Whitepaper

Consensus Algorithms in Low Latency Systems

By Sivakumar Kalyanaraman

©LTIMindtree | Privileged and Confidential 2023

Table of Contents

1. Introduction 3

2. Failover and fault tolerance in financial exchanges 4

2.1 Monitoring and failover of HA cluster 5

2.2 Business constraints 7

3. A raft-based model for low-latency systems 8

4. Conclusion 14

5. References 15

6. Author 16

©LTIMindtree | Privileged and Confidential 2023

01 Introduction

As per the Oxford dictionary, consensus refers to an opinion that all group members agree with. A common example of a consensus issue would be lunch with office peers. You decide a time and place, but one friend has a last-minute meeting, another is caught up in a task, and another maybe 15 minutes late due to a crisis. Since most of your friends are busy and don't agree, your plan gets canceled, and you plan again.

Distributed systems work this way too. A group of servers must agree on some state for a given data. Clustering and failover are parts of distributed computing used to achieve fault tolerance and performance. When processing is done as a finite state machine within the processing node (with memory state updates involved), the input is replicated to the primary and secondary nodes. This way, the state is maintained across the replicas, and failover does not cause data inconsistency. Some critical steps in such replicated state machines are - the replication guarantee, failure identification, and decision on the next node that needs to take over the primary role in case of failure. Some basic ways to achieve these include synchronous replication (the primary sends the input to all nodes, waits for acknowledgment, and then processes the message), using a separate monitor server that listens to the heartbeat from the primary and contains the logic for deciding subsequent primary in case of failure. Such implementations significantly reduce the scalability of the replication cluster and make the monitor node complex to implement. Algorithms such as Paxos and Raft have been published to handle replication and failover scenarios for distributed clusters with unreliable nodes.

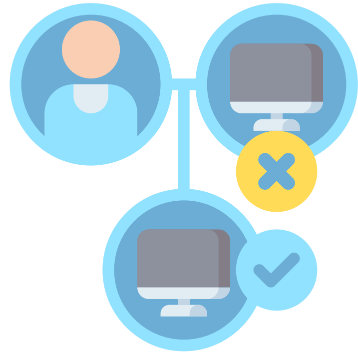
Paxos and Raft are consensus algorithms that allow the nodes in a distributed system to agree on a given state or input sequence. Paxos is older, complex, abstract, and does not provide many implementation details. It only considers the problem of achieving a common value for a single shared state. An input replication technique must be built over Paxos, testing new API and semantics for all cases. Raft is a simpler alternative with implementation details specified for all the operations – it specifies the various Remote Procedure Calls (RPCs) and the behavior of each call. Moreover, Raft addresses the implementation-level problem by consistently replicating input from the primary to the secondary nodes. Thus, we see various open-source projects using Raft.

In this document, we discuss some changes to the basic Raft model to make it more suitable for applications like trading systems, where latency is as much or more critical as fault tolerance. For systems where the latency impact to replication is acceptable, this approach cannot be used when weighed against the degree of fault tolerance needed.

02 Failover and fault tolerance in financial exchanges

Failover is the ability to switch automatically to a reliable backup system. This mechanism works on computer servers that collaborate to ensure High Availability (HA) or Continuous Availability (CA) for server applications. The clusters that guarantee CA are also known as Fault Tolerant (FT) clusters because they allow users to work on applications without witnessing any outages due to a server crash. If one server goes down, another cluster node can handle its workload without interruptions.

In FT clusters, monitoring/heartbeat messages are used for heartbeat/health monitoring of the cluster nodes. The absence of such heartbeats or any inconsistency in the monitoring data sent out by a node can be used to determine if the node is down (either component failure or data inconsistency due to which the node is unusable).
In financial exchanges, FT is one of the critical features that must be implemented at all the layers. Critical components like the trading engine maintain a significant internal state (order book, trades, etc.), and to implement fault tolerance of such components, input replication is used to ensure the same input is processed by the primary node and all secondary nodes of the cluster. Input replication ensures that the primary and secondary nodes are identical and in sync. Another replication method is output replication, wherein the output from the primary is sent to the secondaries so that they can rebuild the internal state based on the output. But here, the secondaries will have a different execution path, and hence is more complex to implement and debug. Hence, log replication in raft consensus algorithms and leader election (identifying the next primary in case a primary fails) are critical for trading applications.



