Ross, P. (2014). *Comparison of High Level Architecture Run-Time Infrastructure Wire Protocols—Part Two.* Fall Simulation Interoperability Workshop, 8–12 September 2014, Florida Mall Conference Center, Orlando, United States. Paper 14F-SIW-002.

Comparison of High Level Architecture Run-time Infrastructure Wire Protocols – Part Two

Peter Ross
Defence Science & Technology Organisation
506 Lorimer St, Fishermans Bend VIC 3207, Australia
peter.ross@dsto.defence.gov.au

Keywords:
HLA, interoperability, RTI, simulation, wire protocol

ABSTRACT: *Under High Level Architecture (HLA), distributed simulation services are provided by middleware known as the Run-Time Infrastructure (RTI). Existing RTI implementations, despite being similar in function and performance, do not interoperate with one another as they each use a different proprietary wire protocol. Appreciating how these protocols converge and diverge is a necessary first step towards developing an interoperability standard. We describe an effort to compare the wire protocols of ten state-of-the-art commercial, government and open-source RTI implementations. This is a continuation of work, presented at SimTecT 2012, that found RTI implementations use similar message encoding and communication systems. Part Two digs deeper, comparing the data structures, and issuance and receipt rules for popular HLA services.*

1. Introduction

The wire protocol is an essential part of High Level Architecture (HLA) as it establishes how information is exchanged 'on the wire' between federates. Unlike earlier distributed simulation technologies, HLA standardises the Application Programming Interface (API) between the federate and the middleware, known as Run-Time Infrastructure (RTI). The standard does not define the embodiment of the wire protocol, or make assurances about its performance or behaviour. These decisions are left up to the RTI implementation. There are many RTI implementations available today, but because each uses a different proprietary wire protocol, they do not interoperate with one another.

The lack of a wire standard means that all federates must use the same RTI implementation to guarantee that the federation will execute. To avoid ambiguity federation agreements specify the RTI implementation by name and software version [1]. For large exercises or experiments this often results in some participants having to change their implementation to satisfy the agreement. The changeover process is not without cost or technical risk, though this claim has never been formally studied. Supporting evidence can be found in the contract notices of established government projects, where the reasons for *not changing* RTIs are sometimes given. The proliferation of interoperability bridges and gateways is another relevant data point [2]. These tools cater for situations where changing the RTI is not feasible.

Since its inception, debate has stirred over the absence of a standard HLA wire protocol. Supporters of the status quo assert that standardisation would restrict developer freedom and performance optimisation, whilst opponents assert that interoperability would reduce integration cost and permit better network communication diagnostics [3] [4] [5]. Informed discussion on this issue is made difficult by the proprietary nature of the technology. Little is actually known about HLA wire protocols because developers do not openly publish their specifications.

This paper describes an effort to study the wire protocols of existing RTI implementations (referred to as *implementations* hereon for brevity). We are motivated to understand the reasons why current implementations do not interoperate, and what practical steps can be taken to achieve better interoperability. Due to the breadth of the study, it has been presented in two parts. Part One compared the communication systems and message formats of nine implementations [6]. Part Two compares the data structures, and issuance and receipt rules for popular HLA services. A reading of the first paper is not required to appreciate the findings presented here.

2. Background

The Simulation Interoperability Standards Organization (SISO) has sponsored two standards activities concerning wire protocol interoperability. The *RTI Interoperability Study Group* was formed after the release of HLA V1.3 to explore issues associated with HLA interoperability. It acknowledged that a standard wire protocol was the best long-term solution, but concluded that premature standardisation may inhibit further development of RTI implementations [7]. The *Open Run-Time Infrastructure Protocol Study Group* was formed several years later after one vendor published a draft wire protocol called HLA Direct. The group investigated the feasibility of developing an interoperability standard, but did not advance further due to lack of volunteer interest [8]. Both study groups have since disbanded.

