SANDTABLE BUGS DESCRIPTIONS

1. PySyncObj

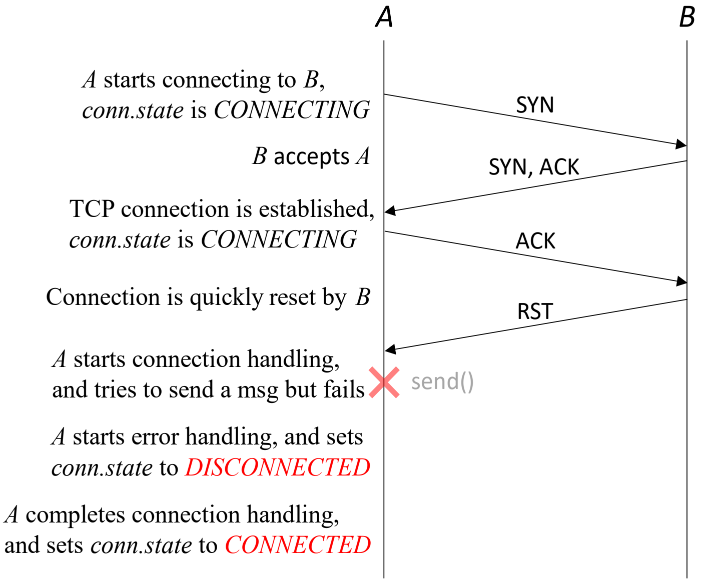
PySyncObj is a Raft protocol library implemented in Python, using TCP sockets for network message transmission, allowing only network partition-related error types.

1.1. PySyncObj#161[1]

This bug exists in the PySyncObj network connection establishment process, resulting in the code throwing an exception.

PySyncObj uses TCP as the network transport layer. After obtaining the socket file descriptor, it sets the file descriptor to non-blocking. The non-blocking connect() system call returns immediately. In the code, it sets the connection state variable to CONNECTING and, in the event loop, obtains the connection result of connect() through I/O multiplexing (such as select() and poll()), and calls the corresponding callback function for processing. When the processing is complete, it sets the connection state to CONNECTED.

The callback function for establishing the connection will call _sendSelfAddress() to send its own address to the other party. If send() returns a failure, it will further call the disconnect() function to close the connection and release resources. After _sendSelfAddress() returns, the connection may have been closed, and the connection state is set to DISCONNECTED. However, PySyncObj does not check the connection state and directly sets the connection state to CONNECTED. In this case, the socket of this connection is null, and attempting to send data using send() will raise an AttributeError exception, causing the raft thread to stop running.


A
B
A starts connecting to *B*,
conn.state is *CONNECTING*
SYN
B accepts *A*
SYN, ACK
TCP connection is established,
conn.state is *CONNECTING*
ACK
Connection is quickly reset by *B*
RST
A starts connection handling,
and tries to send a msg but fails
send()
A starts error handling, and sets
conn.state to *DISCONNECTED*
A completes connection handling,
and sets *conn.state* to *CONNECTED*

[1] "Fix disconnection when connecting by tangruize · Pull Request #161 · bakwc/PySyncObj," *GitHub*. https://github.com/bakwc/PySyncObj/pull/161 (accessed May 28, 2023).

```
    if self.__state == CONNECTION_STATE.CONNECTING:
        if self.__onConnected is not None:
            self.__onConnected() # Calling _sendSelfAddress() may result in
    disconnection
        self.__state = CONNECTION_STATE.CONNECTED
```

1.2. PySyncObj#166[1]

This is a protocol-level bug where under certain conditions, the commit index (referred to as "ci" in the diagram) decreases, violating the safety properties of the Raft protocol.

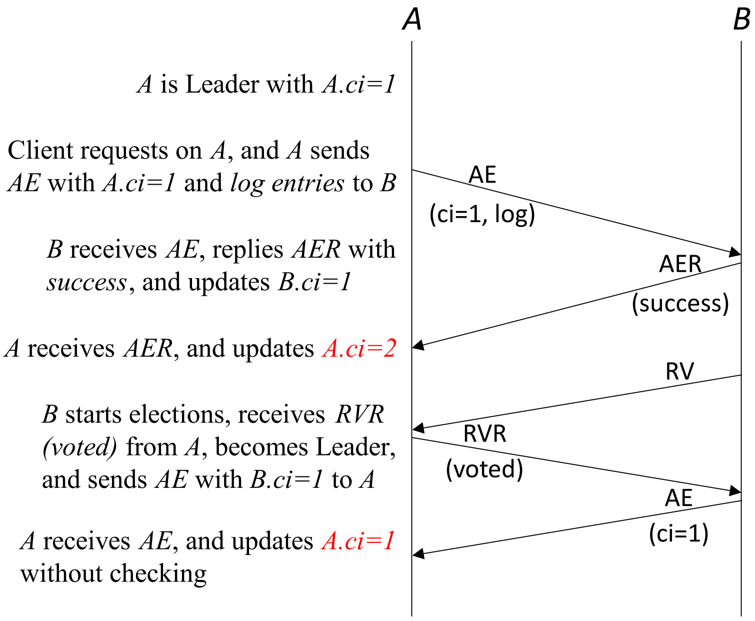
Root Cause: When a follower receives an AppendEntries message (referred to as "AE" in the diagram) from the leader, upon successful processing of the message, the follower sets its current commit index to min(leaderCommitIndex, currentLogIndex). In other words, it sets the commit index to the minimum value between the leader's commit index included in the message and the index of the current log entry. However, there is no check before assigning the commit index to ensure that it is greater than or equal to min(leaderCommitIndex, currentLogIndex), leading to cases where the commit index is not monotonically increasing.

```
    if message['type'] == 'append_entries' and message['term']
    >= self.__raftCurrentTerm:
        # Lack of size check
        self.__raftCommitIndex =
    min(leaderCommitIndex, self.__getCurrentLogIndex())
```

Shortest Triggering Path:

1. The current Leader is A. Both A and B have a commit index of 1.
2. A receives a client request, adds the log entry from the request to the log, and then sends AppendEntries to B. AppendEntries includes the log entry of this request and the current commit index (with a value of 1).
3. B receives the AppendEntries, processes it successfully, and returns AppendEntriesResponse to A.
4. A receives AppendEntriesResponse and updates the current index to 2.
5. Due to some reason (such as network delay), B times out and sends RequestVote to A.
6. A votes for B and returns RequestVoteResponse.
7. B receives RequestVoteResponse, becomes the Leader, and sends heartbeat AppendEntries to A. AppendEntries includes the current commit index (with a value of 1).
8. A receives AppendEntries, updates the commit index to 1. Before the update, A's commit index was 2.

[1] "Raft commit index is not monotonic · Issue #166 · bakwc/PySyncObj," *GitHub*. https://github.com/bakwc/PySyncObj/issues/166 (accessed May 28, 2023).


A
B
A is Leader with A.ci=1
Client requests on A, and A sends AE with A.ci=1 and log entries to B
AE
(ci=1, log)
B receives AE, replies AER with success, and updates B.ci=1
AER
(success)
A receives AER, and updates A.ci=2
RV
B starts elections, receives RVR (voted) from A, becomes Leader, and sends AE with B.ci=1 to A
RVR
(voted)
AE
(ci=1)
A receives AE, and updates A.ci=1 without checking

Feedback on the Fixing Process: Developers successfully reproduced this bug by adding a delay in the random testing component of PySyncObj.

1.3. PySyncObj#167-1[1]

This is a protocol-level bug where a certain scenario causes the Leader's next index (referred to as "ni" in the diagram) to be less than or equal to the match index (referred to as "mi" in the diagram). This violates the safety property of the Raft protocol, leading to unnecessarily re-sending already matched log entries. While this bug may not directly lead to severe consequences, it could degrade performance and pose potential risks.

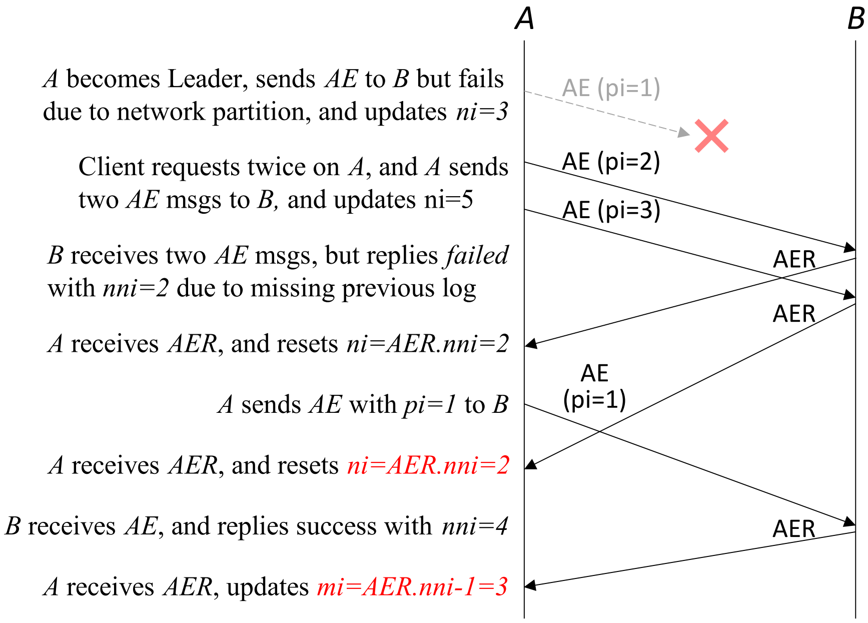
Root Cause: After the Leader receives an AppendEntriesResponse message (referred to as "next_node_idx" in PySyncObj) from a Follower, if the result is successful, it updates the match index without updating the next index. As a result, the updated match index may be greater than or equal to the next index.

```
if self.__raftState == _RAFT_STATE.LEADER:
    if message['type'] == 'next_node_idx':
        reset = message['reset']
        nextNodeIdx = message['next_node_idx']
        success = message['success']
        currentNodeIdx = nextNodeIdx - 1
        if reset:
            self.__raftNextIndex[node] = nextNodeIdx
        if success:
            # Not simultaneously setting the next index
            self.__raftMatchIndex[node] = currentNodeIdx
```

[1] "Raft match index is not monotonic · Issue #167 · bakwc/PySyncObj." https://github.com/bakwc/PySyncObj/issues/167 (accessed May 28, 2023).

