The Coyote approach for Network Centric Service Applications:
Conversational Service Transactions, a Monitor and an Application Style [1]

Asit Dan and Francis Parr
IBM T. J. Watson Research Center
Hawthorne, NY 10532

Abstract: 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 cancelable (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 the COYOTE approach: an application development style and a monitor environment for supporting long running network applications and extended transaction models. We also provide a high level design for this monitor and describe briefly how network applications can be developed in this environment.

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 [10, 5, 16] 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 this paper, 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 [16], or may even follow various extended transaction models already proposed in the literature [13, 15, 22, 25, 6, 9, 1].

1.2 Requirements on Infrastructure for Network Centric 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)*:

- *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.

2

- *Coordinated 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 coordinated purchases of items, such as, services of Hotel and Airline. The underlying runtime environment should ensure this all-or-nothing execution behavior.

- *Service contracts:* A service contract between two interacting parties (i.e., CSTs) documents their expected behaviors. 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, we outline the Coyote approach: the conversational service transaction model, the Coyote monitor and the application development model. We further illustrate these concepts with an example. Subsequently, in Section 3, we describe in detail the essential concepts and building blocks, as well as itemize the functions provided by a COYOTE monitor. We provide further details of the compensation services supported by the COYOTE monitor, reliable execution of service requests, and service contract registration and enforcement in Sections 3.5, 3.6, and 3.7, respectively. In Section 4, we provide an overview of how the proposed COYOTE environment and services can be used for developing such service applications, and comments on the conveniences of this environment to the application developer. We then explore the relationships to earlier works: the ConTract model [25, 24], various Workflow/taskflow systems [1, 2, 12, 7, 20, 23] as well as component models [8, 21] for application development. Section 6 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

- 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 service applications and define a simple integrity model for *conversational service transactions*. Each organization which wants to provide a network centric service application (i.e., CST) defines a set of well defined service access interfaces. These interfaces can be invoked by remote and/or other independent CSTs to access services provided by this one (see Figure 2(b)). We further advocate the use of an extended transaction model and a monitor supporting such pairwise interactions. The key concepts are:

1. Clients get service by having a *conversation* with the monitor at the server node,

2. 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 sub-contracted services potentially at other nodes.

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

4. 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.

5. The monitor also provides runtime service interfaces to its local CSTs for coordinating their requests to other CSTs. A group of service requests to other CSTs are executed by the monitor in an *all-or-nothing* manner.

6. 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.

2.1.2 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 [4, 16, 10]. 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 behavior of the applications which it monitors. Definition of standard ACID transactional properties (i.e., atomicity, consistency, isolation and durability) can be found in [17, 16]. 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).
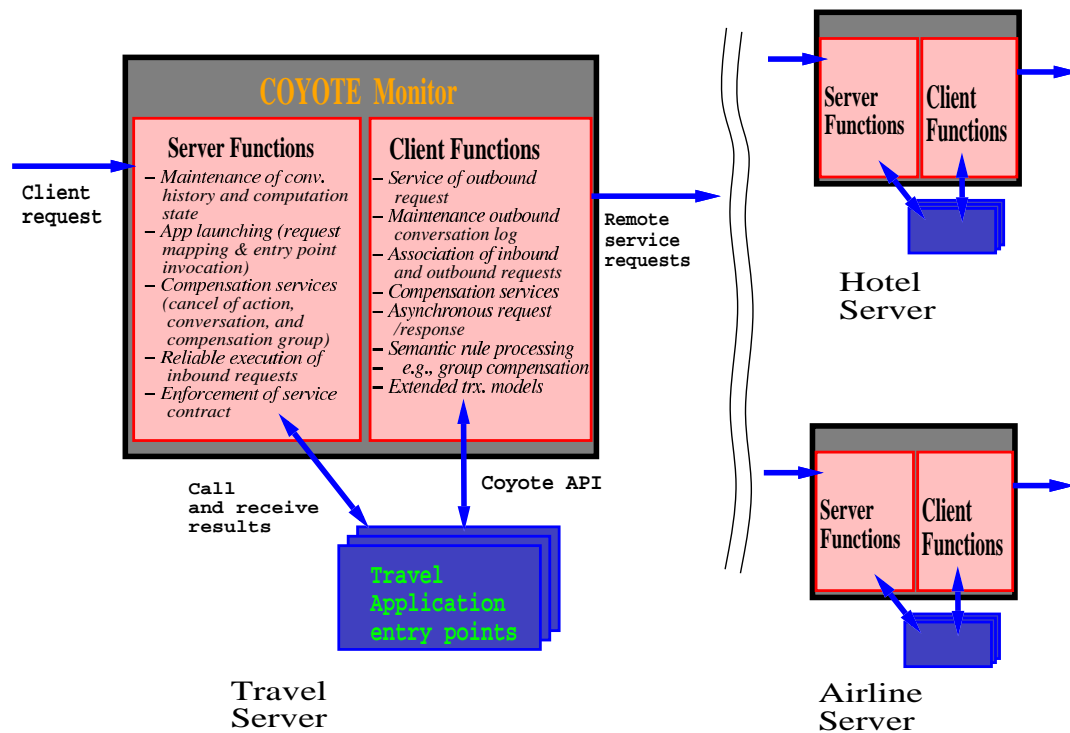

Figure 1: Coyote monitor

A schematic view of a Coyote monitor is shown in Figure 3. At this level the architectures of

5

a classical ACID transaction monitor and a Coyote monitor are essentially the same. However, a Coyote monitor differs from the above summary description of standard transaction monitor functions in several respects.

1. 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.

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

3. 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.

4. Multiple related interactions that follow a logical conversation thread may be dispersed in time, and hence, the monitor maintains persistent conversation states and an interaction history.

5. 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 applicate defined compensation action) if the user so requests.

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

7. The monitor also enforces service contract, i.e., checks to see if the user is allowed to invoke this method.

8. 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.

2.1.3 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 fulfillment, 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, timeouts 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 behavior.