2.1 Monitoring and failover of HA cluster

Monitoring and failover in such a setup can be done in several ways, as described below.

Using an observer/controller node

In this design, an observer/controller node is used to listen to the heartbeat/health monitor messages from all the nodes. In case of a missing heartbeat/incorrect health monitor messages, the controller node acts by marking the node as down or initiating failover if the primary node is identified as down/inconsistent. In this model, the controller node should run on fault-tolerant hardware to ensure it does not go down. This node becomes a single point of failure for the entire system. Also, the controller system software must be simple, and the Finite State Machine must be validated for all valid and invalid inputs. A variation of this design is where all the nodes monitor the heartbeat/health monitor messages, but all discrepancies are reported to the controller node for action.

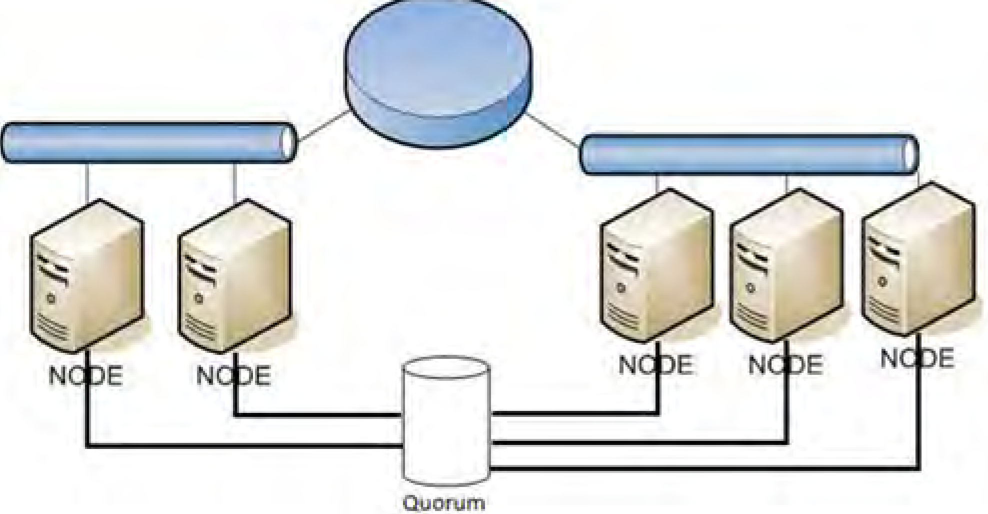


Fig 1: Observer-Controller: Windows Clusters and Quorum configurations explained, JeffOps-The Scripting Dutchman, December 22, 2011: https://jeffwouters.nl/index.-/2011/12/windows-clusters-and-quorum-configurations-explained/

©LTIMindtree | Privileged and Confidential 2023

Homogeneous cluster with any node being the leader

In this design, any of the nodes in the cluster can be the leader (master node), and other nodes are the followers (secondary or slave node). All the nodes process heartbeat and monitoring information. If a heartbeat is missed or an information mismatch, the nodes together determine the action to take by exchanging control messages. There is no separate control node.

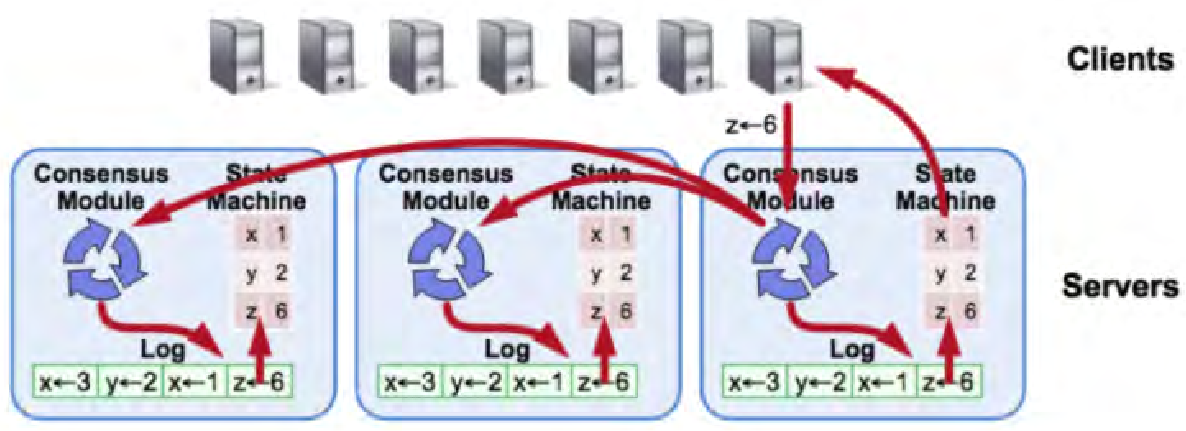


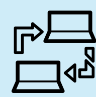
Fig 2: Raft consensus algorithm, YugabyteDB :
https://www.yugabyte.com/tech/raft-consensus-algorithm/

