



SAGE User Guide Documentation

Release v2.0.28

U.S. Naval Research Laboratory

Jan 23, 2020

GETTING STARTED

1	Features	3
2	SAGE Server	7
3	SAGE Node	9
3.1	SAGE Agent	9
4	How It Works	11
5	SAGE App	15
5.1	Main window	15
5.2	Server settings	17
5.3	Network Tree View	18
5.4	Network Graph View	20
5.5	Starting the Server	21
5.6	Active Server	22
5.7	Creating a Node on the Server machine	29
5.8	Creating an Agent	29
5.9	Behavior Manager	30
5.10	Adding a Behavior to an Agent or Node	31
5.11	Sending a message	31
5.12	Sending files to a Node	33
5.13	SAGE Robot Interface	36
5.14	Log viewer	36
5.15	Import and export network	40
6	Download	43
7	Installation	45
7.1	Requirements	45
7.2	Windows	45
7.3	Environment variables	48
7.4	Installation on Linux	50
8	Starting a Server Instance	51
9	Connecting a Node to the Server	53
10	Building an Agent Network	55
11	Writing a SAGE Behavior	57

11.1	Inheritance from the base class	58
11.2	Result object	58
11.3	Behavior constructor	59
11.4	Behavior setUp	60
11.5	Behavior action	61
11.6	Behavior message	62
11.7	Behavior tearDown	63
11.8	C++ header file	64
11.9	Example Behavior file	64
11.10	Packaged Behaviors	69
12	Base Class - SAGE Behavior Methods	71
12.1	addBehavior	71
12.2	createAgent	72
12.3	createState	72
12.4	getState	72
12.5	getStateNames	73
12.6	removeAgent	73
12.7	removeBehavior	74
12.8	removeState	74
12.9	setAgentActive	75
12.10	sendFile	75
12.11	sendMessage	76
12.12	setState	76
13	Retrieving Information About Your Agent Network	77
14	Sending Messages Between Agents	83
15	Responding to a Message Sent to an Agent	85
16	Managing Behavior Files	89
17	Adding a Behavior to an Agent	91
18	Using Supplemental Files in Your Automation	93
19	Activating Agents	95
20	Creating A Test Case Using Robot Framework - Controller Application	97
21	SAGE Robot Framework Keywords	101
22	Running Your Automation Test	103
23	Capturing the Test Results	105
24	Creating Your Own Controller Application	107
24.1	addBehavior	107
24.2	connect	108
24.3	clearExecutionResults	108
24.4	createAgent	109
24.5	disconnect	109
24.6	getAgentNames	109
24.7	getExecutionResults	110
24.8	getNodeNames	111

24.9	sendFile	111
24.10	sendMessage	111
24.11	setAgentActive	112
24.12	removeAgent	113
24.13	removeBehavior	113
24.14	waitForResult	113
25	Help and FAQ	115
26	Example	119
26.1	Test case	119
26.2	Running test	123
26.3	Generated results	124
27	Robot Framework Background	125
28	Changelog	129
29	SAGE makes automating complex systems simple.	135
	Index	137

Introduction

SAGE Framework is a multi-agent system designed as an enabling technology for the research, development, and deployment of agent systems that address wide range of problems including: automated testing of distributed systems, emergent agent configurations, novel agent reasoning algorithms, and distributed simulation. SAGE maintains a small footprint; it's implemented in C++ and uses a highly efficient, and widely used IPC/RPC framework called Remote Call Framework (www.deltavsoft.com).

SAGE is free software and has no dependencies that require paid licenses or impose restrictions on distribution. SAGE is an open system that can interact with external systems, and can run on a variety of platforms from servers to IOT devices.

SAGE was designed to be pragmatic so that it can easily integrate into new or existing software systems. SAGE provides interface libraries that allow external applications to build, control, and interact with SAGE agent networks. SAGE Agents can also interact with their surrounding software environment wherever that may be. SAGE includes an interactive agent development environment called SAGE App where you can interactively build, experiment with, visualize, and store agent networks.

SAGE is simple to use. It consists of four parts:

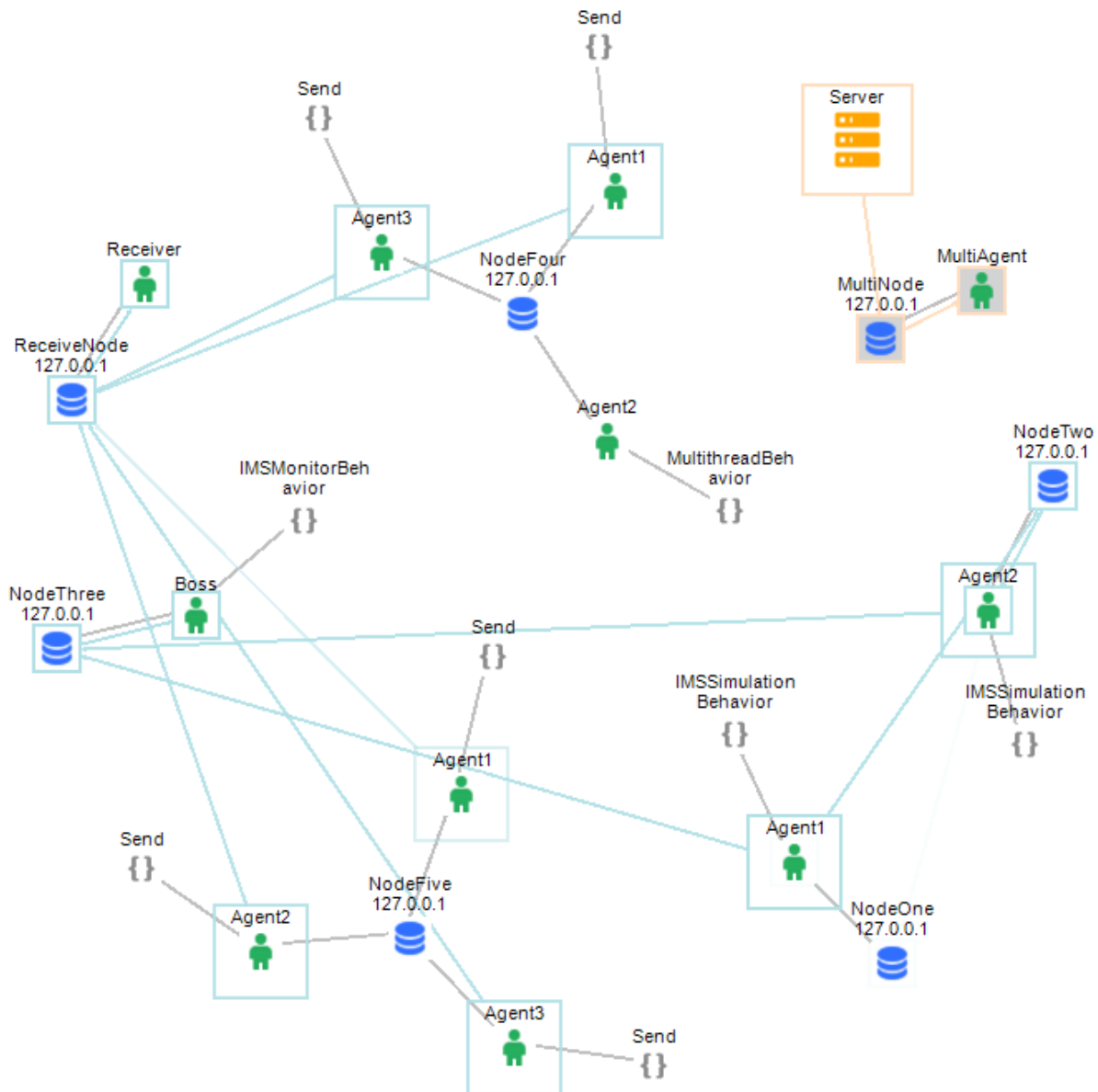
SAGE Server is a console application available for both 32 and 64 bit version of Windows and Linux. SAGE Server acts as the hub of a SAGE agent network. It interacts with other SAGE components to provide them with services, manage communications, maintain system state, log system events, and provide introspection services.

SAGE Node is also a console application available for both 32 and 64 bit versions of Windows and Linux. SAGE Node acts as a container for SAGE Agents managing their execution and maintaining connectivity to the SAGE Server. SAGE Node is designed to be portable across a wide variety of system architectures. Multiple instances of SAGE Nodes can be distributed across a network of computing devices of various architectures. SAGE Framework manages the data translation between different architectures transparently.

SAGE Agents are autonomous agents consisting of a repository of Behaviors and a state. SAGE Agents exist within a SAGE Node and can communicate with other agents using message passing. Agents can have any number of Behaviors each consisting of a mix of C++, Java, or Python modules. Behaviors are stored in a central repository on the SAGE Server and are sent to agents at runtime on demand.

SAGE Behaviors are objects that are derived from base classes provided with the SAGE Framework. A SAGE behavior endows an Agent with the capability to act either proactively or reactively. Features

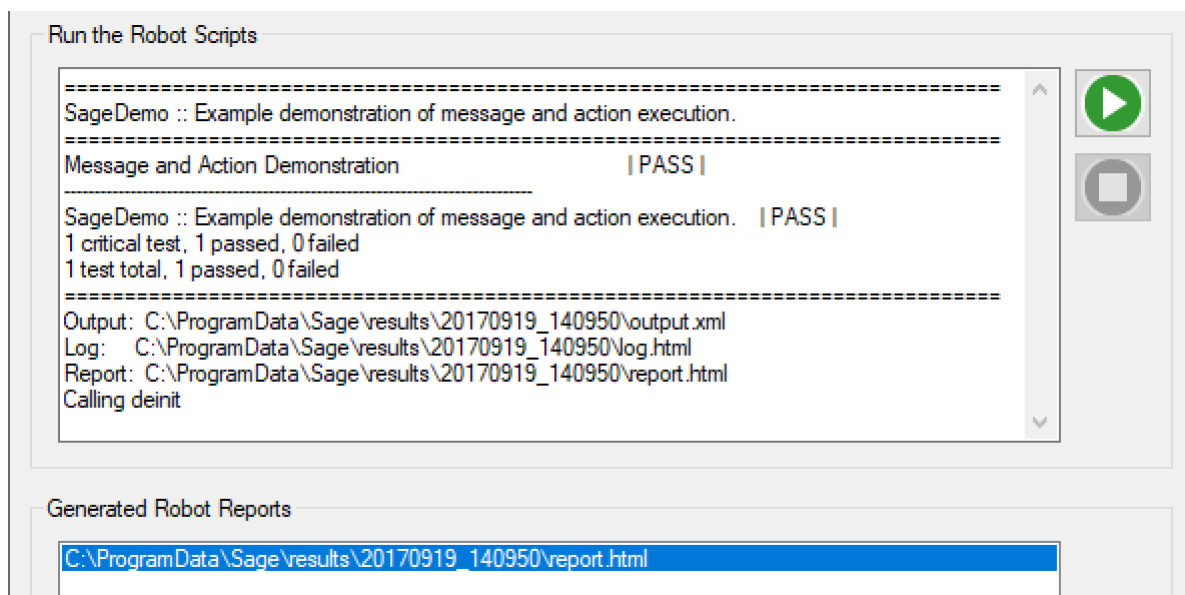
Fast. Simple. Seamless.




```

52
53 public boolean message(Message message, Result result)
54 {
55     // Create a new agent
56     createAgent("NodeA", "Agent1");
57
58     // Add Behavior to agent
59     addBehavior("NodeA", "Agent1", "ExampleBehavior", "ExampleBehavior.class", topi
60
61     // Activate agent
62     setAgentActive("NodeA", "Agent1", true;
63
64     result.m_executionResult = ExecutionResultType.CompletionSuccess;
65     return true;
66 }
67
68 public boolean tearDown(Result result)
69 {
70     // Deactivate agent
71     setAgentActive("NodeA", "Agent1", false);
72
73     // Log event

```

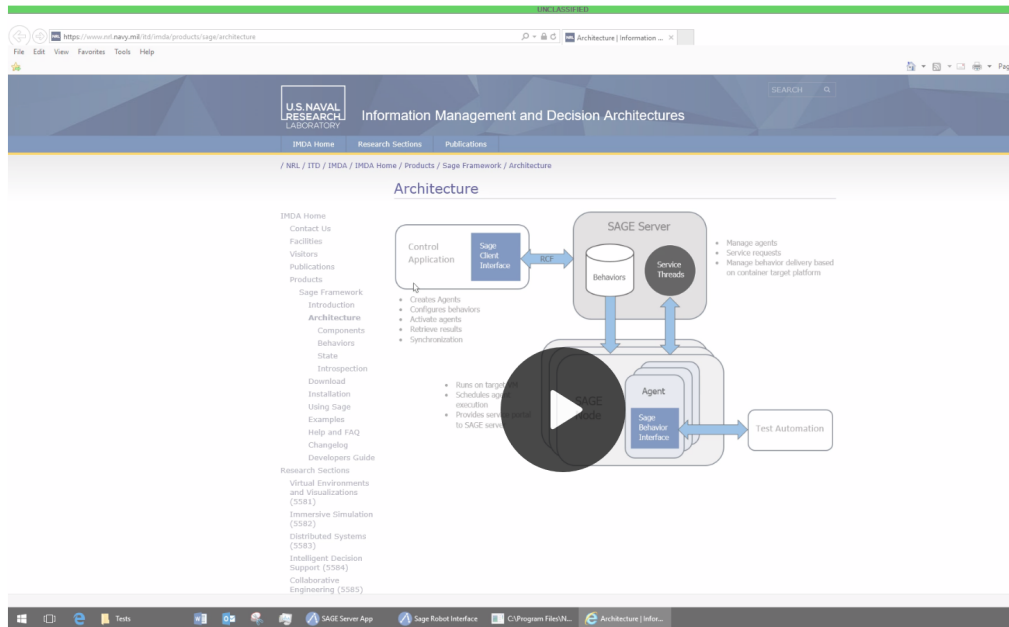


KEYWORD SageRobotKeywords.Report Result \$(ELEMENT)

Start / End / Elapsed: 20170919 15:58:27.226 / 20170919 15:58:27.226 / 00:00:00.000

15:58:27.226 **INFO** Agent_Bond in NRLNode executing behavior RecorderBehavior reported the following results:

15:58:27.226 **INFO** Agent_Bond started recording.



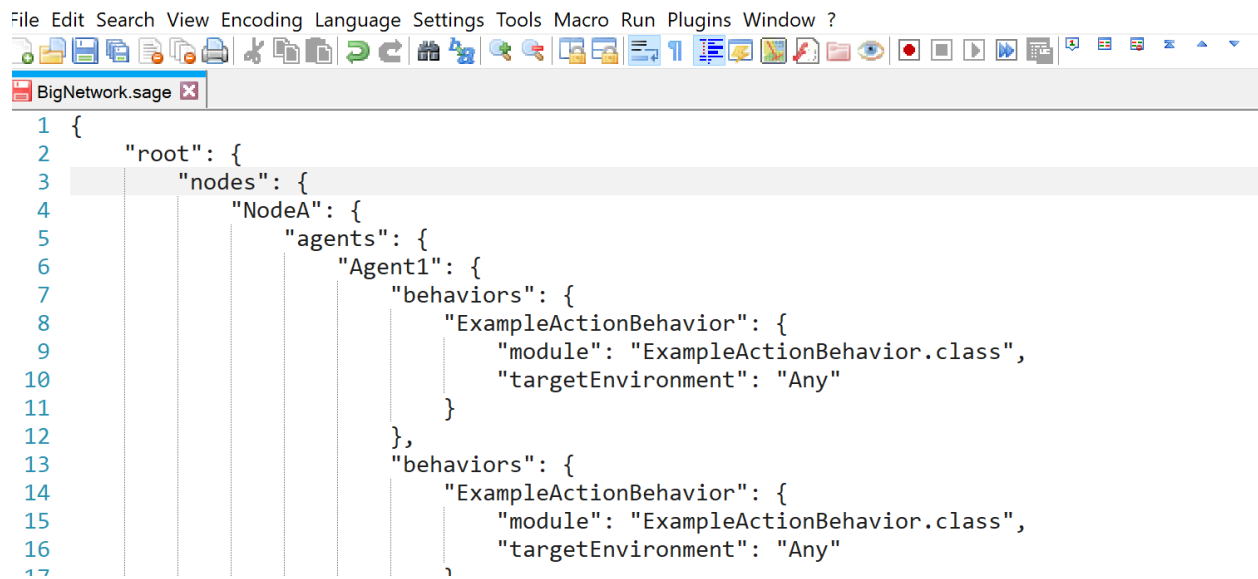
SAGE Framework permits easy language agnostic extensibility and enhancement to support the introduction of new systems and ad hoc test tools. Seamlessly combine, otherwise disconnected, third-party tools and libraries to easily extend functionality.

Persistent network capability

Store network state to facilitate the automation of large scale systems. Speed up automation processes by loading SAGE node, agent, and behavior configurations to quickly restore previously saved network states.

“Robot Framework” by Robot Framework Foundation is licensed under the [Apache License 2.0](#)

SAGE Server



```
1 {
2   "root": {
3     "nodes": {
4       "NodeA": {
5         "agents": {
6           "Agent1": {
7             "behaviors": {
8               "ExampleActionBehavior": {
9                 "module": "ExampleActionBehavior.class",
10                "targetEnvironment": "Any"
11              }
12            },
13            "behaviors": {
14              "ExampleActionBehavior": {
15                "module": "ExampleActionBehavior.class",
16                "targetEnvironment": "Any"
17              }
18            }
19          }
20        }
21      }
22    }
23  }
```

SAGE SERVER



The SAGE Server establishes and maintains a connection between SAGE Nodes and any integrated external Controller application.

All communication shared across the agent network propagates through the SAGE Server enabling it to provide the following services:

- Create and destroy Agents
- Manage Agent behaviors
- Activate and deactivate Agents
- Send messages to Agents
- Retrieve Behavior execution results
- Synchronize their execution with SAGE behaviors.
- Hosts the Behavior repository
- Manage Behavior delivery

SAGE Node

SAGE NODE

SAGE Nodes are containers for your SAGE Agents. A single Node can manage hundreds of Agents.

Nodes connect directly to the SAGE Server and must be uniquely identifiable by name. Nodes are usually deployed on systems based on the locality of the automation. For example, in a SOA test automation scenario, SAGE Nodes would be deployed on the system under test (SUT).

A SAGE Node fulfills a number of functions including:

- Handling incoming requests from other Agents for message delivery, including file download and upload.
- Agent creation and deletion
- Agent Behavior management
- Agent activation and deactivation, .
- Scheduling the execution of Agent Behaviors.
- Communicating with the SAGE Server to deliver requests to remote Agents.
- Reporting the results of Behavior execution back to the Server.
- Implementing synchronization between Agents.

3.1 SAGE Agent

SAGE Agents are autonomous, active objects that execute your automation code (Behaviors). Agents exist within a SAGE Node instances.

They contain a state space that consists of name-value tuples, providing Agents with the information they need to carry out their behaviors.

Agent names only have to be unique within the Node that contains them.

Agents in SAGE are constructed dynamically, at runtime, through requests from the SAGE Server or from other Agents. This capability enables Agents to create and configure other Agents at runtime.

SAGE Agent services include:

- Create and destroy Agents

- Configure Agents
- Activate and Deactivate Agents
- Send messages to Agents
- Execute Behaviors

Addressing an Agent in a SAGE runtime requires both the Node name as well as the Agent name.

```
Create Agent  TestNode  TestAgent
Add Behavior  TestNode  TestAgent  ExampleBehavior  ExampleBehavior.jar
Activate Agent  TestNode  TestAgent
```

How It Works

HOW IT WORKS

SAGE execution is summarized in these following steps:

1. *Starting a SAGE Server instance.*
2. *Connecting SAGE Node instance to SAGE Server.*
3. *Building an agent network.*
4. *Distributing Behaviors to target Nodes.*
5. *Activating agents.*
6. *Creating a test case that runs a behavior.*
7. *Logging the events, collecting the results.*

An instance of a SAGE runtime consists of:

- *ONE* instance of a SAGE Server
- *ONE or MORE* SAGE Node instances running on either the Server computer or on remote computers with network connectivity to the Server Computer.
- In addition, a SAGE Node instance may have *MULTIPLE* SAGE Agents which may each have *MULTIPLE* Behaviors.

1. Starting a SAGE Server instance

To begin automating your environment, you must first stand-up a single SAGE Server instance.

The SAGE Server is the central component for your automation. It serves as a single-point of access for managing your Agent network, by handling communication and code (Behavior) distribution.

2. Connecting SAGE Nodes

The next step would be to stand-up your Node instances on all computers that are expected to execute any automation. This includes any local and remote machines.

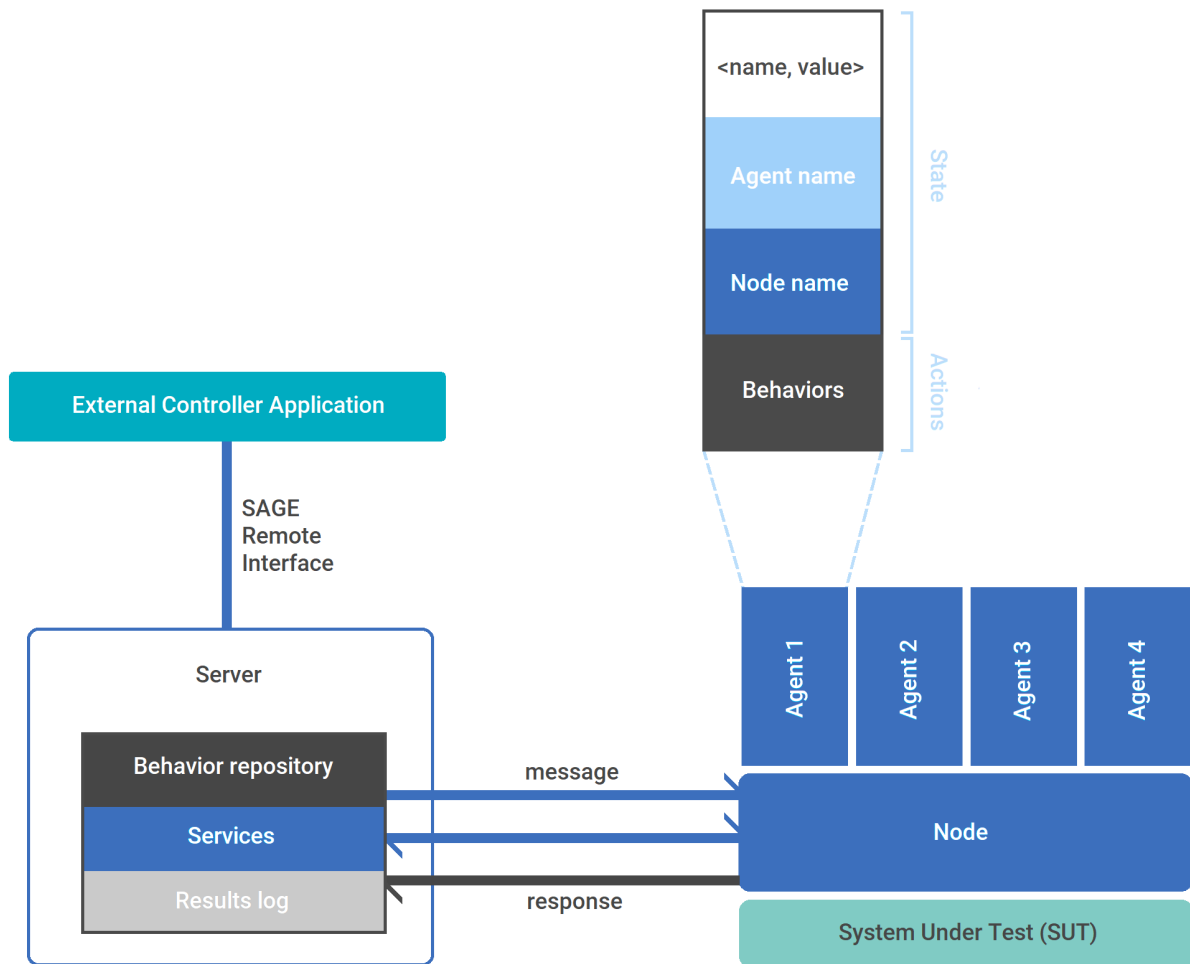
Nodes act as a containers for your Agents. While your Agents execute your automation code, the Nodes can be viewed as conductors for Agents. This means that the Server has fewer components to communicate with as it bypasses the need to have to communicate with an Agent directly.

This also means that as a user, you will not have to manually configure multiple Agents across multiple machines. Instead, you can stand up a single Node and later automate the creation and deletion of Agents tied to that Node.

This architecture was designed with scalability in mind.

3. Adding agent entities to Nodes

You can begin populating your Nodes with Agents as soon as your Nodes are initialized. This can be done via manual configuration or it can be automated by either coding it into Behaviors or through an external controller mechanism.



This gives you the possibility of dynamically building an agent network.

4. Managing Behaviors

Behaviors are only sent to Agents on an as-needed basis. This minimizes overhead and gives Agents the flexibility of learning and unlearning skills on demand.

All Behaviors are stored in a single repository on the Server machine. When requested, the Behavior module will be sent from the Server to a remote Node. From there, the selected Agent will be granted access to the file.

This means that you will not have to worry about manually copying code across multiple machines during configuration.

5. Activating agents

You are ready to activate your Agents once you have established connection between your Server and Nodes and have populated your Server with the necessary Behavior modules.

Agents must first be activated in order to receive any updates from the network. Thus, an agent that is not activated will not receive incoming messages.

Upon activation, your Agents may respond reactively or proactively depending on what Behaviors it contains and how those Behaviors are constructed.

Reactive execution occurs when an Agent performs some action upon receiving any incoming message.

Proactive execution can occur in multiple ways based on how an Agent's Behaviors are configured.

6. Running tests

You can build out a step-by-step test case using the Robot Frameworks integration into SAGE Framework as a controller mechanism.

This integration allows you to send messages from the Controller. This message will propagate through the Server to the Node to instruct an Agent.

Upon receiving messages, your Agents will begin executing their coded automation abilities.

7. Capturing test results

During automation, any pertinent information can be captured by Behaviors and reported as an execution result.

All information, including execution results as well as SAGE log events, are propagated back to the Server to be stored in a single location. This means you will not have to manually check multiple computers to verify your results.

Instead, users only have to check the logs residing on the Server machine.

SAGE App

SAGE APP

The SAGE App provides a graphical user interface for viewing and controlling the SAGE runtime environment. supports the following functions:

- Manage behavior modules
- Start and stop the SAGE server
- View connected nodes
- Create nodes on the local server machine
- Control node access to the server machine
- Send files and messages to nodes
- Create, activate, deactivate and remove agents
- Send messages to agents
- Add and remove agent behaviors
- Select, sequence and run SAGE Robot files
- View a generated Robot reports
- View SAGE server log output, including behavior execution results
- Import and Export SAGE networks

SAGE App is Windows compatible and distributed with the Windows x64 - Server installer.

To run SAGE App, double-click on the shortcut icon, or right-click on the icon and select “Open” from the context menu.

