Towards a Standardized Federate Protocol for HLA 4

Björn Möller
bjorn.moller@pitch.se

Mikael Karlsson
mikael.karlsson@pitch.se

Fredrik Antelius
fredrik.antelius@pitch.se

Pitch Technologies
Repslagaregatan 25
582 22 Linköping, Sweden

Keywords: HLA, Interoperability, Protocol

ABSTRACT: *HLA is a powerful interoperability standard with a rich set of services for information exchange, synchronization and management of federations. These services are accessed through a local RTI library, installed on the same computer as the simulation itself. As new and improved RTI versions are released, or if the user wants to switch RTI supplier, these libraries need to be replaced. What if there were instead a simple protocol that a simulation could use to access the HLA services?*

This paper proposes such a protocol for HLA 4. It partly builds on experiences from the Web Services API in HLA Evolved. The WS API proves the concept, but has several shortcomings dues to its use of blocking calls and use of XML. An optimized, streaming, binary protocol is instead suggested. Such a protocol would make it easy to add a small and generic library to any federate. Switching RTI libraries would then be a simple operation of connecting to a different network address.

Additional advantages are that it makes it easy to provide native HLA support for any language, like C# or Python, to execute in CPU-constrained or hard real-time environments, to communicate in mobile environments, like 3G or 4G, or even to embed HLA support in hardware equipment. It can also be used to avoid re-accreditation of simulations, since the accredited simulator need not be updated.

Some design considerations include discovery, session management and latency handling.

Early test implementations have shown performance close to current RTI performance and improved fault tolerance over WAN links. To be able to easily swap between different RTI implementations, a standardized protocol is now being proposed to the HLA 4 Product Development Group.

1. Introduction

The High-Level Architecture (HLA) [1], standardized as IEEE-1516, is an interoperability standard for building distributed simulations. It provides services for exchanging information, coordination and synchronization as well as management of participating simulations. Some examples of the functionality that the HLA services provide are:

- **Information exchange**: Publish/subscribe of objects with attributes and interactions with parameters.
- **Coordination and synchronization**: Time managed exchange of time-stamped data. Federation-wide synchronization using synchronization points. Transfer of ownership.
- **Management**: Managed set of federates. Monitoring of federates, services usage and data exchange.

The recommended way for describing the logical structure of simulations that interoperate using HLA is the "lollipop" diagram, as shown in Figure 1.

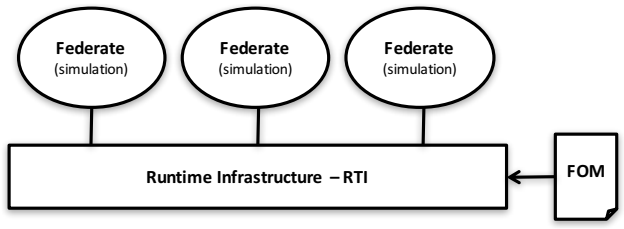

Figure 1: Lollipop diagram of an HLA federation

Each participating system is called a federate and together they form a federation. Each federate connects to the Run-time Infrastructure (RTI) that provides the HLA services defined in the standard. In addition to this, a Federation Object Model (FOM) is used. This is an XML file with federation specific information, such as the types of objects and interactions that are used in a particular domain.

The HLA standard was originally developed in the 90's and is revised regularly. The most recent version was released in 2010. A new version, currently nick-named "HLA 4", is planned to be released 2018-2019.

1.1. Providing HLA services using an API

The HLA services are defined in a generic format, with parameters, return values, preconditions and postconditions. In addition to the generic specification, there are bindings for the C++ and Java programming languages. In order for a federate to use HLA, a C++ or Java library that implements these services is required. This is called the Local RTI Component (LRC). In most cases a separate application, called the Central RTI Component (CRC) is also used, in particular for centrally coordinated services, such as keeping track of the current set of joined federates and their logical time. This means that the physical deployment of a typical HLA federate looks like Figure 2.

