Model Integrated Test and Benchmarking suite for Component Middleware Implementations

Arvind S. Krishna
arvindk@dre.vanderbilt.edu
ISIS
Vanderbilt University, TN 37203,USA

Abstract

This documents presents the high level interpreter design of a model-integrated benchmarking suite, CCMPerf, that can be used to benchmark component middleware implementations. The interpreter has been designed to consists of code generation engines, each of which generates a given file-format, for example, IDL files. The structure of the various files that will be generated as a part of CCMPerf will be explained. Sample generated code snippets will be provided to illustrate structure of code snippets. The generated code will run on standard operating systems such as Widows, Linux, and Unixes. The generated, code, however, is targeted towards CIAO, an open-source implementation of CCM.

1 Introduction

Distributed real-time and embedded (DRE) systems are increasingly becoming widespread and important. Common DRE systems include telecommunication networks (*e.g.*, wireless phone services), tele-medicine (*e.g.*, robotic surgery), and defense applications (*e.g.*, total ship computing environments). DRE systems are increasingly used for a wide range of applications where multiple systems are interconnected using wireless and wireline networks to form system of systems. Such systems possess stringent quality of service (QoS) constraints, such as bandwidth, latency, jitter and dependability requirements. A challenge requirement for these new and planned DRE systems therefore involves supporting a diverse set of QoS properties, such as predictable latency/jitter, throughput guarantees, scalability, 24x7 availability, dependability, and security, that must be satisfied simultaneously in real-time. Conventional distributed object computing (DOC) middleware frameworks, such as DCOM and Java RMI, do not provide capabilities for developers and end-users to specify and enforce these QoS requirements simultaneously in complex DRE systems.

Component middleware [1] is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. The CORBA Component Model (CCM) [2] is a standard component middleware technology that addresses limitations with earlier versions of CORBA middleware based on the DOC model. The CCM specification extends the CORBA object model to support the concept of components and establishes standards for implementing, packaging, assembling, and deploying component implementations.

Component middleware in general – and CCM in particular – are a maturing technology base that represents a paradigm shift in the way DRE systems are developed. Several implementations of CCM are now available, including (1) CIAO (Component Integrated ACE ORB) [3], (2) MICO-CCM [4] (MICO's CCM implementation), and (3) Qedo [5].

2 Design of CCMPerf – Meta Model Synopsis

2.1 Motivation

In this section we describe the design of sf CCMPerf [6], an open-source benchmarking suite integrated with CoSMIC that enables the component developer to model component assemblies and synthesize benchmarking experiments. The use of CCMPerf enables creation of experiments that measure black box, *e.g.*, latency, throughput metrics that can be used to know the consequences of mixing and matching component assemblies on a target environment. CCMPerf, can also be used to synthesize experiments on a per component basis, the motivation being to generate unit and regression tests.

A model based approach to benchmarking allows the modeler to generate tests at the push of button. Without modeling techniques, these tedious and error-prone code would have to be written by hand. In a hand crafted approach, changing the configuration would entail re-writing the benchmarking code. In a model based solution, however, the only change will be in the model and the necessary experimentation code will be automatically generated. A model based solution also provides the right abstraction to visualize and analyze the EXperiment rather than looking at the source code. In the ensuing paragraphs we describe the design of CCMPerf.

2.2 Experimentation Categories:

The experiments in CCMPerf can be divided into the following three experimentation categories:

- *Distribution middleware* tests that quantify the performance of CCM-based applications using black box techniques
- *Domain-specific middleware* tests that quantify the suitability of CCM implementations to meet the QoS requirements of a particular DRE application domain, such as static linking and deployment of components in the Boeing Bold Stroke avionics mission computing architecture [7].

2.3 CCMPerf Meta Model

The modeling paradigm of CCMPerf is defined in a manner that will allow its integration with other paradigms, *e.g.*, COMPASS and Component Assembly and Deployment Modeling Language (CADML) []. To achieve the aforementioned goal, CCMPerf defines *Aspects*, *i.e.*, visualization of the meta model that allows the modeler to model component interconnection and metrics captured through the above interaction. The following three aspects are defined in CCMPerf

- *Configuration Aspect*, that defines the interface that are provided and required by the individual component,
- *Metric Aspect*, that defines the metric captured in the benchmark, and
- *Inter-connection Aspect*, that defines how the components will interact in the particular benchmarking experiment.

