

Interactive Environment to Support Agent-Based Graph Programming in MASS Java

Daniel Blashaw

Master of Science in Computer Science & Software Engineering

University of Washington | Bothell

June 11, 2021

Project Committee:

Dr. Munehiro Fukuda, Chair

Dr. Bill Erdly, Member

Dr. Erika Parsons, Member

Abstract

Interactive Environment to Support Agent-Based Graph Programming in MASS Java

Daniel Blashaw

Chair of Supervisory Committee:
Munehiro Fukuda, Ph.D.
Computing & Software Systems

MASS is an Agent-Based Modeling (ABM) library that allows for parallelized simulation over a distributed computing cluster. In these simulations, Place objects act as the environment for Agents to interact with and may be internally organized as multi-dimensional arrays, graphs, or trees. Graph-based simulations are best suited for systems where Places' relationship to one another may dynamically change or where Places have an indefinite number of neighbors, such as in social networks. However, these graphs are often very complex and present increased difficulty of debugging and verification for the programmer. To address this problem, the goal of this project is to extend the MASS Java library to include a development environment which allows the programmer to step through a graph-based ABM and visually inspect associated Places and Agents. To accomplish this successfully, we have incorporated Java's JShell for line-by-line execution, checkpointing, and rollback of a simulation; expanded MASS-Cytoscape integration with a full control panel, Agent visualizations, and choice to view subgraphs from MASS; and added Agent Tracking functions to the MASS API. These additions result in a development environment which allows programmers the flexibility to rapidly explore and iterate graph-based ABMs, free to focus on the logic of their simulations and not the infrastructure needed to validate their output. Further, although the functionality discussed in this project was designed for graph-based ABMs, their implementation benefits many other non-graph applications and provides a solid foundation for further expansion of the MASS Java library, such as with real-time cluster monitoring and visualization of other simulation data structures.

Table of Contents

| | |
|--|-----------|
| Chapter 1: Introduction | 1 |
| 1.1 Problem Definition..... | 1 |
| 1.2 Research Objective and Contribution | 2 |
| 1.3 Report Structure..... | 3 |
| Chapter 2: Previous Work | 4 |
| 2.1 MASS with Graphs..... | 4 |
| 2.2 Interactive MASS..... | 5 |
| Chapter 3: Related Work | 6 |
| 3.1 Cytoscape | 6 |
| 3.2 Repast Symphony | 7 |
| 3.3 Alternatives in Agent Tracking | 8 |
| 3.4 Summary..... | 8 |
| Chapter 4: MASS GUI Support for Graphs and Agents | 10 |
| 4.1 MASS Library..... | 10 |
| 4.2 MASS Agent Tracking API | 11 |
| 4.3 Enhanced Simulation Controls (InMASS)..... | 21 |
| 4.4 MASS-Cytoscape Integration for Visualization | 24 |
| 4.5 Usage Scenarios | 30 |
| 4.6 Contribution Summary | 30 |
| Chapter 5: Evaluation | 32 |
| 5.1 Results | 32 |
| 5.2 Discussion | 39 |
| Chapter 6: Conclusion | 42 |
| 6.1 Future Work | 42 |
| References | 44 |
| Appendix A: Developer Guide | 45 |

List of Figures

| | |
|---|----|
| Figure 4.1: MASS Library Model | 11 |
| Figure 4.2: Agent Tracking Class Diagram | 13 |
| Figure 4.3: AgentsBase Constructor Sequence Diagram | 15 |
| Figure 4.4: AgentsBase manageAll() Sequence Diagram | 16 |
| Figure 4.5: Sample Graph..... | 17 |
| Figure 4.6: Agent History data before parent propagation..... | 17 |
| Figure 4.7: Agent History data after parent propagation | 18 |
| Figure 4.8: Agent Tracking API Process Flow Diagram..... | 20 |
| Figure 4.9: Checkpoint / rollback class diagram | 23 |
| Figure 4.10: MASS-Cytoscape Integration Architecture Diagram | 27 |
| Figure 4.11: N-Neighbors Graph Retrieval (Centroid = 0, DoS = 1)..... | 29 |
| Figure 5.1: Summary of graph-based Triangle Counting performance testing..... | 33 |
| Figure 5.2: Simulation performance using History Passing and Agent Tracking API..... | 34 |
| Figure 5.3: Agent data extraction time using History Passing and Agent Tracking API.... | 35 |
| Figure 5.4: Qualitative comparison of History Passing and Agent Tracking API | 36 |
| Figure 5.5: Qualitative comparison Repast Symphony versus MASS | 37 |
| Figure 5.6: Agent Path Visualization | 38 |
| Figure 5.7: Heat Map Visualization | 39 |

Chapter 1:

Introduction

1.1 Problem Definition

Agent-Based Modeling (ABM) simulation is a powerful tool for researchers of any discipline exploring a scenario in which the problem can be modeled using autonomous entities, called “agents”, that operate under a predetermined set of rules to interact with the simulation environment and other agents. Particularly, designers of ABMs often seek to observe and analyze high-level shifts to the simulation environment based on the many low-level actions of individual agents, such as in social sciences, economics, and physics domains.

The Multi-Agent Spatial Simulation (MASS) Java library has been developed by the Distributed Systems Lab (DSLab) group at the University of Washington with the goal of making ABMs accessible to researchers with any level of computing experience. To do this, the MASS Java library provides an intuitive programming framework that abstracts away the communication and agent coordination complexities of designing ABMs in distributed computing environments. In MASS Java, simulations may consist of two entities, `Agents` and `Places`. These `Places` represent the simulation environment with which `Agents` may interact and are stored in multi-dimensional arrays allocated over the distributed system.

Storing `Places` in a multi-dimensional array is ideal for simulations where the space is continuous, such as in strategic battle games in which the environment might resemble a checkerboard. However, not all systems can be easily modeled in continuous space. For example, imagine each `Place` represents an individual user in a social network: in this scenario, there will be some users with many connected friends and others with very few. Further, the number of friends for any individual user may be constantly changing as new friends are added and removed from the network. For simulations like this, a graph structure is much more

intuitive to our understanding of the system in the real world. Unfortunately, the variable nature of these graph structures also makes them very difficult to debug and validate, and MASS currently lacks graphical inspection tools to assist.

1.2 Research Objective and Contribution

The aim of this research is to create an interactive environment which allows the user to easily code, debug, and validate graph-based simulations in MASS. The user needs to be able to visually inspect the graph structure and then scrutinize Agent movement over the graph incrementally during simulation execution. Further, since MASS Java runs in a distributed environment capable of handling graphs much larger than can fit on the resources of a single machine, the visualization solution needs to be able to show only a subsection of the overall graph from MASS based on the user's selection. Toward this vision, there are three high-level goals for this project:

1. **MASS Agent Tracking API** must be implemented to allow for visualization of Agent movement in the developed environment. This ensures proper formatting of the Agent data, can be optimized for performance, and will be a valuable tool for any application developers that rely on Agent movement patterns for evaluating their simulation.
2. **Enhanced Simulation Controls** which will enable the user to step through their simulation to determine correctness and impact of each statement in the application incrementally. This project will be expanding on the work of a previous graduate student that used Java's JShell¹ to provide this functionality [3].
3. **Expand MASS-Cytoscape integration** to include agent data transfer and visualization options which allow the user to investigate simulation state at any point in execution.

Actions toward achieving these goals are completed in three phases. First, we add support for Agent tracking by (a) creating classes for managing and representing the Agent history

¹ JShell is a Read-Evaluate-Print-Loop (REPL) tool originally shipped with the Java 9 JDK. JShell is often used as a learning tool for those new to Java, or a particular library in Java, because it offers immediate feedback to the user if a line is logically or syntactically incorrect.

information and (b) making required changes to the MASS library to utilize these new classes during execution. Second, we reimplement core JShell functionality with specific focus on (a) simulation state capture for checkpoint and rollback features and (b) support for dynamically created classes in a distributed environment. Third, we expand MASS-Cytoscape integration by (a) creating *import-agent* plugin to retrieve and store Agent history information, (b) creating a control panel for visualization control and managing interactions with MASS, and (c) modifying *import-graph* plugin to support retrieval of a partial graph from MASS.

1.3 Report Structure

The remainder of this paper is divided into six chapters. Chapter 2 includes a summary of the work from previous students who have enabled this project's extension to the MASS Java library. Chapter 3 presents the technologies that are key to understanding this implementation and includes a discussion of competing ABM simulation software Repast Symphony. Chapter 4 discusses the implementation and architecture details of this project and how each deliverable fits into the final solution. Chapter 5 covers some typical use cases for this interactive solution. Chapter 6 presents the result of this research and discusses evaluation of this work in both quantifiable and qualitative measures. Finally, Chapter 7 contains our closing thoughts and recommendations for future work.

Chapter 2:

Previous Work

The work presented in this document would not be possible without the collective, compounding effort of the many students and faculty members who have built MASS into what it is today. Of note for this work, specifically, are the contributions of previous graduate students Justin Gilroy and Nasser Alghamdi who each developed foundational blocks on which this project is based.

2.1 MASS with Graphs

The infrastructure for graph support in MASS Java was originally added by previous graduate student Justin Gilroy. His contributions to the library include: the addition of the `GraphPlaces` class, which contains the bulk of graph construction and maintenance functionality; support for importing graphs of HIPPIE [7] and MATSim [6] file types; and integration of third-party bioinformatics software, Cytoscape, for visualization of graph structures through addition of two Cytoscape extensions: *import-network* and *export-network* [2]. These additions to MASS Java offer the foundation for this research by adding support for execution and simple visualization of graph-based simulations in MASS without requiring custom graph code from the MASS Java user.

Justin's work was limited, however, to visualization of only the graph structure in Cytoscape and does not include support for Agents. Additionally, his implementation assumes the full graph from MASS can be shown in Cytoscape, which is untrue for simulations fully utilizing the resources available in the distributed environment. Finally, the current visualization layer does not synchronize with the MASS simulation and only provides visibility of the graph structure after the simulation has completed.

2.2 Interactive MASS

Interactive MASS (InMASS) in MASS Java is the work of Nasser Alghamdi whose contribution enables MASS simulations to be written from the command-line interface and executed line-by-line, as if being written in Python. To accomplish this, Nasser created a wrapper around Java's JShell interface and then added supporting classes to ensure JShell would function properly in MASS' distributed environment. Further, Nasser's implementation leveraged this JShell functionality to enable MASS to *checkpoint* an in-process simulation and later *rollback* core simulation data to those previously checkpointed states. Importantly, all this can be done without having to stop the active simulation and then recompile, distribute, and reinitialize the program on the cluster [3].

In addition to the JShell implementation, Nasser's work incorporated various other potential additions to the MASS Java library including, but not limited to, (a) a monitoring system that allowed for MASS simulation state to be viewed from a web browser, (b) Agent and Place builder classes that simplified their instantiation in the user program, (c) additional parallelization of MASS startup and shutdown processes, and (d) improvements to inter-process communication through simplified function calls. Unfortunately, these improvements are beyond the scope of research for this project and were not included in the implementation for the sake of simplicity and compatibility with existing MASS Java simulations.

Chapter 3:

Related Work

This section provides an overview of related work to provide context to this project. Specifically, we discuss the software utilized to visualize our data, competitor ABM simulation library Repast Symphony, and previous Agent Tracking implementations.

3.1 Cytoscape

Cytoscape is an open-source network visualization tool, originally developed for use in analysis of biomolecular interaction networks, that has grown to be widely used by various disciplines for general graph support [4]. Three reasons for Cytoscape's wide adoption, all compelling factors in DSLabs' decision to integrate with MASS Java [2], are that it: (1) has extensive file support for importing graphs into Cytoscape; (2) has native functionality for dynamic manipulation of existing graph structures; and (3) is based on the OSGi framework making its components modular and easily extensible. These features suggest potential for Cytoscape to provide a visualization interface for MASS Java, but careful consideration still needs to be made during implementation. Specifically, Cytoscape is a desktop application while MASS supports distributed computation which means that the amount of data MASS supports is much greater than the amount of data Cytoscape can process for visualization.

3.1.1 Open Service Gateway Initiative (OSGi)

OSGi is a specification for a Java-based component framework in which a collection of independent components, known as "bundles", can be deployed independently inside an OSGi container [8]. Each bundle must be completely self-contained allowing it to be started, stopped, or removed from the OSGi container without affecting the functionality of other bundles. This modularity is core to OSGi as it enables new bundles to be created without worrying about

impacting existing components. To facilitate coordination between components, each bundle can register its services with the OSGi service API, allowing it to be invoked by other bundles which would like to subscribe to that service. Thinking about Cytoscape specifically, the Cytoscape application is the OSGi container, and all functionality is implemented or extended through the addition of new bundles.

Although the inherent modularity of the OSGi makes it appealing for this work, the need for complete bundle self-containment does create some problems. Namely, to create a seamless user experience in our visualization GUI, we must establish a mode of inter-bundle communication which allows our extensions to share information indirectly while still adhering to OSGi specifications.

3.2 Repast Symphony

REcursive Porous Agent Simulation Toolkit (Repast) Symphony [10] is a widely used, open-source ABM platform designed to interface with the Eclipse integrated development environment (IDE). Repast Symphony is designed to be modular and allow extension of core functionality through the development and incorporation of plug-ins at runtime for things such as simulation visualization and monitoring. This flexibility makes Repast Symphony a very strong competitor in the ABM space and, therefore, makes it the best candidate for feature comparison of this project's work.

Repast Symphony is an excellent tool for ABMs and offers many features, like its modularity, native GUI interface in Eclipse, and menu-based execution, which make it an appealing choice for researchers. However, the Repast Symphony toolkit falls short when compared to the objectives of this work. Specifically, Repast Symphony can simulate graph-based ABMs but forces the programmer to code the graph logic into a two-dimensional simulation space which leads to the same issues we are trying to solve with this work. That is, programming graphs into a continuous space is an unnecessarily complicated and error prone endeavor. Additionally, Repast Symphony requires that all graph and agent information be set prior to execution of the simulation and does not support incremental backtracking or manipulation of a running

simulation. Finally, Repast Symphony runs on a single CPU and, therefore, cannot support the same volume of data as MASS operating on a cluster of computing nodes.

3.3 Alternatives in Agent Tracking

Data Provenance in MASS, also known as “ProvMASS,” provided a novel approach for tracking data provenance in a distributed setting [9]. This data provenance included Agent data in addition to Place and cluster information and was captured to file at run-time. In the end, these data provenance features proved to be useful in analyzing Agent behavior but had a significant impact on simulation performance which made the implementation undesirable for this project’s goals. Specifically, our Agent Tracking implementation must be lightweight enough to keep in memory during simulation and its use should not significantly impact system performance.

Repast Symphony, on the other hand, has a lightweight implementation for tracking Agent data, but settings must be pre-configured before running the simulation and recorded Agent data can only be written to console or file [11]. This is useful for review of Agent information but does not facilitate programmatic use of this data in a running simulation. This work seeks to create an API in MASS which returns Agent data either to the visualization layer or directly to a running application so the programmer can utilize its output in the simulation.

