SRTI - SIMPLE RUN-TIME INTERFACE

User and Developer Documentation

v1.00.01 – 2019-03-07

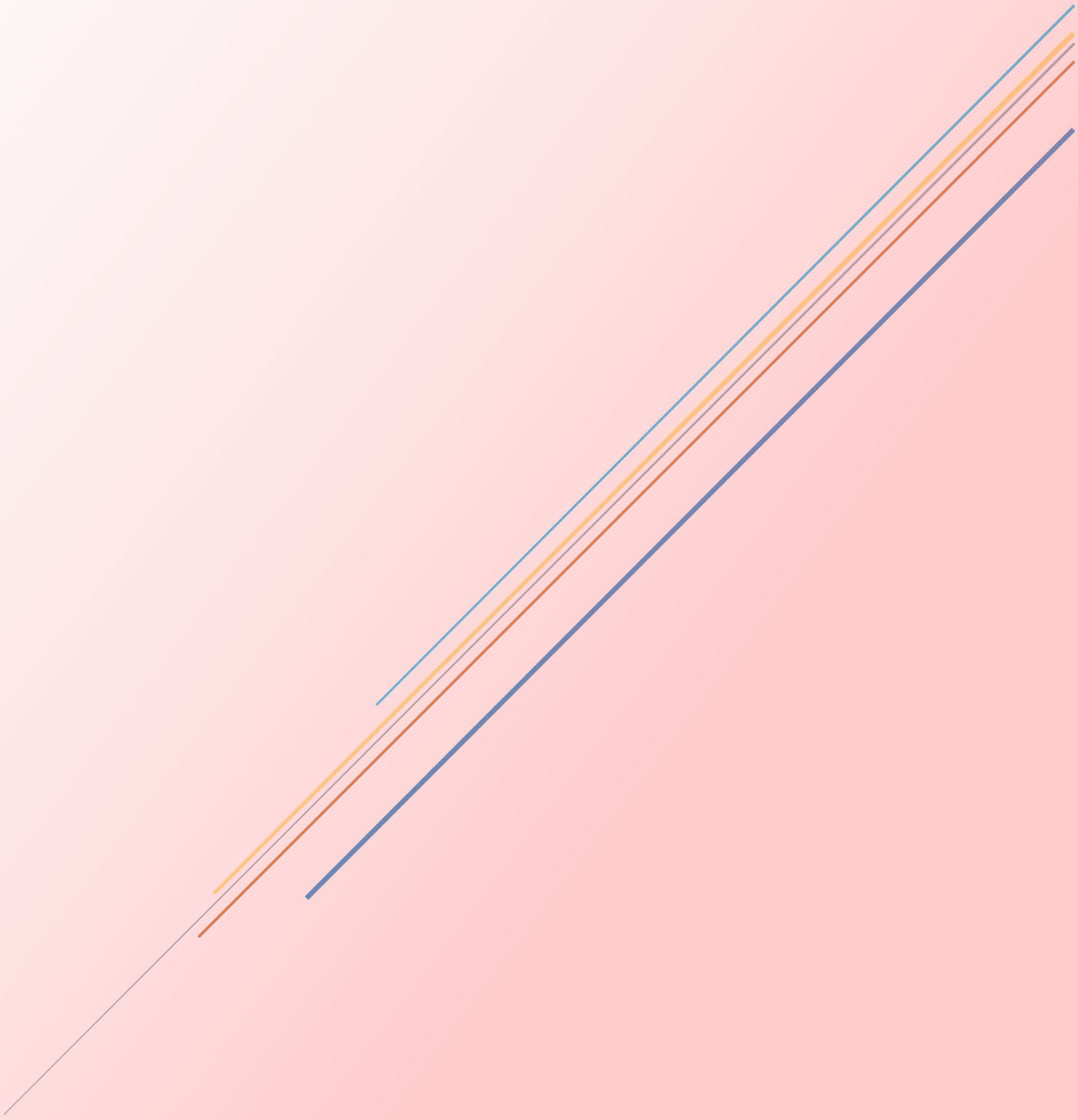

Table of Contents:

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

ii. Third-Party Licenses

<<SECTION PENDING REVISION – 2018-04-13>>

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

2018-07-04

- This covers release v0.61. This SRTI adds further functionality and bug improvements, including fixing issues with backwards compatibility between older and newer versions of the SRTI. Significant feature additions include a) TCP (the optional ability to require message confirmation from within the SRTI logic, optional ability to have RTI Lib attempt reconnection under certain conditions), and b) new “Debug GUI” to trace live debug output (from server or client side) alongside existing “Server GUI”. The GUI systems are only available in the Java versions of the SRTI.

2018-05-31

- This covers release v0.54. This version includes new features including: a) TCP (optional confirmation between server and client for message transfer), b) improved RAM management in the RTI Server (regarding keeping message history), c) “subscribeToMessagePlusHistory” to subscribe to a specific message and its past instances, d) a settings.txt file that helps configure the RTI Server.

2018-04-11

- This covers release v0.50. The SRTI retains basic features to send and receive data in JSON format, relying on the ability to parse JSON and Socket communication (the details of which are hidden from the user). The server is available as a Java executable (.jar), and the client library is available in Java (.jar, or source code) or C++ (.dll, or source code). A simulation must be edited to call upon appropriate API functions in the client library in its programming code.
- Compared to versions prior to v0.50, the message parameter “fromSim” is changed to “source” for better understanding.

Chapter 1 – Introduction

i. Definitions

IoT: Internet of Things.

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

RTIServer: 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.

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).

