An Object implementation of Network Centric Business Service Applications (NCBSAs):

Conversational Service Transactions, a Service Monitor and an Application Style

Asit Dan and Francis Parr

IBM T. J. Watson Research Center
Hawthorne, NY 10532

*The Internet stretches traditional strict transaction processing concepts in several directions. First, transactions spanning multiple independent organizations may need to address enforcement of pairwise legal agreements rather than **global data consistency**. Second, a new transaction processing paradigm is required that supports different views of **unit of business** for all participants, i.e., service providers as well as end consumers. There may be several related interactions between any two interacting parties dispersed in time creating a **long running conversation**. Hence, persistent records of business actions need to be kept. Additionally, some actions and groups of actions may be cancellable (however, this may not mean that all effects are undone, e.g., non refundable payments). Finally, the greater variability in response time for network computing creates a need for asynchronous and event driven processing, in which correct handling of reissued and cancelled requests is critical. This paper presents a framework for development of NCBSA using CORBA services while satisfying the above requirements.*

1. Introduction

1.1 Network Centric Computing is a New Context for Transaction Processing

The Internet and related technology trends (e.g., in end-user presentation interfaces and networking) have revolutionized the business transaction processing environment. *Business-to-end-user* and *Business-to-Business* interactions are both based on automated processing of asynchronous messages (i.e., service requests and responses). These interactions need to follow a long running conversation flow and need to have their effects made persistent. In such a *network-centric* business environment, independent business applications are created by autonomous organizations as ***Conversational Service Transactions (CST),*** where the global or complete knowledge of interactions amongst all applications can not be known. Since transactions and transaction processing systems lie at the core of business computing, it is important to explore the extensions required to the classical transaction monitor [8,3,15] for supporting such new types of applications and how *network applications* can be developed in this new environment. (The additional requirements of such applications are summarized in Section 1.2).

In [10], we advocate a service independence approach to application development and propose an execution environment for supporting long running conversational applications. We call the proposed approach *COYOTE:* for *COver Yourself Transaction Environment*, to emphasize that when several organizations participate in a business interaction across a public network, each organization can manage and be responsible for its own commitments and expectations but there is no party globally responsible for all activities contributing to the business interaction. In fact different participating organizations may have different views of where the boundaries of this particular business interaction may lie. Localized execution of an interaction of a service application may well be a classical ACID transaction[15] or may even follow various extended transaction models already proposed in the literature[12,14,21,4,24,7,1].

In this paper we first briefly summarize the requirements of a BSA , and the framework for development of a BSA. We then propose extensions to CORBA services for object based implementation of such business service applications.

1.2 Requirements on Infrastructure for Network Centric Business Service Applications

In addition to the traditional requirements for application development and runtime environment, service applications in a network centric environment need to have the following additional properties making them *Conversational Service Transactions (CST):* [10]

- **Service independence**: Each CST may be developed by an independent party where the complete interactions amongst all applications can not be known. Hence, the application development process is incremental.

- **Localized execution environment:** The processing of business logic is distributed and each component (i.e., CST) is run under a local application monitor. This also implies that application development and execution environments may be heterogeneous, and very little should be assumed of other execution environments.

- **Well known interfaces:** The interfaces of each CST as well as their operation semantics need to be known to other invoking applications. The interfaces are either known universally (e.g., EDI transactions) or are defined by the interacting parties via *service contracts* (see below).

- **Long running conversation:** Between any two parties, there may be many related interactions dispersed in time (e.g., delayed cancellation or modification of an earlier service request). Hence, the intermediate computation states of a conversation need to be durable.

- **Asynchronous invocations:** individual service action requests (making up the long-running conversation between client and server) are asynchronous, i.e., response to a request may not arrive immediately. There are many factors contributing to this requirement. First, since very little can be assumed of other CSTs and their execution environments, the response time of other CSTs may be unpredictable. Second, a CST itself may request services of other CSTs, adding to this unpredictability. Some CSTs may also require inputs from human operators. Finally, network response time and disconnected operations will also add to this unpredictability.

- **Compensation:** Since a CST may be long running, and traditional transactional execution of business logic across multiple organizations can not be assumed (see, further details in subsequent sections), *Compensation* of an earlier service request is a strong requirement. In real-world operations, this translates to refund of an earlier purchased item, cancellation of a reservation, exchanging items or changing the attributes of a previous request.

- **Co-ordinated invocations:** In many applications, it is natural to request a set of complementary services from independent applications in an *all-or-nothing* manner of execution. An example of this requirement is co-ordinated purchases of items, such as, services of Hotel and Airline. The underlying runtime environment should ensure this all-or-nothing execution behaviour.

