Distributed Print on Demand Systems in the Xpect Framework

Jean-Marc Andreoli and François Pacull

Xerox Research Centre Europe, Grenoble
6 chemin de Maupertuis, Meylan, France
{jean-marc.andreoli,francois.pacull}@xrce.xerox.com
Tel: +33 4 76 61 50 21, fax: +33 4 76 61 50 99

Abstract. The Internet is an extremely rich source of online information and services. However, complex client requests, involving and combining several types of services, are difficult to handle without some form of support. This is particularly true in the case of electronic commerce, where complex transactions may involve several independent good providers, bankers, delivery services, etc. Hence the need for "brokering" services, whose offers combine in the best possible way offers coming from existing, specialized services publicly available on the Net, in order to match customers' constraints. The XPect framework for electronic commerce has been developed for that purpose. In this paper, we illustrate it through a case study in the context of distributed print-on-demand. We propose an architecture and implementation of the case study based on CLF, a distributed application development tool relying on a rich object model and its corresponding coordination scripting facility.
Keywords: Print-on-demand, multi-agent negociation

1 Introduction

The Internet is a fast growing infrastructure which tends to make available online on most desktops a tremendous number of independent services. However, finding and accessing such services is a major task for a non expert user of the Web, and combining them together in order to satisfy complex requests is an even bigger challenge. Hence the need for "brokering" services, helping the user perform complex tasks involving and combining several complementary or competing services. Typically, brokers do not implement the services they combine, but make use of publicly available ones in order to provide users with combinations which best match users requirements.

Electronic commerce is a typical domain where such brokering facilities are essential. The challenge here is to combine existing services such as good providers, bankers, delivery facilities, etc. within a single, high-level service directly available to customers through a standard Web browser. For this purpose, we have developed the Xpect framework [2], which provides support for the coordination of various actors in the electronic commerce field. We illustrate it here by a case study in the domain of distributed print-on-demand systems.

Xpect quite naturally models each electronic commerce actor as an object offering specific interfaces, thus abstracting away how each individual service is implemented. The point here is in the coordination of the various services, not their implementation. However, the object model on which Xpect relies is richer than the traditional one, which only supports the basic invocation-reply protocol. Xpect is built over the Coordination Language Facility [1] (CLF), which defines a rich object interaction protocol together with a high level scripting language specifying coordination behavior among objects supporting this protocol. In particular Xpect makes use of two crucial features of the CLF model:

Dynamicity : a service, defined in the interface of an object, can dynamically propose new offers, even after it has been invoked. For example, it may offer several ways to perform a task, with different completion date and prices (e.g. "I can do this by tomorrow for $100, by the day after for $60, and in one week's time for $40"). These different offers may occur asynchronously any time after the invocation. The CLF protocol contains features, at the basic object level, supporting such kind of interactions.

Multi-party negotiation : typically, a customer request results in a negotiation with multiple services, so as to achieve the best agreement. For example, the customer may gather offers from different providers and delivery facilities, and possibly choose an item from a more expensive provider if the incurred cost of delivery is lower. The CLF coordination scripting language typically supports such behaviors.

Section 2 describes an electronic commerce scenario in the context of "print-on-demand" books. Our goal is to design the architecture of an electronic commerce application supporting such a scenario, and, in particular, including a broker capable of processing complex print-on-demand requests. Section 3 gives a quick overview of the CLF, the tool used in our Xpect framework to design and implement such an architecture, described in Section 4.

2 Case Study: A Distributed Print on Demand System

As a case study, we consider a distributed print on demand scenario that goes beyond the traditional centralized scheme where only one entity is responsible for the brokering, the storage, the printing, the delivery and the payment management. Indeed, all these services require a high level of competency which cannot be realistically handled by a single service provider.

