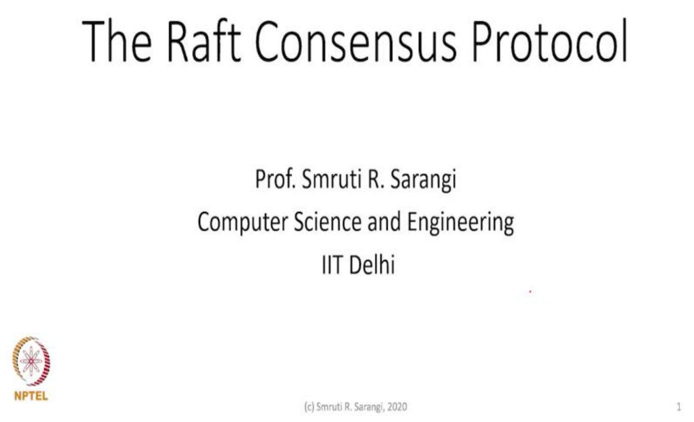
Advanced Distributed Systems
Professor. Smruti R Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture No. 13
The raft consensus protocol

(Refer Slide Time: 0:17)



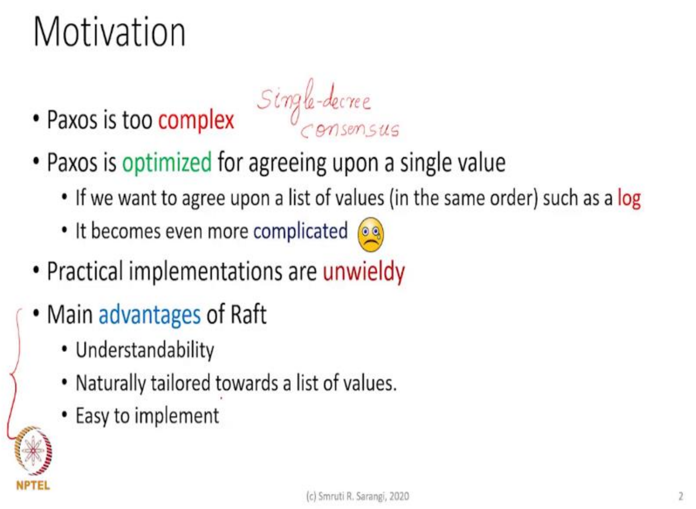
The Raft Consensus Protocol

Prof. Smruti R. Sarangi
Computer Science and Engineering
IIT Delhi

(c) Smruti R. Sarangi, 2020 1

In this lecture, we will discuss the raft consensus protocol which is different from the paxos protocol. So, it is different in many senses. So, paxos is a far more complex protocol. So, raft is substantially simpler.

(Refer Slide Time: 0:40)



Motivation

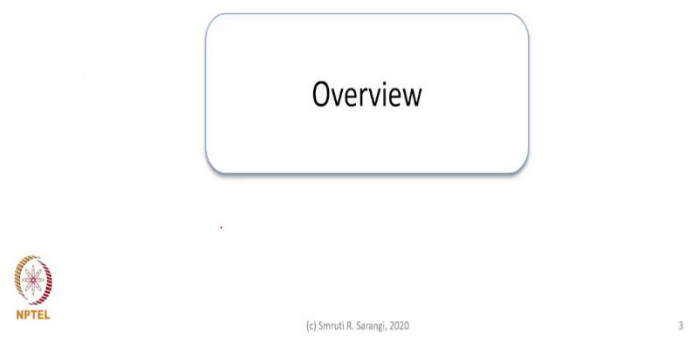
- Paxos is too complex Single-decree consensus
- Paxos is optimized for agreeing upon a single value
  - If we want to agree upon a list of values (in the same order) such as a log
  - It becomes even more complicated
- Practical implementations are unwieldy
- Main advantages of Raft
  - Understandability
  - Naturally tailored towards a list of values.
  - Easy to implement

(c) Smruti R. Sarangi, 2020 2

So, let us look at the main features of raft. So, the primary motivation of creating raft was that paxos is perceived to be too complex and. So, that is the reason when paxos is taught typically a simple version of the protocol is taught, which is called a single degree consensus, which essentially means we are agreeing on a single value not on a list of values. The problem is that most of the time we do not want to agree on a single value but rather we want to agree on a series of values such that in any distributed system different nodes will apply the same set of values same set of changes one after the other on the state machines.

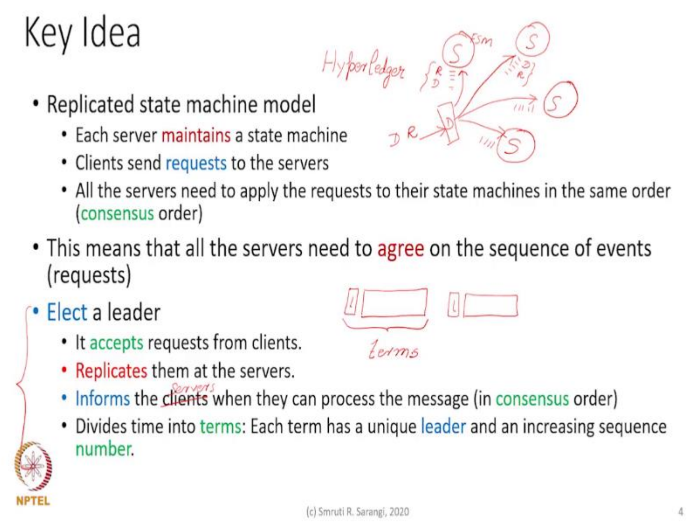
So, this is primarily, so paxos can do it. So, the full version of the paxos protocol can do it but since it is complex a simpler protocol was devised which is easy to understand debug and verify. The main advantages of the raft consensus protocol are understandability, it is simple. It is naturally tailored towards a list of values not just a single value and it is easy to implement.

(Refer Slide Time: 2:11)



Overview

(c) Smruti R. Sarangi, 2020 3



Key Idea

- Replicated state machine model
  - Each server maintains a state machine
  - Clients send requests to the servers
  - All the servers need to apply the requests to their state machines in the same order (consensus order)
- This means that all the servers need to agree on the sequence of events (requests)
- Elect a leader
  - It accepts requests from clients.
  - Replicates them at the servers.
  - Informs the ~~clients~~ servers when they can process the message (in consensus order)
  - Divides time into terms: Each term has a unique leader and an increasing sequence number.

(c) Smruti R. Sarangi, 2020 4

The overview, so the key idea here is that we have a replicated state machine model which means that each server maintains a state machine. So, we essentially have multiple servers and so let us say anything like providing a web page or checking email. So, instead of one email server we have multiple email servers each of them has exactly the same state machine the same finite state machine. So, clients send a request to the servers, so what a dispatcher does, a dispatcher is that it replicates the same so the same request is replicated to all the servers all of them apply the clients request so the client command to the state machines.

And so since we can have a series of commands from different clients there is a need to create a list of commands that are seen by all the servers and there needs to be complete agreement a complete consensus on this list such that they can be applied in the same order otherwise we can have interesting violations in consistency.

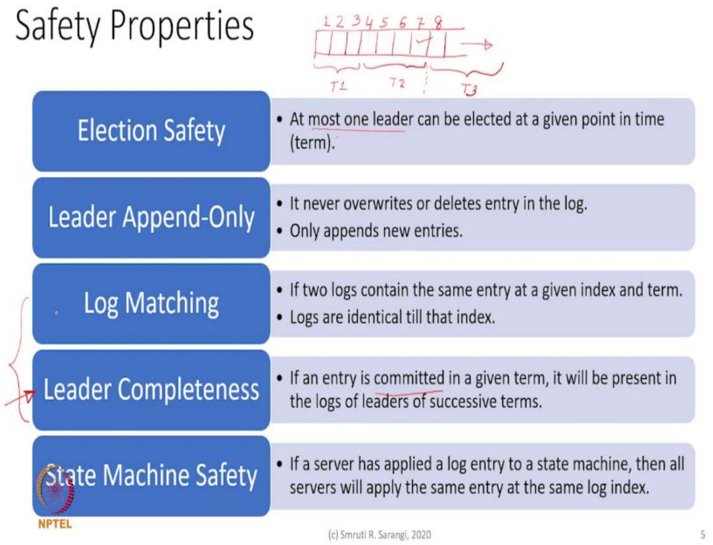
So, let us say for example there are two commands; the first command is to read an email the second is to delete an email. So, if let us say the read comes here first and then the delete it is fine we first read the contents and then we delete the email but it is possible that another server unless we do something might say delete first and then the read that is an issue. Because in this case the mail will be deleted first and the read will fail and this is not something that we want hence we want all the servers to see the same set of commands in the same order.

So, this of course is a consensus problem, and this is a multiple degree instead of a single degree it is like a list consensus, and whether multiple orders that are given out, and this of course is hard to do in paxos. So, we that is the reason we would typically use raft and we will also find when we study blockchain and bitcoin and all of these technologies that raft is one of the key protocols behind IBMs.

Well, IBM led hyper ledger fabric which is a very, very common blockchain that is used but we will have a later lecture on blockchains, where we will discuss this in great detail but the important point that needs to be kept in mind is that raft is one of the key foundational technologies of hyper ledger like blockchain systems. So, the idea here is that we first elect a leader, the leader then accepts the requests from the clients, it replicates them at the servers as we just saw, then it informs the clients when they can process the message in consensus order.

