Interaction Components Between Components based on a Middleware

van Cam Pham, Ansgar Radermacher, Önder Gürcan

Interaction Components Between Components based on a Middleware

Van Cam Pham, 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 and helps raise organizational efficiency. Up until now, various middleware have been proposed to facilitate the distributed system construction. However, from the modeling perspective, the transition from interaction components to middleware implementation is still not clear. This paper reports how to model the interaction components by using ZeroMQ middleware due to the several advantages it offers. In order to test our approach, we designed and implemented several different case studies. 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, *UML connectors* illustrate such abstract communication links between the application components. However, the UML specification does not define the behavior of connectors. One solution is to integrate the implementation (i.e. interaction components[1]) into application components directly. In other words, the interaction component is a part of the application component that processes data. Nevertheless, the management of application components becomes more difficult as their number increases and also their corresponding interaction components cannot be reused by other applications. It is therefore necessary to separate interaction components from application components; hence developers can focus on application components without taking into account the communication.

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

Based on this observation, the aim of this paper is *to define the behavior of connectors in distributed systems where application components are allocated onto heterogeneous platforms.* The presented paper is based on previous work in this area, notably the support of connectors [8] for the UML profile MARTE and the support of simple socket interactions in [9]. Thus, in the modeling level, it is necessary to first model the distributed system based on UML. This model has to be transformed into an intermediate model in order to transform UML connectors to interactions components. After this, the implementation can be generated from the intermediate model. Since we are dealing with distributed applications, these interaction components then need to be split into pieces called *fragments* that are co-located with the applications components that they connect.

In case of heterogeneous platforms, the implementation of connections between fragments of an interaction component 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.

In order to realize this approach, we model the interaction component for the asynchronous method invocation (AMI) pattern since this pattern allows clients to achieve high performance and then we use this interaction component in different case studies to testify it. For modeling, the Papyrus [6] UML editor is used. For model to model transformations, we use the Qompass Designer of the Papyrus project since it allows the usage of the Flex-eware Component Model (FCM) profile [8] that provides stereotypes to model interaction components. For the implementation level, we use the ZeroMQ middleware to connect fragments since it offers powerful socket connections. The use of AMI and ZeroMQ is a primary novelty compared to [9].

The remaining of this paper is organized as follows. Section 2 gives the background and Section 3 presents interaction components modeling by means of ZeroMQ. Section 4 shows case studies to test our implementation. Section 5 gives the related work and the paper is concluded by Section 6.

2 Background

In this section, we introduce the methods and tools used for our study. There are basically three main steps for realizing interaction components[3] (1) modeling the interaction component based on UML, (2) transforming the UML model into an intermediate model, and (3) generating the implementation code from the intermediate model.

The first step is modeling the interaction component based on the UML profile. We use Qompass Designer that is coming as an extension of the Pa-

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

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

pyrus modeling tool[4]. Papyrus [6] is a modeling tool that aims at providing an integrated and user-friendly environment for editing UML models and related modeling languages such as SysML[5] and MARTE[6]. Qompass Designer is its dedicated extension for code generation and deployment.

For representing interaction components, we use the Flex-eware Component Model (FCM) profile that extends UML composite structures by enriching ports of components. An FCM port has a type and a port kind that determine the required and provided interfaces of this port. To model an interaction component, we need to define a UML connector that applies the *Connector* stereotype of the FCM profile. This connector 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 [9] (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 study, we focus on *asynchronous method invocation (AMI) callback communication pattern* [10] 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 Figure 1).

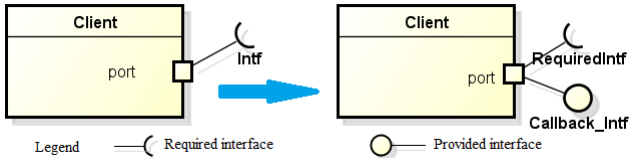

Fig. 1. The AMI port has two interfaces (right), one required and one provided, derived from a original port interface (left). The provided interface is needed since it contains callback functions that are invoked through the AMI callback port.

