Fault Tolerance in MASS Graph Computing: Mid-Computation Checkpointing and Serialization Benchmarks

Zewen Gong

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering
University of Washington
2026

Committee:
Dr. Munehiro Fukuda ,Committee Chair
Dr. Kelvin Sung, Committee Member
Dr. Robert Dimpsey, Committee Member

University of Washington

Abstract

MASS Java Fault Tolerance and Graph Streaming

Zewen Gong

Chair of the Supervisory Committee:

Dr. Munehiro Fukuda

Department of Computing & Software Systems Faculty

Abstract — Distributed graph computing frameworks are vulnerable to node failures that can invalidate hours of computation. This paper presents a two-semester investigation into fault tolerance for the Multi-Agent Spatial Simulation (MASS) Java framework. In the first phase, a ring-topology backup architecture was designed and validated, enabling each compute node to persist graph state to local disk and transfer backup files to its successor in the ring. In the second phase, this work evaluates whether mid-computation checkpointing is feasible in MASS, analyzes three approaches to agent state serialization, and benchmarks Java native serialization against Kryo serialization for graph state persistence. Benchmark results across graphs of 750, 1250, and 2500 vertices demonstrate that Kryo achieves up to 3.48× smaller file sizes, 2.13× faster writes, and 3.42× faster reads compared to Java native serialization. The implemented fault tolerance subsystem is shown to be robust and extensible, and the paper concludes by outlining how the disk backup infrastructure lays the groundwork for extending MASS graph pipelining to graphs of arbitrary size.

Keywords — fault tolerance, distributed computing, graph computing, serialization, MASS framework, Kryo, checkpointing, graph pipelining

I. Introduction

Distributed graph computing frameworks partition large graphs across multiple compute nodes and execute iterative algorithms in parallel. This model is well-suited to problems such as triangle counting, shortest path search, and connected components analysis, where the graph structure determines the pattern of inter-node communication. However, this architecture is inherently vulnerable to node failures. A single node failure can corrupt the global computation state, rendering all prior work unrecoverable and forcing a full restart from the beginning.

The Multi-Agent Spatial Simulation (MASS) Java framework [4] is a distributed computing platform developed at the University of Washington Bothell. MASS supports both place-based and agent-based computation over graphs, using a message-passing model in which agents migrate between graph nodes (places) and perform local computation on arrival. Unlike data-parallel frameworks such as Apache Flink [1], MASS does not provide built-in fault tolerance mechanisms. When a node fails during a long-running graph algorithm, the entire computation must be restarted from scratch.

This paper addresses the problem of fault tolerance in MASS through two phases of investigation. The first phase, conducted in the preceding semester, focused on designing and implementing a ring-topology backup architecture. In this architecture, each node persists its local graph state to disk and transfers backup files to its successor node in a ring, ensuring that each node's data is replicated on at least one other machine. The second phase, reported here, investigates whether mid-computation checkpointing is feasible in MASS, with particular attention to the challenge of serializing agent state alongside graph state.

The contributions of this paper are as follows:

- A complete fault-tolerance subsystem for MASS, including disk-based graph state persistence, ring-topology file transfer, and a clean public API (GraphPlaces.backup()) callable between computational supersteps. GraphPlaces is the MASS class that holds a distributed graph instance, including all related objects and associated metadata.
- A systematic analysis of three approaches to agent state serialization in MASS, with a discussion of why a universal solution is infeasible and a proposal for an application-defined checkpoint interface.
- An empirical benchmark comparing Java native serialization and Kryo serialization for graph state persistence across multiple graph sizes, demonstrating consistent and significant performance advantages for Kryo.

- A discussion of how the implemented fault tolerance infrastructure extends the MASS graph pipelining work of Hong and Fukuda [2] toward graphs of arbitrary size.

The remainder of this paper is organized as follows. Section II reviews related work. Section III describes the ring-topology backup architecture developed in the first phase. Section IV analyzes the three approaches to agent serialization. Section V presents the serialization benchmark methodology and results. Section VI discusses the implications of these results. Section VII concludes and outlines future work.

II. Related Work

Fault tolerance in distributed computing frameworks. Apache Flink uses a distributed snapshot algorithm based on Chandy-Lamport [1, 3], inserting barrier messages into data streams to trigger consistent checkpoints across all operators. This approach is deeply integrated into Flink's execution model and requires that the framework control the flow of data between operators. Applying a similar technique to MASS would require modifications to the agent migration and message-passing subsystem, since agent movement is not a controlled data flow but an autonomous behavior triggered by application logic. This fundamental difference motivates the application-defined checkpoint interface proposed in Section IV.

