

Employing Hierarchical Federation Communities in the Virtual Ship Architecture

Anthony Cramp and Michael Oudshoorn

Department of Computer Science
University of Adelaide
Adelaide, South Australia, 5000.

{crampy,michael}@cs.adelaide.edu.au

Abstract

This paper discusses work underway to develop a framework for the use of hierarchical federation communities as a tool for distributed simulation. The Virtual Ship Project is the application driving the development of the framework. The specific problem within the Virtual Ship Project is one of having to filter unwanted data. It is expected that a hierarchical federation community structure will implicitly provide the necessary data filtering.

There are two main goals in establishing hierarchical federation communities as a simulation tool. The first is in the implementation of a communication structure that provides the necessary functionality for performing simulations. The second goal is a distribution framework for autonomously distributing the components of the distributed simulation, executing the simulation and returning results. Initial frameworks for these two goals are presented in the paper. The paper also presents some initial work in the development of the communication framework for hierarchical federation communities.

Keywords: HLA, Federation Communities, Federations of Federations, Hierarchical Federations, Virtual Ship.

1 Introduction

The Virtual Ship Project provides a mechanism by which simulations of ship systems are brought together in a distributed manner to create a virtual representation of a warship. How these simulations of ship systems are brought together is defined in [1] and is referred to as the Virtual Ship Architecture (VSA).

The VSA builds on the High Level Architecture (HLA), the IEEE standard for modelling and simulation [2,3,4]. The HLA is particularly suited for performing distributed simulation. In the terminology of the HLA a distributed simulation is a federation. A federation is comprised of federates, the individual software processes participating in the distributed simulation. The federates communicate data documented in a Federation Object Model (FOM). The data capabilities of a federate, i.e., the data the federate provides to the federation and the data which the federate requires from the federation, is documented in a Simulation Object Model (SOM).

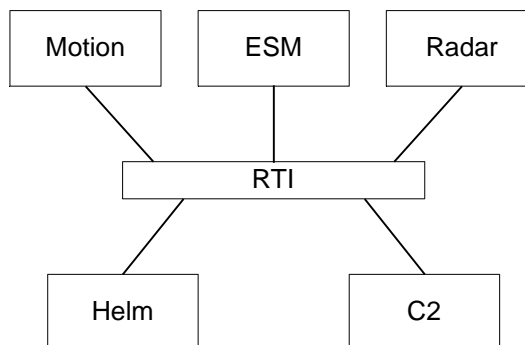


Figure 1: Single Ship Federation.

The Object Model Template (OMT) [4] defines the structure of the FOMs and the SOMs. The OMT defines an XML Document Type Definition (DTD) [6].

The federates in a federation communicate via a process known as the Run Time Infrastructure (RTI). The RTI provides services defined in the HLA Federate Interface Specification [3]. Each federate provides a set of RTI initiated services to the RTI allowing the RTI to perform callbacks on the federate. These callback services are also defined in the HLA interface specification.

The term federation community is defined in [5] as a collection of RTIs and federations working together to achieve a common goal. In a hierarchical federation community, federates are grouped to form federations and federations are grouped to form federation communities. These federation communities can then be grouped to form higher order federation communities.

2 Motivation

The motivation for looking at hierarchical federation communities as a simulation tool for use within the Virtual Ship stems from the “systems of systems” approach to simulation used by the Virtual Ship Project.

In a Virtual Ship federation, each federate is modelling a component of the ship. The ship is simulated as a whole through the interaction of these models. A federation simulating a single ship is shown in Figure 1. The federation consists of a Motion federate, modelling the kinematics of the ship, a Helm federate, providing steering of the ship, a C2 federate, modelling the command and control system of the ship, a Radar federate and an ESM federate modelling a radar sensor and an

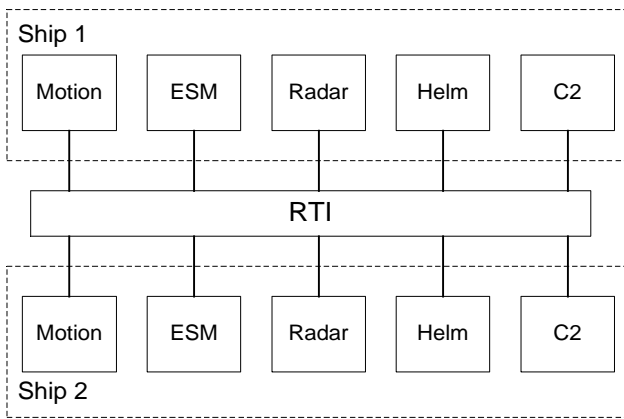


Figure 2: Multiple Ship Federation.

ESM¹ sensor respectively. The federates communicate via the Run Time Infrastructure (RTI). In a federation simulating a single ship, it is guaranteed that the data received by a federate is generated by a federate modelling a component on the same ship.

