HLA 4 Federate Protocol –

Requirements and Solutions

Björn Möller, Mikael Karlsson, Fredrik Antelius, Pitch Technologies
Repslagaregatan 25
582 22 Linköping
Sweden
bjorn.moller@pitch.se, mikael.karlsson@pitch.se, fredrik.antelius@pitch.se

Tom van den Berg, TNO
P.O. Box 96864
NL-2509 JG The Hague
Netherlands
tom.vandenberg@tno.nl

Doug Wood, MAK Technologies
150 Cambridge Park Drive
Cambridge, MA
USA
dwood@mak.com

Keywords:
HLA Interoperability, Protocol, Firewall, WAN, Cloud, MSaaS, 5G, Encryption

ABSTRACT: *HLA provides a range of generic services that, together with standardized FOMs, form a stable foundation for simulation interoperability. Historically, HLA federations were deployed in Local Area Networks with simulators written in C++ and Java on a few common operating systems. But network technology evolves, and deployment requirements are also evolving. Today, there are new requirements:*

- *Simulations often need to operate across wide area networks, private or public, across routers and firewalls.*
- *Such communication needs to be encrypted and authenticated.*
- *Some simulations need to operate across 3G/4G/5G networks where glitches and interruptions are common.*
- *Simulations may be implemented in a variety of programming languages and environments that thus need HLA support, for example Python, C#, PHP, Swift, Rust, MATLAB and game engines.*
- *On the server side, simulations may need to be deployed as services with elastic CPU provision. This requires improved support for virtualization and containerization.*

To meet the above real-world requirements, it has been proposed, as part of HLA 4, to provide the HLA services using a language neutral protocol. It provides fault-tolerant, encrypted point-to-point communication, and can be implemented in almost any language and on any platform. This paper provides a closer look at the design of this protocol and how it meets these new requirements.

Some insights into early implementations and test federates are also provided. The HLA 4 Federate Protocol extends the capabilities of HLA and offers many exciting new opportunities.

1. Introduction

The High Level Architecture (HLA) [1] is a standard for distributed simulation, used when building a simulation for a larger purpose by combining (federating) several simulations. The standard was developed in the 90s under the leadership of the US Department of Defense and was later transitioned to become an open international IEEE standard, known as IEEE 1516.

HLA provides a robust set of services for simulation interoperability, including the ability to establish a well-defined set of members, to exchange data (objects and interactions) for any domain, to synchronize systems and data, and to provide management of systems. These services are specified in the HLA Interface Specification part of the standard. HLA can be used for any application domain, for example defense, space, and medical, since HLA Federation Object Models can be developed and evolved using a well-defined object modeling format, specified in the HLA Object Model Template

part of the standard. With a proven set of features, together with an evolving number of FOMs (both standardized, and proprietary) HLA provides a solid foundation for developing distributed simulation solutions.

HLA is maintained by the Simulation Interoperability Standards Organization (SISO) and published through IEEE. A Product Development Group is established to assist when a new version of a standard is developed. Throughout the life of a standard, a Product Support Group provides support for the standard and may also collect input for future versions of a standard.

Three official versions of HLA have been published: US DoD HLA 1.3 (1998), HLA IEEE 1516-2000 (2000) and 1516-2010 (2010) also known as "HLA Evolved". A fourth version of HLA, also known as "HLA 4" is currently under development. As of 2021, two draft versions have been produced. The HLA Federate Protocol, described in this paper, is one of the proposed updates.

1.1 HLA Overview

HLA builds on a standard set of components and concepts that can be seen as a physical view of the interoperating systems. The concepts are shown in Figure 1, also known as a "lollipop" diagram.

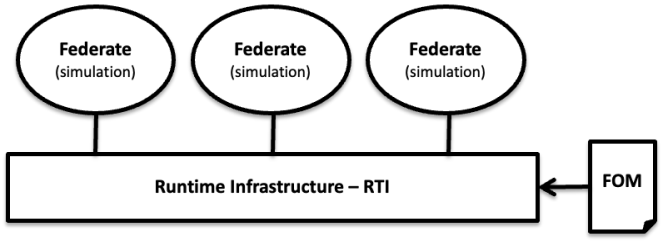

Figure 1: HLA Lollipop Diagram

The main concept in HLA for simulation systems that interoperate is the Federation. The components of an HLA Federation are as follows:

- One or more Federates, which could be systems that simulate entities or processes, tools like data loggers and monitors, and interfaces or bridges to other systems.
- One Runtime Infrastructure (RTI) that provides services for information exchange, coordination and management. These services are grouped into seven service groups: Federation Management, Declaration Management, Object Management, Ownership Management, Time Management and Data Distribution Management, each providing a well-specified set of services.
- Federation Object Model (FOM), specifying, among other things, object classes with attributes, interaction classes with parameters and data types, used for the information exchange.