3.4 Summary

Each of the technologies and approaches discussed in this section provide functionality like the goals set out in this project, but each falls short of our desired outcomes in one aspect or another. (a) Cytoscape provides a flexible and feature rich foundation for visualization but lacks the ability to support large-scale graphs that can be processed using distributed resources. To address this, we will implement functionality to retrieve partial graphs from MASS into Cytoscape for visualization. (b) Repast Symphony provides an IDE-based ABM solution with extensive plug-in support, but lacks desired graph structure, forces users to pre-configure visualizations, and lacks the mid-simulation control features targeted by this work. To improve upon this, we seek to extend MASS’ graph-based simulation support to provide the users with

greater ability to control a running simulation. Further, this project seeks to add visualization support which users can call upon as needed in their applications, asynchronous of simulation execution. Finally, (c) related Agent Tracking implementations are either too computationally expensive for general use or have been limited in their output capabilities. This project aims for a general-purpose Agent Tracking feature that is lightweight and flexible enough to use in the visualization layer or as a replacement for conventional data management techniques in ABM simulations.

Chapter 4:

MASS GUI Support for Graphs and Agents

This chapter introduces some technical aspects of the MASS Library. It then goes on to describe the key considerations and implementation details of each major deliverable of this project: (1) MASS Agent Tracking API, (2) enhanced simulation controls, and (3) expanded MASS-Cytoscape integration for visualization.

4.1 MASS Library

Figure 4.1 provides an overview of the MASS Library Model. In this model, ABMs can be represented using the two available modeling objects: Places and Agents. The Places represent the simulation space, while the Agents are the actors within the simulation. As shown in the application layer of Figure 4.1, Places are distributed among the computing nodes on the cluster, and each is mapped using a globally known set of coordinates which are used to reference or locate the Place. Exactly how Places are mapped depends on the type of Place structure being used. Figure 4.1 illustrates a two-dimensional mapping of Places with x and y coordinates. Regardless of mapping structure, all Places are mapped to processes and may exchange data amongst themselves or with visiting Agents. Agents are then stored in bags on each process and may traverse the cluster to visit Places throughout program execution. When traversing, Agent data is serialized and passed between cluster nodes via TCP communication. Not all simulations utilize both Agents and Places; a simulation may use only Places, if that is all that is needed for computation.

The MASS library functions using a master-worker pattern to control the simulation. In Figure 4.1, Computing Node 0 represents the master node, also referred to as the host node, and the other nodes represent the worker, or remote, nodes. User applications interact with the

MASS host node and then MASS controls all supporting communication with the worker nodes internally.

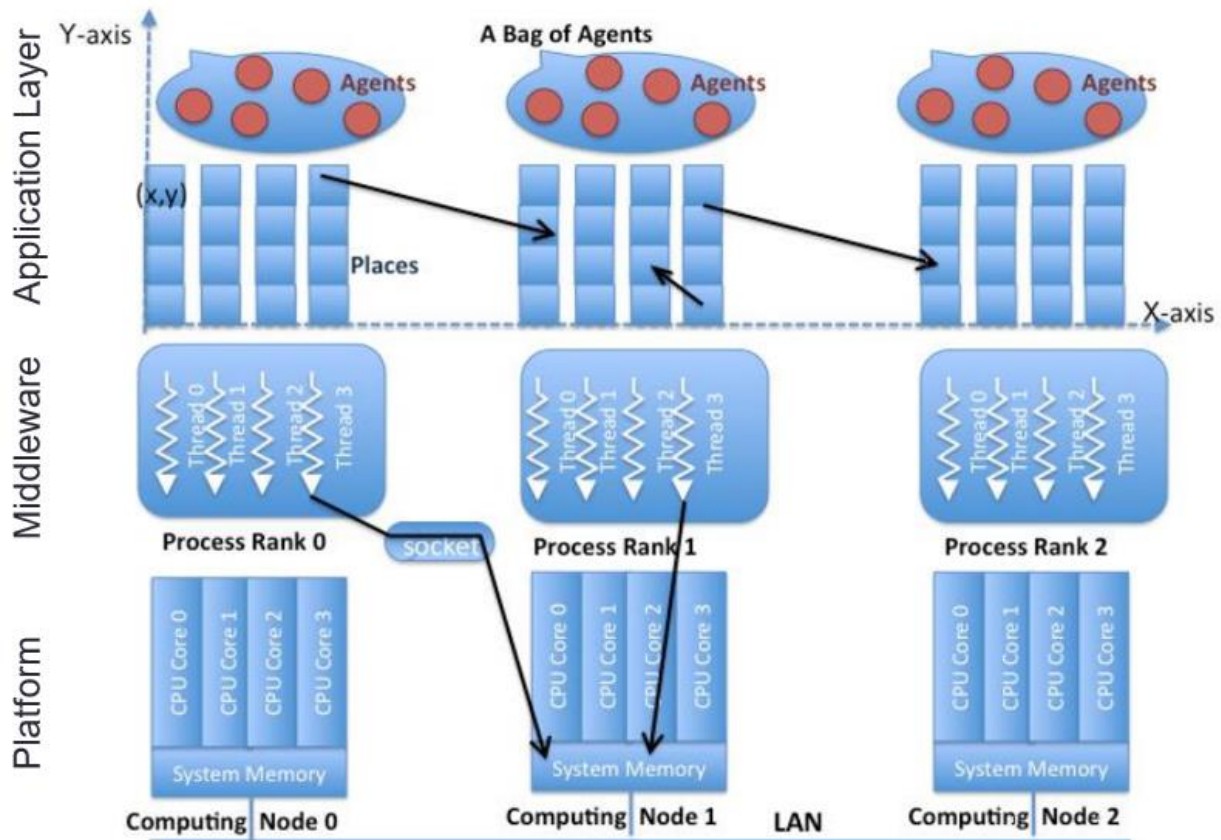


Figure 4.1: MASS Library Model [1]

4.2 MASS Agent Tracking API

To facilitate Agent visualization in our new development environment, we must first add functionality to MASS that will capture the Agent visitation data to be visualized. Toward this objective, a set of Agent Tracking calls have been added to the MASS API.

4.2.1 Key Considerations

When evaluating possible implementations for this feature, the following considerations influenced our design decisions:

- **Performance.** Use of Agent Tracking functionality will introduce extra overhead to simulation execution, due to additional network communications, but the impact to performance must be minimal enough to make Agent Tracking API functionality a viable replacement to user-defined methods.
- **Consistency.** Output of Agent Tracking should be unambiguous and free of errors. Each Agent must be shown only once, and history must be verifiable and consistent with output of user-defined methods.
- **Usability / Ease of Use.** Solution must be straightforward for regular MASS users and output must be easily consumable for general use cases, such as finding all Agents alive at a particular time or determining the number of visits an Agent made.

4.2.2 Overview

The implementation of Agent Tracking functionality in MASS has resulted in the addition of three new classes to the MASS Java library: `AgentHistoryManager`, `AgentHistoryModel`, and `AgentHistoryCollection`. Figure 4.2 shows a class diagram of these new classes as well as all associated new or updated methods and variables on existing classes.

First, as shown in Figure 4.2, if Agent Tracking features are being used in a simulation, then each `Place` will initialize a local `AgentHistoryManager`. This class is responsible for managing which Agents or classes of Agents are being tracked and then recording history each time a tracked Agent visits that Place. Importantly, this means Agent history data is stored on the Places and not the Agents; this maintains simulation performance by ensuring Agents remain lightweight for serialization and transfer between computing nodes. The tradeoff is that Agent history is distributed amongst the cluster nodes during execution which introduces some complications when extracting the data, especially when child Agents are involved (see section 4.2.4 for more discussion on this issue).

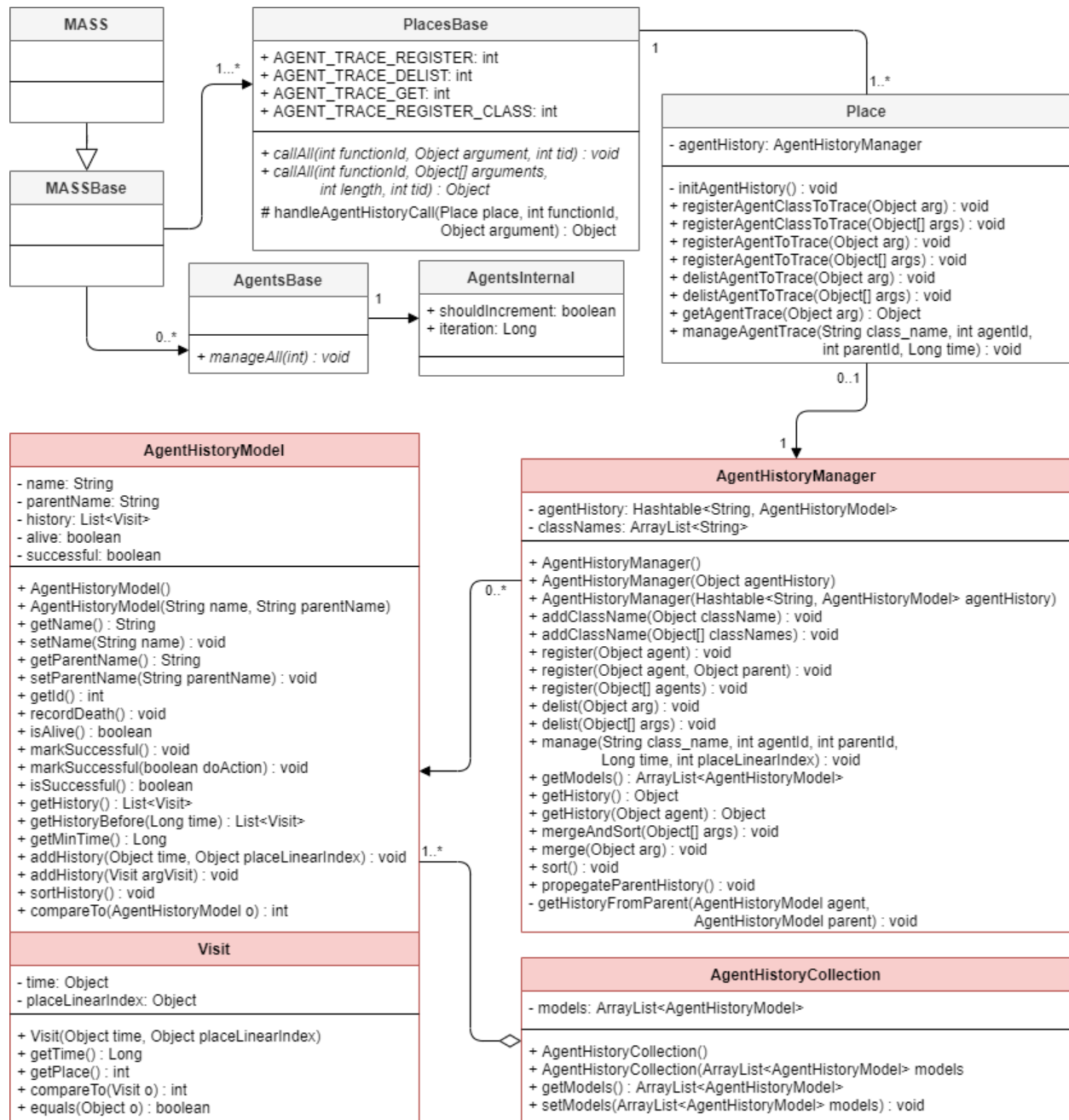


Figure 4.2: Agent Tracking Class Diagram

Due to the complexity of the MASS library, this diagram only shows the classes affected by the addition of Agent Tracking APIs. Methods in *italics* are existing methods which were modified to support the implementation.

Internally, `AgentHistoryManagers` maintain history in a hash table of `AgentHistoryModel` objects using the Agent's name² as the hash key. Each model object represents a single Agent and maintains basic information as well as a list of `Visits` for that Agent. The `Visit` is a simple data storage class which resides inside the `AgentHistoryModel` and consists of two values: `Time` and `Linear Index of the Place`. Managing simulation `Time` is itself a challenge and is discussed in detail in section 4.2.3.

Lastly, `AgentHistoryCollection` objects are simple containers utilized to facilitate movement of `AgentHistoryModels` in a form that can be verified by the JVM and easily consumed by the user. These objects are only used when serving Agent data to Cytoscape or returning data directly to the user.

4.2.3 *Managing Simulation “Time” for History Tracking*

Maintaining `Time` for Agent Tracking purposes is done with a set of new variables which are managed by the `AgentsBase` class. These variables are `iteration`, which is the time value, and `shouldIncrement`, which is a helper Boolean used to ensure the `iteration` counter does not increment more often than it should. Specifically, the `iteration` counter should only increment once per `manageAll()` invocation and only if there is at least one Agent migrating. An Agent must be migrating because, in simulations where Agents are spawning child Agents, there may be multiple `manageAll()` invocations per logical cycle, e.g., one call for parent Agents to migrate and then a second call for child Agents to spawn. In these instances, we ensure history for both calls is captured using the same `iteration` value.

Figures 4.3 and 4.4 show sequence diagrams for `AgentsBase` Constructor and `manageAll()` methods, respectively. Modifications to logic to support Agent Tracking functionality are highlighted in each diagram. In Figure 4.3, each time an Agent is initialized on a `Place`, we use the `Place`'s `manageAgentTrace()` method to signal the arrival of the new Agent. During `AgentsBase` construction, the time recorded will always be zero, and the `shouldIncrement` variable will end as `True`, ensuring `iteration` will increment the next time an Agent migrates.

² An Agent's name is defined as the Agent's class name and ID separated by an underscore. (Ex. “SimpleAgentClass_1”)

In Figure 4.4, we see a similar procedure for `manageAll()`: every time an Agent migrates or spawns, a signal is sent to the receiving Place, and the process always ends by setting `shouldIncrement` to `True`. This process differs, however, the first time an Agent migrates: the iteration counter first increments, to advance the simulation one tick, and then `shouldIncrement` is set to `False`, to avoid incrementing for each migrating Agent.

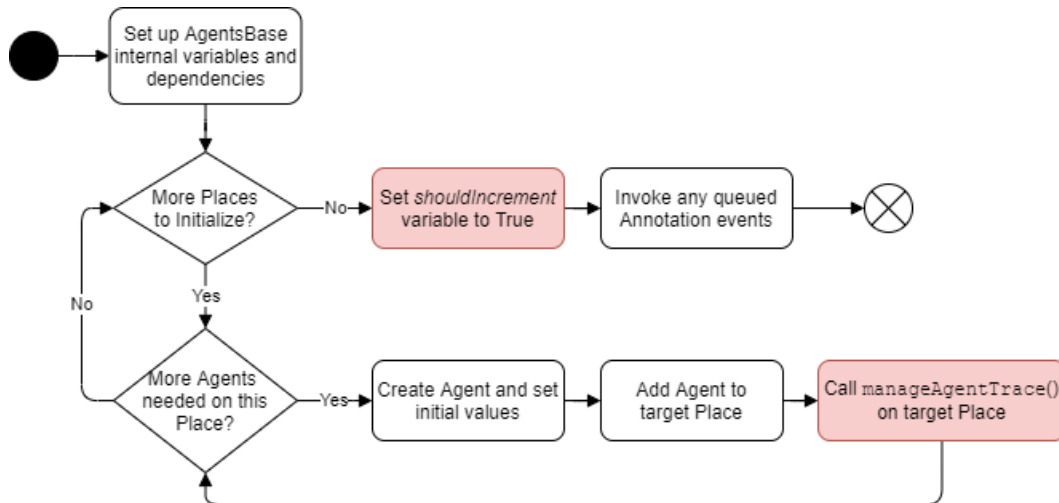


Figure 4.3: AgentsBase Constructor Sequence Diagram

Another option considered for resolving the Time issue was to use the `GlobalLogicalClock` class which supports annotated events functionality in MASS. This option was eventually rejected because it increments with each call of `manageAll()` and would show spawning and migration events occurring at different times.

