Supporting Multiple RTIs within a Single Process

William Luebke, John Baker, Adrian Porter
Raytheon Virtual Technology Corporation
Alexandria, VA
wluebke@raytheonvtc.com, jbaker@raytheonvtc.com, aporter@raytheonvtc.com

ABSTRACT

Achieving simulation interoperability between autonomous federations is always a challenging problem. Despite the fact that different federations might accomplish seemingly similar tasks, they frequently implement solutions using drastically different approaches. A recent federation bridge development project implemented a unique approach to federation interoperability between differing Run-Time Infrastructure (RTI) solutions, Federation Object Models (FOMs), and federation level protocols. The ability to provide interoperability between two High Level Architecture (HLA) federations in a single software process using *different versions* of the RTI allows for an interoperability solution that requires no implementation changes to either federation while demonstrating the collective benefits combining the two federations.

Providing interoperability between two HLA federations in a single software process using *different versions* of the RTI poses a unique challenge, as one normally cannot compile and link an application in this way. This challenge can be overcome using a specialized proxy that enables different versions of the RTI to simultaneously coexist in a single software process. This paper details the technological approach of using such a proxy for a federation bridge, including its applicability, architecture, and performance characteristics. The approach is proven via the successful implementation of a federation bridge that enables interoperability between two federations using the DMSO 1.3 NG v4 and Raytheon VTC NG Pro v2.0.4 RTIs. Examples of using the techniques presented in this paper in other situations are also given, as well as alternative approaches.

ABOUT THE AUTHORS

William Luebke is a Senior Software Engineer with Raytheon Virtual Technology Corporation. He holds an M.S. in Computer Science from the University of North Carolina at Chapel Hill and a B.S. in Computer Science and Mathematics from the Virginia Polytechnic Institute and State University.

John Baker is a Senior Systems Engineer with Raytheon Virtual Technology Corporation. He holds an M.S. in Industrial Engineering from the Pennsylvania State University and a B.S. in Industrial Engineering from the University of Central Florida.

Adrian Porter is a Senior Software Engineer with Raytheon Virtual Technology Corporation. He holds a B.S. in Computer Science from the Virginia Polytechnic Institute and State University.

Supporting Multiple RTIs within a Single Process

William Luebke, John Baker, Adrian Porter
Raytheon Virtual Technology Corporation
Alexandria, VA
wluebke@raytheonvtc.com, jbaker@raytheonvtc.com, aporter@raytheonvtc.com

INTRODUCTION

Interoperability between HLA federations is widely sought after. This is because interoperability potentially extracts the greatest value-added functionality for any given investment that was made in building a federation. In particular, training federations are developed for a specific purpose and set of objectives that may not be perfectly suited for future requirements or potential growth. Providing a means in which to interoperate an existing federation with other operational federations is likely more cost effective than rearchitecting the legacy federation. For example, the U.S. Navy wanted to enable interoperability between two of their F/A-18 Hornet flight simulator federations, the legacy F/A-18 E/F Tactical Operational Flight Training (TOFT) devices and a newer F/A-18 C/D Distributed Mission Trainer (DMT) suite of devices. Interoperability between these federations has tremendous value to the U.S. Navy, as it enables mixed section and mixed division exercises and allows training with more virtual aircraft in a scenario than otherwise possible. These additional training capabilities open the door for more realistic training for operations that may take place in actual combat.

Federation interoperability, however, is faced with a myriad of technological challenges. One such challenge is that the two federations may use different RTI implementations and/or RTI versions.

In some cases, it may be feasible to upgrade the RTI for one of these federations to match the second's RTI. This could involve code changes to the federates in the modified federation or it may be possible to interoperate by exploiting the link-compatibility of many RTIs (SISO-STD-004-2004 and SISO-STD-004.1-2004).

Unfortunately, upgrading the RTI of one of the federations likely requires extensive development and/or re-testing costs. In addition, there can be lost training time if the federations are for training purposes. This reality places the burden of supporting multiple RTIs onto the federation interoperability developer. This is the theme of this paper.

