**Appears in** *the Journal of Distributed and Parallel Databases, special issue on Electronic Commerce.*

# Distributed Print on Demand Systems in the Xpect Framework

Jean-Marc Andreoli and François Pacull
Xerox Research Centre Europe, Grenoble, France

**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 negotiation, electronic commerce brokering, workflow

## 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 builds over the Coordination Language Facility [3] (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 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 server* 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 one. 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 that proposes 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 server* 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. Four distinct solutions for $book_b$ are currently available:

1. Print from $PS_B$ and deliver by $del_A$ (from USA)
2. Print from $PS_B$ and deliver by $del_B$ (from USA)
3. Print from $PS_C$ and deliver by $del_A$ (from France)
4. Print from $PS_C$ and deliver by $del_B$ (from France)

At some point, the customer may decide to consider the current best solution satisfying her deadline: 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.

Here the whole scenario is executed manually by the customer, who would then have to access each service separately, and manually combine them. The purpose of a generic brokering service is precisely to relieve the customer from such a burden, and to offer a high-level service which coordinates all the others according to the customer's requirements (and under her control). This should in particular allows the customer to benefit from new offers that could become available during the process and then to reconsider previous offers. 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 [3], 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

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 managing resources. 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 **Inquiry**). Agent $B$ returns a handle representing a possibly infinite stream of answers, each of

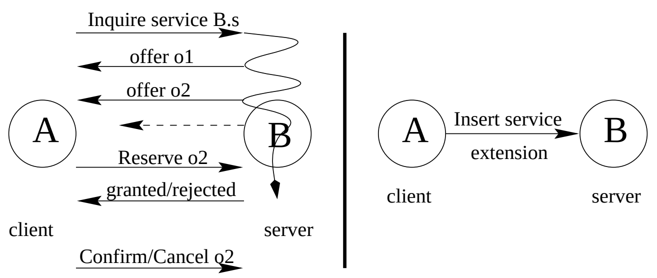

Figure 1: The CLF interaction model

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 either be 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 either be 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 [8]. It allows on the one hand flexible dynamic redefinition and refinement of services and, on the other hand, transactional 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. The main advantage is that more or less all programming languages have either full HTTP libraries, or at least primitives to access URLs, in which the name and parameters of the direct methods can be encoded. In particular, this makes it possible to invoke such methods directly from any Web browser and if the result of the invocation is in HTML format, then it is displayed directly on the browser. It is therefore very easy to quickly add a 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 Inquire 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 this section is to illustrate with a toy example the syntax and operational meaning of CLF scripts. The main point in this example is that it shows in an intuitive maner how scripts search and combine resources, and solve conflicts in their usage. More realistic scripts 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 script and must be assigned to a real service provided by the coordinator. By default, a coordinator provides (*i*) basic computational services, and (*ii*) surrogates of remote services identified by a name looked-up in an application-wide name service.

Apart from the assignment of tokens to services (the `interfaces` section), a CLF script also contains a specification of a coordinated invocation of these services (the `rules` section). A CLF rule consists of a left-hand side and a right-hand side separated by the symbol `<>-` (read "become"). Each side specifies a list of tokens separated by the symbol `@` (read "par").

For example, the following script specifies the process of allocating hotel rooms to people.

```
interfaces:

   customer(a,b) :  -> a,b is LOOKUP Agency.customer
   roomReservation(a,b) : a,b -> is LOOKUP Agency.roomReservation
   vacancy(a,b,c) : a,b -> c  is LOOKUP Hotel.vacancy

rules:

   customer(name,date) @ vacancy(date,"single",roomNumber)
   <>- roomReservation(name,roomNumber)
end
```

There are three tokens involved:

- The token `customer(a,b)` is assigned a service `Agency.customer` owned by a reservation agency and holding customer requests for hotel rooms. The first parameter `a` is the name of the customer and the second one is the requested date of the reservation. In this script, both are output parameters, since the script is aimed at the satisfaction of the requests, not their production.
- The token `roomReservation(a,b)` is assigned another service, `Agency.roomReservation`, of the agency, holding the satisfied reservation requests. Parameter `a` denotes a customer's name and parameter `b` denotes the room s/he has been assigned.
- The token `vacancy(a,b,c)` is assigned a service `Hotel.vacancy` owned by a hotel and holding available rooms at different dates. The parameters `a`, `b` and `c` denote, respectively, the date of availability, the type of the room (e.g. single or double) and the room number. In this script, the first two parameters are input and the third one is output, so that token `vacancy(a,b,c)` allows to retrieve then reserve available rooms of a given type at a given date.

