RELIABLE MULTICAST IN THE STOW RTI PROTOTYPE

Harry Wolfson <HarryWolfson@LL.MIT.EDU>
Steven B. Boswell, PhD <Boswell@LL.MIT.EDU>
Daniel J. Van Hook <dvanhook@LL.MIT.EDU>
Stephen M. McGarry <smcgarry@LL.MIT.EDU>
MIT Lincoln Laboratory
Lexington, MA 02173
Phone: 617-981-7450
FAX: 617-981-7455

KEYWORDS
RTI, HLA, Multicast, Reliable, Discovery

ABSTRACT

The High Level Architecture (HLA) Runtime Infrastructure (RTI) provides a variety of message Transport Services for different communication needs. These Transport Services include Best Effort, State Consistent, Minimum Rate and Reliable. A Reliable Message Transport Service requires an implementation of a reliable multicast protocol for efficiency in large scale simulation systems. This paper will briefly describe reliable multicast technologies, while focusing on the emulation that was chosen for the prototype implementation of the HLA RTI for STOW.

The Discovery Service is a mechanism for allowing distributed clients and providers of an arbitrary service to locate one another dynamically. The RTI's Reliable Service makes use of Discovery to allow Reliable message clients to find and connect to Reliable Distributor servers; and to allow the servers to find and connect to each other.

This paper will describe the implementation of the Reliable and Discovery Services in the prototype HLA RTI for STOW, as well as a description of the APIs for simulation designers.

1.0 INTRODUCTION

As part of a DoD/DMSO team, MIT Lincoln Laboratory is developing software for an initial "freeware" version of the Run-Time Infrastructure (RTI) software needed to support DoD's transition to the High Level Architecture (HLA) for Modeling and Simulation. In particular, the Lincoln Laboratory group is implementing software to support the RTI Data Distribution Management Services and related underlying communication components. This software is being prototyped and tested as part of DARPA's Synthetic Theater of War (STOW) Advanced Concept Technology Demonstration (ACTD). After appropriate testing, this software will be integrated into the government-furnished RTI software package and will be made available to the broad Modeling and Simulation community.

The Reliable Message Transport Service, which is described in this paper, is one of the underlying communication components that is part of the RTI. The reliable multicast that was developed for the prototype RTI is an emulation of a true multicast protocol. The tradeoffs of design and performance as a result of this emulation will be described. Also described in this paper is the Discovery Service, which is a general purpose service that can be utilized by any distributed set of clients and providers within the RTI. Discovery is used by the Reliable Service to provide autonomous connectivity between the its servers and clients.

2.0 MESSAGE TRANSPORT SERVICES

The RTI provides a set of Message Transport Services that the Federate may choose from when sending updates or interactions. This variety allows the Federate to select an optimum service based on the requirements of a particular message. The following is a summary of descriptions for these services that was presented at the last Workshop[1].

2.1 Best Effort

This work is sponsored by the Defense Modeling and Simulation Office (DMSO) under Air Force Contract No. F19628-95-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Air Force.

The best effort (BE) transport service transmits interactions and attribute values with no reliability mechanism. Using BE, the messages may be received out of order. This service is available to support construction of application protocols, or for transport over lossless media such as shared memory.

2.2 Minimum Rate

The minimum rate (MR) transport service most closely emulates DIS transport mechanisms. Interactions and attributes are sent at a minimum rate even if no attribute values have changed. Like BE, there is no reliability mechanism and messages may be received out of order. MR is an efficient and appropriate service for interactions and attributes that change value fairly frequently.

2.3 State Consistent

The state consistent (SC) transport service ensures delivery of the latest attribute values. It does not, however, guarantee delivery of any intermediate messages. Attribute value updates may be requested from the client multiple times by SC to boost reliability.

2.4 Reliable

The reliable (RE) transport service guarantees delivery of all interactions and attribute values to all subscribed receivers, in the order sent.