Using a federation bridge to enable interoperability between two federations that each use different RTIs is an idea first attributable to Braudaway and Little (1997). However, Bréholée and Siron (2003) are the first to attempt such a configuration and mention some of the problems encountered. In their case, they were able to circumvent the problems by making source code changes to one of the RTIs – which was under their control. Unfortunately, having control of the source of an RTI is a luxury that is not commonplace. In contrast, this paper presents and solves the issue more generally; the solutions presented do not require the source code and build environment of the RTI.

This paper is structured as follows: the next section, 'Challenges,' discusses in greater detail the problems encountered when creating a software process that links two different RTIs. The two subsequent sections, 'Proxy Library' and 'Run-time Library Modification,' discuss two techniques that, when used together, solve these problems. The section following, 'Operating System Specific Details,' describes some of the subtleties with applying the techniques in different operating system contexts. Subsequently, the 'Experiences' section details the authors' experience using these techniques, the 'Applicability' section clarifies when the solutions are applicable, and the 'Alternative Approaches' section includes a comparison with other approaches. Finally, the 'Conclusions' section contains closing remarks.

CHALLENGES

The fundamental challenge when constructing a federation bridge between two federations that use different RTIs is enabling the federation bridge to simultaneously use multiple RTIs. An initial architecture might have a dependency graph for its libraries similar to that in Figure 1.

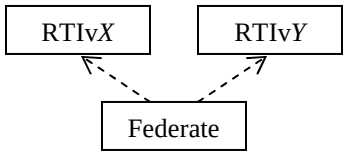

Figure 1. Straightforward but problematic library dependency graph linking multiple RTIs.

However, there are three problems with the above architecture, the first two of which are identified by Bréholée and Siron (2003). First, the code of the bridge federate is ambiguous with respect to which RTI it uses. RTI implementations typically use the same set of header files to declare their interface and enable link-compatibility. This also means they typically use the same set of programmatic symbols, e.g., `RTI::RTIambassador`. This is especially true of different versions of the RTI from the same vendor.

In a bridge federate that uses multiple RTIs, it is unclear to the compiler which symbol would be for which RTI. For example, in the code

```
RTI::RTIambassador rti_vX_Amb;
RTI::RTIambassador rti_vY_Amb;
```

the symbol `RTI::RTIambassador`, used by both RTIs, offers no hints as to which RTI it refers. Assuming the federate compiles, the result is a federate that will only use one RTI for all of its HLA-related communication. This is problematic as it should use both RTIs.

The second problem with the scenario in Figure 1 is that only one RTI's Local RTI Component (LRC) is loaded when the federate is executed. The operating system loads only the first RTI library with a given name that it finds. Since both RTIs' libraries have the same name (particularly if they are different versions from the same vendor) only the first library will be loaded. As a result, only the RTI that is loaded first will be used.

The third problem emerges from the fact that multiple RTIs share the same environment of the federate. Many RTIs load their configuration options from an RTI Initialization Data (RID) file. The location of this file is specified in the `RTI_RID_FILE` environment variable. The RID file stores many RTI-related settings including which network interface the RTI should use. If a federate is going to use more than one RTI, it is highly likely that it will require different RID file settings for each. However, despite having more than one RTI linked into a process, there is only one `RTI_RID_FILE` environment variable. This causes the same set of RTI configuration options to be loaded for both RTIs.

PROXY LIBRARY

Resolving the first problem – compile-time symbol ambiguity – can be accomplished by introducing a software proxy to disambiguate and differentiate between different RTI implementations in the bridge federate's source code. This proxy is a thin wrapper library around the RTI with an interface that uses different symbols. The bridge federate's code is changed to use the proxy in place of one of its RTIs and is linked to the proxy library instead of that RTI. The code given in the previous section would change to:

```
RTI::RTIambassador rti_vX_Amb;
RTIProxy::RTIambassador rti_vY_Amb;
```

The resultant dependency graph would be as in Figure 2.

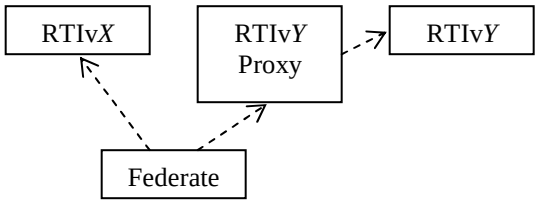

Figure 2. Dependency graph with a proxy library.