2.2   An example: conference trip reservation

Consider a typical multi-organization long running business application: *Travel planning including conference registration.* [2] Figure 2(a) 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 coordinates 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

---

[2]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 [25, 6, 13].

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.

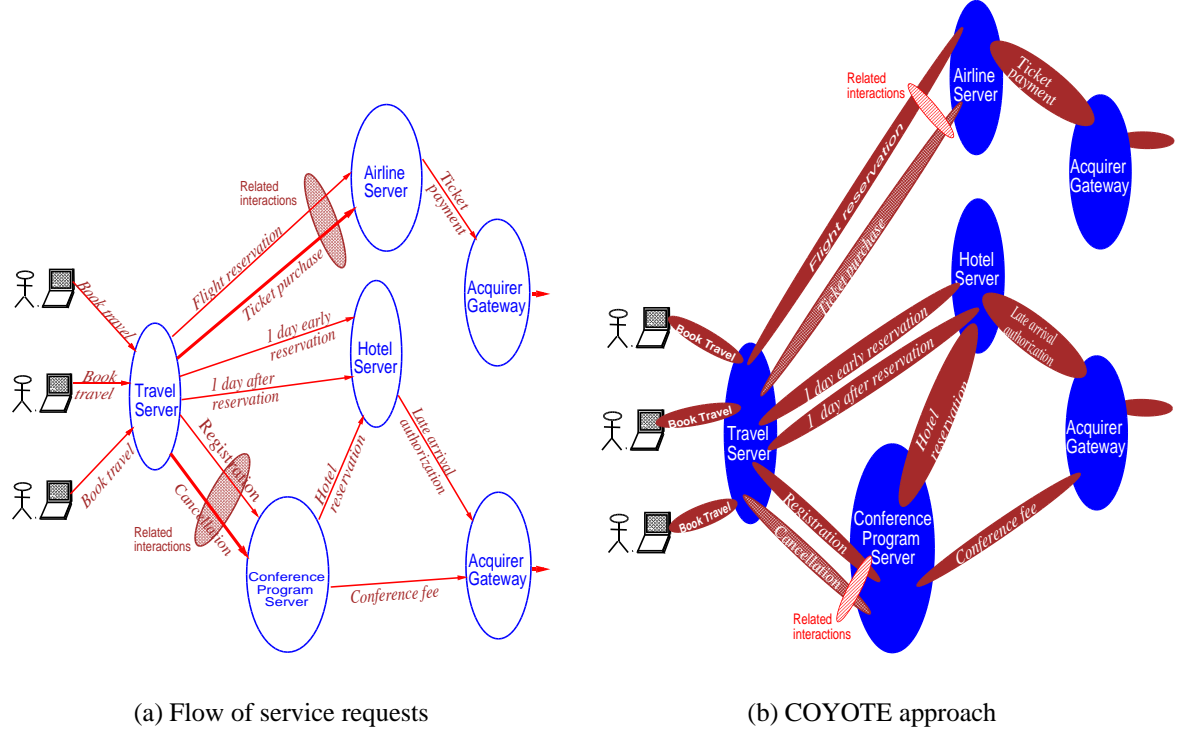(a) Flow of service requests   (b) COYOTE approach

Figure 2: Conference registration example

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.

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.

Figure 2(b) illustrates the use of a COYOTE monitor for providing support for pairwise interactions. The lightly shaded areas (e.g., Registration, Cancellation) represent request interfaces, while dark shaded areas (e.g., Hotel, Airline) indicate server nodes whose participation is typically not visible through the defined service request interfaces. 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. We shall present this proposed extended transactional model (Coyote) by describing a monitor which supports it and provides useful services to server applications constructed with this approach.

In the above example, the travel server coordinates 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 coordinates 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 backend 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.

3   Concepts and Services in the Coyote monitor

This section introduces the essential concepts of the Coyote model more formally to aid in the discussion. The central concept is that of a Coyote monitor. (In subsequent discussions, we will use the term Coyote to represent both the Coyote monitor and Coyote model depending on the context.) This is a system component located at a single node or processing engine. It receives messages, manages transactions and schedules server application processing.

3.1   Users, Conversations and Service Invocations, Action Requests

Users: The Coyote monitor understands the concept of a user on behalf of whom transactions can be executed, i.e., processing can be performed. The user is the individual asking for the transaction. Typically the user is located on some node different from the node where the the Coyote monitor resides and communicates by sending messages to this Coyote monitor. Local users could also exist at the Coyote monitor node but they would be treated by Coyote the same way as a remote user. The

example expected to be typical is that of a person on a remote system running a Web browser and sending HTTP requests to the Coyote monitor. A workstation based user executing an application at a remote site is the next leading example.

The Coyote monitor will keep a persistent record of the users which it knows and has executed work for as part of its directory and log. To identify a user it could eventually keep some Universal User Id generated by a certification agency but until standards for this become widely used it will probably keep a table of passwords for users established by this Coyote monitor.

The Coyote monitor will also keep a list of user ids which its local applications use to request services of other servers. These ids are defined as part of the application registration.

In a typical interaction, an end-user will browse through web content located at a particular Coyote monitor, looking for services of interest in an unstructured way. During this period, the user will typically not have identified himself by specifying a userid or password; Hence the interaction is anonymous at this point. When the user selects particular services and starts the process of requesting something, or attempts to discover what has happened to requests made by him/her at a previous time, identification via some sign-on process will be required.

Conversations: An identified user gets transactional services by starting a conversation with a Coyote monitor and flowing service requests on that conversation. The conversation is a grouping of service requests which is convenient and meaningful to the user. For example a user may choose to make all requests associated with a single travel trip a single conversation so that they are grouped together, can possibly be cancelled as a unit, and are separated from purchases of other items.

Each conversation has a single defined user who initiated it. Future work may address how these concepts can be extended to multi-party conversations.

Service invocations: Service invocations are the functional requests which flow on a conversation; examples would include making a specific hotel reservation or issuing tickets for a specified itinerary. To provide any useful business service usually requires more than one interaction with the end user, hence a service invocation is made up of multiple actions. An application available at a Coyote monitor, is defined as a named service with a collection of action methods.