A key principle is that all information exchange, according to the FOM, is carried out using HLA service calls between the federate and the RTI. No direct data exchange takes place directly between two federates. This facilitates upgrades, replacement, and extension of federates with minimal impact on other federates.

1.2 Physical View and Networking for HLA using C++ and Java

The standardized HLA Services are mapped to a C++ and a Java API in the HLA standard. This means that an RTI provides libraries (DLL, jar-file), called Local RTI Components (LRCs) to deliver these services to a federate. A Central RTI Component (CRC) is also provided by many RTIs. This means that the physical view of a federation could be depicted as shown in Figure 2

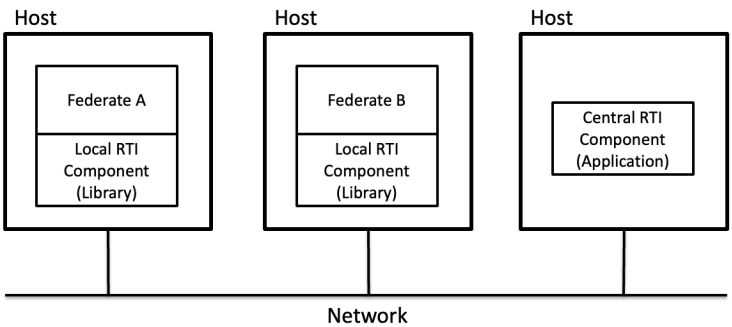

Figure 2: Physical view of a federation with Central and Local RTI Components

Proprietary protocols are used for communication between RTI components. Commonly used approaches are to use TCP/IP unicast for reliable transportation and UDP/IP for best-effort transportation, often using multicast or broadcast for scalability on Local Area Networks. RTIs have also been developed using other communication technologies, like shared memory.

When HLA was created, it was an explicit design decision to specify generic services as an API and to leave room for evolution of underlying technologies such as networking. The advantage of this is that RTI protocols have evolved over time, adding substantial improvements in performance, scalability, and fault tolerance. It has also been possible to optimize RTIs for different environments, such as LAN or WAN. The drawback is that an LRC library from the selected RTI provider must be installed on each federate host.

1.3 The HLA 1516-2010 Web Services (WSDL)

The Web Services API [2] was introduced in HLA Evolved (IEEE 1516-2010). It is based on the Web Services Description Language 1.1 (WSDL) standard [3]. The HLA service names and arguments, as well as responses and exceptions are encoded into XML. The services are exchanged using requests and responses over http, or https in case encrypted communication is required. Early tests showed that around 10 000 to 20 000 updates per second could be exchanged between two prototypical federates on a LAN, which is good enough for many applications, but considerably lower performance than when using native protocols.

There are several advantages with this approach:

- Calls can be made over the protocol from federates implemented in any programming language, running on any operating system.
- Only the information relevant to a particular federate, based on its subscriptions, is delivered over the network.
- There is no need to install a vendor specific LRC on each federate host.
- The protocol works well over routers and Wide Area Networks, for example with a federate deployed behind a firewall.
- If the TCP link is broken, the federate can establish a new connection and continue the http(s) session, since a session cookie that survives the interrupt uniquely identifies the session.

Some disadvantages were also observed:

- The http communication pattern is based on requests initiated by the client, followed by a response from the server. HLA services on the other hand are based on calls and callbacks. The request/response pattern makes it difficult to deliver callbacks. The federate needs to poll the RTI for callbacks regularly. This is impractical and increases latency.
- Implementing callbacks as WSDL responses doesn't match the normal structure of WSDL, making Web Services frameworks and code generators less suitable. This makes it complicated writing federates, although some of this can be handled using middleware.
- XML is bulky, requiring a lot of bandwidth even for a minimal update.
- XML provides great flexibility, but a drawback is that XML processing requires a lot of CPU. In an early test on a LAN it was found that the CPU usage limited the performance more than the increased bandwidth usage that XML added.

To summarize, the HLA Evolved WSDL API is a very promising approach, but it has some drawbacks. Some key aspects worth keeping are the WAN-friendly and fault-tolerant networking and the independence of particular programming languages and vendor-specific LRCs.

