European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

A Network of Excellence funded by the European Commission

A Pattern for Conversational Protocols

Michael Parkin, Rosa M. Badia
{michael.parkin,rosa.m.badia}@bsc.es
Universitat Politècnica de Catalunya
Nexus II, Jordi Girona 29
08034 Barcelona, Spain

Josep Martrat
josep.martrat@atosorigin.com
Atos Origin Research & Innovation
Avinguda Diagonal 210–218
08018 Barcelona, Spain

CoreGRID Technical Report
Number TR-0163
August 22, 2008

Institute on Resource Management and Scheduling

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

A Pattern for Conversational Protocols

Michael Parkin, Rosa M. Badia
{michael.parkin,rosa.m.badia}@bsc.es
Universitat Politècnica de Catalunya
Nexus II, Jordi Girona 29
08034 Barcelona, Spain

Josep Martrat
josep.martrat@atosorigin.com
Atos Origin Research & Innovation
Avinguda Diagonal 210–218
08018 Barcelona, Spain

CoreGRID TR-0163
August 22, 2008

Abstract

This technical report describes a pattern for implementing 'conversational' style protocols in an object-oriented manner. We discovered this pattern in our recent work developing protocols that assume asynchronous, unreliable messaging between two participants. It is hoped that by abstracting this implementation pattern out into a clear template, future development and implementation of such conversational protocols will be made more straightforward.

1 Introduction

Our recent work has involved the design and implementation of network protocols that have a 'conversational' style, that is they involve asynchronous, unreliable message sending between two participants. During this work we found that a common implementation pattern emerged and we were able to factor out this pattern into an abstract description and use this abstraction to develop further protocols that assume the same conditions. It is that pattern which is presented in this report.

The protocols we have implemented have at their heart the concept of a stateful 'conversation' within which co-related messages are sent and received to change the state of the conversation. This report describes the general implementation pattern we found as part of our work developing 'conversational' protocols for the negotiation, agreement, re-negotiation and termination of Service Level Agreements (SLAs) and contracts in general whilst meeting the requirement of handling 'unreliable messaging'.

1.1 Why a design pattern?

A design pattern like this has several benefits. First, it can provide us with a way to solve an issue related to software development using a solution derived from experience, not theory. Secondly, a design pattern "isolates the variability that exists in the system... making the overall system easier to understand and maintain" [1], i.e. a pattern provides an abstraction of a problem and its solution that can be applied in different implementation scenarios. Finally, and related to the second point, using a design pattern makes communication between software developers more efficient;

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

by using patterns developers can immediately understand and picture a high-level design in their heads when they refer the name of the pattern used to solve a particular issue when discussing system design, i.e. it introduces elements of a common vocabulary developers can use to describe their software.

1.2 Context & Motivation

Recently, our research has been focussed around the negotiation, agreement, re-negotiation and termination of Service Level Agreements (SLAs, or "contracts between a provider and a user that specifies the level of service that is expected during the term of the contract" [2]) for the provision of network-based services and applications. Because the current standard for creating SLAs, WS-Agreement [3], is limited to a simple offer-accept protocol [4] there is an outstanding requirement for a bi-lateral multi-round negotiation and agreement protocol which has yet to be met. Further, WS-Agreement does not specify how re-negotiation of a SLA should be carried out and the termination protocol is limited. Thus, we set out to design and implement protocols that allow the negotiation, agreement and re-negotiation of contracts with the constraints that the specification is implementation independent and that the maximum possible interoperability is achieved.

1.2.1 Achieving Interoperability

In order to specify our negotiation and re-negotiation protocols we started with the two requirements that:

1. The protocol specification should be implementation independent, i.e. it should not mandate the use of any particular architectural style, such as the Web Services architecture.
2. The network between the two parties negotiating the contract should not be assumed to be reliable. That is, the network between the parties participating in the negotiation protocol may malfunction or fail completely at any point in time during their interactions.

The first requirement is hopefully self explanatory and allows developers to use any implementation method they wish to realise the protocols.

The second requirement, we feel, is mandatory when two services communicate over a network they do not have complete control over, e.g. when messaging is taking place between two independent organisations over the Internet. Because we assume such a network, messages on it at the time of malfunction or failure may be lost, duplicated, delayed and/or re-ordered. As a result of this behaviour messages may not be processed by the receiver in the order they were sent (i.e. there is asynchronous message processing) and the sender of a message has no knowledge about when or if the recipient processed a message they have sent and how many copies of the message it will receive.

