Autumn Report 2019

MASS Monitoring Tool
Nasser Alghamdi

Master of Science in Computer Science & Software Engineering
University of Washington | Bothell
December 12, 2019

Project Committee:
Munehiro Fukuda, Ph.D., Committee Chair
Arnie Lund, Ph.D., Committee Member
Kelvin Sung, Ph.D., Committee Member
Contents

1 Introduction 3
   1.1 MASS .................................................. 3
   1.2 InMASS .................................................. 4

2 Related Work 5
   2.1 MASS Java Debugger .................................... 6

3 Goals 6

4 Architecture 6
   4.1 Components ........................................... 8
   4.2 Connectors .............................................. 9
   4.3 Data Elements .......................................... 9

5 Implementation 11
   5.1 Handlers and MonitorConnector Classes ................. 12
   5.2 Monitoring Process .................................... 12
   5.3 Selection Mechanism ................................... 13
   5.4 Querying Mechanism ................................... 14
   5.5 UI Code .................................................. 14

6 Non-Interactive Applications 15

7 User Manual 17
   7.1 Sample .................................................. 18

8 Next Steps 29

9 Conclusion 30

References 32
1 Introduction

This report is the first part of ongoing work for a capstone project, *Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System*. It lays the foundation for the next steps in the project. The report contains nine sections: introduction, related work, goals, architecture, implementation, non-interactive applications, user manual, next steps, and conclusion.

In this section, we discuss the organization of this report and we provide a high-level overview for the MASS library. The related work section discusses monitoring in distributed systems, previous work within DSLab, and the correlation to MASS Monitoring Tool. Next, we have the goals section discussing the objectives and goals of the tool. The architecture section discusses the functions and uses of components, connectors, and data elements of the system at an architectural level. The implementation section discusses the construction aspects of the architecture. Next, non-interactive applications section illustrates how the monitoring tool can be used for standalone applications, too. User Manual section explains the setup process for the tool. Also, it has a sample that simplifies, not to say it is complex, the setup process. In the next steps section, we discuss our plan for the next quarter. In the conclusion section, we discuss the limitations and future work opportunities.

1.1 MASS

MASS facilitates a programming model for parallelizing a wide range of agent-based micro-simulation programs including physical simulation (e.g., wave dissemination and molecular dynamics) as well as social, behavioral, and economic simulation (e.g., social network, artificial life, and bank/investor/firm). It also facilitates agent-based data sciences and optimization such as ant colonial optimization and particle swarm optimization. The parallelization process in MASS is built around two main classes: Places and Agents. Places are elements that are statically allocated to different threads across multiple computing nodes. On the other hand, agents are execution entities that autonomously move over different processes. The library exposes a set of APIs that conceal the complexity of thread and process spawning, mapping, execution, communication, and termination under the hood, yet the end-user must learn how to use them [1].

<table>
<thead>
<tr>
<th>MASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASS is an abbreviation for for <em>Multi-Agent Spatial Simulation</em> library which is developed by Distributed Systems Laboratory (DSLab) at Computing and Software Systems Department, University of Washington</td>
</tr>
</tbody>
</table>

Both MASS paper [1] and MASS User Manual [2] follow theoretical and technical approaches for explaining MASS programming model, respectively. In this report, we reference the model from a technical point of view. The MASS library exposes five classes: (1) MASS,
MASS class has two distinct functions: one for initializing the cluster and the other for terminating. Internally, it maintains connections between the node that user interacting with and the remote daemon process or processes. Place and Agent are intended to be extended by the user to define the place and agent models. Extension classes maintains data and actions which both represent the application state.

**Application State In MASS Library**
An application state in context of MASS library is represented by Place & Agent components. Application state includes data and actions defined by the user.

Places and Agents are controlling units in which they provide a set of functions that operate on place and agent instances respectively. Operations performed by Places class include creating elements, exchanging messages, and invoking methods on Place instances. While Agents class operations include creating, spawning, killing, migrating entities, and invoking methods on Agent instances.

