A LOTOS Framework for Middleware Specification

Nelson Souto Rosa and Paulo Roberto Freire Cunha

Universidade Federal de Pernambuco, Centro de Informática
Recife, Pernambuco
{nsr, prfc}@cin.ufpe.br

Abstract. This paper presents a LOTOS framework for the specification of middleware systems. The framework consists of a library of basic middleware components and some guidelines on how to compose them. The components of the framework facilitate the formal specification of different middleware systems.

1 Introduction

Middleware specifications are not trivial to be understood, as the middleware itself is usually very complex [4]. Firstly, middleware systems have to hide the complexity of underlying network mechanisms from the application. Secondly, the number of services provided by the middleware is increasing, e.g., the CORBA specification contains fourteen services. Finally, in addition to hide communication mechanisms, the middleware also have to hide fails, mobility, changes in the network traffic conditions and so on. On the point of view of application developers, they very often do not know how the middleware really works. On the point of view of middleware developers, the complexity places many challenges that include how to integrate services in a single product [6] or how to satisfy new requirements of emerging applications [Blair 98].

Formal description techniques have been used together middleware in the RM-ODP, in which the trader service is formally specified in E-LOTOS. The Z notation and High Level Petri Nests have been adopted for specifying CORBA services [2][3], the Naming service [5], and the Security service [1]. Most recently, Rosa [8] adopted software architecture principles for structuring LOTOS specifications of middleware systems. Despite the adoption of formal techniques, they focus on specific aspects of middleware systems, i.e., they address either a specific service or a specific middleware model.

The main objective of this paper is to propose a framework that helps to formally describe middleware behaviour in LOTOS by providing a set of basic abstractions. These abstractions are generic in the sense that may be combined in different ways in order to specify several middleware systems. Main in our approach is the fact that the abstractions are defined and organised according to their role in relation to the message request. Hence, instead of adopting the traditional approach of organising middleware systems in layers [9], the proposed abstractions are defined considering their role in the message request. For example, the abstractions are grouped into classes related to storage, communication, dispatching, and mapping of message requests. A

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 136 – 142, 2006.

message request is any message that an application (e.g., client, server, sender, transmitter) sends to another application.

This paper is organised as follows: Section 2 presents a general overview of proposed framework. Section 3 presents how the proposed framework may be used to specify client-server and message-oriented middleware systems. Finally, last section presents an evaluation of the research until now and some future work.

2 LOTOS Specifications of Middleware Components

As mentioned before, the proposed framework consists of a set of abstractions that addresses a number of common functionalities of middleware systems. The framework also defines how these abstractions work together to formalise different middleware models. For example, the abstractions may be combined to produce the specification of a message-oriented middleware, whilst they also may be combined to define a procedural middleware (client-server applications) or a tuple space based middleware.

The whole framework is "message-centric" in the sense that basic elements of the framework are grouped according to how they act on the message. Figure 1 shows a general overview of the proposed approach in which the message is intercepted by both middleware elements on the transmitter and receiver sides. It is worth observing that the message may be either a request in which the transmitter ask for the execution of a task on the receiver side or a simple information between loosely-coupled applications.

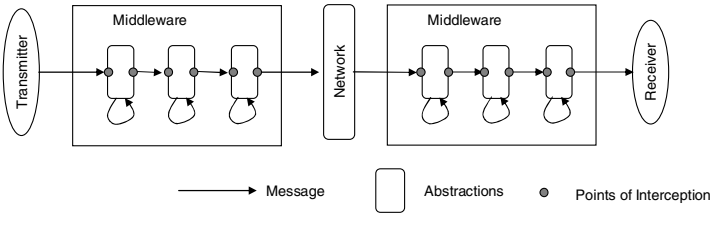
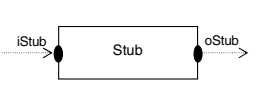


Fig. 1. Message-centric approach

The abstractions of the framework are categorised into four classes: mappers (e.g., stub and skeletons), multiplexers (e.g., dispatcher), communication (e.g., communication channel), and storage (e.g., queue and topic). Whatever the class of the middleware element, it intercepts the message, processes it and forwards the message to the next element. The next element may be a local or remote one. Only communication elements may forward the message to a remote element, i.e., an element only accessible through the network. A non-communication element may need to communicate with a remote element to carry out its task, but it does not send the message itself to a remote element. For example, a transaction service may need to obtain a remote lock before pass the request to the next element of the middleware.

