NISTIR 7680

A RESTful Web Service for Virtual Supply Chain Time Management

Guillaume Radde

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 7680

A RESTful Web Service for Virtual Supply Chain Time Management

Guillaume Radde
Manufacturing Systems Integration Division
Manufacturing Engineering Laboratory

February 2010

U.S. Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Director

Table of Contents

Abstract

A common issue when building distributed simulations is the issue of time management. Each discrete event simulation has its own notion of time, and a mechanism needs to be set up to allow each simulation to run at the same pace. Different architectures exist to perform time synchronization and are usually optimized for a certain scenario and certain types of simulations. The requirements for the type of data the simulations will be exchanging, the performance of the system used, the quality and latency of the network that will be used to run the distributed simulation, and the level of collaboration between the authors of the different individual simulations lead to different design decisions for the architecture of the distributed simulation.

This paper describes a time management mechanism, based on a Service Oriented Architecture, that can be used in the construction of a virtual supply chain simulation.

Keywords: simulation, architecture, design, integration, REST, web, SOA

1 Introduction

Discrete Event Simulation (DES) is commonly used to optimize the profitability of a manufacturing company. By modeling the processes used to create products, engineers can find bottlenecks and experiment with different solutions for lowering costs. DES is commonly used to optimize the processes inside a company, but the performance and profitability of retailers and manufacturers don't solely depend on their internal processes. Each retailer depends on several manufacturers, that, in turn, depend on several suppliers. Hence, the performance of one actor in the supply chain depends on the performance of the other actors in the supply chain.

If each actor in the supply chain can use simulation to optimize its internal processes, it makes sense to try to optimize the whole supply chain by connecting the individual simulations together. However, modeling the whole supply chain crosses the boundaries of a single company and cannot be done easily by a single person. Engineers can easily model the processes of a company since they know the internal processes of their company. However, they can hardly be expected to model the processes of other companies since often that information is confidential. This is a common problem in the area of integration of heterogeneous systems, which can be solved by designing the distributed system as a Service Oriented Architecture (SOA) [1].

Using SOA, businesses that would like their systems to be integrated with the systems of other businesses expose the features of their systems as Web Services defined in an Application Programming Interface (API). They can then provide this API to their partners. Web Services allow for loose integration of systems since the internal behavior of each system is hidden behind an API. Several standards and commonly used protocols exist to design such applications, such as the Simple Object Access Protocol (SOAP) [2] or the Representational State Transfer (REST) architecture [3].

In a traditional Service Oriented Architecture, each actor defines an API to provide services to its partners, and connects to its partners using their APIs. Once the different systems are integrated, each actor can run its system and exchange data with its partners.

Using SOA to integrate simulations involves an extra level of difficulty compared with using SOA to integrate traditional systems. Each simulation has its own notion of time and advances time at its own pace. In a distributed simulation, the advancement of time for all of the simulations must be coordinated in some way, so that a common notion of time is maintained. One approach to solving the time management problem in a service-oriented way is to create a service that will be in charge of managing the time. This forces all simulations of a group to run at the same pace.

Using this concept, a Simulation Synchronization (SimSync) Web Service has been developed to support the integration of supply chain simulations by providing time management functions.

This paper provides a detailed description of the SimSync Web Service. Section 2 presents The High Level Architecture and The Representational State Transfer architecture. Section 3 introduces the conceptual model and explains how SimSync can be used. SimSync's architecture is described in Section 4, and the time synchronization algorithm is explained in Section 5. Sections 6 and 7 detail the Java and Web Service Application Programing Interface, and Section 8 shows the graphical user interface. Section 9 presents a conclusion and proposes directions for future research.

2 Background

2.1 The High Level Architecture

SimSync borrows concepts from the High Level Architecture (HLA) [4]. HLA is an architecture for the development of distributed simulation from the Department of Defense. HLA makes use of a central bus called the Run Time Infrastructure (RTI) that offers time management functionality. HLA's RTI, however, was designed prior to the standardization of today's Web Services technologies and hence doesn't support them. Efforts to support SOAP-based Web Services in HLA are currently underway in the HLA-Evolved specification [5].

In the HLA terminology, each simulation component part of a distributed simulation is referred to as a federate. The set of simulation components forming the distributed simulation is referred to as a federation. SimSync reuses those concepts but refers to a federation as a Simulation Group and to a federate as a GroupMember.

2.2 The Representational State Transfer architecture

The Web Service API provided by SimSync follows the Representational State Transfer (REST) architecture paradigm. REST is an architecture style for integrating heterogeneous systems by managing the creation and exchange of resources that define information about those systems. Each application exposes its state as a resource, and different applications communicate by exchanging representations of those resources through a constrained set of operations. Resources are identified uniquely on a network using Uniform Resource Locators (URLs) [6]. An architecture defined in terms of the REST paradigm is referred to as a RESTful architecture. The REST architecture offers great scalability and is considered to be the architecture that made the World Wide Web successful. The different resources exposed by SimSync in its REST Application Programming Interface are detailed in Section 7 .