2. New requirements and Solution Approaches

Early federations were usually deployed on a well-controlled Local Area Network. Such a network typically supports both point-to-point TCP/IP as well as multicast and broadcast. The same RTI was used for most executions for a given federate. Technology, in particular networking, has since evolved significantly. This has resulted in new requirements and new opportunities, where a standardized federate protocol can add significant advantage. Some areas are:

2.1 Need for a point-to-point TCP protocol

Federates need to work better across Wide Area Networks, through routers and firewalls. Being designed for LANs means that traditionally RTIs communicate directly from host to host using TCP connections as well as UDP and/or Multicast. On a WAN, it is usually not possible for hosts to reach every other host. UDP and Multicast are particularly complicated. While RTIs have used TCP forwarding to bridge both TCP and UDP over the WAN, a direct point-to-point TCP/IP connection, much like what web browsers and commercial games use, would be preferable. The WSDL API showed that this was possible, although with some limitations. When running across open networks, like the Internet, encryption and authentication would also be required for many applications.

2.2 Need for fault tolerance

Federates also need to be able to handle short interruptions in the communications, which are not uncommon in mobile 3G/4G/5G and Wi-Fi networks. This is particularly important when the federate is mobile, like in live simulations. Longer interruptions tend to make a federation execution invalid, since federates may then simulate based on outdated information and possibly lose data that is unrecoverable. It's not feasible to support interrupts past certain limits. However, in the cases of short intermittent interrupts, the direct point-to-point TCP/IP connection between the federate and the RTI makes handling fault tolerance more feasible through message buffering.

2.3 Need to support more environments

Federates are developed in a wide range of environments. There are additional programming languages that need to be supported, such as Python, C#, PHP, Swift and Rust. Calling a C++ library or a Java library may not always be an optimal or even feasible approach (e.g., web-based applications). There are also more stove-piped proprietary environments, like MATLAB and game engines, where a better integration of HLA services could be achieved using a network protocol. Given that the environment can read and write to a socket, it could use the network protocol to join an HLA federation.

2.4 Need to switch RTIs more easily

A wide range of HLA-based simulations tools exist today. Replacement of RTI libraries and verification of the functionality is a recurring challenge for such applications. Using a network protocol removes any direct linkages to vendor specific RTI libraries. Vendor-independent software modules that integrate with any RTI would be an advantage. The use of a centralized "federate protocol server", for instance, would limit the amount of configuration and management of the RTI. This would make it possible to provide an RTI as a service.

2.5 Need to deploy in Cloud environments

Cloud based deployment of federates offers several advantages over a more classic deployment of federates in a local area network. These include, but are not limited to, the ability to replicate a federation many times, the ability to provision additional, virtual, hardware resources (memory, CPU) on demand, and the ability to pool resources between different users. However, in some cases federates should be deployed in a local infrastructure rather than a cloud infrastructure. A federate may be bound to specific, physical, hardware resources, or a federate is under test and requires local access for troubleshooting. In these cases, a network protocol enables the seamless integration of such federates without any vendor-specific elements inside. A "local" federate can mix with "remote" cloud-based federates by connecting to a network endpoint offered by, for instance, a cloud based "federate protocol server".

2.6 Proprietary or standardized solutions?

Some of the above requirements have been addressed in proprietary ways for different RTIs or projects, but experiences from the WSDL API shows that a standardized solution is possible. As part of the HLA 4 product development, a Tiger Team with members from BAE Systems, Johns Hopkins University/APL, MAK Technologies, Pitch Technologies and TNO set out to develop a standardized protocol specification that would keep the advantages of the WSDL API but remove most of the limitations. Several use cases for such a protocol have been presented in an earlier paper [4]. A standardized solution also provides an on-the-wire standard protocol that can be inspected with tools like WireShark. This capability has been a long-requested feature in HLA development.

3. The HLA 4 Federate Protocol in Detail

A number of requirements and design goals were considered when designing the protocol, including:

- Clear and unambiguous mapping to the HLA Services.
- Communication over TCP/IP sockets (point-to-point).
- Ability to use encrypted transport.
- Ability to recover from temporary communication glitches.
- Encoding/decoding support for a wide range of programming languages and operating systems.
- High performance and low CPU and bandwidth utilization.
- Maximize the use of established, commonly understood, and supported technologies and frameworks.

Most of these design goals were met, but some special constructs had to be developed as described below. This section provides an overview of the requirements and design. For a complete specification, see the HLA 4 drafts.

3.1 Three Layer Overview

The Federate Protocol communicates using three layers that build upon each other, as shown in Figure 3.

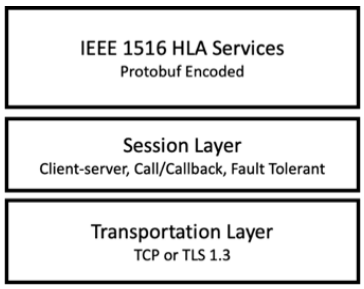

Figure 3: Federate Protocol layers

The layers have the following roles:

IEEE 1516 HLA Services Layer. This layer provides the standard HLA service calls and callbacks, exchanged using the session layer.

