EE122 Project 2

Fall 2010

Version 0.1

Part A due at 11:50pm on Wednesday, November 17, 2010
Part B due at 11:50pm on Friday, December 10, 2010

Change History:

Preface

A couple of students have asked us if we can make project 2 use some libraries or frameworks. We have given these requests much thought and would like to explain our decision and rationale. First of all, it is definitely true that complex applications written these days rarely use sockets directly. A higher level library can save you time by managing your connection and shielding you away from the hairy select() interface. It can also provide other functionality like timers, implement a higher level protocol (e.g. HTTP), and provide a more intuitive interface for packet construction and parsing. Knowing all of these important benefits it was rather hard to decide on the course of action, but we decided to stay with sockets for the following reasons.

The goal of this project is to expose to you to some of the higher level networking issues that appear in distributed and peer-to-peer systems. You will maintain a set of active servers dynamically, will use a simple mechanism to broadcast a message to all clients independently of which server they are connected to, will see a very cool and novel idea for efficient state distribution called consistent hashing, and will write a cracker app that uses our primality service (cloud computing). As you can see there are quite a few higher level ideas that we want you to have some experience with.

Asking you to use an extra library would divert the focus and cause you to spend a lot of time learning the library. Most of the libraries worth learning have a non-trivial learning curve and we think that it might be too much of a burden in the available time. Moreover, debugging your code written in a higher level library is usually much harder - you cannot easily look at the packets in wireshark and there is a level of abstraction between you and the network so that when something goes wrong you cannot be sure if your logic is wrong or you are simply using the library in a wrong way. On the flip side, you are already familiar with sockets and can directly dive in, and as a side note, sockets is a very good API for its level of abstraction. Finally, we will spend a portion of a lecture talking about libraries worth noting, their goods, and bads.

Introduction

Your distributed primality computation system has been proving very successful. But you have

encountered two problems. First, as the number of clients grow, your server cannot keep up assigning jobs and gathering the results. Second, there is currently no way of actually using the database of prime number calculated by your system. You will solve these problems in project 2.

As you started thinking about the first problem, you immediately realized that the only solution is to run multiple servers. For example, if you have 1000 clients and two servers, you can distribute the clients and have 500 clients per server. If you have 50 servers, and clients are well distributed, you will have an average of only 20 clients per server. However, to run this distributed system successfully, you realize that there are a few things that you need to address.

The first issue is how a client will know to which server it should connect. Luckily, you remember that in one of the lectures you learned about a name resolution system called DNS that can return a lsit of IP addresses given a name. So, you figure that you will name your service something like bears-primality-service.com and DNS will return a list of IP addresses of your servers. Then client can chose one address randomly and connect to it.

The second issue is how to add a server to a running system. You figure that you can update the DNS entry by hand, but how will the already running servers know that a new server joined. You remember that the cluster you will be running your servers on has a networked file system and you come up with a very simple idea that will work for your purposes. You will have a special file called “peers.lst” that will contain the ip addresses and the ports of your servers. When you start a new server, it will read this file to learn about running servers and will add an entry for itself to the file. This way a new server A will learn about all the running servers and the subsequently started servers will learn about all the running ones including A. But, how will the running ones learn that A has joined? Among many possible solutions, you decide that upon startup A will contact each running server notifying it of A’s arrival. For this, you will need a new message type - NEW_SERVER.

The third issue is that you still want the SPEAK messages to be broadcasted to all clients, independently of what server they are connected to. It is clear that when a server gets a SPEAK message from a client it should broadcast it not only to all of the connected clients but also to all of the running servers. Then, each receiving server can broadcast the message down to all of its clients. The question then is how to broadcast the SPEAK message to all of the servers. One possible solution is to open a TCP connection between each pair of servers. However, if you have just 100 servers, you will have 10,000 connections. And if you imagine that in google one query is answered with 12,000 servers, how many connections would there be if each server was directly talking to all other servers - 144,000,000. So, this solution is not very scalable. Another option is to establish TCP connections on demand - when we get a speak, we open a connection, send the packet, and close the connection. However, you immediately remember that TCP has a 3-way handshake and so you will be sending 3x more packets than necessary. Finally, you remember that loosing a SPEAK message is not critical and that UDP does not require connection establishment. So, you choose to send inter-server SPEAK messages over UDP. With this decision, you feel comfortable about the first problem and jump on the second one.