3.0 RELIABLE MULTICAST

The DIS method of exchanging information among simulations was for each information producer to **broadcast** each message it generated, insuring that every other simulation would receive that message, as well as every other computer connected on the network. This leads to increased network traffic and an increase in the processing load on simulations not interested in each and every message. To solve this problem, the HLA RTI uses a variety of methods to reduce network traffic and reduce processor loading including “multicast addressing to route all relevant data, and minimal irrelevant data, from producers to consumers” [2].

Most multicast protocols are implemented with UDP, which provides for multicast addressing. However UDP suffers from several disadvantages, namely: relatively poor message reliability, messages arriving out of order, and poor fault-tolerance. Some method of reliable message transfer is required by the HLA RTI to solve these problems.

3.1 Reliable Multicast Protocols

Several reliable multicast protocols have been proposed in recent years, including RAMP[3], RMP[4] and SOM[5]. Some of these protocols are based on UDP, to take advantage of its inherent multicast abilities, and then build the reliable message communications on top of that. Some assume a highly reliable transmission channel with low bit error rate, and then provide reliability based on NACK (negative acknowledgments) from the intended receivers. This usually requires a third party (in addition to the sender and receiver) who keeps track of message ids and coordinates the retransmission of lost messages. Some of these protocols are optimized for a single sender with several receivers (one-to-many), or are optimized for the scenario where the set of potential senders and receivers is relatively fixed.

3.2 Emulation of Reliable Multicast in the Prototype RTI

The message traffic environment in the HLA/RTI is very dynamic. As opposed to a relatively simple one-to-many scenario, any sender may wish to send its messages to any number of receivers. Due to the nature of these simulations, and the way that subscriptions are based on filter space[2], a receiver will change the multicast groups to which it is subscribed numerous times during a simulation exercise. Also a single simulation will nominally be subscribed to many multicast groups, and it will potentially be transmitting to many completely different (and dynamically changing) sets of receivers.

In order to provide for reliable message service, the prototype RTI for STOW (RTI-s) uses TCP instead of a true UDP/multicast protocol. This design choice was made after exploring the use of CORBA in an earlier prototype of the RTI. (CORBA provided a reliable message service but was deemed to be an inappropriate framework for the current RTI.) By using TCP, we gain the inherent advantages of TCP that are required for a reliable message service, namely: reliable delivery, sequential ordering of messages, and a measure of robustness, or fault-tolerance. Of course since TCP is a point-to-point communication protocol, we need to provide a way to distribute messages to numerous recipients (point-to-multipoint). This is accomplished by implementing a client/server approach, where the client only needs to maintain a single connection with one server, and the server is responsible for distributing messages to all intended recipients. Although this requires setting up a server and having clients connect to it, by using the Discovery Service, described below, most of that task is autonomous and there are only two API calls required to fully implement this client/server model.

The obvious disadvantage of this emulation over a true multicast protocol is increased traffic on the network, which is why this implementation of the Reliable Transport Service is not recommended for heavy message users. Rather it is more suited for those that require reliable and robust message delivery. One of the more subtle advantages of our emulation is that reliable message service is not restricted to a limited number of multicast groups as a function of the particular network or interface available to the host computer.

4.0 DISCOVERY SERVICES

The Discovery Service is a mechanism provided within the RTI-s software to facilitate contact between distributed providers and clients of an application or system service. The Discovery Service consists of three fundamental capabilities. First, the supplier of a resource may register to have messages emitted at periodic intervals, advertising the availability of that resource to other members of the network, and supplying contact instructions and status information to potential clients. Second, a consumer of a service may ask to "discover" the service, that is, to be provided with contact and status information for providers currently advertising the desired service, if any. Third, the consumer of a service may ask to "monitor" a particular provider, that is, to be notified if the stream of advertisement messages from the designated provider is interrupted. A previous prototype of the Discovery Service was implemented as part of the STOW RITN program, where it was used to discover the subscription agent dynamically.