A small quantity of literature is available on the subject of RTI wire protocols. Some early implementations published brief descriptions of their protocols [9] [10] [11] [12]. Since all of these implementations have changed in the decade (or more) following initial release, the information is insightful, but not especially relevant. Several authors have compared the distributed computing algorithms for Time Management and Data Distribution Management (DDM) [13] [14] [15]. Direct comparisons between implementations have also been made [16] [17]. While these comparisons primarily consider federation performance and scalability, they also discuss wire protocol design limitations.

3. Method

The provision of RTI diagnostic tools by some commercial vendors, and emergence of mature open-source implementations, makes meaningful comparison of HLA wire protocols now possible. A dataset was built characterising the communication systems and message formats of ten implementations. Only the services defined in the HLA V1.3 interface specification, or the equivalent IEEE 1516 services, were analysed. This enabled a wide variety of implementations to be considered.

For each implementation, the technical documentation and source code were first reviewed to understand its concept of operation and capabilities. Some implementations provided diagnostic tools which display the contents of wire protocol messages in human readable form. These tools aided in the identification of message types and their structure. RTI communication was also captured and studied using the Wireshark network protocol analyser[1]. Each implementation was evaluated using a suite of test federates. These federates invoked HLA services in difference sequences and varied the input parameters. By running the test federates and monitoring the diagnostic tools, we were able to appreciate the issuance and receipt rules for each wire protocol.

[1] Wireshark website – http://www.wireshark.org/

Table 1: RTI implementations. Modus operandi (MO) may be centralised (C), decentralised (D) or hierarchical (H). Implementations marked with an asterisk were only partially analysed due to lack of technical documentation and diagnostic tools.

Implementation	MO	Version	Release date
BH RTI	H	2.2	2006
CERTI	C	3.4.2	2013
HLA Direct	D	0.1	2003
MAK RTI	C | D	4.1	2012
OHLA	C	0.6	2013
Open RTI	C	0.5	2014
Portico	D	2.0.0	2013
pRTI 1516 (*)	C	3.2.2	2007
RTI NG Pro (*)	C | D	4.0.4	2006
RTI-s	D	D27D	2012

Table 1 lists the implementations and software versions analysed. These were chosen on the basis of accessibility to the author, and present a balanced mix of commercial, government and open-source offerings. Further details on each implementation can be found in the open literature. RTI NG, the implementation made freely available by the United States Defense Modeling & Simulation Office (now known as the Modeling & Simulation Coordination Office) was not included as it shares lineage with RTI NG Pro. Not all implementations provided technical documentation and the tools necessary for the wire protocol to be fully appreciated. Analysis of these implementations was therefore limited to the *support*, *update attribute values* and *send interaction* services (Sections 4 and 6.3).

3.1. Modelling the RTI

By design, the HLA standard gives no insight into the internal workings of the RTI. To discuss and reason effectively about wire protocols, an abstract model of how the RTI operates internally is needed. The model was introduced in Part One, and the main elements are summarised bellow. It is valid for all implementations analysed.

Components. Run-time Infrastructure is made up of components called Local RTI Components (LRCs) and Central RTI Components (CRCs). While these terms are not defined in the HLA standard, they appear in RTI documentation and design literature. The LRC is a software library that each federate links to. It provides the application programming interface to the federate developer, and is named 'librti1516' or similar. Coordination of the LRCs, if necessary, is performed by the CRC. This usually takes the form of a standalone program, and is named 'rtiexec' or similar.

Communication system. Components exchange messages via communication channels. These may be formed across different media (such as Internet Protocol or shared memory) and employ different interconnection methods (such as multicast, unicast, or relay).

Messages. Components exchange information using messages. A message consists of a fixed-length header identifying at least the message type and total length, followed by a message-specific body. The messaging system may incorporate additional capabilities, such as versioning, fragmentation, bundling and compression.