PySyncObj has optimized the original Raft protocol such that when a Leader sends AppendEntries messages, it automatically updates the next index to the index of the current maximum log entry plus one. If a previous AppendEntries message has not been received by a Follower, the prev log index in subsequent AppendEntries messages cannot match the Follower's log. In such cases, the Follower returns an AppendEntriesResponse message indicating failure, along with the next node index to suggest where the Leader should send log entries next time.

```
    def __sendAppendEntries(self):
        if nextNodeIndex <= self.__getCurrentLogIndex():
            entries = self.__getEntries(nextNodeIndex,
    None, batchSizeBytes)
            # Optimization: Sending AppendEntries updates the next index
            self.__raftNextIndex[node] = entries[-1][1] + 1
```


A
B
A becomes Leader, sends AE to B but fails due to network partition, and updates ni=3
AE (pi=1)
Client requests twice on A, and A sends two AE msgs to B, and updates ni=5
AE (pi=2)
AE (pi=3)
B receives two AE msgs, but replies failed with nni=2 due to missing previous log
AER
AER
A receives AER, and resets ni=AER.nni=2
A sends AE with pi=1 to B
AE (pi=1)
A receives AER, and resets ni=AER.nni=2
B receives AE, and replies success with nni=4
AER
A receives AER, updates mi=AER.nni-1=3

Shortest Triggering Path:

1. A and B are initialized as Followers, with the log containing only one no-op log entry.
2. A becomes the Leader through an election, adds a no-op log entry to the log, and sends an AppendEntries message with prev log index 1 (abbreviated as "pi" in the diagram) to B. After sending, A updates the next index recorded for B to 3. However, due to network partition, this AppendEntries message is lost. Later, the network is restored.
3. A receives two client requests, sends two AppendEntries messages with prev log index 2 and 3 respectively to B. After execution, next index becomes 5.
4. B processes the two AppendEntries messages, but since there is only one log entry in the current log and the prev log index for the messages are 2 and 3 respectively, the log entry is missing, resulting in failure. AppendEntriesResponse includes next node index = 2.
5. A receives AppendEntriesResponse and sets next index to 2.

6. A (broadcast timeout) sends AppendEntries (prev log index 1) to B.
7. A receives AppendEntriesResponse and sets next index to 2.
8. B receives AppendEntries, the log matches, returns success, and the next node index in AppendEntriesResponse message is 4.
9. A receives AppendEntriesResponse, updates match index to 3, but does not update next index.

1.4. PySyncObj#167-2[1]

This is a protocol-level bug where a certain scenario causes the match index (abbreviated as "mi" in the diagram) in the Leader to not monotonically increase, violating the safety property required by the Raft protocol. Since the match index is used to determine which log entries can be committed, a decrease in the match index may potentially lead to severe data loss or errors.

Root Causes:

1. The Leader fails to set the next index when receiving a successful AppendEntriesResponse, as mentioned in PySyncObj#167-1.
2. The optimization in PySyncObj#167-1 where the Leader updates the next index when sending AppendEntries.

   When the Leader receives a successful AppendEntriesResponse, it does not perform a size check on the match index about to be set. This leads to situations where the match index value requested to be set in the AppendEntriesResponse message is smaller. (PySyncObj uses TCP as the transport layer, which guarantees that network messages will not be reordered, but scenarios where the match index is smaller still exist, as in the bug mentioned in the next point.)

```
if self.__raftState == _RAFT_STATE.LEADER:
    if message['type'] == 'next_node_idx':
        reset = message['reset']
        nextNodeIdx = message['next_node_idx']
        success = message['success']
        currentNodeIdx = nextNodeIdx - 1
        if reset:
            self.__raftNextIndex[node] = nextNodeIdx
        if success:
            # No check on the size of currentNodeIdx
            self.__raftMatchIndex[node] = currentNodeIdx
```

3. When a Follower receives an AppendEntries message containing log entries and returns success, it incorrectly sets the nextNodeIdx in the return message to the index of the

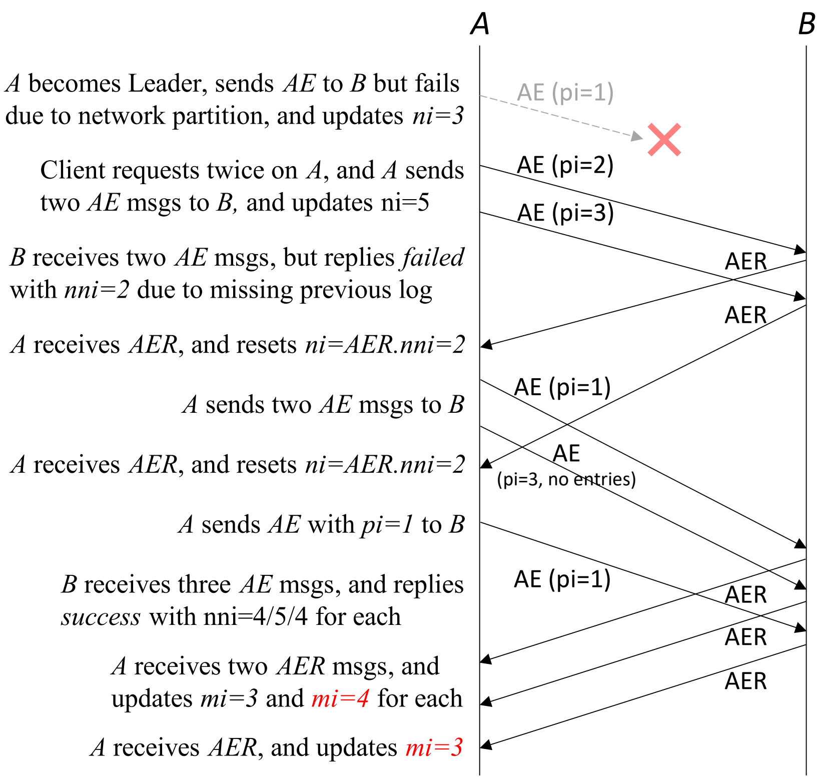
[1] "Raft match index is not monotonic · Issue #167 · bakwc/PySyncObj." https://github.com/bakwc/PySyncObj/issues/167 (accessed May 28, 2023).

current maximum log entry. It should actually be set to that index plus one.

```
if message['type'] == 'append_entries' and message['term']
>= self.__raftCurrentTerm:
    nextNodeIdx = prevLogIdx + 1
    if newEntries:
        nextNodeIdx = newEntries[-1][1] # Missing +1
    self.__sendNextNodeIdx(node, nextNodeIdx=nextNodeIdx, succ
ess=True)
```

Shortest Triggering Path:

1. A and B are initialized as Followers, with the log containing only one no-op log entry.
2. A becomes the Leader through an election, adds a no-op log entry to the log, and sends an AppendEntries message with prev log index 1 (abbreviated as "pi" in the diagram) to B. After sending, A updates the next index recorded for B to 3. However, due to network partition, this AppendEntries message is lost. Later, the network is restored.
3. A receives two client requests, sends two AppendEntries messages with prev log index 2 and 3 respectively to B, and after execution, next index becomes 5.
4. B processes the two AppendEntries messages, but since there is only one log entry in the current log and the prev log index for the messages are 2 and 3 respectively, the log entry is missing, resulting in failure. AppendEntriesResponse includes next node index = 2.
5. A receives AppendEntriesResponse and sets next index to 2.
6. A sends AppendEntries (prev log index 1, with log entries) to B, and sets next index to 5.
7. A sends AppendEntries (prev log index 3, with no log entries) to B.
8. A receives AppendEntriesResponse and sets next index to 2.
9. A sends AppendEntries (prev log index 1, with log entries) to B, and sets next index to 5.
10. B receives three AppendEntries messages, all return success because the logs match. The next node index in the returned AppendEntriesResponse messages is incorrectly set due to the code bug, missing a +1 when there are log entries in the AppendEntries messages (the first and third messages). They return next node index as 4/5/4 respectively.
11. A receives three AppendEntriesResponse messages, updates match index because all return success. However, there is no check on the size of the match index to be set. Processing the second message updates match index to 4, while processing the third message updates match index to 3, resulting in a decreasing match index behavior.


A
B
A becomes Leader, sends AE to B but fails due to network partition, and updates ni=3
AE (pi=1)
Client requests twice on A, and A sends two AE msgs to B, and updates ni=5
AE (pi=2)
AE (pi=3)
B receives two AE msgs, but replies failed with nni=2 due to missing previous log
AER
AER
A receives AER, and resets ni=AER.nni=2
A sends two AE msgs to B
AE (pi=1)
A receives AER, and resets ni=AER.nni=2
AE
(pi=3, no entries)
A sends AE with pi=1 to B
AE (pi=1)
B receives three AE msgs, and replies success with nni=4/5/4 for each
AER
AER
AER
A receives two AER msgs, and updates mi=3 and mi=4 for each
A receives AER, and updates mi=3

1.5. PySyncObj#169[1]

This is a protocol-level bug where there exists a scenario in which the Leader commits a log entry with a term that is not the current term, violating the safety properties of the Raft protocol. The Raft protocol paper illustrates the serious consequences of such scenarios, which may lead to severe data loss.

