CMPS109 Spring 2018 : Lab 8

In this lab you will implement a Berkeley Socket Datagram (UDP) based client-server C/C++ multi-threaded most significant digit radix sorter to satisfy functional requirements only.

This lab is worth 7% of your final grade.

Submissions are due NO LATER than 23:59, Sunday May 27, 2018.

Late submissions will not be graded.

Background information

In Lab 7 you created a Berkeley Stream Socket (TCP) client-server interface to your truly parallel MSD Radix sorter. Here you will replace that reliable stream connection with an unreliable datagram (UDP) connection.

To give you an insight into the complexity of TCP implementations, you will need to deal with cases where the UDP packets arrive out of order and/or go missing entirely.

Details of UDP socket programming can be found in the lecture handouts and in many on-line locations.

Setup

SSH in to any of the CMPS109 teaching servers using your CruzID Blue credentials:

```
$ ssh <cruzid>@<server>.soe.ucsc.edu
```
(use Putty http://www.putty.org/ if on Windows)

Authenticate with Kerberos and AFS: ***(do this EVERY time you log in)***

```
$ kinit
$ aklog
```

Create a suitable place to work: ***(only do this the FIRST time you log in)***

```
$ mkdir -p ~/CMPS109/Lab8
$ cd ~/CMPS109/Lab8
```

Install the lab environment: ***(only do this once)***

```
$ tar xvf /var/classes/CMPS109/Spring18/Lab8.tar.gz
```

Build the skeleton system:

```
$ make
```

Calculate your expected grade:

```
$ make grade
```

What to submit

In a command prompt:

```
$ cd ~/CMPS109/Lab8
$ make submit
```

This creates a gzipped tar archive named `CMPS109-Lab8.tar.gz` in your home directory.

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.

Requirements

Your completed network-enabled radix sorter must in all cases do one thing:

1. Correctly MSD radix sort moderately sized vectors of unsigned integers sent to it across the network

In addition, it must deal with the following situations:

Basic: Handle the trivial case were UDP packets do not go missing or arrive out of order.

Advanced: Deal with out-of-order and missing datagrams but assume the batch termination datagram is delivered successfully.

Stretch: Handle loss of the batch termination datagram.

Extreme: Deal with a combination of Advanced and Stretch.

All are functional requirements.

These requirements are NOT equally weighted - see Grading Scheme below.

What you need to do

The classes `RadixServer` and `RadixClient` are provided, but unimplemented. You need to implement them without changing the existing signature. You can add member variables and functions, but do not change existing signatures or override the default or given constructors.

It is recommended your implementation of `RadixServer` delegate you your Lab 6 or Lab 7 implementations of `ParallelRadixSorter`, but this is not required if you wish to create a totally new implementation.

Basic steps are as follows:

1. Implement client and server ends of the radix sorter
2. Have the server delegate to sorting implementations from previous labs
3. Ensure you stick to the on-wire protocol as described below

To execute your UDP client-server MSD radix sorter after running `make`:

```
$ ./radix 500 1 10
```

Where “1000” is the number of random unsigned integers that the test harness will generate, “1” is the number of times it will generate that many random unsigned integers, and “10” is the maximum number of CPU cores to use when sorting the single list.

`./radix` accepts three optional arguments that can be combined in any way:

`-d` Requests a sampled dump of the sorted vectors to demonstrate (hopefully) correct ordering.
`-r` Forces the internal client and server to drop some datagrams forcing you to request a resend.
`-l` Forces the internal client and server to drop the list termination datagram.

You should start by testing the truly trivial case where all random numbers fit in a single datagram:

```
$ ./radix 100 1 1
```

Then move on to multiple datagrams:

```
$ ./radix 300 1 1
```

Then deal with dropped datagrams:

```
$ ./radix 500 1 1 -r
```

Finally deal with dropped list termination datagrams:

```
$ ./radix 500 1 1 -l
```


As in previous labs, if there's something you don't understand, do this, in this order:

1. Google it
2. Post a discussion on Canvas
3. Come along to section and/or office hours and ask questions

Note that the test harness has embedded implementations of both client and server. Your implementations are tested against them, NOT against each other.

Client-Server Protocol

Your client and server should obey the following on-wire protocol:

1. Unsigned integers are exchanged in network byte order, in batches, over datagram sockets.
2. The list termination marker is a datagram with a special flag; see `radix.h` for details.

A simple implementation of this protocol assuming no transmission errors might be as follows:

- Client receives list of unsigned integers from test harness
- Client sends all unsigned integers in the list to server in batches with sequence numbers
- In the last batch, client indicates in is the last datagram in the list
- Server stores unsigned integers as they are received
- On receipt of the last datagram, server sorts the unsigned integers and returns them to the client in a similar manner.
- Client returns sorted unsigned integers to test harness.

However, the `-r` and `-l` flags force transmission errors to occur and your client and server implementations should handle them appropriately; see `radix.h` for details.

Grading scheme

The following aspects will be assessed by executing your code on a machine with an identical configuration to the CMPS109 teaching servers:

1. (70 Marks) **Does it work?**

    a. Basic (30 marks)
    b. Advanced (10 marks)
    c. Stretch (10 marks)
    d. Extreme (10 marks)
    e. Code free of warnings (10 marks)

In all cases, marks are deducted for any sort operations failing to produce the correct answer.

2. (-100%) **Did you give credit where credit is due?**

    a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%)

    b. Your submission is determined to be a copy of another CMPS109 student's submission (-100%)

    c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

        - < 50% copied code — No deduction
        - 50% to 75% copied code — (-50%)
        - > 75% — (-100%)