The Discovery Service is implemented in C++ in the local distribution manager portion of the RTI-s, via `discovery_manager` and certain companion classes. Following is a synopsis of the implementation and the programming interface used to access discovery services, omitting much detail.

Figure 1 presents a Rumbaugh diagram[6] that

summarizes the RTI-s discovery mechanism. In this diagram, a class is represented as a rectangular box, and named in boldface. Two optional subdivisions in the box indicate class variables and class functions, respectively.

As indicated in the uppermost box, `discover` and `monitor` are public functions invoked by users of the discovery mechanism. An invocation of `discover` causes a `discovery_request` to be added to the `requests` list of `discovery_manager`. The arguments to `discover` are passed into the entry, `discovery_request`, in addition to a client function, `discovery_callback`, which the user supplies to take receipt of any providers that satisfy the discovery request. The discovery request instructs the `discovery_manager` to listen for and accumulate advertisement messages for at least a period `collection_interval`, and to call back with an empty list if no advertisements have been received after a waiting period `timeout`. The `monitor` method places an entry in the `monitors` list of the specified provider. The user supplies a callback function to receive notification if a period `timeout` elapses without new advertisements from the specified

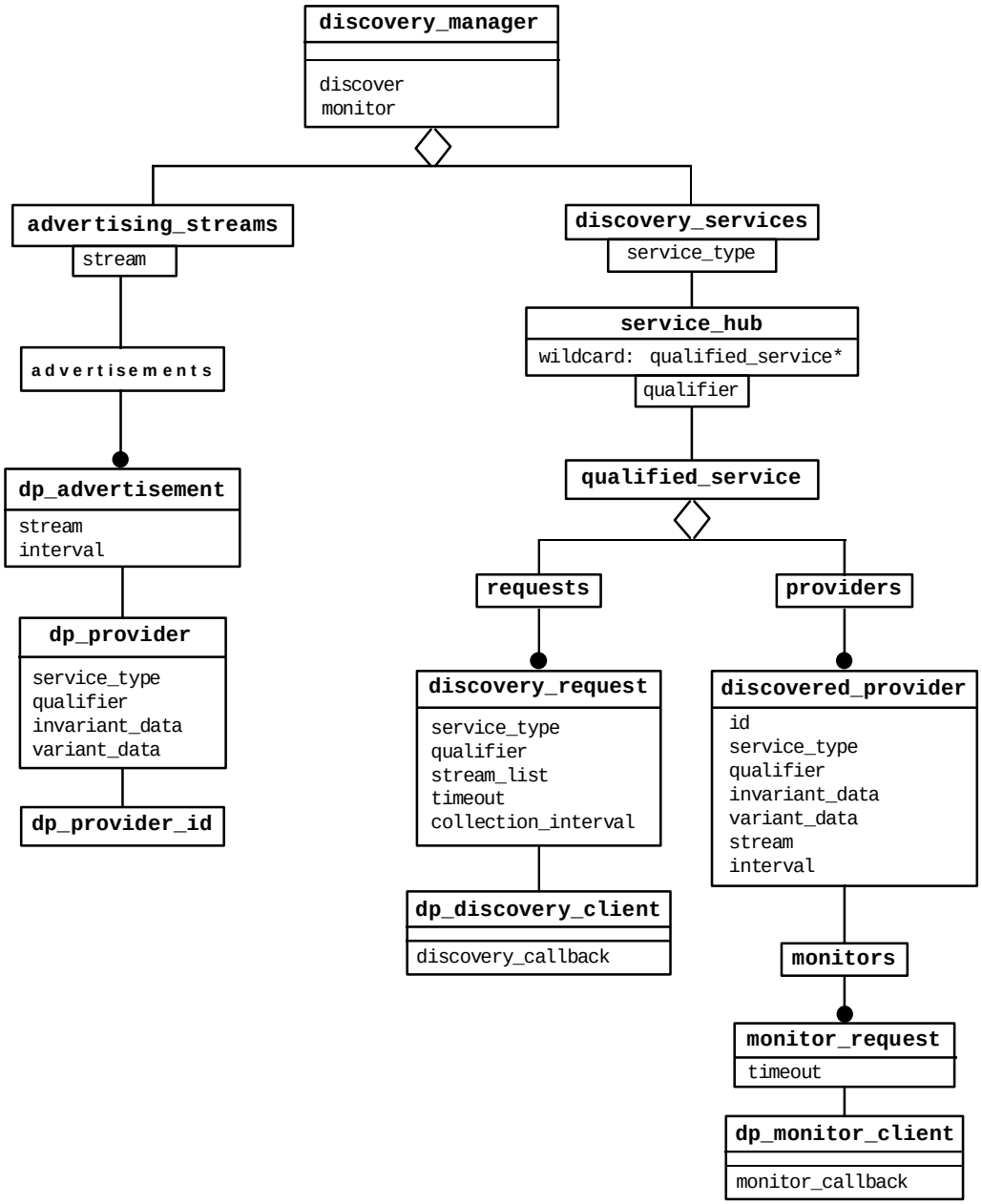

Figure 1. Rumbaugh depiction of RTI-s Discovery Service mechanism.

provider. Because each advertisement message reports the interval at which it is reissued, the consumer of a service may tailor the `timeout` threshold to match the characteristics of the advertisement being monitored.

The supplier of a service is described to the Discovery Service as an object of class `dp_provider`. The constructor for the `dp_provider` class generates an object identifier guaranteed to be unique among all service providers in the Federation Execution. The service provider may optionally give a `qualifier`, which is used for filtering, since a discovery request may ask to be satisfied only by providers matching a particular qualifier value. The member `invariant_data` is effectively an untyped byte array, which should normally contain information enabling a potential client anywhere in the Federation Execution to establish contact with the provider (*e.g.