As a second step, the modeled UML connectors (that apply the FCM *Connector* stereotype) are transformed into interaction components in an intermediate model (Figure 2) by using Qompass Designer. This is necessary since the

[4] Papyrus, `http://www.eclipse.org/papyrus/`, accessed on 17/07/2014.
[5] Systems Modeling Language (SysML), `http://www.sysml.org/`, accessed on 18/07/2014.
[6] The UML Profile for MARTE, `http://www.omgmarte.org/`, accessed on 18/07/2014.

UML connector is represented by a graphic conductor element between application components, but it is not a classifier. In the intermediate model, the UML connectors applying the FCM *Connector* stereotype is replaced by interaction components that are represented as UML components[7]. The ports of application components then connect to the ports of the generated interaction components instead of the end points of the UML connectors.

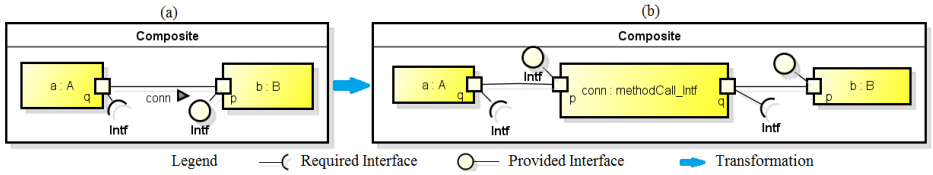

Fig. 2. Transformation from a system with (a) line of connector to (b) a composite structure of connector

Lastly, the implementation code is generated from the intermediate model. The code generator is basically generating C++ code from UML model by taking into account the libraries provided by ZeroMQ[8] (also known as zmq) middleware 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.

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. As mentioned before, an AMI callback port kind dedicated to ports is defined by the Qompass Designer. This port kind is dedicated to asynchronous requesting components such as clients in Client/Server applications.

An interaction component is defined as a UML component represented by a UML class. This interaction component contains fragments that define the behavior of connectors and provides interfaces to connect to application components through its ports. The interaction component often needs to have two ports to connect to two end points of a connector.

Figure 3 illustrates an example of interaction component dedicated to the AMI callback communication pattern (ZMQAMI_InteractionComponent). The

[7] We cannot model UML components for representing interaction components directly in the upper since it is not possible to reuse them.

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

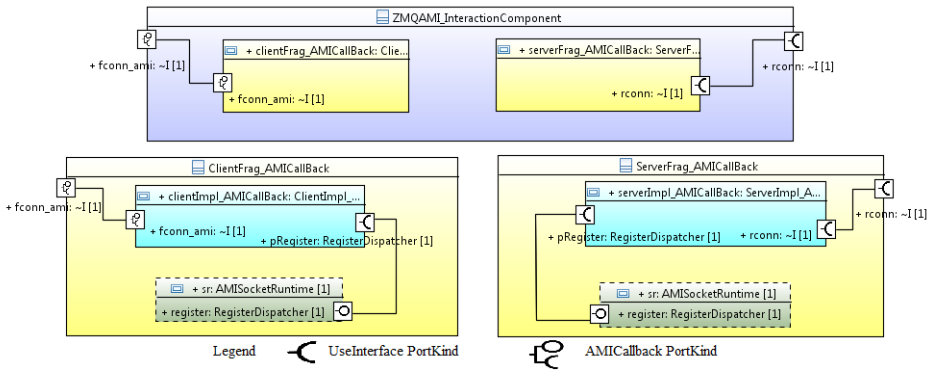

Fig. 3. Interaction component composite for AMI Callback model

client fragment (ClientFrag_AMICallback) is asynchronous and the server fragment (ServerFrag_AMICallback) is synchronous. The ports of corresponding application components must match with ports of these fragments. We want this interaction component 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.

