CSci551 Spring 2010 Project C

Assigned: Oct. 31, 2011. Project C Due: noon, Nov. 30, 2011 (not a class day)

You are welcome to discuss your project with other students at the conceptual level. You may reuse algorithms from textbooks. You may possibly reuse functions from libraries, but if you're using anything other than the C or C++ standard library (libc, STL), or libraries mentioned on the TA's web page, you *must* check with the TA and the professor and identify it in your write-up. You need to check allowed libraries in Project Information on class moodle. Otherwise, each student is expected write *all* of his or her code independently. All programs will be subject automated checking for similarity. Any cases of plagiarism will result in an F for the entire course. **If you have any questions about policies about academic integrity, please talk to the professor.**

Please note: you should expect to spend a significant amount of time on the class projects this semester, probably 20 hours or more when they're all put together. Please plan accordingly. If you leave all the work until the weekend before it is due you are unlikely to have a successful outcome. That said, Project A is a "warm-up" project. Projects B and C are be much larger. Although Project C is smaller than Project B, you should expect it will still take a significant amount of time.

Changes: 2011-10-26: none yet (although please make sure you got the several updates to Project B)

2011-11-03: added simplifying assumption: the first client is never a tkcc client.

2011-11-08: questions for stage 7 were previously incorrectly numbered (as 6.x); they should be 7.x.

2011-11-14: two new messages, ext-stores-q and ext-stores-r, added to stage 7. I think you'll find they're necessary to complete that stage.

1 Overview

The purpose of this project is to get some hand-on experience building a substantial distributed application running on a multi-process computing infrastructure, then to use it to study networking. It also has a secondary purpose of implementing a program using network programming (sockets) and processes (fork, in stage 1) and Unix development tools (make).

This homework will be done as three separate but related projects. First (Project A), you need to demonstrate that you can read the config file do basic process control, and submit a complete assignment. (Project A doesn't really evaluate anything advanced, but it should confirm that everyone's on the same page and is comfortable with our software environment.) Project A has an early due date. Project A should be relatively short for students used to working on Solaris or Unix; if not, it should help get you up to speed.

In Project B you will use this facility to implement Triad, a simplified version of a Chord-like Distributed Hash Table as described in the paper by Stoica et al. (see [Stoica00a] in the class syllabus). It will probably be due just after the midterm. Project B will be *much* larger than Project A; you should plan your time accordingly. Project B will be assigned later and may overlap partially with Project A.

Project C will involve extending your DHT to consider additional cases and perhaps apply it to an application. It will probably be due the last week of class.

Each project will build on the previous one.

2 Project A: Getting Started

(We don't reproduce Project A here, but you may want to refer to it for background.)

3 Project B: Triad

We don't reproduce it here, but you may wish to refer back to your Project B for hints and background. In fact, Project C is required to use the same messages, message formats, and logging rules as Project B.

4 Project C: Extending Triad

4.1 Stage 6: Unexpected Node Departure

Stage 5 (in Project B) explored controlled node departure, as a node announced it was leaving and other nodes take over its job. In a real peer-to-peer system, nodes aren't so friendly—they go away when computers are turned off or suspended. And even if computer users tried to intentionally shut them down, they can still fail unexpectedly and reboot, so all real systems have to deal with unexpected failure.

In this phase, we'll add support for unexpected node failure. All nodes need to watch the ring and repair a single unexpected failure. To accomplish this repair, each node will need to keep both a successor pointers and a double-successor pointer. Each node will periodically poll its successor; if it doesn't get a reply or if it gets an error message, it should assume the successor failed and rebuild the ring accordingly.

Note that because data is stored on only one node, any such failure may loose data. Those are the breaks; you won't recover from them in this stage.

Each node should do stabilization, confirming it's successor exists reasonably frequently, say every 10 seconds. You will need to implement this test to run in the background, while you're also doing other message passing. You may find the timer library is helpful to accomplish this goal. To check the predecessor of your successor, a client must use the following two messages:

15. **hello-predecessor-q** (ni) ask target ni to confirm that it's alive, and what it's predecessor

16. **hello-predecessor-r** (ni, pi, pp) reply stating that n's is alive and has a given predecessor has id pi and port pp

Note that these are the same as the standard predecessor-q and -r messages, but they have different ID codes so we can easily tell why you're using them.