*, IP address and port number). All of the above members are permanent characteristics of a service provider, and all are encoded in each advertisement message authorized by the service provider. The member `variant_data` is an optional virtual function which, if supplied, is invoked during the construction of each advertisement packet, and may be used by the service provider to convey time-varying status information to potential consumers.

A service provider initiates an advertisement process by instantiating an object of class `dp_advertisement`, providing a reference to itself, a `stream` value, which the RTI-s uses to indicate a communication channel [1], and an integer `interval`. The `dp_advertisement` object installs itself in an `advertisements` list of the `discovery_manager` class. So long as the `dp_advertisement` object is instantiated, the Discovery Service issues an advertisement packet every `interval` milliseconds using Best Effort transport, subject to the resolution of the timer used in the `discovery_manager` class. The application user may terminate this perpetual process by deleting the `dp_advertisement` object.

The `discovery_manager` employs a timer that executes maintenance functions at regular intervals, nominally once per second, and it uses this timer to mark timeout thresholds contained in monitor and discovery request. The `discovery_manager` caches every service provider that it discovers advertising on a communication channel to which it is joined, whether a discovery request exists for the service or not. This permits the Discovery Service to provide an immediate "all currently known" response to an incoming discovery request, if the argument `collection_interval` is zero. Service providers are implicitly monitored by `discovery_manager`, and are purged from the cache if their advertisement stream becomes dormant.

5.0 DESIGN AND IMPLEMENTATION OF RELIABLE MULTICAST IN RTI-s

Reliable multicast message service is emulated in the RTI by using a set of TCP point-to-point connections. A server, called the Reliable Distributor or `reldistr`, is created to service a number of clients who wish to send and/or receive reliable messages. Once the server has established itself (turned on), it advertises itself to the Federation Execution via the Discovery Protocol. Potential clients learn about what servers are available via Discovery. Once a client has chosen a suitable server, a TCP connection between the two is established. Then the client informs the server what streams it has joined to allow for stream based filtering and forwarding.

