-only operations on snapshots are optimized in common cases, only requiring communications to a small subset of the bricks, in particular, a single replica set or three bricks in FAB. The algorithm has been prototyped in FAB and has been tested with trace-based experiments.

## 1 Introduction

A snapshot of data is a record that reflects the state of the data at a particular point in time. Snapshots can be used for a variety purposes, including data archiving, recovery after a hardware failure or software error, remote mirroring, report generation and decision making. As a particular example, a snapshot taken prior to the occurrence of data corruption resulting from a software error may be used to return the data to an uncorrupted state.

A *consistent* snapshot at time $t$ records the results of all updates to the data before time $t$ and does not record any results of the updates after time $t$. For example, for replicated data, a snapshot is consistent when the recorded replicas of the data are identical; the snapshot is inconsistent if an update has occurred to a replica of the data and not to another replica, so that the recorded replicas are no longer identical. For distributed data, a snapshot is consistent when it reflects the state of the data across all locations at the same point in time; a snapshot is inconsistent if it records a first update after time $t$ to data in a first location but does not record a second update before time $t$ to data in a second location. It is important to ensure that a snapshot is consistent. Otherwise, the snapshot may not be useful for its intended purpose. Typically, this means that no updates to the data can occur while the snapshot is being taken. Due to communication delays and clock skews, taking a consistent snapshot in a distributed system typically requires the suspension or alteration of other activities and takes a considerable amount of time. Thus, taking snapshots in a distributed system like FAB can significantly interfere with the performance of the system, particularly if frequent snapshots are desired.

Therefore, improved techniques for taking snapshots in large distributed systems are needed.

## 2 Background: data structures in FAB

Figure 1 shows the structure for snapshot volume and maps.

A volume is a collection of segment groups, which is typically created by a human operator or by an application. A segment is a fixed number of data blocks and is the unit of physical to logical address mapping. A segment group is the unit of redundant storage. The set of bricks that redundantly store a segment group is called a replica set.

There exist one or more versions of a volume. A *version* of a volume is a copy of the volume that reflects the updates to the volume up to a certain point in time. The *current* version of the volume reflects all the updates in the past and changes as new updates arrive. A *snapshot* version of the volume reflects the updates up to the point in time when the corresponding snapshot operation is started. Each version has its own copy of data structures, such as names, maps, etc. The name of the current version is also referred to as the volume name, and the name of a snapshot version is referred to as a snapshot name. Read and write operations are allowed on all versions. Snapshot operations are allowed on the current version.

There is a map for each snapshot or current version. The maps of the versions of the same volume are organized into a bi-directional linked list. The next map pointer (nextMap) of a map points to the map of the next newer version. The previous map pointer (prevMap) points to the map of the next older version. The physical map pointer (phyMap) points to the data structure that actually maps logical address to physical. In addition, the map of a snapshot has an optional private map pointer (privMap) that points to a physical map that contains locations of data that belongs to this snapshot but was written after the snapshot was created. The first map in the linked list (with a null prevMap pointer) is called the *initial* map, and the last (with a null nextMap pointer) is the *current* map.

Each map contains two timestamps: creTs is the time when the map was created and dataTs is the last time when any data in the map is updated or prepared for update.

When a new data block is written to the current volume, a new physical block is allocated for the data and the physical map is updated with the new logical to physical mapping. When an existing block is written in the current volume, the data is overwritten in place and the physical map is not updated. When a block is written in a snapshot volume for the first time after the snapshot is created, a new physical block is allocated for the new data and the private map is updated with the new mapping. When the block is written for a second time, the data is overwritten in the location recorded in the private map, and the private map is not updated.

To map a logical address (volName, logAddr) to a physical one (devName, phyAddr), one looks in the physical maps in the following order: private map → physical map → physical map of previous map → physical map of previous map of previous map → ... until a mapping is found.

# 3 Correctness criteria

In order for the snapshots to be consistent, the protocol for creating, accessing and deleting snapshots shall maintain the following invariants:

1. A majority of bricks must agree on whether or not a snapshot exists in the system.
2. A majority of bricks must agree on the timing order of a snapshot in the globally serializable sequence of snapshot/data operations.

# 4 Protocol design

## 4.1 Creating and deleting snapshots

The snapshot creation is operated in two phases. In the first phase, the coordinator sends a PrepareSnapshot command with a new timestamp (newSnapTs) to all bricks in the system. Each brick compares newSnapTs to the dataTs and creTs of its local current map. If newSnapTs is newer than the local timestamps, the brick creates a new version with the snapshot name and a new empty map with creTs=newSnapTs and dataTs=the current map's dataTs. The new version points to the current map and the current volume points to the new map. The new map's prevMap pointer points to the current map. The current map's SNAPPING flag bit is set and the new map's TENTATIVE flag bit is set. The brick will then return an OK status to the coordinator. On the other hand, if newSnapTs is not newer than the local timestamps, the brick immediately returns an NOK status to the coordinator. If the coordinator receives a *global quorum* (i.e., a quorum in every replica set) of OK replies, it proceeds to the second phase by sending a CommitSnapshot command to all bricks. Each brick then unsets the SNAPPING flag bit in its current map and unsets the TENTATIVE flag bit in its new map. The snapshot operation is completed thus far. On the other hand, if the