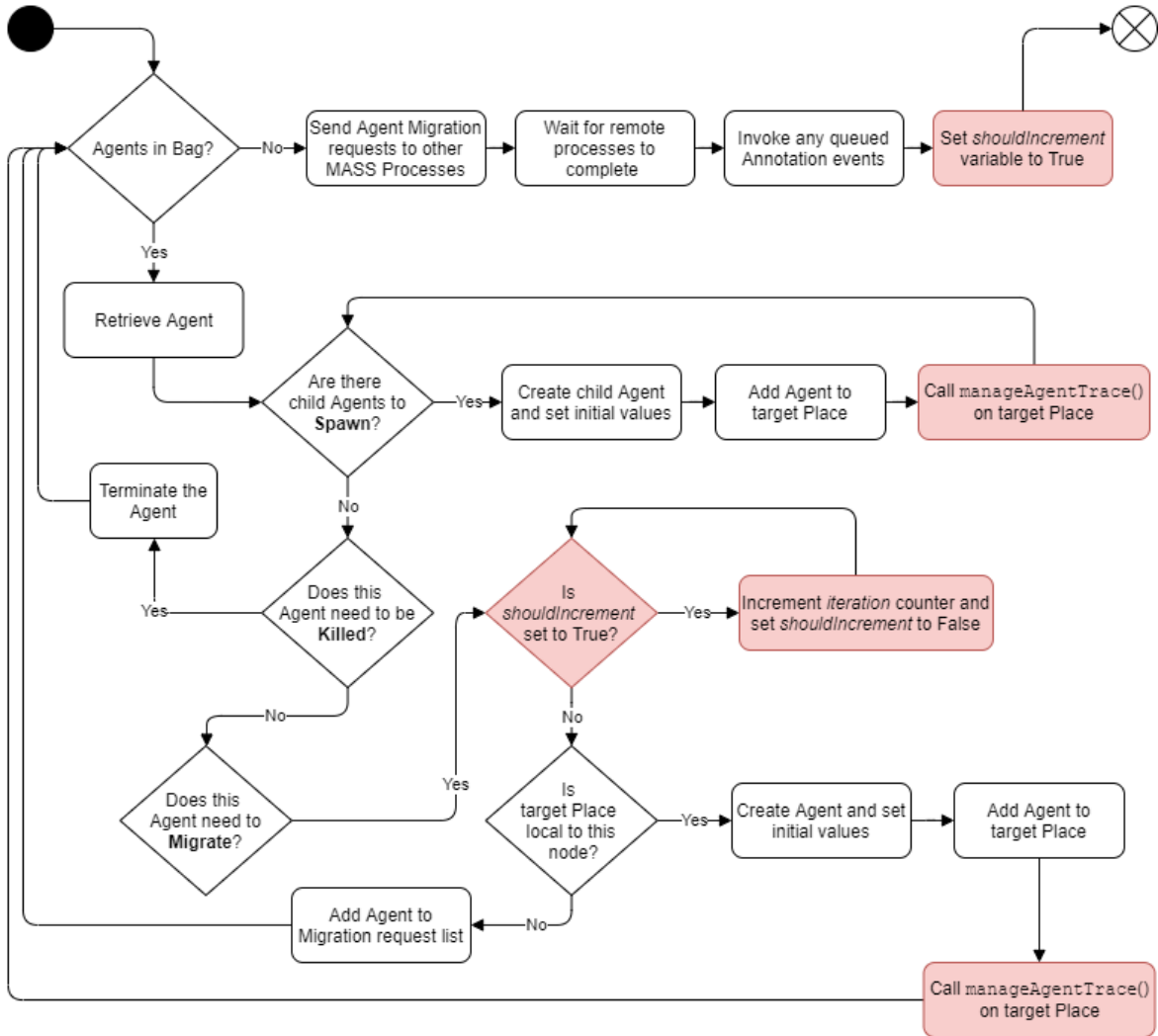


Figure 4.4: AgentsBase manageAll() Sequence Diagram

4.2.4 Parent-Child Data Propagation Problem

In many simulations, Agents will come to decision points at which their instructions indicate they need to travel to multiple Places at once. In these instances, the parent Agent will move to one Place, and then a child Agent will be spawned for each of the other available Places. At this point, if the Agents are being tracked by class name, then the Places will begin gathering data on the newly spawned child Agents. This pattern may continue throughout the simulation, causing multiple waves of child Agents to spawn at various times in the simulation. The issue that arises from this process is that the child Agents will have an incomplete history, because they did not exist at the beginning of execution.

To help illustrate this problem, consider the Triangle Counting benchmark application running on a sample graph shown in Figure 4.5. The Triangle Counting benchmark is solved in four simulation cycles: (Time 0) one Agent is spawned on each available Place; (Time 1-2) each Agent travels to all Places with a lessor index value than current the Place and spawn child Agents if more than one Place fits this criterion; and (Time 3) all remaining Agents attempt to return to Place of Time 0. Each Agent which can return home at Time 3 represents a discovered triangle. Figure 4.6 shows the history captured for each Agent as this benchmark application plays out; note that child Agents are missing data from before they were spawned from the parent Agents. This missing data is needed for the user application to correctly determine the path the Agents traveled along, i.e., the edges of the triangles.

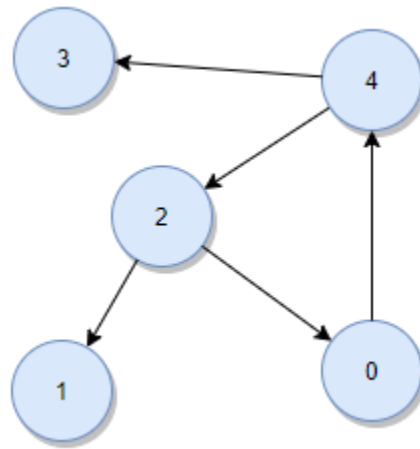


Figure 4.5: Sample Graph

| Agent (Parent) | Time 0 | Time 1 | Time 2 | Time 3 |
|----------------|--------|--------|--------|--------|
| 0 | [0,0] | | | |
| 1 | [0,1] | | | |
| 2 | [0,2] | [1,1] | | |
| 3 | [0,3] | | | |
| 4 | [0,4] | [1,3] | | |
| 5 (4) | ? | [1,2] | [2,1] | |
| 6 (2) | ? | [1,0] | | |
| 7 (5) | ? | ? | [2,0] | [3,4] |

Figure 4.6: Agent History data before parent propagation

Legend: [Time, Place Index] and “?” indicates missing segments of Agent history.

As shown in Figure 4.6, Agent 7 is the only Agent that completed its path at Time 3, but its complete path is unknown because Agent 7 is a child and did not spawn until Time 2. Further, Agent 7 is the child of Agent 5 and Agent 5 is also a child Agent of Agent 4. So, even if we retrieve information from Agent 5, the data will remain incomplete unless we also pull information from Agent 4. Thus, illustrating the need to solve this problem depth-first recursively at the time the data is extracted to the user program.

Figure 4.7 illustrates the result of propagating data from parent Agents, with the red arrows indicating the flow of information from parent to child. In the case of Agent 7, we see that it retrieved results directly from parent Agent 5, after Agent 5 retrieved its own history from parent Agent 4. Now, from Agent 7's movement history, we can correctly conclude the triangle found from this simulation is between Place nodes 4, 2, and 0.

| Agent (Parent) | Time 0 | Time 1 | Time 2 | Time 3 |
|----------------|--------|--------|--------|--------|
| 0 | [0,0] | | | |
| 1 | [0,1] | | | |
| 2 | [0,2] | [1,1] | | |
| 3 | [0,3] | | | |
| 4 | [0,4] | [1,3] | | |
| 5 (4) | [0,4] | [1,2] | [2,1] | |
| 6 (2) | [0,2] | [1,0] | | |
| 7 (5) | [0,4] | [1,2] | [2,0] | [3,4] |

Figure 4.7: Agent History data after parent propagation

An alternative to this implementation would be to populate missing history information at the time each child Agent is spawned. This implementation was considered but rejected based on our design decision to capture Agent information on each distributed Place. To populate each child with parent data upon initialization would require either (a) additional networking calls to collect and redistribute parent information with the distributed Places, which would occur during the simulation and have significant impact on performance, or (b) move Agent history information onto the Agents themselves, which would add overhead to serialization and transmission when migrating, adversely affecting performance. In contrast, the chosen

implementation in this work results in minimal additional overhead during the simulation and only significantly impacts performance when the data is finally retrieved.

4.2.5 Interacting with Agent Tracking API

Figure 4.8 shows the process flow for Agent Tracking API calls. In the top of the diagram, we illustrate the five steps associated with registering an Agent class to be tracked in this simulation: (1) user application utilizes existing MASS API `callAll()` function and provides function Id of `AGENT_TRACE_REGISTER_CLASS`³; (2) MASS host node relays this request to remote nodes; (3) all Places then update their local `AgentHistoryManager` with the new request; (4) host node waits for confirmation from remote nodes that the task is complete; and (5) control returns to the user application. Note that, although the diagram depicts this process for registering a class of Agents, the process is the same for registering or delisting a single Agent or group of Agents.

Once an Agent class is registered for tracking, the user program may continue execution without worrying about tracking data. Each time the `manageAll()` function is invoked, the MASS library will keep track of all associated Agent movements.

The bottom half of Figure 4.8 depicts the process for retrieving Agent data out of the system. The first four steps of the process are generally the same, but with a different function Id: `AGENT_TRACE_GET`¹. However, once all processes have returned their local history to the MASS host node, there is an extra step (5) that consolidates all returned models into a single, cleaned, and sorted model. It is at this step that parent-data propagation occurs. Finally, (6) the output is returned to the user in the form of a single `AgentHistoryCollection` object at position 0 of the return array. From here, the user may retrieve all `AgentHistoryModels` using the `AgentHistoryCollection getModels()` method.

³ To avoid collision with user-defined functions, the Agent Tracking functions utilize the previously unused negative integer space (-1 through -4).

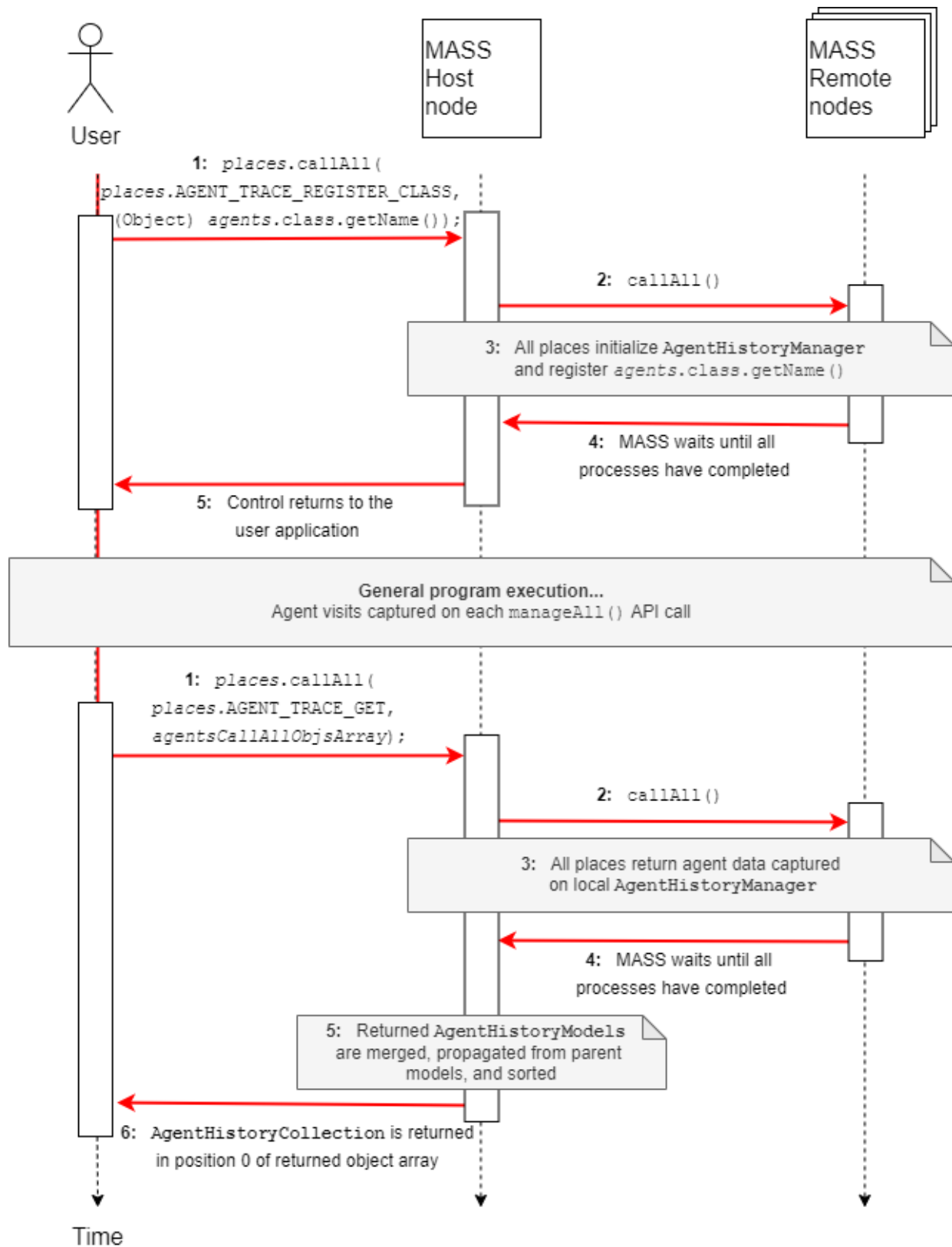


Figure 4.8: Agent Tracking API Process Flow Diagram

4.3 Enhanced Simulation Controls (InMASS)

To provide incremental execution and the ability to checkpoint and rollback a simulation, we have leveraged Nasser Alghamdi's work on InMASS but included only the features necessary to meet this project's objectives.

4.3.1 *Key Considerations*

- **Performance.** Although the functionality included in InMASS brings many possibilities for user interaction with MASS, not all users will want to make use of this functionality; those users should not receive a penalty to system performance.
- **Minimize Impact to Codebase.** Nasser's original implementation of InMASS satisfied primary objectives of this work by enabling line-by-line execution of MASS simulations and including features for checkpointing and rollback but did so in a way that modified more of the MASS library than could be easily incorporated into production versions. For this reason, this work aims to include the same features with less impact on the basic functionality of the MASS Java library.

4.3.2 *Overview*

At the highest level, InMASS is simply a wrapper class that initializes a JShell window, injects MASS startup code into that JShell instance, and then provides hooks for various MASS execution and shutdown functions. This basic functionality alone enables line-by-line execution when running on a single node and effectively eliminates boilerplate code in user applications. The challenges of InMASS implementation, however, revolve around (1) making JShell function properly for all nodes in a distributed environment and (2) deciding how to save and reload simulation state for checkpoint and rollback functionality.

4.3.3 *Managing Dynamically Created Classes*

Built-in Java class loaders rely on classes being included in the application classpath, JVM, or bootstrap loaders, all of which require the class to be known at compile time. However, a key

feature of JShell is that it enables classes to be defined dynamically in a running application. In normal situations, JShell's dynamically created classes would not be a problem, because the computer running JShell would also be running the rest of the application. In cases when a class is defined through JShell, the local class loaders can be notified of the new class. In distributed systems, however, special accommodations must be made for communication of these dynamically created classes with the distributed computing nodes (which are not in direct communication with the JShell client).