Nodes also must maintain their finger tables. In Chord, nodes maintain their finger tables by verifying a random finger table entry periodically. We're going to do something slightly different: the finger tables (other than the successor and predecessor) should be maintained *lazily*. That is, once they're set up, don't do anything to adjust them, *but* verify they're correct before each use. That is to say, if your node wants to use finger table i for id id that points to ni, it should first send a stores-q(ni, id) to ni. If the finger table is correct, then the stores-r reply should indicate that $ni = ri$ (if something changed), or your query may time out (if the target went away). Either way, if the finger table is incorrect, then your node should rebuild its finger table entry before continuing.

Input is just like stage 5, although the stage line will say 6, and in addition to the "end_client" command, there will be a new "kill_client" command. When the manager reads this command, it will send it to the relevant client that should then exit immediately. The manager should then wait some time (say, 60 seconds; *clarification 2011-11-08:* or if you need longer, justify why in your README) to give the ring to rebuild itself.

Sample input:

```
stage 6
nonce 1234
start_client alpha
start_client bravo
start_client charlie
start_client delta
start_client echo
start_client foxtrot
store alpha
store beta
store gamma
kill_client charlie
store delta
store epsilon
search beta
search delta
```

Sample output: Again, the filename should include "stage6". We will not provide sample output for this stage.

Contents of stage6.README.txt: 6.1) Have you implemented all the requirements of this stage? if not, which are missing?

6.2) Did you use any library other than those presented on the moodle page? if so, what one and to do what?

6.3) How do you detect that a given client is down?

6.4) How did you support receiving messages and also handling timeouts to do stabilization, when either reply might happen at the same time?

4.2 Stage 7: An Adversary

A problem with peer-to-peer systems is that it assumes that all nodes will cooperate, yet in the real world they don't always do that. Stage 6 looked at the problem of nodes departing the ring unexpectedly, one kind of failure. We next consider a different kind of failure, and a possible solution to both.

The TKCC (Triad Killers Community Club) is an organization that hates peer-to-peer networks. They want to stop Triad, and to do this they actually create a bunch of bogus Triad nodes. Their bogus nodes do everything normal Triad nodes do, except they replace all content that TKCC doesn't want stored with a file of the appropriate type that says "nyah, nyah". Since there is nothing to stop a client from joining the ring and taking over its piece, this approach allows the TKCC to blockade parts of the ring.

The Triad operators were not happy with this interference, so they hit on a plan: each data item should be replicated and stored at 4 different places around the ring. Replication forces TKCC to put up many more interfering clients, and it also helps the ring avoid loosing data when one node unexpectedly leaves.

For stage 7, implement the following additions to the prior stages: when computing the id of a node, do it like normally. However, when computing the id of a data item as part of a store command, store copies of the item in all four locations found by looking at a all combinations of the top two bits. Thus, if the item foo is stored at 0xa9367d92, please store copies in 0x29367d92, 0x69367d92, 0xa9367d92, and 0xe9367d92.

When searching (with the search command), you must retrieve the two closest copies to the node doing the search (where closest is defined by what will be found in fewest steps searching around the ring in an ascending order, since that's the direction supported by finger tables). It should compare the two readings, and log "search S to node NI and NJ, key PRESENT and VERIFIED" if both contents match, or "search S to node NI and NJ, key PRESENT but DISAGREE" if they don't. (Note that you can also tell which contents are correct, since you can compute your own hash of the contents to see if it is correct. And of course, in this case, you happen to know that the content "nyah, nyah" is incorrect. Thus this double search actually allows error correction, provided one of the two polled nodes is valid.)

Finally, we will add one new configuration command: the `start_tkcc` command should create a client that changes the contents of anything it stores as described above. (*Addition 2011-11-03:* as a simplifying assumption, you may assume that we do not make the first client a tkcc client.)

Addition 2011-11-14: We need to add a new message to allow one to retrieve the contents from another client. We add that below with the ext-stores-r reply; we also add the ext-stores-q to draw this reply. (I think you find these messages are required to complete stage 7.)

17. **ext-stores-q** (ni, di) ask target ni what it's best estimate of the node that stores di (note that its answer may not be correct, in that the reply may not actually cover di, but it is presumably an improvement This version is identical to stores-q, except that it elicits an ext-stores-r reply, described next.

