# Distributed Systems PRAC2

***Joan Maldonado & Josep Rodríguez***

***Consultants: Pedro Antonio García López & Manel Pérez Bondia***

# Table of Contents


Introduction to Raft ..... 3
Decisions in implementation ..... 3
- Multithreaded implementation ..... 3
- RMI Methods ..... 4
- Thread safety ..... 4

Key source code portions ..... 5
- RaftThread ..... 6
- HeartBeatSenderThread ..... 7
- Request ..... 9
- appendEntries ..... 9
- FollowerAppendEntries ..... 10
- grantVote ..... 11
- StartElection ..... 11

Testing the code ..... 13
How to run Raft Consensus algorithm's implementation ..... 13
Conclusions ..... 13
- Known limitations ..... 13

# Introduction to Raft

Raft is a consensus algorithm for managing a replicated log. Raft's main goal is to be more understandable than Paxos and also provides a better foundation for building practical systems.

In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered.

Thus, Raft provides an initial base to build upon algorithms with the need to replicate data in a consistent manner across a cluster.

This paper describes the implementation of Raft algorithm based on RMI invocations.

# Decisions in implementation

## Multithreaded implementation

Our starting point was the skeleton of a RMI cluster that offered the typical raft methods for other servers and clients to invoke:

- requestVote
- appendHeartbeat
- requestOperation

The initial implementation of this skeleton was aimed to take advantage of Java's Timer and TimerTask API's to implement Raft's workflow.

We decided to migrate this approach towards the implementation of different Thread's to gain higher control over the flow of the application. This was mainly due to the fact that Timer and TimerTask's API's do not grant that the cancellation of a task will ensure that it won't be ever executed again.

Furthermore, the aforementioned API didn't gave us the flexibility to decide exactly at which point in time the code would be ran.

In consequence, we created four different thread classes:

- CommitterThread, whose job is to commit the data that has been replicated safely across the cluster. It is created only once at the start of the cluster.
- HeartBeatSenderThread, who is responsible to send heartbeat messages to a single follower with the data to persist if any. This thread is created once for each follower when a server becomes a leader.
- LeaderWatcherThread, whose main goal is to detect that the leader is not sending heartbeats anymore. It is created alongside the committer thread just once in the lifetime of the node.
- VoteRequesterThread is the class in charge to request the vote of the rest of the cluster during a votation. Similarly to the heartbeat sender, it will be created once for every votation and possible follower.

All of the above extend a class named RaftThread that eases up some tasks that are

common across the threads defined previously.

## RMI Methods

Another decision taken during the implementation of the Raft algorithm was to split the RMI methods into different methods depending on the current state of the node. Thus we could safely modify the behaviour of a state without affecting the others.

## Thread safety

To ensure that the cluster remains thread safe, we have guarded using a synchronized block the code fragments that require access to the state of the raft cluster, as for example when evaluating whether an index is committed, or if a vote has already been granted for a given term.