This assumption about network behaviour has two further advantages that are not immediately evident. First, it reduces the dependencies, in terms of implementation, between the participants in the protocol allowing them to be de-coupled as much as possible. For example, if we were to take the approach of specifying a protocol that assumed once-only, in-order, guaranteed delivery of messages, this could only be achieved through implementing the protocol over a message-oriented middleware (MOM) such as IBM's WebSphere MQ [5], Microsoft's Message Queuing (MSMQ) [6], an application implementing the Java Message Services (JMS) application programming interface (API) [7] or a bespoke reliable messaging solution. The WS-ReliableMessaging specification [8] also attempts to provide a solution for reliable messaging between autonomous services. But, as we describe in [9], this specification does not define *how* reliable messaging should be achieved and does not provide the foundation for building interoperable, reliable, distributed applications that require this class of messaging.

Because the MOMs listed above are not interoperable (i.e. a message sent using one MOM cannot be understood by another[1]) two interacting organisations must use the same MOM if a protocol is designed with the assumption of reliable messaging. This couples the organisations implementing the protocol in terms of the middleware used, which in the scenario of dynamic service provision between many independent and autonomous organisations is impossible to enforce or guarantee. Therefore, to de-couple the organisations within their interactions as much as possible the contract negotiation protocol specified here does not assume reliable messaging and instead assumes the opposite, i.e. *unreliable messaging* where messages may be lost, delayed, duplicated and/or re-ordered.

[1] This includes implementations of JMS, which is a specification of a standard API but does not specify the format of the messages sent over the wire. Thus, different implementations of JMS are non-interoperable.

The second advantage of making no assumptions about the reliability of the network is that since messages may be received multiple times and out-of-order many different communications channels can be used to send those messages. For example, if a contract negotiation is at a critical phase and the protocol has been designed with an assumption of unreliable messaging, then if one communications channel is unavailable another can be used to receive and send messages to the other party to find (or change) the state of the contract being negotiated. Thus, a message sent on more than one channel may be received more than once at its destination and, due to the different latencies and reliability of communications channels, messages may be duplicated and/or received out-of-order or not received at all.

As an example of how this works in practice, consider the following scenario: a customer is negotiating the supply of a service with a service provider, initially through the service providers web-interface. After submitting an order for a service in error the network between the customer and the provider fails completely. The customer realises their mistake and when they find the web-interface is unavailable, send an email cancelling the order to the service provider. The network is down so this email is stuck in the outbox of the customer, so they send the same message by fax to the service provider. Later, when the network is available, the cancellation email is sent, resulting in a delayed, duplicate message being received by the service provider. What this scenario illustrates is that the protocol provides greater fault-tolerance when it can cope with out-of-order, duplicate messages being received as an implementation can use multiple communication channels to send and receive messages — that is, a communication channel is not a single point of failure for the negotiation protocol.

1.2.2 Conversation & Message Identification

A further requirement of our 'conversational' protocols was that each protocol should allow multiple conversations to take place between two interacting parties. This required us to introduce an identifier to co-relate messages, so that if multiple messages were received by a party they could determine which message belonged to which conversation. To achieve this we use the concept of an *Asynchronous Completion Token*, first described in [10], which provides a context "that uniquely identifies the actions and state necessary to process the operations completion".

In this pattern we have specified that the value of this token is globally unique so that in the case where a single party is having a several conversations with multiple parties there is no clash of token value. There are freely-available libraries for most programming languages that can generate globally unique or universally unique identifiers according to the format described in [11].

Finally, although an individual message may be correlated within a conversation using the asynchronous completion token a message needs to be uniquely identified within a conversation so that it may be referred to and referenced within the conversation. Thus, we specify that each message must have an identifier that is unique within the conversation (i.e. it must be locally unique). Together the asynchronous completion token and the message identifier form a tuple that uniquely identifies each message.

1.3 Report Structure

The remainder of this report is structured as follows: Section 2 introduces the abstract pattern and the concepts it is based upon and Section 3 presents our conclusions and future work.

2 Pattern Description

The abstract pattern is shown in Figure 1, a UML class diagram illustrating the relationships between the each of the concepts in the pattern. Each of the classes in the diagram and its relationships are introduced in the remainder of this section. In the description class names are given in `Typewriter` font. Throughout the description references are made to our implementation of an SLA negotiation protocol to illustrate how each of the concepts in the pattern relate to a concrete implementation.