Modus operandi (MO). The manner in which an RTI organises its components is called its mode of operation. The mode was found to influence the overall design of the wire protocol. Three modes were identified. A *centralised* operating mode is where LRCs are coordinated by a single CRC. A *decentralised* mode is where there is no central coordination. Finally, a *hierarchical* mode is a hybrid of these where multiple CRCs coordinate LRCs. As only one hierarchical implementation was included in the study, we consider it in both centralised and decentralised discussions. Some implementations provided a configuration option to switch between centralised and decentralised operating modes. Both options are also considered in our discussions.

3.2. Findings for centralised implementations

Full findings of the study are presented in Sections 4–7. From the perspective of the wire protocol, all centralised implementations were found to behave in a similar manner. Specifically, when the federate invoked a service, the LRC would serialise the service parameters into a request message and send this to the CRC. On receipt of the message, the CRC would action the request and return a response message. The CRC would also send 'callback' messages to the LRC asynchronously, for services such as *reflect attribute values*. To avoid unnecessary repetition throughout the paper, only exceptions to this behaviour are discussed.

The wire protocols of centralised implementations can be likened to the Web Services (WS) API introduced in HLA Evolved. This is an alternative to the C++ and Java programming language interfaces. Instead of calling a function to invoke a service, the federate sends an XML message to the RTI's web server[2]. The web server actions the request and replies with a response message. The HTTP client software embedded within a web-enabled federate loosely resembles a LRC, and the web server can be thought of as a CRC. Significant differences between the WS API and wire protocols of centralised implementations are summarised below.

1. XML message encoding is text based and extremely verbose, whereas all other wire protocols were byte-oriented and minimalistic.
2. All centralised implementations were found to use handles, whereas the WS API requires all concepts to be referenced by name. Handles are introduced in Section 4.
3. Federates must poll the web server to obtain 'callback' notifications, whereas the components of centralised implementations operated asynchronously.

4. Support services

Handles provide an efficient mechanism for computers to address concepts by number rather than by name. The *support services* of each implementation were analysed to determine if and how handles were represented on the wire.

When implementing handles, RTI vendors encounter two design questions. *How can handles be generated* to ensure they are unique, and *how large should they be*? In answer to the first question, all centralised implementations used counters that were managed by the CRC.

4.1. Execution handle

When multiple federation executions share the same communication channel, it is necessary for messages to identify which execution they belong to. Where they were used, execution handle sizes varied from 16 to 40 bits.

Two implementations avoided this problem altogether by forcing executions to occur on different communication channels. Another published the execution name verbatim in each message, avoiding the need for handles. Decentralised implementations were found to generate handles by either hashing the execution name, or by combining the first and last letters of the execution name.

4.2. Federate handle

All implementations were found to use federate handles on the wire. Seven implementations used a 32-bit sized

[2]HLA Evolved calls this the Web Services Provider Component (WS PRC).

handle, two were 16-bit and one was 12-bit.

Only one decentralised implementation used a distributed algorithm to ensure the uniqueness of the federate handle. All other decentralised implementations generated handles independently using random numbers, hashing the federate's process identifier, or combination of the two. These approaches do not guarantee uniqueness, and it is possible for handle collisions to occur. To counter this, some implementations provided an option to manually specify the federate handle.

4.3. Object instance handles

Two implementations did not use object instance handles on the wire, instead opting to reference objects by name. All others used a 32-bit handle.

The remaining decentralised implementations all used a per-federate object instance counter. The counter was combined with the federate handle to ensure the object instance handle was unique across the federation execution. See [5] for a worked example. This approach places a limit on the total number of object instances possible per federate. The smallest *default* limit observed was 65535. In recognition of this problem, one implementation provided an option to tune the total number of object instances per federate. The total number of objects permitted per federate could be increased by sacrificing the total number of supported federates (or vice versa).

When a federate registers an object instance with a centralised implementation, the LRC has to request an object instance handle from the CRC. This operations takes time and must be repeated for each new object. One implementation maintained a pool of preallocated handles at each LRC. Only when the pool dropped below a threshold, did the LRC request more handles from the CRC. This approach sought to increase performance of the *register object instance* service.