A request identifies a service or transaction name. When the request arrives at the Coyote transaction manager, it causes processing of an instance of that named application to be scheduled. Various related actions (e.g., cancellation, modification) may be performed subsequently for an instance of a service request.

Action requests: Action requests are the atomic flows, represented by a single message from the user/client into the Coyote transaction monitor. These are specific requests to either start the service invocation, to cancel a previously started service invocation, to cancel or compensate for an active service invocation or a modify action to change what was previously done as part of that service. As an example consider the following sequence of action requests: (1) the initial action to initially make a hotel reservation (2) the modify action to confirm it with credit card number (3) the cancellation to clear it, as making up a service request associated with the application: *making a hotel reservation*.

3.2 Coyote Service Registration

The COYOTE monitor provides support for issuing, cancelling, reissuing and modifying service invocations, where cancellation implies taking some compensation action defined by the service. To provide this support, all services available at a Coyote server and their associated action methods must be **registered** with the monitor. Registration always involves defining:

- the name of the service ,

- each of the action functions: i.e. the primary issuing action for initially making the request, the compensate action (if it exists) and the modify action (if it exists),

- the signatures (prototypes) of each of the action functions, and

- the execution method for each of the action functions.

(We shall see in Section 3.7 some additional information also defined at registration.)

To explain what we mean by execution method we distinguish between registering **inbound** and **outbound** service requests to a Coyote monitor.

For every registered inbound service, the Coyote monitor is prepared to receive incoming action requests requesting that service. When this happens the monitor starts an application program (the indicated action) executing on a thread under its control to provide this part of the service. The execution method for an internal action is essentially the program entry point to be called on an appropriate Coyote managed thread when a request for that action is received.

The outbound services are the services which can be called by applications executing under the Coyote monitor to get at services provided by other service applications. These applications may be independent CSTs running at remote nodes or even in the local node under the control of a Coyote monitor. For actions which are part of outbound services, the registered execution method and the parameters of the IPC, RPC or HTTP requests are used to pass the request for action to the appropriate execution environment.

The function of the Coyote monitor is therefore to use its table of registered inbound services and their actions to schedule application processing when requests are received. It also intercepts all the outgoing action requests made by started applications and forward those requests, following information in its table of registered outbound services and actions.

The fact that a Coyote monitor treats inbound and outbound registrations separately providing different processing and services is illustrated diagrammatically in Figure 1.

3.3 Coyote Conversation Management and Request Processing

Conversation management: A user interaction with a Coyote monitor may optionally begin with an anonymous unstructured web browsing phase, but the proper conversation is established after the

11

user is reliably identified through some logon or password protocol. If the user is identified by some certified UUID in a public directory, then this can be used as identification, otherwise the Coyote monitor will use its own table of defined users and validate access by a password exchange. Once the user is identified and hence a record of previous persistent conversations of that user with this Coyote server is available, the user is given the choice of starting a new conversation or resuming a previously defined conversation. The Coyote monitor will in response either generate a new unique persistent conversation id or retrieve the identifier of the resumed conversation from its persistent log records.

The conversation id established in this way will then be passed in some form on all subsequent requests from this user. The monitor will maintain data structures so that conversation state and owning user identification can be efficiently recovered on receipt of this conversation id with each inbound request. Techniques for passing the conversation id on the request flow will vary depending on the implementation context for the Coyote monitor. If the requests into the monitor from clients are basic HTTP in a Web server environment, embedding hidden variables are one way of accomplishing this. If a Java shell applet is downloaded dynamically to client browsers starting a Coyote conversation, the conversation id may be passed as a formatted field with each request to the server monitor. For Object Oriented servers, the conversation itself could be an object and all subsequent service requests on this conversation rendered as method calls to this remote object using a protocol such as IIOP; in this case the conversation id is effectively being passed as the object reference on which methods are being called.

The conversation management interface to the monitor includes:

- start a new conversation

- close a specific conversation.

The close conversation command is issued when a conversation is completed and will not be used further for any action requests. Log records for the conversation will be kept for audit purposes but may be rolled out to some lower cost archival storage medium.

Processing inbound "new" action requests: Individual service invocations can now be made on the established conversation. A client will typically first issues an action for a "new" service invocation. These messages may not have a unique service invocation number established by the client; Hence when received, the server side Coyote monitor must do some processing to determine whether a received message is a valid new service invocation or whether it is a duplicate. This issue is taken up in subsection 3.6. The new action request starts the service invocation; multiple additional action requests can follow to modify or cancel it.

Each action request message includes fields for:

- the client supplied Service Invocation Number (SIN)

- the server supplied unique SIN

Client side unique numbering of requests is optional. For "new" action requests, server supplied SIN is always null on the incoming request (since it has not yet been allocated by the monitor). If the client does not uniquely number its service requests, (i.e., client SIN is null or omitted), the Coyote monitor will generate a unique SIN provided this does not appear to be a duplicate request. Specifically it will check that previous requests on this conversation for the same service function with the same request parameters for which no follow up modify or cancel has been received indicating that the client has been notified of the server SIN. All the information on requests needed to perform this checking is saved on the Coyote log as described in the following subsection. If the client SIN is supplied, this will be used to determine whether the "new" action is a duplicate. The processing of duplicate "new" actions is described with state diagrams in the section on Reliable Execution in Section 3.6. Having established that a valid (non-duplicated) new action request has been received, the Coyote monitor will generate a unique-within-conversation SIN for it and save it in the log. Parameters for the action request and if present the client's SIN will also be logged. The function identified in the request for action will then be looked up in the monitor's table of registered inbound service request functions. Assuming that a valid function has been requested, the monitor then uses the registered "new" program for this service request function, allocates an execution thread and starts the identified program on that thread. The server SIN, conversation id, user identification and action parameters are all made available to the launched application program instance.

The application program, launched to satisfy the incoming request, may generate during its execution multiple outbound action requests and associated outbound conversations to other services at this node or to other nodes. We will take up the monitor's processing of these outbound requests a little later in this subsection.

