Individual Coursework 1: Distributed Tickertape

Due date: 11:05 AM, 12th November 2018

Value: 15% of marks for module

Introduction

The City of London has hired you to prototype an alternative, more decentralized stock exchange. The City intends that each stock broker put a computer on the Internet that runs your ticker server. Whenever a broker buys or sells stock, the broker will submit a short description of that trade to the local ticker server. The ticker servers will exchange information about all trades, and each server will print out the description of every trade submitted to the system.

A key requirement is that all the servers must print out the trades *in the same order.* Any broker will be upset if it seems that he is not being given the same information as every other broker, because brokers make decisions about what to buy and sell based both on the latest prices and on price trends.

The larger purpose of this assignment is to learn about protocols for maintaining replicated data. If you have some data (a file system, for example) with identical replicas on a number of servers, you might want to perform updates on that data in a way that keeps the replicas identical. If you can arrange to perform the updates in the same order on all the replicas (and the updates are deterministic), then the data will remain identical. An important part of achieving this goal is making sure all the replicas process the updates in the same order. That's what you'll do for this assignment, though under an unrealistically optimistic set of assumptions about network and failure behavior that simplify the problem.

Requirements

Your server will take command-line arguments telling it its own port number, and the host names and port numbers of all the other servers in the system.

We supply you with a client program that submits new trades to the local server, using RPC. Each trade has a text tag. When a server sees a new trade submitted by a local client, it should tell all the other servers in the system about the trade. How this happens is up to you. We supply you with a prototype server that uses RPC (over UDP) to talk to the other servers; we suggest you start by modifying this server.

A server will receive notifications about trades from time to time from other servers. It should print each trade's tag on the standard output (*i.e.,* using `printf`). All the servers should print the trade tags in the same order. The servers need not print the tags immediately when they arrive.

Here are some other rules you must follow:

- Every live server must print out a trade within 13 seconds of when it was originally submitted by a client.

- Your system must continue to operate even if some servers die (which you can simulate with ^C). Dead servers are not required to print trades. If a server originates a trade but dies less than 3 seconds later, other servers are not required to handle that trade correctly.
- We will test your server with the client program we supply, so you should not change the submit RPC interface.
- Your server must work when more than one server runs on the same machine (with different ports), and when servers run on different machines.
- Except as noted above, all servers must print all trades in the same order.
- Your server must pass the `test-ticker` tester program (see below).

Here are some slightly unusual things you are allowed to assume about the system:

- You are allowed to assume that the network will deliver your RPCs within three seconds of when you make them, as long as the sending and receiving hosts are alive. The three seconds include the time required to retransmit lost RPCs, if required.
- If a computer or server fails, it will never come back to life.
- The computers running your servers may fail, but you are allowed to assume fail-stop behavior (*i.e.,* no Byzantine failures).

These are optimistic assumptions, which you might not be able to make about a real system; they should make your task easier.

Getting Started

All programming for this coursework must be done under Linux. We provide a Linux virtual machine (VM) image that you can use if you'd like to do development on your own machine. You also have the option of logging into a set of CS lab Linux machines remotely via `ssh`. We have ensured that the code we give you works correctly on the Linux VM we provide. It should also work correctly on the lab machines, although there may be corner cases where the lab machines may give different results than the VM we provide. **The VM we provide is the same VM we use to test your code. Your grade will be the score you receive when we run your code in our own copy of the VM.** If you consistently (*i.e.,* for many runs) get different results on a lab machine than you do when your code is tested by our auto-grader (which we describe below), please contact the course staff via a Piazza private message. We are happy to answer student questions about difficulties encountered when doing the coursework in the VM we provide or on CS lab machines, but we cannot support any other Linux installation.

