Snapshot in Distributed Bank System – Venkatesan's Incremental Algorithm

Nathan Harvey (997261), Ziyu Wang (1560831)

Abstract – In distributed systems, capturing a consistent global state is fundamental for fault recovery and debugging. This report focuses on Venkatesan's incremental snapshot algorithm, which builds upon the foundations of Chandy-Lamport to provide a lightweight and iterative method for global state recording. We implement the algorithm within a distributed banking system prototype and evaluate its consistency and suitability for real-world deployments.

Keywords – Distributed systems · Global snapshot · Chandy–Lamport · Incremental snapshot · Causal consistency · Fault recovery

I. Introduction and background

In asynchronous distributed systems, snapshots capture a globally consistent state, which is essential for debugging, failure recovery, and verifying the correctness of concurrent executions. However, without shared memory or a global clock, each process operates independently and message delivery can be delayed or reordered, making consistency non-trivial.

Snapshot mechanisms must therefore solve two key challenges:

A. *Message Inclusion*: deciding which in-transit messages should be included in the snapshot
B. *Snapshot Timing*: determining the precise moment when each process records its local state

In financial applications, such as a distributed banking system, global snapshots serve to verify that the total amount of money remains invariant across failures and restarts. Without a coordinated snapshot, independently taken local snapshots might lead to an incorrect global total when the system recovers. Since financial applications often require frequent snapshots to meet strict recovery-time objectives, we chose Venkatesan's incremental snapshot algorithm, which preserves global consistency while significantly reducing coordination costs.

II. Survey of related and direct work

A. Chandy-Lamport Algorithm

Chandy–Lamport (1985) assumes an asynchronous distributed system of processes connected by reliable, FIFO channels and lacking a global clock. Under these constraints, it guarantees that messages recorded as in-transit respect causal ordering. In the initiation phase, one or more initiators record their local state and multicast a special marker on all outgoing channels before sending any further application messages. Once first receipt of a marker on channel c, a process records its current local state, multicasts a marker on all of its outgoing channels, and buffers all subsequent messages arriving on c as the channel state; any later marker on that channel is simply forwarded without additional state recording. In the snapshot completion phase, once every process has recorded its local state and every channel has captured its buffered messages, the union of these local and channel states yields a globally consistent snapshot. By enforcing the Marker Sending Rule, a process must multicast a marker on all outgoing channels before sending any further application messages. The algorithm strictly delimits in-transit messages with markers, thereby preserving causality and ensuring both correctness and consistency. In addition to providing a consistent global state, this snapshot algorithm provides a method for detecting stable properties, which are conditions on the global state that after becoming true in a single state, remain true in all future states.

Having established the workflow, we now turn to its complexity characteristics. The message complexity is linear in the number of channels. Since each channel only receives a marker once, given E is the

number of channels in the system, the message overhead would be O(E). If the system is fully connected, the message overhead would be $O(N^2)$, where N is the number of nodes. In terms of time, considering d is the diameter of the system graph, the time complexity would be O(d). This is because marker messages propagate through the network along the longest shortest-path between any two processes, and a marker must traverse d hops before the farthest process can record its state in the worst case.

Despite its clear complexity bounds, practical deployment reveals further trade-offs. This algorithm is straightforward to implement and directly produces a consistent global snapshot. Its reliance solely on FIFO channel guarantees, which are common in middleware, makes it broadly applicable across an asynchronous distributed system. Moreover, because it cleanly separates application messages from control markers, the system remains non-blocking, normal message processing never stops to wait for snapshot coordination. However, several drawbacks limit its scalability. First, buffering every in-transit message can incur substantial memory overhead, particularly in high-throughput systems. Second, requiring every process to participate in each snapshot imposes coordination costs that grow with system size and dynamism. Finally, broadcasting a fresh flood of markers on every channel for each snapshot generates significant control-message traffic, making the method inefficient for continuously monitored or high-frequency snapshot scenarios.

B. Efficient Algorithms for Global Snapshots in Large Distributed Systems

Given high-performance computing environments, the communication topology is usually considered as a complete graph. Although Chandy–Lamport guarantees consistency, the algorithm incurs $O(N^2)$ message overhead and O(N) space per process in a network of N nodes, making the approach prohibitive in the modern world. To address the scalability limitations, Garg et al. (2010) proposed three algorithms, assuming an asynchronous message-passing model over a fully connected topology, without relying on FIFO channel guarantees.