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

ii. What is SRTI?

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

The basic overview of SRTI is that it contains a server-side application that, when already opened, can be accessed by client-side API (provided within SRTI). The server can be on the same or on a different machine from a simulation system, and the client API must be stored locally to be accessed by the simulations. The Java version of SRTI is stored in a pre-compiled .jar executable, which can be opened as an application to start the server. The same .jar file can be referenced to access client API functions.

It is assumed the simulation is compatible to access Java functions in a .jar file: a C++ version of the client API is also available, but to support other languages, the underlying logic of the client API must be rewritten (the Java server application can be reused for all connecting simulations).

Chapter 1, part iii. Discusses the parts of SRTI from a higher level, and *Chapter 1, part iv.* discusses some lower level detail to how the individual parts function. *Chapter 3* covers the code structure of the SRTI application.

iii. Core Concepts

While this section will not go into detail with how the underlying logic of SRTI works (refer to *Chapter 1, part iv*. for more information), 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 RTILib.

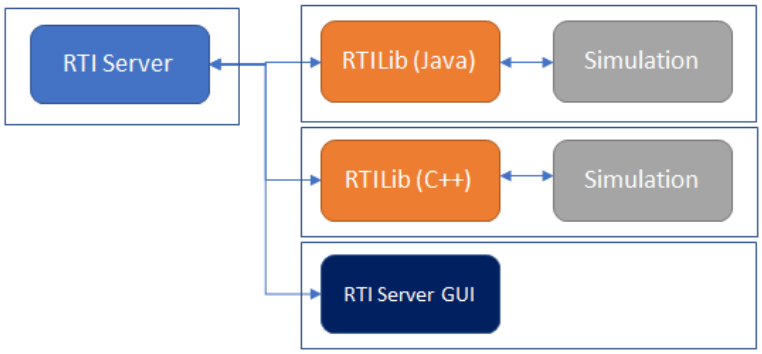

Figure 1 - High-Level Concept of SRTI

There exist three important parts to an SRTI system:

1. *RTIServer*: 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 RTIServer 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 RTIServer, 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. RTIServer is provided in the compiled SRTI.
2. *RTILib*: an API library that must be used by the simulation to connect to the RTIServer. It contains functions to connect, disconnect, publish, and receive messages. The details of socket communication, JSON creation and parsing is handled within RTILib, such that the simulation does not need to know understand anything beyond the provided API. RTILib is provided in the compiled SRTI.

3. *Simulation:* the simulation provided by the user. The user must modify the simulation to reference the RTILib 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. *RTIServerGUI*: a graphical user interface that automatically opens when running RTIServer. 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 RTIServer, 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 RTIServer, the code that the user must add will be meant to do the following (through the RTILib):

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

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), including 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.

<table>
<tr><td rowspan="8">{
"portNumber":4200,
"guiOn":true,
"debugGuiOn":false,
"tcpOn":false,
"oldMessageLimit":100000,
"oldMessageArchive":false,
"debugConsole":false,
"debugFile":true
}</td><td>"portNumber"</td><td>Integer. explicitly sets socket connection on port if available. Use -1 to use randomly-available port.</td></tr>
<tr><td>"guiOn"</td><td>true or false. Determines whether to display the main "RTI Server GUI".</td></tr>
<tr><td>"debugGuiOn"</td><td>true or false. Determines whether to display "Debug GUI".</td></tr>
<tr><td>"tcpOn"</td><td>true or false. Determines whether messages from RTI Server must receive confirmation from simulations.</td></tr>
<tr><td>"oldMessageLimit"</td><td>Integer. Sets limit to how many past messages are saved in memory (good to control memory limit). Number represents character number, not message number. Use -1 to use infinite limit.</td></tr>
<tr><td>"oldMessageArchive"</td><td>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.</td></tr>
<tr><td>"debugConsole"</td><td>true or false. Determines whether to print debug output to console window.</td></tr>
<tr><td>"debugFile"</td><td>True or false. Determines whether to print debug output to file for analysis after execution.</td></tr>
</table>

iv. Detailed Concepts

The SRTI system relies on a shared public server application (which comes included with the software source code), and on client-side API that handles the details of connecting to the server. These connections are made with "socket" communication, a two-way communication system. While different socket protocols exist, SRTI's use of sockets is for the SRTI server to show (through a UI or debug output) a "hostname" (network address) and "portnumber" (port the application is open on the local machine for communication). The client-side API must refer to these two parameters (the user would input this manually from their simulation) to have their simulation connect. After connecting to the main server's thread, the server begins a separate thread and socket dedicated to the reading input and writing output to the specific simulation but refers back to the main thread to distribute a new message out to other simulation threads as required. This type of socket connection is technically an example of TCP communication, but there is no explicit handling of messages within SRTI in situations where a socket connection is

lost, where a message is not received perfectly, or where messages are received out of order due to network delay between different computers (regardless of submission time, SRTI will send out messages in the order the server receives them).

The “hostname” and “portnumber” can change in different instances: the “hostname” is dependent on the computer’s Internet connection and would commonly be through a dynamic-IP router (without an Internet connection, the SRTI can still function through “localhost,” but would not be able to connect with simulations on different computer systems) and the “portnumber” is typically chosen based on which ports on a machine are available at the time.

While the default format of a message shared through SRTI is JSON format, in theory the “content” parameter itself could be in any other format, as long as a simulation is capable of parsing out the format of the data.