Consider a customer living in London, who would like to offer to her daughter, for her birthday, two French books: *"Le Petit Prince"* by *A. de St Exupery* and the French version of *"Jonathan Livingston Seagull"* by *R. Bach*. The birthday is in three day and, if it is not possible to obtain both books on time, the customer would probably have to find another gift.

Being used to shopping on the Web, the customer first contacts a *digital library broker* in order to find one or more digital libraries owning such titles. As a result, three libraries return offers (all in digital form). Library lib_A offers both

books, lib_B only the first one and lib_C only the second. More specifically, $book_a$ (i.e. *"Le petit prince"*) provided by lib_A contains color pictures inside, while lib_B provides only a black and white version. The book $book_b$ (i.e. *"Jonathan Livingston Seagull"*) only contains black and white pictures in both offers from lib_A and lib_C. Libraries lib_A is the most expensive in both cases, but it is also the only one to propose a color version of $book_a$. Finally, the customer decides to buy $book_a$ from lib_A and $book_b$ from lib_C.

The second step is to find print shops able to print the two books. The time constraint (2 days before the birthday) implies that either the books have to be printed near London or at a location from where it is possible to have them delivered in 48 hours by a fast courier company. The *print shop broker* is first asked for printing facilities around London for both $book_a$ and $book_b$. We assume that only one print shop PS_A makes an offer. It is located near the City but proposes only black and white printing facilities and has no color facilities. It could therefore handle $book_b$, but not $book_a$. The broker then is asked to enlarge the search to other print shops offering color facilities for $book_b$. A print shop PS_B located in the USA makes an offer, together with another print shop PS_C located in France. The latter is more expensive, but closer.

The third step is to collect offers for the delivery of $book_b$ either from the USA or from France, and to combine them with the print offers in order to respect the 48 hours time constraint. The courier companies del_A and del_B offer delivery from both USA and France. We end up with four distinct solutions for $book_b$:

- Print from PS_B and deliver by del_A (from USA)
- Print from PS_B and deliver by del_B (from USA)
- Print from PS_C and deliver by del_A (from France)
- Print from PS_C and deliver by del_B (from France)

Finally the chosen solution is to print the book at PS_C and to contract delivery with del_B.

As a final result, the financial transaction involves lib_C, PS_A for $book_b$ and lib_A, PS_C and del_B for $book_a$. The last step, but not the least, is to combine the different payment systems the providers accept and the customer would use in order to finalize the commercial transaction. If, for any reason, one of the partners in this transaction fails to support its offer, the whole transaction should abort.

This whole scenario could of course be executed manually by the customer, who would then have to access each service separately. The purpose of a generic brokering service is precisely to relieve the customer from this burden, and to offer a high-level service which coordinates all the others according to the customer requirements (and under her control). To implement such a service, we assume an infrastructure based on the CLF, described below.

3 Quick Overview of the CLF

The CLF ("Coordination Language Facility") is an object-based distributed application development tool. It has been described in details in [1], and the purpose

of this section is only to give a quick overview of its functionalities so as to make the paper self-contained. CLF assumes an object model in which objects are autonomous agents which may engage in more sophisticated interactions than in the traditional object paradigm.

3.1 The CLF Object Paradigm

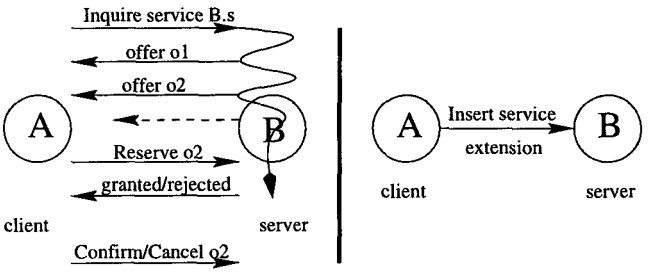

Fig. 1. The CLF interaction model

In the traditional paradigm, a client object A invokes a method on a (possibly remote) server object B which executes the method and returns the reply to object A. Object A may suspend its activity while object B processes the request, or may spawn a thread waiting for the reply while normal execution goes on, with the possibility of joining the spawned thread and waiting for the reply.