4.4. Object model handles

Handles were used by all implementations to represent named elements of the Federation Object Model (FOM). In this section we only examine handles for object and interaction classes, and their attributes and parameters. Space and dimension handles were not considered, neither was the impact of modular FOMs. Seven implementations represented object model handles with 32-bit integers. Only three used 16-bit integers.

Object model handles were generated by the following three distinct methods. See Table 2 for an example of each.

Method 1: Six implementations used an object class counter and a separate attribute counter for each base object class. Interaction classes and parameters were mapped in the same way.

Method 2: Three implementations used an object class counter, and a separate attribute counter shared by all object classes. Interaction classes and parameters were again mapped in the same way.

Method 3: Finally, one implementation used a single counter shared amongst all elements of the FOM.

Table 2: Example of class and attribute handles generated by the three different methods.

FOM element	**Class and attribute handle**		
	Method 1	**Method 2**	**Method 3**
Class A	1	1	1
Attribute a	1	1	2
Attribute b	2	2	3
Attribute c	3	3	4
Class X	2	2	5
Attribute x	1	4	6
Attribute y	2	5	7
Class Z	3	3	8
Attribute z	1	6	9

5. Federation management services

Federation membership is managed by the *create* and *join federation execution* services. The standard requires the RTI to return an error when a federate tries to claim an execution or federate name that is already in use. All centralised implementations relied on the CRC to keep track of federation executions and their memberships. Table 3 lists the different methods used by decentralised implementations. These are described below.

1. *Heartbeat Method.* Four implementations used a heartbeat mechanism to convey the existence of executions and federates to other LRCs. The execution heartbeat was either sent by all federates joined to the execution, or just by the creator of the execution. A flaw exists in the later design. When the federate that created the execution resigns or is interrupted, the heartbeat is no longer sent.

2. *Query Method.* Three implementations used a query mechanism to ask other LRCs if the proposed execution or federate name was in use.

3. *No Method.* Two implementations chose to ignore the requirement for unique execution names.

Table 3: Methods used by decentralised implementations for the *create* and *join federation execution* services. Heartbeat interval is indicated in seconds.

Implementation	Execution deconfliction method	Federate deconfliction method
BH RTI	Query *and* Heartbeat(15)	Heartbeat(18)
HLA Direct	None	Heartbeat(60)
MAK RTI	None	Query
Portico	Heartbeat(3)	Query
RTI-s	Heartbeat(5)	

5.1. File reading and distribution

When a federate invokes the *create federation execution* service it must also tell the RTI the location of the FOM Document Data (FDD) file. The location is expressed as a Universal Resource Locator (URL) or ordinary filename. The HLA standard does not define where and how the file is accessed. All but one implementation relied on the LRC to read the FDD file. The CRC of the other implementation was responsible for reading the FDD file.

The FDD location is not a parameter to the *join federation execution* service[3]. The LRCs of joing federates therefore had to find a way to obtain the file. All centralised implementations relied on the CRC to supply FDD content to joining LRCs. This was supplied either in original text form or a byte-oriented data structure. The methods used by decentralised implementations to read the file are summarised below. Some implementations would attempt more than one method.

Method 1: Request the FDD *content* from another LRC.

Method 2: Request the FDD *location* from another LRC, and attempt to open it locally.

Method 3: If the joining federate also tried to create the federation (and failed because it already existed), try to read from the FDD location passed to the earlier *create federation execution* service.

Method 4: Append a '.fed' or '.fdd' file extension to the execution name, and try to open the filename.

Two decentralised implementations also sought to ensure consistency of the FDD file across the federation. This was achieved by publishing and comparing file checksums.

6. Object management services

Three design problems concerning object management were studied: how to prevent object name conflicts; how to inform new subcribers of earlier created object instances; and how to communicate attribute and parameter value updates efficiently.