Graph pipelining in MASS. Hong and Fukuda proposed pipelining graph construction and agent-based computation over distributed memory [2]. In their design, a large graph file is read in batches; while agents compute over the current batch, a loading thread reads the next batch in parallel. Agents that reach an "incomplete" vertex — one whose edges span beyond the current batch boundary — are suspended and automatically resumed when the next batch becomes available. Evaluated on a 140MB graph with 25K vertices, their single-node pipelining implementation of connected components achieved a 7.7× speedup over non-pipelining execution, and scaled to 8.3× with 24 cluster nodes. A key limitation of their implementation is that the entire graph is eventually held in distributed memory: pipelining reduces construction overhead but does not remove the memory capacity constraint. The fault tolerance subsystem developed in this project directly addresses this limitation, as discussed in Section VII.

Serialization performance. Java's built-in serialization mechanism is known to produce large output and incur high CPU overhead compared to alternative serialization libraries. Kryo is a widely used Java serialization library that produces compact binary output and avoids much of the overhead of Java's reflection-based approach. This paper provides

empirical measurements specific to the MASS graph state data structures, evaluated in the context of fault tolerance checkpointing.

III. MASS Fault Tolerance Architecture

A. Background and Motivation

Prior to this work, MASS provided no mechanism to persist computation state between super steps or across node failures. The only existing persistence mechanism was a shared-memory graph store (SharedGraph) that wrote serialized graph data to /dev/shm, which is process-local and does not survive a node restart. There was no support for transferring backup data to remote nodes.

The fault-tolerance architecture developed in the first phase of this project introduces four cooperating components, all implemented within the edu.uw.bothell.css.dsl.MASS.graph package of mass_java_core.

B. Component Overview

At the heart of the system is a simple agreement, expressed as the DiskGraphStore interface: every machine participating in a computation must be able to write its share of the graph to disk and describe where that saved copy lives. This agreement is not tied to any particular serialization technology — it is just a contract. Any storage strategy that honors it can plug in and work.

The first concrete implementation of that contract, DiskSharedGraph, uses Java's built-in serialization via Apache Commons SerializationUtils. When a backup is triggered, the graph state — which consists of the VertexPlace collection, a distributed key-value map, a queue of recyclable vertex IDs, and a counter tracking the next vertex ID to be assigned — is broken into four pieces and each piece is written as a separate file in a dedicated directory on the local disk. This approach is straightforward, but it has a known weakness: some internal Java types, particularly the synchronized collections that hold the agents residing on each vertex, resist serialization because they were never designed to be saved this way.

A second implementation, KryoDiskSharedGraph, solves that problem by swapping in the Kryo serialization library. Kryo can handle types with no default constructor — including precisely those synchronized collections that DiskSharedGraph cannot touch — by using a fallback instantiation strategy. The two implementations are otherwise identical from the outside, making it easy to switch between them without changing anything else in the system.

Once a machine has saved its own backup locally, the data needs to travel. The machines are arranged in a logical ring, and a FaultTolerantTransferManager on each node is responsible for sending its backup files to its immediate neighbor: node i sends to node $i+1 \pmod N$. This means every machine's data ends up stored in two places — locally, and on the next machine in the ring. The transfer happens over a straightforward TCP connection on a configurable port.

The actual file movement is handled by a small, focused utility called FileTransfer, which knows only one thing: how to send a file or a folder from one machine to another over a socket. It has no knowledge of backup strategies or ring topologies — it simply moves bytes reliably, and the transfer layer calls it for each file that needs to cross the network.

Everything described above is orchestrated by a single coordinator, GraphBackupManager. When a backup is requested, the coordinator first tells the configured DiskGraphStore to write the graph state to disk, then walks through every file in the resulting directory and hands each one to the FaultTolerantTransferManager for delivery to the neighboring machine. The coordinator is created the first time a backup is needed and then kept alive for the duration of the computation, so there is no repeated setup cost across multiple backup cycles.

Complete class and method signatures are listed in Appendix A.