Root Cause: The Leader fails to check whether the term of the log entry being committed is equal to the Leader's current term when committing log entries.

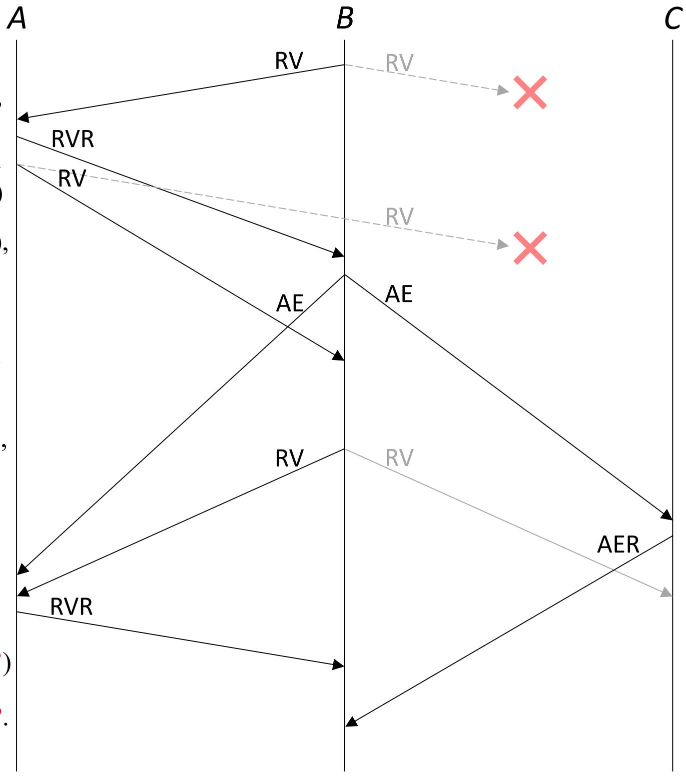
Shortest Triggering Path:

1. Initialize three nodes: A, B, and C. Node C is in a network partition with nodes A and B.
2. B initiates an election (term 1). A receives B's RequestVote and votes for B.
3. A initiates another election (term 2).
4. B receives A's vote, becomes the Leader, adds a NoOp log entry (term 1, index 2) to its log, and sends AppendEntries to A (while A's network partition is still active) and C.
5. B receives A's RequestVote, but since the term of this election request (term 2) is greater than the current term (term 1), B becomes a Follower. As B's log entry has been updated, it does not vote for A.
6. B initiates another election (term 3).

[1] "Leader commits log entries of older terms · Issue #169 · bakwc/PySyncObj," *GitHub*. https://github.com/bakwc/PySyncObj/issues/169 (accessed May 28, 2023).

7. C receives AppendEntries message from B and replies with a successful AppendEntriesResponse.
8. A receives AppendEntries message from B, but since the term in this message is 1, which is less than A's current term (2), A ignores this message.
9. A receives RequestVote from B and votes for B.
10. B receives A's vote and becomes the Leader (term 3).
11. B receives AppendEntriesResponse from C and updates the commit index to 2. The term of the log entry at index 2 in the commit index is 1, while B's current term is 3, leading to the protocol-level bug.

```
while self.__raftCommitIndex < self.__getCurrentLogIndex():
    # No check on the term of the log entry for nextCommitIndex
    nextCommitIndex = self.__raftCommitIndex + 1
    count = 1
    for node in self.__otherNodes:
        if self.__raftMatchIndex[node] >= nextCommitIndex:
            count += 1
    if count > (len(self.__otherNodes) + 1) / 2:
        self.__raftCommitIndex = nextCommitIndex
        self.__raftLog.setRaftCommitIndex(self.__raftCommitIndex)
    else:
        break
```



B starts election (*term=1*), and sends *RV* to *A* and *C* (dropped). *A* votes for *B*

A starts election (*term=2*), and sends *RV* to *B* and *C* (dropped)

B receives *RVR*, becomes Leader (*term=1*), appends a *NoOp* entry (*index=2*), and sends *AE* to *A* and *C*

B receives *RV*, becomes Follower, but doesn't vote for *A* due to longer log

B starts election (*term=3*), and sends *RV* to *A* and *C*

C receives *AE* and replies *success*

A receives *AE* but ignores it due to a higher term, and votes for *B* upon receiving *RV*

B receives *RVR* and becomes Leader (*term=3*)

B receives *AER* from *C*, and update *ci=2*. The term of the log entry is *1*

2. RaftOS

RaftOS is a Raft protocol library implemented in Python that utilizes UDP sockets for network message transmission. It allows for network packet loss, duplication, and out-of-order delivery.

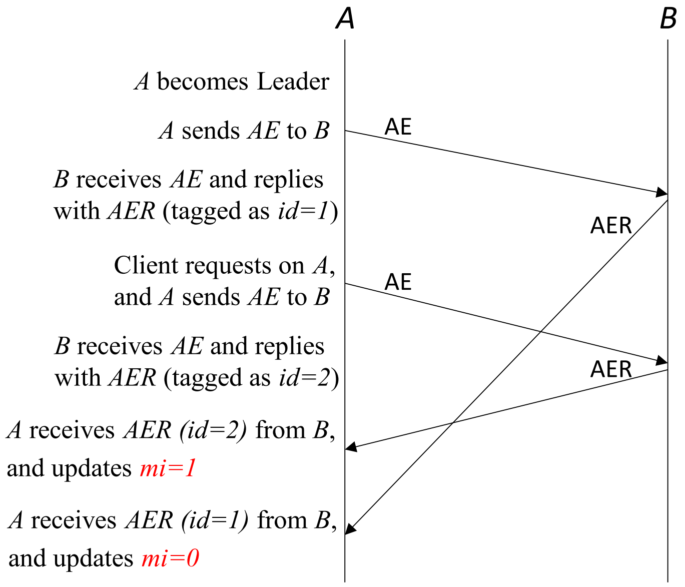
2.1. RaftOS#25[1]

This is a protocol-level bug where there exists a scenario in which the match index (abbreviated as "mi" in the diagram) in the Leader does not monotonically increase, violating the safety properties required by the Raft protocol.

Root Cause: The Leader fails to check the size of self.log.match_index and data['last_log_index'] when handling append_entries_response messages, and directly assigns values. In cases of network disorder, it's easy for the data['last_log_index'] in the message to be smaller.

Shortest Triggering Path:

1. A becomes the Leader and sends an AppendEntries message to B.
2. B receives the AppendEntries message and returns an AppendEntriesResponse, marking this message as id=1.
3. A receives a client request, adds a log entry, and sends an AppendEntries message to B.
4. B receives the AppendEntries message and returns an AppendEntriesResponse, marking this message as id=2.
5. A receives the AppendEntriesResponse with id=2 and updates the match index to 1.
6. A receives the AppendEntriesResponse with id=1 and updates the match index to 0.


A
B
A becomes Leader
A sends AE to B
AE
B receives AE and replies with AER (tagged as id=1)
AER
Client requests on A, and A sends AE to B
AE
B receives AE and replies with AER (tagged as id=2)
AER
A receives AER (id=2) from B, and updates mi=1
A receives AER (id=1) from B, and updates mi=0

[1] “Raft match index is not monotonic · Issue #25 · zhebrak/raftos,” *GitHub*. https://github.com/zhebrak/raftos/issues/25 (accessed May 28, 2023).

```
def on_receive_append_entries_response(self, data):
    if not data['success’]:
        # ..
    else:
        # Direct assignment without size check
        self.log.next_index[sender_id] = data['last_log_index'] + 1
        self.log.match_index[sender_id] = data['last_log_index']
```

2.2. RaftOS#26[1]

This is a protocol-level bug where there exists a scenario in which committed logs are not synchronized to a majority of nodes.

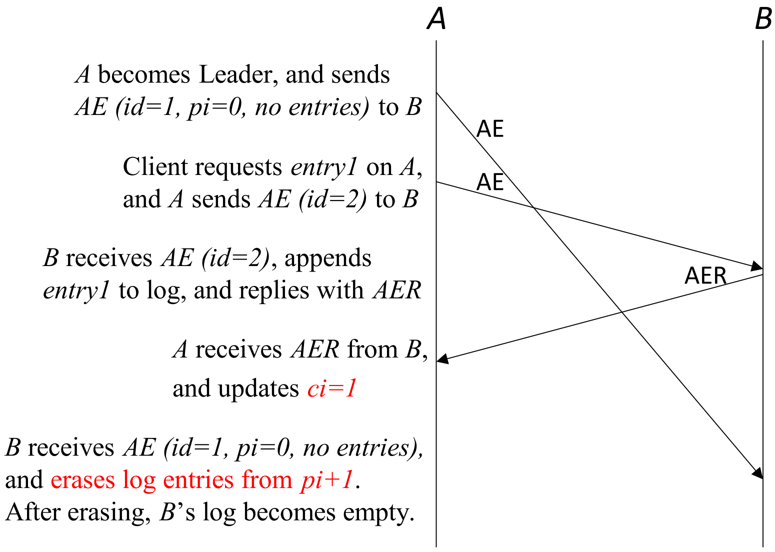
Root Cause: When a Follower receives an AppendEntries message, it incorrectly considers self.log.last_log_index != prev_log_index as a mismatched log condition and deletes logs after data['prev_log_index'] + 1. However, this condition check is unrelated to whether the logs match; the matching of logs only requires checking if the terms are equal. This erroneous check leads to the deletion of logs that have already been matched.

```
def on_receive_append_entries(self, data):
    self.state.set_leader(data['leader_id’])

# Reply False if log doesn’t contain an entry at prev_log_index whos
e term matches prev_log_term
    try:
        prev_log_index = data['prev_log_index’]
        # ...
    except IndexError:
        pass

# If an existing entry conflicts with a new one (same index but diff
erent terms),
    # delete the existing entry and all that follow it
    new_index = data['prev_log_index'] + 1
    try:
        # Only need to check term, and self.log.last_log_index !=
prev_log_index cannot indicate log mismatch
        # In this case, it erroneously deletes already matched logs
        if self.log[new_index]['term'] != data['term'] or (
            self.log.last_log_index != prev_log_index
        ):
            self.log.erase_from(new_index)
    except IndexError:
        pass
```

[1] “Log may erased incorrectly · Issue #26 · zhebrak/raftos,” *GitHub*. https://github.com/zhebrak/raftos/issues/26 (accessed May 28, 2023).