In the CLF paradigm (figure 1), agents are seen as service providers. Three modes of interaction are supported by services and can be combined:

Negotiation : A client agent A retrieves service offers matching a certain profile from a server agent B (operation **Inquire**). Agent B returns a handle representing a possibly infinite stream of answers, each of them specifying an offer. Agent A may then retrieve individual offers from the handle (operation **Next**). Agent A may asynchronously terminate an inquiry, meaning that it is not interested any longer in a service profile it requested (operation **Kill**). Agent A may also asynchronously check whether an offer it received is still valid (operation **Check**).

Performance : A client agent A requests the execution of a service on a server agent B, according to an offer returned during a negotiation, as explained above. This performance phase is in fact decomposed in several steps. Agent A first reserves, if possible, the resources required to execute the service as described in the offer (operation **Reserve**). A reservation may be either accepted by B or rejected. In the latter case, rejection may be hard, meaning the offer is no longer valid, or soft, meaning the offer is temporarily disabled, but may be retried later. Accepted reservations may then be either confirmed or cancelled by A (operations **Confirm/Cancel**).

Notification : A client agent A requests the modification of the services offered by a server agent B (e.g. addition of a new service offer). Agent B may choose whether and when to process the request, so that agent A must not expect an answer (operation **Insert**).

This extended interaction paradigm is formalized in a protocol defined by 8 interaction verbs similar to KQML performatives [4]. It allows on the one hand flexible dynamic redefinition and refinement of services and, on the other hand, multi-party agreement between several service providers to fulfill a complex client request.

3.2 CLF Agents

A CLF agent can be implemented in any language, and can encapsulate any kind of resources, as long as it accepts the CLF protocol. The interface defines what is visible of these resources. It consists of:

Method declarations : which allow access to the agent through the standard method invocation protocol. Method execution requires values for the input parameters of the method, can perform any kind of modification on the resources of the agent, and then returns values for the output parameters. The support for this interaction is the HTTP protocol, for which most programming languages have libraries. When the input parameters are small, they can be encoded in a URL, so that it is possible to invoke the method directly from any Web browser and display the results, e.g. in HTML. This makes it easy to rapidly add a basic graphical user interface to any CLF application.

Service declarations : which allow access to the agent through the CLF protocol. A service declaration can be viewed as declaring a property of some of the resources of the agent. The Inquiry operation on a service requires values for the input parameters of the service. It does not modify the resources of the agent, but each service offer in the stream returned by the Inquiry consists of (*i*) an assignment of the output parameters and (*ii*) an action capable of removing a resource satisfying the property with the assigned parameters (both input and output). The other operations of the protocol then just deal with the action id attached to each offer.

Different kinds of prototypical agents have been designed for CLF applications. The most important is the *coordinator*. A coordinator is an agent whose resources are coordination scripts specified as production rules, and which enacts these rules. The enactment of a CLF rule consists of

- a negotiation between different agents on a set of service offers, as specified by the rule;
- the atomic execution of the services according to the agreed offers;
- the notification of the success of the transaction to different agents specified by the rule.

Since coordinators are themselves CLF agents, they may be part of a coordination which manipulates their rules, thus making the system fully reflexive.

3.3 A sample CLF script

The aim of the following CLF script is to illustrate with a toy example the syntax and semantics of CLF rules. More realistic rules are given in Section 4.