Eventually the launched service request program will complete. It then returns to the monitor with the output message to be returned to the requesting client. The monitor will free up the thread previously used by the application program, log any results, write the allocated server SIN into the reply message and send this message to the requesting client.

Processing inbound cancel and modify action requests: The processing of these requests differs from the processing of new action requests primarily in the method for associating a SIN with the received request. If client request numbering is used this association is used to determine the service request to associate with this action. If client request numbering is not used, then the server SIN must be supplied in cancel and modify actions and this will be used. The monitor then uses the appropriate "cancel" or "modify" program name in its table of registered inbound service request functions, and launches that application program on an allocated thread. In this case the application receives access not only to the user, conversation, SIN and action parameters, but also to the original parameters of the "new" service request which started this SIN. These parameters were logged by the monitor and are retrieved from the log and made available to the application to facilitate the modify or compensate processing.

Processing outbound action requests from a monitor started application: The Coyote monitor will intercept and handle outbound action requests from applications running under its control.

13

This may include requests to subsystems at this node or requests to other server nodes which may or may not be running under a Coyote monitor.

In order to request service from another server, the Coyote started application will establish an outbound conversation. This is similar to the inbound conversations used to pass service requests into Coyote. The application can start an outbound conversation using the Coyote interface OPEN_CONVERSATION(). The Coyote monitor starts a new logging thread as part of this conversation, and returns an unique *Coyote conversation number* to the application for later identification of this conversation.

The monitor will assign unique client SINs to all its outbound service request invocations and will log these SINs and the outbound action parameter lists in the Coyote log. It will also log reply parameters and any remote server SINs when it receives responses for these outbound requests. The monitor will associate these outbound requests with the conversation (on the input side) which launched the application making these requests. When the outbound requests are going to some resource manager or database not executing under a Coyote monitor (the most general and usual case unless Coyote monitors became widespread) there will be no notion of a Coyote conversation outbound, the servers handling these requests will have their own notion if any of conversation. Note that the outbound requests flowing from a CST to other CSTs will use the user id registered with the local CST. However, in some cases, the local CST may also use the user id associated with the inbound requests.

One thing which may also occur is that an application program executing under the monitor requests a service provided at the same monitor. This is processed as an outbound request from the invoking CST and inbound request to the CST to be invoked. The resulting processing is treated as part of the requesting conversation. A performance optimized monitor will typically execute a synchronous internal request on the thread of the caller.

3.4   The Coyote Log

As already alluded to in preceding subsections, the Coyote monitor will build and maintain a persistent log which records all incoming action requests, their associated client and server SINs and their input and reply parameter lists. For outbound service requests, the monitor stores similar information except that server SIN numbering cannot be relied upon in this case (just as client SINs were not required on the input side).

This information is maintained in a hierarchy:

```
user
     conversation
                 inbound SIN + specific actions
                         outbound actions.
```

Logging services are provided for the applications to retrieve this information, i.e., the log is presumed to flow over time into some relational database with indexes based on the hierarchy

above. However relevant action and parameter information is automatically made available to action programs when they are launched; The intent is that general searching is not part of normal "automated" request processing.

The same logging services used to build this system generated log are also available to Coyote application programs to build (as a separate log stream) their own application level audit log. Note that this is a very different intent (with rather different performance and structure requirements) from the recovery logs built by database and ACID transaction monitors to ensure transaction atomicity.

3.5   COYOTE Services for Compensation

In this section, we describe various COYOTE services used by an application for compensating a single or a group of actions.

We have commented above that ACID recoverable transactions are unlikely to be useful in recovery of a service invocation whose implementation is distributed across multiple organizations with no motivation to keep their database consistent. In this context cancellation or compensation is likely to be required (in many cases with multiple short conventional transactions embedded within the overall flow).

Coyote provides several levels of compensation support.

Compensation of action: The application may cancel an already granted service invocation, any time after it is started via an explicit request. The application passes the service invocation number to the Coyote monitor identifying this as a cancellation action. The Coyote retrieves all the parameters associated with the original request as well as the returned results from its log. The application may also be allowed to specify additional parameter values at cancellation time. If no such prior service invocation is found in the log or the service is already cancelled, or the cancellation request is too late, or an action request is still pending against that service invocation, an appropriate error code is returned to the application. In implementing the cancellation request, the Coyote monitor uses the fact that a specific *cancel* action was defined when the service interface was registered. If this were not the case, i.e. a service was registered with no defined cancel action, that would also be reported to the requesting application as an error

Modification of action: The application may also modify an active service invocation adding a new set of parameters. The Coyote monitor, as above, checks to see if the modify request should be honored, i.e., if it is allowed by the service interface. If so, the Coyote retrieves input parameters and returned results associated with the original service invocation. Any additional parameters passed by the client application on the modify action are also passed.

The processing of a modify is very similar to the processing of a cancel action . The differences are that (i) modify does not end the service invocation so that other actions in this service may follow (ii) modify lacks the implicit nesting semantics of cancel, i.e., unlike cancel, modify of a conversation is not defined, and hence, modify is not generated automatically as part of larger cancellation operations.

15

A typical example of a modify action in our example context, would be changing the parameters of a booking, either a date or an additional requirement such as a non-smoking room.

Compensation of conversation: As noted above service invocations are always part of a conversation. One conversation can include several service invocations. In particular in our illustrative example, the conference booking travel application would be implemented with a service request invocation to a hotel to get the room, and a separate service request invocation to the airlines to get the flight booking.

Either a client , or a Coyote application can request compensation of an entire conversation. This has simple nesting semantics. Compensation is called by the Coyote monitor for each of the service invocations in the conversation.

This corresponds to the user cancelling the entire trip. There may or may not be full refunds. Coyote will automatically call the appropriate compensation function in each of the logged service invocations for this conversation.

