2009 Ninth International Conference on Quality Software

Quality Assessment of Mission Critical Middleware System Using MEMS

Yan Liu[1,3], Kate Foster[2], Thong Nguyen[2] and Jacky W. Keung [1]

[1]National ICT Australia Ltd.(NICTA)
[2]Air Operation Division, Defence Science and Technology Organization
[3]School of Computer Science and Engineering, University of New South Wales, Australia
{jenny.liu,jacky.keung}@nicta.com.au, {kate.foster,thong.nguyen}@dsto.defence.gov.au

Abstract

Architecture evaluation methods provide general guidelines to assess quality attributes of systems, which are not necessarily straightforward to practice with. With COTS middleware based systems, this assessment process is further complicated by the complexity of middleware technology and a number of design and deployment options. Efficient assessment is key to produce accurate evaluation results for stakeholders to ensure good decisions are made on system acquisition. In this paper, a systematic evaluation method called MEMS *is developed to provide some structure to this assessment process. MEMS produces the evaluation plan with thorough design of experiments, definition of metrics and development of techniques for measurement. This paper presents MEMS and its application to a mission critical middleware system.*

1 Introduction

Mission critical systems incorporate component-based and distributed computing systems. These systems are built on middleware, which is a class of software infrastructure technologies that use high-level abstractions to simplify the construction of distributed systems. This infrastructure provides a distributed environment for deploying system-level components. The system components rely on the middleware to manage their life cycles and execution, and to provide off-the-shelf services such as transactions and security. Consequently, the system behavior and the middleware architecture are tightly coupled, and the middleware plays a critical role in achieving the requirements of mission critical systems. As a result, the software system acquisition process has come to consider the risk associated with middleware architecture. If the middleware is inefficient or lacking in features, contains subtle errors or the architecture is poorly designed, the projects built on such middleware often incur schedule delays, cost overruns and reduced functionality.

The earlier risks are identified, the greater the probability that the system will be cost effective, delivered to schedule and perform as required. Therefore, the acquisition process should have the capability of evaluating architectures built on middleware.

To remedy this situation, a number of architecture evaluation methods provide general guidelines to assess quality attributes of systems. A comprehensive review can be found in [5]. These methods are applied at the application level, where middleware is considered as features or constraints of implementing the applications. The strong relationship between a middleware technology and the quality of the overall system are not explicitly addressed. While these methods can be extended to incorporate middleware in theory, there are a number of problems with trying to implement them in practice. One problem is that quality attributes (such as performance, liveness and scalability) are abstract entities to the stakeholders, making it difficult for stakeholders to quantify the expected quality of the mission critical systems built on complicated middleware.

The ability of a middleware technology to support given quality attributes depends on the mechanisms and services provided by the infrastructure. This indicates that evaluation methods require detailed technical inputs regarding the middleware infrastructure, including its programming model, APIs, configuration and deployment. This kind of knowledge helps to identify the effect of different middleware architectures on quality attributes. Given the increasing complexity of middleware technologies, improved evaluation techniques are required.

To this end, we have developed a Method for Evaluating Middleware architectureS (MEMS) [10], which measures middleware architectures by rating multiple quality attributes. MEMS was initially designed for comparing several architectures of Grid applications. The output of

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.41

MEMS helps to determine the suitability of an architecture to meet quality goals of the system. However, one limitation of MEMS (as other architecture evaluation methods) is that it only provides general guidelines to access quality attributes, and lack of a structured way to design experiments for evaluation. A unwisely devised experiment may miss the key characteristics of middleware, and produce misleading results. Given the set of factors to consider for experiments (including a number of quality attributes, key scenarios and configurable parameters), designing and carrying out sound experiments are non-trivial tasks.

We noted these problems as we prototyped MEMS in a real-world setting, and in response we made substantial extensions to MEMS to design experiments and measure quality attributes. The extension of MEMS includes:

- Capturing architecture patterns in additional to key scenarios;
- Developing experiments and measurement techniques to fulfill the evaluation plan;
- Conducting both qualitative and quantitative analysis.

The contribution of this paper is twofold. First, it presents a systematic method called MEMS to guide the evaluation of multiple quality attributes; and secondly it devices key experiment techniques to realize MEMS with rigorous measurement.

The structure of this paper is as follows. The related work of architecture evaluation methods is discussed in section 2. A real-world motivating case is introduced in section 3 which setups the context of this paper. Section 4 presents the MEMS method and highlights the key steps. Section 5 focuses on the experiment design and measurement techniques. Section 6 demonstrates how the experiments support the evaluation and presents the final result. The paper concludes at section 7.

2 Related Work

Our approach does not ignore or conflict with existing scenario-based architecture evaluation methods [5] or other approaches to assess risk factors [3]. Rather, our approach extends and complements them by evaluating middleware with dedicated steps. We consider related work from two streams, namely software architecture evaluation and middleware technologies evaluation.

2.1 Architecture evaluation methods

Software architecture evaluation methods and techniques focus on understanding the relationship between software architecture and one or more quality attributes to ensure that the system ultimately achieves its quality goals while still supporting its functional requirements. A review of these techniques can be found in [1] and Chapter 6 of [2]. MEMS falls into the category of scenario-based software architecture evaluation methods. Scenarios are defined to understand how a software architecture responds with respect to attributes such as maintainability, reliability, usability and performance. Examples of scenario based methods are: Software Architecture Analysis Method (SAAM), Architecture Trade-off Analysis Method (ATAM) and Architecture Level Modifiability Analysis (ALMA).