A server can keep track of a single client, or a large list of clients. If there is more than one server in a Federation Execution, then all the servers will find out about each other via Discovery and establish TCP connections between themselves in a fully-connected mesh. The servers will exchange information between themselves so that they know the streams that the clients of each server have joined. Figure 2 shows a sample network of three `reliable_distributor` (RD) servers and their set of `tcp_socket_manager` (tcp) clients. Note that each client is connected to only one server, and that all servers are connected to each other. Also note that each RTI has its own RD server, but they remain inactive unless explicitly turned on via the API.

Whenever a client Federate sends an interaction or update using Reliable Transport, the message is forwarded to that client's server. The server looks at the stream id of that message and will forward the message to all of its clients who have joined that particular stream. The server also forwards the message to all other servers in the Federation Execution who have clients that have joined that particular stream. (Note that each reliable message has a "time to live" counter in its header that gets decremented by each server. This prevents messages from being forwarded more than they should.)

5.1 Robustness

Our Reliable implementation is based on TCP; therefore we benefit from that protocol's inherent reliability, as well as its guarantee of ordered message delivery. In addition, we provide several additional features to improve the robustness of the service.

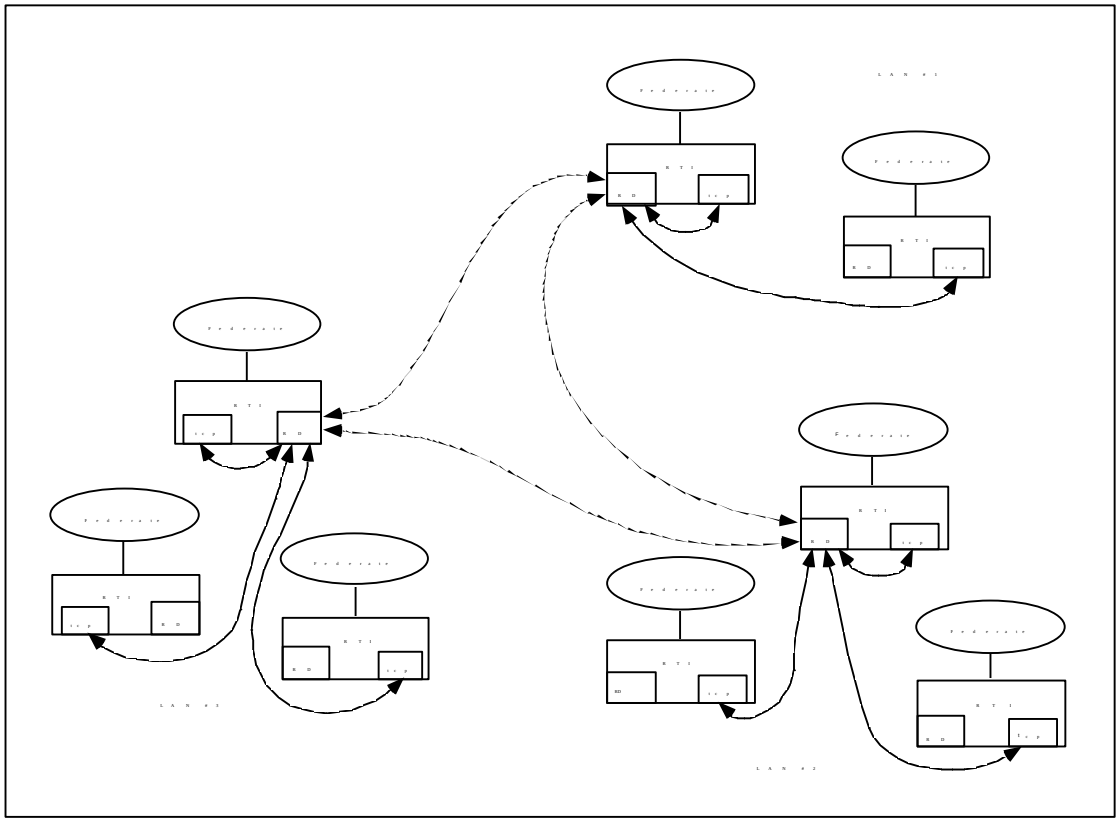

Figure 2. Sample Topology of Reliable Servers and Clients

Message Queue. Each client maintains an internal queue to temporarily hold messages that cannot be transmitted. An inability to transmit could be caused by a client that has not yet established a connection with a server, a broken connection with the server, a full TCP buffer in the kernel, etc. Each server maintains a separate message queue for each of its clients. That way if one client has a very slow connection, its message backlog can be queued so as not to disrupt the flow of messages to clients on a faster link. The size of the message queue can be set by the RID. Once a message queue begins to overflow, the oldest messages are discarded first.

Re-establishing Connectivity. Since our design is based on a client/server model, it is important that a client's connectivity with a server be quickly re-established if a disruption occurs. During normal operation, a ping protocol is used between the client and server to insure connectivity between the two, on a predetermined interval. If a broken connection is revealed during a ping, or during a normal message transfer, both the client and the server take steps to "reset" their connection state.

When a client notices a broken connection, it immediately begins to rediscover a new server. Any messages it was trying to send are saved in its message queue and will be retransmitted as soon as it establishes a connection with a new server.