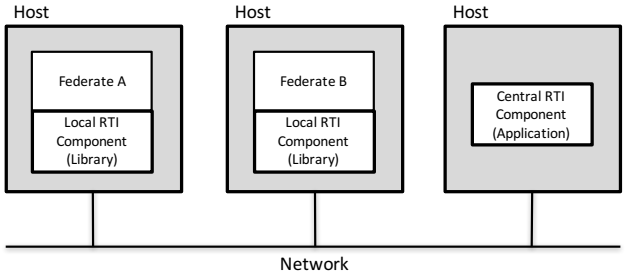

Figure 2: Physical deployment of an HLA federation

A federate makes calls and get callbacks from the LRC library through one of the APIs. The complexity of network communication and synchronization with other federates are handled by the RTI. This architectural style is also known as message-oriented middleware [2].

When developing an RTI and implementing the services, distributed algorithms for proper information exchange and synchronization must be designed. A network protocol that supports these algorithms must also be designed. Different RTI implementations have chosen different algorithms and therefore different protocols and refined them over time for performance, scalability and fault tolerance. Most RTIs use TCP/IP, in many cases with UDP and multicast, but RTIs based on shared or reflective memory have also been developed.

This means that the network protocol is different between different RTI implementations, and often also between different versions of the same RTI. It is thus necessary to have the same version of the RTI library installed for every federate in a federation.

When the initial HLA specification was developed, it was decided to specify HLA as a set of services, and to not specify a protocol. This would enable evolution and innovation, while avoiding lock-in into a particular technology or implementation. Going from a protocol standard, like the predecessor DIS [3], to a services standard, based on message-oriented middleware, caused some criticism within the SISO community.

However, the technical evolution in performance and robustness that has taken place in RTI implementations over the last 20 years wouldn't have been possible if a protocol was specified in the 90's by a standards committee. It is also unlikely that most federate developers

would have been able to use such a protocol directly, without some kind of middleware.

1.2. Providing HLA Services over a Protocol

Still, for some purposes, a standardized protocol can be an advantage. What if the HLA services themselves were provided as a protocol, instead of as an API? In that case, different federate implementations could access different RTI implementations, just like different web browser implementations can access different web server implementations, as long as they properly implement the standardized protocol.

This is the approach that this paper seeks to investigate. In this case, a federate could call an RTI component over a standardized protocol. RTI implementations could still use different algorithms inside. The federate and its technical environment could be separated from the RTI implementation. The difference between the API approach and the protocol approach is shown in Figure 3.

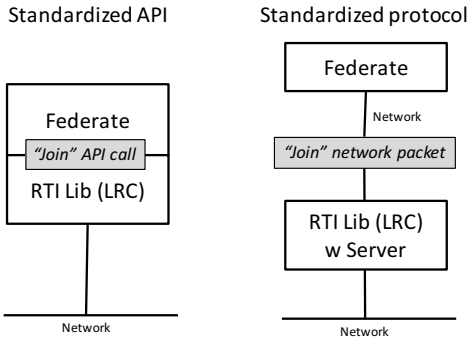

Figure 3: API versus Protocol

In the API case, shown to the left, a service, in this case Join Federation Execution, is called using an API to an LRC library executing in the same process and on the same host. In the protocol case, shown to the right, the same service is called using a communications protocol to an LRC executing in a different process. What traditionally was the LRC can now operate on the same or different host. Due to the smaller software stack required in the federate process, this approach has sometimes been nick-named "Thin HLA". The approach allows for several different topologies, as shown in Figure 4.

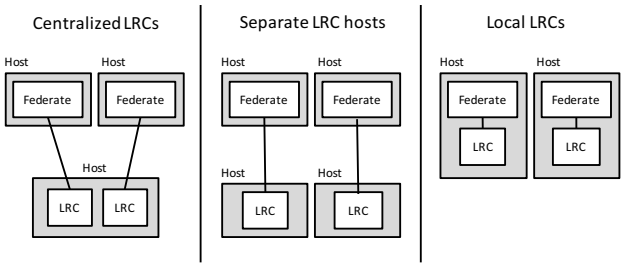

Figure 4: LRCs in different topologies

The full RTI implementation can be installed either centrally, on separate hosts or the same host as the federate. It is also possible to mix these deployments.