This approach is an example of the proxy or adapter design pattern as given by Gamma, Helm, Johnson, and Vlissides (1995). The implementation of each method in the proxy essentially does four things:

1. Converts the input arguments into their RTI equivalents.
2. Calls the equivalent RTI method with the converted arguments.
3. Converts the return value from the RTI method call into its proxy equivalent and returns it, if applicable.
4. Catches any RTI exceptions thrown and throws a proxy equivalent exception.

The implementation of the proxy's `FederateAmbassador` is similar to the above steps, though in the opposite direction (i.e., converting from RTI arguments to their proxy equivalents and calling the user-supplied `FederateAmbassador` instead of the `RTIambassador`).

RUN-TIME LIBRARY MODIFICATION

Implementing and using a proxy library as presented in the previous section will solve the compile-time symbol ambiguity problem. However, alone it is insufficient to solve the remaining two problems. At runtime, one of the RTIs will not be loaded, causing one of the scenarios depicted in Figure 3.

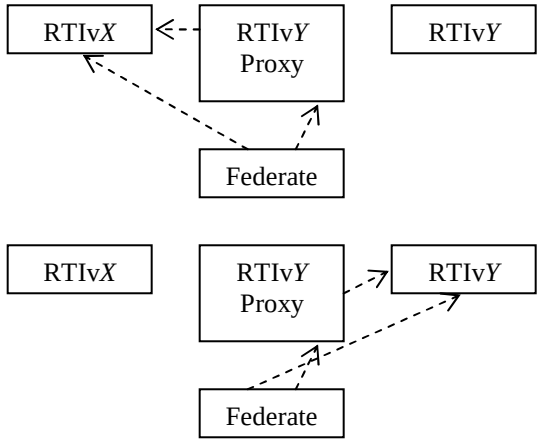

Figure 3. Run-time dependency graphs illustrating the insufficiency of a proxy library to resolve the RTI loading problem.

In order for the operating system to load both RTI libraries, they also need to be disambiguated. One way to accomplish this is to first change the names of the libraries of the RTI and all of its dependencies and then alter those libraries so that they reference their new names. This is best performed in a private copy of the `bin` and/or `lib` directories of the RTI installation that is wrapped by the proxy library. This ensures that the technique does not interfere with other federates that may use the original installation. Note that this change is local to the process that uses multiple RTIs. This disambiguation technique is not required for any other federate in either federation.

Some notes on this process:

- Like most binary files, these libraries will not function properly if bytes are added or removed. Therefore, modifications should only change existing bytes. This constrains the choice of alternate library names. On Windows systems, the authors recommend a naming scheme substituting the first character of each library's name with an underscore character (e.g., "libRTI-NG" becomes "_ibRTI-NG"). On Linux and UNIX systems, the authors suggest changing the first character after the standard library prefix "lib" (e.g., "libRTI-NG" becomes "lib_TI-NG").
- It is important to only modify the portion of the library that lists its dependencies; however in the authors' experience, a global search-and-replace for the entire file name is safe since the library names do not show up in the libraries otherwise.
- The directory with the modified libraries needs to be in the appropriate `PATH` environment variable(s) of the environment the federate is executed from.

The third problem, loading multiple RID files, is solved by performing one additional modification on the RTI libraries: replacing `RTI_RID_FILE` with an alternate name of the same length. The authors recommend using the same re-naming scheme that was used for the libraries, replacing the first character with an underscore. Thus, `RTI_RID_FILE` becomes `_TI_RID_FILE`. This enables the modified RTI to use a different environment variable to locate its particular RID file.

OPERATING SYSTEM SPECIFIC DETAILS

Until this point in the paper, the ideas and techniques have not been operating system-specific. However, the realization of a process simultaneously using multiple RTIs requires some steps that differ depending on the target operating system. The disparity is due to the difference between the file formats the operating systems use for shared libraries. Windows dynamically-linked libraries (DLLs) are structured differently than dynamic shared objects (DSOs) on systems using the Executable and Linker Format (ELF), such as Linux.

