Semantic-Web-enabled Protocol Mediation for the Logistics Domain

Oscar Corcho, Silvestre Losada, Richard Benjamins

Intelligent Software Components, S.A.
Pedro de Valdivia, 10. 28006 Madrid, Spain
Now at Universidad Politécnica de Madrid, Spain

Abstract

Among the problems that arise when trying to make different applications interoperate with each other, protocol mediation is one of the most difficult ones and for which less relevant literature can be found. Protocol mediation is concerned with non-matching message interaction patterns in application interaction. In this paper we describe the design and implementation of a protocol mediation component that has been applied in the interoperation between two heterogeneous logistic provider systems (using two different standards: RosettaNet and EDIFACT), for a specific freight forwarding task.

1 Current Situation

Logistics management is a typical business problem where the use of a Service Oriented Architecture is clearly suited. As pointed out in (Evans-Greenwood and Stason, 2006) the current trend in logistics is to divide support between planning applications, which compute production plans overnight, and execution applications, which manage the flow of events in an operational environment. This disconnection forces users to deal with business exceptions (lost shipments, for example), manually resolving the problems by directly updating the execution and planning applications. However, this human-dependency problem can be ameliorated by using Web technology to create a heterogeneous composite application involving all participants in the process, providing a complete Third-Party Logistics solution, and giving users a single unified view into the logistics pipeline. This consolidated logistics solution greatly simplifies the task of identifying and correcting business exceptions (e.g., missing shipments or stock shortages) as they occur.

Furthermore, (Evans-Greenwood and Stason, 2006) also talk about the possibility of combining multiple Third-Party Logistics solutions into a single heterogeneous virtual logistics network. With such a virtual network, each shipment is assigned a route dynamically assembled from one or more individual logistics providers, using dynamically created virtual supply chains. Most of these business functions are still manual and offline, but most of them can be automated with the use of Service Oriented Architectures, as will be presented in this chapter. Obviously, the main advantages of using such solutions are the decreases in cost and speed in transactions, which influence in a better quality of the service provided to customers.

The main barrier to set up a business relationship with a company in the logistics domain is that it usually requires an initial large investment of time and money. This is ameliorated by the emergence of some industry standards like EDIFACT (EDIFACT), AnsiX12 (AnsiX12) or RosettaNet (RosettaNet), which ease the integration tasks between information systems that comply with them. However, given that these standards have some flexibility in what respects the content and sequencing of the messages that can be exchanged, the integration of systems is still time and effort consuming. Besides, there is sometimes a need to integrate systems that use different standards, what makes the integration task even more time and effort consuming.

This is the focus of one of the four case studies developed in the context of the EU project SWWS[1] (Semantic-Web enabled Web Services), a demonstrator of business-to-business integration in the logistics domain using Semantic Web Service technology. All the features of this demonstrator are described in detail in (Preist et al., 2005), including aspects related to the discovery and selection of relevant services, their execution and the mediation between services following different protocols.

In this chapter we will focus on the last aspect (mediation) and more specifically on protocol mediation, which is concerned with the problem of non-matching message interaction patterns. We will describe the design and implementation of the protocol mediation component applied in this case study to show how to make logistic provider systems using two different standards (RosettaNet and EDIFACT) interoperate for a specific freight forwarding task.

The chapter is structured as follows. The rest of this section introduces a motivating example, focusing on the needs for protocol mediation, and gives some background on how the problem of mediation can be characterised in general and on the approaches for mediation proposed in the context of Semantic Web Service research. Section 2 summarises the protocol mediation approach followed for this case study and the main elements to be considered inside the approach. It also describes the ontology used for the description of the abstract and concrete protocols used by the entities involved in the message exchange. Section 3 provides an overview of the API of the protocol mediation component and gives details about how to configure it for deployment. Finally, section 4 gives some conclusions.

1.1 An example in the logistics domain

Let us imagine that we have a manufacturing company in Bristol, UK, which needs to distribute goods internationally. The company outsources transportation into other companies, which offer *Freight Forwarding Services*. These companies may be providing the transportation service by themselves or just act as intermediaries, but this is not important for the manufacturing compnay. However, the manufacturing company still needs to manage relationships with these service providers, as a *Logistics Coordinator*, being responsible for selecting the service providers, reaching agreements with them with respect to the nature of the service that they will provide, coordinating the activity of different service providers so as to ensure that they link seamlessly to provide an end-to-end service (e.g., if a ship company transports a goods to a port, then the ground transportation company should be waiting for those goods with a truck to transport them to an inland city), etc.

The manufacturing company uses EDIFACT for its exchange of messages with the service providers. However, not all of them use this standard, but in some cases RosettaNet. So the situation can be that two different companies that can offer the same service (e.g., road transportation inside Germany) are using two different standards and the logistics coordinator should be able to use any of them, independently of the protocol that they use in their information systems, taking only into account the business requirements that the parcel delivery may have (quality of service, speed, price, insurance, etc.). In this situation there is a need for a seamless integration of a mediation component that is able to capture the EDIFACT messages sent by the Logistics Coordinator into RosettaNet ones that are sent to the corresponding Freight Forwarding Service, and vice versa, without any change to the information systems of any of the parties involved.

1.2 Mediation in Service Oriented Architectures and in Semantic Web Services

In **service oriented architectures**, mediation services are middleware services that are in charge of resolving inconsistencies between the parties involved in a sequence of message exchanges. Mediation can be considered at different levels:

- Data mediation: transformation of the syntactic format of the messages.
- Ontology mediation: transformation of the terminology used inside the messages.
- Protocol or choreography mediation: transformation of sequences of messages, to solve the problem of non-matching message interaction patterns.

All types of mediation are important to achieve a successful communication between the services involved in an application, and each of them poses different challenges. In this chapter we will focus on aspects related to the last type of mediation, which is the one aimed at ensuring that, from a high-level point of view, the services involved in a message exchange achieve their overall goals. In other words, it aims at mapping the patterns of conceptually similar, but mechanically different interaction protocols sharing a similar conceptual model of a given domain.

