Project 3 Report

Christian Webber
May 7th, 2015
CptS 464, Section 1
Professor David Bakken

In Project 3, we were asked to implement a logical clock in a distributed system and to make a consistent global cut using the Chandy-Lamport snapshot algorithm. In implementing the distributed system, we were asked to use a common scenario, where Caesar must evenly distribute between his three generals and himself an initially random amount of soldiers in each of the general's camps.[1] A full copy of the scenario can be found in Appendix 1. In this report, an explanation of how to run the software, example outputs, a summary of the algorithms involved, and a list of challenges are provided.

Assuming one has access to the WSUCPTS464 projects on DeterLab, running the project is a simple process. First, the project (**cwebb-proj3**) must be swapped in. Once it has been swapped in, a user should login to each of the nodes (*Caesar*, *Brutus*, *Operachorus*, and *Pompus*) using ssh on 4 separate terminals.[2] Once logged into each of the nodes, simply run the correct bash file for the node in the following order:

Pompus: ./RunPompus.sh
Operachorus: ./RunOperachorus.sh
Brutus: ./RunBrutus.sh
Caesar: ./RunCaesar.sh [Cut State]

The bash script will automatically install the necessary prerequisite (the **Go** programming language) onto the node if it hasn't already been installed, although this process isn't entirely automated so some user interaction will be required during the installation. The order of when *Pompus*, *Operachorus*, and *Brutus* are run does not particularly matter, but *Caesar* must be run last. An optional argument can be provided to specify the specific state (for Caesar) for the Cut to be made. If this argument is left out, a random number between 0 and 5 will be chosen instead. The most common cause of a run failing (if it hangs after sending a message) is that the IP addresses have changed (this happened once during testing, but on every other run it remained stable as long as the NS file remained stable. If necessary, the IP addresses can be changed by simply changing the value of the **Address** field in the respective settings file for the node. These IP addresses can be found by going onto the **Details** tab on the experiment's page. The settings files are **setbrutus**, **setcaesar**, **setoperachorus**, and **setpompus**, for *Brutus*, *Caesar*,

[1] David Bakken, *CptS 464/564 Project #3 Implementation of "logical clock" and making "consistent global cut"*, (2015), 2

[2] **Brutus**: Brutus.cwebb-proj3.WSUCPTS464.isi.deterlab.net
Caesar: Caesar.cwebb-proj3.WSUCPTS464.isi.deterlab.net
Operachorus: Operachorus.cwebb-proj3.WSUCPTS464.isi.deterlab.net
Pompus: Pompus.cwebb-proj3.WSUCPTS464.isi.deterlab.net

Operachorus, and *Pompus* respectively. However, no changes should be necessary for the functioning of the system. After the scenario is completed (each process will automatically end when Caesar says everyone is evenly distributed), a file will be made that will contain the information on the cut that was made during that run. This file is called **cutfile.txt**. A fully copy of all of the source code is provided in the folder packages with this report.

While the system is running, Caesar sends orders to his generals to attempt to evenly redistribute their troops. At the beginning of a run, each general has a random amount of troops of each type (Archers, Catapults, Cavalry, Infantry, and Spearmen) between 1 and 100. This value is divided by four (4) to determine the amount of soldiers each general should have, and a remainder value is also calculated to determine how the leeway each number can have for an individual general in regards to extra troops. By the end of the run, each general should have approximately the same amount of troops available in each category. An example of the initial and final states of the system can be found below.