When the server notices a broken connection, it simply deletes that particular client from its list of connections. Any message being transferred to that particular client is lost, but the server will continue to distribute that message to its remaining clients.

5.2 Handling artifacts of TCP

While TCP provides significant advantages and assistance to our Reliable implementation, it also has some idiosyncrasies that must be dealt with to insure ordered message delivery to higher layers in the RTI.

Message Framing. The TCP buffer in the kernel can hold more than a single message and since the communication is asynchronous between client and server, that buffer will fill up with several messages during periods of rapid message transfer. Both the client and server need to frame each message that gets pulled off of the TCP kernel buffer to insure that only a single message is passed up to the rest of the RTI.

Received Message Fragmentation and Reassembly. By using TCP, we are guaranteed that delivery of messages will be in order; however, since TCP is a "byte stream" based protocol and not "message" based, there is no guarantee that each individual message will arrive complete. In other words, TCP might deliver only a fragment of a message in a single packet. Subsequent packets will eventually contain the rest of that particular message. Both the Reliable clients and servers need to provide temporary storage for received message fragments and have the ability to reassemble a single, complete message that can then be passed up to higher layers in the RTI. The server maintains a "received message fragment buffer" for each of its connected clients. This insures that one "slow" client will not degrade performance to the rest of the clients.

Sent Message Fragmentation. Similar to the fragmentation on receive described above, TCP may send only a fragment of a message depending on the message size and the state of its internal buffers. Both the Reliable clients and servers need to provide temporary storage for partially sent message fragments and have the ability to send the remainder of the message when TCP is ready. The server maintains a "sent message fragment buffer" for each of its connected clients. This insures that one "slow" client will not degrade performance to the rest of the clients.

5.3 API

The Application Programmer's Interface (API) for Reliable is defined in `rtiAmbServices.cc`. The following is the set of calls:

```
void
RTI::RTIAmbassador::turnOnReliableServic
e (
       int theListenPort,
       const char *advertString)

void
RTI::RTIAmbassador::discoverReliableServ
ice (
       const char *discvryString)
```

A federate that will act as a Reliable Distributor server must call `turnOnReliableService()`. The argument, `theListenPort`, is the port that will be used to listen for new clients that are trying to connect. `theListenPort` can be any int16 greater than 1023. This is an optional argument that will default to 1666 if not provided. The argument, `advertString`, is an arbitrary string (with a maximum length of 128 characters) that can be used to control how clients and servers connect to each other. This is an optional argument that will default to an empty string if not supplied. Note that if the `advertString` argument is supplied, than `theListenPort` must also be supplied.