Firstly, the grid-based method arranges the N processes into a $\sqrt{N} \times \sqrt{N}$ logical grid and performs row- and column-level broadcasts and convergecasts to compute each node's deficit. The deficit refers to the number of in transit messages, and a convergecast is a process that propagates information starting from edge nodes to a single node as opposed to a broadcast which propagates information from a single node to all nodes. Therefore, each process sends only $O(\sqrt{N})$ messages of size $O(\sqrt{N})$ while keeping O(N) local space.

Secondly, the tree-based algorithm builds on a fixed spanning tree. Each node tracks its deficit counter and participates in a tree-structured convergecast to aggregate deficits, then uses distributed message counting over the same tree to detect when all in-transit messages have arrived. This achieves O(1) control messages, O(1) space, and $O(\log N \times \log \frac{W}{N})$ messages per process, where W is total in-transit messages.

Finally, the centralized approach designates a single coordinator to collect and redistribute deficit information, achieving O(1)-size messages, $O(\log \frac{W}{N})$ message complexity per-process while still using O(1) space.

The primary strength of this work is they dramatically reduce both message and time complexity compared to classical methods, especially in the high-performance computing environment situations. On the other hand, each method introduces practical considerations. To begin with the grid-based algorithm's $O(\sqrt{N})$ message size may be restricted by network maximum transmission unit (MTU) limits. The tree-based technique depends on maintaining a fault-free spanning tree, making it vulnerable to node failures or dynamic membership. Lastly, the centralized approach creates a coordination bottleneck and single point of failure unless augmented with additional fault-tolerance.

C. Message-optimal incremental snapshots

Venkatesan (1989) proposes an extension to the Chandy-Lamport algorithm to allow it to complete snapshots in an optimal number of messages. To do this each process records messages sent on each channel since the last snapshot, to avoid having to send marker messages on channels that weren't used. This works because the marker message is usually used to mark when a process can stop recording

messages in-flight when it takes its own snapshot, but it is unnecessary if the process knows there were no messages in flight because none were sent. To ensure all processes receive a message to start the snapshot at least once the algorithm assumes a spanning tree of the network is known, which is an acyclic graph starting from a primary node which visits each node exactly once.

They are able to prove that Venkatesan's algorithm will always send an optimal number of messages. In the worst case where every process sends a message on every connected channel the algorithm sends the same number of messages as the Chandy-Lamport algorithm. Venkatesan's algorithm performs best when messages are sent non-uniformly, and when snapshots are taken frequently.

Venkatesan's algorithm makes the same assumptions as the Chandy-Lamport algorithm, but also requires a spanning tree be known before starting and a single process be designated the primary process, which is the only process allowed to start a snapshot. Because of this it makes using Venkatesan's algorithm require a rigid structure, and if any processes need to join or leave the network or connections between existing processes are created or destroyed the algorithm needs to restart. If there are changes to the network of processes more frequently than snapshots are taken the algorithm also degrades to the Chandy-Lamport algorithm.

The main advantage of Venkatesan's algorithm is the optimal number of control messages it needs to send to complete the snapshot. This means in ideal scenarios the number of messages required to complete a snapshot is much lower than could be achieved with the Chandy-Lamport algorithm. The only disadvantage of using Venkatesans's algorithm happens when all channels are used to send a snapshot, which should only be happening if the time between snapshots is too low or if the system is unsuited to the algorithm.

D. On Distributed Snapshots

Lai and Yang (1987) propose an algorithm that relaxes the first in first out (FIFO) restriction on channels between processes and requires no control messages. The problem with not having a FIFO guarantee on messages is a process can't guarantee they have received all messages from the previous snapshot, so they don't know when to finish their snapshot. Their algorithm solves this by marking all processes with a colour, either red or white, to indicate if they were sent before or after a snapshot. Any process can start a snapshot randomly or just before they receive a message of the opposite colour, at which point they switch their own colour. Instead of the receiving process recording all messages sent since the snapshot began until it receives a control message, instead each process sends a record of all messages it has sent to other processes to be collected into a global snapshot using the colour of messages sent to tell which messages happen before the snapshot.

Having to send every message sent, and having to piggyback the set of past messages on every message sent, causes a large overhead to sending messages. This makes it very inefficient over slow networks, although they suggest this can be overcome by either using the algorithm for applications that only care about the number of messages sent and not the message's contents, or by resetting the sent and received sets after snapshots in applications that have to take repeated snapshots.

The algorithm also requires processes to be sending messages frequently to function. If a process doesn't need to send an underlying message for a long period of time, it will be unable to convey to other processes that it hasn't sent any messages, and the other processes may be mistaken that messages had been sent and lost.