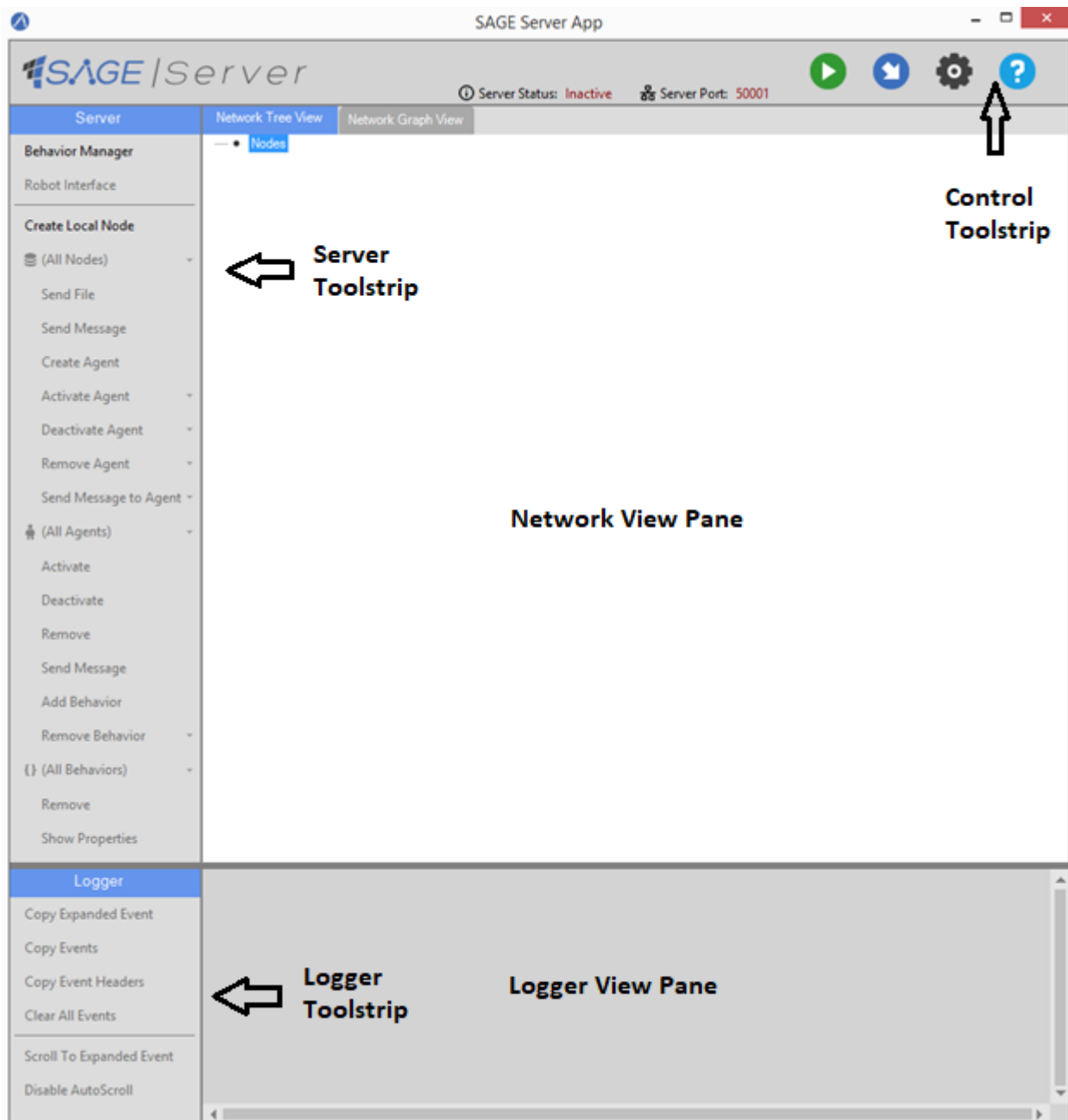
5.1 Main window

Following is an annotated view of the SageServerApp window when the application opens.

Control Toolstrip

Provides buttons for:

- Starting and stopping the SAGE Server.
- Importing and exporting a SAGE network.
- Configuring Server settings.
- Displaying information about the SAGE Server.



Server Toolstrip

Provides buttons for viewing and controlling the SAGE network:

- Create Nodes on the local Server machine.
- Create and remove Agents
- Activate and deactivate Agents.
- Add and remove Behaviors
- View Behavior properties.
- Send files to Nodes.
- Send messages to Nodes and Agents.
- Launch the SAGE Behavior Manager
- Launch Robot Framework Interface.

Network View Pane

Allows you to view and control the SAGE network using the following views:

- Network Tree View
- Network Graph View

Views can be toggled by clicking on the respective tab.

Logger Toolstrip

Allows you to interact with server-generated log events.

Provides buttons for :

- Copying event logs to the Windows clipboard
- Clearing logs displayed in the Logger View Pane.
- Controlling the scrolling of log events when the logger is active.

Logger View Pane

Allows you to view server-generated log events, including behavior execution results as they occur in real-time.

5.2 Server settings

Clicking on the `Settings` button in the “Control Toolstrip” brings up the Server Settings popup dialog.

SAGE Server settings include the following:

- Control Node access to the server machine.
- Set Server IP port number
- Set the maximum capacity of the SAGE event logger.
- Set the scheduler timing resolution for local nodes.
- Adjusting window overlays.
- Changing color theme of application.

Settings are automatically saved when the application terminates.

Server Port

Set the port number that the SAGE Server is listening to.

Node Access Control

Node access control is enforced when `Enable Node Access Control` is checked.

This allows you to whitelist IP addresses in dot-decimal notation. An asterisk may be used as a wildcard character for any of the octet-grouped decimal numbers. Only IP addresses listed are allowed to connect to the server.

Event Logger Maximum Capacity

Setting the maximum capacity of the event logger prevents Server-generated log events from overrunning memory when the Server runs for very long periods of time (day or even weeks). When the maximum capacity is reached, older events are removed to make room for new events.

5.3 Network Tree View

The Network Tree View provides a tree view of the SAGE network.

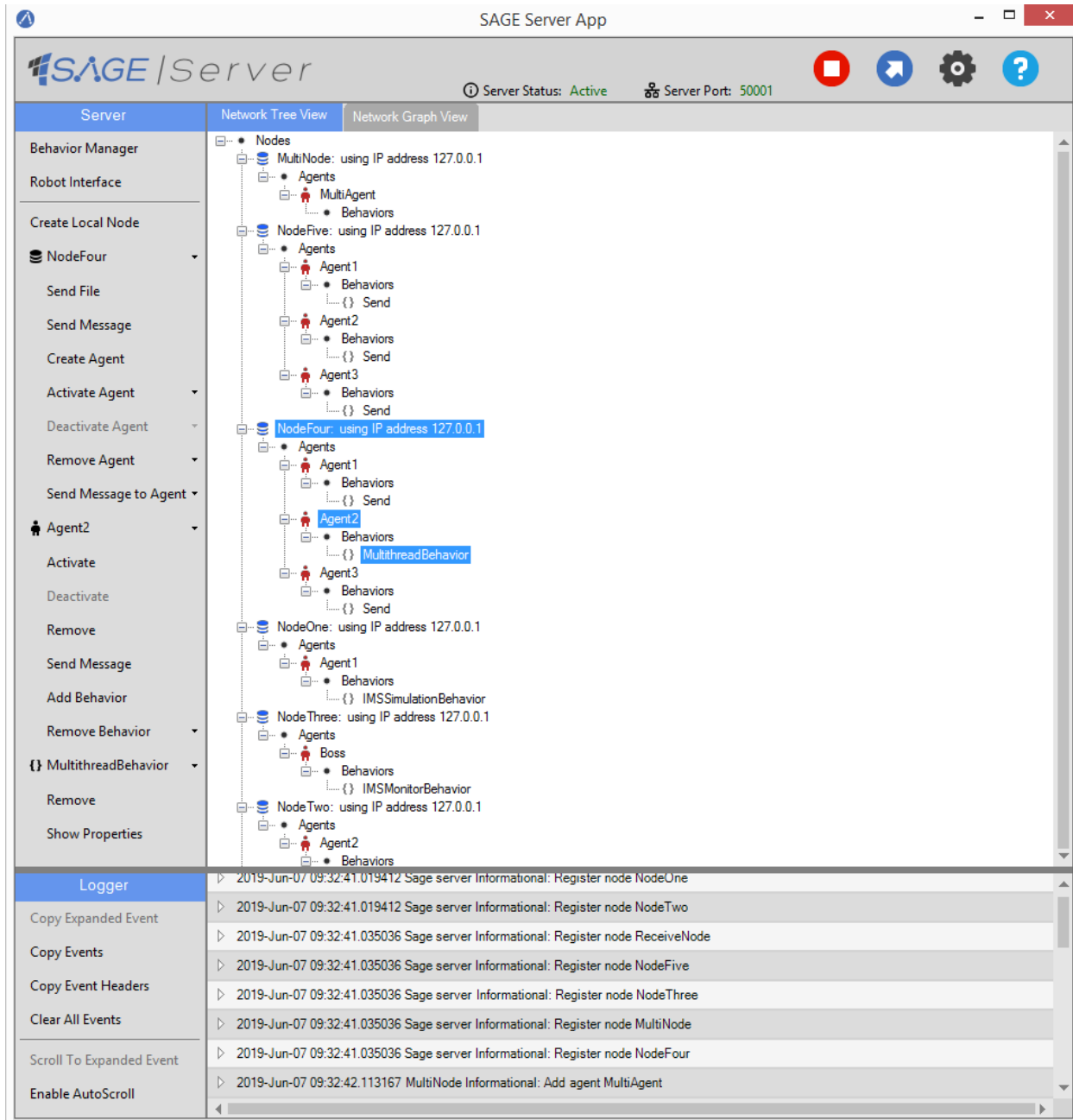
- The root of the network tree is the Nodes tree-node representing the set of all connected SAGE Nodes.
- SAGE Nodes are shown as branches under the Nodes tree-node. Each Node has an Agents tree-node representing the set of all Agents belonging to the Node.
- A Node's Agents are shown as branches under the Agents tree-node. Each Agent has a Behaviors tree-node representing the set of all Behaviors belonging to the Agent.
- An Agent's Behaviors are shown as leaf tree-nodes under the Behaviors tree-node.

Clicking on any tree-node selects and highlights it. Tree-nodes representing ancestor network entities are also selected.

Right-clicking on a tree-node displays a set of cascading context menus with a set of commands relevant to the current selection.

You can expand and collapse tree-nodes by clicking on the plus or minus sign next to a tree-node branch.

Double-clicking on a tree-node fully expands a branch if it is collapsed, and fully collapses a branch if it is expanded.



5.4 Network Graph View

The Network Graph View provides a dynamic graph display of the SAGE network.

- SAGE Nodes are shown as root graph-nodes of their own sub-graphs.
- Graph-nodes representing Agents are connected to their Node graph-node, and graph-nodes representing Behaviors are connected to their Agent graph-node.

Clicking on any graph-node selects and highlights it. Graph-nodes representing ancestor network entities are also selected.

Right-clicking on a graph-node displays a set of cascading context menus with a set of commands relevant to the selected object.

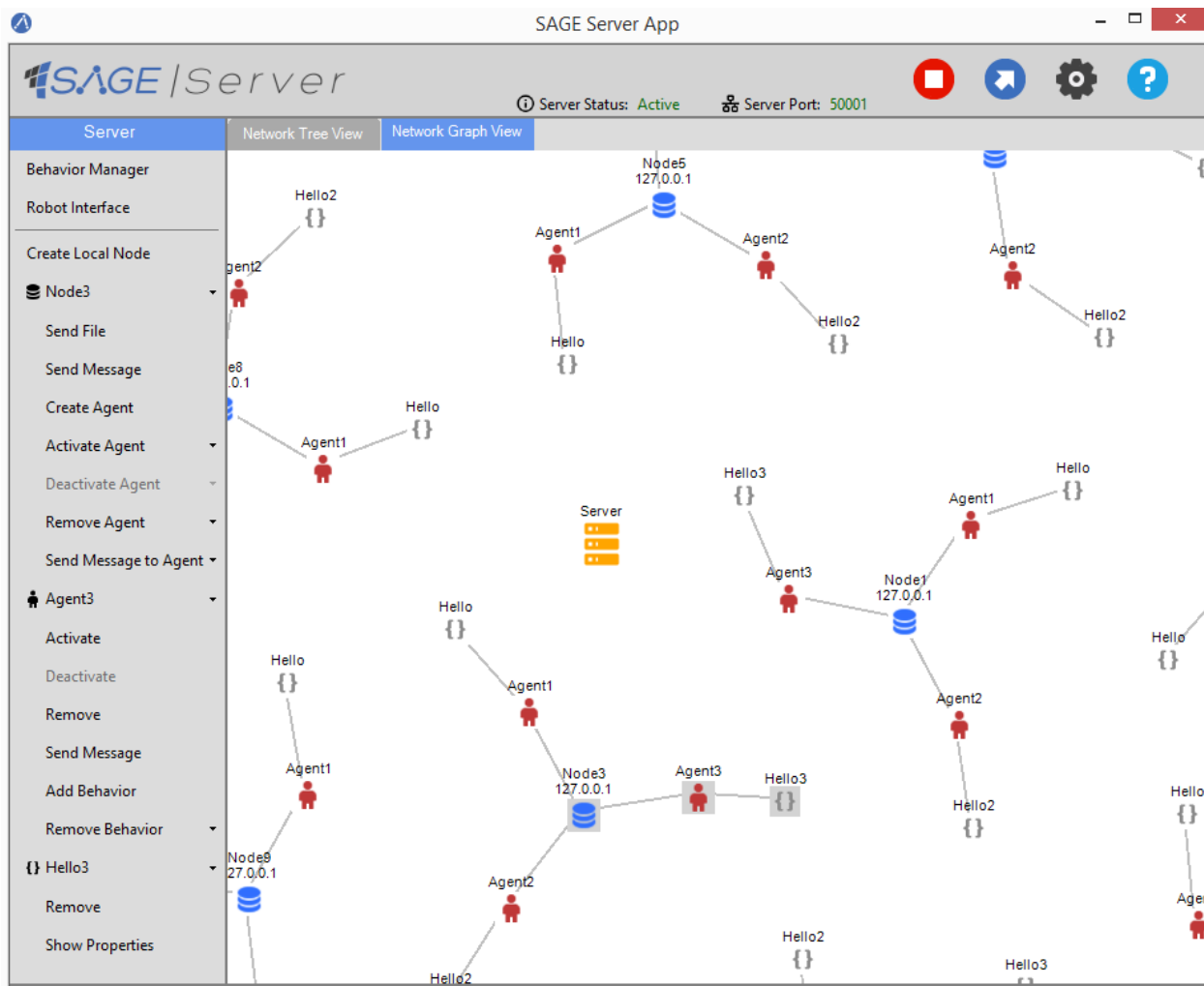
Clicking on the Server graph-node or on the view background displays a context menu relevant to the entire network.

You can reposition entities within the graph region by dragging with the left mouse button.

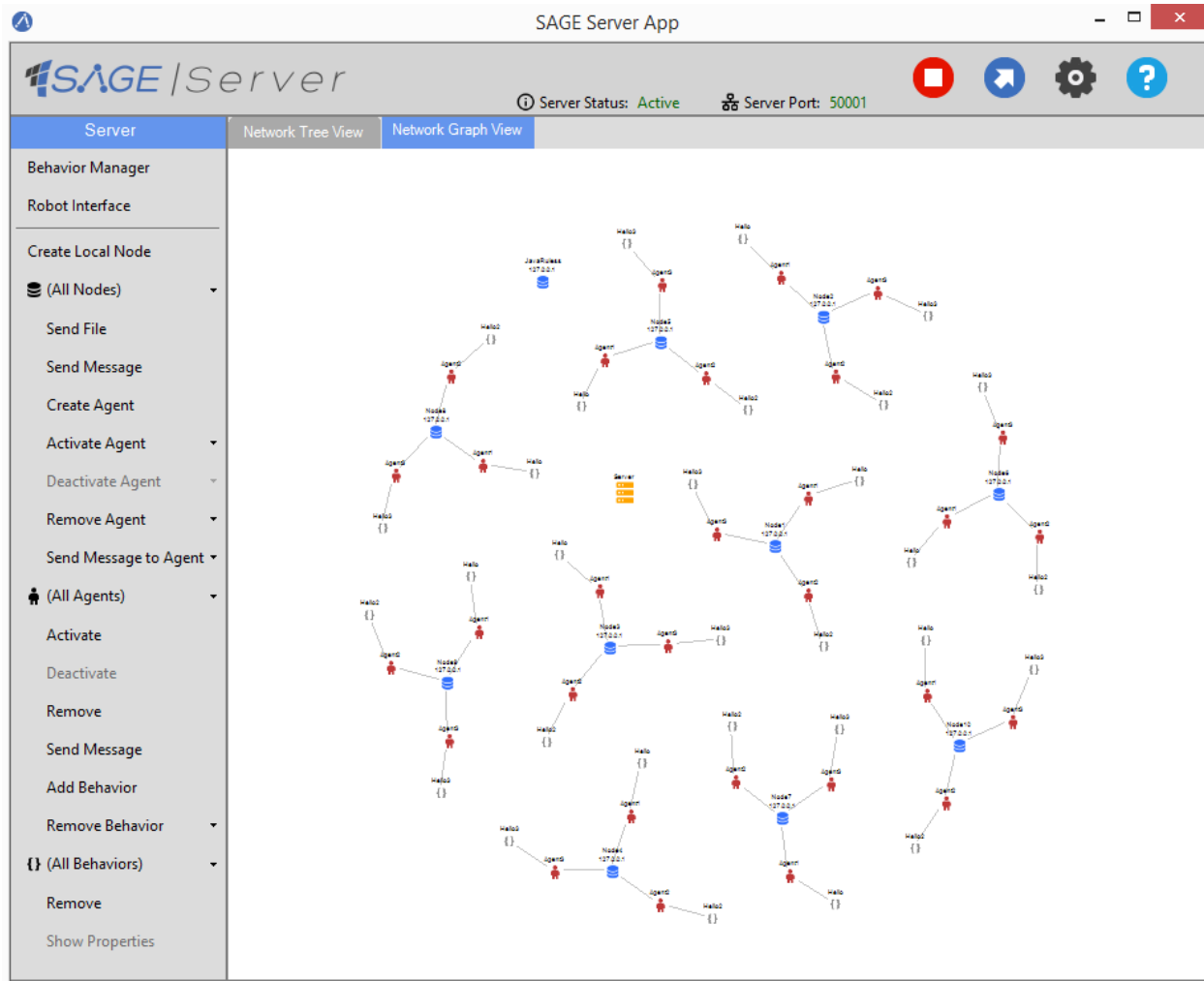
Double-clicking on a graph-node centers the entity within the view pane.

Double-clicking on the view background centers the graph region about that position in the view pane.

The graph view can be panned by clicking down anywhere in the view pane that is not on a graph-node and dragging.



The graph view can be zoomed out and back in using the mouse wheel. When zooming, the scaling is applied about the mouse cursor position.



5.5 Starting the Server

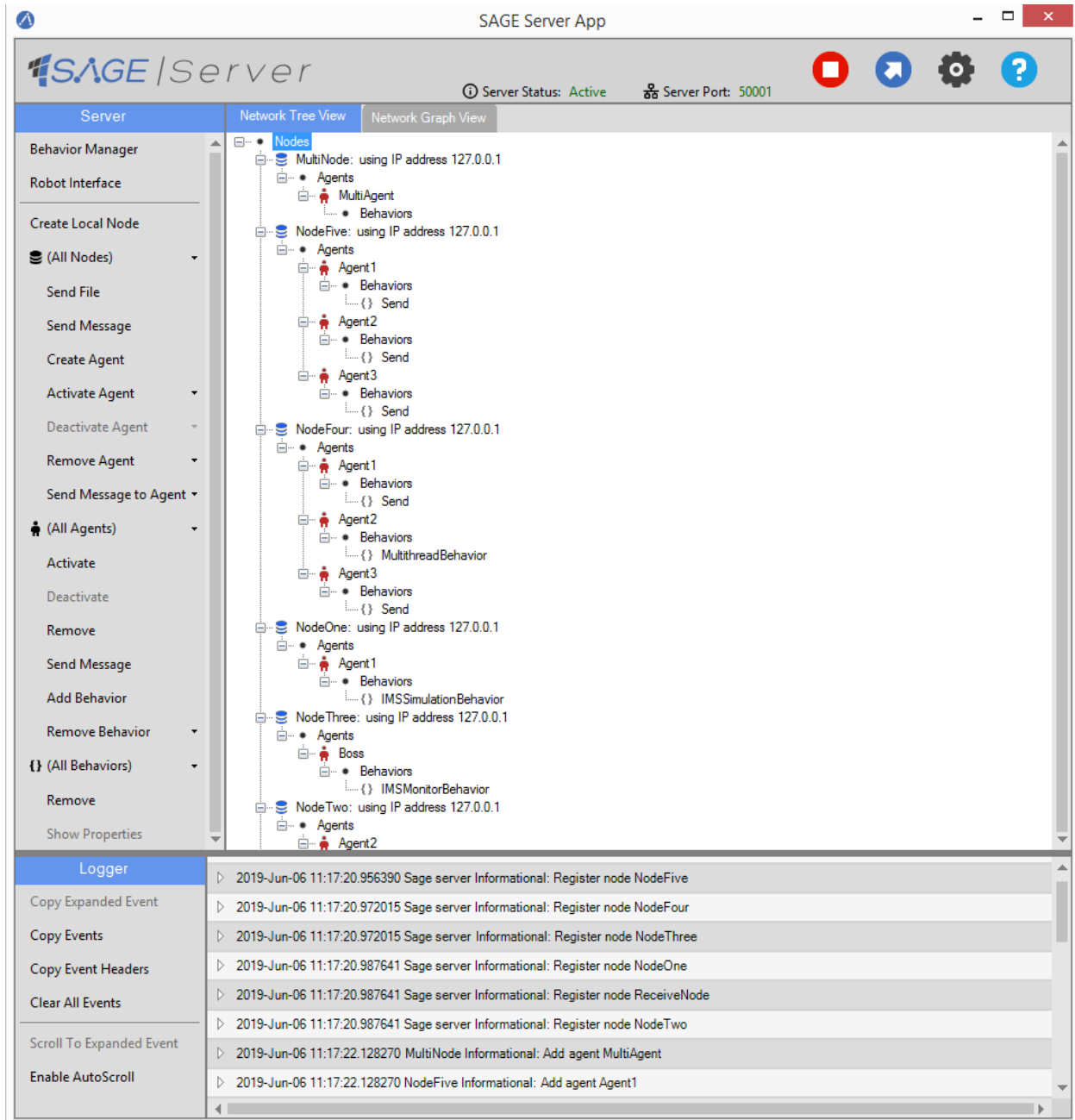
Click the **Start** button or the **Import Network** button in the Control Toolstrip to start the Server instance. After the Server has started, any previously created Nodes will be connected and displayed in the Network View Pane.

If the Server is started using the **Import Network** button, a previously saved network is loaded.

Once the Server becomes active, the **Start** button changes to a **Stop** button. The Server can be stopped at any time by clicking the **Stop** button.

5.6 Active Server

When the Server is active, the current state of the network is reflected in the Server Toolstrip and the Network View Pane.

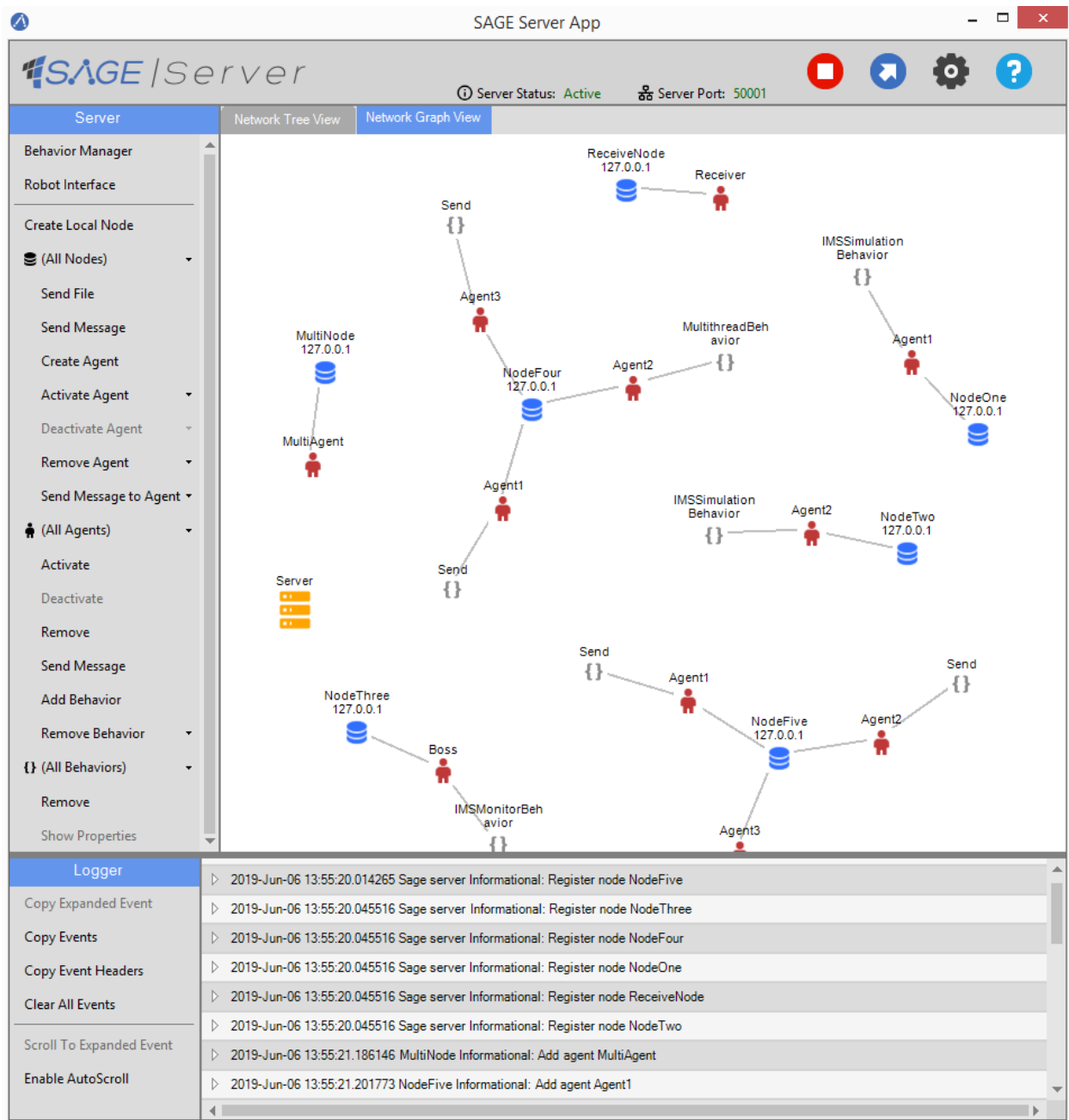


Clicking on the Network Graph View tab displays a graph-based view of the network.

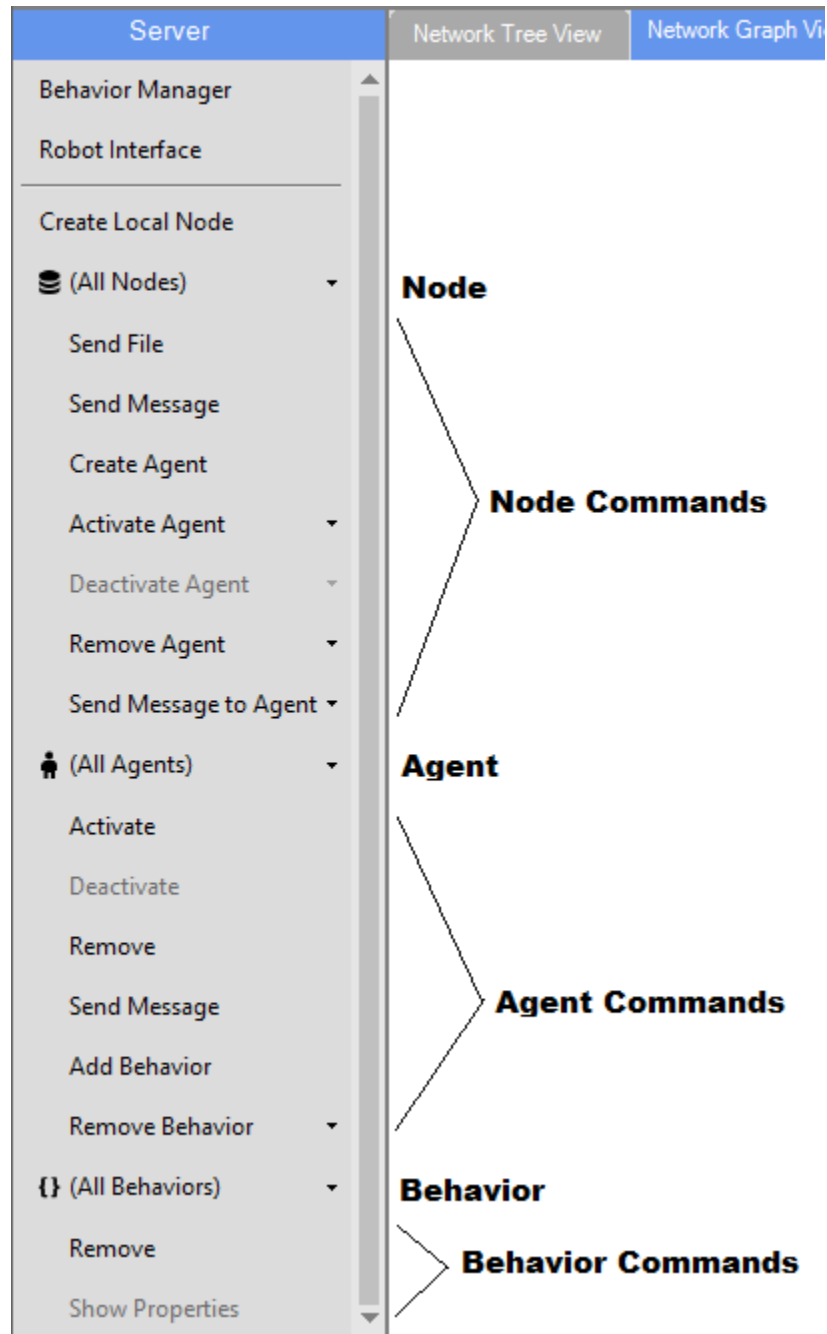
Nodes, their Agents and agent Behaviors are selected using either the ServerToolstrip or Network View Pane.