Windows

The final – and critical – step to enabling a process to use two RTIs on a Windows system is to use the run-time library modification technique one additional time—to alter the proxy's run-time library so that it references the modified RTI libraries. This gives the desired configuration as shown in Figure 4.

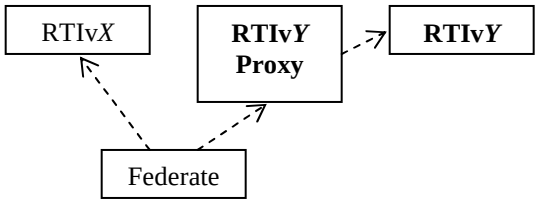

Figure 4. Run-time library dependency graph on a Windows system. Boldfaced items have been modified to alter their dependencies per the above technique.

The effectiveness of this approach on Windows is due to the file format of DLLs. A DLL contains a section that lists its own dependent DLLs. This section also lists what functions each DLL invokes. Performing the replacement in this section of the DLL enables it to use alternate DLLs when making external function calls.

Linux (and Other Systems that Use ELF)

On Linux and other systems using ELF, the problem is more complicated. In contrast to the way Windows DLLs are loaded, distinguishing which library the symbol comes from is less relevant when symbols are loaded from ELF-based libraries. Without intervention, the first library from which a symbol is loaded will be the only library from which it is loaded. This is true even if the libraries have been renamed using the run-time library modification technique.

To avoid this problem, the RTI has to be loaded into a "private" symbol space. This is accomplished by use of the `dlopen` system call, which loads libraries on demand at runtime. Bitwise or-ing the values `RTLD_LOCAL` and `RTLD_DEEPBIND` to the `mode` argument to `dlopen` enables the loading of the library into a private space. Symbols from a library loaded using this mechanism can be referenced via the `dlsym` system call.

The advantage of using these two system calls is twofold. First, libraries loaded and referenced using these functions do not create a dependency for the caller. Such a dependency would create another instance of the problem being solved with this approach. The second advantage is that the library loaded using these calls can have its references resolved in a private space. This prevents any collisions with other libraries.

The disadvantage is that this approach requires additional complexity for its implementation to be realized. The proxy library must be split into two separate libraries. The first library converts the symbols of the RTI so they become accessible via `dlsym`. The second library provides the RTI-like interface to the bridge federate and loads the first via `dlopen`. The authors refer to the first library as the "proxy delegate library" and the second as simply the "proxy library." The resulting architecture is depicted in Figure 5.

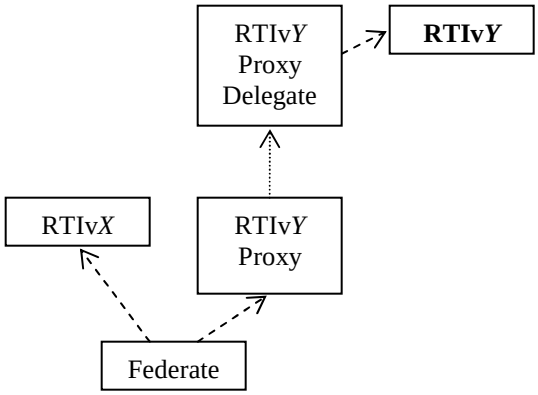

Figure 5. Run-time library dependency graph on a system using ELF. Boldfaced items have been modified to alter their dependencies. The dotted line illustrates a library loaded via the `dlopen` system call.

EXPERIENCES

This section chronicles the authors' experiences using the proxy library and run-time library modification techniques to date.

F/A-18 Federation Bridge

The development of the two presented techniques first occurred during the preliminary phases of the federation bridge project for the U.S. Navy. As mentioned in the introduction, the U.S. Navy desired the additional training capability that could be realized via a federation bridge between two different F/A-18 flight simulator suites at NAS Lemoore, CA. The first simulator suite used the DMSO RTI 1.3 NGv4 to provide networking capability amongst the four F/A-18 E/F flight simulator trainers in the suite. The second suite consisted of four NASMP 1.2.4 compliant F/A-18 C/D DMT trainers. These used the Raytheon VTC NG-Pro v2.0.4 to facilitate network communication. There were several challenges encountered during the design of this federation bridge. One challenge addressed the need to simultaneously support both the DMSO RTI 1.3

NGv4 and the Raytheon VTC NG-Pro v2.0.4. The latter RTI is a derivative of the former, albeit several versions later. Both RTIs also have a dependency on the ADAPTIVE Communication Environment (ACE) (Schmidt, 2007) though each depends on a different version.

