SRTI - SIMPLE RUN-TIME INTERFACE

User and Developer Documentation

v2.20.01 – 2019-09-12

Table of Contents:

Chapter 0 – Release Notes

i. Summary

This document is an introduction to using SRTI from a **developer**'s perspective, providing background information to how the source code is organized. This is meant to help assist a user in how they can utilize the source code files directly, how they might wish to modify or add additional features, and provide a foundation for porting a compatible version of the SRTI client API for other programming languages.

SRTI (Simple Real-Time Interface) is a portable software solution to allow data transfer between different programs, be they in different languages or on different computer systems, be they local or connected through a network. This was built with the intention of being easy to use and maintain, primarily for scientific simulations to cooperate with each other in real time but can also be utilized in related IoT (Internet of Things) projects.

The project began in 2017 and is still under continuous development to add functions and improve efficiency. SRTI is free and open-source and is funded in part by the University of Michigan.

The full public source code, pre-compiled libraries, documentation, and other information can be found at https://github.com/hlynka-a/SRTI .

ii. Third-Party Licenses

As of 2018-04-11, SRTI is free and open-source, and operates under the Apache License, Version 2.0, allowing the right to freely use, modify and redistribute. This will remain unless a document at a later date states otherwise.

The following third-party libraries are included in the source-code and build of the SRTI software:

- Java
    - *javax.json* – provided under either CDDL or GPLv2 open-source license.
- C++
    - *rapidjson* – provided under MIT open-source license, which in turn uses libraries licensed under the BSD and JSON open-source licenses.

iii. Release Updates

2019-03-06

- This covers SRTI v2.05.01, made up of two files: if using the Java version, “SRTI_v2_03_01.jar” and “SRTI_Wrapper_v2_05_01.jar.” The functionality of SRTI v2.xx.xx may make it incompatible with v1.xx.xx.

2019-09-12

- This covers SRTI v2.20.02, which includes Wrapper files for Java, Matlab and Netlogo, allowing for inter-language compatibility between them, and other minor functional improvements and bug fixes.

Chapter 1 – Introduction

i. Definitions

IoT: Internet of Things.

RTI Lib: a client-side API library that allows a simulation to connect to an RTI Server. The RTI Lib must be locally stored and referenced on the same machine as the simulation.

RTI Manager: added-logic to RTI Server (introduced in SRTI v.2.00.00). Allows RTI Server to have some control to maintain a larger simulation system and keep it synchronized.

RTI Server: a server-side application that acts as the shared connection point for simulations in a simulation system. This can be run either on the same machine or a separate computer from individual simulations in a system.

RTI Wrapper: introduced in SRTI v2.00.00. Acts as an intermediary between RTI Lib API and a simulator. Relies on external .json configuration files to interact and utilize a simulator. With its inclusion, a simulator can be entirely independent of an SRTI (with minor restrictions).

Simulation: a computer model of something, especially for the purpose of study. Here, is used to refer to individual programs that might connect to the SRTI server.

Simulation System: Here, is used to refer to a collection of simulation systems that might connect to the SRTI server, such that they would be used together towards a shared goal or towards a representation of a larger, complex model.

SRTI: Simple Real Time Interface.

System: refer to *Simulation System*.

TCP: Transmission Control Protocol, a network protocol that provides reliable, ordered, and error-checked delivery of a stream of bytes between two nodes (a server and client).

Timestep: Used interchangeably in this document with "timestamp." Represents real time (typically, 'system time,' based on epoch time as given by default in Java). For example, if translated to real time: "201903061042123."

UDP: User Datagram Protocol, a network protocol that provides quick one-way delivery of bytes through a network port, not error-checked.

vTimestep: Used interchangeably in this document with "vTimestamp." Represents virtual time, as defined for simulators. For example: "1", "2", etc.

ii. What is SRTI?

SRTI (Simple Real-Time Interface) is a free, open-source and portable software solution that allows external software simulations to connect with each other and share information in real-time.

SRTI v1.00.00 is purely a data-transmission system. It uses a single RTI Server as the shared access point for individual simulator programs to join. Each simulator needs to locally reference the provided RTI Lib API to make a connection, which abstracts most of the low-level details of the connection. Sockets are used for the communication channel between RTI Server and RTI Lib. Messages sent through the system are transmitted in ‘string’ format, for better interoperability between languages, compared to pure byte code. To assist with this, JSON is used as the standard message format (although any string-representation can be used for the content from each simulator).

These design choices make SRTI v1.00.00 a naturally slower solution for software communication, but also allows distribution of the SRTI without the requirement of re-compiling it on different systems (relying mainly of Java code), and being easily used by most languages, or else, easy to port to support new languages that have functions for sockets and json parsing.

SRTI v2.00.00 is a more advanced and specialized update of v.1.00.00. Building upon the original data-transmission protocol, it adds new functionality based on a different goal: to explicitly support artificial simulation systems. In addition to the RTI Server and RTI Lib API, it adds an RTI Manager (coupled with RTI Server) and RTI Wrapper. Instead of having simulators control themselves and how they parse information from the SRTI, the RTI Wrapper takes over much of the responsibility, allowing easier design of larger simulation systems without explicit recoding.

SRTI v1.00.00 removed dependency on other simulators and added dependency on the SRTI itself and defined message types. SRTI v2.00.00 removes those added dependencies from a simulator’s code, and places them into the RTI Wrapper. This is a significant challenge and poses issues with language functionality and simulation system design.

The following chapters introduce a brief overview of SRTI v1.00.00 and introduce the new concepts for SRTI v2.00.00, including instructions to use it, and more detailed information about its backend logic.

iii. Core Concepts – SRTI v1.00.00

While this section will not go into detail with how the underlying logic of SRTI works, it will provide a high-level image to explain how a simulation fits into the system.

Figure 1 shows a representation of what a SRTI system looks like, including the ability to run aspects of it on different machines (or, if you choose, everything can be run on the same machine). Because of the internal use of sockets, a simulation in any language can connect to another simulation of any language, as long as they are compatible to reference the provided RTI Lib.

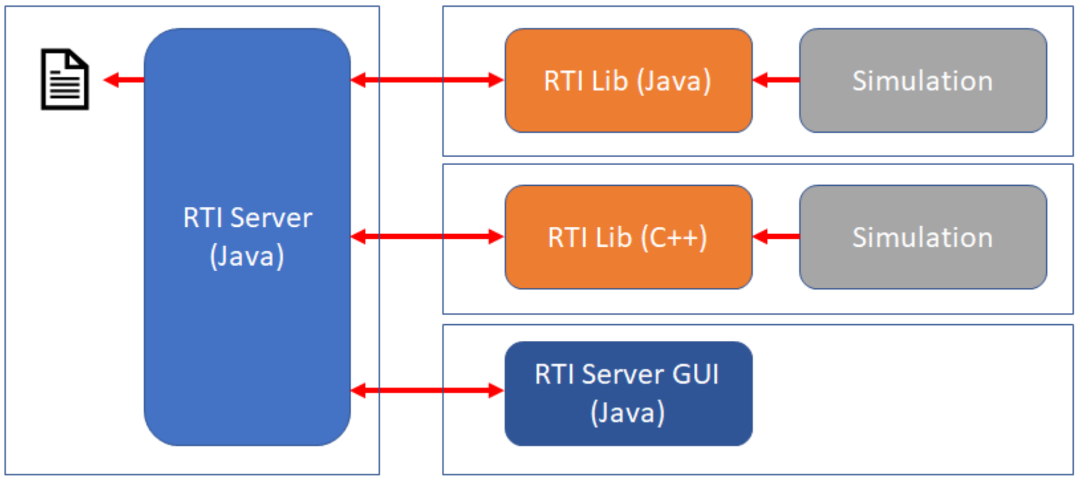


Figure 1 - High-Level Concept of SRTI v1.00.00

There exist three important parts to an SRTI system:

1. *RTI Server*: a shared node all simulations connect to in order to share and receive data. The RTIServer exists as its own application, and the user must start the RTI Server before executing simulations that try to connect to it (otherwise, they will not be able to connect to something that doesn't exist). Additionally, after starting the RTI Server, it will print out the "hostname" and "portnumber" it is connected to, both of which are required to be referenced by the simulation to connect to it. RTI Server is provided in the compiled SRTI.
2. *RTI Lib*: an API library that must be used by the simulation to connect to the RTI Server. It contains functions to connect, disconnect, publish, and receive messages. The details of socket communication, JSON creation and parsing is handled within RTI Lib, such that the simulation does not need to know understand anything beyond the provided API. RTI Lib is provided in the compiled SRTI.
3. *Simulation:* the simulation provided by the user. The user must modify the simulation to reference the RTI Lib and its API as required. Additionally, the user must be aware of the

simulation system and what data to expect, and how to use said data if it is received – the details of a simulation system is blind to the SRTI, it only acts as the gateway to allow communication.

Additionally, there is a fourth optional part to the SRTI system:

4. *RTI Server GUI*: a graphical user interface that automatically opens when running RTI Server. It provides a visual window that describes the "hostname" and "portnumber," as well as a live feed for connected simulations and data messages received by the RTI Server, as well as the ability to print out the simulation system's history to a file for separate analysis.

When updating a simulation to connect to a RTI Server, the code that the user must add will be meant to do the following (through the RTI Lib):

1. Connect to the RTI Server.
2. Subscribe to specific messages by name.
3. Publish messages by name (use RTI Lib to generate the message and set its content).
4. Handle when and how to receive messages and how to handle received data (use RTI Lib to parse out data).
5. Disconnect from the RTI Server.

When receiving messages from the SRTI, the message will be a JSON string with five parts:

1. "name": the name of the message.
2. "content": the actual data the simulation wants to send or receive.
3. "timestamp": set by the RTILib at the time the message is sent, represents the system time in milliseconds, could be used to handle message ordering or track delay amount.
4. "source": the name of the simulation that sent the message. Could be used to handle messages based on where they came from, to prevent using messages sent by itself, etc.
5. "tcp": either "true" or "false," confirms to receiving end whether they are expecting a confirmation response that the message was received.