1.3. Building upon the Web Services API

Such a protocol was indeed introduced in HLA Evolved (IEEE 1516-2010). It is based on Web Services and is defined using the Web Services Description Language (WSDL). The Web Services API [4] implements all HLA services. It can run over http or https, including authentication. The performance is acceptable, typically thousands of updates per second, is useful for some implementations, but not all. It proves that implementing the HLA services as a protocol is feasible.

At the same time, it isn't optimal for many applications. Calls are made using a request-response pattern and the HTTP protocol. This means that, after sending an update, it blocks and waits for a response, which may be time consuming. The requests and responses are encoded using XML and SOAP, which requires a lot of CPU resources and additional libraries to process, and may, in some cases, be cumbersome to handle in an application.

1.4. Towards a Binary Format

This paper proposes a simplistic binary format that would build on experiences from the WSDL API but avoid some drawbacks. The protocol would use a more efficient encoding with smaller footprint. It would not necessarily be blocking. This would mean that updates can have "streaming" characteristics. It would also be easier to implement, for example in resource constrained environments.

2. Opportunities with a Federate Protocol

This protocol would make it possible or easier to build solutions that are difficult or impossible today. Below are a number of use cases for a federate protocol.

2.1. Support any programming language or OS

Programming languages come and go over time. As new languages become popular for simulation development, users request HLA support for them. A federate protocol would make it considerably easier to implement HLA

support in any language, like C#, Pearl or Python. Today, adding a new programming language either means porting the LRC library to another programming language or creating a bridge between code in two different languages. Instead, the federate could make these calls using a binary protocol, possibly with a small reusable stack for each new language.

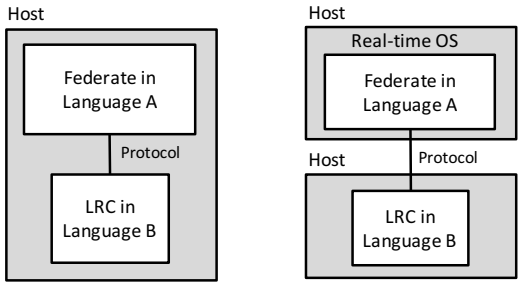

Figure 5: Different implementation languages or OSs

The effort to implement HLA support for a new programming language would be significantly reduced. This approach is also useful for deployment in specialized operating systems, for example real-time operating systems or embedded systems, where it may be difficult or undesirable to deploy the entire LRC functionality.

2.2. Support Live simulation and unreliable links

Another use case is for live simulation and any situation where the network connection has limited bandwidth and may be unreliable. The full LRC is deployed on the network backbone, with reliable connection to the rest of the federation. Only the data that the federate actually needs will be sent over the unreliable link, as shown in Figure 6.

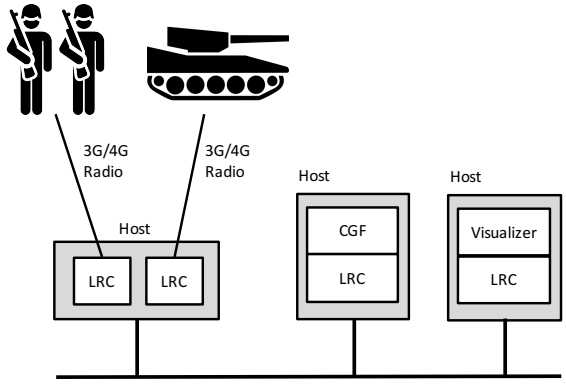

Figure 6: Live Simulation using a Federate Protocol

The fault tolerance is significantly improved. In case of temporary disconnects, the state of the LRC isn't lost. Another advantage is that the protocol would only need one point-to-point connection, which is preferred for communication over for example mobile Internet links.