Compensation group: Sometimes it is desirable to group a set of service invocations, so that if any one fails the entire group is automatically compensated by the Coyote monitor. A Coyote application can define a compensation group using the `OpenCompGroup()` call to the Coyote monitor. The Coyote conversation number is passed as the input parameter, and the Coyote returns the *conversation group number*. Both the conversation number and the conversation group number are passed as parameters for service requests. For requests that are not part of a conversation group a null value is passed as conversation group number. The conversation group is aborted if any service request fails or if the application issues an explicit `CancelCompGroup()` call to the Coyote. A conversation group is closed by a `CloseCompGroup()` call.

Query of state: If less structured compensation and recovery logic is required in a particular application it can build explicit logic around the information available form the Coyote log. This information was described in Section 3.4.

3.6 COYOTE Services for Reliable Execution

An important characteristic of the network centric environment is that the network is inherently unreliable, and servers may be managed to widely varying responsiveness requirements. A request for remote service has an unpredictable response time, or may get no response at all. In order to be able to conduct useful business in this environment, the Coyote monitor provides services for reliable execution of remote requests. On the receiving side the Coyote monitor has to deal with a stream of incoming messages (service invocations and actions) which may have missing or duplicated members.

Human users of the internet have frequently to face the question: "Is there a remote server somewhere still working on my last request - or has it failed?". Application programs implementing network centric services will frequently be faced with this same choice. An important part of their transactional environment is having a consistent way to deal with it. The concepts of a conversational service transaction makes it possible to treat this problem because:

- a request for service is uniquely identified

- actions such as modifying or cancelling a service are architecturally defined.

A Coyote application calling out for services uniquely numbers service invocations. All action messages associated with that service invocation carry the service invocation number. Hence it is very easy for any remote server, Coyote or otherwise, to detect duplicate service invocations, and associate cancellations and modify actions with the proper service invocation. If there is no response to the request:
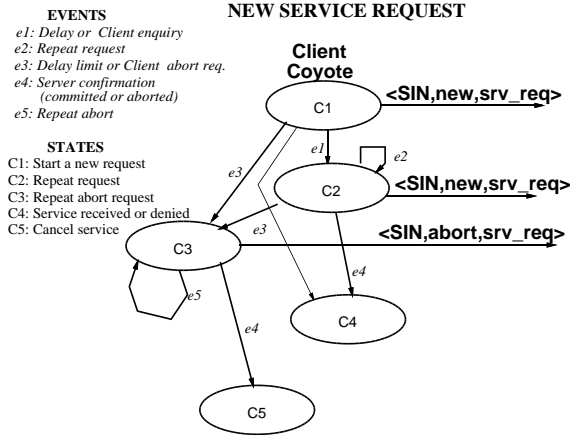
```
SIN# 1234 book hotel room    name: nnn
      hotel: hhh  dates: ddd      ...
```

within some expected timeout, then the resend action can still carry the same service invocation request number SIN#1234 and hence there is no danger of ending up with two reservations.
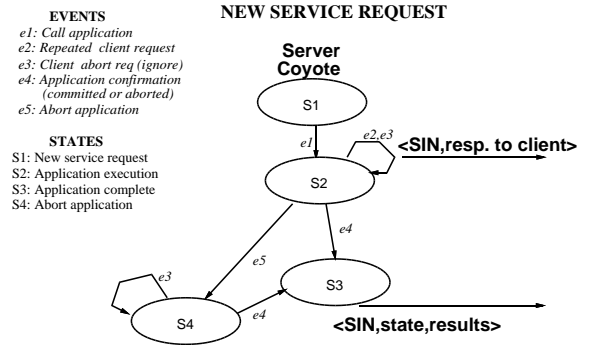
When a Coyote monitor is communicating with a client - perhaps a simple browser - which does not uniquely number service invocations, then some heuristics are required to determine whether a request from the client is a "reissue" or whether it is a genuine independent service invocation. In the example context, if the server receives two reservations for Smith for the same night from the same client, should it assume that the previous reply message to the client got lost,(or the client got frustrated) or should it go ahead and make the second reservation. In our conversational service transaction model, whatever decision may be taken by the server, it will be recorded in the conversation log. This has the advantage that the client will be unambiguously informed using the conversation log whether or not the second message was treated as a separate service invocation (and given its own service invocation number). Furthermore this distinction is made in a generic way by Coyote, rather than as return information from the service application.

The preferred and more functional case is that the client does uniquely number its service invocations. Any client which is an application executing on another Coyote monitor will of course provide this. In this case, the Server Coyote will be able to do full and unambiguous checking for reliable execution. A simple example of the logical states in checking for repeat requests and cancellations between two Coyote monitors is illustrated with finite state machine diagrams in Figure 3.

This unique numbering of service invocations, and organization of the action methods within a service interface, also allows for sequencing rules on the actions within any service invocation. Specifically, as part of the service contract, the service will define the allowable sequences of action requests within any service invocation. For example, a service could allow at least one request for action A, followed by exactly one request for action B, followed by zero or more invocations of action C or D. In addition, the cancel can be issued at any point. These sequencing rules are formally defined as part of the service interface and checked on incoming action requests by the Coyote monitor. This is further explained in Section 3.7 where we discuss service contracts.

17

EVENTS
e1: Delay or Client enquiry
e2: Repeat request
e3: Delay limit or Client abort req.
e4: Server confirmation
   (committed or aborted)
e5: Repeat abort

Client Coyote

STATES
C1: Start a new request
C2: Repeat request
C3: Repeat abort request
C4: Service received or denied
C5: Cancel service

C1   <SIN,new,srv_req>
C2   <SIN,new,srv_req>
C3   <SIN,abort,srv_req>
C4
C5

EVENTS
e1: Call application
e2: Repeated client request
e3: Client abort req (ignore)
e4: Application confirmation
   (committed or aborted)
e5: Abort application

NEW SERVICE REQUEST

Server Coyote

STATES
S1: New service request
S2: Application execution
S3: Application complete
S4: Abort application

S1
S2   <SIN,resp. to client>
S3
S4   <SIN,state,results>

(a) Client Coyote states in reliable execution of a new request

(b) Server Coyote states in reliable execution of a new request

Figure 3: Execution states of a service request in clients and servers

3.7   Service Contract

When a service either inbound or outbound is registered to the Coyote monitor its semantics as seen by Coyote are registered in the form of a *service contract*. This encapsulates the following information:

- the name of the service

- the names and signatures of each of the action methods, and the cancel method

- the sequencing rules on the actions (see Section 3.6)

- authorizations on each action

- the responsiveness committed

   This effectively defines a contract between requester and server. Note that different action sets within a service could be made available to different classes of user by using access control lists on each action method. The sequencing rules are used to manage possible loss or duplication of requests in flowing across the network. This sequencing would be checked and managed at the receiving Coyote server. Responsiveness is intended to give the client application some guidance as to how set useful timeouts. An average and a high percentile response time could be used to convey this.

