Developing Fault Tolerant Federations using HLA Evolved

Björn Möller
Mikael Karlsson
Björn Löfstrand
Pitch Technologies
Nygatan 35
SE-582 19 Linköping, Sweden
+46 13 13 45 45
bjorn.moller@pitch.se
mikael.karlsson@pitch.se
bjorn.lofstrand@pitch.se

Keywords:
HLA Evolved, RTI, Fault Tolerance, Design Patterns, Federation Management, FEDEP

ABSTRACT: *As HLA federations are deployed in more and more distributed environments there is an increasing need to be able to operate in a less than perfect world. A number of extensions that adds fault tolerance support to HLA have been suggested and accepted as a part of HLA Evolved. Two types of faults are introduced: "federate lost" as seen from the federation and "connection lost" as seen from the federate. Some of the potential and limitations of this approach to fault tolerance are described in this paper.*

To handle fault tolerance in a federation it must be addressed early in the Federation Development Process (FEDEP). During federation agreement and throughout the federation and federate design and implementation the level of fault tolerance must also be related to the purpose and goal of the federation. For example, it is necessary to understand what constitutes a valid federation when federates are lost, what the procedures are to determine this and how to recover. Fault tolerant requirements may also vary between training and analysis federations.

A number of design patterns for fault tolerance in federations are presented in the paper, for example, the required federation subset, the optional federation, the fault monitoring federate, the reoccurring federate, the spontaneous federation the fail-over federate and the fail-over RTI.

For federate a number of design patterns for different operations can be implemented to support fault tolerance. Some of these are fault tolerant updates, regular reconnection attempts, fault tolerant save and failure monitoring.

The resynchronization of a federation is an important issue. Some aspects of rejoin are discussed. The approach for resynchronization needs to consider both technical and scenario management aspects to be able to resynchronize at a relevant and convenient time.

1. Introduction

1.1 Faults and HLA

Any successful new technology will experience a transition period when it goes from the well controlled environment of the R&D laboratory to real world deployment and usage [1]. This means both a gradual change in the environment where the technology will be used and in the requirements it will have to meet.

The HLA standard for simulation interoperability is yet another example of this. Early federations included a handful of systems in a resourceful laboratory with experienced staff. We now see deployment underway of HLA federations encompassing hundreds of systems across continents and using systems with real life limitations when it comes to reliability, bandwidth, staff, etc.

1.2 Faults in the HLA federation

Faults can occur at several levels in an HLA federation. Once the fault has been detected, there are several different approaches for recovery. Depending on type of fault and on what level it occurred, the federation can either try to recover or continue with degraded performance. Sometimes the fault will cause the federation to completely stop operating. Degradation or loss of service may be signaled to a higher level directly or indirectly.

The following picture shows where faults can occur in an HLA federation:

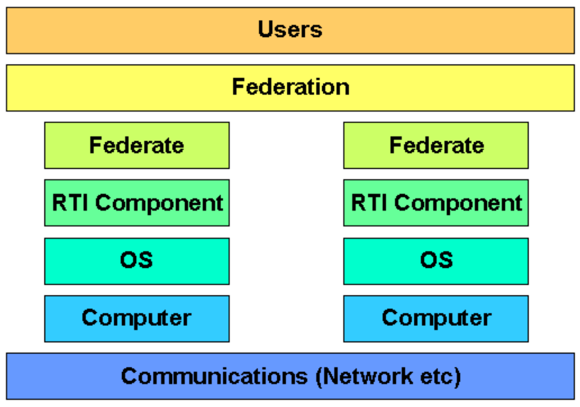

Figure 1: Levels where faults can occur

Level 1: Communications (Network etc). Typical faults occur when a cable is disconnected or a path between two hosts is no longer available. The fault may be resolved at this level by rerouting the packets

Level 2: Computer hardware. Typical faults in hardware may be related to power supplies and hard drives but other components may also fail. Some of this can be compensated by redundancy, for example by doubling the power supply or using RAID storage.

Level 3: Operating system. Typical faults are system freeze or malfunctioning components such as drivers or processes.

Level 4: RTI components. RTI components may crash or become corrupt. The latter may be the more difficult problem to troubleshoot.