So, I am sorry it will inform not the clients the servers when they can process the all the messages in consensus order and such that all of them see the same set of messages in the same order. So, what we do is that so, we divide time as follows we have a leader, then the leader maintains his leadership for a period of time, then if the leader crashes we elect a new leader and again the leader does its role. So, time is thus divided into what we call terms.

(Refer Slide Time: 6:14)



Safety Properties

Election Safety	• At most one leader can be elected at a given point in time (term).
Leader Append-Only	• It never overwrites or deletes entry in the log. • Only appends new entries.
Log Matching	• If two logs contain the same entry at a given index and term. • Logs are identical till that index.
Leader Completeness	• If an entry is committed in a given term, it will be present in the logs of leaders of successive terms.
State Machine Safety	• If a server has applied a log entry to a state machine, then all servers will apply the same entry at the same log index.

(c) Smruti R. Sarangi, 2020 5

Let us now look at some of the key safety properties that we need to ensure in the raft algorithm. So, the first is election safety which means that at any point in time we have at the most one leader or in other words what it means is that in each term we will have at the most one leader, it will never be the case that we have two simultaneous leaders, two leaders at the same time.

The second is, so this is a standard approach in distributed systems that we have an append only log. So, log is treated as something like a array that just keeps growing. So, each of the log entries has an index which is just an increasing sequence of numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and then we have a set of terms. So, pretty much every entry that was added in a given term has that term ID.

So, we can say that the first three entries are in term 1, the second set of four entries are in term 2 and so on. So, the leader particularly never overwrites or deletes any entry in the log it only appends new entries. So, the log only keeps increasing this way entries are never deleted or they are never overwritten by the leader. So, this is not true for followers but at least for the leader this is the case.

Then we will come at two of the most important properties that are log matching and leader completeness. So, log matching basically means that if we consider two logs, so the leader maintains a log all the followers maintain a log everybody maintains a log. So, we want that till some point the logs are exactly identical. So, this is the consensus that is a part of raft.

So, the point to which the logs are identical, so let us say that till 0.7 the logs are identical, then what we can do is the moment we are sure of that all of these entries can be passed on or applied to the state machines. So, what the log matching property says that if two logs contain the same entry at a given index and term.

So, let us say index the entry with index 7 it should have the term T2 and the same contents if that is the case then the logs have to be identical till that index which means that if this index is the same the seventh index is the same even all the entries before it which means all the entries in the indexes 1 to 6 they also need to be the same. So, this is the log matching property and we will see that the algorithm explicitly ensures this.

The other important property which we actually need to prove at the end is the leader completeness property which means that if an entry is committed in a given term. So, let me just explain what it means to commit an entry. So, to commit an entry basically means to ensure that the entire log till that entry. So, let us say we want to commit entry number 7, all the entries still 7 have been successfully replicated across all the servers or a majority of the servers.

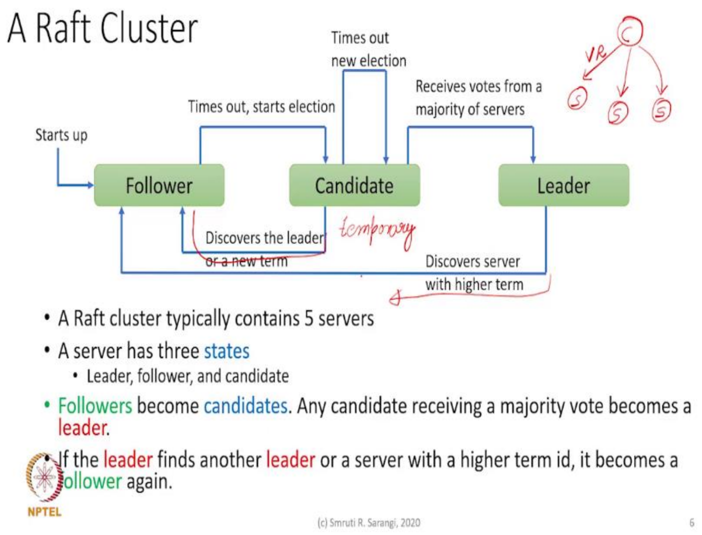
So, at that point we explicitly commit the entry which basically means we tell all the servers that entries from 1 to 7 can be shown to their state machine can be applied to the state machine, this process is called committing. So, if an entry is submitted in a given term it will be present in the logs of leaders of successive terms which means that if a new term begins a current leader crashes, and the new leader is elected, and a new term begins it need not have the uncommitted entries of the previous leader but it definitely has to have the committed entries of all the previous leaders.

So, this ensures that the log grows in a certain way and any change that is applied to a finite state machine that is correctly applied. Otherwise, what would happen if you do not have this property, that the leader of T2 would apply an entry and then T3 will not find it. So, then from the leader 3 who owns the term T3 from his point of view this entry should not have been applied to the finite state machine and that should never be the case.

So, that would cause a correctness issue hence the leader completeness property is required. So, then we come to state machine safety which kind of follows from the rest. So, if a server has applied a log entry to a state machine, then all servers will apply the same entry at the same log index. So, this also follows from the notion of the way we create the logs and the way we create and the leader completeness property and the way we commit the entries.

So, the state machine safety property naturally follows from the first four, so we will not prove this separately. We will essentially look at the log matching and the leader completeness properties because they are sufficient to ensure the rest given these assumptions in the rest of the property.

(Refer Slide Time: 12:36)



A Raft Cluster

Times out new election
Times out, starts election
Receives votes from a majority of servers
Starts up
Follower
Candidate
Leader
Discovers the leader or a new term
temporary
Discovers server with higher term
VR
C
S
S
S

- A Raft cluster typically contains 5 servers
- A server has three states
  - Leader, follower, and candidate
- Followers become candidates. Any candidate receiving a majority vote becomes a leader.
- If the leader finds another leader or a server with a higher term id, it becomes a follower again.

(c) Smruti R. Sarangi, 2020 6

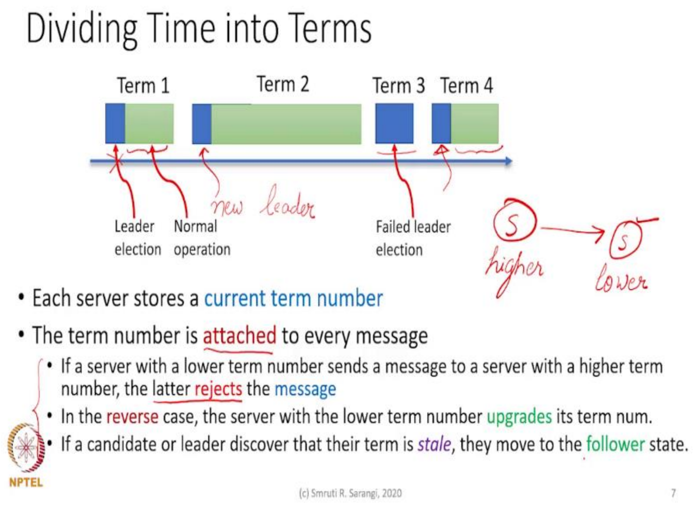
So, this is the way that a raft cluster looks like we have all the servers start in the follower state, then in the follower state once the servers timeout they start an election. So, every time an election is started we increment the term number. So, then the servers become candidates and they request for votes from the rest of the servers. So, what essentially happens is that one server which is a candidate it will send a vote request a message to the rest of the servers.

So, if it gets votes from a majority of servers it is sure that it can be the leader otherwise what happens if there is a split vote, which means nobody is able to garner a majority in that case we increment the term number and start a new election. When the server is a candidate if it discovers that a leader has already been elected which means the term has been incremented and there is a new leader.

So, it relegates itself to the follower state which is essentially this path and it is further more possible that a leader actually crashes and wakes up a long time later just to find that the term has been incremented and a new leader has been elected. So, in this case also the old leader realizes it is not the leader anymore. So, it goes back to being a follower. So, a raft cluster does has the typically has 5 servers or more not less and so the followers become candidates. So, candidate is a temporary state, so follower and leader these are stable states but the candidate state is a temporary state.

And so followers become candidates still a leader is elected then they become followers once again. If the leader finds another leader of course at the higher term ID as we discussed over here it becomes a follower once again.

(Refer Slide Time: 15:11)



Dividing Time into Terms

Term 1
Term 2
Term 3
Term 4
new leader
Leader election
Normal operation
Failed leader election
S
higher
S
lower

- Each server stores a current term number
- The term number is attached to every message
  - If a server with a lower term number sends a message to a server with a higher term number, the latter rejects the message
  - In the reverse case, the server with the lower term number upgrades its term num.
  - If a candidate or leader discover that their term is *stale*, they move to the follower state.

(c) Smruti R. Sarangi, 2020 7

So, we are dividing time into terms. So, a term begins with a leader election we have a new leader subsequently normal operation commences in the normal operation clients send messages to servers they are replicated our consensus algorithm runs, then it is possible that the leader crashes.

So, what happens is a new leader is elected. Subsequently, normal operation commences and then again we have one more election once the leader for term 2 crashes. So, in term 3 it is possible that no leader could be elected because of split votes. So, in that case we will have a timeout we will increment the term number and we will then elect one more leader.

So, note that there is a probabilistic process it is not guaranteed to terminate and this is one of the cases where, so we know that in an asynchronous system because of the FLP result it is possible that we will never achieve a consensus. So, this means that any protocol any consensus protocol in an asynchronous system has to have a set of events where consensus is not possible and this is one such case where we will simply not be able to elect a leader.