E. A Cooperative Partial Snapshot Algorithm for Checkpoint-Rollback Recovery of Large-Scale and Dynamic Distributed Systems

Distributed systems that are large enough for algorithms with high time complexity often are made of clusters of processes that communicate frequently with other processes in their cluster, but rarely with processes outside their cluster. Processes are often in clusters made up of the processes that joined the network by connecting to the same process. Moriya and Araragi (2005) take advantage of this to propose the SubSnapShot (SSS) algorithm which is able to take a global snapshot by having clusters of processes take a local snapshot of their cluster (sub-snapshot), then is able to rollback after a partial system crash using those sub-snapshots.

Each process keeps track of a set of processes called its dependency set consisting of all processes it sends messages to, receives messages from, that connect to it, or that it connected to when it first joined the system. It sends this set when taking a snapshot, and can use the dependency sets of processes that send it marker messages during a snapshot to know when all processes in a cluster have communicated with it about the snapshot, even if it was not by directly sending it a message.

The dependency sets of processes changes during execution of the system, and is reset after a snapshot is taken. If a process rarely communicates with another, they will only be considered part of the same cluster during snapshots where they have communicated with each other.

The clustered nature of the algorithm means the time complexity of the algorithm is no longer bound by the total number of processes in the system, but instead by the number of processes in the cluster. If the distributed system does not display clusters of processes that are more likely to communicate with each other than with other processes in the system, this algorithm will introduce more overhead than simply using the Chandy-Lamport algorithm.

Although not specified in the paper, the algorithm needs a solution for 2 processes in the same cluster initiating the snapshot at once. A previous initiator does not necessarily know when the cluster changes, so always using the last initiator is a good method but not foolproof. Running an election algorithm for every snapshot would also be inefficient, doubling the amount of messages required to complete a snapshot. A solution might be to cancel the snapshot with the lower initiator id, and as each process receives the first marker of the continuing snapshot it should invalidate the snapshot progress made by the cancelled snapshot. Because the cancelled snapshot can't have finished before the collision takes place, the continuing snapshot should be able to capture information from all processes in the cluster without them deleting information due to believing the cancelled snapshot had completed.

The algorithm works best when the system is composed of many small clusters that frequently communicate with each other, but don't often communicate between clusters. The algorithm has 2 worst cases, the first is when all processes commonly communicate with each other process uniformly. The second worst case is when the system is made up of many small clusters that infrequently communicate with each other, but which communicate with each other commonly when many clusters' snapshot algorithms are already running, which need to be cancelled in favour of a single snapshot algorithm over the entire system. This causes a large amount of work that has to be cancelled, while control messages need to be resent between processes for the snapshot which is continuing.

III. Discussion of our application

We chose to implement the incremental algorithm proposed by Venkatesan (1989) in a simple banking system. We thought that Venkatesan's algorithm had the most real world use for taking snapshots for crash recovery due to its optimal message overhead and incremental nature. The ideal scenario for using Venkatesan's algorithm is a system in which nodes only communicate with a small number of other processes for each period of time covered by 1 snapshot. This is a common setup for financial systems, where processes will commonly make many financial transactions with a few other closely related processes. We decided to not go with Moriya and Araragi's (2005) SubSnapShot algorithm due to its complexity, and worse case performance. The worst case for Venkatesan's algorithm is it falls back to Chandy-Lamport, but SubSnapShot can create duplicate work with many times more messages than the Chandy-Lamport solution.

We modeled a system that starts with a set amount of money in the system. Each process is able to send money to any connected process as long as it has at least that much money itself, doing so reduces the amount of money it has by the amount sent. At all times the amount of money in the system should remain constant, no money should enter or leave the system. For the sake of simplicity all processes are aware at the start of the system which processes they are connected to, and no process can leave or enter the system. We also decide on a primary process and define a spanning tree before the system starts.

When the system starts, each process first opens a socket to listen for incoming connections, then tries to connect to all connected processes based on the config. For all pairs of processes one of the processes will start first and the outgoing connection will fail, and one of the processes will start second and the outgoing connection will succeed. Each process also opens a socket connection to itself for sending messages to itself. Next each process will begin to send money randomly to connected processes. It

selects a single connected process, sends a single message with the amount of money contained, then waits a short amount of time. We do this to enable snapshots where not all connections have had messages sent to demonstrate the use case of Venkatesan's algorithm, but need to balance this with sending enough messages to demonstrate messages captured in the snapshot in channels. While this is happening the primary process is also starting a snapshot every 5 seconds, with a small amount of randomness added. We also added a small delay after handling snapshot control messages, to give the system time to send messages to other processes that would arrive after the snapshot started and would be captured in the snapshot as a message in the channel.