All federates that wish to send or receive Reliable message traffic must make the call `discoverReliableService()`, including a federate that is acting as the Reliable Distributor. However, a stand-alone application that is a Reliable Distributor server, and does not need to send or receive any message traffic for its own use, should not call `discoverReliableService()`.

If the creator of the Federation Execution wishes to maintain control over which clients connect to which servers, than they would supply the `discvryString` argument, which would otherwise default to an empty string. If `discoverReliableService()` is called with a `discvryString` argument supplied, than that Federate client will **only** connect to a Reliable Distributor server that uses an exactly matching `advertString` in its call to `turnOnReliableService()`.

If `discoverReliableService()` is called without supplying the argument, `discvryString` , the client will choose a server based on the following criteria:

1st choice: server is in same process as client
2nd choice: server is on same host as client
3rd choice: server is in same sub-net as client
4th choice: first server that the client discovers

Note that the order in which these API calls are made is immaterial. The implementation of Discovery insures that once a Federate makes these calls to initialize the Reliable Service, the connections will happen autonomously.

Simple Network Topology. The simplest model has several clients all connected to a single server. One Federate is selected as the Reliable Distributor. (It can be a stand alone entity that does nothing other than act as a server, or it can be a normal Federate.)

The application selected as the server calls:

```
turnOnReliableService()
```

All Federates must then initiate discovery of, and connection to, a server by calling:

```
discoverReliableService()
```

Complex Network Topology. If you have a more complex network topology that requires multiple servers, then each server would call `turnOnReliableService()`. The Discovery Service that the RTI implementation of Reliable uses will insure that all servers are interconnected into a full mesh topology. Then each Federate would call

`discoverReliableService()` once, in order to find and connect to an appropriate server.

5.3 Protocol Messages

Several special messages are used as part of the protocol for the reliable multicast in RTI-s. They allow communication between the server and its clients, as well as between all of the servers in a Federation Execution. All of the following messages are read at the `tcp_socket_manager` level and then discarded, i.e. they are not handed up to the `stream_manager`.

`JOIN_STR, LEAVE_STR.` These two messages are passed to a server by a client that has been instructed by the stream manager to join or leave a stream. These are also passed by the server to any other connected servers to insure that all desired messages are forwarded to the appropriate clients.

`RECEIVE_ALL_1, RECEIVE_ALL_0.` These messages are used when logging has been activated, or inactivated, respectively. When logging is turned on in a client, it's server will forward all messages, regardless of stream id.

`IM_A_RELDISTR.` This message is exchanged when two reldistr servers connect to each other. The servers need to know which of their clients are other servers so that they can pass on the state of their clients when they join or leave streams.

`PING.` The "ping" message is used to insure connectivity between servers and clients, and between servers. The ping message verifies an intact connection during periods without normal message traffic. Errors detected during the ping will reset the connection state to prepare the clients and servers for any subsequent real message traffic.

`TEST_MSG.` The "test message" is used for performance measurements and diagnostics.

6.0 PERFORMANCE OF RTI RELIABLE MESSAGE SERVICE

Throughput and message latency performance are functions of message size, since the underlying TCP protocol will fragment messages based on their size and the allocated TCP internal buffers. In addition, since the RTI Reliable Service depends on a server to distribute messages to all of its clients on a "point-to-point" basis, the overall performance is directly affected by the number of clients connected to each server. Measured performance data will be presented at the 97S-SIW conference.

7.0 CONCLUSIONS

The design and implementation of the Reliable Message Transport Service in the Prototype RTI for STOW has been presented. We have described how it emulates reliable multicasting, and presented comparisons between it and other multicast protocols. This emulation provides a functionally correct baseline which has been shown to have adequate performance for the RTI-s. Future work will focus on more detailed performance measurements in the context of STOW. This characterization will then be used when evaluating other reliable multicast protocols to determine their suitability for the RTI. These protocol candidates could provide an alternate choice for reliable communications with a different set of performance characteristics.