Shortest Triggering Path:

1. A becomes the Leader and sends an AppendEntries message (id=1, prev_log_index=0) to B. This message is delayed due to network issues.
2. A receives a client request, adds entry1 to its log, and sends an AppendEntries message (id=2, containing entry1) to B.
3. B receives the AppendEntries message (id=2), adds entry1 to its log, and responds with an AppendEntriesResponse.
4. A receives the AppendEntriesResponse and updates the commit index to 1.
5. B receives the delayed AppendEntries message (id=1, prev_log_index=0), incorrectly assumes log mismatch, and starts deleting log entries from index 1 onwards. After deletion, B's log becomes empty. It's evident that the log entry already committed in A is not replicated to a majority of nodes.


A
B
A becomes Leader, and sends *AE (id=1, pi=0, no entries)* to *B*
AE
Client requests *entry1* on *A*, and *A* sends *AE (id=2)* to *B*
AE
B receives *AE (id=2)*, appends *entry1* to log, and replies with *AER*
AER
A receives *AER* from *B*, and updates *ci=1*
B receives *AE (id=1, pi=0, no entries)*, and erases log entries from *pi+1*. After erasing, *B*'s log becomes empty.

2.3. RaftOS#27[1]

This is a code-level bug where, in a certain scenario, a node attempts to retrieve a non-existent key from a message, resulting in the code throwing an exception.

The root cause is that when a node receives a request message (such as RequestVote or AppendEntries), if the node's term is greater than the term in the message, it directly replies with failure. However, the reply message does not contain the 'request_id' key. Subsequently, when the method for receiving AppendEntries responses attempts to check the 'request_id' key, the absence of this key causes the code to throw an exception.

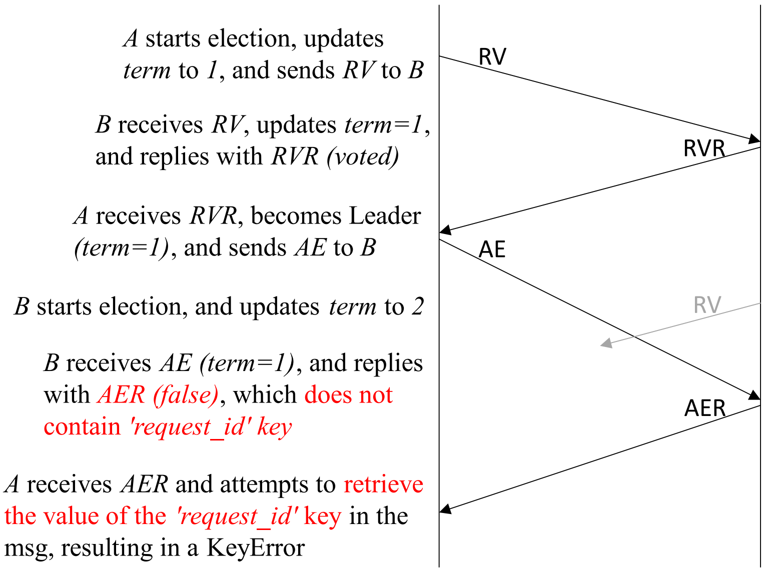
[1] "KeyError in handling append_entries_response message · Issue #27 · zhebrak/raftos," *GitHub*. https://github.com/zhebrak/raftos/issues/27 (accessed May 28, 2023).

```
def on_receive_function(self, data):
    # ...
    if self.storage.term > data['term'] and
not data['type'].endswith('_response’):
        # No request_id
        response = {
            'type': '{}_response'.format(data['type']),
            'term': self.storage.term,
            'success': False
        }
```

```
def on_receive_append_entries_response(self, data):
    sender_id = self.state.get_sender_id(data['sender’])
    # Obtain request_id upon receiving AER
    if data['request_id'] in self.response_map:
        self.response_map[data['request_id']].add(sender_id)
```

Shortest Triggering Path:

1. A initiates an election, incrementing its term to 1, and sends a RequestVote to B.
2. B receives the RequestVote and votes for A.
3. A receives the vote from B, becoming the Leader with term 1, and sends an AppendEntries to B.
4. B initiates an election, incrementing its term to 2, and sends a RequestVote to A (this message is not important).
5. B receives the AppendEntries from A. Since the term in the message (term=1) is smaller than B's term (term=2), B replies with an AppendEntriesResponse (false). However, the response message does not contain the 'request_id' key-value pair.
6. A receives the AppendEntriesResponse and tries to retrieve the 'request_id' key-value pair. However, since the message does not contain it, a KeyError exception is triggered.


A starts election, updates *term* to *1*, and sends *RV* to *B*
RV
B receives *RV*, updates *term=1*, and replies with *RVR (voted)*
RVR
A receives *RVR*, becomes Leader *(term=1)*, and sends *AE* to *B*
AE
B starts election, and updates *term* to *2*
RV
B receives *AE (term=1)*, and replies with *AER (false)*, which does not contain *'request_id'* key
AER
A receives *AER* and attempts to retrieve the value of the *'request_id'* key in the msg, resulting in a KeyError

2.4. RaftOS#30[①]

This bug is a protocol-level bug where, under certain circumstances, the commit index fails to update, causing the cluster to stall.

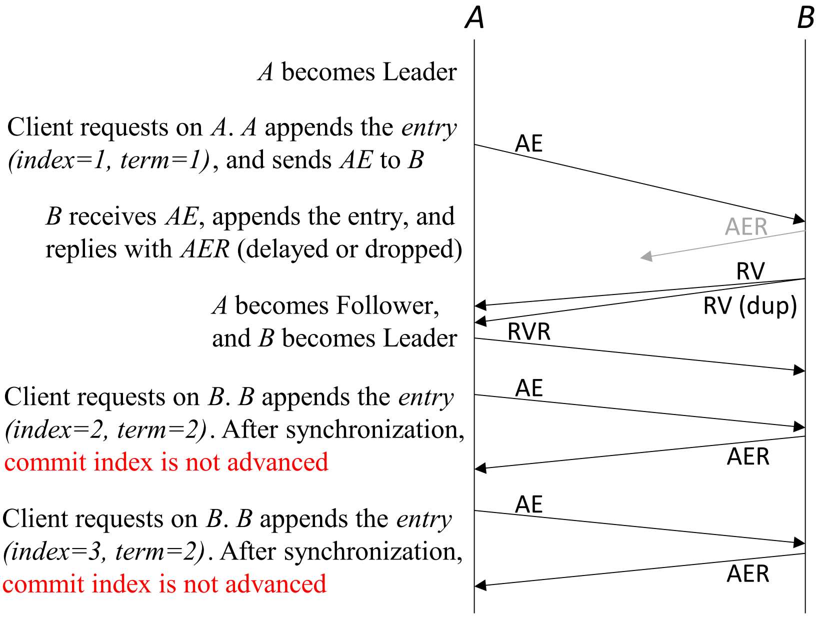
The root cause lies in violating the requirement of the Raft protocol, which prohibits committing indices from a term other than the current term. In the code implementation, when updating the commit index, a loop is used to iterate from the beginning. However, when encountering an index with a term other than the current term, the loop breaks. Consequently, subsequent indices with the current term and are eligible for commitment are not checked, leading to the inability of the commit index to advance.

This bug can be defined as a liveness bug, but it can also be checked using a safety property. We used the safety property to check it and obtained the following shortest triggering path:

1. A becomes the Leader (term=1).
2. A receives a client request and adds a log entry with index=1 and term=1 to the log, then sends AppendEntries to B.
3. B receives the AppendEntries and replies with an AppendEntriesResponse. However, due to network delay or packet loss, this message is not immediately received by A.
4. B initiates an election and becomes the Leader (term=2). Due to the characteristics of RaftOS, A only transitions to Follower upon receiving the first RequestVote and votes for B only upon receiving the second repeated RequestVote.
5. B receives a client request and adds a log entry with index=2 and term=2 to the log, then sends AppendEntries to A. Upon receiving AppendEntries, A responds with AppendEntriesResponse. After receiving the AppendEntriesResponse, B should update the commit index to 2. However, due to the bug described in this section, B fails to update the commit index.
6. Repeating step 5, the commit index still cannot be updated, revealing the liveness bug (or it can also be considered a safety bug).