ATAM is a two-phase method. The first phase's inputs include general scenarios (or requirements from stakeholders), software architecture design documentation and the formation of the evaluation team. The tasks in the first phase are to transform general scenarios into specific scenarios and evaluate the software architecture against specific scenarios. The second stage of ATAM presents the results to stakeholders, who provide the business goals, and matches the software architecture with the business goals to analyze the impact of architecture changes based on each scenario. ATAM also deals with multiple quality attributes. ATAM collects stakeholder's requirements in brainstorming sessions on the scenarios related to the business goals.

Technically, MEMS targets middleware, which is a component of the overall software architecture to be evaluated. Therefore MEMS demands middleware specific techniques and tools to evaluate the architecture. This also means the roles involved in MEMS are more technical, requiring architects and designers who have considerable knowledge and experience of using middleware [7].

2.2 COTS middleware acquisition

The i-Mate process has been applied to evaluate COTS middleware technologies, especially for acquisition of middleware for enterprise applications [9]. i-Mate is similar to the first phase of ATAM, and requires stakeholders to input the business requirements for the middleware to be acquired. The evaluation of performance and scalability is conducted in a lab environment by running a predefined benchmark application on all the candidate middleware with the rest of the evaluation test environment remaining identical.

Both i-Mate and MEMS require middleware infrastructure specific techniques, because prototyping with the middleware is essential to conducting the assessment. MEMS is different from i-Mate in that the business goal of acquisition is imposed on outputs of MEMS and is not a portion of the method. The evaluation is driven by concerns about the quality attributes for specific designs using middleware. In this sense, MEMS is more lightweight and agile than i-Mate.

3 Motivating Case Study

In the context of this paper, the acquisition process focuses on the software system managing tracks in the airborne mission system. In the aviation environment, an aircraft operates amongst a number of targets such as other aircrafts and ships. The sensors such as radars provide their airborne mission system with sensor reports, which are detections of the targets. These sensor reports are correlated and fused to generate one track for each target in the aviation environment. These tracks are displayed on screens so that operators can view a surveillance picture that represents the activity in the aviation environment.

To evaluate the COTS middleware acquired to develop the airborne mission system, a Hybrid Mission System Testbed (MST) is used. The Hybrid MST is not a simulation of the airborne mission system; rather it incorporates components similar to those found in such systems. The software stack consists of the operating system, the Software Architecture Framework (SAF) - a COTS middleware hosting software components, and the Solipsys Tactical Display Framework (TDF).

The key software components are briefly described below (as shown in Figure 1). A *Test Track Generator* populates and updates the *Track Manager* component with random tracks. A Track Manager component maintains the tactical state of the aircraft. It achieves this through a common repository of all tracks in the environment and a collection of capabilities that can be applied to these tracks. An *Ownship* component maintains the kinematic state of the aircraft. An Adapter component provides an interface between the Hybrid MST and a COTS tactical data link gateway. A *Track Monitor* component periodically accesses and displays the details of all tracks in the Track Manager to an operator. The *TDFAdapter* is technically not a component, but an object activated by the Track Monitor. The TDFAdapter provides an interface to the Solipsys TDF application to enable visualisation of tracks contained in the Track Manager.

Together these components and SAF form a multi-threaded, component-based, distributed environment that is able to create, update, manage and use data objects of tracks. These components are implemented using a component technology similar to CORBA Component Model (CCM), and it is running on top of SAF. The overall testbed is running in a networked environment. Detailed description of the Hybrid MST can be found in [6].

A key service provided by the SAF is concurrency. The SAF provides shared access to resources within a single process by extensively employing threads. Communication between threads is more efficient than many other concurrency constructs as memory address spaces are not swapped until a process is context switched. However, this introduces its own complexity, requiring further synchronization

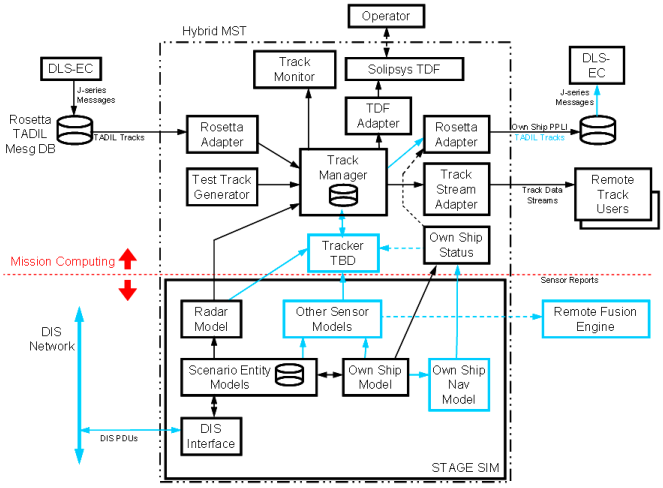

Figure 1: Hybrid MST software components and interactions (from [6])

and notification patterns to ensure the quality and correctness of code is maintained.

