Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 95 (2016) 201 – 208

Complex Adaptive Systems, Publication 6
Cihan H. Dagli, Editor in Chief
Conference Organized by Missouri University of Science and Technology
2016 - Los Angeles, CA

MMO Smart Servers Using Neural Networks for Intelligent, Client-Handling Decisions and Interactions

Ben Smith*

Graduate Student at Missouri University of Science & Technology, 1825 E. Republic Rd. Apt 5203 Springfield, MO. 65804

Abstract

This paper proposes a complex, adaptive System of Systems architecture whose goal is to intelligently and dynamically host Massive Multi-Player Online (MMO) games. Furthermore, the use of a Perceptron employed with a Sigmoidal Membership Function trained by the Least Mean Squares learning algorithm is proposed. The network intelligently manages communications and interactions between equal, subordinate, and client level servers based on a reward returned by each respective neural network. The success of these techniques allows for a fully open and interactive world with minimal server/client maximums, minimal load times, and minimal network down-time. This freedom is due to the dynamic trading of resources and/or hosted clients during execution. This paper also outlines a proof of concept application designed to demonstrate the viability of this concept. Experimental results show that the complex system of systems is able to quickly adapt to the quickly changing environment thereby proving its plausibility as an adaptive and dynamic server management system.

Keywords: System of Systems; Server-Client; Least Mean Squares Algorithm; Perceptron; Sigmoidal Membership Function

1. Introduction

In real world applications, Neural Networks have proven to be an effective, expedient, and resource light solution to solving complex problems. The very nature of a Neural Network is that with a little bit of training and resources up front, you can save a tremendous amount of resources and architecture design headaches in the end. A good example of this concept would be the use of a Neural Network to identify letters in an image or to track objects travelling in space with little to no assistance from a human being. These concepts, if written in raw programming

doi:10.1016/j.procs.2016.09.314

code, could take hundreds if not thousands of lines of logic to cover all the basis of one of the concepts. With a Neural Network, this coding effort can be reduced tremendously requiring only a bit of training logic and the implementation of the equation in the program. It was for this reason the author chose to implement this concept into a complex System-of-Systems architecture for Server-Client cross-talk on a Massive Multi-Player Online (MMO) Server Network. The goal of this proof-of-concept application is to allow an abnormally large number of users to seamlessly integrate with one another, and their completely open world, without load or lag while using minimal server resources at all times. Optimistically, this would allow the user to move throughout a vast, open world with no load times and with an unrestricted amount of players moving around the world with them, synchronously. Furthermore, it would also allow for a more dynamic and fluid maintenance platform for developers and server administrators.

The use of simulated Neural Learning in Smart Servers is not entirely unique as can be seen in examples such as [1] [2], however, the concept of Servers intelligently spawning and communicating with other servers, to the knowledge of the author, is unique as the majority of servers today suffer cross-talk limitations that, either, do not allow unrestricted, synchronous player interaction or cut the world into smaller, separate loading areas of the world. In using a Neural Network, the author aims to allow a network of servers the ability to make intelligent, trained decisions to determine the optimal method of serving all clients while constantly equalizing the load amongst all possible servers.

2. The Structure

The System-of-Systems architecture proposed includes the use of three primary server types which are used recursively and a client type. The server types are the *Host Server*, *Map Server*, and *Player Server*. Each server entity has a responsibility to maintain a specific aspect of the network, but may overlap in their coverage.

2.1. Hosting Server

The *Hosting Server* is the primary point of contact for all *Subordinate Servers*. It is important to note that Hosting Servers may nest themselves in order to provide better coverage over clients and fellow servers more effectively. For this reason, the Hosting Server may, itself, be an “equal subordinate” of a different Hosting Server, but as it doesn’t function with a Neural Network, nor is it submissive to any other server than one of its own pedigree, it is not referred to as a Subordinate throughout the remainder of this paper.