The issue of SRTI’s runtime being slower to similar software solutions may concern some users: early testing suggests that SRTI’s system runs approximately five-times slower than similar software with an identical simulation system. Faster solutions typically require a message format to be predefined as a data structure, which must be compiled each time the format is redefined by the user, the resulting code allowing sending data as pure bytes representing data types like “int” or “boolean.” The insistence for SRTI to use “string” format allows any message to be sent without compilation, but requires more bytes for the same amount of information: one “int” typically requires 2 bytes, and one character in a “string” takes 1 byte, so to send the same information as a string could require significantly larger amounts of data (the exact amount of bytes a data type uses can vary depending on the language and system implementation). Additionally, the computational expense of parsing a JSON message to search for different parameters is greater than storing data into a defined struct directly, this also plays a part in the efficiency. The flexibility SRTI provides (including not having to compile its code in order to use it with your own simulation program) is considered to be worth the trade-off, especially for smaller simulation systems that can execute within a relatively shorter amount of time.

Recent tests have suggested that printing debug output (especially to files), and parsing/generating JSON objects to send between simulations, are the most expensive operations from a time perspective for the SRTI.

Chapter 2 – API List

The following is a list of the available public API in RTILib. The format references the API corresponding with the Java implementation but is also available with the same name and parameters in the available C++ RTILib. If an API is available but not listed in the below format, it's functionality may not be complete and is not promised to work as expected. In cases where a function returns an integer, it represents an error code, where "0" means operation was successful, anything else suggests an error.

public RTILib()	Constructor to create a new instance of RTILib to be able to use the API.
public RTILib(RTISim rtiSim)	Constructor to create a new instance of RTILib to be able to use the API. Passes in an instance of the simulator if it extends from "RTISim" (interface within RTILib), allows automatic callbacks.
public void setSimName(String newName)	Set simulator name to be used in RTILib, to set message "source" and make easier to trace on server side. Highly recommended.
public void setTcpOn(boolean tcp)	Introduced in v0.54. Set TCP on or off (feature where all messages sent through RTILib must receive a response to confirm it was received, else it is resent up to three times.
public void setReconnectTimeLimt(long timeLimit)	Introduced in v0.61. Set a thread that checks every "timeLimit" milliseconds to see if any updates were received by the RTI Server in that time. If not, assume disconnection occurred, and begin reconnection.
public int connect(String hostName, String portNumber)	Connect to the RTIServer. The RTIServer must already be open, and the hostname and portnumber must match the instance of the RTIServer to connect successfully.
public int reconnect()	Introduced in v0.61. Close socket (if open), open new socket using same parameters last used in "connect" function, subscribe to messages subscribed in the past, plus "latest" message in Server memory.
public int reconnect (String lastMessageName, String lastMessageContent)	Introduced in v0.61. Same as "reconnect()," but also resends message with "lastMessageName" and "lastMessageContent." Used when reconnect occurs after a sent message doesn't get TCP confirmation.
public int disconnect()	Disconnect from the RTIServer. Recommended to call at the end of a simulation for safe disconnection.
public int subscribeTo(String messageName)	Subscribe to a message. If the RTIServer receives any messages by this name, it will forward message to this simulation. This only needs to be called once per message name, at the beginning of a simulation, and all messages by this name after this function call can be received.
public int subscribeToMessagePlusHistory (String messageName)	Introduced in v0.54. Subscribe to a message, plus all messages of that type the server had received before the simulation joined.
public int subscribeToMessagePlusLatest(String messageName)	Introduced in v0.61. Subscribe to a message, plus the latest message (based on timestamp) of that type the server had received before the simulation joined.
public int subscribeToAll()	Automatically subscribe to all messages. The simulation will not miss any messages received by the RTIServer, but most of the data may be irrelevant and network traffic might slow down system.

public int subscribeToAllPlusHistory()	Automatically subscribe to all messages, plus all messages the server had received before the simulation joined. The RTIServer automatically saves all messages in memory to allow this.
public int publishTo(String messageName)	Set simulation to be able to publish messages by name. Currently not in use as of v0.50: any simulation can send any message without permissions check.
public int publish(String name, String content)	Publish message "name" with content "content." "Name" is the name of the message to allow subscribing simulations to receive it. "Content" is a JSON-formatted message with relevant data to share.
public int sendWithoutAddingToTcp (String name, String content, String timestamp, String source)	Introduced in v0.61. NOT USED BY SIMULATION. Used to resend a TCP message without re-adding the message again to the TCP buffer (list being checked for whether to resend messages).
public int receivedMessage(String message)	NOT USED BY SIMULATION, but by another part of RTILib. Handles action when receiving new message from RTIServer, either to callback simulation or add to buffer to be accessed later.
public int setTcpResponse(boolean setResponse, String message)	Introduced in v0.61. NOT USED BY SIMULATION. Used by thread receiving messages from RTI Server to confirm a message was received.
public int handleTcpResponse(String name, String content, String timestamp, String source, String message)	Introduced in v0.61. NOT USED BY SIMULATION. Add message to TCP buffer to check if confirmation gets received later, if tcp setting is true.
public int checkTcpMessage()	Introduced in v0.61. NOT USED BY SIMULATION. Check TCP buffer periodically, resend messages if not confirmed to be received, and if not received after multiple attempts, begin "reconnect()" to RTI Server.
public String getNextMessage()	Get next message saved in RTILib buffer, removes from list after retrieval. Returned as full JSON message. If no messages received yet, return null or empty string.
public String getNextMessage(int millisToWait)	Get next message saved in RTILib buffer, removes from list after retrieval. Returned as full JSON message. If no messages received, wait up to "millisToWait" milliseconds for message before returning null.
public String getNextMessage(Integer millisToWait)	Same as "public String getNextMessae(int millisToWait)", but with different data format for "millisToWait."
public String getNextMessage(String messageName)	Get next message of name "messageName" saved in RTILib buffer, removes from list after retrieval. Returned as full JSON message. If no messages received yet, return null or empty string.
public String getNextMessage(String messageName, int millisToWait)	Get next message of name "messageName" saved in RTILib buffer, removes from list after retrieval. Returned as full JSON message. If no messages received yet, wait up to "millisToWait" milliseconds.
public String waitForNextMessage()	Get next message, wait indefinitely until a message is received. Returned as full JSON message. If message is never received from RTIServer, stays within infinite loop.
public String getJsonObject(String name, String content)	Use RTILib to parse out object from JSON message. "name" is the name of the parameter in the JSON message, and "content" is the original JSON message in String format. May return a sub-JSON object that requires further parsing.
public String getJsonObjectFast (String name, String content)	Introduced in v0.61. Experimental option to get the value of a parameter from a JSON object without JSON parsing for faster execution. Best used on simple messages.
public String getJsonString(String name, String content)	Same as "public String getJsonObject(String name, String content)" but forces "string" value instead of "object." SRTI forces string format regardless, so should return similar value, possibly different formatting.
Public String getJsonStringFast (String name, String content)	Introduced in v0.61. See "getJsonObjectFast," but for "getJsonString."