Commands operate on the current selection and are accessible from either the Server Toolstrip or Network View Pane context menus.

On the Server Toolstrip:



- node-related commands are under the `Node` dropdown button
- agent-related commands are under the `Agent` dropdown button
- behavior-related commands are under the `Behavior` dropdown button.

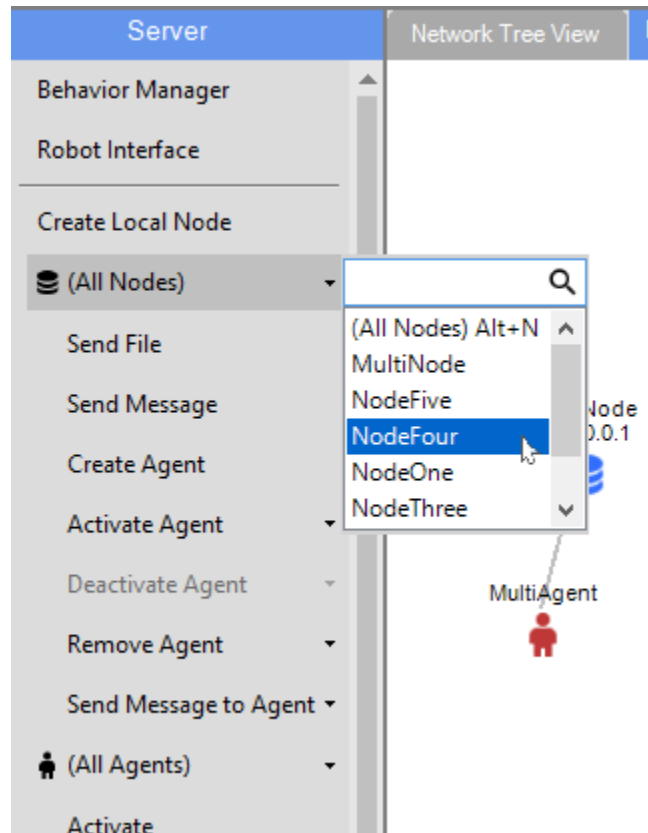


Selecting a SAGE Node, Agent or Behavior

A SAGE object (Node, Agent or Behavior) can be selected from the Server Toolstrip or the Network View Pane.

To select a Node, Agent or Behavior from the Network View Pane, click on the corresponding tree-node or graph-node.

To select a Node, Agent or Behavior from the Server Toolstrip, click on the corresponding dropdown button and select the object's name from the popup combo box, as shown below.



The current selection is reflected in both the Server Toolstrip and Network View Pane (tree view or graph view).

The mechanism for selecting SAGE objects and applying commands is enforced by the topology of the Network Tree and Graph Views and by the Server Toolstrip menu items.

You can quickly search for existing Agents and Nodes using the search menu located in the “Server Toolstrip”

Activating an Agent

Select a Node and Agent option from their respective dropdown list.

When an individual Node, Agent or Behavior is not selected, the corresponding dropdown button will display “(All Nodes)”, “(All Agents)”, or “(All Behaviors)”, and the commands will apply to all objects of that type.

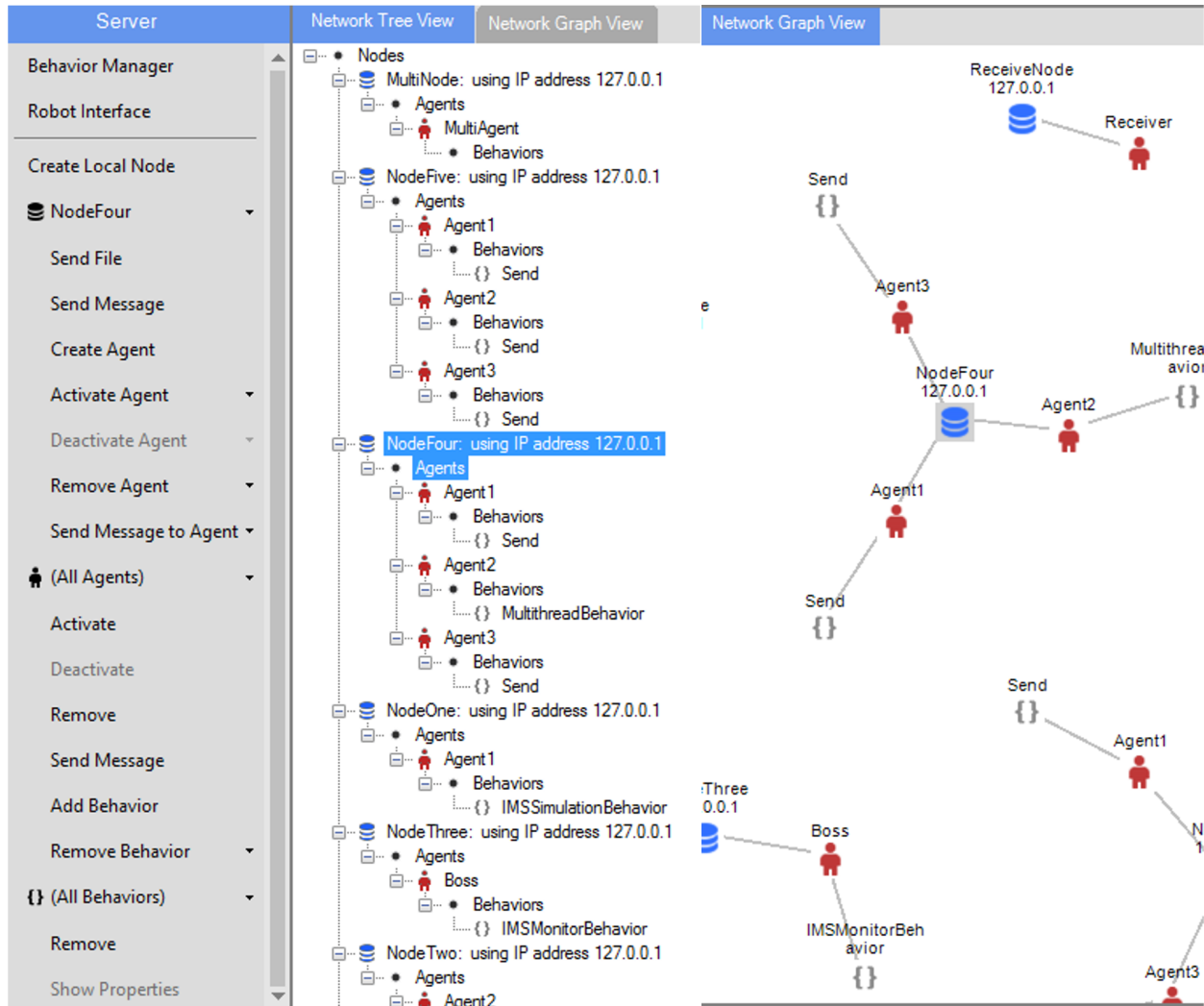
Clicking the agent `Activate` button on the Server Toolstrip activates the agent and node option selected in the dropdown.

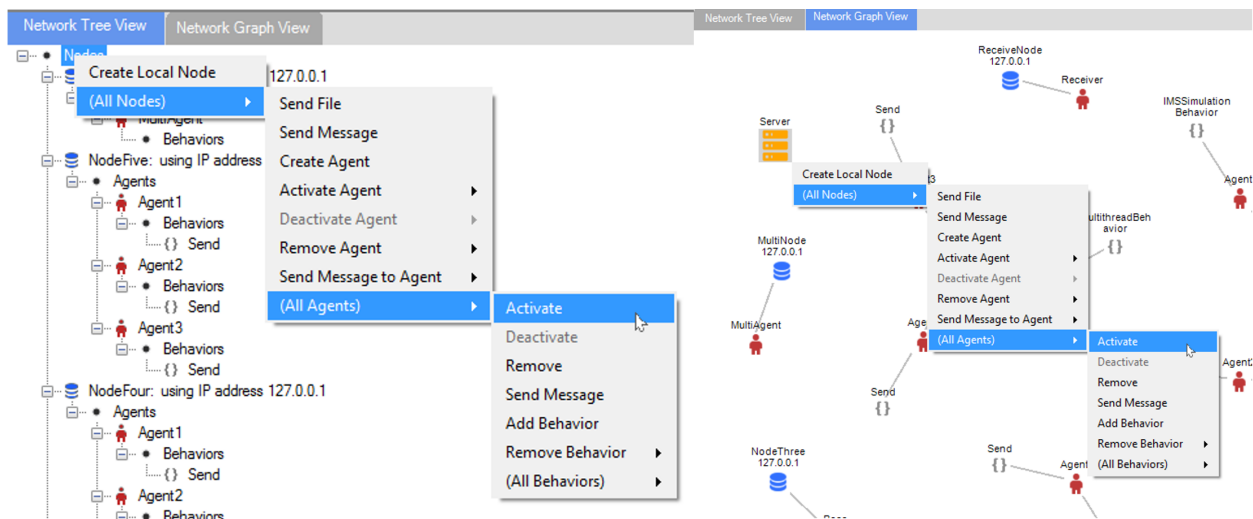
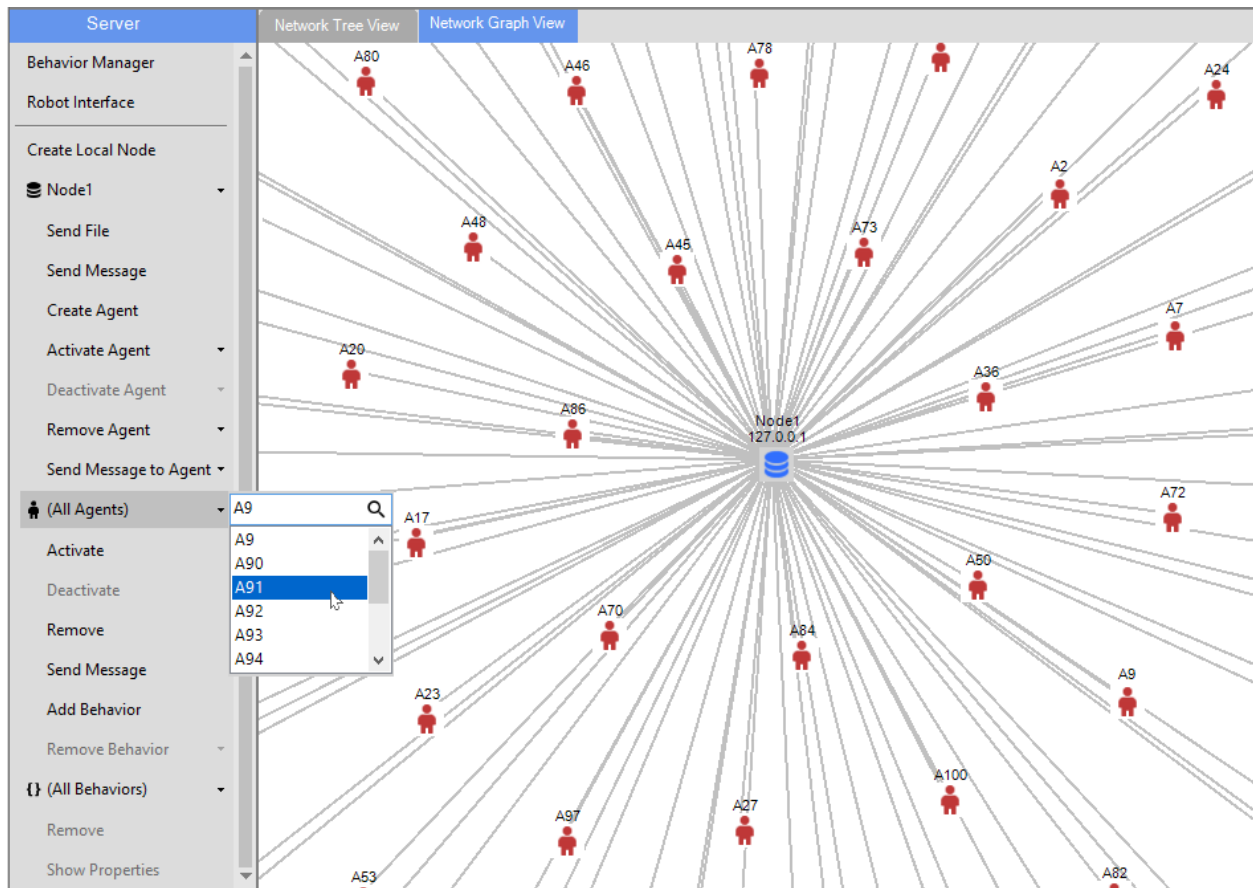
To perform the same operation using the Network View Pane:

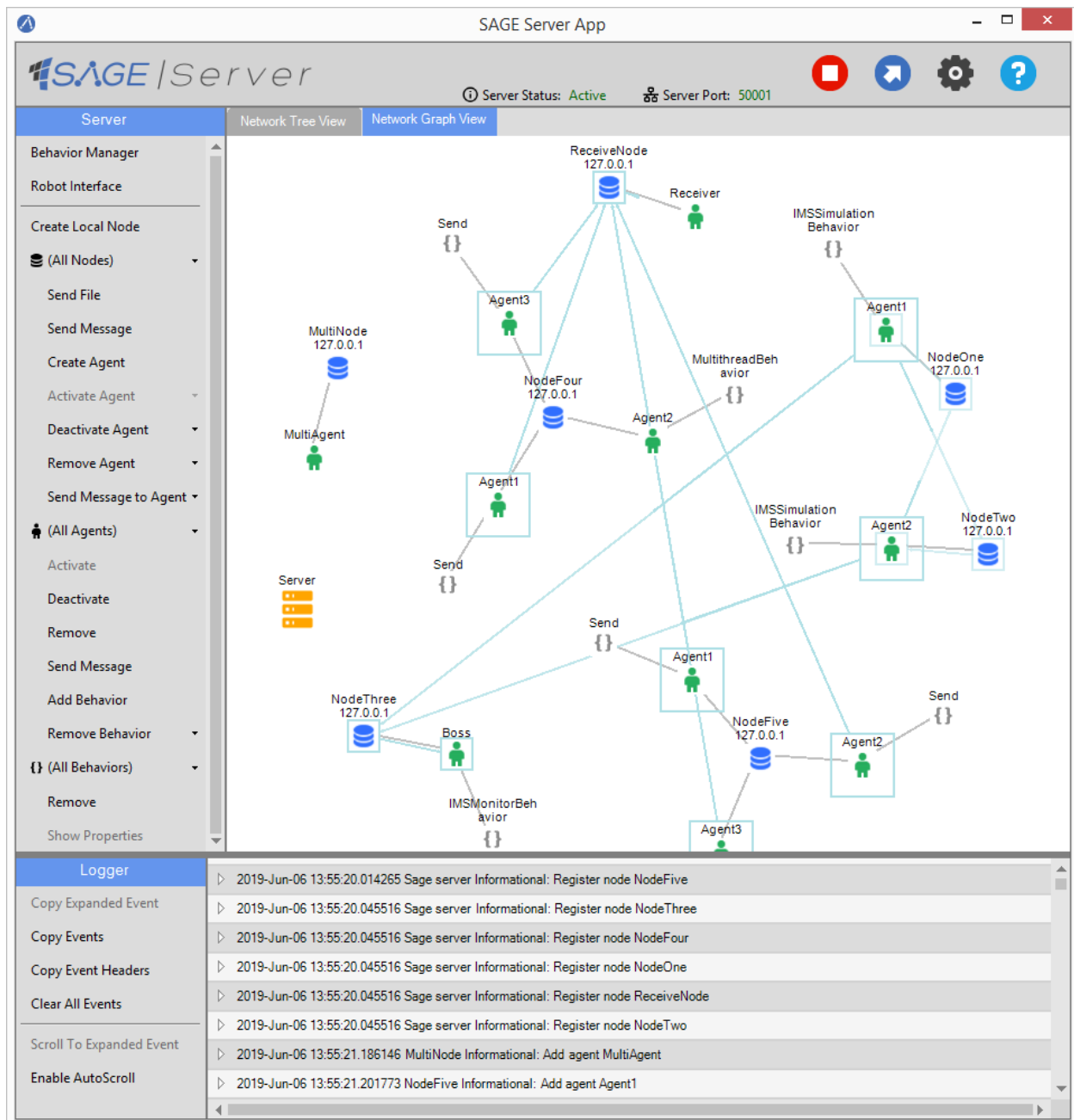
- 1) In the Tree View right-click on the Nodes root-node, or in the Graph View right-click on either the Server graph-node or on the panel background.
- 2) Move the mouse over `(All Nodes)` to display the cascading node context menu.
- 3) Move the mouse over `(All Agents)` to display the cascading agent context menu.
- 4) Click the `Activate` button on the agent context menu.

In the Network Graph View, icons representing the newly activated Agents change color from red (inactive) to green (active), and dynamic messaging lines show the flow of messages sent between Agents.

Click on the Agent `Deactivate` button to deactivate the Agent and Node option selected.





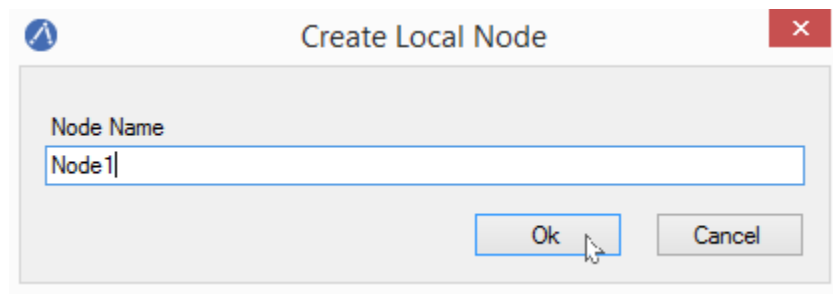


5.7 Creating a Node on the Server machine

A local (Server-based) Node can be created at any time, even when the Server is inactive, from the Server Toolstrip or Network View Pane (tree view or graph view).

To create a local Node:

- 1) Click the **Create Local Node** button from either the Server Toolstrip or the Network View global context menu.
 - To display the global context menu on the Network Tree View, right-click on the Nodes tree-node.
 - To display the global context menu on the Network Graph View, right click anywhere on the view pane background.
- 2) In the popup dialog, enter the Node's name.
- 3) Click OK to confirm, or Cancel to cancel the operation.



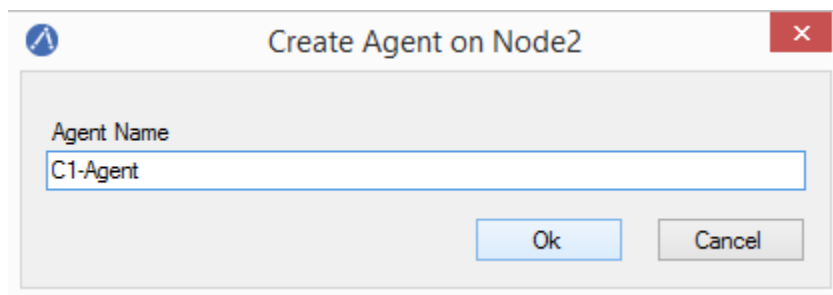
If the Server is inactive, there will be no visible evidence of the Node since the Node is not yet connected to the Server. A Node instance will connect once the Server is started.

5.8 Creating an Agent

To create an Agent on a SAGE Node:

- 1) Select a Node or All Nodes option, then click **Create Agent** from either the “Server Toolstrip” or the “Network View Pane” context menu.
- 2) Enter the Agent's name in the popup dialog.
- 3) Click OK to confirm, or Cancel to cancel the operation.

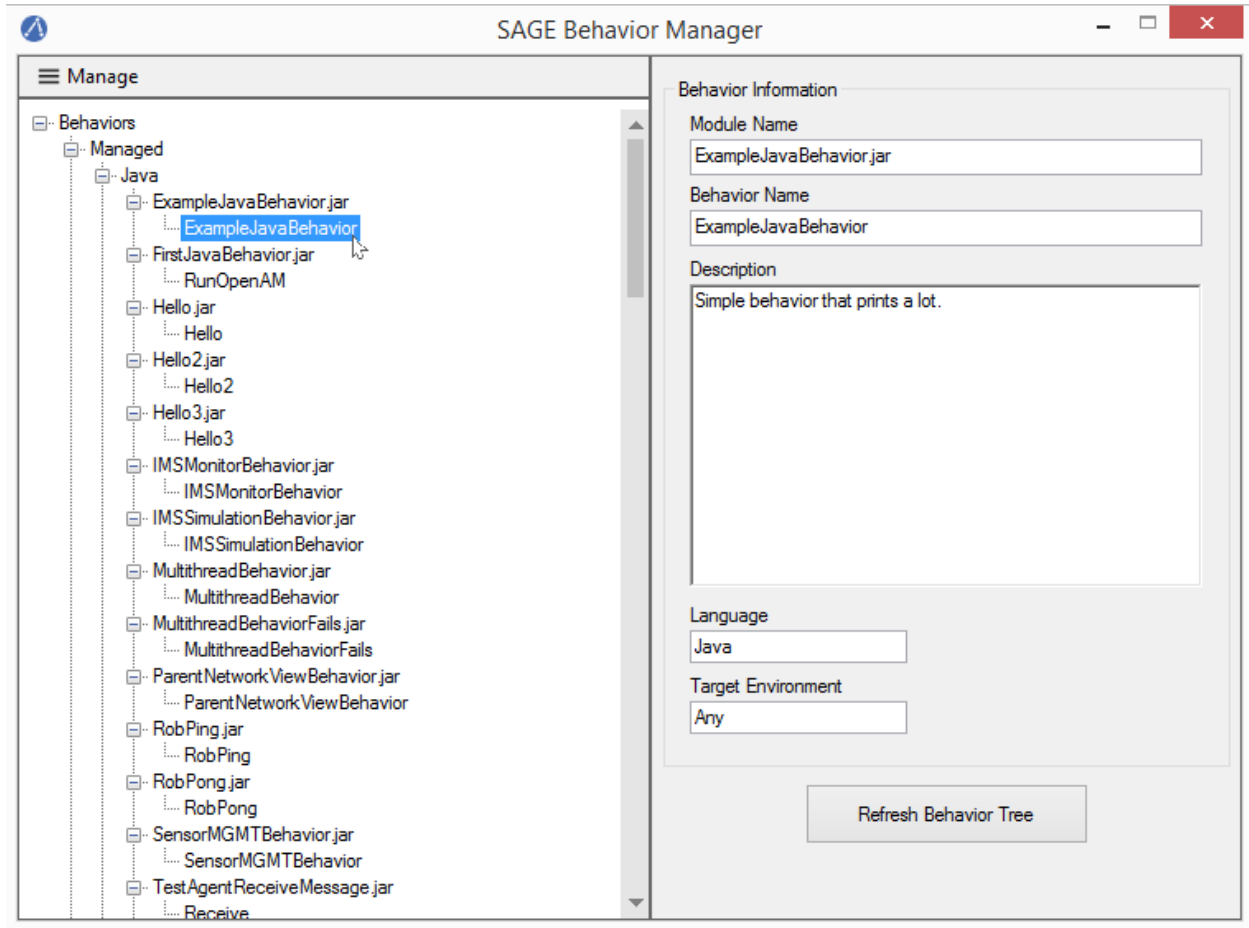
If an agent of that name already exists, an error message is displayed.



5.9 Behavior Manager

Use the Behavior Manager to view and update installed Behaviors.

Click the `Behavior Manager` button on the Server Toolstrip to open the Behavior Manager window.



The left pane contains a tree view of the currently installed behavior modules and their Behaviors.

The right pane contains information about a Behavior when its corresponding tree-node is selected.

Tree-nodes at the first three levels represent hierarchical collections of behavior module types, classified by language and target environment.

- Under `Managed` are the currently supported managed types: “*Java*”, “*JavaScript*” and “*Python*”.
- Under `Native` are the currently supported native types: “*Linux32*”, “*Linux64*”, “*Windows32*” and “*Windows64*” (32-bit and 64-bit Windows and Linux modules).

Management capabilities include adding and deleting Behavior modules.

To add or delete a Behavior from the repository:

- 1) Click the `Manage` button.
- 2) Select either the `Add Modules` or `Delete Modules` button.
- 3) Select the appropriate behavior files.

When a Behavior module is added or deleted, the module is automatically added to or removed from the proper Behavior repository sub-directory.

Behavior Type	Behavior Path
Java	C:\ProgramData\Sage\behaviors\Java
Javascript	C:\ProgramData\Sage\behaviors\JavaScript
Python	C:\ProgramData\Sage\behaviors\Python
Windows32	C:\ProgramData\Sage\behaviors\Windows32
Windows64	C:\ProgramData\Sage\behaviors\Windows64
Linux32	C:\ProgramData\Sage\behaviors\Linux32
Linux64	C:\ProgramData\Sage\behaviors\Linux64

Click the `Refresh Behavior Tree` button to synchronize the behavior tree with the Behavior repository when behaviors are installed or uninstalled from the repository outside of the app while the Behavior Manager window is open.

The Behavior Manager is available when the Server is both active and inactive and can remain open while the user interacts with the main form window.

5.10 Adding a Behavior to an Agent or Node

To add a Behavior to an Agent or Node:

- 1) Select a Node and Agent option from the Server Toolstrip or from the entity's network view context menu.
- 2) Select the Behavior module from the list of available Behaviors.
- 3) Click `Ok` to confirm or `Cancel` to cancel the operation.

The following popup dialog will appear.

To specify topics for the Behavior:

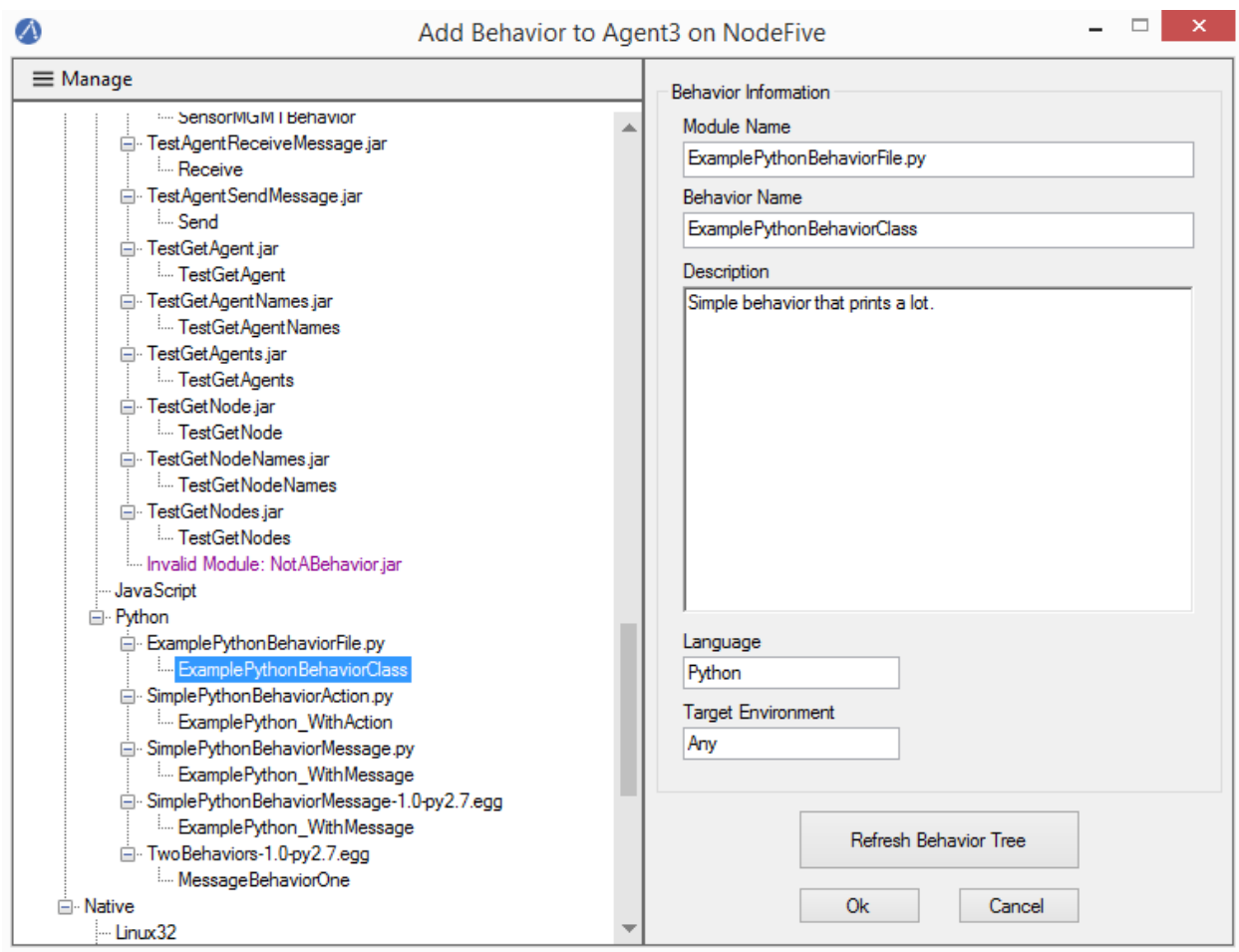
- 1) Enter the desired behavior topic strings. If topics have been added to a previous Behavior, button `Paste Previous Topics` will be enabled. This will automatically add the previous topics to this behavior.
- 2) Click `Ok` to continue or `Cancel` to continue without adding a topic.