The atomic types of mismatches that can be found between a set of interaction patterns are (Cimpian and Mocan, 2005):

- **Unexpected Messages**. One of the parties does not expect to receive a message issued by another. For instance, in a request for the delivery of a parcel the logistics provider sends the parcel weight and size, the departure place and the arrival place, while the freight forwarding service does not expect the parcel weight and size, since it will not use this information.
- **Messages in Different Order**. The parties involved in a communication send and receive messages in different orders. In the previous case the sender may send the messages in the order specified above while the receiver expects first the arrival place and then the departure place.
- **Messages that Need to be Split**. One of the parties sends a message with multiple information inside it, which needs to be received separately by the other party. In the previous example, the sender sends the arrival and departure places in one message, while the receiver expects it as two messages.
- **Messages that Need to be Combined**. One of the parties sends a set of messages that the receiver expects as a single message with the multiple information. We can think of the inverse situation to the one aforementioned.
- **Dummy Acknowledgements or Virtual Messages that Have to be Sent**. One of the parties expects an acknowledgement for a certain message, but the receiver does not issue such acknowledgement; or the receiver expects a message that the sender is not prepared to send.

One of the purposes of the work on **Semantic Web Services** is the automation of some of the tasks involved in the development of applications that follow a Service Oriented Architecture. As a result, some work on mediation has been done in the area. If we focus on protocol mediation, we can find the following two approaches:

Priest and colleagues (2005) and Williams and colleagues (2006) describe the approach followed in the context of SWWS, and which will be described in more detail in the next section. This approach is based on the use of a general abstract state machine that represents the overall state of the communication between parties, and a set of abstract machines for each of the parties in the conversation, which specify their state and the sets of actions to be performed when they receive a set of messages or when they have to send a set of messages.

In the context of the WSMO initiative, Cimpian and Mocan (2005) describe the approach taken for the design and implementation of the process mediator for the Semantic Web Service execution engine WSMX. This approach is similar to the previous one, since it is also based on the use of an abstract machine with guarded transitions that are fired by the exchange of messages and the definition of choreographies for each of the parties involved in the communication.

2 Proposed Solution: The SWWS approach for protocol mediation

This section describes briefly the main components involved in our protocol mediation approach. A more detailed explanation is provided in (Williams et al., 2006), and fig. 2 shows an example of the use of all these components in the logistics domain described in the introduction.

2.1 Communicative Acts

Communicative acts are the basic components of the communication. They are modelled as sequences of four events that are exchanged between systems and the underlying communication infrastructure when sending a message (see fig. 1), as follows:

- `.request`. The initiator sends a message to the communication infrastructure.
- `.indication`. The responder receives the message from the communication infrastructure.
- `.response`. The responder acknowledges the receipt of the message.
- `.confirm` . The initiator receives the acknowledge receipt.

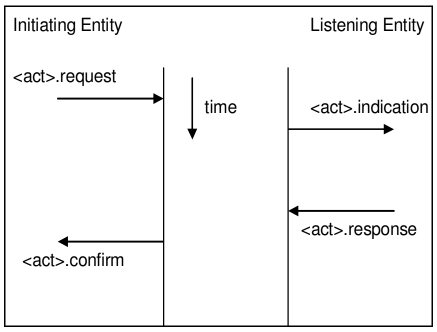


Fig. 1. A communicative act and its four events (Williams et al., 2006).

Both the `.response` and `.confirm` primitives model an acknowledgement that the communication has reached its intended receipient. Any substantive response motivated by the communicative act itself is modelled as a subsequent communicative act in the opposite direction.

At the initiator, the outcome of a communicative act may be a success (the initiator knows that the communication has reached the intended recipient), an exception (the initiator knows that the communication has failed to reach the intended recipient), or indeterminate (the initiator does not know the outcome of the communication).

2.2 Abstract Protocols and Roles

When we described protocol mediation, we commented that systems involved in a message exchange have conceptually similar interaction protocols. This high-level conceptual protocol is described by means of an abstract protocol.

The abstract protocol can be then defined as a multi-party choreography that describes the constraints that govern the sequencing of communicative acts between the systems engaged in an interaction. Each system takes on one or more roles (e.g., buyer, seller, logistics provider, freight forwarder, etc.) with respect to a choreography. The choreography then describes each of these roles in terms of the sequencing constraints on the exchange of primitives between the communication infrastructure and the system adopting the role.

2.3 Concrete Protocols

Each of the systems involved in a message exchange may have different mechanics by which communicative acts are managed. For each communicative act event in each system we will have then a concrete protocol that describes this behaviour.

Hence concrete protocols describe what happens at an initiating system in response to an admissable `.request` primitive and prior to (and after) the corresponding `.confirm` primitive. Likewise, at a responding system in response to the stimuli that give rise to an `.indication` primitive, the behaviours that occur between that and the corresponding `.response` and the behaviours that occur after that.