It's important to know that all data received and sent by the SRTI is in "string" format, which both allows the flexibility of the system and causes it to be slower than other systems. As such, the user will have to convert numbers and other data to string format and will have to convert such content from a string to a number when necessary. This type of conversion is commonly available in modern programming languages.

In addition to messages defined by the simulations, the RTIServer will send SRTI-specific messages with generic information to inform each simulation of system changes.

To start the RTIServer, you can double-click on the executable .jar file containing the RTILib API. To set certain features on the RTIServer, such as enabling the GUI, explicitly setting the port number to open a socket connection, or enabling tcp (confirming messages have been received), include a "*settings.txt*" file in the same directory as the .jar file. The *settings.txt* file is of JSON format and looks like the below diagram. These settings much be set individually through API calls to enable whether applicable for a simulation.

Example "settings.txt" file:
 	
{
"portNumber":4200,
"guiOn":true,
"debugGuiOn":false,
"tcpOn":false,
"oldMessageLimit":100000,
"oldMessageArchive":false,
"debugConsole":false,
"debugFile":true,
"concurrentProcessing": true,
"oldMessageGUILimit": 10,
"compressOn":false
"retryConnection":false
}
	"portNumber"	Integer. explicitly sets socket connection on port if available. Use -1 to use randomly-available port.

	"guiOn"	true or false. Determines whether to display the main "RTI Server GUI".
	"debugGuiOn"	true or false. Determines whether to display "Debug GUI".
	"tcpOn"	true or false. Determines whether messages from RTI Server must receive confirmation from simulations. SRTI already uses TCP Socket connections, but setting "true" here adds extra safety logic to confirm messages are being received.
	"oldMessageLimit"	Integer. Sets limit to how many past messages are saved in memory (good to control memory limit), messages are thrown away periodically in RTI Server's memory when limit is reached. Number represents character number, not message number. Use -1 to use infinite limit.
	"oldMessageArchive"	true or false. Determines whether to save past messages to files after limit is reached. If true, then these can still be referenced to send old messages.
	"debugConsole"	true or false. Determines whether to print debug output to console window.
	"debugFile"	true or false. Determines whether to print debug output to file for analysis after execution.
	"concurrentProcessing"	true or false. Experimental feature. If false, insists that threads wait for each other before accessing same function. Decided to be unnecessary (shared-access bugs have been fixed where necessary).
	"oldMessageGUILimit"	Integer. Sets limit of messages seen in the GUI, prevents memory leak and makes easier to read only most recent messages.
	"compressOn"	true or false. Experimental feature, currently does nothing.
	"retryConnection"	true or false. Experimental feature, currently does nothing.

iv. Core Concepts – SRTI v2.00.00

Figure 2 shows an example of what an SRTI system looks like using the SRTI v2.00.00 design.

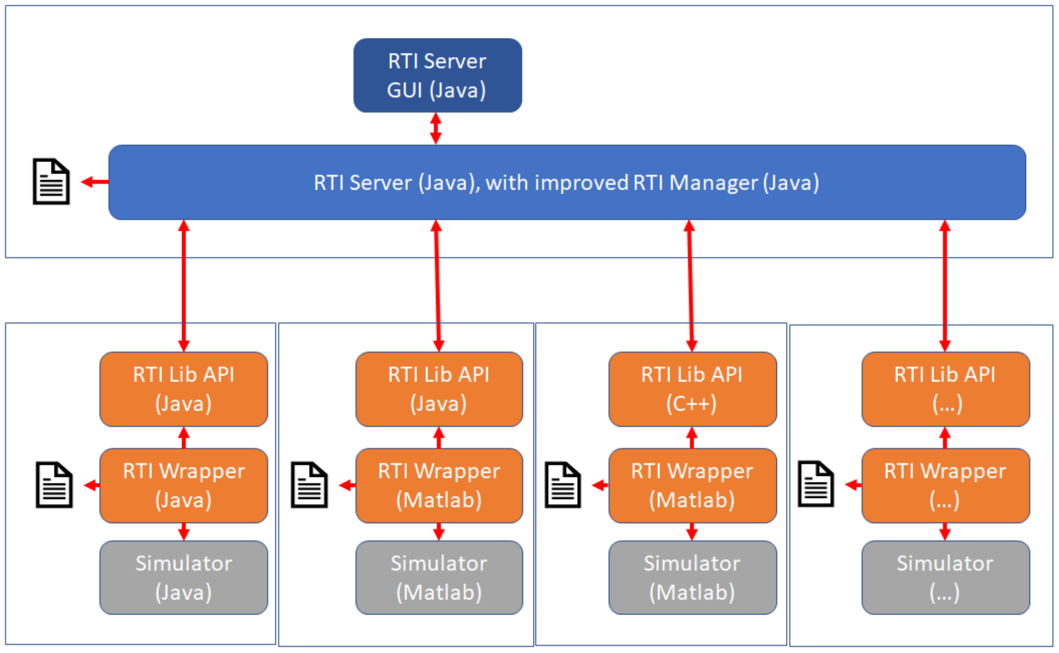


Figure 2 - High-Level Concept of SRTI v2.00.00

Notice that there is now an inclusion of a “RTI Wrapper” with every User Simulator. The Simulator itself doesn’t have explicit dependencies on any element of the SRTI. Instead, the RTI Wrapper is dependent on the Simulator, and also on the RTI Lib. RTI Lib is still used as the method of communication to the RTI Server. The RTI Server still uses an optional “settings.txt” file to enable certain features. Like SRTI v1.00.00, all of these components can be pre-compiled (in Java) without the Simulator for easy use.

But how does the RTI Wrapper know how to call upon a Simulator that was not defined at the time the RTI Wrapper was written? The RTI Wrapper relies on external configuration files. Specifically, there are 3 different files: “settings.json,” “global_def.json,” and “config.json.” These are each JSON-formatted files. “settings.json” only has two variables: “global” and “configuration,” each to define the file name of the other two files, to more easily allow the other files to have any name (“settings.json” must have this exact name and be present in the same directory the RTI Wrapper is launched from, otherwise the RTI Wrapper won’t find it). “global_def.json” is meant to define the message and variable types used in the simulation system, but implementation has determined this file was not necessary for a Wrapper in Java or Matlab.

“config.json” defines a Simulator and its relation to the simulation system as a whole. It still uses the concept of “publishing and subscribing to” messages to separate dependencies on specific simulators. To relate this to a Simulator, the Simulator must have global public variables that the RTI Wrapper can read and write from, either passing data from a message in the SRTI, or obtaining data to create a message to pass towards the SRTI. Similarly, each Simulator must have public functions that can be accessed to “Simulate” one timestep: there can be multiple functions, but the function should return “void” and have no input parameters. Other features the “config.json” file can represent include defining order of simulator execution, variable time unit advances, and different stages of simulation (where different messages or different simulators are involved at a specific point). The design of the RTI Wrapper and the “config.json” file is entirely dependent of other simulators that might be in the system, and it is assumed there are no “system-wide” parameters: for example conditions to “end the system” cannot be dependent on two conditions from different simulators being met; conditions must be able to be met and fulfilled within a single simulator. This better enforces that simulators are not directly dependent on each other.

These features are listed in the below table. Generally, modifications and additions to this file can be made without affecting the format of existing features, but there may be cases where a new feature design requires a complete re-format of the file structure.

v. Simulator Configuration File Format – SRTI v2.00.00

Key	Value Example	Value Type	Definition
hostname	"localhost"	String	The host name of the RTI Server this simulator is trying to connect to.
portNumber	"4200"	String	The port number of the RTI Server this simulator is trying to connect to.
simulatorName	"SomeSim"	String	Cosmetic name for simulator, to read in 'debug files' or to differentiate on RTI Server side.
simulatorRef	"Package.SomeSim"	String	Name of simulator file/reference. If inside 'package' (Java), full path must be provided.
debugConsole	true	Boolean	Whether or not to print out debug info to the console window.
debugFile	false	Boolean	Whether or not to print out debug info to a file (not supported in Matlab).
dataOutFile	true	Boolean	Whether or not to print out file tracing variable values during the time of the simulation (not supported in Matlab, to be updated).
stageChannels		JSON array	An array of objects that contain variables that partially define this simulator, at different 'stages.'
stage	0	Int, [>=0]	Allows stating that these variables are only true for specified stage. Stage = 0 by default.
order	0	Int, [>=0]	Specifies order this simulator should run in in relation to others during this stage, if not all simulators should be parallel. For example, "order = 0" should run first, "order = 1" should run second, etc.
timestepDelta	1	Int, [>= 1]	Specifies how many steps should occur before this simulator runs again. For example, 2 means this simulator would not run again until 2 vTimesteps later.
timestepVarDelta	"nextTimestep"	String	Alternative to setting "timestepDelta": if "timestepVarDelta" != "", Wrapper will read int variable from simulator to decide next time to run.
timestepMul	1	Int, [>=0]	If using simulators that each have different internal concept of time, use "timestepMul" to update one simulator to have same timestep as others, without affecting internal logic.
initializeChannels		JSON array	Lists public functions inside simulator that should be called at the beginning of each stage.
functionName	"initializeNow"	String	Name of public function inside simulator.
stage	0	Int, [>=0]	Stage where this function should occur.
simulateChannels		JSON array	Lists public functions inside simulator that should be called during each timestep of a specific stage.
functionName	"simulateStep"	String	Name of public function inside simulator.
timestepDelta	1	Int, [>= 1, mul. of stageChannels . timestepMul]	Specifies that this should occur every n steps. To avoid unexpected behavior, it is recommended that this is a multiple of "stageChannels.timestepMul".
stage	0	Int, [>= 0]	Stage where this function should occur.
subscribedChannels		JSON array	List messages this simulator should be subscribed to.
messageName	"messageA"	String	Name of the message.
onetime	False	Boolean	Should this simulator be subscribed to this message only at the beginning of the stage?
varChannel		JSON array	List JSON variables that need to be copied to the simulator's public variables.
valueName	"A_time"	String	Name of the key in the JSON message.
varName	"time_var"	String	Name of the public variable in the simulator.