A problem arises when more than one ship is simulated in a single federation. A two-ship federation is shown in Figure 2. In a multi-ship federation, the data received by a federate has potentially been generated by a federate modelling a component residing on another ship. There exists a requirement to associate data flows with the ship that contains the component that generated the data. The approach currently used within Virtual Ship federations is to tag intra-ship data with an identifier indicating the ship within which the data was generated. Producers of the intra-ship data must add this identifier to all outgoing messages while consumers of intra-ship data are required to filter incoming data based in the ship identifier. This solution is referred to as local data filtering [1].

Performing local data filtering is not an ideal situation for a number of reasons:

- It imposes additional coding requirements on the federate developer.
- There is a requirement to send additional data between federates, increasing the network traffic.
- In the case where the ships are being simulated at geographically remote locations, there is no constraint on data intrinsic to a ship at one location being communicated to the ship at the remote location only to be filtered.
- It is counter-intuitive to the physical system, i.e., the components of a real ship are not presented with data generated by a component on another ship.

An alternative to local data filtering is desired.

3 Hierarchical Federation Communities

There is no need to perform local data filtering when there is only one ship being simulated within a federation.

¹ An ESM (Electronic Support Measures) sensor is a passive sensor that detects electromagnetic radiation.

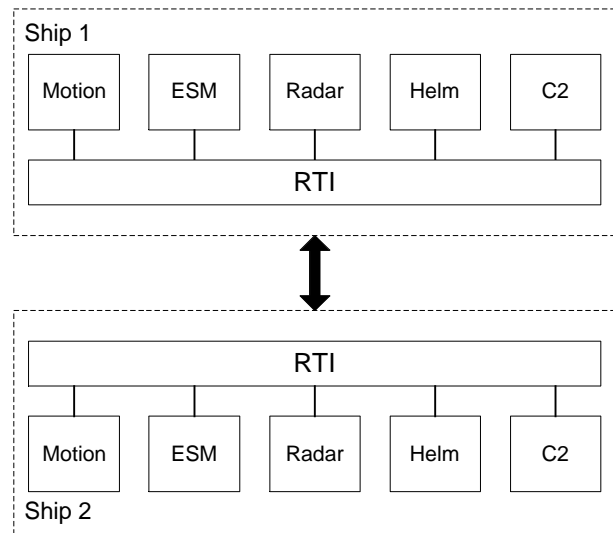


Figure 3: Multiple Ships Multiple Federations.

This suggests an alternative for performing multiple ship simulations; simulate each ship within its own federation and enable inter-ship communication via some means of inter-federation communication. A two-ship federation community is shown in Figure 3.

This architecture addresses the issues identified with local data filtering:

- Since each ship is contained within its own federation, data intrinsic to the ship can be kept from being presented to other ships (by not passing the data to the other federations). Thus, federates no longer need code to perform local data filtering.
- There is no need to augment data flowing within a federation as it is known to be intrinsic to the ship being simulated in that federation.
- Complete federations can be located at geographically remote locations, with communication between federations kept to a minimum.
- The simulation architecture now more readily reflects the physical system being simulated.

This architecture still has the potential for requiring local data filtering to be performed. Consider a scenario consisting of opposing naval task groups, each task group containing three ships. The ships are simulated as federations and all the federations are brought together as a federation community. There exists a problem in that data generated by one ship is received by all ships, including those of the opposing force. Local data filtering can be used as a solution, tagging data with a force identifier, but the problems mentioned previously are applicable.

Applying the federation community solution again allows for a natural containment of data. The federations representing the ships of a common force are grouped into distinct federation communities. The two federation communities are then grouped to form a higher order federation community that represents the simulation as a

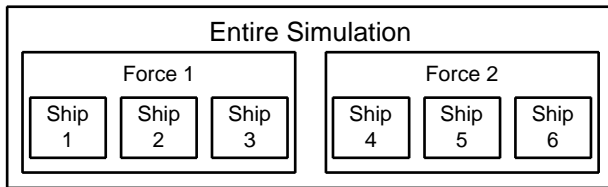


Figure 4: Multiple Ships Hierarchical Federation Community.

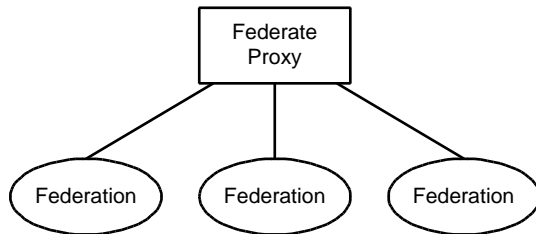


Figure 5: A Federate Proxy.

whole. Figure 4 shows this simulation structure. As can be seen, the simulation structure has a hierarchical nature.