The basic building block of a CLF script is the token. Tokens are used to access CLF services. For example, the token **p(x,y)** specifies an access to a service locally named **p** with one input argument **x** and one output (underlined) argument **y**. Invocation of such a service requires a value for **x** and produces, with each offer, a value for **y**. The name **p** is local to the coordinator and must be assigned to a real service provided by the coordinator. By default, a coordinator provides basic computational services, and surrogates of remote services identified by a name looked-up in an application-wide name service.

```
Rlocation(where) @ Jlocation(where) @ Rhand(thing)
<>- Jhand(thing) @ Rlocation1( "Ground") @ Jlocation( "Room")

Jhand(thing) @ 'Jlocation( "Room") @ equal(thing, "Rose")
<>- Jmood( "in love") @ Vase(thing)

Ladder( "Crash") @ Rlocation( "Balcony") <>- Rlocation( "Ground")
```

The first rule expresses a rendez-vous between Juliet and Romeo since the location **where** held by **Jlocation** is shared with that held by **Rlocation**. Moreover, Romeo should have something in his hand to complete the rule instantiation. For example, let

```
Rlocation1( "Balcony") @ Jlocation( "Balcony") @ Rhand( "Rose")
```

be the result of the instantiation. The transaction phase is enacted and as a result the *"Rose"* and the **location** of Romeo and Juliet are removed from their respective services and new resources are inserted: *"Ground"* in **Jlocation**, *"Room"* in **Jlocation** and *"Rose"* in **Jhand** (i.e. Juliet takes the rose and goes into the room while Romeo goes down the ladder).

The second rule may now be instantiated since the service **Jhand** has a resource verifying **Jhand(thing)** where **equal(thing,"Rose")** is verified (**equal** is assigned to a basic computational service). Thus, the transaction is enacted and consumes all the involved resources except the location of Juliet because the operator ' placed before the token **Jlocation** means that the resource has to be present for enacting the transaction but it is not effectively consumed during the commit operation. New resources are inserted: a *"Rose"* in service **vase** and *"in love"* in service **mood** of Juliet.

The third rule illustrates, in addition to synchronization and transactional operations on independent services, non determinism and competition on resources. Indeed, if an external event such as the break of the ladder (materialized by the resources **Ladder(*"Crash"*)**) occurs, then Romeo will fall down and the resource *"Balcony"* will be consumed by the transaction while the new resource *"Ground"* will be inserted as his location. So, if no resource *"Crash"* is available, Romeo will meet Juliet on the balcony. Conversely, if the resource is present,

rules 1 and 3 will compete for the resource *"Balcony"* and in case rule 3 "wins", Romeo never meets Juliet.