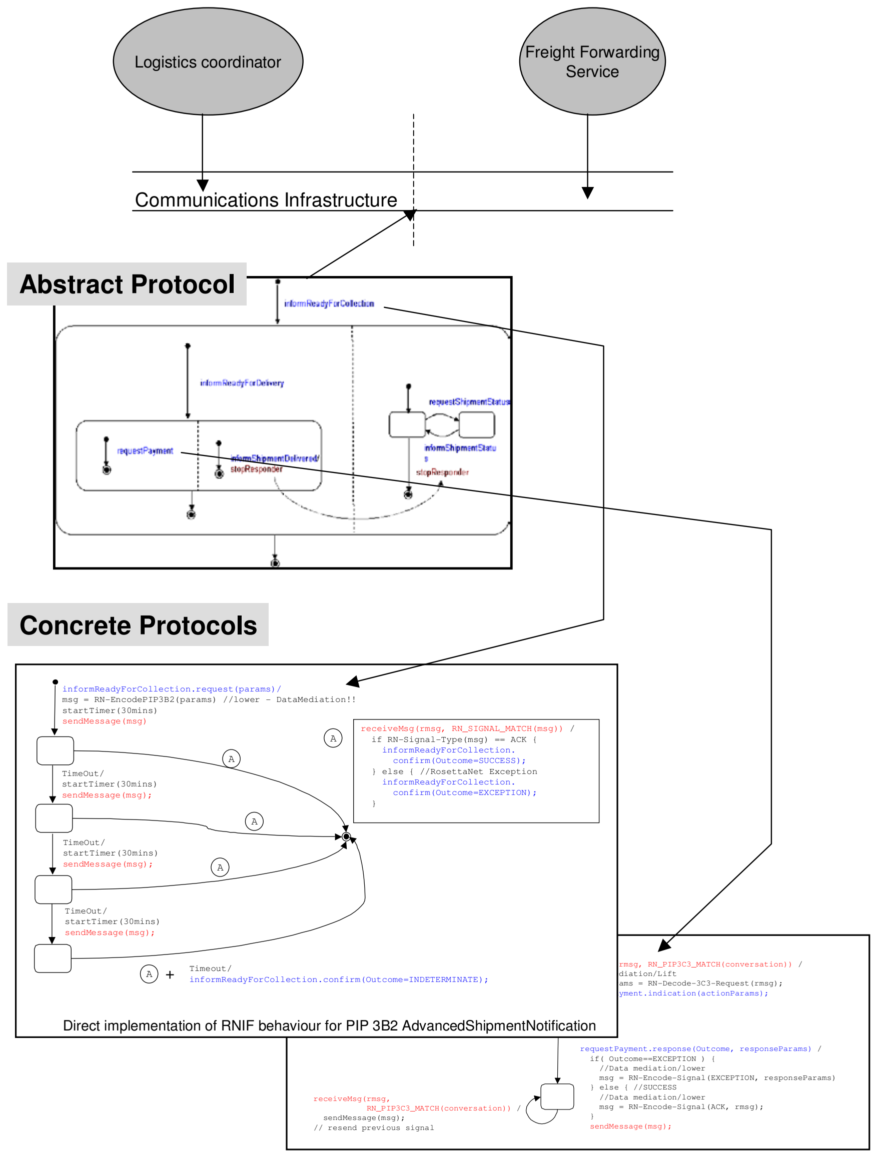
2.4 Processes as abstract state machines

The abstract and concrete protocols are described by means of processes, which in our approach are implemented by concurrent finite state machines. For abstract protocols a state represents the state of the high-level conversation in the context of the common multi-party choreography (e.g., a request for payment has been issued by the freight forwarding service and received by the logistics coordinator). For concrete protocols a state represents some intermediate state in the behaviours associated with the issuing of `.request` and `.confirm` primities or issuing `.indication` and `.response` primitives.

Transitions between states may be driven by different internal and external actions, as follows:

1. *PrimitiveDriven Transitions*. In abstract protocols they can be any of the primitives of a communicative act. In concrete protocols, they can be only `<act>.request` or `<act>.response` primitives, since these primitives can initiate the state machines associated to a concrete protocol.
2. *EventDriven transitions*. They are used to communicate between concurrent processes (a process may raise an event that is being waited for by another process). They are normally used in communication exchanges between more than 2 parties and in concrete protocols (e.g., two processes are waiting for the same payment under different payment procedures, credit card or cheque, and one of them is satisfied).
3. *TimeDriven Transitions*. They occur on the expiry of a time interval following the entry to the state that has the time driven transition associated. They can be used in any type of protocol (e.g., in an abstract protocol, the system will have a timeout feature to send another communicative act if a response has not been received in a given time).
4. *MessageDriven Transitions*. They occur only in concrete protocols, when a message is received from the communication infrastructure and filtered according to a template, so that the relevant information is extracted (e.g., for a freight forwarding service, if a request for a shipment service is broadcasted through the communication infrastructure, this could activate it so that it provides its service to the logistics provider).

All the transitions have associated a transition condition guard (a boolean expression that determines whether the transition can be actually performed given the state where the state machine is and the external and internal conditions) and a transition behaviour. Transition behaviours model the actual transition logic to be done besides moving from one state to another. They include (both for abstract and concrete protocols): raising `.indication` or `.confirm` primitives, raising events to concurrent processes, and instantiate concurrent processes. For concrete protocols they may also include: perform transformations on received message structures, generate message structures for transmission, and extract, maintain and manipulate information taken from message fields.



```
informReadyForCollection.request(params)/
msg = RN-EncodePIP3B2(params) //lower - DataMediation!!
startTimer(30mins)
sendMessage(msg)
```

```
receiveMsg(rmsg, RN_SIGNAL_MATCH(msg)) /
  if RN-Signal-Type(msg) == ACK {
    informReadyForCollection.
      confirm(Outcome=SUCCESS);
  } else { //RosettaNet Exception
    informReadyForCollection.
      confirm(Outcome=EXCEPTION);
  }
```

```
TimeOut/
startTimer(30mins)
sendMessage(msg);
```

```
TimeOut/
startTimer(30mins)
sendMessage(msg);
```

```
TimeOut/
startTimer(30mins)
sendMessage(msg);
```

```
Timeout/
informReadyForCollection.confirm(Outcome=INDETERMINATE);
```

```
rmsg, RN_PIP3C3_MATCH(conversation)) /
diation/Lift
ams = RN-Decode-3C3-Request(rmsg);
yment.indication(actionParams);
```

```
requestPayment.response(Outcome, responseParams) /
  if( Outcome==EXCEPTION ) {
    //Data mediation/lower
    msg = RN-Encode-Signal(EXCEPTION, responseParams)
  } else { //SUCCESS
    //Data mediation/lower
    msg = RN-Encode-Signal(ACK, rmsg);
  }
  sendMessage(msg);
```

```
receiveMsg(rmsg,
          RN_PIP3C3_MATCH(conversation)) /
  sendMessage(msg);
// resend previous signal
```

Fig. 1. Abstract and some concrete protocols in the logistics domain (adapted from Williams et al., 2006).