Session Layer. This layer provides a well-defined session between the federate and RTI on top of the transportation layer. It supports session setup and teardown, calls initiated from the federate, and callbacks initiated from the RTI. It also provides fault tolerance by providing session resumption in case the transportation layer has been interrupted and then reconnected.

Transportation Layer. This layer provides connectivity between the federate and the RTI, enabling messages to be exchanged in both directions based on TCP/IP. If secure transportation is required, TLS 1.3 [5] is used.

These layers are described in further detail below.

3.2 Services Layer

The services layer provides the HLA services including their supplied arguments, return values, and exceptions, all binary encoded. A number of generally available protocol encoders were evaluated and Protobuf [6], originally developed by Google, was selected due to its efficient encoding, wide range of languages supported for encoding/decoding libraries and code generators, and widespread use. Examples of languages supported by Protobuf are C++, C#, Objective-C, Python, Ruby, Java, JavaScript, Kotlin, Php, Dart, Go, Swift and Rust.

The HLA service calls are mapped into Protobuf messages as follows. For a given HLA Service, two messages are used:

- A Call Request message representing the call and its arguments, for example the Create Federation Execution Request message. This message is sent from the federate to the RTI.
- A Call Response message providing the return values or exceptions thrown, for example the Create Federation Execution Response message or, in case of an exception, the generic Exception Data message. This message is sent from the RTI to the federate.

For a given HLA Service callback, two messages are used:

- A Callback Request message representing the callback and its arguments, for example the Discover Object Instance message. This message is sent from the RTI to the federate.
- A Callback Response message indicating if an exception occurred, using the generic Exception Data message. This message is sent from the federate to the RTI.

Figure 4 illustrates what a request and a response definition look like in Protobuf, in this case for the Register Object Instance With Name service:

```
message RegisterObjectInstanceWithNameRequest {
      ObjectClassHandle theClass = 1;
      string theObjectName = 2;
}

message RegisterObjectInstanceWithNameResponse {
    ObjectInstanceHandle result = 1;
}
```

Figure 4: Sample HLA service request/response definition in Protobuf

The Session layer (described below) provides a sequence number in the message header which enables a federate to match call requests with call responses and an RTI to match callback requests with callback responses.

3.3 Session Layer

A number of requirements and design goals were considered when designing the session layer, including:

- Communicate the encoded HLA Services requests and responses between a federate and the RTI.
- Allow calls and callbacks to be independently initiated both from the federate and the RTI side.
- Match requests and responses.
- Maintain a federate-to-RTI session even if the transportation layer connection is broken.
- Be able to recover from a broken connection.

While it was assumed that the solution would be based on TCP/IP sockets, no established session layer that matches the requirements could be found. A new session layer protocol was thus developed, with a session handling inspired by Web Services. A network message has the following general layout.

Byte	Name	Interpretation
0..3	Packet size	4-byte unsigned integer, big endian
4..7	Sequence no	4-byte unsigned integer, big endian
8..15	Session ID	8-byte opaque data
16..19	Last received message no	4-byte unsigned integer, big endian
20..23	Message type	4-byte unsigned integer, big endian
24..n	Payload	Depends on message type, see below

Figure 5: Network message layout

The fields of a message are:

- Packet size in bytes.
- Sequence number, where the sender (federate and RTI) maintains their own sequence numbering.
- Session ID uniquely identifying the federate's session with the RTI. This is used by the federate when reconnecting to an RTI after an interruption. This approach is inspired by how session cookies are used within web technology.
- Last received message sequence number, indicating successful reception from the receiving side.
- Message type, which is one of HLA Call Request, HLA Call Response, HLA Callback Request, HLA Callback Response, or a Control message.
- Payload, containing the encoded HLA Service requests and responses. For responses a dedicated field is used to indicate the sequence number of the request that resulted in this response.

The use of sequence numbers is shown in Figure 6.

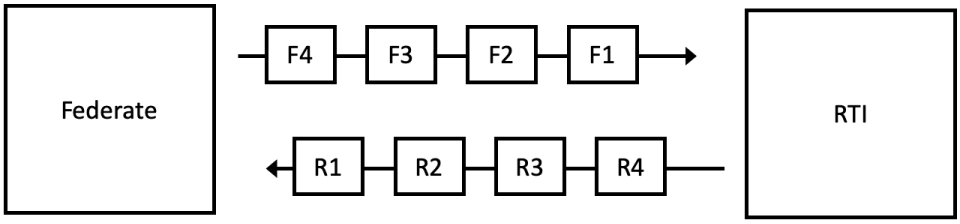

Figure 6: Sequence numbering for federate and RTI messages

The figure shows sequence numbers of the federate messages prefixed with "F" and RTI messages prefixed with "R". Both the federate and the RTI may also maintain a history buffer of already sent messages. This is useful for resending lost messages when resuming a session where the connection was temporarily broken.