mandatory	true	Boolean	Indicates if the simulator cannot finish a step before receiving this message. * Notice that this value does not exist in ‘publishedChannels’ or ‘simulateChannels’: without any way to indicate that these shouldn’t occur at a given moment, it is assumed they will occur at a consistent frequency within a given stage.
relativeOrder	0	Int, [0, 1, 2, 3]	Internal code to describe what message to retrieve, based on timing. 0 = get oldest message of this name. 1 = get newest message of this name, remove all others from queue. 2 = get newest message of this name less than (vTimestep + maxTimestep), remove older than this from queue. 3 = get newest message of this name greater than (vTimestep + maxTimestep), remove older than this from queue.
maxTimestep	-1	Int, [-1, 0, 1]	Alone with “relativeOrder,” specifies what message (based on timestep) should be retrieved. Example: -1 means get message from previous timestep, 0 = get message from current timestep, etc.
timestepDelta	1	Int	Specifies that this should occur every n steps. To avoid unexpected behavior, it is recommended that this is a multiple of “stageChannels.timestepMul”.
stage	0	Int	Stage where this subscription should occur.
publishedChannels		JSON array	List messages this simulator should publish to.
messageName	“messageSomeSim”	String	Name of the message.
initial	true	Boolean	Should an ‘initial’ version of this message be published at the very beginning, to help initialize other simulators?
varChannel		JSON array	List JSON variables that need to be copied from the simulator’s public variables.
valueName	“Sim_Time”	String	Name of the key in the JSON message.
varName	“newTime”	String	Name of the public variable in the simulator.
timestepDelta	1	Int	Specifies that this should occur every n steps. To avoid unexpected behavior, it is recommended that this is a multiple of “stageChannels.timestepMul”.
stage	0	Int	Stage where this publishing should occur.
endConditions		JSON array	List of conditions that will cause the entire simulation system to end (can only reference variables inside this simulator or Wrapper instance).
(array)		JSON array (at least 2 elements)	Why is “endConditions” an array of arrays? The internal arrays represent conditions that should ALL be met to end the system. The external arrays represent conditions where one OR another can be met to end the system. Why at least 2 elements? While not an issue for all languages, Matlab automatically simplifies the data type, which may misread how to check a condition is met. To get around this, the internal array(s) must have at least 2 elements, even if the user just duplicates the first element a second time. This is only required if a second OR condition exists in the list.
varName	“outputValue”	String	Name of public variable in the simulator.
condition	“>=”	String [“=”, “>”, “<”, “>=”, “<=”, “!=”]	Condition to be met.

value	100	String / Int / Float / Booolean / Other	Value to compare "varName" against.
varName2	""	String	Name of variable inside simulator to compare "varName" against. If set to "", it will be ignored, and "value" will be used instead. Otherwise, this allows the user to compare two dynamic values inside their own simulator, instead of relying on a strict hardcoded value in this file.
stageConditions		JSON array	List of conditions that will cause the entire simulation system to switch to a different stage.
(array)		JSON array (at least 2 elements)	Why is "endConditions" an array of arrays? The internal arrays represent conditions that should ALL be met to end the system. The external arrays represent conditions where one OR another can be met to end the system. Why at least 2 elements? While not an issue for all languages, Matlab automatically simplifies the data type, which may misread how to check a condition is met. To get around this, the internal array(s) must have at least 2 elements, even if the user just duplicates the first element a second time. This is only required if a second OR condition exists in the list.
oldStage	0	Int	Stage to check this condition at.
varName	"condition_var"	String	Name of public variable in the simulator.
condition	"="	String ["=", ">", "<", ">=", "<=", "!="]	Condition to be met.
value	"start"	String / Int / Float / Booolean / Other	Value to compare "varName" against.
varName2	""	String	Name of variable inside simulator to compare "varName" against. If set to "", it will be ignored, and "value" will be used instead. Otherwise, this allows the user to compare two dynamic values inside their own simulator, instead of relying on a strict hardcoded value in this file.
newStage	1	Int	New stage to transition to.

Example "config.json" file:

```
{
   "hostName": "localhost",
   "portNumber": "42010",
   "simulatorName": "A_Sim",
   "simulatorRef": "airport_20190104.A",
   "debugConsole":true,
   "debugFile":true,
   "dataOutFile":true,
   "initializeChannels":[
   ],
   "stageChannels":[
            {
                  "stage": 0,
                  "order": 0,
                    "timestepDelta": 1,
                    "timestepVarDelta": "",
                    "timestepMul": 1
            }
    ],
    "simulateChannels":[
            {
                    "functionName": "Simulate",
                    "timestepDelta": 1,
                    "stage": 0
            }
    ],
    "subscribedChannels": [
      {
          "messageName": "D",
          "oneTime": false,
          "varChannel": [
            {
                    "valueName": "D_numberOfPeople",
                     "varName": "changeInPeople"
            }
          ],
         "mandatory": true,
         "relativeOrder": 0,
         "maxTimestep": 0,
         "timestepDelta": 1,
         "stage": 0
      }
    ],
    "publishedChannels": [
    ],
    "endConditions": [
            [
               {
                    "varName": "people",
                    "condition": "<=",
                    "value": 0
                    “varName2”:””
               }
            ]
    ],
    "stageConditions": [
    ]
 }
```

The design of the RTI Wrapper is meant to be expanded further with a GUI application to help define “config.json” and to help launch a command in a single click. This GUI and any other features not described in this document are still under development.

SRTI v1.00.00 was originally developed in **Java** and **C++**, with the expectation that most languages would be able to load its .jar or .dll libraries as required (or else, the RTI Lib API was simple enough to rewrite in other languages as required). Generally, this type of readability isn’t possible the other way around, so RTI Wrapper has a significant limitation: the Wrapper and the

Simulator must be written in the same language. Additionally, the Simulator must have public global variables to represent data to read/write from messages in the SRTI, and must have public functions to represent executing a single simulation step within that simulator. And while SRTI v1.00.00 can support objects of different formats being contained in a JSON message, SRTI v2.00.00 further enforces the message be strictly JSON – a Simulator would need to have a String variable to represent the full proprietary format they want to use to get around this, but this would also defeat the purpose of other Simulators being able to utilize the data.

At the time of this writing, RTI Wrapper's for **Java**, **Matlab,** and **NetLogo 6.1** have been made available.

DISCLAIMER: Versions of SRTI prior to v2.16.02 were unable to share 1-dimensional and multi-dimensional arrays between simulators of different languages. As of v.2.16.02, the SRTI Wrappers for Java, Matlab and NetLogo are able to translate arrays (Java), matrices (Matlab) and Lists (NetLogo) of multiple dimensions to JSON arrays to send through the SRTI, and parse back to the corresponding data type.

Additionally, suppose you want to pass in specific values in use by the RTI Wrapper to a simulator? For example, virtual timestep ("0", "1", "2", etc.)?

It is highly recommended to not depend on these types of values inside a simulator in your design, as it makes it more difficult to adapt to different system designs from another user. But if it is necessary, you can pass "vTimestep," "stage," and two versions of "stageVTimestep."

For "subscribeChannels"	
Set "messageName" to "**RTI_**".	
Set "varChannels"."valueName" to…	
"**vTimestep**"	Integer. Returns virtual timestep for simulation system, which is persistent, even when transitioning between stages.
"**stage**"	Integer. Returns current stage of simulation system, as referenced inside "RTI_StartStep" message.
"**stageVTimestepMul**"	Integer. Returns virtual timestep for simulation system, which is updated by "+ (timestepMul * timestepDelta)" at each step, to correctly correlate with other simulators in the system. This is not persistent across stage transitions (at beginning of every stage, this value is 0).
"**stageVTimestep**"	Integer. Returns virtual timestep for simulation system, NOT updated in the way "stageVTimestepMul" is; this is typically

	updated by “+ 1” at each step. A more accurate representation of how a simulator might internally understand the concept of time.

For “endConditions” and “stageConditions”	
Set “varName” to…	
“RTI_**vTimestep**”	Integer. Returns virtual timestep for simulation system, which is persistent, even when transitioning between stages.
“RTI_**stage**”	Integer. Returns current stage of simulation system, as referenced inside “RTI_StartStep” message.
“RTI_**stageVTimestepMul**”	Integer. Returns virtual timestep for simulation system, which is updated by “+ (timestepMul * timestepDelta)” at each step, to correctly correlate with other simulators in the system. This is not persistent across stage transitions (at beginning of every stage, this value is 0).
“RTI_**stageVTimestep**”	Integer. Returns virtual timestep for simulation system, NOT updated in the way “stageVTimestepMul” is; this is typically updated by “+ 1” at each step. A more accurate representation of how a simulator might internally understand the concept of time.

Additionally, you can choose to subscribe to proprietary messages being sent automatically by the RTI Server to the RTI Wrapper, some of which will include this type of information. This would allow you to use the standard format of subscribing to a message by name, and a variable inside “content” by name. Additional information about internal proprietary messages (“RTI_...”) can be found in Chapter 1, Section vi, “Detailed Concepts (for Developers).”

vi. Detailed Concepts (for Developers) (v2.00.00)

Figure 2 gave a high-level view of how the SRTI is organized, and how different components can be separated to different machines. Figure 3 gives a more detailed view, including direct references to class names inside the SRTI code.

