Heterogeneous component coordination: the CLF approach

Damián Arregui, François Pacull and Michel Rivière
Xerox Research Centre Europe
6, chemin de Maupertuis, 38240 Meylan, France
{Damian.Arregui, Francois.Pacull, Michel.Riviere}@xrce.xerox.com

Abstract

This paper presents a concrete experiment in assembling components from different "protocol-aware" *component frameworks such as Java/Jini (Sun), Corba 3.0 (Object Management Group), Enterprise JavaBeans (Sun) and our own framework called CLF (Coordination Language Facility). We describe how the CLF coordinator can be used to negotiate and perform distributed transactions across several components belonging to the previously cited frameworks, taking advantage of the native transactional capabilities they provide. Since the classical two-phase commit protocol ensures not only serializability but also linearizability, we are able to combine operations in a linearizable way on the whole set of components and thus enforce the serializability of the resulting transaction. We illustrate this approach through a showcase application implementing the coordination of heterogeneous components.*

1 Introduction

A clear message was sent out at the end of the EDOC'99 conference: as the component-based frameworks Java/Jini (Sun)[6], Corba 3.0 (Object Management Group)[15] and Enterprise JavaBeans (Sun)[12] seem to be the emerging standards for middleware technology, it would be a real challenge to present at EDOC2000 a concrete experiment in combining components from these frameworks. This paper describes an attempt in this direction.

Interactions between heterogeneous environments - or even between Corba frameworks released by different vendors - are generally restricted to remote method invocation (RMI, RPC, ...)[13, 16] of components hosted by different frameworks. Conversely, in the experiment described here, we have extended some of the concepts introduced in CLF/Mekano[1, 5] to provide coordination between these components. By coordination we mean higher level mechanisms such as negotiation and distributed transactions.

This article is structured as follows. Section 2 presents the problem. Section 3 describes our approach in terms of the architecture. Section 4 details how we deal with several standard platforms and how these techniques can be extended to any transaction-aware component with a minimum amount of work. Section 5 gives an example of an application built on top of our architecture. Section 6 concludes the article.

2 The problem

The CLF/Mekano environment was a pioneer [2] of the nowadays popular class of *"protocol-aware"* component-based distributed environments that provide a framework for the design and interaction of coarse grain components. In CLF, the core protocols (e.g. distributed transactions) are openly accessible so that foreign clients (e.g. a transaction monitor which does not belong to the platform) can unroll a transactional protocol.

Most other platforms keep the notion of *"protocol-aware"* components but do not open the protocols to foreign clients. For instance, Jini components may be involved in a transaction but this transaction is an opaque object that can be manipulated by default only by a Jini transaction manager and can involve only Jini components. Such an approach prevents the use of enhanced clients such as the CLF coordinator that is able to combine negotiations and transactions. In other words, a CLF coordinator is an object that is able to enact scripts describing the interaction among different components. The enactment of a script consists of an initial phase to query several components in order to find a combined solution to a problem and, in a second phase, to perform in an atomic way[1] the actions the components proposed to do in the first phase.

In the CLF/Mekano framework, we consider the components as resource managers with a very extended definition of resources. A resource may be a database record, a document or fragment of a document, a slot in a print shop schedule, a train ticket, the potential execution of a task

[1] All actions or none are performed.

possibly involving human action and expertise, a piece of information attached to an entry in a name service,...

A Jini JavaSpaces[17] entry is a typical resource in the CLF sense. However, any Jini service can be modeled in the CLF approach if we consider that the result of the service execution is a resource.

2.1 Theoretical aspect

The approach we take is to substitute the classical transaction managers of different heterogeneous platforms with a CLF coordinator. "Substitute" may be not the exact word because a CLF coordinator can even be used concurrently with any of these native transaction managers without causing any inconsistency.

In order to justify this property we have to introduce some results from the theory of consistency criteria. To be able to ensure consistency in the transaction theory, we have to enforce serializability[7]. However, many platforms rely on a two-phase commit protocol that actually ensures the linearizability introduced by Herlihy[10]. The linearizability has two good properties: first it enforces the serializability and second, it is a local property. Local property means that if we consider several objects and can ensure linearizability of each individual action on these objects then we ensure the linearizability at the level of the whole set of objects. This fundamental result, combined with the fact that each platform ensures linearizability for the transaction management, allows us to enforce the linearizability of the transactions distributed across these platforms and thus the serializability.

In other words, we use the embedded mechanisms of each framework to ensure that resource manipulations enacted by the CLF coordinator can co-exist with those enacted by the native transaction managers (because they manage the same resources) and also, that the overall transaction involving several frameworks respects the serializability criterion.

Thus, we are able to design distributed applications where some aspects are implemented within a given framework and other aspects by coordinating different frameworks.

3 Architectural solution

The architectural solution we propose is based on the work we did in the CLF/Mekano[1] project that started several years ago. This platform has already been used for building different applications[3, 4] that integrate both CLF/Mekano and legacy components.