4   COYOTE Application Development

In Section 2.1.3 we gave an overview of the application development process indicating that the developer had to provide (1) service interfaces, (2) action definitions and (3) scheduling rules based on events.

The preceding section has described in more detail both the service contract defining the interface and the role of actions within a service invocation. In this section we will focus on the events and scheduling rules used to trigger valid sequences of actions to provide a complete service invocation

Events and scheduling rules play an important role in the design of network centric service applications, because the network environment drives the application designer to use parallel invocations of other network services and asynchronous messaging. Thus determining the "next step" in providing service may require determining which sets of responses, timeouts and additional requests from the client have been received. Trying to do this with conventional application processing logic is complex and potentially inefficient. A better solution is to allow the application to declaratively define events and triggering actions.

We use a travel reservation example to illustrate this point further.

4.1   Travel Booking

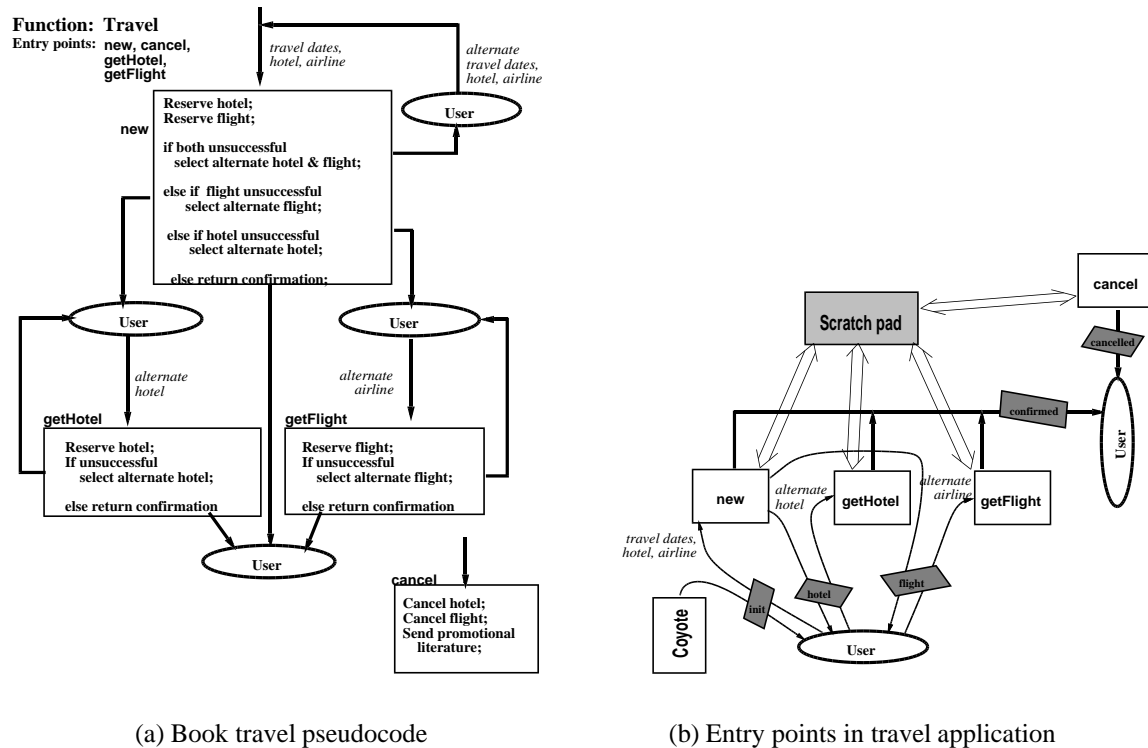| (a) Book travel pseudocode | (b) Entry points in travel application |

Figure 4: Application development

Figure 4(a) shows the logic of a travel booking application using sample pseudo-codes. The application logic is similar to the logic executed in the Travel Server of the conference registration example (see Figure 2). Upon receiving an user request along with the input parameters, the application first requests in parallel the Hotel and Airline (and also the Conference) servers for their services.[3] The customer later may cancel the BookTravel request (i.e., the entire set of travel arrangements on behalf of this customer), in response to which, the Coyote monitor in the Travel Server node finds all successful entries in its log and cancels corresponding requests. The log is maintained by the Coyote monitor and entries are created by intercepting all incoming and outgoing service requests. The Coyote monitor also facilitates reliable execution of all requests (see Section 3.6). The application code is executed upon receiving incoming service requests.

The above service requests in the BookTravel application can also be executed under various extended transaction semantics and compensation model. For example, the entire application may be viewed as an atomic operation. Under this semantics, if the execution of a service request to any of the other server fails, the application will be aborted and all the previous service requests under this application will be automatically compensated by the Coyote monitor. Alternatively, a subset of the requests (say, to the Hotel server and the corresponding car reservation request to the Rental server) may be combined as a compensation group. The Coyote monitor provides easy constructs for expressing this in the application, and leaves the details of its execution to the Coyote monitor.