### 1.2 InMASS

**InMASS**
InMASS is an acronym for *Interactive computing feature for MASS library*. Initial implementation was the focus of CSS600, Independent Study class, in Spring 2019.

InMASS benefits from both MASS communication model and JShell Tool functionalities. Basically, it is a wrapper around JShell Tool. First, it injects an initialization code into JShell. Next, a JShell prompt is returned where the user can interact with MASS components: MASS, Places, Agents, Place, and Agent classes. Finally, cleanup is automatically called on an exit event. It uses MASS communication functionalities to disseminate any class that is defined by the user at runtime.

**JShell Tool**
JShell Tool is an interactive shell for Java programming language which is shipped with JDK, Java Development Kit, version 9 or higher. It is implemented on top of the JShell APIs[3].

(2) Place, (3) Agent, (4) Places, and (5) Agents.
1.2.1 Motivation

My first use of MASS library was in the CSS534, Parallel Programming in Grid and Cloud 2018 class, for parallelizing an optimization algorithm, simulated annealing, to solve the Travelling Salesman Problem (TSP). As a new parallel-programming user, MASS has a smooth learning curve[2] that draws my attention to MASS library and its intuitive programming model.

However, tracking the application state and tuning the code was a tiresome process and it is the case for distributed systems [4, Chapter 14]. In MASS, the user must log the application state through logging files each resides on a different node. Thereafter, the user searches each file to drive a decision upon that information. Next, the user makes changes, compile, and run the application again. This cycle keeps going until the user reaches the expected outcome.

In conjunction with ongoing efforts in developing the library at the DSLab, I found an opportunity to incorporate interactive computing features, thanks to professor Fukuda’s lecture at the Faculty Seminar (Winter 2019). By doing so, I believe that it will reinforce MASS’s usability also it will accelerate the development process by providing results to developers or testers much faster than the compile-then-run approach. In Spring 2019, we designed and implemented InMASS. Also, we submitted a capstone project proposal committed to meet InMASS full potential. The proposal, Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System, aims to add two features for InMASS, one of which is discussed in this report and the other is planned for next quarter (Winter 2020).

2 Related Work

In the distributed systems realm, some monitoring tools such as Ganglia [5] provide a resource-based approach for monitoring a cluster state in which they collect resources’ metrics such as CPU, memory, and network usages across the cluster. Such solutions can be easily integrated with existing libraries. However, they do not reflect the state of client applications for two reasons. First, the application state is a library-dependent in which the state composition is governed by the library model. Therefore, it is hard to provide a generic solution for all different kinds of libraries. Second, it is against the architectural design decisions of such tools since they are high-level monitoring tools. Distributed libraries use such monitoring tools to drive management decisions rather than watching the application state.

Some libraries such as Spark [6] take a step further by collecting metrics related to their predefined execution path. Collected metrics in this approach give insights related to the utilization, execution status, or/and bottlenecks issues. Yet, it does not reflect the application state.

In this work, we follow the later approach where a generic status about various library calls is collected. Also, the application state, which is represented by Place and Agent...
extension classes in MASS library, is selectively collected. We believe that MASS is the first multi-agent simulation library to allow selective and efficient monitoring for the system components.

2.1 MASS Java Debugger

There is a previous work done by Niko Simonson and Sean Wessels [7] within the DSLab group. In this paper we deviate from that work for two reasons: (1) it is geared toward visualization rather than monitoring and (2) its design poses a limitation when large sets of data are aggregated from all cluster nodes into a single process, the visualization process in that work. That being said, we follow a similar approach to incorporate support for non-interactive applications.

3 Goals

The overall goals for the capstone project are: (1) a rollback feature that allows the user to checkpoint a program’s state and retrieves it as needed and (2) a monitoring tool for MASS application state for validating the former.

This work, the Monitoring Tool, is to verify the correctness of the Rollback feature, the second part of the capstone project, through observing changes on an application state across MASS cluster. For instance, in the Rollback feature, an application state at a checkpoint must be the same as the application state when it is rolled back to that checkpoint.