So, now what happens over here is that, so let us assume we have a period of split votes and then we have a new leader and regular execution commences. So, each server stores a current term number and the term number is attached to every message this is required such that everybody knows what is the term and who is the leader. So, it is possible that servers who are either followers or candidates or leaders would crash and recover.

So, they would use a term number to find out about the stillness of messages. So, if a server with a lower term number sends a message to a server with a higher term number then it means that clearly the server with the lower term number is not aware of the changes that have happened when it was in the crash state.

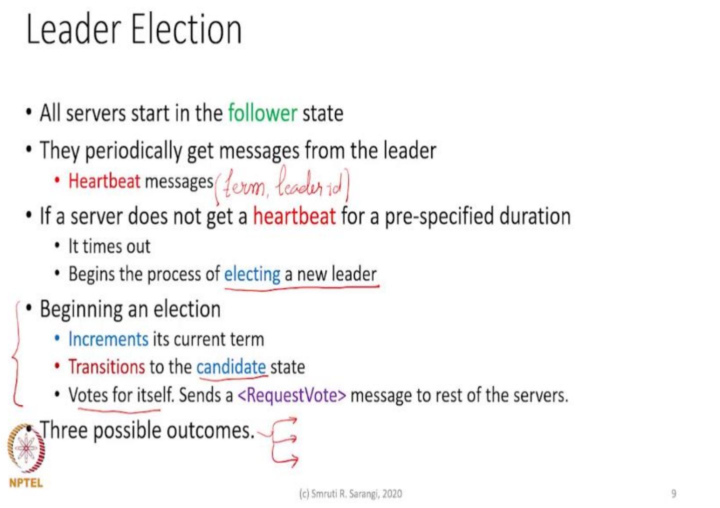
So, in this case the latter which is the server with the higher term number rejects the message drops the message. In the reverse case, if a server with a higher term number. So, let us say a server with a higher term sends a message to a server with a lower term number then the latter which is this server will just upgrade its term.

Furthermore, if a candidate or leader discover that the term is still that their term is still which means that they are getting a message from another server with a higher term number they will realize that maybe they have crashed and then they have recovered or there was some partition in the network something like that has happened. So, they will move to the follower state which we have described in the previous slide.

(Refer Slide Time: 18:42)

Details

(c) Smruti R. Sarangi, 2020 8



Leader Election

- All servers start in the follower state
- They periodically get messages from the leader
  - Heartbeat messages (term, leader id)
- If a server does not get a heartbeat for a pre-specified duration
  - It times out
  - Begins the process of electing a new leader
- Beginning an election
  - Increments its current term
  - Transitions to the candidate state
  - Votes for itself. Sends a <RequestVote> message to rest of the servers.
- Three possible outcomes.

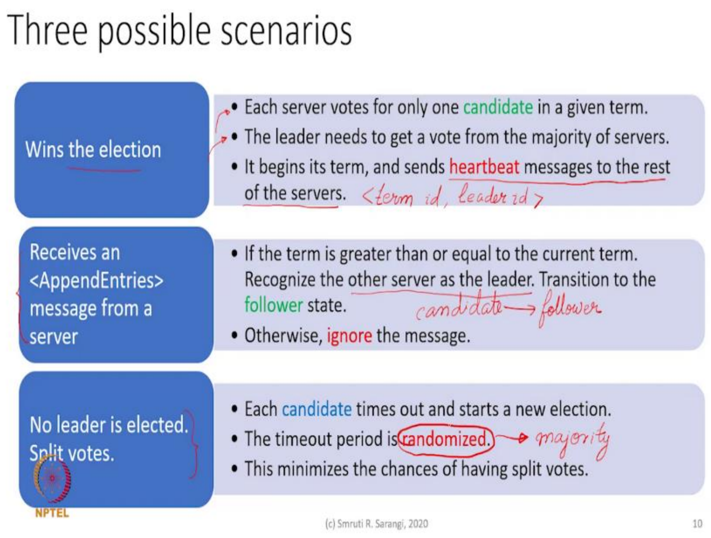
(c) Smruti R. Sarangi, 2020 9

Now let us come to the details. So, all the servers as we just mentioned start in the follower state. They periodically get messages from the leader which are heartbeat messages, each heartbeat message definitely contains the term and the leader ID. So, they always know who the leader is and what is the current term and they use this to figure out if a message is stale or not.

So, if a server does not get a heartbeat for a pre-specified duration this essentially means that maybe the leader has crashed. So, what it does is that it times out and it begins the process of electing a new leader it becomes a candidate itself and begins the process of electing a new leader.

So, now let us see what happens while beginning an election. So, for beginning an election the server increments its current term, so this is required. So, every election be it successful or unsuccessful has to happen with a new term and it transitions to the candidate state. It votes for itself and it sends a request vote message to the rest of the servers.

(Refer Slide Time: 20:23)



Three possible scenarios

Wins the election	• Each server votes for only one candidate in a given term. • The leader needs to get a vote from the majority of servers. • It begins its term, and sends heartbeat messages to the rest of the servers. <term id, leader id>
Receives an <AppendEntries> message from a server	• If the term is greater than or equal to the current term. Recognize the other server as the leader. Transition to the follower state. candidate → follower • Otherwise, ignore the message.
No leader is elected. Split votes.	• Each candidate times out and starts a new election. • The timeout period is randomized. → majority • This minimizes the chances of having split votes.

(c) Smruti R. Sarangi, 2020 10

Let us look at the three possible outcomes and see what will happen in each of these cases. So, let us say it wins the election. So, if it wins the election that is great. So, which means that each server majority of servers vote for it. So, kindly know that in an election each server votes for only one candidate in a given term, so that is important. And so, it basically means that in a given term the server will only vote for a single candidate and so the candidate is also not allowed to vote for, so, a double voting is not allowed and that also includes the candidate voting for itself that also is not allowed in a given term.

The leader especially needs to get a vote from a majority of servers, so this is required and this is where we can have a never-ending cycle. So, then the leader begins its term and sends heartbeat messages to the rest of the servers. So, with the heartbeat message what do the rest of us rest of the servers get they definitely get the term ID the new term ID and the leader ID now let us look at the second scenario.

So, let us say that it either did not participate in the election or it did not win but after that it either gets a heartbeat message or a regular append entries append entries to the log message from a server say the term is greater than or equal to the current term it means that the new server is the leader.

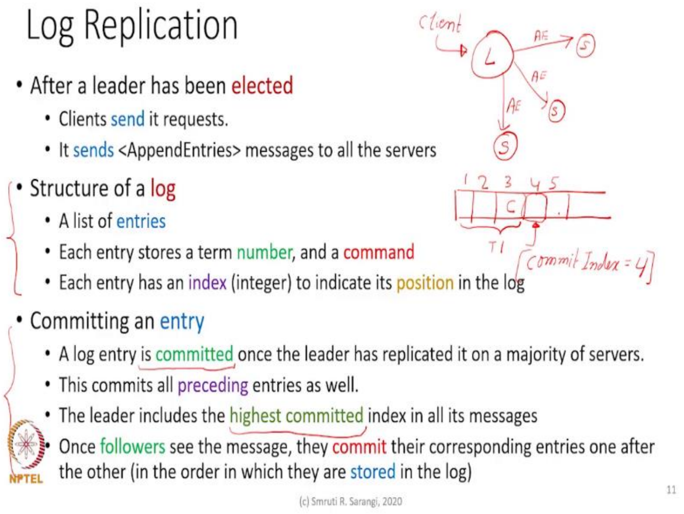
So, we recognize the other server as the leader and the candidates will transition to the follower state, which we have been describing that just in case it is not able to win or before sometime in the middle of the election it gets a message with a higher term number, it should convince itself that it has lost the election and so it will just recognize the new leader and transition to the followers state.

If that is not the case, it is a stale message, we just ignore the message. So, let us consider this case where no leader is elected we have split votes. So, what happens is that in this case the candidate will time out mainly because it is not getting enough votes and it will start a new election a new round of elections. So, one way, so we can always argue that look all the candidates start, then they do not get enough votes then the time out again they start again they do not get enough votes so on and so forth. So, it by this process we never achieve a leader, we never nobody ever wins selection.

So, to solve that the nodes have a randomized timeout which means that for random periods the nodes kind of sleep in do not try to elect themselves leaders. So, because of this randomized time out what happens is that we are trying to minimize the contention trying to minimize the overlaps of when the leader like elections will happen.

So, if the randomized timeout is see if this is a function that can kind of separate the leader election periods it is possible that we will have only a few servers or maybe just one or two that are vying to be the leader at a given point in time. So, we can then assure them a majority probabilistically of course. So, this kind of minimizes the chances of having split votes.

(Refer Slide Time: 24:37)



Log Replication

- After a leader has been elected
  - Clients send it requests.
  - It sends <AppendEntries> messages to all the servers
- Structure of a log
  - A list of entries
  - Each entry stores a term number, and a command
  - Each entry has an index (integer) to indicate its position in the log
- Committing an entry
  - A log entry is committed once the leader has replicated it on a majority of servers.
  - This commits all preceding entries as well.
  - The leader includes the highest committed index in all its messages
  - Once followers see the message, they commit their corresponding entries one after the other (in the order in which they are stored in the log)

(c) Smruti R. Sarangi, 2020

11

Now let us consider the fact. So, now we have elected a leader we have taken care of the split votes issue. So, let us look at log replication how is it that we achieve consensus in the stored logs. So, after a leader has been elected, the clients send it requests the leader sends append entries messages to the rest of the servers and so these servers, then append the entries. So, we will see in a second how it happens.