The life cycle of a federate is shown in Figure 7:

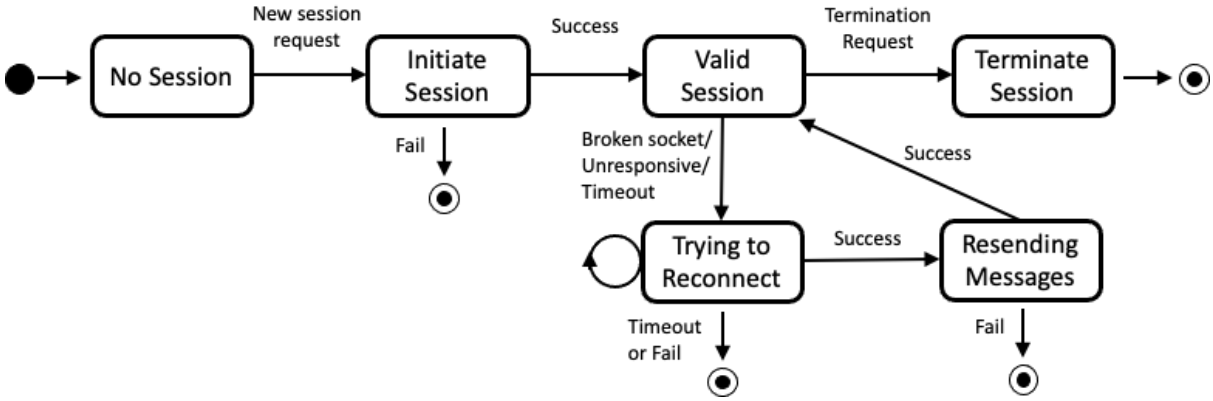

Figure 7: Federate life cycle

The main steps are as follows:

- The federate starts by connecting to the RTI and entering the "Initiate Session" state, establishing a new session with the RTI and getting a Session ID, to be used in later messages.
- The federate then enters the "Valid Session" state during which HLA Calls and Callbacks (with responses) can be exchanged.
- Finally, the federate requests to enter the "Terminate Session" state, resulting in termination of the session. The federate then disconnects from the RTI.
- In case the connection is broken, or a timeout occurs, the federate enters the "Trying to Reconnect" state. If a technical reconnection can be established, the federate and the RTI verifies that there are enough messages in the history buffers to resend lost messages.
- If so, the federate enters the "Resending Messages" state and both the Federate and the RTI resends messages that were lost during the interrupt.
- After the resend, the federate enters the "Valid Session" state and the execution can continue.

A corresponding life cycle for the RTI with respect to each connecting federate is also specified in the standard.

3.4 Transportation Layer

A number of requirements and design goals were considered when designing the transportation layer, including:

- Provide a point-to-point, reliable connection that works well across routers and wide area networks, including a federate communicating from behind a firewall.
- Encrypted communication should optionally be supported.
- Certificate-based verification of the RTI and federate identity should optionally be supported.
- Provide backpressure and flow-control between the sender and receiver.
- Maximize the use of established, commonly understood, and supported technologies.

The resulting choice for unencrypted communications is TCP/IP, which provides bidirectional communication. For the optional encrypted communication, TLS 1.3 was selected. This includes support for server certificates that can be used to authenticate an RTI.

4. Practical Experiences

The current design of the HLA Federate Protocol builds on experiences from the HLA IEEE 1516-2010 WSDL API as well as several other protocols. Two particularly interesting experiments are the Java RMI [7] prototype and the gRPC [8] prototype, both described below. During the development of the standard, engineering support was collected by implementing prototypes in Java and C++, also described below. This also led to discussions about the optimal way to implement middleware for the new protocol.

4.1 Early Experiments: Java RMI

Java Remote Method Invocation (Java RMI) is a Java API that allows for the invocation of remote object methods. In a typical use-case a server application creates objects and registers references to these objects in a RMI registry (located at a well-known address). A client application can obtain the object references from the RMI registry and make method invocations to these remote objects as if these objects are present locally. Method parameters and method return values are transparently and automatically serialized and transferred across the network to and from the remote object. Also, class definitions can be transferred across the network and loaded dynamically by the client, for instance a Java Local RTI Component.

The Java RMI mechanism was exploited in early experiments to learn more about Java RMI, but also for practical reasons. Java RMI is relatively easy to implement due to the native support in Java, requiring relatively little effort to separate the LRC in a client and a server part.