Getting and Running the VM

To run the Linux VM we provide on your own computer, you will need to first download and install VirtualBox. You can find links to installer packages for the latest version of VirtualBox for Windows, Mac OS X, Linux, and Solaris online at:

```
https://www.virtualbox.org/wiki/Downloads
```

On that page, download and install the *platform package* for your computer's installed OS. Next, download the VM image by retrieving *both* files in the directory at:

```
http://www.cs.ucl.ac.uk/staff/B.Karp/0133/f2018/cw1/
```

Start VirtualBox (on Linux, with the shell command `VirtualBox`) and open the local copy of the VM image that you downloaded. A window will open, within which you will find running a complete Linux OS. The username and password to log in are `user` and `user`. The VM image we provide has all you need to do this coursework, including the C and XDR compilers, git, and popular editors (emacs, vim, and nano). Do your work within the VM: edit your code there, run tests there, and manage your code using git and GitHub there (more on git below).

Using the CS Lab Machines

You can also, if you choose, work on CW1 on the CS lab machines. Note that these machines are accessible over the Internet, so you may work on the coursework either from home or in the labs.[1] The Linux lab machines are those with the following hostnames:

```
niagara frontal parietal occipital sphenoid ethmoid maxilla
 palatine zygomatic lacrimal
```

Managing Your Code with `git`

You will manage the revisions of your code, including submitting it to the instructors for testing and grading, using the `git` source code control system and GitHub. `git` is useful for a great many things, from keeping the revision history of your code to making it easy to share your code on different machines (if you wind up wanting to use the VM on your own box and also develop on the CS lab machines, for example, you can keep your multiple working copies in sync via your "master" repository on GitHub). If you've not used `git` before, you can find a wealth of documentation online; we offer only a bare-bones introduction below.

`git` manages a set of source code files you are working on in a *repository*. You keep a local copy of the repository on a machine where you are editing your code and testing it, and use `git` to keep your local copy synchronized with a "master" copy of the repository on a server. In 0133, you will use GitHub to host the master copy of your repository. As you do your work (adding code, fixing bugs, *etc.*) it is good practice to update the master copy of your repository on GitHub with the changes you have made locally. There are two steps to doing so: first, you `commit` the changes to indicate

[1] To log into a lab machine remotely, you must first log into a CS departmental gateway such as `knuckles.cs.ucl.ac.uk` using `ssh`, then from there log into one of the lab machines using `ssh`.

that they are ready for shipping to the master repository, and second, you `push` your committed changes to the master repository.

To start the coursework, though, you must first retrieve a copy of the files we provide for you to start from. You can set up your GitHub master repository for your CW1 code by visiting the following GitHub URL:

```
https://classroom.github.com/a/b_A_Rf0y
```

If you already have a GitHub account, visiting the above URL will create a new private repository for CW1 (*i.e.,* visible only to you—no other GitHub user or the public will be able to read the contents of your repository). If you don't yet have a GitHub account, you'll be prompted to create one, after which GitHub will create a new private repository for CW1 in your new account. In both cases, the repository will be pre-populated with the code you need for CW1: a complete client, an XDR RPC interface definition file, a skeleton server, and a program that tests your code.

This new repository for your CW1 code is on GitHub's servers. You need a local working copy in your work environment—either within your VM if you are working on your own machine, or in your home directory of your CS Linux account if working on a CS lab machine. To create a local working copy, you need to `clone` your private CW1 repository from GitHub to your work environment. When you visit the above URL, your browser should eventually display a page that shows the contents of your new GitHub CW1 repository. Click on the "Clone or download" button toward the right on the page showing the contents of your new CW1 repository. Copy the link shown to the clipboard. To clone the CW1 repository from GitHub to your work environment, use the command:

`git clone` [paste URL here]

You will be prompted for your GitHub username and password. After you enter them, you will have a local working copy of the CW1 repository in a subdirectory with the same name as the GitHub repository (of the form `cw1-ticker-`[GitHub username]).

As you write your code and improve it (*e.g.,* by fixing bugs, adding functionality, *etc.*), you should get in the habit of syncing your changes to the master copy of your CW1 repository on GitHub. Doing so keeps the history of changes to your code, and so allows you to revert to an older version if you find that a change causes a regression. It also serves to back up your code on GitHub's servers, so you won't lose work if your local working copy is corrupted or lost. To bring GitHub up to date with changes to your local working copy, you must first use the `git commit -a` command (which will prompt you for a log message describing the reason for your commit, *e.g.,* "fixed crash on rapid trade submissions"), and then the `git push` command to copy your changes to GitHub.

