ECEC 356: Final Project Write-Up

Multi-Device System

By Darius Remeika

Abstract: The goal of this project was to build a system which allows interconnection of different sensors. The approach that was taken allows the connection of multiple clients through a central server which manages the states of the clients and maintains the application that is run and also to share the data from the sensors. The use of JRE allowed portability and a short development time of the application. Separate Android and Leap Motion client applications were written to demonstrate the flexibility and concurrency achieved by the system. The final demonstration consists of a web player and mouse control which communicates the data bi-directionally with the clients using the help of the server while also allowing the sharing of the same sensor in multiple applications.

Introduction:

The main goal of this final project was to demonstrate the embedded system concepts learned throughout the course. Basic requirements of the project included a sensor which would communicate with some system and affect its state. During the course, the labs demonstrated a good example of the concepts of concurrency which were not trivial to achieve/understand. Microsoft DSS and CCR mechanisms were used to achieve the safe concurrency practices which allowed sensors to communicate through services. While this approach demonstrated the use of concurrency principles - it did not show how exactly such concurrency was achieved. The DDS presented a steep learning curve and a much longer development time – even when a basic application needed to be built. Besides that, the erroneous state of the system made development unpredictable and frustrating. Also, portability became a concern, when operating systems that did not run Windows DSS needed to be used efficiently. Rather than extending the application from lab materials, I've decided to create a portable, efficient system which would interconnect a wide variety of applications and sensors using the concurrency concepts learned during the course. The following paper will cover the process in which the construction of the system was possible as well as the development process of the application which is used to control web player and mouse concurrently with the use of multiple shared sensors and clients.

Design Process:

To build such system, an overview layout of the system was made (Figure 1). The system consists of the main application which should receive and send data from/to clients (which can be on different hosts) as well as the application which the server would be controlling.

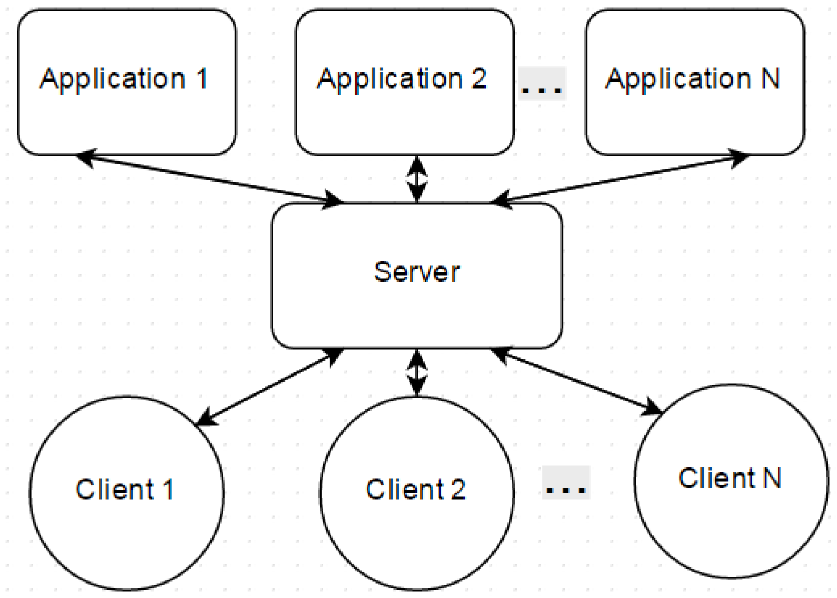


Figure 1 Diagram showing basic layout of the system.

A diagram shows a main server which is a separate entity from the main application – this is made so that the server can be easily expandable to control multiple applications. Such requirement creates a need for the server to know which application the server is sending data to and there is also a need for the server to know where the data is going. There also arises a need for the clients to properly direct the data to the specific application or let the server decide where the data needs to go. The few key aspects that need to be achieved are that the server should be able to handle a huge variety and number of clients, the data that is received and sent from clients and applications should not interfere, the whole system should be portable and easily reusable for any application which needs to use or share sensors.