For instance, let's say two application components client1 and client2 have a port with the ICompute1 interface and the ICompute2 interface respectively. In such a case, the interaction component has to carry ICompute1 in the case of client1 and ICompute2 in the case of client2. However, this is not a good solution to change the interfaces of the ports of an interaction component in each use. To handle this problem, one solution can be to define a template interface. 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, i.e., the I template is bound to the ICompute1 interface of a port of client1 when the client uses the connector containing the interaction component. The template binding defined here is realized by model to model transformations in Qompass Designer.

For the connector decomposition, an interaction component contains several fragments that play role of transferring data; hence it has a sending fragment and a receiving fragment at least. As in Figure 3, the fragments of an interaction component for a Client/Server application are ClientFragment and ServerFragment. These fragments will be co-located with appropriate application components on specific nodes of platforms, i.e., they will be co-located with the client on the client node and the server on the server node. 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 is ClientImpl and ServerImpl respectively that *dispatch* the requests or callbacks to right ad-

dresses. 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., client1) calls a function through the AMI method invocation, the 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 oriented to the parameters of the appeals of 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. These parameters are then decoded for calling callback functions. The chain of bytes also include an operation ID and a handler ID. operation ID is used by the server to determine the right service (processing function). hander ID is used to find again the input parameters saved corresponding to the right results received. The callback functions 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 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[9] socket of ZeroMQ to send requests and a ROUTER socket to asynchronously receive replies from ServerFragment (recipient). The DEALER socket connects to a ROUTER socket of the recipient. These sockets offer asynchronous data transfers.

4 Case studies

In this section, we present two case studies to testify our interaction component implementation. The first case study 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 section shows the case study about a distributed client/server system. This system consists of a client and a server. The client requests to the server through AMI callback communication with the interface ICompute. The interface has four operations: add, mult, sumOfArray, and findMax, as shown in Figure 4. 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.

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

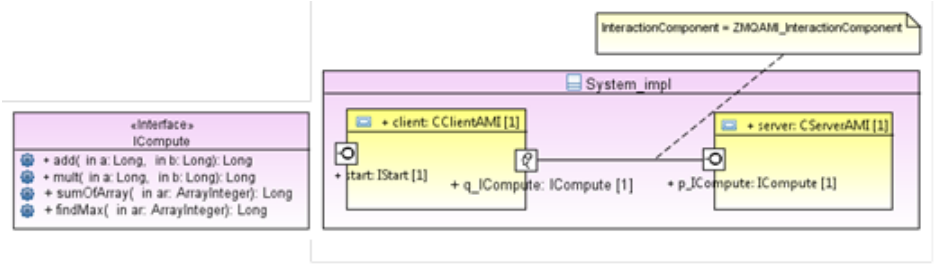

Fig. 4. Client/Server using AMI Call back case study

The client is asynchronous. It applies the port kind AMICallback for its port q_ICompute. The connector between the client and the server uses the interaction component implemented in the previous section.

For the deployment, the system is distributed on two different nodes. The client is deployed on ClientNode, the server on ServerNode as exposed in Figure 5. 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[10].

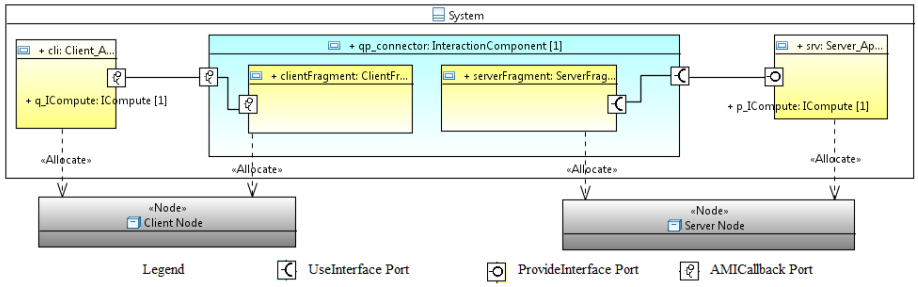

Fig. 5. Client/Server AMI callback example deployment

4.2 Interactions between components in the load balancing model

Client/Server 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 bottle neck since it has to treat all requests. Load balancing model has been proposed as a solution to overcome these issues.