4 Distributed Print on Demand System in XPect

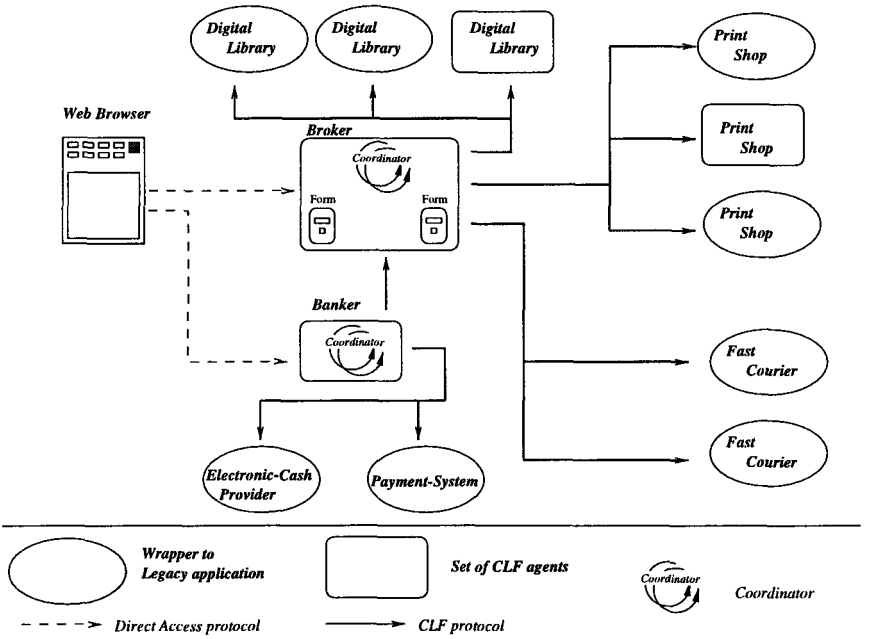

Fig. 2. Overall architecture

We now sketch the design of an electronic commerce application in the context of distributed print on demand systems. We consider that all the required services are managed by separate providers with specific expertise and/or equipment. External service providers are accessible in the XPect framework through CLF wrappers, and can thus participate in CLF coordinations. The components may be invoked either through direct methods mainly for GUI purpose (dashed arrows in figure 2) or through the CLF protocol (plain arrows). The components, spread over the Web, are:

- digital libraries handling various types of documents;
- print shops offering various quality of services in terms of delay (e.g. page-per-minute), printing (e.g. color, dot-per-inch), and service (e.g. binding);
- fast delivery companies responsible for delivering the printed books from the print shop to the customer site;
- a payment mediator and several payment systems handling various modes of payment: electronic cash, credit cards, checks;

– a broker offering a single entry point to the customer.

As described in the scenario of section 2 a typical session may be decomposed in several steps. Each step requires a specific high level coordination scheme (i.e. *Search, Negotiation, Transaction, Mediating and Tracking*). These schemes are described with CLF rules enacted either by the *broker* coordinator or by the *payment mediator* coordinator.

4.1 Search Phase

Let us first illustrate how a simple search in a single digital library may be handled. It is triggered via a Web browser offering a simple form to the customer. The latter fills the form and submits it. This produces the resource `("Jonathan Livingston Seagull","french")` in the service **simpleSearchRequest** of the **broker** agent. The three services **title**, **version** and **bookInfo** belong to the digital library agent, the service **bookMatching** is provided by the **broker** and holds the result of the search.

```
  simpleSearchRequest(title,version)
@ 'title(ref,title) @ 'version(ref,version)
@ 'bookInfo(ref,description,pp,quality,price)
<>-
bookMatching(ref,description,pp,quality,price))
```

Thus, the service **simpleSearchRequest** retrieves the requests coming from the customers, then collects all the book reference **ref** corresponding to the **title** (i.e. "Jonathan Livingston Seagull"). The service **version** acts as a filter that keeps only the French version of the books. Finally the service **bookInfo** is responsible for fetching all the information related to the book matching the request.

Once instantiated, the resources are atomically removed. However, in this case only the resource from **simpleSearchRequest** is actually consumed, since the last three tokens in the left hand side of the rule are tagged with '. A resource describing the matching book is inserted into the service **bookMatching** containing all the information required for the rest of the process (i.e. reference of the book, human readable description, number of pages, quality required for the printing and price).

However, this simple rule is only able to search into one digital library and returns at most one offer. To extend the search capability, we rely on a special type of service called *dispatch*. The first two arguments of a dispatch service are input parameters and denote, respectively, the name of an object to be looked up in the name service, and the name of a service defined in the interface of that object. Thus, when a token is bound to a dispatch service, its occurrences in a rule will in fact invoke remote services on objects specified by the instantiation of the rule. In other words, dispatch services allow late (dynamic) binding of the tokens. For commodity, the first two parameters of a dispatch token are written

in square brackets after the token name. If the second parameter is identical to the token name, it is omitted.