Level 5: Federate. Federates may crash or degrade. One particular problem is when a federate hangs or gets stuck in debugger. A federate producing inaccurate data is not considered a fault in this context.

Level 6: Federation. This level may experience faults when each federate works technically well but does not follow the federation agreement. The federates may not format or interpret data as agreed or produce data at the wrong rate. This is a well-known issue for anybody who has done any real-life federation integration.

Level 7: Users. Users can trigger any fault on lower levels. They can also unexpectedly do technically clean shutdowns of federates at the wrong time. On the other hand, it is sometimes possible to compensate for faults at lower level by handling simulation tasks manually.

1.3 Scope of the fault tolerance

HLA is a standard that mainly specifies the interface and interplay between the RTI and the federate, it is not an implementation. Fault tolerance in the HLA standard only addresses this interface. A federation manager that wants to ensure high availability of his federation may want to address all of the above levels.

2. Fault Tolerance in HLA Evolved

The current version of HLA is formally called IEEE 1516-2000 [2]. HLA Evolved is the working name for the next version of HLA IEEE 1516 that is expected to be completed in 2005 or early 2006. During the first comment round a set of functions that handle fault tolerance have been suggested and accepted. This is an overview of these additions. A number of alternate approaches have been examined and rejected [3].

Some engineering support also exists for the current approach. One of the examples is the federate RTIperf [4] for HLA IEEE 1516-2000 (source code publicly available) that makes educated guesses when general faults occur and applies some of these approaches. The problem is that there is no obvious way to understand this from the source code and there is no guarantee that this code would work with another RTI implementation than the one used in this experiment (pRTI 1516).

2.1 Design criteria

The most important design criteria used for the fault tolerance additions are:

Preserve classic HLA Compliance. We don't want to redefine HLA as it is known today.

Well-defined federations. This concept should be preserved. A federate is either joined or not. This is necessary for federation-wide coordinated operations like time management.

Minimal impact on federates. Especially for federates that do not want to implement fault tolerance the impact of the additions should be minimal.

Modest implementation effort. It should be possible to implement it in RTIs and federates/federations with minimal effort.

2.2 Life cycle of federates experiencing faults

A federate may go through the following life cycle with respect to a fault:

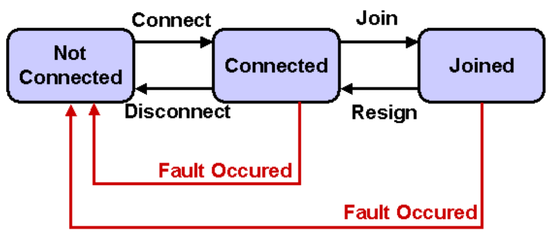

Figure 2: Federate life cycle from a fault perspective

1. Not (yet) connected to the RTI. Initially the candidate federate will not be connected to the RTI. It will now attempt to connect using the Connect call which is a new function introduced in HLA Evolved. Faults during this phase are not handled through the HLA Fault tolerance although the Connect call will clearly indicate whether it completed successfully or not.

2. Connected. If the Connect is successful the system is now connected to the RTI and can Create a FederationExecution and Join. Faults are handled through the HLA fault tolerance.

3. Joined. If the Join is successful the federate can now participate in the federation. Faults will still be handled through the HLA fault tolerance.

4. Connected (after Resign). After resigning the federate is still connected and faults will still be handled through the HLA fault tolerance.

5. Disconnected (After Disconnect). After disconnecting from the RTI the federate is back to step 1. Disconnect is also a new call introduced in HLA Evolved.

6. Disconnected (After fault) If a fault occurs the federate is no longer considered a joined or connected federate by the RTI. The federate is now back to step 1.

For a description of the life cycle of a federation when a fault occurs, see section 6.

2.3 Fault definitions

In HLA Evolved a fault is defined as "*a problem that occurs in the federation or its environment that prevents the entire federation from interoperating in an HLA compliant manner*". There are two types of faults:

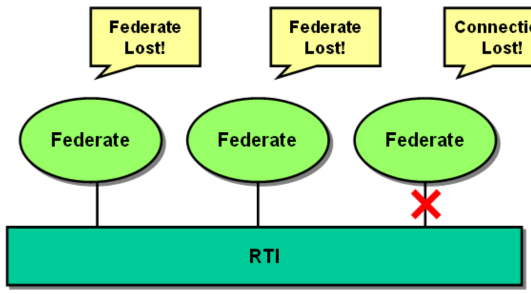

Figure 3: Types of faults

Federate Lost occurs when a federate has been unexpectedly lost from a federation as a result of a fault. This fault is signaled using the MOM interaction "*HLAreportFederateLost*" to any federate in the remaining federation that subscribes to this interaction. The MOM interaction also provides information about the last known-good timestamp to which the lost federate was granted. This value is provided on a best-effort basis and may not represent the highest time value to which the federate was actually granted.