So, let us look at the structure of a log. So, the log is maintained by the leader in all the servers it is simple list as we said but each of them has an index and a term. So, this is the index and of course we have a term where multiple contiguous indices will have the same term and then each entry stores a command. So, the important operative parts are the index the term and the command.

Now to commit an entry, a log entry is committed once the leader has replicated it on a majority of servers. So, the majority of servers the entry has to be replicated with the same index and terms in the majority of the servers. So, which means that in this case entry number 4 has to be present with the same command in the majority of the servers that is when we say that the entry is committed and if the entry is committed, this commits all the preceding entries as well.

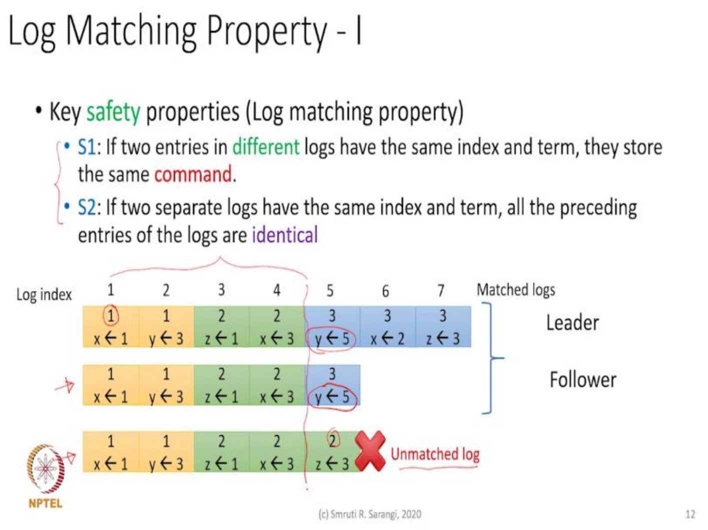
So, all the entries before this are committed. So, if we would see we had the log matching condition that if essentially they match at one point they will match at all the previous points. So, because of log matching we need to ensure that this genuinely happens.

So, the leader then what the leader would do. So, the commit is associated with a state. So, the state in this case if the leader commits entry 4. Well when will it commit entry 4 when entry when the entry with index 4 is present in the logs of a majority of servers with the same index and the same term. So, to commit the leader has a commit index and internal variable.

So, it just sets that to 4 which means that all the entries 1, 2, 3, 4 are committed say any message that the leader sends this will include the highest committed index in the message which means it will include 4. So, any server that gets the message, it will know that the entries 1 to 4 are committed and these entries then can be applied to its finite state machine.

So, once the followers see the message they commit their corresponding entries one after the other in the order in which they are stored in the log. So, of course let us say they have applied the commands at indices 1 and 2 then they will now apply the command at indices 3 and 4 but not 5 that is important not the one at 5.

(Refer Slide Time: 28:44)



Log Matching Property - I

- Key safety properties (Log matching property)
  - S1: If two entries in different logs have the same index and term, they store the same command.
  - S2: If two separate logs have the same index and term, all the preceding entries of the logs are identical

Log index	1	2	3	4	5	6	7	Matched logs
	1 x ← 1	1 y ← 3	2 z ← 1	2 x ← 3	3 y ← 5	3 x ← 2	3 z ← 3	Leader
	1 x ← 1	1 y ← 3	2 z ← 1	2 x ← 3	3 y ← 5			Follower
	1 x ← 1	1 y ← 3	2 z ← 1	2 x ← 3	2 z ← 3			Unmatched log

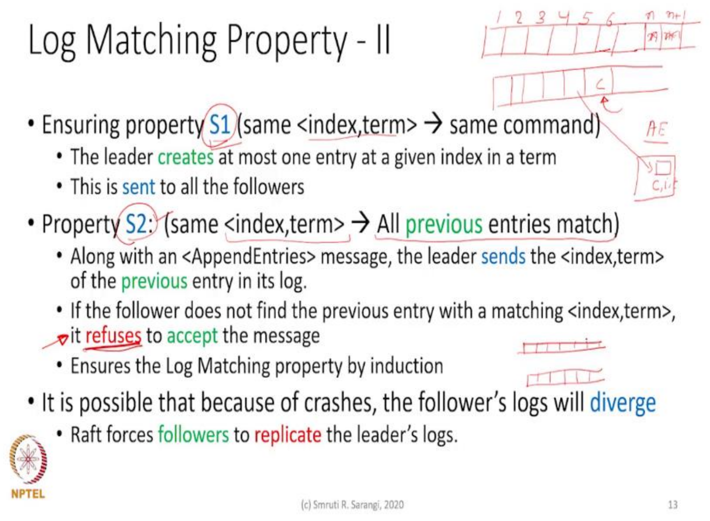
So, let us now look at the log matching property. So, the key safety properties that we have discussed out of them the two important properties are the log matching property and the leader completeness property. So, the log matching property let us break it into two sub conditions S1 and S2. So, if two entries in different logs have the same index and term they store the same command this is the first property. Second if two separate logs have the same index and term all the preceding entries of the logs are identical. So, let us take a look at these three logs.

So, here the color and this entry over here is the term number and the indices are 1, 2, 3, 4, 5, 6, and 7. So, the first row is the log of the leader, the second and third rows are the logs of the followers respectively. So, now what we get to see is that for the leader there are three terms. So, let us first consider the second row which is follower 1.

So, in this case if we consider term 3 index 5, the entries are the same and so for the first property S1 if the index and the term match they have to have the same command which they actually do both are setting y to 5 and by S2 if two separate logs have the same index and term all the preceding entries of the logs are identical, which in this case holds true that all the preceding entries which are entries 1, 2, 3, and 4 are identical.

Now let us consider the log for the second follower. So, in this case the first 4 entries are identical but the fifth entry for index 5 it has a different term number. So, for index 5 the term number should be 3 it is 2, so this entry is unmatched. Given the fact that this entry is unmatched, so we will say that this log is an unmatched log and so since, so it as per the log matching property we can say that this log is not acceptable to us. So, this is an unmatched state whereas all the logs are matching till the first 4 entries.

(Refer Slide Time: 31:32)



Log Matching Property - II

- Ensuring property S1 (same <index,term> → same command)
  - The leader creates at most one entry at a given index in a term
  - This is sent to all the followers
- Property S2: (same <index,term> → All previous entries match)
  - Along with an <AppendEntries> message, the leader sends the <index,term> of the previous entry in its log.
  - If the follower does not find the previous entry with a matching <index,term>, it refuses to accept the message
  - Ensures the Log Matching property by induction
- It is possible that because of crashes, the follower's logs will diverge
  - Raft forces followers to replicate the leader's logs.

(c) Smruti R. Sarangi, 2020 13

So, to ensure property S1 that the same index term tuple leads to the same command, what the leader does is it creates at most one entry at a given index in it a given index in a certain term. So, which basically means that the array is considered a monotonically increasing array and for a given term, so the index keeps on increasing. So, essentially what the leader would do is that if the log is full till let us say the nth index it would add an entry at the n + 1th index and also attach its term number along with it.

And given the fact that this entry is being replicated all the replicated logs would also have the same command at the same index otherwise the log matching property will not hold and we want it to hold and the way that we would actually do that is that the leader will essentially add the entry in its own log and then broadcast that entry in an append entries message with the index and all the followers have to insert it at that index only. So, that ensures property S1.

Next, let us consider property S2 that for the same index term tuple which means that if two logs at the same index are the same term all the previous entries are going to match. So, to ensure this what the leader does is that along with an append entries message the leader sends the index and term of the previous entry in its log. What would that translate to here is that let us say this is the current entry for which the leader is adding a command for the previous entry.

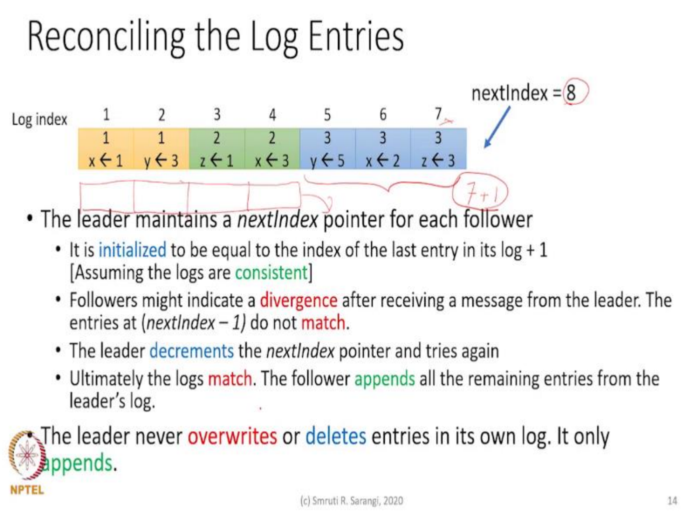
So, for doing this it would create an append entries message and in the append entries message it would essentially add if this is the command it will add this command with the index and the term it will also send the index and the term of the previous entry. So, the advantage of this is that if the follower does not find the previous entry in its log with a matching index term it refuses to accept the message and if it accepts the message it means that the previous entry matches so the current entry will also match. So, this ensures the log matching property essentially by induction that is why because the base case is that the logs are empty.

So, the logs are the same across all the servers and the induction case is that only if the previous entry matches how does it match because the previous entry is sent along with every append entries message only if that matches we add a new entry. Otherwise, the follower refuses to accept the message. So, we will see what happens in this case. So, essentially this refuses this case has to be understood but if it does accept the message then the logs match.