In this scenario, most of the alive nodes must arrive at a consensus on the change in system state (node down or node faulty) and the action to be taken (who is the new master). We can use consensus algorithms such as Raft for state change confirmation and new leader election.

2.2 Business constraints

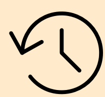
Adopting either of these failover models depends on the business constraints the failover solution must satisfy. These business constraints are as critical as the fault tolerance requirements in financial exchanges. These constraints are:

Multi-node failure



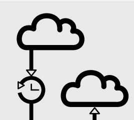
In case of multi-node failure, the ratio number of nodes failed to the total number of nodes in the cluster determines the impact on business. If the ratio is < 0.5, the system will still be fault tolerant. If the ratio is > 0.5, some consensus algorithms may not work (their preconditions would not be met). Hence the system may lose its fault tolerance features. This impacts the business and requires manual intervention to restore fault tolerance.

Time to failover



The time to failover is from when a node failure is confirmed and a new primary is identified to when the new node takes over as primary and is fully connected to other subsystems and servicing requests. This will include any state synchronization to be done among the nodes, state changes in all the cluster nodes, notification acknowledgments, and synchronization needed with other subsystems. This time must be minimal so external users do not feel a high latency during failover. The state transition implementations and the amount of synchronization needed should be kept to a minimum to optimize this time.

Impact on latency



The decision on a node confirmed to be down and designating a new node as the primary could take time, depending on the approach. During this time, the system will not be responding to any external requests and hence will affect the system latency. The approach (observer-based vs. identical nodes) and the algorithms for failure detection and master identification are critical to ensure minimal latency. Moreover, in the case of the homogeneous nodes approach, the heartbeat/health message processing should incur minimal or no overhead to the regular processing; otherwise, it could cause a significant latency overhead even when all nodes are healthy.

03 A raft-based model for low-latency systems

As discussed in the previous sections, log replication and synchronizing inputs across replicated nodes help achieve fault tolerance against hardware faults. Raft handles log replication, node failures, and new leader elections, serving as a protocol for input replication in a distributed system. One of Raft's key tenets/requirements is that the master node begins processing the input message from the replicated log only when the log entry is committed. A committed log entry is one for which most followers have acknowledged receipt of the log entry.

Given the load and throughput requirements, low-latency systems used in the financial exchanges domain have latencies in milliseconds or microseconds range. In such systems, the leader waiting for most followers to acknowledge receipt of a message before processing the message could cause a significant increase in message latency. For example, let us consider a system with a latency of 25 microseconds. Considering the leader sends the AppendEntries RPC and immediately gets a response from all the followers, the network's send + receive time would be three microseconds. This may be seen as just a 12% increase in the latency, but simulating the three micro-seconds extra latency due to replication for a system processing 100K messages per second shows that the average response time more than doubles.

Scenario	Average response time	Max response time
100K messages, 5 processing steps	533	566
100K messages, 5 processing steps + 3 microseconds for replication of messages	1196	1256

This indicates a delay incurred before the message reaches the state machine can significantly impact the overall system latency.