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

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 case study, 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

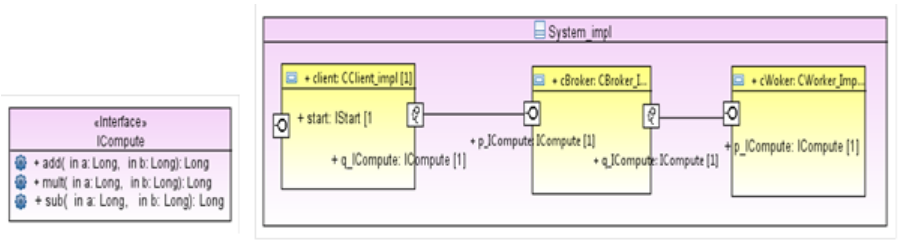

Fig. 6. Simple application follows load balancing model

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 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, i.e. if there are three workers, request 1 to worker 1, request 2 to worker 2, request 3 to worker 3, request 4 to worker 1, and so on.

5 Related Work

In the literature, Arulanthu et al. [1] provide the implementation of asynchronous method invocation model for CORBA. Their implementation is in TAO [10]. They use the IDL compiler to generate callback functions from the original interface. However, this does not resolve the AMI in MDE. The connection code cannot be reused in other applications, i.e., different components use different interfaces, the implementation of each interface needs different connection code. In order to overcome this issue, we create a template interface. This template interface is bound to a specific one in use.

In the literature, Bures et al. [4] also propose the decomposition of connectors. They identify four basic communication style driven connectors. These styles are

procedure call, messaging, streaming and blackboard. Bures et al. also define a set of non-functional properties for each communication type. Connector configurations and deployment models are also shown. 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. This proposal does not however mention the automatic adaptation of connector in use. Moreover, modeling languages are not used. It does not present clearly how parts of a connector are allocated to specific nodes in the deployment model.

Related to the *decomposition of connectors*, Fractal component model has been proposed. Fractal [5] is a hierarchical and reflective component model [3]. It is intended to implement, deploy, and manage complex software systems, including in particular operation systems and middleware [2]. A Fractal component consists of a membrane, which supports interfaces to introspect and reconfigure its internal features, and a content, which consists of a finite set of other components. Fractal also mentions the definition of connector. Connectors are Fractal binding components. A composite binding component is a communication path between an arbitrary number 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 context of remote method calls). However, Fractal does not provide the definition of connector's behavior.

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 sockets due to the several advantages it offers.

After the modeling of the interaction component, in order to test our interaction component, we used it in two case studies. One is a simple Client/Server application with asynchronous client and synchronous server; the other case study is a simple load balancing application[11]. We find that the modeling of interaction components separated from application components simplifies the development process of application components of distributed systems. Moreover, 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. Our purpose is to

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

contribute to the implementation of interaction patterns of Unified Component Model (UCM) [7].

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. E. Bruneton, T. Coupaye, and J.B. Stefani. *The Fractal Component Model*, February 2004. Version 2.0-3.
3. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.
4. 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.
5. Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In *Proceedings of the 2006 Conference on Object-oriented Technology: ECOOP 2006 Workshop Reader*, ECOOP'06, pages 117–129, Berlin, Heidelberg, 2007. Springer-Verlag.
6. Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. 19 papyrus: A uml2 tool for domain-specific language modeling. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, pages 361–368. Springer Berlin Heidelberg, 2010.
7. OMG. Omg unified component model for distributed, real-time and embedded systems. Specification, OMG, May 2014. http://www.omgwiki.org/ucm/doku.php.
8. 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.
9. 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.
10. Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Comput. Commun.*, 21(4):294–324, April 1998.