Connection Lost occurs when a federation has been unexpectedly lost from a federate as a result of a fault. This fault is signaled through a callback from the RTI which in C++ and Java is implemented as a call to the FederateAmbassador. The federate then enters the *Not Connected* state. The federate now needs to do a Connect to the RTI again otherwise a *Not Connected* exception will be thrown when an RTI call is made. Note that this callback is done to a federate experiencing a fault. It is done on a best-effort basis and may or may not succeed since this call may be done by an LRC that may have lost contact with the rest of the federation or is trying to execute on a computer without power.

Why are two different mechanisms used for signaling? It is assumed that not every federate is interested in reports about other failing federates. Those who do would like to get it in a context of what other federates that are available to be able to assess the situation. This is why it was added to the MOM.

Losing the connection to the federation on the other hand is something that most federates may want to be aware of. A federate may choose not to be fault tolerant and ignore this callback (which would be the default behavior when using a C++ or Java nullFederateAmbassador). This would, however, result in exceptions in later RTI calls.

2.4 Actions taken by the RTI

When a federate has been lost the RTI is responsible for reporting this and then do a resign on behalf of the lost federate using the "*Automatic Resign Directive*". This directive is specified in the FOM and can also be accessed using support services. The MOM information and relevant advisories are updated as with a usual resign action. This also means that all federation-wide synchronized operations like time-management are recovered. If for example all federates are waiting for a specific federate to advance time and that federate has crashed this would otherwise have prevented the entire federation from advancing. The final result of the resign and recovery will be a reduced but HLA compliant federation.

A particularly interesting situation occurs when a federation is split by a fault into two groups of federates both of which are potentially HLA compliant. Only one of them may be considered HLA compliant by the RTI. All RTI implementations are required to define a mechanism for making this decision, which is usually not a problem for RTIs with a Central RTI Component. While it would have been powerful to form two new federations that can later on be re-synchronized the fundamental problems of this are intriguing.

2.5 Implications for RTI implementations

The descriptions above say nothing about how the fault tolerance should be implemented only how faults have to be signaled. It can be expected that different RTI implementations will have different performance when it comes to fault detection. Different RTI implementations may also want to do different trade-offs between performance goals such as throughput, latency, fault tolerance and bandwidth utilization. Two obvious candidates for detecting faults are the following:

Closed TCP socket. True multiprocessing operating systems will in many cases detect crashing processes and clean up after them. This means that they will explicitly close any open communications sockets. The system on the other end of the link will be notified. If any RTI component detects that a link to another federate was explicitly but unexpectedly closed it can be assumed that the remote federate crashed. When it comes to bandwidth and CPU utilization this detection method costs close to nothing. It is however limited to the detection of well-handled program crashes only.

Polling. RTI components may poll each other at regular intervals. If no answer is received within a specific time frame the other component may be considered lost. This type of detection may not be 100% accurate under varying load and latency conditions and also requires some bandwidth and processing resources. It does, on the other hand, detect a wide range of faults.

Other detection methods also exist. For a shared memory implementation for example a corrupt communication area may trigger a fault.

When verifying an RTI for fault tolerance it may be necessary to know how to trigger a fault that can be detected by the specific RTI implementation so that the behavior can be verified.

2.6 Scope of the Fault Tolerance additions

Are these fault tolerance additions the definite solution to all problems with faults in an HLA federation? The answer is clearly no.

While these additions gives us a well-defined way of signaling and handling faults there will still be cases where the faults are so large and frequent that a federation cannot execute in such a manner that the federation goals are achieved.