The alternative we propose is for the replication checkk after message processing by the state machine is completed. This goes against one of the key tenets of Raft; however, it will help to avoid the latency increase due to replication. This is a feasible approach with network architectures that guarantee higher reliability and commodity servers having higher uptime.
We propose changes in Raft's RPC semantics for log replication and matching in our approach.
The following table summarizes our proposed changes:

Impact area	Raft behavior	Proposed change
Pass Input to the state machine	Input is sent to the state machine only after log replication is completed, i.e., replication Acknowledgement (Ack) is received from n+1 followers in a cluster of 2n+1 nodes	Input is sent to the state machine in parallel to log replication. Replication completion is verified before the processed output is sent out.
Followers having inconsistent log entries with the leader – different entries at a given Log Index	The leader flushes the log at the follower until the logs are in sync, and then all subsequent entries are replicated from leader to follower	The only recovery option is to start from a clean state and replicate the leader's log from the beginning. Entries present in the follower's log cannot be deleted, as they have already been sent to the state machine.
Waiting for ACK from n+1 followers (assuming cluster of 2n+1 nodes)	If Ack is not received from n+1 followers, the leader pauses. The system is halted until at least n+1 nodes are back in the cluster	None

Impact area	Raft behavior	Proposed change
logIndex variable	The commitIndex is the log entry that the leader has processed, and all entries up to this are expected to be in sync, in all the active current nodes	This is also the log entry that the given node has processed, and it is expected that the logIndex of the leader will always be >= logIndex of all active current nodes
Log sync with the followers	Log sync is guaranteed by a consistency check performed by AppendEntries RPC. If the consistency check fails, the leader decrements nextIndex and retries to match the index and term of the entry. Once a match is found, conflicting entries in the follower's log are deleted	Log sync is guaranteed by a consistency check performed by AppendEntries RPC. In case of consistency check failure, the follower responds with the Index up to which it has the data. The leader now updates this and sends the log from the next Index to that follower. In case the leader's term and the follower's are different, the follower must request a log sync from the beginning of the leader's log

Table 1: Summary of proposed changes

The following tables illustrate the change in AppendEntries RPC.

AppendEntries RPC

Invoked by leader to replicate log entries, also used as heartbeat

Arguments

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[ ]	log entries to sore (empty for heartbeat)
leaderCommit	leader'scommitIndex

Results

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver Implementation

1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit> commitIndex, set commitIndex= min(leaderCommit, index of last new entry)

Fig 3: Original Implementation

AppendEntries RPC

Invoked by leader to replicate log entries, also used as heartbeat

Arguments

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries [ ]	log entries to store (empty for heartbeat;)
leaderCommit	leader's commitIndex

Results

term	currentTerm, for leader to update itself
success	true if follower contained entry matching
purgeClient	true if follower is purging it's log (leader must now sync the log from beginning)
prevLogIndex prevLogTerm	The term and Index up to which the follower has the Log data

Receiver Implementation

1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), purge the log and set purgeClient to true. Return false
4. Append any new entries not already in the log
5. Send the Log Entries to the Finite State Machine for processing

Fig 4: Modified Implementation

The below illustration describes the possible cluster of order matching engines (used in the trading domain), with the modified Raft consensus algorithm used for the replication and leader election. Only the leader communicates external systems. The first diagram shows the replication flow with the Raft protocol, and the second shows the proposed modified flow. Some basic characteristics of order matching are:

1. All the processes are single-threaded, and the order messages are processed sequentially.
2. The processor Finite State Machine (FSM) mentioned below is responsible for the actual order processing and updating of the order book (maintaining the internal state of the node). The state is accessible only to this process.
3. If the processor encounters a fatal error during processing, it shuts down, marking that node as down (in case of this being the leader, this will trigger the leader election).
4. While the followers receive the messages from the leader and process them (and update their internal state), the output handlers act as message sinks and do not send any outputs to the other connected systems. Only the output handler of the leader sends out messages.