## Key source code portions

# RaftThread

The threads that have been implemented to perform Raft's work flow all override from this abstract class that offers a simple API to a never ending thread.

```java
/**
 * Auxiliar thread that handles a loop inside its run method.
 *
 * @author josep
 *
 */
public abstract class RaftThread extends Thread {

	private AtomicBoolean stop = new AtomicBoolean(true);
	private AtomicBoolean sleeping = new AtomicBoolean(false);

	/*
	 * (non-Javadoc)
	 *
	 * @see java.lang.Thread#run()
	 */
	@Override
	public final void run() {
		// on entering run, set stop to false.
		stop.set(false);
		// If stop is true, finish the thread.
		while (!stop.get()) {
			if ( connected.get() ) {
				try {
					// Execute abstract operation that performs its work.
					onIteration();
				} catch (Exception e){
					// Nop
					e.printStackTrace();
				}
			}

			// Do not sleep if the thread has been stopped during
			// onIteration.
			if (stop.get()) {
				break;
			}

			// Sleep until needed again.
			doSleep();
		}
		onGoodBye();
	}

	/**
	 * Lets subclasses do things on end of the thread.
	 */
	protected void onGoodBye() {
		// Nop
	}

	/**
	 * Sleeps until the thread is needed again.
	 */
	private final void doSleep() {
		try {
			sleeping.set(true);
			long l = getAwakeTimestamp() - System.currentTimeMillis();
			// Minimum sleep for 10 millis.
			l = Math.max(l, 10l);
			sleep(l);
		} catch (InterruptedException e) {
			// Thread has been awoken by another one to start working ASAP.
		} finally {
			sleeping.set(false);
		}
	}

	/**
	 * Stops the thread.
	 */
	public final void stopThread() {
		stop.set(true);
		// To make it finish as soon as possible.
		awakeThread();
	}

	/**
	 * Awakes the thread if it is sleeping to start working ASAP.
	 */
	public final void awakeThread() {
```

```
82.                     // Interrupt if thread is sleeping needed.
83.                     if (sleeping.get()) {
84.                             interrupt();
85.                     }
86.             }
87.
88.             /**
89.              * Executed every time the thread has to be awakened.
90.              */
91.             protected abstract void onIteration();
92.
93.             /**
94.              * @return the timestamp at which the thread should be awoken.
95.              */
96.             protected abstract long getAwakeTimestamp();
97.     }
```

# HeartBeatSenderThread

As previously mentioned, this class is responsible to send appendEntries to the followers nodes.