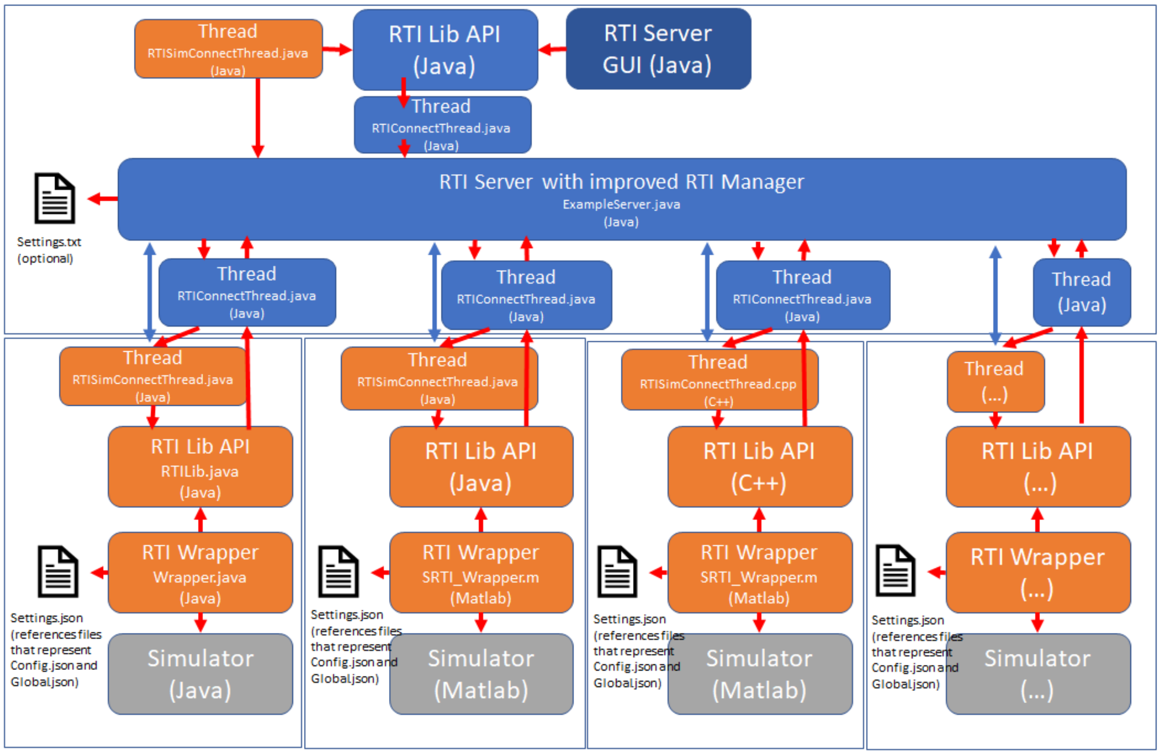

Thread RTISimConnectThread.java (Java)
RTI Lib API (Java)
RTI Server GUI (Java)
Thread RTIConnectThread.java (Java)
RTI Server with improved RTI Manager ExampleServer.java (Java)
Settings.txt (optional)
Thread RTIConnectThread.java (Java)
Thread RTIConnectThread.java (Java)
Thread RTIConnectThread.java (Java)
Thread (Java)
Thread RTISimConnectThread.java (Java)
Thread RTISimConnectThread.java (Java)
Thread RTISimConnectThread.cpp (C++)
Thread (...)
RTI Lib API RTILib.java (Java)
RTI Lib API (Java)
RTI Lib API (C++)
RTI Lib API (...)
RTI Wrapper Wrapper.java (Java)
RTI Wrapper SRTI_Wrapper.m (Matlab)
RTI Wrapper SRTI_Wrapper.m (Matlab)
RTI Wrapper (...)
Settings.json (references files that represent Config.json and Global.json)
Simulator (Java)
Simulator (Matlab)
Simulator (Matlab)
Simulator (...)

Figure 3 - Detailed view of SRTI (code organization)

Figure 3 is more accurate to how different threads are created and managed, and how communication occurs between them. Like SRTI v1.00.00, the RTI Lib API is a crucial part of how the RTI Server and a simulator (or in this case, the RTI Server and a RTI Wrapper) communicate with each other. RTI Lib API will use TCP Socket communication to connect with the RTI Server, a process that can allow both local system and Internet-connected system data transfer. But like both 2-way socket examples in programming, this requires each side use 2 socket ports: one for listening to the other side, and one for speaking to the other side.

At the beginning, RTI Lib API (“RTILib”) will create a thread (a class called “RTISimConnectThread”) specifically designed to receive messages from the RTI Server. At the VERY beginning, the thread will attempt to connect to a single “hostname” and “portnumber” address: when the RTI Server (“ExampleServer”) is opened, it automatically opens a single socket for this purpose. The “hostname” will always be regulated by the local computer, and whether it is connected to the Internet with a static IP (it may change irregularly based on a local Internet router or service provider). The “portnumber” is normally automatically assigned by the local system, and can change with each instance the RTI Server opens. For consistency, the user can specify a preferred port number inside “settings.txt” (in the same directory as the SRTI .jar file), and the RTI Server will attempt to use that port number first, if no other process is using it. When RTISimConnectThread successfully connects to the single connection port from RTI Server, RTI Server creates a new instance of RTIConnectThread with a new socket, and using the main socket, tells RTISimConnectThread to close its previous connection and to open a new connection to a new dedicated port number. This finalizes a dedicated connection between the RTI Server and RTI Lib API; each simulator-server connection would be using 2 socket ports and 2 threads (not including main process thread of RTI Lib API or RTI Server, which would mean 3 threads + 1 shared thread across all sims) during regular use.

While the RTI Server GUI is part of the .jar file containing RTI Server, it itself is also treated as its own simulator, for testing purposes to confirm messages can be sent and received. The RTI Server will have a single line to open an instance of RTI Server GUI before letting it act independently. Note that closing the RTI Server GUI will also automatically close the RTI Server, unless a command line interface was used to open the RTI Server, or unless it was opened as a background process with no ties to any foreground interface.

Messages being sent and received through SRTI are JSON strings (key-value pairs, where any variable can be accessed by its name). RTI Lib API contains functions to help build a JSON message, but even without their use, it will construct a JSON message using the content provided by a user at the time it is asked to publish. The “content” can be of any format, giving the user complete freedom (and responsibility to being compatible with other simulators) without the need to pre-compile or pre-define messages. The “name” of the message differentiates different types of messages: a simulator can subscribe to or publish messages based on its “name.” The format of a message can be seen below. Unless otherwise updated, it is assumed all of these values are formatted and sent as “string” values, even if it could be simplified to other data types.

“name”	Defined by user.	User sets this when they call “publish”
“content”	Defined by user.	User sets this before they call “publish”
“isTcp”	Defined by RTI Lib API.	User sets this as true or false using settings.txt (Server side) or in code (Sim side). It adds additional logic to require confirmation from a receiver (Server or Sim) that a sent message was received, else it will be sent again up to 3 times. SRTI already uses real TCP sockets, so this value’s name is misleading.
“timestep”	Defined by RTI Lib API.	System timestep at time a message was originally sent.
“vTimestep”	Defined by RTI Lib API.	NEW VALUE FOR SRTI v2.00.00. Defines a virtual-timestep value of system execution (“1”, “2”, etc.).

“source”	Defined by RTI Lib API.	Sim name (or name of Server) that first sends a message. User can set this at beginning, before connecting to RTI Server.

When a Simulator (or Wrapper) subscribes to a message (only once – usually near beginning, after connecting to RTI Server), the RTI Lib API will receive and retain all future submissions of that message. The Simulator (or Wrapper) would have to manually call RTI Lib API to retrieve a message from its buffer. After retrieving a message from RTI Lib API, it is removed from its buffer. Failure to do this for messages sent often may be a form of a memory leak. Likewise, there is no explicit method for the RTI Lib API to tell the RTI Server to resend something: if the message was ever sent, the RTI Lib API should already have it.

There are a few “proprietary” messages that are sent internally from the RTI Server and the RTI Lib API to organize themselves. Their names and content is usually irrelevant for a user, but might be helpful for understanding bugs, or if trying to built RTI Lib API in a different language. Some of these messages include:

“name”: ”RTI_InitializeSim”	“content”: (includes simulator name)	Sent during initial connection process from RTI Lib.
“name”: ”RTI_SubscribeToMessagePlusLatest”	“content”: “{“subscribeTo”: “somemessagename”}”	Sent during reconnection or “subscribeToMessagePlusLatest” process. Tells RTI Server to immediately resend the most recent version of a message back to RTI Lib.
“name”: ”RTI_SubscribeToMessagePlusHistory”	“content”: “{“subscribeTo”: “somemessagename”}”	Sent during reconnection or “subscribeToMessagePlusLatest” process. Tells RTI Server to immediately resend ALL messages of a type to RTI Lib.
“name”:”RTI_SubscribeTo”	“content”: “{“subscribeTo”: “somemessagename”}”	Sent after connection is successful, to tell RTI Server to forward any future messages of type “somemessagename.”
“name”: ”RTI_SubscribeToAll”	“content”: “”	Sent after connection is successful, to tell RTI Server to forward all future messages.
“name”: “RTI_SubscribeToAllPlusHistory”	“content”: “”	See “RTI_SubscribeToAll” and “RTI_SubscribeToMessagePlusHistory.”
“name”: “RTI_ReceivedMessage”	“content”: (full copy of some message)	Used if “isTcp” is true to confirm a message was received.

The RTI Lib API has a full list of functions included in the Documentation file for SRTI_v01_00_00. Since the RTI Wrapper handles all calls to these automatically in v2.00.00, this list has been excluded from this document.