An application developer first creates the application logic flow as shown in Figure 4(a). Each interaction with an user represents a break in the logic flow. This results in grouping of application steps into many small blocks, and a separate code segment can be developed for each such block (as supported by traditional TP monitors). The steps within a block can be expressed as transaction flow using the ConTract [25] model. However, the flow across blocks are guided by user interactions. Note however, the code segments for the blocks may share common variables that need to be persistent. The Coyote monitor stores the values of these variables in a conversational scratch pad which is hardened to a log upon exiting a block. These values are reinstantiated from the log and made available in the scratch pad when another related block for that application is invoked. Finally, the interaction with an user or a remote service node may require additional logic/processing. For example, an HTML form needs to be generated or an HTTP POST request processed for interactions with a human user. A different messaging format (e.g., EDI) may be used for interaction with an automated service node. The Coyote environment provides utilities and services for writing such applications. The complete application is defined to the Coyote environment by registering a set of ENTRY points and their associated application code segments and FORMs used by the application.

In this example an action triggering rule might be that:

```
 after requests have been sent to airline and hotel services:

if both responses are received:
        => schedule action 1 (*advances to next step*)
```

---

[3]For ease of illustration, the request for conference service is omitted from the flow chart. Also in general such applications could be more complex, and may have many additional interactions with the user. For example, if the hotel is away from the conference location, the client may be asked if a car need to be rented from the Car Rental Server.

```
if one but not both responses received
and 90 secs have elapsed
        => schedule action 2 (*reports partial progress,
                                options for user*)
on late response (*user is told of partial progress*)
        => schedule action 3 (*saves response persistently
                                for next user interaction*)
```

Having these event and triggering rules of this sort stated explicitly simplifies the definition of the conversational service transaction and enables the monitor to schedule processing of it efficiently. The application may also use the rules based on a success or failures of a compensation group to trigger subsequent actions.

In summary, the Coyote monitor provides several important services to make writing business applications easier.

1. First, the Coyote monitor (as any transaction monitor) frees an application writer from the details of process management, application launching, state maintenance via log entry creation and management.

2. Second, it provides various extended transaction semantics and compensation constructs that are very useful in expressing application logic.

3. Third, it provides support for reliable execution of service requests that are useful in building distributed applications.

4. Next, Coyote approach provides an uniform model to write independent applications which are integrated via the monitor.

5. The Coyote provides services for form/message generation and processing for interactions with users or remote service nodes.

6. Finally, an important point to note here is that in the absence of Coyote services, individual applications need to duplicate these codes. Generally speaking, the code for these services is complex, but needs to be reliable. Hence, factoring out these code modules as Coyote services is important.

5   Comparison with Existing Work

5.1   Comparison with Traditional ACID Transaction Model

Many authors have identified shortcomings of the classical transaction model [13, 15, 22, 25, 9, 1]. These works address the issues of task dependency, semantics and correctness of execution. Here, we just recall the defining (ACID) properties of the classical transaction model [17, 16] to aid subsequent discussions. The key properties required of a transaction are: (a) *Atomicity* of the entire

set of operations performed by the transaction, (b) *Consistency* or data integrity across various data elements for preserving certain relationships (e.g., sum of checking and savings accounts of an user is constant after a transfer of money from one account to the other), (c) *Isolation* of operations and (d) *Durability* of operations. These properties made it feasible to maintain global integrity and consistency of data in a set of databases and helped in understanding the execution of transactions as a serializable stream of operations.

The above ACID properties have less relevance in the new Internet environment where end users are initiating potentially global business interactions, each one of which may span multiple independent enterprises. A participating organization cares much more about what it has committed to deliver and is legally obliged to do, rather than whether its database is consistent with the databases in partner organizations. For example, when a credit card company authorizes a charge amount against a particular card, it does not do so conditionally on whether some requesting transaction aborts; the authorization will just have some expiry date and will be valid if some confirmation request is received before then. Whereas the ACID transaction model emphasizes the ability of the transaction system and the resource managers to undo all effects on persistent data of a transaction which aborts, practical multi-party business interactions are more concerned with (1) have the business actions been durably recorded? (2) what application defined compensation actions are available if cancellation is desired? (3) what automatic expiry periods are required by the business and legal agreements between pairs of parties?

Note however, that the proposed conversational service transaction model for addressing the needs of the business service applications does not entirely replace the need for classical ACID transactions. It complements ACID and other advanced transaction models [13], and extends existing transaction processing environments for dealing with service independence and long running conversations. Within a single CST or conversational service transaction where data consistency issues are well understood and data integrity needs to be preserved (e.g., while executing individual service requests), ACID and other advanced transaction models will still be used. As has been suggested by previous works on extended transactions, there may be extended business interactions (i.e., non-ACID transactions) where the subportions may include ACID transactions. The earlier proposed various advanced transaction models seek flexibility in data dependency as well as improved concurrency [13, 9]. The Coyote monitor provides these services wherever needed. However, the Coyote monitor provides many additional services as detailed in earlier sections of this paper.

5.2 Conversational Service Transactions and Workflows

The advanced transaction [13, 9, 15, 22, 6] workflow [14, 20] and taskflow models [25, 24, 1, 2, 7, 12, 23] share many similarities in dealing with certain application abstractions, i.e., an application is composed of a sequence of (possibly) ACID steps. However, their design points are significantly different and their primary focus on application development issues and/or runtime environments are in general complementary. We first briefly summarize the design points for these systems and then contrast the Coyote monitor to these systems.

Objective: The design point for workflow systems emphasizes (1) easy modeling of business processes by composing pre-developed tasks, (2) role management (i.e., allocation of work to pools of human operators), and (3) maintenance of the workflow pattern in a general database structure. The implications are that the execution of these tasks can not monitored at a detailed level by the workflow monitor. In addition, the system is based on interpretation and the performance is traded for ease in expression. In contrast, the focus of advanced transaction models is on defining semantic language constructs and runtime services that improve fine-granularity concurrency and flexibility in data dependency expressions (e.g., nested transactions).

Recent works on transactional workflow systems [1, 2, 7, 12, 23] attempt to bridge this gap by creating semantic constructs at the script level for describing data or execution dependency. The monitor maintains execution state in terms of the tasks already executed and their results. Upon a system or application failure, the system can do both forward or backward recovery. Backward recovery is based on invoking a predefined set of compensating tasks.