As we discussed in section 1, InMASS is built to enhance MASS’s usability and to speed up the development process. This work, the Monitoring Tool, adds to the learnability aspect of MASS library by quickly conveying the underlying interactions of the system components. Therefore, the user can build a solid understanding of the library calls and their side effects. Also, the library developers can verify the expected outcome for their changes much faster than the current practice.

4 Architecture

The Monitoring Tool architecture is a mix and match of three-tier server-client and publish-subscribe architecture styles. We consider the following principal design decisions while developing the system:

1. Library calls are the only actions that introduce side effects on the state.

2. Data are collected once per library call and only the necessary portion of data is streamed to the presentation layer.
3. Data must not be aggregated holistically in a single node. In other words, each cluster node maintains its local portion of the data.

4. Only the most recent data of both metrics and state of the application are presented.

Figure 1 depicts the architecture and its components. Each cluster node in MASS cluster maintains Node, Monitor, and Repository components. The node component is embedded in MASS daemon process. Monitor and Repository are contained within a separate process, which will be referred to as the monitoring process in this paper, which resides on the same machine. Both processes are connected via a pipe. UI is a client (browser) logic that manages both data querying and visualization for the user. The UI and the monitoring process are connected via an HTTP/WebSocket protocol.

Figure 1: Monitoring Tool Architecture
Collected data can be a metrics data or a user-selected application state. For efficiency purposes, the monitoring tool must have an explicit selection mechanism for the application state. Otherways, the system will collect large sets of data which will cause huge delays when collecting the state. The same for the UI when it queries the data, the tool must provide a selective querying mechanism for the data. Otherways, aggregating all data in the UI will cause a memory bottleneck. The architecture is agnostic of these mechanisms yet they are captured by its principal design decisions. We provide a detailed explanation for selection and querying mechanisms in sections 5.3 and 5.4 respectively.

4.1 Components

Here we list the architectural components and their purposes:

Node

The node component manages whether to launch the monitoring process or not. Also, it is responsible for generating the required data and initiating an update request to the monitor component.

Monitor

The monitor component manages interactions between MASS daemon and the monitoring process. Also, it manages client connections and a resume signal. It works as a broker that passes requests/responses to their appropriate destination.

Repository

The repository component is a collection of resources. A client can subscribe to or fetch from any resource within the repository. This component manages resource update requests and it forwards requests to the appropriate resource. Each resource manages update, fetch requests and client subscriptions.

User Interface

UI component is the representation view of the system. It manages a connection to each monitoring process in the system. Next, it initiates query requests to the repository component and it does the visualization of responses.
4.2 Connectors

We have two types of connectors in the tool: Pipes and HTTP/WebSocket. WebSocket protocol in its current implementation requires HTTP during the connection establishment.

Pipe

Pipe connector connects MASS daemon process with the monitoring process. Both processes reside on the same machine. Data flow from MASS daemon process to the monitoring process except for the resume signal. Resume signal is triggered from the front-end then it is passed to the MASS daemon process via another pipe.

HTTP & WebSocket

HTTP server serves static HTML files with a JavaScript code. These files are provided by the master node only. JavaScript contains necessary logic to setup communication to all nodes. Also, it handles user interaction with the UI.

4.3 Data Elements

Data elements of the Monitoring Tool are JSON objects. Both stored data and most communication messages are simple flat JSON objects. In order to avoid complexity with a deeply nested JSON object, a flat representation of the data is preferred. There is an exception for user-defined data. Since we don’t know beforehand what type of data will be annotated by the user, there is no explicit restriction on how many levels are allowed to nest an object neither the type of the object. The tool defines two types of communication messages: querying and monitoring messages.

Querying Messages

Querying messages take place between the UI component and the monitoring process. The querying message consists of four parts as shown in Figure 2: action, handle, query, and msg. The action tells the monitoring process what is the client, UI, intention. The handle defines which resource is targeted by the client. The query part holds additional information which is not mandatory. While the msg represents the message payload associated with a given action.
Monitoring Messages

Monitoring messages take place between the Node component and the monitoring process. A monitoring message is a string that starts with one of the predefined headers and might contain data. Headers are defined via a configuration file that is used by both processes. The header helps the monitoring process deciding which action to take.