public String[] getJsonArray(String content)	Returns one-dimensional array of strings parsed from JSON string “content.” “Content” must be an explicit JSON array, else error might occur in parsing.
public String getStringNoQuotes(String content)	Optional function to remove outside quotations on a string if exists. For example, content “ “hello” “ is returned as “hello”. Sometimes necessary to fix JSON parsing using RTILib API.
public String getMessageName(String originalMessage)	Return message “name” from JSON message “originalMessage”. The name is used as an ID to subscribe or for simulation to know what to expect inside.
public String getMessageTimestamp(String originalMessage)	Return message “timestamp” from JSON message “originalMessage.” Represents system time in milliseconds (epoch time) when message was originally sent from source.
public String getMessageSource(String originalMessage)	Return message “source” from JSON message “originalMessage.” Represents the name of the simulation that originally sent the message, other simulations might handle choose to handle based on source.
public String getMessageContent(String originalMessage)	Return message “content” from JSON message “originalMessage.” Represents the actual content being shared from the simulation, typically in JSON format.
public String setJsonObject(String originalJson, String nameNewObject, String contentNewObject)	Add JSON object parameter to “originalJson” (can be “” if making brand new JSON string) of name “nameNewObject” and string content “contentNewObject”. Returns a new JSON string with the new object.
public String setJsonObject(String originalJson, String nameNewObject, int contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “int”.
public String setJsonObject(String originalJson, String nameNewObject, float contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “float”.
public String setJsonObject(String originalJson, String nameNewObject, long contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “long”.
public String setJsonObject(String originalJson, String nameNewObject, double contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “double”.
public String setJsonObject(String originalJson, String nameNewObject, char contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “char”.
public String setJsonObject(String originalJson, String nameNewObject, boolean contentNewObject)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being of type “boolean”.
public String setJsonArray(String originalJson, String nameNewObject, String[] contentNewArray)	Same as “public String setJsonObject(String origianlJson, String nameNewObject, int contentNewObject)”, but with the new content value being a one-dimensional array of type “string.”
public void printVersion()	For debugging purposes, prints out the version of SRTI (example: “v0.50”).
public void setDebugOutput(boolean setDebugOut)	Sets whether or not to print out additional debugging information during execution, passing “setDebugOut” to “false” will result in less information taking up console output.
public void setDebugFileOutput(Boolean setFileDebugOut)	Introduced in v0.61. Set whether or not to print out debugging information to file during execution, to be studied by user after execution.
public setDebugGUI(Boolean setGUIOut)	Introduced in v0.61. Set whether or not to print out debugging information to visual interface during execution, to be studied live by user during execution.

public void printLine(String line)	NOT USED BY SIMULATION. Used internally to handle printing out debugging information.

Chapter 3 – Example Use Case

i. Java – Publishing Message

The following example is written in Java and shows how one can write a simple Java simulation to use RTILib to publish messages.

```
// location inside SRTI_v050.jar where RTILib exists. Externally, need to
compile with reference to .jar file.
import mainServer.RTILib;


import java.util.concurrent.TimeUnit;

public class ExampleSim_01 {

    public static void main(String [] args){
        // These 2 values may need to be updated.
        String hostname = "35.3.75.84";
        String portnum = "4200";

        RTILib rtiLib = new RTILib();
        rtiLib.setSimName("ExampleSim_01");
        rtiLib.connect(hostname, portnum);

        // Publish message every second for 100 seconds.
        for (int i = 0; i < 100; i++){
            String messageContent = rtiLib.setJsonObject("", "Message",
    "Hello " + i);
            rtiLib.publish("Greeting", messageContent);
            TimeUnit.SECONDS.sleep(1);
        }

        rtiLib.disconnect();
    }
}
```

