Project 2 (100 points)
Assigned: Thursday, March 22, 2018
Due: Thursday, April 5, 2018, 11:59 PM

CS-4513, Distributed Systems
D-Term 2018

CS4513 Project 2 — A Distributed Shell

For this project, you are strongly urged to work in teams of two or three.

Description

The main purpose of this project is to provide some basic experience in writing a distributed system, albeit a simple one, while reinforcing some of basic OS knowledge. You will write a basic *remote shell* in which a user specifies an arbitrary shell command to be executed on a local computer, and that command is sent over a network and executed on a remote server, with the results sent back over the network to be displayed on the local computer.

You are to write a server and a client, described as follows:–

- *Server:* The server runs on the remote machine. It binds to a TCP/IP socket at a port known to the client (there should be a way to change the port at run-time in case of conflict). When it receives a connection, it calls **`fork()`** to spawn a child process to handle the connection. The parent process loops back to wait for more connections. The child process first authenticates the client via the protocol:–
    - Client sends user-name to server (but *not* password)
    - Server responds by sending back unique random number (using **`rand()`** and **`srand()`**)
    - Client encrypts using user's password plus number as key
    - Client sends hashed/encrypted value back to server
    - Server encrypts using the user's same password plus number as key
    - Server compares two hashed/encrypted values, if same then ok

  If authenticated, the server executes the given shell command via an **`exec()`** call, returning all data displayed via **`stdout`** and **`stderr`** to the client. For this project, the server may assume that the shell command does *not* use **`stdin`**. Upon completing the command, the child process exits. Note, that the original server process will still be around, waiting for additional connections.

- *Client:* The client runs on your local Linux (virtual) machine. From the command line, the user specifies the host where the server resides and the command to be executed. The client then connects to the server via a socket and transmits the username to start the authentication protocol shown in the server above. If authentication is successful, the client sends the command. The client displays any output received from the server to **`stdout`** and then exits.

You should develop and debug the client and server locally using two Linux virtual machines such as provided by this course or by your Operating System course. With your teammate(s), set up the server on one virtual machine and one or more clients on different virtual machine(s).

Cloud Computing

After implementing the Distributed Shell (client and server above) and debugging carefully, port your *server* to the Amazon Elastic Compute Cloud (EC2). Specifically:–

1. Start with Getting Started with Amazon EC2 Linux Instances. Following the documentation, do the following steps:

   a) Sign up for Amazon Web Services (you will be using the AWS Free Tier).[1]
   b) Launch a Linux virtual machine (in Amazon Cloud terminology, *an instance*). Recommended is the Ubuntu HVM (Hardware-assisted virtual machine), but other distributions may work.[2]
   c) Connect to your instance (preferably, via Putty or SSH).
   d) Clean up your instance (when done).

2. Install needed software on your AWS Linux instance. This includes development software and performance evaluation software. Specifically, you will want to install a minimum of:

   `a) gcc` (or **`g++`**)
   `b) make`
   `c) sysbench`

   This is most easily done from the command line, **`sudo apt-get install {name}`** where **`{name}`** is one of the packages above (e.g., **`sudo apt-get install gcc`**.

   By default, Amazon automatically blocks TCP (and ICMP) requests. To fix this, go to *Security Groups* (on the left panel), select your new instance (not the default), under Inbound, hit **`edit`** and ADD TCP. See Authorizing Inbound Traffic for Your Linux Instances or Why can't I **`ssh`** or **`ping`** my brand new Amazon EC2 instance? for more information.

3. Copy your server source code to your AWS instance. Build and run it. Make sure you can connect to it from your client and run it normally.

---

1 Although you will be signing up for the Free Tier, Amazon will still want your credit card before it accepts you.
2 Note that the Linux Virtual Machine provided by AWS is different from the Linux Virtual Machines distributed in your OS course, even though both are Ubuntu Linux systems.

Refer to the Amazon Elastic Compute Cloud Documentation for more information, including guides and the API reference.