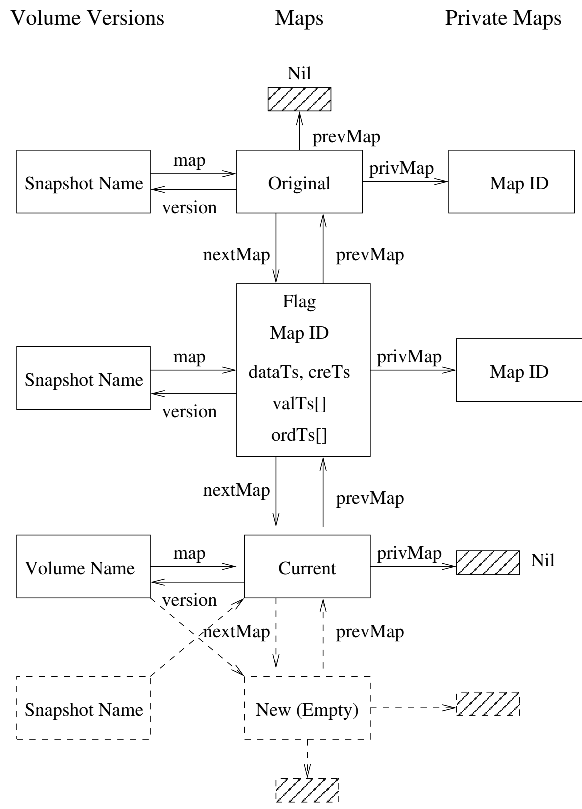


Figure 1: Maps for snapshots

coordinator does not receive a global quorum of OK replies, it sends an AbortSnapshot command to all bricks in the second phase. Each brick merges the new updates in its new map to its current map, deletes the new map and unsets the SNAPPING flag bit in its current map. Figure 2 illustrates the creation of a snapshot.

The snapshot deletion is also operated in two phases. In the first phase, the coordinator sends a PrepDelSnapshot command to all bricks in the system. Each brick checks the SNAPPING flag bit in its local map. If it is unset, the brick sets the DELETING flag bit in the map and returns an OK status to the coordinator. On the other hand, if the SNAPPING bit is set, the brick immediately returns an NOK status to the coordinator. If the coordinator receives a global quorum of OK replies, it proceeds to the second phase by sending a CommitDelSnapshot command to all bricks. Each brick merges the content of the map to be deleted (excluding its private map) into its next map and removes the map from the linked list. On the other hand, if the coordinator does not receive a global quorum of OK replies, it sends an AbortDelSnapshot command to all bricks in the second phase. Each brick then unsets the DELETING flag bit in its local map. Figure 3 illustrates the deletion of a snapshot.

#### 4.1.1 Optimization with future timestamps

In case the timestamp of a snapshot creation is older than some of the timestamps of data blocks, the snapshot operation needs to be aborted. The failure of timestamp comparison can be caused by clock skews and/or network delays. To avoid repeated abortion of the snapshot creation, the coordinator can retry the operation with a timestamp in the future. The future timestamp can be determined by adding to the current time the difference between the timestamp of the aborted snapshot creation and the newest timestamp of the data. Estimated network delay difference can be added to the future timestamp as well. The coordinator can record this timestamp adjustment for future use, and can update the adjustment based on the frequency of snapshot abortion.

### 4.2 Accessing snapshots

The basic question in reading snapshot information is: how do readers of the snapshot information (such as the coordinators of data operations) agree on whether the snapshot is valid or not? A harder question is: how does a reader know whether a snapshot is valid or not by contacting only a single replica set (i.e. by using a local quorum)?

To help answer the questions, let us recall the six states of a map in a brick: NULL (the brick does not have any local information about the map or the corresponding snapshot), TENTATIVE (the map is newly created and not committed), DELETING (the map is prepared for deletion and not actually deleted yet), SNAPPING (the map is being snapshot), SNAPSHOT (the map represents a successfully created snapshot, rather than the current version), and CURRENT (the map represents the current version). Figure 4 shows the typical state transitions of a map.

A snapshot is *valid* if and only if both of the following two conditions are true:

1. There exists at least one brick that has the snapshot in a non-tentative state, or there exists a global quorum of bricks that have the snapshot in any state other than null; and

2. There exists a global quorum of bricks that have the snapshot in in a non-deleting and non-null state.

The above two conditions mean that the snapshot has been successfully created or prepared for creation (i.e., in a global quorum of bricks), and has *not* been successfully deleted or prepared for deletion.

Reversing the two conditions above, a snapshot is *invalid* if and only if either of the following two conditions is true:

1. There does *not* exist any brick that has the snapshot in a non-tentative state, and there exists a global quorum of bricks that have the snapshot in the null state; or

2. There exists a global quorum of bricks that have the snapshot in deleting or null state.

A reader can try to determine whether a snapshot is valid or not by contacting all bricks in the system. If not enough bricks are alive at the time of reading, the reader will not be able to determine whether a global quorum does *not* exist, and will have to wait until enough bricks come back. Therefore, a validity check on a snapshot could return *valid*, *invalid* or *undetermined*. In addition, a snapshot that has been checked to be valid could become invalid after it is successfully deleted, or after a number of bricks that have a valid copy fail and are replaced with empty bricks.