So, essentially our protocol is designed in such a way that properties S1 and S2 which are sub properties of the log matching property they continue to hold if a message is accepted. It is possible that because of crashes the followers logs will diverge, so it is possible that if this is the leader's log and this is the followers log it might match till this point after that there might be a divergence. So, the divergence has to be fixed.

So, we saw one example of the divergence over here which was not an acceptable entry. So, the divergence has to be fixed the way that raft would do it is that it would force followers to replicate the leader’s logs. So, this fixes the divergence. So, the leader is assumed to be always correct and the followers simply replicate the leader’s logs so this kills the diversions.

(Refer Slide Time: 35:58)



Reconciling the Log Entries

Log index	1	2	3	4	5	6	7
	1	1	2	2	3	3	3
	x ← 1	y ← 3	z ← 1	x ← 3	y ← 5	x ← 2	z ← 3

nextIndex = 8
7+1

- The leader maintains a *nextIndex* pointer for each follower
  - It is initialized to be equal to the index of the last entry in its log + 1 [Assuming the logs are consistent]
  - Followers might indicate a divergence after receiving a message from the leader. The entries at (*nextIndex – 1)* do not match.
  - The leader decrements the *nextIndex* pointer and tries again
  - Ultimately the logs match. The follower appends all the remaining entries from the leader’s log.
- The leader never overwrites or deletes entries in its own log. It only appends.

(c) Smruti R. Sarangi, 2020 14

Now let us try to reconcile the log entries. So, consider a log over here with 3 terms and 7 indices. The leader maintains a next index pointer for each follower, what this essentially means that, it basically maintains the degree of how much the logs of a follower match with the leader. So, let me explain. So, the next index pointer is initialized to be equal to the index of the last entry in the leader’s log + 1.

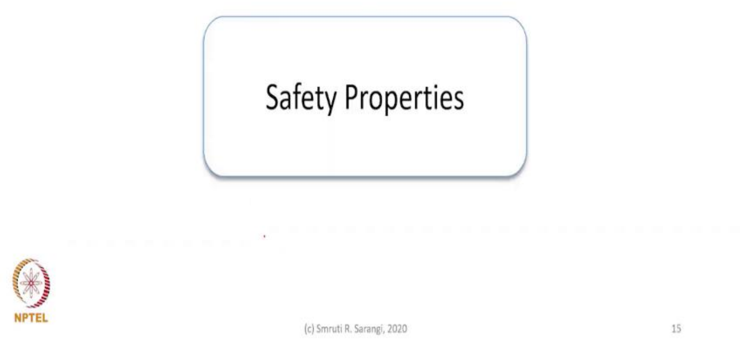
So, if this is the leader's log over here with 7 entries then the last entry + 1 = 8. So, it is assumed by the leader that all the followers have matching logs and the next index at which they will insert an entry is actually index number 8. Now followers might indicate a divergence after receiving a message from the leader.

So, why would that be, well the reason that would be the case when the follower refuses to accept the message from the leader because the previous entry does not match. So, in this case what would happen is that the leader will be aware that a given follower there is a divergence. So, in this case the leader would send entry number 8 but follower would say that look entry number 7 does not match.

So, the leader in this case will decrement the next index pointer and try again. Ultimately the logs will match. So, ultimately the logs will match at some point. So, let us say they will match at this point till 0.4. So, one thing that the leader is sure is that the follower the first 4 entries match after that of course things diverge then when the logs match what the follower will do is that it will append all the remaining entries from the leaders log which are these three entries 5, 6 and 7.

So, this ensures that the follower and the leader will ultimately come in sync but the way that this will happen is that the follower first has to indicate a divergence and the follower has to refuse messages. So, the leader will continue to decrement the next index pointer. It will then wait till the logs match and after that the follower will just copy the rest of the log entries from the leader. So, the leader never overwrites and never deletes entries in its own log, it only continues to append them.

(Refer Slide Time: 38:52)

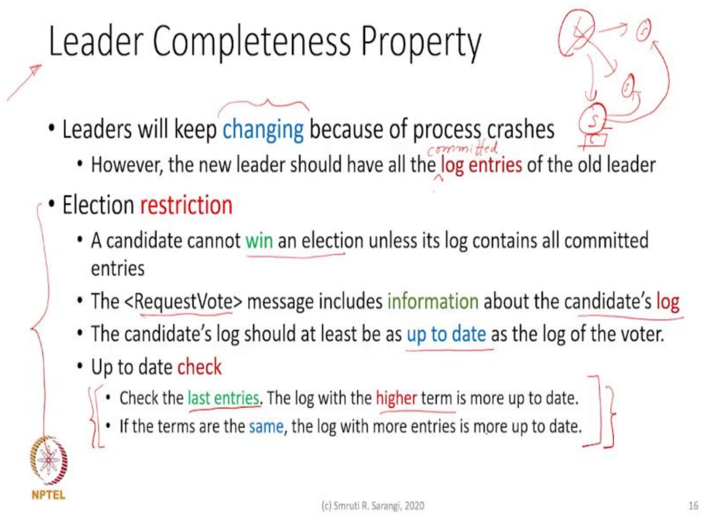


Safety Properties

NPTEL

(c) Smruti R. Sarangi, 2020

15



Leader Completeness Property

- Leaders will keep changing because of process crashes
  - However, the new leader should have all the committed log entries of the old leader
- Election restriction
  - A candidate cannot win an election unless its log contains all committed entries
  - The <RequestVote> message includes information about the candidate's log
  - The candidate's log should at least be as up to date as the log of the voter.
  - Up to date check
    - Check the last entries. The log with the higher term is more up to date.
    - If the terms are the same, the log with more entries is more up to date.

(c) Smruti R. Sarangi, 2020 16

So, given that we have discussed this let us discuss more of the safety properties. So, the next property, so we have already discussed log matching, we have already discussed how the logs are reconciled if we discover a divergence. So, that also we have discussed. So, next let us discuss the leader completeness property. So, leader completeness property is like this that the leaders will keep changing because of process crashes.

So, any kind of a consensus algorithm does take crashes into account. So, the leaders will keep changing. So, however the new leader should have all the log entries of the old leader. So, the new leader whoever is elected will have is expected to have all the log entries of the old leader. Otherwise, all the committed log entries, so, it should be the committed log entries of the old leader, if that is not the case there is a problem.

So, we add an election restriction which is that a candidate cannot win an election unless its log contains all the committed entries. So, what is the idea? The idea is that look the leader is over here the leader keeps on sending messages to each of the servers once the leader is dead one of the servers needs to elected a leader but the server over here can only stand in an election only if it has all the committed entries of the old leader.

So, this is a crucial and critical election restriction that we put to ensure that servers whose entries are not up to date do not stand in the election. So, the request vote message includes information about the candidates log. Particularly, the index of the last committed entry, the index of the entry that was committed the last time. So, the candidate's log should at least be as up to date as the log of the voter.

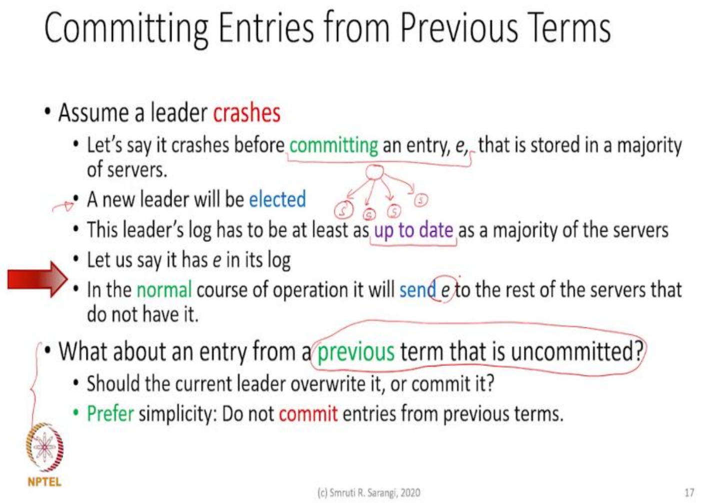
So, how did the leader commit an entry? The leader committed an entry because it was there with the majority of the voters. So, whenever let us say this candidate this server becomes a candidate it sends a request vote message to the rest too they will see if their log is more up to date as compared to this or not. So, the candidate's log which is this is the candidate should at least be as up to date as the voters only then will the election go through.

So, the basic insight of this is that we want to ensure that whatever the leader has committed is present in the logs of the next leader is at least is present in the logs to the next successful candidate and since leaders only come out of candidates we are ensuring the leader completeness property via this.

So, what is the up to date check? Well the up to date check is we check the last entries the log with the higher term is more up to date well that is common sense that the log with the higher term means it has seen more of more leaders come and commit their entries. So, the one with the higher term is clearly more up to date and if the terms are the same the log with more entries is more up to date.

So, we will connect this with the number of committed entries later in the proof but this is our up to date check that what we do is that we check the last entries of the logs and clearly, the winner in this case is the one with a higher term and if the terms are the same, then the log with more entries is more up to date.

(Refer Slide Time: 42:59)



Committing Entries from Previous Terms

- Assume a leader crashes
  - Let's say it crashes before committing an entry, *e*, that is stored in a majority of servers.
  - A new leader will be elected
  - This leader's log has to be at least as up to date as a majority of the servers
  - Let us say it has *e* in its log
  - In the normal course of operation it will send *e* to the rest of the servers that do not have it.
- What about an entry from a previous term that is uncommitted?
  - Should the current leader overwrite it, or commit it?
  - Prefer simplicity: Do not commit entries from previous terms.