3 Design of CCMPerf Interpreter

From the meta-model, an interpreter generates the necessary files to configure the component assemblies and run the experiment. This section describes each of the files and their sample formats.

3.1 Code-generation an overview

In particular the interpreter should generate the following files:

- *Component-executor code*, header and source file for the executor logic that will be run as a part of the experiment,
- *IDL code*, this will be the IDL interface code that will illustrate the contract, *i.e.*, interface, exchanged between the client and the server, and
- *script files*, these will be perl scripts that will allow the experiment to be run on various platforms.

Additionally, the experiment, will require descriptor files that provide meta-data to configue component servers and daemons including:

- *CORBA assembly descriptor (.cad)*, which is a meta-information file with details of an assembly archive,
- *CORBA component descriptor (.ccd)*, which is a meta-information file that describes the features of a single component, and
- *CORBA software descriptor (.csd)*, which is a compressed file that contains one or more implementations of a component or an interface.

CCMPerf's interpreter would only need to decorate the aforementioned files, *i.e.*, add configuration specific information. An example of such an information is collocation information that indicates if components are in the same host or are remote. This meta-data is also required by other stages in the CosMIC tool chain, for example, the Packaging phase and will be generated by COMPASS[1]. This CCMPerf will not generate the meta-information. Additionally, the configuration files will not be available until the project will be completed. To balance the above two forces, CCMPerf will use CADML's descriptor generator to produce the required meta-data information. This scheme is compatible with CIAO, as CIAO's current implementaion is based on XML descriptors generated by CADML. The difference in CADML and COMPASS generated descriptor is that the former is based on XML DTDs while the latter is based on schemas.

In the following sections, we describe the design of the various code generation engines in CCMPerf.

3.2 CCMPerf code generation engines

CCMPerf consists of three genration engines that traverse the model. All three concrete engines have the same interface as `Codegenerator`. Each of the engines are associated with a corresponding aspect in the meta-model. Below we describe each of the engines in details

Benchmark code engine This code generator generates the necessary component code *i.e.*, executor code, both client and server, that will be run as part of the experiment. The excutor code will include a header file and a source file (.cpp file). This engine will be associated with the the *Metrics aspect*, *i.e.*, place where the end-user specifies what metric will be associated with the experiment. Given below is a sample .ccp file that will be generated by this engine. The code snippet illustrates, the code structure for round-trip latency measured at the client side.

IDL Engine This engine will generate the required IDL files that represent the contract between the client and the server and component. This engine is associated with the *Configuration Aspect* in the meta model. The IDL code generated will be compiled by the Component IDL compile bundeled within the CIAO ORB implementaion to futher generate the "glue-code" or plumbing code that will provide the infrastructure to excute the components. The format of these

[1]COMPASS is being developed by Krishnakumar Balasubramanian ¡kitty@dre.vanderbilt.edu¿ as part of the MIC course.

files is beyond the scope of this document. The IDL files illustrate the client and server IDLs. These include the required and provide interface for the components. The IDL files generated can be categorized into:

- CORBA 2.0 IDL, that describe the interface, the operations associated with the components
- Component IDL, that describes the structure of the components

Script engine This engine generates the script files that will be used to run the experiment. Generation of this file plays an important role as running component based implementation also necessitates manging several entities, *e.g.*, CIAO Daemon that starts Component Servers, Component Servers that host the homes of components and homes that manage the life cycles of components. These entities have to be started and stopped in a orderly manner to ensure proper functioning of the system as a whole. This engine will be associated with the *inter-connection* aspect in the metamodel.

References

[1] C. Szyperski, *Component Software—Beyond Object-Oriented Programming*. Santa Fe, NM: Addison-Wesley, 1998.

[2] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

[3] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, pp. 45–54, mar 2003.

[4] M. is CORBA, "The mico corba component project." `http://www.fpx.de/MicoCCM/`, 2000.

[5] Qedo, "Qos enabled distributed objects." `http://qedo.berlios.de`, 2002.

[6] A. S. Krishna, J. Balasubramanian, A. Gokhale, D. C. Schmidt, D. Sevilla, and G. Thaker, "Epirically Evaluating CORBA Component Model Implementations," in *Proceedings of the OOPSLA 2003 Workshop on Middleware Benchmarking*, (Anaheim, CA), ACM, Oct. 2003.

[7] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.