2.5 An ontology for describing abstract and concrete protocols

Fig. 3 and fig. 4 show different parts of the VSCL (Very Simple Coreography Language) ontology, which is available at *http://swws.semanticweb.org/*. This ontology can be used to describe the abstract and concrete protocols presented in the previous section, together with all their components, and is used to configure the protocol mediation component described in the next section.

As shown in fig. 3, choreographies are divided into abstract and concrete protocols. An abstract protocol specifies a set of roles that identify the role that a system is playing in an exchange of messages (logistics coordinator, freight forwarding service, etc.). Each role contains a set of communicative acts that are considered in the shared abstract protocol and that allow defining the shared conceptual model of the message exchange patterns to be followed by all the systems participating in a conversation. For each of these roles in each abstract protocol and with each specific implementation of any of the systems involved there is one role behaviour, that implements a set of concrete protocols that correspond to the behaviour that the actual system for the different communicative acts that are defined for it.

The admissible sequences of communicative acts are specified in what we call a process, whose common implementation will be a state machine, as we will see in the next figure. The primitives that are considered are those that were described when we discussed communicative acts: request, indication, confirm and response.

Finally, each concrete protocol contains one or more instances of RoleBehaviour. Each instance of RoleBehaviour declare a role that may be adopted by a peer to interact with the service provider agent via its interface. Each RoleBehaviour and carries a PrimitiveBinding for each RequestPrimitive and IndicationPrimitive associated with the role. This divides PrimitiveBinding into two subclasses, InitiatingPrimitiveBinding for binding instances of RequestPrimitive and ListeningPrimitiveBinding for bindings associated with instances of IndicationPrimitive. Each instance of PrimitiveBinding associates an instance of Process with the corresponding primitive. The Process(es) associated with an InitiatingPrimitiveBinding are instantiated when an admissible invocation of the corresponding RequestPrimitive occurs. The Process(es) associated with a ListeningPrimitiveBinding are instantiated either when the corresponding conversation is instantiated or as the conversation progresses and the IndicationPrimitive associated with the binding becomes admissible.

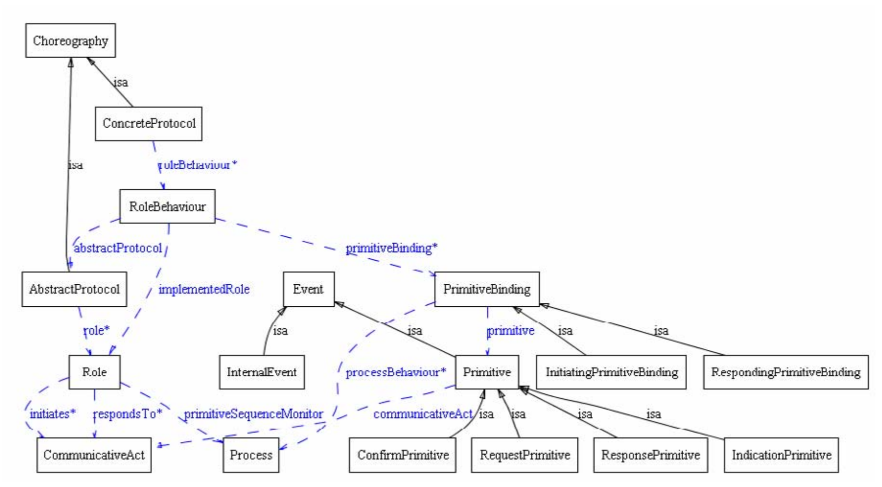


Fig. 2. Ontology excerpt related to abstract and concrete protocols and communicative acts.

Fig. 4 illustrates the classes used to represent state machines in VSCL. A state machine is a type of process that is composed of a set of states (some of which can be end states). Each state can have a set of associated transitions, which specify the next state, a set of guards and a set of transition behaviours. Transitions can be of different types, as described in the previous section (event driven, time driven, or message driven). The primitive driven transitions were already specified in fig. 3 as intiating and responding primitive bindings, since they are responsible for starting a state machine.

Transitions behaviours are of different types, as pointed out in the previous section. From them, the most relevant is the script, which can be provided by a reference to a URL (external) or as part of the instance values (internal). We will analyse them in more detail later, when we discuss the component API.

In our logistics application we have a state machine for each of the protocols aforementioned.