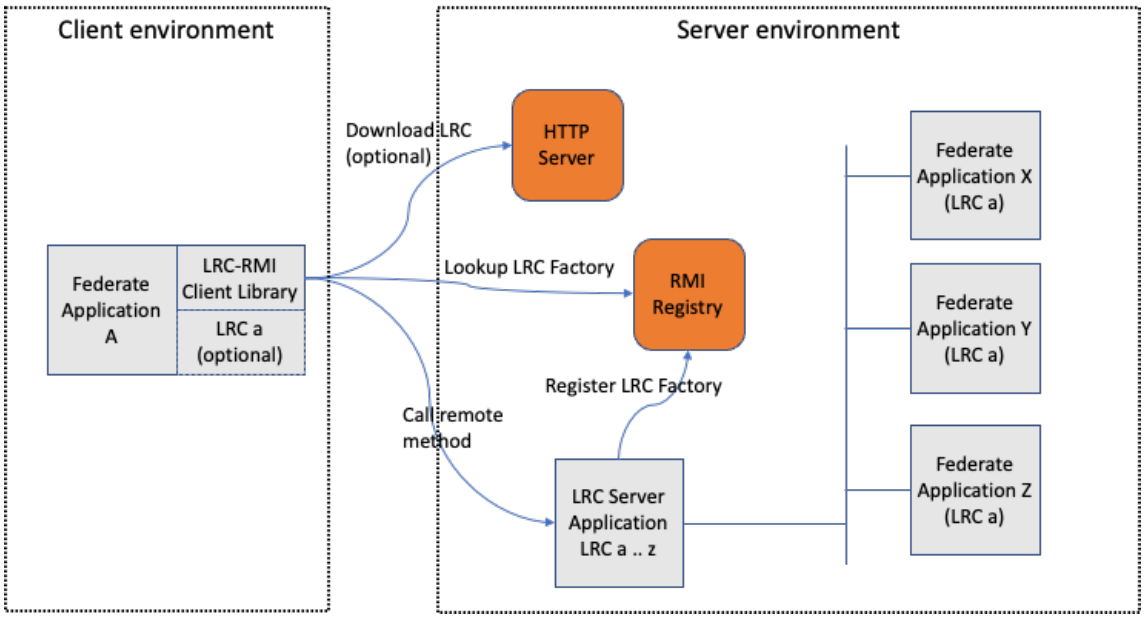

Figure 7: Federate life cycle

The main components in this initial experiment are illustrated in figure 7. The federate application A in the client environment includes the LRC-RMI client library (providing an HLA compliant Java API) on its class path. This seamlessly sets up the communication with the RMI Registry and the LRC Server application. When Federate Application A connects to the RTI and joins the federation, it will actually join the federation in the server environment. The class definitions of the actual LRC ("LRC a", like the Pitch LRC) should be included on the application's classpath or can be downloaded on demand from an HTTP Server. The LRC Server includes several LRC versions, such as Pitch or MAK LRCs (here shown as a..z). A client can select what LRC to use, as long as the used class definitions on the client and server side are identical. This mechanism has been demonstrated successfully with the Pitch RTI, MAK RTI (for the Java binding), and Portico RTI. Several applications were connected, for instance the MAK VR-Forces GUI in the client environment and the VR-Forces SIM in the server environment. Also cloud-based deployment was demonstrated, where "local" federates can connect to a cloud-based LRC Server and subsequent cloud-based federation.

Main lessons learned from this experiment are:

- This set up turned out to be flexible and useful for connecting local federates against a remote federation, in cases where performance is not that important. One or more LRC Servers can be deployed on demand to allow local federates to join cloud based federation executions.
- Some federates request all FOM attribute handles during initialization. On a low bandwidth connection (e.g. home VPN to a remote cloud environment) this can be a problem; functionally the connection works, but it takes more time to initialize the federate. A proposed solution to this is to add HLA services for requesting multiple handles in one call.
- Out of the box Java RMI uses insecure communication; this can be secured if needed.
- Java serialization within Java RMI adds some overhead to the transmission of messages, although no specific measurements were done; alternative protocols should be investigated.
- A few Java classes in the HLA Java API were not "Serializable" (required for Java RMI); a workaround was implemented, and this issue will be resolved in HLA 4.

4.2 Early Experiments: gRPC

Another experiment was based on gRPC, also known as Google Remote Procedure Call. gRPC is an open-source remote procedure call system. It uses http/2 for transport and Protocol Buffers (Protobuf) to describe the available services. A code generator generates cross-platform client and server code in many different languages. This generated code is then used together with gRPC libraries that provide general services.