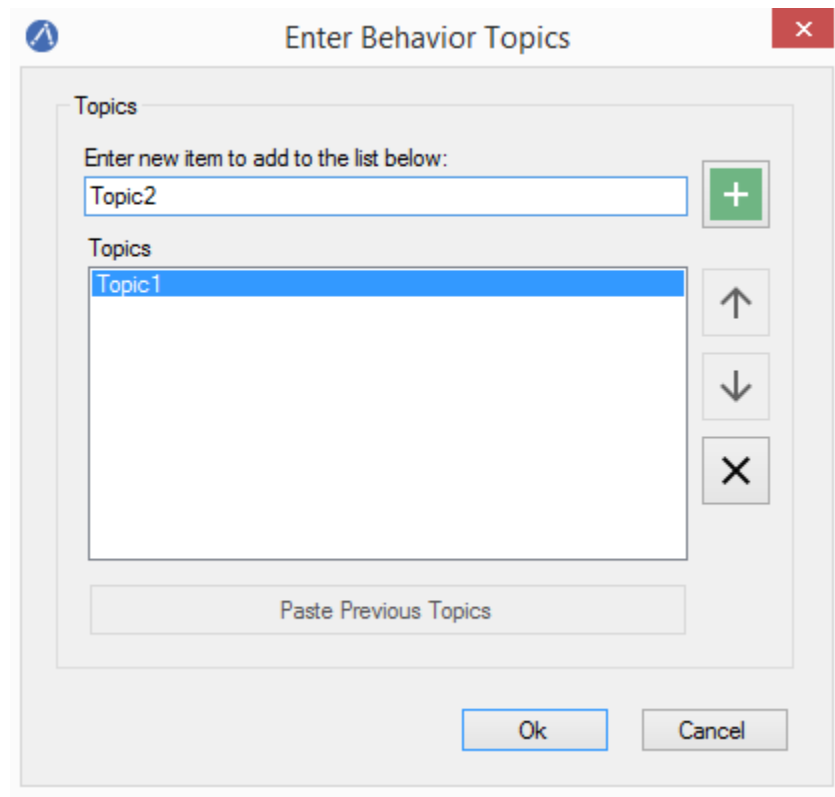
5.11 Sending a message

The `Send Message` command allows the user to send a message from the SAGE server to one or more Nodes and their Agents.

To construct and send a message:

- 1) Select `Node` and `Agent` option from the Server Toolstrip or from the entity's network view context menu.
- 2) Click `Send message` button.
- 3) Enter the message text.
- 4) Enter the topic of the message.
- 5) Enter any data items.
- 6) Click `Ok` to confirm or `Cancel` to cancel the operation.





Message sending is graphically depicted in the Network Graph View. Messages are shown traveling from the source (either the Server or an Agent) to one or more destination (SAGE Nodes and Agents).

The source entity is highlighted with a large rectangle and the destination entity are highlighted with small rectangles.

Server messages (generated using the Send Message command) are shown in orange.

Agent messages (generated by agent Behaviors) are shown in light blue.

5.12 Sending files to a Node

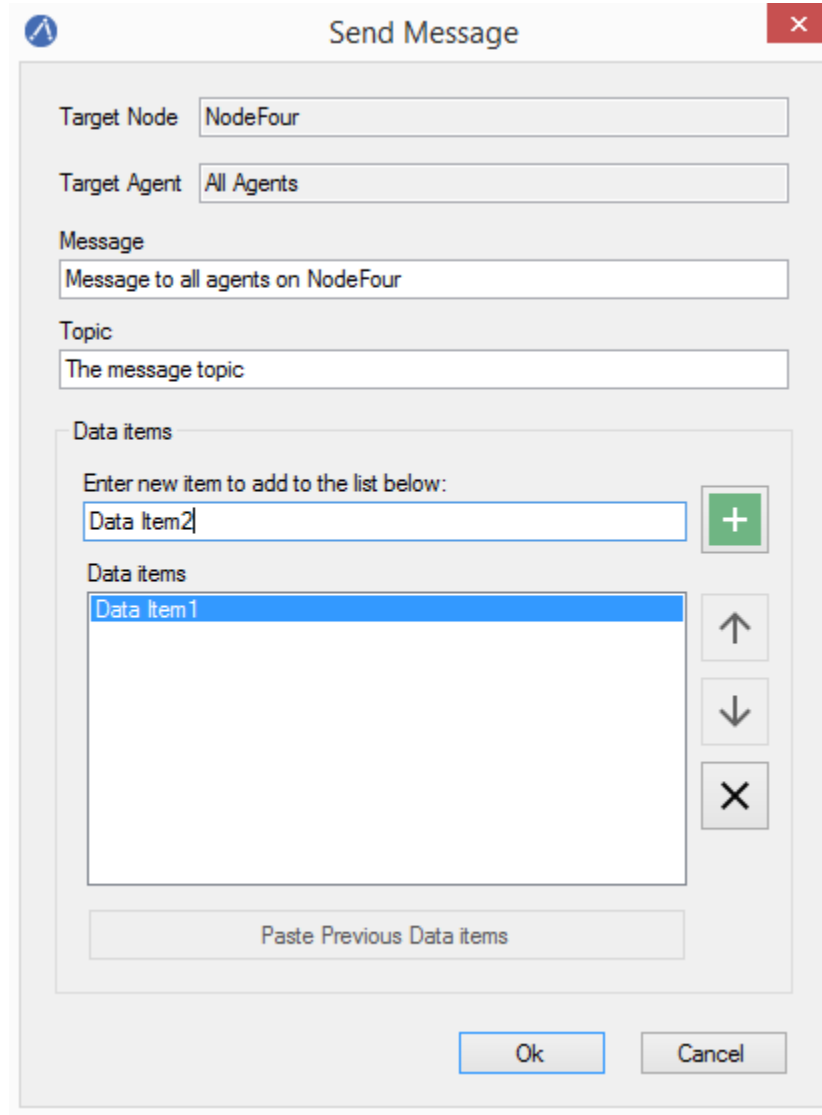
To send one or more files to a node:

- 1) Select the Node from the Server Toolstrip or Network View context menu.
- 2) Click `Send File` button.
- 3) Navigate to the desired folder and select the file or files to send.
- 4) Click `Open` to confirm, or `Cancel` to cancel the operation.

A sub-folder with the same name as the Node will be created (if it does not already exist) in the SAGE data directory on the targeted Node machine where the sent files will be copied to.

`C:\\ProgramData\\SAGE\\data\\NodeName`

Files will be sent to all Nodes if no Node is specified.



The "Send Message" dialog box is a standard Windows-style window with a title bar containing a help icon, the text "Send Message", and a close button. The main area contains several input fields and a list. The "Target Node" field is set to "NodeFour". The "Target Agent" field is set to "All Agents". The "Message" field contains the text "Message to all agents on NodeFour". The "Topic" field contains the text "The message topic". Below these is a "Data items" section. It includes a label "Data items", a text input field with "Data Item2", and a green "+" button. Below this is a list box containing "Data Item1", which is selected. To the right of the list box are three buttons: an up arrow, a down arrow, and a close button (X). At the bottom of the "Data items" section is a button labeled "Paste Previous Data items". At the bottom of the dialog are "Ok" and "Cancel" buttons.

Send Message

Target Node: NodeFour

Target Agent: All Agents

Message: Message to all agents on NodeFour

Topic: The message topic

Data items

Enter new item to add to the list below:

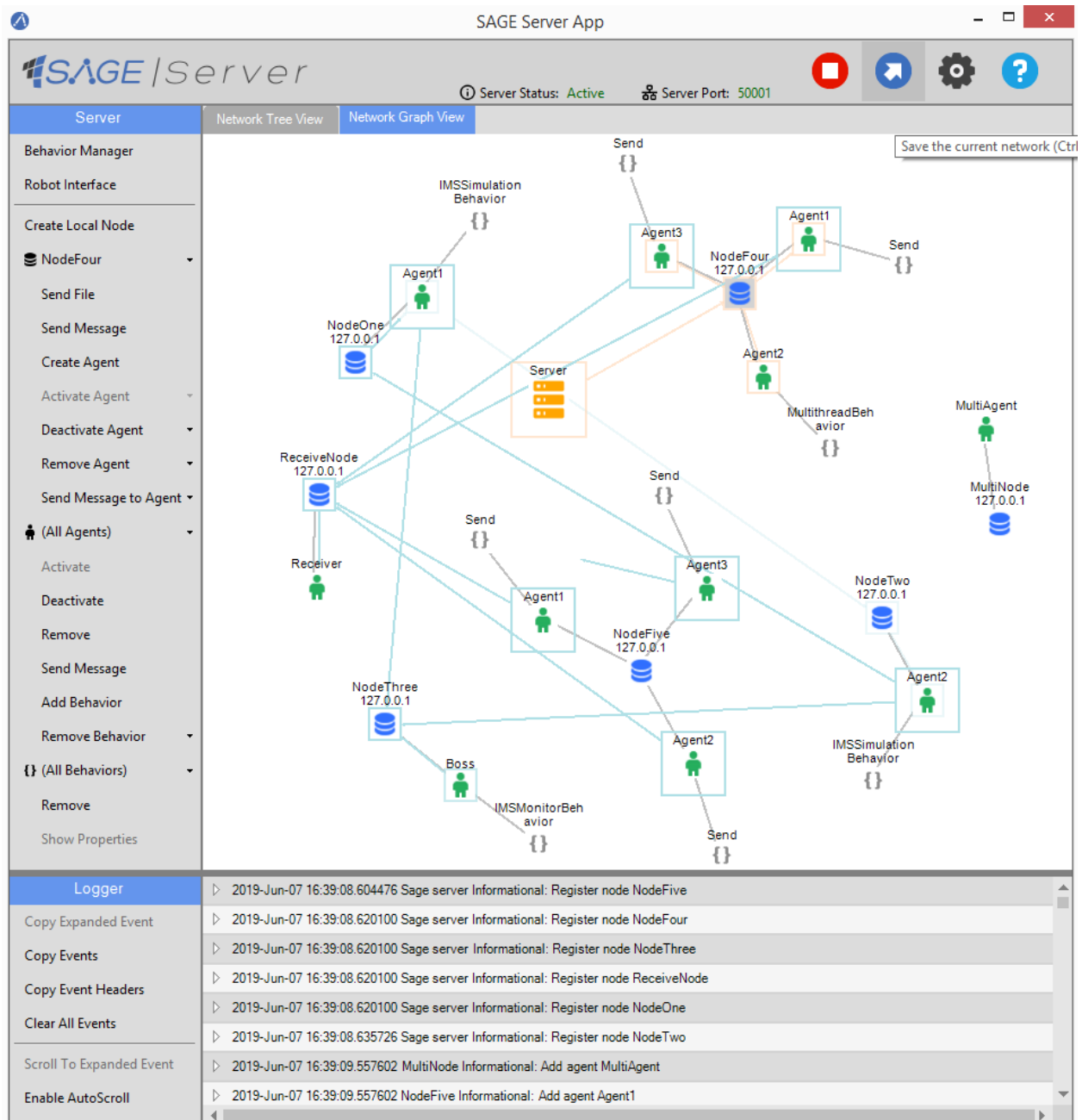
Data Item2

Data items

Data Item1

Paste Previous Data items

Ok Cancel



5.13 SAGE Robot Interface

The SAGE Robot Interface allows you to run Robot scripts and view the results from within the SAGE App.

Click the `Robot Interface` button on the “Server Toolstrip” to start a new instance of the SAGE Robot Interface, as shown below.

At the top of the window are controls to select and sequence the Robot script files.

Robot scripts are added, removed and sequenced using the buttons to the right of the Robot scripts list.

The Add and Remove operations are also available from the **File** menu.

To run a Robot script:

- 1) Open the Robot Interface
- 2) Click the Add button.
- 3) Select the Robot file and click Open.
- 4) Repeat steps 1 and 2 to add additional Robot scripts.
- 5) Click the Start button to start the run.

The Robot scripts will run one at a time in the sequence specified.

Clicking the red `Stop` button will abort the run.

When a Robot script successfully completes, the pathname of the generated report file appears in the report list at the bottom of the window.

To view a report, select it from the report list and click the `Report` button located in the bottom-right corner. You may also right-click the report pathname and select `View Report` from the context menu.

5.14 Log viewer

The display of Server-generated log events are handled by the Log Viewer.

The types of log events include exceptions, errors, warning, informational messages, and behavior execution results.

By default a single-line header is displayed showing the timestamp, event type, and a brief description.

Message and result events can be expanded to show additional information.

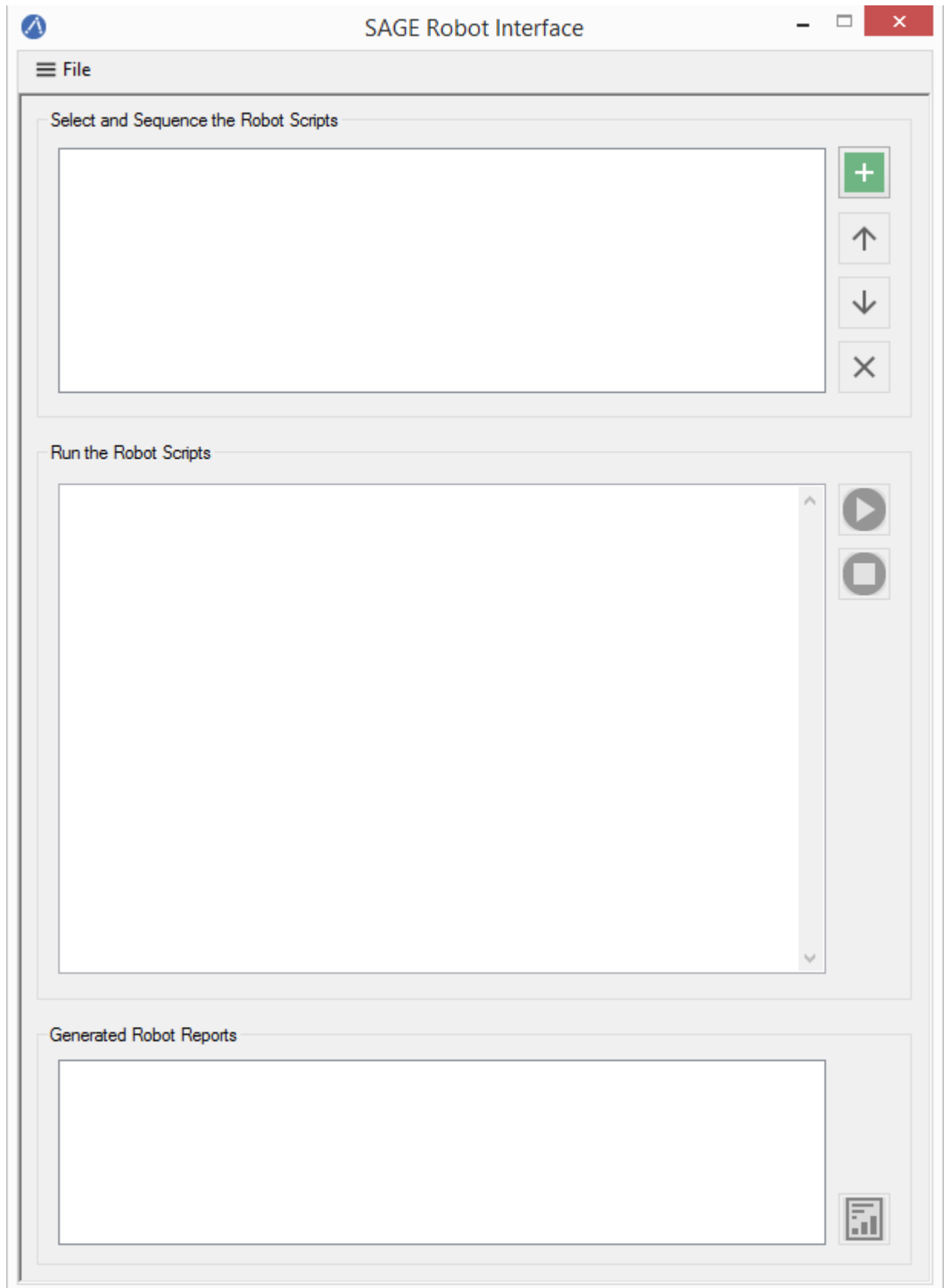
Only one log event can be expanded at a time.

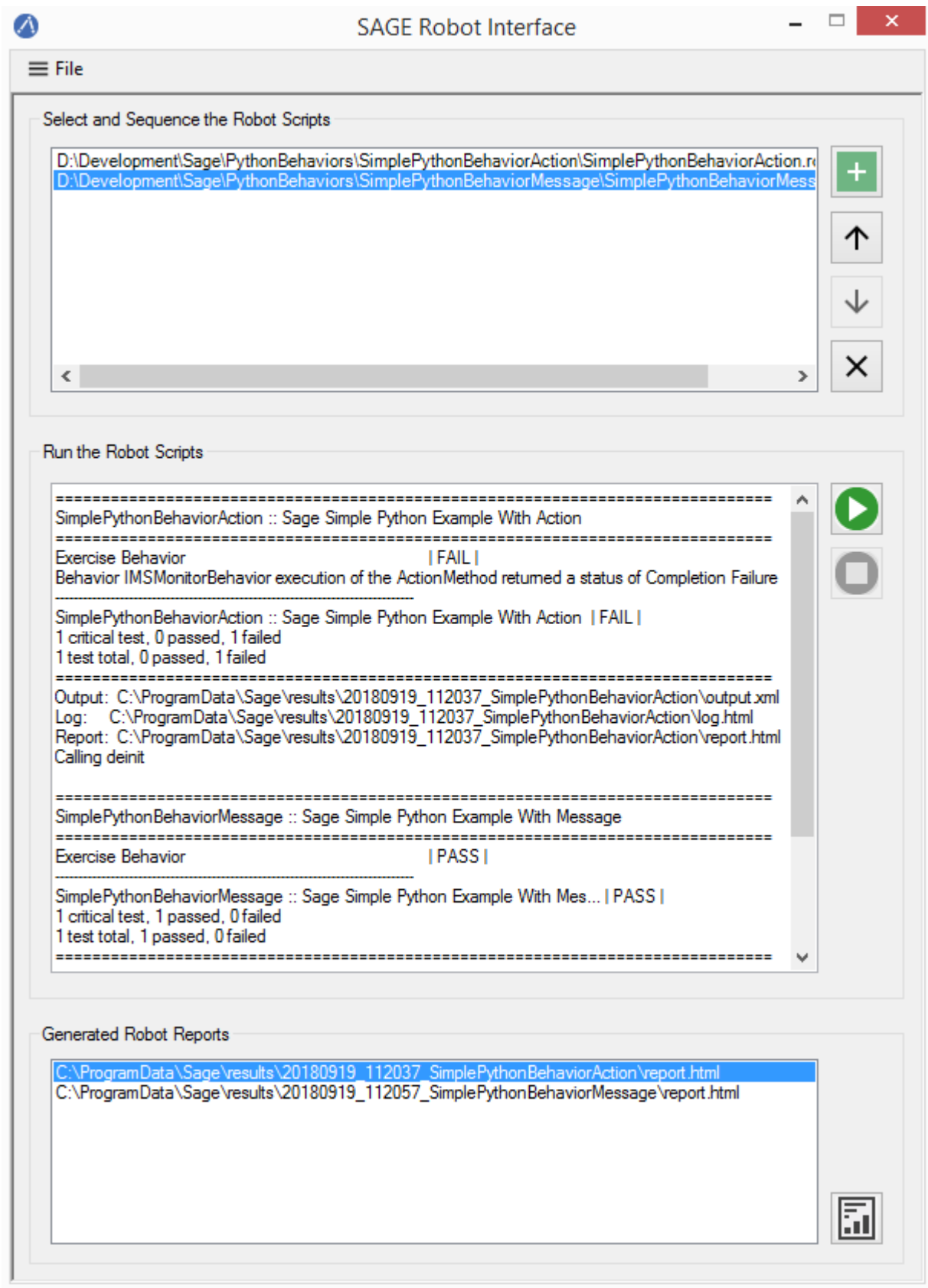
The Logger Toolstrip provides buttons for copying log events to the Windows clipboard and for clearing the Log Viewer. These commands are also available on the context menu displayed when right-clicking on a log event.

When the Server is stopped, currently displayed event logs remain.

All events, including removed events, are saved to a log data file located in the log directory

`C:\\ProgramData\\SAGE\\logs`





SAGE Server App

Server Status: Active Server Port: 50001

Server

Behavior Manager

Robot Interface

Create Local Node

(All Nodes)

Send File

Send Message

Create Agent

Activate Agent

Deactivate Agent

Remove Agent

Send Message to Agent

(All Agents)

Activate

Deactivate

Remove

Send Message

Add Behavior

Remove Behavior

(All Behaviors)

Remove

Show Properties

Network Tree View

Network Graph View

IMSSimulation Behavior

Agent1

NodeOne 127.0.0.1

IMSMonitorBehavior

Boss

NodeThree 127.0.0.1

Send

Agent3

NodeFive 127.0.0.1

Agent1

Send

MultithreadBehavior

Agent2

NodeFour 127.0.0.1

Agent1

Send

Server

Agent2

NodeTwo 127.0.0.1

Agent2

IMSSimulation Behavior

MultiAgent

MultiNode 127.0.0.1

ReceiveNode 127.0.0.1

Receiver

Logger

Copy Expanded Event

Copy Events

Copy Event Headers

Clear All Events

Scroll To Expanded Event

Enable AutoScroll

2019-Jun-08 10:44:31.810972 Sage server Object: Result

2019-Jun-08 10:44:32.123474 Sage server Object: Message

target node: ReceiveNode

source node: NodeFive

target agent: Receiver

source agent: Agent1

topic:

message: TestMessage

data[0]: data1

data[1]: data2

2019-Jun-08 10:44:32.123474 Sage server Object: Message

Copy Expanded Event

Copy Events

Copy Event Headers

Clear All Events

Scroll To Expanded Event

Enable AutoScroll

5.15 Import and export network

Agent network configurations can be saved and loaded. The files are saved as a JSON file with the extension “.sage”.

Export agent network

The current network configuration can be exported (saved) when the Server is active.

To export an Agent network:

- 1) Click the `Export Network` button on the “Control Toolstrip”.
- 2) Navigate to the desired folder
- 3) Enter a filename for your network configuration.
- 4) Click the `Save` button.

Import agent network

A previously saved network configuration can be imported (loaded) when the Server is inactive.

To import an Agent network:

- 1) Click the `Import Network` button on the “Control Toolstrip”.
- 2) Navigate to the folder containing the previously saved network file and select the file.
- 3) Click the `Open` button.

Prior to building the network, SAGE will perform a validation check on the JSON tree by ensuring that all Nodes referenced in the network exist and that all Behaviors to be added to the Agents exists in the Behavior repository.

Any missing local Nodes will be automatically created. If any other errors remain, an “Agent Network Load Errors” dialog window will appear listing the errors.

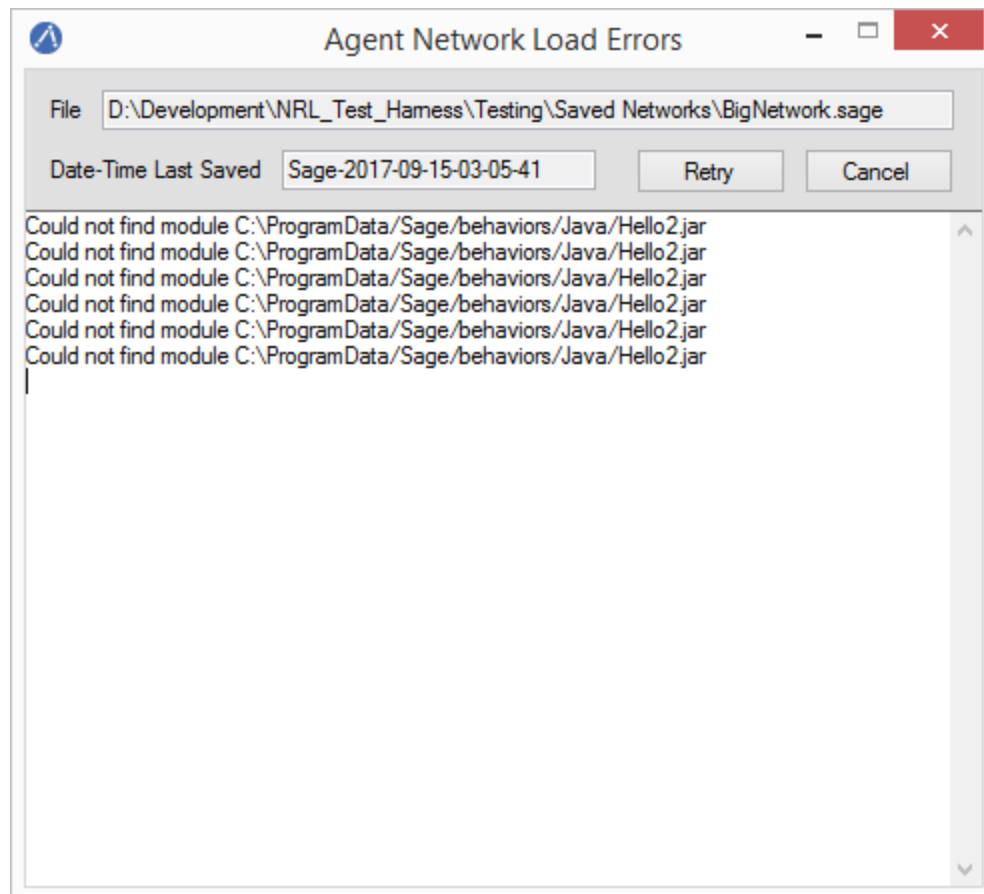
The dialog window remains open while the user attempts to resolve the issues.

Click the `Retry` button to retry loading the network configuration, or click `Cancel` to abort the operation.

If no errors persist, the Server is started and the saved network is fully configured.

Download

Start using SAGE to automate your processes.



DOWNLOAD

Note that the SDK is required to develop C++ Behaviors.

SAGE Framework v2.0.28	
Releases	
Windows x64 - Server	Download
Windows x86 - Server	Download
Windows x64 - Node	Download
Windows x86 - Node	Download
Windows x64 - SDK	Download
Windows x86 - SDK	Download
Linux x64 - Server	Download
Linux x64 - Node	Download
Linux x64 - SDK	Download
Example - Sender and Receiver	Download

SAGE Behavior Template Files	
C++	Download
Java	Download
Python	Download

Installation

5 easy-to-follow steps.

INSTALLATION

7.1 Requirements

Note that SAGE may be compatible with other versions of Windows and Linux distributions not listed.

- Windows 7, 8, 8.1, 10, Server 2013
- Red Hat Enterprise Linux 7
- Ubuntu 14

7.1.1 Optional

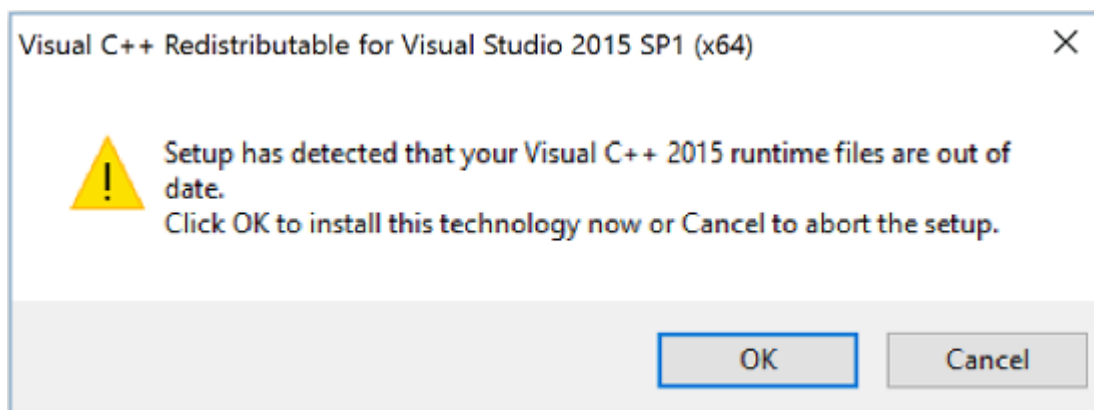
- Python 2.6 or 2.7
- Java SE Runtime Environment 8 and up.