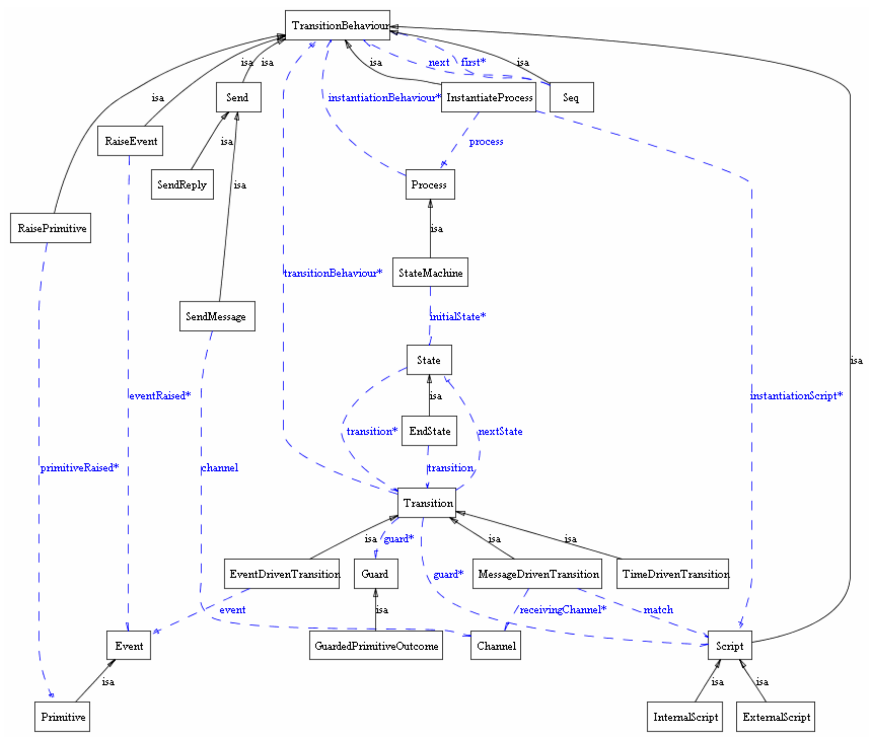


Figure 3. Ontology excerpt related to the state machine descriptions.

In summary, in our logistics application we have the following instances of this ontology (available at *http://swws.semanticweb.org/*):

- One abstract protocol with two roles defined for it: FreightForwardingServiceConsumer and FreightForwardingServiceProvider.
- 14 processes (state machines) for concrete protocols.
- Six communicative acts: InformReadyForCollection, RequestShipmentStatus, InformShipmentStatus, InformReadyToDeliver, InformShipmentDelivered, and RequestPayment, with their corresponding primitives (four for each of them).
- 10 event driven transitions with 20 scripts for their transition behaviours.

3 Solution Details: The SWWS Protocol Mediation Component

Here we provide a general description of the protocol mediation component architecture and of important implementation details, including a broad description of the component API, so that it can be used in other similar applications with protocol mediation needs.

Though the usual deployment of a protocol mediation component would be as part of the communication infrastructure between services in a service-oriented application, in our case this component has been deployed as shown in fig. 5: A consumer application incorporates the protocol mediation component inside its environment in order to control the exchange of messages with the provider application. In our logistics application, the selection of one system or another as consumer or provider is arbitrary. Our decision has been to use the logistics coordinator as a consumer and the freight forwarding service as a provider.

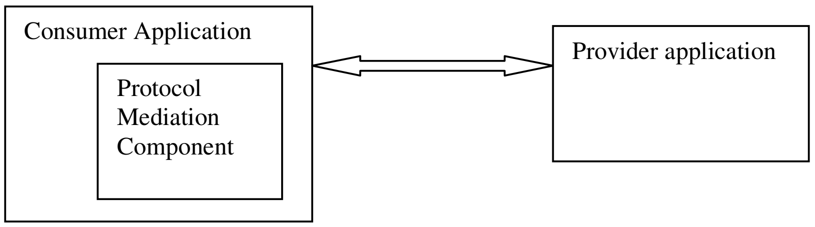


Fig. 4. Location for the deployment of the protocol mediation component.

The protocol mediation component has 5 essential subcomponents, which are described in detail in the next sections:

- **Local agent** (package com.isoco.swws.conversation.local_agent). It is the subcomponent directly used by the final user. Basically, the component allows creating conversations, initiating them in an active or a passive mode and later, by means of the ConversationManager, explicitly invoking the different CommunicativeActs and tracing the interactions with the remote conversation partner.
- **Protocol** (package com.isoco.swws.conversation.abstractprotocol). It is the internal representation of the protocols (either abstract or concrete) that rule the conversation. This is based on the ontology described in the previous section.
- **ChoreographyHandler** (package com.isoco.swws.conversation.mediation.vscl). It is the bridge between the application and the external choreography that is included in the VSCL ontology.
- **Message transfer plugin** (package com.isoco.swws.conversation.plugins). Internally, a specific communication protocol (HTTP, SMTP, etc.) is used for the communication between the consumer and the provider. This plugin serves as an interface for the protocol. This implementation of the component includes an HTTP plugin, but other plugins could be easily created and deployed.
- **Rhino facilites** (package com.isoco.swws.conversation.environment). They are used to execute the Javascript scripts included in the choreography. The mechanism used in the component is Rhino (Mozilla) and there is an abstraction layer to ease its use and to adapt it to the application needs.

3.1.1 Local Agent

The local agent groups the collection of classes that the Consumer needs to create and control a conversation. A conversation is initiated with the creation of a *ConversationManager*, which receives the following parameters in its constructor:

- A set of roles (the systems involved in a conversation). The InterfaceRole contains the *remoteInterface*, the URL that holds the address of the conversation's partner, and the *localRole,* the URL of the role adopted by the local agent with respect to the choreography and this conversation.
- The URL where to find the choreography (that is, the place where the VSCL ontology instances are stored).
- An indication handler, which is used in the case that an external system has to contact this system or send it and event. Normally this handler is used when the system receives a message from the provider that causes a <CommunicativeAct>.indication. This is the way that the protocol mediation component has to inform an application that an indication has arrived. It is also responsibility of the IndicationHandler to respond to the indication of the CommunicativeAct. Responding to the .indication means to model the .response. The user must calculate the outcome and the results of that CommunicativeAct.

The implementation of the IndicationHandler is robust enough to deal with situations where it could be blocked or fail, where the response will be launched again.

A conversation consists in the coordinated exchange of communicativeActs. The local agent can send CommunicativeActs either in a **synchronous** or an **asynchronous** way. In the synchronous exchange the communicative act is sent and the system waits for the confirmation of the remote partner. In the asynchronous exchange the communicative act is launched and the control is returned back to the application. When the confirmation from the user is received, the *confirm* method of the ConfirmHandler interface that has been specified as a parameter is invoked.

The creation of a new ConversationManager implies the following tasks: initializing the abstract and concrete protocols and initializing of the ChoreographyHandler for the successive uses of the choreography. A conversation that has been created can be initiated in two modes: **active** and **passive**.