A monitoring message might contain data associated with a given header. While some messages might have no header which are considered to be logging or error messages. Also, some might have only a header which are considered to be signal messages: pause and resume signals. Nine headers are considered to define all monitoring messages types.

Six headers are reserved for signals: (1) one is for starting an HTTP with WebSocket server, (2) another is for starting a WebSocket server only, (3) one is for an acknowledgment signal, two for (4) resume and (5) pause signals, (6) the last one is to stop the server. The other three headers are reserved for: (7) hosts, (8) data, and (9) status messages.
5 Implementation

The tool is implemented for MASS Java version. We consider three independent processes when implemented the tool as shown in Figure 3. One is the MASS/MPLProcess which represents the MASS daemon process. Second is the monitoring process which manages data and the querying mechanism. The other is the UI client which is a browser process that executes the JavaScript code.

![Figure 3: Implementation View](image)

We did not touch the MASS implementation except for probing the application state and the library calls. Also, we added two MASS functions see section 6. The probing does not conflict with MASS execution and it is a synchronized logic that occurs before and after most of the exposed library functions.
The probing code is written in Java as well as the Monitor process. While the UI is a combination of HTML, JavaScript, and CSS that is executed by a browser.

The interactive feature code is available on BitBucket [8]. Implementation code for the Monitoring Tool resides under the following path:

```
// Monitoring Tool
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring
```

### 5.1 Handlers and MonitorConnector Classes

The probing code consists of a set of classes that are called Handlers. A Handler class is responsible for transforming MASS component into JSON representation. Handlers collectively produce a one-line string (a JSON object) that represents the node (MASS/MProcess) state. The tool has five handlers: MASSHandler, PlacesHandler, AgentsHandler, PlaceHandler, and AgentHandler.

MASSHandler, PlacesHandler, and AgentsHandler collects metrics for MASS, Places, and Agents, respectively. While PlaceHandler and AgentHandler collect metrics and the state for Place and Agent, respectively.

Also, it has MonitorConnector and MsgUti classes which are responsible for interfacing with the monitoring process. The implementation codes for the Handlers, MonitorConnector, and MsgUti classes reside under the following paths:

```
// Handlers classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/handlers

// MonitorConnector class
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/MonitorConnector.java

// MsgUti class
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/MsgUti.java
```

### 5.2 Monitoring Process

The monitoring process depends on Vert.x library [9]. Vert.x provides a lightweight HTTP server that can be easily embedded in the monitoring process.

The monitor process consists of two types of classes. The first type is classes that manage the data: Repository and Resources classes. Repository forwards requests (read or write) to the appropriate Resources instance. Resource handle requests such as updating the data (write) and fetching the data (read). Also, it manages client subscriptions on the data.

The second type is classes that manage the monitor process: Launcher, HTTPServer,
WebSocketServer, WebSocketHandler, and RequestHandler class. The Launcher is an entry point of the monitoring process which keeps listening for MASS/MProcess messages. The HTTPServer manages an HTTP/WebSocket server. The WebSocketServer manages a WebSocket server only. The WebSocketHandler handles WebSocket events such as opening and closing client connection. The RequestHandler handles client messages that are sent via the WebSocket protocol.

The implementation code for the monitoring process resides on the following paths:

```java
// Data classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/repository

// Management classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/server
```

### 5.3 Selection Mechanism

The tool uses Java Annotation and Reflection for enabling state selection mechanism. We introduce two custom annotations: Inspect and Watch annotations. The following is a code snippet illustrating how the tool utilizes Java Annotation. **Inspect()**, a class annotation, indicates that the annotated class has at least one field to monitor while **Watch()**, a field annotation, indicates which fields to monitor. By doing so, only data that are selected (annotated) by the user will be collected.

```java
@Inspect() // Indicate class intended to inspect
public class Agent {
    ...

    @Watch() // Indicate property intended to watch
    private String data;

    ...
}
```

PlaceHandler and AgentHandler classes will look for these annotations using Java Reflection when collecting the state.