In the past we have encapsulated legacy components that provide their own transactional features such as databases. However, this was done in an hod-hoc fashion and in spite of the speed-up obtained by the know-how we acquired over the time, the development of new encapsulator is still a time-consuming activity. The current development phase of our integration framework is to consider a generic way of encapsulating components from other frameworks such as Jini, EJB and Corba.

In order to make this document self-contained we hereby provide a brief description of the CLF/Mekano environment. More details can be found in [1, 5]

3.1 CLF/Mekano

CLF/Mekano is an object-oriented architecture and development environment, targeting highly distributed applications over multiple intranets, and where the coordination of existing coarse grain entities is a major issue. In our experience, the component-oriented approach followed by Mekano, and the CLF object model, protocol, and scripting facility, based on the view of objects as resource managers, are particularly suited for that kind of applications.

3.2 CLF object model

CLF enforces the traditional object encapsulation policy in that the resources of a CLF object are not accessible directly, but rather through an interface. A CLF object, unlike traditional objects, gives access through two kinds of interfaces: services and direct methods.

3.2.1 CLF services and the CLF protocol

A service is used when resource manipulations have to be performed within a coordinating transaction involving several CLF objects.

More precisely, resource manipulations through a CLF service can be decomposed into three phases called, respectively, negotiation, performance and notification.

- The first phase (negotiation) basically allows the query of a CLF service about offers for action on resources, satisfying given constraints (verb **Inquire**). The stream of returned offers is potentially infinite and can be enumerated by the requester (verb **Next**). When no resource matching the inquiry is currently available, the **Next** operation "is pending" until a new offer becomes available, either due to internal changes in the object or to the insertion of new resources (see notification phase). A stream of offers can also be closed, on the initiative of the object (by raising the NO-MORE-VALUE exception) when the service can guarantee that no new offer matching the inquiry will ever occur, or at the requester initiative (verb **Kill**) when the requester is no longer interested in the offers. Finally, the requester may at any time, check the validity of an offer

(verb **Check**). The verbs involved in the negotiation phase therefore have the following abstract signature:

Inquire:	*input-service-parameters* -> *inquirer*
Next:	*inquirer* -> <*output-service-parameters, action*> | NO-MORE-VALUE
Kill:	*inquirer* -> VOID
Check:	*action* -> YES | NO

- The second phase (performance) unrolls a classical two-phase commit protocol that ensures the atomic execution of a set of actions returned by different services in the previous phase. To achieve atomicity, the requester first attempts to reserve the resources needed by each action (verb **Reserve**) then, if successful, enacts all the actions (verb **Confirm**) or, if one reservation fails, cancels all the successful reservations (verb **Cancel**). The verbs of the performance phase have the following abstract signature:

Reserve:	*trans, action* -> ACCEPT | REJECT
Confirm:	*action* -> VOID
Cancel:	*action* -> VOID

- The third phase (notification) allows for asynchronous creation of new resources (verb **Insert**), which can be used to notify a situation.

Insert:	*service-parameters* -> VOID

3.2.2 Direct Method Invocations

A direct method is used to manipulate the resources of a single CLF object through a traditional remote method call. User interfaces are typical invokers of direct methods. The abstract signature is:

Perform:	*input-parameters* -> *output-parameters* | EXCEPTION

Direct Method Invocations are similar to Remote Procedure Calls as found in conventional middleware (e.g. Corba), renamed "Remote Method Invocations" in the Java world. In fact, CLF provides bridges to these types of middleware for the implementation the direct methods, in addition to its own implementation based on HTTP, where you can invoke a direct method by opening an URL, with various encodings of the input parameters and the returned results. In particular, simple user interfaces can very quickly be added to a CLF application using HTTP-based direct methods whose parameter encodings are directly supported by standard Web browsers (so called "form-data" or "url-encoding" for input parameters and HTML for the result). The perspective of a new generation of XML browsers makes this approach even more valuable.

3.3 Coordination scripting

The CLF coordination scripting facility takes full advantage of the object model. It enables high-level declarative specification of coordinated invocations of CLF object services. A coordination is viewed here as a complex block of inter-related manipulations (removal, insertion) of resources, held by a set of objects (called the participants of the coordination). Using rules, CLF scripts describe, the expected global behavior of such blocks in terms of resulting resource transformations, but abstract away the detailed sequencing of invocations of the CLF interaction verbs required to achieve such a behavior. It is this abstraction feature which considerably simplifies the design and verification of coordination scripts and makes them highly platform independent and hence, portable. It is also one of the main advantages of (and reasons for) the use of rules in the context of CLF. Of course, if, for a specific coordination purpose, rules are not judged appropriate, nothing prevents programmers from writing ad-hoc coordination programs explicitly unrolling the CLF protocol.

In a CLF application, coordination scripts are enacted by CLF objects called "coordinators", which are instances of a prototype developed with the CLF library and available in the CLF distribution. Like any CLF/Mekano object, coordinators manage resources, accessible through CLF services: these resources are CLF coordination scripts and the rules which compose them. When a script is inserted in a coordinator, it is immediately interpreted. Being CLF objects, coordinators can participate (i.e. behave as participants) in higher level coordinations, thus offering a reflexive model of coordination.