[①] "Change in update commit index by meteam2022 · Pull Request #30 · zhebrak/raftos," *GitHub*. https://github.com/zhebrak/raftos/pull/30 (accessed May 28, 2023).

```
    def update_commit_index(self):
        commited_on_majority = 0
        for index in range(self.log.commit_index +
    1, self.log.last_log_index + 1):
            commited_count = len([
                1 for follower in self.log.match_index
                if self.log.match_index[follower] >= index
            ])
            is_current_term = self.log[index]['term']
    == self.storage.term
            if self.state.is_majority(commited_count + 1)
    and is_current_term:
                commited_on_majority = index
            else:
                # When is_current_term is false, the else branch is
    triggered, causing the loop to break, which may prevent subsequent
    indices from being committed. It should be changed to 'continue'
    instead of 'break'.
                break
        if commited_on_majority > self.log.commit_index:
            self.log.commit_index = commited_on_majority
```


A
B
A becomes Leader
Client requests on *A*. *A* appends the *entry (index=1, term=1)*, and sends *AE* to *B*
AE
B receives *AE*, appends the entry, and replies with *AER* (delayed or dropped)
AER
RV
RV (dup)
A becomes Follower, and *B* becomes Leader
RVR
Client requests on *B*. *B* appends the *entry (index=2, term=2)*. After synchronization, commit index is not advanced
AE
AER
Client requests on *B*. *B* appends the *entry (index=3, term=2)*. After synchronization, commit index is not advanced
AE
AER

3. WRaft

WRaft is a Raft protocol library implemented in the C programming language. This library does not make assumptions about the underlying network, thus allowing for injectable error types including those permitted by UDP and TCP.

All bugs in WRaft have been consolidated into a single pull request (PR), which contains nine bugs. This approach is not recommended as it complicates the understanding for developers and makes it harder to receive responses. We will elaborate on three bugs (1, 2, 4) that result in severe issues.

3.1. WRaft#118.1[1]

This is a protocol-level bug where, under certain circumstances, a Follower receiving AppendEntries messages may redundantly add previously added log entries, leading to data anomalies or corruption.

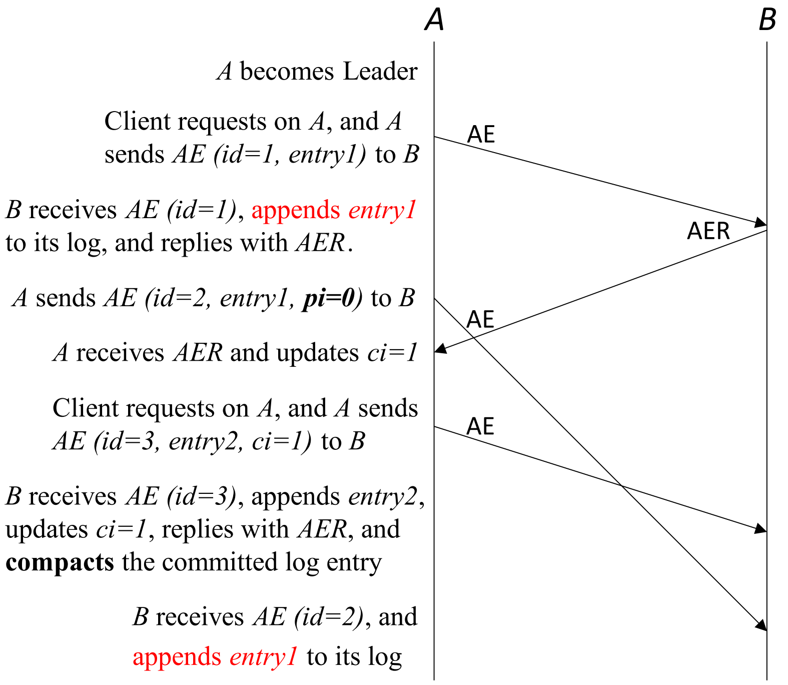
The root cause lies in how a Follower handles AppendEntries messages after taking snapshots of already committed logs. When a Follower receives an AppendEntries message with a previous log index (abbreviated as pi) of 0, the log entries in the AppendEntries message are unconditionally added to the log.

The shortest triggering path (showing only up to where the bug in the code is triggered, as the violation of the model's safety property is easily inferred from the diagram):

1. A becomes the Leader.
2. A receives a request from the Client, adds entry1 to the log, and sends AppendEntries (entry1, prev_log_idx=0), message labeled id=1.
3. B receives the AppendEntries with id=1, adds entry1 to the log, and replies with an AppendEntriesResponse message.
4. A, upon a heartbeat timeout, sends AppendEntries (entry1, prev_log_idx=0) to B, message labeled id=2.
5. A receives an AppendEntriesResponse message from B and updates the commit index to 1.
6. A receives another request from the Client, adds entry2 to the log, and sends AppendEntries (entry2, commit_idx=1) to B, message labeled id=3.
7. B receives the AppendEntries with id=3, adds entry2 to the log, updates the commit index to 1, and replies with an AppendEntriesResponse message (this message is not relevant to the bug and is not depicted in the diagram).
8. B meets the conditions for taking a snapshot and deletes the already committed log entry (entry1) from the log.
9. B receives the AppendEntries with id=2 and adds entry1 to the log.
10. Clearly, the Leader has only two log entries, while the Follower has three (the first one has been compacted). When the Leader commits the third log entry, it becomes evident that the committed logs are inconsistent across different nodes (leading to inconsistent state machines), thus violating the safety property of the model.

[1] "Fix bugs mainly caused by snapshot by tangruize · Pull Request #118 · willemt/raft," *GitHub*. https://github.com/willemt/raft/pull/118#:~:text=raft_recv_appendentries%23L432 (accessed May 28, 2023).

```
/* Not the first appendentries we've received */
/* NOTE: the log starts at 1 */
// Special handling for cases where previous log index is greater
than 0, does not enter this branch when equal to 0
if (0 < ae->prev_log_idx)
{
    raft_entry_t* ety = raft_get_entry_from_idx(me_, ae-
>prev_log_idx);
    /* Is a snapshot */
    if (ae->prev_log_idx == me->snapshot_last_idx)
        // ... Irrelevant

/* 2. Reply false if log doesn't contain an entry at prevLogIndex
       whose term matches prevLogTerm (§5.3) */
    else if (!ety)
    {
        // If prev_log_idx is greater than 0 and entry does not
exist (not received or already snapshot), return failure
        __log(me_, node, "AE no log at prev_idx %d", ae-
>prev_log_idx);
        goto out;
    }
    else if (ety->term != ae->prev_log_term)
        // ... Irrelevant
}
r->success = 1;
r->current_idx = ae->prev_log_idx;
/* 3. If an existing entry conflicts with a new one (same index
   but different terms), delete the existing entry and all that
   follow it (§5.3) */
int i;
for (i = 0; i < ae->n_entries; i++)
{
    raft_entry_t* ety = &ae->entries[i];
    raft_index_t ety_index = ae->prev_log_idx + 1 + i;
    raft_entry_t* existing_ety =
raft_get_entry_from_idx(me_, ety_index);
    if (existing_ety && existing_ety->term != ety->term)
        // ... Irrelevant
    else if (!existing_ety)
        // If entry does not exist, break the loop, all entries
after the position of i when the loop exits will be appended
        break;
    // ... Irrelevant
}
/* Pick up remainder in case of mismatch or missing entry */
for (; i < ae->n_entries; i++)
{
    // Append all entries from AE that were not added in the
previous loop
    e = raft_append_entry(me_, &ae->entries[i]);
    // ... Irrelevant
}
```



A becomes Leader

Client requests on *A*, and *A* sends *AE (id=1, entry1)* to *B*

B receives *AE (id=1)*, appends *entry1* to its log, and replies with *AER*.

A sends *AE (id=2, entry1,* ***pi=0****)* to *B*

A receives *AER* and updates *ci=1*

Client requests on *A*, and *A* sends *AE (id=3, entry2, ci=1)* to *B*

B receives *AE (id=3)*, appends *entry2*, updates *ci=1*, replies with *AER*, and **compacts** the committed log entry

B receives *AE (id=2)*, and appends *entry1* to its log

It seems that RedisRaft independently fixed this bug by removing the check for whether the previous log index is greater than 0 in RedisRaft#148[1].

3.2. WRaft#118.2[2]

This is a protocol-level bug where a Follower incorrectly updates the commit index upon receiving erroneous AppendEntries messages, leading to the commitment of log entries that should not have been committed. These mistakenly committed log entries may later be deleted during subsequent synchronizations, violating the safety property that committed logs cannot be deleted. A more severe consequence is that a Follower, after erroneously updating the commit index, may become a Leader, leading to the deletion of correctly committed log entries.

The root cause of this issue lies in the boundary condition checks in the Leader's logic when determining whether to send AppendEntries or Snapshot messages. Consequently, messages that should have been sent as Snapshots are erroneously sent as AppendEntries. However, since the relevant logs have already been snapshotted, the AppendEntries messages contain empty entries. When a Follower receives such an AppendEntries message with a previous log index of 0, it incorrectly determines a match on the previous log and proceeds to the code for adding entries. Since the AppendEntries message contains no entries, it does not delete mismatched entries. The Follower then updates the commit index to the commit index in the AppendEntries message, which may correspond to a log entry that does not match the Leader's log.

[1] "Refactor raft_recv_appendentries() by tezc · Pull Request #148 · RedisLabs/raft," *GitHub*. https://github.com/RedisLabs/raft/pull/148 (accessed May 28, 2023).

[2] "Fix bugs mainly caused by snapshot by tangruize · Pull Request #118 · willemt/raft," *GitHub*. https://github.com/willemt/raft/pull/118#:~:text=raft_send_appendentries%23L901 (accessed May 28, 2023).