There are also faults that cannot be handled by these additions as can be seen from the following picture:

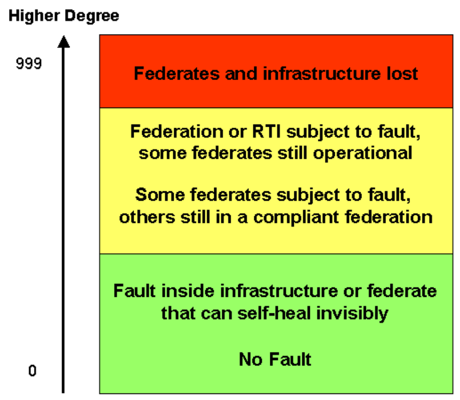
Higher Degree
999
0
Federates and infrastructure lost
Federation or RTI subject to fault, some federates still operational
Some federates subject to fault, others still in a compliant federation
Fault inside infrastructure or federate that can self-heal invisibly
No Fault

Figure 4: Different degrees of faults

In the bottom part (green) are faults that occur on a low level, for example network failures that can be automatically solved through rerouting. These faults are invisible to higher levels, for example current and previous HLA standards.

In the middle part (yellow) are faults where a subset of a federation or an ex-federate simulation is still operational. This area is addressed by the current additions.

In the upper part (red) are faults where the HLA infrastructure and federates are no longer operational. This may happen during a training exercise due to power loss. This will have to be handled by falling back to manual procedures.

This is a first broad approach the problems with faults. As we gain more experience from building fault tolerant federations the functionality can be expected to develop in future versions of HLA.

3. Applying fault tolerance to a federation

Fault tolerance at the HLA level needs to be designed into the federation to be successful. It needs to be aligned with the goal and budget of the federation. This additional feature will come at a cost but as always it will be substantially less costly if designed and added at an early stage.

3.1 Fault tolerance throughout FEDEP [5]

In the purpose and goals of the federation, the fault tolerance aspects are often implicit. Depending on the requirements, the federation can be designed to be more or less fault tolerant. It is however very important that the issue is raised and a well-informed decision is made early in the federation development process. Since fault tolerance can be a significant cost driver it should also be considered in the budget planning process.

During “Conceptual Model Development” step the conceptual analysis and the scenario should also address to what degree different model components are required or optional. This information is then used in the “Federation design” step where for example the design patterns described later in this paper can be used.

The federation agreement should specify at least

- What degree of degradation is acceptable to the federation.
- How and by whom this is evaluated and what actions to take at each level of degradation.
- What type of fault tolerance that should be implemented in each federate, including any interaction between federates.
- Standard design patterns to use for the fault handling.

During the “Test and integration” step procedures to invoke or simulate failures should be determined and implemented.

During the “Federation Execution” step faults some faults may be fully handled by the federation but it is likely that major faults may require manual intervention and in some cases dynamic adjustments to the scenario. All faults may need to be logged.

During the “Analysis and evaluation of results” step it is necessary to take into account any faults that occurred.

3.2 Analysis versus training

When evaluating what degree of fault tolerance that should be implemented in a federation the approach may differ between analysis and training federations:

Analytical federations may need a very detailed tracking of faults that occurred during the federation execution. A thorough analysis may be necessary do determine to what degree the output is useful. If the output is considered useless if any major degradation has occurred it may be of greater importance to track the faults than to compensate for them.

Training federations on the other hand may want to focus on the capability to provide an effective training experience even though systems may at times be at fault. It may be more important to provide capabilities to execute a degraded federation than to log the faults for future analysis.

4. Design Patterns for Fault Tolerant Federations

A number of design patterns have been developed both on the federation level and for federate implementations. They are described together with the problem and context when they can be used and a number of consequences that need to be evaluated for each case before applying it.

4.1 The required federation subset

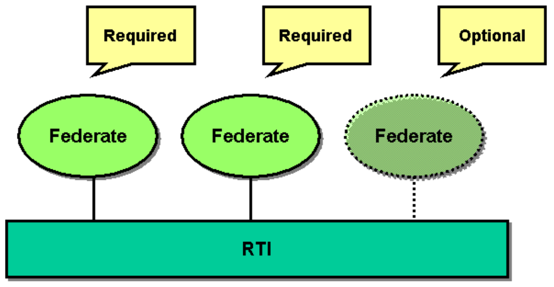

Figure 5: The required federation subset pattern

Problem and context: A federation cannot reach its goals if a required subset of the federates is not available.