- In the active mode, the Consumer can invoke the *synchSay* and the *asynchSay* methods (for synchronous and asynchronous exchanges of messages) to start the exchange of CommunicativeAct with the remote partner.

- In the passive mode, the *listen* method must be invoked to initiate a conversation in a passive mode. This action prevents the use of the *synchSay* and the *asynchSay* methods and the conversation waits for an indication from the remote partner. It should be noted that once the *listen* method is invoked, the conversation will only be activated by a remote message from the partner. There is no explicit method to transfer the conversation to the active mode.

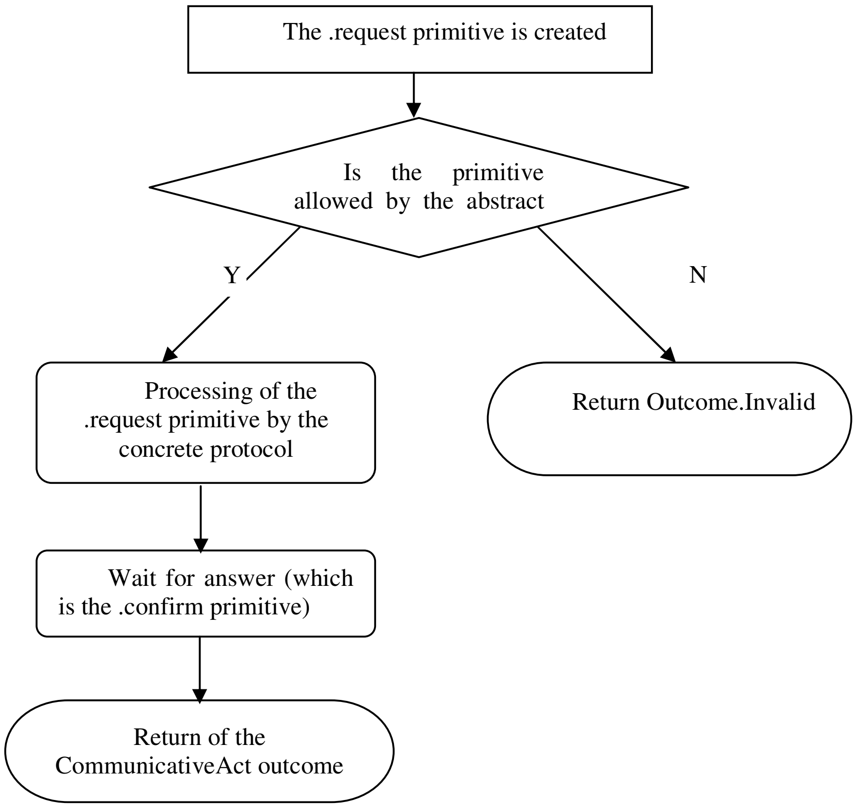

The .request primitive is created
Is the primitive allowed by the abstract
Y
N
Processing of the .request primitive by the concrete protocol
Return Outcome.Invalid
Wait for answer (which is the .confirm primitive)
Return of the CommunicativeAct outcome

Fig. 5. Usual process followed for a communicative act being sent.

Fig. 6 shows how this works in an active mode: the .request primitive is created for a CommunicativeAct. This primitive is sent to the abstract protocol to know if the CommunicativeAct can be initiated in the current context of the conversation.

- If it cannot be initiated, the execution is aborted and Outcome.INVALID is returned to the entity to inform that it is impossible to execute that action in the current situation.
- If it can be initiated, the primitive is sent to the concrete protocol in order to execute the set of scripts and other relevant actions associated to this primitive. It is important to emphasize that the primitive is sent, that is, there is no explicit communication from the abstract protocol to the contrete protocol. The idea is that the abstract protocol allows executing the primitive but it does not consumes it.
  Afterwards, we wait to receive the .confirm primitive and the Ouctome associated to the CommunicativeAct of the primitive is returned. The outcome can be: Outcome.SUCCESS, Outcome.EXCEPTION, or Outcome.INDETERMINATE.

When the entity is waiting for an indication, the process is different. When a message arrives, it is evaluated in the MessageDrivenTransitions of the active processes of the concrete protocol. If any of them matches, that transition is executed and it will be its responsibility, among other responsibilities, to launch an .indication primitive to the abstract protocol to check if in the context of this conversation that primitive is allowed. If the primitive is allowed, the entity will be informed about it by the indication method of the IndicationHandler.

3.1.1.1 Multiple conversations

The exchange of messages between the consumer and the provider is executed in a multiple simultaneous conversations scenario. To know which conversation should process each message, the protocol mediation component associates a unique conversation id to each ConversationManager.

Whenever a conversation is initiated by a partner, a message is sent with a paraneter that informs that it is a new conversation. A new conversation id is created for this conversation and the following messages of this conversation must include this id.

The ConversationDispatcher class is responsible for registering all the existing conversations. Basically there are two lists: a list of active conversations and a list of potential conversations (those that are waiting to receive a message to start, by the invocation of the method *listen*). When a message to start a conversation arrives, all the conversations that are waiting are checked to inquire which one can process it. If a conversation can process it, that conversation is moved to the list of active conversations.

The ConversationDispatcher is also responsible for initializing all the available plugins once a conversation starts.

3.1.2 Protocols

Conversations are ruled by a choreography, which contains two types of protocols (abstract and concrete). Both protocols are specified by means of the ontology described in section 2.5. For each class in the ontology there is a Java class in this package, including the states and transitions.

Each ConversationManager has references to its own abstract and concrete protocols. When a conversation is created, the ConversationManager loads the initial processes with all their associated structure, using those Java classes (as explained in the following section). The list of processes and active states is updated as the transitions are executed.