The rule of the sample script processes reservation requests. The requests are first retrieved from service `customer` which requires no input parameter (both its parameters are output). Let's assume it returns the following flow of offers:

1. (*"Helmut"*, *"May 2nd"*)
2. (*"Tony"*, *"May 2nd"*)
3. (*"Jacques"*, *"May 3rd"*)

   ...

Each of these offers creates a specialized instance of the rule, and all the instances are executed asynchronously and concurrently:

1. customer(*"Helmut"*, *"May 2nd"*) @ vacancy(*"May 2nd"*, *"single"*,roomNumber)
2. customer(*"Tony"*, *"May 2nd"*) @ vacancy(*"May 2nd"*, *"single"*,roomNumber)
3. customer(*"Jacques"*, *"May 3rd"*) @ vacancy(*"May 3rd"*, *"single"*,roomNumber)

   ...

The execution of each instance proceeds by requesting service `vacancy` with the corresponding input parameters (e.g. `"May 2nd"` and `"single"` for the first instance). The resulting flow of offers may be:

1. (*"May 2nd"*, *"single"*, *"501"*)
2. (*"May 2nd"*, *"single"*, *"502"*)

3. (*"May 2nd"*,*"single"*,*"503"*)

   ...

which produce further specializations of the rule:

1. customer(*"Helmut"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"501"*)
2. customer(*"Helmut"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"502"*)
3. customer(*"Helmut"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"503"*)

4. customer(*"Tony"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"501"*)
5. customer(*"Tony"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"502"*)
6. customer(*"Tony"*, *"May 2nd"*) @ vacancy(*"May 2nd"*,*"single"*,*"503"*)

   ...

At this point, the instances are complete and are ready to enter the transaction phase. Notice that each instance denotes a combination of offers which is a potential candidate for the problem we are solving. However, it is easy to see that there exists conflicts between the different propositions. Indeed a person does not need several rooms and a room cannot be booked by different people at the same time.

The CLF ensures that the actions promised by the services (namely the offers) will be executed atomically. So, if the first proposition above is realized, i.e. the first instance of the rule is successfully executed, then the resources supporting the offers `customer("Helmut", "May 2nd")` and `vacancy("May 2nd","single","501")` will be removed atomically from their respective services, and the creation of a resource supporting `roomReservation("Helmut","501")` will be requested in the corresponding service. This atomic removal of resources will invalidate the propositions 2,3 and 4, i.e. the execution of the corresponding instances will be aborted. Similarly, only one of the propositions 5 and 6 will succeed.

Conflicts may occur not only between instances of the same rule, but also between different rules and even between different scripts, possibly run by separate coordinators on different machines. The transactional protocol of the CLF is also capable of handling these cases.

Our sample script involves only one agency and one hotel. It would be easy to introduce, say, another agency, by adding another rule similar to the one already given, in which the tokens relative to the agency (`customer` and `roomReservation`) are replaced by new tokens assigned to the services of the new agency. This would be a typical case where the two rules would compete for the same hotel resources. More generally we can imagine that several agencies and several hotels - the lists of which could evolve dynamically - have to be considered. To do so 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.

The following rule uses the dispatch mechanism to allow several agencies and hotels to be considered. 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 initial rule is modified in order to get first a list of possible hotels and agencies able to solve the original room reservation. Then the logical name of the service is not known at compilation time but computed at the execution time.

```
interfaces:

   hotelList(hotel) : -> hotel is LOOKUP Broker.hotelList
   agencyList(agency) : -> agency is LOOKUP Broker.agencyList

   customer(object,service,name,date) :
     object,service -> name,date is DISPATCH

   vacancy(object,service,date,type,roomNumber) :
     object,service,date,type -> roomNumber is DISPATCH

   roomReservation(object,service,name,roomNumber) :
     object,service,name,roomNumber is DISPATCH

rules:

   `hotelList(hotel) @ `agencyList(agency) @
   customer[agency](name,date) @ vacancy[hotel](date,"single",roomNumber)
   <>- roomReservation[agency](name,roomNumber)
```

```
end
```

The two services accessed through the tokens `hotelList` and `agencyList` are owned by a "broker" agent which tries to solve the room allocation problem for a set of agencies and a set of hotels. They return the names of hotels and agencies registered for the transaction. These lists of names may evolve dynamically allowing new hotel or agency to be taken into account or already proposed names to be disabled.