Implementation codes for annotations and reflections resides under the following path:

```java
// Annotations
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/annotations
```
5.4 Querying Mechanism

After the Handlers produce a local state representation, it will be streamed to the monitoring process. The application state is represented in a large JSON object which might be scattered across multiple monitoring processes. Collecting such an object in the UI component (the browser process) is an inefficient solution and it might be invisible for applications that have a large state.

We provide a querying mechanism to avoid the memory bottleneck at the UI component. It tackles this issue by querying the necessary data for the UI component rather than collecting the whole state. As illustrated in Figure 2, a querying message has a field called query. This field is used to specify a small set or even a single field from the state.

For instance, the query field can be used to query data for a specify Agent. All needed is to provide the Agents' handle and the id for the specified Agent. To make this example concrete, let us assume the Agents handle is 1234 and the Agent id is 5678. The query field will be set to 1234.5678. The querying message can be constructed as follows:

```
// This will fetch the agent data only once
{ action: "FETCH", handle: "AGENT", query: "1234.5678" }

// This will fetch the agent data once and every time it is changed
{ action: "SUBSCRIBE", handle: "AGENT", query: "1234.5678" }
```

5.5 UI Code

The UI is written using Angular framework [10] and TypeScript [11]. The output of this framework is HTML, JavaScript, and CSS files. The monitoring process must have these files in order to serve them via an HTTP protocol thanks to Vert.x library. We designated `webroot` folder (inside the resources folder) to hold these files.
6 Non-Interactive Applications

Monitoring Tool is built for InMASS, interactive application, in mind. However, we added a few changes to port its benefits for non-interactive applications. For that purpose, the tool operates in two modes: AUTO or ONPAUSE. AUTO mode is intended for the interactive shell, InMASS. When running in AUTO mode, data will be automatically collected per each system call.

ONPAUSE mode is intended for non-interactive applications. When running in ONPAUSE mode, the cluster must explicitly be instructed when to collect the state. For that purpose, we add two functions to MASS arsenal. (1) MASS.collect(), it instructs each node to forcibly collect the state. (2) MASS.pause(), it collects the state much the same as the MASS.collect() does. However, it will hold control right after collecting the state. Execution control will be released upon a resume signal that is triggered by the front-end. Figures 4 and 5 depict these functions and their behaviours.

![Figure 4: Collect Function](image-url)
Figure 5: Pause Function

It is worthwhile to mention that AUTO mode can be used in non-interactive and the ONPAUSE can be used in interactive applications, too. However, in a non-interactive application, one must hold the execution control in order to successfully observe the state. Otherwise, the application might run faster than the user opening a browser window.
7 User Manual

Setup Configuration

The Monitoring tool setup is fairly simple, all needed is a single XML tag, inserted into the MASS configuration file. As shown in the Listing 1, line 8. This tag informs the underlying implementation to set up and run the monitoring process.

Two additional tags can be used as needed. One is for specifying the monitoring mode line 11 (The monitoring modes are explained in section 6). The other is for specifying the HTTP/WebSocket server port line 14. This can be useful when the default port 8080 is unavailable.

```
<?xml version="1.0" encoding="UTF-8"?>
<nodes>
  <!-- Global Configuration -->
  <!-- Monitoring Tool configuration -->
  <!-- Monitor Flag Mandatory-->
  <monitor>true</monitor>
  <!-- Mode, Optional: Default is AUTO mode -->
  <monitor-mode>ONPAUSE</monitor-mode>
  <!-- WebSocket Port, Optional: Default is 8080 -->
  <monitor-port>45295</monitor-port>
  <!-- Nodes Configurations -->
</nodes>
```

Listing 1: nodes.xml, MASS configuration file

State Selection

The user can use the custom Java annotations we introduce in section 5.3 to select a state of interest. This step is not mandatory yet if there is no field selected, only the metrics data will be collected by the tool.