2.1 Basic Abstractions

Mapper elements typically represent remote objects, serve as input points of the middleware, their basic function is to (un)marshal application data (arguments and results) into a common packet-level (e.g., GIIOP request), and are usually found in middleware systems that support request/reply applications in heterogeneous environments. Additionally, non-conventional mappers may also compress data. The specification of a typical mapper, named `Stub`, is defined as shown in Figure 2.

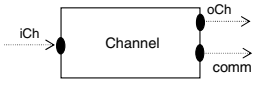



```
(1) process Stub [iStub, oStub] : noexit :=
(2)     iStub ?m : Message;
(3)       oStub !marshalling (m);
(4)         iStub ?m : Message;
(5)           oStub !unmarshalling (m);
(6)             Stub [iStub, oStub]
(7) endproc
```

Fig. 2. Mapper Element

In this specification, the `Stub` receives a message sent by the transmitter and intercepted by the middleware (2), marshals it (3), passes it to the next element (4), and then waits for the reply from the receiver. The reply is also intercepted by the middleware and passed to the `Stub` (4) that takes responsibility of unmarshalling the reply (5).

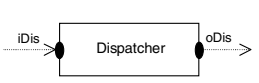
Communication elements get a message and communicate it to a remote element. They act as an interface between the middleware and the operating system. The structure of a communication element, named `Channel`, is shown in Figure 3.




```
(1) process Channel [iCh, oCh, comm] : noexit :=
(2)   Send [iCh, oCh, comm] ||| Receive [iCh, oCh, comm]
(3)  where
(4)     process Send [iCh, oCh, comm] : noexit :=
(5)       iCh ?m : Message;
(6)       comm !m;
(7)       oCh;
(8)       Send [iCh, oCh, comm]
(9)     endproc     …
(10) endproc
```

Fig. 3. Communication Element

In a similar way to `Stub`, the input (`iCh`) and output (`oCh`) ports serves as interception points of the element. However, communication elements have an additional port, named `comm`, used to communicate the message to a remote element. Additionally, the `Channel` is composed by `Send` and `Receive` processes that are responsible to send and receive messages, respectively. In this case, the `Channel` receives the message intercepted by the middleware (5) and then communicates it to a remote element (6).Dispatchers get the request and forward it to the right object (service). The destination object is defined by inspecting the message, in which the destination has been set during the binding. In practical terms, the dispatcher acts as a multiplexer inside the middleware. The general structure of a `Dispatcher` is depicted in Figure 4. The dispatcher receives a message (2) and inspects it, through the function `multiplexer`, to define the destination object (3).




```
(1) process Dispatcher [iDis, oDis] : noexit :=
(2)       iDis ?m : Message;
(3)          oDis !m ! multiplexer(m);
(4)             Dispatcher [iDis, oDis]
(5)    endproc
```

Fig. 4. Dispatcher Element

Finally, storage elements express the need of some middleware systems of store the message prior it to be sent, e.g., for asynchronous communication or to keep a copy of the message for recovery reasons. The general structure of a `Storage` element is shown in Figure 5.

```
(1) process Storage [iSto, oSto] (q: Queue): noexit :=
(2)       hide enq, fst, empt, deq in
(3)          Manager [iSto, oSto, enq, fst, empt, deq]
(4)          |[enq, fst, empt, deq]|
(5)          Queue [enq, fst, empt, deq] (q)
(6)     where
(7)        …
(8) endproc
```

```
process Queue [enq, fst, empt, deq] (q : Queue) : noexit :=
   enq ?n : Nat;
      Queue [enq, fst, empt, deq] (enqueue (q, n))
   [] fst !first (q);
      Queue [enq, fst, empt, deq] (q)
   [] deq;
      Queue [enq, fst, empt, deq] (dequeue (q))
endproc
```

