Interaction Components Between Components based on a Middleware

Văn Cam Phạm, Önder Gürcan, Ansgar Radermacher

CEA, LIST, Laboratory of Model driven engineering for embedded systems,
Point Courrier 174, Gif-sur-Yvette, F-91191 France
name.surname@cea.fr

Abstract. One of the problems of systems based on distributed architectures is the communication between applications running on different platforms on a network. The appearance of middleware reduces the complexity in transferring data between heterogeneous platforms of such systems. Up until now, various middleware have been proposed to facilitate the distributed system construction. In the context of component-based development, connectors represent links that realize the communication between application components. However, from the modeling perspective, the transition from the behavior of connectors to middleware implementation is still not clear.
This paper reports how to model the interaction components that define the behavior of connectors by using the ZeroMQ middleware due to several advantages it offers such as effective asynchronous communication patterns. In order to test our approach, we designed and implemented several different examples. Based on these examples, we observed that implementing interaction components between components based on a middleware simplifies the connection between components in a distributed system.

1 Introduction

A distributed system consists of multiple different *application components* that connect together to exchange data. These components usually run on heterogeneous platforms and thus have to handle platform differences such as byte-order. In model-driven approaches, this problem is often tackled by abstracting the communication logic from its implementation. In the UML specification, *connectors* illustrate such abstract communication links between the application components. However, the UML specification does not define the behavior of connectors. Therefore, an additional refinement is required on the model level.

On the implementation level, it is possible to integrate the connection code into application components directly. In other words, the connection code is integrated into the application components. Nevertheless, the management of application components becomes more difficult as their number increases and the embedded connection code cannot be reused. It is therefore necessary to

separate interaction components[1] from application components; hence developers can focus on application components without taking the communication into account.

In case of heterogeneous platforms, the implementation of connections needs to take several issues into account, notably different conventions for the ordering of bytes within a word[2]. In addition, it is also difficult to directly manage complicated connections from the application using socket connections since many sockets need to be created. Middleware is a way to overcome such difficulties since it offers a higher level of abstraction and does not depend on the underlying operating system.

The presented paper is based on previous work in this area, notably the support of connectors [9] for the UML profile MARTE and the support of simple socket interactions in [10]. The Qompass designer tool chain has been developed in the context of this work. It is a code generation and deployment extension of the UML modeler Papyrus[3]. The novelties of this paper are (1) the presentation of an additional interaction component based on the ZeroMQ middleware and (2) the support of asynchronous requests with return values (also called deferred synchronous calls).

The remaining of this paper is organized as follows. Section 2 outlines the methods and tools. Section 3 presents the modeling of ZeroMQ interaction components. Section 4 shows examples to test our implementation. Section 5 gives the related work and Section 6 concludes the paper.

2 Background

In this section, we introduce the method and tools used for our study, in particular Qompass Designer. It is used to transform models and deploy an application. Besides a model of the application software, the input model consists of a library of interaction components (and container services), a platform description and a deployment description that declares, configures and allocates instances. In the context of this paper, we only focus on application and interaction components. The component model is enriched by means of the Flex-eware Component Model (FCM) profile. It provides (among other extensions) a means to enrich ports of components. An FCM port has an additional port kind (extensible) that denotes whether the port is for instance a client/server, data-flow or event port. From an implementation perspective, the port kind determines the required and provided interfaces of this port.

There are basically two main steps for using interaction components[4] (1) transforming the UML application model into an intermediate model, and (2)

[1] As a common terminology, components that implement a UML connector are called *interaction components*.

[2] Orderding of bytes, http://www.gnu.org/software/libc/manual/html_node/Byte-Order.html, accessed on 07/07/2014.

[3] Papyrus, `http://www.eclipse.org/papyrus/`, accessed on 17/07/2014.

[4] It is basically a UML component (class) tagged as interaction component.

generating the implementation code from the intermediate model. The first transformation step replaces UML connectors with interaction components, as detailed later (see Fig. 2).

From the perspective of a developer who wants to incorporate new interaction components, a preliminary step is the modeling of this interaction component. This is done in form of a stereotyped class. The interaction component has ports to connect to the ports of application components. To be able to allocate these ports on different platforms, interaction components are logically decomposed into several fragments [10] (fragment per node). For example, a uni-directional communication interaction component has a sending fragment and a receiving fragment. These logically connected fragments are physically connected by using programming languages such as C++, Java in the implementation level. In this work, a new interaction component on top of the ZeroMQ (also known as ZMQ) middleware is developed[5] since we want to apply the AMI callback pattern and ZeroMQ offers a set of asynchronous socket APIs that transfers messages quickly and efficiently over the network. These sockets run on top of the standard sockets of operating systems and carry atomic messages across various transports such as in-process, inter-process, TCP, and multicast. The modeling of the interaction component is detailed in the next section.