As such, it has to send the correct data to each of them, and deliver the response of the follower according to Raft specifications.

```
1.  /**
2.   * Thread managing heartbeat sending to each host.
3.   */
4.  public class HeartBeatSenderThread extends RaftThread {
5.
6.          /** The host to whom we send him the heartbeats. */
7.          private Host host;
8.
9.          /** The interval that should be delayed between heartbeats. */
10.         private long interval;
11.
12.         /**
13.          * Default constructor takes a host and an interval to start sending
14.          * heartbeats.
15.          *
16.          * @param host
17.          * @param interval
18.          */
19.         public HeartBeatSenderThread(Host host, long interval) {
20.                 this.interval = interval;
21.                 this.host = host;
22.         }
23.
24.         /**
25.          * Sends a heartbeat to the given host.
26.          */
27.         @Override
28.         protected void onIteration() {
29.                 synchronized (guard) {
30.                         // the runnable can be ran if the term hasn't changed and
31.                         // localhost is leader
32.                         if ( !isLeader() ) {
33.                                 log("Stopping heartbeat thread because we are not the current leader.",
    WARN);
34.                                 stopThread();
35.                                 return;
36.                         }
37.                 }
38.
39.                 int lastLogIndex,prevLogindex;
40.                 long prevLogTerm,term;
41.                 List<LogEntry> entries;
42.
43.                 // Load in a guarded section
44.                 int nextIndexInt = nextIndex.getIndex(host.getId());
45.
46.                 synchronized (guard) {
47.                         // My term
48.                         term = getCurrentTerm();
49.
50.                         // Get the last index in our log.
51.                         lastLogIndex = persistentState.getLastLogIndex();
52.
53.                         // * If last log index ≥ nextIndex for a follower: send
54.                         // AppendEntries RPC with log entries starting at nextIndex
55.                         if ( lastLogIndex >= nextIndexInt ) {
56.                                 entries = persistentState.getLogEntries(nextIndexInt); // Starts at
    position 1.
57.                         }
```

```
58.                             // No entries to send.
59.                             else {
60.                                     entries = new ArrayList<LogEntry>();
61.                             }
62.
63.                             // Get prev index and term for the append.
64.                             prevLogindex = nextIndexInt - 1;
65.                             prevLogTerm = persistentState.getTerm(prevLogindex);
66.                     }
67.
68.                     try {
69.                             log("Starting to send heartbeat", DEBUG);
70.                             AppendEntriesResponse appendResponse = communication.appendEntries(host,
71.                                             term, leader, prevLogindex,
72.                                             prevLogTerm, entries, commitIndex); // Send entries from nextindex
   on.
73.
74.                             log("Sent heartbeat to " + host.getId(), DEBUG);
75.
76.                             if ( entries.size() > 0 && appendResponse.isSucceeded() ) {
77.                                     log ("Sent to " + host.getId() + " entries from index " + nextIndexInt + "
   containing : " + entries, WARN);
78.                             }
79.
80.                             // Controlling null pointer exception
81.                             if (appendResponse == null) {
82.                                     // Do not retry, the other host has returned null, but the
83.                                     // communication hasn't failed. This should never happen.
84.                                     log("<<<<<<<<< RECEIVED NULL APPEND RESPONSE FROM HOST " + host + ". THIS
   SHOULD NEVER HAPPEN.", ERROR);
85.                                     return;
86.                             }
87.
88.                             // If the other server has a higher term, we are out of date.
89.                             if (persistentState.getCurrentTerm() < appendResponse.getTerm()) {
90.                                     log("Host " + host + " had a higher term: " + appendResponse.getTerm(),
   WARN);
91.                                     // I've no leader
92.                                     synchronized (guard) {
93.                                             changeState(RaftState.FOLLOWER);
94.                                             setLeader(null);
95.                                             persistentState.setCurrentTerm(appendResponse.getTerm());
96.                                     }
97.                                     return;
98.                             }
99.
100.
101.                            //                              * If successful: update nextIndex and matchIndex
   for follower (§5.3)
102.                            if ( appendResponse.isSucceeded() ) {
103.                                    log("<<<<<<<<<<< AppendEntries accepted from host " + host,    INFO);
104.                                    synchronized (guard) {
105.                                            // Update indexes
106.                                            if ( matchIndex.getIndex(host.getId()) < lastLogIndex ) {
107.                                                    log("Updating match index to " + lastLogIndex,WARN);
108.                                                    matchIndex.setIndex(host.getId(), lastLogIndex);
109.                                                    // Only after recalculate commit index is this one.
110.                                                    recalculateCommitIndex();
111.                                            }
112.                                            if ( nextIndexInt < lastLogIndex + 1) {
113.                                                    log("Updating next index to " + (lastLogIndex + 1),WARN);
114.                                                    nextIndex.setIndex(host.getId(), lastLogIndex + 1);
115.                                            }
116.                                    }
117.                            }
118.                            //                              * If AppendEntries fails because of log
   inconsistency:
119.                            //                              decrement nextIndex and retry (§5.3)
120.                            else {
121.                                    log("<<<<<<<<<<< AppendEntries rejected from host " + host + " decreasing
   its index to " + (nextIndexInt - 1), ERROR);
122.                                    synchronized (guard) {
123.                                            nextIndex.decrease(host.getId());

124.                                    }
125.                            }
126.
127.
128.                    } catch (Exception e) {
129.                            // Retry if some error has happened (probably IOException).
130.                            log ("Exception happened during the heartbeat delivery: " + e.getClass(), INFO);
131.                            //                              e.printStackTrace();
132.                            return;
133.                    }
134.            }
135.
136.            @Override
137.            protected long getAwakeTimestamp() {
138.                    // Now + interval.
139.                    return System.currentTimeMillis() + interval;
140.            }
```

# Request

As previously introduced, all the RMI methods have been splitted depending on the current state of the node.

