CSci551 Spring 2010 Project B

Assigned: Wed. Sept. 28, 2011 Project B Due: noon, Wed. Oct. ~~19~~ 26, 2011 (not a class day) (*Change 2011-10-06:* deadline extended by one week.)

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.

Changes: 2011-09-21: none yet.

2011-09-29: bug fix in update-q specification

2011-10-06: clarified in stage 2 that it is a guarantee that nodes to not arrive or depart after all are created, and that this guarantee goes away in stage 5.

2011-10-06: added the has field to closest-r

2011-10-06: the closest-q and closest-r messages were changed to stores-q and stores-r, and the *has* value was added to stores-r.

2011-10-06: incorrect comment about phase 2 sample output not having a nonce removed.

2011-10-06: deadline extended one week, to Oct. 26. (There will *not* be any additional deadline extensions, although students may choose to use your slip day if you want.)

2011-10-10: removed statement suggesting passing the nonce through fork(). (It's already passed through TCP connection, and that is required, so through fork() adds nothing.)

2011-10-11: all hash values should be printed in hexadecimal. The specification incorrectly said in its the first log message the hash should be printed base-10.

2011-10-11: added "with hash SH" to the log output of storing a value. This change is manditory.

2011-10-11: clarified that the "client S created..." message is printed out only just after the node joins the ring.

2011-10-14: removed comment about "a sync line" in stage 2.

2011-10-15: clarification: the end_client command needs an "OK", just like all the other manager-client commands.

2011-10-16: two corrections to stage 2 sample ouptut: 1) The 8th line of stage2.foo.out should be "update-q received (0xa9367d92 0xdabfd86c 34089 1)" and 2) The 4th line of stage2.baz.out should be "update-q sent (0xa9367d92 0xdabfd86c 34089 1)"

2011-10-18: corrected an incorrect description of in the "search" command of stage 2.

2011-10-18: added an *optional* **update-r** message in stage 2.

2011-10-18: added discussion of iterative and recursive updates in stage 4.

2011-10-18: added a no-warnings requirement to this document (previously only on the TA page).

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.

Project C will be assigned later. It will be smaller than Project B but bigger than Project A. It will build on Project B. We will offer a the TA's implementation of Project B for students who wish to use it instead of their own Project B implementation, but we do not promise to help you understand it.

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

Now that Project A has demonstrated running basic processes and communication, Project B gets down to the business of implementing *Triad*, a simple Distributed Hash Table modeled after Chord.

3.1 Concepts

Please refer to [Stoica00a] from class for a detailed description of Chord and the Chord algorithms.

Triad is like Chord, but slightly simpler. Like Chord, nodes form themselves into a ring and store keys. Each node and key has an id; a node is responsible for all keys that have its id value up to just before the id of the next key in the ring. In the worst case, nodes find data by linear search through the ring.

Like Chord, Triad can use finger tables to speed search, providing $O(\lg n)$ search time in the best case. Unlike Chord, the first stages of Triad will be done without such acceleration and compare versions.

Both Chord and Triad use hash values to map data into the storage ring, but Chord uses the full 160-bit SHA-1 value. Triad will use a 32-bit value using a hash function provided by the TA (it will actually be just the first 32-bits of the SHA-1). Chord also talks about hash values abstractly. For Triad, we will provide string names and part of a function to compute the underlying hash value.

While both Chord and Triad store data in a ring. Unlike Chord, we will also store data with those keys. (We will add this feature in a later stage.)

Chord includes algorithms to deal with ring maintenance as nodes come and go, possibly without notice. In Triad we will build this capability up gradually. In early stages all nodes will join at the beginning and no nodes leave. In later stages nodes will leave with warning to their neighbors, then join after some keys have been stored, and eventually we will support nodes coming and going without warning.