The RTI Wrapper is a relatively new concept in the SRTI, built on the idea that many of the steps required to connect to and subscribe/publish messages remains the same for most users. It also makes explicit use of "vTimestep," which required a minor redesign of the RTI Server to better control and synchronize each connected simulator. The concept of "stages" and "order of execution" is also handled by the smarter RTI Server. These changes make use of the older RTI Server obsolete with the RTI Wrapper.

A general overview of the pattern RTI Wrapper uses is as follows:

1. Connect to open RTI Server.
2. Subscribe to specified messages (plus history).
3. Wait for "RTI_StartStep" from RTI Server (step 0).
4. For any messages that need to be received right away, get.
5. For any functions that need to be called at the beginning, initialize.
6. For any messages that need to be published right away, publish.
7. Initialize variables now based on being in stage "0".
8. Send "RTI_FinishStep".
9. Start main loop:
   - 9.1. Wait to receive "RTI_StartStep" from RTI Server.
   - 9.2. If content in "RTI_StartStep" show change in stage, then repeat steps 3-6, update current step, send "RTI_FinishStep," go back to 9.1. If new stage is -1, then break the loop and disconnect from RTI Server.
   - 9.3. Subscribe to messages for this stage and step.
   - 9.4. Simulate as required for this stage and step.
   - 9.5. Publish as required for this stage and step.
   - 9.6. Check if any "end conditions" are met (if so, tell RTI Server to stop system-wide execution of simulators).
   - 9.7. Check if any "stage conditions" are met (if so, tell RTI Server to change stage at next step).

On the RTI Server side, it follows the following pattern:

1. Wait for "RTI_StartSim", content empty (sent by RTI Server GUI when button is clicked).
2. Tell all simulators "RTI_StartStep," set that it is expecting "RTI_FinishStep" from each.
3. For each received "RTI_FinishStep," observe message content to determine next time (step and order) it is supposed to send "RTI_StartStep" again for that specific simulator. If all expected "RTI_FinishStep" messages have been received, check for smallest step and order

for connected simulators to determine who needs to execute next, and send “RTI_StartStep” to them again.

4. Repeat step 3.

In theory, one would write a “custom” Wrapper in the programming language of their choice if they follow a similar design to how the official Wrapper does, waiting for “RTI_StartStep” and sending “RTI_FinishStep” at the correct moments.

There are a handful of new proprietary messages that are required for this new design. Those are listed below:

“name”: ”RTI_StartSim”	“content”: “”	Tells RTI Server to start all the simulators (for step 0).
“name”: ”RTI_StartStep”	“content”: “{“timestep”: “(some vTimestep)”, “stage”: “(some stage number)”}”	Sent by RTI Server to start simulator for specific step.
“name”: ”RTI_FinishStep”	“content”: “{“nextStep”: “(some future vTimestep)”, “nextOrder”: “(some future order)”}”	Sent by RTI Wrapper to inform RTI Server that it has finished one step, and to tell when it should be called again to execute a new step.
“name”:”RTI_EndSystem”	“content”: “”	Sent by RTI Wrapper to inform RTI Server to set stage = -1 for next step, would tell all simulators to close at next step (but not actually close the RTI Server).
“name”: ”RTI_UpdateStage”	“content”: “{“nextStage”: “(some stage number)”}”	Sent by RTI Wrapper to inform RTI Server of a stage change, will occur at next time “vTimestep” needs to increase.
“name”: “RTI_PauseSystem”	“content”: “”	Tells RTI Server to pause an executing system, to not send “RTI_StartStep” to any open simulators after they’ve returned “RTI_FinishStep.”
“name”: “RTI_ResumeSystem”	“content”: “”	Tells RTI Server to resume a paused executing system, sending “RTI_StartStep” to any simulators that had finished their previous step.

It is not by magic that the RTI Wrapper is able to call upon the functions and variables of a simulator that does not exist at the time the RTI Wrapper was programmed. Using variables specified in a “config.json” file (see Chapter 1, part v.), it must utilize some form of dynamic referencing, either through a design pattern like “reflection” (Java, C#, Python) or dynamic function calls (“eval()” in Matlab). For more strict languages like C++, the Wrapper would have to be explicitly programmed to properly call functions and variables by name: one prototype solution produced internally was to write a program to generate a C++ Wrapper after reading the config.json file, having the user compile the new Wrapper, and utilizing that for the specific simulator.

In addition to limiting the number of languages that can easily use a pre-compiled RTI Wrapper, a Wrapper cannot call upon another simulator of any language. While some languages can support access to code written in different languages, that has to be explicitly set. So without knowing the language in advance, a strict requirement is made that any RTI Wrapper must be written in the same language as the simulator in question. A Java Wrapper must be used with a Java simulator, but cannot be used with a Matlab simulator, for example. The SRTI NetLogo Wrapper is an exception: the Wrapper is written in Java, but utilizes NetLogo libraries to be able to call back to a NetLogo simulator.

Because of these limitations, there is still value in using the design of SRTI v1.00.00, where a user would hardcode the use of functions from RTI Lib API in the manner they wish. But to be compatible with other simulators (and the updated RTI Server) in SRTI v2.00.00, the user must pay close attention to the new design pattern that involves “RTI_StartStep” and “RTI_FinishStep.”

Chapter 2 – User Example

This Chapter provides a series of examples for the user to follow, step by step. It is recommended the user attempt at least one of the examples; ideally, the example that most closely resembles their project.

i. Airport Example – Synopsis

This is an introduction to the simple example that following sections will build off of.

Suppose there exist four simulators. Their goal is to simulate a simple airport, with a single airplane that lands to take passengers, leaves, and returns again. The simulation system ends when the airport has no more passengers. (It might make more sense to imagine this as a train or bus station.)

A = Airport (contains number of waiting passengers)

B = Body of Plane (where a limited number of passengers get on)

C = Controls of the Plane (controls "autopilot" for plane to fly and come back)

D = Additional system-wide variable (to make results seem less trivial).

This type of system requires two distinct stages: one where passengers slowly get onto the plane, and a second that simulates the time it takes for a plane to reach a destination to let passengers off and return again. Figures 4 and 5 describe the two stages.

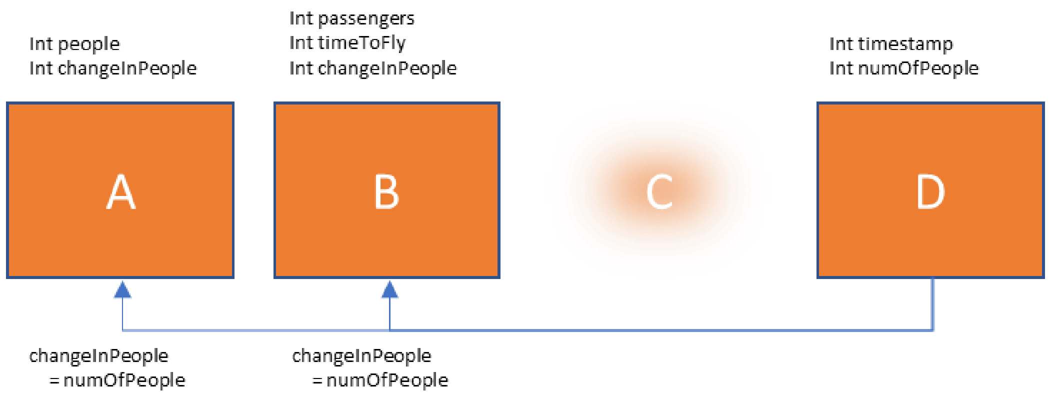


Figure 4 - Airport Example (Stage 1)

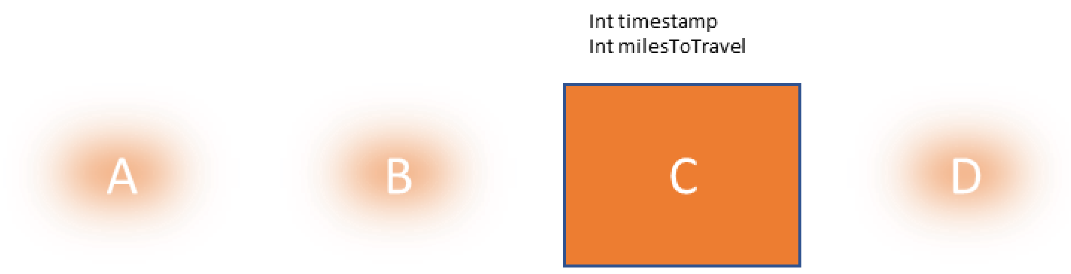


Figure 5 - Airport Example (Stage 2)

At each "timestep" of the simulation in Stage 1, A and B are supposed to update as the following:

A.people = A.people – A.changeInPeople;

B.passengers = B.passengers + B.changeInPeople;

B.timeToFly = B.timeToFly – 1;

If either of the following conditions are met, the system transitions from Stage 1 to Stage 2.

If (B.passengers >= 100)

If (B.timeToFly <= 0)

At each "timestep" of the simulation in Stage 2, C updates (at a different rate from Stage 1) as the following:

C.milesToTravel = C.milesToTravel – 30;

If the following condition is met, the system transitions from Stage 2 to Stage 1.

If (C.milesToTravel <= 0)

The simulation system should end execution if the following condition is met:

If (A.people <= 0)

If not using SRTI v2.00.00, a user could easily create a "Manager" class to handle the interactions between these separate simulators, to pass and utilize variables accordingly. Instead, we'll have the SRTI be the management system that synchronizes them and controls their update logic.

ii. Airport Example – Version 1 (Java)

Using the SRTI, let's suppose we want A, B and D to update at the same time each timestep.