4 Hierarchical Federation Community Architecture

Although hierarchical federation communities solve the problems of the local data filtering approach, an entirely new simulation framework needs to be established. There are two main tasks identified in establishing a simulation framework for using hierarchical federation communities as a simulation tool for Virtual Ship simulations; the definition of a communications framework and the definition of an autonomous distribution framework. These goals are discussed below.

4.1 Communications Framework

A simple means of performing communication between federations is via a Federate Proxy [5]. A Federate Proxy is a process that is joined to two or more concurrently executing federations, as shown in Figure 5.

With respect to a federation, the Federate Proxy acts as a proxy for the remote federations. All data sent from a remote federation appears to have been sourced from the Federate Proxy. The Federate Proxy is also responsible for filtering out any data in which the destination federation is not interested.

There is a problem with this approach for federation communities that are geographically distributed. In this case the Federate Proxy is local to at most one of the federations. This has the implication that data from remote federations must traverse the wide area network linking the Federate Proxy before potentially being filtered by the Federate Proxy.

A solution to this problem is to distribute the Federate Proxy into components capable of executing locally to a federation. These local components are able to filter irrelevant data before communicating to the other components. Communication between federations is then enabled via communication between the local federate proxy components. We term this architecture a

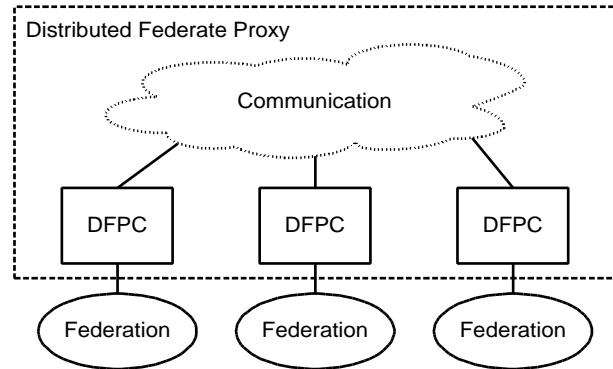


Figure 6: Distributed Federate Proxy.

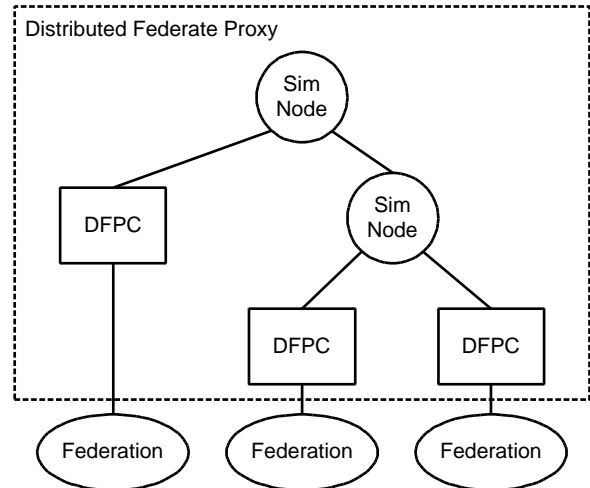


Figure 7: A Hierarchical Federation Community.

Distributed Federate Proxy (DFP) with the component executing local to a federation termed a Distributed Federate Proxy Component (DFPC). The structure of a DFP is shown in Figure 6.

The undefined communication element in Figure 6 needs to support communication between hierarchically related simulation components. As such it is natural to structure the communication framework as a hierarchy of simulation nodes, or SimNodes. Each SimNode represents the root of a federation community connected below it. At the lowest level a SimNode connects only DFPCs. At higher levels a SimNode connects a collection of DFPCs and SimNodes. A hierarchical federation community constructed from SimNodes is shown in Figure 7.

Just as the DFPC's are responsible for filtering data from the federations they are connected to, a SimNode is responsible for filtering data from the federations/federation communities that are connected below it. This architecture also allows the opportunity to perform data transformation and/or data aggregation between levels in the hierarchy. Data transformation could be required when a data type is represented in different units at different levels. Data aggregation allows a systems of systems view of an entity (such as a Virtual Ship) to be represented as a single entity at higher levels.

Communication between the components of the simulation hierarchy (DFPCs and SimNodes) must

provide the information necessary for performing simulations as if the entire set of federates were connected to a single federation. This information is defined for a single federation in the HLA Federate Interface Specification [3]. It therefore makes sense to utilise this information in the communications between components in the simulation hierarchy. The information can be utilised by having each SimNode appear as an RTI to the components connected below it. Thus, a component uses the RTI services for communication to the parent SimNode, while the SimNode uses the RTI initiated services to communicate to its connected children. This poses the questions, why not use an existing RTI as the SimNode? The answer is two-fold. Firstly, the SimNode performs custom processing, such as data transformation, not found in the standard implementation of an RTI, and secondly the current RTI in use by the Virtual Ship Project does not allow a federate to simultaneously connect to more than one RTI process.