C. Integration with the MASS Framework

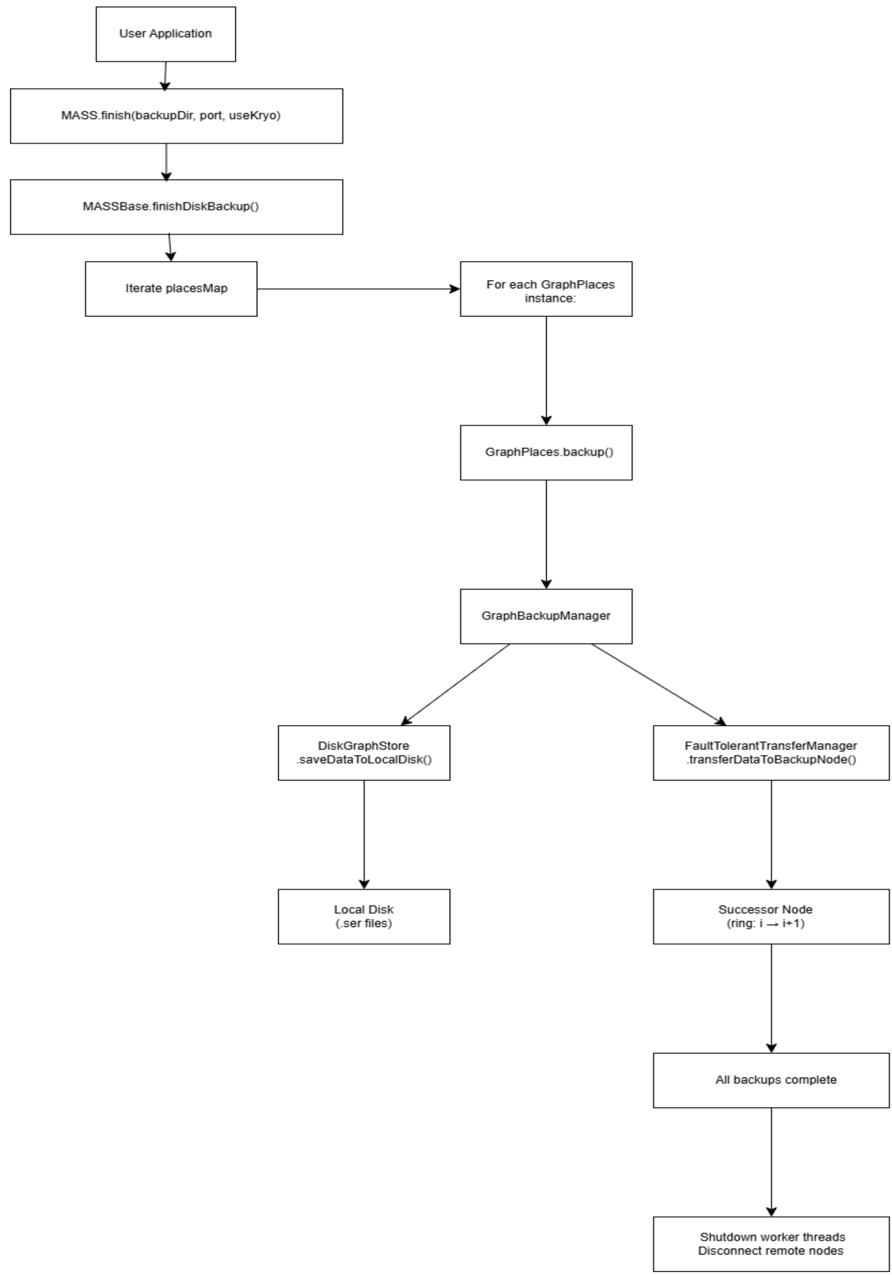
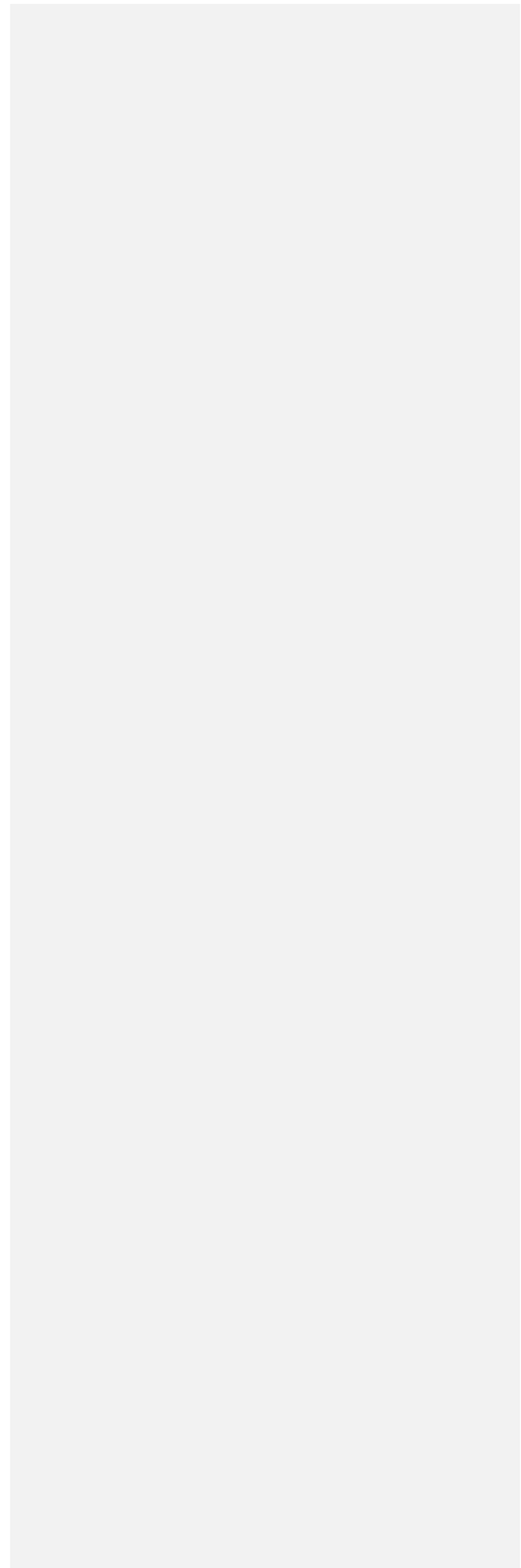