Note, if doing your local experiments (see below) on a virtual machine from a WPI course, you will also need to install **sysbench** there. You do this by building from the source:

```
git clone https://github.com/akopytov/sysbench.git
cd sysbench
./autogen.sh
./configure --without-mysql
make
```

The **sysbench** executable will be under the **sysbench** directory.

Experiments

After setting up your Amazon cloud server instance, design experiments to measure:–

1) network throughput, and
2) CPU and I/O performance.

Then, create a simple algebraic model to predict how much data would need to be transferred for cloud computing to be more efficient than local.

Network performance

Measure the following:–

1) the amount of time required (the latency, in milliseconds) to setup a connection to the server, authenticate, and tear it down, and
2) the maximum throughput (in bits per second) from the server to the client.

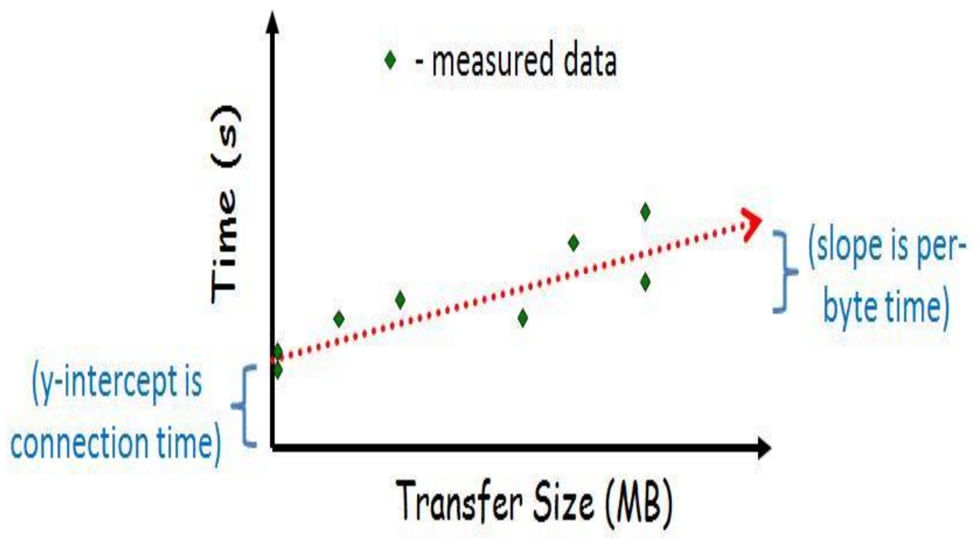
For both sets of measurements, you need to do multiple runs in order to account for normal variance in the data between runs.

To measure the connection-tear down time, consider forcing the client to make a call to the server that does not have the server to do an **exec()** or any other significant processing. Since the time scale for the first test is very small, you will measure the time for many operations and then divide by the number of operations performed. You need to build a harness (a program, shell script, **perl** script or something similar) to make repeated connection setup-tear down requests.

To measure throughput, consider forcing the server to send an amount of data (of a known size) to the client. Note, the client output, by default, goes to **stdout** so you may want to consider redirecting **stdout** on the client to a file (via the "**>**" redirection shell operator).

In order to record the time on your computer (instead of, say, looking at the clock on the wall) you should use the **gettimeofday()** system call from a program, or the **time** command from a shell (note, some shells have a built-in **time** command, too). You can also do something similar in a scripting language of your choice (e.g., **localtime()** in **perl**).

For illustration purposes, a possible graph is:



CPU and I/O performance

You will use **`sysbench`** to benchmark *both* your local machine (e.g., your WPI virtual machine) and your Amazon Linux instance.

Benchmark the CPU performance with the following:

```
  sysbench --test=cpu --cpu-max-prime=20000 run
```

There will be a lot of numeric output, but the most important is the total time (e.g., **`3.5160`**s).

Benchmark file I/O performance by first creating a test file that is much bigger than your RAM (otherwise, you get memory performance). Check the RAM on your system with **`free -h`** and use at least twice this value for the file size. Say this is 16GB, then prepare files via:

```
  sysbench --test=fileio --file-total-size=16G prepare
```

Then, run the benchmark:

```
sysbench --test=fileio --file-total-size=16G --file-test-mode=rndrw
--init-rng=on --max-time=30 --max-requests=0 run
```

The important number is the Mb/sec value (e.g., 1.19779 Mb/sec).