4.2 Distribution Framework

Hierarchical federation communities using the simulation architecture presented above have the potential to contain a large number of simulation components. As an example, consider the federation community presented in Figure 4. Here the top-level federation community contains two federation communities, each containing three federations. Each federation contains five federates plus an RTI. So before counting the components of a simulation hierarchy required to link the federations there are 36 (30 federates and 6 RTIs) processes. Adding the simulation components yields six DFPCs (one for each federation) and three SimNodes (one for each distinct federation community). Thus the number of processes in this federation community totals 45.

Distributing such a large number of processes to computers, beginning the simulation and retrieving results from the simulation manually is sure to be an arduous task, a task that stands to be automated.

There is also a potential problem with finding the necessary resources required for performing such a large simulation. One place where computing resources are available is the Internet. Making the computers and LANs connected to the Internet available for performing distributed simulations forms another part of the distribution framework.

A framework for making computing resources available for use in distributed simulations revolves around the notions of a host agent, a LAN agent and a host registry. A host registry is a service, possibly web based, that maintains listings of currently available computing resources.

A host agent is a software process that executes on a computing resource and registers that resource with a host registry. The host agent has the ability to deregister itself from a host registry, which is necessary when a computing resource is needed for other purposes or is shutdown. Additionally, a host agent needs to periodically renew its registration with a registry to allow for a degree of robustness against the possibility of

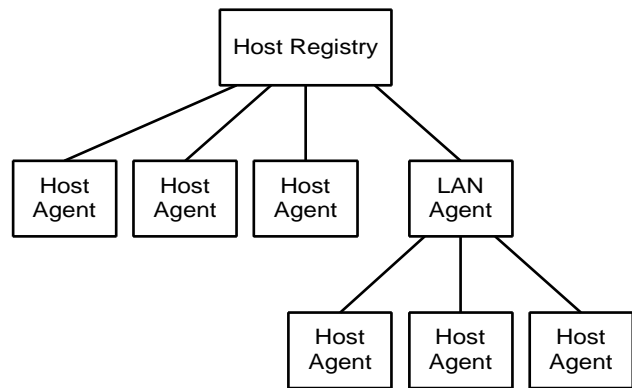


Figure 8: Computing Resource Accessibility Framework.

computing resources being unexpectedly unavailable. The host registry removes any host agents that have not renewed their registrations within a certain time period.

Many computers connected to the Internet are situated on LANs that are protected by firewalls. In such a case, it is unlikely that a host agent running on such a computer is permitted to register with a global host registry. It may be feasible for the LAN as a whole to register with the registry. A LAN agent could be run on a computer allowed to register with the global registry and also act as a proxy registry for host agents running on the computers connected to the LAN.

Such a structure resembles the simulation hierarchy presented previously. A computing resource hierarchy is shown in Figure 8.

The other key component in a distributed simulation is the software. Software resources can be made available simply through a web or ftp server. The generic software server is termed a software registry. Uploading software to the software registry is a manual process requiring a user to provide information, such as hardware and operating system requirements, about the software. There exists a possibility that the source code could be autonomously inspected to determine, or approximate, computing requirements.

During the distribution of the simulation, host agents and LAN agents identified to take part in the simulation are notified of which software to download from the software registry. In the case of a LAN agent, it downloads the software and then notifies appropriate host agents on its LAN to download software from the LAN agent's computer. As an optimisation, host agents and LAN agents may be able to cache downloaded software to improve simulation startup times for subsequent simulation executions.

In order for a distributed simulation to actually be distributed and executed, it must first be specified. The specification of a distributed simulation can occur at various levels. At the lowest, most detailed level all components of the distributed simulation are defined. The federates that are to partake are chosen, grouped into federations and allocated to hosts. The linkage of the federations is defined via the definition of a hierarchy of SimNodes, and each SimNode is allocated to a host.

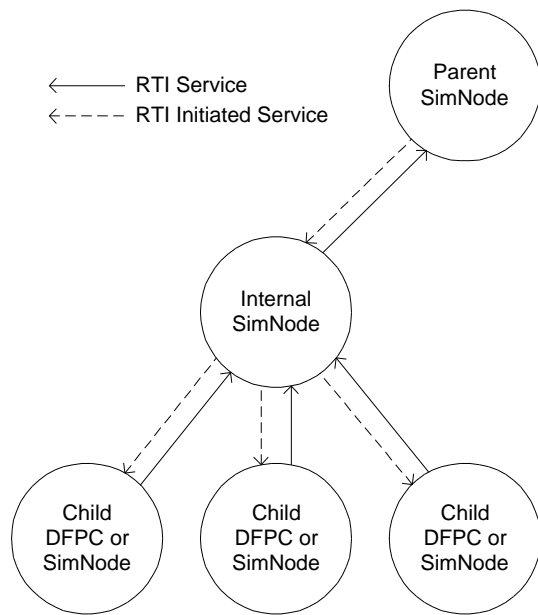


Figure 9: In/Out communication of an internal SimNode.