The responsibilities of the Hosting Server include:

1. Listen for New Client connections
    a. Poll Subordinate Servers for Hosting Bids based on individual clients
    b. Evaluate Hosting Bids to determine optimal client ownership
    c. Serve New Clients with Map and Player Servers
2. Monitor and Serve Subordinate Servers
    a. Notify subordinates of proximity with other, like subordinates
    b. Negotiate “Transition Requests” to trade clients between subordinates
    c. Spool new subordinates when no subordinates are available
    d. Spool new Host servers when unable to maintain its own subordinates

2.2. Subordinate Servers

Subordinate servers may be any server controlled by another server. The primary Subordinate relationship is between the Hosting Server and its Map and Player subordinates. The fine print of a subordinate is that it evaluates its client-server ownership relationship by running their distances through a simple Neural Network called a *Perceptron,* described in detail in section 4, is used to produce a single value or reward. This reward is then used to determine the ownership of that client. Evaluation takes place for three reasons, *Creation, Optimization,* and *Transition*.

Creation is the act of finding an initial Map and Player to own a client when it first connects to the network. The request is sent from the Hosting Server to evaluate a new client, each of the Hosting Server’s subordinates evaluate said client and return a Bid which is an accurate representation of the Subordinates’’ capability of optimally owning that client. Due to the large impact of the distance between currently owned clients on the Bid, the scores reflect the optimal Map and Player servers for the client ensuring the Map is only having to serve a single, reasonable map and that the Player server is not stretched too thin by covering owned, straying clients.

After creation, the subordinate's clients may run freely, even if that means they run out of the optimal service range of their Map or Player servers. For this reason, each Subordinate engages in constant *Optimization*. This act is characterized by the subordinate evaluating each of its clients recursively looking for outliers or "weakest links". Furthermore, the Host Server monitors the proximity of like subordinates to determine when said subordinate's clients may start to "intermingle". If formerly connected clients begin to stray or start to intermingle with other subordinates' clients, then a *Transition*, seen in **Error! Reference source not found.**, is requested by either the subordinate owner of the stray, or by the Hosting server who has noticed the proximity collision of two subordinates.

In a *Transition*, the subordinates submit and evaluate one another's weakest clients to determine if the other subordinate is a better candidate for ownership. The Host Server calculates the bids and conducts the trades resulting in optimized client ownership for each subordinate. Essentially, should Subordinate B, with which Subordinate A collided, bid higher on A's owned client than A did, then the Hosting Server notifies the client of the trade and the client immediately assumes the patronage of B; the new, more dominantly bidding subordinate.

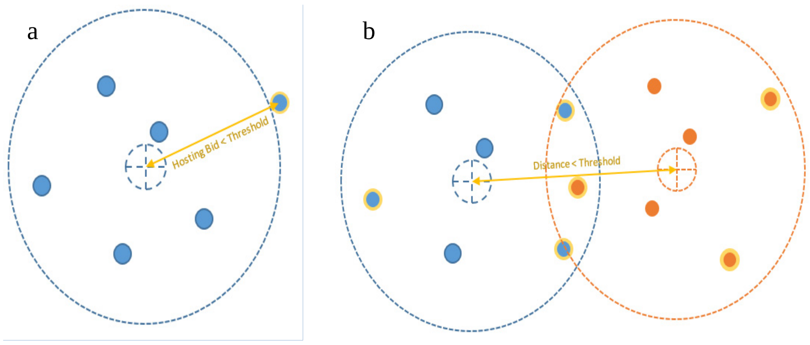


Figure 1(a) Subordinate Initiated Transition (b) Host Initiated Transition

2.3. Map Server

The *Map Server* is a subordinate server that is very nearly self-explanatory; it serves the map to clients. Using evaluation, its goal is to serve clients who are tightly grouped allowing the Map Server to make fewer calls for Map updates from storage, thereby resulting in quicker served maps for clients. Without a Map Server, the client would be in a black abyss. The Map Server also sends the client a larger map than is necessary. This is to reduce the amount of times the Map Server must serve the map. Essentially, the map is served with "Host Padding" of an administrator defined distance that allows the client to run through the buffered map before absolutely having to be updated by the Map Server, as seen in Figure 2.