Transitions are modelled with the `Transition` Java class and its subclasses. The following methods are called for a transition:

- Evaluate initTransition. This function must be redefined by all the subclasses of Transition. It has two responsibilities: verify that the object that it receives is the instance that it knows how to process. For example, the EventDrivenTransition must guarantee that the object type is 'Event'. Not only must it guarantee that it has the correct type, but also that it is the instance that sets off the transition (for example, that it is the RequestShipmentStatus.request primitive). Its other responsibility is to initiate whatever is necessary to execute the transition. For example, to set some variable in the RhinoEnviroment (section 3.1.5) or some other task.
- Evaluate initGuard. The transitions can have an associated guard that must be satisfied to continue with the transition. In general, it is a method that does not have to be redefined by the subclasses.
- Execute doBehaviours. As a consequence to the execution of the transition, a set of TransitionBehaviours must be executed. These behaviours represent what the transition does. This method should not be modified. As we will see in section 3.1.5, transition behaviours are specified in Javascript and executed by the RhinoEnvironment.
- Execute advanceToNextState. A change to the new state is performed in order to end the execution of a transition. This process entails several tasks such as the loading of all the structure of the new state from the choreography, the initialization of the associated TimeDrivenTransitions, etc.

3.1.3 Choreography Handler

It serves as a bridge between the application and the choreography. It is used to create instances of the classes included in the Protocols package from the choreography information available in a URL. As aforementioned, the whole choreography is not loaded completely from the start but incrementally according to the transitions done through the abstract and concrete protocols. Two significant methods from this class are:

- o *createProcessByName*, which creates a state machine from the information available in its location (URL). It returns the state machine and all the structure associated to it (states, transitions, transition behaviours, scripts, etc.).
- o *createStateByName*, which creates a state from its name (URI). It returns the state and all the structure associated to it (transitions, transition behaviours, scripts, etc.).

This component uses the KPOntology library[2] to navigate the RDF graph that models the choreography.

3.1.4 Message transfer plugin

This component deals with the specific communication protocol (HTTP, SMTP, etc.) used for the communication between consumers and providers. An HTTP plugin is included with the current implementation, and other plugins can be also created.

The HTTP plugin provided is made up of the HttpPlugin class and an auxiliary web application that manages the queue of received messages, with two services:

- Receive a message. This service is used when a remote partner, e.g. the provider, must send a message to the consumer. The web application receives the message and puts it in the queue of received messages.
- Recover the message. This service allows the HttpPlugin class to recover the messages received from the web application

The HttpPlugin class has two main objectives:

- Send messages to remote partners, using the sendMessage method. This method receives a remote address where to send the message, the conversation id, and the message.
- Transfer messages from the web application to the component. The HTTP plugin has a thread that is constantly polling the web application for the arrival of new messages.

The Web application always responds to the petition of messages by means of an XML that contains the following elements:

- conversationId: id of the conversation under way.
- newConversation: it indicates if it is a new conversation.
- Message. Depending on the types of message, it will have different types of structures. For instance, in the case of the RosettaNet messages, it will be divided into: “Preamble”, “DeliveryHeader”, “ServiceHeader” and “Content”.

It is the responsibility of the plugin to find the appropriate Conversation Manager from the conversation id, to build the internal structure of the protocol for the representation of the messages and to send the resulting message to the Conversation Manager for its processing.

3.1.4.1 Messages and Filters

All messages are vectors of XML structures, so that they can accommodate multi-part messages that are typical in B2B interactions. The underlying messaging system plugins are responsible for encoding/decoding between the internal XML structures (typically XML DOMs or more abstractly XML Infosets) and the packaged and encoded wire format - this includes XML validation of inbound messages against the relevant DTDs and/or XML schema. Directly or indirectly the concrete interface descriptions MUST provide message DTD/Schema and lift/lower transformations.

In addition, received message structures also carry low-level connection and endpoint information. Typically this will not be used directly in processing the message, but is essential for the plugins to correctly formulate a response message - in particular if a response/reply needs to be returned on the same connection as a given received message.

Message are filtered and classified accordint to the various pieces of typing information that they carry: internet media type, XML DOCTYPE and XML root element type of the primary part of the message; and identification of the endpoint via which they were received. This associates a received message with a collection of processes which serve messages of a given kind. Concrete Role behaviour descriptions contain a static description of the message types they are able to receive.

Messages with the same conversation id are bound to a particular conversation and queued to be processed by the concrete role behaviours associated with that process - in particular messages are consumed by message driven transitions. When a message matches a message filter in the initial transition of a listening role behaviour a factory behaviour is invoked which instantiates a new instance of a conversation (controller) and passes that new message to that controller - a new conversation id value becomes associated with the new conversation.

So coarse filtering is used to associate messages with a class of conversational role where they may either be queued at an existing conversation or used to instantiate a new conversation. Messages queued at a conversation are then visible to the processes that realise the concrete role behaviours for that conversation. As discussed earlier these may or may not be processed in strict arrival order.

3.1.5 Message filtering

This component eases the use of Rhino, the Javascript interpreter used by the protocol mediation component to express message filters, transition pre-conditions and some (scripted) transition behaviours. Each process specified in the choreography has a Rhino Environment, and each environment will have a defined scope. This scope has a set of variables and functions defined in the scripts. In this way, the processes do not share the execution environment when they execute the scripts.

The abstraction layer of Rhino is achieved through the RhinoEnviroment class. The most distinguishable of its functions are:

- execute, which receives a script as a parameter and executes it.
- match, which receives a script that returns a boolean value, executes it and returns that boolean value.
- setMessage, which receives a variable name and its value, and is in charge of creating in the Javascript environment a variable with that value.
- getMessage, which returns the value of a variable name in the Javascript environment.

3.1.6 Deployment and installation

The protocol mediation is a framework designed to be used by a client application. The typical scheme for its use would be:

- Initialize the ConversationDispatcher.
- Create a ConversationManager, specifying the choreography, the participating agents and the Indicationhandler. The implementation of the IndicationHandler must guarantee that all the possible .indication communicative acts that the remote partner can send are processed and for each one of them, it must compute the Outcome and the adequate results.
- Initiate the exchange of CommunicativeActs with the remote partner.