We have shown how Discovery is used by the Reliable Service to provide autonomous connectivity between distributed clients and servers. In the RTI-s, the Discovery Service is an internal mechanism, visible only within the RTI-s. However, the programming interface is quite general, and largely independent of details of the RTI. An option for future development is to add the Discovery Service to the RTI API, and make it available to application users of the RTI. To do so will require public mechanisms to allow applications to define and register arbitrary types of service, and it will require an API level mapping to the RTI stream construct.

8.0 REFERENCES

[1] Van Hook, Daniel J., McGarry, Stephen M. "A Prototype Approach to the Data Communications Component of the RTI" (previously titled "An Update on the RTI Design"), 96-15-085, Fifteenth Workshop on the Interoperability of Distributed Interactive Simulations, September 16-20, 1996.
<http://dss.ll.mit.edu/dss.web/96.15.085.html>

[2] Van Hook, Daniel J., Rak, Steven J., Calvin, James O. "Approaches To RTI Implementation of HLA Data Distribution Management Services," 96-15-084, Fifteenth Workshop on the Interoperability of Distributed Interactive Simulations, September 16-20, 1996.
<http://dss.ll.mit.edu/dss.web/96.15.084.html>

[3] Koifman, A., Zabele, S., "RAMP: A Reliable Adaptive Multicast Protocol," white paper from TASC, Inc., Reading, MA, 1996.
<http://www.tasc.com/simweb/papers/RAMP/ramp.htm>

[4] Montgomery, Todd "Design, Implementation and Verification of the Reliable Multicast Protocol," Thesis, West Virginia University, 1994.
<http://research.ivv.nasa.gov/projects/RMP/RMP.html>

[5] C.Bormann, J.Ott, H.-C. Gehrcke, T.Kerschat and N. Seifert: “MTP-2: Towards Achieving the S.E.R.O. Properties for Multicast Transport,” International Conference on Computer Communications and Networks, 1994.
<http://user.cs.tu-berlin.de/~nilss/som/som.htm>

[6] Rumbaugh, James, et. al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

9.0 ABOUT THE AUTHORS

Harry Wolfson has been a Member of the Technical Staff at MIT Lincoln Laboratory since 1985. He is responsible for the design and implementation of the Reliable Message Transport Service, and its underlying protocol, for the STOW RTI-s Prototype. Prior to joining Lincoln, Harry was a Staff Member of TRW in Redondo Beach, Ca. He received a Master of Science in Electrical Engineering from the University of Southern California with a major in Microwave Design and Communication Systems. Harry received his Bachelor of Science in Electrical Engineering from Northeastern University.

Steve Boswell is a staff member with the Distributed Simulation Systems Group at Lincoln Laboratory, MIT. He implemented the discovery mechanisms described in this paper. At Lincoln Laboratory, he has modeled airport capacity, delays and network flows in commercial air traffic, and he has worked on system design for satellite-based space surveillance. Prior to joining Lincoln Laboratory, he was an instructor and research statistician at Harvard Medical School, and faculty member in the School of Mathematics at Georgia Tech.

Daniel J. Van Hook is on the staff of MIT Lincoln Laboratory. He is a member of the development team that is producing the STOW RTI-s Prototype. His primary technical interests are in the areas of distributed systems, realtime systems, communications, signal and image processing, and software engineering. Prior to joining Lincoln Laboratory, he was employed by Loral Advanced Distributed Simulation and by BBN Advanced Simulation Division, working on distributed simulation systems including the SIMNET program.

Stephen M. McGarry is a Member of the Technical Staff at MIT Lincoln Laboratory. He is a member of the development team that is producing the STOW RTI-s Prototype.