The first step was to create Protobuf definitions for a subset of the HLA services. The subset was selected to support a typical Chat federate. The gRPC code generator was used to generate code for the HLA services.

Then, server code was created that mapped gRPC services to corresponding HLA service invocations. Callbacks were implemented with a gRPC service that returned a stream of callback objects encoded using protobuf.

On the client side, gRPC provided methods to create both a synchronous and an asynchronous version of the subset RTIambassador API. Two different Chat federates were created, one that used the asynchronous API and one that used the asynchronous API.

Important lessons learned from the gRPC experiment are:

- Protobuf is a good choice to describe services.
- The Protobuf code generator can generate code for building and reading service messages in many different languages.
- The messages generated by Protobuf are compact and efficient.
- The overhead caused by http/2 headers are quite large.
- gRPC did not provide any session handling that was needed to maintain the association between the server-side HLA RTIambassador and the client-side gRPC RTIambassador. Session handling had to be added manually.
- The synchronous variant suffered from latency issues.
- gRPC did not provide any built-in flow control. A federate using the asynchronous API could easily flood the server with service invocations. Some sort of flow control would be needed.

4.3 Java Prototyping of the HLA 4 Federate Protocol

A Java prototype was built in several iterations using Pitch pRTI. In the first iteration, a Protobuf definition file was created by the HLA PDG Federate Protocol Tiger Team. The definition was based on the existing HLA Evolved API, since an RTI supporting a draft HLA 4 API does not currently exist. This Protobuf definition was then used as input to the *protoc* Protobuf compiler that generates code for building and reading Protobuf messages. The generated code was combined with Protobuf libraries to form the Services layer.

In parallel, prototypes of the Session layer and Transportation layer were implemented in Java. As work progressed, tests were made with the C++ implementation developed by MAK Technologies to ensure compatibility.

A server component was implemented that used the session layer to listen for and accept connections from prototype clients. The server accepted service messages from the client and converted them to regular HLA calls. These calls were passed on to Pitch pRTI that acted as a backend for the server. Put differently, the server acted as a federate that used Pitch pRTI.

The first test clients used the generated Protobuf code to build service messages. To simplify the creation of test federates, an adapter was created that presented a standard HLA Evolved Java API. Calls to this Java API were translated, using the Protobuf code, to messages that were sent to the server.

The API adapter and the server implementations were then expanded to cover all HLA services. The adapter and server were then used to successfully run the Pitch pRTI test suite.

During prototyping, the experiences were used to improve the Federate Protocol specification and suggested standard text. In addition, work in the Tiger Team and interaction with the HLA PDG led to updates of the specification which were incorporated in the prototype.

The methods presented by the API adapter were all synchronous which means that each call waits for a return message from the server before returning control to the federate. As discovered during the experimentation phase, this can make the calls slow. To investigate possible alternatives, asynchronous variants were created for a few methods.

4.4 C++ Prototyping of the HLA 4 Federate Protocol

Independently from the Pitch Java prototype implementation of the federate protocol, a C++ prototype implementation was developed using the MAK RTI. The C++ prototype supports the session layer but only implemented enough of the service layer to demonstrate that different implementations of the federate protocol could interoperate. The implemented services were based on the HLA Evolved API. The supported services were connect, list/report federation executions, create/destroy federation execution, join/resign federation execution, publish/subscribe interaction and send/receive interactions. A chat client was implemented in C++ as well as a federate protocol server.

Session layer message definitions along with the Protobuf C++ service definitions were shared between the implementation of the federate protocol server and the federate protocol client.

The federate protocol server processes connections from federate protocol clients using the transportation layer (TCP sockets) and the session layer messages. For new connections, a federate proxy is created which takes over the processing of the connection. The federate proxy creates traditional C++ RTI and Federate Ambassadors to represent the client federate in the federation. The federate proxy processes the service call request messages from the client to invoke the traditional HLA C++ API services on behalf of the client, returning any result or exception to the client in a corresponding call response message. Any Federate Ambassador callbacks are put into callback request messages and sent to the client. The client will return a corresponding callback response message. A federate proxy manager handles the reconnect process matching up a reconnecting connection with its existing federate proxy.

The federate protocol client uses wrappers around the federate protocol mimicking the traditional RTI and Federate Ambassadors. The RTI Ambassador wrapper transforms the traditional service calls and parameters into the corresponding Protobuf definitions and sends them to the server using the federate protocol call request message. For initial prototyping, a blocking approach was used so that the service call does not return until the corresponding call response is received. Similarly, the Federate Ambassador wrapper transformed the callback request messages into Federate Ambassador callback invocations.

