Jared Barneck

USU CS 6210

Homework 1

# Simulated Disease Surveillance

Messaging frameworks and Vector Timestamps

## Primary Goals

The goal of this project is to

1. Use ZeroMQ to simulate a distributed system.
2. Observe Vector Timestamps in a distributed system.

## Project Repository

The project is located on GitHub here: <https://github.com/rhyous/MastersOfComputerScience/tree/master/Rhyous.CS6210.Hw1>

Included with the source is a Docs folder that includes this document as well as the Visio Diagram.

### Language

The language used is C#.

## Architecture

With any distributed system, architecture is critical to both the success of creating a project and the success of maintaining a project.

Note: A more complete architecture can be found in the Visio diagram here: <https://github.com/rhyous/MastersOfComputerScience/blob/master/Rhyous.CS6210.Hw1/Docs/Visio%20Diagram.vsdx>

The target architecture can be seen in Figure 1 – Visio Diagram. While this is the target, not all of it will be complete by homework 1. In Visio Diagram, there is another screen shot representing what of the below diagram is complete.

### Larger boxes vs small boxes in the diagram

In the diagram, there are larger boxes, which represent larger systems. The smaller boxes represent servers or clients. If the smaller boxes share a larger box, then those clients and servers are on the same system.

### Lines and Arrows

The lines connect clients and servers. Usually the arrow indicates the direction that traffic flows, however, when it comes to request and reply, the line has arrows on both sides, but doesn’t indicate which direction is first. However, the small box indicates via color whether it is a server or client and the ZSocketType is in the boxe’s text.

Figure 1 – Visio Diagram



## Architecture of the Code

The C# port of ZeroMQ fails to expose or implement any interfaces. This makes architecture more difficult. All systems were forced to be tightly coupled to ZeroMQ. To solve this, I started wrapping ZeroMQ into interfaces, wrappers, and adapters to allow for dependency injection and mocking of objects. This cut significantly into my homework time. Had I not had an inclination to use this in a production system, I would have abandoned the effort.

The interfaces are designed taking into account the I in SOLID.

I = The **interface-segregation principle** (**ISP**) states that no client should be forced to depend on methods it does not use.[[1]](https://en.wikipedia.org/wiki/Interface_segregation_principle#cite_note-ASD-1)

There are Interfaces such as the following IConnect interface that have only one or two items.

public interface IConnect

{

void Connect(string endpoint);

bool IsConnected { get; }

}

This does not prevent meta-interfaces from being created, such as ISusbscribe

public interface ISubscribeSocket : ISubscribe, IDisposable, IConnect

{

}

This interface doesn’t implement any of its own contracts but instead is a just a combination of existing interfaces.

### Arguments or Configuration File

I have a minorly popular arguments library, Rhyous.SimpleArgs, which is available as a NuGet package. The code is open source and available here:

GitHub: <https://github.com/rhyous/SimpleArgs>

Before this assignment, it didn’t allow for providing the command line parameters as a file, so I updated it. SimpleArgs uses a name=value syntax for its command line parameters. So

The configuration file syntax is to just have the parameters defined in the same syntax, with each parameter on a separate line.

Name="Disease Analyzer: Measels"  
AnalyzerEndpoint=tcp://127.0.0.1:5555  
OutbreakNotifierEndpoint=tcp://127.0.0.1:5555

### Unit Tests

I started writing Unit Test following the industry standard rule that they cannot alter the build system, including opening ports on the build system. This required mocking the Interfaces used with a mocking library. I use Moq.

## Log Server

It made sense to put all the logs in one place. So while it wasn’t a requirement of the homework, I created an additional PUSH and PULL system that is a log server.

Internally, the LogServer uses Log4Net to log to a file.

All systems log to the LogServer. Here is an example snippet:

2018-02-07 18:55:24,084 (DEBUG) LogServer: LogServer has started.  
2018-02-07 18:56:20,882 (DEBUG) Influenza:[0,0,1]: Starting Influenza on tcp://127.0.0.1:5555.

### Vector Timestamp

The vector timestamp was first implement as a simple object, with each system being a fixed property of the object.

public class VectorTimeStamp : IVectorTimeStamp

{

public int Simulator { get; set; }

public int HealthDistrict { get; set; }

public int Analyzer { get; set; }

}

I implemented a ToString() method that returned the following format: [1,0,0].

However, while this worked, it quickly showed its limitations. The assignment asks that we create three Analyzers. The above code doesn’t allow for extensibility or scaling. In short, it was a failure. Good news is, this is a homework assignment and not a production system, and it can be improved for homework 2.

### Vector Timestamp Planned Improvements

For homework 2, the plan is to change the VectorTimeStamp to be a system identifier paired with the log response. Then each system should have a short name or identifier. Also, the VectorTimeStamp may not know about a system until it is used. So in Homework 2, the first VectorTimeStamp coming from a client, C1, will only have the following:

[C1:1]

The second system, a server, S1, would then append to it.

[C1:1,S1:1]

This would allow for better scaling and extensibility. I’ll implement it with a custom IDictionary<string,int> that writes that format using ToString().

## Running the System

To test the full system, PowerShell was used to launch the systems. There is a Run\Run.ps1 file in the project.

Run.ps1 launches three Outbreak Analyzers, multiple Health Districts and one or two Simulators per Health District.