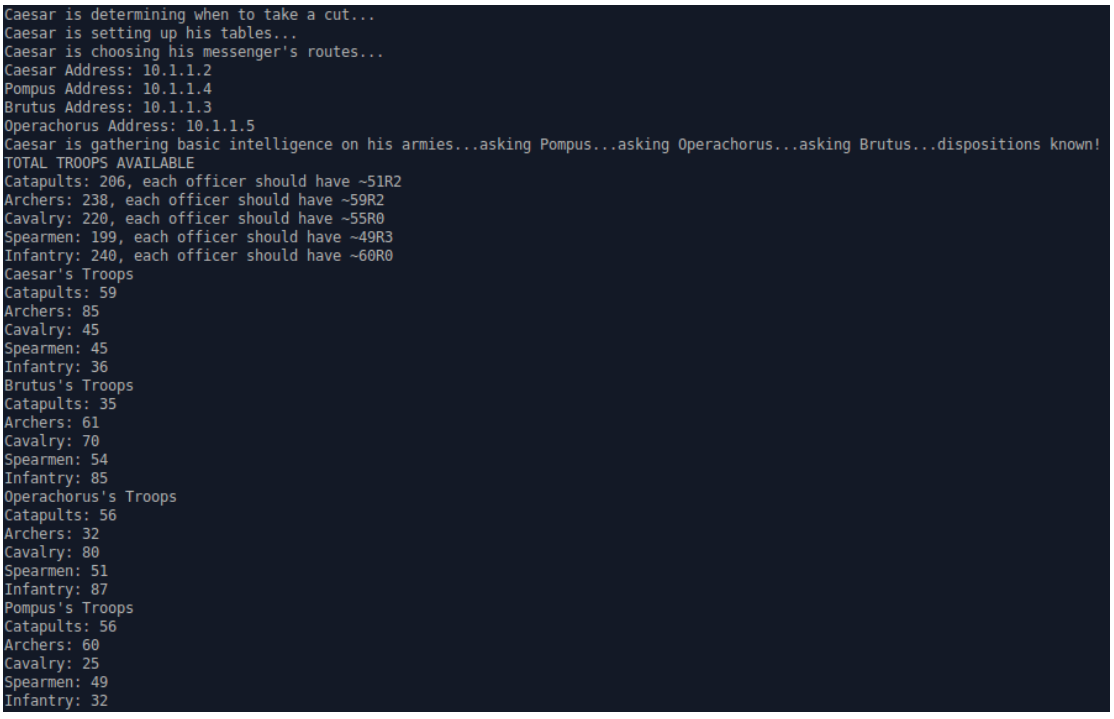

Figure 1 - Pre-Run Distribution

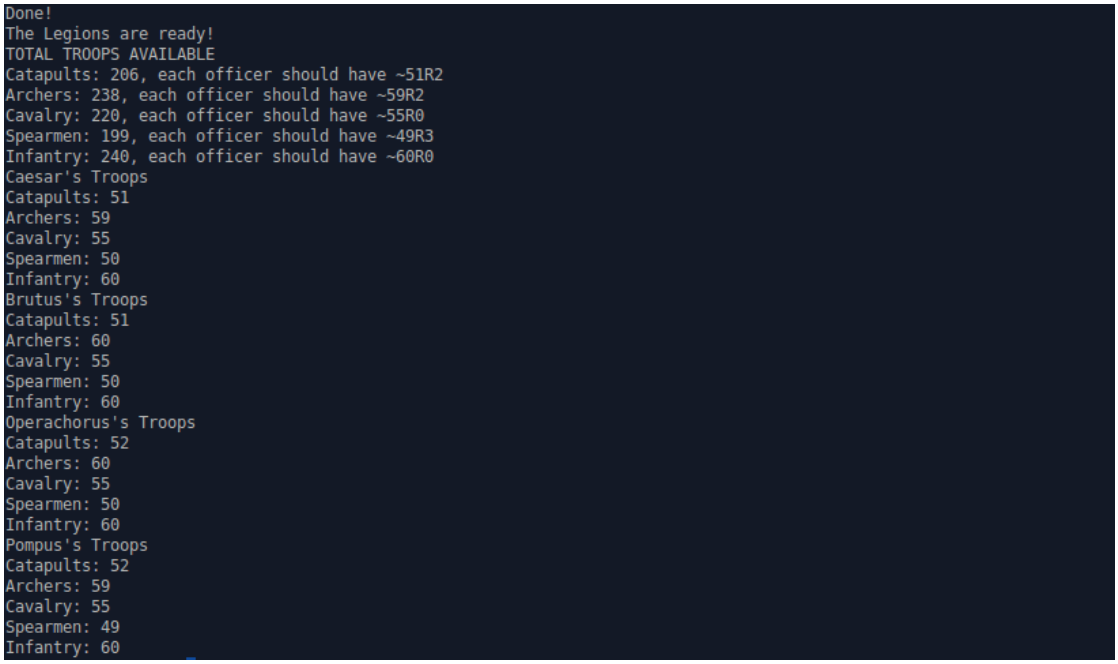

Figure 2 - Post-run Distribution

In Figure 1, we can see the initialization process at the beginning, and then a list of the total troops, including how many each general should have and how many extra they can have. Following this, we can see the troop counts of each unit type for each general. In Figure 2, we again see the total troops count and what the split should be, as well as the final distribution of troops for each general. Note that the values for each unit type for each general matches the approximate amount that they should have according to the earlier calculations. At some point during the scenario, a "cut" will be taken of the system. The cut uses the Chandy-Lamport snapshot algorithm to make a consistent global cut (or picture) of the system at a 'moment' in time.[3] A basic overview of this algorithm is available in Appendix 2. This snapshot will show the global state of the entire system at a single "moment," and will be exported to an external file, **cutfile.txt**, to allow it to be easily viewed at any moment. Examples of a snapshot are displayed on the next page.

[3] K. Mani Chandy and Leslie Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, (1985).

