CS 505: Exercise 6

Due: 03/23/2016 Wed. 11:59 PM

1 Overview

The goal of this programming assignment is for you to get some hand-on experience in implementing a Leader Election algorithm in the presence of crash failures, on top of a message passing based consensus algorithm.

2 Requirements

2.1 Languages

We do not restrict on the languages you can use for this programming assignment (e.g. C / C++ / Java / Python, etc.), as long as it could **compile** and **run correctly on XINU machines**. You will need to provide some scripts for compiling and/or executing your program on XINU machines. The grading will be done on those machines.

However, no matter which language you choose, you will need to use socket programming for network communications. Besides, each node should use **UDP** as the underlying transport protocol for communication with other nodes. If you are not very familiar with socket programming yet, here are some materials that might be helpful:

- C/C++: `http://beej.us/guide/bgnet/`
- Java: `https://docs.oracle.com/javase/tutorial/networking/sockets/`
- Python: `https://docs.python.org/2/howto/sockets.html`

2.2 Command Line Inputs

Ultimately, your program (named as *leader*) **must accept** the following command line arguments:

```
> leader -p <Port> -h <Hostfile> -f <MaxCrashes>
```

-p Port
Port identifies which port the program will be using for incoming/outgoing messages. It can take any integer from 1024 to 65535. All the nodes will use this port for communication.

-h Hostfile
Hostfile is the path to a file that contains a list of <id, hostname> pairs. It includes all the nodes that should be running this leader election program. You can assume that each host is running only one instance of the program. The file will be in the following format:

```
1 xinu01.cs.purdue.edu
2 xinu02.cs.purdue.edu
...
```

Hence, the first line means there is one program running on "xinu01.cs.purdue.edu" and it will be assigned id 1.

-f MaxCrashes
MaxCrashes is a number indicating the maximum possible number of crash failures. The value will be non-negative. And the total number of nodes will be no less than (MaxCrashes+2). Here we assume any process can crash, and a process will not recover once it has crashed.

2.3 Outputs

You will also need to print out to console at pivotal moments as specified below. But do remember to turn off all your own debugging outputs in the submission. There could be penalty for not doing that.

For the purpose of grading, print out to console when

- Every time when a node starts another leader iteration procedure, print out:
  [<time>] Node <self>: begin another leader election.

- Every time when a leader is elected, print out:
  [<time>] Node <self>: node <leader> is elected as new leader.

- Every time the leader node is detected to have crashed, print out:
  [<time>] Node <self>: leader node <crashed> has crashed.

In all above outputs, "self" is the assigned ID to current node. For variable "time", printing out using format "hh:mm:ss" is enough.

3 Lab Tasks

Here are some suggested steps in implementing a Leader Election algorithm that can handle crash failures. Each step is extending the previous step a little bit.

Eventually, the Leader Election algorithm should work correctly when at most MaxCrashes nodes crash; And when the elected leader crashes, it should redo the leader election algorithm automatically and elects a new valid leader.

3.1 Step 1: Basic Synchronous Consensus algorithm

It's recommended to start with implementing a synchronous consensus algorithm that doesn't consider any failures yet. Algorithm 1 below (similar to that in the homework) is the basis you could begin with:

Algorithm 1 n-process consensus; code for process p_i; $i \in \{1, \ldots, n\}$

$propose_i(v_i)$

1: **Send** value v_i to every other process $p_j \neq p_i$
2: **Receive** value v_j from every other process $p_j \neq p_i$
3: Let S_i denote the set of received values $\cup\ v_i$
4: **return** $max(S_i)$

Being *synchronous* indicates that it is possible to determine whether some node crashes or

is just being very slow. You could use timeout to achieve this, categorizing those nodes of no response for a long time to be "crashed". Since right now we assume no failures can happen, each message will arrive before it times out.

