Distributed sort

This project is an extension of the sort program you had implemented in project 1. In this second project, you are going to implement a multi-node sorting program with sockets and Go's *net* package. The main objective of this project is for you to get familiarized with basic socket level programming, handling servers and clients, and some popular concurrency control methods that golang provides natively.

Starter code

- The starter code invitation is located at https://classroom.github.com/a/gHnj2dVq

Updates

- Oct-1: Project released
- Oct-3: Added a note about socket mesh creation, suggesting that net.Dial() calls should be retried until they are successful

Distributed sort specification

Basic specification

Like the previous project, the project will read files consisting of zero or more records. A record is a 100 byte binary key-value pair, consisting of a 10-byte key and a 90-byte value. Your sort should be ascending, meaning that the output should have the record with the smallest key first, then the second-smallest, etc. The input file won't have any duplicate keys.

Distributed sort specification

In this project, we will have multiple servers instead of one server as in project 1. Each server has its own single input file. The distributed sorting program will concurrently run on all servers. The program should broadly have the following architecture/specs:

- read the input file,
- appropriately partition the records,
- send relevant records to peer servers,
- receive records from peers,
- sort the merged data (a fraction of records from its own input list augmented with a fraction of records from each of the other servers),
- write the sorted data to a single per-server output file

In the end, each server will produce a sorted output file with only records **belonging to** (based on the data partition algorithm below) that server. If one were to concatenate the output files from server 0 then server 1 etc then there would be a single big sorted file.

Basic assumption

1. **Number of servers**: We make the simplifying assumption that the number of servers running the distributed sorting program is a power of 2 (so 2, 4, 8 servers, etc). The set of servers is defined in a configuration file specified on the command line. You can assume that there are at most 16 servers.
2. **Data partition algorithm**: Given a record with a 10 byte key and assuming 2^N servers, we would use the most significant N bits to map this record to the appropriate server. For example, in a system with 4 servers, if we encounter a record with a key starting with 1101… , it would **belong to** serverId: 3.

Network protocol

Each of your netsort instances will send (and receive) 100-byte records over the network socket. You will implement a very simple binary protocol as follows. You will send a one-byte boolean value called stream_complete (0 = false, 1 = true) indicating whether all of the data is sent, followed by the 10-byte key, then the 90-byte value. If stream_complete is false, then the contents of the 100 bytes after it represent a valid record. If stream_complete if true, then the contents of the 100 bytes after it are not defined (you can set them to all 0s if you want) and no more records will be sent over the TCP socket.

0	1	2	3	4	5	6	7	8	9	10	11	12	...	99	100
stream_complete (1 byte)	Key (10 bytes)										Value (90 bytes)				

Project objective

You are to write a distributed sorting program that reads in an input file and produces a sorted output file with only records belonging to that server. Use src/netsort.go as a starting point. Your program should support the following interface:

Usage:

```
$ netsort <serverId> <inputFilePath> <outputFilePath> <configFilePath>
```

- **serverId**: integer-valued id starting from 0. For example, if there are 4 servers, they would have ids 0, 1, 2, 3.
- **inputFilePath**: input file
- **outputFilePath**: output file
- **configFilePath**: path to the config file

While we could run each server on its own physical machine (or virtual machine), it is easier to simply run multiple server processes, each with its own TCP port and separate directory to store input/output files. This is how the small demo provided with the starter code is structured.