Accessing the UI

After starting the application, the user can interact with UI through a browser window. The master node URL is used for opening the UI webpage. One must use the configured port if it is set up in the nodes.xml file. Otherwise, the default port 8080 must be used for opening the webpage.
7.1 Sample

In this section, we demonstrate how the user can select, query the application state, and interact with the UI component. We consider the QuickStart application [12] for demonstrating the Monitoring Tool.

Models

Place and Agent models are represented as shown in Listing 2 lines 1-55 and 57-155 respectively. Inspect annotations at line 1 and 57 indicate that the user want to watch the state of each model. At the place model, lines 7, 10, 13, and 16 demonstrate the use of Watch annotation as well as in the Agent model line 67.

Listing 2: QuickStart Application: Agent and Place Models

```java
@Inspect()
public class Matrix extends Place {
    public static final int GET_HOSTNAME = 0;

    @Watch()
    public int touches = 0;

    @Watch(key = "nullArray2")
    public Object[] nullArray;

    @Watch(key = "generic")
    public Object genericObject = new Object[] {"Hi", 123, 7L};

    @Watch(key = "private")
    private final Object secureValue = 1234567890L;

    public Matrix(Object obj) {
        touches++;
        Vector<int[]> placeNeighbors = new Vector<int[]>() {
            placeNeighbors.add( new int[] {0, -1, 0 } );
            placeNeighbors.add( new int[] {0, 1, 0 } );
            placeNeighbors.add( new int[] {0, 0, 1 } );
            placeNeighbors.add( new int[] {0, 0, -1 } );
            placeNeighbors.add( new int[] {1, 0, 0 } );
            placeNeighbors.add( new int[] {-1, 0, 0 } );
            setNeighbors( placeNeighbors );
        }

        public Object callMethod(int method, Object o) {
            touches++;
        }
    }
```
switch (method) {
    case GET_HOSTNAME:
        return findHostName(o);
    default:
        return new String("Unknown Method Number: " + method);
}

public Object findHostName(Object o){
    String result;
    try{
        result = (String) "Place located at: "
            + InetAddress.getLocalHost().getCanonicalHostName()
            + ": " + Integer.toString(getIndex()[0])
            + ": " + Integer.toString(getIndex()[1])
            + ": " + Integer.toString(getIndex()[2]);
    }catch (Exception e) {
        result = "Error : " + e.getLocalizedMessage() + e.getStackTrace();
    }
    return result;
}

@Inspect()
public class Nomad extends Agent {
    public static final int GET_HOSTNAME = 0;
    public static final int MIGRATE = 1;
    public static final int MIGRATE_REVERSE = 2;
    public static final int MIGRATE_RANDOM = 3;
    public static final int KILL_RANDOM = 4;
    public static final int KILL = 5;

    @Watch()
    public int touches = 0;

    Random generator;

    public Nomad(Object obj) {
        this.touches++;
        this.generator = new Random();
    }

    public Object callMethod(int method, Object o) {
        this.touches++;
        switch (method) {

case GET_HOSTNAME:
return findHostName(o);

case MIGRATE:
return move(o);

case MIGRATE_REVERSE:
return moveBack(o);

case MIGRATE_RANDOM:
return randomMove(o);

case KILL_RANDOM:
return this.randomKill(o);

case KILL:
return this.kill(o);

default:
return new String("Unknown Method Number: " + method);
}

public Object findHostName(Object o){
String result;
try{
result = (String) "Agent located at: "
+ InetAddress.getLocalHost().getCanonicalHostName()
+ "\": " + Integer.toString(getIndex()[0])
+ "\": " + Integer.toString(getIndex()[1])
+ "\": " + Integer.toString(getIndex()[2]);
} catch(Exception e) {
result = "Error : " + e.getLocalizedMessage() + e.getStackTrace();
}
return result;
}

public Object move(Object o) {
int xModifier = this.getPlace().getIndex()[0];
int yModifier = this.getPlace().getIndex()[1];
int zModifier = this.getPlace().getIndex()[2];

