# An Implementation of Multi-User Distributed Shared Graph

Yuan Ma

Term Report

submitted in partial fulfillment of the

requirements of the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

March 12, 2024

**Project Committee:**

Professor Munehiro Fukuda, Committee Chair

Professor Kelvin Sung, Committee Member

Professor Robert Dimpsey, Committee Member

## Project Overview

Multi-Agent Spatial Simulations Library (MASS) is a parallel-computing library for multi-agent and spatial simulation over a cluster of computing nodes. MASS mainly contains two classes: Agents and Places. The former represents a collection of mobile objects in a simulation, each named agent. The latter represents a multi-dimensional array space of entities, each named place. Agents can migrate between places, regardless of the specific node or thread they are associated with [1].

Most big-data computing handles text data with data-streaming tools such as MapReduce [2], Spark [3], and Storm [4]. However, for distributed data structures such as distributed graph, these data-streaming tools need to disassemble the data into texts that cannot remain in their original shape over distributed memory before processing the data. On the other hand, many graph applications including graph database such as Neo4j [5] requires maintaining the original structure of the graph over distributed memory to function. Therefore, it's reasonable to introduce agent-based graph computing in which we deploy agents to graphs without modifying the original shape of the data structure [6].

Currently, agent-based modeling (ABM) libraries including MASS focus on parallelization of ABM simulation programs. However, database systems need to accept, handle, and protect many queries from different users, and ABM libraries do not have this capability. Therefore, in order to apply ABM libraries to database systems, in particular to graph database, the underlying graph must be accessible and modifiable by multiple users simultaneously. Given the above motivation, this project aims at investigating multi-user distributed shared graph (DSG) and trying to add this new feature to the MASS library.

## Project Goals

1. The project will get started with surveys on several platforms that facilitate shared space: Unix Shared Memory, Hazelcast [7], Redis [8], Oracle Coherence [9], and AWS SimSpace Weaver [10]. Particularly, we will check whether they can be used to create a distributed shared graph and compare their speed of running the same graph algorithm.
2. Based on the survey and prototyping, we will propose and implement a high-performance multi-user DSG.
3. Our implementation will replace MASS GraphPlaces and furthermore facilitate relevant methods.
4. After integrating DSG with MASS, we will complete verification and performance measurement for DSG-integrated MASS.

## Progress

During the autumn quarter we encountered many performance problems, but we managed to solve them and proposed a high-performance distributed shared graph system. Therefore, during this quarter, my first work is to implement the whole system.

Before explaining the whole system, I first need to introduce the tools that I used in my implementation.

To make the graph distributed, we use JSCH [11]. JSCH is a java utility for establishing remote connections using SSH2, this allows users to execute commands in remote machines, so we can use it to launch remote processes. After launching remote processes, the channel between the current java program and the remote java program will be established and we can also use it to send or receive messages.

As for the shared feature, mainly two tools are used. The first is Unix Shared Memory, which has been introduced in the term report of autumn quarter. Shared Memory is a memory shared between 2 or more processes, it's a way of inter-process communication. In order for the graph to be shared among different user's processes, we want the graph to be stored in shared memory. To be more specific, in Linux, the directory "/dev/shm" is an implementation of the shared memory concept, and it is in the virtual memory which guarantees good performance. We can create a file under this directory so that it can be accessed by multiple users. The shared memory access operations can be achieved by the class MappedByteBuffer [12] in Java programming.

Another tool is called Aeron [13]. Aeron offers a low-latency message transport system that guarantees performance and reliability. During the autumn quarter I was testing the speed of those common communication protocols such as TCP, UDP and Aeron, and I came into the conclusion that Aeron is the best solution for implementing the shared feature. Aeron uses publication and subscription mechanisms; publishers can send messages to specific channels and subscribers can subscribe to the channel so that they receive those messages. A channel is specified by an IP address, a port, and a stream ID, which means even if we only use one IP and one port, we can create multiple independent streams by changing the stream ID. The detailed technique of Aeron can be found in Fig.1. We can see that all subscribers of a channel will receive messages from publishers of the same channel. If there's a subscriber who joined the channel late, then he will only receive the messages sent later than his join time. This model is suitable for our shared feature because all the users in the same node only needs to join the same channel to get the latest updates from other users.

