QoS for Distributed Objects by Generating Tailored Protocols

Matthias Jung, Ernst W. Biersack
Institut Eurécom, 2229 Route des Crêtes, 06190 Sophia Antipolis, France
{jung,erbi}@eurecom.fr

In ECOOP'00 Workshop on QoS for Distributed Object Systems, Cannes, France

Abstract

This paper presents a way to introduce quality of service management in distributed object systems based on the idea of tailorable and configurable protocol components.

1 Introduction

Distributed object systems (DOS)[1] (like DCOM [4], CORBA [9], or RMI [10]) allow to simplify the implementation of distributed systems by hiding distribution concerns and allowing the programmer to concentrate on the application logic. However, todays DOS provide almost no support to accommodate the quality of service requirements of distributed applications. Developers thus either must accept service mismatches or make large efforts to integrate possibly complex communications related functions into the application, which clearly contradicts the objectives of middle-ware systems.

Integrating QoS management into DOS is a challenging research topic that concerns different components of the system. One research direction proposes to integrate QoS management into the middle-ware system by modifying and extending it. TAO [8] extends a CORBA Object Request

[1]Also refered to as middle-ware systems

Broker by real-time facilities. The Object Management Group (OMG) published specifications for real-time and fault-tolerant services. Other examples for specialized middleware systems are Electra [5], Eternal [6], or DOORS [11]. A second approach proposes to extend the programming model to make QoS concerns explicit. Examples for this approach are QualityObjects (QuO) [12], MAQS [1], and the Squirrel project [3]. Another approach proposed by [7] introduces *interceptor* layers between the middleware infrastructure and the application to avoid changing neither the middleware system nor the application logic.

We believe that one reason for the inflexibility of common DOSs is due to the fact that distributed object services are generally built on top of TCP. The stream based nature of TCP does not match very well the request/response character of distributed object calls and does not provide any QoS differentiation. Consider a distributed game in the Internet: multicast, real-time services, and security functions are required in one distributed application, even expected from a single distributed object. None of these services are supported by TCP and are impossible to be realized efficiently on top of TCP.

Building a distributed object system on top of a light-weight protocol like UDP, which performs only de-multiplexing and guarantees that a delivered packet is not corrupted, gives more flexibility in applying QoS. However, building own protocols requires a lot of efforts in design and implementation, especially when reliability is required. If it would be possible to generate protocols dynamically in dependence of interface specifications of distributed objects and allow to integrate them with specialized protocols (each of which is responsible for a certain QoS criterion), the efforts of integrating QoS into the protocols of distributed objects can be minimized.

2 Proposal

Our work comprises two parts: the first part tackles structuring of protocol software and responds to the question how protocol software can be structured and organized to

- reduce crosstalk of protocol streams of different objects
- assure different QoS for different methods of distributed objects
- allow easy modification, extension, and tailoring of protocol software

We identified a system of architectural design patterns [2] that follow a vertical structure to overcome the problems of simply layering protocol services. We propose to vertically slice protocols into *data paths*, divide data paths into a chain of *protocol function objects*, encapsulate message headers in objects, and decouple function and header objects. The architecture has been implemented in a protocol environment prototype in Java. The fine granularity of the protocol function objects and the highly reflective character of the framework allow to easily combine and configure new protocols out of re-usable components. Furthermore, the vertical structure allows to extend and remove new services without interfering with other services. This is the key requirement to allow for different QoS in a single protocol session.