The CLF coordination scripting facility does not specify, per-se, any computing feature of its own. If computation is needed in a coordination (arithmetical computation, string manipulation etc.), it must be handled by a participant. This ensures a high degree of portability of the coordination scripts, which are completely independent of any computational model or language. However, in the CLF distribution, basic stateless computing participants are delivered with the coordinator prototype.

3.4 Component-based distributed architectures

CLF/Mekano is part of the new generation of component-based distributed architectures that integrate within the components, some basic functionalities which can then be used by generic middleware services. Components become "aware" of their potential involvement in a middleware service (e.g. "query awareness" for messaging services, "transaction awareness" for transaction services). The CLF protocol captures the aforesaid in a precise and concise manner: CLF components are query-aware

(through the negotiation and notification verbs of the CLF protocol) and transaction-aware (through the performance verbs of the protocol). The runtime libraries provided by Mekano to deal with these features, based on the systematic view of objects as resource managers, greatly simplify the design of such components.

Coordination awareness In CLF/Mekano, unlike other architectures, query awareness and transaction awareness appear as two sides of the same coin, namely "coordination awareness". They are not extra features added in the course of the evolution of an existing architecture (e.g. what Enterprise JavaBeans are to JavaBeans): coordination was present from the start[2] in a very synthetic form, and in fact leads the development of the architecture. The most visible practical consequence of coordination awareness is the existence of "coordinators" in CLF, which are very generic components tailored to coordination purposes. Coordinators, and in particular their resources (the coordination scripts) have no equivalent in other architectures. Coordination scripts offer a very compact way to describe operations which, if they had to be explicitly programmed, would result in heavy, error-prone code (even with the help of separate transaction and messaging services). Furthermore, the view of scripts as resources enables both flexibility and dynamicity in the design of coordinated behaviors: coordination is not drowned into code, but can be manipulated like any other resource. A typical CLF application requires about twenty rules for its coordination core. The amount of explicit code needed to achieve the same result would be incomparably higher, and would be extremely difficult to maintain and evolve.

The CLF coordinator and scripting facility appear naturally well suited to enact coordination of heterogeneous components, with a small amount of work, to make all of these components CLF-aware, i.e. compliant with the CLF protocol. The how-to and related issues are presented below.

Openness Another salient feature of CLF/Mekano is its openness, not bound to any particular language, communication protocol, or operating system. Moreover, the different phases of the protocol are open and the application builder is free to implement any behavior for each of the CLF verbs.

The Mekano library, in this case written in Python, provides a harness from which it is possible to call upon any piece of code written in C, Java, Python,...(to name only those languages that have actually been used in our applications). Parameters are assumed to be passed by value and are coded as simple strings which can encode any piece of information like a stringified structure (à la Python), a language dependent serialization of an object (Python, Java), a language independent serialization (Corba), an object name or identifier, a raw file, etc.

CLF interactions can be implemented on top of any communication protocol. The Mekano library currently allows for the building of components capable of performing CLF interactions through multiple communication protocols *at the same time*. This may be particularly useful when different cryptographic or session-based protocols have to be used with different participants in a CLF coordination.

The current Mekano library has been tested on top of NT and Unix (Solaris, Linux). We took advantage of the excellent portability of Python to essentially reuse the same code on the different platforms (in fact, only a small daemon used for application deployment had to be adapted to each platform).

Finally, the implementation of the different phases of the CLF protocol is fully open, and can easily be adapted to reuse existing functionalities provided by either the encapsulated component (e.g. databases, document management systems) or by the underlying framework (e.g. EJB).

3.5 Transaction and object model

The CLF transaction model does not specify per-se that all components involved in a transaction have to be fully compliant with the ACID[9] properties (Atomicity, Consistency, Isolation, Durability). These properties may be satisfied for a given transaction only if all components involved support ACID transaction or have been encapsulated by a CLF object in a way to achieve these ACID properties. However, it appears in practice that the target applications do not require such ACID properties primarily adapted to database management or bank account manipulation. Flexibility at this level simplifies the encapsulation of legacy components. Moreover, no support for nested transactions[14] has to be provided at the object level. Because of these few constraints it is very easy to integrate most components of the different platforms considered (i.e. Jini, Corba, EJB) and make them participate together in a CLF driven coordination. To make a component CLF-aware and thus able to participate in a CLF transaction the easiest way is very often to encapsulate the component with a thin layer that maps the CLF protocol API to the API of the component.

This approach has two main benefits. First, it is not necessary to modify the component and so legacy or non-proprietary objects or applications can be more easily reused. Second, it is possible, with the help of the Mekano library, to extend some properties (i.e. ACID properties) of a component especially if it is a resource manager.

This approach does not however enforce the use of the transactional services of the native framework of an encapsulated object. Which means that, the consistency of the

concurrent transactions - with those in the CLF environment - are not ensured. To avoid this problem, it is more appropriate to use the native transactional services of these environments. The next sections dedicated to implementation illustrates precisely the caveats and issues related to the use of these native transactional services.