The marker ‘ in front of a token means that the offer found for this token should still be valid when the rule is actually enacted, but it is not effectively executed by the rule. This is used for the two services provided by the `Broker` object since the execution of a reservation should not remove the resources describing the hotels and agencies from the broker's lists. For the rest of the rule the behavior is the same as previously.

## 4 Distributed Print on Demand System in Xpect

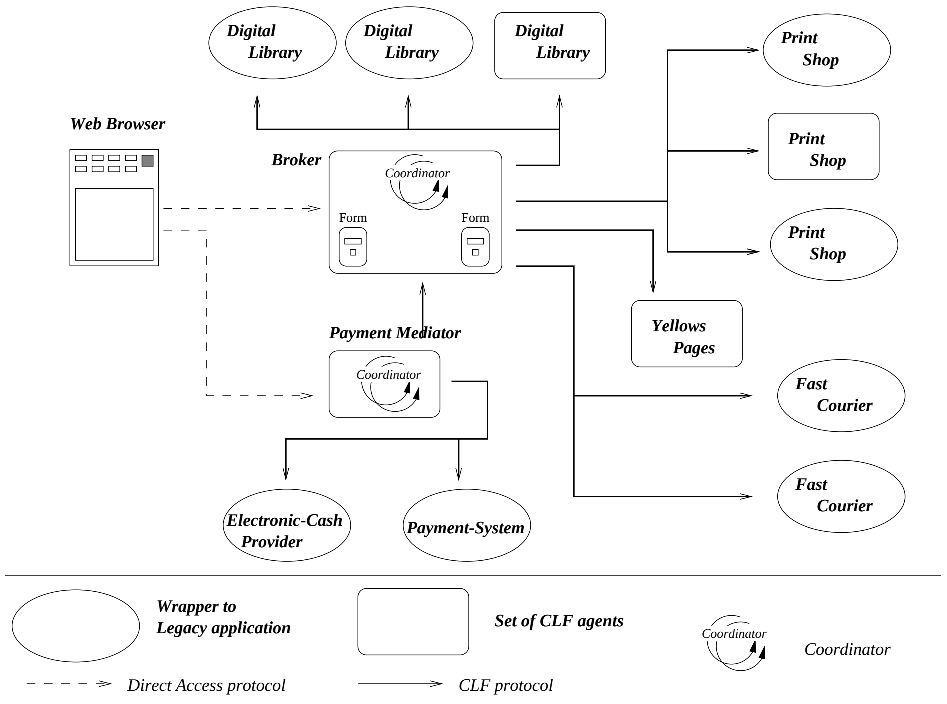

Figure 2: Overall architecture

We now sketch the design of an electronic commerce application in the context of distributed print on demand systems. The architecture of the system (Figure 2) is composed of two kinds of components, implemented as CLF objects:

- Wrappers to external service providers: they can wrap either legacy applications which are already available in electronic form and accessible by some electronic means, such as databases (e.g. good providers catalogs), or disconnected services which have their own operating interface, which may involve non fully electronic means such as fax (e.g. to inquire or reserve a delivery service).
- Xpect-specific services: they consist essentially of coordinators, which execute CLF coordination scripts, together with local storage services (essentially tuple banks).

More precisely, 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 side-services (e.g. binding);
- fast delivery companies in charge of delivering the printed books from the print shop to the customer site;
- payment systems handling various modes of payment: electronic cash, credit cards, checks;
- yellow pages classifying the other services (for clarity purpose, we assume here a very simple classification: `digitalLibrary`, `printShop`, `fastDelivery`);
- payment mediators and brokers offering the main entry points to the customer.

An instance of the Xpect framework for distributed print on demand will typically include several instances of the above-mentioned components. In particular, multiplicity of brokers and payment mediators, even if they share exactly the same behavior, avoids creating bottlenecks in the overall architecture. Each broker knows one and only one payment mediator (which may serve several brokers). A customer may choose the broker which is closest to its location, or which is accessible, in case of network cuts. One Xpect session can be seen as a long lived workflow process instance, involving the customer, the broker s/he has selected and its attached payment mediator, plus any number of instances of the other components mentioned above, depending on the customer's specific requests. The only assumption here is the presence of a (distributed) name server, which is also implemented as a set of CLF objects conforming to the CLF protocol, allowing brokers to retrieve the information needed to actually connect to the different services. The information contained in the name service component is structured by the yellow pages component, which returns logical names (entries in the name service) attached to functional descriptions.