At a slightly more abstract level just the structure of the federation community is defined, i.e., the selection of federates, the grouping of federates into federations and the grouping of federations into federation communities. The actual allocation of simulation components to computing resources is automated. An experienced user may wish to design in this fashion and then manipulate the allocation of software resources to computing resources generated by the allocation algorithm. The decisions encoded in the allocation algorithm must be designed to minimise network usage. An initial rule to achieve this minimisation is to allocate the federates of a federation to computing resources on a single LAN. The allocation algorithm must also consider the hardware requirements of a software resource. Thus, a host agent is required to provide some information to the host registry as to the capabilities of the computing resource that it represents. Similarly, when uploading a software resource to the software registry, hardware requirements must be specified.

At a completely abstract level the distributed simulation can be designed by specifying data requirements. An algorithm responsible for selecting the federates, grouping the federates into federations and grouping federations into federation communities is then employed. Once again a user may want to manipulate the output of the algorithm.

Once specified a distribution manager is responsible for notifying host and LAN agents of software to download. Once downloaded, the distribution manager sets the simulation running and, at the conclusion, retrieves output generated and returns it to the user.

5 Progress

Progress to date has focused on the implementation of the DFPC and SimNode processes and establishing a communications mechanism between them. The communication discussed in Section 4.1 is structured on

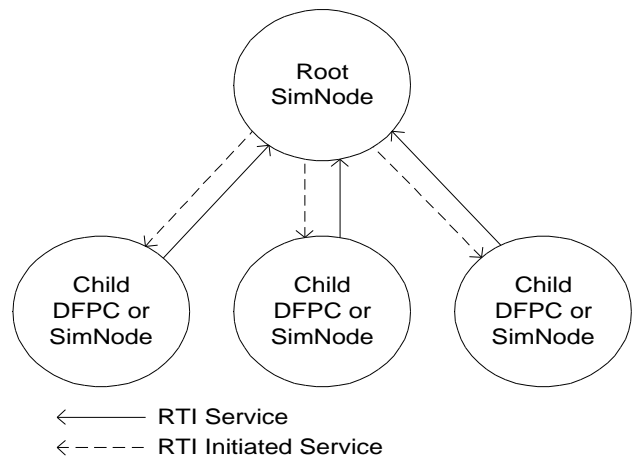


Figure 10: In/Out communication of a Root SimNode.

the services defined by the HLA Federate Interface Specification. Focusing on a child-parent pair in the simulation hierarchy, the child issues RTI service calls to the parent, while the parent issues RTI initiated service calls to the child. So each parent in the simulation hierarchy acts as an RTI to its child nodes.

A SimNode is either an internal node of the simulation hierarchy, or is the root of the complete simulation hierarchy. In the case of an internal node, a SimNode presents an RTI to its child nodes while connecting to the RTI of its parent. Thus, communication from the SimNodes children (in the form of RTI service calls) that need to proceed up the simulation hierarchy can be mapped directly onto the same RTI service call for communication to its parent. Similarly, when communication is received from its parent (via an RTI initiated service call), it can be mapped onto the same RTI initiated service call for communication to its children. This communication flow is shown in Figure 9.

When the SimNode is the root, it need only present an RTI to its children. Thus, the SimNode need only receive RTI service calls and only send RTI initiated service calls. This is illustrated in Figure 10.

A DFPC is always a child node in the simulation hierarchy, and therefore connects to the RTI of a parent SimNode. However, a DFPC also connects to the RTI of the federation to which it is assigned. In effect the DFPC has two RTI connections. This results in communication received from the DFPC's parent in the simulation hierarchy (an RTI initiated service call) needing to be mapped onto an RTI service call for forwarding to the federation. Similarly, when the DFPC receives an RTI initiated service call from the federation, it must be mapped onto an RTI service call for forwarding to the DFPC's parent in the simulation hierarchy. This communication is illustrated in Figure 11.

An XML messaging format has been chosen to encode the services and responses for communication between simulation components. There were two reasons for choosing XML:

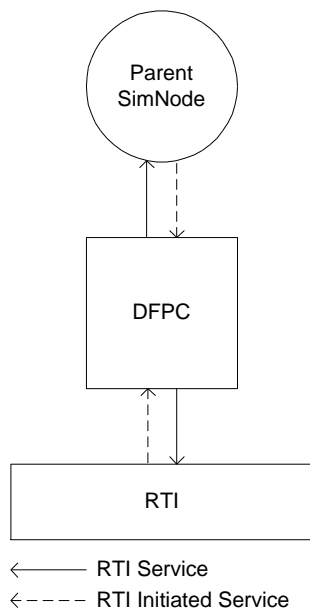


Figure 11: In/Out communication of a DFPC.