In this specific request, the client will be redirected to the leader if it is not us, or it will be performed by ourselves if we are the current leader.

```
1.  public RequestResponse Request(Operation operation) throws RemoteException {
2.          // If follower and candidate, redirect him to the last leader known, so
3.          // it will retry.
4.          log("Received request from client", INFO);
5.          switch (state) {
6.          default:
7.          case FOLLOWER:
8.          case CANDIDATE:
9.                  return new RequestResponse(leader, false);
10.         case LEADER:
11.                 log("Leader performs the request", INFO);
12.                 return leaderRequest(operation);
13.         }
14. }
```

# appendEntries

Once more, this RMI method is splitted into a method for each of the available states of a node in the cluster.

```
1.  /*
2.   * (non-Javadoc)
3.   *
4.   * @see recipesService.raft.Raft#appendEntries(long, java.lang.String, int,
5.   * long, java.util.List, int)
6.   */
7.  @Override
8.  public AppendEntriesResponse appendEntries(long term, String leaderId,
9.                  int prevLogIndex, long prevLogTerm, List<LogEntry> entries,
10.                 int leaderCommit) throws RemoteException {
11.         log(" Received appendEntry for state: " + state, DEBUG);
12.         log ("With prev index: [" + prevLogTerm + "," + prevLogIndex + "]" , DEBUG);
13.         log ("With entries : " + entries , DEBUG);
14.         switch (state) {
15.         case FOLLOWER:
16.                 return followerAppendEntries(term, leaderId, prevLogIndex,
17.                                 prevLogTerm, entries, leaderCommit);
18.         case CANDIDATE:
19.                 return candidateAppendEntries(term, leaderId, prevLogIndex,
20.                                 prevLogTerm, entries, leaderCommit);
21.         case LEADER:
22.                 return leaderAppendEntries(term, leaderId, prevLogIndex,
23.                                 prevLogTerm, entries, leaderCommit);
24.         }
25.
26.         // Should never happen.
27.         return null;
28. }
```

# FollowerAppendEntries

This is the actual implementation of the flow that a follower node has to execute when a leader tries to append entries to him.

It is coded and commented to match Raft's specification document instructions on this particular method.

```
1.  /**
2.   * Follower code on invokation of appendEntries from the leader.
3.   * Invoked by leader to replicate log entries; also used as heartbeat.
4.   * @param term : leader's term
5.   * @param leaderId : so follower can redirect clients
6.   * @param prevLogIndex : index of log entry immediately preceding new ones
7.   * @param prevLogTerm : term of prevLogIndex entry
8.   * @param entries : log entries to store
9.   * @param leaderCommit : leader's commitIndex
10.  * @return AppendEntriesResponse: term (currentTerm, for leader to update itself),
11.  *       success (true if follower contained entry matching prevLongIndex and prevLogTerm)
12.  */
13. private AppendEntriesResponse followerAppendEntries(long term,
14.                     String leaderId, int prevLogIndex, long prevLogTerm,
15.                     List<LogEntry> entries, int leaderCommit) {
16.             synchronized (guard) {
17.                     // From raft pdf: Receiver implementation:
18.                     // 1. Reply false if term < currentTerm (5.1)
19.                     if ( term < persistentState.getCurrentTerm()) {
20.                             return new AppendEntriesResponse(persistentState.getCurrentTerm(), false);
21.                     }
22.
23.
24.                     // If leader with higher term, update ourselves.
25.                     else if (term > persistentState.getCurrentTerm()) {
26.                             log("Received append entry with newer term.", INFO);
27.                             synchronized (guard) {
28.                                     persistentState.setCurrentTerm(term);
29.                                     setLeader(leaderId);
30.                             }
31.                     }
32.
33.                     // Same term, with different leader update it.
34.                     else if (term == persistentState.getCurrentTerm()) {
35.                             log("Received append entries for my term. Updating leader if needed", DEBUG);
36.                             synchronized (guard) {
37.                                     setLeader(leaderId);
38.                             }
39.                     }
40.
41.                     // If not a stale leader, count from now the heartbeat.
42.                     onHeartbeat();
43.
44.                     // 2. Reply false if log doesn't contain an entry at prevLogIndex
45.                     // whose term matches prevLogTerm (5.3)
46.                     if ( persistentState.getTerm(prevLogIndex) != prevLogTerm ) {
47.                             // 3. If an existing entry conflicts with a new one (same index
48.                             // but different terms), delete the existing entry and all that
49.                             // follow it (5.3)
50.                             if ( prevLogIndex > 0 ) {
51.                                     log("Deleting invalid entries from log.", WARN);
52.                                     persistentState.deleteEntries(prevLogIndex);
53.                             }
54.                             log("Rejecting append entry because term " + prevLogTerm + "!=" +
    persistentState.getTerm(prevLogIndex), ERROR);
55.                             return new AppendEntriesResponse(persistentState.getCurrentTerm(), false);
56.                     }
57.
58.                     // 4. Append any new entries not already in the log
59.                     for ( LogEntry e : entries ) {
60.                             persistentState.appendEntry(e);
61.                             log("Appended entry " + e + " at position " + persistentState.getLastLogIndex(),
    WARN);
62.                     }
63.
64.                     // 5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, last log index)
65.                     if ( Math.min(leaderCommit, persistentState.getLastLogIndex()) > commitIndex ) {
66.                             commitIndex = Math.min(leaderCommit, persistentState.getLastLogIndex());
67.                             log("Commit index updated to " + commitIndex + " from leader. Start committing
    data." , WARN);
68.                             // Start committer thread.
69.                             committer.awakeThread();
70.                     }
71.                     // Ok to leader.
72.                     return new AppendEntriesResponse(persistentState.getCurrentTerm(), true);
73.             }
74. }
```