3.6 Architectural solution retained

To summarize, our problem was to integrate coarse grain components from different heterogeneous environments and perform coordination across these components using CLF/Mekano. For this reason, we up to now presented the relevant constraints and problems we considered and what CLF/Mekano provides in order to make this feasible and, moreover, why it is so well suited to this kind of challenge.

CLF/Mekano is effectively well designed to involve components from other component-based platforms. It offers through the CLF scripting language and coordinator, an original and robust way to unify heterogeneous components and external transaction services while preserving a minimum of important properties needed to realize and ensure coherent transactions. The fact that these facilities are highly portable and system independent eases their integration without losing the possibility of achieving transactions or concurrently sharing resources managed by these components, through their native transaction services.

In conjunction with these possibilities, the CLF/Mekano environment is open and thus facilitates the wrapping (i.e. encapsulation) of heterogenous components, often without making the slighest modification. Moreover, it offers an infrastructure that can concurrently handle multiple communication protocols, is easily portable to different systems and remains language independent.

4 Implementation

To validate the architectural solution presented in the previous section, we considered three well-known component-based object-oriented distributed environments: Jini, EJB (Enterprise JavaBeans) and Corba 3.0. Our goal was to define and implement a bridge between each of these technologies and CLF. This includes mapping the CLF object model, the resource-based paradigm and the CLF communication protocol previously introduced. By doing this, we ensure that any Jini service, EJB bean or Corba component is seen as a regular component of the CLF/Mekano library, and can therefore be transactionally accessed and coordinated by the CLF coordinator. As described in section 2, we use a native transaction context per component (Jini, EJB, Corba) and we combine all of these transaction contexts into a general transaction managed by the CLF coordinator.

We have included pieces of “pseudo-code” within the following subsections, where we do not intend to detail all the aspects of the implementation but rather show the global mechanisms used and highlight some of the important interactions with the native APIs of each environment. On the purely technical side, we have used JPython[11] a Python interpreter built on top of a Java Virtual Machine to combine CLF components (mainly written in Python) with the APIs of Jini and EJB (written in Java).

4.1 CLF component

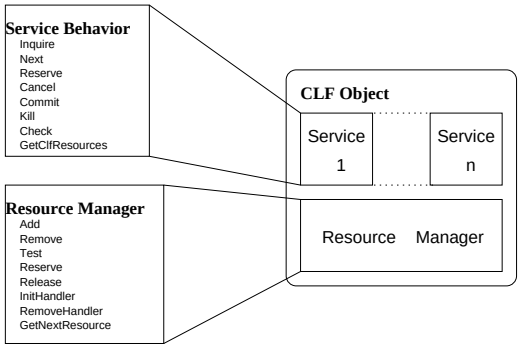

Figure 1. CLF component internal structure

The internal structure of a CLF object is given in Figure 1. Basically, we can act upon a CLF component at two levels: the resource manager and the service behavior level.

Service behavior Service behavior may be defined as the specific synchronous and sequential behavior of a service. For instance, to process a **Next** request, a service may have to block until some new matching resource becomes available. The service behavior specifies only the synchronous part of this operation, namely to check if a resource is available satisfying the service request, while the CLF runtime included in the component takes the responsibility of the asynchronous part of the operation, blocking the response and waking it up when required. Most of the time, the programmer may reuse a service behavior class of the Mekano library, either as such or in order to derive new behaviors. The kind of service behavior currently provided ranges from Simple Bag to Catalog or CloseDictionary[1].

A service behavior class should be compliant with the API sketched in Figure 1 that mainly defines the methods that are synchronously invoked by the service handler (part of the component CLF runtime).

Resource manager A service behavior defines the particular behavior of a service with respect to the CLF verbs, but abstracts away the precise nature of the resources it manipulates. On the other hand, a resource manager defines the way the physical resources (if any) are stored. The Mekano library provides a set of predefined resource manager classes, which can be customized for application purposes, ranging from simple TupleSpace or PersistentTupleSpace to FileSystem, LDAP or MySQL [1].

A Resource Manager should be compliant with the API sketched in Figure 1 that provides basic functionality for adding and removing resources, sequentially browsing a set of resources and managing reservations (invoked by the CLF runtime included in the component).

In the following sections we present, as examples, both the *service behavior* and *resource manager* encapsulation of external components. For Jini we have chosen to act at the level of the resource manager while for the EJB we acted directly at the level of the service behavior.

4.2 Jini

When looking for a Jini service to support our proof-of-concept implementation, we naturally came across JavaSpaces. It was the only freely available service supporting Jini transactions at the time of writing.