Building and Running the Code

Change directories into your local working copy and build the skeleton code:

```
% cd cw1-ticker-[Github username]
% make
```

```
rpcgen -C -h -o ticker_prot.h ticker_prot.x
cc -c -Wall -g client.c
...
cc -Wall -g -o ticker-server server.o ticker_prot_xdr.o
ticker_prot_svc.o ticker_prot_clnt.o minirpc.o -L/usr/local/lib
%
```

(Don't worry if there are warnings from the C compiler about unused variables while compiling `ticker_prot_*.c`; rpcgen automatically generates those .c files, and sometimes includes unused variables in the C functions it generates. Remember: you should never edit the C source code for the `ticker_prot_*.c` or `ticker_prot_*.h` files. When you change the `ticker_prot.x` file, rpcgen will re-generate these automatically.)

The server expects command-line arguments as follows: a unique numeric ID, a local port number on which to receive RPC/UDP packets, and (for each other server) a host name and port number. To test the server, you might run the following on zygomatic:

```
zygomatic% ./ticker-server 1 2001 occipital 2002
```

And this on occipital:

```
occipital% ./ticker-server 2 2002 zygomatic 2001
```

You may have to modify the port numbers in case someone else is running this test at the same time. You can also run multiple copies on the same machine, using different UDP port numbers. (If you're running in a VM, you only have one machine, so should open a separate window for each server you wish to run, use `localhost`—the one machine you're on—as the name of the other machine to connect to, and be sure to use different UDP port numbers for each server process.) Now in a third window (on either machine), submit a few trades. For example:

```
zygomatic% ./ticker-client localhost 2001 IBM-90
zygomatic% ./ticker-client localhost 2001 DELL-16
```

Ideally, both servers would print either

```
IBM-90
DELL-16
```

or else they would both print

```
DELL-16
IBM-90
```

Either is acceptable, unless 7 or more seconds passed between submitting the two trades, in which case they must appear in order of submission. Notice that the server we supply you with doesn't work; only one of the servers prints each trade.

Interfaces and Hints

You will need to modify the files `server.c` and `ticker_prot.x` to complete this coursework. You may not modify the files `minirpc.c` or `minirpc.h`. You may not modify the `TICKER_PROC` procedure in `server.c`, as it is used by the `ticker-client` program. Instead, you will probably want to add at least one procedure to the `TICKER_PROG` RPC interface, for ticker servers to use when contacting each other. You can do this by adding appropriate structures to the `ticker_prot.x` protocol definition file. For instance, you might add something along the lines of the bold text below:

```
struct xaction_args {
  /* XXX - You must place fields you want in the argument here */
};

program TICKER_PROG {
  version TICKER_VERS {
    submit_result TICKER_SUBMIT (submit_args) = 1;
    void TICKER_XACTION (xaction_args) = 2;
  } = 1;
} = 400001;
```

For each procedure that you add, you must add a server side dispatch routine. This function will be named by the procedure name you have chosen (translated into lower case), with the version number (`1`) and `svc` appended, separated by underscores. In the example above, you would want to add the following procedure to your program:

```
void *
ticker_xaction_1_svc (xaction_args *argp, struct svc_req *rqstp)
{
  /*
   * XXX - You must write this code
   * Arguments are in argp.  (You can ignore the rqstp parameter.)
   */

  return NULL;
}
```

Note that one thing you probably don't want to do is block waiting for RPCs to other servers. If, for example, when your ticker server received a trade it made synchronous RPCs to every other ticker server, you could easily end up with deadlock when two servers receive trades simultaneously. (Each will be waiting for an RPC result to return from another server before servicing any other RPCs.)