Figure1.Shutdown-Level Backup Interaction

When the application finishes its computation as Figure1, it calls MASS.finish() with a backup directory, a transfer port, and a serialization preference. This call delegates to MASSBase.finishDiskBackup(), which iterates through the global placesMap to locate every registered GraphPlaces instance. For each instance found, it invokes GraphPlaces.backup(), which in turn activates the GraphBackupManager. The coordinator performs two steps in sequence: it first calls the configured DiskGraphStore to serialize the four components of graph state into separate files on the local disk, and then hands those files to the FaultTolerantTransferManager, which transmits them over TCP to the next node in the ring. Once all GraphPlaces instances have been backed up and replicated, MASS.finish() proceeds to shut down the worker threads and disconnect all remote nodes. This is a one-way process — after the backup completes, the system terminates.



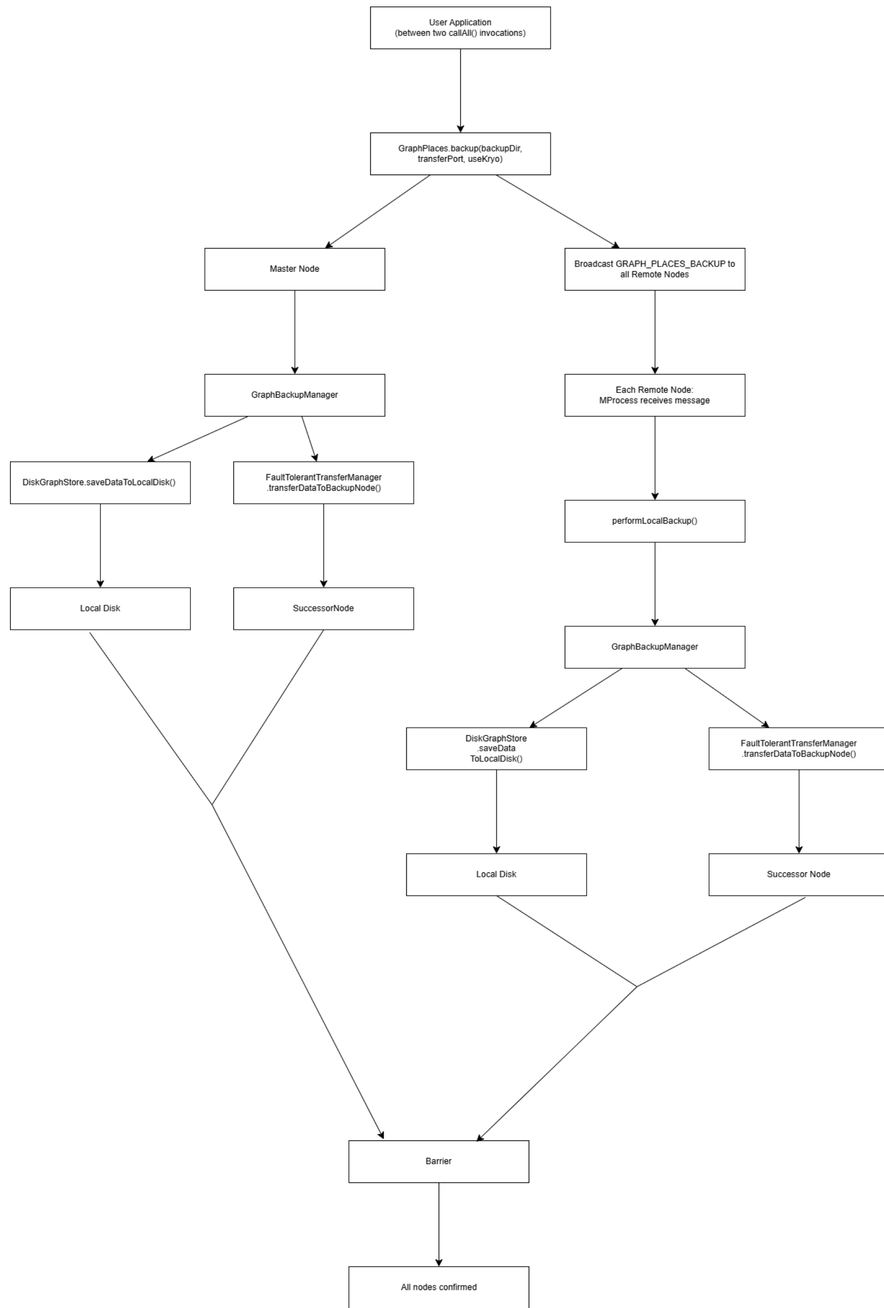

User Application
(between two callAll() invocations)
GraphPlaces.backup(backupDir,
transferPort, useKryo)
Master Node
Broadcast GRAPH_PLACES_BACKUP to
all Remote Nodes
GraphBackupManager
Each Remote Node:
MProcess receives message
DiskGraphStore.saveDataToLocalDisk()
FaultTolerantTransferManager
.transferDataToBackupNode()
performLocalBackup()
Local Disk
SuccessorNode
GraphBackupManager
DiskGraphStore
.saveData
ToLocalDisk()
FaultTolerantTransferManager
.transferDataToBackupNode()
Local Disk
Successor Node
Barrier
All nodes confirmed

Figure2. Superstep-Level Backup Interaction

Between two callAll() invocations during a live computation, the application calls GraphPlaces.backup() to create a mid-computation checkpoint. On the master node, this call does two things simultaneously: it begins its own local backup through GraphBackupManager, and it broadcasts a GRAPH_PLACES_BACKUP message to all remote nodes. Each remote node's MProcess receives this message and calls performLocalBackup(), which triggers the same GraphBackupManager sequence — DiskGraphStore serializes