To address this issue, a custom class loader, `InMASSLoader`, has been added to keep track of any dynamically created classes. Then, `InMASSLoaderClient` and `InMASSLoaderServer` have been added to facilitate distribution of the classes' bytecode throughout the distributed system. Once all computing nodes are aware of the new classes, they use new `MASSObjectInputStream` and `MASSObjectOutputStream` functions to assist in serialization and deserialization of these dynamic classes. These are variations of Java's standard `ObjectInputStream` and `ObjectOutputStream`, respectively, and simply reroute the program through the `InMASSLoader` when an `Object` cannot be resolved using built-in class loaders.

4.3.4 State Storage and Recovery

Figure 4.9 shows a class diagram of all class additions and revisions that went into adding support for state checkpointing and rollback. As shown, this implementation consists of four major functionality groupings which have been color coded for clarity.

First, the `IInternal` interface along with two implementations, `AgentsInternal` and `PlacesInternal`, (indicated with red in Figure 4.9) have been added to facilitate serialization and deserialization of simulation data. These classes now hold all data elements previously in the `AgentsBase` and `PlacesBase` classes, respectively. When a checkpoint occurs, it is these `IInternal` classes which get serialized and saved. `AgentsBase` and `PlacesBase` now extend the `IRef` interface which provides them with a reference to these `IInternal` classes. This architecture allows for a simple update of the `IRef` reference variable with a new `IInternal` class when rollback is performed: eliminating the need to completely reinitialize the base class.

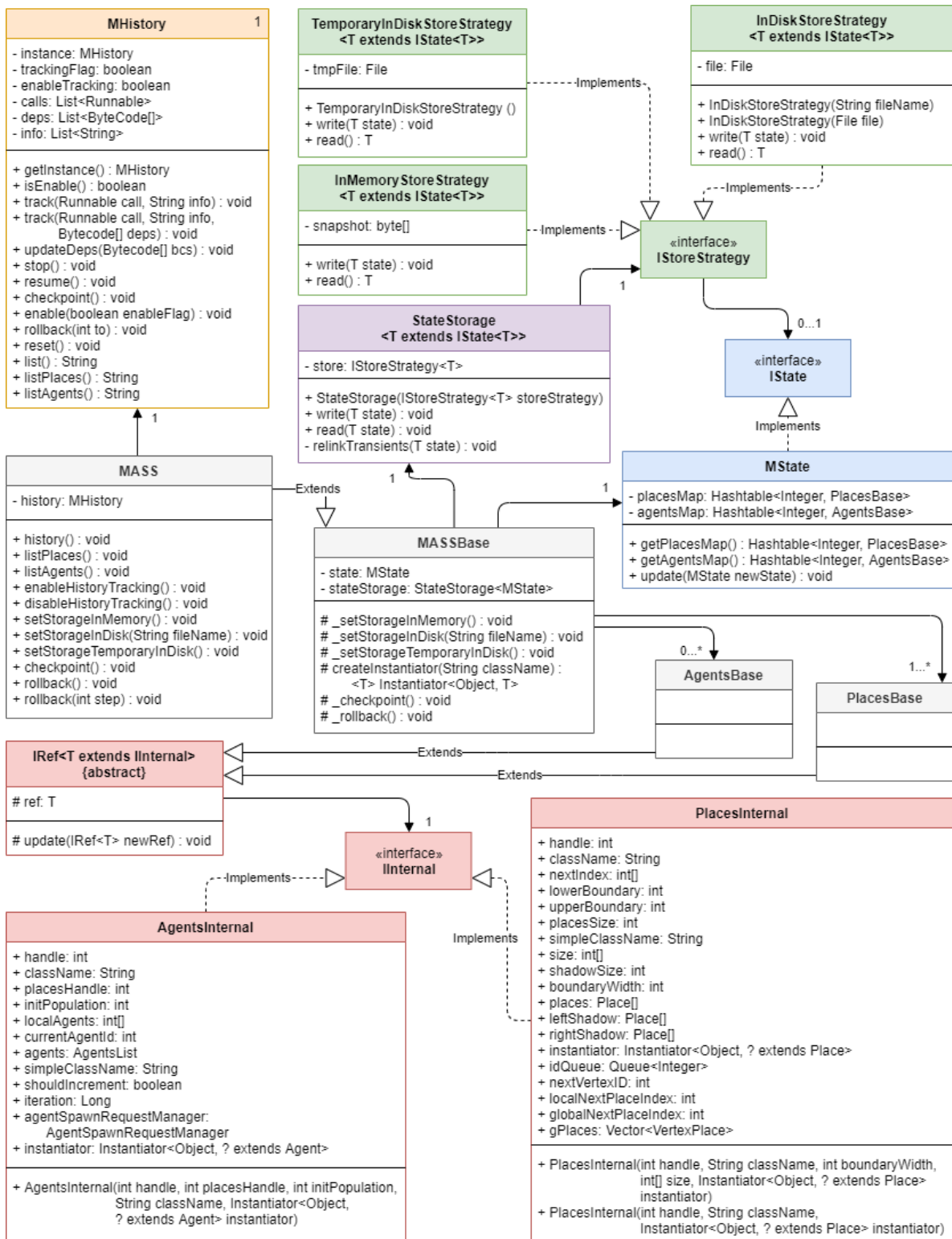


Figure 4.9: Checkpoint / rollback class diagram

Due to the complexity of the MASS library, this diagram only shows the classes affected by the addition of checkpoint and rollback features. API calls in AgentsBase and PlacesBase were also modified to support these MHistory execution logging, but those changes are not reflected here for simplicity.

Second, the `IState` interface and corresponding `MState` implementation (indicated with blue in Figure 4.9) are included to provide a more concise way for MASS to reference the simulation state internally. These are container classes which hold hash tables containing all initialized `AgentsBase` and `PlacesBase` instances. Their inclusion provides a single object to be saved and updated on checkpoint and rollback.

Third, (indicated with green in Figure 4.9) we added the `IStoreStrategy` interface along with three corresponding implementations: `InMemoryStoreStrategy`, `InDiskStoreStrategy`, and `TemporaryInDiskStoreStrategy`. As their class names imply, these contain the procedures for writing and reading `IState` information to active memory, temporary disk location, or a specified file in disk. Inclusion of these `IStoreStrategy` classes decouples the storage process from the data and provides opportunities for future implementations to save data to other locations and formats.

Finally, to facilitate user ability to rollback to states other than the original checkpoint we have added the `MHistory` class (indicated with orange in Figure 4.9). The class keeps a log of all API calls to `AgentsBase` and `PlacesBase` and stores their bytecode to enable re-execution on demand. Consequently, when a user requests rollback to step 5, for example, the original checkpoint will be loaded, and then `MHistory` will execute the next five API calls that follow the checkpoint. Note that all API calls, such as `callAll()`, `manageAll()`, and `exchangeAll()`, have been modified to support this feature, but, for the sake of simplicity, we have not shown these changes in Figure 4.9.

4.4 MASS-Cytoscape Integration for Visualization

To facilitate visualization and validation of simulation data, we have (1) expanded the existing MASS-Cytoscape integration to include transfer of Agent data to Cytoscape, (2) added two types of visualizations for viewing this Agent information, and (3) built a control panel to improve overall user experience. Additionally, (4) to allow large-scale graphs to be shown in Cytoscape we have added the capability to retrieve partial graphs from the MASS library using an n-neighbors approach.

4.4.1 Key Considerations

- **Usability.** The solution developed in this project must be user-friendly and intuitive to use. The final implementation should allow the user to focus on building their application in MASS and not configuring or managing the visualization solution.
- **Scalability.** MASS can run on a distributed system while the Cytoscape client must run on a single machine. Therefore, the visualization solution must allow options for scenarios where large-scale networks in MASS need to be partially shown due to resource constraints.
- **Expandability.** A core benefit of using Cytoscape and the underlying OSGi framework is the modularity and expandability of the system. This implementation should provide a foundation and template for further integration with MASS Java.

4.4.2 Overview

Figure 4.10 presents an overview of the MASS-Cytoscape architecture. On the left side of the figure is the user space. We have illustrated the user's two points of interaction, with the JShell window for executing their simulation in MASS and with the MASS Control Panel for managing their data flow and visualization in Cytoscape.

The MASS Control Panel serves three main functions. First, it provides a single point of interaction for the user by internally managing the data transfer plugins: *import-network*, *export-network*, and *import-agents*. Second, it provides the ability to manipulate the MASS Configuration tables which inform the data transfer plugins of how to find the MASS simulation and what data to pull back into Cytoscape. Lastly, it provides the interface and logic for visualizing Agent movement through manipulation of the Cytoscape data tables and Network View.

In MASS, the `CytoscapeListener` class must be started by the user application to open a TCP-based communication port for MASS-Cytoscape communication. This listener will then field any requests from Cytoscape by first parsing the request, then obtaining reference to the corresponding `GraphPlaces` method, and finally invoking that method and returning the results

to the requesting Cytoscape plugin. Internally, the `GraphPlaces` methods utilize standard MASS internal APIs, such as `callAll()`, to communicate with the rest of the cluster and set or retrieve the appropriate information.

4.4.3 *Synchronizing OSGi Plugins*

OSGi is a flexible specification due to the modularity of each of its component bundles. However, because each bundle must be completely self-contained, we must address how to coordinate the bundles and provide bundle-to-bundle communication of data. Further, OSGi does not guarantee bundles are loaded in a particular order on startup. Therefore, we must also consider how to obtain reference to another bundle that may or may not exist at that point in time.

We handle bundle-to-bundle communication using shared tables within the Cytoscape environment. Specifically, the “`MASS_Configuration`” table acts as the central point for all MASS-related plugins which locate and manipulate this table using a common `MASSConfig` class. The MASS Control Panel uses this class to write new values to the table after taking inputs from the UI and the data transfer plugins read in relevant information each time a new data transfer task is created. In the current implementation, the “`MASS_Configuration`” table includes fields for the MASS hostname and port as well as other fields used for partial graph streaming. By placing all common data in a shared location, we avoid the need for direct bundle-to-bundle communication entirely and, therefore, adhere to OSGi’s philosophy regarding clear separation of concerns.

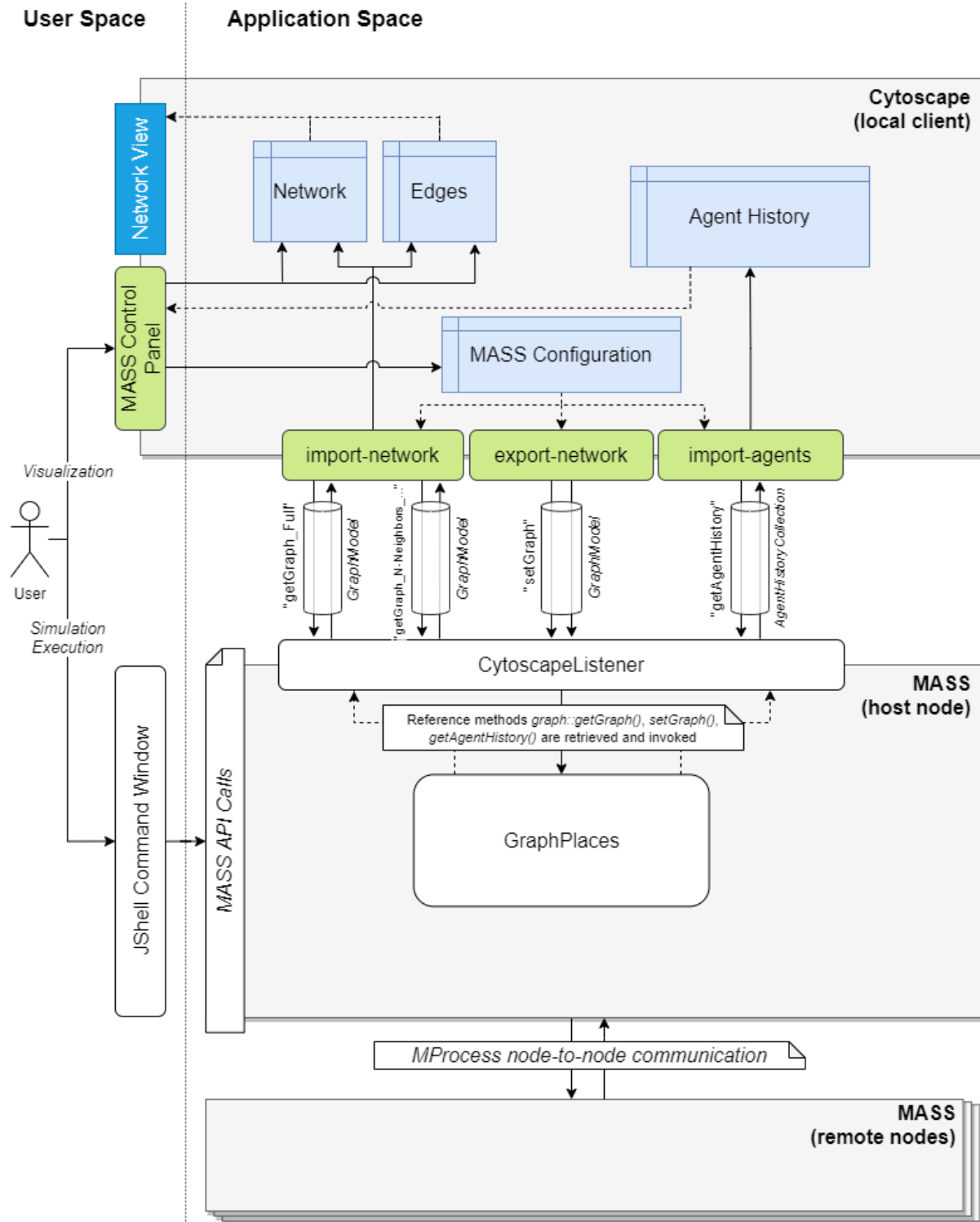


Figure 4.10: MASS-Cytoscape Integration Architecture Diagram

Due to the complexity of the MASS library, the MASS side of this architecture has been simplified to only classes in use in the MASS-Cytoscape integration.

To address the lack of bundle ordering in OSGi, in the MASS Control Panel we have implemented a reference to each of the data transfer plugins and the ability to test and reacquire the reference if it is lost. This ensures the MASS Control Panel is tolerant to whatever order in which the OSGi bundles are started. The panel furthermore maintains performance by only reacquiring the reference when it is needed. If the reference is not present and cannot be reacquired, then control returns to the user without an error message.

4.4.4 Visualizing Agent Movement (manipulating the Network View)

Visualizations in Cytoscape are all controlled by two factors: the *layout* and the *network view*. The *layout* defines the spatial orientation of network nodes and edges within the viewer. For this implementation, we have utilized the “Circular” layout which is a default layout in Cytoscape and appropriate for visualization of networks with large amounts of interconnectivity. Cytoscape allows for creation of custom layouts, but this is beyond the scope of this project. The *network view* then defines the visual attributes of each node or edge in the viewer. Customization of the network view allows for changes to attributes such as coloring, borders, line weights, and tooltips. These settings can be the same for all nodes or edges but can also be linked to individual fields in the input tables that contain categorical or numerical values. An example of this in the current implementation, we have linked the edge’s line weight attribute with a numerical field called *plot_edge_weight* in the edge table. The network view then parses this field to determine how thick each individual line should be. Importantly, this gives the MASS Control Panel a field to manipulate to create custom visualizations.

