Assignment 5
Distributed Algorithms

Version: March 23, 2021

Prerequisites

1. The assigned readings in module 5 on Canvas
2. Lecture videos from Canvas/ or class
3. Running and understanding the examples listed on the Canvas Module 5 page

Learning outcomes of this assignment are:

1. Understand basics of Distributed Algorithms
2. Evaluate distributed algorithms
3. Understand the basics of Consensus algorithms

Preliminary things

I strongly advise you to work on Git and GitHub, to version control and also to practice.
If you work on GitHub make sure your repository is private.
Submit your assignment as always on GitHub in the appropriated directory.
Please watch the videos supplied with this assignment to get some more insight.

What you definitely need:

1. Structure: you will have to create two programs one for each Activity. So your assignment 5 folder should have two subdirectories called **Sorting** and **Leader**. Each project needs a README.md and a build.gradle file.
2. A README.md for each project
   a) Design your calls and user interaction in a way that they are easy. Remember we have over 100 assignments to grade, design it so it is easy for you, most of this is given anyway.
   b) More details for the Readme will follow in the activities directly.

1 Activity: Distributed Algorithm: Sorting (30 points)

Background

For this activity you will use the code provided with the Assignment for activity 1.
This is not about implementing things but more about working with a simple distributed algorithm and evaluating the pros and cons.

You will still need to submit your code in an Activity 1 folder, so we see if you made any adjustments and see your test cases.
Some basics:

- Branch: is a node that splits an array in half and gives it to its two children (children can be branch or Sorter)
- Sorter: uses a priority queue to sort the given array
- Branch always pulls the first value out from the children from the priority queue, compares it and puts the smaller in its own queue
- Repeat step 3 until children queues are empty

You should not need to change the code much but you can if you like to for testing. For your test cases you of course need to create new arrays.

Task 1: Getting started

First analyze, understand and run the given code (on localhost – so all nodes on localhost). You should have a good understanding of what is going on before you continue. Open up Wireshark to look at the traffic. Do the steps below and answer the questions in your Readme under a Task 1 section, number the questions according to the number in the following list (eg. Task 1, Point 1):

1. Explain the main structure of this code and the advantages and disadvantages of the setup of the distributed algorithm.
2. Run the code with different arrays to sort (different sizes, numbers etc.) and include code to measure the time (you can just enter start and end times). In your Readme describe your experiments and your analyzes of them. E.g. why is the result as it is? Does the distribution help? Why, why not? See this as setting up your own experiment and give me a good description and evaluation.
3. Experiment with the "tree" setup, what happens with more or less nodes when sorting the same array and different arrays? When does the distribution make things better? Does it ever make things faster? As in the previous step experiment and describe your experiment and your results in detail.
4. Explain the traffic that you see on Wireshark. How much traffic is generated with this setup and do you see a way to reduce it?

Task 2: Running it outside of localhost

Right now the program runs only on localhost. Run the program not just on localhost, e.g. branch on your local computer, then two sorters on AWS. So that traffic needs to leave your computer.
Open up Wireshark to look at the traffic. Answer the questions in your Readme under a Task 2 section:

1. Do you expect changes in runtimes? Why, why not?
2. Do you see a difference how long it takes to sort the arrays? Explain the differences (or why there are not differences)

Task 3: How to improve

Based on the questions above you hopefully have a good idea where time is lost.

1. Where is the most time lost and could you make this more efficient?
2. Does it make sense to run the algorithm as a distributed algorithm? Why or why not?

2 Activity: Leader Election (70 points)

You have a given Peer-2-peer network which is meant as a very simple chat tool. See the video and provided code. Video is linked in the Readme.
You should play with this example for a bit before getting started and decide on some things (these are questions for you, no answer in the Readme needed):
- What protocol do you plan on using? (JSON, Protobuf) - Do you want to stick to the simple BufferedReader/PrintWriter or exchange it with something you like better? - Understand where the client and where the server parts are in the code - Do you want to keep this code or make adjustments or rather start from scratch (not sure I advise that but it is up to you)
(8 points) As always you will need a Readme which has a little screencast where you show your project in action, explain everything we need to know about it. Also explain your project in your Readme and which requirements you were able to fulfill and explain your protocol.
(4 points) We want to be able to run the client and server exactly as defined in the Readme and as is in Gradle. You should not need to change that. If you do for some reason, please ask first on Slack so we are aware and can discuss options.

Assumptions

You can assume that the nodes are initially started correctly, meaning first the leader than the nodes and that only one leader is started when you start up all your nodes.

Requirements

The requirements will add up to more than 58 points, so you can get some extra credit if you get everything done. In here we will also be grade with partial credit if you get part of it going and we can see these parts (based on when we run it, your Readme and your video). If we cannot run it we cannot award points.

1. (3 points) Send the message "Hello what's up?" which will not work in the chat. Figure out why it does not work and fix it.
2. (8 points) Check that if a non leader node (pawn) disconnects that it will be removed correctly, meaning if one peer detects another non leader node not responding, the peer should inform the leader. The leader should then inform all other peers about the not responding node. If the leader detects it, it will just inform the others to remove this node.
3. (5 points) Make sure a peer is not entered twice into the list of peers, it is your job to figure out when this situation can occur.

4. (15 points) Instead of just a chat we also want to be able to have every peer save a list of jokes and add new jokes. Any peer can add a joke – a user calls "joke" in the terminal, which should be caught by the peer when it detects user input and the peer should then ask for the joke. The user enters a joke, which would then be send to the leader node. The leader node would then add that joke to its potential list of jokes (make sure it is thread save) and would inform all peers to now also add that joke (would need to send the joke to them) to their potential list. The peers would **all** (all that are currently available) have to send their ok and save the joke as a potential add on! Only if all send their ok is the joke actually added to the real joke list. The leader would have to send another "ok now add" (does not have to be that, base it on your protocol) to the peers which would then mean the joke is actually added to the list.

   If not all of the nodes send their ok, then the joke will be discarded on all nodes.

   We now created a simple consensus, since all nodes need to "agree" on adding the joke. Congratulations!

5. We want to be able to start a leader election in case the leader goes off-line.
   a) (15 points) Leader election: you can go by highest peer number (if you assign them numbers), go by each peer choosing a random number and comparing that number (smallest, highest wins), go by whoever detect the leader is "dead" tries to become leader or even Raft (but that is more complicated). Nodes need to communicate and a node can only become leader by a specified rule you define (explain it in your Readme). See the lecture videos for some suggestions and also the linked documents.
   b) (4 points) The new leader is now responsible for the handling of the jokes list. If nodes still have potential jokes they will be discarded at this stage.
   c) (5 points) Your program needs to be able to handle that any node might not respond (leader or not). Your program should not come to a standstill if a node "dies". If the leader fails a new one needs to be chosen. You should try to set this up as fail safe as you can, e.g. have the leader send heart beats if no heart beat is received the nodes will elect a new leader. If you did the above this hopefully works already.
   d) (5 points) New nodes can join the network, they can contact the new leader (thus setting the correct leader when calling Gradle), the leader will then send them the current joke list.
   e) (5 points) New nodes can join the network by contacting ANY of the nodes and that node will forward the request to the leader (who will then inform everyone about the new peer)

6. Overall good error handling, no crashing, users are informed of what is going on. Should go without saying.

Submission

Push your Assignment 5 folder to GitHub and make sure that you also include the link to the folder on Canvas in your submission.