MASS JAVA LIBRARY TOWARDS IT'S USE FOR A GRAPH DATABASE SYSTEM

ATUL AHIRE

Term Report - Winter Quarter

Master of Science in Computer Science & Software Engineering

University of Washington

March 26, 2025

Project Committee

I. Introduction

In this quarter's work, I explored optimizations for the MASS Java framework to enhance its capabilities as a graph database system. The Multi-Agent Spatial Simulation (MASS) library provides a powerful foundation for distributed computing, but faces significant challenges when scaling graph operations across computing clusters. Initially, my research focused on addressing the inefficient vertex distribution mechanisms in MASS. The standard round-robin approach distributes vertices across nodes without considering graph topology, leading to excessive remote communication during traversals. To solve this, I implemented two advanced partitioning strategies: a Hazelcast-inspired partitioning logic that creates evenly distributed fixed-size partitions, and a METIS-based approach that minimizes edge cuts between partitions. Benchmarking these partitioning strategies revealed an interesting phenomenon. While the METIS-based partitioning successfully reduced remote agent migration requests by a significant margin as shown in Figure 1, this reduction did not translate to proportional performance improvements in applications like Triangle Counting. This unexpected outcome triggered a deeper analysis of the system's performance characteristics.

The investigation revealed that the bottleneck wasn't just in where vertices were placed, but in how agents migrated between nodes. The current MASS implementation faces two critical limitations: (1) it creates and destroys threads for each migration cycle, adding substantial overhead, and (2) it relies on synchronous TCP communication that becomes inefficient with increasing cluster size and graph complexity. To address these foundational issues, I developed a new communication infrastructure for MASS based on Aeron, a high-performance UDP messaging framework. This approach introduces a dual communication pattern: a master-worker structure for control operations and global synchronization, and a peer-to-peer mesh network optimized for high-throughput agent migrations. The solution implements custom reliability mechanisms on top of UDP through acknowledgment protocols and retry logic, while leveraging Aeron's efficient zero-copy operations and message fragmentation capabilities.

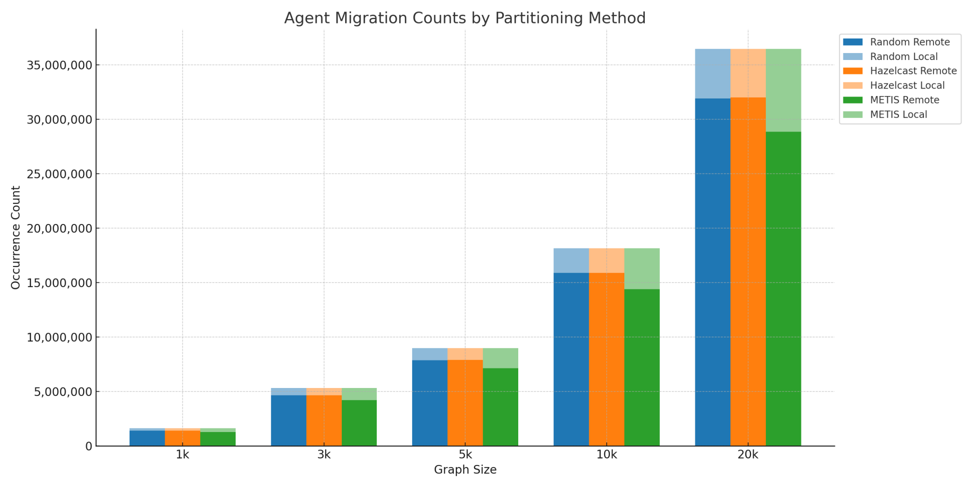

Figure 1: Agent migrations metrics with different partitioning techniques.

II. Challenges

While Aeron-based UDP communication offers significant performance benefits for MASS, it also introduces several challenges.

Key Challenges in using Aeron in MASS

1. **Message Reliability**

   Unlike TCP, UDP provides no built-in guarantee of message delivery. When a message is sent via UDP, there is no automatic confirmation that it has reached its destination. This unreliability is particularly problematic for MASS, where missed messages could result in lost agent migrations, incomplete graph traversals, or corrupted distributed state. The system must function correctly even when packets are dropped due to network congestion or temporarily unavailable nodes, making reliability a fundamental challenge that needed addressing at the application level. We implemented a comprehensive acknowledgment framework using CountDownLatch mechanisms and timeout-based retries. Each message generates a unique ID tracked in a ConcurrentHashMap, with acknowledgments from receivers triggering latch countdowns. Messages without acknowledgments within configurable timeframes trigger automatic retransmission, with exponential backoff to prevent network congestion during periods of instability.