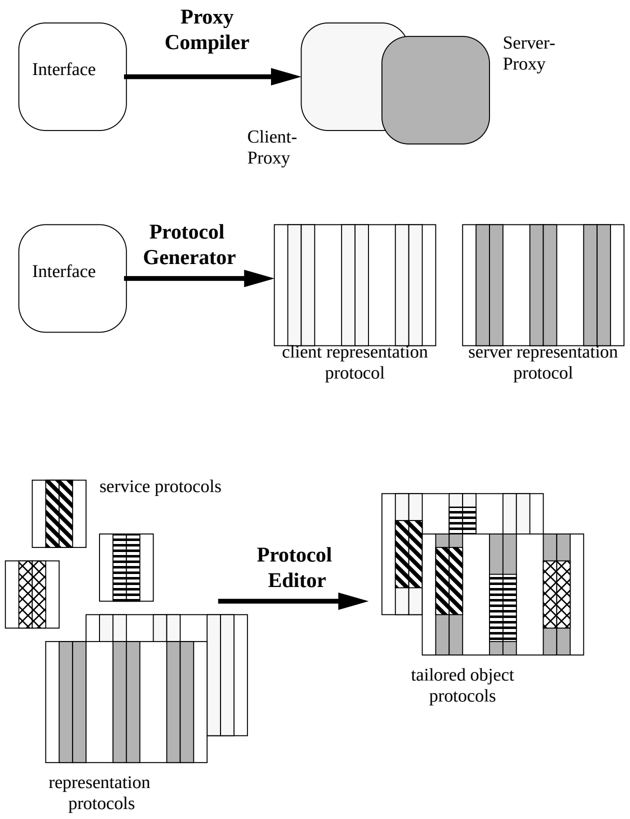

Figure 1: Generation tools

The second part consists of a set of tools to generate and integrate protocols and proxies. The *protocol generator* takes a Java interface as input, maps methods to data paths, constructs message formats in dependence of parameters, and produces *representation protocol* code. A representation

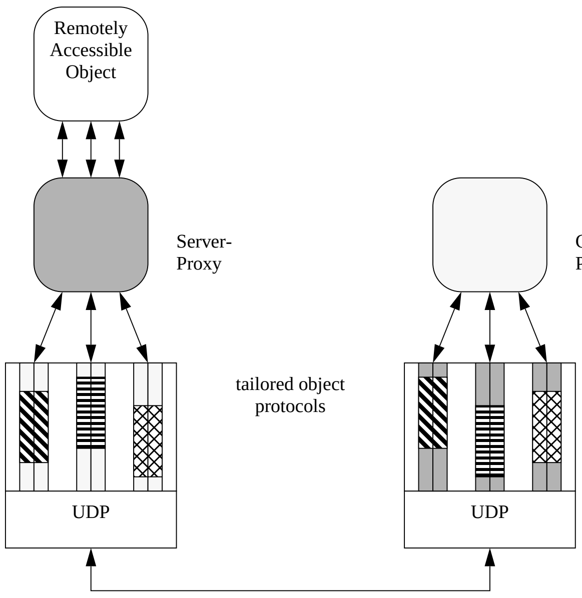

Figure 2: System Integration

protocol performs parameter conversion and maps communication data to methods. Example code for a client protocol based on an interface that defines the method `int add(int,int)` is depicted in Figure 3.

The *protocol editor* allows to integrate the generated object specific *representation protocol* code with pre-implemented *service protocol* code such that different methods can be associated with different service characteristics. A service protocol may be a real-time protocol, it may manage multicast groups, or perform admission control; service protocols are thus the core of QoS management. Some kind of specification is needed for each method to express how representation and service protocols should be combined (e.g. in form of an IDL extension for QoS). In our prototype implementation of the protocol editor, the server side programmer does this ”by hand”, i.e. he is free to choose any existing service protocol he wants and combine it accordingly with the generated representation protocol.

The *proxy compiler* generates server and client proxies for the Java interface and links them with the generated protocol code. The server proxy reads requests from the generated protocol, performs method calls to its remotely accessible object, and sends the responses via the generated protocol. The client proxy represents the server object in the client application. Figure 5 and Figure 4 give an impression how the generated code looks like. The tools used are illustrated in Figure 1.