Instead, we have supplied you with two functions for making asynchronous RPCs. These functions make an RPC and return immediately, without waiting for the response, but keep retrying in the background in case a UDP packet is lost. These functions are declared in the header file `minirpc.h`:

- `void rpc_send (struct sockaddr_in *dest, u_int32_t prog, u_int32_t vers, u_int32_t proc, xdrproc_t argxdr, void *arg);`

This function sends an RPC to the server at UDP port `dest`, but returns immediately without awaiting a reply. (It will keep trying in the background in case the UDP packet is lost.)

`prog`, `vers`, and `proc` are as in the `.x` file, for instance `TICKER_PROG`, `TICKER_VERS`, and `TICKER_SUBMIT`, respectively.

`argxdr` is the auto-generated XDR marshaling routing for the argument type. For example, for type `submit_args`, the function is `xdr_submit_args` (in general, just prepend `"xdr_"` to the name of the type). `arg` is a pointer to the actual arguments. *e.g.,* for `TICKER_SUBMIT`, this would be of type `submit_args *`.

- `void rpc_broadcast (struct sockaddr_in **dest, u_int32_t prog, u_int32_t vers, u_int32_t proc, xdrproc_t argxdr, void *arg);`

  This function is like `rpc_send`, except that it sends the same RPC to several servers. Here `dest` is now an array of pointers to UDP socket addresses. There must be a `NULL` pointer after the last socket address.

Note that the skeletal `server.c` file, when it parses the command line arguments, creates a `NULL`-terminated array called `others` containing the addresses of all the other servers. Thus, if you want to send an RPC to all the other servers, you can write:

```
rpc_broadcast (others, ...);
```

Because you may want to wait around for a while before printing a trade (to make sure there aren't any previous trades you haven't heard of), `server.c` contains a skeletal function `timer` that you can ask to have called once per second. The following variables (declared in `minirpc.h`) are useful for `timer`:

- `extern time_t elapsed;`

  This variable always contains the number of seconds that have elapsed since your ticker server program started. You can use the value to timestamp RPCs you receive and decide how long ago you received them.

- `extern int want_timer;`

  You must set this to a non-zero value for your timer function to get called. When `want_timer > 0`, function timer will be called once per second. You should set `want_timer` to a positive value when you need timer events, and set it to zero if you don't need any timers and can sleep until the next RPC is received.

Testing Your Server

Once you have a server that you think might work, you can do a quick test as follows. Start two servers as above. Then run the client program with these arguments:

```
% ./ticker-client -r 5 zygomatic 2001 occipital 2002
```

This generates 5 submissions to each of the servers indicated, in rapid succession. (This isn't quite correct, since the client is only allowed to submit to the local server, but

it's just for testing.) If your servers always agree on the order of outputs when you run the client this way, you're well on your way to finishing the coursework.

Your code's correctness will be assessed with the `test-ticker` program. You should run the tests to see if your server works before concluding that you've completed CW1 successfully. There are two ways to run the tests: on our grading server, and manually yourself in your working copy of the code.

Grading server: Every time you push your updated code to GitHub, our grading server will retrieve a full copy of your code, build it (inside a VM disconnected from the Internet), run the tests, and push a report containing the results of the tests back into your CW1 repository on GitHub. The test results file is named `report.md`. It will take several minutes for the file to appear in your repository, as the tests take that long to run. The results file will contain the same information as what you see when you run the tests locally in your working copy. The results from the grading server are authoritative: it is the test results on the grading server at the deadline that determine your grade.

Manually in your working copy: This is probably more convenient and faster than waiting for the grading server to produce a report. However, as only the grading server is authoritative—we don't award grades on the basis of tests you run yourself, but only for tests run by the grading server—you will want to make sure that you pass the tests on the grading server before concluding that you've completed the coursework successfully. You can run the tests in your working copy (same command whether in the VM or on a lab machine) as follows:

```
% ./test-ticker ./ticker-server ./ticker-client
One server, one transaction (no points): passed
Two servers, one transaction (1 point): passed
Two servers, two transactions (1 point): passed
Two servers, ten concurrent transactions (2 points): passed
Five servers, continuous transactions (3 points): passed
One of two servers fail (1 point): passed
Three of six servers fail (2 points): passed
FINAL SCORE: 10/10
%
```

Turning in the Assignment

In order to turn in your completed assignment, you must do as follows **before 11:05 AM on Monday the 12th of November 2018:**

1. Before the deadline, be sure you've updated the GitHub repository to reflect any changes you've made to the finished version of the code in your working copy using `git commit -a` and `git push`.