1. The clear text nature of XML makes the communication between simulation components easier to understand and debug.
2. The XML structure of the messages allows for the use of an XSLT [7] processor for performing data transformations.

The obvious disadvantage of using XML as a messaging format is the larger message sizes due to the extra information conveyed in the XML structure.

The structure of the XML messages mimics the SOAP [8] structure for XML documents. The XML Schema [9] of the containing structure for the XML messages is shown in Figure 12.

The root of an XML message is an element named Envelope. The two direct children of the Envelope element are the Header and Body elements, in that order. The Header element contains a single Identification child element. The Identification element is used for two purposes:

1. Indicating whether the message is a request, i.e., a service invocation, or a response, i.e., the response from a service invocation.
2. A unique identifier is associated with the message to allow matching of request and response messages in the case of asynchronous service invocation. Presently only synchronous service invocation is being used so the identifier is redundant.

The structure of the Body element is dependent on the service or response being passed. Some of the RTI service invocations implemented to date are InitializeRTI, CreateFederationExecution, JoinFederationExecution, ResignFederationExecution, DestroyFederationExecution and SendInteraction. In addition, the RTI initiated service ReceiveInteraction and the services dealing with synchronization points (both RTI implemented and RTI initiated) have been implemented.

```

<xsd:element name="Envelope">
  <xsd:element name="Header">
    <xsd:element name="Identification"
      type="xsd:long">
      <xsd:attribute name="type">
        <xsd:simpleType>
          <xsd:enumeration type="xsd:string">
            request
          </xsd:enumeration>
          <xsd:enumeration type="xsd:string">
            response
          </xsd:enumeration>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:element>
  </xsd:element>
  <xsd:element name="Body">
    <!-- envelope content is encoded here -->
  </xsd:element>
</xsd:element>
  
```

Figure 12: XML Schema for XML message container.

```

public java.util.Properties initializeRTI(
    java.util.Properties properties )
throws
    InitializePreviouslyInvoked,
    BadInitializationParameter,
    RTIinternalError;
  
```

Figure 13: Java API structure for the InitializeRTI service.

The InitializeRTI service is used to initialise the RTI and is called before any other service. The Java API format of this service is shown in Figure 13.

The XML Schema for the service invocation is shown in Figure 14.

The XML Schema for the InitializeRTI service contains zero or more instances of the Property element, which encodes each of the entries of the Properties instance supplied to the method call. A document conforming to the schema is inserted as the only child of the Body element of the Envelope container presented in Figure 12.

The XML Schema for the response from the InitializeRTI service is shown in Figure 15.

The root of the InitializeRTI response is once again the InitializeRTI element. The child of this element is either a Properties element or an Exception element. The structure of the Properties element contains a Property element for each entry in the Properties instance returned from the InitializeRTI service call. If the InitializeRTI instance threw an exception, it is returned via the Exception element, which is structured as in the XML Schema shown in Figure 15. Once again, a document conforming to the response schema is inserted as the only child of the Body element of the Envelope structure.

The CreateFederationExecution, JoinFederationExecution, ResignFederationExecution and DestroyFederationExecution services have XML Schemas defined for

```
<xsd:element name="InitializeRTI">
  <xsd:element name="Property"
    minOccurs="0"
    maxOccurs="unbounded">
    <xsd:element name="Name"
      type="xsd:string"/>
    <xsd:element name="Value"
      type="xsd:string"/>
  </xsd:element>
</xsd:element>
```

Figure 14: XML Schema for the InitializeRTI service invocation.

```
<xsd:element name="InitializeRTI"
  type="InitializeRTIUnion"/>

<xsd:complexType name="InitializeRTIUnion">
  <xsd:union memberTypes="Properties
    Exception"/>
</xsd:complexType>

<xsd:complexType name="Properties">
  <xsd:element name="Property"
    minOccurs="0"
    maxOccurs="unbounded">
    <xsd:element name="Name"
      type="xsd:string"/>
    <xsd:element name="Value"
      type="xsd:string"/>
  </xsd:element>
</xsd:complexType>

<xsd:complexType name="Exception">
  <xsd:element name="Exception">
    <xsd:element name="ExceptionName"
      type="xsd:string"/>
    <xsd:element name="ExceptionMessage"
      type="xsd:string"/>
  </xsd:element>
</xsd:complexType>
```

Figure 15: XML Schema of response to the InitializeRTI service.

service call and response. Their definitions are omitted but it is worth noting that all RTI service calls have the potential to throw an exception. Thus a common theme in all the response schemas of the RTI services is the presence of the Exception schema shown in Figure 15.