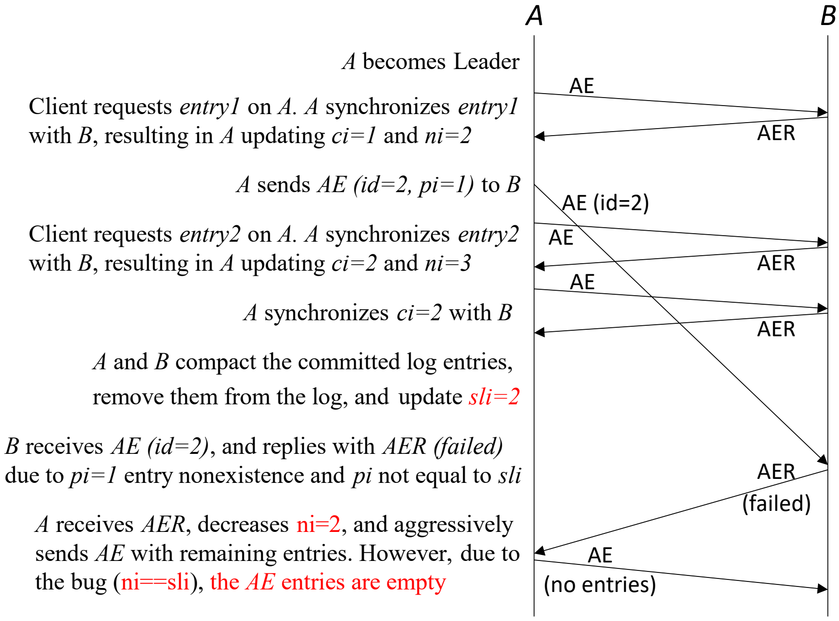
```
    int raft_send_appendentries(raft_server_t* me_, raft_node_t* node)
    {
        raft_server_private_t* me = (raft_server_private_t*)me_;
        // ...
        raft_index_t next_idx = raft_node_get_next_idx(node);
        /* figure out if the client needs a snapshot sent */
        // Incorrect condition for sending snapshot, should
    be next_idx <= me->snapshot_last_idx
        if (0 < me->snapshot_last_idx && next_idx < me-
    >snapshot_last_idx)
        {
            if (me->cb.send_snapshot)
                me->cb.send_snapshot(me_, me->udata, node);
            return RAFT_ERR_NEEDS_SNAPSHOT;
        }
        // ...不关心
    }
```

The second mistake lies in not sending snapshots to nodes at the boundary conditions while taking snapshots, but this does not cause the bug described in this section:

```
    int raft_end_snapshot(raft_server_t *me_)
    {
        raft_server_private_t* me = (raft_server_private_t*)me_;
        // ...
        for (i = 0; i < me->num_nodes; i++)
        {
            raft_node_t* node = me->nodes[i];
            raft_index_t next_idx = raft_node_get_next_idx(node);
            /* figure out if the client needs a snapshot sent */
            // Same error as the second mistake, but it will not cause a
    severe bug
            if (0
    < me->snapshot_last_idx && next_idx < me->snapshot_last_idx)
            {
                if (me->cb.send_snapshot)
                    me->cb.send_snapshot(me_, me->udata, node);
            }
        }
        return 0;
    }
```

The first minimal triggering path (due to the complexity of the TLA+ trace, only showing up to the point of triggering the code bug, as described above, this bug would violate the safety property described in this subsection):

1. A becomes Leader.
2. A receives a client request, adds entry1 to the log, and synchronizes the log with B. After synchronization, A updates the commit index to 1, and A updates its next index for B to 2.
3. A's heartbeat timeout triggers, sending AppendEntries to B with the previous log index as 1, noted as AE (id=2). Due to network delay, it hasn't reached B temporarily.
4. A receives another client request, adds entry2 to the log, and synchronizes the log with B. After synchronization, A updates the commit index to 2, and A updates its next index for B to 3.
5. A's heartbeat timeout triggers again, sending AppendEntries to B with the commit index as 2. After synchronization, B updates the commit index to 2.
6. Both A and B reach the condition for taking a snapshot, updating the snapshot last index to 2, and respectively removing the committed log entries from the log (i.e., deleting all log entries).
7. B receives AE (id=2). Since the log entry at previous log index 1 does not exist, and the previous log index is not equal to the snapshot last index, B replies with AppendEntriesResponse (failed).
8. A receives AppendEntriesResponse (failed) and decrements the next index to 2. Since the next index is not greater than the maximum log length, A considers B's log not synchronized and initiates fast synchronization. Due to the bug described in this subsection, when next index equals the snapshot last index, A sends AppendEntries to B instead of InstallSnapshot. During the sending process, since the log entry at the next index position has been deleted, the AppendEntries contains empty entries, which is not consistent with the purpose of fast synchronization.
9. Obviously, if B has uncommitted log entries different from A (i.e., log entries with the same index but different terms), if the incorrectly sent AppendEntries message contains a commit index greater than or equal to the index of such differing log entry, and under certain conditions where the log is deemed to match (e.g., when the previous log index is 0 and the log unconditionally matches), due to the empty entries in the message, it will not cause the differing log entry to be deleted, resulting in the erroneous commitment of that log entry and leading to more serious consequences.

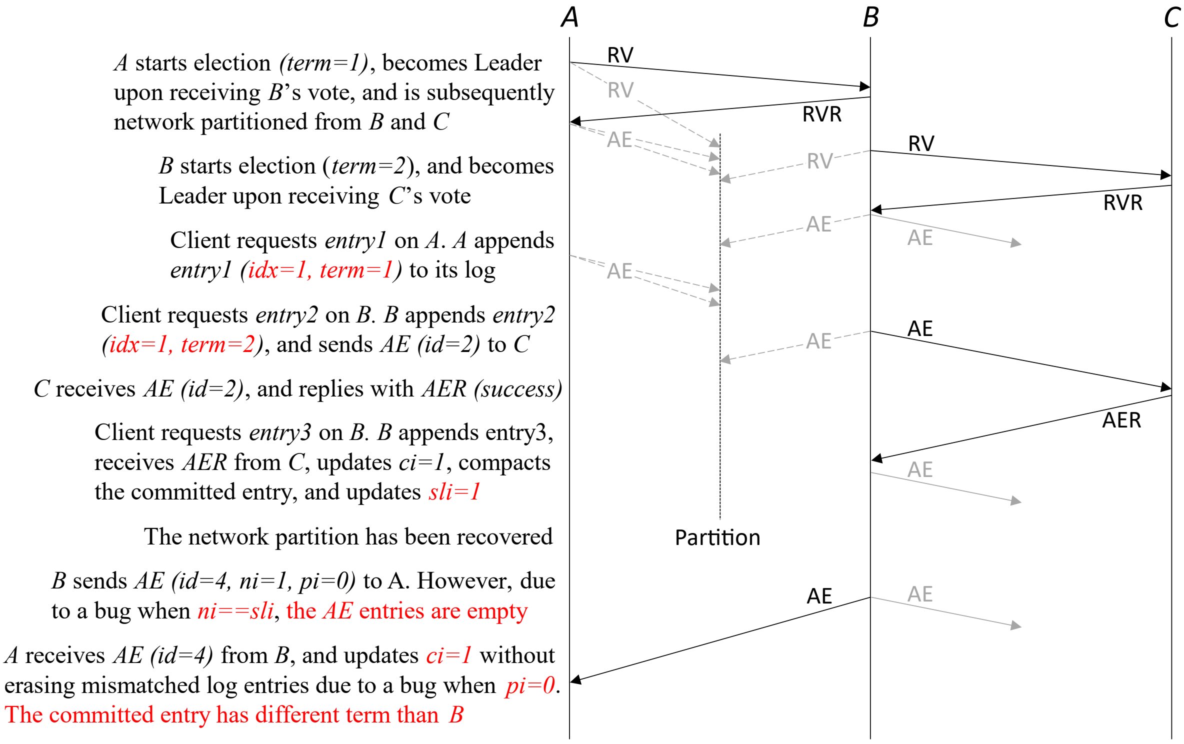

A
B
A becomes Leader
Client requests entry1 on A. A synchronizes entry1 with B, resulting in A updating ci=1 and ni=2
A sends AE (id=2, pi=1) to B
Client requests entry2 on A. A synchronizes entry2 with B, resulting in A updating ci=2 and ni=3
A synchronizes ci=2 with B
A and B compact the committed log entries, remove them from the log, and update sli=2
B receives AE (id=2), and replies with AER (failed) due to pi=1 entry nonexistence and pi not equal to sli
A receives AER, decreases ni=2, and aggressively sends AE with remaining entries. However, due to the bug (ni==sli), the AE entries are empty
AE
AER
AE (id=2)
AE
AER
AE
AER
AER (failed)
AE (no entries)

The second minimal triggering path (showing up to the point of the committed log having different terms on different nodes, combining the bugs WRaft#118.1 and WRaft#118.2):

1. A initiates an election and becomes Leader after receiving a vote from B (term=1).
2. A becomes partitioned from B and C.
3. B initiates an election and becomes Leader after receiving a vote from C (term=2).
4. A receives a client request and adds the log entry entry1 (index=1, term=1) to its log. Due to the network partition, AppendEntries sent by A cannot reach B.
5. B receives a client request and adds the log entry entry2 (index=1, term=2) to its log, then sends AppendEntries (id=2) to C.
6. Due to the out-of-order arrival of network packets, C receives AppendEntries with id=2 and replies with a successful AppendEntriesResponse.
7. B receives another client request and adds the log entry entry3 to its log (WRaft requires at least two log entries to take a snapshot).
8. B receives an AppendEntriesResponse from C and updates the commit index to 1.
9. B takes a snapshot, updating the snapshot last index to 1 and snapshot last term to 2, and removes the committed log entry (i.e., log entry with index 1) from its log.
10. Network partition between A, B, and C is restored.
11. B's heartbeat timeout triggers, and B's next index for A is 1. Due to the bug described in this subsection, B erroneously sends AppendEntries to A when the next index equals the snapshot last index (should send InstallSnapshot instead). In this AppendEntries message, the commit index is 1, and the previous log index is 0, and the entries are empty due to compression.
12. A receives the AppendEntries from B. Due to the bug WRaft#118.1, when the previous log index is 0, A considers the previous log to be matching. As the entries in the

message are empty, the non-matching log entry (i.e., log entry with index 1) is not deleted. A updates the commit index to the commit index in the message, which is 1. At this point, the non-matching log entries at index 1 on A and B are both committed, leading to severe data corruption.