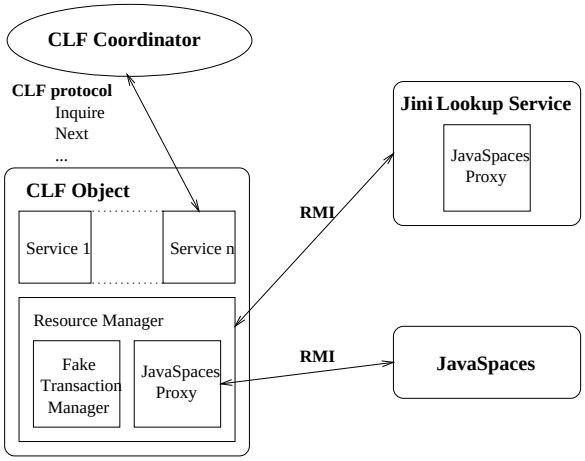

Figure 2. CLF and Jini

Mapping the CLF resource-based paradigm was quite straightforward, as JavaSpaces is nothing more than a classical tuplespace enhanced with the notion of type (JavaSpaces entries are full Java objects) and the Mekano library already offered a tuplespace resource manager.

A single CLF object encapsulates a JavaSpaces service, however, as CLF resources are not typed, we had to define several CLF services in the object, each one corresponding to a single entry class from the JavaSpaces. The complexity then moved to the CLF resource manager in charge of interfacing all these services with the JavaSpaces instance. Its main responsibility is to retrieve at initialization time, a JavaSpaces proxy from a Jini Lookup Service and then act as a client of the JavaSpaces reading, writing and taking entries. In addition to that, the resource manager does the packaging (resp. unpackaging) of CLF resources to (resp. from) JavaSpaces entries.

JavaSpaces Resource Manager

```
Add( rsc )
    entry = buildEntryFromRsc( rsc )
    JSproxy.write( entry )
    return entry.key

Remove( key )
    txn = txnDict[ key ]
    p = FakeTM.getParticipant( txn )
    p.commit( txn )

Reserve( key )
    txn = new Transaction( FakeTM )
    txnDict[ key ] = txn
    entry = buildEntryFromKey( key )
    JSproxy.take( entry, txn )
    p = FakeTM.getParticipant( txn )
    p.prepare( txn )

Test( key )
    if txnDict.hasKey( key )
        return true
    else
        entry = buildEntry( key )
    return ( JSproxy.read( entry ) != null )

TestStatus( key )
    if Test( key )
        return txnDict.hasKey( key )
    else
         return false

Release( key )
    txn = txnDict[ key ]
    p = FakeTM.getParticipant( txn )
    p.abort( txn )

 (InitHandler(), RemoveHandler() and GetNext()
omitted here)
```

Figure 3. Mapping the CLF onto Jini

The tricky part of the implementation was to map the two transactional models. Even if JavaSpaces provides an API to control its behavior as a transaction participant, this internal API is not directly accessible to every client. Only a Jini transaction manager, which manages the execution of the two-phase commit protocol, has this privilege. As we needed such fine-grained control, we had to include in our resource manager a lightweight “fake” Jini transaction manager which gives us access to the transaction API of a JavaSpaces object.

Figure 3 details how the CLF resource manager API has been implemented using the Jini APIs. The JavaSpaces service is remotely accessed through the *JSProxy* object. A

FakeTM object implementing the Jini *TransactionManager* interface gives access to the Jini *TransactionParticipant* interface of the JavaSpaces (this allows in particular the control of both phases of the two-phase commit protocol). The *txnDict* dictionary is used to store associations between resource keys and Jini transactions. The *buildEntryFromRsc* and *buildEntryFromKey* helper functions, construct a JavaSpaces entry from a CLF resource or a CLF resource key.

A call to the **Reserve** operation for a given resource (identified thanks to its resource key) starts a new Jini transaction inside which, an entry is taken from the JavaSpaces and then the transaction is set as "prepared to commit". Of course this Jini transaction may fail at any stage (e.g. a Jini transaction manager is accessing the resource concurrently with the CLF coordinator). Such a failure would be detected and handled as a *reject* of the CLF **Reserve** operation. Finally, either a call to the **Remove** operation commits the Jini transaction or a call to the **Release** operation aborts it.

This approach can easily be extended to other Jini services as the mapping of the transactional model and the techniques used in the implementation will still be valid. However, the mapping of the resource-based paradigm has to be different for services that do not manage "real" resources but rather have various side-effects or return the result of a computation (traditional method call). This is the case for the Enterprise JavaBeans, and an appropiated mapping is described in the next section.

4.3 Enterprise JavaBeans

The aim of this experiment was to build a generic CLF component to access any business method of any enterprise bean, taking advantage as much as possible of its transactional capabilities.

The resource-based paradigm has been adapted to match the semantics of a method call: a resource tuple contains the input parameters of the call. This is only one possible mapping, where we assume that the return value of the call is not important. Otherwise, we would have considered defining the first element of a resource tuple as the output parameter, and the others as input parameters.

A single CLF object corresponds with an enterprise bean, and each of its services corresponds with a method. There is no resource manager associated with the CLF object (Figure 4), as these methods do not manage "physical" resources but rather contain some complex business logic. Thus, building directly on the CLF service behavior, is a more suitable way to implement the mapping between a CLF service and an enterprise bean method.