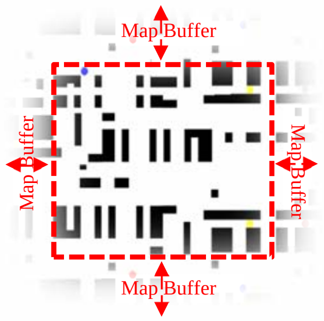


Figure 2 Dramatized Map Service

2.4. Player Server

The *Player Server* is a bit vaguer; it serves all a clients' interaction with its surroundings including other clients. Essentially, the Player Server's goal is to provide all content within eye shot of the client in a timely fashion. Due to this interaction being less data intensive, a Player Server can host much larger distances between clients as well as many more clients than a Map Server, but still aims to clump proximate clients together to allow for better resource management (i.e. no servers serving a single client). Without a Player server, the client would be alone on an empty map.

2.5. Clients

Clients present little in the way of the network functionality, but provide all necessary feedback for gameplay to the end user. The client sends constant updates to its assigned Hosting Server, Map Server, and Player Server to ensure that it as well as other players are up-to-date. The client is entirely submissive, only requesting an initial connection and being completely directed from the Hosting Network. This is unique in MMO Server architectures as the Client generally selects a Genre or specific server of which to connect.

3. The Process Flow

The communication process, seen in Figure 3, is as follows. The single Host is initiated by the server administrator when the application is executed. The Host immediately connects and listens to a communication link (for example, in the POC project, it listens over a port), then, when ported, spools up a Subordinate Map and Player server who also listen to their own port. The end user spools a client and connects to the Host Server via the "Host Connection Screen". Once the connection is initiated, a request is sent to the Host Server containing the clients' unique information and a location of the client's spawn in the Map. This request is forwarded to the Map and Player Servers. When received the Map and Player Servers Evaluate the client's request via their Neural Networks and return the Hosting Bid produced by said networks to the Hosting Server.

Once an adequate bid is received for both service requirements, the Hosting Server selects the highest bidder from the subordinate responses and notifies the client of its newly appointed Map and Player Servers. The client may then navigate the map via the "WASD" navigation keys on the keyboard. The clients' locations are then monitored closely by their owners and the subordinates are watched closely by the Hosting Server. As mentioned above in the description of Subordinates, Transitions are used to ensure a re-optimization of the network throughout performance.

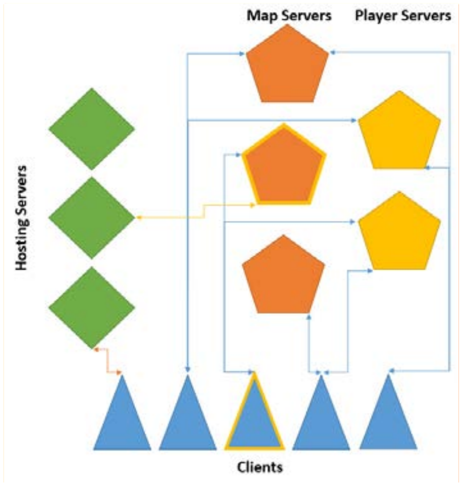


Figure 3 Communication Structure

4. The Neural Network

The Subordinate Servers are assigned and *Transitioned* clients based on the Hosting Bids produced by the Perceptron contained in each subordinate. A perceptron described in [3], with examples of application in [4] [5], is a very well known, basic Neural Network model that allows the client to receive flexible, unmapped inputs and convert them into generalized target values. In the POC, the NN is trained before implementation, and it's inputs, seen in Figure 4 are: the distance of the client from the subordinate (whose location is calculated as the centre of all of its clients), *D*, the current performance index (a score of the subordinate's current service based on numerous factors such as network throughput, hardware resources being used, currently owned clients, etc.), *P*, and the subordinate's pre-set Performance Threshold, *T*, which acts as a point of reference at which the server is considered "Stressed" and needs assistance and is not recommended for ownership of more clients. Once each input is multiplied by its previously calculated weight, ω, all are summed together in addition to a previously calculated bias, β. This summed total is then processed by an equation or *Activation Function* producing a single, generalized reward or "Hosting Bid". The processing power and