4.4.5 Partial Graph Streaming with N-Neighbors

Visualization of a partial graph is important in situations where the MASS cluster is operating on a graph too large to be stored in a single machine. To allow the visualization environment to support the scale of these graphs, we have implemented an optional “N-Neighbors” approach to graph retrieval from MASS. This requires the user to provide a centroid node id as well as determine the degrees of separation (DoS) that should be imported. DoS is interpreted as the number of neighbor rings that we would like to visualize. Shown in Figure 4.11, if the user selects

“1” DoS then the graph retrieval will bring back the centroid node as well as one ring of immediate neighbors. Specifying “2” DoS would bring back all the centroid’s neighbors as well as the neighbors of those nodes in the first ring, and so on.

To manage this request on the MASS side of the program, the MASS host node first receives the request, saves centroid and DoS information to MASSBase because method references may not accept parameters, and then invokes the `getGraphNNeighbors()` method on `GraphPlaces`. This method then iteratively queries the remote nodes for each DoS requested, passing a list each time to ensure only the required graph vertices are sent back. Importantly, this approach results in additional overhead from iterative network calls but ensures that the data returned to the host node is limited which is critical given that we are solving for situations where the graph is too large for a single machine.

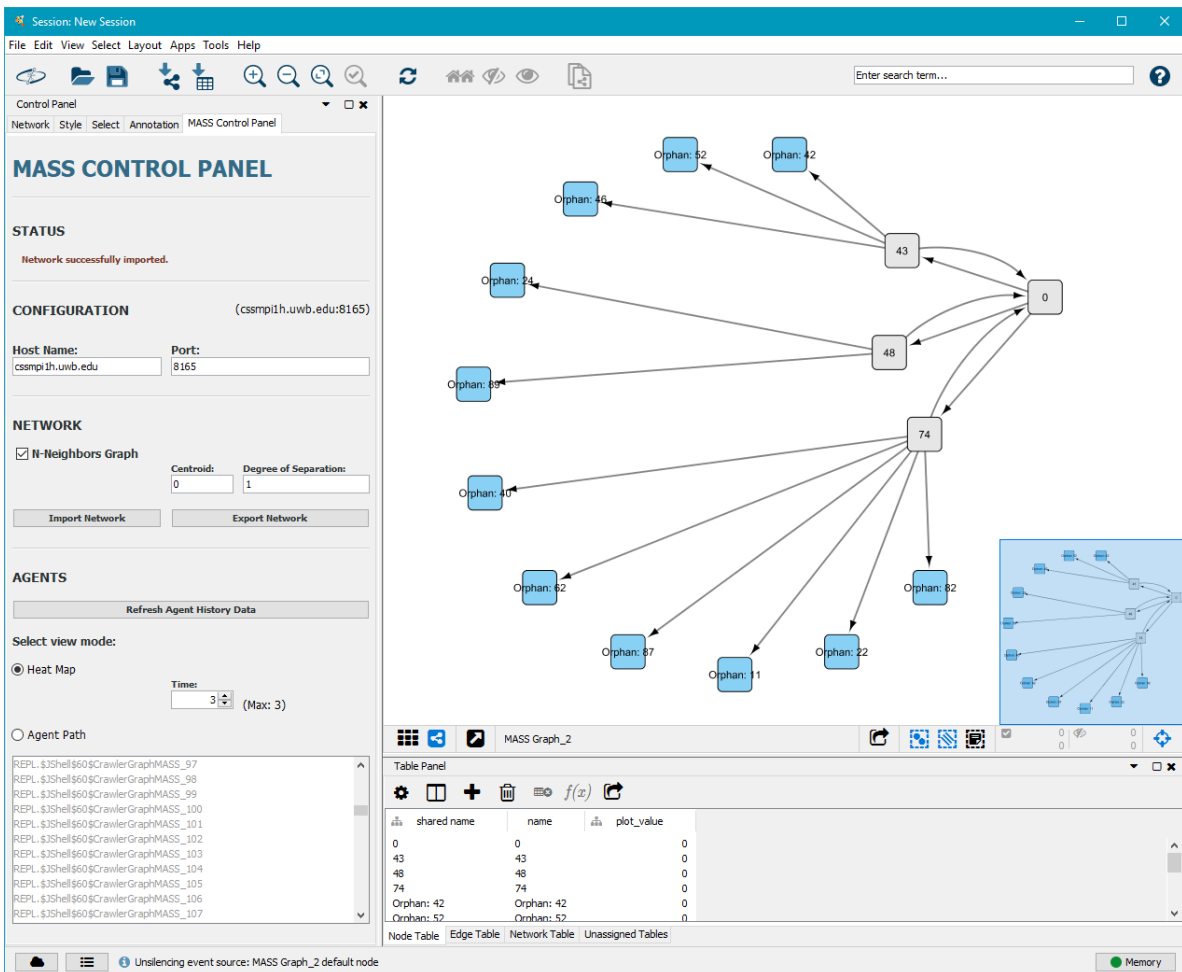


Figure 4.11: N-Neighbors Graph Retrieval (Centroid = 0, DoS = 1)

4.4 Usage Scenarios

The work presented here can be a useful tool for anyone working on graph-based simulations in MASS Java. However, due to the latency of required network communication this will not likely be an option for real-time visualization of an in-process simulation. Instead, these tools are most useful for those who are either (1) just learning MASS and exploring the functionality offered; (2) experienced in MASS and would like to use the tools for rapid prototyping and incremental debugging of the new graph simulations; and (3) reviewing the data of a completed simulation for accuracy and insights.

4.5 Contribution Summary

The functionality described in this implementation represents both original code as well as reimplementations and extensions of the work of other MASS Java contributors. In this section we summarize the scope of work completed in this project.

- **MASS Agent Tracking API.** This set of features are implemented from scratch in this project. This implementation includes three brand new classes (`AgentHistoryManager`, `AgentHistoryModel`, and `AgentHistoryCollection`) as well as associated changes to internal MASS code to invoke and manage these classes and supporting variables, such as “iteration” in `AgentsBase` for keeping track of simulation time.
- **Enhanced Simulation Controls (InMASS).** This set of features is a reimplementations of Nasser Alghamdi’s work [3] related to “Interactive MASS.” For incorporation into this project, we kept Nasser’s classes for state storage and dynamic class handling relatively unchanged from his original work. Instead, this project focused on how these classes are incorporated into the existing MASS architecture. Specifically, we have incorporated the InMASS features without impacting existing benchmark applications and left behind Nasser’s additional features related to monitoring and parallel instantiation.
- **MASS-Cytoscape Integration.** For implementation of the visualization layer in Cytoscape, this project extended Justin Gilroy’s original work [2] (implementing

GraphPlaces and creating the first Cytoscape plugins: *import-network* and *export-network*). Specifically, this project added two new Cytoscape plugins (*import-agents* and *agent-panel*) for importing and visualizing Agent data in Cytoscape, enhanced existing plugins to facilitate indirect component-to-component communication for improved user experience, and modified Justin's *import-network* plugin to facilitate partial graph streaming. Further, this work included several revisions to the MASS side of the integration to ensure GraphPlaces could handle new Agent data and partial graph requests from Cytoscape.

Chapter 5:

Evaluation

In this chapter we will (1) review performance and programmability of Agent Tracking API, (2) qualitatively compare added features with those of Repast Symphony, and (3) survey Agent visualizations added to Cytoscape.

5.1 Results

The following trials were performed using the csmmpi-h computing cluster located at the University of Washington Bothell. This cluster consists of 8 machines, each running an Intel Xeon Gold 6130 CPU at 2.10GHz and 20 gigabytes of system memory.

5.2.1 Agent Tracking API

Figure 6.1 shows the results of performance testing of the Agent Tracking API compared to the conventional History Passing technique (i.e., maintaining and passing history from parent to child directly through Agent spawn arguments) used by existing MASS applications. We utilized the graph-based Triangle Counting benchmark⁴ application for these tests and the execution times shown represent the average of five trials for each combination of input graph and data management technique. Finally, all tests were performed using a single computing node which removes the impact of network latency from the results.

These trials were conducted using graph sizes between 100 and 1,500 vertices with the maximum number of neighbor vertices per node scaling up with the size of the graph. As the size of the

⁴ In the Triangle Counting benchmark the simulation begins with an Agent on each node. All Agents then traverse the graph structure by first traveling twice to lower valued Place nodes, spawning child Agents when multiple nodes meet this criterion, and then trying to return to their original node. If the Agent is successful, then a triangle has been found.

graph increases, we also observe a corresponding exponential increase in the total number of Agents needed to perform the analysis. Due to variability in complexity from one graph to the next, and the management of Agent data being the primary point of interest for these features, we will use the total number of Agents to provide context to our results.

Analyzing the largest graphs tested in these trials (1,000 and 1,500 vertices), we observe approximately 1,700ms (~11-18%) slower performance in simulation execution times when using the Agent Tracking API. The difference in Agent data extraction times is then even more pronounced: in the case of 1,500 vertices, extraction took 5,844ms longer (or ~84% slower) when using the Agent Tracking API. For a better representation of these trends, simulation performance and Agent data extraction times with respect to the number of Agents are shown in Figure 5.2 and Figure 5.3, respectively.

| Vertices | Max Neighbor Vertices | Total # Agents | Total # Triangles | Agent History Passing | | Agent Tracking API | |
|----------|-----------------------|----------------|-------------------|--------------------------------|---------------------------------|--------------------------------|---------------------------------|
| | | | | Simulation Execution Time (ms) | Agent Data Extraction Time (ms) | Simulation Execution Time (ms) | Agent Data Extraction Time (ms) |
| 100 | 8 | 373 | 8 | 2,208 | 709 | 2,652 | 1,353 |
| 500 | 19 | 7,734 | 145 | 2,895 | 725 | 3,224 | 1,625 |
| 750 | 28 | 24,818 | 526 | 3,944 | 789 | 4,616 | 2,056 |
| 1,000 | 50 | 143,486 | 4,578 | 7,429 | 889 | 9,085 | 4,830 |
| 1,500 | 66 | 342,802 | 9,124 | 13,888 | 1,134 | 15,613 | 6,978 |

Figure 5.1: Summary of graph-based Triangle Counting performance testing

In Figure 5.2, we observe a slight decrease in simulation performance when using the Agent Tracking API. This increase in processing time is due to the added overhead from data capture methods invoked when Agents are spawning and migrating. This gap widens slightly when running in multi-node configurations due to added network latency, but the correlation between the two techniques remains consistent.

Conversely, in Figure 5.3, we observe a significant gap in performance between the two techniques regarding Agent data extraction time. There are two primary explanations for this disparity in performance:

1. The amount of data being returned to the user is significantly different. Using the Agent Tracking API returns all Agent data captured in the simulation while History Passing returns only data for Agents that are alive at the time of retrieval. Since this data was captured at the end of the simulation, the only Agents alive are those that found Triangles. In terms of the 1,500 vertices trial, this means that the History Passing technique returned data for 9,124 Agents while the Agent Tracking API approach returned results for 342,802 Agents.
2. The Agent Tracking API takes additional steps to clean, sort, and propagate the data upon retrieval. The impact of this extra data processing is somewhat limited in the Triangle Counting benchmark, however, because the three-step algorithm limits parent-child propagation to at most two levels. Longer simulations with more parent-child relationships are likely to see further increase in Agent data extraction times.

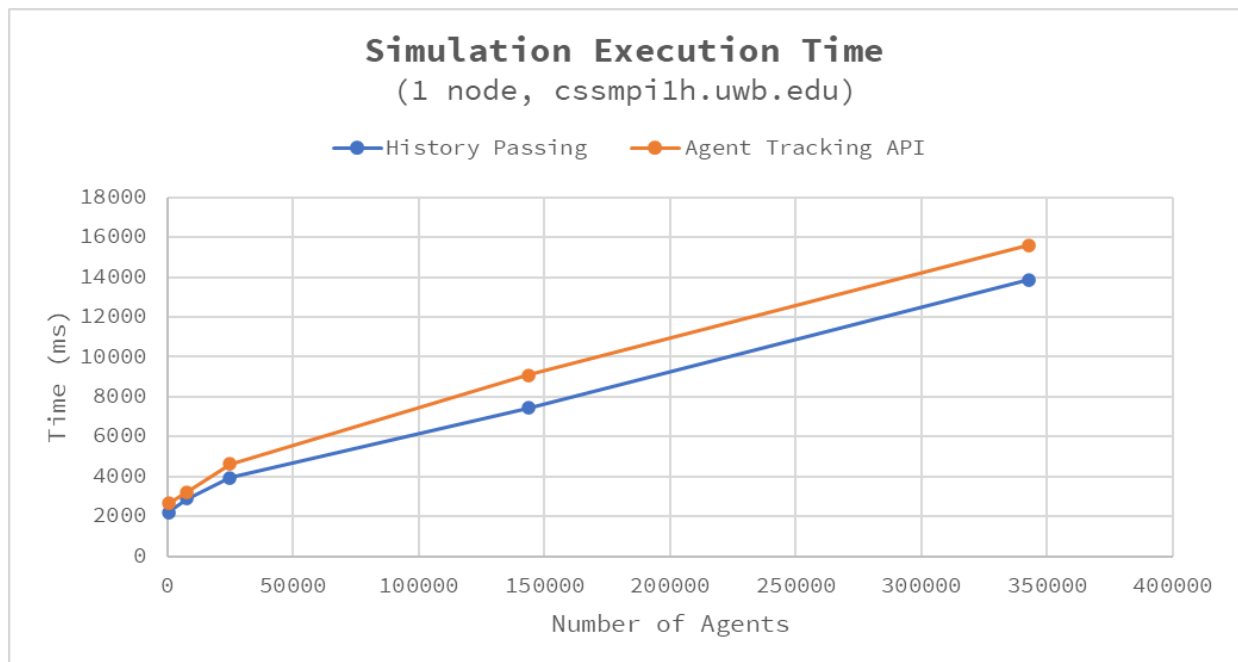


Figure 5.2: Simulation performance using History Passing and Agent Tracking API

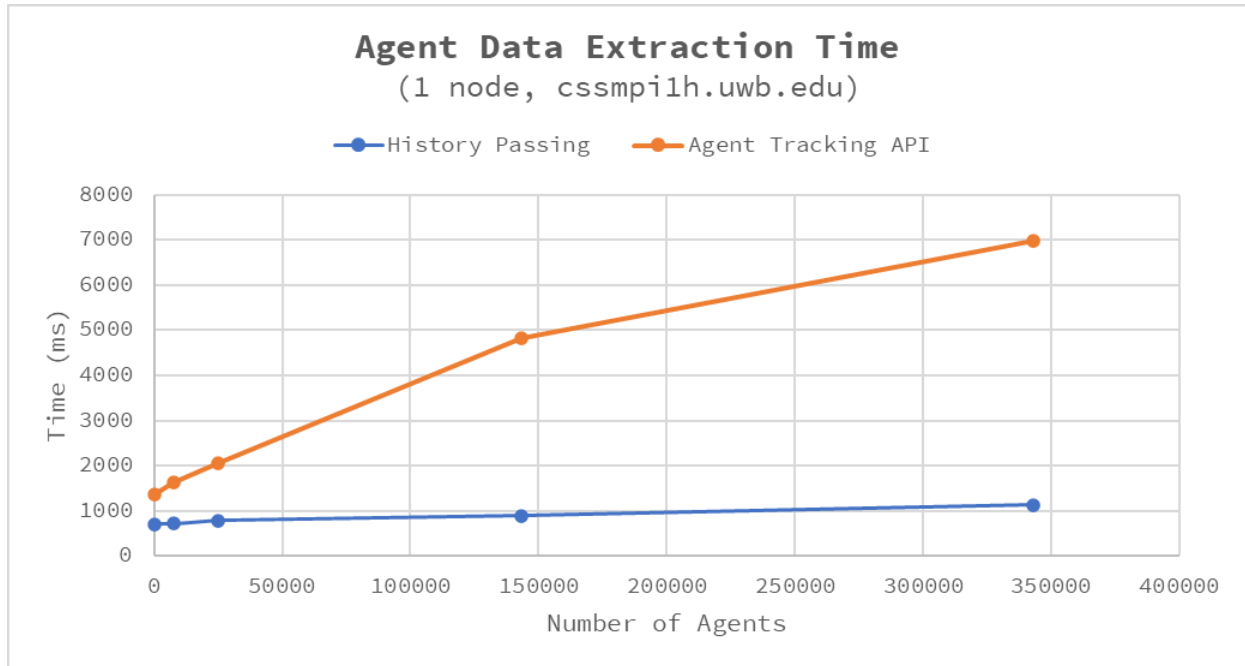


Figure 5.3: Agent data extraction time using History Passing and Agent Tracking API