When done, delete the prepared files from your system:

```
  sysbench --test=fileio --file-total-size=16G cleanup
```

Model

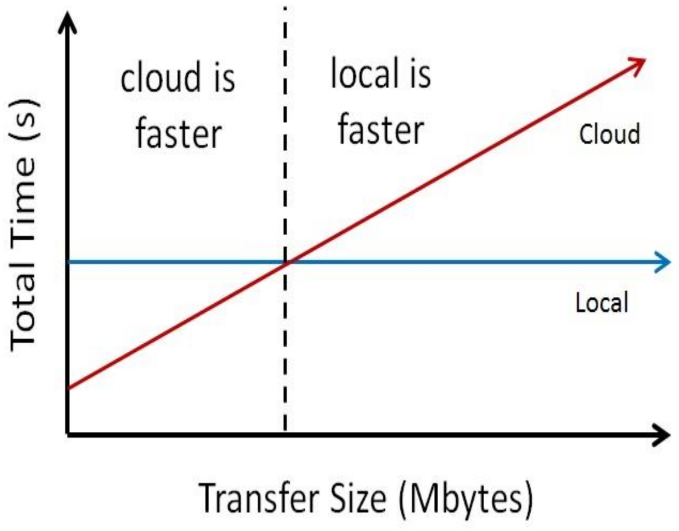
Develop a simple, algebraic model to compute when (for how much data that needs to be transferred) computing in the Amazon EC2 cloud provides better performance (i.e., a faster turn-around time on the task). Assume the CPU task used in your **`sysbench`** benchmarking (computing and testing 20000 primes) and File I/O of 16 GB.

Specifically, solve the equation:

`Local_CPU + Local_File_I/O = n * Network + Remote_CPU + Remote_File_I/O`

for *n* (in bytes), where all terms are in time. Network is the network performance, determined earlier and CPU and I/O are the CPU and I/O performance, respectively, determined earlier.

Visually, the relationship may be as in the below graph:



Write-up

When your experiments are complete, you must turn in a brief (two pages or so) write-up with the following sections:–

1) *Design* – describe your experiments, including: a) what programs/scripts you ran in your harness and what they did (use pseudo-code); b) how many runs you performed; c) how you recorded your data; d) what the system conditions were like (including information on hardware and software); e) and any other details you think are relevant.
2) *Results* – depict your results *clearly* using a series of tables or graphs. Provide statistical analysis including at least mean and standard deviation. Include graphs for network per-byte cost and local versus remote costs.
3) *Analysis* – interpret the results. Briefly describe what the results mean and what you think is happening and any subjective analysis you wish to provide.

Hints

To get help information about specific Unix commands, use the "**`man`**" command. For instance, entering "**`man tcsh`**" will display the manual page entry for the **`tcsh`**. If you specify a number after the word "**`man`**" it looks in the indicated section number (e.g. "**`man 2 bind`**") to display the **`bind()`** system call instead of bash's built-in **`bind`** command.

The following system calls for setting up your sockets may be helpful:

- **`connect()`**
- **`accept()`**
- **`socket()`**
- **`listen()`**
- **`bind()`**
- **`close()`**
- **`send()`**
- **`recv()`**
- **`getservbyname()`**
- **`gethostname()`**
- **`gethostbyname()`**
- **`gethostbyaddr()`**

You might also see Beej's Guide to Network Programming for socket information.

The following system calls for the shell aspects of your programs might be helpful:

- **`fork()`** – to create a new process.
- **`execve()`** – to execute a file. The call `execvp()` may be particularly useful.
- **`getopt()`** – help in parsing command line arguments.
- **`strtok()`** – to help in parsing strings.
- **`dup2()`** – to redirect stdout, stderr to socket

The following system call will be useful for the authenticating part of your program:

- **`crypt()`** – create a cryptographic hash. Note, need **`#define _XOPEN_SOURCE`** *before* **`#include <unistd.h>`**.