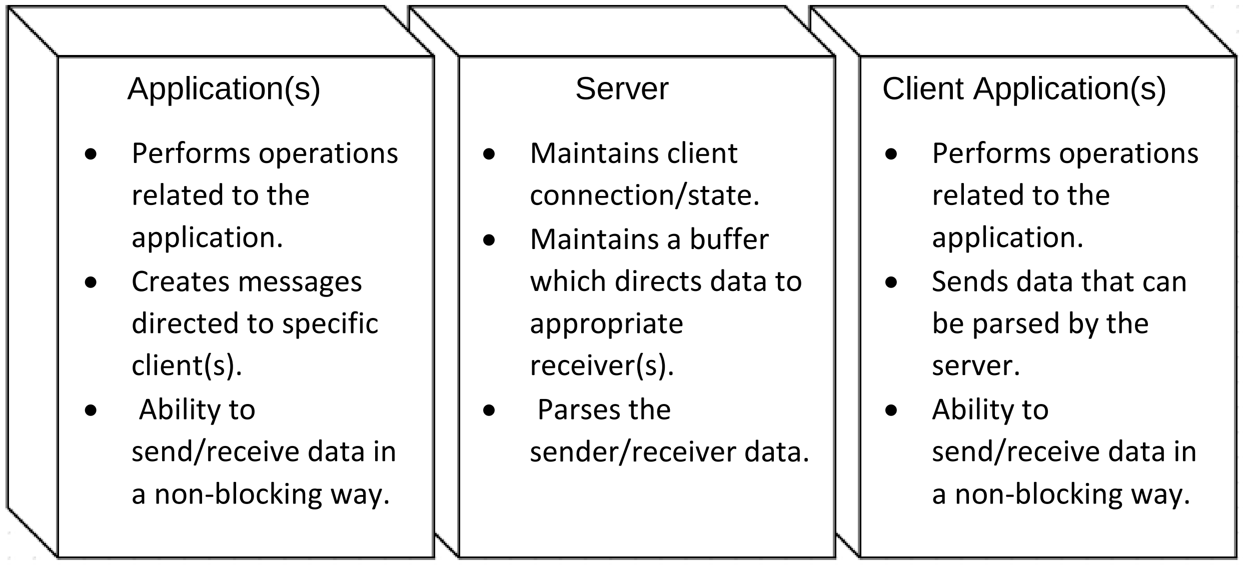
In order create such system in a short period of time, a few keys decisions needed to be made right away. The mechanism of how the data is communicated needed to be resolved. The mechanism that was shown during the course was the WebSocket mechanism which used a browser to communicate data via HTTP like protocol. While this method of communication method seemed tempting, the regular socket communication provided a much higher throughput and lower latency even when multiple clients were connected (Kemmerling 1). The simple socket communication was therefore chosen. In a case where information needs to be made available to public, a server can easily post the data received from the clients to a webpage.

To make the server portable the Microsoft's DSS and CCR mechanism couldn't be used, since it would not run on non-Windows platforms. JRE (Java Runtime Environment) provides many concurrency features as well and it's also portable, the server can be run on any machine which has Java runtime environment (Sander). Therefore JRE seemed like a good candidate to build the system in a short period of time.

In order to communicate between the server and applications a mechanism which has small overhead, is quick and is orderly, and has a bounded buffer was needed. Since the choice of JRE was already made, a JRE's concurrent libraries provided a mechanism which satisfies these requirements. A BlockingQueue provides a way to create a bounded buffer which is thread-safe (Paul). Therefore such buffer can be used to pass the data between the server and application and perhaps to store the data received from clients. Such mechanism is needed to store and distribute data while it's being parsed. Since this mechanism implements a nice way to ensure that the buffer is bounded, there is little concern with the clients sending too much data for the server to handle.

Building the System (Results):

The building process of the application was split into 3 major parts – server, application and client projects. Due to the time limitation, the priority was placed on the server project (although all of the parts were finished) which is visually least observed – but in reality it's the backbone of the system. The major tasks of each part are listed as seen in Figure 2.



Application(s)

- Performs operations related to the application.
- Creates messages directed to specific client(s).
- Ability to send/receive data in a non-blocking way.

Server

- Maintains client connection/state.
- Maintains a buffer which directs data to appropriate receiver(s).
- Parses the sender/receiver data.

Client Application(s)

- Performs operations related to the application.
- Sends data that can be parsed by the server.
- Ability to send/receive data in a non-blocking way.

Figure 2 Figure that lists important tasks that need to be performed in each project.