As you started thinking about the second problem, you realized that for your service to be useful you actually need to remember the results. You imagine that if you have 10 servers, you can partition the space of all numbers (whose primality you want to check) into 10 ranges and assign each range to one server. Each server will then assign jobs from its own range. You also imagine that you will have a cracker application using our service to factor numbers. When the cracker application will need to check the primality (or get a prime factor) of a number it can know which server to ask for this number if we tell it about the ranges. You also realize that you can use the existing JOB_ASSIGNED and

JOB_COMPLETED messages to implement the cracker app querying the server and getting the reply, respectively. The only difference is that these messages will be traveling over UDP to minimize the latency and not burden the server with extra TCP connections from cracker apps. All of these seem great as you are reusing much of the existing design. However, when you try to think what should happen with the ranges when a new server is added you realize that it is not immediately obvious.

To see the problem, lets explore some simple possible options. Imagine that you have 10 servers S_1, S_2 ... S_{10} and you split the space of numbers into 10 continuous ranges r_1, r_2 ... r_{10}. Then, a new server S_{11} joins. If we now split the space into 11 continuous ranges, around 1/3 of the state will need to be transfer between server to redistribute the ranges. Moreover, some servers will need to transfer much more than others creating bottlenecks. So, this approach seems rather bad. Another approach is to have all of the existing 10 servers split their ranges in two (in proportion 10 to 1) and transfer the smaller range to the new server. Then, all servers will have equal (1/11th) portions of the space but the new server will have 10 small ranges and other servers will have one large continuous range. You can see that if we try to accommodate one more server, the picture will look even more complex. You don't even want to think how to redistribute the ranges when some server leaves. So, you ask if there a better solution where whenever a server joins or leaves, minimal state is transferred, each server transfers similar amount of state (so there are no bottlenecks and inhomogeneity), the number of ranges does not grow with more servers, and it is easy to calculate. Luckily the answer is "yes" and the technique is called "consistent hashing".

The consistent hashing technique was first proposed in 1997 and since then has greatly grown in popularity, especially in cloud computing applications like our primality service. Virtually every large web services company uses consistent hashing. We won't describe the details of consistent hashing in here because there is plenty of material on-line. Some good references are: http://weblogs.java.net/blog/2007/11/27/consistent-hashing http://www8.org/w8-papers/2a-webserver/caching/paper2.html (one of the original papers)

Having learned about consistent hashing you think you have all the pieces to decide on the details of the implementation that are described in the following sections.

Part A

In part A, you will implement everything except for the consistent hashing and simple cracker application. In the following sections, we will describe all the pieces.

To simplify the problem and decrease the number of annoying details we make a few assumptions.

- We won't handle the case of a server leaving. We will be only adding servers to the pool.
- We will run all servers on the same machine and from the same directory so that all of them have easy access to peers.lst.
- We will not worry about how the workers will select the server to connect. It is not hard to implement the scheme described in the introduction, but it will require much setup and will add a large source of possible problems. When starting our workers, we will specify the server to contact on the command line ourselves.
- We will not worry about invalid/malformed packets. In other words, you can assume that everyone in the system is precisely following the protocol. (There are a few places were you would want to print an error message for your own sake. We will ask you to do it below.)

Server Command and Client Ports

In Project 1, our server had a single port that it was listening on. Let's call this port a **client TCP port**. In project two the server must also be able to accept connections from new servers. While it is possible to have the servers connect to the same client TCP port, in practice people prefer to have a separate port for connections from "inside". For example, if client port is different from the server port, one can add a firewall rule to disallow connections to the server port from outside of the cluster. Thus, we will have a separate TCP port on which servers will be listening for connections from other servers. We call this port a **command TCP port**.

Recall that our servers will also need to listen for UDP packets to receive SPEAK messages from other servers and JOB_ASSIGNED message from cracker applications. We will have a single **UDP port** for both of these messages and to avoid extra configuration will will make the port number be the same as the **client TCP port** number. If binding to the UDP port fails, your server must call **on_udp_bind_failure()**.