Interaction between the customer and the broker / payment mediator can be done from a Web browser invoking direct methods on the corresponding CLF objects (dashed arrows in Figure 2). All the other interactions (plain arrows in Figure 2), generated by the coordinators, conform to the CLF protocol. In the implementation, the CLF objects are built on top of some traditional distributed object infrastructure and their interactions translate into traditional method invocations. Since only basic data values (integer, strings) are exchanged during these interactions, little requirements are put on the object infrastructure itself. In fact, the development library available with the CLF package offers support for several such infrastructures, including, of course, CORBA. The strict modular architecture of a CLF object allows to fully decouple its communication capabilities from its CLF-specific features, so that the same CLF object can at the same time be involved in several interactions supported by different communication infrastructures (e.g. HTTP and CORBA). The reader is referred to [3] for more details on this topic.

As described in the scenario of section 2 a typical session may be decomposed in several steps, similar to a workflow map. Each step implements a specific high level coordination scheme (i.e. *Search, Negotiation, Transaction, Mediating and Tracking*). These schemes are described using CLF scripts enacted either by the *broker* coordinator or by the *payment mediator* coordinator. These coordinators assume and make use of specific CLF services offered by the components, summarized in Table 1.

| | |
|---|---|
| Digital libraries | `title(dlRef,title,version)`<br>`bookInfo(dlRef,description,npages,quality,price)`<br>`item(dlRef,price)` |
| Print shops | `location(loc)`<br>`quality(psRef,quality)`<br>`printInfo(psRef,npages,price,delay,weight)`<br>`item(psRef,price)` |
| Delivery companies | `deliver(fdRef,fromLoc,toLoc)`<br>`deliveryInfo(fdRef,weight,price,delay)`<br>`item(fdRef,price)` |
| Yellow pages | `digitalLibrary(name)`<br>`printShop(name)`<br>`fastDelivery(name)` |
| Brokers | `searchRequest(title,version,location)`<br>`matchingBook(dl,ref,description,npages,quality,price)`<br>`localPrintReq(dlName,dlRef,quality,npages,delay,loc,psName,psRef,psPrice)`<br>`remotePrintReq(...)` |
| Payment mediators | `withdrawal(Account,Amount)`<br>`credit(Account,Amount)` |

Table 1: Services used by the coordinators

Notice however a major difference between traditional workflow maps, which enforce a rigid sequentialization of the steps they involve, and the kind of flexible coordination of steps required here: all the steps overlap and, for example, an ongoing negotiation step can be influenced by new results asynchronously coming out of the search step.

## 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 displaying 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 which can then be used by following rule in the `broker` coordinator.

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

This rule makes use of two services provided by the digital library agent:

- The service `title` retrieves book references corresponding to the given title constraint (here "Jonathan Livingston Seagull", "french" version).
- The service `bookInfo` provides information about a given book reference.

The rule also involves two services in the `broker` agent, which act as simple tuple bags:

- The service `simpleSearchRequest` collects the pending customer requests (for simplicity purpose, the identification of the customer has been omitted).
- The service `matchingBook` collects matching offers.

The rule collects and aggregates informations from the first three services (i.e. reference of the book, human readable description, number of pages, quality required for the printing and price) into the last service, `matchingBook`, which is used in the continuation of the process. Notice that the rule actually consumes only the resource from `simpleSearchRequest`, since the last three tokens in the left hand side of the rule are tagged with ‘: they correspond to pure information retrieval operations which need not go through the transactional part of the CLF protocol.