If a single process in our system crashes, all processes in the system crash. The system can then be started with the same config, and also specifying a global snapshot to recover at. This overrides the starting money for each process and specifies a list of messages that were on the channel at the time of the snapshot that need to be received before the system can start. For simplicity we decided to not allow processes to exit the system and to recover from a total system crash instead of a single process crash.

We made the decision to not handle receiving messages from channels in parallel, and to instead handle messages in series one after another from all channels. We did this because we wanted to focus on the complexities brought by the parallel execution between processes without having to also handle parallel processing of threads in a single process. Our method of handling the messages in series achieves the same effect as using a mutex lock on the process when handling a message but allows the message handling to occur in the main thread, which makes debugging easier.

We made 2 changes to Venkatesan's algorithm. The psuedocode provided in Venkatesan's paper saves a snapshot when it receives an init_snap message for the next snapshot, but we made the decision to save the snapshot immediately when the current snapshot finishes. We also made the decision for each process to send its snapshot to its parent, so the primary process that initiated the snapshot finishes with all the processes snapshots. We did this to allow inspection of the snapshot at runtime to assert that the snapshot was correct and the amount of money was consistent, but we could have also saved each processes' snapshot individually.

A possible extension we could make in the future would be to calculate the spanning tree when the system starts up. A simple method of calculating a spanning tree is to make the parent of a process the first process it actively connects to. The primary process would be the process that actively connects to 0 processes and receives a connection on its listening socket for all connections. This would also allow processes to join during execution. They would have to start with no money to not change the total amount of money in the system, but it would be able to join the spanning tree as a child of the process it connects to, and its set of connections that it has sent messages to would start empty.

It would be harder to extend our application to support processes leaving the system. Processes would have to leave with 0 money in their state to not remove money from the system, and all spanning tree children would have to connect to the spanning tree parent of the leaving process, and set it as their new parent. This would not work if a process left the system abruptly, as these things would need time to complete. The only way the system would be able to continue if a process left abruptly would be to wait for the process to rejoin and roll back to a snapshot before it left.

IV. Conclusions

We surveyed existing snapshot algorithms for their suitability for taking snapshots of a financial system. We found several interesting extensions to the Chandy-Lamport algorithm, and chose Venkatesan's algorithm because it was designed to be optimal when taking incremental snapshots. We then built an application modeled off of financial systems to demonstrate Venkatesan's algorithm.

We mention several ways our application could be extended, and we also believe there are interesting ways to extend the snapshot algorithm further. One way Venkatesan's algorithm could be extended would be to reduce the complexity in systems with a large number of highly connected processes, which could be achieved by finding a minimal spanning tree or optimising the spanning tree to use connections that are commonly used and would always need a marker message sent on them if they weren't part of the spanning tree.

References

Chandy, K. M., & Lamport, L.. (1985). Distributed snapshots. ACM Transactions on Computer Systems, 3(1), 63–75. https://doi.org/10.1145/214451.214456

Garg, R., Garg, V. K., & Sabharwal, Y.. (2010). Efficient Algorithms for Global Snapshots in Large Distributed Systems. IEEE Transactions on Parallel and Distributed Systems, 21(5), 620–630. https://doi.org/10.1109/tpds.2009.108

Lai, T. H., & Yang, T. H. (1987). On distributed snapshots. *Information Processing Letters*, *25*(3), 153-158. https://doi.org/10.1016/0020-0190(87)90125-6

Venkatesan, S.. (1989). Message-optimal incremental snapshots. 53–60. https://doi.org/10.1109/icdcs.1989.37930

Moriya, S., & Araragi, T.. (2005). Dynamic snapshot algorithm and partial rollback algorithm for internet agents. *Electronics and Communications in Japan (part III: Fundamental Electronic Science)*, *88*(12), 43–57. https://doi.org/10.1002/ecjc.20208

- Title, Author information, Abstract, Keywords **- Ziyu**
- Introduction and background **- Ziyu**

- Survey of related and direct work **- Split**
- Critical analysis of the algorithms reviewed **- Split**

- Future directions and applications **- Ziyu**

- Discussion of chosen algorithm, implementation details and formal description **- Nathan**
- Conclusions **- Nathan**

Chandy, K. M., & Lamport, L.. (1985). Distributed snapshots. *ACM Transactions on Computer Systems*, *3*(1), 63–75. https://doi.org/10.1145/214451.214456 **- Ziyu**