Our understanding of Hybrid MTS indicates that the software components and their interactions under concurrency enabled by middleware determine the key scenarios of quality attributes. Therefore, the MEMS approach should be applied within the context of the middleware technology used to implement these software components.

4 Two Stages of MEMS

The process of evaluation involves defining and documenting the evaluation artifacts. It also includes detailed descriptions of test cases for experiments and measurements. As there are a number of factors to consider, organizing these artifacts following MEMS provides a structured way of evaluation. In this section, we first introduce MEMS and then describe how these artifacts are produced from MEMS steps.

MEMS are driven by key scenarios that describe the behaviour of a middleware architecture with respect to particular quality attributes. The quality goals and their expression in the form of key scenarios drive the evaluation process. MEMS was presented in more details in [10]. The evaluation process can be divided into two stages. The first stage is the development of the evaluation plan. This stage encompasses the first four steps of MEMS:

- Determine critical quality attributes;
- Identify key architecture patterns;
- Develop key scenarios;
- Define metrics for each quality attribute.

The outcomes of this stage include:

- Well-defined key scenarios with a common understanding between the evaluation team and stakeholders;
- Well-defined metrics for quality attributes;
- A plan for prototyping and deploying experiments.

The second stage of the evaluation process involves the last three steps of MEMS namely prototyping, carrying out the experiments, analyzing the measurements. Techniques, tools and mechanisms needed for taking measurements are developed. The outcomes of this stage are the evaluation results.

4.1 Determine Critical Quality Attributes

The critical quality attributes are nominated by software architects or engineers who have architectural knowledge of the system and can transform or map the acquisition goals from stakeholders to quality attribute descriptions. These quality attributes determine the key scenarios for the evaluation process.

Two architectural design patterns are identified that they strongly impact the behavior of the Hybrid MST. These two patterns are *Active Object* and *Leader-Follower* [12], both dedicated to concurrency. The Active Object pattern decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. The Test Track Generator and Track Updater are reader/writer components using this concurrency pattern.

The SAF middleware also implements the leader-follower pattern. SAF structures a pool of threads to share a set of event sources (i.e. Test Track Generators) efficiently by taking turns demultiplexing events (i.e. track updating requests) that arrive on these event sources and synchronously dispatching the events to application services (i.e. Track Updater) that process them.

Quality attributes are selected to reflect the key characteristics that these patterns impose to the system. Table 1 lists the quality attributes for evaluating the Hybrid MST running on the SAF middleware.

Table 1: Critical quality attributes of the Hybrid MST

Quality Attributes	Brief Description
Performance	The responsiveness of the system under different workload patterns in terms of throughput and response time of method invocations.
Scalability	The ability of the system to handle increasing workload for two cases: (1)the number of tracks generated from one source increases; and (2)the number of track sources increases while the number of tracks generated remains constant.
Modifiability	How easily the system can be modified to meet changes in the functional specification, environment or requirements. The following scenarios are considered: (1)configure different threading policies to optimise concurrency control; and (2)add different input sources and display outputs.
Configur -ability	To what extend the mission critical system can be extended by programming with APIs and/or configuration options.
Liveness	The ability to execute in a timely manner. In this case, liveliness is indicated by the possibility of deadlock.

4.2 Develop Key Scenarios and Define Metrics

Key scenarios are discussed among members of the evaluation team. From these discussions ten key scenarios required to evaluate quality attributes are identified, seven scenarios for Performance, Scalability and Liveness, and three for Modifiability and Configurability. These scenarios are ranked according to the priority of quality attributes. Performance and scalability are the two most important quality attributes. Scenarios for liveness, modifiability and configurability are also related to achieve performance and scalability goals. For example, SAF provides a number of configuration options, each may have an indication on performance. How easily and efficiently to modify the Hybrid MST built with different SAF configuration options is of the concern of both modifiability and performance. Due to the space limitation, we only listed the scenarios of performance/scalability/liveness to later illustrate the experiment design and setup in Section 5.

Scenario 1 A single Track Writer (i.e. no updates) generates the required number of tracks for a number of writing rates and sends the tracks to the Track Manager as they are created.

Scenario 2 Multiple Track Writers (i.e. no updates) generate the required number of tracks each and the tracks are sent to the Track Manager with the maximum writing rate from Scenario 1. The number of Track Writers is increased until the performance of the Track Manager degrades (throughput drops).

Scenario 3 A single Track Updater (i.e. Track Writer stopped after writing the required number of tracks)

updates and sends tracks to the Track Manager for a number of update rates.

Scenario 4 Multiple Track Updaters (i.e. Track Writers stopped after writing the required number of tracks) update and send tracks to the Track Manager for the maximum update rate from Scenario 3. The number of Track Updaters is increased until the performance of the Track Manager degrades (throughput drops).

Scenario 5 Measure the invocation time for displaying tracks with different numbers of tracks in the Track Manager.

Scenario 6 A mixed workload of writing, updating and displaying tracks is generated.

Scenario 7 There is a possibility of thread deadlock occurring during Scenarios 1 to 6. Directly measuring these occurrences is difficult, therefore the time taken to acquire locks (measured from Scenario 1-6) is used as an indication of deadlock.