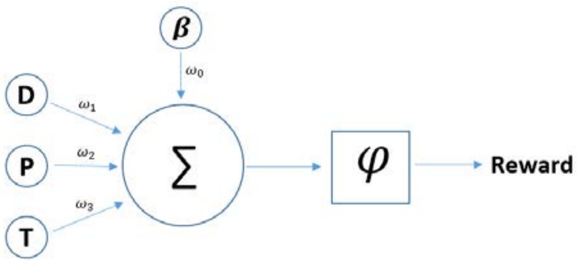


Figure 4 Perceptron Diagram

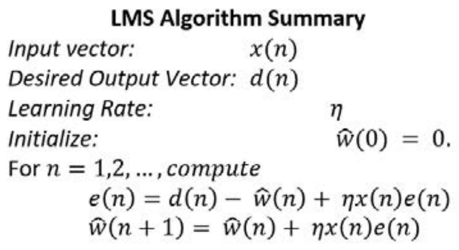
resources necessary to compute these generalizations are little to nil. The perceptron was chosen for this POC due to its simplicity in implementation and for readers to immediately understand how it works. Future research will be conducted to experiment with the use of other more advanced Neural Network and computational intelligence concepts and will, likely, greatly increase the performance of this POC application.

4.1. The Components

To briefly surmise above, the inputs, are distance between entities, the Current Performance of the Subordinate, and the Threshold at which the network is considered "stressed". These inputs are multiplied by uniquely trained weights and then fed into a "Combiner" which computes a single value from all of the weights. The single value is then fed through another mathematical component known as the *Activation Function*. In the case of the Proof of Concept application, the Activation Function used is the Sigmoid Membership function. This equation serves as a means of "squashing" the values, possibly infinitely high, back into a reasonable range. The process of running the value through this perceptron is extremely simple once trained; the real magic of a perceptron is in its ability to be trained.

4.2. The Training Algorithm: a dirty intro

Neural Networks, including perceptron may utilize numerous training algorithms to learn to generalize between inputs and outputs but, due to its simplicity, only the Least Mean Squares Algorithm, Figure 5, is defined within the scope of this paper. Furthermore, the author assumes a general understanding of Machine Learning algorithms of the reader. Basically, this method employs a supervised learning technique in which the network is given a learning rate *η*, generic weights *w*(0,1,…, n), then fed training inputs, x(1,2,…, n), and the desired outputs of the network when given said inputs. The *desired output, d(n),* then has the *actual output*, $w^T(n)x(n)$, of the network subtracted from it to find the *error*, *e(n)*. The difference in the errors is computed by finding the mean of all received errors squared. The mean squared error, when tracked from one "Epoch", or training iteration, to the next is a calculation of the gradient descent of the error toward an acceptable error level. The network then uses the error, learning rate, the input, and the original weight to produce the new weight. This process is repeated maintaining the steepest descending gradient value per epoch until the error is reduced below a user defined threshold. The network, having trained weights, can now be used to "generalize" an expected output.



LMS Algorithm Summary

Input vector:	$x(n)$
Desired Output Vector:	$d(n)$
Learning Rate:	η
Initialize:	$\hat{w}(0) = 0.$

For $n = 1,2, \ldots, compute$

$$e(n) = d(n) - \hat{w}(n) + \eta x(n)e(n)$$
$$\hat{w}(n+1) = \hat{w}(n) + \eta x(n)e(n)$$

Figure 5 LMS Algorithm as defined in [3]

5. The Proof of Concept

The application written by the author was developed to demonstrate the basic implementation of this concept in action and was greatly scaled down to make the concepts easily reproducible and visible. The design choices and scaling resulted in interesting results both in implementation and training.

5.1. Original Design

Some design choices, such as Console application implementation for servers, WPF implementation for clients, and the use of object-oriented, .Net C# are completely flexible choices and were selected only due to the author's familiarity with these implementations. The author concedes to the superior run-time performance of the C++ language for high performance applications, but the scale of the POC application was deemed too minimal for performance of language implementation to be a factor. Furthermore, the implementation of basic, predefined data types and address-accessed arrays allowed for quicker performance than the use of iteratively searched data types such as lists. The MVVM design pattern was selected for its implementation benefits with the WPF framework and UDP socket connections were selected to emulate "Fire and Forget" internet communication, even though the ports were all locally hosted on a single PC.