6.1. Preventing object name conflicts

All centralised implementations relied on the CRC to track object use and prevent name conflicts. Only one decentralised implementation addressed this requirement. During object registration Portico would send a query message to all fellow LRCs asking if the name was in use. If no protest message was received after one second, object registration would proceed. This facility was disabled by default as it slows down the object registration process.

When no object name is supplied to the *register object instance* service, the RTI must generate a unique object name. All implementations derived a name from a numeric representation of the object instance handle, e.g. 'HLA<ObjectInstanceHandle>'. As one implementation did not use object instances handles on the wire, the name it generated included as much entropy as possible to prevent collisions. This included the host IP address, process identifier, and a local counter.

6.2. Late discovery problem

When a federate subscribes to an object class, the HLA standard requires it to discover all existing instances of that object class. We call this the late discovery problem. Table 4 lists the methods used by decentralised implementations to solve this problem. They are described in detail below. Note this aspect of centralised implementations was not studied.

1. *Reactive Method.* Three implementations used a reactive approach. When a new federate joined an execution, or subscribed to a new object class, the LRC would send a 'join' or 'subscribe' message to all other LRCs. On receipt, LRCs would respond with a list of relevant object instances that they owned. This was achieved via a single message describing all the object instances, or separate messages for each instance.

[3] The *join federation execution* service was revised in HLA Evolved to accept an optional FDD location parameter.

2. *Heartbeat Method.* One implementation used heartbeat messages. The LRC of a federate that owned an object was responsible for sending its heartbeat.

3. *Lazy Method.* One implementation pushed the late discovery problem onto the federate. New subscribers to an object class only became aware of object instances when they were next updated.

All these approaches involve some form of compromise. Heartbeats waste bandwidth and require federates to wait until the next heartbeat is published. Reactive methods can create large message bursts when federates identify themselves. We did not check to see if any implementations used message throttling to guard against this. Lazy discovery assumes object values will be updated routinely, which is not the case for all simulations.

Table 4: Methods used by decentralised implementations to prevent object name conflicts and to discover object instances. Heartbeat interval is indicated in seconds.

Implementation	**Object name deconfliction method**	**Late discovery method**
BH RTI	None	Heartbeat(18)
HLA Direct	None	Reactive–Join
MAK RTI	None	Reactive–Sub
Portico	Query	Reactive–Join
RTI-s	None	Lazy

6.3. Communicating attribute and parameter values

The *update attribute values* service is the workhorse of the RTI, as this service communicates changes in attribute values to other interested federates. All implementations were found to send similarly formatted messages to other components describing the updated values. The messages indicated the object instance handle being updated, and contained a list of attribute handles, value sizes and value content. This corresponds closely to the parameters of the *update attribute values* service. Some implementations included additional information in the message, such as the communication channel, transport type and user supplied tag.

There were subtle differences in the way the information was arranged within the messages. Four implementations used a single list of records, where each record described the attribute handle, value size, and value content. An example of this, taken from Open RTI, is shown in Table 5. Other implementations split the handles, value sizes and value content into separate lists. While these appear to be arbitrary design choices, the layout will affect message compressibility and cache utilisation.

The maximum permitted attribute size varied slightly between implementations. Eight implementations used a 32-bit field to store the attribute value size. The maximum size for MAK RTI was 16 or 32-bits depending on mode of operation. HLA Direct, showing its vintage, was limited to a 16-bit field.

Table 5: Example update attribute values message

Record	**Field**	**Data type**
Header	Magic number	$8\times$ uint8
	Message size	uint32
	Message type	enum
Body	Federation handle	uint16
	Object instance handle	uint32
	User tag size (T)	uint32
	User tag value	$T\times$ uint8
	Transport type	enum
	Number of attributes (N)	uint32
Attribute #1	Attribute handle	uint32
	Attribute size (S_1)	uint32
	Attribute value	$S_1\times$ uint8
$\vdots$		
Attribute #N	Attribute handle	uint32
	Attribute size (S_N)	uint32
	Attribute value	$S_N\times$ uint8

Invocation of the service did not always result in a single message being sent. Where separate communication channels were allocated for reliable and unreliable data, the update was split into two messages and sent across the appropriate channels. Our analysis does not consider DDM, time management or bundling, which may result in more or less messages being sent.