the local partition to disk, then FaultTolerantTransferManager sends the files to the node's ring successor. The master node follows the identical sequence for its own data. After every node — master and remote alike — has completed both serialization and transfer, all nodes synchronize at a barrier. Only when every node has confirmed completion does the barrier release, and the computation resumes with the next superstep. Unlike the shutdown-level backup, this process is repeatable: it can be called after every superstep to create a series of incremental checkpoints throughout the computation.

IV. Agent Serialization: Analysis of Three Approaches

A. The Agent Serialization Problem

The backup architecture described in Section III checkpoints graph state — the set of VertexPlace objects (each VertexPlace represents a single graph vertex, storing its neighbor list and edge weights), the distributed vertex ID map, the ID queue, and the next vertex ID counter.In MASS, agents are mobile computational entities that migrate between places; their current position and internal state are not captured by the graph backup.

For a mid-computation checkpoint to support full recovery, it must be possible to restore not only the graph structure but also the agents that were active at the time of the checkpoint. This section analyzes three approaches to agent serialization and argues that only one is practically feasible.

To illustrate the problem concretely, consider the CrawlerGraphMASS agent used in the TriangleCounting application. Each crawler carries an itinerary array of four integers recording the vertices visited so far. A complete checkpoint of the active agent population requires persisting the itinerary of every live agent at the time of the checkpoint.

B. Approach 1: Universal Full Agent Serialization

The most straightforward approach is to serialize the complete state of every active agent using Java's built-in serialization mechanism. CrawlerGraphMASS already implements java.io.Serializable, and in principle, an ObjectOutputStream could write each agent to disk.