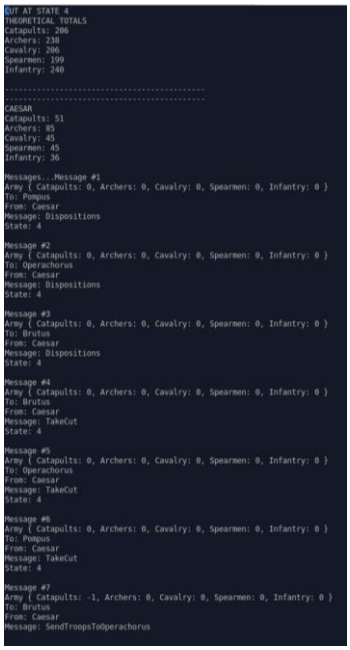

Figure 3 - Cut Log Part 1

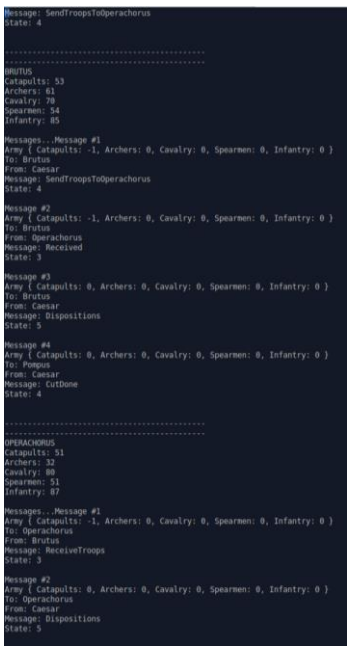

Figure 4 - Cut Log Part 2

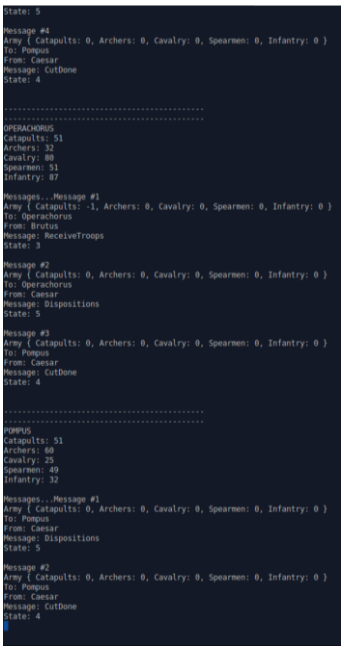

Figure 5 - Cut Log Part 3

The log file displays the theoretical number of troops that there should be in the army at the beginning. Below, each general is listed (separated by two rows of dashes), with their army count at the beginning of the cut. In addition, a full listing of each message passed during the cut to that general is listed below their army counts. Each general has 2 bi-directional channels coming into it, for a total of 8 channels (and 16 total messages in the system being possible during a cut). As can be seen in the log files, the total numbers of each category of unit for each general sums up to the estimated totals, proving that the cut is a consistent one. Additional screenshots (full-size) are provided in the 'Screenshots' folder packages with this report.

The single biggest challenge that was experienced during this project was unfamiliarity with the **Go** programming language. Although it is similar to a combination between **Python** and **C**, both of which I have experience with, it still took a while to become familiar enough with it to create the programs for this project. This also limited me somewhat in creating the programs in particularly well-designed ways. I went through 3 complete rebuilds before finally settling on one that allowed me to complete the programs, and partial rebuilds of major sub-systems occurred along the way. I was also limited in my abilities to use some more advanced libraries and techniques due to this unfamiliarity, as well as the strange lack of a while loop in **Go**. Right before turning this project in, I ran into another problem as I realized that I had forgotten to give Caesar an army also, so I had to restructure some of the code (which was hard-coded with the assumption that there was only 3 generals with armies) to handle the addition

of an extra army. Another challenge that I experienced was in using DeterLab. Initially it was confusing to get around, as DeterLab's provided tutorials are fairly dense. While this is good once you're used to getting around the system, a lighter weight tutorial is necessary for the basics. Luckily, this challenge was solved, mostly thanks to the help and basic tutorials provided by Jeremy and Zachariah. I feel that it would have been better if we had had more experience with using DeterLab before this project. The final challenge was the algorithm for the scenario. For some reason, it took much longer than it should have to come up with a half-way decent algorithm to redistribute the troops. Although I know better ones now, and would have been able to use them if I was more familiar with Go or was using a language I was more familiar with, this remained a significant early challenge for completing the project.