However, the simple rule above is only able to search into one digital library and returns at most one offer. To extend the search capability, we rely on the type of service called *dispatch* already described in section 3. A dispatch service allows late (dynamic) binding of the tokens based on the first two parameters that denote respectively, the name of an object and the name of a service defined in the interface of that object to be looked up in the name service (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,location)
@ ‘digitalLibrary(dl) @
@ ‘title[dl](dlRef,title,version)
@ ‘bookInfo[dl](dlRef,description,pp,quality,priceLocal)
@ currencyConverter(priceLocal,price,location)
<>-  matchingBook(dl,dlRef,description,pp,quality,price))
```

The service `digitalLibrary` is provided by the `yellowpages` 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 (holding its name as known from the name service). Each digital library agent also has to provide the services `title` and `bookInfo` as previously. They are accessed through *dispatch* services defined at the broker agent level. It is possible to dynamically add new digital libraries to the system (via the `yellowpages` agent) without changing the rule nor interrupting the system, nor even interrupting the ongoing sessions. The service `currencyConverter` is a simple wrapper to a Web service that translates a price from the local currency into the currency used by the customer. It holds no resource and has a void behavior during the transaction phase. The service `matchingBook` is the same as previously, except that the library agent name is also stored in its arguments. 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:

1. ("$lib_A$", "1-567-36789-6","le petit prince, ...", "75", "color", "25£")
2. ("$lib_A$", "2-277-21562-7", "jonathan livingston le goeland, R. Bach, trans. P. Clostermann,...","130", "BW", "25£")
3. ("$lib_B$", "Ex324","le petit prince, ...", "75", "BW", "15£")
4. ("$lib_C$", "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 were possible, it would decrease the cost and reduce the delivery delay. The customer, via her Web browser, fills a form indicating the following constraints: *best* price, printed *near London* within *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`:

1. ("$lib_A$", "1-567-36789-6", "75", "color", "48", "london", "", "", "")
2. ("$lib_C$", "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 rule the role of which is to find the best offer satisfying the constraints imposed by the customer.

```
localPrintReq(dl,dlRef,quality,pp,delayMax,destination,bestPrice,_,_)
@ ‘printShop(ps)
@ ‘location[ps](destination) @ ‘quality[ps](psRef,quality)
@ ‘printInfo[ps](psRef,pp,price,delay,_) @
@ ... # check the constraint  delay < delayMax and price < bestPrice
<>- localPrintReq(dl,dlRef,quality,pp,delayMax,loc,price,ps,psRef)
```

For each print shop returned by `printShop` and each request held by `checkedBook`, the constraints in term of *location*, *quality*, *delay* and *price* are checked, using some basic computation service assumed to be available to the broker and informally described in comment for readability. 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 offer by a print shop or the dynamic registration of a new print shop may trigger a new instance of the rule and improve the current best offer. For instance, after a few applications of the rule, the resources of `localPrintReq` may be:

1. ("$lib_A$", "1-567-36789-6", "75", "color", "48", "london","" ,"" ,"" )
2. ("$lib_C$", "Bach70-2345", "130", "BW", "48", "london","10£", "$PS_A$", "BW-600dpi-48h")

indicating that the current best offer for the printing of the second book costs 10 pounds, while there is still no offer for the first book (last three parameters are empty).

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.

```
remotePrintReq(dl,dlRef,quality,pp,delayMax,destination,bestPrice,_,_,_,_)
@ ‘printShop(ps)
@ ‘location[ps](loc) @ ‘quality[ps](quality,psRef)
@ ‘printInfo[ps](psRef,pp,pricePrintLocal,delayPrint,weight)
@ ‘fastDelivery(fd)
@ ‘deliver[fd](fdRef,loc,destination)
@ ‘deliveryInfo(fdRef,weight,priceDeliveryLocal,delayDelivery)
@ ... # check  delayDelivery + delayPrint < delayMax
@ currencyConverter(pricePrint,pricePrintLocal,loc)
@ currencyConverter(priceDelivery,priceDeliveryLocal,loc)
@ ... # check  (price = pricePrint + priceDelivery) < bestPrice
<>-
remotePrintReq(dl,dlRef,quality,pp,delayMax,destination,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, e.g.:

1. ("$lib_A$", "1-567-36789-6", "75", "color", "48", "50£","$PS_C$", "co-300dpi-24h", "$Del_B$", "24h")

## 4.3 Transaction Phase

As soon as the search and negotiation phases start producing complete offers satisfying the overall customer's requirements, the customer may wish to select one of them and finalize the corresponding commercial transaction. 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 within the required delay.