3.2 Stage 2: Booting Triad

For stage 2 we will bring up the smallest possible correct Triad: you will boot a network of several nodes that will form a Triad network with successor pointers. This stage is about correctness, not performance, so we will not yet implement a finger table. Instead, each node will link directly to its immediate neighbors. In addition, to simplify this stage, nodes will (or should!) never fail and leave the network, so you only need to deal with network growth, not change.

To be able to exercise your Triad network, the configuration file will include a script to create nodes and make them perform operations.

As with stage 1, we expect the manager to read the configuration file. Unlike stage 1, here we will create clients dynamically.

You should expect a “stage 2” and “nonce N” lines in the configuration file (as well as comments). The nonce will be used in computing hashes. It is guaranteed to be a unsigned integer less than 2^{32}. The manager should pass it to all clients, (*Correction 2011-10-10*) ~~perhaps through a variable that is preserved across the fork()~~ through the TCP socket as in stage 1.

Creating clients: After the nonce line there will be several `start_client S` lines, ~~each followed by a sync line~~ (*Removed 2011-10-14:* this was left over from an earlier draft of the project). Each start_client line will specify a name of the client S (some string of 80 characters or less). The manager should create the client by forking and setting up a TCP connection as in stage 1. The manager will write four lines of text to the client to get it going: first the nonce N, then the client’s name S, a port number FP (described below), and a second name FS all in ASCII terminated by a newline. (The TCP connection should stay open; it will be used to send later commands from the manager to the client.)

For the first client created, the port number FP will be zero, and the second name FS will be the same as the first name S. For all other clients, the port number FP will be the UDP port of the first created client, and the second name FS will be the name of the first created client. Subsequent clients will use FP and FS to connect themselves to the ring.

Each client should compute its Triad identity and enter the Triad ring. To compute the triad identity, it should take the nonce as a 4-byte, network-byte-order value, concatenate the client's name S, not including the newline nor the `'\0'` end-of-string terminator, and pass that to class-provided hash-function that will return a 32-bit unsigned integer. While we provide you the hash function with the signature `unsigned int projb_hash(unsigned char *buffer, int buffer_length)`, but you must assemble the buffer that is hashed as described above. (The TA will provide this code on the project moodle page.) You may check your job in that you should see these mappings from (nonce, name) to hash: (1234, foo) to 0xa9367d92, (12346, foo) to 0xcba8f538 (1234, bar) to 0xf970bbc8.

To enter the Triad ring, it should dynamically allocate a UDP port and start receiving Triad messages on that port. It should then determine and set up its successor and predecessor links. If it is the first node, it can do this itself (it knows it has no successor or predecessor, or it's probably cleaner to consider it to be its own predecessor and successor). If it is not the first node, it can do this by talking to the port of some other established node given to it by the manager. It will need to search around the ring to find it's proper location, then tell the old successor and predecessor it belongs between them. Finally, when it's set up, it should reply to the manager on the manager-client TCP port. This reply should have two lines, each in ASCII and terminated by a newline. The first should be the modified nonce as computed in Stage 1. The second line should be its UDP port for Triad messages.

Note that the client creation is carefully constructed so that the manager knows when the client is really ready. If the manager waits to hear this reply before continuing, then there should be no race conditions where the next client needs to contact another client that is not yet initialized.

Clarification 2011-10-06: For stage 2 and some following stages, you may assume that no clients are created after the first store or search. (This was said before as a comment in the sample input, but you may assume this is true for *all* stage 2 inputs.) This assumption is true *only* through stage 4; stages 5 and later allow nodes to come and go more flexibly.

Each clients should create a separate log file "stage2.S.out", where S is the name of the node (as the client finds out from the manager). When the client is created (*Clarification 2011-10-11:* just after it has joined the ring, but before it does anything else), it should log "client S created with hash SI", where S is the name and SI is the hash value it creates (in ~~base 10~~ *Change 2011-10-11* hexadecimal).

Triad Messages: Once the client sends its Triad UDP port to the manager, it needs to loop waiting for both Triad message (on the UDP port) and control messages from the manager (on the TCP connection). (Hint: I recommend you use a select loop so you can multiplex these two operations.)