2. **Message Ordering**

   UDP does not preserve message order, which introduces significant complexity for operations that depend on sequential processing. Messages sent in a specific sequence may arrive at their destination in a completely different order. This issue becomes particularly acute in graph operations where the order of updates matters, such as multi-step migrations or modifications to connected vertices. Without ordering guarantees, the system could enter inconsistent states where, for example, a vertex is modified after it's been deleted, or an agent moves to a location that no longer exists in the expected form. Our solution involved implementing logical timestamps and message sequencing at the application layer. For critical operations like agent migration, we developed a transaction-based approach where migrations occur in atomic operations with rollback capabilities should any part of the sequence fail. Additionally, we implemented message buffering at the receiver side to reorder messages before processing when sequence integrity is essential.

3. **Buffer Management**

   Aeron's high-performance approach relies on off-heap memory management using direct byte buffers that operate outside Java's garbage collection. While this approach offers significant performance benefits by avoiding GC pauses and memory copies, it introduces complex memory management requirements. Large agent migrations can require buffers exceeding default allocation sizes, potentially leading to out-of-memory conditions or excessive fragmentation. Since these buffers are not automatically managed by the JVM, improper handling can result in memory leaks that gradually degrade system performance and stability. We encountered significant challenges when handling large agent migrations, where buffers exceeded default sizes. Our implementation dynamically resizes buffers based on message payload requirements while implementing safeguards against excessive memory consumption.

4. **Architectural Refactoring**

   The existing MASS framework was built around a synchronous communication model where operations typically block until completion. Aeron, by contrast, uses a fundamentally asynchronous approach based on non-blocking I/O and event notifications. This paradigm shift affects core assumptions throughout the codebase. Operations that previously completed in a predictable sequence now occur asynchronously, potentially in parallel, creating race conditions and state management challenges. Thread synchronization points, error handling strategies, and state maintenance all require reconsideration when moving from synchronous to asynchronous processing. We needed to redesign core components to operate in an event-driven manner, implementing callback chains and completion handlers throughout the codebase. This refactoring extended beyond just communication layers, affecting thread management, agent lifecycle handling, and Place operations.

III. Proposed Solution

The proposed solution transforms MASS's communication infrastructure from TCP to a high-performance Aeron-based UDP implementation using a dual communication pattern approach as shown in Figure 2. The solution implements a hybrid communication approach to address the inefficiencies in MASS agent migration for large graph applications:

1. **Master-Worker Pattern** Master-Worker Pattern forms the backbone for control operations. Phase 1 establishes the Master Node (PID=0) through MASS.init(), which becomes the foundation for the Master-Worker Pattern. In phase 2, the Master Node uses SSH to launch MProcess instances on remote nodes, establishing the "worker" components of the Master-Worker Pattern. These remote processes will be directed by the Master Node, creating the hierarchical structure necessary for maintaining system-wide consistency. Dedicated Aeron streams in Phase 3 connect the master to each worker node, creating direct channels for operations that require coordination, such as Graph initialization , synchronization points, and global state updates. This hierarchical structure provides the necessary oversight for maintaining consistency across the distributed system.

2. **Peer-to-Peer Pattern** Peer-to-Peer Pattern addresses the high-volume data transfer needs of the system. Implemented in the ExchangeHelper class, this pattern creates a mesh network allowing any node to communicate directly with any other node without routing through the master as shown in Phase 4. This direct communication approach significantly