In this study, we focus on *asynchronous method invocation (AMI) callback communication pattern* [11] since it allows clients to achieve high performance. For example, in a client/server application, a client sends a request to a server. Instead of blocking and waiting for a reply from the server (as synchronous calls), it provides callback functions to be invoked in order to process results received. These callback functions are called once replies are received. In the sense of component-based development, we use ports dedicated to the AMI callback pattern that are used by applying the AMI callback element of the FCM profile (see Fig. 1).

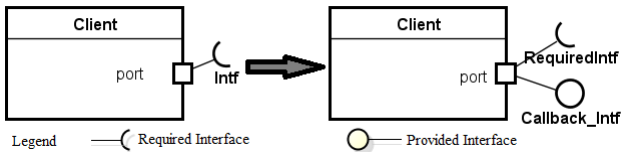

Fig. 1: The AMI port has two interfaces (right), one required and one provided, derived from a original port interface (left). The provided interface is needed since it contains callback functions that are invoked through the AMI callback port.

During application deployment, the modeled UML connectors are transformed into interaction components in an intermediate model (Fig. 2) by using Qompass Designer. The FCM *Connector* stereotype references the interaction

[5] ZeroMQ, `http://zeromq.org/`, accessed on 18/07/2014.

component that should be used. The transformation adapts the interaction component automatically to the application components that are connected, e.g. the ports of the interaction component need to be compatible with the ports of the connected application components. The ports of application components then connect to the ports of the generated interaction components instead of the end points of the UML connectors.

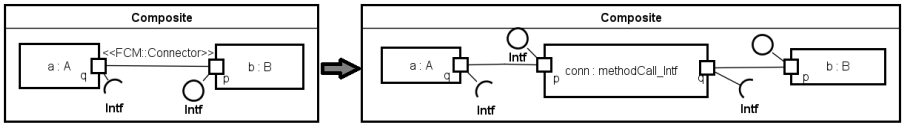

Fig. 2: Transformation from a system with (a) line of connector to (b) a composite structure of connector

The implementation code is generated from the intermediate model (a UML2 model with expanded interaction components). The code generator is basically generating C++ code from the UML model.

3 Interaction components modeling based on ZeroMQ

In this section, we present the decomposition of connectors and how AMI callback ports are used for modeling the asynchronous communication pattern. The AMI ports are dedicated for asynchronous requesting components such as clients in Client/Server applications.

This interaction component (see Fig. 3) contains fragments that define the behavior of connectors, provides interfaces to connect to application components through its ports and are co-located with appropriate application components on specific nodes of platforms. Interaction components often have two ports to connect two application components, but the concept is not limited to this case.

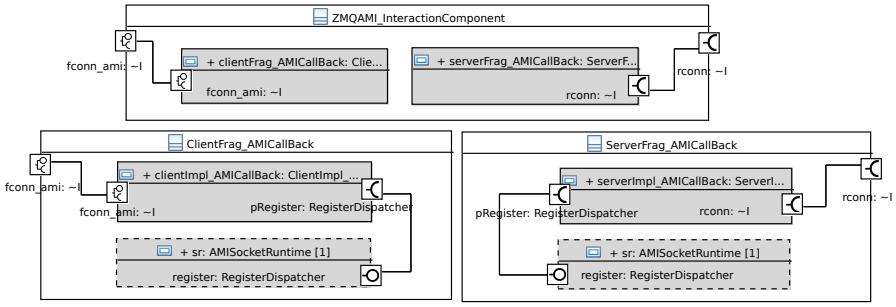

Fig. 3: Interaction component composite for AMI Callback model

The client fragment (ClientFrag_AMICallback) is asynchronous and the server fragment (ServerFrag_AMICallback) is synchronous. The interaction component needs to be reusable in other applications; hence the interfaces of the ports of the interaction component must match with the interfaces of different application components. In other words, when the interfaces of a port change, the interaction component has to adapt with the new interfaces.

To do this, we use an interface I as a formal parameter in a template and the ports of the interaction component are typed with this template. I is then bound to a specific interface when it is in use. The template binding defined here is realized by model to model transformations in Qompass Designer.