The proxy library and run-time library modification techniques were applied to resolve this simultaneous RTI support need. To mitigate implementation risks, they were first proven using a prototype to ensure they solved the problem of a single process using multiple RTIs. A proxy library was then created to wrap the DMSO RTI 1.3 NG v4. One of the ways the proxy library was tested was by using the RTI's User Test Suite (UTS), a battery of 275 tests used to verify the RTI is working properly. The UTS was modified to use the proxy library instead of the RTI directly. Tests were then executed to ensure that the behavior of the proxy library was equivalent to the behavior of the RTI itself.

The processing overhead of the proxy library needed to be ascertained. A simple program was created to compare using the RTI directly versus using the RTI via a proxy library. This program was designed to provide a reasonable but worst-case situation for the overhead incurred while using the proxy library to perform an update. An "update" is defined as the complete process of instantiating an attribute handle value pair set, populating it with attributes, sending it, and destroying the attribute handle value pair set. The program performed the following algorithm for both the RTI and the proxy library:

- Capture the system time, t_1
- Do the following 1,000,000 times:
  - Create an attribute handle value pair set
  - Put 10 attributes in the attribute handle value pair set, each with a payload of 64 bytes
  - Call `updateAttributeValues` using the attribute handle value pair set
  - Delete the attribute handle value pair set
- Capture the system time, t_2

Notice that each loop of the algorithm makes 13 RTI API calls (or their proxy library equivalents). The authors found that the delta between t_1 and t_2 was 50 seconds for the RTI and 60 seconds for the proxy library. This results in the RTI being able to process an update in 50 microseconds, and the proxy library being able to process an update in 60 microseconds. The overhead incurred for using a proxy library is 10 microseconds per update, which is less than 1 microsecond of additional processing time per proxy library RTI API call. This experiment was performed on an IBM Thinkpad laptop with a 2 GHz Intel Centrino processor and 1 GB of RAM. The operating system was Windows XP, and the proxy library architecture was as depicted in Figure 4.

The proxy library's performance could be improved if it utilized a caching strategy for the attribute handle value pair set. This would eliminate the additional memory allocation/de-allocation per iteration of the loop.

In short, applying these techniques to the F/A-18 federation bridge was enormously successful. There were no issues attributable to the use of the proxy library or the run-time library modification of the RTI. Furthermore, there were no detrimental side-effects with using these approaches. It was originally believed that there might be the risk that the run-time library modification for one RTI LRC would cause it to become not operational. However, the federation bridge was able to participate in both federations seamlessly. Pilots in trainers on one federation were able to perform a large set of common procedures with pilots on trainers in the other federation and vice versa. This resulted in greatly increased capability for training at a very low cost for the U.S. Navy. As a final note, the U.S. Navy specifically required that the federation bridge be completely removed with zero footprint on either federation within 10 minutes. Because these solutions were external to the federations, removing the federation bridge was a matter of shutting down the federation bridge's software. This took less than one minute.

NASMP Interoperability Interfaces

These techniques have been used on several other projects. Many of these projects are related to building and customizing federates to participate in Navy Aviation Simulator Master Plan (NASMP) federations. "NASMP-compliant" federates adhere to a specific FOM and federation agreements document (FAD). Many current and future training systems achieve NASMP-compliance via a NASMP Interoperability Interface (NII), a NASMP-compliant federate that communicates NASMP federation data to and from the training device.

One challenge in creating an NII for a trainer is that the latest version (1.4.*x*) of the NASMP FAD requires use of data-distribution management (DDM) in a specific way. To facilitate this, the engineers involved used the proxy library technique. Since all RTI API

calls are made through a proxy library, it becomes a convenient location in the system to implement the required DDM functionality. This effectively leverages the adapter design pattern (Gamma, Helm, Johnson, & Vlissides, 1995). The proxy implementation for this purpose is straightforward: for each non-DDM RTI API call, determine the appropriate region to use and invoke the DDM analog of that API call using the determined region.