3 Conceptual Model and Usage

SimSync provides a Web Service that allows simulations that are intended to be a part of a distributed simulation to run at the same pace. First, when a distributed simulation is to begin, SimSync is used to create a Simulation Group. A Simulation Group represents a group of simulations that should run at the same pace. Next, each simulation participating in a run of the distributed simulation registers itself as a Member of this Simulation Group. Each Member will then periodically ask SimSync for permission to advance its local time, and SimSync will grant permission to advance to the Members in a manner such that a common notion of time is created and maintained.

SimSync by itself doesn't provide a way for simulators to exchange data. In an effort to apply the concepts of RESTful architecture to distributed simulations, SimSync focuses on providing the time synchronization service. Existing data standards and communication protocols such as the Core Manufacturing Simulation Data (CMSD) [7] and the Hyper Text Transfer Protocol (HTTP) [8] can be used to perform data transfer. Tools to facilitate data transfer using those standards are being developed in another project and are not the focus of this paper.

4 Software Architecture

The SimSync framework has been implemented in the Java programming language. The framework is implemented in terms of several software components that can be configured to provide different functional capabilities. Some of the components implement functionality that allows the creation of Simulation Groups, the joining of simulation Members to those groups, and the coordination of time advancement of the group. This is referred to as the server functionality of SimSync. Other components provide the client functionality of SimSync that allows simulations to access and interact with the functionality provided by the server. The relationships between components are described in Figure 1. Components may be configured in different ways, allowing SimSync to be run as a stand-alone server or be embedded as a library in a simulation application.

The role of each component is as follows:

- The **SimSync-API** component defines the application programming interface. Any application that wants to make use of SimSync as a library should use this API instead of a specific implementation. Developers who need to change the time synchronization algorithm used by SimSync can implement the interfaces defined in this component. The details of the API are explained in Section 6 .

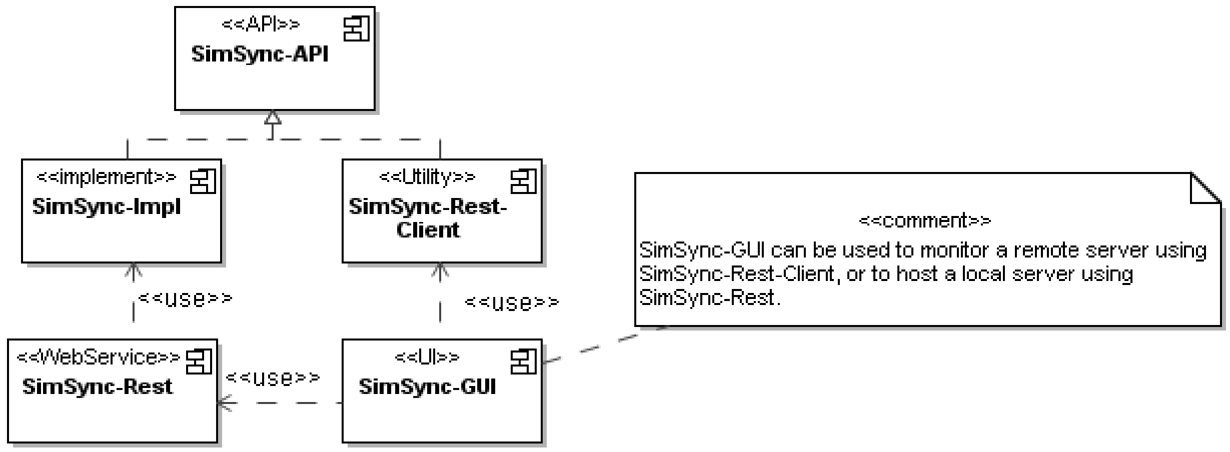


Figure 1: SimSync Components

- The **SimSync-Impl** component is the main implementation of the services defined by SimSync-API. It implements the time synchronization algorithm and is the component that contains most of the business logic for the functionality of SimSync. See Section 5 for a detailed explanation of the implemented algorithm.
- The **SimSync-Rest** component provides a Web Service API for SimSync. SimSync-Rest embeds a mini web server called Jetty [9] and can be started as a standalone application or used as a library. See section 7 for details on using SimSync-Rest.
- The **SimSync-Rest-Client** component is a client library that can be used to connect to the REST Web Service API of SimSync. This component also provides an implementation of the SimSync-API, and hence can be used as a local proxy for a SimSync server.
- The **SimSync-GUI** component contains a graphical user interface (GUI) to SimSync. It is a standalone application that can be used to either start a local SimSync server or to monitor a remote SimSync server. See Section 8 for more details on SimSync-GUI.