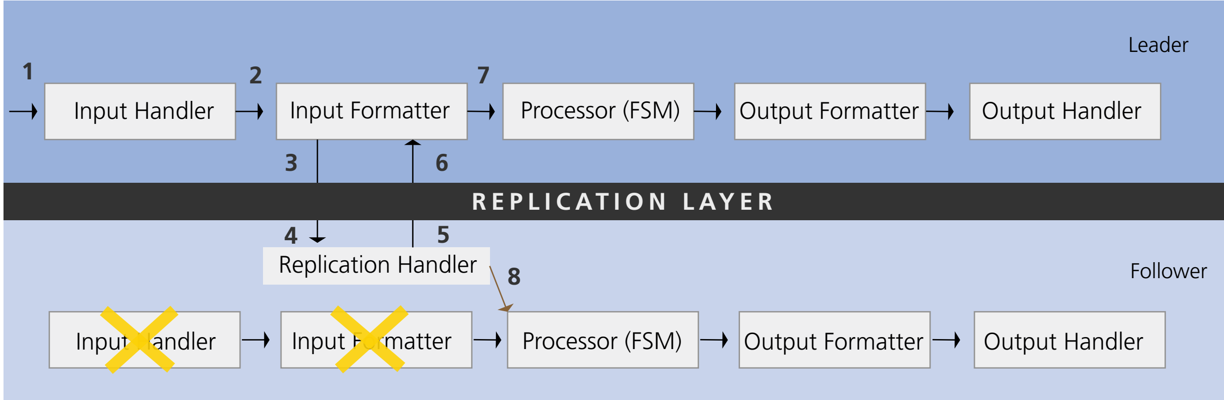


Fig 5: Replication with regular Raft flow

The flow of the steps is as follows:

1. Message received from the client
2. Input handler and formatter format the message
3. Raft is used to replicate the message to the follower(s). The log is sent to the follower, and the leader waits for Ack before proceeding
4. Now the message is sent to the processor for actual processing
5. The message is also sent to the processor at the follower
6. After processing the message, the response is sent to the client

The following diagram depicts the modifications in the Raft protocol:

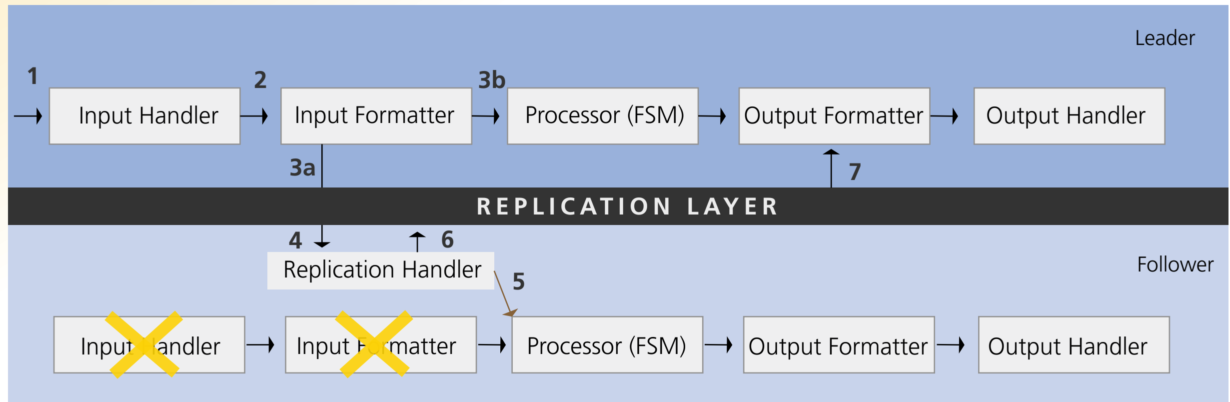


Fig 6: Replication with modified Raft flow

The flow of the steps is as follows:

1. Message received from the client.
2. Input handler and formatter format the message.
3. Raft is used to replicate the message to the follower(s). The log is sent to the follower.
4. Now the message is sent to the processor for actual processing.
5. The message is also sent for processing at the follower.
6. Follower acknowledges the message.
7. After processing the message, the leader waits for Ack from followers.
8. Thereafter the response is sent to the client.

The leader does not wait for the network hop of followers to receive the log and send acknowledgment before processing the message. It processes the message and then checks for acknowledgment. Since these two operations happen in parallel, there is no impact on the latency during normal operations. If n+1 nodes do not acknowledge, the leader will wait for the acknowledgment, as in the current case of Raft.

We believe such tweaks to Raft will increase its adoption and allow enterprises to use proven algorithms for fault tolerance capabilities against building in-house solutions. In-house solutions need to cater to multiple state transition scenarios requiring significant testing before they can be deployed.

04 Conclusion

Input replication and leader election are critical for achieving fault tolerance in distributed systems. While these could be achieved using custom implementations, protocols like Paxos and Raft are designed for the same. Raft is a more complete and detailed protocol that can be used to arrive at a usable implementation with limited ambiguities. Various open-source projects use variations of Raft, as shown above.

The simulation above shows that input log replication and sending confirmations could significantly increase the latency. In systems where low/ultra-low latency is a crucial Non-Functional Requirements (NFR) along with fault tolerance, our proposed change to the Raft protocol will guarantee fault tolerance without increasing the latency.