18. **ext-stores-r** $(ni, di, ri, rp, has, SL, S)$ reply stating that ni's best estimate of the node

that stores di has id ri at port rp. The *has* value is 1 if $ni = ri$ and ri has the exact id di, otherwise 0. (In other words, if you queried the node holding the exact hash and that node has stored that data locally.) In addition, this extended version of stores returns the contents via SL and S.

Sample input:

```
stage 7
nonce 1234
start_client delta
start_tkcc bravo
start_client charlie
start_client alpha
start_client echo
start_client foxtrot
store alpha
store beta
store gamma
store delta
store epsilon
search gamma
search delta
```

Sample output: Again, the filename should include "stage7". We will not provide sample output for this stage.

However, for the sample input, we think you should find that searching for gamma results in disagreement, while delta is OK.

Contents of stage7.README.txt: (*Correction:* these questions were numbered 6, but they should of course be 7.) 7.1) Have you implemented all the requirements of this stage? if not, which are missing?

7.2) Did you use any library other than those presented on the moodle page? if so, what one and to do what?

7.3) From a technical point of view, TKCC is attacking Triad by putting up false nodes. One way to judge the effectiveness of this attack is to evaluate how many IDs m false nodes will block. First, *without* the replication added in stage 7, suppose there are n nodes in Triad, and 1 false node is added. What fraction of the ID space is preempted in this case?

7.4) Of course, stage 7 adds replication. With replication (as per stage 7) and 1 false node, what fraction of the ID space is now impeded?

7.5) Do you have one other suggestion about how to *prevent* Triad from working? (In other words, answer the question: if you were TKCC, what technical step would you take to impede Triad.)

7.6) Do you have one other suggestion about how to make Triad robust to attacks? (In other words, answer the question: if you were the Triad developers, what technical step would you take to protect Triad from your biggest fear of attack.)

5 Submitting and evaluating your project

To submit each part the assignment, put *all* the files needed (Makefile, README, all source files, and source to any libraries) in a gzip'ed tar file and upload it to the class moodle with the filename `projc.tar.gz`. *Warning:* when you upload to the moodle, please be careful in that you must both do "Upload a file" *and* do "Save changes"! When you are done you should get a message "File uploaded successfully", and you should see a list with your file there and an option to "update this file". If you do *not* see "file uploaded successfully", then you have *not* completed uploading!

We *strongly* recommend that you confirm that you have included everything needed to build your project by extracting your tarfile in a different directory and building it yourself. (It's easy to miss something if you don't check.)

It is a project requirement that your project come with a Makefile and build with just the make command. To evaluate your project we will type the following command:

```
% make
```

Structure the Makefile such that it creates an executable called `projc`. (Note: *not* `a.out`.) For more information please read the make(1) man page. (Your program must run with Solaris make, so be careful not to use any make extensions such as those in GNU make (gmake).)

We will then run the manager using a test configuration file. You can assume that the topology description will be syntactically correct. After running the program, we will grade your project based on the output files created by the manager and the clients.

It is a project requirement that your implementation be somewhat modular. This means that you should follow good programming practices—keep functions relatively short, use descriptive variable names. You must use at least one header file, and multiple files for different parts of your program code. (The whole project should be broken up into *at least* two C/C++ files. If you have a good file hierarchy in mind you can break up into more files but the divisions should be logical and not just spreading functions into many files.) Indicate in a comment at the front of each file what functions that file contains.

Computer languages other than C or C++ will be considered but *must be approved ahead of time*; please contact the professor and TA if you have an alternative preference. The deadline for approving new computer languages is *one week* before the project due date, so get requests in early. The language must support sockets and process creation. (Please ask *before* you start, we don't want you waste your work.)

Although we provide a complete sample input file and output, final evaluation of your program will include other input sources. We therefore advise you to test your program with a range of inputs.

Although the exact output from your program may be different from the sample output we provide (due to events happening in different orders), your output should match ours in format.

Clarification 2011-10-18: **It is a project requirement that your code build without warnings** (when compiled with -Wall). The TA had previously mentioned this in the project Moodle page; this note repeats this requirement here. We will be responsible for making

sure there are no warnings in class-provided code (any warnings in class-provided code are our bugs and will not count against you).

6 Hints

The structure of the project is designed to help you by breaking it up into smaller chunks (compared to the size of the whole project). We strongly encourage you to follow this in your implementation, and do the stages in order, testing them as you go. In the past, some students have tried to read and implement the whole assignment all at once, almost always resulting in an unhappy result.