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: a script is used as a resource manipulated by another script. The commercial transaction script is inserted into the coordinator of the payment mediator which enacts it.

```
  commercialTransaction("CT1")
@ item["lib_A"]("1-567-36789-6","10") @ credit["lib_A"]("10")
@ item["PS_C"]("co-300dpi-24h","6") @ credit["PS_C"]("6")
@ item["Del_B"]("24h","5") @ credit["Del_B"]("5")
@ item["lib_C"]("Bach70-2345","9") @ credit["lib_C"]("9")
@ item["PS_A"]("BW-600dpi-48h","2") @ credit["PS_A"]("2")
@ withdrawall("customerStanley","32")
<>-  notifyProviderDL["lib_A","notify"]("1-567-36789-6")
@ notifyProviderPS["PS_C","notify"]("co-300dpi-24h","lib_A","1-567-36789-6")
@ notifyProviderFD["Del_B","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","1-567-36789-6","10")
<>- notifyCustomer("CT1","lib_A service not available","1-567-36789-6")
```

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 if and when the main rule is successfully executed, all the subsidiary rules will abort 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. On the other hand, satisfying a print offer has a direct incidence on the availability of its equipment and in this case, a resource corresponding to the time slot used for the printing 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 may be more efficient to group together several small financial operations and perform them in a block. The payment system mediator locally manages accounts (for instance in order to group small payments) or forwards the operations to the different payment systems used by the commercial actors. This latter option may involve tradition payment methods such as faxed credit card information or electronic payment service existing on the Web [12, 7]. Basically, the payment service is considered as a particular provider that use money as its own resource. Then the interactions between the payment services, the customer, the traditional banks or electronic cash service is handled in a same way as other kinds of resources.

## 4.5 Tracking

The customer and the different providers would like to visualize the overall workflow in order to monitor the state of a commercial transaction realized by Xpect. For instance, with a classical web browser, the customer may wish to verify that the printing phase is finished and the task is now in the delivery phase. If a delay occurs, the customer may decide to take specific actions. More generally, the customer, or the other commercial actors, may wish to track the execution of the commercial transaction in order to integrate it into a wider process, or workflow, in which it occurs.

On the side of the good and service providers, for example, the actions involved in the commercial transaction may be coordinated by an order processing workflow, which may include accounting steps, stock verification steps, as well as various validation and quality management steps. On the side of the customer, the commercial transaction may be part of a larger process describing the different steps of the execution of a longer term project or operation; this aspect does not appear in our toy print-on-demand case study, where the customer is a private person rather than an organization, whose involvement in the commercial transaction is not part of a structured process (buying a birthday present...).

In case an agreement exists between some, or all, of the commercial actors, the different workflows involved in the commercial transaction may directly inter-operate so that, for example, an exception behavior in the workflow of one provider (e.g. delays in the availability of a service or goods) is immediately taken into account in the workflow of the customer, or even that of other providers, and the exception can be treated in a collaborative manner. The resulting situation is that of a global process enacted autonomously by different inter-operating worksites, but respecting the global policy specified by the process. The broker agent, which has been described in detail in the previous sections, is just one of the involved worksites. The CLF scripts enacted by its coordinators represent the maps of the workflow processes executed on this worksite. Unlike traditional workflow maps, CLF scripts support both transactions and flexible negotiation between tasks. These features, which result from the CLF resource based approach, have been exploited in an essential way in the specification of the broker workflow.

More generally, we believe that the resource based approach taken by the CLF object model and scripting language provides a good framework for workflow interoperation. We have applied it to the design of a distributed workflow system called Webflow [9, 1]. A Webflow engine is not a monolithic entity, but is instead split into distributed independent components, which are viewed as resource managers and are coordinated by CLF scripts of the kind which have been presented above. The resources held by an Xpect broker, and holding the status of the commercial transaction in its different phases, could typically be manipulated in the workflow processes of the participants to the transaction.

Webflow exploits the reflective nature of the CLF to provide support for a range of “meta”-processes (i.e. processes which operate on other processes), such as the dispatching of subprocesses over several worksites, or the dynamic updating of running instances of a process when the process model has been changed. In Xpect, these meta-processes could be used to ensure that the inter-operation between the participants workflows conform to the global policy defined by their agreement.

## 5 Discussion

The Xpect system involves techniques coming from several areas of research, discussed here.

### 5.1 Distributed object coordination infrastructures and middleware

Coordination infrastructures elaborate the basic distributed object interaction model, which nowadays has reached industrial strength through standard architectures such as CORBA [10]. For example transaction processing with its many extensions [11] constitute a major line of research in the domain of coordination. Another one explores coordination through shared message spaces, following the program initiated by the seminal work on Linda [4]. The CLF infrastructure, on which Xpect relies, integrates at the lowest level — the object interaction protocol — concepts coming from both transaction based and message-space based techniques. Furthermore, the CLF offers a powerful scripting language, which allows a high degree of flexibility and reflexivity in the design of coordinated systems.