The map used by the application was created by an "Autonomous Walker" object that was written by the author. This object is initialized, given a blank canvas in the form of an integer array containing all zeros, and issued a specific amount of steps to walk before stopping. Each step the walker takes through the array; it converts the zero at that step to a one, thus, "carving" the map out of the canvas. The walker used to generate the map provided took 3,600,000 steps before completion and generated a usable map over 2000 units^2.

Table 1. Perceptron Training Results.

Distance Range	Mean Squared Error Threshold	Network Accuracy %
1000	.015	99.7
70	.03	82.3

The perceptron utilized by the Subordinate Servers are written in C# as objects and employ explicitly set weights and a Sigmoid Membership activation function. This implementation was chosen for its simplicity and easy expandability; future implementations of this POC will include more complex neural mechanisms as well as more advanced and effective learning algorithms. The author utilized MATLAB to create and train the weights of the perceptron, then set the trained values explicitly in the application. The network was trained using a learning rate of .001 for a maximum of 500 epochs. The training results can be seen in Table 1. One lesson learned by the author was that when the original concept was trained, the distance of 1000 units was used as the maximum of training points which yielded an exceptional accuracy rate. When the distance was scaled down greatly for the POC (greater than 90% reduction), the same training algorithm performed at a greater than 10% decline in accuracy even with an increased acceptable Mean Squared Error threshold. This loss in accuracy was deemed reasonable as the outputs still reflect reasonable target outputs for the distances in the application and the subordinates can directly fluctuate the output of the perceptron by raising and lowering their "Threshold" inputs. It is also a reasonable assumption that accuracy will be lost when more "Fuzzy" concepts are added to the performance such as hardware or network factors.

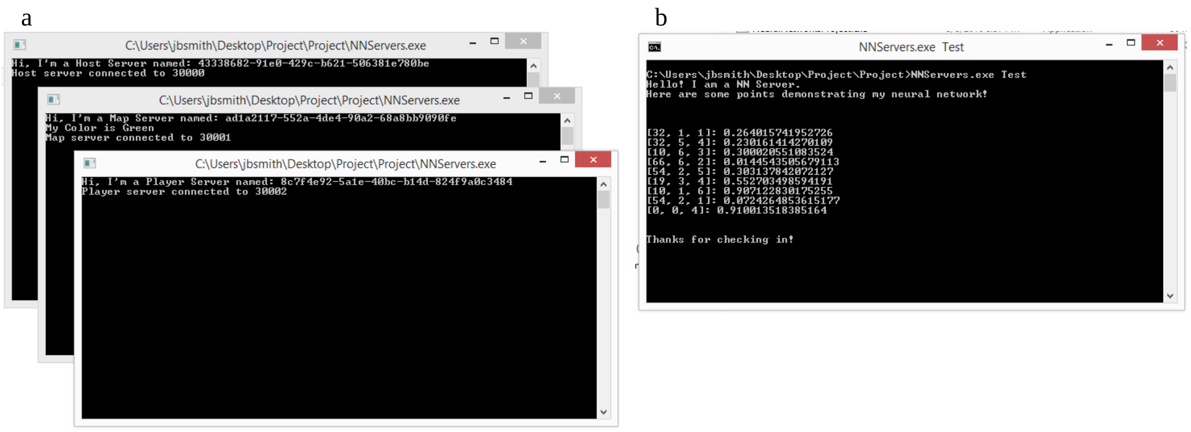
While the "Performance" and "Threshold" inputs are very flexible in that they can be calculated in a real world environment to include factors such as network bandwidth, currently owned clients, CPU and Memory resources, etc., the use of one PC for the POC application resulted in the Performance and Threshold inputs being a single integer value representing the current number of owned clients and a maximum number of clients to be owned. The Subordinates' threshold of maximum clients can be increased or decreased to change the "Hosting Bid" of each subordinate. It was also found that the bandwidth threshold of the Player Servers was far less than that of the Map Server due to the mass amounts of Map data needed to keep the client in the game. The Player, therefore, is capable of serving a much greater number of clients and producing a visible transition of Player servers is quite difficult in the scaled down POC.