When starting the server, we will specify two port numbers as follows:

```
./server -p <client-tcp-port-number> -c <command-tcp-port-number>
```

Peer List

Before starting the first server, peers.lst must be empty but present in the same directory as the server. Each line of peers.lst must have the following format

```
<dotted-ip-address-of-the-server> <cmd-tcp-port> <client-tcp-port>
```

For example,

127.0.0.1 5511 4411

There must be exactly one space between each of the fields and no spaces at the end. It is ok, to hard code the ip address of the server to be 127.0.0.1.

On startup the server must open the peers.lst file, read all the info about running servers, append itself to the file and close it. The server should not do any significant work while the file is open. You don't need to worry about race conditions between servers while working with the file because we won't be starting the servers simultaneously. However, your server should not have the file open for a long time (0.1 second is a very long time).

NEW_SERVER Message

When a new server reads the peers.lst file and finds some already running servers there, it must contact each of them sequentially (in the same order as they appear in the peers.lst) and send them the NEW_SERVER message notifying them that it has joined. Here is a detailed sequence of steps that a newly started server needs to carry out (you can obviously change them as long as your steps have the same effect):

- Read the peers.lst remembering the information about the peers in some local data structure. For each read peer, call the **`on_read_peer_from_list()`** function.
- Append itself to peers.lst and close the file. (It is ok to open and close the file twice)
- For each of the peers from the file, in the same order, do the following
    - Connect to the TCP command port of the peer. If connection fails, call **`on_notify_peer_failure()`** function.

- Construct the NEW_SERVER message.
- Send the NEW_SERVER message to the peer using a **blocking** send. (In part B, you will receive a STATE_TRANSFER message in response). If send fails, call **`on_notify_peer_failure()`** function.
- Close the connection.

From the other side, when your server starts it should start listening not only on the client TCP port, but also on the command TCP port (and UDP port). When it receives a new connection on the command TCP port, it should accept it and can assume that it is a peer server. The server must not be blocked on incoming connections from peer servers (i.e. you should use the same select() mechanism). When the server receives NEW_SERVER message from the peer, it should remember the peer information to be able to later send UDP SPEAKs to it. It must also call **on_new_server_from_peer()** function. The receiving server must not close the connection. The connection must be closed by the initiating server.

If helpful, you can assume that at any time at most one peer will be connected sending NEW_SERVER message.

The NEW_SERVER message looks very similar to the messages from project 1.

version (8 bits)	length (16 bits)	length (cont'd)	type (8 bits)
ip (4 bytes)	ip (cont'd)	ip (cont'd)	ip (cont'd)
cmd_port (2 bytes)	cmd_port (cont'd)	client_port (2 bytes)	client_port (cont'd)

- **ip** is the IP address as a 32bit number in network byte order.
- **cmd_port** is the command TCP port number in network byte order.
- **client_port** is the client TCP port number (and the UDP port number) in network byte order.

UDP SPEAK Message

UDP SPEAK message has exactly the same format as the regular SPEAK message. When the server receives a SPEAK message from the client, it must broadcast it to all the connected clients as in project 1 and also send a UDP SPEAK to all the peers (both the ones it learned from peers.lst and the ones that sent it a NEW_SERVER message).

You can use blocking send() when sending UDP messages. The reason is that UDP almost never blocks. It will block only if you have used up all kernel buffer space, which means that you must be sending faster than your NIC's speed (or, if you are sending to the same machine, faster than the kernel can copy). It is a safe assumption that our SPEAK messages will not reach such speeds.

When a server receives a UDP SPEAK from a peer, it should simply broadcast it to all the connected clients (Hint: since it is the same format and we don't check for error, you can just copy the packet as is :)).

Part B

To be filled in.

Provided Files

Most of the files are self explanatory and are largely the same from project 1. Here are the differences.

--- server.c ---
Contains the reference implementation of project 1's server slightly adopted to project 2 so that it compiles and runs with project 2 testing script.

--- Makefile ---
Contains rules to build the client (not needed for project 2), server, and cracker application. Here are the commands that you can use: make, make client, make server, make cracker, make clean, and the hand-in commands.

--- ranges.h/c ---
These files are currently empty, but will soon contain a consistent hashing library for part B.

--- server.c ---
Contains the reference implementation of project 1's server slightly adopted to project 2 so that it compiles and runs with project 2 testing script.