5 Time Synchronization Algorithm

5.1 Overview

A Discrete Event Simulation (DES) consists of a list of events that are scheduled at a given time. During a run, the simulator processes each event in chronological order. The simulator also updates the clock each time it starts processing an event. Figure 2 shows the algorithm of a DES engine in pseudo code.

In a distributed simulation, each simulation has its own list of events that are scheduled at different times. Also, the time it takes to process each event can differ in each simulation. This creates an issue when two simulations of a distributed simulation need to communicate, since each simulation might have different local time during the communication, leading to incoherent processing of messages.

```
$EventList = \{e_1, \ldots, e_n\}$
$CurrentTime = 0$
$for\ i = 0..n$
    $CurrentTime = e_i.time$
    $process(e_i)$
```

Figure 2: Typical algorithm of a Discrete Event Simulation engine

Several well known approaches exist to solve this issue, such as those present in the High Level Architecture (HLA) [10]. Each approach has benefits and trade offs and is suitable for a certain scenario. Criteria to select a time synchronization algorithm include the latency of the network, the maximum allowed time difference between simulators, and the ability of simulators to go back in time.

SimSync is designed specifically with the goal of connecting simulations of the different links of a supply chain. In order to choose the most appropriate time synchronization algorithms, the following assumptions have been made:

- Most communications between links of the supply chains are orders and orders deliveries. Since a real company might take hours to process an order and take days to deliver the order, it is acceptable if the simulated time of different simulations are off by a few hours.
- Simulations will be exchanging messages over the Internet. A round-trip message between two simulators over the Internet takes approximately between 30 ms and 300 ms (this is obtained empirically by pinging computers at different locations over the Internet).
- The resulting distributed simulations will be used to simulate several years of production of a company.
- Simulating a year of production should take less than 10 minutes.
- Commercial off-the-shelf (COTS) simulation packages do not necessarily have the ability to perform a rollback (i.e. go back in time).

Based on those assumptions, the time synchronization algorithm implemented in SimSync is as follows:

Simulators periodically ask permission to SimSync to move their clock forward. SimSync will receive a query and will only reply once it is approved. Finally, a simulator S_i will be granted permission to move its clock to the simulated time t_i once all other simulations have requested to move time past or at t_i.

This algorithm is non-optimistic, since each simulation has to wait for SimSync's approval at the end of a simulation cycle. In an optimistic time synchronization algorithm, simulators don't wait for an approval to move their clock forward and rewind their clock when they receive a message that was supposed to be processed beforehand. These algorithms generally offer better performance, especially on a high latency network such as the internet. However, the implementation of optimistic algorithms is sometimes impossible in COTS simulation packages that do not have the ability to move their clock backward in time.

In the algorithm presented above, performances of the distributed simulation are mainly affected by the frequency with which each simulator communicates with SimSync. This is due to the fact that communications over the Internet are very slow and are the main bottleneck of the system. Therefore, it is important for simulation implementers to define a frequency of synchronization that is as low as possible but appropriate for their models.

Finally, it is important to notice that as previously stated in Section 4, the time synchronization algorithm in SimSync can be replaced. Developers who would like to experiment with other time synchronization algorithms can provide their own algorithm by implementing the SimSync API and replacing the SimSync-Impl component.

5.2 Example

Figure 3 is a UML sequence diagram that shows an example of the messages exchanged to perform time synchronization during a run. In this example, two simulations, called sim1 and sim2, are advancing at different paces and ask SimSync permission to advance their local time. This example doesn't show the messages exchanged during the initialization of the distributed simulation and only shows messages exchanged during a small time period.

On the left side of the diagram, the two vertical time lines represent the two simulations sim1 and sim2. On the right side of the diagram, the two vertical time lines represent the two group-member resources that represent sim1 and sim2 inside SimSync. For more details on the GroupMember Application Programing Interface, see Section 7.1.3.