- **Service contracts:** A service contract between two interacting parties (i.e., CSTs) documents their expected behaviours. It includes issues such as the interfaces that can be invoked, how long a

conversation needs to be maintained and whether or not a service request can be compensated. The service contract needs to be enforced during execution time.

- **Query of interaction history:** A CST may need to query all its durable past interactions with other CSTs and the current states of interactions.

- **Site Autonomy, Privacy and Implementation Hiding:** A CST should be able to conceal from its requesters, whether parts of the service have been subcontracted out into the network; conversely the identity of the requester may sometimes need to be unavailable to any providers of subcontracted services.

1.3 Outline of the Paper

The remainder of the paper is organized as follows. In Section 2.1, we outline the Coyote approach: the conversational service transaction model, the Coyote monitor services and the application development model. We further illustrate these concepts with an example in Section 2.2. In Section 3, we describe how an object implementation of NCBSA and service monitor can be realized by extending the CORBA services. Section 4 contains the summary and conclusions of the paper.

2. An overview of the COYOTE Approach

In this section we explain at a high level our proposed solution to meeting the needs of network centric service applications in terms of three components:

- *Conversational Service Transaction model*
- *Coyote Monitor services*
- *Coyote application development model.*

We then illustrate the approach with an example.

2.1 Components of the Approach

2.1.1 Conversational Service Transaction Model

We treat requests for services as the fundamental building block for network centric business service applications and define a simple integrity model for **conversational service transactions**. Each organization which wants to provide a network centric business service application (i.e., CST) defines a set of well defined service access interfaces. The examples of such independent NCBSAs are Hotel services, Airline services, etc. The interfaces provided by each NCBSA can be invoked by remote and/or other independent NCBSAs to access services provided by this one (see Figure 2). We further advocate the use of an extended transaction model and a monitor supporting such pairwise interactions. The key concepts are:

- Clients get service by having a *conversation* with the application monitor at the server node.
- The service requests which flow on these conversations are *encapsulated*, i.e., the client sees only a well defined service interface and does not know whether the implementation of the service involves subcontracted services potentially at other nodes.

- Valid and allowable service requests are defined in service contracts between interacting parties, and the service requests are monitored for enforcing this contract.

- A client can request cancellation of a specific service request, a predefined grouping of requests or an entire conversation. However, there is no guarantee that the identified services will be completely undone; rather an application defined *compensation* action will be executed.

- The monitor also provides runtime service interfaces to its local CSTs for co-ordinating their requests to other CSTs. A set of service requests to other CSTs can be grouped as a *compensation group* in which case either all will be executed successfully or any which were executed will be compensated.

- The monitor also provides to its local CSTs a persistent application level log and execution of service requests reliably in the face of reissued requests. This is required especially when response times (for the possibly distributed processing) do not meet the clients expectations.

2.2 Transaction Monitors and Specific Features of a Coyote Monitor

Coyote is a transaction monitor and hence, provides all the basic services expected of a transaction monitor. The basic functions of a standard transaction monitor are well known, e.g. CICS, Encina, Tuxedo [3,15,9]. We summarize them briefly here in order to highlight the additional properties of a Coyote monitor.

Transaction monitors are responsible for scheduling transaction invocations. Transaction programs or applications are registered to the transaction monitor in an administrative operation. Each registered application has a function name known to clients so that they can request it; associated with this function name is a registered entry point or program which is scheduled when a request for that function is received.

Incoming requests to the transaction monitor appear as messages typically from remote clients. An incoming request is logically queued by the transaction monitor until computing resource is available to execute it. This occurs when some other transaction completes and frees up a process or thread in a pool managed by the monitor. When a thread becomes available, the transaction monitor allocates it to the first queued incoming request, starts on the allocated thread the application program registered to the requested function, passing it any additional parameter data also in the request.

Having started the transactional application in response to an incoming message, the transaction monitor is responsible for supervising the execution of the transactional application. In particular it will intercept all outbound requests (to recoverable resource managers such as databases that may be local or remote) and to other remote transaction monitors if this transaction monitor can participate in managing a distributed transactional protocol.

Finally the transaction monitor is also responsible for the transactional behaviour of the applications which it monitors. Definition of standard ACID transactional properties (i.e., atomicity, consistency, isolation and durability) can be found in [16,15]. A primary function of standard transaction monitors is to provide a commit/abort protocol. This allows a transactional application to be ABORTED if either a serious error is detected by the monitor during processing of the transactional application or if the application explicitly requests it. The transaction monitor will then ensure that all effects of processing on behalf of this transaction instance are undone in the persistent resources of the participating resource managers (e.g., databases).