The inside of each of these two fragments is divided into two parts to differentiate between dispatching (xImpl) and communication (SocketRuntime) tasks. For the client and the server, there are ClientImpl and ServerImpl respectively that *dispatch* the requests or callbacks to right addresses. SocketRuntime, on the other hand, permits the dispatching component to register the dispatch interface (RegisterDispatcher) to the corresponding port (pRegister). RegisterDispatcher is called when the SocketRuntime receives some data. To realize this mechanism, SocketRuntime uses a set of ZeroMQ sockets to connect to the application components.

When a requesting component (e.g., client) calls a function through the AMI method invocation, the in/inout parameters of the function are marshalled into a chain of bytes. These parameters are stored in a buffer of the interaction component, ClientImpl in particular. These parameters are then passed as the parameters of the callbacks. This storage is essential to distinguish callbacks from multiple invocations since different callbacks corresponding to different input parameters may process results received in different ways. The parameters marshaling is generated from an Acceleo template. An example of an interface with two operations (int sum(int a, int b) and int square(int value)) is shown in Fig. 4. The chain of bytes also includes an operation ID and a handler ID. The operation ID is used by the server to determine the right processing function and the hander ID to find again the input parameters saved corresponding to the right results received. The callbacks therefore execute with its results and input parameters. The called function returns immediately after saving its parameters. The requesting component can go ahead without waiting for results. Data are actually sent and received in background threads. The socket runtime at the server side receives the request, calls dispatch to de-marshal parameters and then execute the right function to get the result. The de-marshaling of parameters is also generated from Acceleo.

The maximum number of input data has to be configured by users. For network applications with high calls number density or high computational time on servers, this number should be large enough to prevent the data of previous requests from overwriting. ClientFragment (sender) has a DEALER[6] socket of ZeroMQ to send requests and a ROUTER socket to asynchronously receive replies

[6] See the ZeroMQ web site for more information.

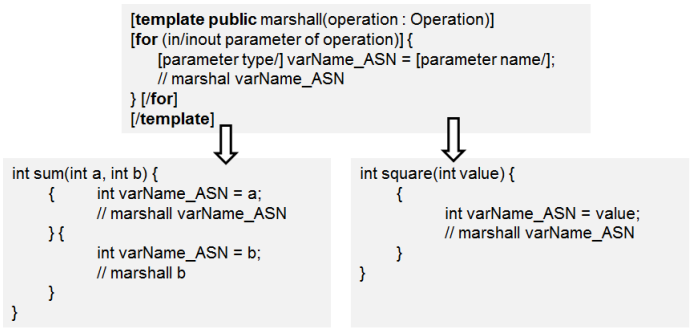

Fig. 4: Transformation from Acceleo to C++ code

from ServerFragment (recipient). The DEALER socket connects to a ROUTER socket of the recipient. These sockets offer asynchronous data transfers.

4 Examples

In this section, we present two examples to testify our interaction component implementation. The first example is about a client/server application. The second one is about a simple load balancing application.

4.1 Client/Server application using AMI callback

This system consists of a client and a server. The client is asynchronous and requests to the server through AMI callback communication with the interface ICompute as shown in Fig. 5. The interface has two operations: add(in a:Long, in b:Long):Long and mult(in a:Long, in b:Long):Long. The client needs to initiate requests. For this need, the FCM provides a simple convention: the client possesses a port start providing the interface IStart. This interface contains a run method that is automatically called pending the system start-up. The connector between the client and the server refers the interaction component implemented.

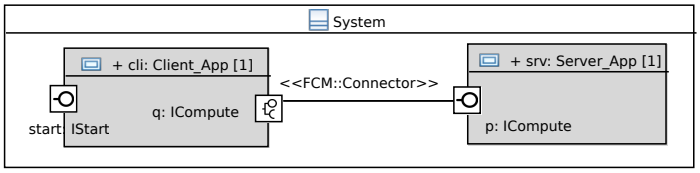

Fig. 5: Client/Server using AMI Callback example

For the deployment, the system is distributed on two different nodes. The client is deployed on ClientNode, the server on ServerNode as exposed in Fig. 6.

The fragments of the connector are co-located with the application components. The model transformed by Qompass Designer is then the input of the code generation process. This process is realized by using Acceleo[7].