Fig. 5. Storage Element

In this particular element, the storage element (left side) is modelled as a `Queue` that is administered by the `Manager`. It is worth observing that with minor changes to the storage element, it may be defined as a buffer or a file.

2.2 Putting the Basic Abstractions Together

By using the basic abstractions defined in the previous section, middleware systems may be specified by composing them according to the desired distribution model. The general structure of any middleware specified according to the framework is defined as follows:

```
specification TemplateMiddleware [invC,terC,invS,terS,comm] : noexit
     …
behaviour
  (Transmitter[invC,terC]|[invC,terC]|LocalMiddleware[invC,terC, comm])
    |[comm]|
 RemoteMiddleware [invS,terS,comm] |[invS,terS]| Receiver[invS,terS])
      …
endspec
```

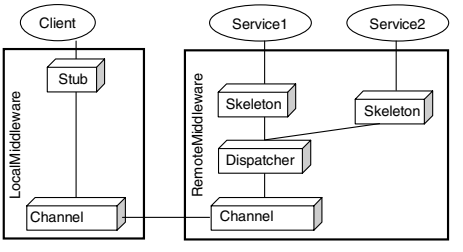
where a `Transmitter` sends a message to the `Receiver` through the middleware, which is made up of a local (`LocalMidleware`) and remote middleware (`RemoteMidleware`) that communicates through the port `comm` (e.g., it may abstract the whole network). Whatever the middleware model, its internal structure is defined as follows (except for the number of components):

```
process Middleware [invC, terC, comm] : noexit :=
   hide iC1, oC1, iC2, oC2 in
        ((C1 [iC1,oC1]  |||  C2  [iC2,oC2,comm])
           |[iC1, oC1, iC2, oC2]|
         Interceptor [invC,terC,iC1,oC1,iC2,oC2])
   where …
endproc
```

The middleware is composed of a set of components (e.g., `C1` and `C2`), depending on its complexity. The composition is expressed in the process `Interceptor`. As our approach is message-centric, each component "intercepts" the request in the port `iCN` (`iC` refers to "input port of component `CN`" that represents the point where the request enters in the component). Next, the request is processed inside the component and then passed to the next component through the port `oCN` (`oC` refers to the "output port of component N" that represents the point where the request exits the component) according to the constraints imposed by the process `Interceptor`.

3 Adopting the Framework Elements

In order to illustrate how the elements introduced in the previous session may be used to facilitate the middleware specification, we present the specification of a simple middleware that has a similar structure as CORBA and a message-oriented middleware (MOM).

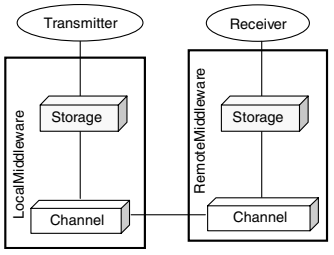



```
process RemoteMiddleware [invS, terS, comm] : noexit :=
  hide iSkeleton, oSkeleton, iTcp, oTcp, iDis, oDis in
  ( ( Skeleton [iSkeleton, oSkeleton] (1)
        ||| Skeleton [iSkeleton, oSkeleton] (2)
        ||| Channel [iTcp, oTcp, comm]
        ||| Dispatcher [iDis, oDis] )
    |[iSkeleton, oSkeleton, iTcp, oTcp, iDis, oDis]|
Interceptor [invS, terS, iSkeleton, oSkeleton, iTcp, oTcp, iDis, oDis] )
  where …
endproc
```

Fig. 6. Client-Server Middleware

Figure 6 presents a client-server middleware where the local middleware is a composition of a stub and channels elements. On the server side (remote), the middleware is more complex, as it is composed by a communication element (`Channel`), a dispatcher (`Dispatcher`) that forwards the request to the proper skeleton, and some skeletons (`Skeleton`). It is worth observing that additional middleware elements are easily added to the middleware just including them in the parallel composition (|||) and changing the `Interceptor` element.