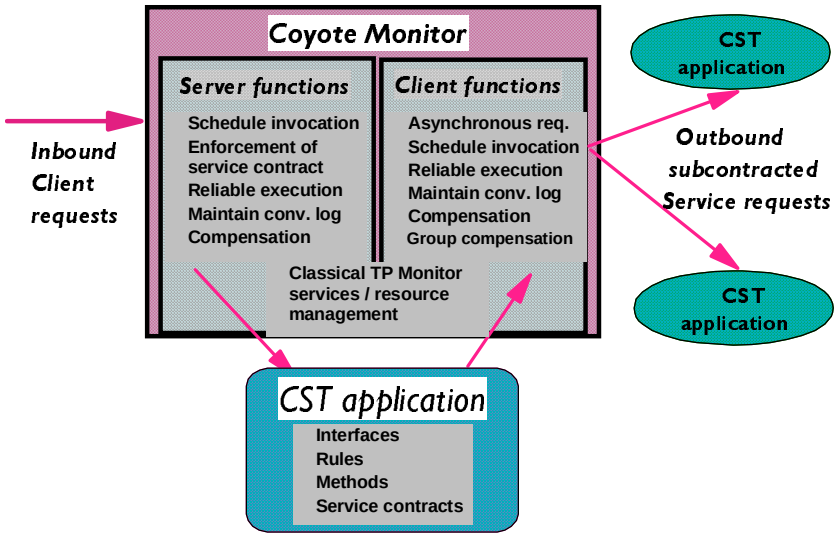

Figure 1: Diagram of Coyote Monitor

A schematic
view of a Coyote monitor is shown in Figure 1. At this level the architectures of a classical ACID transaction monitor and a Coyote monitor are essentially the same. However, a Coyote monitor differs from above summary description of standard transaction monitor functions in several respects.

- A server application started by a Coyote monitor is not by default an ACID transaction; rather it is a durable conversation for which there is no ``system guaranteed" abort operation to undo all its effects.

- An application level log of all interactions with the client and other remote resources and transaction managers is maintained by the Coyote monitor.

- A service application registered with the Coyote monitor can have a group of programs or interfaces associated with it. The interfaces define the primary action, and its associated application defined compensation and modify actions.

- A logical conversation thread may involve related interactions dispersed in time. Hence, the monitor maintains persistent conversation states and an interaction history.

- The Coyote monitor has special support for enabling reliable execution of a service request on behalf of a particular conversation (with a particular user) even if action requests are reissued, and is reliably compensated (by the application defined compensation action) if the user so requests.

- The Coyote monitor checks validity of invocation sequence as defined by the application during its registration.

- The monitor also enforces service contract, e.g., checks to see if this user is allowed to invoke this method at this point in the conversation. Other service contracts may enforce even more general

state validation.

- The Coyote monitor provides additional features to compensate an entire conversation or a defined group of service requests within the conversation, should compensation of the group be explicitly requested or automatically if some essential request within the group fails to execute successfully.

In short the novel features of a Coyote monitor is that it shifts the definition of a transactional application away from the concept of a system guarantee that all persistent effects are removed on failure, towards supporting the concept of persistent conversations in which it is easy to provide and manage application defined compensation of actions and provide the end-user with a reliable view of cancelling, reissuing and modifying particular service requests.

2.3 Coyote application Development model

A network centric service application has to guide a client which may be an end-user or possibly another unknown program through a unit of business. This is typically a long running interaction, in which there are short bursts of automated activity at the Coyote server, interspersed with periods of waiting for, subcontracted servers in the network to respond, the user to decide what to do next or activities to happen in the real world as part of the service fulfilment, goods to be moved, money to transfer or time between a reservation and use to elapse.

Thus a Coyote application consists of:

- **Interfaces:** its service interfaces (specifically service contract) presented to clients; the application will also depend on the service interface definitions for any services which it uses from other applications in the network in a subcontracting mode.
- **Action methods:** the implementations of each short running action which can occur in providing the service; each action involves responding to a specific event, processing based on this event and the state of the service conversation, and sending out messages and action requests to the client and other servers.
- **Scheduling rules:** the definition of the application's trigger events, i.e., combinations of response messages, time-outs and user requests, which will trigger execution of an action by the application, and the rules to determine which action is to be scheduled when events occur.

