NWEN 303 Project 2

Daniel Braithwaite

October 14, 2016

1 Control Flow

The control flow between the KeyManager and Client is detailed below.

1. Client opens connection to KeyManager and sends session type.
2. Session type is REQUEST
   (a) Client then sends the chunk size
   (b) KeyManager then sends information in the following order. cipher text, plain text, starting key (as int), key size.
3. Session type is SUBMIT
   (a) Client sends whether the key was found. If the key was found the Client will also send the correct key (as int).
4. Connection is closed by key manager and client.
5. If latest task found key then terminate.

1.1 System Requirements

1. **Clients only need to be aware of the location of the key manager.** The client is only aware of the minimum amount of information, as described in the control flow sequence the client is told the key to start at, what the plain text is and what the cipher text is.
2. **Clients can join or leave but will complete the work they have been assigned.** Once a client has requested a job, the job must be completed as the manager doesn't keep track of assigned work, however when a client has submitted its job there is no requirement to collect another task.
3. **Clients request work from the key manager.** The only part of this system that gives out tasks is the key manager so clients must request work from there.
4. **Connections between clients and the master only exists long enough to request work or to return results.** When a job is requested and all information has been transferred the socket is closed on both ends (by key manager and client). Same for when a job is submitted.

5. **When the key is found, the key manager will shutdown** When a solution is found the event loop is terminated, thus terminating the program.

2 Testing

2.1 KeyManager Requirements

1. **Key manager can be invoked from the command line.** The key manager can be invoked with the following command $java\ KeyManager < startkey > < keysize > < cyphertext > < plaintext >$

2. **Command line arguments are correctly parsed.** By debugging the program we can inspect the variables and see that they have been correctly received and parsed.

3. **Should request random port number and print it out.** When the key manager is created it makes a new Socket Server as follows $= new\ SocketServer(0)$. As specified by the java doc, when given a parameter of 0 the port is automatically assigned to a free port. The key manager then prints the string **Waiting for connections on [port]**.

4. **When key is found message should be printed along with the correct key.** By running the command detailed in number 2. And then connecting a client we find the key immediately (as the starting key is correct), then the serve prints the following

   Key Found!
   3185209680

   Thus this requirement is stasfyed.

5. **If no correct key is found then message should be printed.** After sending out a job the key manager checks to see if the current key exceeds the maximum possible key, if it is then we will wait to receive all active jobs before terminating. If none of the jobs have found a correct solution then we print out a failure message.

6. **Manager should keep track of how long it takes to find the key or come to the end of the search space.** If we run the same

example that has been run before, we get the following output right before termination.

Key Manager Exiting
Total search time: 0s

The time is 0s because we only start the clock once the first connection is made. This makes sense as we don't want to skew the timing results by how long it takes to start the clients.

So all the system requirements are stasfyed.

2.2 Client Requirements

1. **Client can be invoked from the command line.** The client can be invoked with the command $java\ Client\ <hostname><port><chunksize>$.

2. **Command line arguments are correctly parsed.** By debugging we can inspect the variables of the program and see that the have been correctly parsed.

3. **Client keeps requesting tasks until no more tasks are available or user shuts down the client.** If the client attempts to connect to a KeyManager but the connection fails (i.e. there are no tasks left) then the client will shut down.

4. **Client searches all the assigned key space.** The client is sent the start key and knows the chunk size. The client iterates from 0 to $chunk\ size\ -\ 1$ and for each iteration we test the current key followed by incrementing the key. So we are testing the range of keys $(startKey, startKey + chunkSize\ -\ 1$. Which is all of the assigned key space.

5. **Client does not search outside the key space.** Before trying a new key, like with the key manager we ensure that the current key is not greater than the maximum possible key (which we know to be $2^{keySize*8}$).

3 Performance Evaluation

3.1 Experiment Process

To run each of the experiments we use a Java program to create a KeyManager and then spawn off N independent client programs. Then we run this program for 30 replications. We also design the problem its trying to solve to be simple so we can run multiple tests quickly. To get accurate results we run this program on lighthouse, the number of cores will allow us to better simulate the N clients. To start we will fix the number of clients to 10 and experiment with adjusting the chunk size. The chunk sizes we will use are 500, 1000, 5000, 10000, 25000. Following this we will experiment with different numbers of cores to see how this effects the time taken and if the trends in the time are the same between different numbers of clients.

To check if the changes are statistically significant we construct a 95% confidence interval for the difference between the means of the two populations. Then if 0 is in this interval we can conclude that the changes are **not** significant. And if 0 is not in the range then the changes are significant. We use the following formula to calculate our confidence interval

$$\bar{X_1} - \bar{X_2} \pm 1.96 * \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

3.2 Results

The table below is the results from running tests with 10 nodes and varying chunk size.

chunk size	500	1000	5000	10000	25000
mean time	18	17	16	15	19
std div	0	2.236	1.412	5.196	2.828

Table 1: Results of changing varying chunk size with 10 clients

Now we check to see if these changes are significant. When comparing the populations with chunk size 500 and 1000 from the table above we get a 95% confidence interval $(1.876, 0.124)$, indicating that the change **is statistically significant**. We want to see of the rest of the results are statistically

significant. The following shows the comparison of the populations with 10 nodes.