ii. Java – Subscribing to Message without Callback

The following example is written in Java and shows how one can write a simple Java simulation to use RTILib to subscribe to messages, and to use the RTILib buffer to access new messages when available.

```
// location inside SRTI_v050.jar where RTILib exists. Externally, need to
compile with reference to .jar file.
import mainServer.RTILib;

import java.util.concurrent.TimeUnit;

public class ExampleSim_02 {

    public static void main(String [] args){
        // These 2 values may need to be updated.
        String hostname = "35.3.75.84";
        String portnum = "4200";

        RTILib rtiLib = new RTILib();
        rtiLib.setSimName("ExampleSim_01");
        rtiLib.connect(hostname, portnum);
        rtiLib.subscribeTo("Greeting");

        // Publish message every second for 100 seconds.
        for (int i = 0; i < 100; i++){
            String message = rtiLib.getNextMessage("Greeting");
            if (message != null){
                String content = rtiLib.getMessageContent(message);
                System.out.println("I received a message that says: " +
rtiLib.getJsonObject("Message", content));
            }
            TimeUnit.SECONDS.sleep(1);
        }

        rtiLib.disconnect();
    }
}
```

iii. Java – Subscribing to Message with Callback

The following example is written in Java and shows how one can write a simple Java simulation to use RTILib to subscribe to messages, and to use the RTILib "RTISim" interface definition to allow RTILib to automatically callback to the simulation when a new message is received. This is generally more efficient and a better design, but requires the simulation to be explicitly written in Java to be able to implement the "RTISim" class, where the previous examples could be written in non-native Java code compatible with Java API access.

```
// location inside SRTI_v050.jar where RTILib exists. Externally, need to
compile with reference to .jar file.
import mainServer.RTILib;
import mainServer.RTISim;

import java.util.concurrent.TimeUnit;

public class ExampleSim_03 implements RTISim {

    public static void main(String [] args){
        ExampleSim_03 thisSim = new ExampleSim_03();
    }

    String receivedMessage = "";
    RTILib rtiLib;
    public ExampleSim_03(){
        // These 2 values may need to be updated.
        String hostname = "35.3.75.84";
        String portnum = "4200";

        rtiLib = new RTILib();
        rtiLib.setSimName("ExampleSim_01");
        rtiLib.connect(hostname, portnum);
        rtiLib.subscribeTo("Greeting");

        // Publish message every second for 100 seconds.
        for (int i = 0; i < 100; i++){
            System.out.println("At step " + i + " I received this message :
" + receivedMessage);
            receivedMessage = "";
            TimeUnit.SECONDS.sleep(1);
        }

        rtiLib.disconnect();
    }

    @Override
    public String getStringName(){
        return "ExampleSim_03";
    }

    @Override
    public void receivedMessage(String name, String content, String
timestamp, String source){
        if (name.compareTo("Greeting") == 0){
            System.out.println("By the way, I received a message: " +
content);
            receivedMessage = rtiLib.getJsonObject("Message", content);
        }
    }
}
```

Chapter 4 – Source Code Organization

While the following programs appear to be separate, they are contained within the same .jar file if using the pre-compiled version.

i. Java – RTIServer - Organization

Figure 2 shows the variable and function organization within the RTIServer.