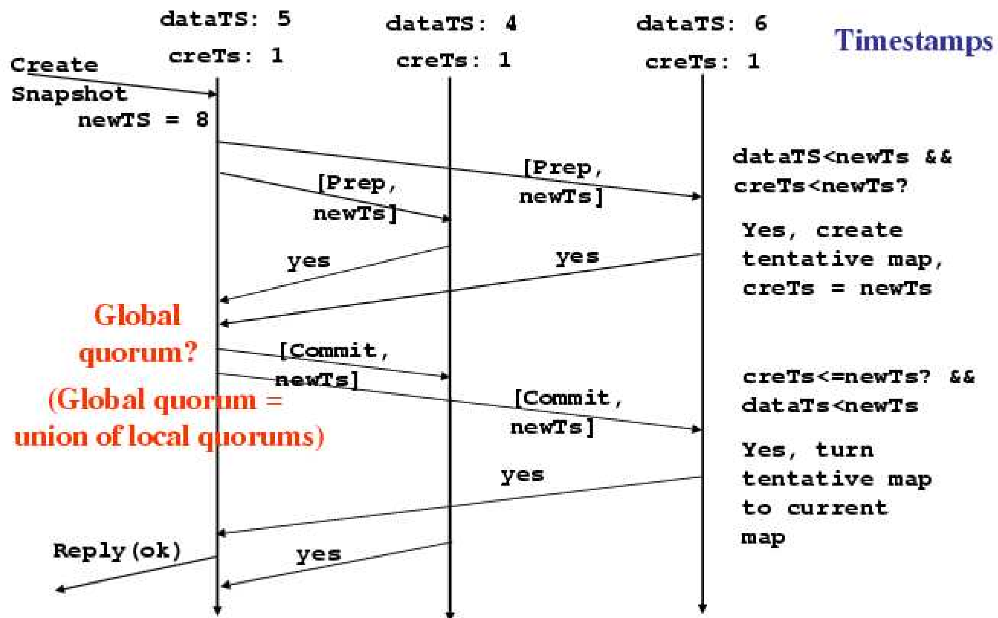


Figure 2: Snapshot creation

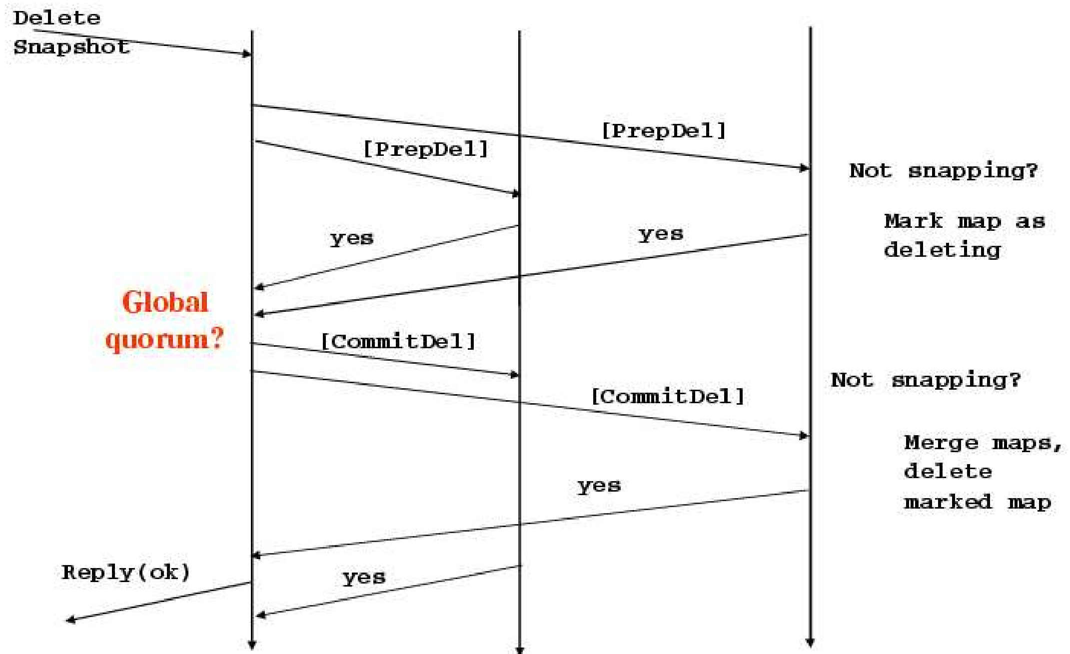


Figure 3: Snapshot deletion

### 4.2.1 Optimized validity check

It is desirable to have a reader contact only a small set of bricks (e.g., a single replica set) for validity check in common cases (e.g., read/write operations). Derived from the global conditions above, the reader can determine the validity of a snapshot using a local quorum in the following cases:

1. If there exists a local quorum of bricks that have the snapshot in the NULL state, then the snapshot is invalid.
2. If there exists at least one brick that has the snapshot in a non-tentative state, and there exists a local quorum of bricks that have the snapshot in non-null and non-deleting state, then the snapshot is valid.

If condition 1 above is true, then there does *not* exist a global quorum of bricks that have the snapshot in any state other than null. It follows that the first phase of the snapshot creation must have failed or may not even have been started. Therefore, the reader can determine that the snapshot is invalid if condition 1 is true. If condition 2 above is true, then the first phase of the snapshot creation must have succeeded (based on the existence of the non-tentative state) and the first phase of the snapshot deletion must have failed or may not even have been started (based on the lack of quorum of non-null and non-deleting state). Therefore, the reader can determine that the snapshot is valid if condition 2 is true.

To read information on a snapshot, the coordinator sends a ProbeSnapshot command to a single replica set, which may include itself. Each brick in the replica set replies with its local information on the snapshot, including the map Id and the flags. If it cannot determine the validity of the snapshot using the local replies, the coordinator will have to contact all accessible bricks and invoke a recovery procedure. In the recovery procedure, the coordinator first tries to determine the validity of the snapshot using the global replies; and if it succeeds, it then tries to enforce the result on all accessible bricks. Depending on the replies, the enforcement command could be CommitSnapshot, AbortSnapshot, CommitDelSnapshot, AbortDelSnapshot or some combination of them.

## 4.3 Reading and writing data

There is an implicit read operation on the snapshot information in every read/write/recovery operation on data. Since the coordinator will contact the replica set of the data for the data operation, the ProbeSnapshot command is piggy-backed on the data command, rather takes a seperate round-trip message.

In the first phase of a write/recovery operation, the newTs of the data is compared to the creTs timestamps of the maps in the volume at each brick in a replica set. The newest version of the volume whose creTs is older than newTs is selected as the destination for the write or recovery operation. If such a destination is found and the timestamps ValTs and OrdTs for the data in the destination map are older than NewTs, the device returns an OK response and the destination map id to the coordinator. If the coordinator receives a quorum of OK replies and sufficient information to determine the validity of the returned map locally, it will proceed to the second phase of the operation with the returned map id. The write will be committed in the map identified by the id. If the coordinator receives a quorum of OK replies, but not sufficient information to determine the validity of the returned map locally, then the coordinator invokes a recovery procedure on the snapshot before it retries the write/recovery operation on the data.