Pop 1	Pop 2	95% CI	Result
500	1000	$(1.876, 0.124)$	**Significant**
1000	5000	$(1.943, 0.057)$	**Significant**
5000	10000	$(5.744, -3.745)$	**Not Significant**
5000	25000	$(0.935, -8.935)$	**Not Significant**

Table 2: Significance tests for populations for 10 clients

From these results it seems that a chunk size between 5000 and 25000 give the best results as the three populations 5000, 10000 and 25000 are statistically the same. Now that we have seen the chunk size can have an effect on the total time take we want to see if this choice of chunk size is dependent on the number of clients.

We run the same series of tests with 5, 15 and 20 nodes. Which give the following result tables.

chunk size	500	1000	5000	10000	25000
mean time	28	27	26	24	26
std div	2	1	0	0	0

Table 3: Results of changing varying chunk size with 5 clients

chunk size	500	1000	5000	10000	25000
mean time	15	14	14	14	12
std div	1.412	4	2.645	0	3.316

Table 4: Results of changing varying chunk size with 15 clients

chunk size	500	1000	5000	10000	25000
mean time	16	15	15	13	15
std div	0	3.606	2.236	3.317	5.656

Table 5: Results of changing varying chunk size with 20 clients

Which we then want to perform statistical significance tests, the same way we tested the results for 10 clients.

Pop 1	Pop 2	95% CI	Result
500	1000	$(1.722, 0.277)$	**Significant**
1000	5000	$(1.175, 0.824)$	**Significant**
5000	10000	$(2, 2)$	**Significant**
10000	25000	$(-2, -2)$	**Significant**

Table 6: Significance tests for populations for 5 clients

Pop 1	Pop 2	95% CI	Result
500	1000	$(3.826, -1.826)$	**Not Significant**
500	5000	$(2.275, -0.275)$	**Not Significant**
500	10000	$(1.349, 0.651)$	**Significant**
10000	25000	$(5.222, -1.222)$	**Not Significant**

Table 7: Significance tests for populations for 15 clients

Pop 1	Pop 2	95% CI	Result
500	1000	$(3.279, -1.279)$	**Not Significant**
500	5000	$(1.876, 0.124)$	**Significant**
5000	10000	$(4.118, -0.118)$	**Not Significant**
5000	25000	$(5.675, -5.675)$	**Not Significant**

Table 8: Significance tests for populations for 20 clients

We also want perform significance tests between the number of clients to see if increasing the number of clients has a significant effect. With the various numbers of clients we see that using the chunk size 10000 is always in the optimum range so we will fix this chunk size to compare between the populations.

N_1	N_2	95% CI	Result
5	10	$(13.732, 4.267)$	**Significant**
10	15	$(5.732, -3.732)$	**Not Significant**
15	20	$(2.928, -0.928)$	**Not Significant**

3.3 Interpretation

From these results we see a few observations

1. Apart from with 5 nodes changing the client chunk size doesn't tend to make much of a significant change (even when the change is statistically significant the difference in mean is around a second).

2. Once we get to 10 clients there is no longer any significant changes with regards to the time taken.

Consider our assumption of the problem, we are assuming that the amount of work to be done in parallel is fixed. Because of this we can apply Amdahl's Law. This law is consistent with item 2 above.

4 Realistic System

The idea for solving this problem is to assign each job a job id. When a job is requested we assign a job id and the current time when the job was handed out. We have some fixed interval which after the interval has passed we hand out the job again.

The simplest way to re hand out jobs is to check all our currently active jobs and ensure that the time interval is not up. We do this every time a job is requested.

We store the reclaimed jobs in a list where each entry in the list is (job id, job start key, chunk size), and any job request will receive something from this list if its non empty. To account for the fact that different clients can have different chunk sizes we will say that if a client requests a chunk of size x then they will get a chunk of size at most x but never more. So therefor if a client requests a chunk of 50 and we have a reclaimed chunk of size 30 the request will only receive 30 keys to check. On the other hand if there is a request for 30 and we have a reclaimed chunk of 50, then the request for 30 will be fulfilled and the reclaimed list updated to show there are only 20 reclaimed remaining.

With this implemented if a client requests a job and doesn't submit a result the job will be sent out again and still get processed. If it turns out that the initial client was running very slowly and submits the result anyway then this will be fine. We have two cases ether the duplicated task contained the

key we are looking for. In this case the first submission we receive will tell us the key is found and the server will shut down, therefor we wont receive the second submission. On the other hand if the job doesn't receive then we will receive multiple negative result which doesn't effect the outcome of the system.

4.1 Control Flow

Below is the modified control flow outline, the modifications are in bold.

1. Client opens connection to KeyManager and sends session type.
2. Session type is REQUEST
   (a) Client then sends the chunk size
   (b) KeyManager then sends information in the following order. **Job id**, cipher text, plain text, starting key (as int), key size, **chunk size**.
3. Session type is SUBMIT
   (a) Client sends the **Job id**, followed by whether the key was found. If the key was found the Client will also send the correct key (as int).
4. Connection is closed by key manager and client.
5. If latest task found key then terminate.

4.2 Testing

The only requirement to change is system requirement 2. It changed from *Clients can join or leave but will complete the work they have been assigned* to **Clients may not return work, and these jobs should be re sent after a period of time**.

This requirement is simple to check, we take a small example problem. First we connect a client that will take a chunk which includes the correct key. Before this client can find the key and report back we stop it. Now if the key manager isn't reclaiming jobs if we start another client we will never find the key as the section with the key has been given out. But when we do start another client (after a set period to allow the reclaiming to happen) we get the same chunk and are able to find the key. So we can see this requirement

is satisfied.

All the other requirements are still satisfied by the same reasons as above.