From these discussions the performance critical components, operations and parameters of the Hybrid MST are further specified in Table 2.

Table 2: Components, operations and parameters

Components	Operations or Parameters	Purpose
Track Writer	1.Number of Track Writers 2.Number of tracks per Track Writer 3.Track generation rate	Workload generation
Track Updater	1.Number of Track Updaters 2.Number of tracks per Track Updater 3.Track update rate	Workload generation
Track Manager	1.Update track repository 2.Retrieve tracks 3.Iteration operations 4.Thread pool size 5.Time to acquire locks	Processing tracks

The scenarios are documented using these components, operations and parameters in a template of three fields, namely descriptions, metrics and configuration. A sample scenario documentation is shown in Table 3.

Table 3: Sample of Scenario 1: Performance/Scalability

Description	A single Track Writer (i.e. no updates) generates the required number of tracks for a number of writing rates and sends the tracks to the Track Manager as they are created.
Metrics	1.Response time/throughput of update track repository 2. Time taken to acquire locks 3. CPU utilization 4. Number of active threads
Configuration	1. Number of tracks generated 2. Track writing rate

5 Experiment Design and Measurement

5.1 Modeling Experiments

Common software components and metrics exist in several scenarios. This indicates that some artifacts in one scenario can be reused in another scenario. Now the question is how to implement these scenarios with reusable and extensible elements. This can be answered by modelling the basic elements of experiments that implement individual scenarios. The following four elements are abstracted:

Metrics have the same meaning of the metrics in each scenario. It needs to be realized with customized metrics listed in each scenario description;

Test Track Generator should be configurable to generate different mix of workload;

Configuration captures configurabe parameters of each scenario;

Constraints are attached to each scenario to further scope the test case. For example, threading policy of blocking and dropping requests can only be applied when the worst-case workload occurs.

These four elements form the basic notations for modeling. All experiments need the Test Track Generator to drive the experiments. Each experiment consists of specific configuration (such as the rate of track generated), and constraints. An experiment also has a set of metrics to measure. Some configurations, metrics and constraints may be used in several experiments and their relations are modeled using UML.

A sample model to describe experiments for Scenario 1 and 2 is shown in Figure 2. In this template, the *Test Track Generator* has *Metrics*, *Configuration* and *Constraints* as its members. As configurations and constraints are optional to a scenario, they are not strong aggregations of the Test Track Generator. Individual scenarios can be described by extending or implementing these four elements. In this sample template, Scenario 1 (Single Creation Track Generator) and Scenario 2 (Multiple Creation Track Generator) both have common metrics captured in the *Single Creation Metric* element. Scenario 2 has some extra metric as the number of time out events, and therefore the *Multiple Creation Metric* extends the *Single Creation Metric*. Similarly, this

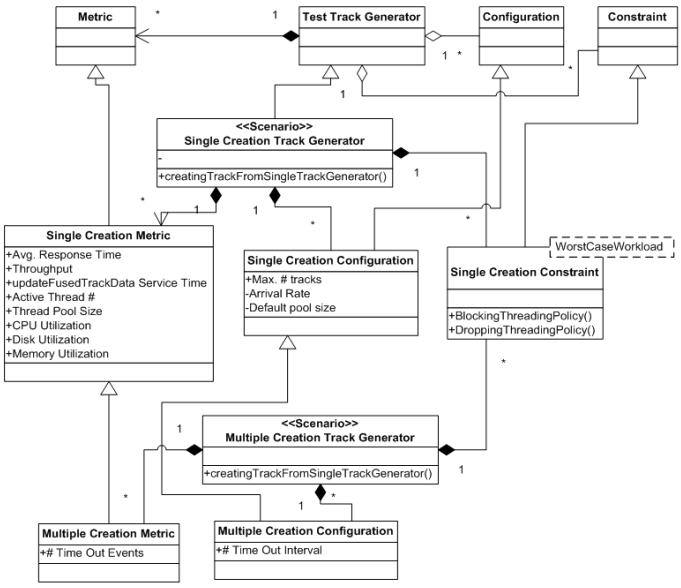

Figure 2: Test Case Generation Template

template can be extended to cover the remaining evaluation scenarios.

The scenario descriptions help to produce the experiment models. The experiment model captures the dependencies of multiple scenarios and helps to develop experiments with reusable software code or scripts. Together the scenario descriptions and experiment models are used to organize the experiments by scenarios and with incremental complexity.

5.2 Measurement Techniques

The metrics listed in all scenario definitions guide the measurement of the Hybrid MST. Even these metrics are well defined, not all of them are straightforward to measure. Hence, a number of instrumentation techniques are introduced. Although they are employed for specific scenarios, the method to devise them is in a general architecture evaluation context. In this section, we present the details of these techniques including *approximation*, *instrumentation*, *configuration files*, *logging and data tagging* and *synchronization*.