With slower simulation performance and longer data extraction times, the discussion must then shift to what aspects of the Agent Tracking API make it a valuable addition to the MASS library. Figure 5.4 shows a qualitative comparison of the two techniques. First, as touched upon earlier, the History Passing method only works for Agents that are alive at the time of data extraction; whereas the Agent Tracking API manages history for all registered Agents and makes that data available at any point in execution. Second, the Agent Tracking API is arguably more intuitive for inexperienced MASS users. The History Passing method requires the user to have additional understanding of the MASS library: to understand that they can pass arguments from the parent to the child Agents to maintain a log of visits. In contrast, the Agent Tracking API can be initiated and then ignored until needed, allowing the user to instead focus on the logic of their application. Additionally, the Agent Tracking API will function the same way in all applications allowing knowledge of its use to be easily transferred to new projects. Finally, the Agent Tracking API shows a small reduction in LOC and, more importantly, a consolidation of those lines to one initialization statement and then a single block of code to retrieve and process the data.

| | Agent History Passing | Agent Tracking API |
|-----------------|------------------------------|---------------------------|
| Completeness | Only living Agents | All tracked Agents |
| Ease of Use | Less Intuitive | More Intuitive |
| Lines of Code * | 15 | 13 |

* Based on Graph-based Triangle Counting benchmark application.

Figure 5.4: Qualitative comparison of History Passing and Agent Tracking API

5.2.2 *Qualitative Comparison to Repast Symphony*

Figure 5.5 provides a comparison of key features between previous versions of MASS Java, this implementation, and competitor software Repast Symphony with regards to graph-based simulations. Compared features are then sub-divided into execution and visualization categories.

Repast Symphony provides a GUI for both execution and visualization of simulations through their integration with the Eclipse IDE. This integration also provides plugin support for two- and three-dimensional visualization of the simulation “Context”, which is like MASS’ Places, and Agents. These visualizations, known as “Projections”, are configured through the IDE before starting the simulation and are strictly synchronized with simulation execution. Repast Symphony does provide statistics and logging features for reviewing historical data, but the simulation itself is limited to only stepping forward through execution or running the simulation at full speed.

MASS, this implementation specifically, separates the execution and visualization aspects of the system into two windows. Simulation execution is handled through the command-line interface, which also enables forward and backward stepping through the inclusion of JShell with checkpointing and rollback features. While visualization is then managed separately through the Cytoscape GUI and MASS-specific extensions. Adding support for Agent visualization in this work brings MASS on par with Repast Symphony as far as what objects can be visualized, but our visualizations are still limited to two-dimensional views. Most importantly, the separation of view and execution concerns allows for the visualization to be completed asynchronously and the visualizations to be adjusted as desired through the Cytoscape GUI

without the need to pre-configure or restart a simulation. This is particularly useful in long-running simulations where the need for visualization was not considered ahead of time or when exploring which visualizations may best fit the application.

| | | MASS (Graphs) | MASS (Graphs) w/ New Features | Repast Symphony |
|----------------------|---------------|------------------|----------------------------------|------------------|
| Execution | Interface | CLI | CLI | GUI |
| | Flow Control | Forward Complete | Forward / Backward Stepping | Forward Stepping |
| Visualization | Interface | GUI | GUI | GUI |
| | Objects | Graph Only | Graph + Agents | Graph + Agents |
| | Type | 2D | 2D | 2D / 3D |
| | Synchronicity | Retrieved at end | Asynchronous | Synchronous |
| | Flexibility | Set at start | As needed, selectable | Set at start |

Figure 5.5: Qualitative comparison Repast Symphony versus MASS

5.2.3 Agent Visualizations in Cytoscape

To facilitate user exploration and simulation understanding, we have implemented two Agent visualizations in Cytoscape: *Agent Path* and *Heat Map*. Both visualizations are generated using the MASS Control Panel and each can be freely manipulated in the network view.

Agent Path is shown in Figure 5.6 and provides the user with the ability to review the complete path of any individual Agent. The more recent movements are represented with a darker node and thicker edge. If an Agent ever traverses an edge that does not exist in the edge table, then a dashed line is created to signal the issue to the user. This view is particularly useful in simulations, such as Triangle Counting, where the pattern of Agent movement determines the success of the application. In Figure 5.6, we see the selected Agent was successful in finding a triangle because the Agent was able to return to its origin.

The *Heat Map* visualization, shown in Figure 5.7, provides the user a representation of all Agents active in the simulation at a point in time with darker nodes representing higher

concentrations of Agents. The user is then able to cycle through the time variable of the simulation to observe movement patterns of the entire group. This visualization is best applied to use cases such as in the Sugarscape benchmark, in which we seek to observe aggregate movement patterns centralized around particular points of interest in the simulation.

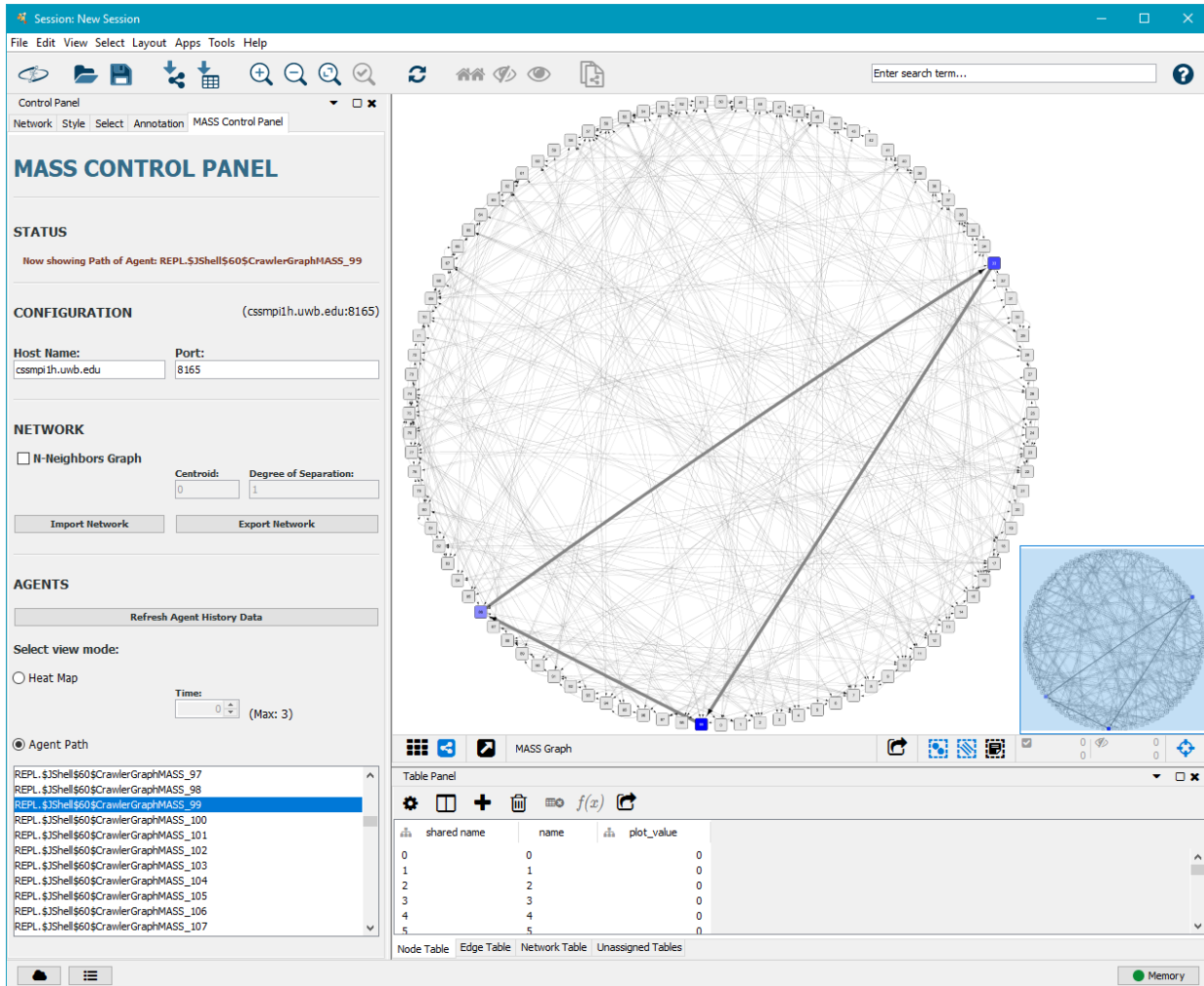


Figure 5.6: Agent Path Visualization

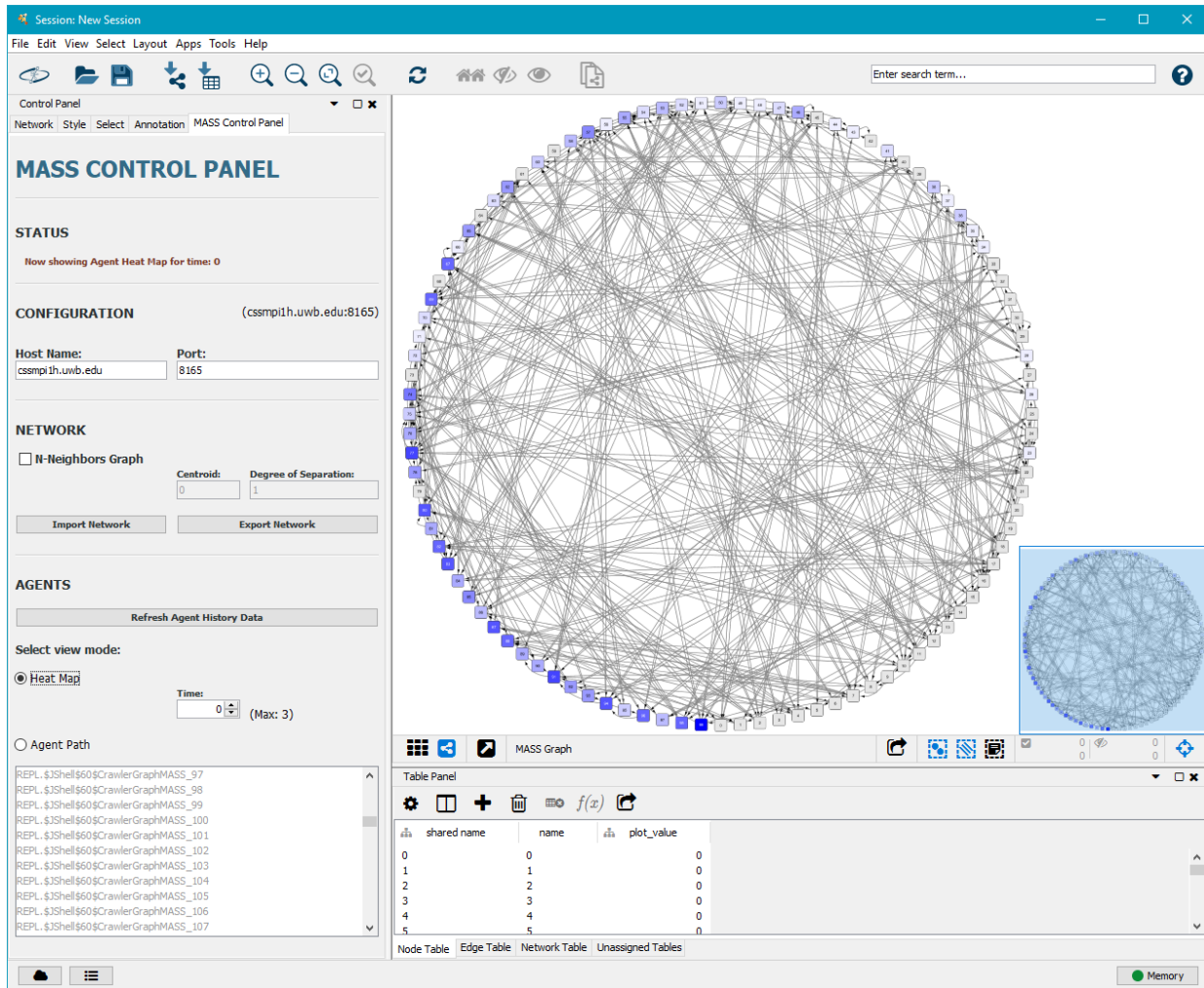


Figure 5.7: Heat Map Visualization

5.2 Discussion

The features developed in this work were successful at improving approachability and programmability in MASS, particularly when working with graph-based applications. The addition of Agent Tracking API provides a simple and intuitive mechanism to track and retrieve complete Agent history for all applications. This feature may not be appropriate for performant applications, however, due to increased processing time when retrieving history from the cluster. JShell integration with checkpointing and rollback functionality allows greater control over execution of MASS simulations while Cytoscape visualization tools enable the user to easily visualize and explore their results. Together, these features offer prospective MASS users with a lower knowledge barrier to entry by providing the ability to rapidly experiment with their

simulation and then review results of their experimentation in real-time without needing to recompile, redistribute, and restart the simulation with each iterative trial.

The current implementation has several limitations, we describe three of those limitations here and discuss how each of these issues may be mitigated by the user or addressed through future work on the MASS library.

1. Java's JShell provides a CLI which may be unfamiliar or awkward for users just getting started. This difficulty may be particularly acute for users who normally rely heavily on their IDE for auto-correction and suggestion support. Although these features exist in JShell their use is not as smooth as in most IDEs. These challenges will fade over time, however, as users become more familiar with the JShell interface and available features, such as the `"/open"` command which allows the user to open and run a pre-written text file. This command is particularly powerful in that it allows the user to continue developing in their chosen IDE and then simply run the `"/open"` command on their file when they are ready to test execution.
2. The Agent Tracking API allows for registration and retrieval of data by calling for individual Agents or an entire class of Agent. At this time, however, the Agent History propagation from the parent feature is only functional when tracking an entire class and then retrieving history for all Agents tracked. The limitation exists because the history of Agent visits is distributed amongst the Places and the propagation occurs upon retrieval, once all history data has been coalesced on the master node: if the data is not present on the master process, then it cannot be propagated. Further, the current MASS-Cytoscape integration only allows for retrieval of full Agent history. Addressing these limitations may lead to improved performance in instances where only a selection of Agents, for example "successful" Agents, is required.
3. Agent visualization in Cytoscape does not allow for much customization by the user and has only been optimized for instances where the user would like to track a single class of Agents. Further, the color gradient is based on only fifteen shades of the base color

which leads to instances where a node may show no Agents present, even when they are, because the number of Agents on the Place is not significant enough with respect to the most populated Places.