However, this approach faces two significant obstacles. First, VertexPlace.agents is a java.util.Collections$SynchronizedSet, which is a JDK-internal wrapper class that does not have a public no-arg constructor. While this specific issue has been resolved in KryoDiskSharedGraph through the StdInstantiatorStrategy fallback described in Section III-B, the deeper problem lies in the scale of serialization.

A graph with 10,000 vertices may generate on the order of one million active agents at peak during TriangleCounting. Each agent object carries not only application-specific fields (16 bytes for a four-integer itinerary) but also the full object graph inherited from GraphAgent and Agent base classes, including references to thread-local state, place references, and migration queues. A conservative estimate of 400 bytes per serialized agent yields approximately 400 MB of checkpoint data per super step. At this scale, serialization time would dominate computation time, and disk I/O would become the primary bottleneck, making this approach impractical for production-scale graphs.

C. Approach 2: Transparent Framework-Level Checkpointing

A second approach, inspired by Apache Flink's distributed snapshot mechanism [1], would have the MASS framework automatically capture a consistent global snapshot of all agent state at barrier points between super steps, without any involvement from the application developer.

This approach is theoretically attractive but practically infeasible in the current MASS architecture. A consistent global snapshot requires that all in-flight agent migrations be either completed or rolled back before the snapshot is taken. In MASS, agent migration is implemented as a message-passing operation in MProcess, interleaved with callAll() execution and manageAll() barrier synchronization. Inserting snapshot barriers into this protocol without modifying the agent migration model would require changes to MProcess, MThread, the message handling loop, and the Agents class API. Such changes would alter the observable semantics of the framework for all existing applications.

Furthermore, transparent checkpointing does not address the scale problem identified in Approach 1: a full snapshot of one million agents still requires hundreds of megabytes of I/O per checkpoint interval, regardless of how the snapshot is triggered.

D. Approach 3: Application-Defined Checkpoint Interface (Recommended)

The recommended approach introduces a lightweight checkpoint interface on the Agent base class:

```
// To be added to Agent.java in mass_java_core

public byte[] getCheckpointData() { return null; }

public void restoreFromCheckpoint(byte[] data) { }
```

Application developers override these methods to serialize only the fields that cannot be reconstructed from the graph structure. For CrawlerGraphMASS — a GraphAgent that traverses three-hop paths in the graph and records each visited vertex in a four-element itinerary array to detect triangles — the only irreducible state is itinerary[0] — the origin vertex. intermediate hops that can be derived from the agent's current position and the graph topology; itinerary[3] is always -1 at a super step boundary.

Under this model, checkpointing one million CrawlerGraphMASS agents requires storing one integer (4 bytes) per agent, yielding 4 MB of checkpoint data — a reduction of approximately 100× compared to full agent serialization.

The framework is responsible for iterating active agents, calling getCheckpointData(), and writing the results to disk alongside the graph state backup. Recovery proceeds by restoring the graph state via the existing DiskSharedGraph or KryoDiskSharedGraph mechanism, then reinstantiating agents at their checkpointed origin vertices and calling restoreFromCheckpoint() to initialize their internal state.

This design respects the principle of separation of concerns: the framework owns the checkpointing mechanism, and the application owns the checkpointing content. It imposes a minimal burden on application developers — a single method returning a small byte array — while making the checkpoint size proportional to the irreducible state of the application rather than the full object graph of the agent hierarchy.

V. Serialization Benchmark

A. Experimental Setup

To evaluate the performance trade-off between Java native serialization and Kryo serialization for graph state persistence, a benchmark was implemented within the CountTrianglesGraphMASS application (the main driver class that loads the graph, instantiates CrawlerGraphMASS agents, and orchestrates the triangle counting iterations) and executed on a representative server node. The benchmark is invoked via the --benchmark flag:

```
java --add-opens=java.base/sun.nio.ch=ALL-UNNAMED \
    -jar TriangleCounting.jar --benchmark <graph_file>
```

Three graph files were used, containing 750, 1250, and 2500 vertices, respectively, generated in the MASS DSL graph format (a text-based file format that defines the graph structure including vertices, edges, and their weights). Each benchmark run loads the graph into a live GraphPlaces instance using the standard loadDSLFile() method, ensuring that the serialized data structures are identical to those produced during a real computation.

For each graph size, the benchmark performs two warmup iterations followed by five timed iterations for each serialization method. Each timed iteration measures:

- **Write time:** the total wall-clock time to call saveDataToLocalDisk(), which serializes all four components of graph state to separate files on disk.
- **Read time:** the total wall-clock time to call all four read*FromDisk() methods, which deserializes all components back into memory.
- **File size:** the total size of all serialized files in the backup directory, measured after the final write iteration.

