Term Report: Development and Testing of AWS Environments to Run MASS Applications

Christopher Bowzer
CSS 499 with Dr. Fukuda
Winter 2016

For this quarter, the subject of my undergraduate research was the Development and Testing of AWS Environments to Run MASS Applications. This ended up being an extremely complex project for me as it was my first time working with distributed programming and AWS. While currently, MASS applications are not running as intended on AWS, I believe this quarter was productive in discovering issues to be considered before the public release of MASS.

Discovery

The first stage of this quarter for me was developing a good working knowledge of both the MASS library, and AWS. Without understanding both of these, it was impossible to move forward.

Understanding MASS

The MASS (Multi-Agent Spatial Simulation) Library is a library developed by Dr. Fukuda for use with C++ and Java. MASS enables the simplification of parallelizing programs featuring simulations of 2D space (with or without independently acting agent) and for parallelizing big data processing. Though there are several other parallelization libraries available, one of the goals of MASS is the ease of use for programmers.

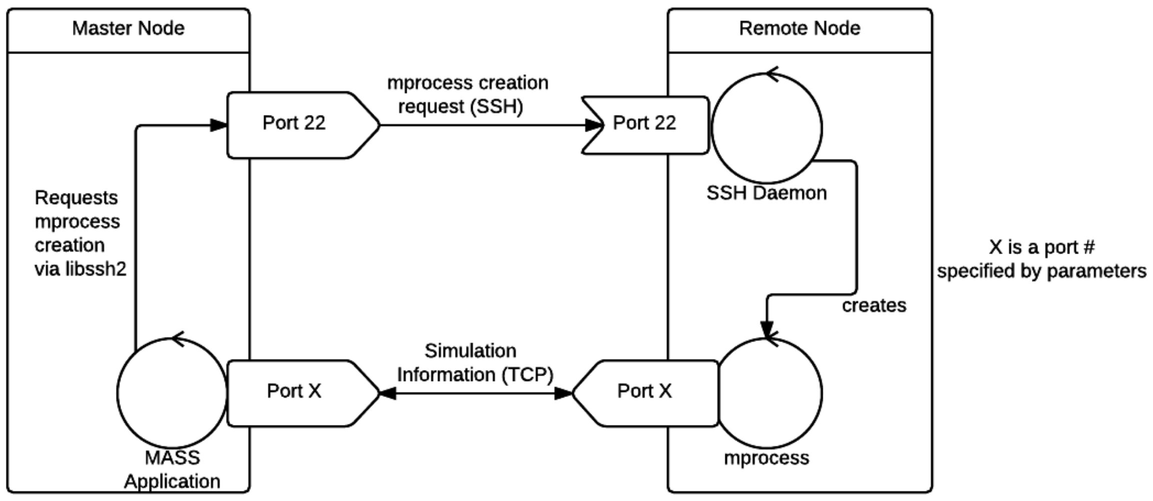
Agents and Places

MASS achieves ease of programming by allowing application specific agents and places to extend MASS's agents and places. After that whenever the application programmer makes calls to perform movements or exchange data, the MASS library takes care of synchronizing the process between utilized machines. MASS also parallelizes multithreading on each individual machine by mapping groups of places to specific threads.

Network Protocols and Mprocess

To achieve communication and parallelization across a clustered network, MASS uses two separate communication protocols between nodes. The first is SSH (secure shell connection), aided by the libssh2 open source library. This use of SSH is not to perform communications between nodes during the actual simulation, but to establish mprocess on the remote node. This is done when the SSH Daemon (a program bound to port 22 to intercept and act on SSH requests) on the remote node receives and executes a request from the master node to establish mprocess and bind it to a specified port. Once bound to the specified port, the mprocess establishes a TCP (Transmission Control Protocol) connection with the master node. All further communications between the master node and remote nodes is done via this TCP connection. This process of establishing a connection for simulation communication is shown in figure 1.

Figure 1: Creation of mprocess and establishing TCP connection


Master Node
Remote Node
mprocess creation request (SSH)
Port 22
Port 22
SSH Daemon
Requests mprocess creation via libssh2
X is a port # specified by parameters
creates
Simulation Information (TCP)
Port X
Port X
MASS Application
mprocess

Understanding AWS

AWS (Amazon Web Services) is a cloud computing service offered by Amazon. Because of AWS's popularity with developers, it is predictable that many people will attempt to use AWS for their MASS applications once the library goes public.

Ec2 Instances

AWS's most popular service is the Ec2 virtual machine. These virtual machines can be quickly launched from standard images (Ubuntu, Windows Server, Redhat Linux, etc.) with varying hardware. Additionally, it is possible to save images for relaunch in the future.

Setting up Ec2 Environment

From the default Ec2 Ubuntu image, creating an environment involved several steps, and coming to a better understanding of the AWS environment. The simplest step necessary to creating the environment is as simple as installing a C++ compiler, as the standard Ubuntu image has been stripped down for efficiency to not include one by default. Additionally, installing the libssh2 library was required, as it is a dependency of the C++ version.

Setting up MASS

Since MASS had previously been used almost exclusively at the UW Bothell Linux Lab cluster, most of the path identifiers were machine specific. This includes the Makefile, as well as the compile.sh for most applications. These paths had to be rewritten to be only relational to the directory the scripts were contained in. After compiling, the machinefile.txt and run.sh for each program must be edited, and a symbolic link to mprocess and killMProcess.sh made.