Chapter 6:

Conclusion

In this project we successfully implemented a set of new tools and functionality for MASS Java users to enable more rapid development and exploration when building graph-based ABM simulations. We accomplished this by introducing new APIs for tracking Agent data, incorporating an interface for line-by-line control of a running simulation, and expanding integration of Cytoscape for visualization of graph-based Places and associated Agents. Verification of this work was done by examining each major deliverable using the graph-based Triangle Counting benchmark application which showed that the new functionality provided results consistent with previous methods and did so with minimal impact on simulation performance, though greater impact on Agent data extraction times.

6.1 Future Work

In addition to addressing some of the limitations already discussed, there are many other opportunities to extend or optimize this work further. Some of these opportunities are:

- Rework MASS-Cytoscape communication to use Aeron or other UDP-based strategy to pulse MASS data to Cytoscape plugins. This could include both simulation data, Places and Agents, as well as MASS system status including uptime and processing status. This will be dependent upon OSGi framework's ability to maintain a persistent task to hold the listener port open. The Cytoscape plugins in the current implementation hold control of the system while they are executing, though there may be another component structure that would allow the communication task to run concurrently with other plugins.
- Expansion of Cytoscape visualization capabilities to non-graph-based simulations through implementation of custom layouts and modifications to graph import plugins.

As an example, two-dimensional places could be represented in Cytoscape utilizing a grid layout with edges between local neighbors.

- Leveraging added ability to checkpoint simulation to file, further extension of this feature could be to add functionality to reload complete simulation state from the saved file. This complete reinitialization from file would enable users to come back to simulations after restarting MASS and to share simulations with other interested users.
- Additional quality of life improvements to the MASS Control Panel in Cytoscape to refine the user experience and improve flexibility of the tool. For example, it would be beneficial for users to have the ability to filter the Agent list to only Agents alive at a particular time or to only those Agents which have been marked successful. (To that end, MASS implementation would also need to be expanded to accept success scenarios or a way to mark a particular Agent successful in the user program.)
- User experience with the MASS Control Panel, as well as the rest of this implementation, has not been adequately studied with new or active MASS Java users. Therefore, usability studies and subsequent feature iterations are needed to better understand and refine the impact of this work.
- Cytoscape visualizations in this project have been created primarily to show Agent movement over a static graph. In real-world situations, however, the most interesting aspects of the simulation may be related to the evolving graph itself, as in social networks where nodes and edges are constantly being created and removed. To that end, additional work could be done to capture graph structure incrementally and then import that data to Cytoscape to allow visualization of the evolution of the graphs alongside associated Agents.

References

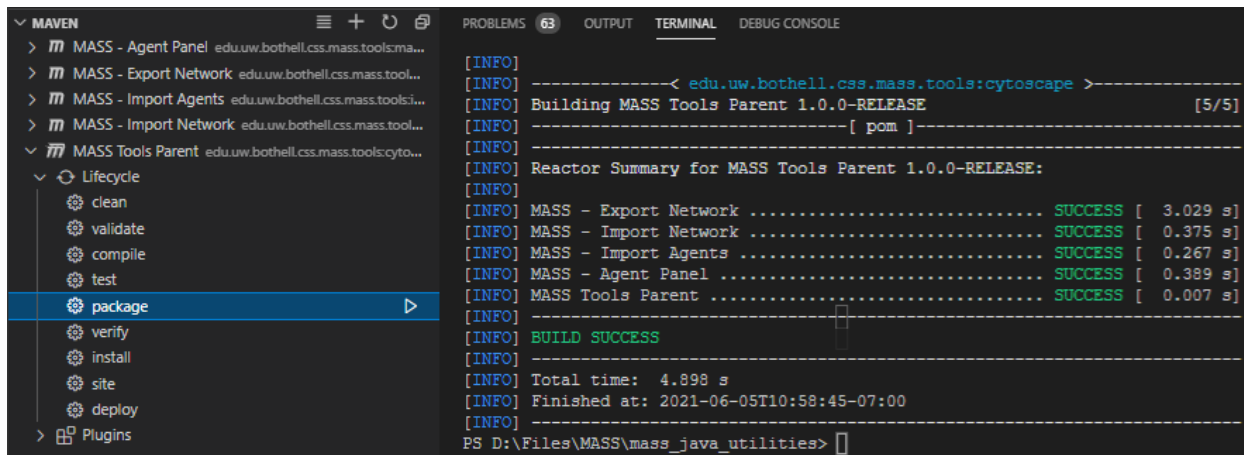
- [1] University of Washington. MASS Java Manual, 2016.
<https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf>
- [2] Gilroy, Justin, "Dynamic Graph Construction and Maintenance," (Master's paper). University of Washington, Bothell, WA, 2020.
- [3] Alghamdi, Nasser, "Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System," (Master's paper). University of Washington, Bothell, WA, 2020.
- [4] Ono, K. (2018). Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization. Retrieved July 27, 2020, from <https://cytoscape.org/>
- [5] Fukuda, M. (2020). Distributed Systems Laboratory (DSL). Retrieved July 30, 2020, from <http://depts.washington.edu/dslab>
- [6] Horni, A., Nagel, K. and Axhausen, K.W. (eds.) 2016 The Multi-Agent Transport Simulation MATSim. London: Ubiquity Press. DOI: <http://dx.doi.org/10.5334/baw>. License: CC-BY 4.0
- [7] "HIPPIE: Human Integrated Protein-Protein Interaction rEference" Cbdrm01.zdv.unimainz. de, 2020. [Online]. Available: http://cbdrm-01.zdv.unimainz.de/~mschaefer/hippie/hippie_current.txt. [Accessed: 29- July- 2020].
- [8] Baeldung. "Introduction to OSGi." 14 Aug. 2019. Web. 17 Dec. 2020.
- [9] Davis, Delmar B.; Featherston, Jonathan; Vo, Hoa N.; Fukuda, Munehiro; Asuncion, Hazeline U. 2018. "Data Provenance for Agent-Based Models in a Distributed Memory" Informatics 5, no. 2: 18. <https://doi.org/10.3390/informatics5020018>
- [10] North, MJ, NT Collier, J Ozik, E Tatara, M Altaweel, CM Macal, M Bragen, and P Sydelko, "Complex Adaptive Systems Modeling with Repast Symphony", Complex Adaptive Systems Modeling, Springer, Heidelberg, FRG (2013). <https://doi.org/10.1186/2194-3206-1-3>.
- [11] North, Collier, "Repast JAVA Getting Started", [Repast.github.io](https://repast.github.io), 2021. Retrieved May 20, 2021, from <https://repast.github.io/docs/RepastJavaGettingStarted.pdf>
- [12] Using layouts. (n.d.). Retrieved May 23, 2021, from <https://blog.js.cytoscape.org/2020/05/11/layouts/>
- [13] Network Layout. (n.d.). Retrieved May 23, 2021, from <https://cytoscape.org/cytoscape-tutorials/protocols/network-layout/#/7>

Appendix A: Developer Guide

A.1: Cytoscape Installation

The following installation steps are based on Windows 10 OS but should be similar on other platforms.

1. **Download and install Cytoscape** on local machine from [Cytoscape.org](https://cytoscape.org). (This project was completed using Cytoscape version 3.7.2.)
2. **Download Cytoscape Plugins from Bitbucket**, all Cytoscape extensions can be found in the `mass_java_utilities` repository. Project work completed in `develop_dblashaw` branch but should be merged to `master` soon.
(https://bitbucket.org/mass_utility_developers/mass_java_utilities/src/master/)
3. **Build and install Cytoscape Plugins.**
From the command line or within chosen IDE (pictured in Figure A.1 using VS Code) ‘package’ each of the plugins.



```
MAVEN
> m MASS - Agent Panel edu.uw.bothell.css.mass.tools:ma...
> m MASS - Export Network edu.uw.bothell.css.mass.tool...
> m MASS - Import Agents edu.uw.bothell.css.mass.tools:i...
> m MASS - Import Network edu.uw.bothell.css.mass.tool...
> m MASS Tools Parent edu.uw.bothell.css.mass.tools:cyto...
Lifecycle
  clean
  validate
  compile
  test
  package
  verify
  install
  site
  deploy
  Plugins

PROBLEMS 63 OUTPUT TERMINAL DEBUG CONSOLE
[INFO] -----< edu.uw.bothell.css.mass.tools:cytoscape >-----
[INFO] Building MASS Tools Parent 1.0.0-RELEASE [5/5]
[INFO] ----- [ pom ]-----
[INFO]
[INFO] Reactor Summary for MASS Tools Parent 1.0.0-RELEASE:
[INFO]
[INFO] MASS - Export Network ..... SUCCESS [ 3.029 s]
[INFO] MASS - Import Network ..... SUCCESS [ 0.375 s]
[INFO] MASS - Import Agents ..... SUCCESS [ 0.267 s]
[INFO] MASS - Agent Panel ..... SUCCESS [ 0.389 s]
[INFO] MASS Tools Parent ..... SUCCESS [ 0.007 s]
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 4.898 s
[INFO] Finished at: 2021-06-05T10:58:45-07:00
[INFO]
PS D:\Files\MASS\mass_java_utilities>
```

Figure A.1: Packaging MASS Java Utilities in VS Code

Then move the newly packaged .jar files to the Cytoscape apps folder, as shown in Figure A.2.

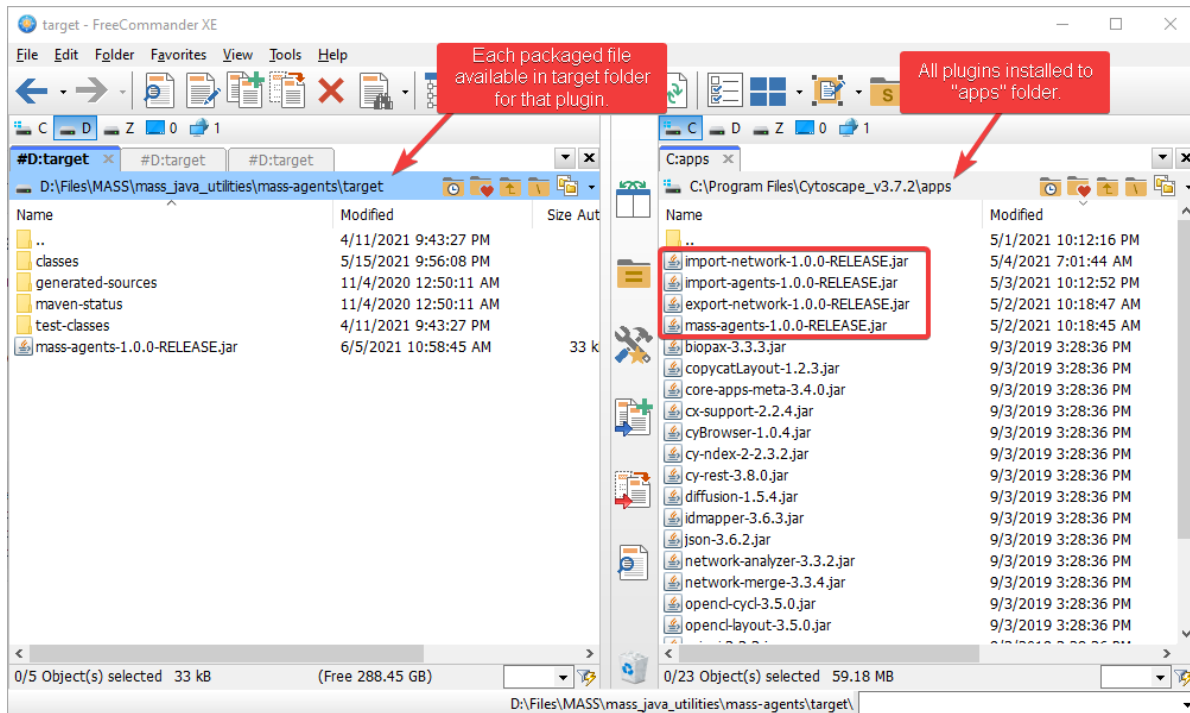


Figure A.2: Installing Cytoscape plugins to Apps directory

All plugins inside Cytoscape apps folder should be automatically started the next time Cytoscape desktop application is initialized. (If MASS Control Panel is shown upon Cytoscape initialization, then continue to section A.2 *Configuring Cytoscape Visualizations.*)

If for some reason the plugins do not start automatically, then open the “App Manager” inside Cytoscape (**Apps** → **App Manager...**) and then “Install from File...” (shown in Figure A.3).

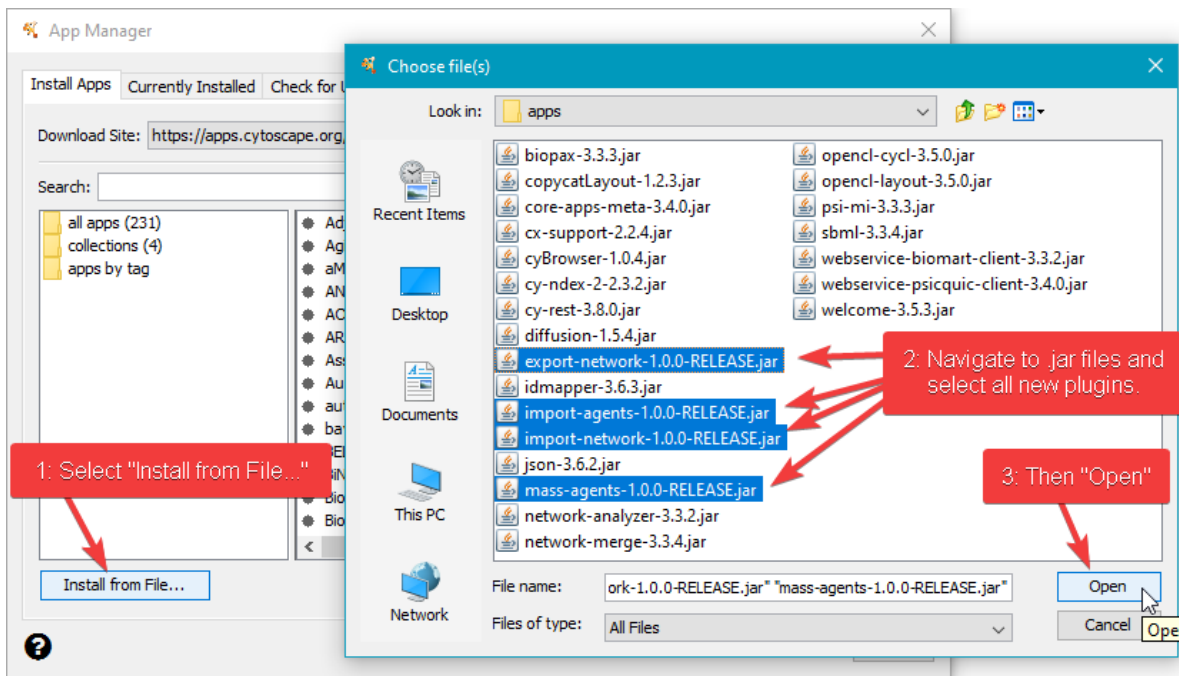


Figure A.3: Installing Cytoscape plugins using App Manager

At this point, the “Currently Installed” tab of the App Manager (see Figure A.4) should show the new plugins and the MASS Control Panel should be opened in the main Cytoscape window.