Note: You need to take care of some corner cases. For instance, all the processes may not start execution at the same time. If not properly handled, this situation may have processes agree on wrong values. **Describe in the report how you handle such corner cases and its limitation, if any.**

3.2 Step 2: Leader Election without failures

Based on the implemented consensus algorithm in §3.1, it's not difficult to have a Leader Election application building on top of that. The decision of choosing which node as leader is completely up to you, as long as it satisfies the

- Agreement: All non-faulty nodes decide on the same *leader*, *i.e.* process id.
- Validity: only a process participating in the protocol is elected as leader
- Termination: Every non-faulty node decides in a finite time.

properties as discussed in class.

3.3 Step 3: Leader Election with crash failures

To move on, you will first need to devise some synchronous consensus algorithm that can handle crash failures and every correct process must return a response after a finite number of communication rounds. **Describe your enhanced algorithm in the report.** Specifically, you must assume a parameter $f \leq n - 2$, where is the f is the upper bound on the number of process crashes and n is the total number of processes in the protocol. You must provide a precise algorithmic pseudocode of the protocol you implemented. Try to present a concrete proof for why your algorithm solves consensus. Be as formal as possible.

Note: There is a famous lower bound for solving consensus in synchronous message-passing systems with $f \leq n - 2$ crash failures which states that at least $f + 1$ rounds of communication is needed for reaching consensus (`http://www.sciencedirect.com/science/article/pii/S0020019099001003`). If your algorithm violates this lower bound, something is wrong!

The machines in XINU lab can be assumed to locate in a perfectly reliable network, where there are no failures unless you manually affect it. To simulate crash failures, you could manually stop the executing program on specific nodes, or just don't start the program at all. Command "pkill" may also be helpful.

3.4 Step 4: Leader Re-Election when leader crashes

The last step is to empower the existing leader election algorithm a bit more. Now when the elected leader crashes, the rest nodes should be able to redo the leader election automatically.

To do this, you will need to figure out some simple mechanism for detecting whether the leader is alive or not (typically based on *time-outs*). This is achievable since we are assuming a synchronous model. **Describe in the report about your approach.**

One idea to realize your failure detector is to force each process to ping every other process from "time to time"; the other process is presumed to have *failed* if it does not respond to the ping within twice the maximum round-trip time for any previous ping.

This is also a good time to read carefully consensus protocols like *Raft*: a popular consensus protocol based on Paxos used today. Here is the extended paper: `https://pdos.lcs.mit.edu/6.824/papers/raft-extended.pdf`. This will give you an idea of how consensus protocols are implemented and how failures are handled in large distributed systems.

Note: we have deliberately left unspecified several details of how you might realize this project. Feel free to state clearly in your report how you implemented this project specification and justify any assumptions you wish to make.

4 Submit Instructions

Similar to last programming assignment, this one could be done as a group exercise, of size no more than 3. If doing in groups, only one submission is needed for each group. Do remember to write all the names in the submitted PDF report.

This is also a good chance for you to form groups for your course project, which is also done in groups of size up to 3.

ONLY submit electronically using *turnin* command. Your submission should include the following files:

- All **SOURCE** files of your implementation (no object files of binary files).
- Some **SCRIPTS** (e.g. Makefile) that can compile your programs automatically.
- **README** if you would like us to know any particular instructions on compiling/executing your program.
- Some **REPORT** (PDF). In addition to the questions asked previously (see $\xi 3$), you should also discuss your overall structures, any design decisions, any implementation issues, etc.

Electronic turn-in instructions:

1. Log in to any CS machines (e.g. those in XINU lab) using your purdue account.
2. Now assume the file to submit is located at "~/ex6/". Go to the same directory of it, in this case it's your home directory "~/".
3. Clean the generated binary files, debug files before submission.
4. Type the following command: **turnin -c cs505 -p ex6 ./ex6/**
5. After submission, if you want to validate it's indeed submitted, you could use command: **turnin -v -c cs505 -p ex6** to do so.

Submission to "ex6" will be disabled after deadline. For late submission, please submit to "ex6late". Thus the command is: **turnin -c cs505 -p ex6late ./ex6/** and also **turnin -v -c cs505 -p ex6late** for validation.