Design pattern: The federation will not start running unless the required federates are joined. It will stop executing if any required federate disappears. This monitoring can be controlled manually or using the pattern “The fault monitoring federate” below.

Consequences: Need to handle the situation where the federation unexpectedly stops running.

4.2 The optional federation

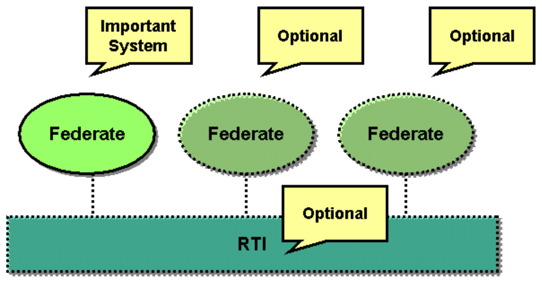

Figure 6: The optional federation pattern

Problem and context: An important simulation, for example a trainer, needs to keep executing, no matter if the federation is available or not. The stand-alone system may be more important than the federation.

Design pattern: The federate performs regular checks to decide whether a federation is available. If so it will join and interoperate. If not the federate will keep running but it will not make any RTI calls.

Consequences: Some computing resources used for the reconnection attempts. Need to consider resynchronization.

4.3 The reoccurring federate

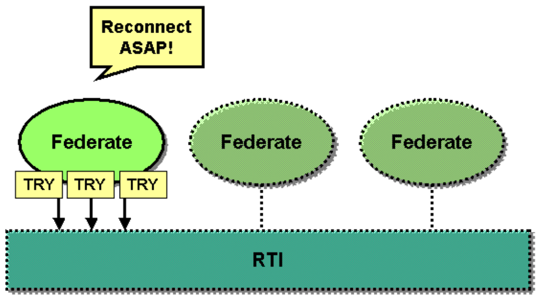

Figure 7: The reoccurring federate pattern

Problem and context: A federation is not always available but when it is it needs the contribution of a specific federate that is available to a larger extent. Alternatively a federate wants to participate in a particular federation. The federate may be difficult to start for example if it is located at another site.

Design pattern: The federate tries to join the federation at regular intervals.

Consequences: Some computing resources used for the reconnection attempts. A mechanism for preventing the federation from accepting the new member may be required.

4.4 The fault monitoring federate

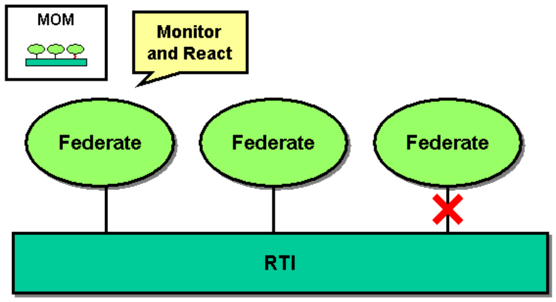

Figure 8: The fault monitoring federate pattern

Problem and context: Specific actions need to be taken whenever the members of a federation changes.

Design pattern: The monitoring federate subscribes to MOM data about joining and lost federates and take appropriate actions. This may include signaling to operators or to other federates to take specific actions.

Consequences: Other federates need to understand what to do and when. It is necessary to communicate to operators what actions that are done automatically and what needs to be handled manually.

4.5 The spontaneous federation.

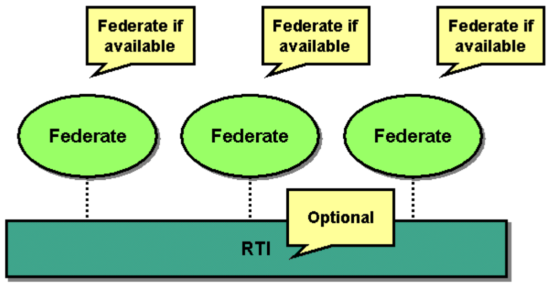

Figure 9: The spontaneous federation pattern

Problem and context: A federation is valuable even though no centralized federation management takes place or no formal federation agreement has been made except for sharing a FOM.

Design pattern: Whenever there are enough participants a federation will occur. Each federate is responsible for monitoring this from their own point of view. Lost federates may not be considered a major problem.