Input and output

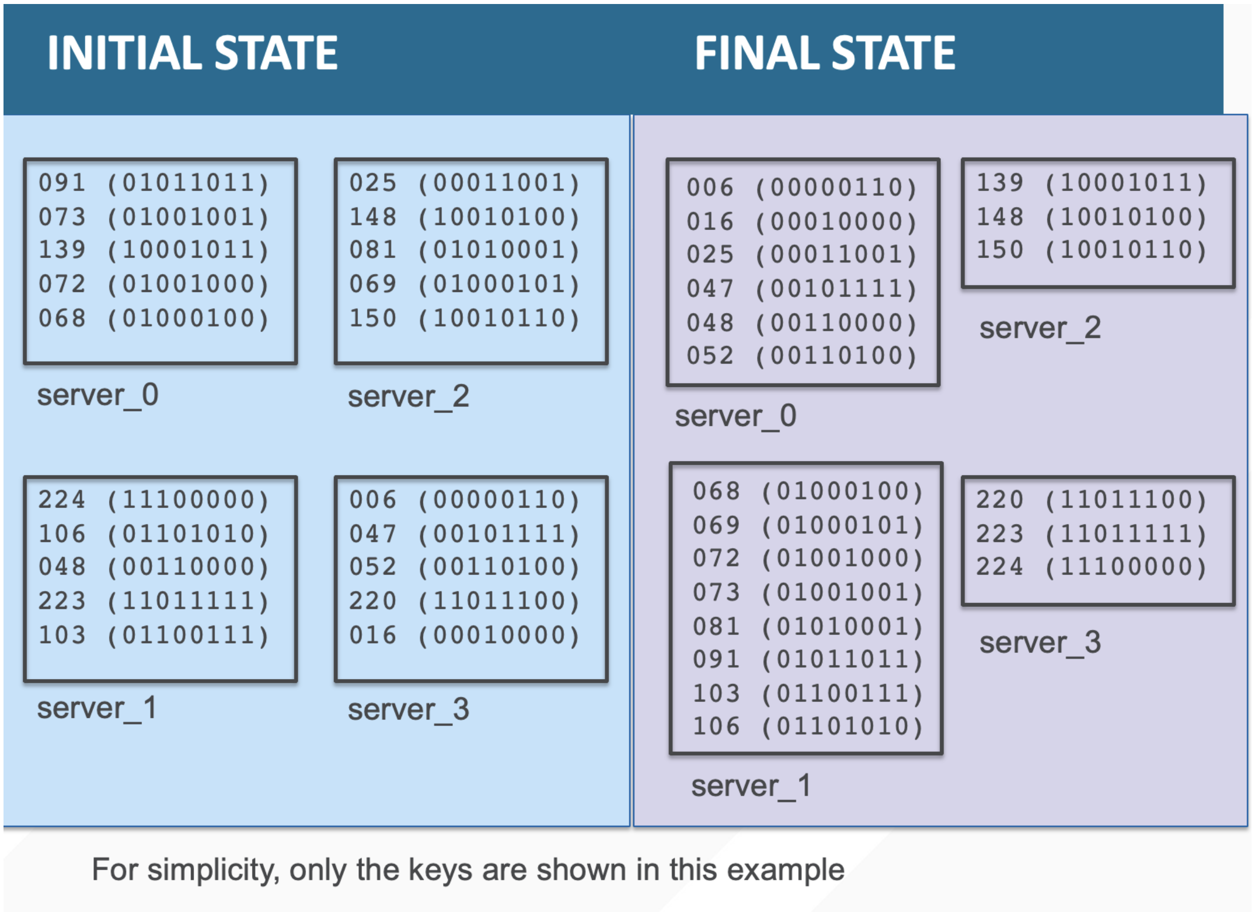
As an example of what behavior we expect from your program, assume that the number of servers is N=4 (in reality it could be any power of 2 up to 16). Each server starts with its own input file, so server 0 has input-0.dat, server 1 has input-1.dat, etc up to input-3.dat. After your sort is finished, then server 0 should have output-0.dat, server 1 should have output-1.dat, etc up to output-3.dat.

output-0.dat should contain a sorted set of records that all begin with bits 00. output-1.dat should contain a sorted set of records that all begin with bits 01, output-2 should have bits 10, and output-3 should have bits 11.

Thus if you concatenated output-0.dat, output-1.dat, output-2.dat, then output-3.dat together into a single big file output.dat, then output.dat would be a sorted set of key-value pairs and should contain the data that was in all the input files.

Example

Consider four servers shown on the left, below. For simplicity, only the keys are shown. At the end of a successful distributed sort operation, the keys will be stored on the servers as shown to the right. Note that due to randomness, each server may have a differing number of records in its partition of the sorted output, as shown in the example.