The approach uses the Java Transaction API (JTA) available in every EJB container to have fine-grained control over the transactional behavior of the method call. It is worth noticing that the JTA is the interface for bean developers to the Java Transaction Service (JTS) bundled with the J2EE server. As no client-side support for the JTA is required in the J2EE specification, we create on the server (in fact on the EJB container) what we call a CLF gateway for each method call. This gateway is a stateful session bean that holds the transaction context for a particular method call and will be accessed from the CLF object through a proxy object.

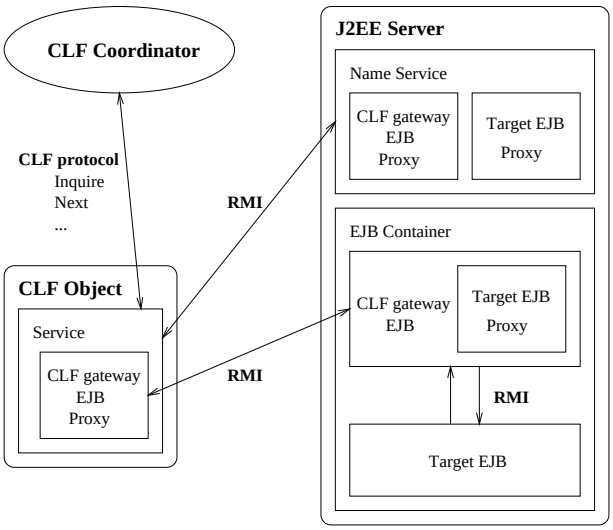

Figure 4. CLF and EJB

Figure 5 details how the CLF service API has been implemented using the EJB APIs and the previously described gateway mechanism. The *EJB* parameter holds a proxy to the bean encapsulated by the CLF object, and the *methodName* parameter holds the name of the bean method encapsulated by the CLF service. The *inqDict* dictionary is used to store associations between inquiries and method call parameters. The *gwDict* dictionary is used to store associations between inquiries and gateways. Each gateway is in fact a proxy object allowing remote access to a CLF gateway bean. These beans keep the whole context of a particular method call, i.e. the target bean reference, method name and arguments, and of course the transactional state which may be modified through the JTA.

A call to the **Inquire** operation locally stores the method call parameters. Then, a call to the **Next** operation creates a CLF gateway bean on the remote EJB container and a local proxy. At this point the context is set up. The **Reserve** operation performs the method call under a transaction. Either a call to the **Commit** operation then commits the EJB transaction or a call to the **Cancel** operation rolls it back.

EJB Service

```
Inquire( filter )
    inqId = inqId + 1
    inqDict[ inqId ] = filter
    return inqId

Next( inqId )
    rsc = inqDict[ inqId ]
    gw = CLFgateway.create( EJB, methodName, rsc )
    gwDict[ inqId ] = gw
    return inqId, rsc

Reserve( inqId )
    gw = gwDict[ inqId ]
    try
        gw.call()
        return true
    except
        return false

Commit( inqId )
    gw = gwDict[ inqId ]
    gw.commit()

Cancel( inqId )
    gw = gwDict[ inqId ]
    gw.rollback()

 (Insert(), Check() and Kill() omitted here)
```

CLF gateway

```
create( bean, method, args )
    (Using reflection we retrieve
    method from methodName)

call()
    SessionContext.getUserTransaction().begin()
    res = method.invoke(bean, args)
    return res

commit()
    SessionContext.getUserTransaction().commit()

rollback()
    SessionContext.getUserTransaction().rollback()
```

Figure 5. Mapping the CLF onto EJB

4.4 Corba

CLF currently provides the capability of interoperating with Corba objects using the FNorb ORB[8] through the IIOP protocol. Using these facilities and following the same approach as for the EJB framework[2], we can offer a comparable level of integration for at least the transactions. We are still working on this aspect but current results are encouraging.

5 Components in practice

In this section we introduce the prototype of an application where we take advantage of CLF to enforce a coordinated transaction across a set of services belonging to different infrastructures (namely Jini and EJB). Using the results of the work done around interoperability (see section 4), we were able to:

- define a set of CLF objects and services giving access to the existing infrastructure ;
- express interactions between services using rule scripts ;
- ensure that rule instances are transactionally executed for all the services.

5.1 Showcase application overview

The scenario considered for this particular application is one in which several customers want to purchase sets of items. These items are available from various providers at different costs. The goal of the application is to provide each customer with an optimal solution (in terms of cost) at a given point in time. If the customer approves a given offer then the order is committed.