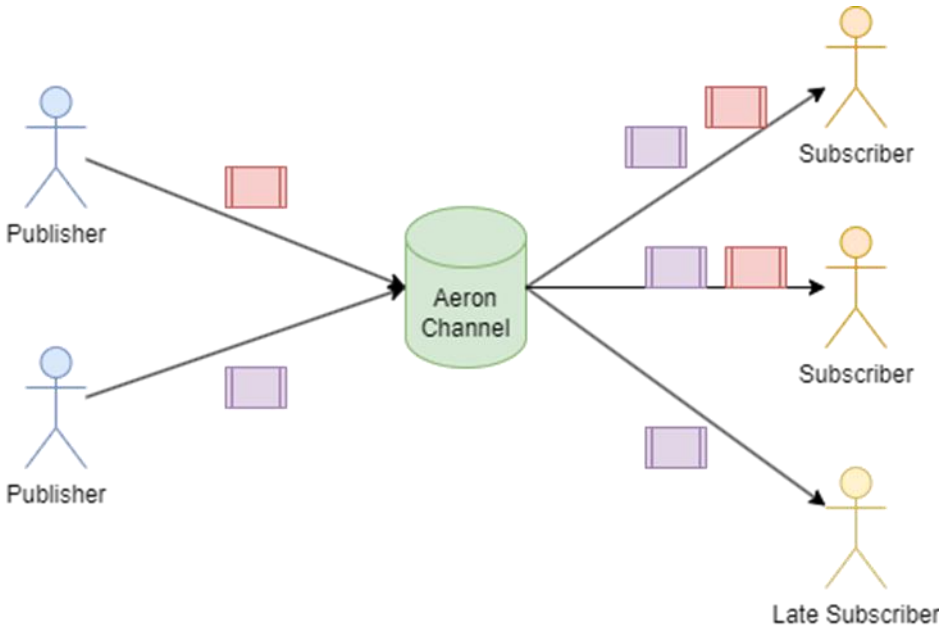


Fig.1 Aeron Mechanism Explanation

Now, I would like to explain our implementation strategy with the system design shown in Fig.2. First, from the main node, we can use JSCH to launch remote processes in other computing nodes. We backup our graph in the files stored in shared memory, so that multiple users can access the same shared graph, and each node only stores a portion of the graph, so as a result the graph is distributed and shared. For a fast performance, we introduce a cache in the process that reads the local graph data from the shared memory when the program started. Although the cache and the shared memory both contain the most up-to-date data, Aeron operations are in general quicker than shared memory operations so when one user modified the data the update will be sent to all other users working on the same graph using Aeron.

The two basic operations for a graph are vertex insertion and vertex access. For a vertex insertion or update operation, a process writes this modified vertex information through to the shared file in “/dev/shm” and at the same time send this vertex information to all other processes running on same computing node through Aeron. This caching scheme is a software implementation of write-through write-update policy, most found in snoop cache. As for vertex access, simply read from the cache in process memory. The procedure just mentioned is for the scenario that the vertex we want to access, or insert is in the same node as the user’s process. If the vertex is stored in another node, we use the TCP connection between master node and other nodes to inform and let the remote node perform local operations and send back the results.