INITIAL STATE

```
091 (01011011)
073 (01001001)
139 (10001011)
072 (01001000)
068 (01000100)
```

server_0

```
025 (00011001)
148 (10010100)
081 (01010001)
069 (01000101)
150 (10010110)
```

server_2

```
224 (11100000)
106 (01101010)
048 (00110000)
223 (11011111)
103 (01100111)
```

server_1

```
006 (00000110)
047 (00101111)
052 (00110100)
220 (11011100)
016 (00010000)
```

server_3

FINAL STATE

```
006 (00000110)
016 (00010000)
025 (00011001)
047 (00101111)
048 (00110000)
052 (00110100)
```

server_0

```
139 (10001011)
148 (10010100)
150 (10010110)
```

server_2

```
068 (01000100)
069 (01000101)
072 (01001000)
073 (01001001)
081 (01010001)
091 (01011011)
103 (01100111)
106 (01101010)
```

server_1

```
220 (11011100)
223 (11011111)
224 (11100000)
```

server_3

For simplicity, only the keys are shown in this example

Program structure

You are free to any structure your code any way that makes the most sense for you, however one possible structure is as follows:

1. The starter code includes a `main()` method that reads in the contents of the configuration file for you including the IP address and port for each server, plus the input file location and output file location for each server
2. A command-line option to each instance of netsort tells that instance which of the various servers it is
3. You can start by building a “mesh” of TCP socket connections. Each client opens a socket to every other server
4. Once every instance has connected to every other instance, and received incoming connections from every other instance, you can begin reading in your input data and partitioning it across the different nodes and sending it to remote nodes as appropriate
5. Using goroutines, you should be sending data to remote nodes in parallel with receiving data from remote nodes (if you serialize the sending of data before you read any data, or vice-versa, you may encounter deadlock. For more details, refer to the class lecture on sockets internals and deadlock)
6. Once you know you’ve received all the data, you can begin to sort the data (in memory is fine) and write it out to your output file. You can re-use any of the code from your first project if you’d like.

Building your socket mesh

When your program starts up, each process will need to establish (N-1) sockets to the other (N-1) nodes. Because each process will start up at slightly different times, you will need to make sure to keep retrying your net.Dial() calls until all the sockets are set up. One way of doing that is to include your calls in a loop, and to retry the net.Dial() call after a short delay (e.g. 250 milliseconds).

Building and running

```
$ go build -o netsort netsort.go
```

There is a testcases directory with a simple input test case for a small example running on four servers. You can examine and run the run-demo.sh script to see this in action.

Verifying your sort implementation

There are a few ways to check that your program is correct. Here are two possibilities, though of course you might think of others.

<u>Setup</u>

Concatenate all the input files into a file called INPUT

```
$ cp input-0.dat INPUT
$ cat input-1.dat >> INPUT
$ cat input-2.dat >> INPUT
$ cat input-3.dat >> INPUT
```

Concatenate all the output files into a file called OUTPUT

```
$ cp output-0.dat OUTPUT
$ cat output-1.dat >> OUTPUT
$ cat output-2.dat >> OUTPUT
$ cat output-3.dat >> OUTPUT
```

<u>Method 1: Using Project 1 sort</u>

You can use your project 1 sort code to verify if your output is correct

```
$ bin/sort INPUT REF_OUTPUT
$ diff REF_OUTPUT OUTPUT
```

The output of your P1 sort code should equal the output of your P2 netsort

Method 2: Using showsort

```
$ bin/showsort INPUT | sort > REF_OUTPUT
$ bin/showsort OUTPUT > my_output
$ diff REF_OUTPUT my_output
```

In both cases, you are simply comparing a sort of a concatenation of the input files with a concatenation of the output files.

Autograding

You should check that your code works on a single node, on two nodes, on four nodes, on eight nodes, and on 16 nodes. You should try with one record per server, multiple records per server, an uneven number of records per server, and a few larger runs as well (e.g. a few megabytes of input per server). Does your code work if a server doesn't have any records sent to it?

Expected effort level

To set expectations for how much work is involved, note that our implementation of netsort.go added 250 lines of code to the starter code.

Good luck!

###