Configuration File. A scenario has a set of configurations defined in its description. The experiments follow these configurations and setup the properties in a configuration file. These properties are read when the SAF server is started and used to configure the appropriate component. The list below shows an example script to configure the Test Track Generator for Scenario 1. Some code of Hybrid MST may be modified to enable the configuration. This includes settings of the maximum size of the thread pool, the maximum number of tracks created by the Test Track Generator, the track creation rate of the Track Writer and the number of active objects (Track Writers) to synchronize.

```
[config.properties.experiment1]
# Test Track Generator properties
TestTrackGen.MaxTracks=200
TestTrackGen.WriterRate=1 #tracks/sec
TestTrackGen.UpdaterOn=0 #0 = not on
TestTrackGen.UpdaterRate=0 #tracks/sec
# Track Manager Properties
TrackManager.EvictorOn=0 #0 = not on
TrackManager.ThreadPolicy=default
TrackManager.PoolSize=200
```

Instrumentation. It is a technique to insert timers to a piece of code and record the readings. The time taken to process the code is estimated as the difference between readings. In this evaluation, the quality attribute Liveness can be measured by recording how many deadlock cases occur. However, it is currently difficult in the SAF to set the maximum time a calling thread is blocked when trying to acquire a lock. Therefore, the time taken to acquire locks metric provides an indication of the liveness of the Hybrid MST by wrapping synchronisation points in high resolution timers.

This is done in the following way. An high resolution timer is created and started just before the synchronization point. At the synchronization point, a thread attempts to take the lock and waits if the lock is already taken. When the thread acquires the lock, the timer is stopped and the thread proceeds with the locked code. Once the program flow is out of the locked code, the time taken to acquire the lock is output and caught by the Logging Service (discussed later in this section).

The instrument code has to be carefully placed so that it is not inside locked code as it would significantly increase the time taken to traverse the locked code and consequently the time waiting threads spent in the blocked state. The experiments show that for 1-20 Track Writers, the lock acquisition time is fairly consistent and generally remains within 3-7 μs for the write lock. This indicates the increasing of workload has little effect on the lock contention and therefore has little possibility on causing liveness problems.

Logging and Data Tagging. The instrumentation class collects performance data (such as response time of methods). Data output from this class is captured by the Logging service. Data output from the logging service is tagged with other information such as machine name, process ID and piped to a file. A number of Perl scripts are developed to process the files produced by the Logging service and eventually produce the MATLAB plots like Figure 3.

Approximation. When it is technically infeasible to measure the exact metric, an approximation is introduced with caution that the approximation can still represent the

performance characteristics of the original metric. One example is the total response time of the method of updating track data - measuring the time from the Track Manager receiving a request to when a track is added or updated in the Track Manager. It was difficult to measure because this method involves the concurrency operations, and not being able to measure the time taken to schedule the thread from the thread pool. However, an approximation of this metric can be provided by measuring the time taken to handoff a track to the thread pool and the time taken to add or update the track in the Track Manager.

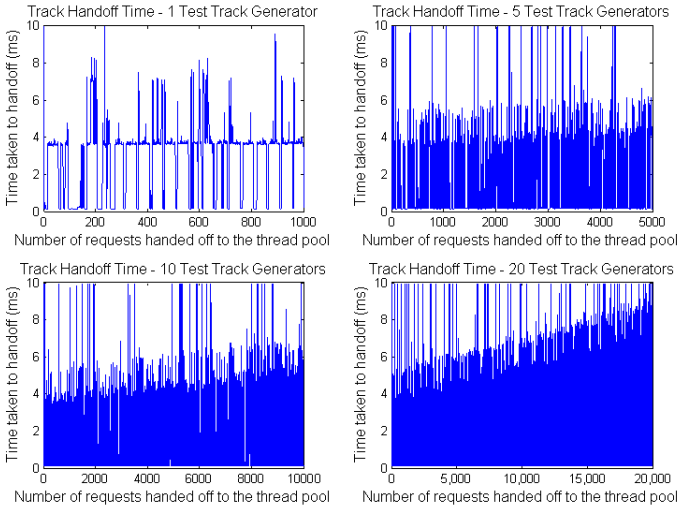

Figure 3: Tack Handoff Time for 1-20 Track Writers

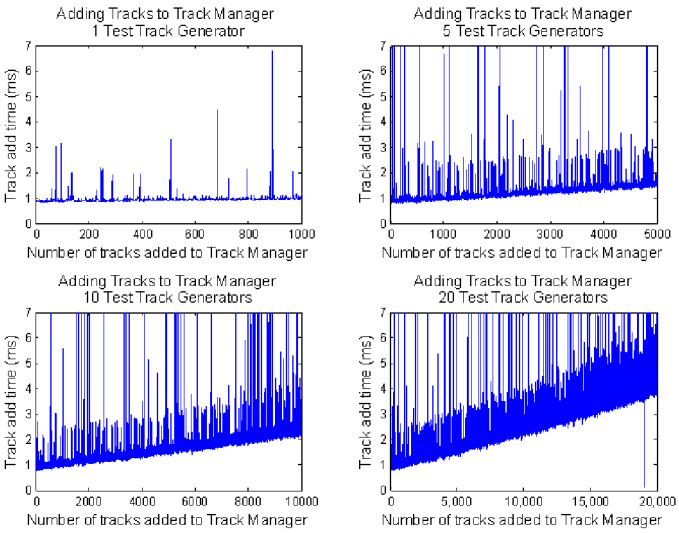

Figure 4: Tack Writer Time for 1-20 Track Writers

Figure 3,4 show the approximation by two separate metrics used in experiments of Scenario 1 and 2 for examples. In Figure 3, there is an increase (increased density) in overall track handoff time as the number of Track Writers increases from 1 to 20. In Figure 4, there is also an increase (increased sloped) in track adding time as the number of Track Writers increases from 1 to 20. A closer inspection at the implementation reveals the reason for these results. This is due to the increasing number of tracks contained in the Track Manager. Tracks are stored in the Track Manager in a C++ Standard Template Library (STL) map. A larger track population means that there are a greater number of leaf nodes to traverse in the map, which in turn increases the amount of time it takes to add a track to the Track Manager. We can see that the two separate metrics produce consistent trends to approximate the original metric of total time taken.