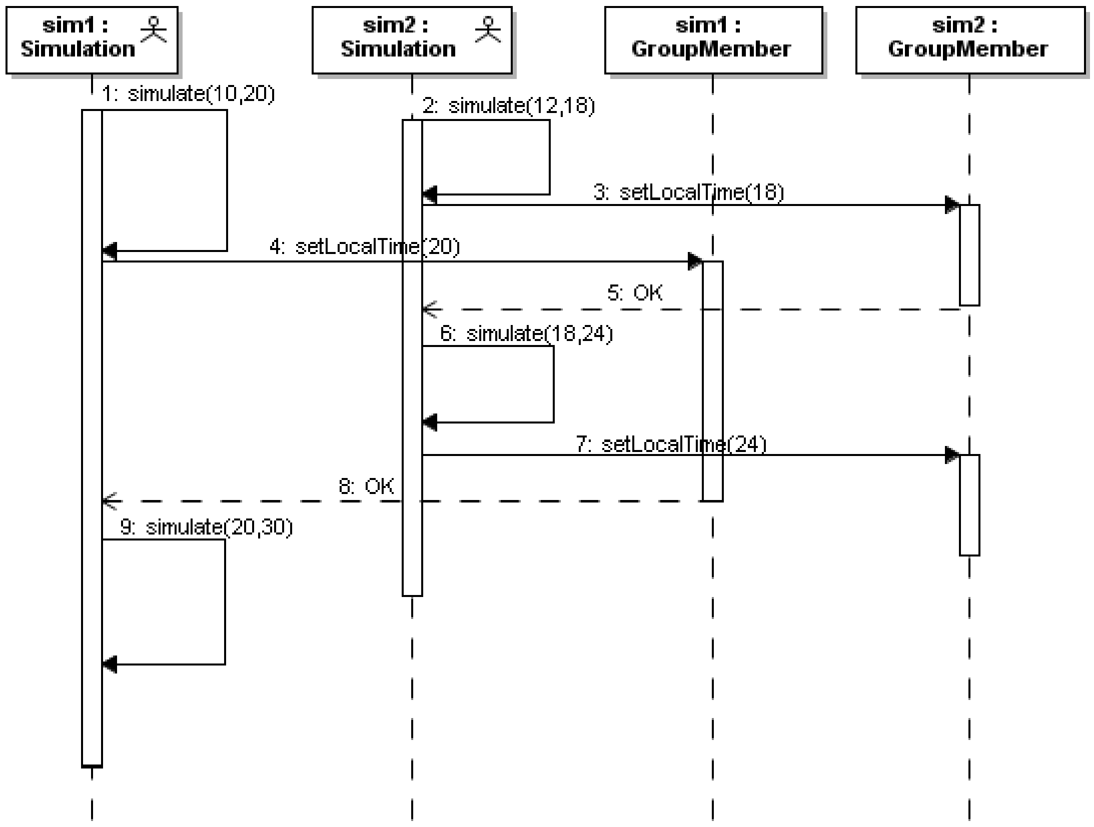


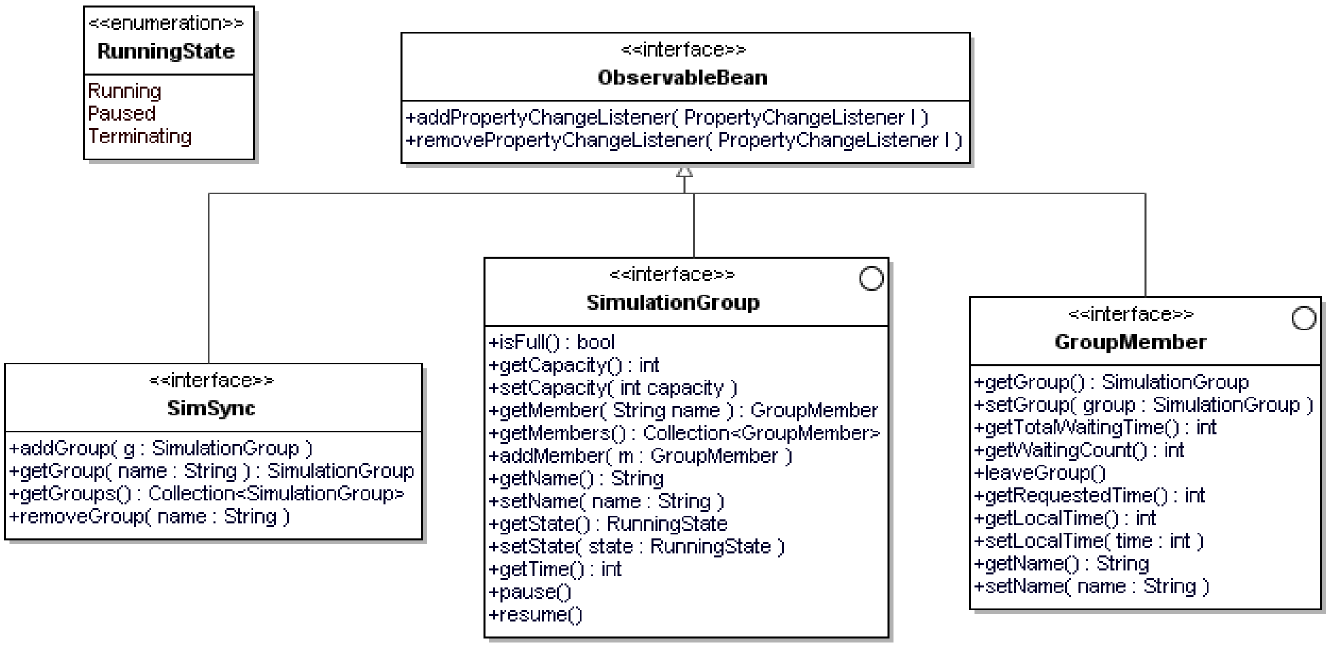
Figure 3: Example of messages sent during a run with 2 simulations.