All entities are set with timers that trigger at 100ms by default, however, the Map Server was allowed an exception to this. The size of the client's served map was selected to be 25 units^2 while allocating a 10 unit^2 buffer around the edges of the client served map to create the illusion of a constantly updated map. Due to this buffer availability, the Map servers service rate was decreased to once per 500ms allowing better bandwidth management. A "Color" attribute was added to Map Servers to allow the User Interface Client to distinguish the difference between clients when hosted by different servers.

The User Interface created for the application was designed to be used by multiple users where all could join the game, however, a hard-coded IP address pointing at the localhost of the hosting PC was used, therefore, multiple players are not available. For testing purposes, however, the author created a client server option in the NNServers.exe structure that generates an autonomous client who is spawned and acts independently of user interaction (Fun Fact: The autonomous clients' walking choices are powered by the same Autonomous Walker object that carves the Map Server's maps, but the maximum step threshold is turned off making these clients walk infinitely).

Finally, as a necessary change, the server applications were broken out into individual console applications due to Windows application resource restrictions. The original POC was developed using Multi-Threading to emulate the use of multiple, separate programs, however, the thread overlap proved to be too demanding and resulted in poor service results for the Map and Hosting server implementations. To circumvent this challenge, the author created a single Server program, NNServers.exe, to be called recursively to generate the different server types, and a single client User Interface Application called NeuralNetworksProject.exe for the use of live end users.

5.2. The End Result



```
Hi, I'm a Host Server named: 43338682-91e0-429c-b621-506381e780be
Host server connected to 30000
```

```
Hi, I'm a Map Server named: ad1a2117-552a-4de4-90a2-68a8bb9090fe
My Color is Green
Map server connected to 30001
```

```
Hi, I'm a Player Server named: 8c7f4e92-5a1e-40bc-b14d-824f9a0c3484
Player server connected to 30002
```

```
C:\Users\jbsmith\Desktop\Project\Project>NNServers.exe Test
Hello! I am a NN Server.
Here are some points demonstrating my neural network!


[32, 1, 1]: 0.264015741952726
[32, 5, 4]: 0.230161414270109
[10, 6, 3]: 0.300020551083524
[66, 6, 2]: 0.0144543505679113
[54, 2, 5]: 0.303137842072127
[19, 3, 4]: 0.552703498594191
[10, 1, 6]: 0.907122830175255
[54, 2, 1]: 0.0724264853615177
[0, 0, 4]: 0.910013518385164


Thanks for checking in!
```

Figure 6 (a) Server Windows During Execution (b) Example Neural Network Outputs from Client

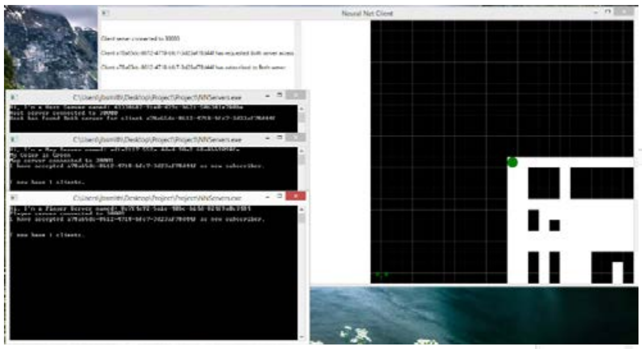
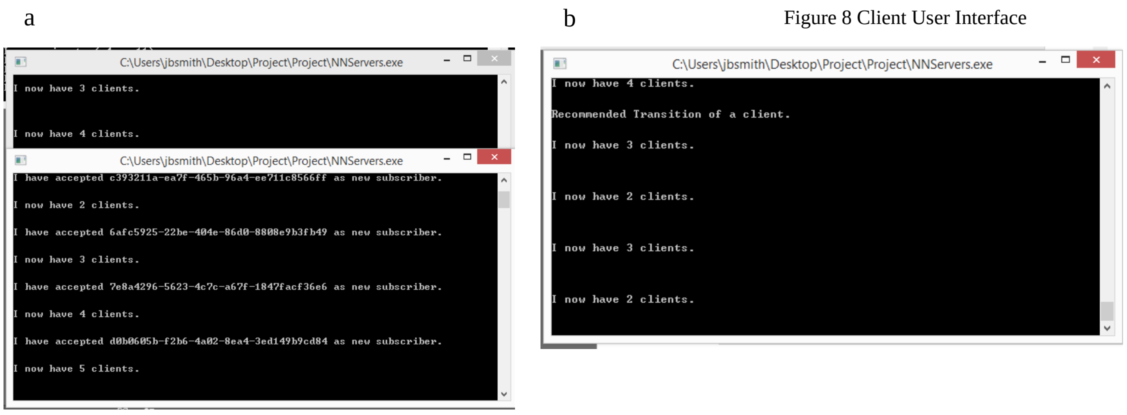