A MOM is characterised by the use of a buffer to the asynchronous communication and it is widely adopted to communicate loosely coupled applications. Figure 6 shows a simple MOM specified by using the basic abstractions defined in Section 2.




```
(1) Process LocalMiddleware [send, receive, comm]: Noexit :=
(2)     hide iSto, oSto, iCh, oCh in
(3)     (( Storage [iSto,oSto] ||| Channel [iCh, oCh, comm])
(5)       |[iSto, oSto, iCh, oCh]|
(6)     Interceptor [send, receive, iSto, oSto, iCh, oCh])
(7)     where
(8)     …
(9)     endproc
```

Fig. 7. Message-Oriented Middleware

This MOM has two elements, namely `Channel` and `Storage`. The abstraction `Channel` is similar to Figure 6, whilst `Storage` is defined as presented in Section 2. MOMs that execute on the transmitter side are usually similar to one on the receiver (remote) side.

4 Conclusion and Future Work

This paper has presented a framework useful to formalise middleware behaviour based on LOTOS. The framework consists of a set of common elements usually found in the development of middleware systems. The framework is now being defined, but it is possible to observe that a formalisation approach centred on the message request instead of middleware layer facilitates the treatment of middleware complexity: simple abstractions are highly reusable (see abstraction `Channel` in Section 3) and easier to find specification errors and verify desired behaviour properties; and the way of composing middleware abstractions considering the order they intercept the message request enormously facilitate the composition of middleware abstractions.

We are now extending the proposed set of abstractions including more sophisticated communication and concurrent elements. Meanwhile, it is also planned to include the specification of middleware services in such way that composition constraints may also consider middleware service composition.

References

[1] Basin, David, Rittinger, Frank and Viganò, Luca (2002) “A Formal Analysis of the CORBA Security Service”, In: Lecture Notes in Computer Science, No. 2272, pp. 330-349.

[2] Bastide, Rèmi, Palanque, Philippe, Sy, Ousmane and Navarre, David (2000) “Formal Specification of CORBA Services: Experience and Lessons Learned”, In: OOPSLA’00, p. 105-117.

[3] Bastide, Rèmi, Sy, Ousmane, Navarre, David and Palanque, Philippe (2000) “A Formal Specification of the CORBA Event Service”, In: FMOODS’00, p. 371-396.

[4] Campbell, Andrew T., Coulson, Geoff and Kounavis, Michael E. (1999) “Managing Complexity: Middleware Explained”, IT Professional, IEEE Computer Society, Vol 1(5), pp. 22-28, October.

[5] Kreuz, Detlef (1998) “Formal Specification of CORBA Services using Object-Z”, In: Second IEEE International Conference on Formal Engineering Methods, pp., December.
[6] Venkatasubramanian, Nalini (2002) “Safe Composability of Middleware Services”, Communications of the ACM, Vol 45(6), pp. 49-52, June.
[7] Vinoski, Steve, (2002) “Where is Middleware?”, IEEE Internet Computing, Vol. 6(2), pp. 83-85.
[8] Rosa, Nelson and Cunha, Paulo (2004) “A Software Architecture-Based Approach for Formalising Middleware Behaviour”, Electronic Notes in Theoretical Computer Science, Vol. 108, pp. 39–51.
[9] Schmidt, Douglas and Buschmann, Frank (2003) “Patterns, Frameworks, and Middleware: Their Synergistic Relationships”, Proceedings of the 25th international conference on Software Engineering, pp. 694-704.