7.2 Windows

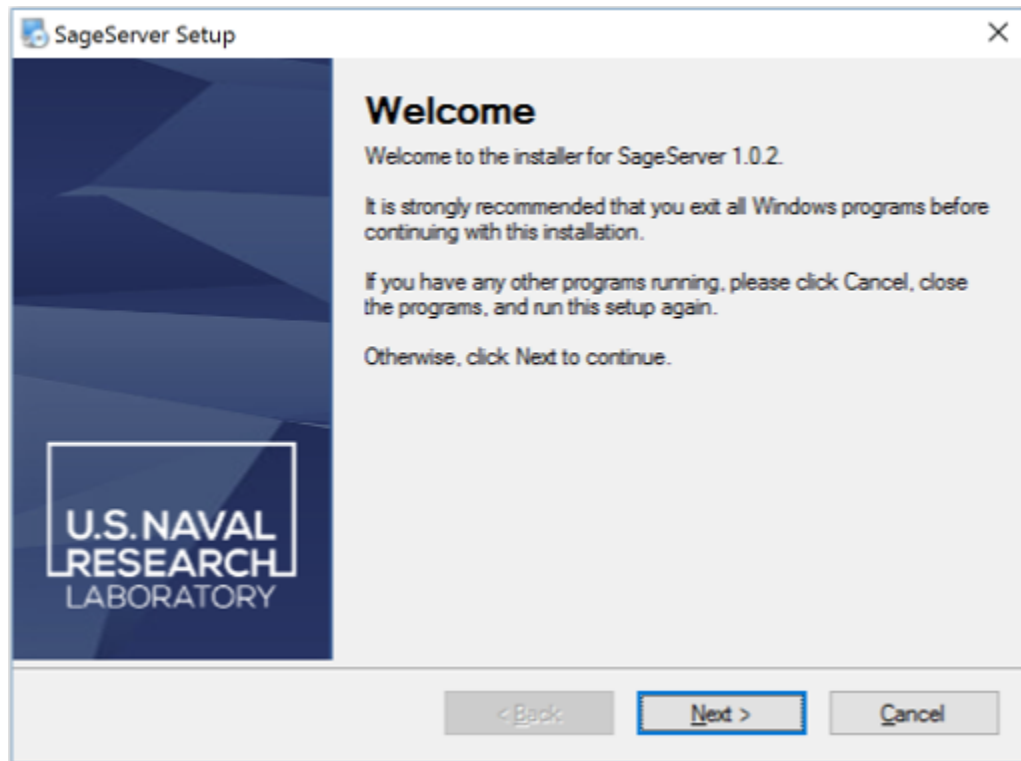
7.2.1 1. Start the SAGE Server or Node Installer executable

SAGE Server and Node softwares are packaged independently and include separate installers.

After the SAGE distribution has finished downloading it should be available in your download directory. Start the SAGE Server or SAGE Node Installer executable. You may be asked to install Visual C++ Runtime. If so, select 'OK' and proceed. Select repair if needed.



Select the 'Next' button to continue with installing SAGE.



7.2.2 2. Review the SAGE license agreement

Review the terms of agreement. To proceed, agree to the terms of the license agreement.

Select the 'Next' button to continue.

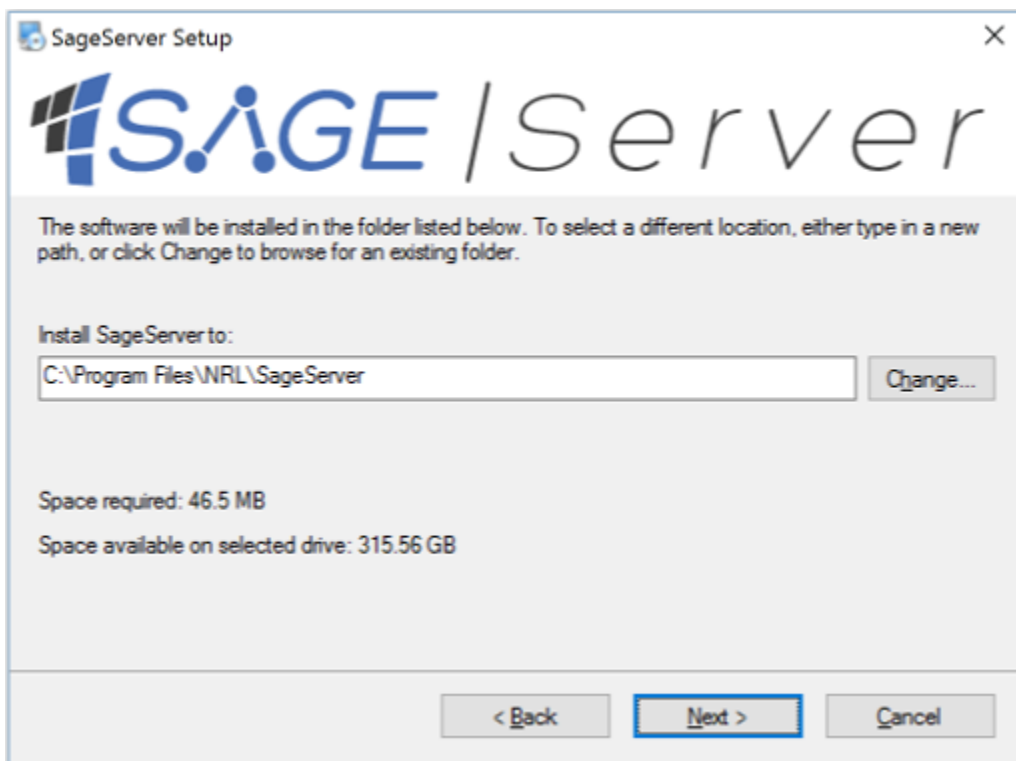
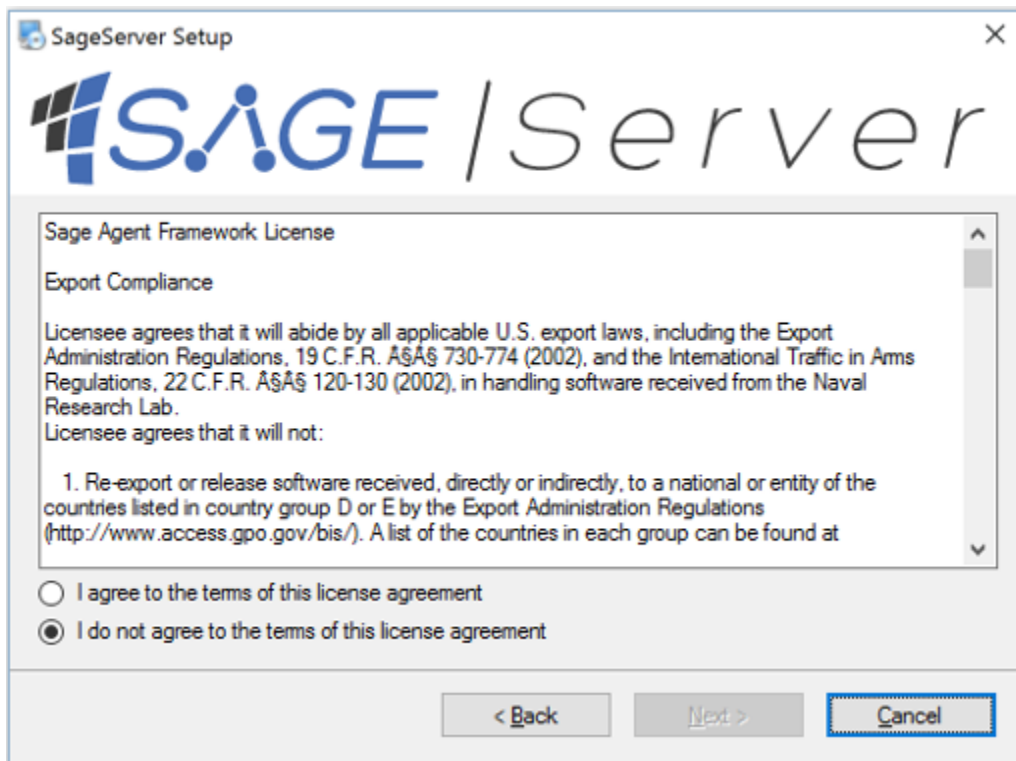
7.2.3 3. Select your installation folder

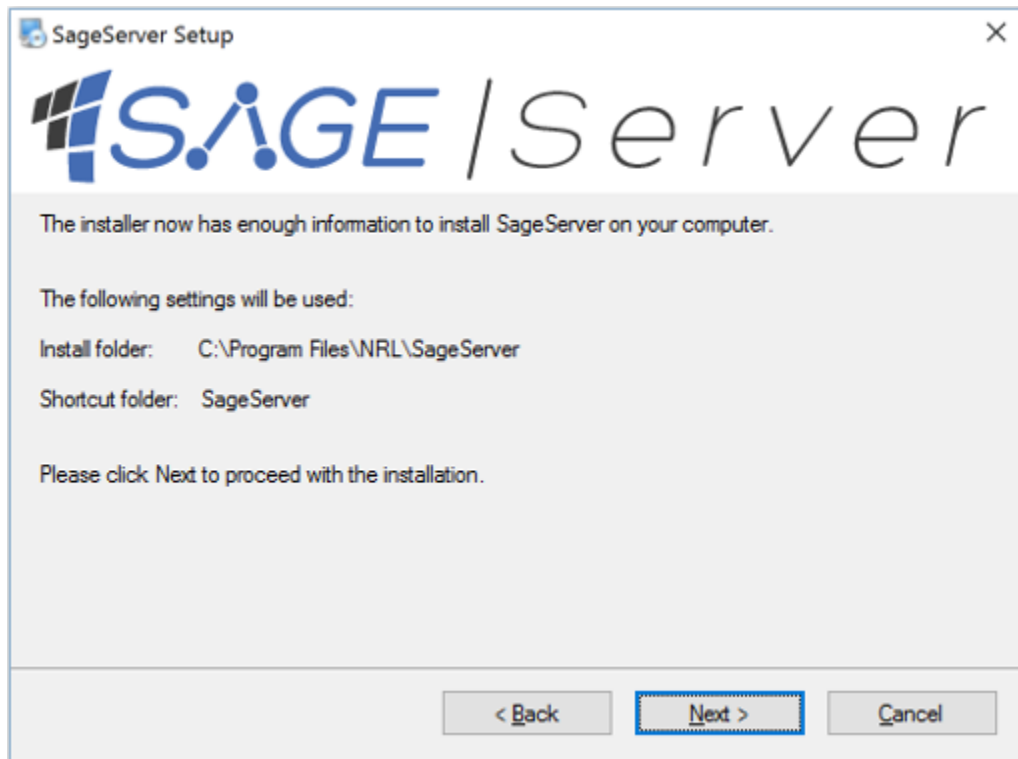
Specify the folder where you want SAGE to be installed. The default folder will be in your Program Files directory.

Select the 'Next' button to continue.

7.2.4 4. Confirm SAGE installation

Select the 'Next' button to begin the installation. You may get a security warning to run this file. Select 'Yes' to allow SAGE to proceed with installation.





7.2.5 5. Installation complete

Once the installation is complete you can now launch SAGE.

7.3 Environment variables

The following System variables will be automatically added:

SAGE_SERVER_HOME

```
C:\Program Files\NRL\SageServer
```

SAGE_CLASSPATH

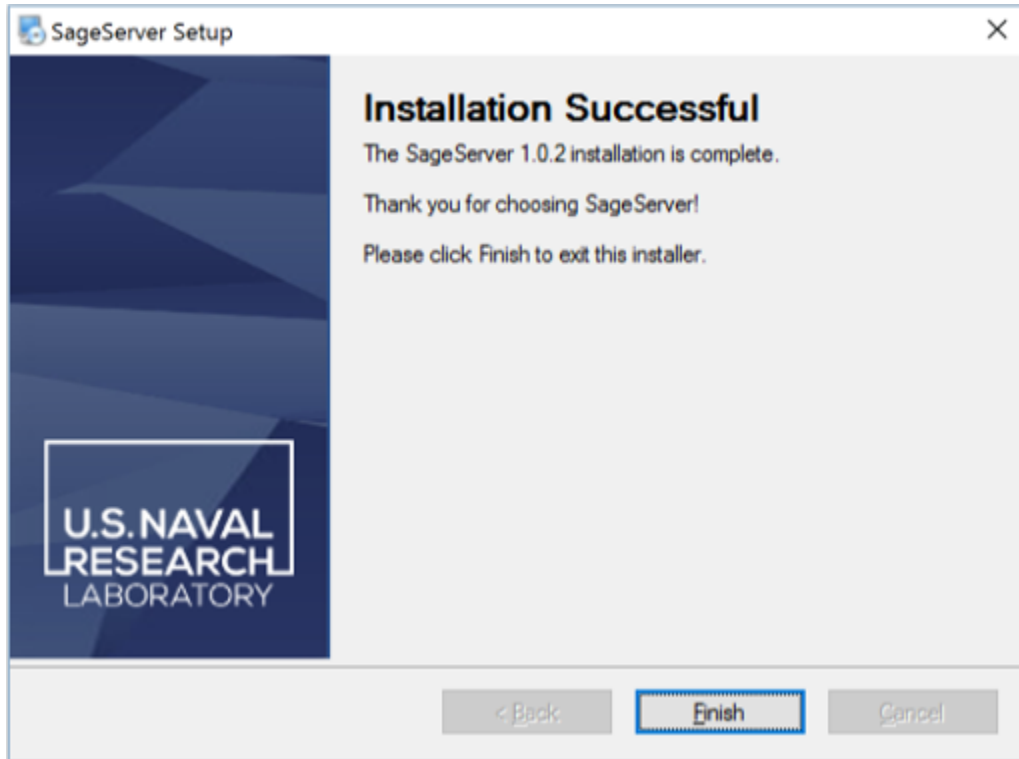
```
%SAGE_SERVER_HOME%;%SAGE_SERVER_HOME%\robotframework-2.9.2.jar;%SAGE_SERVER_HOME%\  
↪SageRemoteInterface.jar;%SAGE_SERVER_HOME%\SageJavaBehaviorInterface.jar
```

SAGE_NODE_HOME

```
C:\Program Files\NRL\SageNode
```

Additional User variables

Add the following User variables if you would like to include support for Java or Python Behaviors.



Your actual path may be set with a different name or version number. Ensure each variable points to the directory of your distribution.

JAVAHOME

```
For 64-bit: "C:\Program Files\Java\jre1.8.0_xx" or "C:\Program Files\Java\jdk1.8.0_xx"
For 32-bit: "C:\Program Files (x86)\Java\jre1.8.0_xx" or "C:\Program Files (x86)\Java\jdk1.8.0_xx"
```

PYTHONHOME

```
C:\Python27
```

Update your PATH System variable

Edit your PATH System variable to append your Users variables.

Java installation requires access to *jvm.dll* and *java.exe*.

Python requires access to *python.exe*.

PATH

```
For Java JDK users: "C:\Program Files\Java\jdk1.8.0_xx\bin;C:\Program Files\Java\jdk1.8.0_xx\jre\bin\server"
For Java JRE users: "C:\Program Files\Java\jre1.8.0_xx\bin;C:\Program Files\Java\jre1.8.0_xx\bin\server"
```

(continues on next page)

(continued from previous page)

```
For Python users: "%PYTHONHOME%"
```

7.4 Installation on Linux

On a terminal enter the following command, assuming the current sage distribution rpms are in the local folder:

```
sudo rpm -Uvh nrl-sage-node-*.rpm
```

Note that the SDK package is optional and needed only for C++ behavior development. Java and Python Behaviors can be developed without it. Note that the Linux Server is still considered to be an experimental version. To install the SDK and Server:

```
sudo rpm -Uvh nrl-sage-sdk-*.rpm
sudo rpm -Uvh nrl-sage-server-*.rpm
```

If you're on a Debian based Linux that does not have rpm, such as Ubuntu, you can install the alien package. Then the above rpm commands will work.

```
sudo apt-get install alien
```

Environment variables

SAGE Node needs the Java JVM dynamic library to be in the runtime library search path. This is accomplished by including its location in the `LD_LIBRARY_PATH` environment variable. This is typically done by including the definition in `.bashrc` file or `.profile` file.

It can also be temporarily set using the same command, typically:

`LD_LIBRARY_PATH`

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:{JRE_HOME}/lib/amd64/server
```

If you intend to use the Robot Framework on Linux with the SAGE Server you will need to define the `SAGE_ROBOT_LIBRARY` environment variable.

`SAGE_ROBOT_LIBRARY`

```
export SAGE_ROBOT_LIBRARY=SageRemoteInterface
```

Starting a Server Instance

STARTING A SERVER INSTANCE

Initialize Server

On a Windows command prompt enter the following command:

```
SageServerConsole_x64.exe {port_number}
```

On a Linux terminal enter the following command:

```
SageServerConsole {port_number}
```

To verify a running SAGE Server instance, check the prompt window and confirm that the following text is displayed, SAGE server started listening on port: {port_number}.

This can also be accomplished using the SAGE App:

- 1) In the SAGE App, configure your *Server settings* by clicking the gear icon Settings button.
- 2) Enter a port number in the Server Port dialog box and click the Ok button.
- 3) Click the green Start button to initialize the Server.

Note that there is no default port number for SAGE. It is recommended to avoid using dedicated port numbers, such as 21 (FTP), 22 (SSH), and 80 (HTTP). For a list of unassigned port numbers visit [Internet Assigned Numbers Authority \(IANA\)](#).

Connecting a Node to the Server

CONNECTING A NODE TO THE SERVER

Initialize node

On a Windows command prompt enter the following command:

```
SageNodeConsole_x64.exe nodeName {server_IP_address} {port_number} *  
↪ {scheduler_mode}
```

On a Linux terminal enter the following command:

```
SageNodeConsole nodeName {server_IP_address} {port_number} *{scheduler_mode}
```

scheduler_mode is optional and accepted mode values depend of the current CPU load:

- “high” (Default) - Scheduler runs with sub millisecond accuracy and yields the processor between runs of the scheduler. Running in high mode is not recommended for battery powered devices.
- “medium” - Scheduler runs every 1-2 milliseconds.
- “low” - Scheduler runs every 15 milliseconds.

To verify if a SAGE Node is operational, check the prompt window and confirm that the following text is displayed, Connected to SAGE runtime using ip address {server_IP_address}:{port_number}.

Building an Agent Network

BUILDING AN AGENT NETWORK

You can start to build out an agent network once you have established a connection between your SAGE Server and Node components.

Agent entities can be created within any active Node using the SAGE App, a Controller Application, or within a Behavior.

SAGE App - You can populate your existing Nodes with agents using the SAGE App. This is the manual way to configure your agent network with the proper Behaviors prior to executing your automation.

- 1) In the SAGE App, select the targeted Node name and click `Create Agent`.
- 2) Type in a name for your agent and click the OK button.

Controller Application - You can create agents using a Controller Application, such as Robot Framework. This method is a fast way to generate a large number of agents across any active Nodes.

- 1) In your robot file, connect the controller to the Server by using the keywords `Start Sage` and `Connect to Sage Runtime`.
- 2) Use the `Create Agent` keyword to populate a targeted Node name with an expected Agent name.

<code>Create Agent</code>	<code>nodeName1</code>	<code>newAgentName1</code>
<code>Create Agent</code>	<code>nodeName1</code>	<code>newAgentName2</code>
<code>Create Agent</code>	<code>nodeName2</code>	<code>newAgentName1</code>

Behavior - Agents have the ability to spawn and configure other agent entities. This is a dynamic way to build your agent network to be more robust. This unique SAGE capability is supported by initializing agents within your SAGE Behavior code. This enables agents to create other agents.

- 1) In your Behavior file, use the `CreateAgent` method to populate a targeted Node name with an expected Agent name.

<code>createAgent ("nodeName1", "newAgentName1");</code>
--

Writing a SAGE Behavior

WRITING A SAGE BEHAVIOR

SAGE Behaviors are classes that are developed, outside of the main SAGE codebase, and ran by SAGE Agents. An example Behavior might be a Python module that when invoked uses a Java package to test an external application and make the test results available to the SAGE Server.

Behaviors can be written so that agents can perform either **reactively in response to an incoming message**, or **proactively on a continuous or one time basis**.

Behavior objects must contain four methods that constitute their execution repertoire:

- `setUp` This method is called when a Behavior is instantiated and is expected to perform initialization activities. Behaviors are instantiated when an Agent becomes active, or when a new Behavior is added to an active Agent.
- **`action` This method implements the proactive execution of a Behavior based on the execution mode specified.**
 - OneShot execution activates the `action` method once when the Agent is activated.
 - Periodic execution activates the `action` method continuously.
 - TimedPeriodic execution activates the `action` method every n milliseconds.
 - OnMessage activates the `action` method upon the receipt of a specific message.
- `message` This method is called when an Agent receives a message. Behaviors can specify a topic filter consisting of a list of topics that the Behavior is interested in. Incoming messages that do not specify one of those topics in the topic filter are not propagated to the Behavior.
- `tearDown` This method is called on each Behavior when an Agent is deactivated. It is expected to perform any de-initialization activities necessary to bring the system to a known state.

Each method can populate a Result object that includes the result of the execution as well as a log of the activities performed during the execution. Result objects that specify an execution result other than `NotSet` (see below) are passed back to the Server.

Behaviors are run by (within) a SAGE Node process. When a Behavior is added to an *SAGE Agent*, its executable module is sent to the Node containing the Agent so that it can be scheduled for execution by that Node.

11.1 Inheritance from the base class

Behaviors can be written in C++, Java, or Python. In each language a SAGE Behavior is a class and is required to conform to a specific interface. This includes inheriting from a base class and using specific objects provided by the SAGE Framework. To develop C++ Behaviors you will need to install the SAGE SDK. Behaviors developed in Java or Python do not require the SAGE SDK.

Every C++ Behavior class, in this example named “ExampleCppBehavior”, is a subclass (child) of NativePluginInterface. The NativePluginInterface header file is available in the SAGE SDK.

C++

```
class ExampleCppBehavior : public NativePluginInterface
```

In Java every Behavior is a subclass of SageBehavior. The Java SageBehavior class code is in the SageJavaBehaviorInterface.jar file distributed with the SAGE Node installer. The Java source files are distributed in the SAGE SDK.

Java

```
public class ExampleJavaBehavior extends SageBehavior
```

In Python every Behavior is a subclass of SageBehavior. The Python SageBehavior code is in the file SagePythonBehaviorInterface-1.0-py2.7.egg which is distributed with the SAGE Node and SAGE Server installers. The code itself is distributed in the SAGE SDK.

Python

```
class ExamplePythonBehavior(SageBehavior.SageBehavior):
```

11.2 Result object

A Result object provides a reporting mechanism back to the Server to indicate the results of the execution of a Behavior’s setup(), action(), message(), or teardown() methods.

A Result object contains the following three members that should be assigned appropriate values by the Behavior developer:

C++

```
std::vector<std::string> m_logMessages;  
std::string m_exception;  
ExecutionResultType m_executionResult;
```

Java

```
public ArrayList<String> m_logMessages = new ArrayList<String>();  
public String m_exception = "";  
public ExecutionResultType m_executionResult = ExecutionResultType.CompletionSuccess;
```

Python

```
self.m_logMessages = []  
self.m_exception = ""  
self.m_executionResult = ExecutionResultType.CompletionSuccess
```

- **m_logMessages** is a vector of strings and is intended to provide a log of events that occurred during the execution of a Behavior.

- **m_exception** if the execution of code in the method generates an exception, the m_exception string should contain text describing the cause of the exception.
- **m_executionResult** this member consists of an enumeration of type `ExecutionResultType`. It can have one of three values:
 - `CompletionSuccess` - the method completed execution successfully
 - `CompletionFailure` - the method completed execution but failed
 - `ExceptionThrown` - code in the method caused an exception to be thrown. It is highly recommended that Behavior developers execute any code that can potentially generate exceptions within a try - catch statement.
 - `NotSet` - the method has not yet assigned m_executionResult a value.

11.3 Behavior constructor

It is a requirement for SAGE Behaviors that their **constructors only set member variables**. There should be no significant code executed in a Behavior constructor. This is because the SAGE Framework may instantiate and destroy Behaviors when doing inquiry functionality. Any startup related code should be put in the `setUp()` method.

Behaviors in all three languages have the same members declared in the base class.

There are six members:

- **m_name [a string]** This is the Behavior's name. Other parts of the SAGE system depend on this name (and the Behavior's filename) to find and execute the Behavior.
- **m_description [a string]** A description of the Behavior.
- **m_executionType** Set to `OneShot`, `Cyclical`, `TimedCyclical`, `OnMessage`, or `NoExecution`. These values all pertain to when the Behavior's `action()` method will be invoked.
 - `OneShot` `action()` is called once when the Agent is activated.
 - `Cyclical` `action()` is called continuously after the Agent is activated.
 - `TimedCyclical` `action()` is called continuously every `m_period` milliseconds after the Agent is activated.
 - `OnMessage` `action()` is called upon the receipt of a message that matches `m_triggerMessage`. Do not confuse this with the Behavior's `message()` method.
 - `NoExecution` `action()` is not called. The Behavior is still created, `setUp()` is invoked, and the Behavior can be used via the `message()` method.
- **m_triggerMessage [a string]** If the Behavior's **m_executionType** is `OnMessage` then `action()` will be invoked when a message is sent from elsewhere in the SAGE system that matches the string **m_triggerMessage**.
- **m_period [time in in milliseconds]** If the **m_executionType** is `TimedCyclical` then the `action()` method will be called every **m_period** milliseconds.
- **m_delay [time in in milliseconds]** This is the initial amount of time the Behavior's Agent will wait before calling the Behavior's `action()`. A Behavior is instantiated when its Agent is activated or when a Behavior is added to an active Agent. At that time the Behavior's `setUp()` method is also called. If the Behavior has defined an **m_executionType** of `OneShot`, `Cyclical`, or `TimedCyclical` then the `action()` will first be called after **m_delay** milliseconds have passed.

In C++, SAGE enforces this requirement by requiring that a Behavior's constructor be created using a macro **SAGE_BEHAVIOR** (defined in `NativePluginInterface.h`). Use of this macro also allows a SAGE Server running on Windows to inspect an `.so` file created under Linux.

C++

```
SAGE_BEHAVIOR(CLASS_NAME, BEHAVIOR_NAME, BEHAVIOR_DESCRIPTION, EXECUTION_TYPE,   
↳TRIGGER_MESSAGE, PERIOD, DELAY)
```

The first parameter should be the name of the class. The remaining map to each of the six members.

In Java and Python, the members should be set in the constructor.

Java

```
public ExampleJavaBehavior()  
{  
    m_name = "SimpleExample";  
    m_description = "Simple description";  
    m_executionType = ExecutionType.OneShot;  
    m_triggerMessage = "";  
    m_period = 0;  
    m_delay = 0;  
}
```

Python

```
def __init__(self):  
    super(ExamplePythonBehavior, self).__init__()  
    self.m_name = "SimpleExample"  
    self.m_description = "Simple description"  
    self.m_executionType = ExecutionType.ExecutionType.OneShot  
    self.m_triggerMessage = ""  
    self.m_period = 0  
    self.m_delay = 0
```

11.4 Behavior setUp

This method is called when a Behavior is instantiated and is expected to perform initialization activities. Behaviors are instantiated when an Agent becomes active, or when a new Behavior is added to an active Agent.

- **result [Result object]** Results of this method's execution should be reported using Result object. Result objects contain `m_logMessages`, `m_exception`, or `m_executionResult`.

It returns true or false; false indicates that there is a SAGE Behavior internal error meaning this Behavior should not be run again.

C++

```
bool ExampleCppBehavior::setUp(sageframework::Result& result)  
{  
    result.m_logMessages.push_back("This is log message string3.");  
    result.m_exception = "Nothing exceptional here.";  
    result.m_executionResult = sageframework::ExecutionResultType::NotSet;  
    return true;  
}
```

Java

```

public boolean setUp(Result result)
{
    result.m_logMessages.add("This is log message string3.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult = ExecutionResultType.NotSet;
    return true;
}

```

Python

```

def setUp(self, result):
    result.m_logMessages.append("This is log message string3.")
    result.m_exception = "Nothing exceptional here."
    result.m_executionResult = ExecutionResultType.ExecutionResultType.
↪NotSet
    return True

```

11.5 Behavior action

This method is called based on the execution type (`m_executionType`) specified in the Behavior constructor. Execution types include, `OneShot`, `Cyclical`, `TimedCyclical`, `OnMessage`, or `NoExecution`. This is method is a proactive response.

- **result [Result object]** Results of this method's execution should be reported using Result object. Result objects contain `m_logMessages`, `m_exception`, or `m_executionResult`.

It returns true or false; false indicates that there is a SAGE Behavior internal error meaning this Behavior should not be run again.

C++

```

bool ExampleCppBehavior::action(sageframework::Result& result)
{
    result.m_logMessages.push_back("This is log message string.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult =
↪sageframework::ExecutionResultType::CompletionSuccess;
    return true;
}

```

Java

```

public boolean action(Result result)
{
    result.m_logMessages.add("This is log message string.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

```

Python

```

def action(self, result):
    result.m_logMessages.append("This is log message string.")
    result.m_exception = "Nothing exceptional here."

```

(continues on next page)

(continued from previous page)