With both a C++ and a Java implementation available, the following combinations were tested for interoperability:

- C++ clients using Java server
- Java clients using C++ server
- C++ and Java clients using Java server
- C++ and Java clients using C++ server

4.5 Middleware/adapter approaches

The HLA 4 Federate Protocol specifies a number of messages and how they are exchanged. It is expected that most developers will use a middleware stack for exchanging such messages. There are several approaches for this.

- Develop your own solution. This includes taking the proto files, generating code using protoc, writing your own protocol stack, building service messages using Protobuf classes and sending the messages.
- Use a protocol stack (transportation layer and session layer) provided by other developers (in-house, commercial off-the-shelf, government off-the-shelf, open source, etc). Build service messages using Protobuf classes and send using methods in the protocol stack.
- Use a protocol stack and an adapter that presents a traditional Java or C++ HLA API. This adapter builds service messages and passes them to the protocol stack. The main advantage of this is that it makes it easy to switch between traditional LRCs and the protocol stack.

Option three has a limitation. The standard API uses regular synchronous method calls where the federate invokes a method and then waits for the method to finish and return a result. An adapter that presents the standard API will suffer from the fact that each method call corresponds to a network message from the client to the server and the result corresponds to a message from the server to the client. Therefore, each call gets an overhead of two times the latency of the network.

To get optimal performance for the protocol a middleware should support making several HLA service calls, for example a large number of attribute updates or interactions, without waiting for return values or exceptions. Such a middleware could provide an HLA API extended with asynchronous delivery of return values and exceptions. Middleware may alternatively make commonly used HLA services without return values, like *update attribute values* and *send interaction*, asynchronous by ignoring and not waiting for any exceptions. If an asynchronous middleware is used, the number of pending call requests in-flight needs to be limited to provide flow control and avoid overloading the server with calls faster than they can be processed.

5. Conclusions

The standardized HLA 4 Federate Protocol extends the capabilities of HLA and offers many exciting new opportunities. Some of the key advantages are the ability for federates to connect securely to an RTI in the cloud, to work over unreliable links, such as 3G/4G/5G, to support additional languages and environments, and to remove the requirement for RTI-specific libraries in federates that are reused in different environments. It can be expected to co-exist with the classic C++ and Java APIs with LRCs since the two approaches are optimal for different use cases.

References

[1] IEEE 1516-2010. High Level Architecture (HLA), [online], August 2010, available at www.ieee.org.

[2] Björn Möller, Staffan Löf: "A Management Overview of the HLA Evolved Web Service API", Proceedings of 2006 Fall Simulation Interoperability Workshop, 06F-SIW-024, Simulation Interoperability Standards

Organization, September 2006.
[3] W3C: "Web Services Description Language (WSDL) 1.1", W3C, URL: www.w3.org/TR/wsdl
[4] Björn Möller, Mikael Karlsson, Fredrik Antelius: “Towards a Standardized Federate Protocol for HLA 4”, Proceedings of 2018 Winter Simulation Interoperability Workshop, 18W-SIW-037, Simulation Interoperability Standards Organization, January 2018
[5] IETF RFC 8846, "Transport Layer Security (TLS) Protocol," version 1.3, ISSN, August 2018
[6] “Protocol Buffers - Google's data interchange format”, https://github.com/protocolbuffers/protobuf, retrieved 1-Feb-2022
[7] “Remote Method Invocation Home”, https://www.oracle.com/java/technologies/javase/remote-method-invocation-home.html, retrieved 1-Feb-2022
[8] gRPC home page, https://grpc.io, retrieved 1-Feb-2022

Author Biographies

BJÖRN MÖLLER is the President and co-founder of Pitch Technologies. 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 SISO RPR FOM Product Development group, chairman of the SISO Space FOM Product Development Group. and the vice chairman of the SISO HLA Evolved Product Development Group.

FREDRIK ANTELIUS is the Head of Product Development 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 twenty 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.

TOM VAN DEN BERG is a senior scientist in the Modeling and Simulation department at TNO, The Netherlands. He holds an M.Sc. degree in Mathematics and Computing Science from Delft Technical University and has over 25 years of experience in distributed operating systems, database systems, and simulation systems. His research area includes simulation systems engineering, distributed simulation architectures, systems of systems, and concept development & experimentation. Tom is a member of several SISO Product Development / Support Groups, participates in a number of NATO M&S standardization activities.

DOUG WOOD is a principal software engineer at MAK Technologies, working on the Link family of products involved in facilitating distributed simulation interoperability. He holds a M.S. in Computer Science from the University of Central Florida. He has over thirty years of experience developing and managing software for distributed simulation and training including distributed simulation architectures and protocols, automated behavior, electronic warfare simulation, emergency management simulation, and training device databases. He is currently serving as editor of the HLA Interface Specification in the HLA Product Development Group.