Timing is performed using System.nanoTime() and reported in milliseconds averaged over the five timed runs. The benchmark implementation is available in CountTrianglesGraphMASS.runDiskSerializationBenchmark().

B. Results

Table I presents the benchmark results across all three graph sizes.

TABLE I. Serialization Benchmark Results

Graph Size	Method	Size (MB)	Write (ms)	Read (ms)
750 vertices	Native	1.62	68.0	86.2
750 vertices	Kryo	0.47	46.2	33.3
1250 vertices	Native	2.72	79.6	87.6
1250 vertices	Kryo	0.78	50.1	41.5
2500 vertices	Native	5.51	157.7	165.3

Graph Size	Method	Size (MB)	Write (ms)	Read (ms)
2500 vertices	Kryo	1.67	73.9	48.3

TABLE II. Kryo Speedup Relative to Java Native Serialization

Graph Size	Size Reduction	Write Speedup	Read Speedup
750 vertices	3.46×	1.47×	2.58×
1250 vertices	3.48×	1.59×	2.11×
2500 vertices	3.29×	2.13×	3.42×

C. Analysis

Kryo outperforms Java native serialization on all three metrics across all graph sizes. File size reduction is remarkably consistent, ranging from 3.29× to 3.48×, indicating that the structural overhead of Java's serialization format (class descriptors, field metadata, and object references) accounts for a stable fraction of total output size regardless of graph scale.

Write speedup increases monotonically with graph size, from 1.47× at 750 vertices to 2.13× at 2500 vertices. This trend suggests that the per-object overhead of Java serialization scales worse than Kryo's binary encoding as the number of VertexPlace objects grows. Read speedup follows a similar trend, reaching 3.42× at 2500 vertices. The read advantage is consistently larger than the write advantage, which is consistent with the general observation that Java's deserialization path incurs higher overhead than serialization due to class loading and object graph reconstruction.

Extrapolating to the target production scale of 10,000 vertices, the performance gap is expected to widen further. If write time scales roughly linearly with vertex count — which is consistent with the observed data — a 10,000-vertex graph would require approximately 630 ms to write with Java native serialization versus approximately 296 ms with Kryo, based on the 2500-vertex scaling factor.

VI. Discussion

A. Kryo as the Preferred Serialization Backend

The benchmark results of Section V provide a clear empirical basis for selecting Kryo as the default serialization backend for the MASS fault tolerance subsystem. The

DiskGraphStore interface makes this selection transparent to application code: changing the serialization backend requires only passing useKryo = true to GraphPlaces.backup() or MASS.finish().

One consideration that favors Java native serialization is compatibility. Any class serialized with Java's built-in mechanism can be deserialized by any JVM without additional dependencies, provided the class definition has not changed. Kryo-serialized data, by contrast, is sensitive to changes in class field layout. For short-lived operational backups intended to survive a node failure within a single computation, Kryo's performance advantages clearly outweigh this concern.

B. Mid-Computation Checkpointing in TriangleCounting

TriangleCounting is an application that counts and enumerates all triangles (cycles of three mutually connected vertices) in a given graph. The GraphPlaces.backup() API introduced in this project enables mid-computation checkpointing between any two callAll() invocations. In TriangleCounting, this means a checkpoint can be inserted between any of the three iterations:

```
for (int i = 0; i < 3; i++) {
  if (i < 2)
    crawlers.callAll(CrawlerGraphMASS.propagateDown_, currStep);
  else
    crawlers.callAll(CrawlerGraphMASS.migrateSource_, currStep);

  crawlers.manageAll();

  // Mid-computation checkpoint after each super step
  network.backup("/var/tmp/backup", 29055, true);
}
```

If the computation fails after the checkpoint following iteration 1, the graph state can be restored from disk. However, the current checkpoint captures only the graph state and does not record application-level metadata such as the iteration index. Upon recovery, the application cannot determine which superstep to resume from; extending backup() to accept optional metadata for this purpose is left to future work. Furthermore, as

discussed in Section IV, agent state is not currently checkpointed. Recovery from this backup would restore the graph structure but not the agent population, requiring a full restart of the agent-based computation phase. Full mid-computation resume — in which agents are restored to their checkpointed state and computation continues from the interrupted iteration — awaits the implementation of the getCheckpointData() / restoreFromCheckpoint() interface described in Section IV-D.**C. Limitations**

The current implementation has the following known limitations. First, backup is triggered synchronously and blocks computation during the write and transfer phases. For large graphs, the write latency reported in Section V would be visible as pauses in computation throughput. Asynchronous backup, in which graph state is snapshotted and written in a background thread while computation continues, is a potential optimization. Second, the ring topology transfer requires all nodes to be reachable at the time of backup. If a node has already failed before the backup is triggered, the transfer to its successor will fail silently. A more robust implementation would detect transfer failures and fall back to a direct write to a shared network file system.