One benefit of this approach is that the proxy library is also reusable among NIIs or other NASMP-compliant federates. Furthermore, extensive design and development work is not necessary for each federate that uses this approach to participate in a NASMP 1.4.*x* federation using this proxy library. As a corollary, if a federate is ever upgraded to a later version, the changes to re-add the proxy library are smaller than the re-engineering that would be required to re-fit the federate with the new DDM solution. Conversely, if the federate requires compliance to a newer version of the NASMP FAD, all of the changes to support the DDM portion of this change are confined to the proxy library. Parnas (1972) points out other advantages of decoupling in this manner.

This technique has been used in other contexts as well. For instance, many trainers are unable to internally support the number of entities that may exist on a NASMP federation. In these cases, NIIs make use of a proxy library to filter entities based on proximity to the simulated trainer entity, among other factors. Implementing filtering functionality closely to the RTI also makes sense for performance reasons.

In the above scenarios, the run-time library modification technique is not used because it is only necessary when using multiple RTIs. However, trainers often use different mechanisms to communicate between their different subsystems. One such case used HLA-based communication via the STOW RTI. In addition to using the proxy library technique to implement DDM and filtering capabilities, this particular NII had to simultaneously use two different RTIs from two different vendors (the STOW RTI to communicate to the trainer and the Raytheon VTC NG Pro RTI to participate in the NASMP federation). The run-time library modification technique was used to support this.

The two techniques in this paper have been used together on other smaller projects and in other situations. These situations are discussed more generally in the next section.

APPLICABILITY

Table 1 captures and summarizes the different situations in which a proxy library and run-time library modification may be required. Each column merits an explanation: the 'Case' column serves to label the rows of the table so that they may be referenced.

The second column, 'Multiple `RTIambassador` Instances,' and the third column, 'Different RTI Versions,' determine how many `RTIambassador`s are required in a federate implementation and to which RTI each belongs. It is possible to have multiple `RTIambassador` instances from one RTI. It is also possible to have multiple `RTIambassador` instances, each from a different RTI. The second column denotes whether there is simply more than one `RTIambassador` in the federate, while the third column determines if all of the `RTIambassador`s are from the same RTI or are from different RTIs.

The 'RID File Conflict' column denotes whether each RTI implementation requires its own `RTI_RID_FILE` environment variable setting. Cases where the multiple RID files are not necessary when multiple `RTIambassador` instances are present from different RTIs can occur when one or more of the RTIs do not make use of the `RTI_RID_FILE` environmental variable.

The 'Symbolic Ambiguity' column addresses whether the different RTI versions use the same symbols in their RTI APIs. A "yes" in this column indicates that the federate code will be unable to determine which RTI it is using when it makes calls to the RTI. A "no" indicates that this is not a problem.

The 'Library Name Collision' column denotes whether the different RTI versions use different library names that are loaded at runtime. A "yes" in this column indicates that the two different versions of the RTI use the same name for their library or for a dependency library. A "no" specifies that different library names are used.

The 'Is Proxy Library Required?' and 'Is Run-time Library Modification Required?' columns answer whether one or both techniques presented in this paper are required to support the federate based on the entries in the previous columns.

Next, each case is considered. Most federates fit the profiles of case 1 in that they only require one

Table 1. Summary of Applicability

Case	Multiple `RTIambassador` Instances	Different RTI Versions	RID File Conflict	Symbolic Ambiguity	Library Name Collision	Is Proxy Library Required?	Is Run-time Library Modification Required?
1	No	N/A	N/A	N/A	N/A	No	No
2	Yes	No	No	No	No	No	No
3	Yes	No	Yes	Yes	Yes	Yes	Yes
4	Yes	Yes	No	No	No	No	No
5	Yes	Yes	No	No	Yes	No	Yes[1]
6	Yes	Yes	No	Yes	No	Yes	Depends[2]
7	Yes	Yes	No	Yes	Yes	Yes	Yes
8	Yes	Yes	Yes	No	No	No	Yes[3]
9	Yes	Yes	Yes	No	Yes	Yes	Yes[1]
10	Yes	Yes	Yes	Yes	No	Yes	Yes[4]
11	Yes	Yes	Yes	Yes	Yes	Yes	Yes

1. On systems that use ELF, a proxy delegate library is not necessary due to the lack of symbol ambiguity.
2. No for Windows systems, but required for Linux and other systems that use ELF
3. Only to modify the RID file environment variable
4. On Windows systems, only to modify the RID file environment variable. On systems that use ELF, the entire library disambiguation process is required.