2. When the deadline arrives, GitHub will automatically take a snapshot of your last commit (the last version you pushed before the deadline) and label it for the instructors as your submission. Your grade will be the score you receive on the tests run on this version of your code on our grading server.

3. Write a design document: in a plain-text or PDF file named either `design.txt` or `design.pdf`, respectively, provide pseudocode for the algorithm used in your solution and a clear explanation in English of why it will always print trades in the same order on all participating hosts, given the assumptions stated at the start of this coursework, but *no further assumptions*. This design document *may not be longer than one side of one page of A4 paper.* **Be sure that this file is pushed to the GitHub repository!** Use the `git add` command to add this file to your local working repository, and then `commit` and `push` as usual to sync it to GitHub.

75% of your mark on this coursework will be whatever score you obtain from the automated tests on our grading server. Note that while the grading server will only test your code once each time you push new code to GitHub before the submission deadline, the teaching staff will run the automated tests on your code several times, and you will receive the *minimum* score your code receives across these tests. We are awarding marks for reliable operation of your code, and intermittent failures (on some runs but not others) indicate that your code does not operate reliably.

The remaining 25% of your mark on this coursework will be determined by the instructors' evaluation of the correctness of the algorithm and completeness of explanation you provide in the design document you submit.

If you have any problems with submitting the coursework, please contact the instructor.

Late Work Policy

As explained in the first lecture of term and on the class web site, 0133 uses a late days system for late coursework; this policy is different from the general departmental policy. If you turn in this coursework late, please **write at the top of your design document** how many late days you would like to use.

If you want to submit late, you will need to indicate to our grading server that you wish for a version you commit after the deadline to be graded. The way to do so is to include the string `LATESUBMIT` (all capitals, no space in between words) in your commit log message. We will grade the latest commit with that string present in its log message. We will *always* give you a grade for the latest commit with this string in its log message. So if you find your latest `LATESUBMIT` commit is for a version that works less well than a prior one, be sure to revert to the prior version that worked better. We will not allow students to request that we grade an earlier `LATESUBMIT` version of their code than the latest submitted, so please heed these instructions carefully.

Academic Honesty

This coursework is an **individual coursework**. Every line of code you submit must have been written by you alone, and must not be a reproduction of the work of others—whether from the work of students in the class from this year or prior years, from the Internet, or elsewhere.

Students are permitted to discuss with one another the definition of a problem posed in the coursework, but not the design, details of, or code for a solution. Students are strictly prohibited from showing their solutions to any problem (in code or prose) to a

student from this year or in future years. In accordance with academic practice, students must cite all sources used; thus, if you discuss a problem with another student, you must state in your solution that you did so, and what the discussion entailed.

Any use of *any* online question-and-answer forum (other than the 0133 Piazza web site) to obtain assistance on this coursework is strictly prohibited, constitutes academic dishonesty, and will be dealt with in the same way as copying of code.

You are free to read reference materials found on the Internet (and any other reference materials). You may of course use the code we have given you. **Again, all other code you submit must be written by you alone.**

Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities (normally resulting in exclusion from all further examinations at UCL). The course staff use extremely accurate plagiarism detection software to compare code submitted by all students and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Read the Piazza Web Site

You will find it useful to monitor the 0133 Piazza web site during the period between now and the due date for the coursework. Any announcements (*e.g.*, helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions there. *Please remember that if you wish to ask a question that reveals the design of your solution, you must mark your post on Piazza as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about RPC or C that do not relate to your solution, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

References

The following references may be useful in completing this assignment:

- Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In Communications of the ACM, 21(7):558-565, July 1978. (This is the original paper describing logical clocks.)
- RFC 1832: XDR data representation standard
- Linux manual page for RPC
- RFC 1831: RPC protocol specification