# grantVote

This method is called by the RequestVote RMI implementation, and returns a boolean representing whether the vote is granted to the candidate asking for it or not.

```
1.  /**
2.   * Guarded method to obtain a vote.
3.   *
4.   * @param term
5.   * @param candidateId
6.   * @param lastLogIndex
7.   * @param lastLogTerm
8.   * @return
9.   */
10.  private boolean grantVote(long term, String candidateId, int lastLogIndex, long lastLogTerm) {
11.          synchronized (guard) {
12.                  // Receiver implementation:
13.                  // 1. Reply false if term < currentTerm (§5.1)
14.                  if ( term < getCurrentTerm() ) {
15.                          log("Not accepting other leader because his term is less than ours.", WARN);
16.                          return false;
17.                  }
18.
19.                  // If higher term, update our own. This will clean the voted for object.
20.                  if ( term > persistentState.getCurrentTerm() ) {
21.                          persistentState.setCurrentTerm(term);
22.                          changeState(RaftState.FOLLOWER);
23.                          log("Updating my term to " + term, INFO);
24.                  }
25.
26.                  //                                  2. If votedFor is null or candidateId,
27.                  if ( persistentState.getVotedFor() != null && !
    persistentState.getVotedFor().equals(candidateId) ) {
28.                          log("Not granting vote because i've already voted to " +
    persistentState.getVotedFor(), INFO);
29.                          return false;
30.                  }
31.                  //                           and candidate's log is at least as up-to-date as receiver's log,
    grant vote (§5.2, §5.4)
32.                  if ( fresherLog(lastLogIndex, lastLogTerm) ) {
33.                          log("Not granting vote because our term is fresher: "
34.                                                  + "[" + persistentState.getLastLogTerm() + "," +
    persistentState.getLastLogIndex() + "]"
35.                                                  + " > "
36.                                                  + "["    + lastLogTerm + "," + lastLogIndex + "]", WARN);
37.                          return false;
38.                  }
39.
40.                  // If we are here we can grant him the vote.
41.                  persistentState.setVotedFor(candidateId);
42.
43.                  // Add some time to election timeout.
44.                  leaderWatcher.onHeartbeat();
45.
46.                  log("Granting my vote to " + candidateId, INFO);
47.                  return true;
48.          }
49.  }
```