Assume that a snapshot on a volume is created at a time $t_1$ using a timestamp of $t_2$, where $t_1 < t_2$. For example, a write operation may occur at a time $t_3$, where $t_1 < t_3 < t_2$. Writes that occur during the interval between $t_1$ and $t_2$ may be directed to the snapshot. However, if a read operation is performed on the snapshot and then a write is performed on the snapshot, the read operation will be incorrect since the snapshot will have changed after the read. In other words, read operations should only be directed to a snapshot after all of the writes for the snapshot have been performed. Thus, read operations are disallowed unless the timestamp for the read operation is greater than the timestamp for the snapshot. Thus, in the example, a read operation having a timestamp of $t_4$ is disallowed unless $t_2 < t_4$. Once such a read occurs on a block of data in the snapshot, each device in the segment group for the block may update its timestamp ordTs and valTs for the data to be equal to $t_2$. Thus, any write operation attempted after such a read operation, but with a timestamp older than $t_2$, will be aborted rather than redirected to the snapshot (and any write operations with a timestamp newer than $t_2$ will be directed to the current version). Thus, no further writes to a block in a snapshot are allowed after the first successful read. This is done per data block rather than per snapshot because it lowers the possibility of aborting writes after the read and because there is no ordering requirement on writes to different blocks in FAB anyway.

# 5 Correctness

As stated in Section 3, the snapshot protocol needs to meet the two correctness criteria, validity and serializability, in order to maintain the consistency of snapshots. In Section 4.2, we have discussed the validity of snapshots. In this section, we focus on the serializability of the snapshot operations and data operations, i.e., a snapshot created at time $t$ includes the results of all writes before time $t$ and excludes any results of writes after time $t$.

Our reasoning about the serializability is based on the assumption that the basic protocol for data operations in FAB [22] maintains serializability of operations. That is, the value of a block after two writes, $write(v_1, t_1)$ and $write(v_2, t_2)$, is $v_2$ if and only if $t_2 > t_1$.

A snapshot creation generates a new version for each data block in the volume, where the old version belongs to the newly created snapshot and the new version belongs to the current version of the volume. It achieves this by creating a new version of the volume for data written after the snapshot creation and directing writes to the right version based on their timestamps. It maintains the invariant that every write with timestamp $t_1$ is directed to the newest version with a creation timestamp older than $t_1$.

We assert that a snapshot creation with timestamp $t$ has the same effect as a write operation in every block of the volume at time $t$ with the value of the latest completed write before time $t$. In particular, the first successful read since the snapshot creation on the snapshot version of the block has the same effect as a write on the snapshot version at time $t$. This operation gets its timestamp $t$ from the snapshot creation time recorded in during the snapshot preparation phase. It leaves the value of the block intact. That is, the value $v$ "written" in this operation is the same as that of the latest completed write before this operation. Since only writes with timestamps older than $t$ can be directed to the old version, and reads are not allowed on the old version until after time $t$, $v$ must be the value of the latest completed write before time $t$.

Therefore, the snapshot version of each block is the result of the latest completed write on the block before time $t$. This concludes our reasoning about the serializability of the snapshot operations and data operations.

# 6 An alternative

An alternative to the approach based on tentative data structures is to log the writes that overlap with a snapshot creation. The snapshot creation is still operated in two phases. In the first phase, the coordinator sends a PrepareSnapshot command *without* a timestamp to all bricks in the system. Upon receiving this command, each brick starts to process future write requests by appending the new data to a persistent log rather than updating in place, and returns to the coordinator the newest timestamp of data in the volume that was updated in place. If the coordinator receives a *global quorum* (i.e., a quorum in every replica set) of replies, it proceeds to the second phase by sending a CommitSnapshot command to all bricks, with the newest timestamp in the received replies. Each brick then moves the data in its log with timestamps older than the snapshot timestamp to the old map (i.e., the snapshot map), creates the new data structure for the current volume, moves the other data in the log to the new map, and ends the logging for future writes in this volume. The snapshot operation is completed thus far. On the other hand, if the coordinator does not receive a global quorum of replies, it sends an AbortSnapshot command to all bricks in the second phase. Each brick moves the new data in its log to its current map and ends the logging.

An issue with the log-based approach is the cost for first writing data in the log and then moving it to the maps. It may require an additional write and an additional read compared to the tentative data structure approach, even in normal cases. On the other hand, the additional disk accesses may not be a siginificant overhead if there are only a small number of them. Another issue with this approach is how to end the logging in a brick if the brick did not receive the second phase command from the coordinator for some reason. Unlike in the other approach, it is not desirable here to rely on a reader of the snapshot to invoke a recovery procedure, because the cost increases as the logging goes on. Therefore, a timeout should be set on each brick for receiving the second phase command. After the timeout, a brick can decide to abort the snapshot as if it had received a AbortSnapshot command from the coordinator.

The snapshot deletion in this approach is the same as that in the approach based on tentative maps.

