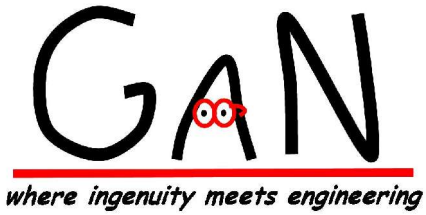

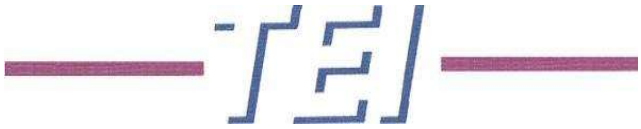

DTE-ARCH Milestone 4: Customized Logging for TENA Middleware

Written By:

Ben Matthews,
GaN Corporation
Computer Engineer

Gary T. Fee,
Technologies Engineering, Inc.
Senior Software Engineer

ABSTRACT: *The DTE-ARCH (Distributed Test and Evaluation - Architecture) Milestone 4 was an event undertaken by the United States (US) Developmental Test Command (DTC) to analyze the future of the distributed M&S environment. This event took place on the 25th and 26th of January, 2005 with participants including Aberdeen Test Center (ATC), Aviation Technical Test Center (ATTC), Dugway Proving Grounds (DPG), Electronic Proving Grounds (EPG) at Fort Lewis and Fort Huachuca, Redstone Technical Test Center (RTTC), One-SAF Testbed Baseline (OTB), White Sands Missile Range (WSMR), and Yuma Proving Grounds (YPG). This event utilized the Test and Training Enabling Architecture (TENA) Middleware Release 4.0.4 to simulate the various components within the operational scenario. Preliminary data was collected during this event from one object model's (OM) published stateful distributed objects (SDO). This data aided in the understanding of how a customized logger could be implemented within a TENA application. This paper presents one method of creating a customized logger that will include additional information not present within the TENA Middleware Logging capability.*

1.0 Introduction

The DTE-ARCH Milestone 4 event was executed over the Defense Research and Engineering Network (DREN). It included test centers (TC) from across the CONUS including ATC, ATTC, DPG, EPG-Ft.Lewis, EPG-Ft. Huachuca, RTTC, WSMR, and YPG. The following figure shows the DREN network and all of the participating sites during the DTE-ARCH Milestone 4 event.

Figure 1: ATEC Sites participating in DTE-ARCH Milestone 4.
(Modification of picture obtained from http://www.spc.noaa.gov/products/ wwa/warnsummary.html)

The DTE-ARCH Milestone 4 event performed a preliminary data collection from published SDO's of one OM that provided insight into how a customized logging function should operate within TENA Middleware Release 4.0.4 (This customized logging method within TENA Middleware Release 5.0.1 has not been tested by this group). TENA Middleware has a built in logging capability that logs everything that is in an OM SDO, along with a publication version number, and puts a time stamp with one second precision on each piece of data. This one second precision obtained from the TENA Middleware Logger is acceptable for some environments, but the DTE-ARCH team needed a higher precision for the time stamped data to provide insight into

latency issues. Since this logger is handled solely in TENA and cannot be easily modified for individual needs, the DTE-ARCH community preceded with the production of a customized logger that could provide a higher time stamp resolution. This was accomplished by adding another layer of logging on top of the TENA Middleware logging to provide a time stamp with microsecond precision.

From the analysis of the SDO data collected by this customized logger used during the Milestone 4 event, observations were made that can help with the development of a customized TENA logger. This paper presents a method of customized logging based on these observations that will aid in future application development.

2.0 Customized SDO Logging

When creating a customized logger within a TENA application, there are differences between implementing the logger for a publisher or a subscriber. The custom publication logger can be implemented in the main source file for the application, while the subscription logger must be implemented in TENA files. This section gives an explanation of the implementations of a customized logger within a subscribing or publishing TENA application.