# StartElection

This method is called by the leader watcher thread when it detects that the election timeout has passed without any valid leader sending a heartbeat. It will start the election.

```
1.  /*
2.   * Leader election
3.   */
4.  private void startElection() {
5.          // Steps
6.          synchronized ( guard ) {
7.                  // 2-Change to candidate state
8.                  if (state == RaftState.CANDIDATE) {
9.                          log(">>>>>>>> The election ended without choosing a leader, therefore, it will be
    restarted <<<<<<<<<< ", INFO);
10.                 } else {
11.                         log(">>>>>>>> Starting election on host " + getServerId(), INFO);
12.                 }
13.
14.                 // 1-Increment current term
15.                 persistentState.nextTerm();
16.                 long term = persistentState.getCurrentTerm();
```

```
17.                     log("Incrementing my term to " + term, INFO);
18.
19.                     // 2 - Clear list of received votes
20.                     this.receivedVotes = new HashSet<Host>();
21.
22.                     // 3 - Change state
23.                     changeState(RaftState.CANDIDATE);
24.                     setLeader(null);
25.
26.                     // 4 - Vote for self
27.                     this.persistentState.setVotedFor(getServerId());
28.                     this.receivedVotes.add(localHost);
29.
30.
31.                     // Stop previous vote requesters
32.                     for ( RaftThread t : voteRequesters ) {
33.                             t.stopThread();
34.                     }
35.                     voteRequesters.clear();
36.
37.                     // Start voteRequesters.
38.                     for (Host h : otherServers) {
39.                             VoteRequesterThread t = new VoteRequesterThread(h,
    persistentState.getCurrentTerm());
40.                             voteRequesters.add(t);
41.                             t.start();
42.                     }
43.             }
44.     }
```

# Testing the code

To ensure the correct behaviour of every part of the implementation, we followed a structured test plan, consisting in:

- Every phase is tested separately.
  - Test phase without disconnections
  - Test phase adding the disconnection parameter demanded on the assessment
- Test the overall application in our local machines
- Test the application in Amazon's machines provided by the consultants
- Test the application in the SDLAB

To perform every phase of the test plan, we executed our implementation using the script "start.sh" provided by the consultants, adding to it the correct parameters to run the desired phase to test. In cases where it was necessary, we modified the config file config.properties.

# How to run Raft Consensus algorithm's implementation

To build our implementation, we followed thoroughly the structure given by the consultants of this subject and, therefore, it can be run using the same commands and parameters as per the consultant's implementation.

In addition to this, it can be run with the requested parameters set by the consultants in config.properties file.

# Conclusions

This assignment provided us the opportunity of understanding how the basic structure of a distributed system is constructed and test how it works on a real environment. We had to deal with the decisions related to the pursuit of a better performance in our implementation.

Also, we've seen the importance of the KISS principle (Keep it Simple, Stupid). Raft has as its main goal to offer a simple yet effective approach to create a Consensus Algorithm. Even with the efforts of the creators, we've struggled during weeks to get all the cluster working properly. If we've had to implement Paxos that is not aimed towards simplicity, the effort required to obtain the same outcome would have probably been considerably higher.

## Known limitations

During the evaluation of the Raft algorithm some limitations have been discovered that should be known to the reader.

First of all, the most common cause for an error during a run is for a given process to be

disconnected by the testing environment without being connected again to the cluster.

In this scenario, the node's log will fall behind the rest of the cluster and on comparing it with the rest of the cluster will be marked as an invalid result. Up to the day of the composition of this deliverable no work around has been found to force a node to reconnect prior to the tear down of the system.

Furthermore, during some of the many tests that have been performed on the platform log divergence has been found. Nevertheless, this minor divergences have been impossible to debug as the only mean to detect the core of the issue is through the platform's output and it appeared to be correct during the whole execution.

Finally, the probability of a run to end with log divergences was detected to be higher as the probability of disconnection of cluster node's increased. For the initial value set in the file *config.properties*, the probability of log divergence was actually 0 (meaning that it hasn't occurred to us in all the tests performed with the given disconnection probability), but as we increase the probability of disconnection (up to 0.25 was tested) the possible log divergences increased in concordance.