<u>Building Server:</u>

The server needs to have an ability simultaneously perform multiple tasks, such as handling new clients, taking care of disconnected clients, updating the states, etc., but most importantly parsing and directing the data received from each client. In order to allow the server to do these tasks concurrently, the use of threads is necessary. The layout of server part of the application can be seen in Figure 3. The server has a single client listener thread which spawns and assigns new client thread object to newly connected clients. The server also maintains the unique ID and type of the client; therefore it can be used to reach specific client or a set of clients through the sender queue. Each client thread contains a sender and receiver queues to which outgoing and incoming messages are placed.

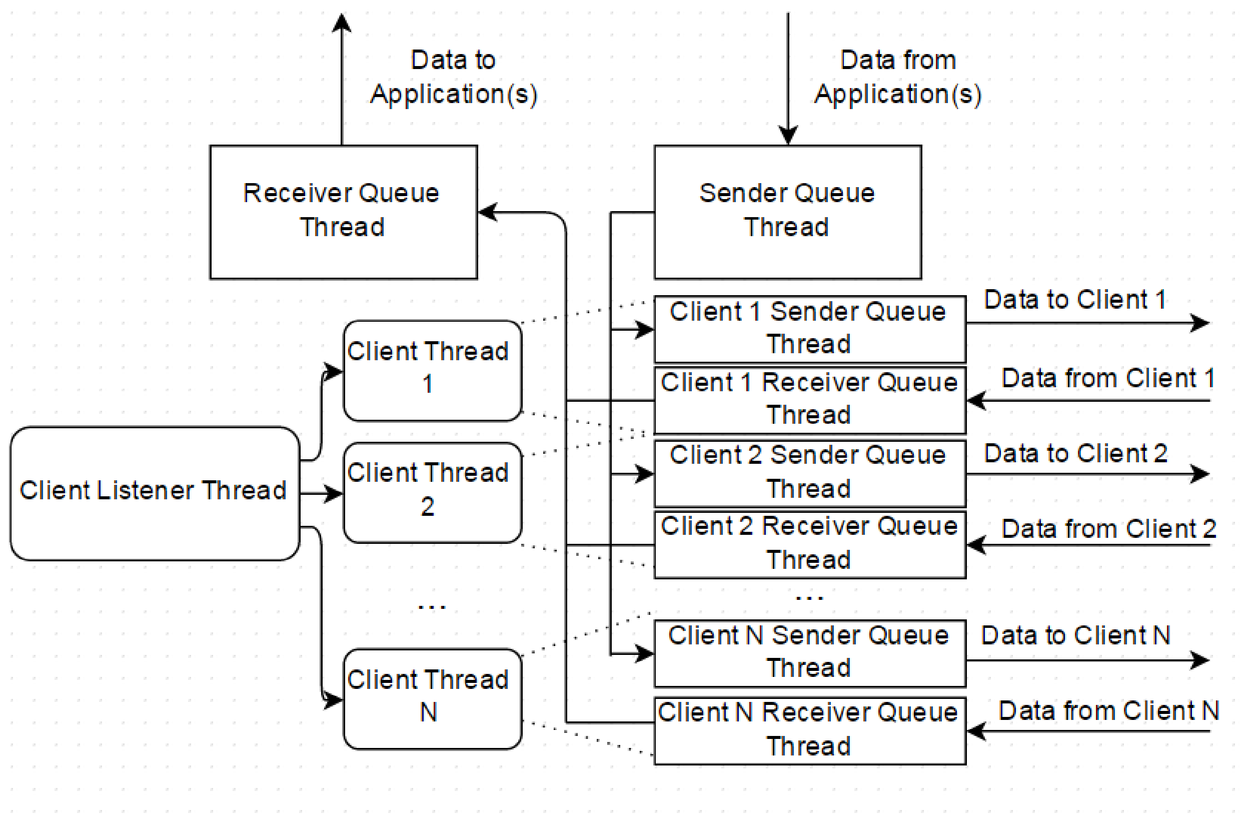
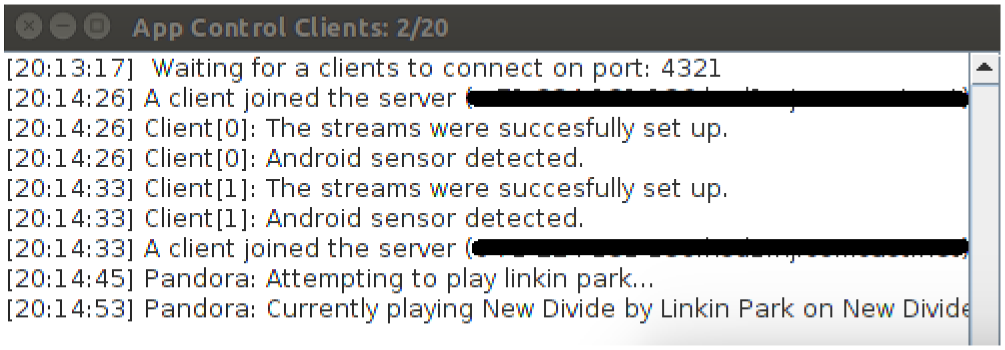