The following rule uses the dispatch mechanism to allow search within a set of digital libraries that may be dynamically changed, and allows each of them to return several offers.

```
  'searchRequest(title,version,rLocation)
@ 'digitalLibrary(dl) @
@ 'title[dl](dlRef,title) @ 'version[dl](dlRef,version)
@ 'bookInfo[dl](dlRef,description,pp,quality,dlPrice)
@ currencyConverter(dlPriceLocalCurrency,dlPrice,rLocation)
<>-
bookMatching(dl,dlRef,description,pp,quality,dlPriceLocalCurrency))
```

Basically, the service **digitalLibrary** is provided by the **broker** agent. Every digital library agent that would like to be involved in the search has to register itself in this service by inserting an appropriate resource (namely the agent name). Each digital library agent also has to provide the services **title**, **version**, **bookInfo** as previously. These services, specific to each digital library agent, are accessed through *dispatch* services. It is possible to dynamically add new digital libraries to the system without changing the rule nor interrupting the system. Indeed, the resources proposed by the service **digitalLibrary** define the scope of the search. The service **currencyConverter** is a simple wrapper of a Web service that translates a price into the local currency. It holds no resource and has a void behavior during the transaction phase. The service **bookMatching** is the same as previously, except that the library agent name is also stored in the tuple. Another difference with the previous rule is the ' tag before the token **searchRequest**. This prevents the consumption of the resource defining the request and allows multiple offers satisfying a request to be stored.

In the scenario of section 2, four offers are received (at least: other offers may occur later), so the service **matchingBook** now contains:

- ("lib_A", "1-567-789-6", "le petit prince, ...", "75", "color", "25£")
- ("lib_A", "2-277-562-7", "jonathan livingston le goeland, R. Bach, trans. P. Clostermann,...","130", "BW", "25£")
- ("lib_B", "Ex324", "le petit prince, ...", "75", "BW", "15£")
- ("lib_C", "Bach70-2345", "jonathan livingston le goeland, R. Bach, trans. P. Clostermann,...","130", "BW", "15£")

From the User Interface point of view, once the search is triggered from the web browser, an HTML page is returned and regularly updated, displaying the current content of the service **matchingBook**. The customer may select one or more of the displayed items, triggering a direct method which removes the corresponding resource from the service **matchingBook** and inserts a resource in the service **checkedBook**, used afterwards in the negotiation.

4.2 Negotiation Phase

Having selected two items out of the list of matching books, according to the scenario of section 2, the customer, via the broker, asks for the print of both books locally. Indeed, if it is possible, it would decrease the cost and reduce the delivery delay. The customer, via her Web browser, fills a form indicating the following constraints: price *best*, printed *near London* and delay < 48 *hours*. These informations are combined with the resources held by the service `checkedBook` and produce the following two resources in the service `localPrintReq` of the `Broker`:

- ("lib_A", "1-567-789-6", "75", "color", "48", "london", "", "", "")
- ("lib_C", "Bach70-2345", "130", "BW", "48", "london", "", "", "")

The components of these tuples are: the name of the digital library, the identification of the book, its number of pages, its color feature, the deadline, the expected location of the print shop, the price of the print service, the print shop agent name and the reference of the printing service (the last three fields are initially unknown). These insertions trigger the following rules the role of which is to find the best offer satisfying the constraints imposed by the customer.

```
  'printShop(ps)
@ localPrintReq(dl,dlRef,rQuality,pp,rDelay,rLoc,currentBestPrice,_,_)
@ 'location[ps](rLoc) @ 'quality[ps](rQuality,psRef)
@ 'delay[ps](psRef,pp,psDelay) @ lessThan(psDelay,rDelay)
@ 'price[ps](psRef,pp,psPrice) @ lessThan(psPrice,currentBestPrice)
<>-
localPrintReq(dl,dlRef,rQuality,pp,rDelay,rLoc,psPrice,ps,psRef)
```

For each print shop returned by `printShop` and for each request held by `checkedBook` the constraints in term of *location*, *quality*, *delay* and *price* are checked. Each time the constraints are satisfied, the transaction phase is triggered and the resource held by the service `localPrintReq` is removed and a new one containing a best offer is inserted. Thus, the proposition of a new service by a print shop or the dynamic registration of a new print shop may trigger the rule and improve the current offer. For instance, after a few applications of the rule, the resources of `localPrintReq` are:

- ("lib_A", "1-567-789-6", "75", "color", "48", "london","" ,"" ,"")
- ("lib_C", "Bach70-2345", "130", "BW", "48", "london", "10£", "PS_A", "BW-600dpi-48h")

At any time the customer may consult the current best offer and decide to either consider it or start, in parallel, another request, so as to feed in more offers. In our example, she may try to seek an offer from a remote print shop, with the printed book being delivered by a fast delivery company. For this, a specific resource is inserted in the service `remotePrintReq`, which has a similar role as `localPrintReq`. The last components of a `remotePrintReq` service specify the current price, the print shop agent name, the reference of the printing service, the delivery company agent name and the reference to the delivery service.

```
  'printShop(ps) @ 'fastDelivery(fd)
@ remotePrintReq(dl,dlRef,rQuality,pp,rDelay,rLoc,rPrice,_,_,_,_)
@ 'location[ps](psLocation)
@ 'quality[ps](rQuality,psRef) @ 'weight[ps](psRef,pp,psWeight)
@ 'delayPS[ps, "delay"](psRef,pp,psDelay)
@ 'pricePS[ps, "price"](psRef,pp,psPrice)
@ 'deliver[fd](psLocation,rLoc,fdRef)
@ 'delayFD[fd, "delay"](fdRef,fdDelay)
@ 'priceFD[fd, "price"](fdRef,psWeight,fdPrice)
@ plus(delay,psDelay,fdDelay) @ lessThan(delay,rDelay)
@ currencyConverter(psPriceLocalCurrency,psPrice,rLoc)
@ currencyConverter(fdPriceLocalCurrency,fdPrice,rLoc)
@ plus(price,psPriceLocalCurrency,fdPriceLocalCurrency)
@ lessThan(price,rPrice)
<>-
remotePrintReq(dl,dlRef,rQuality,pp,rDelay,rLoc,price,ps,psRef,fd,fdRef)
```

In the same manner, the combined offers for the printing and delivering services are compared against constraints and the best offer is computed. After a while, the service **remotePrintReq** contains a resource defining the current best offer:

- ("lib_A", "1-567-789-6", "75", "color", "48", "50£", "PS_C", "co-300dpi-24h", "Del_{A_F}", "24h")

4.3 Transaction Phase

After the search and negotiation phases, the customer wishes to enact the commercial transaction using the different service offers she has chosen. The customer would like that transactional properties be respected, and, in particular, to ensure that all the negotiated offers are committed or none of them. For instance, buying the offer from a remote print shop would be useless if finally the fast delivery company is not able to realize its offer to deliver the books from this print shop in less than 24h.

The transaction is initiated via the Web browser displaying the offers for each request. Once all the offers are checked by the customer, she fills the different information needed for the transaction (e.g. her address, banking details), which triggers the construction of the following CLF script that handles the commercial transaction. Notice here the use of the reflexive capabilities of the CLF model: scripts are used as resources manipulated by another script. The commercial transaction script is inserted into the coordinator of the payment mediator which enacts it.

```
  commercialTransaction( "CT1")
@ item["libA"]("1-567-789-6","10") @ credit["libA"]("10")
@ item["PSC"]("co-300dpi-24h","6") @ credit["PSC"]("6")
@ item["DelAF"]("24h","5") @ credit["DelAF"]("5")
@ item["libC"]("Bach70-2345","9") @ credit["libC"]("9")
```

```
@ item["PS_A"]("BW-600dpi-48h","2") @ credit["PS_A"]("2")
@ withdrawall("customerStanley","32")
<>-
  notifyProviderDL["lib_A","notify"]("1-567-789-6")
@ notifyProviderPS["PS_C","notify"]("co-300dpi-24h","lib_A","1-567-789-6")
@ notifyProviderFD["Del_AF","notify"]("24h","PS_C","<StanleyAddress>")
@ notifyProviderDL["lib_C","notify"]("Bach70-2345")
@ notifyProviderPS["PS_A","notify"]("BW-600dpi-48h","lib_C","Bach70-2345")
@ notifyCustomer("CT1","Transaction accepted","")

  'commercialTransaction("CT1")
@ unsufficientCredit("customerStanley","32")
<>-
notifyCustomer("CT1","Not Enough Money","32")

  'commercialTransaction("CT1")
@ notAvailable("lib_A","notAvailable","1-567-789-6","10")
<>-
notifyCustomer("CT1","lib_A service not available","1-567-789-6")
```