Overall, the project proved to be an interesting and fun challenge. While I might wish for some changes to the project (an easier scenario so that I'd have had more time to focus on the algorithm, for example), it was still a decent project. I also feel like I have learned a lot about distributed systems, much more than I did in any previous assignment, as well as in making cuts to a distributed system. Copies of the source code for the nodes as well as screenshots of the program in action are provided in folders packaged with this report (the Screenshots folder contains various screenshots of a run, and the Source Code folder contains the complete source code necessary for running the project and building it on DeterLab).

Sources

David Bakken, *CptS 464/564 Project #3 Implementation of "logical clock" and making "consistent global cut"*, (2015), 2. http://www.eecs.wsu.edu/~cs464/assignments/Project3.pdf

David Bakken, *Slides for Chapter 14: Time and Global States*, (2015), 13-15. http://www.eecs.wsu.edu/~cs464/slides/Chap14Slides.pdf

K. Mani Chandy and Leslie Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, (1985). http://www.eecs.wsu.edu/~cs464/assignments/chandy-lamport-snapshot-paper.pdf

Appendix 1

Taken from the CptS 464/564 Project 3 Specifications by Professor David Bakken:

"Caesar summoned three of his favorite generals Brutus, Pompus, and Operachorus to share his latest strategy to attack the Gaul village. "This time, we'll attack them from four directions", said Caesar. The generals agreed and all four armies began their march towards the village right away. Upon arriving near the village, the generals realized that the village is surrounded by four hills. So, Caesar ordered each army to set camp on each of the hills and wait for further orders. Caesar is a highly methodical person and wants all generals to be in-sync. Roman generals are known for their excellent vector clock skills so Caesar plans to use it to synchronize his armies. As his first order, Caesar sends a message to all three generals to report him back with the count of their army units – catapults, archers, cavalry, spearmen, and infantry. He wants each army 1 to have an equal number of units. Those days, the only form of communication was to use a human messenger. Messengers are usually the fastest soldiers in the army but they are not reliable. Each general obeys Caesar's order and sends their respective unit counts in a single message. Thus assume trustworthy generals. Upon receiving the unit counts from each general, Caesar calculates how many units should be relocated to and from each army and instructs the respective generals. For example, one such relocation command could be "Brutus, send 30 infantry to Pompus". Each general carries out the order he receives by redeploying the units mentioned in their message."