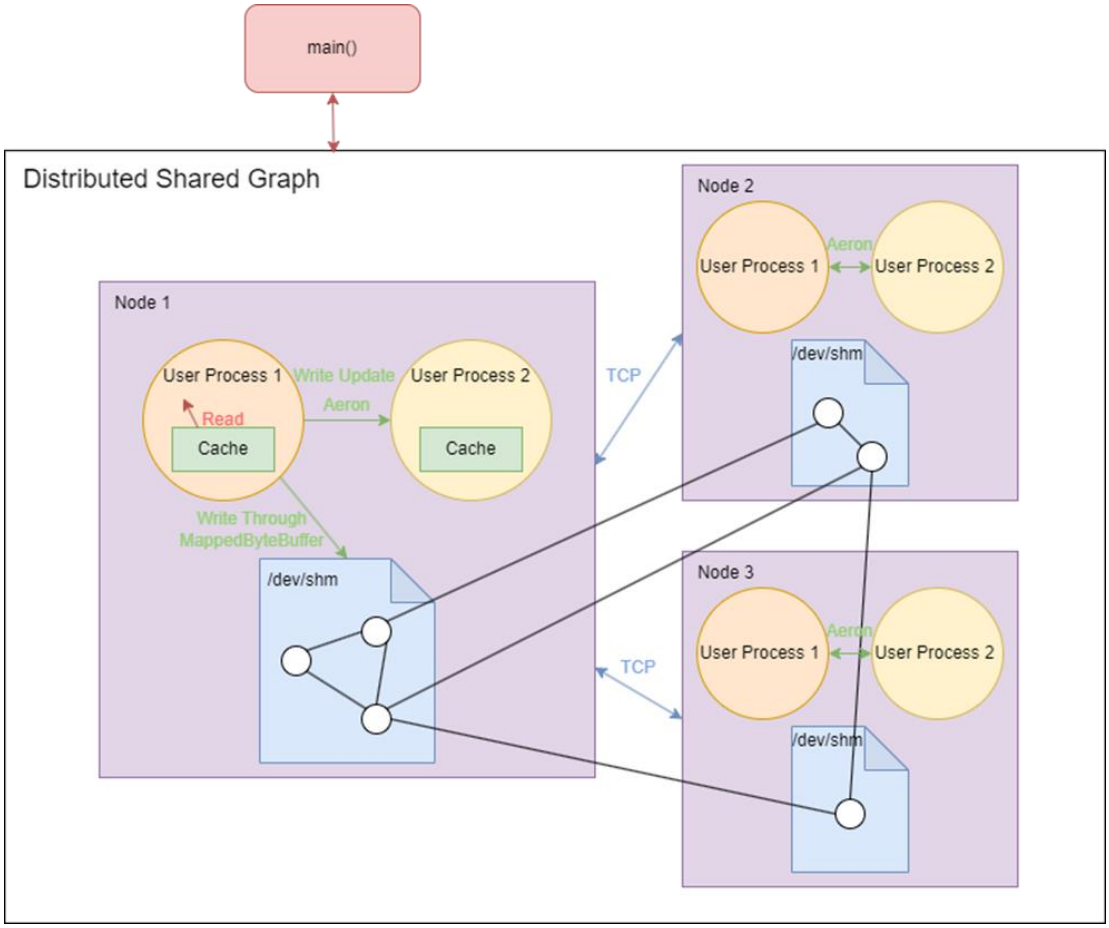


Fig.2 Distributed Shared Graph System Design from Autumn Quarter

Early this winter quarter, I implemented the whole system and compared the performance with the Hazelcast implementation (which serves as a baseline competitor) by Michael Robinson, an undergraduate student of our lab. When the graph size is small, the performance of DSG is quite good. However, when we start to test the program with graph of over 3000 vertices, the performance dropped significantly, and we found that both graph insertion and access operations are much slower than Hazelcast. Therefore, I started to test and analyze our implementation again and try to find solutions.

For graph insertion operation, as I mentioned before we used write through mechanism which guarantees both the cache and the shared memory contains the most up-to-date data. The shared memory operation is achieved by MappedByteBuffer in Java, and for the write operation we are only allowed to serialize an object and then write it to the shared memory. In each node, we are maintaining a collection of key-value-pairs where key is the vertex ID and value is the list of neighbors of that vertex. With the limitation of MappedByteBuffer, each time when we insert or update the vertex, we need to serialize and write the whole collection, this makes the performance poor when graph size increases. After realizing the bottleneck of shared memory

operations in Java, we decided to switch to write-back write-update policy, which means we only write back all data to shared memory when the program terminates. However, this introduces another problem, which is the data stored in shared memory is no longer up to date.