6.1 Where to run

Please test and debug your processes on ~~the Suns in SAL 125 and LVL library~~ (now gone) *Correction 2011-09-12:* on the server `cs-server.usc.edu`. Please do not run jobs or do lots of development with many processes on aludra or nunki; there are process limits that may prevent them from running. (Hint: make sure you check the exit status of fork() so you're not surprised when something doesn't run.) However, you should test your final program on ~~nunki~~ cs-server since that's where we will test it.

~~The best place to develop and test your code is on the Sun workstations maintained by USC. (On a workstation you won't bother other people like you might developing on a server.)~~ ITS provides several computing centers listed at `http://www.usc.edu/its/pcc/index.html`. *Addition 2011-09-12:* you may find the terminals in these sites helpful to access the USC servers, and the MacOS boxes helpful to provide a near-Unix platform on which to develop.

However, of course everyone wants to develop in the comfort of your own home. This should be *very* easy to do if you run Linux or BSD on your computer (or if you install your computer to dual boot). Note that you *will* have to port your code to run at USC on our testing computer, but if you do a careful job that should be very, very easy. If you're running MS-Windows, you might consider trying Cygwin, a Unix emulation package for Windows. (We cannot support or answer questions about it, however.) If you're running a Macintosh, MacOS X includes a complete BSD Unix that should be very similar to Solaris. (And please remember, regardless of where you develop your code, we test it at ~~nunki~~ cs-server so it *must* run there.)

Addition 2011-09-12: Another option to get a Unix development environment that you can run locally is to get a virtual machine with pre-installed Linux of some flavor. Virtual machines like VMware, VirtualBox, Xen, qemu, and KVM are widely available are free (both libre and usually gratis) and portable to different degrees. One can get per-built virtual machine images for most (search the web for, say "ubuntu vm image" or "fedora vm image"). (Again, please remember, regardless of where you develop your code, we test it at ~~nunki~~ cs-server so it *must* run there.)

6.2 Common pitfalls

Please do not hardcode any directory path in your code! If you hardcode something like `/home/scf-...` or `/auto/home/scf-...` in your code to access something in your home directory and the grader cannot access these directories during grading, your code will not work (and this will be your fault)! If your code does not work, you get no credit! Instead, assume paths are given external to your program, and that you read and write files in the current directory (wherever that is).

One thing to watch out for on Solaris is that all the useful bits of the network are in libraries "socket" and "nsl". You will need to link against these libraries on Solaris, but not on other versions of Unix.

6.3 Doing multiple things at once

Later stages of the project may require you to handle both timers and I/O at the same time. *Clarification 2011-09-17:* Note that stage 1 is not complex enough to require timers. One approach would be to use threads, but most operating systems and many network applications don't actually use threads because thread overhead can be quite large (not context switch cost, but more often memory cost—most threads take at least 8–24KB of memory, and on a machine with 1000s of active connections that adds up, and always in debugging time, in that you have to deal with synchronization and locking). Instead of threads, we strongly encourage you to use timers and event driven programming with a single thread of control. (See the talk "Why Threads Are A Bad Idea (for most purposes)" by John Ousterhout, `http://www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf` for a more careful explanation of why.)

Creating a timer library from scratch is interesting, but non-trivial. We will provide a timer library that makes it easy to schedule timers in a single-threaded process. You may download this code from the TA web page. There is *no* requirement to use this code, but you may if you want. If you want to use it, download it from the class web page. There is no external documentation, but please read the comments in the `timers.hh` and look at `test-app.cc` as an example. If you do use the code, you must add it to your Makefile and you must document how you used it in your README.

6.4 Other sources of help

You should see the Unix man pages for details about socket APIs, fork, and Makefiles. Try `man foo` where foo is a function or program.

The TA can provide *some* help about Unix APIs and functionality, but it is not his job to read the manual page for you, nor is it his job to teach how to log in and use vi or emacs.

You may wish to get the book *Unix Network Programming*, Volume 1, by W. Richard Stevens, as a reference for how to use sockets and fork (it's a great book). We will *not* cover this material in class.

The README file should not just be a few sentences. You need to take some time to describe what you did and especially anything you didn't do. (Expect the grader to take off

more points for things they have to figure out are broken than for known deficiencies that you document.)