In the log-based approach, a snapshot is valid if and only if both of the following two conditions are true:

1. There exists a global quorum of bricks that have a committed data structure of the snapshot; and

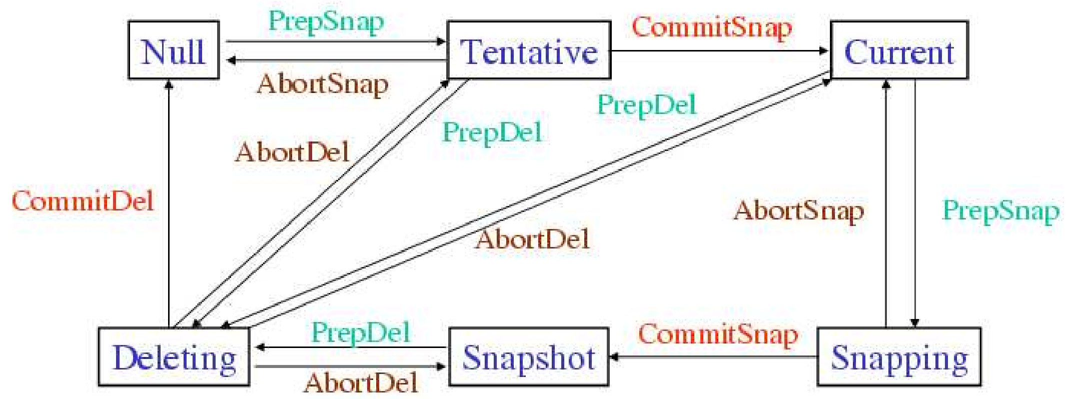


Figure 4: Map state transition

2. There exists a global quorum of bricks that have the snapshot in in non-deleting and non-null state.

Condition 1 above is stricter than its counterpart in the tentative maps based approach due to the possibility of timeout. It also makes it harder (if not impossible) to optimize the common cases by determining the validity of a snapshot with only a local quorum.

# 7 Implementation and test

The snapshot creation and its interaction with data operations described in this paper has been implemented in the FAB prototye. The following test has been run on the prototype. FAB is started with 4 virtual bricks on the same physical machine. A Buttress2 [21] process then starts up as a FAB client, and creates a number of volumes in FAB. After that, the Buttress2 client issues I/O requests to the FAB volumes according to I/O traces on the real volumes. Another process is forked to issue snapshot commands on one of the FAB volumes at a given interval, concurrently with the I/O requests. The traces that were used in the experiments are located in /mount/cello/traces/harp/sanitized/day10.srt.gz.

# 8 Performance

Each snapshot creation, deletion or recovery operation requires two global broadcast messages and replies. Each snapshot read operation requires one local broadcast messages and replies (within a replica set) if the validity of the snapshot can be determined locally, otherwise requires two additional global broadcast messages and replies for the recovery.

# 9 Related work

Existing snapshot solutions for storage systems include EMC TimeFinder, EMC SnapView, SUN StorEdge Instant Image, Veritas Storage Checkpoint, HP OpenView OmniBack, and Compaq Enterprise Volume Manager. Each of these existing solutions has at least one of the following limitations:

1. Need to pause the applications or change the operation mode of file system/database when a snapshot is taken.
2. Need coordination on the hosts.
3. Snapshot is stored in volatile memory only.
4. There is a limit of one snapshot per LU.
5. Require physical mirrors to be established and split to produce the snapshot.
6. Require a centralized controller or controller pair.

There is extensive research on consistent snapshots or checkpoints for distributed processes in the literature.

The Chandy-Lamport algorithm [8], Spezialetti-Kearns algorithm [10], Venkatesan's incremental algorithm [15] and Helary's wave synchronization algorithm [12] are non-blocking algorithms for FIFO channels that take into account internal messages only. Algorithms that block communications when a snapshot is taken are blocking algorithms, and others are non-blocking ones. Two messages sent to the same FIFO channel are received in the same order as sent. In the Chandy-Lamport algorithm, an initiator takes a snapshot and broadcasts a marker (a snapshot request) to all processes. Each process takes a snapshot upon receiving the first marker and rebroadcasts the marker to all processes before sending any application message. The Spezialetti-Kearns algorithm optimizes concurrent initiation of snapshot collection and efficiently distributes the recorded snapshot in the Chandy-Lamport algorithm. The Venkatesan algorithm optimizes the basic snapshot algorithm to efficiently record repeated snapshots of a distributed system that are required in recovery algorithms with synchronous checkpointing. The Helary algorithm incorporates

the concept of message waves in the Chandy-Lamport algorithm. A process begins recording the local snapshot when it is visited by the wave control message.

The Helary's non-FIFO algorithm , Lai-Yang algorithm [13] and Li et al algorithm [14] , Elnozahy et al algorithm [17], Silva algorithm [18] and Mattern algorithm [12] are non-blocking algorithms for non-FIFO channels that take into account internal messages only. In a non-FIFO channel, messages can be lost, duplicated or reordered. The Helary algorithm uses message inhibition to avoid an inconsistency in a global snapshot. After a process i has sent a marker on the outgoing channel to process j, it does not send any messages on this channel until it is sure that j has recorded its local state. The Lai-Yang algorithm fulfills the role of a marker in a non-FIFO system by using a coloring scheme. Every process is initially white and turns red while taking a snapshot. Every message has the same color as the sending process. Every white process takes its snapshot no later than the instant it receives a red message. The Li et al's algorithm tag markers so as to generalize the red/white colors of the Lai-Yang algorithm to accommodate repeated invocations of the algorithms and multiple initiators. In the Elnozahy algorithm and Silva algorithm, snapshot indices serve the role of markers, where a snapshot is triggered when the receiver's local snapshot index is lower than the piggybacked snapshot index. The Mattern algorithm is based on vector clocks. The initiator ticks its local clock and selects a future vector time s for the global snapshot. It then broadcasts s and freezes all activity until it receives an acknowledgement from every process. After all acknowledgements are received, the initiator increases its vector clock to s and broadcast a dummy message to all processes. Each process increases its clock to a value ¿= s upon receiving the dummy message. Each process also takes a local snapshot just before its clock reaches a value $\geq s$.

In the Tong et al algorithm [20], loosely synchronized clocks are used to trigger the local snapshots of all processes at approximately the same time without an initiator. Each process then waits for a period that equals the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process. To guarantee checkpoint consistency, either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked.

In the Koo-Toueg algorithm [19], only those processes that have communicated with the snapshot initiator either directly or indirectly since the last snapshot take new snapshots. After a process takes a snapshot, it does not send any message until all involved processes have taken snapshots, although receiving a message after the snapshot is allowable.