The Java code for A.java, B.java, C.java, and D.java look like this:

```
package airport_20190104;

public class A {


     public int people = 450;
     public int changeInPeople = 0;

     public void Initialize() {
          people = 450;
     }
     public void Simulate() {
          people = people - changeInPeople;
     }
}
```

```
package airport_20190104;

public class B {

     public int passengers = 0;
     public int timeToFly = 30;
     public int changeInPeople = 0;

     public void Initialize() {
          passengers = 0;
          timeToFly = 30;
     }

     public void Simulate() {
          passengers = passengers + changeInPeople;
          timeToFly = timeToFly - 1;
     }
}
```

```
package airport_20190104;

public class C {

     public int milesToTravel = 200;
     public int timestamp = 0;

     public void Initialize() {
          milesToTravel = ((timestamp % 4) + 1) * 50;
     }

     public void Simulate() {
          milesToTravel = milesToTravel - 30;
     }
}
```

```
package airport_20190104;

public class D {

     public int timestamp = 0;
     public int numberOfPeople = 1;

     public void Initialize() {

     }

     public void Simulate() {
          if (timestamp % 50 > 25)
               numberOfPeople = timestamp % 9;
          else
               numberOfPeople = timestamp % 5;
     }
}
```

Step 1: Compile these Java simulators into a .jar file. You can choose to compile them into separate .jar files, but for this example, we place them in the same .jar file, called “airport.jar.”

Step 2: Download “SRTI_v2_00_00.jar” and “SRTI_Wrapper_v2_00_00.jar” (or whatever the most recent version of each file is). Prepare one directory to place “SRTI_v2_00_00.jar” in, and four directories (one for each simulator) to copy “SRTI_Wrapper_v2_00_00.jar” and “airport.jar” in. If desired, also copy the “settings.txt” file that “SRTI_v2_00_00.jar” uses, and place in the same directory as “SRTI_v2_00_00.jar.”

Step 3: For each directory that contains a simulator, we need to prepare three files: “settings.json,” a global.json file, and a config.json file. At this time, the ‘global.json’ file’s content doesn’t really matter, so just copy the example provided online with the SRTI files.

Let’s start with Sim A: name the .json files “Settings.json,” “global.json” and “configA.json.” The files should look like the following:

```
“Settings.json”

{
    "global": "global.json",
    "configuration": "configA.json"
}
```

“global.json”

```
{
    "Sum": {
        "value": "int"
    },
    "Difference": {
        "value": "int"
    }
}
```

“configA.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "A_Sim",
    "simulatorRef": "airport_20190104.A",
    “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "D",
             "oneTime": false,
             "varChannel": [
               {
                    "valueName": "D_numberOfPeople",
                    "varName": "changeInPeople"
               }
             ],
               "mandatory": true,
               "relativeOrder": 0,
               "maxTimestep": 0,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
          [
          {
               "varName": "people",
               "condition": "<=",
               "value": 0
          }
          ]
     ],
     "stageConditions": [
     ]
}
```

These files should be in the same directory as the Wrapper file and sim file. The following pages show how “configB.json,” “configC.json” and “configD.json” should look. Keep in mind that “settings.json” should also be updated to reference the correct config file each time.

Also notice “configD.json” shows D is subscribed to “RTI_StartStep.” This is one of multiple proprietary internal JSON messages the SRTI sends and receives between the Server and the Wrapper. Its content includes “timestep” and “stage.” This is a way to be able to get the “virtual timestep” (for example, “0,” “1,” “2,” etc.) from the system for a sim’s internal logic.

“configB.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "B_Sim",
    "simulatorRef": "airport_20190104.B",
    “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
          {
               "functionName": "Initialize",
               "stage": 0
          }
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "D",
             "oneTime": false,
             "varChannel": [
               {
                    "valueName": "D_numberOfPeople",
                    "varName": "changeInPeople"
               }
             ],
               "mandatory": true,
               "relativeOrder": 0,
               "maxTimestep": 0,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
     ],
     "stageConditions": [
          [
          {
               "oldStage": 0,
               "varName": "passengers",
               "condition": ">=",
               "value": 100,
               "newStage": 1
          }
          ],
          [
          {
               "oldStage": 0,
               "varName": "timeToFly",
               "condition": "<=",
               "value": 0,
               "newStage": 1
          }
          ]
     ]
}
```

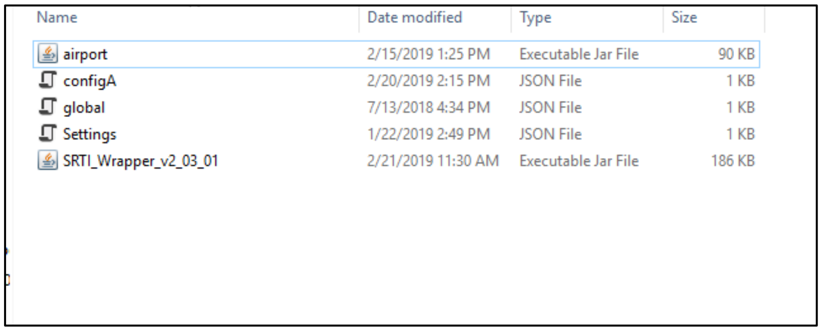
“configC.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "C_Sim",
    "simulatorRef": "airport_20190104.C",
    “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
          {
               "functionName": "Initialize",
               "stage": 1
          }
     ],
     "stageChannels":[
          {
               "stage": 1,
               "order": 0,
               "timestepDelta": 30,
               "timestepMul": 1
          },
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 30,
               "stage": 1
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "RTI_",
             "oneTime": false,
             "varChannel": [
               {
                    "valueName": "vTimestamp",
                    "varName": "timestamp"
               }
             ],
               "mandatory": true,
               "relativeOrder": 0,
               "maxTimestep": 0,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
     ],
     "stageConditions": [
          [
          {
               "oldStage": 1,
               "varName": "milesToTravel",
               "condition": "<=",
               "value": 0,
               "newStage": 0
          }
          ]
     ]
}
```

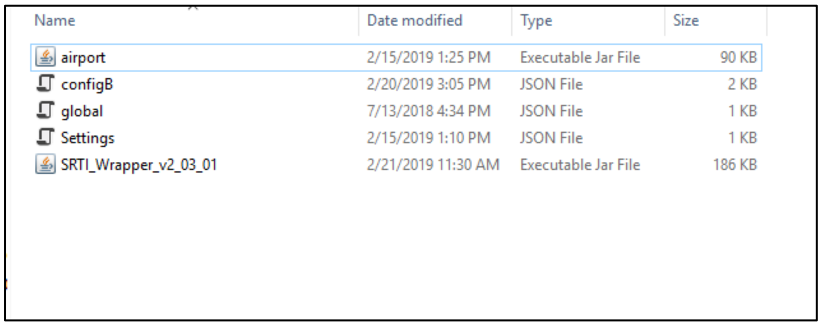
“configD.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "D_Sim",
    "simulatorRef": "airport_20190104.D",
     “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
          {
             "messageName": "RTI_",
            "oneTime": false,
            "varChannel": [
              {
                   "valueName": "vTimestamp",
                   "varName": "timestamp"
              }
            ],
              "mandatory": true,
              "relativeOrder": 0,
              "maxTimestep": 0,
              "timestepDelta": 1,
              "stage": 0
        }
    ],
    "publishedChannels": [
          {
               "messageName": "D",
            "initial": false,
            "varChannel": [
              {
                   "valueName": "D_numberOfPeople",
                        "varName": "numberOfPeople"
              }
            ],
              "relativeOrder": 0,
              "maxTimestep": 0,
              "timestepDelta": 1,
              "stage": 0
        }
    ],
     "endConditions": [
     ],
     "stageConditions": [
     ]
}
```

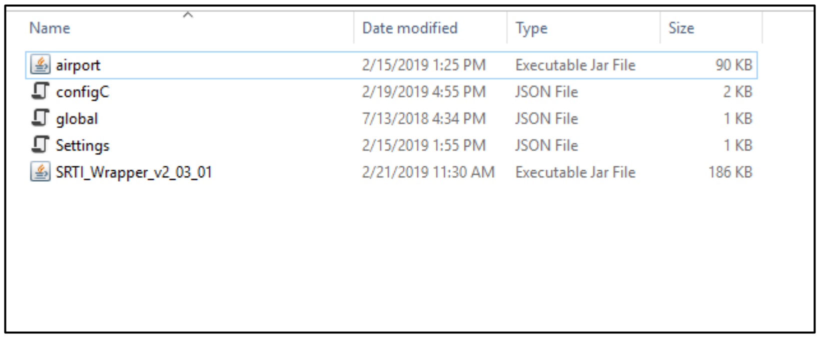
At this point, the file system on your computer (if running Windows OS) should look something like this:



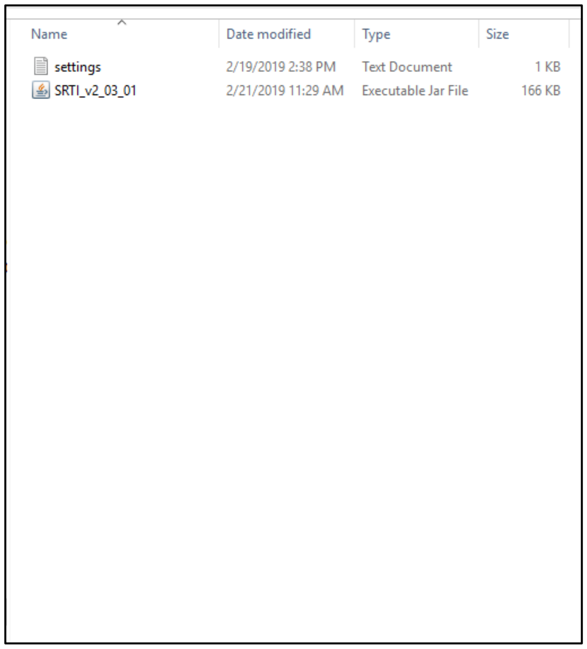
Name	Date modified	Type	Size
airport	2/15/2019 1:25 PM	Executable Jar File	90 KB
configA	2/20/2019 2:15 PM	JSON File	1 KB
global	7/13/2018 4:34 PM	JSON File	1 KB
Settings	1/22/2019 2:49 PM	JSON File	1 KB
SRTI_Wrapper_v2_03_01	2/21/2019 11:30 AM	Executable Jar File	186 KB