2.1 Conversation Class

The abstract `Conversation` class represents a conversation and the context within which all messages can be correlated. In the contract negotiation protocol this is sub-classed into a `Negotiation` class which can be instantiated. The `Conversation` class must contain a globally unique identifier (a GUID, introduced in Section 1.2.2) to provide an identifier that allows the correlation of messages within a specific dialogue. Each instance of a `Conversation`

has exactly one `State` and zero or more sent or received `Messages`. In Figure 1, attributes to record when the conversation was started and when the conversation was last modified (i.e. when a `Message` was last sent or received) have been included, though these are not mandatory in any implementation.

2.2 State Class

The `State` class represents the current status of a conversation. This is an abstract class that must be sub-classed in a protocol that implements the conversational pattern with the states the conversation can take. Thus, in the example of a contract negotiation protocol the `State` represents the current position of the negotiation, such as `Negotiating`, `Contracted` or `Terminated`. The cardinalities reflect that a `Conversation` must have a status, that no `Conversation` instances may be in a particular `State` and that different `Conversation` instances may share the same `State`.

2.3 ProtocolMessage Class

Like the `Conversation` and `State` classes, the `ProtocolMessage` class is also abstract and represents the messages which may be sent or received within the protocol. For example, a negotiation protocol may have concrete subclasses of `ProtocolMessage` such as `QuoteRequest`, `Quote`, `Offer` and `Accept`. As described in Section 1.2.2 each unique `ProtocolMessage` instance has an identifier that is unique within the context of the conversation, or a locally unique identifier (LUID).

The cardinalities shown on the class diagram allow each `ProtocolMessage` instance to be sent multiple times in different `Message` instances allowing the conversation participants to send duplicates of the same `ProtocolMessage` instance over the same or different communications channels more than once, a requirement of the messaging conditions described in Section 1.2.1.

2.4 Message Class

The abstract `Message` class represents the application-protocol 'wrapper' of a `ProtocolMessage` instance. The inclusion of this class allows the de-coupling of the communications medium from the representation of the actual message, a requirement we described in Section 1.2.1 to achieve the goal of being able to send messages over multiple communications channels for greater fault tolerance. Thus, each concrete subclass of the `Message` class represents a message sent over a particular communications channel. These subclasses could be, for example, `HttpMessage`, `JmsMessage`, `SmtpMessage` and `XmppMessage` to represent messages sent or received via the HTTP, JMS, SMTP or XMPP application protocols. Each `Message` must contain an identifier unique within the context of the conversation, the identity of the sender, the direction of the message (outbound or inbound, i.e. sent or received) and the date and time the message was sent or received. Each subclass of this class should contain the destination to which messages should be returned (e.g. an SMTP address, the URL and name of a JMS queue, a single URL accessed over HTTP or an XMPP identity).

The cardinalities with the `Conversation` class reflect that a `Message` instance may be sent or received multiple times as part of a conversation. Also, a `Message` instance must contain at least one `ProtocolMessage`. In practice this means that, for example in the negotiation protocol, a `Message` instance may contain once or more `Quotes` instances (i.e. instances of a subclass of the `ProtocolMessage` class) and, conversely, a single `ProtocolMessage` can be sent in many different `Message` types (i.e. over many application protocols).

2.5 Detail Class

The final class in Figure 1 is the `Detail` class. An instance of this class contains the 'payload' or the essential domain-dependent data that should be completed by the sender of the `ProtocolMessage` to which the `Detail` instance belongs. The receiver should interpret the contents of the `Details` in order to decide what to do next in order to progress the conversation. For example, in a contract negotiation protocol instances of this class contained in `QuoteRequest` or `Offer` messages may contain information about a resources capabilities (advertised or required) and the period of time the resource is required for (e.g. start time and end time).

The cardinalities shown on the class diagram allow each `ProtocolMessage` to contain one or more instances of a `Detail`. This capability is required when, in a contract negotiation protocol for example, a resource requester asks for the availability or co-allocation of several resources within the same `QuoteRequest` message `Offer` message.

3 Conclusions & Future Work

This report has described a design pattern, or a "general, reusable solution to a commonly occurring problem" [12], which provides a template for how to solve a problem that can be used in many different situations. In our work we have used this design pattern to implement SLA negotiation, agreement and re-negotiation protocols that have a 'conversational' style, i.e. where messages may be sent asynchronously over different, unreliable communications channels multiple times. Our future work is to experiment more with this design pattern, determine its applicability to further types of protocol and refine it if necessary.