Clients will send each other Triad messages to manage the ring:

Messages:

1. **successor-q** (ni) ask target with identity ni what it's successor is
2. **successor-r** (ni, si, sp) reply stating that ni's successor has id si and port sp
3. **predecessor-q** (ni) ask target ni what it's predecessor is
4. **predecessor-r** (ni, pi, pp) reply stating that n's predecessor has id pi and port pp
5. **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 *Change 2011-10-06:* this query is changed from the previous closest-q query.
6. **stores-r** (ni, di, ri, rp, has) reply stating that ni's best estimate of the node that stores di has id ri at port rp. In addition, 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.) *Change 2011-10-06:* this query is changed from the previous closest-r query, and the has parameter is added.
7. **update-q** (ni, si, sp, i) request node ni update its ith finger table entry, or the ~~successor~~ predecessor (*correction 2011-09-29*) entry if $i = 0$, to point to identity si at port sp. Entry $i = 1$ must be the successor, and in later stages, i can be as high as 32. Entry $i = 0$ is a special case and specifies the predecessor. For stage 2, i will always be 0 or 1 since we do not yet implement full finger tables, just successors and predecessors.
8. **store-q** (ni, SL, S) store string S of length SL at node id ni. Presumably the hash of S should be in ni's currently covered portion of the ring.
9. **store-r** (ni, R, SL, S) reply from ni with reply code R about storing string S with length SL, with reply code R. R should be zero if the store failed (for example, the S doesn't belong to ni), 1 if the store succeeded, and 2 if the string was already stored at the node.
10. *Added 2011-10-18:* **update-r** (ni, R, si, sp, i) *optional* reply to an update-q message. The reply code R shoudlb e zero if the update failed, otherwise 1. The other parameters are reflected back (in the unlikely event you have multiple outstanding update-q requests :-). This message is *optional*, it was added 2011-10-18 to allow a client to know when and update request completes. We recommend you use it because it can avoid race conditions when there are multiple sequential updates that must be done. However, because it is optional, we will not adjust your score if it is present or absent. (We may, however, deduct points if your program is not reliable because it has race conditions.) Please note that this message's code is out of order.

Each message should be encoded as the message code (1 for successor-q, 2 for successor-r, etc.), and each of the parameters stored as a 4-byte, network-byte-order integer. For store-q and store-r, the string length is given by the SL parameter and the string should follow with exactly that many additional bytes.

In the queries, passing *ni* is actually redundant, because the client receiving the message should know its id already. However, the receiver may wish to check that against its id to detect, to detect the unlikely event that the code might sometime have a bug.

The client should log all Triad messages it sends and receives to this log file in the order they are sent or received. Each log line should be the message name (listed above), the word "sent" or "received", the name of the message from the above list, and each parameter. Node identities should be printed in hexadecimal, and other integers in base-10. For example, a successor-r log statement might be "`successor-r sent (0xa9367d92, 0xf970bbc8, 5473)`".

If the client's name (given in the start_client configuration line) is "logmsg", then that client must also log the raw contents all messages that it receives, after it logs their text format. These log entries should start "raw 1234abcd..." where the 1234abcd are replaced with hexadecimal values of each byte. (It should not log messages it sends.) For example, if logmsg received the above successor-r message, it would log "`raw:  00000002a9367d92f970bbc800001561`".

Client interactions: The manager will read commands from the configuration file and respond in these ways:

start_client Handled as described above

store S store a key with name S (some string) in Triad.

The manager should pass a store command to the first created client. That client should then execute the store by finding the right client in the ring (presumably by sending repeated closest-q messages), then sending a store-r message.

The client executing the store command should convert the identity into a hash value with the same algorithm as node identities. (We don't actually store any value with the key, so clients will just track the presence or absence of a key.) You can assume S is at most 79 characters long.