```

        result.m_executionResult = ExecutionResultType.ExecutionResultType.
↪CompletionSuccess
        return True

```

11.6 Behavior message

This method is called when an Agent receives a message. This method is a reactive response. Elsewhere in the SAGE system Behaviors are specified to have a topic filter consisting of a list of topics that the Behavior is interested in.

Incoming messages that do not specify one of those topics in the topic filter are not propagated to the Behavior.

- **Message [Message object]** Provides the necessary mechanism to retrieve the message content sent to the message recipient. Messages may contain `m_targetNodeName`, `m_targetAgentName`, `m_topic`, `m_message`, and `m_data`.
- **result [Result object]** Results of this method's execution should be reported using Result object. Result objects contain `m_logMessages`, `m_exception`, or `m_executionResult`.

It returns true or false; false indicates that there is a SAGE Behavior internal error meaning this Behavior should not be run again.

C++

```

bool ExampleCppBehavior::message(const sageframework::Message& message,
↪sageframework::Result& result)
{
    std::cout << "Got message " << message.m_message << " on topic " << message.
↪m_topic << std::endl;
    std::cout << "len of m_data is " << message.m_data.size() << std::endl;
    result.m_logMessages.push_back("This is log message from message string1.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult =
↪sageframework::ExecutionResultType::CompletionSuccess;
    return true;
}

```

Java

```

public boolean message(Message message, Result result)
{
    System.out.println("Got message " + message.m_message + " on topic " +
↪message.m_topic);
    System.out.println("length of m_data is", message.m_data.size());
    result.m_logMessages.add("This is log message from message string1.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

```

Python

```

def message(self, message, result):
    print("Got message ", message.m_message, " on topic ", message.m_topic)
    print("length of m_data is", len(message.m_data) )

```

(continues on next page)

(continued from previous page)

```

result.m_logMessages.append("This is log message from message string1.")
result.m_exception = "Nothing exceptional here."
result.m_executionResult = ExecutionResultType.ExecutionResultType.
↪CompletionSuccess
    return True

```

11.7 Behavior tearDown

This method is called when the Agent is deactivated. It should perform any clean-up and shutdown activities.

- **result [Result object]** Results of this method's execution should be reported using Result object. Result objects contain m_logMessages, m_exception, or m_executionResult.

It returns true or false; false indicates that there is a SAGE Behavior internal error meaning this Behavior should not be run again.

C++

```

bool ExampleCppBehavior::tearDown(sageframework::Result& result)
{
    result.m_logMessages.push_back("This is log message string3.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult = sageframework::ExecutionResultType::NotSet;
    return true;
}

```

Java

```

public boolean tearDown(Result result)
{
    result.m_logMessages.add("This is log message string3.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult = ExecutionResultType.NotSet;
    return true;
}

```

Python

```

def tearDown(self, result):
    result.m_logMessages.append("This is log message string3.")
    result.m_exception = "Nothing exceptional here."
    result.m_executionResult = ExecutionResultType.ExecutionResultType.
↪NotSet
    return True

```

11.8 C++ header file

C++ requires a header file. In addition to providing declarations of the required four methods, the header file must include a line required by the Boost DLL library that SAGE uses to manage dynamic load libraries.

The static create() method is a factory method that creates instances of the behavior class.

The BOOST_DLL_ALIAS statement exports that symbol so that SAGE can find it in the dynamic load library and invoke it.

The use of a factory method to load the behavior class means that each C++ dll/so can only contain one behavior class. While this is a restriction, its likely a good practice to follow in general.

Here is a example header file called ExampleCppBehavior.h:

```
#pragma once

#include "NativePluginInterface.h"

// This class will be compiled into ExampleCppBehavior.dll under Windows
// or ExampleCppBehavior.so under Linux.
//
class ExampleCppBehavior : public NativePluginInterface
{
public:
    ExampleCppBehavior();
    bool setUp(sageframework::Result& result);
    bool action(sageframework::Result& result);
    bool message(const sageframework::Message& message, sageframework::Result&
↳result);
    bool tearDown(sageframework::Result& result);

    // Factory method
    static boost::shared_ptr<ExampleCppBehavior> create()
    {
        return boost::shared_ptr<ExampleCppBehavior>(new
↳ExampleCppBehavior());
    }
};

BOOST_DLL_ALIAS
(
    ExampleCppBehavior::create,    // <-- this function is exported with...
    create_plugin                 // <-- ...this alias name
)
```

11.9 Example Behavior file

This is an example Behavior in its entirety. Other parts of the SAGE system would refer to this as the Behavior named ExampleBehavior. This Behavior does not do anything useful, but it defines each of the four required methods, will print out when each is called, and fills in the result parameter.

C++

```
// ----- Header file -----
#pragma once
```

(continues on next page)

(continued from previous page)

```

#include "NativePluginInterface.h"

// This class is exported from the ExampleCppBehavior.dll
class ExampleCppBehavior : public NativePluginInterface
{
public:
    ExampleCppBehavior();
    virtual bool setUp(sageframework::Result& result);
    virtual bool action(sageframework::Result& result);
    virtual bool message(const sageframework::Message& message,
↳sageframework::Result& result);
    bool tearDown(sageframework::Result& result);

    // Factory method
    static boost::shared_ptr<ExampleCppBehavior> create()
    {
        return boost::shared_ptr<ExampleCppBehavior>(new
↳ExampleCppBehavior());
    }
};

BOOST_DLL_ALIAS
(
    ExampleCppBehavior::create,    // <-- this function is exported with...
    create_plugin                 // <-- ...this alias name
)

// ----- cpp behavior file -----
// ExampleCppBehavior.cpp : Defines the exported functions for the DLL.
//

#include "stdafx.h"
#include "Common.h"
#include "Message.h"
#include "Result.h"
#include "ExampleCppBehavior.h"

SAGE_BEHAVIOR(ExampleCppBehavior, SimpleExample, does something cool,
↳sageframework::ExecutionType::OneShot, "", 0, 0)

bool ExampleCppBehavior::setUp(sageframework::Result& result)
{
    std::cout << "Setting up behavior" << std::endl;
    result.m_executionResult =
↳sageframework::ExecutionResultType::CompletionSuccess;
    return true;
}

bool ExampleCppBehavior::action(sageframework::Result& result)
{
    std::cout << "Hello World from Action" << std::endl;
    result.m_logMessages.push_back("This is log message string1.");
    result.m_logMessages.push_back("This is log message string2.");
    result.m_logMessages.push_back("This is log message string3.");
    result.m_exception = "Nothing exceptional here.";
    result.m_executionResult =
↳sageframework::ExecutionResultType::CompletionSuccess;

```

(continues on next page)

(continued from previous page)

```

        return true;
    }

    bool ExampleCppBehavior::message(const sageframework::Message& message,
    ↪ sageframework::Result& result)
    {
        std::cout << "Got message " << message.m_message << " on topic " << message.
    ↪ m_topic << std::endl;
        std::cout << "len of m_data is " << message.m_data.size() << std::endl;

        for (int i=0; i < message.m_data.size(); i++)
        {
            std::cout << "Data[" << i << "] = " << message.m_data[i] <<
    ↪ std::endl;
        }

        result.m_logMessages.push_back("This is log message from message string1.");
        result.m_exception = "Nothing exceptional here.";
        result.m_executionResult =
    ↪ sageframework::ExecutionResultType::CompletionSuccess;

        return true;
    }

    bool ExampleCppBehavior::tearDown(sageframework::Result& result)
    {
        std::cout << "Tearing down behavior " << m_name << std::endl;
        result.m_executionResult =
    ↪ sageframework::ExecutionResultType::CompletionSuccess;
        return true;
    }

```

Java

```

/* Example SAGE Behavior in Java.
   Shows the four methods that have to be defined:
   setUp, action, message, tearDown

   Use the following command line to build:
       javac -cp "%SAGE_SERVER_HOME%\SageJavaBehaviorInterface.jar
    ↪ ExampleJavaBehavior.java

   Create jar file
       jar cf ExampleJavaBehavior.jar ExampleJavaBehavior.class
*/

import nrl.sage.BehaviorInterface.*;

public class ExampleJavaBehavior extends SageBehavior
{
    public ExampleJavaBehavior()
    {
        m_name = "SimpleExample";
        m_description = "Simple behavior that prints a lot";
    }

```

(continues on next page)

(continued from previous page)

```

        m_executionType = ExecutionType.OneShot;
        m_delay = 15000;
    }

    public boolean setUp(Result result)
    {
        /* The class needs a setUp. Return True or False */
        System.console().printf("Setting up behavior %s\n", m_name);
        result.m_executionResult = ExecutionResultType.CompletionSuccess;

        return true;
    }

    public boolean action(Result result)
    {
        /* The class needs an action fill in result. Return True or False */
        System.out.println("Hello World from Action");
        result.m_logMessages.add("This is log message string1.");
        result.m_logMessages.add("This is log message string2.");
        result.m_logMessages.add("This is log message string3.");
        result.m_exception = "Nothing exceptional here.";
        result.m_executionResult = ExecutionResultType.CompletionSuccess;

        return true;
    }

    public boolean message(Message message, Result result)
    {
        /* The class needs a message message method. It is passed in a
        ↳message object. Return True or False */
        System.out.println("Got message " + message.m_message + " on topic " +
        ↳+ message.m_topic);
        System.out.println("length of m_data is", message.m_data.size());
        for (int i=0; i<message.m_data.size(); i++)
        {
            System.out.println("Data[" + Integer.toString(i) + "]");
            ↳+ " + message.m_data.get(i));
        }
        result.m_logMessages.add("This is log message from message string1.");
        result.m_exception = "Nothing exceptional here.";
        result.m_executionResult = ExecutionResultType.CompletionSuccess;

        return true;
    }

    public boolean tearDown(Result result)
    {
        /* The class needs a tearDown. Return True or False */
        System.console().printf("Tearing down behavior %s", m_name);
        result.m_executionResult = ExecutionResultType.CompletionSuccess;

        return true;
    }
}

```

Python

```
#!/usr/bin/env python

# Example SAGE Behavior in Python.
# Shows the four methods that have to be defined:
# setUp, action, message, tearDown
#

from nrl.sage.BehaviorInterface import *

class ExamplePythonBehavior(SageBehavior.SageBehavior):
    """ This is a simple SAGE Behavior """
    def __init__(self):
        super(ExamplePythonBehavior, self).__init__()
        self.m_name = "SimpleExample"
        self.m_description = "Simple behavior that prints a lot."
        self.m_executionType = ExecutionType.ExecutionType.OneShot
        self.m_delay = 15000

    def setUp(self, result):
        """ The class needs a setUp. Return True or False """
        print("Setting up behavior ", self.m_name)
        result.m_executionResult = ExecutionResultType.ExecutionResultType.
↳CompletionSuccess

        return True

    def action(self, result):
        """ The class needs an action fill in result. Return True or
↳False """
        print("Hello World from Action")
        result.m_logMessages.append("This is log message string1.")
        result.m_logMessages.append("This is log message string2.")
        result.m_logMessages.append("This is log message string3.")
        result.m_exception = "Nothing exceptional here."
        result.m_executionResult = ExecutionResultType.ExecutionResultType.
↳CompletionSuccess

        return True

    def message(self, message, result):
        """ The class needs a message method. It is passed in a message
↳object. Return True or False """
        print("Got message ", message.m_message, " on topic ", message.m_
↳topic)

        print("length of m_data is", len(message.m_data) )
        i = 0
        while i < len(message.m_data):
            print("Data[", i, "] = ", message.m_data[i] )
            i += 1
        result.m_logMessages.append("This is log message from message string1.
↳")

        result.m_exception = "Nothing exceptional here."
        result.m_executionResult = ExecutionResultType.ExecutionResultType.
↳CompletionSuccess

        return True
```

(continues on next page)

(continued from previous page)

```
def tearDown(self, result):  
    """ The class needs a tearDown. Return True or False """  
    print("Tearing down behavior ", self.m_name)  
    result.m_executionResult = ExecutionResultType.ExecutionResultType.  
↪CompletionSuccess  
  
    return True
```

11.10 Packaged Behaviors

SAGE Behaviors **MUST** be packaged into .JAR or .EGG files for Java and Python behaviors respectively. C++ behaviors are packaged as .DLL or .SO modules depending on the target platform.

Note that there can only be one SAGE Behavior per package.

Base Class - SAGE Behavior Methods

BASE CLASS - SAGE BEHAVIOR METHODS

The SAGE Behavior base class provides a number of powerful methods that allow Agents to communicate with other Agents, create and remove Agents, add and remove Behaviors from Agents, activate and deactivate Agents, and manage their Agent's state space.

This capability in SAGE supports the idea of meta-agents that can reason about their current environment and create other Agents based on that reasoning.

12.1 addBehavior

This method enables SAGE Agents to populate other SAGE Agents with new Behaviors.

- **nodeName [string]** This is the name of a SAGE Node where the target Agent exists.
- **agentName [string]** This is the name of the SAGE Agent that will receive the new Behavior.
- **behaviorName [string]** This is the name that was assigned to the SAGE Behavior by its developer.
- **module [string]** This is the file name of the module containing the Behavior to be added. Depending on the language the Behavior was written in, this could be a Java .jar file, a Python .egg file, or a native dynamic load library (.dll or .so).
- **topics [string vector]** The topic vector specifies the set of topic names that the new Behavior will be interested in. Messages sent to the Behavior's Agent will only be routed to this new Behavior if the topic specified in the message matches one of the topic names in this vector. Note that messages that do not specify a topic are sent to all Behaviors and conversely Behaviors that don't specify a topic list receive all message irrespective of the topic the message specifies.

C++

```
void addBehavior(std::string nodeName, std::string agentName, std::string_  
↳behaviorName, std::string module, std::vector<std::string> topics);
```

Java

```
public void addBehavior(String nodeName, String agentName, String behaviorName,_  
↳String module, ArrayList<String> topics)
```

Python

```
def addBehavior(self, nodeName, agentName, behaviorName, module_, topics)
```

12.2 createAgent

This method enables SAGE Agents to create other SAGE Agents through their Behaviors. New Agents can be created on an Agent's local Node or a remote Node.

- **nodeName** [string] This is the name of a SAGE Node where the new Agent will be created.
- **agentName** [string] This will be the name assigned to the new SAGE Agent. Agent names must be unique within the Node that contains them.

C++

```
void createAgent(std::string nodeName, std::string agentName);
```

Java

```
public void createAgent(String nodeName, String agentName)
```

Python

```
def createAgent(self, nodeName, agentName)
```

12.3 createState

This method enables Behaviors to create a new, named state in its Agent's state space. State names must be unique so this method returns false if a state with the specified name already exists.

- **name** [string] This is the name of the new state to be added to state space.

C++

```
bool createState(std::string name);
```

Java

```
public Boolean createState(String name)
```

Python

```
def createState(self, name)
```

12.4 getState

This method returns the value of a named state in an Agent's state space. The method throws an exception if the specified state name is not found. The state value is returned as a generic value.

- **name** [string] This is the name of the state whose value will be returned.

In C++, the value is returned as a `boost::variant` that can contain long, double, or `std::string` types. The `boost::variant` `which()` method can be used to determine the type held. The `which()` method returns the 0-based index of the type currently held by the `boost::variant`. In SAGE, a value of 0 is a long, a value of 1 is a double, and a value of 2 is a `std::string`.

To retrieve the actual value of the `boost::variant`, use the `boost::get<T>(value)` function template. For a long value use `boost::get<long>(value)`, for a double value use `boost::get<double>(value)`, and for a `std::string` value use `boost::get<std::string>(value)`.

C++

```
sageframework::StateValueType getState(std::string name);
```

In Java the value is returned as a generic `Object`. Java's `getClass()` method can be called on the `Object` to discover its type. The object can then be cast to the appropriate type to retrieve its value.

Java

```
public Object getState(String name) throws Exception
```

In Python, the `type(value)` function returns the type of a variable. Python variables are dynamically typed so the value returned from the `getState` can be assigned to Python variable.

Python

```
def getState(self, name)
```

12.5 getStateNames

This method returns a vector of all the state names that currently exist in an Agent's state space.

C++

```
std::vector<std::string> getStateNames();
```

Java

```
public Set getStateNames()
```

Python

```
def getStateNames(self)
```

12.6 removeAgent

This method enables SAGE Agents to remove other SAGE Agents through their Behaviors. Agents can be removed from an Agent's local Node or a remote Node.

- **nodeName [string]** This is the name of a SAGE Node where the target Agent exists.
- **agentName [string]** This is the name of the SAGE Agent that is removed.

C++

```
void removeAgent(std::string nodeName, std::string agentName);
```

Java

```
public void removeAgent(String nodeName, String agentName)
```

Python

```
def removeAgent(self, nodeName, agentName)
```

12.7 removeBehavior

This method enables SAGE Agents to remove existing Behaviors from other Agents.

- **nodeName** [string] This is the name of a SAGE Node where the target Agent exists.
- **agentName** [string] This is the name of the SAGE Agent that will have its Behavior removed.
- **behaviorName** [string] This is the name of the Behavior that will be removed.

C++

```
void removeBehavior(std::string nodeName, std::string agentName, std::string_  
↪behaviorName);
```

Java

```
public void removeBehavior(String nodeName, String agentName, String behaviorName)
```

Python

```
def removeBehavior(self, nodeName, agentName, behaviorName)
```

12.8 removeState

This method enables Behaviors to remove a named state from its Agent's state space. If the specified state name is not found, this method returns false.

- **name** [string] This is the name of the state to be removed from state space.

C++

```
bool removeState(std::string name);
```

Java

```
public Boolean removeState(String name)
```

Python

```
def removeState(self, name)
```


12.9 setAgentActive

This method enables SAGE Agents to activate and deactivate other SAGE Agents. Agents are created in an inactive state. Activating an inactive agents causes the `setUp()` methods of its Behaviors to be called and its `action()` and `message()` methods to be invoked based on the properties of the Agent and in response to incoming messages. Deactivating an active Agent causes the `tearDown()` method of its Behaviors to be called and the Agent then becomes dormant.

- **nodeName [string]** This is the name of a SAGE Node where the target Agent exists.
- **agentName [string]** This is the name of the SAGE Agent that will be activated or deactivated.
- **isActive [boolean]** This is a flag indicating if an Agent is to be activated (true) or deactivated (false).

C++

```
void setAgentActive(std::string nodeName, std::string agentName, bool isActive);
```

Java

```
public void setAgentActive(String nodeName, String agentName, boolean isActive)
```

Python

```
def setAgentActive(self, nodeName, agentName, isActive)
```

12.10 sendFile

This method transmits a file to the SAGE Server. This capability is useful for sending auxiliary data or arbitrary files to a centralized location. Files transmitted to a SAGE Server running on the Windows operating system are placed in the `sage\downloads` subfolder of the `ProgramData` folder (usually `C:\ProgramData\SAGE\downloads`).

- **filePath [string]** This is a fully qualified path to the file.

C++

```
void sendFile(std::string filePath);
```

Java

```
public void sendFile(String filePath);
```

Python

```
def sendFile(self, filePath)
```

12.11 sendMessage

The sendMessage method can be called by Behaviors to communicate with other Agents in the SAGE Agent network.

C++

```
void sendMessage(const sageframework::Message& message)
```

Java

```
public void sendMessage(Message message)
```

Python

```
def sendMessage(self, message)
```

12.12 setState

This method sets the value of named state in an Agent's state space. The method returns false if the state name is not found. In C++ setState is a templated function that is instantiated for values of type long, double, and std::string. It can be called with values that are one of those three types.

- **name [string]** This is the name of the state whose value will be modified.
- **value [long, double, or std::string]** This is the new value that the state will take on. Valid values must of type long, double, or std::string.

C++

```
template<typename T> bool setState(std::string, T value);
```

Java

```
public Boolean setState(String name, Object value)
```

Python

```
def setState(self, name, value)
```

Retrieving Information About Your Agent Network

Agent Introspection and State

RETRIEVING INFORMATION ABOUT YOUR AGENT NETWORK

SAGE Agents will often need to find the names of SAGE Nodes that are part of the network as well as the names of SAGE Agents residing at a specific node. SAGE uses a query message send and reply protocol to provide this introspection capability.

SAGE Agents can send query messages to the SAGE Server by specifying `sage` as the target Node in the Message object (`m_targetNodeName`). The type of query requested is specified in the message itself (`m_message`).

Replies are sent by the SAGE Server to the originating SAGE Agent and contain the original query in the `m_message`, while the results of the query are returned data element, `m_data`, of the Message object. Simple queries that request an enumeration of Node and Agent names receive a reply with the Node or Agent names as a string vector in `m_data`. Deeper queries that request all the relevant information about Nodes or Agents receive a reply with a JSON object in the first element of the `m_data` vector (`m_data[0]`).

The following introspection queries are supported by SAGE:

Querying the names of all the SAGE Nodes in the network

To get the names of all the SAGE Nodes in the network, the message (`m_message`) should equal `getNodeNames`.

Here's an example of a `getNodeNames` query generated in a Java behavior:

```
Message myMessage = new Message();

myMessage.m_message = "getNodeNames";
myMessage.m_targetNodeName = "sage";
sendMessage(myMessage);
```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```
m_message = "getNodeNames"
m_topic = "sage"
m_data[0] = "nodeName1"
.
.
m_data[n - 1] = "nodeNameN"
```

Querying the names of all the SAGE Agents in the network

To get the names of SAGE Agents at a specific SAGE Node, message (`m_message`) should equal `getAgentNames` with the name of the SAGE Node included as the first string data element in the Message object (`m_data[0]`).

Here's an example of a `getAgentNames` query generated in a Java behavior:

```
Message myMessage = new Message();

myMessage.m_message = "getAgentNames";
```

(continues on next page)

(continued from previous page)

```
myMessage.m_targetNodeName = "sage";
myMessage.m_data.add("nodeName1");
sendMessage(myMessage);
```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```
m_message = "getAgentNames"
m_topic = "sage"
m_data[0] = "newAgentName1"
.
.
m_data[n - 1] = "newAgentNameN"
```

Querying all of the SAGE Nodes in the network

To get detailed information on all of the SAGE Nodes in a network, message (m_message) should equal getNodes.

Here's an example of a getNodes query generated in a Java behavior:

```
Message myMessage = new Message();

myMessage.m_message = "getNodes";
myMessage.m_targetNodeName = "sage";
sendMessage(myMessage);
```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```
m_message = "getNodes"
m_topic = "sage"
m_data[0]=
{
  "nodes": {
    "Node1": {
      "agents": {
        "Agent1": {
          "behaviors": {
            "ExampleJavaBehavior": {
              "module":
↪ "ExampleJavaBehavior.jar",
              "targetEnvironment": "Any"
            }
          },
          "Agent2": {
            "behaviors": {
              "Hello": {
                "module": "hello.jar",
                "targetEnvironment": "Any"
              }
            }
          },
          "runtimeEnvironment": "Windows64",
          "ipAddress": "127.0.0.1"
        },
        "Node2": {
          "agents": {
```

(continues on next page)

(continued from previous page)

```

        "Agent3": {
            "behaviors": {
                "TestGetNode": {
                    "module": "TestGetNode.jar",
                    "targetEnvironment": "Any"
                }
            }
        },
        "Agent4": {
            "behaviors": {
                "ExampleJavaBehavior": {
                    "module":
↪ "ExampleJavaBehavior.jar",
                    "targetEnvironment": "Any"
                }
            }
        }
    },
    "runtimeEnvironment": "Windows64",
    "ipAddress": "127.0.0.1"
}
}

```

Querying a specific SAGE Node in the network

To get detailed information of a single SAGE Node, message (m_message) should equal `getNode` and the first element of m_data (m_data[0]) should contain the target Node's name.

Here's an example of a `getNode` query generated in a Java behavior:

```

Message myMessage = new Message();

myMessage.m_message = "getNode";
myMessage.m_targetNodeName = "sage";
message.m_data.add("Node1");
sendMessage(myMessage);

```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```

m_message = "getNode"
m_topic = "sage"
m_data[0] =
{
    "Node1": {
        "agents": {
            "Agent1": {
                "behaviors": {
                    "ExampleJavaBehavior": {
                        "module": "ExampleJavaBehavior.jar",
                        "targetEnvironment": "Any"
                    }
                }
            },
            "Agent2": {
                "behaviors": {
                    "Hello": {

```

(continues on next page)

(continued from previous page)

```

        "module": "hello.jar",
        "targetEnvironment": "Any"
    }
}
},
"runtimeEnvironment": "Windows64",
"ipAddress": "127.0.0.1"
}
}

```

Querying all of the SAGE Agents in the network

To get detailed information on all of the SAGE Agents in a network, message (m_message) should equal `getAgents` and the first element of m_data (m_data[0]) should contain the target Node's name.

Here's an example of a `getAgents` query generated in a Java behavior:

```

Message myMessage = new Message();

myMessage.m_message = "getAgents";
myMessage.m_targetNodeName = "sage";
message.m_data.add("Node1");
sendMessage(myMessage);

```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```

m_message = "getAgents"
m_topic = "sage"
data[0] =
{
  "agents": {
    "Agent1": {
      "behaviors": {
        "ExampleJavaBehavior": {
          "module": "ExampleJavaBehavior.jar",
          "targetEnvironment": "Any"
        }
      }
    },
    "Agent2": {
      "behaviors": {
        "Hello": {
          "module": "hello.jar",
          "targetEnvironment": "Any"
        }
      }
    }
  }
}

```

Querying a specific SAGE Agent in the network

To get detailed information on a specific SAGE Agent in a network, message (m_message) should equal `getAgent` and the first element of m_data (m_data[0]) should contain the target Node's name. The second element of m_data (m_data[1]) should contain the Agent's name.

Here's an example of a `getAgent` query generated in a Java behavior:

```

Message myMessage = new Message();

myMessage.m_message = "getAgent";
myMessage.m_targetNodeName = "sage";
message.m_data.add("Node1");
message.m_data.add("Agent1");
sendMessage(myMessage);

```

In response to this query, the SAGE Server would reply to the sender with a Message containing the following:

```

m_message = "getAgent"
m_topic = "sage"
data[0] =
{
  "Agent1": {
    "behaviors": {
      "ExampleJavaBehavior": {
        "module": "ExampleJavaBehavior.jar",
        "targetEnvironment": "Any"
      }
    }
  }
}

```

Getting the name of the originating Agent and its Node

There may be instances where you need to know information about the Agent who is currently using your Behavior, such as what Node it resides on. For instance, you may want to ensure that you create a new agent on the same node as the originating Agent.

To achieve this, agents contain a state space that is shared by all the agents' behaviors. State space consists of name-value pairs where name is a unique string identifying a state variable and value is one of the following value types: double-precision floating-point number, a long integer, or a string.

Behaviors can create, remove, or modify the value of state variables. SAGE guarantees that state space is always synchronized across behavior executions.

SAGE provides two standard state values of type string that are available to behaviors:

- `node` is the name of the SAGE Node where the behavior's agent resides
- `agent` is the name of that agent.

Use either the `getState` or `getStateNames` methods to return the Agent's state space information

```

System.out.println(getState("node"));
System.out.println(getState("agent"));
System.out.println(getStateNames());

```

If all goes well, you should see the following response

```

nodeName1
newAgentName1
[nodeName1, newAgentName1]

```

Sending Messages Between Agents

SENDING MESSAGES BETWEEN AGENTS

SAGE Agents have the built-in ability to communicate across the network.

This can be useful for:

- Controlling other agents dynamically.
- Sending pertinent information, such as files, across your agent network.
- Gathering information regarding your agent network.

The Message class provides the necessary mechanism to specify a message recipient and the message content.

- **m_targetNodeName [string]** This is the name of the SAGE Node that is the recipient of this message. If left empty, the message is sent to all the Nodes in the SAGE network.
- **m_targetAgentName [string]** This is the name of the SAGE Agent that is the recipient of this message. If left empty, the message is sent to all the Agents in the specified SAGE Node.
- **m_topic [string]** If specified, a topic name acts to direct a message to only those Behaviors that have that topic included in their topics list. This provides a filtering mechanism so that messages are only sent to those Behaviors interested in them.
- **m_message [string]** This is an arbitrary, application defined string that specifies what the message is.
- **m_data [string vector]** This string vector allows the message sender to include an arbitrary number of data items with the message. The content of that data is application defined.

C++

```
class Message
{
public:
    std::string m_targetNodeName;
    std::string m_targetAgentName;
    std::string m_topic;
    std::string m_message;
    std::vector<std::string> m_data;
}
```

Java

```
public class Message
{
    public String m_targetNodeName;
    public String m_targetAgentName;
    public String m_topic;
    public String m_message;
```

(continues on next page)

(continued from previous page)

```
    public ArrayList<String> m_data = new ArrayList<String>();  
}
```

Python

```
class Message(object):  
def __init__(self):  
    self.m_targetNodeName = str()  
    self.m_targetAgentName = str()  
    self.m_topic = str()  
    self.m_message = str()  
    self.m_data = []
```

Invoke the `sendMessage` method using your `Message` as the argument to communicate with other Agents in the SAGE Agent network.

```
Message myMessage = new Message();  
  
myMessage.m_message = "Hello";  
myMessage.m_targetNodeName = "nodeName1";  
myMessage.m_targetAgentName = "newAgentName1";  
sendMessage(myMessage);
```

As a result, the recipient agent (`newAgentName1`) will receive a message containing the message content “Hello”.