The developer of a network centric service application is required to provide all of the above information. The Coyote monitor plays the role of enforcing constraints defined in the service interface, gathering and saving the event data, interpreting the scheduling rules of the application, and then triggering the execution of action methods for the application following those rules.

We claim that the Coyote approach provides both a convenient and powerful structure for developing network centric service applications and a coherent model for characterizing their behaviour.

2.4 An example: conference trip reservation

Consider a typical multi-organization long running business application: *Travel planning including conference registration*. [1] Figure 2 shows the participating organizations and their interactions. The *end-users* need to make complete travel plans associated with attending this conference. Each may run a local travel arrangement application on his/her desktop/laptop, but is more likely to contact an intermediary, *Travel Service* provider, that co-ordinates various end-user service requests, and maintains special business relationships with various business service providers (e.g., airlines, hotels). As part of the conference registration, the conference organizers provide (via the *Conference service application*) not only a seat at the conference and conference proceedings, but also arranges hotel accommodations for the entire duration of the conference. The conference organizers make special arrangements with the hotel (via the *Hotel Service application*) to provide this accommodation. They also collect appropriate fee (via the *Acquirer gateways*) through the credit service providers (e.g., Amex, Visa, MC). The end-user provides all the necessary information by processing an HTML form, and is typically unaware of all the business activities behind the scene.

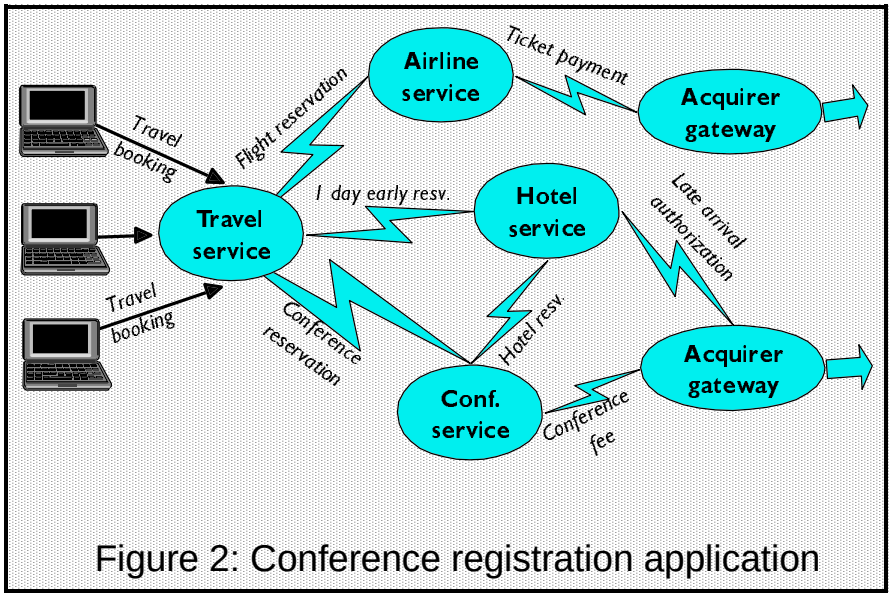

Figure 2: Conference registration application

An end-user may also require a separate arrangement with the same or another hotel for an extended stay unrelated to the conference. For example, the end-user may decide to arrive early for some unrelated business activity (e.g., giving a talk at the local University) and stay a day longer beyond the conference for some sight-seeing. Additional activities may depend on various factors: availability of suitable flights, availability of hotel accommodation, and of course, no conflict in schedule. The conference provider is clearly not interested in providing services beyond what is absolutely required for conference attendance. The unpredictable schedule of a busy client may also require partial attendance or modification of the schedule on-the-fly.

[1] Note the multi-organizational, service independence and long running aspects of this example in contrast to the typical version of this example used in the earlier literature [24,4,12].

The key requirements imposed by this overall application on the participating organizations and their underlying systems are as follows. Multiple organizations (e.g., Service providers for Travel, Conference, Hotel, Airline, etc.) need to interact to provide all the required services to the end-user. All the services demanded by an end-user may not be known in advance, and may change during the course of the conference. For example, the requested and granted services may be cancelled by (or on-behalf of) the end-user at a later time. However, each organization is responsible for delivering its services. Each service provider develops its own application without the a-priori global knowledge of interactions across all organizations. Finally, the organizations do not need to be in constant communication during the course of this long running overall application.

An important point is that cross-discounting deals between parties, i.e., 10% rebate at hotel X if airline Y is used (where half the discount to be paid by airline Y) should be easy to add to an existing set of service applications. The Coyote monitor and the service contract provide exactly the structure which facilitates customization of this nature.