Currently communication between parent and child (and vice versa) is synchronous. This means the sender of a message waits for a reply to the message before proceeding. This is potentially inefficient for two reasons:

1. The sender is not doing work while waiting for a reply.
2. A reply must be sent regardless of whether the sent message generates a response.

It is beneficial to consider asynchronous communication in the future, but this introduces additional complexities.

```
<FederationCommunity name="root">
  <FederationCommunity name="fc1">
    <Federation name="fedone">
      <Federate name="federate1"/>
      <Federate name="federate2"/>
      <Federate name="federate3"/>
    </Federation>
  </FederationCommunity>
  <Federation name="fedtwo">
    <Federate name="federate4"/>
    <Federate name="federate5"/>
  </Federation>
</FederationCommunity>
```

Figure 16: XML element hierarchy describing the structure of a hierarchical federation community.

The messages are being sent from sender to receiver over TCP/IP connections. A child has one dedicated connection for RTI service communication to its parent, and one connection for receiving RTI initiated service calls from its parent. Thus a parent has 2N TCP/IP connections for N child nodes, plus two connections to its parent if it is not the root SimNode. This large number of TCP/IP connections could be a bottleneck for performance and may need to be reconsidered. A possible alternative could be to have all nodes at a given level, and the parent of those nodes, communicate on a multicast address. This would allow all nodes to be a member of at most two multicast addresses.

In addition to the implementation of RTI services, implementation of functionality that uses those services has begun. In particular the Virtual Ship notion of execution management is progressing [1].

In Virtual Ship federations it is necessary for all simulation components to be present and joined to the federation before the simulation execution begins. If this were not enforced, repeatability of simulations is jeopardised due to the distributed nature of the simulation environment. This requirement forms a part of execution management. Execution management also refers to other managerial controls that allow a federation execution to proceed in a controlled and repeatable way.

In the hierarchical federation community framework, the root node is supplied with a script file that documents the managerial requirements for the federation community. This script file is an XML document and contains an element hierarchy that documents the structure of the hierarchical federation community that is to be formed for the simulation.

The element hierarchy shown in Figure 16 shows that the root node (named `root`) of the federation community has

child nodes named `fc1` and `fedtwo`. Node `fc1` is the root of a federation community and contains one child node named `fedone`. Node `fedone` belongs to a federation that contains federates `federate1`, `federate2` and `federate3`, and node `fedtwo` belongs to a federation containing federates `federate4` and `federate5`.

The root of the simulation hierarchy monitors components joining it. Components join through the invocation of the `JoinFederationExecution` service. The root matches the names supplied in this service with entries in the structure defined in the script file. If a match is made, the part of the XML structure that is rooted at the component that just joined is supplied to the component through an invocation of the `ReceiveInteraction` service. The joining component receives this service and the XML document fragment that contains a description of the components that are expected to join to the sub tree rooted at this node.

This continues down to the federation level, which contain the DFPCs. A DFPC joins to its parent node and receives an XML document fragment in reply. The DFPC then monitors the federation to which it is joined for the joining of federates. This monitoring is via discovery of instances of the object class `Manager.Federate` (which is defined in the Management Object Model (MOM) [3]).

When the DFPC determines that all federates have joined, by comparison against the document fragment supplied to it from the original script file, it sends an interaction to its parent acknowledging that its simulation structure is created. Once a node receives such an acknowledgment from all its children, it sends the same interaction to its parent. Eventually the root of the simulation hierarchy receives acknowledgments from each of its child nodes and the entire framework is constructed. The root now knows that the simulation hierarchy is constructed and ready to begin simulation but the rest simulation hierarchy does not know. Informing the rest of the simulation hierarchy is performed using synchronization points.

Services exist in the HLA interface specification for synchronization of the federation at user-defined points. At the single federation level synchronization points work in a number of steps, as outlined below:

1. A synchronization point is registered with the federation. A label (string) is used to identify the synchronization point.
2. The synchronization point is announced to all federates joined to the federation within which the synchronization point was registered.
3. When a federate achieves the functionality defined by the synchronization point, it announces that the synchronization point has been achieved.
4. Once all federates have achieved the synchronization point, an announcement that the federation is synchronized on that point is made.

After the simulation hierarchy is completely created, the root node announces a synchronization point to its

children (the registration of the synchronization point is implicitly done within the root node when it has confirmation that the simulation hierarchy is fully constructed). This synchronization point is used to move the simulation components into their simulating state upon synchronization of the federation community.

Processing of the synchronization point progresses in a similar way to that of the acknowledgments of simulation structure created. A node informs its child nodes of the synchronization points down to the DFPC level. The DFPCs then wait for their federations to synchronize before acknowledgment is passed back up the hierarchy. Each node waits for acknowledgment from its children before acknowledging to its parent. Once simulation components receive the federation synchronized announcement, they are free to begin simulation.