NPTEL

(c) Smruti R. Sarangi, 2020

17

We can have several corner cases when it comes to leaders crashing and then recovering. So, let us consider a few such cases there is an example in the paper, so, the viewers should be convinced of the fact that most consensus protocols are very hard to design. Particularly, if the leader crashes then there are many, many race conditions, many, many corner cases which need to be handled.

It is typically not possible to manually verify all of them that is the reason many of these protocols are actually verified by automated checkers, automated verification engines that actually do a process of automated verification and go through hundreds of these corner cases and verify that for each of them the protocol works correctly.

So, let us give a brief overview of what is it that can happen. So, assume a leader crashes. So, let us say it crashes before committing an entry e. So, what committing an entry basically means here. So, this is just not a single phase. So, this is actually a pretty loaded term. So, what the leader does is that it sends an append entries to all the servers only when they accept the message, is it actually sure that they have actually added it to the log and when it gets an acknowledgement or does not get a refusal from a majority of the servers it is sure that the majority of the servers have the entry in their log.

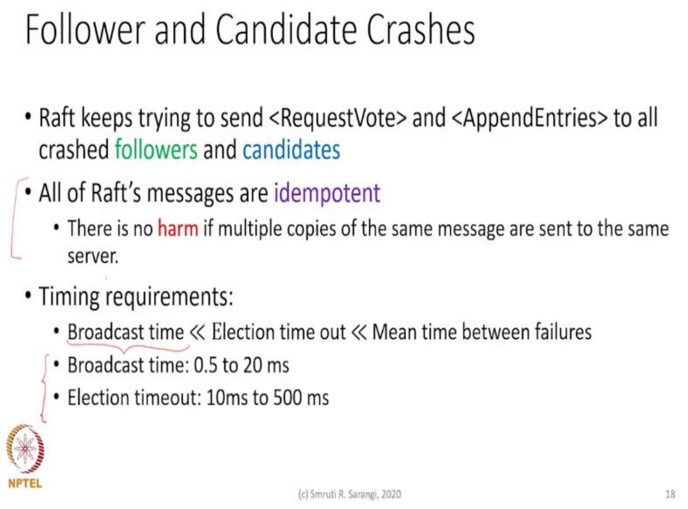
And at that point the leader updates the last commit entry and then the entry is sent and then every subsequent message has the last commit entry has the value of the last commit. So, once the last commit is sent to the server it is also sure that the leader has committed a given entry. So, this process is kind of implicit in raft and it is not that the leader commits an entry and announces it, it happens implicitly and this is a rather complex aspect of the protocol, which is hard to appreciate but there is an example and a section in the paper I would request the viewers to take a look at it.

So, let me give an overview of what exactly it says. So, let us say before it commits an entry which means that before it has sent it to all the servers and it has gotten acknowledgments that the entry has been added the leader fails it crashes. So, in this case a new leader is elected, this leader's log clearly has to be as up to date as a majority of the servers where up to date is defined the way we have defined either the last entry has a higher term or the last entry has the same term ID but the leader just has more greater than equal to the leader size of the log is at least as large as the size of the voter.

So, let us say it has the entry e in its log the new leader. So, what will happen is since the new leader does not delete or overwrite any entries in the normal course of operation it will send that the entry e to the rest of the servers that do not have it. So, this is what is going to happen in a normal course of operation. So, the rest of the servers will have the entry it is also possible, so this is one of the cases where this could be safely handled but in general if this happens in the normal course of operation is fine but in general if there is an entry from a previous term that is uncommitted.

So, for the sake of simplicity what raft does is that it does not overwrite it, it does not committed. If the leader has the entry in its log, it will naturally propagate to the rest of the servers, if it does not have the entry it does not automatically committed and this is where simplicity is preferred that we do not automatically commit entries from previous terms. So, this is not an automatic process if it does happen in the course of this leader's operation it is fine otherwise it does not happen otherwise.

(Refer Slide Time: 47:42)



Follower and Candidate Crashes

- Raft keeps trying to send <RequestVote> and <AppendEntries> to all crashed followers and candidates
- All of Raft's messages are idempotent
  - There is no harm if multiple copies of the same message are sent to the same server.
- Timing requirements:
  - Broadcast time ≪ Election time out ≪ Mean time between failures
  - Broadcast time: 0.5 to 20 ms
  - Election timeout: 10ms to 500 ms

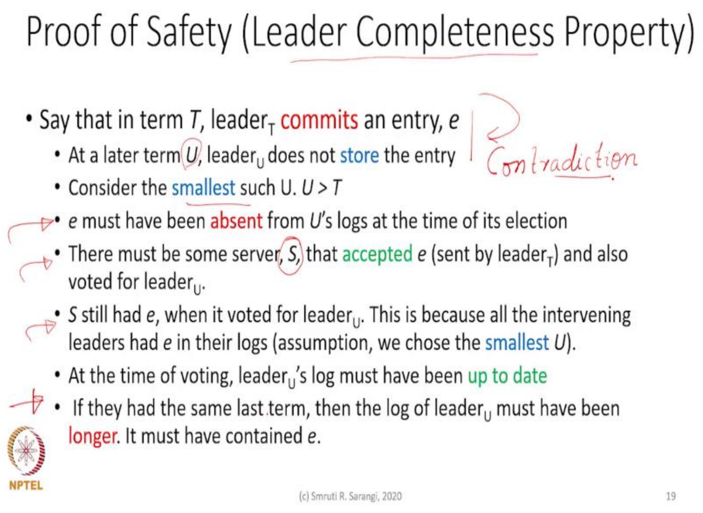
(c) Smruti R. Sarangi, 2020 18

If the follower or the candidate crashes well that is an easy situation to manage. So, raft keeps trying to send request vote and append entries messages to all the crashed followers and candidates. And all of raft's messages are idempotent. What that means is that, there is no harm if multiple copies of the same message are sent to the same server. So, the request vote messages can be sent it is okay append entries messages can also be sent and since the same message can be sent at the number of times there is no issue.

For example, if the append entries message is sent twice or thrice it does not matter if it has been added to the log once it will not be added a second time. So, of course there are some timing requirements the time it takes to broadcast a message for example has to be less than the timeout time of an election that that is very clear.

Because if it times out before that, then we will just be doing broadcast after broadcast and the election time out time has to be much lower than the mean time between failures because if that is not the case then we will actually fail rather often as compared to the timeout and the election will never conclude. So, some typical examples that are there in the paper is that the broadcast time is between 0.5 and 20 milliseconds and election timeout is between 10 milliseconds and 500 milliseconds.

(Refer Slide Time: 49:17)



Proof of Safety (Leader Completeness Property)

- Say that in term T, leader$_T$ commits an entry, e
  - At a later term U, leader$_U$ does not store the entry
  - Consider the smallest such U. $U > T$
  - e must have been absent from U's logs at the time of its election
  - There must be some server, S, that accepted e (sent by leader$_T$) and also voted for leader$_U$.
  - S still had e, when it voted for leader$_U$. This is because all the intervening leaders had e in their logs (assumption, we chose the smallest U).
  - At the time of voting, leader$_U$'s log must have been up to date
  - If they had the same last term, then the log of leader$_U$ must have been longer. It must have contained e.

Contradiction

NPTEL

(c) Smruti R. Sarangi, 2020

19

So, now let us provide a quick proof of safety of the leader completeness property. So, we have already described the log matching property. So, that is easy it happens by induction that is the way that the two sub properties S1 and S2 are actually handled. So, let us now discuss the leader completeness property how we would actually prove it in the next two or three slides. So, let us say that in term T the leader of term T, leader T commits an entry e.

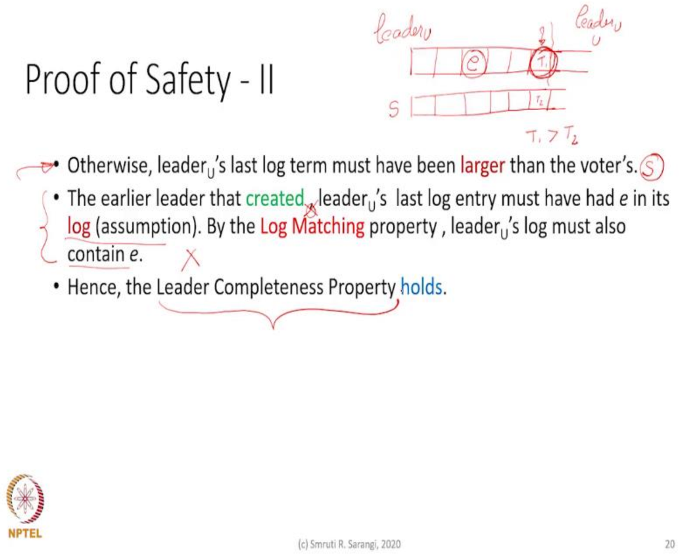
So, at a later term U, leader U does not have the entry. So, this would violate the leader completeness property and this is where we want to prove a contradiction we want to say that this situation will not happen. So, let us consider the smallest search term U where U > T where the entry e is absent in U is absent in the logs of U.

So, the this will only happen if e is absent from U's logs so the time of its election. So, when U was elected. So, it did not have the entry e. So, hence even after the election it does not have the entry e because the process of election per se does not add an entry to the logs. This means that there must be some server S that accepted e which was sent by leader T and that also voted for leader U.

