SRTI - SIMPLE REAL TIME INTERFACE

User Documentation

V0.50 – 2018-04-11

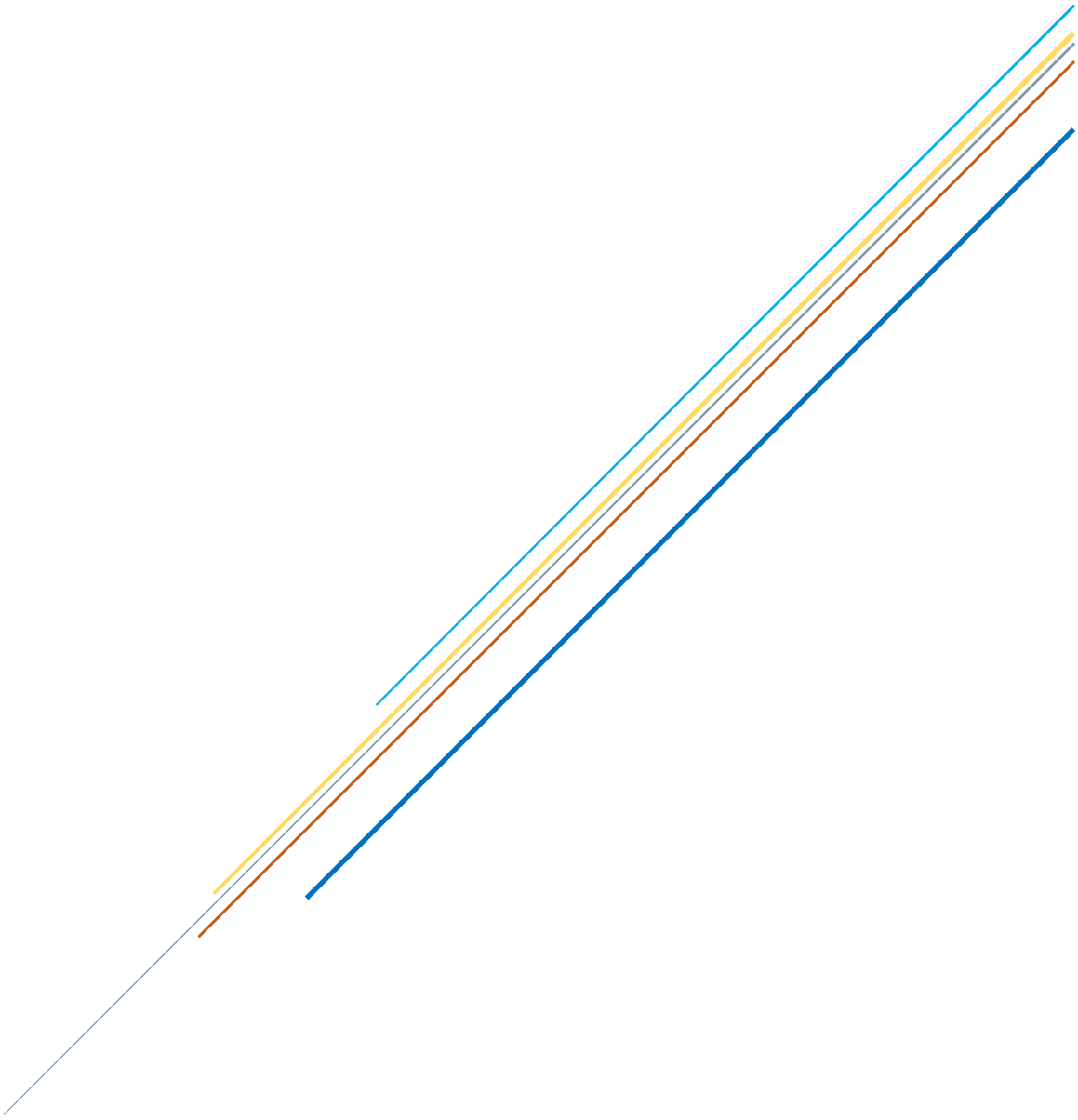

Table of Contents:

Chapter 0 – Release Notes

i. Summary

This document is an introduction to using SRTI from a **user**'s perspective, providing required information to begin using the system in their projects.

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 – 2018-04-11>>

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

2018-04-11

- 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*.

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.

Imagine an example where you have two simulations. Each study and calculate data that is relevant to each other, but with different independent goals. You might decide that it is a good idea for the second simulation to utilize the results from the first simulation.

There are multiple ways to handle this example. You might try to combine the two simulations together into a larger project. This would give you greater control over their execution and the data they share but may require a significant amount of custom programming. If the simulations are written in different languages, or if the user does not have a solid understanding of the separate simulations, this solution may not be possible or easy to implement.

The user could run each simulation in sequential order, executing the first simulation until it is finished, writing its output to a file, and using said file as input for the second simulation. This still requires some minor program modifications to each simulation to decide what parameters to write and read. For large simulation systems, running each program sequentially is not efficient. And bi-dependent simulations, where both simulations have data relevant to each other during their calculations, further complicate this solution.

There exist several third-party solutions made with the purpose of data sharing, either specialized for scientific simulations or for IoT (Internet of Things) projects. Both paid and free solutions exist. A variety of issues exist with these, including (but not limited to): ease of use, support for modern operating systems, support for multiple programming languages, requirement to design simulations and data messages to strict pre-defined formats, and requirement to compile the solution from scratch when a new message format is defined for a new simulation system. While it is possible to utilize these solutions, their limitations make them difficult and intimidating for users not comfortable with computer programming.

The SRTI is built to be as simple as possible from a user's perspective. While the source code exists, a pre-compiled library exists for a user to simply double-click to open an instance of a RTIServer, and connecting to the server only requires a few lines of programming. To publish or receive messages from the server each also only requires a few lines. The format and contents of a data message can be dynamically set, allowing the simulation system complete freedom while also taking on the responsibility to avoid bug issues. This is possible through the hidden use of "sockets" to allow connections between different simulations, and the common "json" format to define and parse messages. While initially supporting a few languages, the ability to both support "sockets" and "json" parsing is commonly available on most modern programming languages, and the API is simplified with the intent to allow developers to rewrite the library into new languages as required.

The trade-off for SRTI to allow this flexibility is that it data transfer, and parsing messages, makes the system slower than other similar solutions.

iii. Core Concepts

While this documentation 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

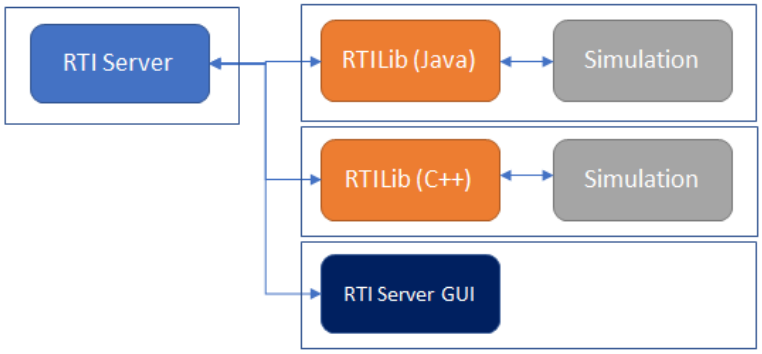

Figure 1 - High-Level Concept of SRTI

another simulation of any language, as long as they are compatible to reference the provided RTILib.

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 four 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.

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.

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 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 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.
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 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 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 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[] 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 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 – 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.

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.