The first rule realizes the commercial transaction if everything goes well. The others (subsidiary rules) handle the different problems that can occur (withdrawal forbidden on the customer account, unavailability of a service) and notify the customer of the problem. So, either the first rule or at least one of the others is committed. The service **commercialTransaction** ensures that when the main rule is enacted none of the subsidiaries will do since the resource *"CT1"* is consumed.

One consequence of the commitment of the first rule is to credit and withdraw the accounts of the different actors. This is performed via the **payment mediator** that provides services **credit**, **withdrawal** and **unsufficientCredit** (see next section for more information about the role of this agent). The **item** service of each agent may react differently depending on the type of offer it concerns. For instance, we can imagine that the resource constituted by an electronic document is infinite and is not in fact physically consumed. However, the work that a print shop should do impacts on the availability of its equipment and in this case, a resource corresponding to the time slot used for the print of the book is really consumed. The rules notify each individual actor of the final outcome of the transaction through the **notifyProvider...** services.

4.4 Mediating

The customer and the different service providers would like to be paid without being bothered by finding a common payment system. Moreover, when small amounts of money are involved it is not always interesting to directly realize the operation. The payment system mediator, locally manages accounts (for instance in order to group small payment) or forwards the operations to the different payment systems used by the commercial actors. This latter option

may involve traditional payment methods such as faxed credit card information or electronic payment service existing on the Web [8,7,6,3].

4.5 Tracking

The customer and the different provider would like to visualize the overall workflow in order to track the global work. For instance, with a classical web browser, the customer may verify that the printing phase is finished and the task is now in the delivery phase. For these workflow aspects the reader can refer to [5].

5 Conclusion

We have presented here a Distributed Print-on-Demand System based on the electronic commerce framework Xpect. The main advantage of our system is that it provides high level scheme allowing not only to search services across the Web but also to optimize their combination in order to realize complex service respecting constraints imposed by the user. Moreover, it offers basic transaction facilities that may directly be used for ensuring multi-party commitment. Finally, it is possible to dynamically register new services directly taken into account in the on-going customer requests.

References

1. J-M. Andreoli, F. Pacull, D. Pagani, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, scheduled for mid-1998.
2. J-M. Andreoli, F. Pacull, and R. Pareschi. Xpect: A framework for electronic commerce. *IEEE Internet Computing*, 1(4):40–48, 1997.
3. Steve B. Cousins, Steven P. Ketchpel, Andreas Paepcke, Héctor García-Molina, Scott W. Hassan, and Martin Roescheisen. Interpay: Managing multiple payment mechanisms in digital libraries. In *Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.
4. T. Finin, Y. Labrou, and J. Mayfield. Kqml as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, Ma, U.S.A., 1997.
5. A. Grasso, J-L. Meunier, D. Pagani, and R. Pareschi. Distributed coordination and workflow on the world wide web. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 6(2):1–26, 1997.
6. P. Panurach. Money in electronic commerce: Digital cash, electronic fund t ransfer, and ecash. *Communication of the ACM*, 39(6):45–50, 1996.
7. Lee H. Stein, Einar A. Stefferud, Nathaniel S. Borenstein, and Marshall T. Rose. The green commerce model. draft, October 1994.
8. Michael Waidner. Development of a secure electronic marketplace for europe. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth ESORICS*, LNCS, Rome, Italy, September 1996. SV.