Multiple interactions between server parties (e.g., Registration and Cancellation) may be related. Note that implementation of the ``registration at the conference site" service, results in invocation of services that atomically perform reservation of a hotel bed and conference registration fee payment, as well as local book keeping such as updating the number of proceedings to be printed, etc.. Here, each transaction accesses/modifies its local database(s) in an atomic fashion, and hence, preserves desired integrity relationships within a site. Each organization is also responsible for managing its own interactions with other organizations and hence needs its own persistent application log of the interactions between them. In the above example, the travel server co-ordinates its interactions with conference, hotel and airline sites. The conference program server is responsible for ensuring all services that come with a conference registration, and co-ordinates its interactions with the hotel and the acquirer gateway servers. The end-user or the travel server never gets involved in the conference program server backed activities.

Each site also passes minimal (essential) information about its clients to other servers as well as hides its implementation of the ways it provides the services, particularly if it is dependent on the services of other sites. Without such information barriers, all participating sites may be aware of the relationship across all other (potentially hostile) sites. It may also jeopardize the businesses of intermediate sites since the end-client may directly contact final service providers. Additionally, the global state maintenance may impose unduly burden on each service provider.

The proposed Conversational Service Transaction model and runtime services provided by a Coyote monitor aids in the development of NCBSA. In the next Section, we will explore how these services can be provided to an oo implemented NCBSA.

3. An Object implementation of Network Centric Business Service Applications (NCBSAs)

The preceding sections defined the need for NCBSAs and the requirements which must be met by an infrastructure for them. However, this was done with no reference to objects. In this section we take up the question of how one would implement both the NCBSAs and their infrastructure using Object technology and concepts. This adds a level of detail both to the description of NCBSAs and their required infrastructure; it also raises some interesting questions about ways in which Object standards could be extended to assist in the evolution of cross enterprise electronic services.

3.1 Service Invocations and Service Contracts

Being able to contact, uniquely identify, and interact with service invocations is a key part of the NCBSA economy. Hence we will need a *ServiceInvocation* class. Particular services such as:

- combined conference trip booking service
- airline reservation service
- hotel booking service

would be subclassed from this *ServiceInvocation* class.

It is usually the case that *ServiceInvocation* objects have to be persistent, e.g., a travel booking has to survive from the time that the booking is made to the time when it is used, possibly months later. During this lifetime there will be a number of specific interactions with the client of the service defined as *actions* in the preceding section.

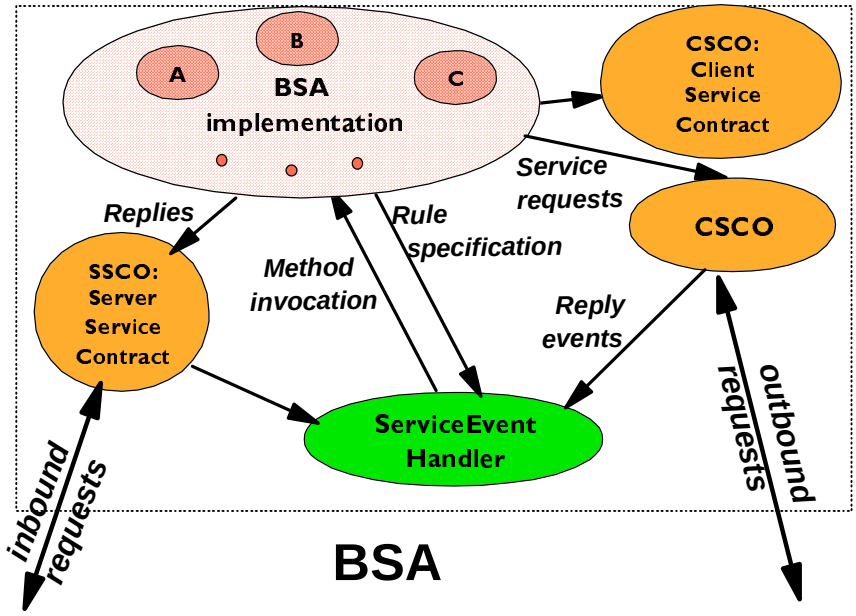

Figure 3: Network Centric Business Service Application

These actions are an important part of the interface of the service invocation, but are associated with the implementation of the service invocation only via a level of indirection. Hence rather than treat these actions as methods of the *ServiceInvocation* object directly, we introduce a *ServiceContract* object associated with each *ServiceInvocation* object defining its interface (see Figure 3). The methods of the *ServiceContract* object will be the actions by the of the service invocation, i.e., the individual requests the client can make on this service. To illustrate, in the hotel booking example, creation of booking, upgrade or

date change of a booking and cancellation are independent actions on the hotel service invocation. These functions are then the defined methods on the service contract.