2.3. Mix Cloud and local resources

Another use case is to deploy the LRC as a service in the Cloud, for example with a commercial cloud provider or a private cloud. This would enable federates to connect from many different geographical locations, even behind firewalls, and form a federation as shown in Figure 7

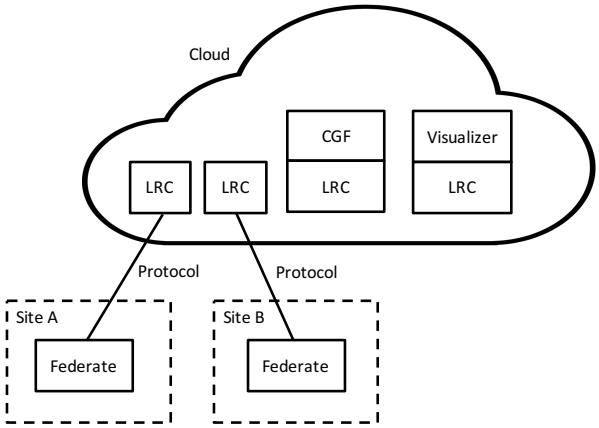

Figure 7: Cloud deployment using a Federate Protocol

Some federates, such as Computer Generated Forces (CGFs), could run in the Cloud environment. Other federates, for example virtual simulations, could run locally.

2.4. Simplify switching between RTIs

To switch between different RTI implementations or to upgrade from one version to another today requires installing new software on the same computer as the federate. If several different RTI implementations need to be available, it is common to perform parallel installations and to provide different search paths to the federate application. With a federate protocol, it is only necessary to provide the network address of a different RTI

2.5. Reduce cost for accreditation

Today, when a federate has already been accredited, it may be a major and costly effort to switch RTI version or RTI provider. Such a switch may be necessary if an accredited simulator needs to participate in a federation that use a different RTI implementation. If the federate protocol is used, no files need to be installed or replaced on the accredited simulation host.

2.6. Security

A federate protocol is useful for secure simulation in at least two ways since it can make both the network communications and the protocol stack easier to inspect.

When running federates in the same federation in different security domains, the data exchange may need to be inspected at the security domain borders. A standardized

federate protocol makes it more attractive to build reusable or even COTS/GOTS solutions for this purpose.

The Local RTI Components of an RTI contain a lot of vendor-specific, proprietary code that may not be easily available for auditing and inspection. This means that the LRCs may not be approved for use in certain high-security environments.

The client side of the Federate Protocol can be quite small, even constructed in-house, thereby making it possible to use HLA in a high-security environment.

2.7. Add native HLA support to hardware equipment

With a standardized federate protocol it is easier to include HLA support inside hardware equipment, for example sensors.

3. Lessons Learned from Earlier Work

This section takes a deeper look at lessons learned from practical experimentation.

3.1. Lessons learned from the WSDL protocol

The Web Services API hasn't been widely adopted by users, although RTIs that implements it exist and it has been mentioned in numerous academic papers. Federate development is somewhat cumbersome. There is good support in many languages for making Web Services calls, but not for parsing the RTI callbacks that are returned from an Evoke Callback service call. Processing a high rate of XML calls and callbacks consumes a lot of CPU cycles, which in many cases is a bigger problem than the bulkiness of XML.

Another, not so obvious issue, is that the http request/response pattern is blocking which means that each request must be confirmed by a response before a new request can be made. This makes it sensitive to latency. As an example, with a round-trip latency of 10 ms it is only possible to make 100 calls per second. HLA service calls may depend on each other so they cannot be performed in parallel. As an example, the result of calling Join Federation Execution and Send Interaction in parallel would be unpredictable.

Mixing different programming languages between the federate and the LRC has been tested using the WSDL API. One use case is for supporting C# using the Web Services API. It worked well, but parsing call-backs was cumbersome, as previously described.

3.2. Issues with moving to a protocol

When an API is used, many syntactic errors, like providing the wrong number of parameters, are detected during development and compilation. In a protocol-based solution, these kinds of errors may occur at runtime and create a need for new types of error messages. To minimize problems like this, e.g. missing parameters, malformed messages, etc, it is popular to develop and reuse software stacks, which would be a good idea also for a federate protocol. For a protocol, debugging is considerably more difficult, since the data is exchanged over a network as a one piece of raw data.