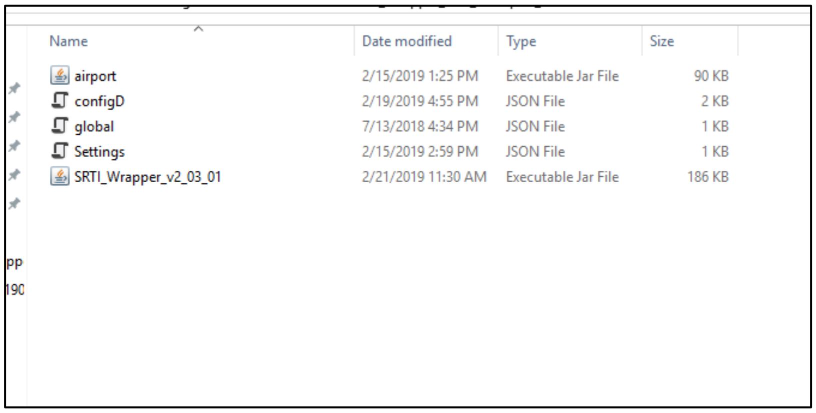
Name	Date modified	Type	Size
airport	2/15/2019 1:25 PM	Executable Jar File	90 KB
configB	2/20/2019 3:05 PM	JSON File	2 KB
global	7/13/2018 4:34 PM	JSON File	1 KB
Settings	2/15/2019 1:10 PM	JSON File	1 KB
SRTI_Wrapper_v2_03_01	2/21/2019 11:30 AM	Executable Jar File	186 KB



Name	Date modified	Type	Size
airport	2/15/2019 1:25 PM	Executable Jar File	90 KB
configC	2/19/2019 4:55 PM	JSON File	2 KB
global	7/13/2018 4:34 PM	JSON File	1 KB
Settings	2/15/2019 1:55 PM	JSON File	1 KB
SRTI_Wrapper_v2_03_01	2/21/2019 11:30 AM	Executable Jar File	186 KB



Name	Date modified	Type	Size
settings	2/19/2019 2:38 PM	Text Document	1 KB
SRTI_v2_03_01	2/21/2019 11:29 AM	Executable Jar File	166 KB



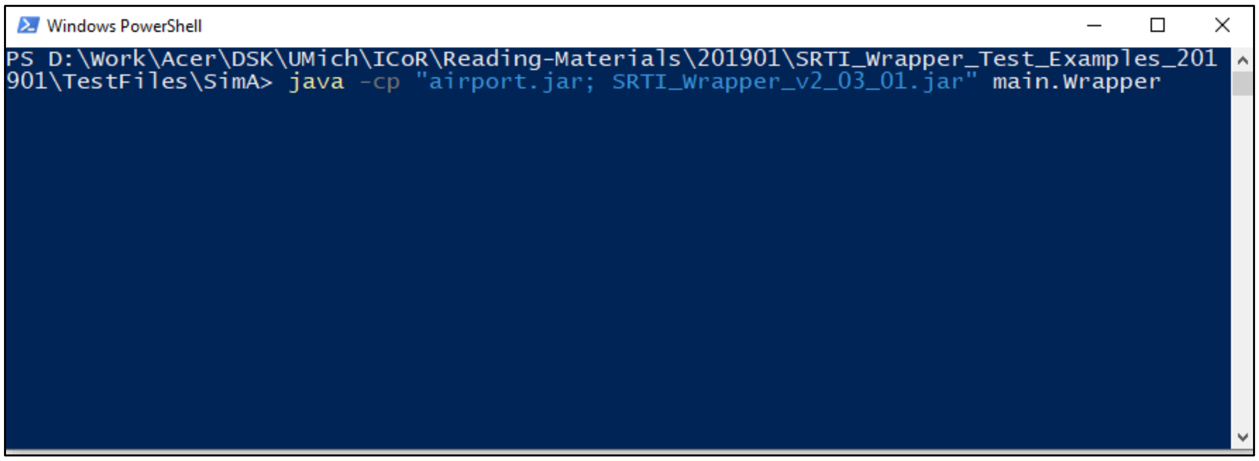
Name	Date modified	Type	Size
airport	2/15/2019 1:25 PM	Executable Jar File	90 KB
configD	2/19/2019 4:55 PM	JSON File	2 KB
global	7/13/2018 4:34 PM	JSON File	1 KB
Settings	2/15/2019 2:59 PM	JSON File	1 KB
SRTI_Wrapper_v2_03_01	2/21/2019 11:30 AM	Executable Jar File	186 KB

Step 4: Now that the files are prepared, we just have to run the system.

Start by running the RTI Server. It opens automatically when you double click on "SRTI_v2_00_00.jar."

Next, run each copy of the Wrapper separately. Unlike "SRTI_v2_00_00.jar," you cannot simply click on "SRTI_Wrapper_v2_00_00.jar" to run it. By itself, it does not know about "airport.jar," and would throw an error. We need to run the Wrapper with a reference to that dependency. To do that, open a command prompt at the location where each simulator is, and run the following Java command:

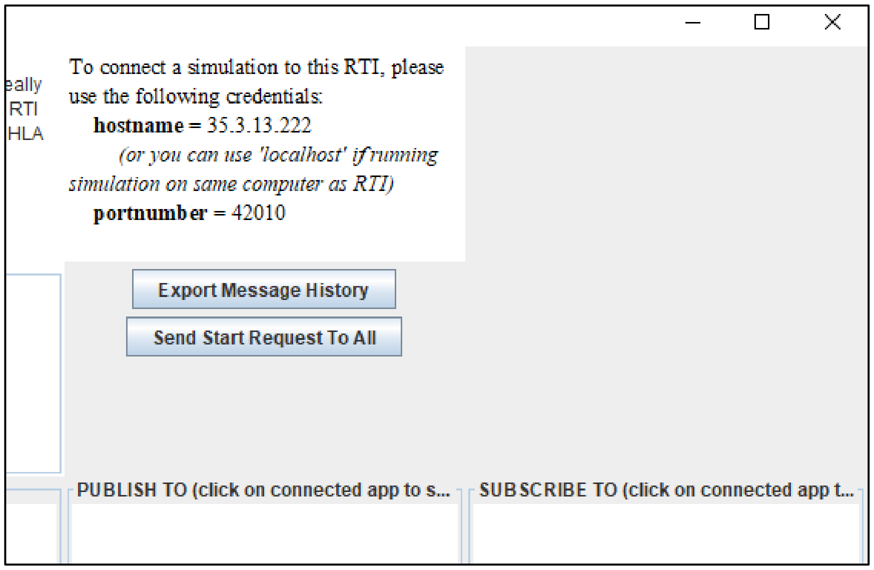
>> java -cp "airport.jar;SRTI_JavaWrapper_v2_20_02.jar" main.Wrapper



```
PS D:\Work\Acer\DSK\UMich\ICoR\Reading-Materials\201901\SRTI_Wrapper_Test_Examples_201
901\TestFiles\SimA> java -cp "airport.jar; SRTI_Wrapper_v2_03_01.jar" main.Wrapper
```

Run this for each simulator. In this example, you should have four command prompts open.

Finally, when you started the RTI Server, it should have opened a simple GUI that shows the sims have connected. At this point, all four sims (A, B, C and D) should be listed. There is a button called "**Send Start Request To All**" on that GUI. Click on it. Until you click on it, each Wrapper will wait until they are told to start (this makes it easier to not start the system until you are ready). When you do click on it, the system will start "Step 0," for "Stage 0." It will proceed, and in this case, end, automatically.


To connect a simulation to this RTI, please use the following credentials:

hostname = 35.3.13.222

(or you can use 'localhost' if running simulation on same computer as RTI)

portnumber = 42010

ally
RTI
HLA
Export Message History
Send Start Request To All
PUBLISH TO (click on connected app to s...
SUBSCRIBE TO (click on connected app t...

iii. Airport Example – Version 2 (Java)

"Airport Example – Version 1 (Java)" assumed that the user would want all sims to act at the same time during each timestep. Also, each simulator currently uses the oldest copies of a "subscribed" message that hasn't been used so far (the message received from the end of the previous timestep). This is better simulation design to take advantage of parallel computing, but there may be instances where the user requires that a specific simulator runs before another at each timestep, in order to use immediate data generated within that step.

In this example, let's suppose for Stage 1, we want D to run first, then we want A and B to run, explicitly utilizing the message used from within that step.

SRTI v2.00.00 supports this type of design too. To allow it to occur, we are going to make a modification to the four "config" files created in **Step 3** of "Airport Example – Version 1 (Java)." The rest of that example's instructions remain the same.

Refer to the updated versions of each config file. Changes have been highlighted in red.