Figure 3 A diagram showing data communication inside a server project.

For example, the received data from the client application is placed in the receiver queue, the client thread then parses the data and creates a user defined type object which is then placed to the main receiver queue. The receiver queue then determines (based on the type of message) an application to which the data needs to be sent. To send the information back to the clients, the message is placed in the sender queue (based on the type of the message or the unique client ID). The sender queue places the message(s) into specific client(s) sender queue(s) where they are properly formatted and sent to the client.

Each sender/receiver queue utilize the blocking queue mechanism which provides thread-safe operations, therefore one (or more) thread(s) serves as a consumer while the other serves as a producer. Each blocking queue has a limited buffer size therefore a case where faster production creates unbounded buffer is possible but is bounded.

As seen in the screenshot captured from the server console window, it shows two separate android clients connecting to server (Figure 4). It provides a user a good way to find out information about the client and issue commands.



```
[20:13:17]  Waiting for a clients to connect on port: 4321
[20:14:26] A client joined the server ([redacted]
[20:14:26] Client[0]: The streams were succesfully set up.
[20:14:26] Client[0]: Android sensor detected.
[20:14:33] Client[1]: The streams were succesfully set up.
[20:14:33] Client[1]: Android sensor detected.
[20:14:33] A client joined the server ([redacted]
[20:14:45] Pandora: Attempting to play linkin park...
[20:14:53] Pandora: Currently playing New Divide by Linkin Park on New Divide
```

Figure 4 Sample output of the server console window.

<u>Building the Client Application:</u>

In general, the application has very few restrictions, but using this system it needs to have an access to TCP/IP interface card (physical or virtual) if run locally or the internet to reach the server. It also needs to be able to receive data continuously, and an ability to have messages formatted according to the server specifications. The focus of this section is to show a brief overview how the android application was built.

The goal of this client application is to utilize the Android client's touch capabilities and the processing power to demonstrate the ability of the system to communicate quickly and via server's help distribute the state/data to similar clients. Figure 5 shows the example application which was made to

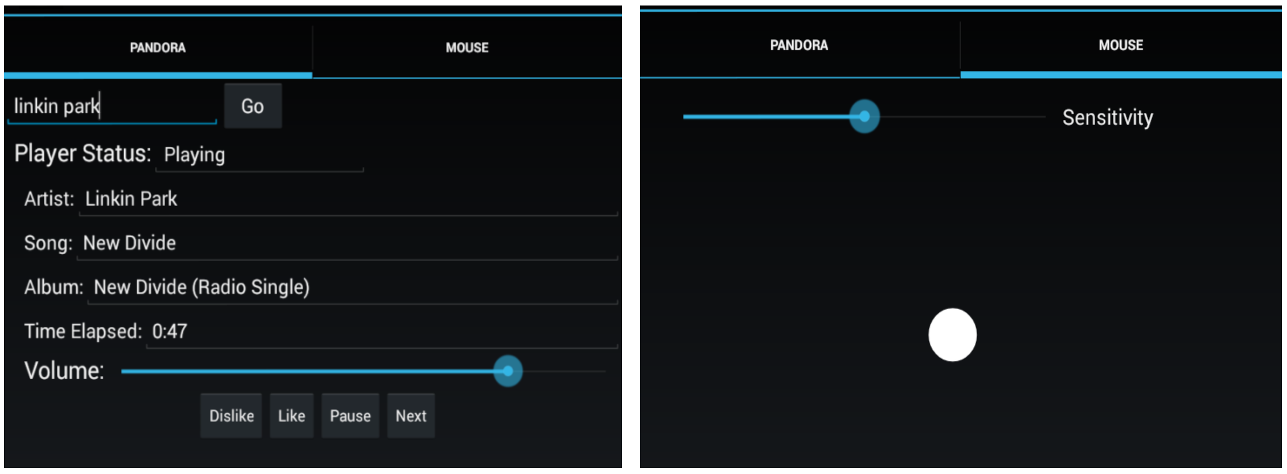


Figure 5 Android 2-tab application demonstrating the music player (left) and mouse control (right), both of which communicate data (bi-directionally) to the server.

demonstrate the ability of the server to communicate data efficiently.