Building a distributed application comprises the following steps:

1. the application developer writes a Java interface, which declares all methods that a distributed object is supposed to implement
2. the application developer implements a class considered to be remotely accessible (based on the defined interface)
3. the protocol-generator uses the interface as input to produce code that constructs a representation protocol
4. the proxy generator tool generates code that serves as proxy between application (server or client) and the protocol code
5. the application developer uses the protocol-editor tool to combine the produced representation protocol with protocol components required by the application (e.g. real-time, multicast-management, admission control)
6. by using the generated proxies, a distributed object can be easily and transparently be integrated in the client application

The interaction of the components at runtime is depicted in Figure 2.

3 Benefits

Our approach promotes an open, extensible system that allows to integrate any QoS criteria at any time – by just integrating a new service protocol that supports it. Tackling QoS integration at the lowest level promises to be the most efficient approach, since QoS are managed close to where they originate. It also fits well into existing middle-ware systems since it avoids complexity and modification on higher levels. Transparency for the application is also guaranteed; compared to Java's RMI things are getting even easier since distributed objects are not obliged to extend special classes.

Acknowledgments

This work is sponsered by Siemens ZT IK2, Munich.

References

[1] C. Becker and K. Geihs, "MAQS - Management for Adaptive QoS-enabled Services", In *IEEE Workshop on Middleware for Distributed Real-Time Systems and Service*, San Francisco, USA, 1997.

[2] M. Jung and E. Biersack, "Order-Worker-Entry: A System of Patterns to Structure Communication Protocol Software", In *Proc. of Euro-PLoP*, Bad Irsee, Germany, July 2000.

[3] R. Koster and T. Kramp, "Structuring QoS-Supporting Services with Smart Proxies", In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, Hudson River Valley, USA, April 2000.

[4] D. Krieger and R. M. Adler, "The Emergence of Distributed Component Platforms", *IEEE Computer*, pp. 43–53, March 1998.

[5] S. Maffeis, "Adding Group Communication and Fault Tolerance to CORBA", In *Proceedings of USENIX Conference on OO Technologies*, Monterey, CA, June 1995.

[6] L. Moser, P. Melliar-Smith, P. Narasimham, L. Tewksbury, and V. Kalogeraki, "The Eternal System: An Architecture for Enterprise Applications", In *3rd International Enterprise Distributed Object Computing Conference (EDOC)*, University of Mannheim, Germany, sep 1999.

[7] J. Pruyne, "Enabling QoS via Interception in Middleware", , Hewlett-Packard Labatories, 2000.

[8] D. Schmidt, D. Levine, and T. Harrison, "The Design and Performance of a Real-time CORBA Object Event Service", In *Proceedings of OOPSLA*, Atlanta, Georgia, April 1997.

[9] J. Siegel, *CORBA – Fundmentals and Programming*, John Wiley and Sons, Inc., 1996.

[10] Sun Microsystems, "The Java Remote Method Invocation Specification", 1999, http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi-spec.ps.

[11] S. Yajnik, "DOORS: Fault Tolerance for CORBA Applications", In *FIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, England, September 1998.