Synchronization. To evaluate the performance of updating the tracks contained in the Track Manager, the Track Writer have to first create the required number of tracks and then add them to the Track Manager before the Track Updater can start. A lock using SAF libraries is implemented to synchronize the Track Writer and Track Updater active objects within a single Test Track Generator. The lock is created and acquired before the Track Writer and Track Updater are started. Once the Track Writer has written all the tracks and the lock is released. This notifies the Track Updater which is blocked on the lock. For multiple Test Track Generators, a synchronization point is needed for all the Test Track Writers to finish writing the required number of tracks before any Track Updaters are started. A Barrier component is created to enable the synchronization of multiple Track Writers and Updaters (required for Scenario 4). Once all Track Writers check in, they are all released from the Barrier component, and then they release the lock in their own Test Tack Generators so that their associated Track Updaters are notified and can run. The Barrier component has a property to specify how many active objects are synchronized, and therefore the attribute *WT.Barrier.Parties* is added to the configuration file. For scenarios that only has a single Test Track Generator, the value of *WT.Barrier.Parties* is set to one.

5.3 Testbed Deployment

The testbed consists of a Sun Blade 1000 hardware platform connected to a network in a laboratory. The platform consists of one CPU (900MHz UltraSPARC-III+) and 1 GB of RAM. The COTS middleware SAF is running on operating system Solaris 8 together with a number of components developed by DSTO (as introduced in Section 3). The deployment of key components for each scenario is shown in Figure 5. The use of single processor machine for the evaluation aims to provide a baseline performance. The evaluation can be repeated using a multiprocessor platform and/or a number of platforms distributed across a network to measure the impact on the critical quality attributes identified in Table 1.

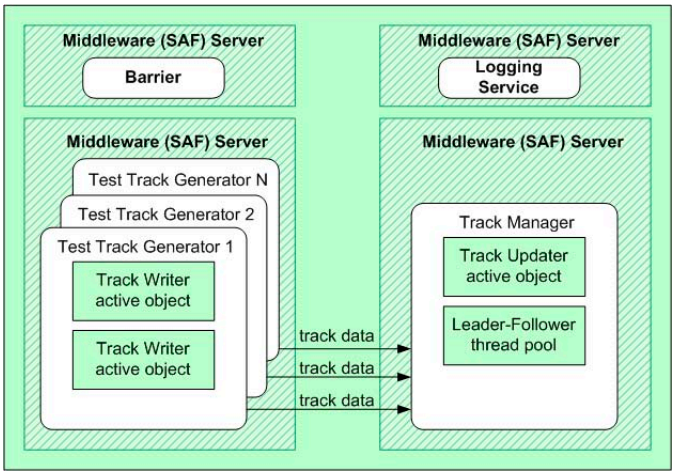

Figure 5: Sample Component Deployment

6 Evaluation Results

Presenting evaluation results involves two aspects. The first aspect is analyzing the measurements collected from the experiments, and drawing rational conclusion on the system's quality. This is initially done by the engineers who conduct the experiments. The second aspect is to present both the analysis and the early conclusion to other evaluators who have practical knowledge of the system under test and can reason about the rating of a quality attribute. We use some sample results to illustrate these two steps.

6.1 Analysis of Measurement

The experiments are designed with incremental complexity of testing. For example, Scenario 3 starts with a single Track Updater within a Test Track Generator. Three experiments are designed, each with different track update rate (1, 10 and 50 updates per second respectively). Then Scenario 4 involves multiple Track Updaters (each with their own Test Track Generators) continually generating track updates and sending these to the Track Manager for the maximum update rate from Scenario 3 (i.e. 50 updates per second). Such experiment design establishes rationale between scenarios and helps to diagnose the behavior of Hybrid MST across experiments.

Figure 6 shows the active threads count plot for Scenario 3 and Scenario 4. Figure 7 shows the CPU utilization plot accordingly. The CPU utilization shows that the Track Manager process initially increases sharply with increasing workload, but eventually reaches a steady state for Test Track Generators up to 10. Figure 6 also shows that the thread pool quickly spawns many threads. These two plots demonstrate that the SAF middleware scales well to handle increasing workload upto 1000 updates per second (i.e. 50 updates per second multiples 10 Test Track Generators). For 20 to 50 Test Track Generators, the comparison of the two plots shows that when the number of active threads increases sharply at the start of the track updating stage, the CPU utilization decays to a lower level and then remains steady. This may due to at this stage, enough threads are available to handle the increasing workload, therefore no more CPU overhead incurred by spawning threads.

