UCL CS M030/GZ03 Brad Karp

Individual Coursework 1: Distributed Tickertape

Due date: 9:05 AM, 2nd November 2011

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 on the department's lab machines. We have ensured that the code we give you to build upon works correctly on these 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:

```
auerbach calder fulla goya hals judd kubin lowry munch opie
pollock quarton rubens shitao valdes whistler zorach
faulkner heine hesse swift carey levi hortensia
```

We supply you with a complete client, an RPC interface definition file, a skeleton server, and a test program, in `~ucacbnk/gz03-2011/ticker.tar.gz`.[2] To unpack and build the skeleton code, run:

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

[2] Note that the character before `ucacbnk` is a tilde (located to the left of the "Z" key on a UK Mac keyboard, to the left of the Enter key on a UK PC keyboard, and to the left of the "1" key on a US keyboard.

```
% tar xvfz ~ucacbnk/gz03-2011/ticker.tar.gz
ticker
ticker/minirpc.c
ticker/server.c
ticker/minirpc.h
ticker/ticker_prot.x
ticker/client.c
ticker/.cvsignore
ticker/Makefile
% cd ticker
% 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 levi:

```
levi% ./ticker-server 1 2001 lowry 2002
```

And this on lowry:

```
lowry% ./ticker-server 2 2002 levi 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. Now in a third window (on either machine), submit a few trades. For example:

```
levi% ./ticker-client localhost 2001 IBM-90
levi% ./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 levi 2001 lowry 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 it manually to see if your server works before handing it in. You can run it as follows:

```
% ~ucacbnk/gz03-2011/bin/test-ticker ./ticker-server
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
[internal use only: student:10:1081213431:78d892ccade421ccf638:9a26ed
763484f3407f98]
%
```

(Ignore the last line; it is used during grading.)

Turning in the Assignment

In order to turn in your completed assignment, you must do as follows **before 9:05 AM on Wednesday, the 2nd of November, 2011:**

1. Run the command `make handin`. That command will produce two files: `score` and `ticker.tar.gz`. When you run `make handin`, you should see output like the following:

   ```
   % make handin
   make clean
   ...
   `echo ~ucacbnk`/gz03-2011/bin/test-ticker ./ticker-server | tee score
   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
   [internal use only: student:10:1081214349:78d892ccade421ccf638:fc5504
   adf0f2a980dd76]
   %
   ```

   Be sure that the line including `internal use only` is present.

2. **Without modifying any file in your directory,** submit the following **four files** as Coursework 1 on the M030/GZ03 Moodle web page:

- `score` (produced by `make handin`)
- `ticker.tar.gz` (produced by `make handin`)
- `ticker-server` (the executable program for your server that was built in the directory when you ran `make handin`)
- A design document: in a plain-text or PDF file named either `design.txt` or `design.pdf`, respectively, 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.*

The `ticker-server` file you submit via Moodle must be the one that was present in your directory when you ran `make handin`. The marking software can detect if the `ticker-server` executable file you submit was used to produce the `score` file you submit. If they do not match, you will receive zero marks for the coursework.

75% of your mark on this coursework will be whatever score you obtain from the automated tests. 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, M030/GZ03 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.

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 and the general outline of an approach to a solution, but not the 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.

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 Web Forum

You will find it useful to monitor the M030/GZ03 Web Forum on Moodle 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.

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