3.3. Lessons learned from testing in Live environments

A prototypical federate protocol has been tested in an application with live entities. Several vehicles in a test range were connected using GPRS/3G mobile broadband. The traditional LRC topology and the "thin HLA" topology were both tested. The federation contained around one hundred vehicles using the RPR FOM [5]. The most interesting observation was when the 3G link suffered interrupts every now and then. The time to re-join the federation, for a federate that lost connection, was in the range of minutes for the traditional LRC approach, but only a few seconds when using the federate protocol.

The challenge here is to maintain the integrity of the distributed state, since the Local RTI Components need to stay in constant touch with each other. Whenever the connection between one LRC and other LRCs is lost, e.g. due to a temporary disruption, the LRC may need to resign and re-join the federation to ensure correct state. This puts a very heavy burden on the federation.

Using the federate protocol, this problem can be solved by locating the LRC on a reliable network and then have the federate connect to the LRC over the less reliable network. Any disruptions on the less reliable network will not cause the LRC to drop out of the distributed state.

4. Design Challenges

Developing a binary protocol based on the experiences from the Web Services API is quite straightforward. There are however some challenges that need to be solved.

4.1. Supporting different environments

The technical approach chosen shall have support for a large number of technical environments. If a code generator for encoding/decoding is used, it shall support many relevant programming languages. If communications or encoding libraries are required at runtime, these shall be available in as many relevant environments as possible. If possible, the use of such libraries shall be optional .

4.2. Optimized encoding

The encoding/decoding and data exchange shall require as little CPU and memory as possible. The need for copying of data between buffers shall be minimized.

4.3. Optimizing for high-latency environments

A protocol should be able to handle high-latency environments. One important approach is to enable federates to send many updates and interactions, without necessarily waiting for the outcome of a previous call, i.e. "blocking calls". This would create an asynchronous behaviour in the protocol, as shown in Figure 8.

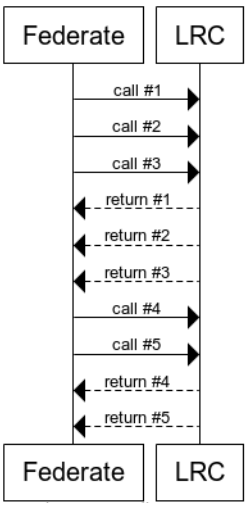

Figure 8: Asynchronous mode

This can be seen as postponed exception checking. A federate may want to perform some operations, like Join Federation Execution, with a conservative approach (waiting for a response), but others, like updates, with postponed exception checking.

4.4. Optimizing for fault tolerance

The federate protocol should also be designed for maximum fault tolerance. It could build upon TCP in order to get automatic retransmission of lost packages. Faults either need to be handled or propagated. Faults that could be handled include recovery from a broken TCP connection. This could be achieved by introducing a session token and retransmission of unconfirmed packages (unless they use best effort transportation).

5. Discussion

5.1. Need for a standard

While proprietary versions of a binary HLA federate protocol have been developed, a standardized protocol has several advantages. It would make it possible for federate developers to switch RTIs, and avoid vendor lock-in. It would make it more attractive to developers to use the protocol. It would also capture the expertise of more HLA experts during the design. It would increase the support for more technical environments without needing to rely on a particular RTI supplier to support each effort.

5.2. Reuse the standard C++ and Java APIs

Considering a client software stack that implements the HLA federate protocol, should this client software have the same API as the programming language bindings in the HLA standard. For previously unsupported languages, like C#, Perl or Python, there are no such APIs, so this is less of a concern. For C++ and Java, there are several advantages with keeping the same API. A federate can then easily switch between a standard LRC and the "Thin HLA" software stack. The drawback is that this may force the stack to use blocking calls, which would reduce the performance significantly, compared to the asynchronous approach, where the calls aren't blocking.

5.3. Asynchronous API

An API that supports the asynchronous model could be very useful for federate developers. Such an API could be closely aligned with existing APIs or it could be very different. One approach would be to reuse the current API but to introduce a delayed exception callback that reports exceptions asynchronously.