13. In terms of liveness, if A's log is empty (which is common during node initialization or when first joining the cluster), A receives the erroneous AppendEntries. The subsequent AppendEntriesResponse from A causes B to update its next index but remains unchanged. The next erroneous AppendEntries will continue to be sent. If the snapshot last index does not change, it will lead to the inability of A and B to synchronize their states. With the effect of WRaft#118.8[1] (canceling the sending of AppendEntries to subsequent nodes if InstallSnapshot is sent during the heartbeat process), the fault tolerance of the cluster may decrease.



A starts election *(term=1)*, becomes Leader upon receiving *B*'s vote, and is subsequently network partitioned from *B* and *C*

B starts election (*term=2*), and becomes Leader upon receiving *C*'s vote

Client requests *entry1* on *A*. *A* appends *entry1 (idx=1, term=1)* to its log

Client requests *entry2* on *B*. *B* appends *entry2 (idx=1, term=2)*, and sends *AE (id=2)* to *C*

C receives *AE (id=2)*, and replies with *AER (success)*

Client requests *entry3* on *B*. *B* appends entry3, receives *AER* from *C*, updates *ci=1*, compacts the committed entry, and updates *sli=1*

The network partition has been recovered

B sends *AE (id=4, ni=1, pi=0)* to A. However, due to a bug when *ni==sli*, the *AE* entries are empty

A receives *AE (id=4)* from *B*, and updates *ci=1* without erasing mismatched log entries due to a bug when *pi=0*. The committed entry has different term than *B*

Follow-up: RedisRaft submitted a PR addressing this bug (RedisRaft#47[2]). DaosRaft submitted a PR containing a fix for this bug (DaosRaft#10[3]).

[1] "Fix bugs mainly caused by snapshot by tangruize · Pull Request #118 · willemt/raft," *GitHub*. https://github.com/willemt/raft/pull/118#:~:text=raft_send_appendentries_all%23L952 (accessed May 28, 2023).

[2] "Send snapshot when node's next index is snapshot's last included index by tezc · Pull Request #47 · RedisLabs/raft," *GitHub*. https://github.com/RedisLabs/raft/pull/47 (accessed May 28, 2023).

[3] "Log compaction by liw · Pull Request #10 · daos-stack/raft," *GitHub*. https://github.com/daos-stack/raft/pull/10 (accessed May 28, 2023).

Follow-up on WRaft#118.8[1]: In RedisRaft#49[2], developers confirmed the issue, and it was fixed in RedisRaft#68[3] (November 15, 2021). The issue was also addressed in DaosRaft#10.

Discovery during the fix: It was discovered that WRaft had a test case function named **TestRaft_leader_sends_appendentries_when_node_next_index_was_compacted**, implying that AppendEntries should be sent when the next index was compacted. **This indicates that the developers had carefully considered and intended to handle this edge case in this manner**. However, without an oracle, human intuition can sometimes be incorrect, leading to erroneous test case design. When RedisRaft addressed the issue, the test case function name was changed to TestRaft_leader_sends_snapshot_when_node_next_index_was_compacted, indicating the correct behavior of sending a snapshot. Although DaosRaft did not change the function name of this test case during the fix, the content of the test case was modified to send a snapshot instead.

3.3. WRaft#118.4[4]

This is a protocol-level bug that leads to non-monotonicity of the current term.

Root Cause: When loading a snapshot, the node's current term is not compared with the current term in the snapshot. Instead, it is directly assigned the value from the snapshot. If the current term in the snapshot is smaller, it results in the current term becoming non-monotonic.

```
int raft_begin_load_snapshot(
    raft_server_t *me_,
    raft_term_t last_included_term,
    raft_index_t last_included_index)
{
    raft_server_private_t* me = (raft_server_private_t*)me_;
    // ...
    // Directly assigns the value to current_term without checking
if the assigned value is greater
    me->current_term = last_included_term;
}
```

Shortest Triggering Path:

1. A, B, and C are network partitioned.
2. B becomes the leader (term=1).

[1] "Fix bugs mainly caused by snapshot by tangruize · Pull Request #118 · willemt/raft," *GitHub*. https://github.com/willemt/raft/pull/118#:~:text=raft_send_appendentries_all%23L952 (accessed May 28, 2023).

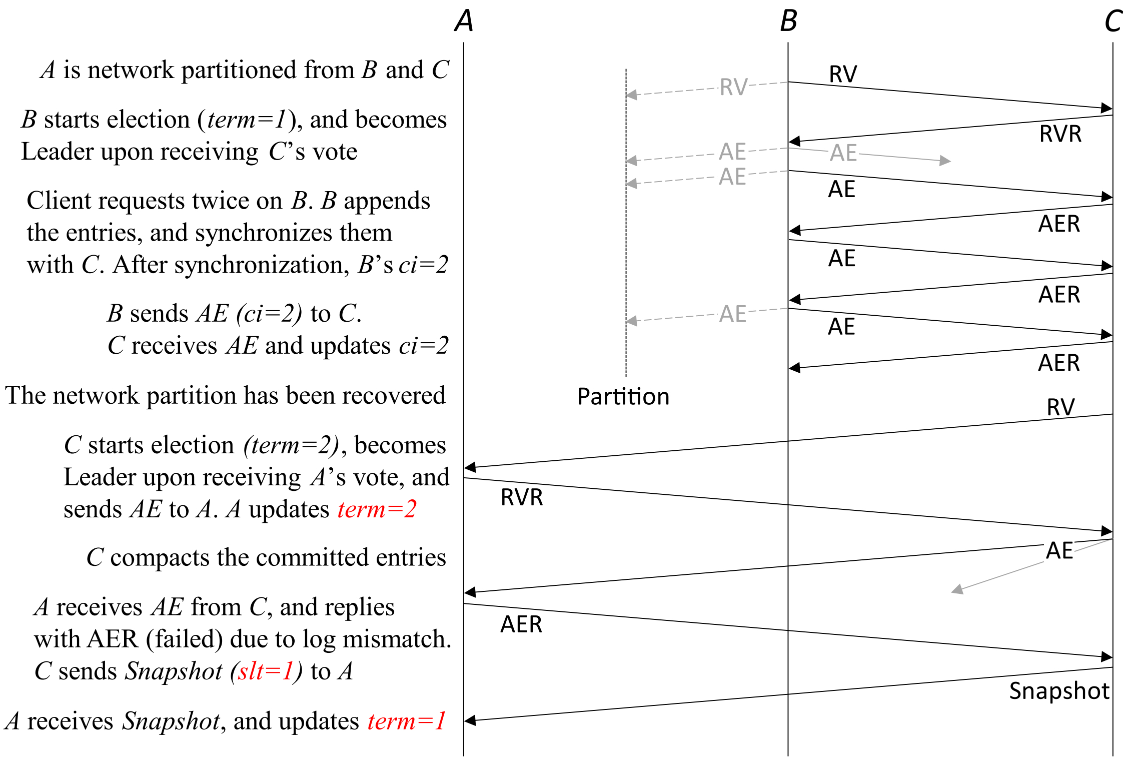
[2] "Found bugs mainly caused by snapshot · Issue #49 · RedisLabs/raft," *GitHub*. https://github.com/RedisLabs/raft/issues/49 (accessed May 28, 2023).

[3] "Snapshot RPC by tezc · Pull Request #68 · RedisLabs/raft," *GitHub*. https://github.com/RedisLabs/raft/pull/68 (accessed May 28, 2023).

[4] "Fix bugs mainly caused by snapshot by tangruize · Pull Request #118 · willemt/raft," *GitHub*. https://github.com/willemt/raft/pull/118#:~:text=raft_begin_load_snapshot%23L1383 (accessed May 28, 2023).

3. B receives two client requests and synchronizes with C. After synchronization, the commit index of B and C is 2.
4. Network partition between A, B, and C is restored.
5. C initiates an election, receives a vote from A, and becomes the leader (term=2). C sends AppendEntries to A. A updates its term to 2.
6. C takes a snapshot, removes two committed log entries from the log, and updates the snapshot's last term to 1.
7. A receives AppendEntries but fails to match because the log is empty. A responds with AppendEntriesResponse (failed).
8. C receives AppendEntriesResponse and sends a Snapshot (snapshot last term=1) to A.
9. A receives the Snapshot and updates its term to the snapshot's last term, which is 1.

Follow-up: In DaosRaft#10, most of the snapshot-related logic was rewritten, including the removal of the assignment to current_term. This bug was also fixed in RedisRaft@d23f390.