At the conclusion of the simulation execution, the root node again announces a synchronization point, this time with the intention of having the simulation components shutdown. The simulation shutdown is an orderly, bottom up shutdown. This means that the federates first resign from the federations they are joined to. The DFPC's monitor this and when all federates have resigned, they acknowledge (via an interaction) to their parents and then shutdown. A node sends a shutdown acknowledgment message to its parent when it has received shutdown acknowledgments from each of its children. This progresses until the root node has received acknowledgment from each of its children, at which point it terminates and the simulation hierarchy has been completely shutdown.

6 Future

Work in the immediate future will be to more fully develop the DFPC and `SimNode` processes. This will involve the definition of XML Schemas for additional RTI services, RTI initiated services and their responses. Associated with the encoding of the services is the establishment of functionality within the DFPC/`SimNode` to process the services.

As communication is presently XML based it may be worthwhile conforming to the SOAP specification. This may also allow the development of the DFPC/`SimNode` as a web service. This has the advantage of utilising a great deal of existing technology in the form of web servers.

The features of the autonomous distribution framework resemble the features present in the Jini™ architecture [10]. For example, host agent registration with the host registry resembles the Jini™ notions of Discovery and Join. Additionally, the periodic registration updates performed by the host agents resemble the Leasing mechanism present in Jini™. Determining if Jini™ can be leveraged for use within the autonomous distribution framework is to be ascertained.

In the long term the use of hierarchical federation communities as a simulation tool will be evaluated. The evaluation will be against a single federation running federates that perform local data filtering. Some metrics

that will be measured include simulation speed, coding complexity and network usage. The evaluation will be performed in two contexts based on the distribution of components. One context will have all components executing locally, i.e., on the same LAN. The other context will have the components geographically distributed.

7 Conclusions

Hierarchical federation communities offer a unique means of performing distributed simulation. The hierarchical structure of these federation communities make them natural frameworks for performing simulations of physical systems that have an inherently hierarchical nature. Such a simulation is being performed in the Virtual Ship Project.

A new concept for linking the federations in the federation community, the Distributed Federate Proxy, provides data filtering capabilities. This is essential for minimising network usage in large simulations, especially when the simulation is geographically distributed.

Due to the potential large size of these distributed simulations, a framework for managing and automatically distributing the simulations is to be defined. This also provides a degree of transparency in the use of distributed simulations as a simulation tool. This is particularly important if distributed simulations are to be used by non-simulation experts.

Some initial work has been performed in the development of the framework of the hierarchical federation community. This involves the development of two software products, the SimNode and the DFPC. These processes are connected, via TCP/IP connections, to form the hierarchical simulation structure. Communication between the components of the simulation structure is encoded in XML documents.

Future work will revolve around completing the DFPC and SimNode processes and establishing the framework and process for autonomously distributing hierarchical federation communities.

8 References

1. Best J.P., et al. Virtual Ship Architecture Description Document Issue 1.00. *DSTO General Document 0257 (DSTO-GD-0257)*. DSTO Edinburgh, South Australia, 2000. (<http://www.dsto.defence.gov.au/corporate/reports/DSTO-GD-0257.pdf>).
2. IEEE Std 1516-2000. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules. *The Institute of Electrical and Electronic Engineers*. March 2001.
3. IEEE Std 1516.1-2000. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification. *The Institute of Electrical and Electronic Engineers*. March 2001.
4. IEEE Std 1516.2-2000. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification. *The Institute of Electrical and Electronic Engineers*. March 2001.
5. Myjak, M., Clark, D. and Lake, T. RTI Interoperability Study Group Final Report. *Proc: Simulation Interoperability Workshop*. Orlando, September 1999. <http://www.sisostds.org>.
6. W3C Recommendation. Extensible Markup Language (XML) 1.0 (Second Edition) 6 October 2000. *World Wide Web Consortium*. <http://www.w3.org/TR/REC-xml>.
7. W3C Recommendation. XSL Transformations (XSLT) Version 1.0 16 November 1999. *World Wide Web Consortium*. <http://www.w3.org/TR/xslt>.
8. W3C Note. Simple Object Access Protocol (SOAP) 1.1. 08 May 2000. *World Wide Web Consortium*. <http://www.w3.org/TR/SOAP>.
9. W3C Recommendation. XML Schema Part 0: Primer. 2 May 2001. *World Wide Web Consortium*. <http://www.w3.org/TR/xmlschema-0/>.
10. Sun Microsystems. Jini™ Architecture Specification. Version 1.1. October 2000. *Sun Microsystems*. <http://www.sun.com/jini/specs>.