The *send interaction* service was achieved using a similarly structured message across all implementations. The only significant difference was the inclusion of the interaction class handle, instead of an object class instance handle.

7. Time stamp representation

HLA defines two types of message ordering, Receive Ordering (RO) and Time Stamp Ordering (TSO). When a service is invoked with TSO, the time stamp parameter must be communicated to other federates, along with the normal information associated with the service. This was achieved four different ways.

Three implementations reported the time stamp as a 64-bit floating point number within the message header. Fixing the time representation like this prevents use of alternative federation time libraries. It also wastes bandwidth for services that do not use time stamps. All other implementations supported an abstract representation of the time stamp value.

Two implementations defined separate message types for the RO and TSO variants of services. The time stamp value was only included in the TSO variant. Another implementation adopted an opposite approach to this. It used a single message for services that were either RO or TSO, and the time stamp value was always included in the message body. A zero-length time value signalled an RO service, otherwise TSO was assumed.

Finally, one implementation defined a special *time stamp* message that encapsulated other messages. For example, when the *send interaction* service was invoked with TSO, the implementation would construct a generic send interaction message, and then wrap it inside the time stamp message (together with the time stamp value). The special *time stamp* message was able to contain multiple messages. This avoided repetition of the time stamp value for services invoked at the same logical time.

8. Conclusion

This paper concludes an effort to compare the wire protocols of ten RTI implementations. Overall we found all the protocols to be different and incompatible. This was to be expected! Only by examining how individual services were conveyed 'on the wire', and characterising the methods used, could the differences and similarity of wire protocols be appreciated. The significant findings are summarised below.

1. Mode of operation was found to greatly influence the design of the wire protocol. The protocols of centralised implementations were all similar across the services analysed, and can be likened to the WS API protocol. Decentralised implementations were much more varied, employing many different methods.

2. For each service, there was always at least one method shared by multiple implementations. Often this represented the most obvious design choice. Many novel methods were also observed. The existance of these demonstrates that the HLA standard is working as intended. Vendors are innovating!

3. All decentralised implementations took shortcuts in addressing HLA requirements. Shortcuts involved hash functions, random number generators or simply ignoring the requirements.

Implementation diversity is a core principal of HLA, but it also creates uncertainty for the federate developer. In the absence of specific requirements, vendors have had to make decisions about maximum attribute size, handle limits, time stamp representation, and even where to read the FDD file. Vendors of decentralised implementations have also had to balance standards compliance against achieving acceptable RTI performance, sometimes choosing the later. We found these decisions to vary across *all* implementations. It is likely that some of these decisions influence federation development, and contribute to the difficulty of changing RTI implementations at a later date.

Three limitations of the study must be noted. We have discussed only the basic services of the HLA interface specification. Federation save and restore, synchronisation, Dynamic Data Distribution (DDM) and Ownership Management services were not evaluated, nor any of the new capabilities introduced in HLA Evolved. Secondly, our analysis of the strengths and weaknesses of each method is quite shallow. A more detailed comparison, with supporting benchmarks, was beyond the scope of the study. Finally, RTI implementations and their wire protocols are frequently updated. The results presented are only valid for the software versions listed.

As we approach the 20th anniversary of High Level Architecture, is time to revisit the principals of the original design? While our findings are insufficient to define a complete wire protocol standard, the key ingredients have been identified. More work on this topic is needed. The study also highlights the technical challenges faced by decentralised RTI vendors. This kind of RTI makes up half of the implementations evaluated, yet none could be regarded as HLA compliant. We hope that the real-world practices observed in this study will contribute to the next evolution of the HLA standard.