Key Difficulties

The first unforeseen difficulty of running MASS on an Ec2 instance was the use of private/public key pairs. The default SSH authentication on an Ec2 instance is a private key. Because MASS attempts to connect to all remote nodes with the same parameters, it is required to have all remote nodes share the same private key. Additionally, the standard placement for a private key to be used by mass is in the directory ~/.ssh/id_rsa so the key should be placed there. Since libssh2 requires both a public and private key pair, it is also necessary to create a public version of the key at ~/.ssh/id_rsa.pub.

One potential problem discovered with the MASS libraries public release though is that libssh2 requires an absolute path to keys when using them to initialize an SSH connection. Since there is often discrepancies between home directory names, the hard coded path is not guaranteed to work. I am not sure if there is an obvious fix to this, short of having the key paths given as a parameter by the application using MASS. This problem will require further discussion with Dr. Fukuda and the MASS team.

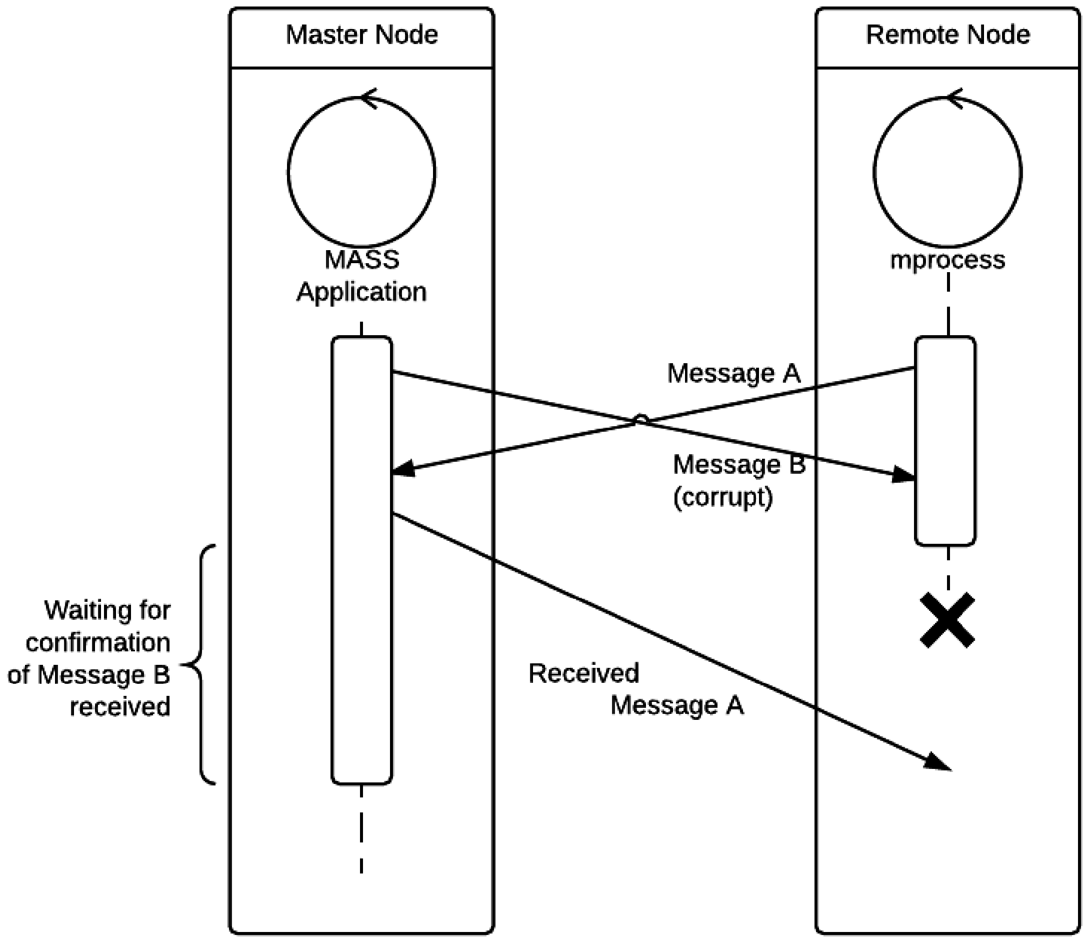
Firewall Difficulties

The second difficulty involved primarily my lack of understanding of the network protocols MASS uses to communicate (figure 1). Because I originally presumed that all communications were occurring via SSH connection, I did not originally customize the default Ec2 firewall that blocks TCP communications on all ports. After realizing this, I created a security group (firewall configuration) that allows TCP connections over port 11111 to enable communications of simulation data between nodes.

Unknown Error

With MASS appearing to work (allowing generation of places to remote nodes and completing initialization of TCP connection) I began attempting to test the performance time. When placing a timing output at the bottom of Sugarscape and Wave2D, I realized that although MASS appeared to be running for a time, it was not reaching the final timer output. Attempting to trace the origin of the problem with Dr. Fukuda's help, we discovered that the problem is probably occurring in ExchangeHelper::recieveMessage(int): Message*. This error is somehow occurring because the message being sent by the master node is becoming corrupted. When the corrupted message is received, the remote node fails to send a response, leaving the master node waiting to receive the response (figure 2).

Figure 2: Possible error in ExchangeHelper::recieveMessage(int)



Summary

Overall, although this term has not been as productive with regards to creating a fully working AWS environment of MASS and thereafter checking the efficiency of MASS relative to other parallelization libraries, I do believe it has been productive in terms of furthering my understanding of MASS, AWS, and distributed systems in general. I look forward to utilizing the knowledge I gained this quarter in future quarters. I think that I will be able to build on the quarter, and achieve good testing results next quarter so that MASS can be better compared to other libraries.