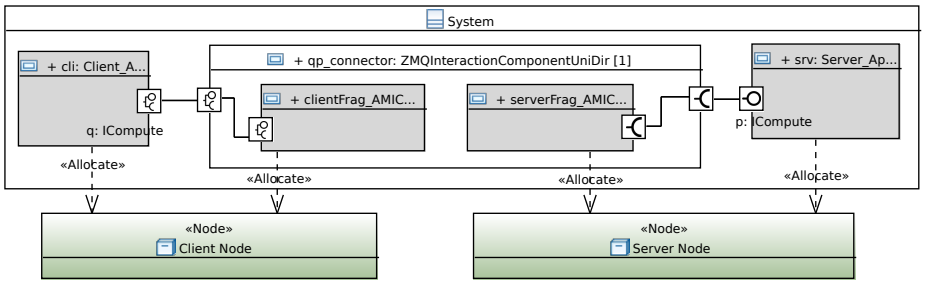

Fig. 6: Client/Server AMI callback example deployment

4.2 Interactions between components in the load balancing model

The Client/Server model is widely used because of its simplicity and facility of implementation. However, the model presents some issues, i.e. it is difficult to scale since the server must always run or the server can be a bottleneck since it has to treat all requests. Load balancing model[8] has been proposed as a solution to overcome these issues.

Load balancing is offered by ZeroMQ for distributing workloads of an application onto several servers called workers. Workloads distribution is performed by a broker component. The workers have the computational responsibility. They expedite the result to the broker.

In this example, there are one client, one broker and one worker (on the type level). The client needs to implement call back functions. AMI callback port kind is used. The ZMQAMI_InteractionComponent interaction component is applied to the connectors between components. The workers act synchronously. Information about the address and listening ports of the broker is configured. Clients need to know the front end port number and the broker's IP address and workers know about the back end's.

The application components of the system are allocated onto three nodes, client node, worker node, broker node. Many instances of client and worker can be run in different platforms. The broker has to start firstly and listen on the worker side (back end). When a worker begins, it sends a ready signal to the broker and the broker sets it as an available worker. The broker only actives on the client (front end) side if there is one available worker at least. Requests are forwarded from the broker and arrive to the workers alternatively.

[7] Acceleo, `http://www.eclipse.org/acceleo/`, accessed on 17/07/2014.

[8] Load Balanced Cluster, `http://msdn.microsoft.com/en-us/library/ff648960.aspx`, accessed on 12/09/2014.

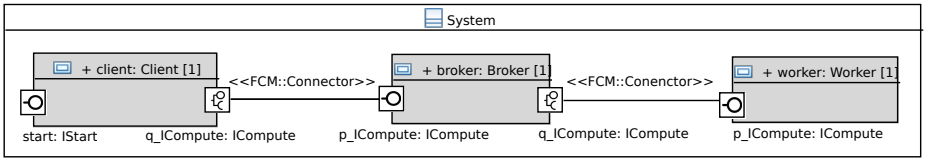

Fig. 7: Simple application follows load balancing model

5 Related Work

The concept of interaction components presented in this paper is supported by multiple component models and tools in other terminologies. The following sketches several works that are categorized into AMI callback implementations, component models supporting interaction components and implementations based on ZeroMQ.

Arulanthu et al. [1] provide the implementation of AMI callback for CORBA. Their implementation is in TAO [11]. They use the IDL (interface definition language) compiler to generate callbacks from the original interface. However, this does not resolve asynchronous messaging in MDE. At a higher level of abstraction, asynchronous messaging has been integrated into the CORBA component model (CCM), called AMI4CCM [7]. An AMI4CCM connector (analogous to the interaction component described in this paper) is responsible for managing the interaction. The connector is part of the extensibility mechanism in CCM, providing a so-called generic interaction support (GIS). The major difference between AMI4CCM and the work presented here is to address the concepts directly at the modeling level and the support for the middleware ZeroMQ. Please note that Qompass designer is inspired by CCM and supports similar concepts.

The interest for a further standardization of component models with extensible interaction support is expressed by a request-for-proposal of a Unified Component Model (UCM) [8] that the Object Management Group (OMG) has issued recently. In the sequel we reference two older component models with this ability, before we talk about a different approach to build higher level services on top of ZeroMQ: ZeroRPC.

SOFA 2[9] [5, 6, 4] is a component system employing hierarchically composed components. SOFA connectors are automatically generated. A connector might support a transport mechanism such as CORBA or low level mechanisms. In this context, they are responsible for marshaling and de-marshaling. The proposed connector architecture consists of a distributor deployment unit and several sender/recipient units. The sender/recipient unit allows sending messages to attached components. The sender/recipient units connect to the distributor unit in a similar way. Connector configurations and deployment models are also shown. However, SOFA 2 does not support UML.