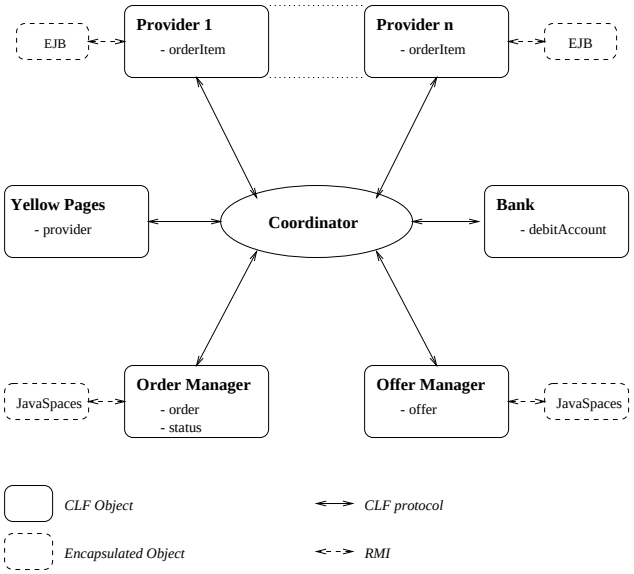

Figure 6. Showcase application

Figure 6 gives an overview of the application's architecture. The *Provider* objects encapsulate EJB-based applications running on each provider's site. The *Order Manager* and *Offer Manager* objects encapsulate Jini JavaSpaces to respectively store persistent data corresponding to customer orders and proposed offers. The *Bank* object is a CLF object already used in an e-commerce framework we have developed on top of CLF[4]. It could also be provided by an EJB application, a database or a legacy system in charge of managing account information for the customers. Finally, the *Yellow Pages* object is one we often use in CLF applications to register and retrieve object references.

[2]The approach of EJB and Corba 3.0 components is very similar.

```
# Finds an optimal offer for a given order
‘order( orderId, userId, itemId, qty ) @ ‘status( orderId, ‘‘open’’) @ ‘provider( providerId ) @
getCost( providerId, ‘‘getCost’’, cost, itemId ) @ offer( orderId, bestProvider, bestCost ) @ isLess( cost, bestCost )
<>- offer( orderId, providerId, cost )

# Enacts the ordering process for a given offer
offer( orderId, providerId, cost ) @ status( orderId, ‘‘offerAccepted’’) @ ‘order2( orderId, userId, itemId, qty ) @
orderItem( providerId, ‘‘orderItem’’, userId, itemId, qty) @ debitAccount(userId, cost*qty )
<>- status( orderId, ‘‘closed’’)
```

Figure 7. Showcase application rules

Figure 8 specifies the signatures of the service interfaces referenced in the rule scripts (Figure 7). A signature defines the input and output parameters (respectively on the left side and right side of the "`->`" symbol). Moreover, it describes how the mapping is performed by the *Coordinator* in order to resolve the physical address of the remote service. For instance, let's consider the *offer* interface. It has one input parameter, *orderId*, and two output parameters, *bestProvider* and *bestCost*. The **LOOKUP** directive specifies that the *Coordinator* will look up the *Offer* service of the *OfferManager* object. For a **DISPATCH** directive the object and service are specified at instantiation time by the first two input parameters. Finally, **ASSERT** or **COMPUTE** refer to a function locally executed by the *Coordinator* in order to perform a simple test or computation.

Figure 7 details the rules specifying the behavior of the application interpreted by the *Coordinator*. The first rule deals with finding an optimal offer for a given order. For each order that does not yet have an associated accepted offer, the rule goes to each provider and finds the one proposing the lowest price for a desired item. Once an offer has been approved by a customer the second rule enacts the ordering process. This second rule is particularly interesting as it illustrates the interaction of heterogeneous components within the same transaction; we describe it in detail in the next section.

5.2 A sample execution

An order is committed each time a complete instance of the second rule detailed in Figure 7 is executed by the *Coordinator*. This execution can be divided into three phases, corresponding with the three phases of the CLF protocol: negotiation, performance and notification (see section 3).

In the negotiation phase, using the **Inquire** and **Next** verbs, the *Coordinator* does its best to find a complete instance of the rule (i.e. an instance where all the tokens in the left part of the rule have been mapped to existing offers returned by the services). In our case this means that we have found an `offer` for which the `status` is *"acceptedOffer"* and from its `orderId` we have retrieved the corresponding `order`. We then have all the required parameters for the last two tokens: `orderItem` and `debitAccount`.

```
# Retrieves all orders, or details for an order id
order( orderId, userId, itemId, qty )
    -> orderId, userId, itemId, qty
    is LOOKUP OrderManager.Order

order2( orderId, userId, itemId, qty )
    orderId -> userId, itemId, qty
    is LOOKUP OrderManager.Order

# Retrieves status for an order id
status( orderId, code )
    orderId -> code
    is LOOKUP OrderManager.Status

# Retrieves offers for an order id
offer( orderId, bestProvider, bestCost )
    orderId -> bestProvider, bestCost
    is LOOKUP OfferManager.Offer

# Retrieves all providers
provider( providerId )
    -> providerId
    is LOOKUP OrderManager.Provider

# Retrieves cost for an item from a provider
getCost( providerId, method, cost, itemId )
    providerId, method, itemId -> cost
    is DISPATCH

# Orders a quantity of items for a user from a provider
orderItem( providerId, method, userId, itemId, qty )
    providerId, method, userId, itemId, qty ->
    is DISPATCH

# Verifies that cost is less than bestCost
isLess( cost, bestCost )
    cost, bestCost ->
    is ASSERT

# Debits an amount from a user’s account
debitAccount(userId, amount )
   userId, amount ->
   is LOOKUP Bank.DebitAccount
```

Figure 8. Showcase application services

In the performance phase, using the **Reserve**, **Commit** and **Cancel** verbs, the *Coordinator* enacts a two-phase commit protocol.