--- peers.lst ---
You can use this file as your peers.lst file. It should be empty when the first server starts. A quick way to empty it is "echo -n > peers.lst"

--- utlist.h and uthash.h ---
These are external list and hash table libraries that will be used by consistent hashing library. You probably don't need to worry about them.

--- cracker.c ---
This is an empty file where you will write your cracker app using our distributed primality service.

Discussion

We expect the second project to take less time than the first one and Part B to be easier than Part A. The role of this project is to give you some experience with distributed systems. We thought about doing something purely networking - like congestion control - but thought that practical experience with the issues in distributed and peer-to-peer systems is more valuable.

You might feel that a significant part of this project is very similar to the first project and in some sense that is correct. What you should learn from these similar parts is how to juggle different types of

connections and cleanly process them. In some sense it is an exercise in software engineering in the context of
networking.

As mentioned earlier, you are free to base your implementation either on the code we released or on your own implementation of project 1. If you decide to go with our reference implementation, note that it is in no way perfect. Think of it as just some implementation. In particular, its design was not tweaked to easily accommodate project 2 requirements. So, you will probably need to restructure some things.

Policies

[Unchanged from Project 1] We very strongly encourage you to work on the project with your peers. However, you must adhere to the collaboration policy outlined below.

- All source code must be written solely by you.
- You can discuss anything related to a project with anybody. If discussion happens in writing, it must not include code.
- You must not read other students' code, with the exception of helping them debug a problem which they tried to, but could not, figure out themselves.
- You are welcome to read online tutorials and example code. However, you must not "copy-paste" significant (>10 lines) portions of code.
- We strongly encourage you not to use non-standard libraries and build the whole system yourself. It will feel much better :) and you will have learned the fundamental underlying primitives. However, we realize that some students might have had extensive experience in socket programming before and might like to spend their time learning something new. For such cases, **you must contact us and ask for permission to use an additional library**. In return, we might ask you to implement an additional feature.

Be waned that you might use program similarity tests and/or tools like Google code search to identify breaches in collaboration policy. We have all been students and know how tempting it might be especially when the load of other classes is very high. To avoid such situations, and to actually learn a very useful skill, start the project early! Then, if you are stuck and your friends can't help you, come to TA's office hours. You can either bring a laptop, if you have one, or some way to get your files. TAs can put your files in a virtual machine and look at the problem with you.

Grading

Grading will be done similarly to project 1 - we will have a testing script similar to the one we released for your convenience. The only difference is that you are still responsible for project 1 server tests. The reason is that while implementing new features you should not break existing ones. Your final grade is the total number of points you get from project 2 minus the total number of points you broke in project 1. So, if you don't do anything and submit the code that we provided you get zero (0 - 0 = 0). If you pass all project 2 tests and don't break any project 1 tests you get 100 (100 - 0 = 100). We weight project 1 tests lower than project 2 tests, so the penalty for breaking project 1 tests is not very high.

Project 2 is 20% of your final grade. Part A is 10% and Part B is 10%. If you implement an additional networking related feature, you can earn an additional 10% for each part.

Not following hand-in instructions can cost you up to 20% of the project grade.

Hand-in Instructions

[slightly changed from project 1] You must submit a single tar archive through bspace for each of the two parts of the project. If you don't use any additional libraries and place all of you code in server.c (for part a) and server.c cracker.c (for part b) (let's call this the "normal case"), the tar archive for submission can be generated using the following commands:

```
STUD_ID=12345678 make handin_2a   # for part a
STUD_ID=12345678 make handin_2b   # for part b
```

where 12345678 must be replaced by your student id. The first (second) command will generate project2a_12345678.tar.gz (project2b_12345678.tar.gz) file that must be uploaded.

Besides the source code, the submission archive must contain **Makefile** and **readme.txt** files. In the normal case, you don't need to touch the Makefile and the readme.txt must be empty. If, however, you are using a non-standard library you provide instruction in readme.txt on how to install it in the virtual machine. Also, if you have implemented an additional feature, you must describe it in the readme.txt file and provide instructions how we can see the feature in action.

After installing any additional libraries (if any), we will run "`make server`" to build your server executable for part A. We will run "`make server`" and "`make cracker`" to build both server and the cracker binaries for part B. If you choose to follow a path different from the "normal case", you must make sure that these commands produce the server and the cracker binaries named "server" and "cracker", respectively.