`RTIambassador` instance, through which they communicate with other federates. As this case involves only one `RTIambassador` instance, the problems associated with having multiple RTIs in a single software process are not applicable.

Federates in case 2 are uncommon but nonetheless possible. The ModelEngine (Geyer, Glintz, Thomas, & Luebke, 2004) is an example of one such federate. This federate enables multiple models (such as mission computers, radars, etc.) to be loaded simultaneously. Each model has an associated `RTIambassador`, all of which are used in the same federation. Neither a proxy library nor run-time library modification is required for this case, though a proxy library could be used to add additional functionality as described in the previous section.

Case 3 encompasses the situation in which a federation bridge spans two federations. These federations use the same version of the RTI but are using two different RTI executives (perhaps on two different networks) or are running in connectionless mode with different settings for each federation. Support for this situation requires different RID files for each `RTIambassador` instance.

Unfortunately, this seemingly small problem requires both a proxy library and a run-time library modification. It is worth mentioning that copying RID files around in a controlled way while the federate is initializing or making calls to `setenv` may be a viable alternative. However, this may not work for all RTIs due to the caching of environment variables by their underlying libraries or the way the operating system performs library loading.

Case 4 includes instances when two RTIs are different enough to avoid the problems in this paper. Neither a proxy library nor run-time library modification is required if the two RTIs have no RID file conflict, no symbolic ambiguity, and no library name collision. The authors are not aware of a pair of COTS RTIs existing today that fall into these categories, though cases when one of the RTIs is a "home-grown" RTI implementation very well might.

Case 5 covers situations when two RTIs share the same library names but do not suffer from using the same API. The RTIs also have different mechanisms for loading RID files or otherwise configuring the RTI. In this case, run-time library modification is necessary to change the names of one of the RTI's library files (and possibly its dependencies as well). However, on systems that use ELF, a proxy library delegate is not necessary since the RTIs have no symbols in common.

Situations involving two RTIs with different mechanisms for RID configuration, different library names, but that use the same symbols for their APIs

fall into case 6. If the operating system is Windows, the problem can be resolved by using a proxy library to merely wrap the API. This is the only case where a proxy library is necessary but run-time library modification is not. On operating systems using ELF, the problem requires both techniques due to the way symbols are loaded from the libraries at run-time.

Case 7 is basically the same as case 6 except that the additional issue of a library name collision is present. In addition to a proxy library, this issue requires run-time library modification to overcome.

If a bridge federate uses two RTIs that differ in both API and choice of library names, but still conflict in their use of the `RTI_RID_FILE` environment variable in order to load a RID file (case 8), then one of the RTIs' run-time libraries can be modified to change that symbol.

Case 9, like case 5, covers instances where the RTIs share library names but do not have a symbolic ambiguity problem. Unlike case 5, the RTIs in case 9 both use the same `RTI_RID_FILE` environment variable. The approach is the same as in those previous cases, except that as a final step in modifying the run-time library of one of the RTIs, the name of the environment variable used to locate the RID file is also changed. The bridge federate in the previous section that uses the STOW RTI and the Raytheon VTC RTI-NG Pro is a good example of this case, as the STOW RTI uses slightly modified header files that replace the `RTI` enclosing class with the `rti13` namespace.

The next case (10) encompasses situations where the bridge federate uses different RTI versions that use the same API and RID file loading mechanism, but use different library names and thus have no collision in that regard. On a Windows system, the issue can be resolved by wrapping the RTI's API and making the change to the library's `RTI_RID_FILE` environment variable via a binary modification. On systems using ELF, the entire library disambiguation binary modification process is required.

Case 11, best illustrated by the F/A-18 Federation Bridge, makes use of both a proxy library and run-time library modification in order to span two federations on two networks using two different RTIs.