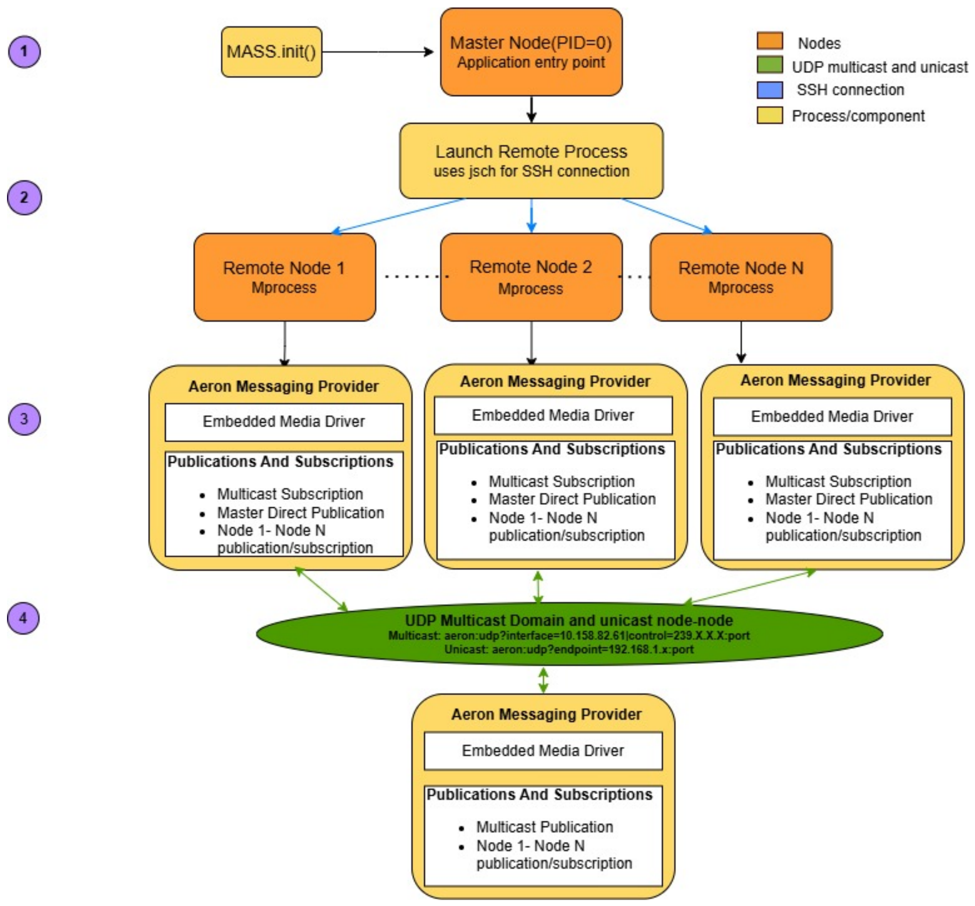
1
MASS.init()
Master Node(PID=0)
Application entry point
Nodes
UDP multicast and unicast
SSH connection
Process/component
Launch Remote Process
uses jsch for SSH connection
2
Remote Node 1
Mprocess
Remote Node 2
Mprocess
Remote Node N
Mprocess
3
Aeron Messaging Provider
Embedded Media Driver
Publications And Subscriptions
Multicast Subscription
Master Direct Publication
Node 1- Node N publication/subscription
Aeron Messaging Provider
Embedded Media Driver
Publications And Subscriptions
Multicast Subscription
Master Direct Publication
Node 1- Node N publication/subscription
Aeron Messaging Provider
Embedded Media Driver
Publications And Subscriptions
Multicast Subscription
Master Direct Publication
Node 1- Node N publication/subscription
4
UDP Multicast Domain and unicast node-node
Multicast: aeron:udp?interface=10.158.82.61|control=239.X.X.X:port
Unicast: aeron:udp?endpoint=192.168.1.x:port
Aeron Messaging Provider
Embedded Media Driver
Publications And Subscriptions
Multicast Publication
Node 1- Node N publication/subscription

Figure 2: Aeron Integration in MASS.

reduces latency and eliminates potential bottlenecks at the master node during intensive operations like agent migrations. Each node maintains direct Aeron publications to every other node in the cluster, creating a fully connected communication mesh.

The solution leverages Aeron's high-performance capabilities including zero-copy operations, efficient message fragmentation, and back-pressure handling. It implements custom reliability mechanisms on top of UDP through acknowledgment protocols and retry logic. For agent migrations specifically, the system dynamically sizes buffers to handle large transfers (up to 16MB per message) and employs concurrent processing with dedicated receiver threads.

This dual-pattern approach combines structured control flow with high-performance data transfer, making it particularly well-suited for large-scale graph applications where efficient agent migration is critical to overall performance.

IV. Source Code

1. Master-Worker Communication pattern

The master-worker communication model in the MASS framework represents a significant architectural enhancement, establishing dedicated channels between the master node (rank 0) and worker nodes (ranks > 0) using Aeron's high-performance messaging capabilities. Let's examine the key components of this implementation.