At the beginning of this example, both sim1 and sim2 are simulating locally (messages 1 & 2). Sim1's local time is going from 10 to 20 while sim2's local time is going from 12 to 18. When sim2's local time reaches 18, it asks SimSync permission to set its local time to 18 (message 3). SimSync doesn't reply right away since it is waiting for other simulations of the group to arrive at the same point in time. When sim1's local time reaches 20, it asks SimSync permission to set its local time to 20 (message 4). This tells SimSync that both sim1 and sim2 have a local time past or at 18. SimSync then sends sim2 authorization to move its local time to 18 (message 5).

Sim2 then starts simulating again, until its local time reaches 24 (message 6). It then asks SimSync permission to set its local time to 24 (message 7). This notifies SimSync that sim2's local time is past 20 and SimSync sends authorization to sim1 to move its local time up to 30 (message 8). Sim1 then starts simulating again until its local time reaches 30 (message 9).

6 Java Application Programming Interface

The SimSync API has 3 main interfaces: SimSync, SimulationGroup, and GroupMember. The main class, called SimSync contains Simulation Groups. A Simulation Group represents a group of simulations that would like to advance at the same pace. A Simulation Group itself is then composed of Group Members, representing each individual simulation. Each class in SimSync follows the JavaBean convention [11] and implements the ObservableBean interface. The ObservableBean interface represents a JavaBean whose properties can be monitored by registering PropertyChangeListeners. Figure 4 shows SimSync's interfaces and their relationships.



<<enumeration>>
RunningState
Running
Paused
Terminating
<<interface>>
ObservableBean
+addPropertyChangeListener(PropertyChangeListener l)
+removePropertyChangeListener(PropertyChangeListener l)
<<interface>>
SimSync
+addGroup(g : SimulationGroup)
+getGroup(name : String) : SimulationGroup
+getGroups() : Collection<SimulationGroup>
+removeGroup(name : String)
<<interface>>
SimulationGroup
+isFull() : bool
+getCapacity() : int
+setCapacity(int capacity)
+getMember(String name) : GroupMember
+getMembers() : Collection<GroupMember>
+addMember(m : GroupMember)
+getName() : String
+setName(name : String)
+getState() : RunningState
+setState(state : RunningState)
+getTime() : int
+pause()
+resume()
<<interface>>
GroupMember
+getGroup() : SimulationGroup
+setGroup(group : SimulationGroup)
+getTotalWaitingTime() : int
+getWaitingCount() : int
+leaveGroup()
+getRequestedTime() : int
+getLocalTime() : int
+setLocalTime(time : int)
+getName() : String
+setName(name : String)

Figure 4: SimSync's main classes

The following sections describe each method of the different interfaces of the Java API.

6.1 The SimSync interface

`void` **`addGroup`**`(SimulationGroup group)`

Add a group to this SimSync instance.

Parameters:
`group` - The Group that will be added to this instance of SimSync.

`SimulationGroup` **`getGroup`**`(java.lang.String name)`

Get the simulation group that has the specified name.

Parameters:
`name` - The name of the simulation group.

Returns:
The SimulationGroup with the specified name, or null if no SimulationGroup has the given name.

`java.util.Collection<SimulationGroup> getGroups()`

Get a collection of the groups that are registered in this instance of SimSync.

Returns:
A collection containing all the groups registered in this instance of SimSync.

`void removeGroup(java.lang.String name)`

Unregister the group with the specified group name.

Parameters:
`name` - The name of the group.

6.2 The SimulationGroup interface

`boolean isFull()`

Returns a 'true' value if the group has reached his expected number of members, otherwise return a 'false' value.

Returns:
true if the capacity equals the number of members of the group.

`int getCapacity()`

Get the capacity of this group. The capacity is the number of members that are expected in a group for the group to be full.

Returns:
The capacity of the group.

`void setCapacity(int capacity)`

Set the capacity of the group.

Parameters:
`capacity` - The capacity of the group.

`GroupMember getMember(java.lang.String memberName)`

Return a group member that has a specified member name.