Appendix 2

Taken from the CptS 464 Chapter 14 slides by Professor David Bakken:

Snapshot algorithm

•By Chandy and Lamport [1985]: determine global states
•Goal: record a set of process AND channel states such that it is consistent (not strongly consisten)
•Assumptions
- Neither channels nor processes fail
- Channels are uni-directional and FIFO ordered
- Graph of processes and channels strongly connected (path between any 2 processes)
- Any process may initiate the snapshot at any time
- Processes don't need to freeze/lock: continue normal operations

• Main ideas
- Terms: incoming channels and outgoing channels for pi
- Each process records its state, and for each incoming channel, set of messages sent to it
- For each channel, process records msgs that arrived after its last recorded state and before sender recorded state
    - •I.e,. Record state at different times but account for messages transmitted but not yet received (these are part of the channel stat)
- Use distinguished marker messages
    - •Tell receiver to save state
    - •Way to determine which messages go in channel state
    - •To initiate the algorith, process acts like it received a marker message

Chandy and Lamport's 'snapshot' algorithm
Marker receiving rule for process pi
On pi 's receipt of a marker message over channel c:
    if (pi has not yet recorded its state) it
    records its process state now;
    records the state of c as the empty set;
    turns on recording of messages arriving over other incoming channels;
    else
    pi records the state of c as the set of messages it has received over c since it saved its state.
    end if
Marker sending rule for process pi
After pi has recorded its state, for each outgoing channel c:
    pi sends one marker message over c (before it sends any other message over c).

Appendix 3

Full instructions on how to run the project.

Basic Requirement: User is running on a computer capable of using the ssh command in a command prompt or terminal.

1. Swap the project in on DeterLab. The name of the project is **cwebb-proj3**.
2. Open 4 separate terminals/command prompts.
    a. In terminal 1: **ssh Caesar.cwebb-proj3.WSUCPTS464.isi.deterlab.net**
    b. In terminal 2: **ssh Brutus.cwebb-proj3.WSUCPTS464.isi.deterlab.net**
    c. In terminal 3: **ssh Operachorus.cwebb-proj3.WSUCPTS464.isi.deterlab.net**
    d. In terminal 4: **ssh Pompus.cwebb-proj3.WSUCPTS464.isi.deterlab.net**
3. (OPTIONAL) Check IP Addresses. They should be valid by default, but I ran into one case where they spontaneously seemed to change. The IP addresses for each node can be found on the Details tab on the DeterLab page for the project. The settings for the current IP addresses can be found in the following files:
    a. Caesar: **setcaesar**
    b. Brutus: **setbrutus**
    c. Operachorus: **setoperachorus**
    d. Pompus: **setpompus**
4. On the node for that general (Caesar = the Caesar node, etc.), run the bash script for that general. Note that these scripts will automatically detect if the Go Language is installed on that node, and if it isn't it will begin the installation process for it. Also, make sure to run the script for Caesar last.
    a. Brutus: **./RunBrutus.sh**
    b. Operachorus: **./RunOperachorus.sh**
    c. Pompus: **./RunPompus.sh**
    d. Caesar: **./RunCaesar.sh [Cut State]**
        i. Cut State: the state for Caesar to take the cut at. If no value is specified, it will choose a random number between 0 and 5 for taking the cut.
5. Wait for the runs to end. They will automatically exit upon completion of the distribution algorithm.
6. If a cut was made, it will be outputted to a separate file: **cutfile.txt**
   This file can be accessed using the command: **vim cutfile.txt**