Responding to a Message Sent to an Agent

Reactive Response vs. Proactive Response

RESPONDING TO A MESSAGE SENT TO AN AGENT

Depending on the Behavior construction, an agent will perform either **reactively** in response to an incoming message or **proactively** upon the receipt of a specific trigger message (`m_triggerMessage`) or at activation. You can support both types of execution within a single Behavior file.

Reactive Response

Reactive responses are handled within your Behavior `message` method. Any code found within the `message` method will be called once your targeted Agent receives any message.

To avoid invoking the wrong Behavior file, you can specify a specific topic (`m_topic`) within the senders Message object which matches the list of topics that the Behavior is interested in. Topics are associated with Behaviors when you add them to an agent.

Here's an example of a Message being sent by `senderAgent`:

```
Message myMessage = new Message();

myMessage.m_message = "Hello";
myMessage.m_targetNodeName = "recipientNode";
myMessage.m_targetAgentName = "recipientAgent";
myMessage.m_topic = "Topic1";
sendMessage(myMessage);
```

Assuming `recipientAgent` has a Behavior that is associated with the topic "Topic1":

```
addBehavior("recipientNode", "recipientAgent", "ExampleBehavior", "ExampleBehavior.jar", "Topic1");
```

And by setting my execution type (`m_executionType`) to `NoExecution` within my Behavior's constructor, I ensure the `action()` method won't be called:

```
public ExampleBehavior()
{
    m_executionType = ExecutionType.NoExecution;
    m_description = "Example Behavior";
    m_name = "ExampleBehavior";
}

public boolean setUp(Result result)
{
    System.out.println("begin");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}
```

(continues on next page)

(continued from previous page)

```

public boolean action(Result result)
{
    System.out.println("Hello World from Action");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean message(Message message, Result result)
{
    System.out.println("Got message " + message.m_message + " on topic " + message.
↪m_topic);
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean tearDown(Result result)
{
    System.out.println("end");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

```

If all goes well, you should see the following response from the recipient (recipientAgent).

```

begin
Got message Hello on topic Topic1

```

Here's the same example of a Message being sent by the Server instead:

- 1) In the SAGE App select Send Message to recipientAgent on recipientNode.
- 2) Type message "Hello".
- 3) Type topic "Topic1".
- 4) Click the Ok button.

If all goes well, you should see the same response as above from the recipient (recipientAgent).

```

begin
Got message Hello on topic Topic1

```

Proactive Response

Proactive responses are handled within your Behavior `action` method. Any code found within the `action` method will be called once your targeted Agent receives the trigger message.

Here's an example of a Message being sent by senderAgent:

```

Message myMessage = new Message();

myMessage.m_message = "Bye";
myMessage.m_targetNodeName = "recipientNode";
myMessage.m_targetAgentName = "recipientAgent";
sendMessage(myMessage);

```

By setting my execution type (`m_executionType`) to `OnMessage` and specifying a trigger message within my Behavior's constructor, I can invoke the `action()` method:

```

public ExampleBehavior()
{
    m_executionType = ExecutionType.OnMessage;
    m_description = "Example Behavior";
    m_name = "ExampleBehavior";
    m_triggerMessage = "Bye";
}

public boolean setUp(Result result)
{
    System.out.println("begin");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean action(Result result)
{
    System.out.println("Hello World from Action");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean message(Message message, Result result)
{
    System.out.println("Got message " + message.m_message + " on topic " + message.
↪m_topic);
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean tearDown(Result result)
{
    System.out.println("end");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

```

If all goes well, you should see the following response from the recipient (recipientAgent).

```

begin
Hello World from Action

```

Here's the same example of a Message being sent by the Server instead:

- 1) In the SAGE App select Send Message to recipientAgent on recipientNode.
- 2) Type message "Bye".
- 3) Click the Ok button.

If all goes well, you should see the same response as above from the recipient (recipientAgent).

```

begin
Hello World from Action

```

You can also make use of OneShot, Cyclical, or TimedCyclical proactive execution types (m_executionType), which respond as soon as the Agent has been activated, to get the best of both worlds:

```
public ExampleBehavior()
{
    m_executionType = ExecutionType.OneShot;
    m_description = "Example Behavior";
    m_name = "ExampleBehavior";
}

public boolean setUp(Result result)
{
    System.out.println("begin");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean action(Result result)
{
    System.out.println("Hello World from Action");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean message(Message message, Result result)
{
    System.out.println("Got message "+ message.m_message);
    setAgentActive("recipientNode","recipientAgent", false);
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean tearDown(Result result)
{
    System.out.println("end");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}
```

Assuming the recipientAgent got a message AFTER the agent was activated, you should see the following response from the recipient (recipientAgent).

```
begin
Hello World from Action
Got message Bye
end
```

In this example, tearDown() method was invoked last as the agent was deactivated using the setAgentActive(false) method.

Managing Behavior Files

MANAGING BEHAVIOR FILES

The Server hosts and manages a Behavior repository located in the following directory:

	Behavior Path
Windows	C:\ProgramData\Sage\behaviors
Linux	/etc/Sage/behaviors

Behavior files must be placed on the Server machine in order for the Server to distribute them across the network. This is because the SAGE runtime looks inside these directory paths before deploying an executable module to a target Node.

Manually updating files

The behaviors directory contains sub-directories for SAGE supported languages C++, Java, and Python.

Windows64, Windows32, Linux64, Linux32 C++ behavior files directory - Behavior.cpp

Python Python behavior files directory - Behavior.egg

Java Java behavior files directory - Behavior.jar

You can manually move your Behaviors to their corresponding sub-directory.

Updating files using SAGE App

It is recommended to use the Behavior Manager found within the SAGE App to manage your Behaviors. It provides a syntax check to ensure that the file you are attempting to upload is a properly written and supported SAGE Behavior.

- 1) In the SAGE App, click the Behavior Manager button.
- 2) Select the Manage drop-down menu and click Add Modules.
- 3) Select the targeted file and click Ok.

Upon successful upload, your file will be copied to the proper sub-directory.

Adding a Behavior to an Agent

ADDING A BEHAVIOR TO AN AGENT

The Server will distribute behavior files to the target Nodes by way of a file transfer. From there, a targeted Agent will be given access to the Behavior.

A Behavior contains some action expected to be fulfilled by an Agent residing on a Node instance.

Behaviors can be sent to connected Agents in a number of ways.

Using the Behavior Manager in SAGE App - You can populate your existing Agents with Behaviors using the SAGE App. This is the manual way to configure your agent network with the proper Behaviors prior to executing your automation.

- 1) In the SAGE App, select the targeted Agent name and click Add Behavior.
- 2) Select the Behavior name from the list of available Behaviors and click Ok.
- 3) Enter the desired Behavior topic strings and click Ok.

Assuming the Behavior file is in the proper sub-directory on the Server:

By a Controller Application - You can add available behaviors to agents using a Controller Application, such as Robot Framework. This gives SAGE users the ability to control complex test scenarios, involving multiple nodes and agents, from a single controller entity.

- 1) In your robot file, use the Add Behavior keyword to populate a targeted Node and Agent name with the specified behavior.

```
Add Behavior    nodeName1    newAgentName1    ExampleBehavior    ↵  
↵ExampleBheavior.jar    Topic1    Topic2
```

Through the execution of a Behavior - Agents have the ability configure other agent entities. This is a dynamic way to build your agent network to be more robust. This unique SAGE capability enables agents to send behaviors to other agents.

- 1) In your Behavior file, use the addBehavior method to populate a targeted Node and Agent with the specified behavior.

```
addBehavior("nodeName1", "newAgentName1", "ExampleBehavior",  
↵"ExampleBheavior.jar", "Topic1", "Topic2");
```

Using Supplemental Files in Your Automation

USING SUPPLEMENTAL FILES IN YOUR AUTOMATION

There are many cases in which your automation test may require files other than Behaviors, such as images, executables, or text documents.

These files can be used to support your automation scripts at the **Node** level.

Supplemental files should be manually placed in the following directory on the targeted **Node machine**:

	Data Path
Windows	C:\ProgramData\Sage\data
Linux	/etc/Sage/data

Activating Agents

ACTIVATING AGENTS

Remember that depending on how your Agents' Behaviors are programmed, Agents may execute code in response to being activated.

Agent entities can be activated in multiple ways.

Using the SAGE App - You can activate your existing Agents using the SAGE App. This method is a fast way to activate a large number of agents simultaneously on a single Node.

- 1) In the SAGE App, select the targeted Node name or single Agent name and click `Activate Agent`.

By a Controller Application - You can activate agents using a Controller Application, such as Robot Framework. This method is a fast way to activate a large number of single Agents across multiple Nodes.

- 1) Use the `Activate Agent` keyword to activate a targeted Agent name residing on the targeted Node.

<code>Activate Agent</code>	<code>nodeName1</code>	<code>newAgentName1</code>
<code>Activate Agent</code>	<code>nodeName1</code>	<code>newAgentName2</code>
<code>Activate Agent</code>	<code>nodeName2</code>	<code>newAgentName1</code>

Through the execution of a Behavior - Agents have the ability to activate other agent entities. This is a dynamic way to build your agent network to be more robust. This unique SAGE capability is supported by activating agents within your SAGE Behavior code. This enables agents to activate other agents.

- 1) In your Behavior file, use the `setAgentActive` method to activate a targeted Agent name residing on the targeted Node.
- 2) Set the value to `true` to activate or `false` to deactivate.

<pre>setAgentActive(nodeName1, newAgentName2, true); setAgentActive(nodeName1, newAgentName2, false);</pre>

Creating A Test Case Using Robot Framework - Controller Application

CREATING A TEST CASE USING ROBOT FRAMEWORK - CONTROLLER APPLICATION

A Controller in the SAGE Framework is an external application that interacts with the SAGE Server.

This component is optional. However, it is highly recommended for users using SAGE for test automation to use a Controller to interact with your tests. It provides greater visibility by enabling report generation and step-by-step test case building.

By default, SAGE Framework supports native Robot Framework integration as a SAGE controller mechanism.

By remotely connecting Robot Framework to an running SAGE Server instance, you can manage a network of Agents in a single Robot file using Keyword-based testing.

For users in test automation, controlling your agent network via Robot Framework provides the following benefits:

- You can construct an entire test case in a single file.
- Robot Framework prints a test report with each step producing a result.
- You can quickly create several independent agent networks for multiple test cases.
- All tests are stored and ran from a single server machine.
- You can run multiple Robot files sequentially.

If either of these benefits are not fitting for your automation needs, then you may not require the use of a controller mechanism. Instead, continue using the Server and Node only.

Keywords, such as `Start SAGE`, `Connect To SAGE Runtime`, `Create Agent`, `Add Behavior`, and `Activate Agent` are used to construct an agent network.

```
**Test Cases**
This is an Example Test
    Start SAGE
    Connect To SAGE Runtime    {server_IP_address}    {port_number}
    Create Agent    nodeName1    agentName1
    Add Behavior    nodeName1    agentName1    {behaviorName}    {behaviorModule}
    <--topic
    Activate Agent    nodeName1    agentName1
```

While keywords, such as `Remove Agent` and `Deactivate Agent`, can be used to de-construct your network.

```
**Test Cases**
This is an Example Test
    Start SAGE
    Connect To SAGE Runtime    {server_IP_address}    {port_number}
    Create Agent    nodeName1    agentName1
    Add Behavior    nodeName1    agentName1    {behaviorName}    {behaviorModule}
    <--topic
```

(continues on next page)

(continued from previous page)

```

Activate Agent    nodeName1    agentName1
Deactivate Agent  nodeName1    agentName1
Remove Agent     nodeName1    agentName1

```

Using keywords, such as Send Message and Run Step allows the Server to send a message to a targeted Agent.

```

**Test Cases**
This is an Example Test
    Start SAGE
    Connect To SAGE Runtime {server_IP_address} {port_number}
    Create Agent    nodeName1    agentName1
    Add Behavior    nodeName1    agentName1    {behaviorName}    {behaviorModule}
↪topic
    Activate Agent    nodeName1    agentName1
    Send Message    nodeName1    agentName1    Topic1    Hello
    Deactivate Agent    nodeName1    agentName1
    Remove Agent    nodeName1    agentName1

```

Depending on the Behavior construction, an agent will perform either **reactively** in response to an incoming message or **proactively** upon the receipt of a specific trigger message or at activation.

Here's an example constructed test case:

```

This is an Example Test
    Start SAGE
    Connect To SAGE Runtime    121.0.0.1    50001
    Create Agent    recipientNode    recipientAgent
    Add Behavior    recipientNode    recipientAgent    ExampleBehavior
↪ExampleBehavior.jar    Topic1
    Activate Agent    recipientNode    recipientAgent
    Send Message    recipientNode    recipientAgent    Topic1    Hello
    Deactivate Agent    recipientNode    recipientAgent
    Remove Agent    recipientNode    recipientAgent

```

Looking at the Java code for ExampleBehavior file:

```

public ExampleBehavior()
{
    m_executionType = ExecutionType.NoExecution;
    m_description = "Example Behavior";
    m_name = "ExampleBehavior";
}

public boolean setUp(Result result)
{
    System.out.println("begin");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean action(Result result)
{
    System.out.println("Hello World from Action");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean message(Message message, Result result)

```

(continues on next page)

(continued from previous page)

```

{
    System.out.println("Got message " + message.m_message + " on topic " + message.
↪m_topic);
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}
public boolean tearDown(Result result)
{
    System.out.println("end");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

```

If all goes well, you should see the following results from your executed test case:

```

begin
Got message Hello on topic Topic1
end

```

This is because, the Send Message Keyword points to the Behavior associated with Topic1 (ExampleBehavior). Knowing this, the Server sends message “Hello” to the recipient Agent (recipientAgent) using that Behavior.

If you were to make the following change to Send Message:

```

This is an Example Test
Start SAGE
Connect To SAGE Runtime 121.0.0.1 50001
Create Agent recipientNode recipientAgent
Add Behavior recipientNode recipientAgent ExampleBehavior ↪
↪ExampleBehavior.jar Topic1
Activate Agent recipientNode recipientAgent
Send Message recipientNode recipientAgent Topic10000 Hello
Deactivate Agent recipientNode recipientAgent
Remove Agent recipientNode recipientAgent

```

You would expect the following results from your executed test case:

```

begin
end

```

This is due to the message not being propagated to any Behavior, as no Behavior was added that is associated with Topic10000.

SAGE Robot Framework Keywords

SAGE ROBOT FRAMEWORK KEYWORDS

Running Your Automation Test

RUNNING YOUR AUTOMATION TEST

You are ready to execute your test once you have constructed your Robot file with the proper SAGE Keywords.

All test cases are ran at the **Server level**.

The Server will communicate with each connected Node, whether local or remote. This allows you to store all your test cases in a single location while testing across multiple network connected machines.

Execute a robot file using the command-line

```
java -cp "%SAGE_CLASSPATH%" org.robotframework.RobotFramework path/to/my_
↳ tests/Hello.robot
```

Execute a robot file using the SAGE App

- 1) In the SAGE App click the Robot Interface button.
- 2) Click the Add button.
- 3) Using the finder, select your Robot file.
- 4) Once your Robot file has been added to the list, click the Start button.

Capturing the Test Results

CAPTURING THE TEST RESULTS

A `Result` object provides a reporting mechanism back to the server to indicate the results of the execution of a Behavior's methods.

The `m_logMessages` member is a vector of strings and is intended to provide a log of events that occurred during the execution of a Behavior.

You can capture messages back to your Robot report by appending to this `m_logMessages` list:

```
public boolean action(Result result)
{
    System.out.println("Hello World from Action");
    result.m_logMessages.add("This is log message string3.");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}

public boolean message(Message message, Result result)
{
    System.out.println("Got message");
    result.m_logMessages.add("This is log message from message string1.");
    result.m_executionResult = ExecutionResultType.CompletionSuccess;
    return true;
}
```

As a result, your Robot log should read:

```
This is log message string3.

This is log message from message string1.
```

While your console reads:

```
Hello World from Action

Got message
```

All execution logs and results are pushed back to the Server. Logs and results can be found on the **Server machine** in the following directories:

```
Logs directory: "C:\ProgramData\SAGE\logs"

Results directory: "C:\ProgramData\SAGE\results"
```

Creating Your Own Controller Application

CREATING YOUR OWN CONTROLLER APPLICATION

A Controller in the SAGE framework is an application that interacts with the SAGE Server to create and manage a network of Agents. SAGE Controller applications are external applications that remotely connect to an instance of a SAGE Server running in either a console application (ex. SageServerConsole_x64.exe) or in the SageServerApplication.

The SAGE SDK provides both C++ and Java bindings for creating Controller applications. The SageRemoteInterface class provides an interface for creating a remote connection to an instance of the SAGE Server and for creating and managing agents.

Using the SageRemoteInterface to create and manage an Agent network, a Control application performs the following steps:

- 1) Create an instance of the SageRemoteInterface class.
- 2) Connect to the SAGE Server by providing an IP address and port number.
- 3) Create and manage the Agent network.
- 4) Synchronize events.
- 5) Retrieve results.
- 6) Disconnect from the SAGE server.
- 7) Destroy the SageRemoteInterface object.

The SageRemoteInterface class provides a number of methods to interface with Controller applications.

24.1 addBehavior

This method adds a new Behavior to a SAGE Agent. Behaviors are units of execution that define how an Agent acts either due to a deterministic execution pattern, or in response to an incoming message. This method allows the caller to specify an optional set of topics that act as a message filter for the Agent. If a topic set is specified for a Behavior, then only incoming messages that specify one of those topics will be passed through to that Behavior. If no topic set is specified, then all messages are passed to the Behavior. This method returns a boolean value indicating whether or not the Behavior was successfully added.

- **nodeName [string]** This is the name of the SAGE Node in which the Agent resides.
- **agentName [string]** This is the name of the Agent.
- **behaviorName [string]** This is the name that was assigned to the SAGE Behavior by its developer.
- **module [string]** This is the file name of the module containing the Behavior to be added. Depending on the language the Behavior was written in, this could be a Java .class file, a Python .py file, or a native dynamic load library (.dll or .so).

- **topics [string vector]** The topic vector specifies the set of topic names that the new Behavior will be interested in. Messages sent to the Behavior's Agent will only be routed to this new Behavior if the topic specified in the message matches one of the topic names in this vector. Note that messages that do not specify a topic are sent to all Behaviors and conversely Behaviors that don't specify a topic list receive all messages irrespective of the topic the message specifies.

C++

```
bool addBehavior(std::string nodeName, std::string agentName, std::string_  
↳behaviorName, std::string module, std::vector<std::string> topics);
```

Java

```
public boolean addBehavior(String nodeName, String agentName, String behaviorName,_  
↳String module, StringVector topics);
```

24.2 connect

This method establishes a connection to a running SAGE Server either on the local machine or an a remote machine. It returns a boolean value indicating if the connection was successfully established.

- **serverIPAddress [string]** This is the IP address of the machine where the SAGE Server is running.
- **portNumber [integer]** This is the port number that the SAGE Server is using to accept new connection. The port number is specified when the server is started.

C++

```
bool connect(std::string serverIPAddress, int portNumber);
```

Java

```
public boolean connect(String serverIPAddress, int portNumber);
```

24.3 clearExecutionResults

This method clears the set of Result objects reported to the SAGE Server. Control Applications can use this method after retrieving Result objects using getExecutionResults to avoid duplicates.

C++

```
void clearExecutionResults();
```

Java

```
public void clearExecutionResults();
```

24.4 createAgent

This method creates a SAGE Agent in the specified SAGE Node. The Node is identified by the name that was given when it was created. Node names must be unique across Agent networks. The Agent is created in an inactive state and must be activated before it will initiate any of its Behaviors. This method returns a boolean value indicating whether or not the Agent was created.

- **nodeName [string]** This is the name of the SAGE Node in which the Agent will be created.
- **agentName [string]** This is the name that will be assigned to the new SAGE Agent. Agent names must be unique within a SAGE Node.

C++

```
bool createAgent(std::string nodeName, std::string agentName);
```

Java

```
public boolean createAgent(String nodeName, String agentName);
```

24.5 disconnect

This method ends a connection to a running SAGE Server.

C++

```
void disconnect();
```

Java

```
public void disconnect();
```

24.6 getAgentNames

This method returns the names of all the SAGE Agents residing in the specified Node.

- **nodeName [string]** This is the name of the SAGE Node whose Agent names are to be returned.
- **agentNames [vector of strings]** This is a vector of strings containing SAGE Agent names.

C++

```
bool getAgentNames(std::string nodeName, std::vector<std::string>& agentNames);
```

Java

```
public boolean getAgentNames(String nodeName, StringVector agentNames);
```

StringVector is a Java **class** that provides a **get(int i)** method **for** retrieving the **Agent name at index i**.

24.7 getExecutionResults

This method returns all the Result objects that have been reported to the SAGE Server at the time of the call. This method does not clear the set of Result objects. Subsequent calls to this method will return the cumulative Result objects until the clearExecutionResults method is called.

- **results** [vector of Result objects]

C++

```
void getExecutionResults(std::vector<Result>& results);
```

Java

```
public void getExecutionResults(ResultVector results);
```

ResultVector is a Java **class** that provides a `get(int i)` method **for** retrieving the `Result` at index `i`.

The Result class enables Behaviors to report the results of the execution of either a `action()` method or a `message()` method. It consists of the following members:

- **m_logMessages** while the container type varies across languages, the `m_logMessages` member is a vector of strings and should be intended to provide a log of events that occurred during the execution of a Behavior.
- **m_exception** if the execution of code in the `action()` method or a `message()` method generates an exception, the `m_exception` string should contain text describing the cause of the exception.
- **m_executionResult** this member consists of an enumeration of type `ExecutionResultType`. It can have one of three values:
 - `CompletionSuccess` - the method completed execution successfully
 - `CompletionFailure` - the method completed execution but failed
 - `ExceptionThrown` - code in the method caused an exception to be thrown. It is highly recommended that Behavior developers execute any code that can potentially generate exceptions withing a try - catch statement.

C++

```
std::vector<std::string> m_logMessages;  
std::string m_exception;  
ExecutionResultType m_executionResult;
```

Java

```
public ArrayList<String> m_logMessages = new ArrayList<String>();  
public String m_exception = "";  
public ExecutionResultType m_executionResult = ExecutionResultType.CompletionSuccess;
```

24.8 getNodeNames

This method returns the names of all SAGE Nodes connected to this SAGE Server.

- **nodeNames [vector of strings]** This is a vector of strings containing SAGE Node names.

C++

```
void getNodeNames(std::vector<std::string>& nodeNames);
```

Java

```
public void getNodeNames(StringVector nodeNames);
```

StringVector is a Java **class** that provides a **get(int i)** method **for** retrieving the **Node** name at index **i**.

24.9 sendFile

This method transmits a file to a SAGE Node. This capability is useful for sending auxiliary data or executable library files needed by Behaviors. Files transmitted to a SAGE Node running on the Windows operating system are placed in the sage\data subfolder of the ProgramData folder (usually C:\ProgramData\SAGE\data). On the Linux operating system, files are placed in the /etc/sage/data folder.

- **nodeName [string]** This is the name of a SAGE Node where the file will be sent.
- **filePath [string]** This is a fully qualified path to the file.

C++

```
bool sendFile(std::string nodeName, std::string filePath);
```

Java

```
public boolean sendFile(String nodeName, String filePath);
```

24.10 sendMessage

The sendMessage method enables the Control Application to send messages to Agents.

C++

```
bool sendMessage(const sageframework::Message& message)
```

Java

```
public boolean sendMessage(Message message);
```

The Message class provides the necessary mechanism to specify a message recipient and the message content.

- **m_targetNodeName [string]** This is the name of the SAGE Node that is the recipient of this message. If left empty, the message is sent to all the Nodes in the SAGE network.
- **m_targetAgentName [string]** This is the name of the SAGE Agent that is the recipient of this message. If left empty, the message is sent to all the Agents in the specified SAGE Node.

- **m_topic [string]** If specified, a topic name acts to direct a message to only those Behaviors that have that topic included in their topics list. This provides a filtering mechanism so that messages are only sent to those Behaviors interested in them.
- **m_message [string]** This is an arbitrary, application defined string that specifies what the message is.
- **m_data [string vector]** This string vector allows the message sender to include an arbitrary number of data items with the message. The content of that data is application defined.

C++

```
class Message
{
public:
    std::string m_targetNodeName;
    std::string m_targetAgentName;
    std::string m_topic;
    std::string m_message;
    std::vector<std::string> m_data;
}
```

Java

```
public class Message
{
    public String m_targetNodeName;
    public String m_targetAgentName;
    public String m_topic;
    public String m_message;
    public ArrayList<String> m_data = new ArrayList<String>();
}
```

24.11 setAgentActive

This method sets the active state of an SAGE Agent. Agents are created in an inactive state so that Agent Behaviors are not eligible for execution. This allows the Control Application to populate the Agent with Behaviors before any Behaviors execute. This method returns a boolean value indicating whether or not the Agent state was set.

- **nodeName [string]** This is the name of the SAGE Node in which the Agent resides.
- **agentName [string]** This is the name of the Agent.
- **isActive [boolean]** This is a boolean flag specifying whether the Agent state should be active (true), or inactive (false).

C++

```
bool setAgentActive(std::string nodeName, std::string agentName, bool isActive);
```

Java

```
public boolean setAgentActive(String nodeName, String agentName, boolean isActive);
```

24.12 removeAgent

This method removes a SAGE Agent from the specified SAGE Node. The Node is identified by the name that was given when it was created. It is good practice for Control applications to remove any Agents they created before exiting. This method returns a boolean value indicating whether or not the Agent was removed.

- **nodeName [string]** This is the name of the SAGE Node in which the Agent will be removed.
- **agentName [string]** This is the name of the Agent to remove.

C++

```
bool removeAgent(std::string nodeName, std::string agentName);
```

Java

```
public boolean removeAgent(String nodeName, String agentName);
```

24.13 removeBehavior

This method removes an existing Behavior from an Agent. This will cause the Behavior's `tearDown()` method to be invoked.

- **nodeName [string]** This is the name of a SAGE Node where the target Agent exists.
- **agentName [string]** This is the name of the SAGE Agent that will have its Behavior removed.
- **behaviorName [string]** This is the name of the Behavior that will be removed.

C++

```
void removeBehavior(std::string nodeName, std::string agentName, std::string_  
↪behaviorName);
```

Java

```
public void removeBehavior(String nodeName, String agentName, String behaviorName)
```

24.14 waitForResult

This method causes the Control Application to block until a specified Behavior executes either its `action()` method or its `message()` method and generates a Result object. This capability is useful for synchronizing the execution of the Control Application with events in the Agent network.

- **nodeName [string]** This is the name of a SAGE Node where the Agent resides.
- **agentName [string]** This is the name of the SAGE Agent that has the target Behavior.
- **behaviorName [string]** This is the name of the Behavior that generates the Result object.
- **timeOut [int]** This is the maximum time, in milliseconds, that the Control Application will block waiting for the Result object.

C++

```
bool waitForResult(std::string nodeName, std::string agentName, std::string_  
↳behaviorName, int timeout);
```

Java

```
public boolean waitForResult(String nodeName, String agentName, String behaviorName,_  
↳int timeout);
```

Help and FAQ

HELP AND FAQ

- *Do I need to install both the Server and the Node on every machine?*
- *How can I verify the server is running?*
- *Is there a default port number that is commonly used?*
- *When I deploy SAGE agents to other VMs, how can I verify that they are fully operational?*
- *Why am I getting a python27.dll is missing error when attempting to connect a SAGE Node?*
- *Why am I getting a jvm.dll is missing error when attempting to connect a SAGE Node?*
- *Why am I receiving a 'sageframework.SageLocalInterfaceWrapperPINVOKE' exception upon starting the SAGE App?*
- *How do I resolve "error while loading shared libraries: libjvm.so" when starting a Node on Linux?*
- *Why am I getting "ImportError: No module named robot"?*
- *Why do I receive "WARNING: An illegal reflective access operation has occurred" upon running a Robot file?*
- *Why does the SAGE App not open?*

Do I need to install both the Server and the Node on every machine?

No. Only one Server instance is required. All tests are to be launched from the machine that has the SAGE Server instance. The Server will distribute behavior files to the target Nodes via message. This gives SAGE users the ability to control complex test scenarios, involving multiple nodes and agents across various machines, from a single controller entity.

How can I verify the server is running?

To verify a running SAGE Server instance, check the prompt window and confirm that the following text is displayed, "SAGE server started listening on port: {portNumber}". This indicates that the SAGE Server is active and listening over the selected port.