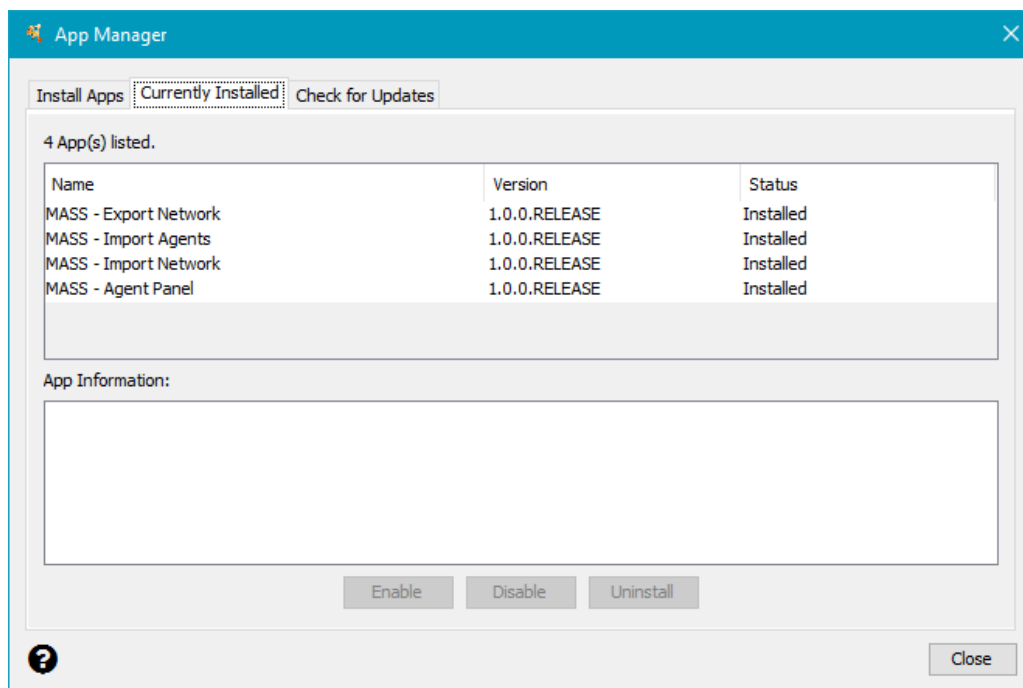


Figure A.4: Output after successful install with Cytoscape App Manager

A.2: Configuring Cytoscape Visualizations

Visualizations inside Cytoscape are governed by two factors: the **Network View** and the **Layout**. The *Network View* holds information on the visual representation of the Node and Edge data (e.g., color, size, thickness, borders, tooltips) and the *Layout* provides Cytoscape instructions on how to then arrange those Nodes and Edges within the viewer.

A.2.1: Network View

At the time of this writing, there is no way for the user to directly modify Network View behavior in the GUI. Customization of the Network View instead must be done by directly editing plugin source code.

- (1) To change the mapping of Network/Edge columns to visualization attributes, e.g., line width, node color, then modify the `createView()` method in `CreateNetworkTask` class of *import-network* plugin.
- (2) To update how mapped columns are updated or to add additional visualizations options, then modify the `updateNodePlotValues()` and/or `updateEdgePlotValues()` methods in `MASSControlPanel` class of *mass-agents* plugin.

A.2.2: Layouts

Cytoscape comes with several built-in Layouts and provides users with the ability to create customized Layouts, also in the form of plugins. Custom Layouts were outside of scope for this project, so the built-in “Circular Layout” was used for demonstration purposes.

Set default layout by...

- (1) Import a graph into memory (**tip:** enter “test” as MASS Host Name and then click “Import Network” button to retrieve a randomly generated graph)
- (2) Select **Layout** → **Settings...** from the Menu bar.
- (3) Switch to the **Preferred Layout** tab and make selection (shown in Figure A.5).

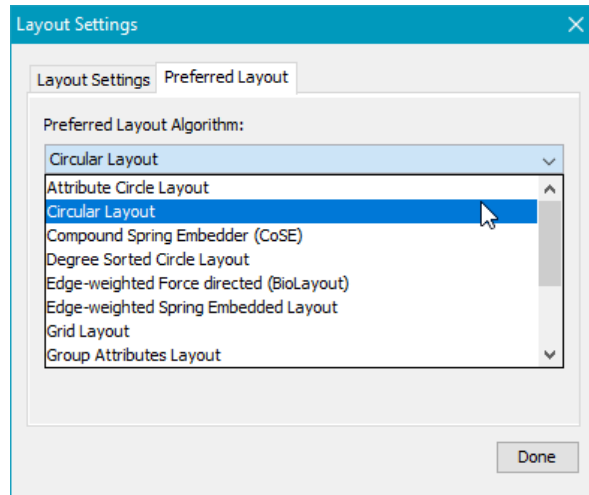


Figure A.5: Setting Preferred Layout in Cytoscape

Apply default layout using the  button in the top ribbon or by pressing F5.

For more information on Layouts see references [12] and [13].

A.3: Using MASS Control Panel

A.3.1: Application Requirements

For visualization of Graph and Agent data in Cytoscape as shown in this project the following conditions must be met:

- (1) **MASS must be utilizing InMASS** to enable paused simulation and incremental execution.

The code snippet in Figure A.6 shows how to compile and run the MASS Java library using InMASS. Once complete, use the “/exit” command to shutdown InMASS.

```
cd ~
cd mass_java_core
mvn clean install
cp ./target/mass-core.jar .
java -cp mass-core.jar InMASS

// ...user interaction with InMASS window...

/exit
```

Figure A.6: Compiling and starting InMASS

- (2) User's MASS application must **create an instance of CytoscapeListener class** to field requests to/from Cytoscape. This CytoscapeListener receives requests from Cytoscape, such as "setGraph" and "getAgentHistory," and then retrieves and invokes corresponding reference methods from the GraphPlaces instance which was provided in its constructor.

The code snippet in Figure A.7 shows the initialization and shutdown of the CytoscapeListener class. Note that the current implementation will only work in conjunction with the GraphPlaces class, support for other Places types may be a goal of future related work. (Code snippet taken from CountTrianglesGraphMASS benchmark application.)

```
Places network = new GraphPlaces(1, NodeGraphMASS.class.getName(), nodes.size());
MASSListener listener = new CytoscapeListener((GraphPlaces) network);

// ...user's MASS application code

if (listener != null) {
    listener.finish();
}
```

Figure A.7: Starting and stopping CytoscapeListener in MASS simulation

- (3) User's MASS application must **use Agent Tracking API** to track Agent movement. **Recommended:** use the AGENT_TRACE_REGISTER_CLASS function to ensure all Agent movement is captured.

In Figure A.8 code snippet, we show how the user calls the Places.callAll() method using the Agent Tracing function IDs to register a class of Agents tracking and then we demonstrate the process for retrieving that data at the end of the application. Note though, to enable visualization in Cytoscape the user only needs to *register* the Agents for tracking. Extract data to user application only when Agent data is useful for programmatic parsing of results. (Code snippet taken from CountTrianglesGraphMASS benchmark application.)

```

Places network = new GraphPlaces(1, NodeGraphMASS.class.getName(), nodes.size());

// Register Agents for tracking
network.callAll(network.AGENT_TRACE_REGISTER_CLASS,
                (Object) CrawlerGraphMASS.class.getName());

// ...user's MASS application code

// Retrieve Agent Trace results
Object[] agentsCallAllObjs = new Object[vertices.size()];
Object[] agentTraceResults =
    (Object[]) network.callAll(network.AGENT_TRACE_GET, agentsCallAllObjs);

AgentHistoryCollection collection = (AgentHistoryCollection) agentTraceResults[0];

for (AgentHistoryModel model : collection.getModels()) {
    // ...do something with the results
}

```

Figure A.8: Registering and retrieving Agent data using Agent Tracking API

A.3.2: MASS Configuration

To begin visualizing MASS simulation data in Cytoscape, we must first update the MASS Control Panel CONFIGURATION section with the location of the running MASS simulation, shown in Figure A.9.

To update a field in the control panel simply click into the text box, update that entry, and then click or tab out of the field. Upon exiting the field, you should then see the current configuration and the STATUS message are updated.

By default, the MASS Control Panel and CytoscapeListener default to port 8165. The port number can be changed in the MASS Control Panel through the associated “Port” field and in the CytoscapeListener by adding port number to the constructor parameters.

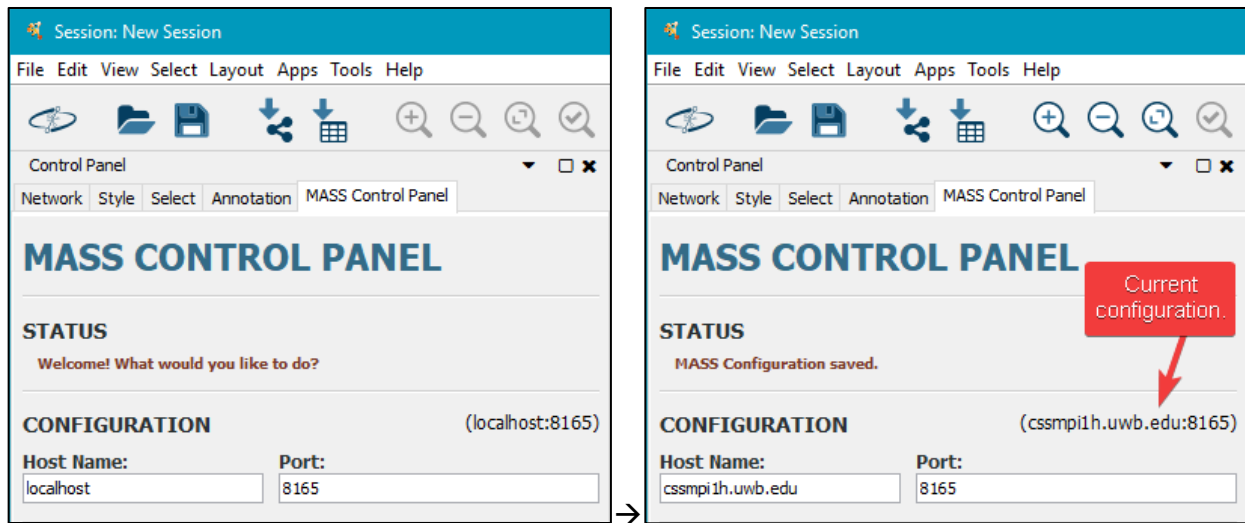


Figure A.9: Updating MASS Configuration in the MASS Control Panel

A.3.3: Network (Places) Data

To import Places data into Cytoscape, the user can use the NETWORK section of the MASS Control Panel, shown in Figure A.10.

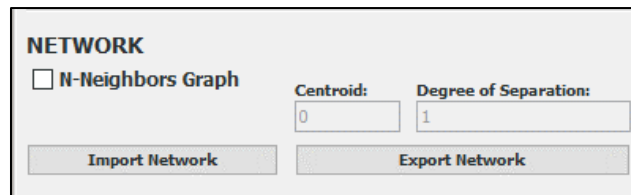


Figure A.10: Import and Export Network features in the MASS Control Panel

In this section, the user may choose to import a partial graph using the “N-Neighbors Graph” checkbox (along with subsequent “Centroid” and “Degree of Separation” fields) or may import the entire graph by leaving the checkbox blank and pressing the **Import Network** button.

After successfully importing graph data into Cytoscape all vertices will be stacked on top of one another. At this time, press the F5 key or use the “Apply Preferred Layout” button to organize the vertices and edges based on selected default layout.

A.3.4: Agents Data

Agent data is imported and manipulated in Cytoscape using the AGENTS section of the MASS Control Panel, shown in Figure A.11.

To begin, user must press the **Refresh Agent History Data** button to create an import-agent task and retrieve data from MASS. Once complete, the visualization should automatically update to the selected visualization type and the list of Agents underneath the “Agent Path” radio button will populate with the name of all Agents imported.

Now that Agent data is available in Cytoscape memory, the user may select either **Heat Map** or **Agent Path** visualization options using the radio buttons. Once visualization type has been selected, the time or Agent list becomes selectable for further exploration of the Agent data.

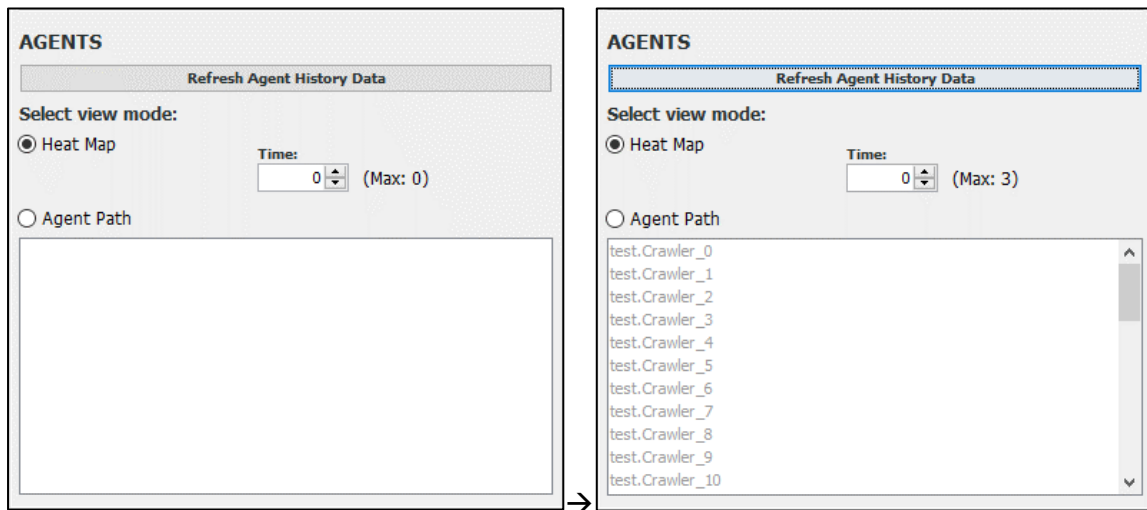


Figure A.11: Importing and using Agent data in the MASS Control Panel

A.3.5: Reviewing Raw Data

Raw data imported into Cytoscape is always available in the “Table Panel” at the bottom of the Cytoscape window. (If its not there by default, the Table Panel can be opened by going to the menu bar and selecting **View → Show Table Panel.**)

Within the Table Panel (shown in Figure A.12), the raw data is partitioned based on the type of information: MASS vertices and edge information are in the Node Table and Edge Table, respectively; the Network Table maintains the name of the most recently imported graph from MASS; and the Unassigned Tables tab has all other tables that do not fit into the Cytoscape default tables mentioned above. In this implementation, the Unassigned Tables section has the Agent History tables as well as the MASS Configuration table.

| UID | Agent_Name | Time | Place | Alive | Successful |
|-------------|----------------|------|-------|-------------------------------------|--------------------------|
| test.Cra... | test.Crawler_0 | 0 | 2 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_0 | 1 | 14 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_0 | 2 | 5 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_0 | 3 | 13 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_1 | 0 | 7 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_1 | 1 | 14 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_1 | 2 | 4 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_1 | 3 | 7 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_2 | 0 | 4 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |
| test.Cra... | test.Crawler_2 | 1 | 1 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |

Figure A.12: Unassigned Tables shown in Cytoscape’s Table Panel

Note that the “Alive” and “Successful” fields have been created in the Cytoscape panel as placeholders. These fields are not currently used by MASS applications and, therefore, do not offer anything meaningful to the visualization layer until they are fully implemented in MASS.