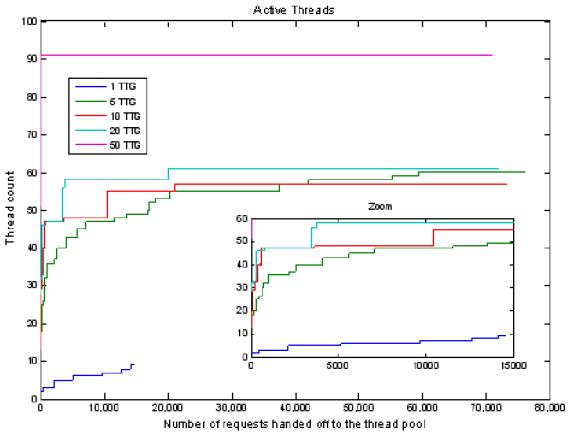

Figure 6: Active threads count

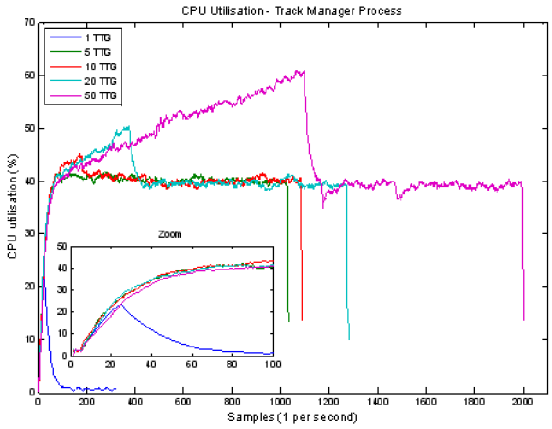

Figure 7: CPU utilization

The analysis of the measurement highlights the advantages of the concurrency patterns used by SAF. The responsiveness of performance actually benefits from the combination of the Track Updater active object and the leader-follower thread pool. The active object pattern allows a number of threads to interleave their access to the repository of tracks, and therefore enhances concurrency and simplifies synchronized access. The leader-follower pattern enhances system responsiveness to requests, and has low overhead handoff and scales to match demand (through the thread pool). The use of these patterns is evidence of good architectural design of the SAF. This enables the Hybrid MST to be responsive, and ensures that CPU is not saturated for intensive workload (e.g. 50 updates per second x 50 Test Track Generators = 2500 updates per second).

6.2 Rating Quality Attributes

The outcomes of the measurements and the initial analysis are the inputs to the second stage of MEMS. There are three key steps:

First, define quality attribute scale. For qualitative attributes, one common approach to consolidating evaluation results is the weighted scoring method (WSM) [8]. The WSM method requires a clear and unambiguous definition of a rating scale, so that evaluators can give weights or scores to qualitative attributes with respect to the architecture under evaluation. Quantitative attributes need to be transformed into the same scale as the qualitative attributes so that all quality attributes can be presented in the same radar diagram. This mapping can follow the same practice of dealing with qualitative attributes. The difference is that the rating scale definition of quantitative attributes should include quantitative expectation. For example, response time of performance is high means 90% response time should be within a second.

Second, rate quality attributes. At this stage, the rating scale will be used to gauge each quality attribute. We revisit the quality attributes of interests in this evaluation and then define the rating scale of each quality attribute. Ozkaya et al. [11] examined 24 different SEI-led evaluations from 1999 to 2007. The empirical analysis shows that modifiability and performance are the top 2 concerns out of 140 concerns. Among the top 20 concerns, other concerns of interest in this project include configurability, reliability and scalability. The findings from SEI are consistent with the quality attributes of interests in this evaluation.

Finally, aggregate rating scores. Three researchers from DSTO who have practical knowledge of the Hyprid MST conduct rating exercise according to the rating scale definition. Before we reach to the conclusion of the ranking for each quality attribute, we apply the inter-rate reliability analysis [4] to examine the agreement between any pair of two evaluators. The original ranking is summarized in pairs, and the Kappa statistic is calculated to examine the agreement level. One interpretation of Kappa statistic is that the value of Kappa statistic ≥ 0.8 means good agreement and between 0.4 and 0.6 means fair agrement. For any pair that gets the Kappa statistic value below 0.6, the evaluators are further interviewed to identify the rationale behind their ranking gap. The necessary emendation is introduced either to remove the ambiguity of the scale definition or redo the ranking between the pair. Eventually the rankings from all the evaluators can be consolidated and listed in Table 4.

The algorithm presented in Figure 8 is applied to transform the qualitative ratings (Table 4) into values with weights. Thus the overall result is presented in a radar diagram in Figure 9 with each axis representing an individual quality attribute. It can clearly demonstrate how the system performs with respect to a specific quality attribute. This rigorous assessment of the quality attributes ensure stakeholders are aware of the quality when they make acquisition decisions for COTS middleware-based systems.