3.1.1 Properties of *ServiceContracts:* sequence checking for reliable execution

Besides defining the set of action interfaces of a service invocation, a *ServiceContract* contains all information which defines the responsibilities undertaken by the provider of the service. The *ServiceContract* Object (SCO) provides information on the responsiveness committed to by the server. This may be provided in the form of the mean and some high percentile response times for responses to action requests which could be used by clients to set reasonable time outs. There could also be a more general indication of whether processing was interactive or batch oriented.

Another function of the SCO is that it defines sequencing rules for valid sequences of actions within a service invocation. This is particularly valuable in a network centric environments where there is great variation both in message delivery time and in processing response time. Service requests and their actions are uniquely numbered. The *ServiceContrac*t object implementation enforces these sequencing rules, thus eliminating invalid or duplicated requests. Thus a series of calls to a *ServiceContract* object, newBooking, newBooking, upgrade, cancel, upgrade, cancel, will actually be received at the *ServiceInvocation* object as the filtered sequence: newBooking, upgrade, cancel. The sequencing rules cause two newBooking actions to a single service invocation to be treated as duplicates and cancel is supposed to be a "final" action which cannot be followed by other actions on the same service invocation. Other invalid sequences would be reported back to the client as errors. Finally, *ServiceContract* object (which is created upon invocation of a BSA by an user and based on the service contract defined a priori between the invoking user and the service provider) decides which
action requests are to be permitted.

In summary, the *ServiceContract* interface defines properties of the service made visible to external clients usually in other organizations. The *ServiceInvocation* offers an interface to the implementation of the service, called from within the service provider's organization.

3.1.2 Separating monitor checking aspects of a service from the service implementation

An alternate way of looking at this separation between *ServiceContract* and Service Invocation is as follows. In providing business services in a conventional network environment the value of transaction monitors is widely accepted. Object Request Brokers (ORBs), as presently defined by OMG standards provide a limited set of the scheduling, interface enforcement and reliability management provided in existing procedural transaction managers. In a network centric business service environment, extensions to transaction manager functionalities are required to handle the greater openness, extended transaction models and more ad hoc cross organizational interactions of a public network environment.

In these terms the *ServiceContract* object represents processing to provide rule and interface enforcement for a BSA. Users of this service see and make method calls on the *ServiceContract* object, not the *ServiceInvocation* object which implements the service. The *ServiceInvocation* object represents the actual implementation of the service and has method interfaces which would be called by the transaction monitor after it had allocated an execution thread to an incoming request.

3.1.3 Client and Server Views of the Service Contract Object

A service contract represents the agreement between client and the provider of a service covering the rules of interactions and any guarantees of service quality to be supported by the server. The enforcement of this contract is actually distributed, and involves state management and checks at the both client and the server of this service. We refere to these objects as the client and server views of the service contract objects, i.e., CSCO and SSCO. Both of these objects are generated from the service contract as part of the Coyote application development tools. This is illustrated in Figure 3. The inbound requests are managed by the SSCO while outbound interactions with one or more sub-service providers are managed by the corresponding CSCO objects.

The CSCOs receive all outbound requests from the *ServiceInvocation* object to the outside world (i.e., various sub-service providers). Part of the CSCO interface, is a method through which requests can be made identifying a target *SSCO* and an action call on it. The *CSCO* just converts these outbound requests into one way method calls on the target S*erviceContract* object. The service provider calls back on completion to a reply method, also defined in the CSCO interface. The SSCO and CSCOs can recognize incoming events and parse when an event is a reply to an outstanding request for service or a new service request. They also persistently log the requests and replies so that the service invocation can understand after a failure what has been asked of the rest of the network.

3.1.4 *ServiceEventHandler* and asynchronous responses to subcontracts of the service implementation

A Network Centric Business Service Application (NCBSA) is typically incremental. It provides a *ServiceContract* interface, but its service implementation makes wide use of other services available in the network. For the best response time to clients, these "subcontract" service invocations will be made in parallel (since it is assumed they are executing on independent server engines scattered across the network). Hence the action requests made to subcontract services will need to be asynchronous calls.