The Acharya-Badrinath algorithm [11] and Alagar-Venkatesan algorithm [9] assume that the underlying system supports causal message delivery. The causal order of message delivery means that two messages destined to the same process (but not necessarily originated from the same process) are received in the same order as sent. It is a stronger delivery guarantee than FIFO. Both algorithms are considerably simplified and use an identical principle to record the state processes. An initiator process broadcasts a token to every process including itself. A process records its local snapshot when it receives the token and sends the recorded snapshot to the initiator.

In summary, our snapshot algorithm differs from each existing algorithm referenced above in at least one of the follow five aspects:

1. Interactions with the outside world: Most existing algorithms take only inter-process messages into account when a snapshot is taken, while our algorithm handles external messages (e.g., a data write request from a FAB client) as well as internal ones.

2. Assumptions on communication channels between processes: Causal, FIFO, and non-FIFO. Our algorithm assumes non-FIFO channels because messages can be lost/resent due to the crash/reboot of bricks.

3. Blocking vs. non-blocking: Our algorithm is non-blocking.

4. Required responses: Most existing algorithms require the responses from all process in order to proceed with a snapshot, while our algorithm requires only a global quorum of responses.

5. Full state duplication vs. copy on write vs. move on write: Some algorithms duplicate the full state of all processes for each snapshot, some copy the old state to a read-only location upon a write and leave the new state in the original location, while our algorithm leaves the old state in the original location and stores the new state in a new location.

# References

[1] Nabil Osorio and Bill Lee, Guideline for using Snapshot Storage Systems for Oracle Databases , white paper, Oracle Corporation, October 2001. This white paper describes the usage of storage snapshot technology with Oracle databases. It shows how local snapshots, which include both split of mirror and copy on write techniques, and the remote snapshot, which uses copy on write technique, can be used for hot backups, cold backups, remote point-in-time copy and disaster recovery.

[2] Compaq Enterprise Volume Manager and Oracle8 Best Practices , white paper, 11EE-1199A-WWEN, Storage Division, Compaq Computer Corporation.

[3] Validation of Oracle8i and Oracle9i with EMC TimeFinder , engineering white paper, part No. H589.1, April 2002.

[4] Using the EMC FC4700 SnapView feature with Oracle 8i , technical paper, EMC mid-range partner engineering team, 2001.

[5] Performing Oracle 8i Split Mirror Backups Using HP OpenView OmniBack II , white paper, April 2000.

[6] Usage Guide for Sun StorEdge Instant Image Software with Oracle8 , white paper, 2001.

[7] Guidelines for Using VERITAS Storage Checkpoint and Storage Rollback with Oracle8i Databases , white paper, August 2001.

[8] K. Mani Chandy and Leslie Lamport, Distributed snapshots: determining global states of distributed systems, ACM Transactions on Computer Systems (volume 3 number 1 pages 63-75) February 1985.

[9] S. Alagar and S. Venkatesan, An optimal algorithm for distributed snapshots with causal message ordering, Information Processing Letters (volume 50 number 6 pages 311-316) 27 June 1994.

[10] Madalene Spezialetti and Phil Kearns, Efficient distributed snapshots, Proceedings of 6th International Conference on Distributed Computing Systems (ICDCS), (Cambridge, MA) Report Catalog number 86CH22293-9, IEEE Computer Society Press, (pages 382-388) May 1986.

[11] A. Acharya and B. R. Badrinath, ”Recording distributed snapshots based on causal order of message delivery”, Information Processing Letters, 1992.

[12] J-M Helary, ”Observing global states of asynchronous distributed applications”, in Proceedings of 3rd International Workshop on Distributed Algorithms, LNCS 392 (Berlin: Springer), pp 124-24, 1989.

[13] T H Lai and T H Yang, ”On distributed snapshots”, Information Processing Letters, 25 153-8, 1987.

[14] H F Li, T Radhakrishman and K Venkatesh, ”Global state detection in non-FIFO networks”, In Proceedings of 7th International Conference on Distributed Computing Systems, pp 364-70, 1987.

[15] S Venkatesan, ”Message-optimal incr4emental snapshots”, Journal of Computer Software Engineering, 1 211-31, 1993.

[16] F. Mattern, ”Efficient algorithms for distributed snapshots and global virtual time approximation”, in Journal of Parallel and Distributed Computing”, Vol. 18, No. 4, 1993.

[17] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. 11th Symposium on Reliable Distributed Systems, 1992.

[18] L. M. Silva. ”Checkpointing mechanisms for scientific parallel applications”, Ph.D. thesis, University of Coimbra, Portugal, March 1997.

[19] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. IEEE Transactions on Software Engineering (volume 13 number 1 pages 23-31) January 1987.

[20] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. Proceedings of 8th Symposium on Reliable Distributed Systems, (Seattle, WA) Report catalog number 88CH2807-6, IEEE Computer Society Press, (pages 12-20) 10-12 October 1989.

[21] Eric Anderson, Mahesh Kallahalla, Ram Swaminathan, Mustafa Uysal, “Buttress: a toolkit for flexible and high fidelity I/O benchmarking”, FAST 2004.

[22] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch, “A De-centralized Algorithm for Erasure-Coded Virtual Disks”.
DSN 2004.