Parameters:
`memberName` - The name of the group member.
Returns:
The GroupMember registered with the given name.

`java.util.Collection<GroupMember> getMembers()`

Return a collection containing all the members of this group.

Returns:
A collection containing all the members of this group.

`void` **`addMember`**`(GroupMember member)`

Add the given member to this group.

Parameters:
`member` - The member that needs to be added to this group.

`java.lang.String` **`getName`**`()`

Get the name of this group.

Returns:
The name of this group.

`void` **`setName`**`(java.lang.String name)`

Set the name of this group.

Parameters:
`name` - The name of this group.

`RunningState` **`getState`**`()`

Get the current state of this group.

Returns:
The current state of this group.

`void` **`setState`**`(RunningState state)`

Set the current state of this group.

Parameters:
`state` - The new state of this group.

`int` **`getTime`**`()`

Get the current time of this group. Note that the time of a group might be calculated in different ways depending on the algorithm being used. Also, the time of a group doesn't necessary match the local time of any of its members.

Returns:
The current time of this group.

`void` **`pause`**`()`

Put the group in the *Paused* state.

`void` **`resume`**`() throws java.lang.Exception`
Put the group in the *Running* state. This method will throw an exception if the group hasn't reached its full capacity.

Throws:
`java.lang.Exception` - If the group hasn't reached its full capacity.

6.3 The GroupMember interface

`SimulationGroup` **`getGroup`**`()`

Get the group that this member belongs to.

Returns:
The group that this member belongs to.

`int` **`getLocalTime`**`()`

Get the current local time of this group member. The local time of a member can be different from its requested time if the member is currently trying to advance time (by calling *setTime(int time)*).

Returns:
The local time of this member.

`int` **`getRequestedTime`**`()`

Get the requested time of this member. The requested time is different from the local time if the method *setTime(int time)* was called and hasn't returned yet.

Returns:
The current requested time of this group member.

`java.lang.String` **`getName`**`()`

Get the name of the group member.

Returns:
The name of the group member.

`long` **`getTotalWaitingTime`**`()`

Get the total time this member spent on waiting for other members. This method can be used to find out which member of a group slows down other simulations.

Returns:
The total time this member spent waiting for other members.

`int` **`getWaitingCount`**`()`

Get the total times this member has been asked to wait for other simulators before it gets approval to move forward in time. This method can be used to find out which simulator slows down the group.

Returns:
An integer indicating how many times this member has been asked to wait for other simulators before it gets approval to move forward in time.

`void` **`leaveGroup`**`()`

Remove the group member from its group. This will put the group in the *Terminating* state.

`void` **`setGroup`**`(SimulationGroup group)`

Set the group that this member belongs to. This method can be called only once.

Parameters:
`group` - The group that this member belongs to.

`void` **`setLocalTime`**`(int localTime)`

Set the group member's local time. When SimSync decides to authorize the member to move to its new local time, the requested time of the group member is changed instantly, otherwise this method is blocked.

Parameters:
`localTime` - The new local time of this group member.

`void` **`setName`**`(java.lang.String name)`

Set the name of this member.

Parameters:
`name` - The name of this member.

6.4 The ObservableBean interface

`void` **`addPropertyChangeListener`**`(java.beans.PropertyChangeListener l)`

Add a property listener to this bean. The listener is notified when the value of a property changes.

Parameters:
`l` – The listener that will be notified when any property of this bean changes.

`void` **`removePropertyChangeListener`**`(java.beans.PropertyChangeListener l)`

Unregister the given listener from this bean. The listener will no longer be notified when a property of this bean changes.

Parameters:
`l` – The listener that will be removed from this bean.

6.5 The RunningState enumeration

`public static final RunningState` **`Running`**

Indicates that the simulation is currently running.

`public static final RunningState` **`Paused`**

Indicates that the simulation group is paused. All calls to move the time forward will be blocked until the simulation group gets to running state.

`public static final RunningState` **`Terminating`**

Indicates that a member of the simulation group asked to leave the group. Simulations are not allowed to move their time forward while the group is in the *Terminating* state and should leave the group.

7 The REST Web Service API

This section describes the Web Service API of SimSync. The API follows the REST architectural style. Each resource has a uniform interface based on the methods present in the HyperText Transfer Protocol (HTTP):

- The HTTP GET method is used to obtain a representation of a resource.
- The HTTP PUT method is used to create a new resource or update the value of a resource.
- The HTTP DELETE method is used to remove a resource.

The Uniform Resource Locators (URL) which are used to identify SimSync's resources as well as the methods and available representations, are described in Section 7.1 , while Section 7.2 provides a detailed description of each representation.