[12] J. Zinky, R. Schantz, J. Loyall, K. Anderson, and J. Megquier, "The Quality Objects (QuO) Middleware Framework", In *RM – Workshop on Reflective Middleware*, New York, USA, April 2000.

```
public void init() throws ProtocolConstructionException {
             Environment env=getEnvironment();
             //ORDER DEFINITIONS
             OrderRegistrar orderReg=new OrderRegistrar();
             env.setOrderRegistrar(orderReg);
             .....
             // Definition of the order-type add_callOut
             OutputOrderType O7_addc=new OutputOrderType();
             orderReg.addOrder(O7_addc);
             StructureRegistrar r_O7_addc=new StructureRegistrar();
             O7_addc.setStructureRegistrar(r_O7_addc);
             O7_addc.setName("add_callOut");
             O7_addc.setPriority(5);
             O7_addc.setNrOutputSAP(0);
             O7_addc.setOrderID(4);
             //Define all ENTRYs
             probeans.entries.IntegerEntryType O7_addc_E6_____=new probeans.entries.IntegerEntryType()
             r_O7_addc.addEntry(O7_addc_E6_____);
             O7_addc_E6_____.setVisibleFlag(true);
             O7_addc_E6_____.setInitFlag(true);
             O7_addc_E6_____.setSize(4);
             probeans.entries.IntegerEntryType O7_addc_E7_____=new probeans.entries.IntegerEntryType()
             r_O7_addc.addEntry(O7_addc_E7_____);
             O7_addc_E7_____.setVisibleFlag(true);
             O7_addc_E7_____.setInitFlag(true);
             O7_addc_E7_____.setSize(4);
             //Define all PARAMETER-relations

             // Definition of the order-type add_respIn
             InputOrderType O8_addr=new InputOrderType();
             orderReg.addOrder(O8_addr);
             StructureRegistrar r_O8_addr=new StructureRegistrar();
             O8_addr.setStructureRegistrar(r_O8_addr);
             O8_addr.setName("add_respIn");
             O8_addr.setPriority(5);
             O8_addr.setNrInputSAP(0);
             O8_addr.setOrderID(11);
             //Define all ENTRYs
             probeans.entries.IntegerEntryType O8_addr_E8_____=new probeans.entries.IntegerEntryType()
             r_O8_addr.addEntry(O8_addr_E8_____);
             O8_addr_E8_____.setVisibleFlag(true);
             O8_addr_E8_____.setInitFlag(true);
             O8_addr_E8_____.setSize(4);
             .....
             return env;
}
```

Figure 3: Generated Client Protocol Code

```
CLIENT-STUB:
public int add(int p1, int p2) {
int methodNr=4;
try {
Object[] param={new Integer(p1), new Integer(p2)};
allWriteAPIs[methodNr-1].accept(param);
try {
synchronized(blocking[methodNr-1]) {
blocking[methodNr-1].wait();
}
}
catch(Exception e) {
throw new RuntimeException("remote error calling add(int p1, int p2)");
}
return ((Integer)results[methodNr-1]).intValue();
}
catch(Exception e) {
throw new RuntimeException("local system error in add(int p1, int p2)");
}
   }

public void deliver(Object[] o, Deliverable d) {
try {
int i=((InputOrder)d).getOrderID()-numberOfMethods;
if (o.length!=0)
   results[i-1]=o[0]; //set return value
synchronized(blocking[i-1]) {
blocking[i-1].notify();
}
}
catch(Exception e) {
throw new RuntimeException("incoming information is unusable");
}
   }
```

Figure 4: Generated Code of Client Stub

```
public void deliver(Object[] o, Deliverable d) {
                try {
                        int i=((InputOrder)d).getOrderID();
                        Address a=((InputOrder)d).getAddress();
                        switch (i) {
                                case 1: {
                                   callMethod1(o,a);
                                   break;
                                }
                                case 2: {
                                   callMethod2(o,a);
                                   break;
                                }
                                case 3: {
                                   callMethod3(o,a);
                                   break;
                                }
                                case 4: {
                                   callMethod4(o,a);
                                   break;
                                }
                        }
                }
                catch(Exception e) {
                        throw new RuntimeException("incoming information is unusable");
                }
   }

private void callMethod4(Object[] o, Address a) {
                //code for add
                int methodNr=4;
                try {
                        int p1=((Integer)o[0]).intValue();
                        int p2=((Integer)o[1]).intValue();
                        Object[] ret={ new Integer(myObj.add(p1, p2)) };
                        allWriteAPIs[methodNr-1].accept(ret,a);
                }
                catch(Exception e) {
                        throw new RuntimeException("local system error in add(p1, p2)");
                }
   }
```

Figure 5: Generated Code of Server Skeleton