Is there a default port number that is commonly used?

There is no default port number for SAGE. It is recommended to avoid using dedicated port numbers, such as 21 (FTP), 22 (SSH), and 80 (HTTP). For a list of unassigned port numbers visit [Internet Assigned Numbers Authority \(IANA\)](https://www.iana.org/).

When I deploy SAGE agents to other VMs, how can I verify that they are fully operational?

It is important to not confuse the terms SAGE agents and SAGE nodes. SAGE agents are created on-the-fly using a SAGE controller interface (Robot Framework by default). Currently, there is only support for agent creation using Robot Keyword, `Create Agent`. When you create an agent, it is deployed to a SAGE Node. SAGE nodes are containers that are deployed to a VM/machine. These containers store agents upon their creation.

When you initialize a node, you will assign it a IP Address and a port number value. The IP Address value should be the IP Address of the server VM/machine. The port number must match that of the recipient server's port number. A matching port number and valid IP is essential for connecting a node to the intended SAGE Server. Once connected, check the prompt window for the following text, "Connected to SAGE runtime using ip address {server_IP_address} : {portNumber}", to verify the node is operational. It is recommended to download the example files to get a better understanding of the SAGE Node deployment and agent creation process.

Why am I getting a python27.dll is missing error when attempting to connect a SAGE Node?

This error either means there are multiple Python instances installed on your machine, a non-supported Python version installed on your machine, or your installed Python bit version differs from that of the installed SAGE. SAGE currently supports [Python 2.6](#) or [2.7](#). Be sure you have one of the two installed. Next, be sure the bit version is consistent with the installed SAGE bit version. To check the Python bit version, start Python in the command prompt and it will display the bit number. Check your Python directory e.g C:\Python27 and verify that the python27.dll file is there. If not, it is recommended to uninstall the current Python instance and to do a fresh install. Be sure your [Environment variables](#) PYTHONHOME and PATH are set and updated, respectively. Visit [here](#) to learn how to add and edit environment variables.

Why am I getting a jvm.dll is missing error when attempting to connect a SAGE Node?

This error either means there are multiple Java instances installed on your machine, a non-supported Java version installed on your machine, or your installed Java bit version differs from that of the installed SAGE. SAGE currently supports [Java 8](#). Be sure you have a supported version installed. Next, be sure the bit version is consistent with the installed SAGE bit version. To check the Java bit version, type `java -version` in the command prompt. Check your Java JDK server directory e.g C:\Program Files\Java\jdk1.8.0_91\jre\bin\server and verify that the jvm.dll file is there. If not, it is recommended to uninstall the current Java instance and to do a fresh install. Be sure your [Environment variables](#) JAVAHOME and PATH are set and updated, respectively. Visit [here](#) to learn how to add and edit environment variables.

Why am I receiving a 'sageframework.SageLocalInterfaceWrapperPINVOKE' exception upon starting the SAGE App?

The SAGE App goes through a behavior discovery process to locate local "Native", "Java", and "Python" behavior files. This exception is thrown if both Java and Python are not found on your machine. Be sure your [Environment variables](#) PYTHONHOME, JAVAHOME, and PATH, are set and updated. Visit [here](#) to learn how to add and edit environment variables.

How do I resolve "error while loading shared libraries: libjvm.so" when starting a Node on Linux?

SAGE needs JVM dynamic library to be in the runtime library search path. This is accomplished by including the location in the LD_LIBRARY_PATH environmentvariable. This is typically done by including the definition in .bashrc file or .profile file. It can also be temporarily set using the same command, typically: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:{JRE_HOME}/lib/amd64/server`

Why am I getting "ImportError: No module named robot"?

This error is returned when the robot framework module cannot be found. By default, robot-framework is distributed with SAGE and appended to the [Environment variables](#) SAGE_CLASSPATH system Environment variable upon install. Be sure to verify that the robotframework-2.9.2.jar is located in the "C:\Program Files\NRL\SageServer" directory. In addition, check the [Environment variables](#) SAGE_CLASSPATH environment variable to ensure the absolute path to the jar "C:\Program Files\NRL\SageServer\robotframework-2.9.2.jar" has been added and appears as the first item in the list.

Why do I receive "WARNING: An illegal reflective access operation has occurred" upon running a Robot file?

This warning is a known issue in the current Robot Framework build and is caused by Java 9+ reflection and package access changes. You can continue with testing until Robot Framework pushes a new update to address the changes.

Why does the SAGE App not open?

If you said yes to Python, make sure you have PYTHONHOME set correctly and that it is in your PATH.

Example

EXAMPLE

Download and extract the “Example - Sender and Receiver” package from the [Download](#) page.

1. Compile `SenderReceiverBehavior.java` file then generate JAR file by opening the terminal and typing the commands:

```
javac -cp "%SAGE_SERVER_HOME%\SageJavaBehaviorInterface.jar_  
↪SenderReceiverBehavior.java  
  
jar cf SenderReceiverBehavior.jar SenderReceiverBehavior.class
```

2. Directory structure

On Server machine copy the Behavior file into the Java Behavior sub-directory:

```
For Windows: C:\ProgramData\SAGE\behaviors\Java\SenderReceiverBehavior.jar  
For Linux: /etc/Sage/behaviors/Java/SenderReceiverBehavior.jar
```

26.1 Test case

SenderReceiverTest.robot

Robot file that uses the SageRobotKeywords [Library] and contains one [Test Case] titled “Test Message Communications”.

```
# "%JAVAHOME_X64%\bin\java.exe" -cp "%SAGE_CLASSPATH%" org.robotframework.  
↪RobotFramework SenderReceiverTest.robot  
  
*** Settings ***  
Documentation    Used to test communications across the Agent network.  
...             Sets up SAGE Agents with their corresponding Behavior and activates_  
↪them.  
...             Waits until the Agents are done communicating to generate report.  
  
Library          SageRobotKeywords  
Test Setup       Connect to SAGE Server  
Test Teardown    Log Results  
  
*** Test Cases ***  
Test Message Communications  
    # Create Agents on existing Nodes  
    Create Agent    NodeOne    Agent1
```

(continues on next page)

(continued from previous page)

```

        Create Agent      NodeTwo    Agent2

        # Add Behaviors to Agents so that they can learn skill
        Add Behavior      NodeOne    Agent1      SenderReceiverBehavior
↪SenderReceiverBehavior.jar talking
        Add Behavior      NodeTwo    Agent2      SenderReceiverBehavior
↪SenderReceiverBehavior.jar talking

        # Activate Agents
        Activate Agent    NodeOne    Agent1
        Activate Agent    NodeTwo    Agent2

        # Wait until each Agent reports back to the Server upon completion
        Wait For Result    NodeOne    Agent1      SenderReceiverBehavior    teardown
↪100000
        Wait For Result    NodeTwo    Agent2      SenderReceiverBehavior    teardown
↪100000

*** Keywords ***
Connect to SAGE Server
    Start Sage
    Connect to Sage Runtime    127.0.0.1    50001

Log Results
    # Gather results that have been logged
    @{RESULTS}      Get Results

    :FOR      ${ELEMENT}      IN      @{RESULTS}
    \      Report Result      ${ELEMENT}

    # Clear results so that no results are cached
    Clear Results

```

[Test Setup] includes running the Keyword named “Connect to SAGE Server”, identified in the *Keywords* table which includes:

- Start Sage
- Connect to Sage Runtime

The [Test Case] involves dynamically constructing two Agent entities on separate connected Nodes. Both Agents are to learn the same Behavior.

In order to add the Behavior to an Agent you must give the Add Behavior keyword the following arguments:

- *nodeName* - refers to target Node.
- *agentName* - refers to target Agent.
- *behaviorName* - refers to the Behavior’s name (m_name) variable found within the Behavior’s constructor.
- *behaviorModule* - refers to the Behavior filename.
- *topics* - allows you to add tags to Behaviors so that you can properly associate a message with a particular Behavior.

Incorporating the Wait For Result keyword ensures that Robot will not conclude prematurely. In this example, you wait until the teardown() method of each Agent returns the results of its execution before concluding the test. This

enables the Server to wait until the Agents have finished communicating before generating a report. Otherwise, Robot will conclude the test after activating Agent2.

After the test has concluded, the [Test Teardown] execution will gather the logged events from the preceding steps, as well as clear those logs once captured.

SenderReceiverBehavior.java

Behavior file that demonstrates the use of having both a reactive and proactive execution response.

```

/* Use the following command line to build:
    javac -cp "%SAGE_SERVER_HOME%\SageJavaBehaviorInterface.jar\u
↪SenderReceiverBehavior.java

    Create JAR file:
    jar cf SenderReceiverBehavior.jar SenderReceiverBehavior.class
*/

import nrl.sage.BehaviorInterface.*;

public class SenderReceiverBehavior extends SageBehavior
{
    int count;
    String senderAgent;
    String senderNode;
    String receiptAgent;
    String receiptNode;

    public SenderReceiverBehavior()
    {
        m_name = "SenderReceiverBehavior";
        m_description = "Periodically sends messages to other agents all\u
↪while receiving messages";
        m_executionType = ExecutionType.TimedCyclical;
        m_delay = 50;
        m_period = 200;
    }

    public boolean setUp(Result result)
    {
        count = 1;
        result.m_executionResult = ExecutionResultType.NotSet;
        return true;
    }

    public boolean action(Result result)
    {
        try{
            // Using Agent state, retrieve the information of the Agent\u
↪using this Behavior
            senderAgent = (String)getState("agent");
            senderNode = (String)getState("node");

            // Knowing who the sender agent is, store the intended agent\u
↪recipient information
            if (senderAgent.equals("Agent1")) {
                receiptAgent = "Agent2";
                receiptNode = "NodeTwo";
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        else {
            receiptAgent = "Agent1";
            receiptNode = "NodeOne";
        }

        // Construct the message content to send.
        Message messageToAgent = new Message();
        messageToAgent.m_message = "send" + " messageNo:" + Integer.
↪toString(count);

        messageToAgent.m_data.add(senderAgent);
        messageToAgent.m_data.add(receiptAgent);
        messageToAgent.m_targetNodeName = receiptNode;
        messageToAgent.m_targetAgentName = receiptAgent;
        messageToAgent.m_topic = "talking";
        sendMessage(messageToAgent);

        // Increase count to know how many messages have been sent
        count++;

        System.out.println(senderAgent + " sent " + messageToAgent.m_
↪message + " to " + receiptAgent);
    }
    catch (Exception e){
        result.m_exception = e.toString();
        result.m_executionResult = ExecutionResultType.
↪ExceptionThrown;
    }

    result.m_executionResult = ExecutionResultType.NotSet;
    return true;
}

public boolean message(Message message, Result result)
{
    // Print the message content in the console
    System.out.println(message.m_data.get(1) + " received " + message.m_
↪message + " from " + message.m_data.get(0));

    // Deactivate agents after 20 messages have been sent
    if (count>20){
        setAgentActive(receiptNode, receiptAgent, false);
        setAgentActive(senderNode, senderAgent, false);
    }

    result.m_executionResult = ExecutionResultType.NotSet;
    return true;
}

public boolean tearDown(Result result)
{
    System.out.println("Tearing down behavior " + m_name);

    // Log event
    result.m_logMessages.add(senderAgent + " signing off. Goodbye!");

    // Report back to the Server indicating a successful teardown
    result.m_executionResult = ExecutionResultType.CompletionSuccess;

```

(continues on next page)

(continued from previous page)

```

        return true;
    }
}

```

The following indicates that the Agent will proactively respond at activation:

- *m_executionType* has been set to TimedCyclical;

Proactive responses invoke the `action()` method. In this example, the response is to construct and send a message to a fellow Agent after identifying the originating Agent.

TimedCyclical execution type continuously invokes the `action()` method. This continuous calling of the `action()` method enables the Agent to send more than one message.

The `message()` method handles the Agents reactive response and is invoked by way of message. In this example, the Agent is assumed to receive messages that contain the following content:

- *message.m_message* - contains a string.
- *message.m_data.get(0)* - contains the sender's name.
- *message.m_data.get(1)* - contains the receiver's name.

Once the Agent has received 20 messages it will proceed to deactivate its fellow Agent, as well as itself.

The `tearDown()` method will be invoked upon deactivation, signaling the Agent to report the result of the `tearDown` back to the Server.

26.2 Running test

Ensure that both the SAGE Server and SAGE Node were installed with Java support. In addition, ensure that the Behavior file has been placed in the correct directory (*C:\ProgramData\SAGE\behaviors\Java*) on the Server machine. If not yet done, add the Behavior file to the specified directory.

Start the SAGE Server by opening the terminal and typing the command:

```
SageServerConsole_x64.exe 50001
```

Create and connect a Node named NodeOne by typing:

```
SageNodeConsole_x64.exe NodeOne 127.0.0.1 50001
```

Create and connect a Node named NodeTwo by typing:

```
SageNodeConsole_x64.exe NodeTwo 127.0.0.1 50001
```

Upon successful connection, you will be ready to run the test.

Start the example by typing:

```
java -cp "%SAGE_CLASSPATH%" org.robotframework.RobotFramework_
↳SenderReceiverTest.robot
```

26.3 Generated results

A report and log HTML file are generated after running test. Report gives you an overview of the test execution by detailing viewable statistics including Pass/Fail ratios and elapsed times. Log details statistics from each step of the test execution, from keyword to keyword. It enables you to drill down on the specifics of the test in case of failure or otherwise [*].

Expand the elements to find returned results for each step execution. For more information regarding report and log files, see [Robot Framework output documentation](#).

This log is based on successfully running SenderReceiverTest.robot.

SenderReceiverTest Test Log

Generated
20190328 11:56:47 GMT-07:00
27 seconds ago

REPORT

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:04	
All Tests	1	1	0	00:00:04	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
SenderReceiverTest	1	1	0	00:00:05	

Test Execution Log

SUITE SenderReceiverTest	00:00:04.916
Full Name: SenderReceiverTest	
Documentation: Used to test communications across the Agent network. Sets up SAGE Agents with their corresponding Beahvior and activates them. Waits until the Agents are done communicating to generate report.	
Source: C:\Users\Desktop\SenderReceiverTest.robot	
Start / End / Elapsed: 20190328 11:56:42.262 / 20190328 11:56:47.178 / 00:00:04.916	
Status: 1 critical test, 1 passed, 0 failed 1 test total, 1 passed, 0 failed	
TEST Test Message Communications	00:00:04.093
Full Name: SenderReceiverTest.Test Message Communications	
Start / End / Elapsed: 20190328 11:56:43.080 / 20190328 11:56:47.173 / 00:00:04.093	
Status: PASS (critical)	
SETUP Connect to SAGE Server	00:00:00.037
Start / End / Elapsed: 20190328 11:56:43.091 / 20190328 11:56:43.128 / 00:00:00.037	
KEYWORD SageRobotKeywords.Start Sage	00:00:00.012
KEYWORD SageRobotKeywords.Connect To Sage Run Time 127.0.0.1, 50001	00:00:00.011
KEYWORD SageRobotKeywords.Create Agent NodeOne, Agent1	00:00:00.004
KEYWORD SageRobotKeywords.Create Agent NodeTwo, Agent2	00:00:00.004
KEYWORD SageRobotKeywords.Add Behavior NodeOne, Agent1, SenderReceiverBehavior, SenderReceiverBehavior.jar, talking	00:00:00.006
KEYWORD SageRobotKeywords.Add Behavior NodeTwo, Agent2, SenderReceiverBehavior, SenderReceiverBehavior.jar, talking	00:00:00.005
KEYWORD SageRobotKeywords.Activate Agent NodeOne, Agent1	00:00:00.004
KEYWORD SageRobotKeywords.Activate Agent NodeTwo, Agent2	00:00:00.004
KEYWORD SageRobotKeywords.Wait For Result NodeOne, Agent1, SenderReceiverBehavior, teardown, 100000	00:00:03.965
KEYWORD SageRobotKeywords.Wait For Result NodeTwo, Agent2, SenderReceiverBehavior, teardown, 100000	00:00:00.004
TEARDOWN Log Results	00:00:00.037
Start / End / Elapsed: 20190328 11:56:47.135 / 20190328 11:56:47.172 / 00:00:00.037	
KEYWORD @({RESULTS}) = SageRobotKeywords.Get Results	00:00:00.009
FOR \${ELEMENT} IN [@({RESULTS})]	00:00:00.018
KEYWORD SageRobotKeywords.Clear Results	00:00:00.002

Robot Framework Background

ROBOT FRAMEWORK BACKGROUND

“Robot Framework” by Robot Framework Foundation is licensed under the [Apache License 2.0](#).
All information presented is derived from [Robot Framework User Guide](#) .

Robot Framework is a generic test automation framework that uses the keyword-driven testing approach.
Keyword-driven testing uses keywords (or action words) to perform a specific task.

- Facilitates simple to use tabular syntax to create test cases.
- Generates easy to read result reports as well as logs in HTML format.
- Independent of application and platform.
- Supports creation of data driven test cases.
- Provides tagging for categorizing and selecting test cases that are to be executed.
- Has built in support for variables to enable testing in different environments.

File extension

Starting from Robot Framework 2.7.6, every robot framework file is saved as a .ROBOT file extension type.

e.g., *sageTest.robot*

Settings

Robot Framework test data is defined in tabular format, thus all content outside of the recognized table types is ignored.

The asterisk (*) symbol is used to distinguish each table. When creating each table you can input “*”, “***”, or “****” before you add the name of the table. For consistency purposes it is common to use “****” before you add the table name. Be sure that the table name is spaced between the asterisks.

```
*** Settings ***
```

Four common settings are *[Documentation]*, *[Library]*, *[Test Setup]*, and *[Test Teardown]*.

Documentation

The *[Documentation]* setting allows you specify documentation of a test case. The supporting text is shown in both the command line output, as well as the test logs and reports.

```
*** Settings ***
Documentation      Written explanation of test case to be documented
```

Library

Importing supporting Robot Framework libraries is essential to make use of its keywords and their functionality. Test libraries may be imported using the *[Library]* setting.

The library name is both case- and space-sensitive. If a library is in a package, the full name including the package name must be used.

```
*** Settings ***
Documentation      Written explanation of test case to be documented

Library           SageRobotKeywords
```

The Library setting is Robot Framework’s equivalent of Java’s and Python’s respective import statement. Import SageRobotKeywords to use SAGE Framework specific keywords.

Test Setup

The *[Test Setup]* setting allows you to run a Keyword before running your test case. This is useful for setting up your environment prior to conducting your test, such as configuring your network or having the proper windows open.

```
*** Settings ***
Documentation      Written explanation of test case to be documented

Library           SageRobotKeywords
Test Setup        Start SAGE
```

Test Teardown

The *[Test Teardown]* setting allows you to run a Keyword after your test case either completes or fails. This is useful for performing any clean-up and shutdown activities, such as deactivating your agents or closing any active windows.

```
*** Settings ***
Documentation      Written explanation of test case to be documented

Library           SageRobotKeywords
Test Setup        Start SAGE
Test Teardown     Deactivate Agent    MyNode    MyAgent
```

Test case

The first column in the **Test Cases** table contains the test case name

```
*** Test Cases ***
TestName
```

“*TestName*” is the name of the test case, while “**Test Cases**” is the table name.

A test case concludes once a new test case or table is declared or if the current test reaches its final keyword.

Keywords

Keywords are considered as functions or methods that can be used for testing. Each keyword correspond to a set of actions.

Keywords are specified in the second column. Keywords are `Start SAGE` and `Connect to SAGE Runtime`. Columns after the keyword name contain possible arguments to the specified keyword. Arguments are **server_IP_address** and **port_number**.

Robot Framework recommends using *four spaces between columns*.

```

*** Test Cases ***
TestName
    Start SAGE
        Connect to SAGE Runtime    {server_IP_address}    {port_number}

```

Comments

In order to comment out a line include the pound (#) symbol.

```

*** Test Cases ***
TestName
    Start SAGE
        Connect to SAGE Runtime    {server_IP_address}    {port_number}
        # This keyword is commented out

```

Arguments

Keywords may accept zero or more arguments. The number of arguments a keyword accepts depends on its implementation, and typically the best place to search for this information is located in a keyword's documentation (if supplied by the developer).

If an argument has a default value you may specify that value by separating the argument name and its value with an equal sign.

```

*** Test Cases ***
TestName
    Start SAGE
        Connect to SAGE Runtime    {server_IP_address}    {port_number}
        # This keyword is commented out
        Create Agent    nodeName=MyNode    agentName=MyAgent

```

Robot file runtime

Robot Framework is executed in these following steps:

1. Collecting test cases, reading and setting variables
2. Running all the steps in every test case
3. Providing the execution statistics (which test cases have passed/failed)
4. Writing the detailed log in xml format
5. Generating the report and log in html format.

All executed test suites and test cases, as well as their statuses, are shown in real time.

Changelog

CHANGELOG

SAGE 2.0.28

Release date: 2019-12-09

Core and Builtins

- Issue #52: Fixed bug in periodic actions that occurred when the period specified is very small. This could lead to a race condition.

SAGE 2.0.24

Release date: 2019-08-05

Core and Builtins

- Issue #51: Fixed bug, Java behaviors within JAR files that include supporting libraries could throw a duplicate class exception due to how Java loads dependencies.

Server Application

- Some UI enhancements.
- Bug fixes.

SAGE 2.0.22

Release date: 2019-06-11

Core and Builtins

- Issue #50: Fixed bug, under some conditions it was possible to get the Server to hang if you were trying to stop it but it was also waiting for a Result.
- Issue #49: Behavior members are now checked on each run of the scheduler. That way action and message methods can change period, execution type, delay, and trigger message at runtime.
- Issue #48: The environment variable SAGE_HOME has been replaced with SAGE_SERVER_HOME and SAGE_NODE_HOME.
- Issue #47: Better error checking to make sure multiple Agents on the same Node can't use different versions of a Behavior (Note: they will use different instances).
- Issue #46: Behaviors will now only be sent from the Server to the Node if the Node does not already have the latest version.
- Issue #45: Better information and error reporting from Nodes back to the Server.
- Issue #44: Fixed bug where under certain conditions a Node that was shut down on purpose would show an error message.

- Issue #43: For Java, Sage no longer uses the system CLASSPATH environment variable and instead uses SAGE_CLASSPATH.
- Issue #42: All Java behaviors now must be in jars and are internally loaded by a custom class loader.
- Issue #41: Fixed a bug where a TimedCyclical behavior could drift over time.
- Issue #40: Fixed an issue where a new version of a Behavior was not properly replacing the old version in a Node.
- Issue #39: Added Keywords to Sage Robot Keywords to allow access to individual parts of a Result object, for example “Get Result Behavior Name”.

Server Application

- Light and dark color scheme enhancements.
- Behavior Manager now a non-modal dialog window.
- Added application settings for specifying which non-modal dialog windows stay on top of the main form window.
- Added double-click event handling for agent tree view nodes.
- Bug fixes.

SAGE 1.2.18

Release date: 2018-10-08

Core and Builtins

- Issue #38: Fixed message broadcasting bug.
- Issue #37: Added JSON based introspection for getting detailed Agent state.
- Issue #36: Fixed introspection bug.
- Issue #35: Added new scheduling mechanism that allows users to control the precision of the Node scheduler to control the CPU usage of SageNodeConsole.

Server Application

- Improved UI interactions for selecting Nodes, Agents, and Behaviors.
- Improved Network graph view layout.
- Added zoom control to Network graph view.
- Added application settings for controlling Node scheduler precision.
- Added light and dark theme selections.
- Fixed core bugs.

SAGE 1.1.14

Release date: 2017-09-28

Core and Builtins

- Issue #34: User’s CLASSPATH is now appended to the runtime CLASSPATH.
- Issue #33: Added new SAGE Robot keyword Clear Results.
- Issue #32: BehaviorModule class now only returns a single behavior instead of a vector of behaviors. This addresses potential memory corruption due to cross DLL allocation/deallocation.
- Issue #31: Fixed bug that would cause JVM to unexpectedly crash.

- Issue #30: Added server methods to load and save agent networks so that clients don't have to instantiate SAGE internal classes.
- Issue #29: Fixed bug that would cause the loss of Result objects.
- Issue #28: Fixed crash of Robot caused by Result objects being corrupted.
- Issue #27: Fixed GDI resource issue.
- Issue #26: Fixed issue with Behavior files not being overwritten on Node machine if Server contained newer version of file.
- Issue #25: Fixed problem with SAGE Server memory leak caused by the server running for very long periods of time.
- Issue #24: Fixed memory leak in SAGE Node code.
- Issue #23: Fixed issue where Node would not reconnect to SAGE Server after stopping then starting Server.
- Issue #22: Added support for sending files back to the server. Base class functionality now includes sendFile method.
- Issue #21: Added support for persistent networks. Network settings can be stored as SAGE extension files for save/load capability.

Server Application

- View SAGE User Guide from Help window.
- Add Network graph view.
- Integrate Import and Export network settings.
- Add "Activate All Agents", "Deactivate All Agents", and "Remove All Agents" buttons.
- Update layout and icons.

SAGE 1.0.12

Release date: 2017-06-27

Core and Builtins

- Issue #20: Fixed problem with SAGE ServerApp not allowing behaviors to be overwritten in the Behavior Manager.
- Issue #19: Fixed issue in SAGE ServerApp where behaviors would not show up in the tree view until the one of the behaviors' methods is called.
- Issue #18: Fixed problem where sending a message back to the sender in the message() method would crash SAGE.
- Issue #17: Added support for packaged behaviors. Behaviors and supporting class files may now be packaged into JAR and EGG files.

Server Application

- Items respond immediately to mouse-clicks when not in focus.
- Update layout and icons.
- Add bi-directional object selection.

Documentation

- Add Robot Framework background.
- Update SAGE Server Application.

SAGE 1.0.11

Release date: 2017-03-07

Core and Builtins

- Issue #16: JVM path automatically appended to the system environment PATH variable.
- Issue #15: Modified Namespaces for SageBehaviorInterface (both Java and Python module) to `nrl.sage.BehaviorInterface`, `SageRemoteInterface` to `nrl.sage.BehaviorInterface`, and `SageLocalInterface` to `nrl.sage.BehaviorInterface`.
- Issue #14: Linux SageNode rpm now correctly sets folder permissions in `/etc/sage`.
- Issue #13: Fixed problem with running robot files in SageServerApp when the user account does not have administrator privileges.
- Issue #12: Added a package name to the SageRemoteInterface.jar library. Java applications using that jar must import `sageremoteinterface.*`
- Issue #11: Added support that enables SAGE to be installed and operate properly on machines that don't have Java and/or Python installed. (Windows version only)
- Issue #10: Fixed problem with importing Java behaviors using the behavior manager in SageServerApp.

SAGE 1.0.10

Release date: 2016-12-15

Core and Builtins

- Issue #9: Improved Robot error reporting to include more informative messages.
- Issue #8: Fixed issue where the SageServer was not updated when an agent was removed by another agent that resides in the same Node.

SAGE 1.0.9

Release date: 2016-10-31

Core and Builtins

- Issue #7: Added source node and source agent fields to messages sent to Behaviors.
- Issue #6: Run Step and Report Result now throw an exception on result = "failure".
- Issue #5: Routines in SageRobotKeywords.java now return a boolean value.

SAGE 1.0.8

Release date: 2016-10-07

Documentation

- Update API Reference section with C++ Behavior file example.

Library

- Update Boost to 1.61.0.
- Removed Pluma dependencies.

SAGE 1.0.7

Release date: 2016-09-28

Core and Builtins

- Issue #4: Added support for asynchronous/multi-threaded calls.

- Issue #3: Wait For Result SageRobotKeyword now takes an additional argument that indicates which method to wait on to generate a result.
- Issue #2: The Behavior methods setUp() and tearDown() now are called with a Result parameter.
- Issue #1: Fixed bug that continued Behavior process after internal error occurred in Behavior methods setUp(), action(), message(), or tearDown().

Server Application

- Add new Options menu.
- Add new Options and Server tool strips.
- Add context sensitive interaction between the AgentTreeView and the menu/toolstrip tools.

SAGE (Sentry Agents) Framework

A dynamic multi-agent based solution for system automation

[Download](#)

Release v2.0.28

Latest Documentation Update: December 9, 2019



SAGE MAKES AUTOMATING COMPLEX SYSTEMS SIMPLE.



Language Agnostic

SAGE supports a wide range of popular programming languages including: C++, Java, and Python. No need to learn a new language to utilize SAGE.



Open Integration

SAGE permits language extensibility and enhancement so that you can easily integrate SAGE with other tools and applications. Built-in support is provided for Robot Framework.



Easy to use

SAGE is a cross-platform effort that was created with all users in mind. There is detailed documentation to support developers and testers alike.

E

environment variable

JAVAHOME, 49

LD_LIBRARY_PATH, 50

PATH, 49

PYTHONHOME, 49

SAGE_CLASSPATH, 48

SAGE_NODE_HOME, 48

SAGE_ROBOT_LIBRARY, 50

SAGE_SERVER_HOME, 48