### 5.2 Workflow

The CLF infrastructure promotes the notion of resource as a first class concept: objects are viewed as resource managers. This makes it particularly suitable for workflow applications, which, to a great extent, consist of more or less complex and entangled resource manipulations (resources being, e.g., task states, documents, or hardware components, time slots, human competencies needed to perform some tasks). Xpect essentially provides a workflow, enacted by the broker agent coordinator, which involves a number of commercial actors which may be either simple service providers capable of reacting to the workflow messages they receive or full workflow processes themselves which may inter-operate with the broker. Xpect uses the transactional properties offered by the underlying infrastructure (CLF) to achieve some form of consistency in the workflow execution (as in many applications of advanced transaction models to workflow [16]) and uses the scripting language both to specify the “normal” execution of the workflow and to handle exceptions (as with Contracts scripts [15]). Notice that the scripting language allows for the asynchronous generation of combinations of service offers, thus leading to a much more flexible form of workflow task coordination (far beyond the usual input-process-output model), used e.g. in the negotiation step (Section 4.2). Also, we make use of the reflexive capability of the scripting language (which can manipulate scripts as resources) to generate taylored pieces of the workflow map "on the fly", e.g. in the commercial transaction finalization step (Section 4.3).

### 5.3 Electronic Commerce

The workflow enacted by the Xpect broker realizes a number of electronic commerce operations, which exist in many comparable systems such as the Eco framework [14]. In this paper, we have illustrated one instance of broker with a specific behavior (search-negotiation-transaction-mediating-tracking) in the context of distributed print-on-demand, and shown that it can easily be encoded as CLF scripts. The search-negotiation steps implement an extended contract-net mechanism [13] involving a customer and multiple good and service providers (digital libraries, print-shops and delivery companies), which is one of several possible "marketware" modules, "buy-sell brokering" in the Eco terminology. It is not difficult to see how other behaviors, such as Kasbah's ads-based negotiations [6], could be specified in a similar way.

The key to the integration realized by the brokers resides in the CLF distributed name service which provides access metadata for the interaction with the commercial actors and the yellow pages service which provides content metadata. As in the Metabroker architecture [5], these two kinds of informations set the appropriate proxies inside the broker coordinator, and all the communications generated by the execution of the scripts go through these proxies.

## 6 Conclusion

We have presented here a Distributed Print-on-Demand System based on the electronic commerce framework Xpect. The main feature of our system is that it provides a high level scheme allowing not only to search services across the Web but also to optimize their combination in order to realize complex service requests respecting constraints imposed by the customer. 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. These functionalities fully exploit the underlying distributed object infrastructure CLF, and in particular its flexible, reflexive scripting language.

## References

[1] J-M. Andreoli, J-L. Meunier, and D. Pagani. Process enactment and coordination. In *Proc. of EWSPT'96*, pages 195–216, Nancy, France, 1996.

[2] J-M. Andreoli, F. Pacull, and R. Pareschi. Xpect: A framework for electronic commerce. *IEEE Internet Computing*, 1(4):40–48, 1997.

[3] J-M. Andreoli, D. Pagani, F. Pacull, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, 31:179–203, 1998.

[4] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Ma, U.S.A., 1990.

[5] S. Caughey, D. Ingham, and P. Watson. Metabroker: A generic broker for electronic commerce. Technical report, Newcastle University, U.K., 1998.

[6] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proc. of 1st Int'l Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 75–90, 1996.

[7] 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 *DL '95 proceedings*, 1995.

[8] 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.

[9] 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.

[10] Object Management Group and X/Open. OMG, CORBA, the common object request broker: architecture and specification, 1991. OMG Document Number 91.12.1, Rev. 1.1.

[11] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academics Publishers, 1997.

[12] P. Panurach. Money in electronic commerce: Digital cash, electronic fund transfer, and ecash. *Communication of the ACM*, 39(6):45–50, 1996.

[13] R.G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computing*, 29(12):1104–1113, 1980.

[14] J. Tenenbaum, T. Chowdry, and K. Hughes. The eco system: Comercenet's architectural framework for internet commerce. Technical report, CommerceNet Inc., Palo Alto, Ca., U.S.A., 1997.

[15] H. Wächter and A. Reuter. The contracts model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, Ca, U.S.A., 1993.

[16] D. Worah and A. Sheth. Transactions in transactional workflows. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures.* Kluwer Academics Publishers, 1998.