After adding a key, the client executing the add (the first client) should log "add S with hash SH to node NI" to its log file. *Change 2011-10-11:* added "with hash SH" to the output; SH is the hash of S (and almost certainly different from the node identifier NI.

search S Find if key named S is in Triad.

Just like ~~"add"~~ "start_client" *(Correction 2011-10-18)*, but check to see if the key is present.

The client executing the add (the first client) should log "search S to node NI, key PRESENT" if the key is found, otherwise "search S to node NI, key ABSENT".

After the first client has executed a command, the it should send text text "ok" and a newline back to the manager over the client-manager TCP connection. This response will let the manager know it can proceed with the next command, avoiding race conditions.

You may assume there are fewer than 100 clients and 100 strings, although good code will not need these constants.

Sample input: Here is a sample configuration input for this stage ~~(we don't need nonce in this stage)~~ (*text removed 2011-10-06*):

```
stage 2
# you are guaranteed a nonce line before any clients are started
nonce 1234
start_client foo
# there may, of course, be comments anywhere
start_client bar
start_client baz
# you can assume for stage 2 that no clients start after the first store or search
store alpha
store beta
store gamma
search beta
search delta
```

(Note that you should also test a configuration file with a client called "logmsg".)

Sample output: File stage2.foo.out:

```
client foo created with hash 0xa9367d92
successor-q received (0xa9367d92)
successor-r sent (0xa9367d92 0xa9367d92 38708)
update-q received (0xa9367d92 0xf970bbc8 58366 0)
update-q received (0xa9367d92 0xf970bbc8 58366 1)

successor-q received (0xa9367d92)
successor-r sent (0xa9367d92 0xf970bbc8 58366)
update-q received (0xa9367d92 0xdabfd86c 34089 1)
stores-q sent (0xdabfd86c 0xbf0a5ea2)
stores-r received (0xdabfd86c 0xbf0a5ea2 0xdabfd86c 34089 0)

store-q sent (0xdabfd86c 5 alpha)
store-r received (0xdabfd86c 1 5 alpha)
add alpha with hash 0xbf0a5ea2 to node 0xdabfd86c
add beta with hash 0x7eba5174 to node 0xa9367d92
add gamma with hash 0x3cee9126 to node 0xa9367d92

search beta to node 0xa9367d92, key PRESENT
search delta to node 0xa9367d92, key ABSENT
```

We will provide sample output, announced on the class moodle, in the next week.

File stage2.bar.out:

```
successor-q sent (0xa9367d92)
successor-r received (0xa9367d92 0xa9367d92 38708)
update-q sent (0xa9367d92 0xf970bbc8 58366 0)

update-q sent (0xa9367d92 0xf970bbc8 58366 1)
client bar created with hash 0xf970bbc8
update-q received (0xf970bbc8 0xdabfd86c 34089 0)
```

There will also a stage2.baz.out.

Contents of stage2.README.txt: 2.1) Have you implemented all the requirements of this stage? if not, which are missing?

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

For this stage, most of the points will be evaluated to assessing if your code runs, and we will do that by feeding it several input configuration files.

3.3 Stage 3: Demonstrating Linear Search

Basic Triad has linear search.

For this stage, you will set the nonce to a value specific to you and look at search performance.

Sample input:

```
stage 3
# note, on the nonce line,
# replace dddd with the first four digits
# of your student id
nonce dddd
start_client alpha
start_client bravo
start_client charlie
start_client delta
start_client echo
start_client foxtrot
store alpha
store beta
store gamma
store delta
store epsilon
search beta
search delta
```

Sample output: Output format is exactly the same as stage 2, but with the filename changed from "stage2.S.out" to "stage3.S.out". Specific output will vary because of the use of different nonces.

Contents of stage3.README.txt: 3.1) If you use 1234 for the nonce, how many steps does it take to search for "beta"?

3.2) If you use 1234 for the nonce, how many steps does it take to search for "delta"?

3.3) What are the first four digits of your student ID?

3.4) With your student ID for a nonce, how many steps does it take to search for "beta"?