Stream Channel Initialization

Listing 1 establishes dedicated stream IDs for the Master-Worker communication pattern. Line 2 defines the `MASTER_TO_WORKER_STREAM_ID` (9001) for messages flowing from the master to worker nodes, while line 3 defines the `WORKER_TO_MASTER_STREAM_ID` (9002) for messages traveling in the opposite direction. These separate channels create a clean separation between the two communication flows.

```java
1  // Stream IDs for dedicated master-worker communication
2  private static final int MASTER_TO_WORKER_STREAM_ID = 9001;
3  private static final int WORKER_TO_MASTER_STREAM_ID = 9002;
```

Listing 1: Streams For Master-Worker Communication

Listing 2 establishes the master-worker communication channels based on the node's role in the system. For the master node (lines 2-11), it adds a publication for sending messages to workers (line 4) and a subscription for receiving messages from workers (line 5). It then initializes a subscriber thread (lines 8-10) to handle incoming messages from workers. For worker nodes (lines 12-21), the configuration is reversed - they add a publication for sending messages to the master (line 14) and a subscription for receiving messages from the master (line 15), with a corresponding subscriber thread (lines 18-20).

```java
1  // Initialize master-worker communication channels
2  if (MASS.getMyPid() == 0) {
3      // Master node setup
4      masterToWorkerPub = aeron.addPublication(url +
           AERON_FLOW_CONTROL_STRATEGY, MASTER_TO_WORKER_STREAM_ID);
5      workerToMasterSub = aeron.addSubscription(url,
           WORKER_TO_MASTER_STREAM_ID);
6
7      // Start subscriber for master node
8      masterWorkerSubscriber = new Subscriber("MASTER_WORKER",
9              receiveMasterWorkerMessage(true), FRAGMENT_COUNT_LIMIT,
                   running, idle,
10             workerToMasterSub);
11     masterWorkerSubscriber.start();
12 } else {
13     // Worker node setup
14     workerToMasterPub = aeron.addPublication(url +
           AERON_FLOW_CONTROL_STRATEGY, WORKER_TO_MASTER_STREAM_ID);
15     masterToWorkerSub = aeron.addSubscription(url,
           MASTER_TO_WORKER_STREAM_ID);
16
17     // Start subscriber for worker node
18     masterWorkerSubscriber = new Subscriber("MASTER_WORKER",
19             receiveMasterWorkerMessage(false), FRAGMENT_COUNT_LIMIT,
                   running, idle,
                masterToWorkerSub);
        masterWorkerSubscriber.start();
}
```

Listing 2: Initialize master-worker communication channels

Message Sending

Listing 3 implements the message sending functionality from master to worker nodes. Lines 2-4 ensure this method is only called by the master node. The method takes a destination worker address and wraps the message content in a `MasterWorkerMessage` container (line 13), which is then serialized (line 14). Lines 16-26 handle the actual transmission of the message using Aeron's publication API, with a retry mechanism that uses the idle strategy if the initial send attempt fails. This ensures reliable delivery of control messages from the master to workers.

```java
1  public <T> void masterSendToWorkerNode(MASSMessage<Serializable> message) {
2      if (MASS.getMyPid() != 0) {
3          throw new IllegalStateException("Only the master node can call masterSendToWorkerNode");
4      }
5
6      int workerRank = message.getDestinationAddress();
7      Serializable msgContent = message.getMessage();
8
9      if (!(msgContent instanceof Message)) {
10         throw new IllegalArgumentException("Message content must be of type Message");
11     }
12
13     MasterWorkerMessage wrapper = new MasterWorkerMessage(0, (Message) msgContent, workerRank);
14     byte[] data = SerializationUtils.serialize(wrapper);
15
16     // Use the master-to-worker publication
17     UnsafeBuffer buffer = new UnsafeBuffer(BufferUtil.allocateDirectAligned(data.length, 64));
18     buffer.putBytes(0, data);
19
20     long result;
21     do {
22         result = masterToWorkerPub.offer(buffer, 0, data.length);
23         if (result < 0) {
24             idle.idle();
25         }
26     } while (result < 0);
27 }
```

Listing 3: Send Message

Similarly, for worker-to-master communication, the `workerSendToMasterNode` method ensures messages are sent on the appropriate channel.