xModifier++;
migrate(xModifier, yModifier, zModifier);
return o;
}
public Object moveBack(Object o) {
    int xModifier = this.getPlace().getIndex()[0];
    int yModifier = this.getPlace().getIndex()[1];
    int zModifier = this.getPlace().getIndex()[2];
    xModifier--;
    migrate(xModifier, yModifier, zModifier);
    return o;
}

public Object randomMove(Object o) {
    int[] randomDest = IntStream.of(this.getPlace().getSize())
        .map(x -> this.generator.nextInt(x)).toArray();
    migrate(randomDest);
    return o;
}

public Object randomKill(Object o) {
    if(generator.nextFloat() < 0.5)
        this.kill();
    return o;
}

public Object kill(Object o) {
    this.kill();
    return o;
}

int x = 50;
int y = 50;
int z = 50;

System.err.println( "Quickstart creating Places..." );
Places places = new Places( 1, Matrix.class.getName(),
    ( Object ) new Integer( 0 ), x, y, z );
System.err.println( "Places created" );

Object[] placeCallAllObjs = new Object[ x * y * z];
System.err.println( "Quickstart sending callAll to Places..." );
Object[] calledPlacesResults = ( Object[] )
places.callAll( Matrix.GET_HOSTNAME, placeCallAllObjs );
System.err.println( "Places callAll operation complete" );

System.err.println( "Quickstart creating Agents..." );
Agents agents = new Agents( 1, Nomad.class.getName(), null, places, x * y );
System.err.println( "Agents created" );

Object[] agentsCallAllObjs = new Object[ x * y ];
System.err.println( "Quickstart sending callAll to Agents..." );
Object[] calledAgentsResults = ( Object[] )
agents.callAll( Nomad.GET_HOSTNAME, agentsCallAllObjs );
System.err.println( "Agents callAll operation complete" );

// can be used instead of the loop function
// calledAgentsResults = ( Object[] )
// agents.doAll( Nomad.MIGRATE, agentsCallAllObjs, z );

for (int i = 0; i < z; i ++) {

// tell Agents to move
System.err.println( "Quickstart instructs all Agents to migrate..." );
agents.callAll(Nomad.MIGRATE);
System.err.println( "Agent migration complete" );

// sync all Agent status
System.err.println( "Quickstart sending manageAll to Agents..." );
agents.manageAll();
System.err.println( "Agents manageAll operation complete" );
}

MASS.pause();

// return all agents back to where they start
// calledAgentsResults = ( Object[] )
// agents.doAll( Nomad.MIGRATE_REVERSE, agentsCallAllObjs, z );

System.err.println(
"Quickstart sending callAll to Agents to get final landing spot..." );
calledAgentsResults = ( Object[] )
agents.callAll(Nomad.GET_HOSTNAME, agentsCallAllObjs );
System.err.println( "Agents callAll operation complete" );
The UI webpage consists of multiple views: Cluster Status, Timers, Query Dialogs, Nodes, Logging, Queries views. Cluster Status view as shown in Figure 6 contains the status of last call and two lists of models. One list for places’ models and the other is for agents’ models. When the execution reaches Listing 2 line 200, the Cluster Status view will have a button to return control back to the application. When the user clicks on the timer icon (at the top-right), a pop-up view will be presented that shows calls stack each with a time elapsed as shown in Figure 7. When the user click on the search icon (at the right side for each model), a pop-up view, , will be presented as shown in Figure 8.

Figure 6: Cluster Status View: Places and Agents Representation
The Query Dialog has two variations, Figure 8. One is for querying from place models and the second is for agent models. Place query dialog allows dimensional or linear index for querying a place while the Agent query dialog except linear id only.
Nodes view illustrates places mapping per each node as well as the agent population as shown in Figure 9. Also, it has a status view that shows the last function call, similar to the Cluster Status view. Each node is labeled whether it is a master or worker node.

Figure 9: Nodes View
Figure 10 shows the Logging view. All error pipes from MASS daemon processes are collected in this window.

