Futures

(Also look at powerpoint slides)

The problem

Look at the following piece of code:

```
byte data1[]=readData(server1,"image1.png");   // line 1

encryptData(data1);                            // line 2

writeData(server1,"image1.png",data2);         // line 3

byte data2[]=readData(server2,"image2.png");   // line 4

encryptData(data2);                            // line 5

writeData(server2,"image2.png",data2);         // line 6
```

Assuming that readData and writeData have a range of run times from 20ms to 2000ms (depending on the size of the file and the how busy the network or server 1 and server 2 are).

If encryptData takes between 400ms to 600ms depending on the length of the data.

We can see that this code can have worst and best run times as follows:

Worst (longest delays) = 2000ms + 600ms + 2000ms + 2000ms + 600ms + 2000ms = 9,200ms = 9.2s

Best (shortest delays) = 20ms + 400ms + 20ms + 20ms + 400ms + 20ms = 9,200ms = 880ms = 0.88s

We can also see from this code, that lines 4 and lines 5 and line 6 do not depend on lines 1 and lines 2 and lines 3.

If line 1 runs very slowly (because server 1 is busy, this will delay lines 4, 5 and 6 even if server 2 is not busy).

Use of futures

The use of futures allows parts of the code to run concurrently in a structure manner. A future is a entity which will return its value sometime in the future. The future itself wraps up the execution of the code in a thread, this is so multiple lines of the program can be executed at the same time. This cuts out the dead time where the execution of the program is held up while waiting for an external resource (for example waiting for a server to respond).

So in the above example we can wrap each of the lines of execution, 1 to 6 in futures allowing them to be executed in turn. Here is an example of a Java base class that can be used to build future entities.

```
abstract class Future <T> extends Thread     {
public Future() {
start();
}

private T futureValue;

public     T resolve() {
try {
            this.join();
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
return(futureValue);
}

protected abstract T calculate();

public void run() {
futureValue=calculate();
}
}
```

We use this class as a base class to construct futures for the operations on lines 1, 2 and 3.

Here is the future class for readData

```
FutureReadData extends Future <byte[]> {

	private String serverURL;

	private String filename;



	// Constructor

	public FutureReadData(String serverURL, String filename) {

		this.serverURL=serverURL;

		this.fileName=filename;

	}



	@Override
	protected byte[] calculate() {
		return(readData(serverURL,serverName);
	}
}
```

Here is the future class for encrypt

```
FutureEncrypt extends Future <byte[]> {

	private FutureReadData reader;

	// Constructor

		public FutureEncrypt(FutureReadData reader {
			this.reader=reader;
	}

	@Override
	protected byte[] calculate() {
		return(encrypt(reader.resolve());
	}
}
```

Here is the future class for writeData, notice it takes it input from the encryptData future.

```
FutureWriteData extends Future <byte[]> {
        private String serverURL;
        private String filename;
        private FutureEncrypt future;
        // Constructor
        public FutureWriteData(String serverURL, String filename, FutureEncrypt future) {
                this.serverURL=serverURL;
                this.fileName=filename;
                this.future=future;
        }

        @Override
        protected byte[] calculate() {
                return(writeData(serverURL,serverName);
        }
}
// We can now modify the original target code using futures
FutureReadData    reader1=new FutureReadData(server1,"image1.png");          // line 1
FutureEncrypt encryptor1=new FutureEncrypto(reader1);                        // line 2
FutureWriteData write1=new FutureWriteData(server1,"image1.png",encryptor1);        // line 3
FutureReadData    reader2=new FutureReadData(server2,"image2.png");          // line 4
FutureEncrypt encryptor2=new FutureEncrypto(reader2);                        // line 5
FutureWriteData write2=new FutureWriteData(server2,"image2.png",encryptor2);        // line 6
write1.resolve();write2.resolve();                    // complete execution
```

Assuming the encrypt function is CPU bound but all other operations take negligible CPU time we can calculate worst case run time for new code.

write1.resolve1(); // will complete by 2000ms + 600ms + 600ms+ 2000ms = 5.2 seconds

write2.resolve2() // will complete by 2000ms + 600ms + 600ms + 2000ms = 5.2 seconds

Notice for each concurrent task you have to assume the encryption from the other thread will

slow this thread down, (this is because each encryption take 600ms of CPU cycles).

So the worst case is 5.2 seconds, much faster than before.

Java Futures Library

See example source code.

Future <T> is a generic class which returns a type of T sometime in the future.

The future itself is generated by an scheduler of type ExecutorService, this is responsible to launching the threads for the futures. The example code generators a new executor like this.

private ExecutorService executor = Executors.newFixedThreadPool(10);

Because it is using a ThreadPool more than 1 future can execute at one time, so the execution is concurrent. In this case a maximum of 10 threads are available at one time.

Once we have the service we can use this to generate the instances of the Futures.

This is done by using the submit method of the executor as follows:

```
public Future <String> calculate(String urlString) {
        return executor.submit(() -> {
                return(“Hello World”);     // example code is more interesting!
        });
}
```

The notation () -> denotes the use of an anonymous function the () is the parameter list to the function, and the -> refers to the anonymous function, in this case wrapped in braces.

This notation is called a lambda expression.
To make an instance of the future, just call calculate:
Future <String> future=factoryMaker.calculate("http://gmail.com/");

You can test if the Future has finished execution using the isDone call, future.isDone();

If isDone returns true you can get the actual return value called get, System.out.println(“Value is “+future1.get()); See the source code for complete working code and adapt as needed.