Message Reception

Listing 4 handles message reception for both master and worker nodes. When the master receives a message (lines 8-21), it stores the message in a map and signals any waiting threads using a `CountDownLatch`. When a worker receives a message (lines 22-39), it first checks if the message is intended for this specific worker or is a broadcast message (line 28), then processes it similarly by storing the message and signaling any waiting threads.

```
1  private FragmentHandler receiveMasterWorkerMessage(boolean isMaster) {
2      return (buffer, offset, length, header) -> {
3          try {
4              byte[] data = new byte[length];
5              buffer.getBytes(offset, data);
6              MasterWorkerMessage message = SerializationUtils.
                   deserialize(data);
7
8              if (isMaster) {
9                  // Master receiving from a worker node
10                 int senderRank = message.getSenderRank();
11
12                 if (message.getMessage() instanceof Message) {
13                     // Store message
14                     masterWorkerMessages.put(senderRank, (Message)
                           message.getMessage());
15
16                     // Signal waiting thread if any
17                     CountDownLatch latch = masterWorkerLatches.get(
                           senderRank);
18                     if (latch != null) {
19                          latch.countDown();
20                     }
21                 }
22             } else {
23                 // Worker node receiving from master
24                 int destRank = message.getDestinationRank();
25                 int myRank = MASS.getMyPid();
26
27                 // Only process messages targeted at this node or
                       broadcasts
28                 if (destRank == -1 || destRank == myRank) {
29                     if (message.getMessage() instanceof Message) {
30                         // Store message
31                         masterWorkerMessages.put(0, (Message) message.
                               getMessage());
32
33                         // Signal waiting thread if any
```

```java
                        CountDownLatch latch = masterWorkerLatches.get(0);
                        if (latch != null) {
                            latch.countDown();
                        }
                    }
                }
            }
        } catch (Exception e) {
            MASSBase.getLogger().error("Error processing master-worker message: " + e);
        }
    };
}
```

Listing 4: Recieve Message

Synchronous Operation Support

A key feature of this implementation is its ability to support synchronous operations through a `CountDownLatch` mechanism. Methods like `waitForWorkerNodeMessage` allow the caller to block until receiving a message from a specific node:

```java
public Message waitForWorkerNodeMessage(int workerRank, long timeoutMs) {
    if (MASS.getMyPid() != 0) {
        throw new IllegalStateException("Only the master node can call waitForWorkerNodeMessage");
    }

    // Check if we already have a message from this node
    if (masterWorkerMessages.containsKey(workerRank)) {
        return masterWorkerMessages.remove(workerRank);
    }

    // Set up a latch for this node
    CountDownLatch latch = new CountDownLatch(1);
    masterWorkerLatches.put(workerRank, latch);

    try {
        boolean received = latch.await(timeoutMs, TimeUnit.MILLISECONDS);

        // Clean up
        masterWorkerLatches.remove(workerRank);

        if (received) {
            return masterWorkerMessages.remove(workerRank);
        } else {
            MASSBase.getLogger().error("Timeout waiting for message from worker " + workerRank);
            return null;
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return null;
    }
}
```

Listing 5: Synchronous Message Support

This architecture provides a dedicated channel for control operations separate from the data plane, allowing for more structured communication patterns between the master and worker nodes while keeping this traffic isolated from the framework's other messaging activities.

2. Peer-to-Peer Communication

This communication is primarily facilitated through the `ExchangeHelper` class, which provides a reliable mechanism for nodes to send and receive messages directly with each other.

Connection Establishment

The communication begins with establishing connections between all nodes in the cluster. Here's how the connection is established:

This method creates an array of Publications (nodePubs) for sending messages to each node (lines 27 in Listing 6). Then Sets up a Subscription for receiving messages (lines 17 in Listing 6). Finally, starts a receiver thread to process incoming messages (lines 48 in Listing 6)

```java
1  
2  public void establishConnection(int size, int rank, Vector<String> hosts, int port) {
3      MASS.getLogger().debug("Initializing Aeron connections to {} nodes", size);
4  
5      nodePubs = new Publication[size];
6  
7      // Get the Aeron instance from MASS messaging provider
8      aeron = MASS.getMessagingProvider().getAeronInstance();
9      if (aeron == null) {
10         MASS.getLogger().error("Aeron instance not available - MASS messaging not properly initialized");
11         System.exit(-1);
12     }
13 
14     // Set up subscription to receive messages first
15     String localEndpoint = hosts.get(rank) + ":" + port;
16     String subChannel = "aeron:udp?endpoint=" + localEndpoint;
17     exchangeSubscription = aeron.addSubscription(subChannel, EXCHANGE_STREAM_ID);
```

```

    // Create publications to all other nodes
    for (int i = 0; i < size; i++) {
        if (i == rank)
            continue; // Skip self

        String endpoint = hosts.get(i) + ":" + port;
        String channel = "aeron:udp?endpoint=" + endpoint;

        nodePubs[i] = aeron.addPublication(channel, EXCHANGE_STREAM_ID)
           ;

        // Implement retry logic for connection
        int maxRetries = 5;
        boolean connected = false;

        for (int retry = 0; retry < maxRetries; retry++) {
            // Check for connection with a shorter timeout per attempt
            long deadline = clock.time() + 5000; // 5 seconds per
               attempt
            while (!nodePubs[i].isConnected() && clock.time() <
               deadline) {
                idleStrategy.idle();
            }

            if (nodePubs[i].isConnected()) {
                connected = true;
                break;
            }
        }
    }

    // Start receiver thread
    receiverThread = new Thread(() -> this.receiveMessages());
    receiverThread.setDaemon(true);
    receiverThread.start();
}
```

Listing 6: Establish Connection

Sending Messages

The sendMessage method handles sending a message to a specific node:

```

public void sendMessage(int rank, Message message) {
    MASS.getLogger().debug("Sending message to rank: {}", rank);

    // Add sender rank to message
    message.setSenderRank(MASSBase.getMyPid());

    // Serialize the message
    byte[] serializedMsg = SerializationUtils.serialize(message);

    // Ensure we have a valid publication for this rank
    if (nodePubs[rank] == null || !nodePubs[rank].isConnected()) {
        MASS.getLogger().error("No valid connection to node " + rank);
        return;
    }

    // Copy serialized message to the send buffer
    sendBuffer.putBytes(0, serializedMsg);

    // Send the message
    long result;
    do {
        result = nodePubs[rank].offer(sendBuffer, 0, serializedMsg.
            length);
        if (result < 0) {
            idleStrategy.idle();
        }
    } while (result < 0);

    MASS.getLogger().debug("Message sent to rank: {}", rank);
}
```

Listing 7: Send Message to Node

V. Performance Benchmark

After integrating the Aeron UDP communication into MASS, I performed performance benchmarking using a triangle counting application on graphs with 5 and 1000 vertices.

In Figure 3, the performance data for triangle counting shows that Aeron-based UDP communication significantly underperforms compared to traditional TCP, counter to expectations for a high-performance messaging system. While Aeron should theoretically offer better throughput and lower latency than TCP, I feel that the implementation in ExchangeHelper appears to introduce bottlenecks that negate these advantages. The primary culprits are likely the synchronous messaging pattern with acknowledgment waits, which create overhead that compounds across distributed nodes. This explains why execution times with Aeron (802ms and 1123ms for 4 and 8 nodes) are much higher than with TCP (474ms and 412ms), despite Aeron's architectural advantages.

Also, Triangle counting with 1k nodes is failing with the exception `IllegalArgumentException: message exceeds maxMessageLength of 16777216, length=32395426` indicates that during agent migration (specifically at iteration 1 when the agent count jumps dramatically from 46,740 to 878,407), the system attempts to send a message approximately 32MB in size, which exceeds Aeron's configured maximum message length of 16MB. This likely happens when