Control flow: Typically, the execution flow of a workflow application is described in terms of the order in which the component tasks are executed. In contrast, general programming constructs capture more complex data and/or state dependency, particularly in the presence of asynchronous execution constructs (i.e., rules and events). Here, the number of execution states can be very large, and applications require appropriate interfaces for dynamic interactions with the monitor.

The steps or tasks can be related to application components in systems based on component architectures (e.g., COM, CORBA, JAVA Beans). However, the task invocation processes are different between these two classes of systems. In workflow systems, the steps or tasks are invoked based on predefined scripts, whereas the components are explicitly invoked by the embedded program logic in the later systems.

Step Granularity: The granularities of program steps (that are composed to create an application) in these systems may also differ significantly. In workflow or taskflow systems, the steps are of coarse granularities. Hence, the monitor (i.e., interpreter) does not coordinate the underlying resource management. In contrast, a TP monitor is part of the underlying runtime environment and coordinates resource accesses of an application via the X/Open protocol. In a taskflow or transactional workflow system, the

The design point of Coyote differs from that of workflow systems in that it emphasizes sub-second response time for each individual action requests. This is needed to be responsive to the end-user as well as to provide high throughput automated services. The workflow systems in contrast in general do not need this responsiveness. This is because they schedule more coarse-grained tasks either to human or to for automated processing. Additionally, in COYOTE we seek additional supports for dynamic non-predefined conversations between independent organizations. The Coyote also provides support for form (e.g., HTML) management and processing for ease in development of conversational applications.

5.3 Coyote, Existing TP Monitors and Component Architectures

Existing transaction systems Tuxedo [3], CICS [11], IMS [18] etc, have begun to offer internet gateways to them and support for network centric languages such as Java. Java itself is offering a transaction standard JTS which is actually a veneer on OMG's OTS transaction protocol. The proposed concept of network service transactions begins to provide a model and captures the essential semantics of service applications that these products expect to support. The Coyote monitor also provides some specific services to help in the mapping of input and output data to HTTP parameter lists and HTML pages.

The service transaction model and Coyote approach to application development also complements the application development based on various component architecture models, e.g., COM [8], CORBA [21] and JAVA Beans [19]. All of these architectures provide support and services for developing application components (e.g., Encapsulation, component registration and invocation, event services, transactional services, etc.). However, they all lack support for long running conversations (i.e., general support infrastructure, management of conversation instance, compensation of service request, group compensation, service contract enforcement, etc.). Hence, all of these platforms would benefit from the service transaction abstraction.

6   Summary and Conclusions

Service Transactions are an effective model to capture the semantics for adding service application in a network centric environment.

With the Coyote monitor 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

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.

Acknowledgments: The authors gratefully acknowledge the contributions of Ambuj Goyal and Tim Holloway who through many discussions helped in defining and refining some of these concepts. The discussion of application development builds on the concepts developed in IBM's MQSeries Three Tier product and discussions with Ian McCallion, Peter Lambros, Ian Robinson and Vernon Green. Dinkar Sitaram worked with us at an earlier stage, and contributed to an earlier prototype. He is also an author of a previous version of this paper.

References

[1] Alonso, G., 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] Attie, P., M. P. Singh, A. Sheth, M. Rusinkiewicz, "Specifying and Enforcing Intertask Dependencies", VLDB 1993, pp. 134-145.

[3] BEA Systems Homepage, *http://www.beasys.com*

[4] Bernstein, P. A., "Transaction Processing MONITORS", *CACM*, 33(11), Nov. 1990.

[5] Bernstein, P. A., M. Hsu, and B. Mann, "Implementing Recoverable Requests using Queues". In Proc. *ACM SIGMOD*, 1990.

[6] Biliris, A., S. Dar, N. Gehani, H. V. Jagadish, K. Ramamritham, "ASSET: A System for Supporting Extended Transactions", SIGMOD 1994, pp. 44-54.

[7] Breitbart, Y, 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.

[8] Chappell, D., "Understanding ActiveX and OLE: A guide for Developers and Managers", *Microsoft Press*, Redmond, Washington, 1996.

[9] Chrysanthis, P., and K. Ramamritham, "ACTA: The SAGA continues", pp. 349-397, in [2].

[10] *Customer Information Control System/ Enterprise Systems Architecture (CICS/ESA)*, IBM, 1991.

[11] IBM CICS Home Page, *http://www.hursley.ibm.com*

[12] Dayal, U., M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions" *ACM SIGMOD Record*, pp. 204-210, 1990.

[13] "Database Transaction Models for Advanced Applications", Ed. A. K. Elmagarmid, Morgan-Kaufmann Publishers, 1992.

[14] *FlowMark*, SBOF-8427-00, IBM Corp., 1996.

[15] Garcia-Molina, H., and K. Salem, "SAGAS," *In Proc. of SIGMOD Conf.*, ACM, 1987, pp. 249–259.

[16] Gray, J., and A. Reuter "Transaction Processing: Concepts and Techniques," *Morgan Kaufmann Publishers*, 1993.

[17] Haerder, T., and A. Reuter "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, 1983, pp. 287–317.

[18] IBM IMS Homepage *http://www.software.ibm.com/data/ims*

[19] Java Beans Homepage, *http://splash.javasoft.com/beans*

[20] Kamath, M. 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.

[21] Object Management Group (OMG), *http://www.omg.org*

[22] Pu, C., G. Kaiser and N. Hutchinson, "Split Transactions for Open-Ended Activities" Proc. *VLDB*, 1988.

[23] Rusinkiewicz, M. and A. P. Sheth, "Specification and Execution of Transactional Workflows" *Modern Database Systems*, 1995, pp. 592-620.

[24] Schwenkreis, F., "APRICOTS - A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System", Proc. of the 12th /em Symposium on Reliable Distributed Systems pp. 12-21, 1993.

[25] Waechter, H., and A. Reuter, "The ConTract Model", Chapter 7, pp. 219-263, in [13].