# A Snapshot operations pseudo C code

```
//Functions that coordinators call in processing clients' requests

CreateSnapshot(volName, snapName){
  ts = NewTimestamp();
  replies = broadcast(allBricks, Message(PrepareSnapshot, ts, volName, snapName));
  if(GlobalQuorum(replies, OK)){
    broadcast(allBricks, Message(CommitSnapshot, ts, volName, snapName));
    return OK;
  }else{
    broadcast(allBricks, Message(AbortSnapshot, snapName));
    return NOK;
  }
}

ReadSnapshot(snapName){
  replies = broadcast(replicaSet, Message(ProbeSnapshot, snapName));
  snapshot = IsValid(replies);
  if(snapshot==VALID){
    return RepliedVersion(replies, ~TENTATIVE);
  } else if(snapshot==NULL){
    return NULL;
  }else{
    return RecoverSnapshot(snapName);
  }
}

DeleteSnapshot(snapName){
  replies = broadcast(allBricks, Message(PrepDelSnapshot, snapName));
  if(GlobalQuorum(replies, OK)){
    broadcast(allBricks, Message(CommitDelSnapshot, ts, snapName));
    return OK;
  }else{
    broadcast(allBricks, Message(AbortDelSnapshot, snapName));
    return NOK;
  }
}

//Helper functions that coordinators call in processing clients' requests;
//todo: convert English to C

IsValid(replies){
  if(replies contain a quorum of maps that have consistent ids and
     non-DELETING state, and the quorum includes at least one non-TENTATIVE;
     state){
    //existing snapshot;
    //snapshot = a non-TENTATIVE snapshot in replies;
    return VALID;
  } else if(LocalQuorum(replies, NULL)){
    //aborted or non-existent snapshot;
```

```
    return NULL;
  } else {
    return UNDETERMINED;
  }
}

RecoverSnapshot(snapName){
  replies = broadcast(allBricks, Message(ProbeSnapshot, snapName));
  if((replies contain at least one non-TENTATIVE state or a
      global quorum with non-NULL state) and
     (replies contain a global quorum with non-DELETING state)){
    ts = RepliedTimestamp(replies);
    volName = RepliedVolumeName(replies);
    broadcast(allBricks, Message(ValidateSnapshot, ts, volName, snapName));
    return RepliedVersion(replies);
  }else{
    if((replies contain no non-TENTATIVE state and contain a global
quorum of NONE state) or
       (replies contain a global quorum of DELETING or NULL state)){
      broadcast(allBricks, Message(InvalidateSnapshot, snapName));
      return NULL;
    }else{
      return UNDETERMINED;
    }
  }
}

//Functions that replicas call in processing coordinators' requests;

PrepareSnapshot(ts, volName, snapName){
  vol = FindVolume(volName);
  if(vol==NULL) return NOK;
  %LockAllBlocks(vol);
  curMap = CurrentVersion(vol)->map;
  status = (ts>max(curMap->creationTs, curMap->dataTs));
  if(status){
    snapVer = DoPrepareSnapshot(ts, vol, snapName);
  }
  %UnlockAllBlocks(vol);
  return status;
}

AbortSnapshot(snapName){
  vol = FindVolume(snapName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  nextVer = NextVersion(snapVer);
  nextMap = nextVer->map;
  if(nextMap->creationTs<nextMap->dataTs){
```

```
    //nextVer has been written since phase 1, merge its data and map to snapMap
    MergeMap(snapMap, nextMap);
  }
  snapMap->next = nextMap->next;
  DeleteMap(nextMap);
  nextMap = nextVer->map = snapMap;
  DeleteVersion(vol, snapVer);
  ClearFlag(nextMap->state, SNAPPING);
  sync();
  %UnlockAllBlocks(vol);
}

CommitSnapshot(ts, volName, snapName){
  vol = FindVolume(volName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  if(snapVer==NULL){
    curMap = CurrentVersion(vol)->map;
    if(ts>max(curMap->creationTs, curMap->dataTs)){
      //tentative snapshot was not created in phase 1, redo it now;
      snapVer = DoPrepareSnapshot(ts, vol, snapName);
    } else {
      return;
    }
  }
  snapMap = snapVer->map;
  if(snapVer==CurrentVersion(vol) and ts>=snapMap->creationTs and ts>snapMap->dataTs){
    ClearFlag(snapMap, SNAPPING);
    SetFlag(snapMap, SNAPSHOT);
    nextMap = snapMap->next;
    ClearFlag(nextMap, TENTATIVE);
    vol->snapTs = 0;
    sync();
  }
  %UnlockAllBlocks(vol);
}

ProbeSnapshot(snapName){
  vol = FindVolume(snapName);
  if(vol==NULL) return NULL;
  snapVer = FindVersion(vol, snapName);
  if(snapVer==NULL){
    return NULL;
  }else{
    return snapVer;
  }
}

PrepDelSnapshot(snapName){
  vol = FindVolume(snapName);
```

```
  if(vol==NULL) return NOK;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  if(IsFlagSet(snapMap, SNAPPING)){
    status = NOK;
  }else{
    SetFlag(snapMap, DELETING);
    status = OK;
    sync();
  }
  %UnlockAllBlocks(vol);
  return status;
}

AbortDelSnapshot(snapName){
  vol = FindVolume(snapName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  ClearFlag(snapMap, DELETING);
  sync();
  %UnlockAllBlocks();
}

CommitDelSnapshot(snapName){
  vol = FindVolume(snapName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  if(IsCurrentVersion(vol, snapVer)){
    //this is the current map, delete it without merging;
    if(snapMap->prev!=NULL){
      snapMap->prev->next = NULL;
    }
    DeleteMap(snapMap);
    DeleteVersion(vol, snapVer);
   } else {
    //merge its nextMap to it and assign it to its nextVer
    nextVer = NextVersion(snapVer);
    nextMap = nextVer->map;
    MergeMap(snapMap, nextMap);
    nextVer->map = snapMap;
    DeleteMap(nextMap);
    DeleteVersion(snapVer);
  }
  sync();
  %UnlockAllBlocks(vol);
}
```

```
ValidateSnapshot(ts, volName, snapName){
  vol = FindVolume(volName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  if(IsFlagSet(snapMap, TENTATIVE)){
    CommitSnapshot(ts, volName, snapName);
  }
  if(IsFlagSet(snapMap, DELETING)){
    AbortDelSnapshot(snapName);
  }
  %UnlockAllBlocks(vol);
}

InvalidateSnapshot(snapName){
  vol = FindVolume(snapName);
  %if(vol==NULL) return;
  %LockAllBlocks(vol);
  snapVer = FindVersion(vol, snapName);
  snapMap = snapVer->map;
  if(IsFlagSet(snapMap, TENTATIVE)){
    AbortSnapshot(snapName);
  }
  if(IsFlagSet(snapMap, DELETING)){
    CommitDelSnapshot(snapName);
  }
  %UnlockAllBlocks(vol);
}

// Helper functions that replicas call in processing coordinators'
// requests

DoPrepareSnapshot(ts, vol, snapName){
  curVer = CurrentVersion(vol);
  curMap = curVer->map;
  newMap = curVer->map = CreateMap();
  SetFlag(newMap, TENTATIVE);
  newMap->creationTs = ts;
  newMap->dataTs = curMap->dataTs;
  newMap->prev = curMap;
  newMap->next = NULL;
  snapVer = CreateVersion(vol, snapName);
  snapVer->map = curMap;
  SetFlag(curMap, SNAPPING);
  curMap->next = newMap;
  sync();
  return snapVer;
}
```