Why? Because essentially commit requires a majority and election requires a majority. So, there has to be an intersection a non-zero intersection and let some server S be in that intersection. So, that must have accepted as well as voted. So, this means that since U is the least such U it is the mean we have taken the smallest such U, S still had e when it voted for leader U this is because all the intervening leaders had e in their logs which is by assumption.

So, the server S still had e it was not removed. So, at the time of voting leader U's log must have been up to date and this means it must have been up to date as compared to the logs of S. This further means that we must do an up to date check which means that if they had the same last term then the log of leader U must have been longer. It, and it must have contained e if it was longer it must have contained e which so this cannot be the case.

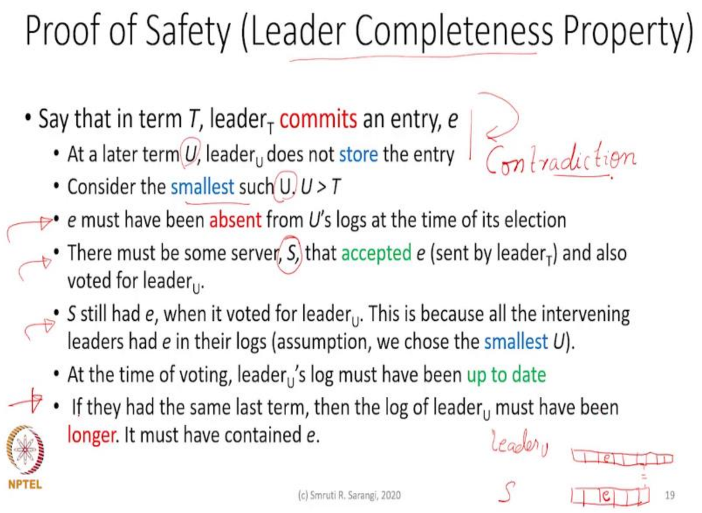
(Refer Slide Time: 52:23)



Proof of Safety - II

- Otherwise, leader$_U$'s last log term must have been larger than the voter's.
- The earlier leader that created leader$_U$'s last log entry must have had *e* in its log (assumption). By the Log Matching property , leader$_U$'s log must also contain *e*.
- Hence, the Leader Completeness Property holds.

leader$_U$
leader$_U$
S
$T_1 > T_2$
NPTEL
(c) Smruti R. Sarangi, 2020
20



Proof of Safety (Leader Completeness Property)

- Say that in term *T*, leader$_T$ commits an entry, *e*
  - At a later term *U*, leader$_U$ does not store the entry
  - Consider the smallest such U. *U > T*
  - *e* must have been absent from *U*'s logs at the time of its election
  - There must be some server, *S*, that accepted *e* (sent by leader$_T$) and also voted for leader$_U$.
  - *S* still had *e*, when it voted for leader$_U$. This is because all the intervening leaders had *e* in their logs (assumption, we chose the smallest *U*).
  - At the time of voting, leader$_U$'s log must have been up to date
  - If they had the same last term, then the log of leader$_U$ must have been longer. It must have contained *e*.

So, if that is not the case that only the other case is possible which is that leader U’s last log and log term must have been larger than the voters which in this case is S. So, the term of the last entry must have been larger than the term of the voters logs last entry. So, let us see, is this possible? So, this is not possible the reason is like this, the earlier leader that created leader U's last log entry must have had e in its log that is an assumption.

So, by the log matching property leader use log must also contain e. So, it is important we go through this particular thing once again. So, let us explain this once again. So, if you would recall the leader completeness property says that look if a given entry has been committed by a previous leader it is there in the logs of all successive leaders. So, we say let this not be the case let us derive a contradiction. So, let us say some later leader U. So, some later term U whose leader is leader U, where U > T let it not have a given entry.

If it does not have a given entry, it would have not had that entry at the time of voting as well and so, this means that there must have been some server S that accepted e and also voted for leader U. So, for this server S let us try to see what would have happened and derive a contradiction. So, what would have happened is, that the server S would have still had e at the time of voting and the leader U’s log would have been up to date.

This means that if U and say this means that a leader U and S have the same last term. So, in their logs, if they are the same last term so consider their logs see if both of them have the same last term then it must be the case the leader U just has the longer log which means it has more entries at least it has greater than equal to 0 extra entries. So, then this means that from the log matching property if e is somewhere in here e has to be in the same position over here. So, there is no other option.

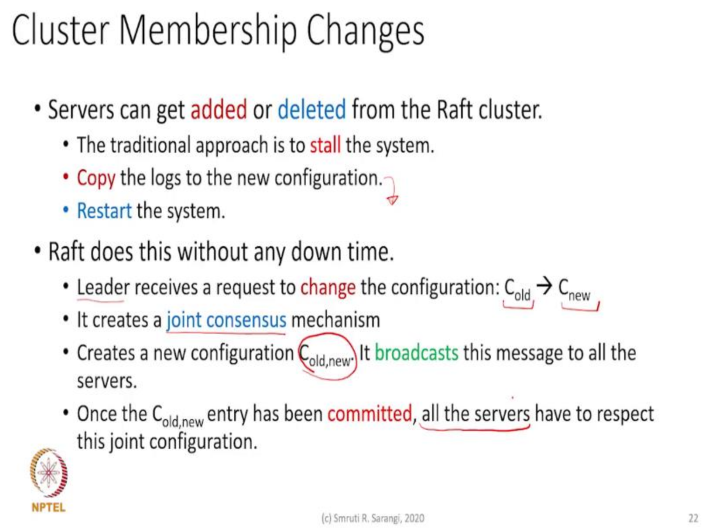
So, since there is no other option this case is precluded. So, this could not have led to the divergence. So, what could have led to the divergence is that the leader U’s last term must have been larger than the voters last term. So, if I were to draw the logs once again. So, what would have happened is that let us say they are last terms if the term of this is T1 and the term of this of T2 then T1 ≥ T2. So, this would have made the leader use logs more up to date.

So, let us understand what would have happened the earlier leader sorry there will be no comma over here. So, the earlier leader that created leader U’s last log entry. So, let this be the last log entry of leader U, what would have happened? Well, what would have happened is that the earlier leader would have had e in its log. So, that is by the assumption because we are assuming the minimum U the earlier leader that created the log for created the last log entry for leader U.

So, basically what happens this is where the last log entry ends and then the earlier leader crashes and this is where leader U becomes the leader and its term starts. So, the earlier guy who created this entry must have had e in its log and then leader U took it from there. So, what would have happened is that because of the log matching property given that both the logs actually match the all the previous entries would have matched which you have already ensured by induction which means that just at the time of voting leader U would have had e in its logs.

So, since the leader U will subsequently not delete an entry it will continue to have e hence we have a contradiction in both the cases and thus the leader completeness property holds. So, it is important that the viewer goes through this argument several times and appreciates all of its nuances. So, this is reasonably complicated to visualize and explain. So, that is why the viewer needs to go through the proof in the paper several times.

(Refer Slide Time: 58:07)



Cluster Membership Changes

- Servers can get added or deleted from the Raft cluster.
  - The traditional approach is to stall the system.
  - Copy the logs to the new configuration.
  - Restart the system.
- Raft does this without any down time.
  - Leader receives a request to change the configuration: $C_{old} \rightarrow C_{new}$
  - It creates a joint consensus mechanism
  - Creates a new configuration $C_{old,new}$. It broadcasts this message to all the servers.
  - Once the $C_{old,new}$ entry has been committed, all the servers have to respect this joint configuration.

NPTEL

(c) Smruti R. Sarangi, 2020

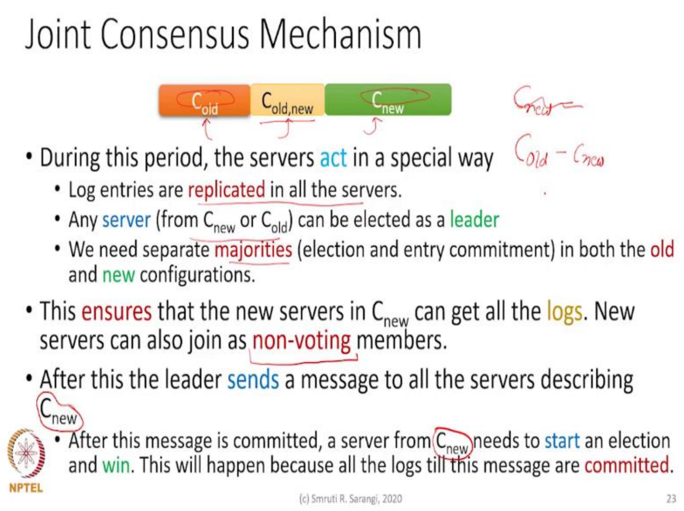
22

A few miscellaneous issues. So, the first is that servers can get added or deleted from the raft cluster. So, a traditional approach would be to stall the entire system copy all the logs from the new configuration to the old configuration and restart the system raft does this without any downtime. So, what raft does is that if the servers are supposed to change it creates something called a joint consensus mechanism, which is kind of like having two leaders at the same time who are working in unison.

So, the leader receives, so what happens is that the old leader receives a request to change the configuration from C old to C new. So, number servers will change. So, then a joint consensus is created called C all new and so this configuration is broadcast to all the servers once the C old new entry has been committed which means a majority of the servers respect this joint configuration. So, this is a special kind of majority which basically means I withdraw the word majority.

So, all the servers in this case have to respect the joint configuration. So, that is kind of simpler. So, if all the servers respect this joint configuration then we go forward.