2.1 Custom SDO Publication Logger

Within a *publish_SDO.cpp* file, the TENA Middleware logger logs the OM data for publications during the `modifyPublicationState()` method, thus the customized logger should be implemented near this command to maintain the integrity of the data. One effective place to put the custom logger implementation is directly before the `modifyPublicationState()` command. A very simple example of the customized logger using XML format (explained later) looks like this:

```
// Update variables
servantUpdater->set_Name(Name);
servantUpdater->set_Payload(Payload);

// Obtain time stamp
ACE_Time_Value systemTime = ACE_OS::gettimeofday();

// Log all the updates in the Custom Logger.
customLogger << "<SDOUpdate>" << std::endl;
customLogger << "<Name>" << Name << "</Name>" << std::endl;
customLogger << "<Payload>" << Payload << "</Payload>" << std::endl;
customLogger << "<localTime>" << std::endl;
customLogger << "  <seconds>" << systemTime.sec();
customLogger << "  </seconds>" << std::endl;
customLogger << "  <microSeconds>" << systemTime.usec();
customLogger << "  </microSeconds>" << std::endl;
customLogger << "</localTime>" << std::endl;
customLogger <<"</SDOUpdate>" <<std::endl;
```

```
// Commit all the updates to the SDO "Servant".
// This will also log in TENA logger if it is implemented.
Servant->modifyPublicationState( servantUpdater );
```

The reason the custom logger should be implemented before the `modifyPublicationState()` command is so the data will get logged as close to the moment when it is disseminated as possible. If something within the delivery of the update causes a momentary pause in the execution of the application, the data will have already been logged and will not wait on the return of the application execution to record the data.

2.2 Custom SDO Subscription Logger

The custom subscribing logger is more involved than the publishing logger. The TENA subscriber works as follows[1]. A TENA SDO subscribing application will execute the command `evokeMultipleCallbacks()` in order to receive any published updates of that SDO. When a subscriber receives an update from a publisher, one of three callback implementations are executed: Discovery, State Change, or Destruction. The Discovery callback is used by the subscriber to execute a piece of code for every SDO that joins the execution. The State Change callback executes code for every SDO state change that takes place during the execution. The Destruction callback executes code when an SDO leaves the execution. When an SDO distributes the first publication, the Discovery callback is executed first, followed by the State Change callbacks for each published update until a Destruction callback is sent to the subscribing application.

In order to obtain all updates from all publishing applications, the subscription logger needs to be present in at least two separate locations. The first location would be in the *DiscoveryCallbackImpl.cpp* file. This allows the logger to include the first update that is sent to the subscriber. This will be followed by State Change callbacks, which means that the logger should also be implemented in the *StateChangeCallbackImpl.cpp* file. This will log all additional updates sent to the subscriber. An example of the code within these two files follows:

```
// Get update variables
Proxy->get_Name(Name);
Proxy->get_Payload(Payload);

// Obtain time stamp
ACE_Time_Value systemTime = ACE_OS::gettimeofday();

// Log all the updates in the Custom Logger.
customLogger << "<SDOUpdate>" << std::endl;
customLogger << "<Name>" << Name << "</Name>" << std::endl;
customLogger << "<Payload>" << Payload << "</Payload>" << std::endl;
customLogger << "<localTime>" << std::endl;
customLogger << "  <seconds>" << systemTime.sec();
customLogger << "  </seconds>" << std::endl;
customLogger << "  <microSeconds>" << systemTime.usec();
customLogger << "  </microSeconds>" << std::endl;
```

[1] For a detailed description of callback handling, please refer to the TENA Middleware Programmer's Guide located at http://support.fi2010.org/doc/.

```
customLogger << "</localTime>" << std::endl;
customLogger <<" </SDOUpdate>" <<std::endl;
```

If this is implemented in both the Discovery and State Change callback implementation, all updates presented to the subscriber should be logged. Specifically, information for the log, such as the current time and SDO state, should be captured in the constructors of the *DiscoveryCallbackImpl* and *StateChangeCallbackImpl* classes. These objects are instantiated by the TENA Middleware, independent of the application's call to the `evokeMultipleCallbacks()` method. These are the only two places in the code where this is necessary, but, if desired, a similar implementation could be included within the *DestructionCallbackImpl.cpp* file to log when each subscribed SDO is destroyed.