9. References

[1] IEEE: "Distributed Simulation Engineering and Execution Process Multi-Architecture Overlay (DMAO)" IEEE Std 1730.1, 2013.

[2] A.E. Henninger, D. Cutts, M. Loper, R. Lutz, R. Richbourg, R. Saunders & S. Swenson: "Live Virtual Constructive Architecture Roadmap (LVCAR) Final Report" Institute for Defense Analyses, 2008.

[3] L. Granowetter: "RTI Interoperability Issues – API Standards, Wire Standards and RTI Bridges" 2003 Spring Interoperability Workshop, 03S-SIW-063.

[4] T.W. Pearce & N.B. Farid: "If RTI's Have a Standard API, Why Don't They Interoperate?" 2004 Fall Simulation Interoperability Workshop, 04F-SIW-100.

[5] K. Mullally, G. Hall, D. Gordon, B. Pemberton & C. Peabody: "Open Message-Based RTI Implementation – A Better, Faster, Cheaper Alternative to Proprietary, API-Based RTIs?" 2003 Spring Simulation Interoperability Workshop, 03S-SIW-112.

[6] P. Ross: "Comparison of High Level Architecture Run-Time Infrastructure Wire Protocols – Part One" SimTecT 2012 Conference Proceedings. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.278.5840`

[7] M.D. Myjak, D. Clark & T. Lake: "RTI Interoperability Study Group Final Report" 1999 Fall Simulation Interoperability Workshop, 99F-SIW-001.

[8] J. Woodyard & K. Mullally: "Open Run-Time Infrastructure Protocol Study Group Final Report" 2004 Fall Simulation Interoperability Workshop, 04F-SIW-018.

[9] J.O Calvin, C.J. Chiang, S.M. McGarry, S.J. Rak, D.J. Van Hook & M. Salisbury: "Design, Implementation, and Performance of the STOW RTI Prototype" 1997 Spring Simulation Interoperability Workshop, 97S-SIW-019.

[10] M. Karlsson, S. Löf & B. Löfstrand: "Experiences from Implementing an RTI in Java" 1998 Spring Simulation Interoperability Workshop, 98S-SIW-062.

[11] D.D. Wood & L. Granowetter: "Rationale and Design of the MAK Real-Time RTI" 2001 Spring Simulation Interoperability Workshop, 01S-SIW-104.

[12] Z. Zhou & Q. Zhao: "Reducing Time Cost of Distributed Run-Time Infrastructure" Proceedings of the 16th International Conference on Artificial Reality and Telexistence, pp969–979, 2006.

[13] B. Watrous, L. Granowetter & D. Wood: "HLA Federation Performance: What Really Matters" 2006 Fall Simulation Interoperability Workshop, 06F-SIW-107.

[14] P. Gupta & R.K. Guha: "A Comparative Study of Data Distribution Management Algorithms" Journal of Defense Modeling and Simulation on Applications, Methodology, Technology, Vol. 4, Issue 2, pp127–146, 2007.

[15] J-B. Chaudron, E. Noulard & P. Siron: "Design and modelling techniques for real-time RTI time management" 2011 Spring Simulation Interoperability Workshop, 11S-SIW-045.

[16] L. Malinga & W.H. le Roux: "HLA RTI Performance Evaluation" 2009 SISO European Simulation Interoperability Workshop, 09E-SIW-005.

[17] D.R. Azevedo, A.M Ambrosio & M. Vieira: "HLA Middleware Robustness and Scalability Evaluation in the Context of Satellite Simulators" 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC), pp312 - 317.

Author Biography

PETER ROSS is a first year PhD student at the School of Computer Science & Information Technology, RMIT University, Australia. He has taken leave from an engineering position at the Defence Science & Technology Organisation (DSTO) to pursue research interests in agent-based modelling and simulation.