Table 4: Final ranking of the quality attributes

Performance 1. Response time = high end of medium 2. Latency = medium to low 3. Real time = medium 4. Resource utilization = low
Scalability Response time vs. workload = medium
Liveness Possibility of deadlock = low
Modifiability 1. New/revised functionality/components = high 2. Portable to other platforms = medium 3. Upgrade components = low end of high
Configurability Flexibility (range of testing scenarios) = high

1. Assign values to High, Middle and Low to convert the rating of each sub-category into values correspondingly.

$$H \leftarrow 1, M \leftarrow 2/3, L \leftarrow 1/3$$

2. Calculate the overall rating based on weight and the converted rating of each sub-category

$$Rate = \sum_{i=1}^{n} Weight_n \times Rate_n \qquad (\sum_{i=1}^{n} Weight_n = 1)$$

where n is the number of sub-categories

3. Convert back the value of the overall rating into the descriptive form.

$$Rate \leftarrow \begin{cases} H & (2/3 < Rate \leq 1) \\ M & (1/3 < Rate \leq 2/3) \\ L & (0 < Rate \leq 1/3) \end{cases}$$

Figure 8: Simple algorithm for calculating rating

7 Conclusion

This paper presents the MEMS method and its experimenting techniques in the process of middleware architecture evaluation. MEMS utilizes scenarios to organize artifacts (including configurations, constraints and metrics) centric to quality attributes. A number of experiments are devised to evaluate each scenario. Each experiment takes inputs of a configuration file and customizes its running according to the settings of configuration parameters, constrains and target metrics. The experiments with incremental complexity also help to observe the trend of the system under workload changes. The outcomes from MEMS

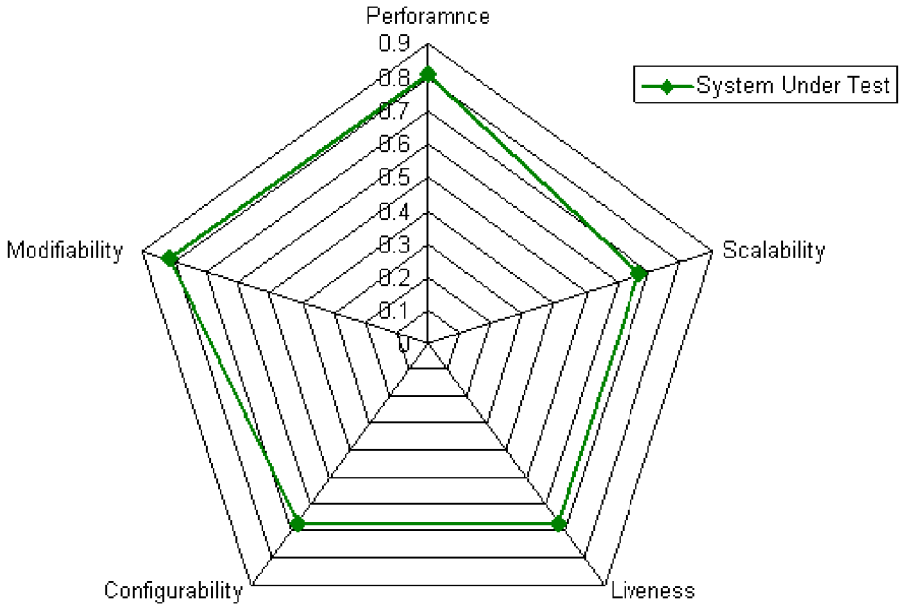

Figure 9: Final Evaluation Result

provide insights on the design and development of mission critical systems to satisfy high quality goals on performance, scalability, liveness, modifiability and configurability. MEMS does not make the decision for the stakeholders, it does serves as a tool to inform architects about quality attributes and help to structure the inquiry to identify the defects.

8 Acknowledgement

National ICT Australia is funded through the Australia Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

[1] M. A. Babar and I. Gorton. Comparison of scenario-based software architecture evaluation methods. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 600–607, Washington, DC, USA, 2004. IEEE Computer Society.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] L. Bass, R. Nord, W. Wood, and D. Zubrow. Risk themes discovered through architecture evaluations. In *WICSA '07: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[4] L. J. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3):297–334, 9 1951.

[5] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, 28(7):638–653, 2002.

[6] K. Foster, A. Iannos, G. Lawrie, P. Templ, and B. Tobin. Exploring a net centric architecture using the net warrior airborne early warning and control node. Technical Report - public release DSTO-TR-2093, Defence Science and Technology Organisation, December 2007.

[7] I. Gorton, A. Liu, and P. Brebner. Rigorous evaluation of cots middleware technology. *Computer*, 36(3):50–55, 2003.

[8] J. Kontio. A case study in applying a systematic method for cots selection. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 201–209, Washington, DC, USA, 1996. IEEE Computer Society.

[9] A. Liu and I. Gorton. Accelerating cots middleware acquisition: The i-mate process. *IEEE Softw.*, 20(2):72–79, 2003.

[10] Y. Liu, I. Gorton, L. Bass, C. Hoang, and S. Abanmi. Mems: A method for evaluating middleware architectures. In *Intl. Conf. on Quality of Software Architecture (QoSA)*, pages 9–26. IEEE Computer Society, 2006.

[11] I. Ozkaya, L. Bass, R. S. Sangwan, and R. L. Nord. Making practical use of quality attribute information. *IEEE Softw.*, 25(2):25–33, 2008.

[12] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.