3.5) With your student ID for a nonce, how many steps does it take to search for "delta?

3.4 Stage 4: Adding Finger Tables

It's important that real Chord has $O(\lg n)$ search time, not linear. While perhaps not critical for our small network, with millions of nodes linear search would be untenable. Finger tables are the key to guaranteeing, with high probability, that we will get logarithmic search time for a large network.

This stage has the same input format as stages 2 and 3, except that the stage number changes, and all clients are expected to create and maintain finger tables that cover the entire hash space.

Comment 2011-10-18: Note that when a node is added you may have to update several finger tables. You may do multiple updates recursively (client A updates B, and B notices it must update C and so it does so before it returns), or you may update them iteratively (client A updates B, then A determines it must update C, so A updates C directly.) ([Stoica00a] Figure 6 suggests recursive updates, but you may do either.)

Sample input: Same as stage 3, except the first line is "stage 4".

Sample output: Output filenames should be like stage 2, but with the filename changed from "stage2.S.out" to "stage4.S.out". The end result should be the same as stages 2 and 3, but you should find you do less searching.

Contents of stage4.README.txt: 4.1) Have you implemented all the requirements of this stage? if not, which are missing?

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

4.3) If you use 1234 for the nonce, how many steps does it take to search for "beta"?

4.4) If you use 1234 for the nonce, how many steps does it take to search for "delta"?

4.5) With your student ID for a nonce, how many steps does it take to search for "beta"?

4.6) With your student ID for a nonce, how many steps does it take to search for "delta?

3.5 Stage 5: Starting Dyanmics with Node Departure

While setting up a Triad network takes some work, the real challenge in peer-to-peer systems like it is network *dynamics*, as nodes come and go. Handling dynamics can become particularly difficult when multiple nodes are entering and leaving the network at the same time, and in a large enough network, it is *always* the case that nodes are coming and going.

The next stage adds controlled node departure to Triad. *Clarification 2011-10-06:* this change removes the guarantee from stage 2 that nodes do not change after the first search or store command. Nodes can now leave after a search, although new nodes cannot (yet) join. (In Project C we will add additional dynamics.)

This stage requires one new configuration line: "end_client S", which indicates that the client with name S should do a controlled departure. (*Clarification 2011-10-15:* as with other manager-client commands, the client needs to send OK after it's completed this command.) The manager will need to keep track of client names so it can send this message to the correct client.

Handling client departure requires four new Triad messages:

leaving-q (ni, di) tell node ni that node di is leaving the network. Node di will send this to its successor ni to allow ni to take over di's job.

On receiving this message, ni should fetch all of di's data using the next-data-q message described below. When it has all the data, it should adjust it's successor link to skip di, and tell it's new successor to update its predecessor. (With the update-q message where $i = 0$.)

leaving-r (ni) is the reply to leaving-q, indicating all data has left the network.

next-data-q (di, id) is a request to di give the next data item stored at that node with hash less then or equal to than id. It responds with next-data-r and the data item.

next-data-r (di, qid, rid, S) is the reply; it indicates that di's next data item after qid is rid with value S. If the di stores no data less than qid, it should return rid as one less than the beginning of its range and an zero-length S.

The idea of next-data-q and next-data-r is to that the node that's taking over starts asking the departing node for its data, then it counts backwards around the id space until it has fetch all data from that node. (Remember that all id arithmetic is modulo around the ring, so 0x0 is before 0x1, and 0xffffffff is before 0x0.)

Sample input:

```
stage 5
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
end_client charlie
store delta
store epsilon
search beta
search delta
```

Sample output: Again, the filename should include “stage5”. We will provide sample output, announced on the class moodle, in the next week.

Contents of stage5.README.txt: 5.1) Have you implemented all the requirements of this stage? if not, which are missing?

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

4 Hints about Project C

Project C is still under revision, but we are expecting it will include some extensions to your Triad implementation, and probably some application using Triad.

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 `projb.tar.gz`.

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 `projb` (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.)