Consequences: Need to determine when a federation is wanted or not. Need to establish a mechanism for locating a common RTI.

4.6 The fail-over federate

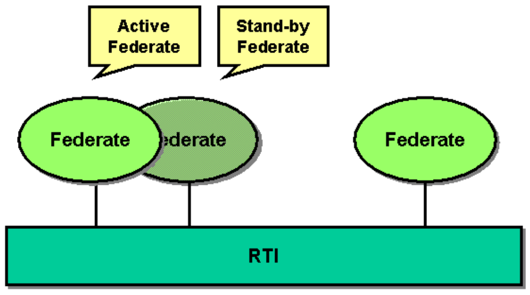

Figure 10: The fail-over federate pattern

Problem and context: The functionality of a specific model is essential to a federation. The environment has a high fault-rate or it is desirable to achieve very high availability of the federation.

Design pattern: A stand-by federate monitors the progress of another federate with overlapping functionality. When the other federate is lost the fail-over federate can take over. The fail-over federate may provide exactly the same model or a model with a different resolution.

Consequences: The exact time for the fail-over event may not be exact so there may be some catch-up time before the full functionality of the federation is restored. Some loss of model availability and accuracy during fail-over can be expected.

4.7 The fail-over RTI

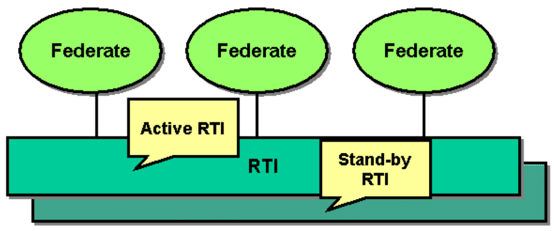

Figure 11: The fail-over RTI pattern

Problem and context: There is a high fault-rate in the environment or the RTI or it is desirable to achieve very high availability of the federation.

Design pattern: Federates have a prioritized list of RTIs. When one RTI disappears they connect to another one.

Consequences: Need to provide alternate RTIs. There is a risk that different federates may fail over to different RTIs for topological reasons.

5. Design Patterns for Fault Tolerant Federates

To make it possible to implement a fault tolerant federation according to the above it is also necessary to look at each federate and make necessary modifications. A number of patterns for the federate implementation are described below.

5.1 Signaling errors to the user

Problem and context: Many faults, for example hardware failures, can only be corrected manually outside of the federation. The users may however experience difficulties locating the exact reason for the fault.

Design pattern: Both the “Federate Lost” MOM interaction and the “Disconnected” callback provide human readable descriptions of the fault. These should be clearly advertised to the user to facilitate correction of the fault

5.2 Fault tolerant RTI calls (updates, etc)

Problem and context: It is not acceptable for the federate to crash when the connection to the RTI is lost. It may keep running anyway and potentially reconnect to the federation later.

Design pattern: Before sending updates the federate check if it is connected.

Consequences: It may be combined with the next pattern.

5.3 Regular reconnection attempts

Problem and context: A federate that has lost its connection to the federation needs to reconnect.

Design pattern: The federate tries to reconnect at regular intervals.

Consequences: Too frequent reconnection attempts may slow down the federate since the reconnection may take some time to time out.

5.4 Fault tolerant save

Problem and context: Some federates may crash or hang during save. This may cause the save to terminate with an exception.

Design pattern: The federate initiates a save. Some federates may be lost which will result in an exception. Save is retried over and over until it succeeds.

Consequences: Only the federates that survived the save attempts were actually saved. If a federate hangs it may need to be killed manually but after that the federation will save. If the initiating federate crashes no save will take place.

5.5 Failure monitoring

Problem and context: A federate need to be up to date about the status of a specific federate and monitor when it joins, resigns or crashes.

Design pattern: The monitoring federate subscribes to the MOM federate object class and monitors it for the other federate. It also subscribes to the FederateLost MOM interaction.

6. Further Implications for Fault Tolerant Federations

After recovering after a fault a federate will want to rejoin the federation. While it is technically possible to rejoin a federation at any time there are several other challenges associated with this particularly related to the scenario. Several of these problems are similar to the ones of “late joining federate” situation.