Garg, R., Garg, V. K., & Sabharwal, Y.. (2010). Efficient Algorithms for Global Snapshots in Large Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, *21*(5), 620–630. https://doi.org/10.1109/tpds.2009.108 **- Ziyu**

Venkatesan, S.. (1989). *Message-optimal incremental snapshots*. 53–60. https://doi.org/10.1109/icdcs.1989.37930 **- Nathan**

Helary, J.-M.. (1989). Observing global states of asynchronous distributed applications. In *Lecture Notes in Computer Science* (pp. 124–135). Lecture Notes in Computer Science. https://doi.org/10.1007/3-540-51687-5_37

Lai, T. H., & Yang, T. H. (1987). On distributed snapshots. *Information Processing Letters*, *25*(3), 153-158. https://doi.org/10.1016/0020-0190(87)90125-6 **-Nathan**

Mattern, F.. (1993). Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, *18*(4), 423–434. https://doi.org/10.1006/jpdc.1993.1075

Taylor, K.. (1989). The role of inhibition in asynchronous consistent-cut protocols. In *Lecture Notes in Computer Science* (pp. 280–291). Lecture Notes in Computer Science. https://doi.org/10.1007/3-540-51687-5_50

Possible other papers

Kshemkalyani, A. D.. (2010). Fast and Message-Efficient Global Snapshot Algorithms for Large-Scale Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, *21*(9), 1281–1289. https://doi.org/10.1109/tpds.2010.24

- Might require knowledge of what a hypercube is, which I don't

Garg, R., Garg, V. K., & Sabharwal, Y.. (2010). Efficient Algorithms for Global Snapshots in Large Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, *21*(5), 620–630. https://doi.org/10.1109/tpds.2009.108

- More efficient time complexity in complete networks

Kim, Y., Nakamura, J., Katayama, Y., & Masuzawa, T.. (2018). *A Cooperative Partial Snapshot Algorithm for Checkpoint-Rollback Recovery of Large-Scale and Dynamic Distributed Systems*. 285–291. https://doi.org/10.1109/candarw.2018.00060 **- Nathan**

- Partial snapshots with coordination for crash recovery, looks interesting

Mark Breakdown
10/100: Description of problem domain
20/100: Synthesis and presentation of background survey
20/100: Comparative analysis
20/100: Discussion of chosen algorithm and its implementation
10/100: Literature selection
10/100: Structure and audience
10/100: Referencing

V. Future directions and applications

Chandy-Lamport Algorithm

Lightweight Incremental Protocols

These protocols track channel activity since the last checkpoint and send markers only on active links. By piggybacking snapshot identifiers onto existing application messages and buffering only new in-transit messages, they reduce control-message volume and memory overhead while still enforcing causal consistency.

Adaptive Snapshot Scheduling

This approach dynamically adjusts checkpoint intervals based on real-time metrics. Under light load, snapshots are deferred to conserve bandwidth and compute resources; during traffic spikes or error bursts, intervals are shortened to minimize the window of unrecovered work. This balance of coordination overhead against recovery latency is especially valuable in environments with bursty workloads.

Garg, Garg, and Sabharwal (2010)

Future work could focus on extending these three algorithms to partially connected or dynamic topologies—for example, by combining adaptive spanning‑tree maintenance with hybrid gossip‑based message counting to accommodate node joins and departures. At the same time, integrating fault‑tolerance mechanisms—such as automatic coordinator re‑election and marker replay upon failure—would enhance overall system robustness. In terms of applications, these high‑efficiency snapshot protocols are not only well suited for MPI‑based HPC checkpoint and IoT or blockchain.

Our prototype demonstrates the correctness and lightweight coordination benefits of Venkatesan's incremental snapshot algorithm in a small-scale, in-memory banking simulation. To extend its utility, we propose two primary directions.

A. Fault-Resilient Financial Systems

   To improve robustness in the system, incorporating dynamic initiator election ensures that snapshot routines resume if the original root fails, while fault injection tests validate system behavior under node crashes. Let process join and exit in the middle of

   These enhancements will enable real-world financial platforms to achieve high availability and compliance with stringent recovery-time objectives.

B. Integration with Cloud-Native Storage Services

   By replacing in-memory buffers with durable and scalable services such as Kafka, AWS S3, or Google Cloud Storage, snapshots can persist beyond application lifecycles and facilitate rapid recovery after crashes. Packaging the snapshot coordinator as a Kubernetes Operator would enable automated triggering, scheduling, and collection of checkpoints. This approach reduces operational complexity and supports multi-region deployments with geo-replication.