5.4. Relationship to WebLVC

A SISO standard called WebLVC [7] is currently under development. The main focus is as follows: "WebLVC is a protocol for enabling web and mobile applications (typically JavaScript applications running in a web browser) to play in traditional M&S federations (which may be using Distributed Interactive Simulation (DIS), High Level Architecture (HLA), Test and Training Enabling Architecture (TENA), or related protocols and architectures)" [8]. A typical WebLVC protocol message presents the data of an object instance, such as a military platform or an interaction, such as a firing event, in Javascript Object Notation (JSON) [8] notation.

The authors would like to argue that the purpose and scope of WebLVC and the proposed federate protocol are very different. The WebLVC standard is best suited for providing data for a particular domain, like defense, to a web browser. The proposed federate protocol, on the other hand, provides the entire set of HLA services to many different environments for any domain, but may not provide the convenience of JSON for Javascripts applications in a web browser.

6. Conclusion and Road Ahead

A standardized HLA federate protocol offers several new opportunities as well as solutions to several issues that exists with HLA today. The most important ones are better support for Cloud deployment, Live applications and additional programming languages.

As HLA is currently under revision, a formal comment proposing a federate protocol has been submitted. This comment includes forming a Tiger Team to capture the input from interested users, vendors and academia.

References

[1] IEEE: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)", IEEE Std 1516-2010, IEEE Std 1516.1-2010, and IEEE Std 1516.2-2010, www.ieee.org, August 2010.

[2] “Message Oriented Middleware”, Wikipedia, retrieved 2017-06-26

[3] IEEE: “IEEE Standard for Distributed Interactive Simulations”, IEEE Std 1278.1-2012, www.ieee.org, December 2012

[4] SISO: “SISO-STD-001.1-2015, Standard for Real-time Platform Reference Federation Object Model (RPR FOM)”, www.sisostds.org, September 2015.

[5] Björn Möller et al.: “RPR FOM 2.0: A Federation Object Model for Defense Simulations”, 2014 Fall Simulation Interoperability Workshop, (paper 14F-SIW-039), Orlando, FL, 2014.

[6] WebLVC, www.sisostds.org

Author Biographies

BJÖRN MÖLLER is the Vice President and co-founder of Pitch Technologies. He leads the development of Pitch's products. He has more than twenty-five years of experience in high-tech R&D companies, with an international profile in areas such as modeling and simulation, artificial intelligence and web-based collaboration. Björn Möller holds a M.Sc. in Computer Science and Technology after studies at Linköping University, Sweden, and Imperial College, London. He is currently serving as the chairman of the Space FOM Product Development group and the vice chairman of the SISO HLA Evolved Product Development Group. He was recently the chairman of the SISO RPR FOM Product Development Group.

FREDRIK ANTELIUS is a Senior Software Architect at Pitch and is a major contributor to several commercial HLA products, including Pitch Developer Studio, Pitch Recorder, Pitch Commander and Pitch Visual OMT. He holds an M.Sc. in Computer Science and Technology from Linköping University, Sweden.

MIKAEL KARLSSON is the Infrastructure Chief Architect at Pitch overseeing the world's first certified HLA IEEE 1516 RTI as well as the first certified commercial RTI for HLA 1.3. He has more than ten years of experience of developing simulation infrastructures based on HLA as well as earlier standards. He also serves on several HLA standards and working groups. He studied Computer Science at Linköping University, Sweden.

.

Appendix A: Sample HLA Federate Protocol Excerpt

To illustrate what a federate protocol could look like, the following example is provided. It is based on early prototyping.

Three sample packet layouts are provided:

1. A Create Federation Execution service invocation packet, with the name "Test1" and using only one FOM module called "a1.xml".
2. A return packet for a successful invocation
3. A return packet with an exception "Federation Execution Already Exists"

The header contains four fields:

1. The length of the package
2. The message code. Codes 0-99 are used for session setup, teardown and similar purposes. Codes 100 and up are used for HLA service invocations.
3. The session ID is used to identify the session, for example when several federates are serviced by the same server.
4. The call ID is used to match call and return packets.