VII. Conclusion and Future Work

This paper has presented a fault tolerance subsystem for the MASS Java distributed graph computing framework, developed across two semesters of investigation. The first phase established a ring-topology backup architecture capable of persisting graph state to local disk and replicating it to a successor node. The second phase evaluated mid-computation checkpointing feasibility, analyzed the agent serialization problem, benchmarked two serialization backends, and delivered a production-ready implementation.

The serialization benchmark demonstrates that Kryo provides consistently superior performance compared to Java native serialization: up to 3.48× smaller files, 2.13× faster writes, and 3.42× faster reads. These advantages grow with graph size, making Kryo the clear choice for the production-scale graphs targeted by the MASS framework. The implemented subsystem — comprising DiskSharedGraph, KryoDiskSharedGraph, FaultTolerantTransferManager, GraphBackupManager, and the GraphPlaces.backup() API — is robust, well-tested, and ready for integration into the develop branch.

Two directions for future work are identified.

Disk-backed graph streaming. Hong and Fukuda's graph pipelining implementation [2] demonstrates that overlapping graph construction with agent-based computation yields substantial performance gains. However, their approach still requires the entire graph to

eventually reside in distributed memory once all batches have been loaded. The disk backup infrastructure developed in this project — specifically KryoDiskSharedGraph and GraphPlaces.backup() — enables a natural extension: after agents complete computation on a batch and before the next batch is loaded, the current graph state is checkpointed to disk via backup(). Once a segment is no longer needed for active computation, it can be evicted from memory and reloaded from disk if agents need to revisit it. This would lift the current constraint that graph size is bounded by total distributed memory, allowing MASS to operate on graphs of arbitrary size using the same pipelining programming model already established in [2].

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 36, no. 4, pp. 28–38, 2015.

[2] Y. Hong and M. Fukuda, "Pipelining Graph Construction and Agent-based Computation over Distributed Memory," in *Proc. IEEE International Conference on Big Data*, 2022.

[3] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[4] M. Fukuda, "MASS: A Multi-Agent Spatial Simulation Library for Parallel Computation," University of Washington Bothell, Tech. Rep., 2017.

Appendix

DiskGraphStore (interface). Defines the contract for disk-based graph state persistence: saveDataToLocalDisk() accepts the four components of graph state (Vector<VertexPlace>, MASSSimpleDistributedMap, ConcurrentLinkedQueue<Integer>, and nextVertexID) and writes them to a named directory. getPrimaryReplicaPath() returns the path of the primary replica for use by the transfer manager.

DiskSharedGraph. Implements DiskGraphStore using Java native serialization via Apache Commons SerializationUtils. Each component of graph state is serialized to a separate .ser file within the backup directory. Supports multiple replica paths for redundancy.

KryoDiskSharedGraph. A drop-in replacement for DiskSharedGraph using Kryo serialization. Configured with Kryo.DefaultInstantiatorStrategy and a StdInstantiatorStrategy fallback, enabling serialization of JDK internal types such as Collections$SynchronizedSet (used by VertexPlace.agents) that lack a no-arg constructor.

FaultTolerantTransferManager. After local backup completes, transfers all backup files to the successor node in a ring topology over a configurable TCP port. Each node i sends its backup files to node $i+1 \pmod{N}$, ensuring that every node's data is replicated on exactly one other machine.

FileTransfer. A low-level utility providing static methods for transferring individual files or entire directories over a socket connection. Used internally by FaultTolerantTransferManager.

GraphBackupManager. Coordinates the complete backup sequence: calls saveDataToLocalDisk() on the configured DiskGraphStore, then invokes FaultTolerantTransferManager.transferDataToBackupNode() for each file in the backup directory. The manager is lazily initialized on first use and reused across subsequent backup calls.

Commented [MF1]: Instead of enumerating class names and methods like a manual, illustrate a system overview and explain each component in a narrative. All these classes and methods can be listed in the Appendix.