Next, we show an example on how to use the component. The objective of this example is to give a guidance on the use of the component. The typical use must be by means of an application that should keep the evolution of the conversation as well as the CommunicativeActs that have been sent and received by the remote partners.

```
String logisticsNamespace = "http://swws.semanticweb.org/logistics#"
ConversationDispatcher.init("http://consumer:8080/");
interfaceRole = new InterfaceRole( new URI("http://provider:8080/"),
      new URI(logisticsNamespace + "FreightForwardServiceConsumer"));
IndicationHandlerImpl indct = new IndicationHandlerImpl();
ConversationManager conversationManager = new ConversationManager(
      new InterfaceRole[]{interfaceRole},
       new URI("http://swws.semanticweb.org/logistics.owl"), indct);
CommunicativeAct communicativeAct = new CommunicativeAct(
 new URI(logisticsNamespace + "InformReadyForCollection"));
conversationManager.synchSay(communicativeAct);
communicativeAct = new CommunicativeAct(
 new  URI(logisticsNamespace + "RequestShipmentStatus"));
conversationManager.synchSay(communicativeAct);
```

The first thing to do is the initialization of the ConversationDispatcher. This initialization also includes the initialization of the plugins. In the previous example, the URL is the address of the local web application that uses the HTTPPlugin.

The second thing to do is the creation of the ConversationManager. In the previous example we talk to the partner that we can reach at "http://provider:8080/". In the conversation we adopt the role of the FreightForwardingServiceConsumer. The choreography is found in `http://swws.semanticweb.org/logistics.owl`. We also have the IndicationHandlerImpl which is an implementation of the IndicationHandler.

Afterwards, a CommunicativeAct is created (in this case: InformReadyForCollection) and we send it in a synchronous way.

To keep the example simple we do not send any parameter in the comunicativeAct, but it would be usual practice.

4 Alternatives, Cost and Benefits

The proposed solution to protocol mediation between heterogeneous applications can be applied not only to the logistics domain, which is the one that has been described in this paper, but also to other similar domains where applications are already deployed and have to interoperate with each other in order to support a specific set of added-value functionalities.

While work on the area of data mediation in service exchanges is quite widespread and there are tools available in the mainstream market for solving these issues, most of the approaches for protocol mediation have been based on ad-hoc solutions that are tightly related to the applications where they are being applied. No easy configurable toolkit exists yet for solving this problem, hence the main alternative for the work proposed here is to create an ad-hoc solution that solves the interaction problem between applications or services for a specific set of functionalities.

Though our approach still requires a lot of effort to be done, and requires more maturity and further evaluations to be applied in production systems, the main advantages with respect to the current state of the art are related to the reusability of the abstract representations of message exchanges for each of the systems involved, as well as the reusability of message filters across different types of applications, what can benefit the agility of developing new added-value applications in the future. Besides, the model is easily extensible and fully declarative, what influences in the lowering of maintenance costs.

5 Conclusions and Future Trends

In this paper we have motivated the need to use some form of protocol mediation to make it possible to different systems in the logistics domain to communicate successfully with each other, even if they use different protocols (RosettaNet and EDIFACT). Furthermore, we have described the approach for protocol mediation developed in the context of the SWWS project, including the ontology used to describe the choreography (that is, how the systems interact with each other) and the software that implements the component that has been developed.

Though this is a first approach to solve the protocol mediation problem between systems, there is still much work to be done in the future to convert this prototype into a production-quality component. Among them, we have to add new message transfer plugins to allow message transfer using other communication protocols, such as SMTP, FTP, etc., which is what it is used by many of the current systems. Besides, a tighter integration and evaluation with existing systems has to be provided, and a library of common interaction patterns should be also implemented, so that the task of protocol mediation is as simple as possible for those developers that want to develop a mediation solution for their systems.

Acknowledgements

This work was supported by the EU under the SWWS and DIP consortia. In addition we want to thank Stuart Williams, for his ideas that have been the basis for this work and for providing good comments to improve this chapter, to Jorge Pérez Bolaño for making part of the component implementation, to Juan Miguel Gómez for the work on the VSCL ontology, and the other members of the consortium, who contributed to the use case and to the ideas presented here.

References

Evans-Greenwood P, Stason M (2006) Moving Beyond Composite Applications to the Next Generation of Application Development: Automating Exception-Rich Business Processes. Business Integration Journal, May/June 2006.

Preist C, Esplugas-Cuadrado J, Battle SA, Grimm S, Williams SK (2005) Automated Business-to-Business Integration of a Logistics Supply Chain Using Semantic Web Services Technology. In : Gil et al. (eds) Proceedings of the 4th International Semantic Web Conference (ISWC2005). Lecture Notes in Computer Science, Volume 3729, Oct 2005, Pages 987-1001

Williams SK, Battle SA, Esplugas-Cuadrado J (2006) Protocol Mediation for Adaptation in Semantic Web Services. In : Domingue and Sure (eds) Proceedings of the 3rd European Semantic Web Conference (ESWC2006). Lecture Notes in Computer Science, Volume 4011, June 2006, Pages 635-649

EDIFACT. ISO 9735, Electronic data interchange for administration, commerce and transport (EDIFACT) – Application level syntax rules. 2002. International Standards Organisation.

RosettaNet Implementation Framework : Core Specification Version 2.00.01. March 2002. http://www.rosettanet.org/

AnsiX12. National Standards Institute Accredited Standards Committee X12

Cimpian E, Mocan A (2005) Process Mediation in WSMX. WSMO Working Draft D13.7 v0.1. http://www.wsmo.org/TR/d13/d13.7/v0.1/

Additional Reading

We recommend reading WSMO deliverables about mediation, in general, and about process mediation in particular. They can be found at http://www.wsmo.org/. Efforts on process mediation are also being done in the context of the SUPER project (http://www.ip-super.org/).