4 Acknowledgements

Michael Parkin is pleased to acknowledge that this work was carried out as part of an industrial fellowship for the CoreGRID IST project N°004265, funded by the European Commission and partly sponsored by ATOS Origin Research and Innovation Spain.

References

[1] J. Maioriello. What Are Design Patterns and Do I Need Them? developer.com Article. `http://www.developer.com/design/article.php/1474561`.

[2] Global Grid Forum Open Grid Services Architecture Working Group (OGSA-WG). Open Grid Services Architecture Glossary of Terms, January 2005.

[3] A. Andrieux *et. al.* Web Services Agreement Specification (WS-Agreement). Proposed Recommendation, Open Grid Forum, September 2006. Grid Resource Allocation Agreement Protocol Working Group (GRAAP-WG).

[4] H. Ludwig, T. Nakata, P. Wieder, and O. Wäldrich. Reliable Orchestration of Resources using WS-Agreement. CoreGRID Technical Report TR-0050, October 2006.

[5] WebSphere MQ. IBM Corporation. Product Overview. `http://www.ibm.com/software/mqseries/`. Last accessed 30 May 2007.

[6] Microsoft Message Queuing. Microsoft Coproration. Technical Documentation. `http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx`. Last accessed 30 May 2007.

[7] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service. Application Progamming Interface Specification v1.1, Sun Microsystems, April 2002. `http://java.sun.com/products/jms/docs.html`. Last accessed 30 May 2007.

[8] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and U. Yalcinalp (Eds.). OASIS Web Services Reliable Messaging Protocol. Technical specification, OASIS WS-RX Technical Committee, August 2006. `http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-rddl-200608.html`. Last accessed 30 May 2007.

[9] D. Kuo and M. Parkin. Negotiating the Minefields in Realising the i2010 Vision – A Position Paper. In M. Cunningham and P. Cunningham, editors, *Proceedings of the 5th eChallenges Conference*, 2007. To appear.

[10] D.C. Schmidt. Asynchronous Completion Token. Siemens AG Technical Document, 1999. `http://www.cs.wustl.edu/~schmidt/PDF/ACT.pdf`.

[11] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. IETF Network Working Group RFC 4122, July 2005. `http://www.ietf.org/rfc/rfc4122.txt`.

[12] E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison-Wesley Professional Computing Series, November 1994.

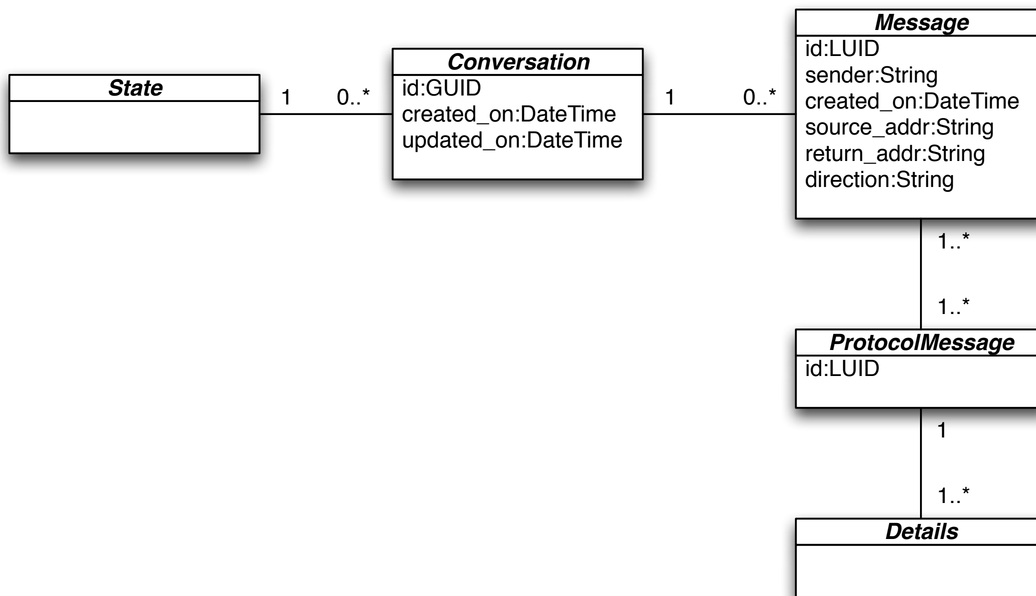

Figure 1: Conversational protocol pattern