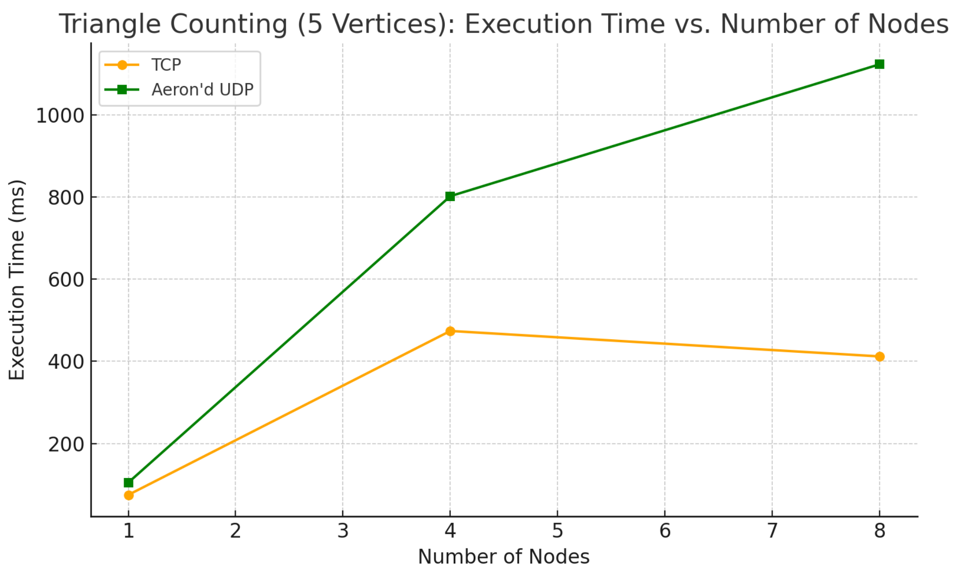

Figure 3: Performance benchmark for triangle counting on a graph with 5 vertices.

numerous agents need to migrate between nodes simultaneously, forcing the system to serialize a large collection of agent objects into a single message.

```
Import complete
Import time: 2.176 s
Number of Vertices is : 1000
Go! Graph
*** iteration = 0************
*** Total Agents = 46740************
*** iteration = 1************
Exception in thread "Thread-20" java.lang.IllegalArgumentException:
    message exceeds maxMessageLength of 16777216, length=32395426
        at io.aeron.Publication.checkMaxMessageLength(Publication.java
            :666)
        at io.aeron.ConcurrentPublication.offer(ConcurrentPublication.
            java:125)
        at io.aeron.Publication.offer(Publication.java:426)
        at edu.uw.bothell.css.dsl.MASS.ExchangeHelper.sendMessageToRank
            (ExchangeHelper.java:503)
        at edu.uw.bothell.css.dsl.MASS.ExchangeHelper.sendMessage(
            ExchangeHelper.java:427)
        at edu.uw.bothell.css.dsl.MASS.
            AgentsBase$ProcessAgentMigrationRequest.run(AgentsBase.java
            :3478)
*** Total Agents = 878407************
}
```

Listing 8: Triangle counting- 1K

To address both the performance issues and the message size limitations in ExchangeHelper, I will have to replace the synchronous approach with a non-blocking asynchronous design and handle large messages streaming using chunks rather than sending whole message.

VI. Project Plan

Quarter	Week	Plan	Deliverables
Spring 2025	1-2	Handle agent migration for large graphs and fine tune performance of aeron based communication.	
	3-4	Benchmark graph applications and database queries performance.	
	5-6	Work on the draft of the white paper and Create final defense slides.	Final defense slides created.
	7-8	Edit the white paper based on the committee recommendation.	White paper revised based on committee feedback.
	9-10	Finalize program deliverables.	Program deliverables finalized.
	11	Finalize the white paper. Update the system documentation and user guides to reflect the changes and provide instructions on utilizing the extended features.	White paper finalized. System documentation and user guides updated.

VII. Conclusion

This Quarter, I explored the implementation and optimization of Aeron based UDP communication in the MASS (Multi-Agent Spatial Simulation) framework, with a specific focus on enhancing the agent migration performance. While Aeron offers theoretical advantages over traditional TCP communication with its high-performance design and efficient memory management, our performance analysis revealed significant challenges in practical implementation. The triangle counting algorithm benchmark demonstrated that the current Aeron implementation actually underperforms compared to the original TCP version, primarily due to synchronous messaging patterns, inefficient serialization, and acknowledgment overhead that negate Aeron's inherent advantages.

The most critical limitation encountered was the message size constraint, as evidenced by the failure when processing a 1,000-vertex graph where message sizes exceeded Aeron's configured maximum. This highlighted the need for more sophisticated approaches to agent migration and data transmission in distributed simulations. The proposed optimizations include implementing message fragmentation, asynchronous processing, efficient serialization, proper Aeron configuration, and adaptive flow control mechanisms. By adopting these strategies, the MASS framework can better leverage Aeron's performance capabilities while handling large-scale simulations that generate substantial inter-node communication.

In the next quarter, my focus will be on completing the integration of Aeron into MASS's graph database engine, fine-tuning its performance under real-world scenarios, and preparing for final submission and defense of this project.

VIII. Appendix

Complete Source Code

Complete source code of this implementation can be found in my branch:
Branch:
`mass_library_developers / mass_java_core`
Pull Request:
#51: aahire-metis-implementation — Bitbucket