With better hardware and network infrastructure, hardware failures (such as N nodes going down in a cluster of 2N + 1 nodes or only one node receiving the replicated messages) are expected to happen less frequently. We believe that the leader can ensure replication completeness after processing the message. The network latency for message sending and receiving an ack will be comparable to the processing time for the message, and hence there should be no impact on latency due to replication. If a follower detects log inconsistency and requests a log sync from the beginning, there will be additional latency before it can sync up and be up to date with the leader.

By slightly modifying the appendEntries RPC, a follower can also request the input log from the beginning from the leader. This can be used by followers whenever they see that the log-in leader and follower are not in sync.
Thus, the suggested change will offer an option to use Raft even in low-latency systems. Such an implementation must be validated for state machine correctness for all possible state changes.

With this proposed change to Raft, we believe that it can be more widely adopted in financial exchanges/trading application domains for building fault-tolerant systems. Raft's leader election and log replication logic can be used, thereby ensuring correctness, which is already established for Raft. This will reduce the testing needed for the actual replication and failover (leader election logic) against testing a custom implementation of the complete replication and failover logic.

05 References

- *The Part-Time Parliament, Leslie Lamport, Association for Computing Machinery, May 2, 1998:* http://lamport.azurewebsites.net/pubs/lamport-paxos.pdf?from=https://research.microsoft.com/users/lamport/pubs/lamport-paxos.pdf&type=path
- *Paxos Made Simple, Leslie Lamport, Association for Computing Machinery, December 2001:* https://www.microsoft.com/en-us/research/publication/paxos-made-simple/?from=https://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf&type=exact
- *Paxos Made Live: An Engineering Perspective, Tushar Chandra, Robert Griesemer, Joshua Redstone, Association for Computing Machinery, June 20, 2007:* https://read.seas.harvard.edu/~kohler/class/08w-dsi/chandra07paxos.pdf
- *Paxos Lecture, John Ousterhout and Diego Ongaro, Stanford University, August 15, 2013:* https://www.youtube.com/watch?v=JEpsBg0AO6o&t=28s
- *Consensus in the Cloud: Paxos Systems Demystified, Ailijiang, Charapko, & Demirbas, The 25th International Conference on Computer Communication and Networks (ICCCN), 2016:* http://www.cse.buffalo.edu/~demirbas/publications/cloudConsensus.pdf%22%20/t%20%22_blank
- *Raft: In Search of an Understandable Consensus Algorithm, Ongaro, Diego, & Ousterhout, USENIX Annual Technical Conference (USENIX ATC 14), 2014:* https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14%22%20/t%20%22_blank
- *The Secret Lives of Data – Visual explanation of Raft :* http://thesecretlivesofdata.com/raft/
- *ARC: Analysis of Raft Consensus, Heidi Howard, Technical Report UCAM-CL-TR-857, July 2014:* https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf
- *Official overview website of materials on and implementations of Raft:* https://raft.github.io/%22 /t %22_blank
- *Raft Lecture, John Ousterhout and Diego Ongaro, Stanford University, August 15, 2013:* https://www.youtube.com/watch?v=YbZ3zDzDnrw
- *The Raft Consensus Protocol, Prof. Smruti R. Sarangi, IIT Delhi, April 2020 :* https://www.youtube.com/watch?v=oBlbpPDxs-M&t=2s

06 Author

Sivakumar Kalyanaraman

Head – Performance Engineering Group, Global Technology Office

Sivakumar Kalyanaraman leads the High-Performance Engineering unit within the Global Technology Office at LTIMindtree. He has worked on various high-performance, low-latency computing projects and spearheaded performance/reliability engineering of critical business systems.

©LTIMindtree | Privileged and Confidential 2023

LTIMindtree is a global technology consulting and digital solutions company that enables enterprises across industries to reimagine business models, accelerate innovation, and maximize growth by harnessing digital technologies. As a digital transformation partner to more than 700 clients, LTIMindtree brings extensive domain and technology expertise to help drive superior competitive differentiation, customer experiences, and business outcomes in a converging world. Powered by 82,000+ talented and entrepreneurial professionals across more than 30 countries, LTIMindtree — a Larsen & Toubro Group company — combines the industry-acclaimed strengths of erstwhile Larsen and Toubro Infotech and Mindtree in solving the most complex business challenges and delivering transformation at scale. **For more information, please visit https://www.ltimindtree.com/**