![Logging Window](image)

**Figure 10: Logging View**

When a place or an agent model is fetched from the monitoring process, the UI will add a new view (we call it Query view) that shows query data. Figure 11 illustrates a place query view. It has three tabs: inspection, neighbors, and occupants. The annotated state can be inspected at the inspection tab as shown in Figure 12. The neighbor places can be inspected at the neighbors’ tab as shown in Figure 13.

![Place View](image)

**Figure 11: Place View**
All selected fields in the model will be collected and shown under the inspection tab, illustrated in Figure 12.

Figure 12: Place View: Inspection Tab Expanded
In neighbors tab Figure 13, the user can navigate to a neighbor by clicking on its index. The UI will add new view for the neighbor if it is not queried yet.

![Figure 13: Place View: Neighbors Tab Expanded](image.png)
8 Next Steps

Next quarter Winter 2020, is the focus of the Rollback feature and the writing. The Monitoring Tool will help us to validate the Rollback implementation. The following is a weekly plan for the next work, Winter 2020:

Week 01, 01/07 - 01/11: Rollback Implementation
- Reserve an appointment for the defense
- Start rollback APIs implementation: checkpoint & rollback

Week 02, 01/14 - 01/18: Rollback Implementation
- Complete rollback APIs implementation: checkpoint & rollback
- Update status logic in Monitoring Tool to capture side effect of Rollback functions
- Implement history tracking logic (call stack recording)

Week 03, 01/21 - 01/25: Rollback Testing
- Revising (Fixing issues with previous implementations)
- Testing with Monitoring Tool

Week 04, 01/28 - 02/01: Rollback Testing
- Collecting result and measurements
- Rollback Feature Demo (Deliverable)
- Plan Revision

Week 05, 02/04 - 02/08: Revising and Writing
- Start writing Final Capstone Report
- Revising Rollback + Monitoring Tool

Week 06, 02/11 - 02/15: Writing
- Continue writing Final Capstone Report
Week 07, 02/18 - 02/22: Writing & Presentation

- Continue writing Final Capstone Report (First Draft Deliverable)
- Start preparing for the defense presentation

Week 08, 02/25 - 02/29: Presentation

- Continue preparing for the defense presentation

Week 09, 03/03 - 03/07: Final Revising & Presentation

- Revising Final Capstone Report
- Continue preparing for the defense presentation

Week 10, 03/10 - 03/14: Deliverables

- Final Capstone Report
- Rollback Implementation

Week 11, 03/17 - 03/21: Presentation (actual date TBA on the first week)

- Defense Presentation

9 Conclusion

The Monitoring Tool is still in its infancy. The current implementation is a proof of concept that brings the tool to the light. Considering that, only one developer is working on the tool, within a time constraint. Further development and optimization are needed.

That being said, using the tool with the current implementation, we were able to point out and fix two critical bugs in the library. We ran the QuickStart application [12] interactively and as a standalone application for demonstrating the state selection, querying mechanism, UI views and UI interaction. In this report, we provide a detailed architecture that can help other developers to develop this work further.
Limitations

Due to the time constraint, we were not able to write enough unit tests for the tool. Also, we did not have extensive tests that provide confidence for maintainers to ship the tool for the next release. Therefore, writing enough unit tests and integration tests are the next steps required before shipping the tool to the releasing phase.

Another limitation is due to an efficiency decision we decided to go with. Taking advantage of the fact that any JSON object always starts with a curly bracket, we consider the curly bracket character as the data header (see section 4.3, Monitoring Messages). Therefore, no string manipulation is required to extract the JSON object from the message. However, the trade-off of doing so is that any logging message cannot be started with a curly bracket.

Future Work

The querying mechanism can be used to develop a plugin that allows users to visualize the application state using JavaScript. Since the querying mechanism is accessible from the browser, users can query the annotated fields and implement their visualization logic in JavaScript. One might provide a JavaScript package that allows the user to interface with the querying mechanism seamlessly.

Monitoring Tool components are implemented in Java and JavaScript. The current implementation for the monitoring process, Monitor and Repository components, can be used with MASS C++. However, the Node component of the architecture must be embedded in MASS C++ implementation. One might implement the Node component and define a selection mechanism in the C++ language.
References