A is network partitioned from *B* and *C*

B starts election (*term=1*), and becomes Leader upon receiving *C*'s vote

Client requests twice on *B*. *B* appends the entries, and synchronizes them with *C*. After synchronization, *B*'s *ci=2*

B sends *AE (ci=2)* to *C*. *C* receives *AE* and updates *ci=2*

The network partition has been recovered

C starts election *(term=2)*, becomes Leader upon receiving *A*'s vote, and sends *AE* to *A*. *A* updates *term=2*

C compacts the committed entries

A receives *AE* from *C*, and replies with AER (failed) due to log mismatch. *C* sends *Snapshot (slt=1)* to *A*

A receives *Snapshot*, and updates *term=1*

4. DaosRaft

DaosRaft is a derivative version of WRaft, used in Intel's Daos storage system. This protocol library does not make assumptions about the underlying network, so the types of errors that can be injected include those allowed by UDP and TCP.

4.1. DaosRaft#70[1]

This is a code-level bug where, under certain specific configurations, an assertion is triggered during the leader election phase, causing the system to crash.

Root cause: In code block 1, if the condition me->timeout_elapsed < me->election_timeout is not met, the code will not enter the if branch. If the configuration sets the leader's heartbeat timeout to be greater than or equal to the election timeout, it can make the above condition not met.

Code block 1:

```
/* Reject request if we have a leader */
// Won’t enter this branch when timeout_elapsed >= election_timeout
if (me->leader_id != -1 && me->leader_id != vr->candidate_id &&
    me->timeout_elapsed < me->election_timeout)
{
    r->vote_granted = 0;
    goto done;
}
```

When the Leader receives a PreVote message, Code block 2 of the code requires that the Leader cannot vote for other nodes. However, when the condition in Code block 1 occurs and the Leader does not go to the "done" label, it proceeds to the voting code, triggering an assertion.

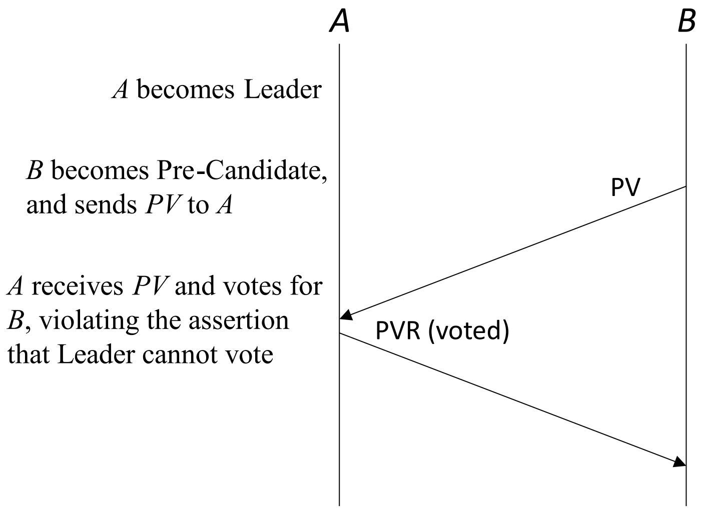
Code block 2:

```
if (__should_grant_vote(me, vr))
{

/* It shouldn't be possible for a leader or prevoted candidate to gr
ant a vote
     * Both states would have voted for themselves
     * A candidate may grant a prevote though */
    assert(!raft_is_leader(me_) && (!raft_is_candidate(me_) || me-
>prevote || vr->prevote));
}
```

Shortest Trigger Path:

1. Configure the Leader's heartbeat timeout to be equal to the minimum election timeout.
2. A becomes the Leader.
3. B becomes a Pre-Candidate and sends a PreVote to A.
4. A receives the PreVote. At this point, A's recorded timeout_elapsed equals election_timeout. A votes for B's PreVote, triggering the Assertion.

[1] “Reject request vote if self is leader by liw · Pull Request #70 · daos-stack/raft,” *GitHub*. https://github.com/daos-stack/raft/pull/70 (accessed May 29, 2023).


A
B
A becomes Leader
B becomes Pre-Candidate, and sends PV to A
PV
A receives PV and votes for B, violating the assertion that Leader cannot vote
PVR (voted)

5. RedisRaft

RedisRaft is a derivative version of WRaft and serves as a pluggable module for Redis. This protocol library does not make assumptions about the underlying network, thus allowing injection of error types permissible under UDP and TCP.

As of now, no bugs have been identified in RedisRaft.

6. Xraft

Xraft is a Raft protocol library implemented in Java, utilizing RPC with a TCP-based underlying network, thereby only susceptible to network partition errors.

6.1. Xraft#33[1]

This is a protocol-level bug that can lead to the emergence of multiple Leaders with the same term, which is the most serious violation of the Raft protocol.

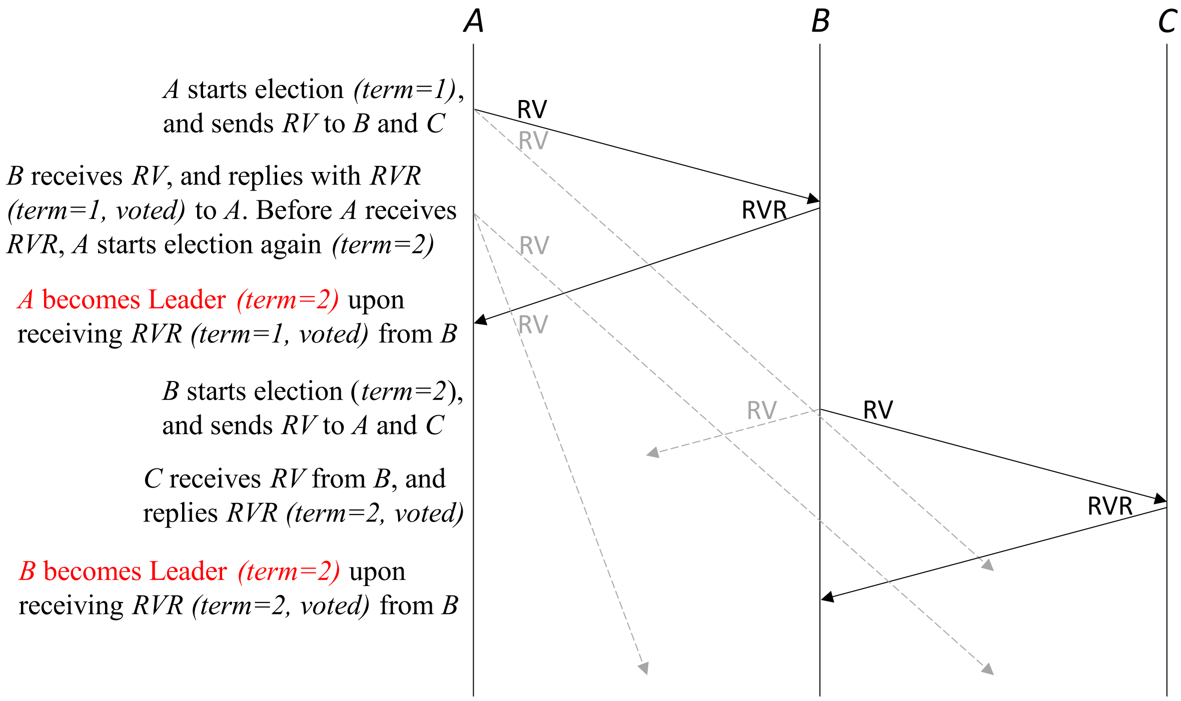
The root cause lies in Candidates not checking the size of the term in the RequestVoteResponse messages they receive. If the term is smaller, the Candidate still accepts the RequestVoteResponse and becomes a Leader.

[1] "Missing check for `result.getTerm() == role.getTerm()` in `doProcessRequestVoteResult()` can result in two leaders with the same term. · Issue #33 · xnnyygn/xraft," *GitHub*. https://github.com/xnnyygn/xraft/issues/33 (accessed May 29, 2023).

```
private void doProcessRequestVoteResult(RequestVoteResult result) {
    // step down if result's term is larger than current term
    if (result.getTerm() > role.getTerm()) {
        becomeFollower(result.getTerm(), null, null, true);
        return;
    }
    // check role
    if (role.getName() != RoleName.CANDIDATE) {
        logger.debug("receive request vote result and current role is not candidate, ignore");
        return;
    }
    // do nothing if not vote granted
    if (!result.isVoteGranted()) {
        return;
    }
    // lacks a check for the term
    int currentVotesCount =
((CandidateNodeRole) role).getVotesCount() + 1;
    int countOfMajor = context.group().getCountOfMajor();
    logger.debug("votes count {}, major node count {}", currentVotesCount, countOfMajor);
    // ...
}
```

Shortest Trigger Path:

1. There is a significant delay in the network connections between A, B, and C. Dotted lines in the diagram represent network messages that have not yet reached their destination due to the delay.
2. A times out for the election (term=1) and sends RequestVote messages to B and C.
3. B receives RequestVote from A, updates its term to 1, and responds with RequestVoteResponse voting for A.
4. A times out for another election (term=2).
5. A receives RequestVoteResponse from B (term=1) and becomes the leader (term=2).
6. B times out for the election (term=2) and sends RequestVote messages to A and C.
7. C receives RequestVote from B and responds with RequestVoteResponse voting for B.
8. B receives RequestVoteResponse from C (term=2) and becomes the leader (term=2).



A starts election *(term=1)*, and sends *RV* to *B* and *C*

B receives *RV*, and replies with *RVR (term=1, voted)* to *A*. Before *A* receives *RVR*, *A* starts election again *(term=2)*

A becomes Leader *(term=2)* upon receiving *RVR (term=1, voted)* from *B*

B starts election (*term=2*), and sends *RV* to *A* and *C*

C receives *RV* from *B*, and replies *RVR (term=2, voted)*

B becomes Leader *(term=2)* upon receiving *RVR (term=2, voted)* from *B*