7.1 Resources

Below is a description of each resource available in SimSync. For each resource, accepted methods, available representations, the status code returned and a short description are specified. Parameters in Urls are shown between braces. The methods used on those resources are standard HTTP methods and the status codes returned by those methods are standard HTTP status codes.

7.1.1 SimulationGroups

URL: /simgroups/

The root of the API. Each resource used by SimSync has a Url that starts with the prefix /simgroups/.

Method	Available/Acceptable Representations	Status Code	Description

GET	text/plain (SimulationGroups)	HTTP_200(OK)
		HTTP_404(Not Found)
			Provide a list of the groups that are registered in this instance of SimSync.

7.1.2 SimulationGroup

URL: /simgroups/{groupName}

The group of simulations. The parameter *{groupName}* is the name of the simulation group.

Method	Available/Acceptable Representations	Status Code	Description

GET	text/plain (SimulationGroup)
	application/json (SimulationGroup)
		HTTP_200(OK)
		HTTP_404(Not Found)
			Obtain a representation of a simulation group.

PUT	application/json (SimulationGroup)	HTTP_201(Created)
		HTTP_200(OK)
		HTTP_404(Not Found)
			Create a new simulation group or update information of an existing group.

DELETE		HTTP_204(No Content)
		HTTP_404(Not Found)
			Delete the group with the name specified in the URL.

7.1.3 GroupMember

URL: /simgroups/{groupName}/members/{memberName}

The simulation, that is a member of a particular group. The parameter*{groupName}* is the name of the group and the parameter *{memberName}* is the name of the simulation in this group.

Method	Available/Acceptable Representations	Status Code	Description

GET	application/json (GroupMember)	HTTP_200(OK)
		HTTP_404(Not Found)
			Obtain a representation of a Group Member.

PUT	application/json (GroupMember)	HTTP_201(Created)
		HTTP_200(OK)
		HTTP_404(Not Found)
			Put a simulation in a group or update information on an existing simulation. This method is most frequently used to update the local time of a simulation. When a simulation wants to move forward in time, it SHOULD ask SymSync permission to change its local time. To do so, the simulation should call the PUT method with its new local time as a parameter. The PUT method will return only when SimSync approves to update the local time.

DELETE		HTTP_204(No Content)
		HTTP_404(Not Found)
			Remove this group member from its group. The group will be placed in the 'Terminating' state.

7.2 Representations

In a REST architecture, the state of resources is communicated by means of representations. SimSync currently supports the JavaScript Object Notation (JSON) [12] as representations of resources. Plain text representations also exist for some resources but are only useful for debugging purpose. Other types of representations such as XML will be added later on if needed. Each type of representation is identified using Multipurpose Internet Mail Extensions (MIME) type [13].

text/plain (Simgroups)

Provide a textual description of the content of the SimSync instance.

text/plain (SimulationGroup)

A textual representation of the content of the simulation.

application/json (SimulationGroup)

A JSON representation of the simulation Group.

Example: {"full":true,"name":"group1","state":"Running","capacity":2}

Parameters:

Parameter	Value	Description
name	(required)	The name of the group.
capacity		The maximum number of members in the group.
state	*One of:* • `Running` • `Paused`	The current state of the simulation group. Can be one of: `Running`

	• Terminating	Indicates that the group of simulations is currently running.
		
		`Paused`
		
		Indicates that the group of simulations is currently paused.
		
		`Terminating`
		
		Indicates that the group is currently terminating. No action can be performed on the group and all remaining members should leave the group.

full		Boolean indicating whether the group has reached its maximum capacity. This parameter is read-only.

application/json (GroupMember)

A JSON representation of a simulation who is a member of a particular group.

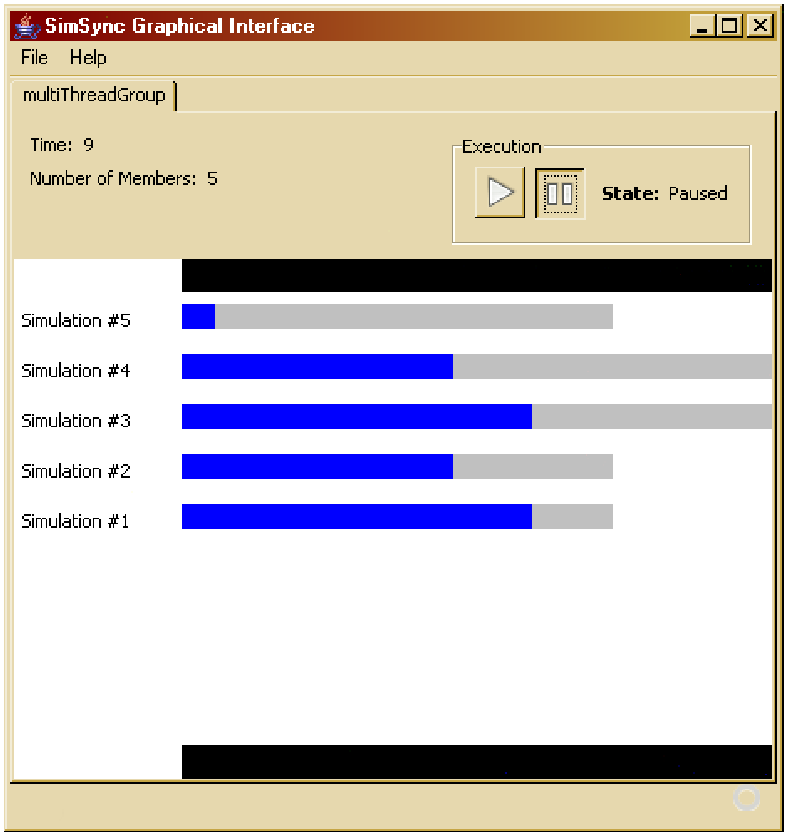
Example: {"groupName":"group1","name":"member1","localTime":0}