Some sample code may be useful:

- **`talk-tcp.c`** and **`listen-tcp.c`** - helpful samples for doing socket code.
- **`fork.c`** - showing the simple use of the **`fork()`** call.
- **`execl.c`** - showing simple use of the **`execl()`** call.
- **`get-opt.c`** - code that parses command line arguments (fairly) painlessly.

You might also see the class slides on sockets.

Beware of the living dead! When the child process of a server exits, it cannot be reclaimed until its parent gathers its resource statistics (typically via a **`wait()`** call or a variant). You must figure out how to do this using **`waitpid(`**, since **`wait()`** blocks; alternatively, you can use **`wait()`** in conjunction with a **`signal()`** that triggers when a child exits.

When running your experiments, you need to be careful of processes in the background (say, a Web browser downloading a page or a compilation of a kernel) that may influence your results. While multiple data runs will help spot periods of extra system activity, try to keep your system "quiet" so the results are consistent (and reproducible).

In Linux, you can find information on hardware (and kernel version) with the commands:–

- **`lscpu`** - CPU information (also **`cat /proc/cpuinfo`**)
- **`lsblk`** - block device information (also **`/df/mount | column -t`**)
- **`free -h`** - available memory
- **`uname -a`** - OS version (also **`cat /proc/version`**)

Performance evaluation experiments should always report information on relevant hardware and software.

The guide How to Benchmark Your System with Sysbench may be helpful for additional information on **`sysbench`**.

You might look at the brief slides for some overview information.

For added security, a "real" server would likely use **`chroot()`** to change the root directory of the process. For example, many anonymous ftp servers use **`chroot()`** to make the top directory level **`/home/ftp`** or something similar. Thus, a malicious user is less likely to compromise the system since it doesn't have access to the root file system. Note, however, **`chroot()`** requires root privilege to run, making it unavailable to the common user (e.g., on course virtual machines). For developing on your own system (say, a Linux box), we encourage you to explore using **`chroot()`** (do a "**`man 2 chroot`**" for more information). Please mention if you do this in your **`README.txt`** file or similar documentation when you turn in your program. Make sure your client programs run on the course virtual machines.

Examples

Here are some examples (in this case, running on the WPI CCC machine). The server:

```
claypool 94 ccc% ./server -h
distributed shell server
usage: server [flags], where flags are:
        -p #    port to serve on (default is 4513)
        -d dir  directory to serve out of (default is /home/clay-
pool/dsh)
        -h      this help message

claypool 95 ccc% ./server
./server activating.
        port: 4513
         dir: /home/claypool/dsh
Socket created! Accepting connections.

Connection request received.
forked child
received: john
password ok
command: ls
executing command...

Connection request received.
```

```
forked child
received: john
password ok
command: ls -l
executing command...

Connection request received.
forked child
received: john
password ok
command: cat Makefile
executing command...
```

The client (login name john) from the same session:

```
claypool 40 capricorn% ./dsh -h
distributed shell client
usage: dsh [flags] {-c command}, where flags are:
        {-c command}    command to execute remotely
        {-s host}       host server is on
        [-p #]          port server is on (default is 4513)
        [-h]            this help message

claypool 41 capricorn% ./dsh -c "ls" -s ccc.wpi.edu
Makefile
client.c
dsh
dsh.c
index.html
server
server.c
server.h
sock.c
sock.h

claypool 42 capricorn% ./dsh -c "ls -l" -s ccc.wpi.edu
total 37
-rw-r-----   1 claypool users         212 Nov  7 22:19 Makefile
-rw-r-----   1 claypool users         997 Nov  1 09:27 client.c
-rwxrwx---   1 claypool users        6918 Nov  9 00:04 dsh
-rw-r-----   1 claypool users        3790 Nov  9 00:03 dsh.c
-rw-r-----   1 claypool users        5374 Nov  8 23:50 index.html
-rwxrwx---   1 claypool users        7919 Nov  9 00:09 server
-rw-r-----   1 claypool users        4383 Nov  9 00:09 server.c
-rw-r-----   1 claypool users         240 Nov  7 22:19 server.h
-rw-r-----   1 claypool users        2638 Nov  1 09:36 sock.c
-rw-r-----   1 claypool users         614 Nov  1 09:27 sock.h

claypool 43 capricorn% ./dsh -c "cat Makefile" -s ccc.wpi.edu
```

```
#
# Possible Makefile for distributed shell program
#

CC = gcc
CFLAGS = -Wall
LIBFLAGS =

all: dsh server

server: server.c
	$(CC) $(CFLAGS) server.c server.o -o server $(LIBFLAGS)

dsh: dsh.c
	$(CC) $(CFLAGS) dsh.c dsh.o -o dsh $(LIBFLAGS)

clean:
	/bin/rm -rf *.o core dsh server
```