Figure 7 All Servers During Initial Connection

Figure 8 Client User Interface



```
I now have 3 clients.


I now have 4 clients.
```

```
I have accepted c393211a-ea7f-465b-96a4-ee711c8566ff as new subscriber.

I now have 2 clients.

I have accepted 6afc5925-22be-404e-86d0-8808e9b3fb49 as new subscriber.

I now have 3 clients.

I have accepted 7e8a4296-5623-4c7c-a67f-1847facf36e6 as new subscriber.

I now have 4 clients.

I have accepted d0b0605b-f2b6-4a02-8ea4-3ed149b9cd84 as new subscriber.

I now have 5 clients.
```

```
I now have 4 clients.

Recommended Transition of a client.

I now have 3 clients.



I now have 2 clients.



I now have 3 clients.



I now have 2 clients.
```

Figure 9 (a) Subordinate Server Accepting Clients (b) Subordinate Server Transitioning Clients

6. Conclusion

As demonstrated in the many screenshots of actual gameplay above, the implementation of this concept demonstrated positive results. All servers ran as individual consoles, seen in Figures 6a and 8, and their internal NNs performed as expected as can be seen in Figure 6b. Several challenges presented themselves during this process including the threading resource issues, the difficulty to trigger Player Server Transitions, and the substantial coding effort that went into creating and designing the application components (Fun Fact: This coding effort utilized two separate Visual Studio Solutions, over 2200 lines of code and had a combined ending “Maintainability Rating” of 87%; a corporately acceptable programming solution).

Throughout gameplay, the user can see the communication of all servers working together, such as Figures 7, 9a and 9b, and by running away from gameplay and spawning new players that not only do the servers transition players between one another, but that the Hosting Server often spawns new subordinate servers to help with the onslaught of needy clients. While, due to the tiny scale, it requires a tremendous distance to get the servers to actually transition players to new servers, it is easily reproducible by travelling 2 or 3 lengths of the map away from like clients.

The proof of concept application proved to be a success. The challenges presented throughout this development all stemmed from the downward scaling of the concept for simple implementation with limited resources, however, it goes without saying that scaling this architecture up using actual hardware, more resource hungry gameplay, and fluctuating network lag will all pose significant challenges in the future. With this in mind, the author feels that future implementations of this architecture may benefit from using more advanced neural network models and optimal resources. Furthermore, the author plans to expand this POC to a real-world application and is optimistic that future implementations will be successful.

References

[1] R. V. Prasad Y and R. Pachamuthu, “Neural Network Based Short Term Forecasting Engine to Optimize Energy and Big Data Storage Resources of Wireless Sensor Networks,” in *IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, Taichung, 2015.

[2] J. Kane, B. Tang, Z. Chen, J. Yan, T. Wei, H. He and Q. Yang, “Reflex-Tree: A Biologically Inspired Parallel Architecture for Future Smart Cities,” in *44th International Conference on Parallel Processing (ICPP)*, Beijing, 2015.

[3] S. Haykin, Neural Networks and Learning Machines, Third Edition, Saddle River: Prentice Hall, 2008.

[4] G. M. Masters and R. L. Mattson, “Threshold Logic synthesis of Sequential Machines,” in *IEEE Conference Record of Seventh Annual Symposium on Switching and Automata Theory*, Berkeley, CA, USA, 1966.

[5] I. Morishita, “Analysis of an Adaptive Threshold Logic Unit,” *IEEE Transactions on Computers,* Vols. C-19, no. 12, pp. 1181 - 1192, December 1970.

[6] W. S. McCulloch and W. Pitts, “A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY,” *BULLETIN OF MATHEMATICAL BIOPHYSICS,* vol. 5, pp. 115 - 133, 1943.