Previously, when each user starts the program, the program will first read the shared memory to get the data, and that data is guaranteed to be the same as data stored in other user's processes. After switching to write-back policy, if a running program from a user has modified the shared graph data but at the same time another user joined the same shared graph, the second user can only get the updated data from the first user. Therefore, I spent another two weeks to design and implement the graph initialization using Aeron. The detailed implementation can be found in Fig.3. When the program starts, it will first read the shared memory, but the initialized flag is set to false. Then an initialization request will be sent to the Aeron channel, and it will be received by all other users currently working on the same shared graph. They will reply with the most up-to-date data, and the new process will update its data correspondingly and set the initialized flag to be true. Once the initialized flag is set, later response messages will be ignored which avoided waste of resources to deal with all response messages. The above procedure happens in all related computing nodes.

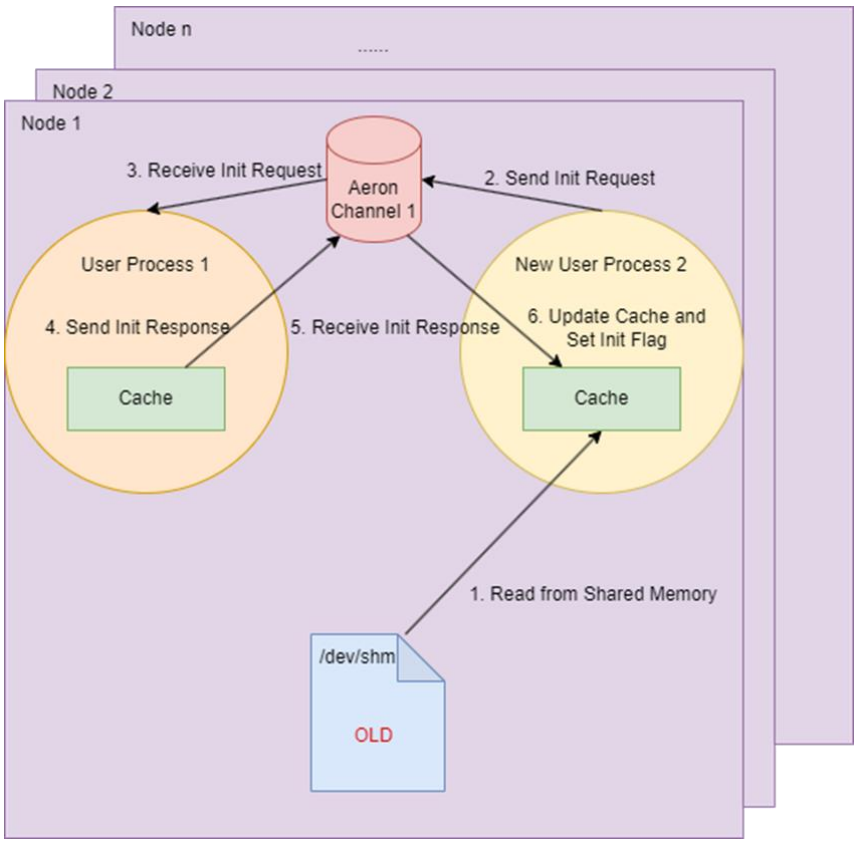


Fig.3 Graph Initialization Mechanism

As for graph access operation, I found that local access operations are quick while remote access operations become slow as graph size increases. To be more specific, when the vertex is stored in a remote node, the main program sends a message to remote node asking for the data and the remote node will reply by sending the list of neighbors of that vertex. When graph size is large, the number of neighbors of the vertices will also increase, so as a result the response message from the remote node is quite large. However, I during the last quarter when I was testing the performance of the communication protocols, TCP performed very well when dealing with large messages. After reviewing the test data, I realized that the connection established by JSCH may be different from a customized TCP connection established by Sockets. Therefore, I tested and confirmed that we need to establish a network of TCP connections between different computing nodes to achieve good performance. Based on the observation, I implemented the TCP network, and the graph access operation is optimized by using the customized TCP connections.

I finalized the DSG system in week 9. The finalized design is shown in Fig.4. The detailed performance analysis will be presented in the next section.