6.1 Life cycle of a fault in a federation

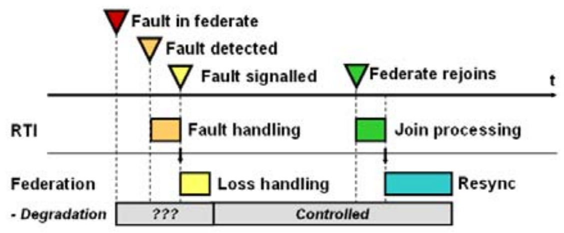

Figure 12: Federation fault handling and degradation

After the fault occurs the RTI may detect the fault and signal it to the federates. During the time this takes the degradation is not well controlled.

The federation can now start handling the loss, for example by removing objects, adjusting ownership or activating stand-by functionality. How the loss handling is done should be covered in the federation agreement. After this is completed the federation executes with a well controlled degradation.

We now assume that the fault is handled and the federate wants to rejoin. After the federate has completed the Join and the exchange of information could in principle start again.

In most real cases however, a resynchronization has to take place. The returning federate needs to go through configuration and re-initialization to prepare itself to re-enter a federation. This can be particularly difficult if the rest of the federation is continuously executing a scenario.

Furthermore, a federate might not be allowed to re-enter a scenario at any given time. Due to the current state of the federation and scenario, a federate might have to wait until the rest of the federation (the other federates) allows it.

Design Patterns to handle these situations are not limited to federates joining after a fault has occurred. The normal execution of a federation might allow late-joining federates or federates that leave the federation and rejoin later. Common for all these design patterns are that they are part of the federation agreement and not implemented as part of the HLA standard.

7 Conclusions

With the fault tolerant additions to HLA Evolved it is finally possible to develop fault tolerant HLA federations in a standardized way. While these additions will not compensate for all types of faults that can occur during a federation execution they still make it possible to reach new levels of fault tolerance.

Since the standard only specifies the API for the fault tolerance, not the implementation, we can expect RTIs from different sources to start competing with regards to their fault tolerance capabilities.

A federation and its federates will need to be modified to take full advantage of the fault tolerance additions. This paper provides a number of design patterns for this.

Fault tolerance has been lagging behind in the HLA standard. The new additions will open new possibilities and applications for HLA technology. As the practical experiences from developing fault tolerant federations expand during the coming years we may also see further fault tolerance additions being made to future versions of HLA.

8. References

[1] “Towards Fault Tolerant RTIs, Federates and Federations”, Trevor Pearce, Björn Möller, 05S-SIW-129, 2005 Spring Simulation Interoperability Workshop, SISO, April 2005

[2] “IEEE 1516, High Level Architecture (HLA)”, IEEE, www.ieee.org, March 2001.

[3] “Developing the Fault Tolerance Support Extensions for HLA Evolved”, Björn Möller, Mikael Karlsson, 05E-SIW-019, 2005 European Simulation Interoperability Workshop, June 2005

[4] “RTIperf 1516 a HLA 1516 Performance Test Tool”, Pitch Technologies, www.pitch.se/rtiperf.

[5] “1516.3-2003 IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP), IEEE, www.ieee.org, 2003

Author Biographies

BJÖRN MÖLLER is the vice president and co-founder of Pitch Technologies, the leading supplier of tools for HLA 1516 and HLA 1.3. He leads the strategic development of Pitch HLA products. He serves on several HLA standards and working groups and has a wide international contact network in simulation interoperability. He has twenty years of experience in high-tech R&D companies with an international profile in areas such as modeling and simulation, artificial intelligence and Web based collaboration. Björn Möller holds an M.Sc in Computer Science and Technology after studies at Linköping University, Sweden and Imperial College, London.

MIKAEL KARLSSON is is the chief architect at Pitch overseeing the world’s first certified HLA IEEE 1516 RTI as well as the first certified commercial RTI for HLA 1.3. He has more than ten years of experience of developing simulation infrastructures based on HLA as well as earlier standards. He also serves on several HLA standards and working groups. He studied Computer Science at Linköping University, Sweden.

BJÖRN LÖFSTRAND is Manager of Modeling and Simulation Services at Pitch Technologies.

He holds an M.Sc. in Computer Science from Linköping Institute of Technology and has been working as with HLA federation development and tool support since 1996. Recent work includes developing design patterns for HLA based simulation in the future Swedish Networked Based Defense.