ALTERNATIVE APPROACHES

A seemingly viable alternative to the solutions presented in this paper is to avoid the issue altogether and change one or both federations so that they use the same RTI. As mentioned previously, this approach is unlikely to be feasible due to cost in time and money. This is especially true in the military training systems community, where such a change would require full regression testing and revalidation of the training device(s). Furthermore, this approach is merely a point solution; it is possible that other federations may need to bridge later as well. This would require that the whole testing and revalidation process be repeated. Granowetter (2003) points out other challenges, both technical and otherwise, with this approach.

In some cases, it may be acceptable to modify the source code and build processes of the RTI itself to overcome this obstacle. This is the approach Bréholée and Siron (2003) used. They controlled the source code of one of the RTIs in use. For example, the STOW RTI is an open-source HLA implementation. It would be straightforward to change its relevant symbols and library names and build a private copy for use with a federation bridge. One caveat of this approach in general is that the library dependencies of the RTIs may also need to be addressed. The DMSO RTI and Raytheon VTC NG Pro RTIs, for instance, are built on top of ACE (Schmidt, 2007). In many cases they may use different and incompatible versions of it. This approach would prevent RTI link-compatibility; however the techniques put forth in this paper also have that same drawback.

Another alternative is to split the federation bridge into multiple processes that communicate via some IPC mechanism. While this approach would work, there are some significant disadvantages. First, designing, implementing, and testing the mechanism by which the processes communicate would be significantly more expensive compared to the effort required by the techniques presented in this paper. A second disadvantage is the performance penalty associated with sending and receiving data between the processes. This penalty varies depending on the mechanism on which the IPC is based and the way it is used. In any case, the overhead falls into two categories: the additional processing required to package and transmit data (including the overhead of copying data if applicable) and the delay inherent to the transmission itself. In contrast, the performance penalty of using a proxy library is typically an additional method call per RTI API call on Windows and typically two additional method calls per RTI API call on Linux and other operating systems using ELF.

CONCLUSION

The two techniques discussed in this paper, wrapping the RTI using a proxy library and run-time library modification, provide a simple, straightforward solution in linking multiple disparate RTIs into a single software process. The use of these two techniques is a low-cost alternative that due to its simplicity of concept and code lends itself well to lifecycle support. The techniques can be broadly applied, though there are subtleties and caveats with each one on Windows and Linux/UNIX-based operating systems. These solutions are not tied to a single set of HLA products; rather, the same techniques can be used for any RTIs. This approach has the potential of being applied in many more federation bridging engagements with very low risk incurred.

REFERENCES

Braudaway, W., & Little, R. (1997). The High Level Architecture's Bridge Federate. In *Proceedings from the Fall 1997 Simulation Interoperability Workshop*, September 1997, 97F-SIW-078.

Bréholée, B., & Siron, P. (2003). Design and Implementation of a HLA Inter-federation Bridge. In *Proceedings of the 2003 Euro Simulation Interoperability Workshop*, June 2003, 03E-SIW-054.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design *Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Geyer, D., Glintz, S., Thomas, N., & Luebke, W. (2004). ModelEngine: An Application Framework for Integrating Simulation Models into HLA Federations. In *Proceedings from Spring 2004 Simulation Interoperability Workshop*, April 2004, 04S-SIW-107.

Granowetter, L. (2003). RTI Interoperability Issues – API Standards, Wire Standards, and RTI Bridges. In *Proceedings of the 2003 Euro Simulation Interoperability Workshop*, June 2003, 03E-SIW-077.

Parnas, D. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM, 15*(12), 1053-1058.

Schmidt, D. (2007). *The ADAPTIVE Communication Environment (ACE)*. Retrieved June 3, 2007 from http://www.cs.wustl.edu/~schmidt/ACE.html

SISO (2004). *Dynamic Link Compatible HLA API Standard for the HLA Interface Specification Version 1.3*. SISO-STD-004-2004, http://www.sisostds.org.

SISO (2004). *Dynamic Link Compatible HLA API Standard for the HLA Interface Specification (IEEE 1516.1 Version)*, SISO-STD-004.1-2004, http://www.sisostds.org.