Hand In

For your client-server shell, the main information you need is:

- name
- login
- system dependencies
- brief notes on how to run client and server
- examples of it running

All this information should appear in a **`README.txt`** file that accompanies your program.

In addition, be sure to have your experiment writeup in a clearly labeled file in **`pdf`** format named **`Project2-username.pdf`** or **`Project2-teamname.pdf`**. *Team names must include the names or usernames of all members.*

Before submitting, "clean" your code (i.e., do a "**`make clean`**") removing the binaries (executables and .o files).

Use **`zip`** to archive your files. For example:

> **`mkdir lastname-proj2 cp * lastname-proj2`** /* copy all the files you want to submit */
> **`zip -r proj2-lastname.zip lastname-proj2`** /* package and compress */

Substitute **`teamname`** for **`lastname`** as appropriate. To submit your assignment (**`proj2-lastname.zip`**), log into the Instruct Assist website:

> https://ia.wpi.edu/cs4513/

Use your WPI username and password for access. Visit:

`Tools → File Submission`

Select "Project 2" from the dropdown and then "Browse" and select your assignment **`(proj2-lastname.zip`**).

Make sure to hit "Upload File" after selecting it!

If successful, you should see a line similar to:

```
Creator    Upload Time             File Name        Size     Status    Removal
Claypool 2016-01-30 21:40:07  proj2-claypool.zip   2508 KB  On Time   Delete
```

Grading

A grading guide shows the point breakdown for the individual project components. A more general rubric follows:

100-90. Both server and client function in the specified way. Each utility is robust in the presence of errors, whether from system or user. Code builds and runs cleanly without errors or warnings. Successful setup of server on Amazon EC2. Experiments effectively test all required measurements. Experimental writeup has the three required sections, with each clearly written and the results clearly depicted.

89-80. Both the server and client meet most of the specified requirements, but a few features may be missing. Programs are mostly robust in the face of most errors, failing only in a few cases. Code builds cleanly and runs mostly without errors or warnings. Mostly successful setup of server on Amazon EC2. Experiments mostly test all required measurements. Experimental writeup has the three required sections, with details on the methods used and informative results.

79-70. Both serer and client are in place, and the client can connect to the server, but core shell functionality is missing. Simple, one-word commands may work properly, but little more complicated. Code compiles but may exhibit warnings. Programs may fail ungracefully under some conditions. Setup of server on Amazon EC2 may be unsuccessful or incomplete. Experiments are incomplete and/or the writeup does not provide clarity on the methods or results.

69-60. Server and client do not function consistently in tandem. Code parts may be in place but do not work properly. Code compiles but may exhibit warnings. Programs may fail ungracefully under many conditions. Setup of server on Amazon EC2 unsuccessful and/or incomplete. Experiments are incomplete and the writeup does not provide clarity on the methods or results.

59-0. Server and client may connect but do little else properly. Code do handle parts of the client/server functionality may be there but does not function properly. Code may not even compile without fixes. No setup of server on Amazon EC2. Experiments are incomplete with a minimal writeup.

Send all questions to the TA mailing list (cs4513-staff at cs.wpi.edu).