The structure of the Android application is fairly complex, but the basic diagram is shown in Figure 6. The Pandora and Mouse Control Activities run in separate windows, while the background services such as receiver and the sender queue's handle the data going to/from server. The information regarding the states, button labels, etc., is not the importance in the context of this system.

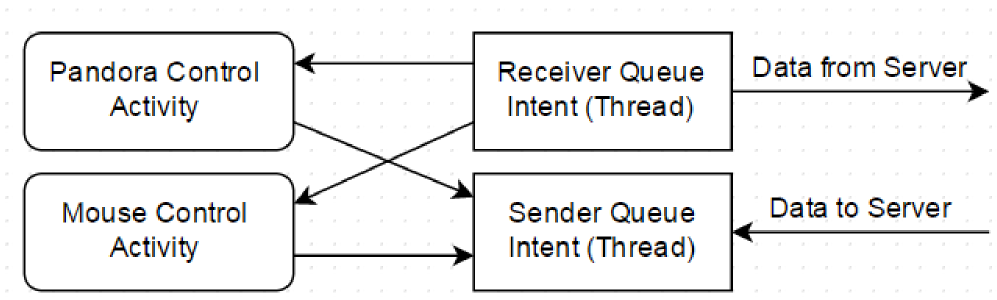


Figure 6 Diagram showing data communication in Android application.

<u>Building the Application:</u>

The application which makes the use of the data that comes from the server utilizes a similar mechanism of blocking queues. The Pandora music player application periodically obtains the state from the web music player and sends the information to the server. The server then parses the message, which requests that the data is to be sent to all Android type clients that are connected.

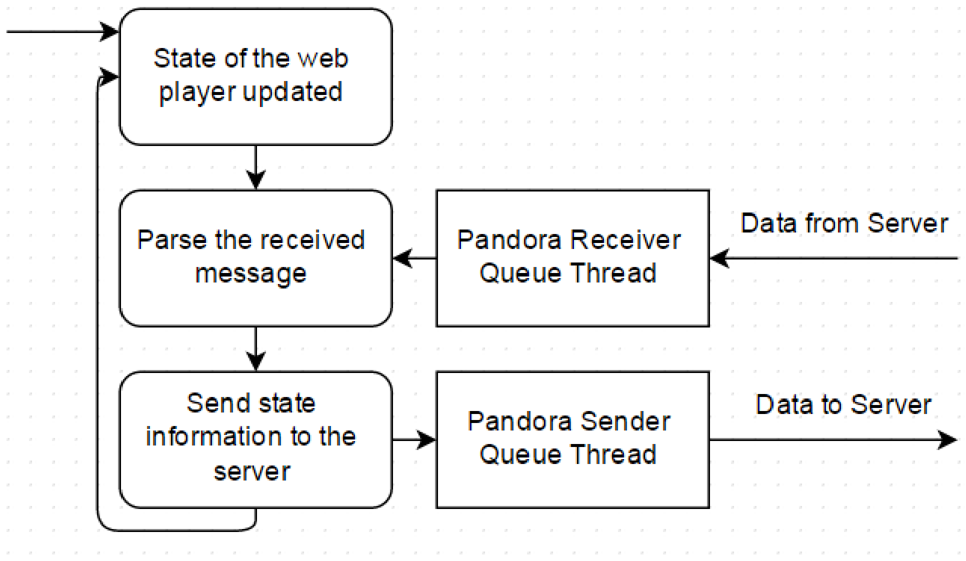

State of the web player updated
Parse the received message
Pandora Receiver Queue Thread
Data from Server
Send state information to the server
Pandora Sender Queue Thread
Data to Server

Figure 7 A diagram showing data communication within Pandora Player the application.