"configA.json"

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "A_Sim",
    "simulatorRef": "airport_20190104.A",
    "debugConsole": false,
    "debugFile": false,
    "dataFileOut": false,
     "initializeChannels":[
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 1,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "D",
            "oneTime": false,
            "varChannel": [
              {
                   "valueName": "D_numberOfPeople",
                   "varName": "changeInPeople"
              }
            ],
               "mandatory": true,
               "relativeOrder": 3,
               "maxTimestep": -1,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
          [
          {
               "varName": "people",
               "condition": "<=",
               "value": 0
          }
          ]
     ],
     "stageConditions": [
     ]
}
```

“configB.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "B_Sim",
    "simulatorRef": "airport_20190104.B",
    “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
          {
               "functionName": "Initialize",
               "stage": 0
          }
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 1,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "D",
           "oneTime": false,
           "varChannel": [
               {
                    "valueName": "D_numberOfPeople",
                    "varName": "changeInPeople"
               }
           ],
               "mandatory": true,
               "relativeOrder": 3,
               "maxTimestep": -1,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
     ],
     "stageConditions": [
          [
          {
               "oldStage": 0,
               "varName": "passengers",
               "condition": ">=",
               "value": 100,
               "newStage": 1
          }
          ],
          [
          {
               "oldStage": 0,
               "varName": "timeToFly",
               "condition": "<=",
               "value": 0,
               "newStage": 1
          }
          ]
     ]
}
```

“configC.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "C_Sim",
    "simulatorRef": "airport_20190104.C",
    “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
          {
               "functionName": "Initialize",
               "stage": 1
          }
     ],
     "stageChannels":[
          {
               "stage": 1,
               "order": 0,
               "timestepDelta": 30,
               "timestepMul": 1
          },
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 30,
               "stage": 1
          }
     ],
    "subscribedChannels": [
        {
               "messageName": "RTI_",
             "oneTime": false,
             "varChannel": [
               {
                    "valueName": "vTimestamp",
                    "varName": "timestamp"
               }
             ],
               "mandatory": true,
               "relativeOrder": 0,
               "maxTimestep": 0,
               "timestepDelta": 1,
               "stage": 0
        }
    ],
    "publishedChannels": [
    ],
     "endConditions": [
     ],
     "stageConditions": [
          [
          {
               "oldStage": 1,
               "varName": "milesToTravel",
               "condition": "<=",
               "value": 0,
               "newStage": 0
          }
          ]
     ]
}
```

“configD.json”

```
{
    "hostName": "localhost",
    "portNumber": "42010",
    "simulatorName": "D_Sim",
    "simulatorRef": "airport_20190104.D",
     “debugConsole”: false,
    “debugFile”: false,
    “dataFileOut”: false,
     "initializeChannels":[
     ],
     "stageChannels":[
          {
               "stage": 0,
               "order": 0,
               "timestepDelta": 1,
               "timestepMul": 1
          }
     ],
     "simulateChannels":[
          {
               "functionName": "Simulate",
               "timestepDelta": 1,
               "stage": 0
          }
     ],
    "subscribedChannels": [
          {
             "messageName": "RTI_",
            "oneTime": false,
            "varChannel": [
              {
                   "valueName": "vTimestamp",
                   "varName": "timestamp"
              }
            ],
              "mandatory": true,
              "relativeOrder": 0,
              "maxTimestep": 0,
              "timestepDelta": 1,
              "stage": 0
        }
    ],
    "publishedChannels": [
          {
              "messageName": "D",
            "initial": false,
            "varChannel": [
              {
                   "valueName": "D_numberOfPeople",
                        "varName": "numberOfPeople"
              }
            ],
              "relativeOrder": 0,
              "maxTimestep": 0,
              "timestepDelta": 1,
              "stage": 0
        }
    ],
     "endConditions": [
     ],
     "stageConditions": [
     ]
}
```

An explanation of the changes:

For configA.json and configB.json, “order: 1” instead of 0. Meanwhile, configD.json retains “order: 0.” The RTI Server will receive “order” within “RTI_FinishStep” from each simulator, and within a given step, the RTI Server will send “RTI_StartStep” to each given simulator, based on both timestep and order. Specifically, “RTI_StartStep” would be sent to D_Sim, then RTI Server will wait to receive “RTI_FinishStep” from D_Sim, then RTI Server will send “RTI_StartStep” to both A_Sim and B_Sim, wait to receive “RTI_FinishStep” for both, then update the virtual timestep, repeat.

configA.json and configB.json also update “relativeOrder” and “maxTimestep” under “subscribedChannels.” Originally, “relativeOrder = 0,” which (according to the documentation) is a code that means to pass the oldest message of type “D” the RTI Lib API has and copy to the simulators. Now, it is set to “3,” which means to retrieve the newest message that has vTimestep greater than (current_vTimestep + maxTimestep). Additionally, “maxTimestep = -1,” so this means at vTimestep = 5, “A_Sim” and “B_Sim” will check to the newest message available of “D” with a vTimestep > 4 (ie. >= 5). If there existed a message “D” at vTimestep = 6, then that would be retrieved instead, but since RTI Server keeps everyone synchronized, “D_Sim” cannot proceed to vTimestep 6 until both “A_Sim” and “B_Sim” have confirmed they finished step 5.

Because of this time-synchronization logic, it would also be correct to simply set “relativeOrder” to 1 for “A_Sim” and “B_Sim,” which means to get the newest available message at each timestep. Using “order = 0” and “order = 1” is still required in this case, but the value of “maxTimestep” would be irrelevant.

If “D_Sim” doesn’t send an initial message “D” at the beginning of the simulation system, then if “order = 0” for all simulators, “A_Sim” and “B_Sim” might start checking for message “D” at the same time “D_Sim” executes, but won’t proceed until “D” is published anyway. Therefore, in our example, the changes in this section aren’t necessary, but are helpful to conceptualize what is occurring.

iv. Airport Example – Version 3 (Matlab)

This example can also be written in other languages: specifically, a SRTI Wrapper is available to do this in Matlab.

There are a few minor changes to keep in mind with the Matlab Wrapper versus the Java Wrapper:

1) The Matlab Wrapper is provided as a Matlab .m script, not pre-compiled. This requires the user to have a licensed installation of Matlab on their machine.
2) The format of the simulator file in Matlab might be a little unusual.
3) Arrays and matrices in Matlab are easy to pass between Matlab variables, but less so with other languages (like Java). A compatible fix is under development to make Matlab compatible with language-independent arrays, but for now, users would have to be wary of formatting.

That's it. The configuration files and general startup process remains the same.

In our example (which doesn't contain arrays), difference 2) is the only concern. The RTI Wrapper and the configuration files rely on the ability to get and set public variables in a simulator class, and to call multiple functions from a specific class. This is possible in Matlab, but contradictory to common coding styles in Matlab: typically, a Matlab script only has one function per file, and cannot support different instances of the same 'global' variables. To get around this, our Matlab example requires a simulator be a 'class' instead of a script, must extend from 'handle,' and must be careful in how it updates its variable values.

Below is how A_Sim.m might look like, using the same design as the Java version:

```
classdef A_Sim < handle

    properties
       people = 450
       changeInPeople = 0
    end

    methods
        function obj = A_Matlab()
            obj.people = 450;
            obj.changeInPeople = 0;
        end

        function Initialize(obj)
           obj.people = 450;
        end

        function Simulate(obj)
            obj.people = obj.people - obj.changeInPeople;
        end
    end
end
```

While we could create all of the simulators in Matlab, instead, let's showcase the ability for SRTI to work with multiple languages, and only have Matlab replace "A_Sim," and keep the original Java code for the other simulators.

Refer to *"ii. Airport Example – Version 1 (Java),"* and follow its steps. For **Step 3**, you do not need to compile the Matlab Wrapper or Matlab Script into a standalone executable, but both scripts should be in the same folder directory. For **Step 4**, run the SRTI_Wrapper.m file (provided with the SRTI) normally, either within the Matlab graphic interface or through a command line, at the same time as you start the other Java simulators. The system should run as smoothly as it would if all simulators were in the same language.

v. Airport Example – Version 4 (NetLogo)

This example can also be written in other languages: specifically, a SRTI Wrapper is available to do this in NetLogo.

NetLogo is a free GUI program and scripting language designed for agent-based models, popular with students, teachers and researchers (Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.).

To write a Wrapper that allows access between a NetLogo simulator and the SRTI, it was decided for the Wrapper to be written in the Java language, to utilize important functions such as JSON parsing and tcp sockets. NetLogo is partially written in core Java, and it is possible to import NetLogo libraries into a Java program to open an instance of NetLogo and a simulator, as well as access public functions and variables.

In fact, the NetLogo Wrapper is compiled with a complete version of NetLogo 6.1 built inside, to allow the reference to its libraries. It is free to redistribute and/or modify under the terms of the GNU General Public License (https://ccl.northwestern.edu/netlogo/docs/copyright.html , referenced 2019-09-12). This means a user can use the SRTI NetLogo Wrapper to launch a NetLogo simulator without installing NetLogo themselves; alternatively, if they require a specific version of NetLogo, the SRTI would be unable to access it without recompiling the Java-language NetLogo Wrapper, even if it was installed on their system. Therefore, users would have to check that their simulator is compatible with NetLogo 6.1 (differences between each version may make certain data types or syntax rules incompatible).

Aside from this rule, the process to run the NetLogo Wrapper is similar to running the Java Wrapper. While the Java Wrapper requires you to reference the simulator file in the command, the NetLogo Wrapper doesn't: the NetLogo .nlogo file just has to be in the same folder directory. The configuration file format is the same as those for Java and Matlab. The command to run would be similar to the following:

```
>> java -cp "SRTI_NetLogoWrapper_v2_20_02.jar" main.Wrapper
```

Below is an example for how A_Sim.nlogo might be written:

```
globals [people changeInPeople]

to Initialize
     clear-all
     set people 450
     set changeInPeople 0
end

to Simulate
     set people people - changeInPeople
end
```

While we could create all of the simulators in NetLogo, instead, let's showcase the ability for SRTI to work with multiple languages, and only have NetLogo replace "A_Sim," and keep the original Java code for the other simulators.

Refer to *"ii. Airport Example – Version 1 (Java),"* and follow its steps. For **Step 3**, you do not need to compile the NetLogo Wrapper or NetLogo Script into a standalone executable, but both scripts should be in the same folder directory. For **Step 4**, when you run the above command to launch the NetLogo Wrapper, you should see the NetLogo interface open for you to observe when the SRTI executes. The system should run as smoothly as it would if all simulators were in the same language.

Chapter 3 – Additional Notes

For assistance or questions, users can contact developer Andrew (Andy) Hlynka at ahlynka@umich.edu .