In the first round it tries to reserve all the actions associated with offers returned in the negotiation phase (i.e. `offer`, `status`, `order`, `orderItem` and `debitAccount`). For the `offer`, `status` and `order` tokens this will result in the creation of a Jini transaction (one per service) under which the resource will be taken from the underlying JavaSpaces. For the `orderItem` token this will re-

sult in the creation of an EJB transaction (in the object referenced by the `providerId`) under which the *"orderItem"* method will be executed with `userId`, `itemId` and `qty` as parameters. Finally, for the `debitAccount`, it is verified that the withdrawal operation can be performed and a lock preventing other withdrawal operations is set.

If any of the reserve operations fail then all the services will be asked to cancel. For the `offer`, `status` and `order` tokens this will abort the corresponding Jini transactions and the resources will not be removed from the underlying JavaSpaces. For the `orderItem` token this will rollback the method execution. Finally, for the `debitAccount` token, this releases the lock.

If all the reserve operations succeed then all the services will be asked to commit. For the `offer`, `status` and `order` tokens this will commit the corresponding Jini transactions and the resources will be removed from the underlying JavaSpaces. For the `orderItem` token this will commit the method execution. Finally, for the `debitAccount` token this will perform the withdrawal and release the lock.

In the notification phase, using the **Insert** verb, a resource composed of the `orderId` and the *"closed"* literal will be inserted in the JavaSpaces encapsulated by the `status` service.

6 Conclusion and future work

The showcase application presented here is mainly focused on showing how CLF tackles the issue of heterogeneous component coordination. Even though the scenario has been relatively simplified, it illustrates the mechanisms used to handle the difficult parts of the problem and that these mechanisms are generic enough to be used in many different contexts. We believe that our approcach is particularly well suited to collaborative applications across virtual enterprises, where each entity relies on its own platform and legacy applications.

In this paper we have proposed a solution to coordinate components from different *"protocol-aware"* component frameworks. In particular we have described a way to involve components coming from different platforms in a distributed transaction. In addition to this, the distributed transaction can be concurrently enacted together with transactions performed by the native transaction manager of each platform. Indeed, applications within an entity of a virtual enterprise need to run in their own restricted environment, yet have to interface some of their components with larger cross-enterprise applications.

As no single platfrom succeeds in solving all the problems, we have no choice but to cope with heterogeneous components. Moreover, interoperability should not be restricted to simple RPCs but rather rely on more elaborated interactions between the components.

Having mainly considered the distributed transaction problem, several other aspects are beyond the scope of this paper such as negotiation, error handling and performance. These will be part of the next steps in our investigations.

Acknowledgements

We are grateful to Jean-Marc Andreoli and Irene Maxwell for their helpful comments on this paper.

References

[1] J.-M. Andreoli, D. Arregui, F. Pacull, M. Riviere, J.-Y. Vion-Dury, and J. Willamowski. CLF/Mekano: a framework for building virtual-enterprise applications. In *Proc. of Enterprise Distributed Object Computing EDOC'99*, Mannheim, Germany, 1999.

[2] J.-M. Andreoli, S. Freeman, and R. Pareschi. The Coordination Language Facility: Coordination of distributed objects. *Theory and Practice of Object Systems (TAPOS)*, 2(2):635–667, 1996.

[3] J.-M. Andreoli and F. Pacull. Distributed print on demand systems in the Xpect framework. *Journal of Distributed and Parallel Databases*, 7(2), 1999.

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

[5] J.-M. Andreoli, D. Pagani, F. Pacull, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, 31(2–3):179–203, 1998.

[6] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[7] P. Bernstein and N. Goodman. Serializability theory for replicated databases. *Journal of Computer and System Sciences*, 31(3):355–374, Dec. 1985.

[8] M. Chilvers. http://www.dstc.edu.au/fnorb.

[9] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1990.

[10] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[11] J. Hugunin. Python and Java - the best of both worlds. In *Proceedings of the 6th International Python Conference*, pages 11–20, San Jose, Ca., Oct. 1997.

[12] V. Matena and M. Hapner. Enterprise JavaBeans. Technical report, Sun Microsystems, Palo Alto, 1998.

[13] S. Microsystems. Java Remote Method Invocation specification. Technical report, Sun Microsystems, 1997.

[14] J. Moss. *Nested Transactions an Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass., 1985.

[15] OMG/CORBA. http://www.corba.org.

[16] Sun Microsystems. RPC: Remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[17] J. Waldo et al. JavaSpaces specification - 1.0. Technical report, Sun Microsystems, Mar. 1998.