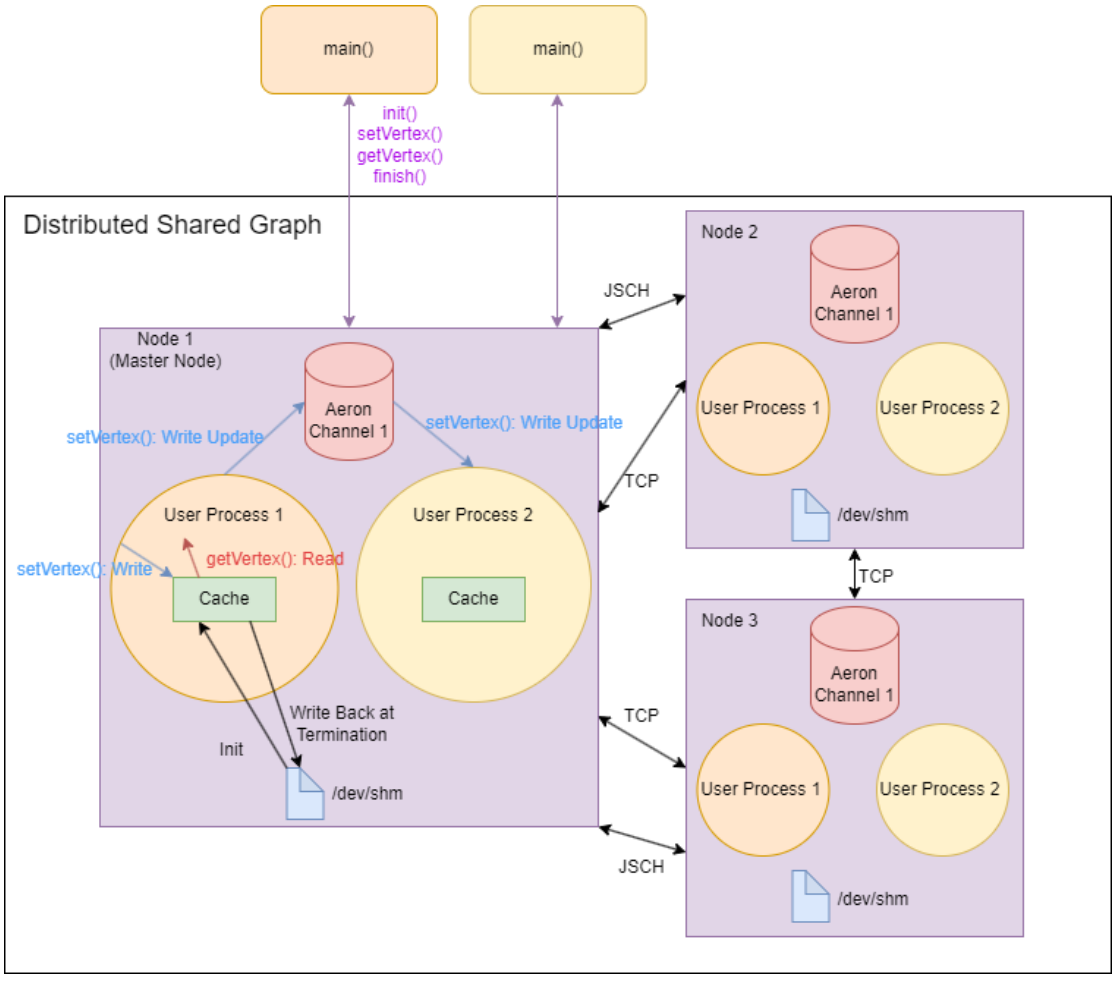


Fig.4 Finalized Distributed Shared Graph Implementation

For the last two weeks of the quarter, I carefully examined the MASS Java Core library and discussed how to add my implementation into our library. Initially, I tried to modify PlacesBase.java by changing the distributed array into a distributed HashMap. However, according to the inheritance structure shown in Fig.5, there are too many dependencies on PlacesBase so considering the time left I decided to only work on GraphPlaces.java. Currently GraphPlaces store vertices in a Vector data structure, and I'm planning to change it to HashMap and further facilitate relevant functions.