2.3 Log File Formats

In the TENA Logger and the customized logger, separate files need to be created for both the publisher and the subscriber. In total, each application should create two log files, one for the TENA Logger and one for the custom logger. If the application is both a publisher and a subscriber, then four log files should be created.

The log files created by TENA Middleware are formatted in XML (eXtensible Markup Language). An example of this format and instructions on how to implement the TENA Middleware Logger can be found in the TENA Middleware Programmer's Guide which can be found at http://support.fi2010.org/doc/. The data collected by the customized logger in the DTE-ARCH Milestone 4 event also used an XML format, but the format for future custom logging methods is left up to the developers. If an XML format is used, the implementation of the format will need to be part of the code (see examples above). Other formats can be used for the custom logger, but the TENA Middleware Logger will remain in an XML format.

2.4 Naming of Log Files

To keep separate executions of the application in separate log files, a creative means of naming the file should be implemented into the customized logger, as well as the TENA Middleware logger. One method is to produce a time stamp that is appended to the end of the log file name for each application execution. For example, the file name(s) could be:

"CustomSDOProxiesLog.22Apr2005.04.09.00.txt"

and/or

"TENASDOProxiesLog.22Apr2005.04.09.00.txt"

This would create a unique file name every time the application is executed. This also protects against the application reopening an already existing file and overwriting the information contained within the file. This problem and possible solutions are discussed in the next section. The following code provides an example of how a file name with a time stamp appended can be

created for TENA Middleware log files.

```
// Variables for FileName string and time stamp
ACE_TCHAR DateTime[35];
ACE_TCHAR * aceTChr (ACE::time stamp(DateTime, 35, 0));
std::string FileName("CustomSDOProxiesLog.");

// Append time stamp and extension to FileName
FileName += DCT::Utils::toString(DateTime);
FileName += ".txt";

// Replace all ':' with a '_' for naming convention.
for (unsigned int i=0; i<FileName.length(); i++)
{
          if (FileName[i] == ':')
          {
                    FileName[i] = '_';
          };
}
```

2.5 Ensuring Against Overwrite

Some overwriting occurred in the data that was obtained from the DTE-ARCH experiment. The application opened the same log file used during the previous execution and began overwriting existing data. Several options can be employed to safeguard against this occurrence. One is to append a time stamp to the end of the name of the file as as explained in the previous section. Another option is to append to the end of existing log files. A combination of these methods could be used to double check that no data is being permanently removed from the log files.

Appending to the end of a log file becomes problematic when an application is exited ungracefully. If *CTRL-C* is used to interrupt and exit the program, the log file may not have been closed and/or the update entry may be incomplete. This could cause the situation where trying to append to the existing log file would cause overwriting. One way to avoid this problem is to create a handler for the *CTRL-C* interrupt. The following segment of code shows how a handler is used. This code will print "CTRL-C was used" to the standard output when the interrupt occurs.

```
Include files:
#include <signal.h>                    // included for Windows
#include </sys/signal.h>               // included for Linux

Include within main function:
signal(SIGINT, handler);               // Handles the CTRL-C event.

Handler function:
void handler(int sig)
{
     std::cout << "CTRL-C was used";  // print to screen
}
```

In order to close the log file, the handler function would need to modify a global variable and

return to the main function, which would then allow the program to exit gracefully. More options may exist for handling the *CTRL-C* interrupt problem, but this is one solution that could be useful for future application development. More development and testing should be conducted to fully test this capability within TENA Middleware to prevent overwriting problems.

3.0 Conclusions

This paper has provided a description of how a customized logger can be implemented to include additional information not present in the TENA Middleware Logging capability. The description of the customized logger addresses specific issues encountered during data analysis from DTE-ARCH Milestone 4 using TENA Middleware Release 4.0.4. While this paper is not meant to provide a comprehensive description of a customized logger, it will provide some insight to developing a logging function within a TENA application. This custom logger has not been tested with TENA Release 4.0.4.1, 4.1, or 5.0.1, but this general implementation should not be effected by the TENA Release version and should be a portable method of implementing a custom logger.