Figure 7 shows the data flow inside the Pandora application, the application uses Selenium API to obtain the information from the live web player using css tags. The same API is used to control the webpage (seen in Figure 8), where the messages received from the server are interpreted.

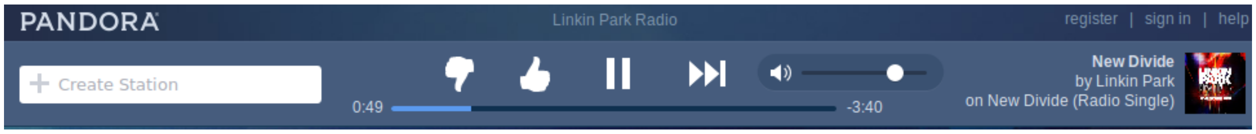


Figure 8 A webplayer which was controlled by pandora application (using css tags)

The mouse control application uses similar mechanism as the Pandora application, but it was used to demonstrate the ability to send/receive messages quickly for a smooth control of the operating system's mouse pointer (even when multiple clients are trying to control it).

The applications did not need a separate process from the server, by simply adding a separate thread for each was satisfactory to demonstrate the function of the system. The mouse and android applications were running smoothly and simultaneously, all while sending the data to multiple clients as well as being controlled by multiple clients at the instance. The result was a non-blocking communication of data, messages were received and executed in order and the server was able to handle high loads of data sent to it from the android mouse controller application.

Related Technology:

The communication of data between systems is usually done using TCP/IP connection, but I wanted to investigate if there were any other data links that would allow for a better communication of data, especially for the cases when critical equipment needs to respond quickly. I've been able to find a communications link called InfiniBand which has a very high bandwidth (speeds of up to 100 Gb/s, 150 million messages per second) and very low latency communication (Khan, Jameel, & Shafi, 2014). InfiniBand also provides emulation layers which allow TCP/IP applications written in Java to be used, but due to such emulation the performance is lost. MPJ Express is a Java messaging system which is able to improve the speed of InfiniBand using an interface which implements an MPI-like interface.

While InfiniBand seems like a good alternative for the applications where low latency and high throughput is required – it's definitely not needed by a regular user. The cost of such system is really high as well, and might only seems practical for super computers.

Conclusion:

During the development of this system I was able to successfully apply the topics presented in the course. The few keys ideas I was able to improve upon were mechanisms of synchronous data communication, multi-threaded communication of data and how to piece together a system developed in multiple separate projects.

In regards to synchronous data communication, I was able to use message like queue systems which allowed for orderly passing of data to specified clients. During the development of the Android application, I needed to implement a way to share the resources between two different activities (which are almost separate processes) - I therefore learned a great deal about the issues that might arise when multiple parts of the applications try to access the shared resources and safe-guards needed to prevent problems.

I expanded my knowledge in regards to the communication of data between separate threads, where again I was able to implement consumer/producer queue's to communicate data. As the system became larger and more complex I was forced to utilize the IDE's debugging resources to observe the application and to quickly debug it.

The biggest area which improved upon is the design of a complex system with multiple parts which were developed individually. To my surprise I was fairly quickly piece together different components resulting in a system, which is able to work very well. I think this was partly due to how the systems were interconnected – where a specific interface was shared between the interconnecting components.

Overall, this project has been a great experience in expanding my knowledge about how to interconnect different types of applications/sensors and the methods learned during the course of the development of the system provided invaluable experience.

References:

[1] Kemmerling, Stephen. "WebSockets vs. Regular Sockets - Kifi Engineering Blog." *Kifi Engineering Blog*. Kifi, 22 July 2013. Web. 09 Dec. 2014.

[2] Khan, Omar, Mohsan Jameel, and AAmir Shafi. "High Performance Message-passing InfiniBand Communication Device for Java HPC." *Procedia Computer Science*. Science Direct, 13 May 2014. Web. 09 Dec. 2014.

[3] Paul, Javin. "BlockingQueue in Java – ArrayBlockingQueue vs LinkedBlockingQueue Example Program Tutorial." *BlockingQueue in Java – ArrayBlockingQueue vs LinkedBlockingQueue Example Program Tutorial*. JavaRevisited, 5 Dec. 2012. Web. 9 Dec. 2014.

[4] Sander, Mak. "Modern Concurrency and Java EE." *Branch and Bound Blog*. Branch and Bound, 22 July 2012. Web. 09 Dec. 2014.