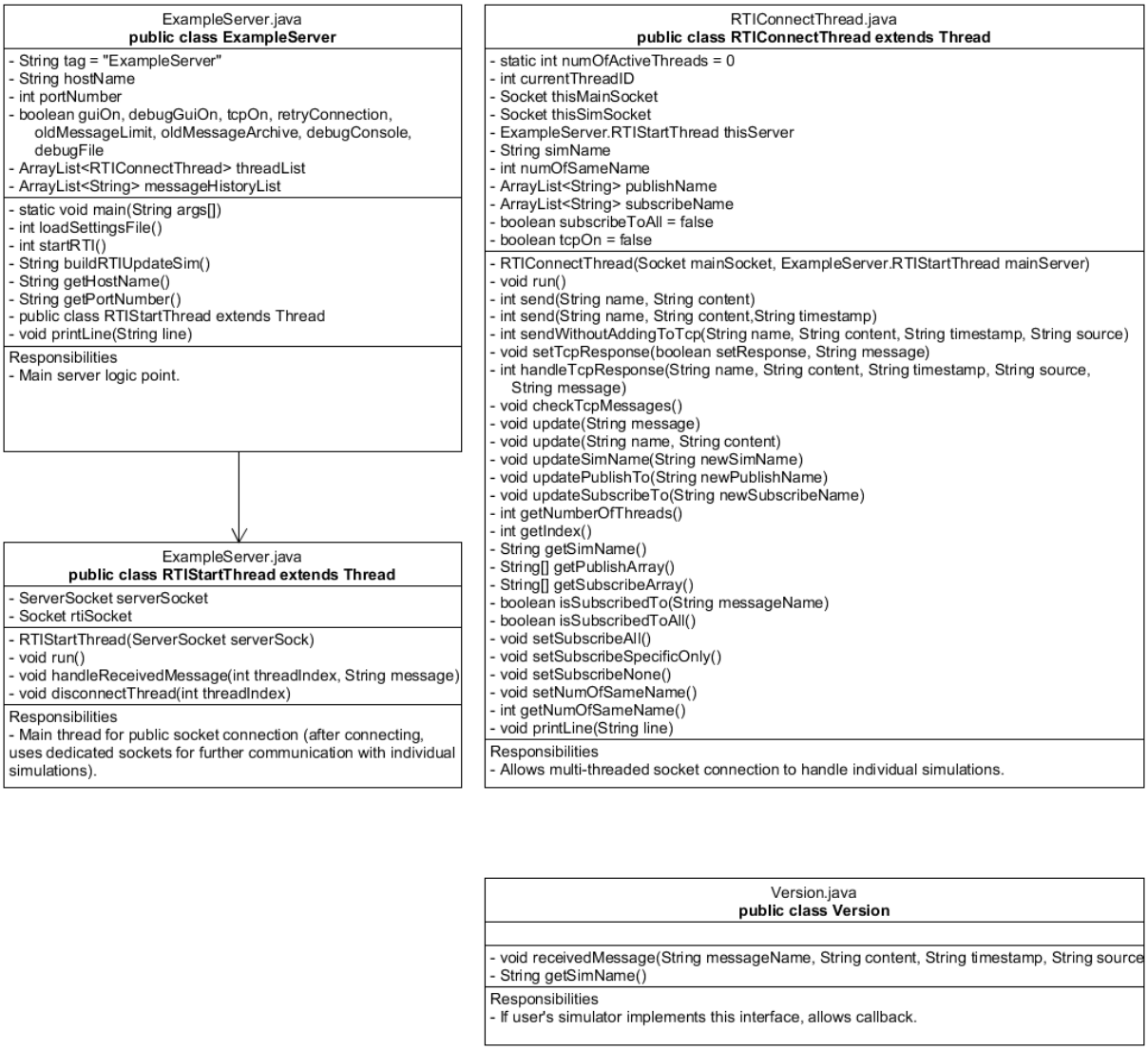

Figure 2 - 3 classes relevant to RTIServer.

ii. Java – RTILib – Organization

Figure 3 shows the variable and function organization within the RTILib (client-side API).

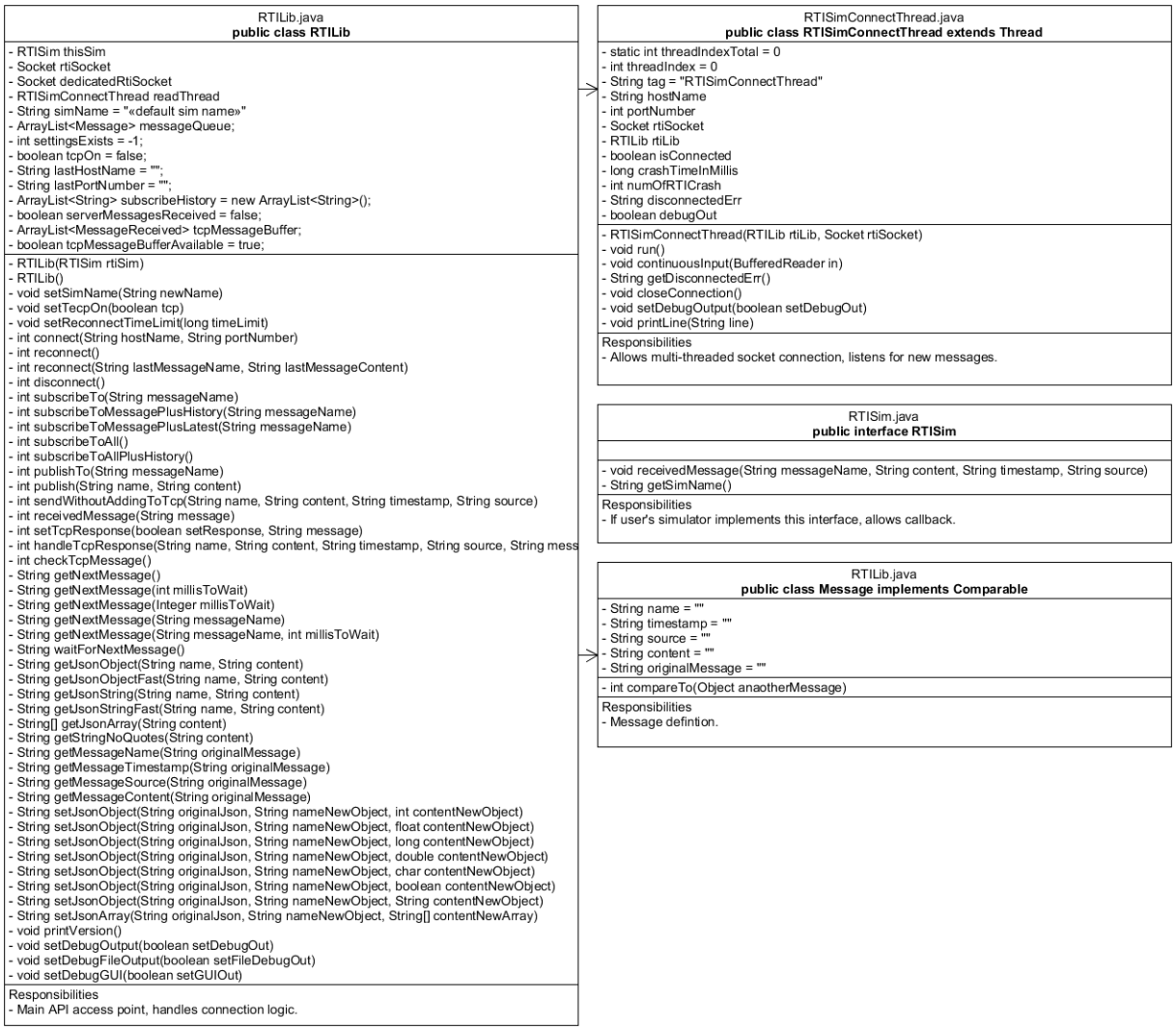

Figure 3 - 4 classes relevant to RTILib (client-side API).

iii. Java – RTIServer – Sequence Diagram

Figure 4 shows the order of events behind-the-scenes when the RTIServer begins.