In this environment, although service invocations last for a long time, they are actively executing only in brief bursts triggered by arrivals of requests from the service invocation client, and asynchronous replies from subcontracted services. It is convenient to structure the *ServiceInvocation* implementation as a set of internal methods and a set of event driven scheduling rules. Once again we separate out the rule driven scheduling functionality of the monitor into a separate object , i.e., namely the *ServiceEventHandler*. There is one *ServiceEventHandler* instance for each *ServiceInvocation* instance. The methods in the ServiceInvocation object are now the segments of algorithmic processing which are shceduled in response to some trigger event following the rules processed by the *ServiceEventHandler*.

More specifically, the *ServiceEventHandler* is driven whenever a new reply or message arrives from a subcontract or a new request arrives from the client of this service invocation. Using knowledge of which other events and replies have already been received, together with the rules defined in the *ServiceEventHandler,* the appropriate method of the *ServiceInvocation* is allocated a thread and called.

This observation reinforces the need to separate out *ServiceContract* object from *ServiceInvocation* object. Although the *ServiceContract* interface defines as its methods the set of action requests which can be made, the implementation (i.e. the set of methods on the *ServiceInvocation* object) will *not* be a set of independent procedures - one for each defined action. In general in the presence of asynchronous responses to subcontracted requests, a service invocation which allows clients to make action requests ***A*** and ***B,*** and processes action ***A*** by sending out subcontract network requests ***AF1, AF2*** would be more likely to have a set of methods:

method1 - in response to ***A***: sends out sub requests ***AF1, AF2***

method2 - timely relies to ***AF1*** and ***AF2*** are received; respond to ***A***
method3 - a request for B is received before responses to ***AF1, AF2*** from a preceding ***A***
method4 - replies to ***AF1*** and ***AF2*** are not both received in time: report progress only on outstanding request
method5 - handle late arriving responses to ***AF1*** or ***AF2*** after progress report to client.

The above example shows that even though the client has a set of actions which can be applied to a service invocation, and are defined as the methods of the *ServiceContract*, the *ServiceInvocation* is best implemented as a *ServiceEventHandler* with rules for choosing what action should be scheduled in response to a particular network event and the *ServiceInvocation* object itself whose methods are the specific processing called as a result. The *ServiceInvocation* methods provide a complete implementation of the service, but not in ***1-1*** correspondence with the actions callable by the client.

3.1.5 Further comparison of this object structure with Transaction Monitors

Transaction monitors classically completely surround the transactional applications which they support. A monitor provides an external interface to clients used to receive requests and drive the appropriate transaction applications. It also intercept all **outbound** requests from the transactional application to databases, file servers or services on other nodes, to insert hooks for transactional co-ordination, scheduling and monitoring. Orbs have not as yet defined interfaces to play this surrounding role. But in order to have a productive environment for developing NCBSAs using object technology we need to be able to define the necessary infrastructure support in object terms. The *ServiceContract* captures functionality which a monitor would have to provide as **inbound** request processing; the *ServiceEventHandler* is providing the monitor services for **outbound** request handling.

Transaction monitors build their **inbound** and **outbound** request handling support for a given application based on metadata provided when the transactional application is registered with or defined to the monitor. In our approach the *ServiceInvocation* object provides the essential implementation while he *ServiceContract* and *ServiceEventHandler* provide the application specific metadata and the general processing which extend the Orb to enable it to function as a more general purpose monitor, meeting the needs of NCBSAs.

3.1.6 Compensation and Compensation Groups

So far we have said nothing about transactional semantics and co-ordination other than mentioning that this responsibility is supported in the *ServiceEventHandler*. Now in the world of business and of NCBSAs there is little reason to expect service invocations to be "abortable" at all times - in the sense that all permanent effects can be removed. Some services involve payments which are not refundable; bookings may have been made which beyond some point in time cannot be cancelled. Hence we talk about *compensation* and *cancellation* of services rather than abort as it used for transactions.

Compensation of service may occur at several levels:

- **compensation of an individual *action*:** - action ***A*** has an inverse action ***undoA***; clients of the service containing action ***A*** may use the inverse to undo it.
- **compensation of a *ServiceInvocation:*** cancellation is a known action defined for many (but possibly not for all) *ServiceInvocation* classes; it will end the *ServiceInvocation* instance stopping any subsequent actions to that instance. it will provide as complete a *cancellation* as is possible.

- **compensation group**: a *ServiceInvocation* may sometimes want to group together several subcontract requests and have either all or none of them succeed. In addition, it may want other parts of the *ServiceInvocation* to be executed conditionally on the success of one of these groups.

The implementation of CompensationGroups is straightforward. If all service invocations in the group succeed, the group has succeeded; if one or more fail then the CompensationGroup is responsible for calling the *cancel* method on all the other successful members of the group. A CompensationGroup is defined by a BSA implementation which passes it as a rule to its ServiceEventHandler.