Fractal is a hierarchical and reflective component model [3]. It is intended to implement, deploy, and manage complex software systems, including in partic-

[9] SOFA, `http://sofa.ow2.org/`, accessed on 15/09/2014.

ular operation systems and middleware. Fractal connectors are Fractal binding components with behavior [2].A composite binding component is a communication path between an arbitrary numbers of component interfaces, of arbitrary language types. These bindings are represented as a set of primitive bindings and binding components (stubs, skeletons, adapters in the con-text of remote method calls).

ZeroRPC[10] is a light-weight, reliable and modern communication library for distributed systems. ZeroRPC builds on top of ZeroMQ and MessagePack. ZeroRPC is more than a typical Remote Procedure Call (RPC) engine and supports multiple ZeroMQ socket types, streaming, heartbeat and more. ZeroRPC is created to satisfy requirements such as exposing arbitrary code with minimal modification, self-document systems, propagate exceptions, trace nested calls and provide brokerless, highly available, fast fan-in/fan-out. However, ZeroRPC focuses on communications between server-side processes and so far is only implemented in Python and Node.js that are not suitable to distributed embedded applications

6 Conclusion and Future Work

In this paper, we have shown the modeling in UML of the AMI interaction component that defines the behavior of connectors. We used the stereotypes of the FCM profile to apply UML connectors and ports for the modeling. A UML connector applying the *Connector* stereotype of the FCM profile is transformed to a composite structure. We used Papyrus to model and Qompass Designer to transform models. At the physical connection level, we used the ZeroMQ middleware due to the several advantages it offers.

After the modeling of the interaction component, we tested it with two examples. One is a simple Client/Server application with asynchronous client and synchronous server; the other one is a simple load balancing application[11]. The separation between interaction and application components simplifies the development process of distributed systems. The interaction component can be reused in other applications. Application components developers can therefore focus on data processing at application level.

As future work, we will enrich the properties of quality of service for the interaction components to provide more reliable communications. We will also further study systems with dynamic adaptation which are currently poorly supported by our approach.

The work presented in this paper is supported by the European project SafeAdapt, grant agreement No. 608945, see `http://www.SafeAdapt.eu`.

[10] ZeroRPC, `http://zerorpc.dotcloud.com/`, accessed on 15/09/2014.

[11] We also support publisher/subscriber and producer/consumer patterns, but we are unable to present them here due to space limitation.

References

[1] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The design and performance of a scalable orb architecture for cobra asynchronous messaging. In IFIP/ACM International Conference on Distributed Systems Platforms, Middleware '00, pages 208–230, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[2] Tomás Barros, Rabea Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural Models for Distributed Fractal Components. Rapport de recherche RR-6491, INRIA, 2008.

[3] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, February 2004. Version 2.0-3.

[4] Lubomir Bulej and Tomas Bures. Using connectors for deployment of heterogeneous applications in the context of omg d&c specification. In Dimitri Konstantas, Jean-Paul Bourrières, Michel Léonard, and Nacer Boudjlida, editors, Interoperability of Enterprise Software and Applications, pages 349–360. Springer London, 2006.

[5] Tomas Bures and Frantisek Plasil. Communication style driven connector configurations. In LNCS3026, ISBN 3-540-21975-7, ISSN 0302-9743, pages 102–116. Springer-Verlag, 2004.

[6] Ondrej Galik and Tomás Bures. Generating connectors for heterogeneous deployment. In Elisabetta Di Nitto and Amy L. Murphy, editors, Proceedings of the 5th International Workshop on Software Engineering and Middleware, SEM 2005, Lisbon, Portugal, September 5-6, 2005, pages 54–61. ACM, 2005.

[7] Object Management Group. Asynchronous method invocation for ccm. Specification Version 1.0, Object Management Group, April 2013.

[8] OMG. OMG Unified Component Model for Distributed, Real-Time and Embedded Systems. Request for proposal, OMG, May 2014. http://www.omgwiki.org/ucm/doku.php.

[9] Ansgar Radermacher, Arnaud Cuccuru, Sebastien Gerard, and François Terrier. Generating Execution Infrastructures for Component-oriented Specifications with a Model Driven Toolchain: A Case Study for MARTE's GCM and Real-time Annotations. SIGPLAN Not., 45(2):127–136, October 2009.

[10] Ansgar Radermacher, Önder Gürcan, Arnaud Cuccuru, Sebastien Gerard, and Brahim Hamid. Split of composite components for distributed applications. In Torsten Maehne and Marie-Minerve Louërat (eds), editors, Languages, Design Methods, and Tools for Electronic System Design, chapter 14, pages 255–267. Springer, Septembre 2014. doi:10.1007/978-3-319-06317-1_14.

[11] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design of the TAO Real-time Object Request Broker. Computer Communications, 21(4):294–324, April 1998.