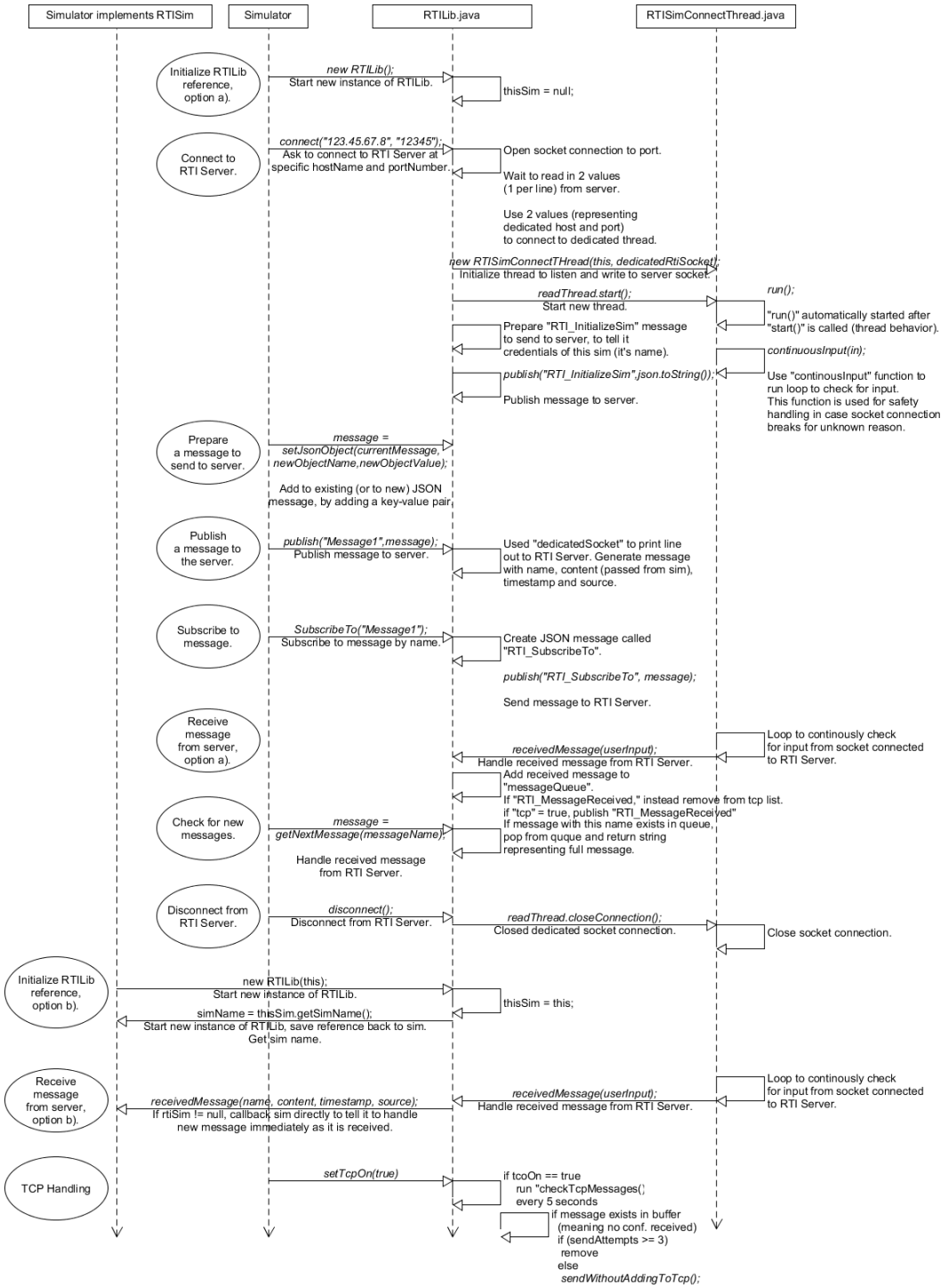

Figure 4 – Order of events for certain functions of RTIServer.

iv. Java – RTILib - Sequence

Figure 5 shows the order of events behind-the-scenes when the RTILib is used.