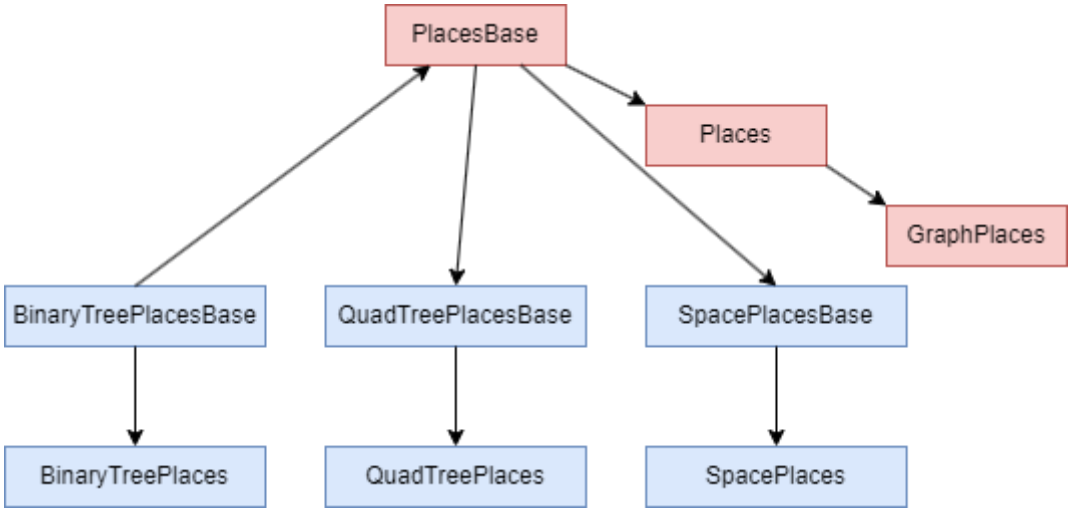


Fig.5 PlacesBase Dependencies

## Results

After finalizing the DSG implementation, I tested and compared the performance of vertex insertion and access operations with Hazelcast implementation with Michael Robinson, and the results are shown in Table.1 and Table.2.

Table.1 Vertex Insertion Total Time DSG vs. Hazelcast

| Vertex Insertion Total Time (ms) | Graph Size 3000 | Graph Size 5000 |
|---|---|---|
| DSG – 1 node | 1870 | 3796 |
| Hazelcast – 1 node | 809 | 1779 |
| DSG – 4 nodes | 2809 | 7448 |
| Hazelcast – 4 nodes | 7228 | 13722 |

Table.2 Vertex Access Total Time DSG vs. Hazelcast

| Vertex Access Total Time (ms) | Graph Size 3000 | Graph Size 5000 |
|---|---|---|
| DSG – 1 node | 4 | 9 |
| Hazelcast – 1 node | 540 | 1400 |
| DSG – 4 nodes | 2390 | 5274 |
| Hazelcast – 4 nodes | 3942 | 7800 |

From the results, we can see for vertex insertion DSG performs much better when we use multiple computing nodes, while for vertex access DSG outperformed Hazelcast in both single-node and multi-node scenarios.

After I implemented the DSG, Michael Robinson helped implementing many graph queries based on DSG and Hazelcast. The graph queries include getting neighbors of a vertex, getting parents of a vertex, getting grandparents of a vertex, getting vertex with minimum edges and getting highest degree vertex. Table.3 shows a complete performance comparison of DSG and Hazelcast when executing the same graph queries. The data is measured by Michael Robinson, and he is using a big graph with 100000 vertices and 3999366 edges.

Table.3 Graph Queries Performance Comparison DSG vs. Hazelcast

| Avg Time (ms) | # Computing Nodes | DSG | Hazelcast |
|---|---|---|---|
| Get Neighbors | 1 | 0.02 | 0.23 |
| | 4 | 1.03 | 0.94 |
| | 8 | 0.71 | 1.09 |
| Get Parents | 1 | 9.83 | 31.53 |
| | 4 | 6.39 | 29.65 |
| | 8 | 5.48 | 16.41 |
| Get GrandParents | 1 | 78.2 | 118.14 |
| | 4 | 28.49 | 101.73 |
| | 8 | 19.15 | 57.71 |
| Get Min Edges | 1 | 2.84 | 24.9 |
| | 4 | 2.03 | 22.96 |
| | 8 | 1.82 | 11.01 |
| Get Highest Degree | 1 | 5.7 | 54.05 |
| | 4 | 3.88 | 49.47 |
| | 8 | 3.42 | 24.6 |

From Table.3, we can see that DSG performs better than Hazelcast in general. The above results show that we have successfully developed a high-performance distributed shared graph system.