(Refer Slide Time: 59:42)



Joint Consensus Mechanism

C_{old} | $C_{old,new}$ | C_{new}

C_{new}

$C_{old} - C_{new}$

- During this period, the servers act in a special way
  - Log entries are replicated in all the servers.
  - Any server (from C_{new} or C_{old}) can be elected as a leader
  - We need separate majorities (election and entry commitment) in both the old and new configurations.
- This ensures that the new servers in C_{new} can get all the logs. New servers can also join as non-voting members.
- After this the leader sends a message to all the servers describing C_{new}
  - After this message is committed, a server from C_{new} needs to start an election and win. This will happen because all the logs till this message are committed.

(c) Smruti R. Sarangi, 2020

23

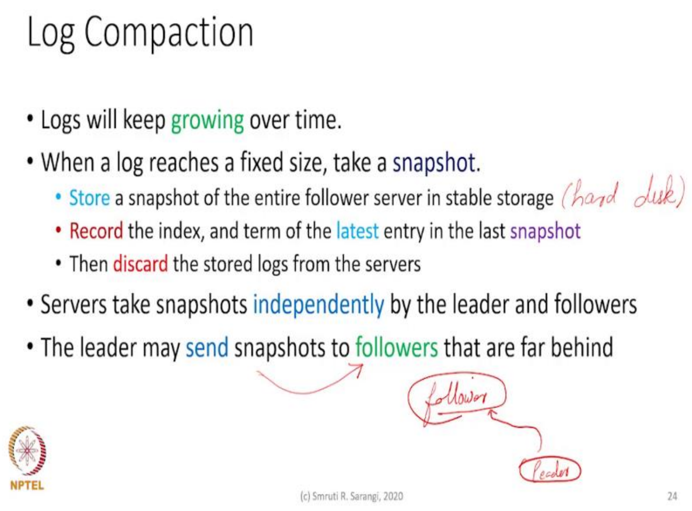
So, what is it that will happen in this case which is will happen in a special way. So, we first start with the C old configuration then we go to C old new which is a new joint configuration and then we go to C new which is a new configuration. So, during this period all the log entries are replicated in all the servers. So, any server from C new or C old during this transition period can be elected as a leader.

So, for any commit we need separate majorities in both the older new configurations which basically means that for sending any log entry we need a majority in C old as well as a majority in C new. So, this ensures that the new servers in C new get to see all the logs and also their logs get up to date over time such that they can ultimately vote and win elections. So, initially they can be non-voting members where they will just be reading and recording the log entries gradually when the logs start getting up to date, they will become voting members after this the leader sends a message to all the servers with C new.

So, after this message is committed a server from C new is expected to start an election and win it. So, this will happen if you can stop the servers from see old in participating from the in the election and the reason it will win is because all the logs are till this message are expected to be committed.

So, that is the reason the candidate will all that the candidate will be eligible because its logs will be at least as up to date as the rest of the servers. Sorry and then the new configuration will take over and the old servers so basically all the servers in C old minus C new which are essentially the servers that are not a part of the new configuration they can be shut off.

(Refer Slide Time: 62:02)



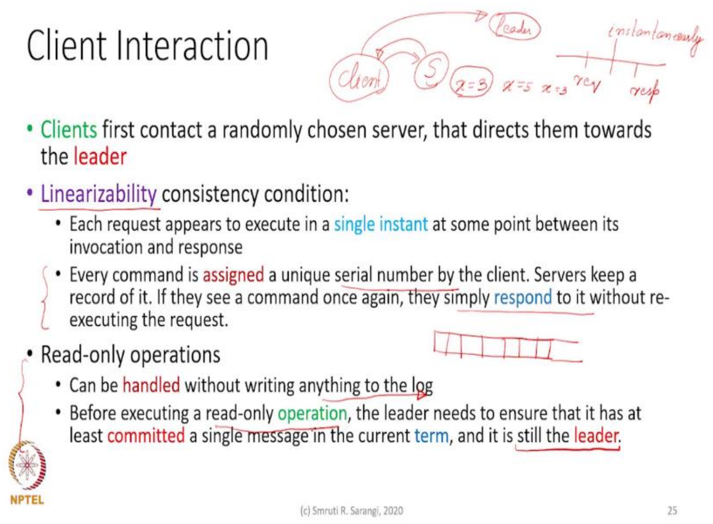
Log Compaction

- Logs will keep growing over time.
- When a log reaches a fixed size, take a snapshot.
    - Store a snapshot of the entire follower server in stable storage (hard disk)
    - Record the index, and term of the latest entry in the last snapshot
    - Then discard the stored logs from the servers
- Servers take snapshots independently by the leader and followers
- The leader may send snapshots to followers that are far behind

The other aspect is log compaction which means that logs will keep growing over time. So, when a log reaches a fixed size, what we can do is that, we need not store the entire log we can take a snapshot. So, a snapshot is just a copy of the log which can be stored in stable storage like a hard disk. We record the index and the term of the latest entry in the snapshot and we discard the logs from the servers.

So, servers can take snapshots independently and furthermore, the snapshot can be used for another good purpose. So, assume a follower is really behind a leader. So, we have a follower but the follower is really running behind a leader. So, in this case what the leader can do is a leader can simply send it a snapshot. It can read all the logs that it does not have and it can add them in its all the log entries it does not have and it can then add them in its log.

(Refer Slide Time: 63:09)



Client Interaction

- Clients first contact a randomly chosen server, that directs them towards the leader
- Linearizability consistency condition:
  - Each request appears to execute in a single instant at some point between its invocation and response
  - Every command is assigned a unique serial number by the client. Servers keep a record of it. If they see a command once again, they simply respond to it without re-executing the request.
- Read-only operations
  - Can be handled without writing anything to the log
  - Before executing a read-only operation, the leader needs to ensure that it has at least committed a single message in the current term, and it is still the leader.

(c) Smruti R. Sarangi, 2020 25

So, let us now discuss the way that the clients interact with the raft system. So, in this case the client first contacts the randomly chosen server. The randomly chosen server returns it the ID of the leader, the client then sends a request to the leader and gets the, and executes, and sends the command. So, command will be sent to the state machines either it can change the state which would be a write operation or it can read the current state which is a read operation.

So, all distributed systems need a notion of correctness in this case we use the linearizability correctness condition which is stronger than sequential consistency. So, here the idea is that between the request and response of a command it appears to execute instantaneously at some point in the middle. So, this is stronger than sequential consistency and most distributed systems try to provide linearizability to some extent and so raft does provide linearizability.

So, to ensure the correctness of this and to ensure that we have some idempotency of the commands. So, of course the read can be issued several times but we will see that is that has complications. But definitely let us say if we send a write and leader crashes, we do not get a response, we send the right once again then it should not be the case that this actually becomes a second right.

So, let us say that we were implementing a key value store and we set x = 3 but the client is not sure if it went through or not and another client sets x = 5. Again, the previous request is retried and we have x = 3. So, this is not a linearizable execution it is not appearing to happen at one instant.

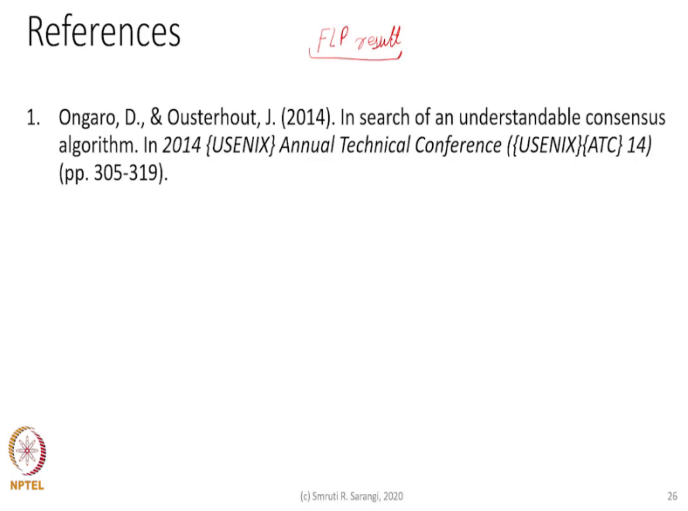
So, what we do is that we assign a unique serial number to each command. So, the client does that to every command it assigns a unique serial number. The servers keep a record of the serial numbers and if they see a command once again they simply respond to it without re-executing the request. So, this ensures that if this request is actually gone through the serial number would be there with servers and they would quickly return the response.

Read operations are difficult in this case because we want to provide the property of linearizability and also we are not writing anything to the log which is reading its state. So, to read the state of the log before executing a read-only operation the leader needs to ensure that it has at least committed a single message in the current term and it is still the leader.

So, being still a leader is important because that is the only way it can execute a request and committing a message in this current means that all the messages in the previous term of all been handled they have either been committed or they have been discarded. So, whatever is the correct state of the log that is present with the leader in the current term. So, it is in a position to actually execute the read request and the read request should always be on the committed state.

So, what do you have to do for a read again, well we will have to commit at least one message in the current term by the current leader and then execute a read on the state machine.

(Refer Slide Time: 67:06)



References

FLP result

1. Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)* (pp. 305-319).

NPTEL

(c) Smruti R. Sarangi, 2020

26

So, this concludes our discussion on the raft consensus protocol. So, the raft consensus protocol similar to paxos does not give us consensus all the time, this is of course according to the FLP result which says that it is impossible to do so. And for when does it not give us a result, when we cannot elect a leader that is one of the cases, when we are not able to create a consensus. But however, we have safety properties in the sense it guarantees us that if something is agreed to, it is agreed to by a majority of the servers it is so there is a consensus and if there is any divergence it can be detected and it can be fixed.