## B Data operations pseudo C code

```
//Functions that coordinators call in processing clients' requests

read(vol, addr){
  ts = NewTimeStamp();
  target = RandomReplica();
  replies = broadcast(replicaSet, Message(Read, vol, addr, ts, target));
  snapshot = IsValid(replies);
  if(snapshot==VALID){
    if(replies containt a quorum that have true status
       and consistent valTs){
      if(NeedFreeze(replies)){
broadcast(replicaSet,
  Message(Write, RepliedMapId(replies), addr, NULL,
  RepliedTimestamp(replies)));
      }
      if(target replied and ){
return RepliedDataValue(replies, target);
      } else {
return read(vol, addr); //select a different random target
      }
    } else {
      return recover(vol, addr);
    }
  } else if(snapshot==UNDETERMINED){
    replies = RecoverSnapshot(vol->name);
    if(IsValid(replies)){
      return read(vol, addr);
    } else {
      return NULL;
    }
  } else {
    return NULL;
  }
}

write(vol, addr, val){
  ts = NewTimestamp();
  replies = broadcast(replicaSet, Message(Order, vol, addr, ts, false));
  snapshot = IsValid(replies);
  if(snapshot==VALID){
    if(replies do not contain a quorum of true status){
      return NOK;
    }
    replies = broadcast(replicaSet,
Message(Write, RepliedMapId(replies), addr, val, ts));
    if(replies contain a quorum of true status){
      return OK;
    } else {
      return NOK;
```

```
    }
  } else if(snapshot==UNDETERMINED){
    replies = RecoverSnapshot(vol->name);
    if(IsValid(replies)){
      return write(vol, addr, val);
    } else {
      return NULL;
    }
  } else {
    return NULL;
  }
}

recover(vol, addr){
  ts = NewTimestamp();
  replies = broadcast(replicaSet, Message(Order, map, addr, ts, true));
  snapshot = IsValid(replies);
  if(snapshot==VALID){
    if(replies do not contain a quorum of true status){
      return NULL;
    }
    val = RepliedDataValue(replies, HighestValTs);
    replies = broadcast(replicaSet,
Message(Write, RepliedMapId(replies), addr, val, ts));
    if(replies contain a quorum of true status){
      return val;
    } else {
      return NULL;
    }
  } else if(snapshot==UNDETERMINED){
    replies = RecoverSnapshot(vol->name);
    if(IsValid(replies)){
      return recover(vol, addr);
    } else {
      return NULL;
    }
  } else {
    return NULL;
  }
}

//Functions that replicas call in processing coordinators' requests;

Read(name, addr, ts, target){
  vol = FindVolume(name);
  map = FindVersion(vol, name)->map;
  //make sure no outstanding writes and no future writes
  status = (map->valTs[addr]>=map->ordTs[addr] and ts>map->creationTs);
  if(!status){
    return Message(Read-R, false);
  }
```

```
  //eventually, all read blocks in snapshot will have the timestamp
  //map->creationTs so that no writes newer than that can be applied
  //to them
  if(map->valTs[addr]<map->creationTs){
    //need to update the valTs to creationTs so that no future writes
    //are allowed
    needFreeze = true;
    //like the order command in write operation
    map->ordTs[addr] = map->creationTs;
    sync();
  } else {
    needFreeze = false;
  }
  if(MyId==target){
    val = ReadDataValue(map, addr);
  } else {
    val = NULL;
  }
  return Message(Read-R, status, map->id,
 max(map->valTs[addr],
     map->ordTs[addr]), val, needFreeze);
}

Order(name, addr, ts, read){
  vol = FindVolume(name);
  map = CurrentVersion(vol)->map;
  while(map!=NULL){
    if(ts<=map->creationTs){
      map = map->prevMap;
    } else {
      break;
    }
  }
  if(map==NULL){
    return Message(Order-R, false);
  }
  status = (ts>max(map->valTs[addr], map->ordTs[addr]));
  if(!status){
    return Message(Order-R, false);
  }
  map->ordTs[addr] = ts;
  if(ts>map->dataTs){
    map->dataTs = ts;
  }
  sync();
  if(read){
    val = ReadDataValue(map, addr);
  } else {
    val = NULL;
  }
  return Message(Order-R, status, map->id, map->valTs[addr], val);
```

```
}

Write(mapId, addr, newVal, ts){
  map = FindMap(mapId);
  if(map == NULL){
    return Message(Write-R, false);
  }
  status = (ts>=map->ordTs[addr] and
    ts>max(map->valTs[addr], map->creationTs));
  if(status){
    if(newVal!=NULL){
      WriteDataValue(map, addr, newVal);
    }
    map->valTs[addr] = ts;
    if(ts>map->dataTs){
      map->dataTs = ts;
    }
    sync();
  }
  return Message(Write-R, status);
}
```