Parameters:

Parameter	Value	Description
groupName		The name of the group that this simulation is member of.
name		The name of the simulation.
localTime		The local time of the simulation.

8 Graphical User Interface

SimSync has a graphical user interface called SimSync-GUI. It can be used to monitor the state of a running instance of SimSync, or to start a new SimSync server. The GUI is a standalone desktop application that has been developed using the Swing framework [14].



Time: 9
Number of Members: 5
State: Paused

Figure 5: SimSync Graphical User Interface

Figure 5 shows the main window of SimSync-GUI. The content of each simulation group is displayed in a separate tab. The current estimated time of the group and the number of members in the group are shown at the top of the tab. In the top right corner, a panel called “Execution” shows the current state of the group and provides two controls: a “play” button to set the group in the *Running* state, and a “pause” button to set the group in the *Paused* state. The central part of the window shows the time advancement of each simulation. A blue bar shows the current time of a simulation and a gray bar shows the requested time of a simulation.

When SimSync-GUI is used to start a local server, the graphical user interface is updated in real time. New simulations appear in the central component as soon as they get connected, and the blue and gray bars expand as the time of each simulation evolves. This feature is not available when monitoring the state of a remote SimSync server, since this would have a significant impact on the underlying server's performance.

9 Conclusion

In this paper, we presented a first piece of software that can be used to leverage RESTful Web Services in distributed simulations. No official release of SimSync has been done so far, but the source code can be downloaded at http://sourceforge.net/projects/simrest/ . SimSync is being developed as a “proof of concept” software in an effort to improve interoperability between simulation packages. As such, it shouldn't be used in a production environment.

Future work will include the development of integration components for Commercial Off The Shelf simulation packages such as Rockwell Arena and Delmia Quest. Those components will use SimSync for time synchronization and will make use of REST Web Services to perform data transfer. The set of tools will then be used to create a real test case of virtual supply chain integration that will be detailed in another paper.

Acknowledgments and Disclaimer

The work described in this paper was sponsored by the National Institute of Standards and Technology (NIST) and is not subject to copyright. Other participants in the project include Damien Bertot and Benjamin Raverdy. Mention of commercial products or services in this paper does not imply approval or endorsement by NIST, nor does it imply that such products or services are necessarily the best available for the purpose.

References

[1] OASIS SOA-RM TC, Reference Model for Service Oriented Architecture 1.0, 2006
[2] W3C, SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007
[3] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, 2000
[4] F. Kuhl, R. Weatherly, J. Dahmann, Creating Computer Simulation systems: An Introduction to the High Level Architecture, 1999
[5] B. Moller, K. Morse, M. Lightner, R. Little, R. Lutz, HLA Evolved - A Summary of Major Technical Improvements,
[6] Tim Berners-Lee, Uniform Resource Locators (URL), 1994
[7] SISO, Standard for: Core Manufacturing Simulation Data - UML Model, 2009
[8] R. Fielding et al., Hypertext Transfer Protocol -- HTTP/1.1, 1999
[9] Jetty - Quick Start Guide (http://wiki.eclipse.org/Jetty/Starting/Quick_Start_Guide)
[10] C. Carothers, R. Fujimoto, R. Weatherly, A. Wilson, Design and Implementation of HLA Time Management in the RTI Version F.0, 1997
[11] Sun Microsystems, JavaBeans (TM) Specification, 1997
[12] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), 2006
[13] N. Borenstein, Bellcore, N.Freed, RFC 1521: MIME (Multipurpose Internet Mail extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies, 1993
[14] John Zukowski, The definitive guide to Java Swing, 2005