4. Summary and Conclusions

Service Transactions are an effective model to capture the semantics for adding service application to a network centric environment. With the Coyote monitor services, we have defined the critical services which the transaction infrastructure need to provide to support network centric service applications. Key technical features provided by this monitor are:

- Transaction monitoring and Logging services
- Compensation model, Group compensation
- Automated invocation of applications and services
- Persistent queryable conversation state
- Reliable execution of service requests
- Services for generating Internet (HTML or Java) formatted output from the transactional environment.

The appropriate model for developing applications in this environment is based on:

- service actions which are themselves atomic
- long running persistent conversations built from them
- scheduling rules identifying the ``next" action in
- response to messages or events
- use of the Coyote coordination and compensation services

Finally, for implementation of these applications and infrastructure, we have introduced the creation of *ServiceContract* object and *ServiceEventHandler* objects dynamically per service invocation instance. The detailed guidelines for implementing these objects providing monitor services can be found in [10]. This set of concepts, service transactions, monitor and application structure has considerable practical implications as to how one can most effectively construct ***middle tier*** servers in a network centric service environment.

References

1. G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, ``Advanced Transaction Models in Workflow Contexts" In 12th *ICDE*, New Orleans, Lousiana, Feb. 1996.
2. P. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz, ``Specifying and Enforcing Intertask Dependencies", *VLDB* 1993, pp. 134-145.
3. P. A. Bernstein, M. Hsu, and B. Mann, ``Implementing Recoverable Requests using Queues". In Proc. *ACM SIGMOD*, 1990.
4. A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, K. Ramamritham, ``ASSET: A System for Supporting Extended Transactions", Proc. *ACM SIGMOD* 1994, pp. 44-54.

5. Y. Breitbart, A. Deacon, H. Schek, A. Sheth and G. Weikum, ``Merging Application Centric and Data Centric Approaches to Support Transaction-Oriented Multi-System Workflows", *ACM SIGMOD Record*, 22(3), Sept. 1993.
6. D. Chappell, ``Understanding ActiveX and OLE: A guide for Developers and Managers", *Microsoft Press*, Redmond, Washington, 1996.
7. P. Chrysanthis and K. Ramamritham, ``ACTA: The SAGA continues", pp. 349-397, in [2].
8. Customer Information Control System/ Enterprise Systems Architecture (**CICS/ESA**), IBM, 1991
9. **IBM CICS** Home Page, *http://www.hursley.ibm.com*
10. A. Dan and F. N. Parr, "The Coyote Approach for Network Centric Business Service Applications: Conversational Service Transactions, a Monitor and an Application Style", HPTS workshop, Asilomar, CA, 1997.
11. U. Dayal, M. Hsu, and R. Ladin, ``Organizing Long-Running Activities with Triggers and Transactions" *ACM SIGMOD Record*, pp. 204-210, 1990.
12. "Database Transaction Models for Advanced Applications", Ed. A. K. Elmagarmid, *Morgan-Kaufmann Publishers*, 1992.
13. **FlowMark**, SBOF-8427-00, IBM Corp., 1996.
14. H. Garcia-Molina, and K. Salem, ``SAGAS," Proc. of *ACM SIGMOD* Conf., 1987, pp. 249--259.
15. J. Gray, and A. Reuter ``Transaction Processing: Concepts and Techniques," *Morgan Kaufmann Publishers*, 1993.
16. T. Haerder, and A. Reuter ``Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, 1983, pp. 287--317.
17. **IBM IMS** Homepage, *http*://www.*software.ibm.com/data/ims*
18. **Java Beans** Homepage, *http://splash.javasoft.com/beans*
19. M. Kamath, and K. Ramamritham, `` Modeling, Correctness and Systems Issues in Supporting Advanced Database Applications using Workflow Management Systems" , *Technical Report, TR 95-50*, University of Massachusetts, Amherst, 1995.
20. Object Management Group (**OMG**), *http://www.omg.org*
21. C. Pu, G. Kaiser and N. Hutchinson, ``Split Transactions for Open-Ended Activities" , Proc. *VLDB*, 1988.
22. M. Rusinkiewicz, and A. P. Sheth, ``Specification and Execution of Transactional Workflows" *Modern Database Systems*, 1995, pp. 592-620.
23. F. Schwenkreis, ``APRICOTS - A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System", Proc. of the 12th *Symposium on Reliable Distributed Systems*, pp. 12-21, 1993.
24. H. Waechter, and A. Reuter, ``The ConTract Model", Chapter 7, pp. 219-263, in [12].