## Spring Quarter Plan

Based on the current progress of the project, I proposed the spring quarter plan as shown in Table.4.

Table.4 Spring Quarter Plan

| Quarter | Week | Plan | Deliverables |
|---|---|---|---|
| Spring 2024 | 1-2 | Preliminary MASS GraphPlaces implementation on top of our DSG implementation. | A preliminary version of MASS GraphPlaces. |
| | 3-4 | Final version of MASS GraphPlaces with all specifications achieved. | Final version of MASS with DSG integrated. |
| | 5-6 | Write white paper, co-author an IEEE Cluser 2024 conference paper | Draft white paper |
| | 7-8 | Write white paper, prepare for final defense. | Draft presentation for final defense. |
| | 9-10 | Prepare for final defense. | Presentation for final defense. |
| | 11 | Complete white paper | Finalized white paper. |

## Summary

Overall, during this quarter we encountered performance problems when comparing the initial version of DSG with Hazelcast but we managed to solve the problem and proposed a better designed system. According to the performance comparison with Hazelcast, we have successfully developed a high-performance distributed shared graph. Also, I started to add my implementation into MASS, and I decided to only focus on GraphPlaces in the remaining time. Although according to the original project plan, I'm a little behind the schedule, I'm still confident that the goals can be met by devoting more effort to the project.

## References

[1] "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory", Accessed on: July 13, 2023. Available at: http://faculty.washington.edu/mfukuda/papers/biggraphs20.pdf
[2] "MapReduce", Accessed on: July 18, 2023. Available at: https://hadoop.apache.org/
[3] "Spark", Accessed on: July 18, 2023. Available at: https://spark.apache.org/
[4] "Storm", Accessed on: July 18, 2023. Available at: https://storm.apache.org/
[5] "Neo4j", Accessed on: July 18, 2023. Available at: https://neo4j.com/
[6] "Pipelining Graph Construction and Agent-based Computation over Distributed Memory", Accessed on: July 13, 2023. Available at:

http://faculty.washington.edu/mfukuda/papers/biggraphs22.pdf
[7] “Hazelcast”, Accessed on: July 18, 2023. Available at: https://hazelcast.com/
[8] “Redis”, Accessed on: July 18, 2023. Available at: https://redis.com/
[9] “Oracle Coherence”, Accessed on: July 18, 2023. Available at: https://www.oracle.com/java/coherence/
[10] “AWS SimSpace Weaver”, Accessed on: July 18, 2023. Available at: https://aws.amazon.com/simspaceweaver/
[11] “JCraft JSch - Java Secure Channel”, Accessed on: 10 March 2024. Available at: http://www.jcraft.com/jsch/
[12] “Class MappedByteBuffer”, Accessed on: July 18, 2023. Available at: https://docs.oracle.com/javase/8/docs/api/java/nio/MappedByteBuffer.html
[13] “Aeron”, Accessed on: December 11, 2023. Available at: https://aeron.io/

## Appendix

I uploaded all my DSG code to mass_java_core bitbucket repository, under the branch “chrisma/develop”. The DSG core library and two testing programs are located under the directory “dsg”. I created a guide to compile and run DSG, and the link is provided below.

DSG Guide Link:
https://docs.google.com/document/d/1HeCqWunFmAAbIXNma9goPufWY8wdtwWd/edit?usp=sharing&ouid=116473289907246068929&rtpof=true&sd=true

Link to DSG Code:
https://bitbucket.org/mass_library_developers/mass_java_core/src/9586b4d0b7f9bdcaa59d620cdb48876382132cf5/dsg/?at=chrisma%2Fdevelop