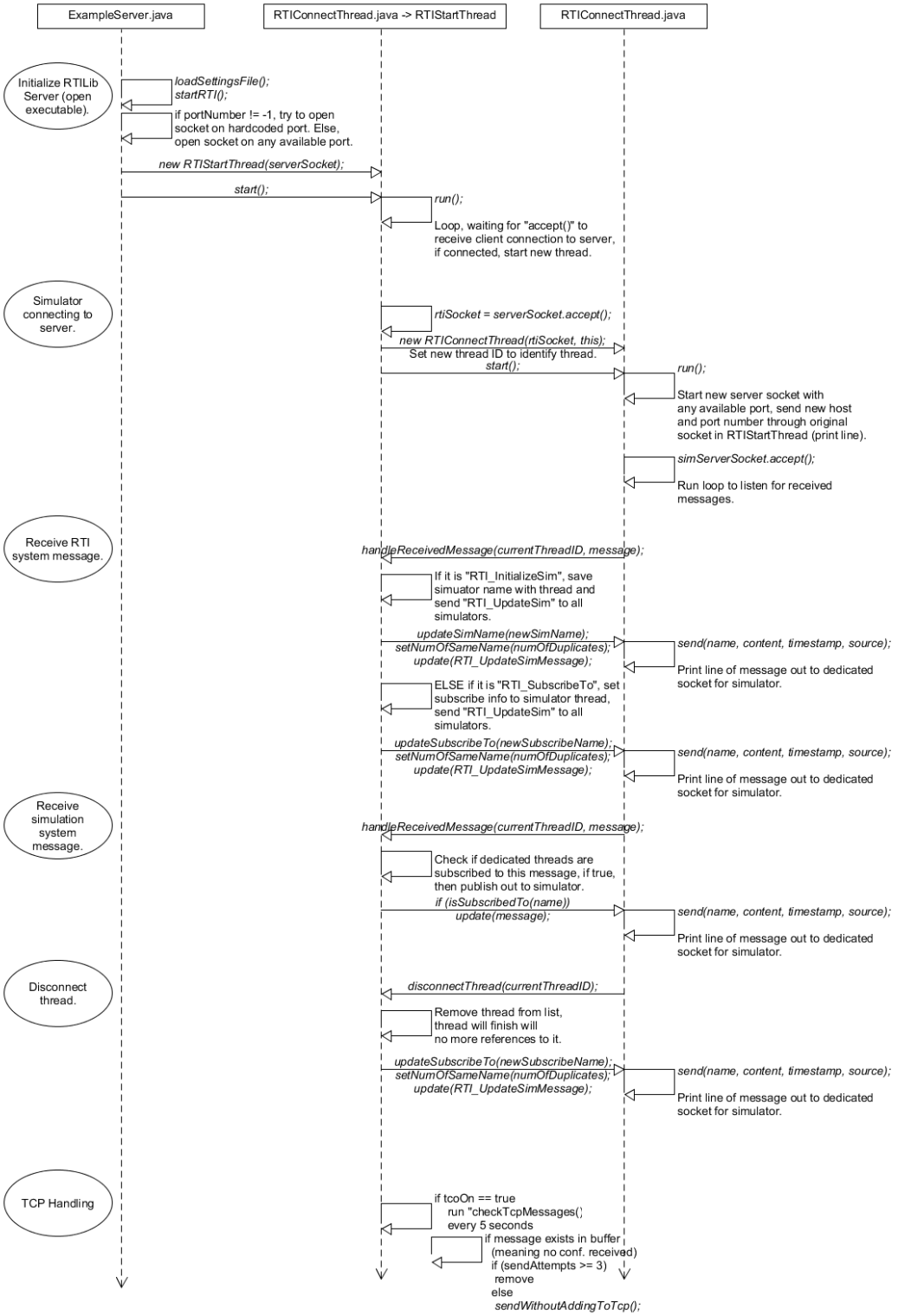

Figure 5 – Order of events for certain functions of RTILib.

v. C++ version.

Aside from some minor internal logic differences (how to initialize sockets, for example), the C++ example of the RTILib API follows the same order of events and has the same function definitions as of v0.61.

One major difference is the existence of the “Debug GUI”: this optional feature only exists in Java, not in C++. This is due to C++ not having a standard graphical system, so it isn’t attempted to avoid requiring additional third-party libraries to compile.

If a user is programming their own version of the RTILib API, it is recommended they try to follow the design pattern described here in Chapter 3.

Chapter 5 – Missing Features

i. In-Development Features

a) The current implementation of the RTILib requires the user add a few lines of programming code to their existing simulation. We want to allow the SRTI solution to be as easy to use as possible (even to users who have never programmed before). In the future, a more complete UI system will be developed to enable the ability to connect simulations without having to modify it to connect to the RTILib API: the UI system will allow a user to graphically define a wrapper. This feature will require compatible simulations to meet specific formatting requirements, simulations not normally compatible can continue to use explicit programming as seen in the current examples.

b) A system-wide time syncing protocol will allow individual systems to not rely in their independent clocks and to execute in sync. Certain simulations may run at different intervals from each other. This is a complicated topic, and until this feature is prepared (or even after), it is highly recommended that time is handled internally on the simulation side to meet all possible requirements.

ii. Additional Features For You To Consider

a) As of v0.50 of SRTI, there is no security to prevent applications from joining the simulation system and potentially reading all data being shared through it, but a user must be able to know the "hostname" and "portnumber" – because they change often, it is difficult for a unknown application to join the system without direct access to the computer running the server in order to see these values. An unknown application would also need to understand how to "subscribe" to all messages the server receives, if not familiar with the documentation a alien simulation would not be able to benefit. An allegory is having a large country with every house having no locked doors and having an SRTI system running in one of the houses: if a simulation knows the address of the server, it is easy to walk in and join. Otherwise, it is very unlikely for an ill-meaning simulation to find the server.

   - A developer might wish to encrypt data before sending through the RTIServer such that only simulations given information ahead of time would know how to decrypt and utilize any shared data, or to add logic on the RTIServer side to block simulations not to join a set system (both the system and simulations with permission would have to be defined by name).

b) As of v0.54, the user has the option to set "tcp" for both the RTI Server and the RTLib separately. If true for RTI Server, then it would expect the RTILib connected to it to return a response confirming that each message is received. If true for RTILib, then it would expect the RTI Server to return a response

confirming that each message is received. If not confirmed, each message is saved in a buffer and sent up to three times before giving up.

Chapter 5 – Additional Notes

The RTIServer is written in Java as an executable.

As of v0.50, the RTILib API is available in both Java and C++ to allow better compatibility with more languages. The RTILib API was originally designed and tested in Java, and potential bugs may be found with extensive testing of the C++ version. Where possible, the Java version is recommended.

As of v0.61, the RTILib API in Java includes an optional "debug GUI" made with the Java Swing library. The C++ version of RTILib API does not include this, in order to avoid utilizing non-standard libraries to make compiling easier.

The source code is freely available and written with the intention to be simple to implement in other languages, should the Java or C++ API not be accessible with your simulation. Socket communication and JSON parsing are two features required in order to implement a new solution compatible with the SRTI system.

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