A Monitor for Extended Transactions on the Internet
COver YOurself Transaction Environment (COYOTE)

Asit Dan, Francis Parr and Dinkar Sitaram
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. 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 COYOTE: a monitor environment for supporting long running network applications and extended transaction models addressing the above requirements. We also provide a high level design for this monitor and describe briefly how network applications can be developed in this environment.

1 Introduction

The Internet and related technology trends (e.g., in end-user presentation interfaces and networking) have revolutionized the business transaction processing environment. The web browser is now ubiquitous and available as free software on almost all workstations. Novel technologies such as automated voice processing provide a new rich set of human interfaces. Organization-wide intranets have greatly enhanced the networking capabilities for most businesses. As a result, these technology trends have made feasible *publicly* available low cost automated processing on behalf of businesses. The effects are already visible in the way businesses, and specifically business Information Technology(IT) systems, operate. Whereas IT systems used to be designed to allow the trained employees of a single enterprise or organization to automate operations within that enterprise, in an Internet context they may equally be expected to:

- advertise to the general public services provided by the enterprise,
- allow convenient direct automated purchase of those services to a general consumer who has received no specialized training and may purchase only occasionally from this user,
- facilitate automation of a business interaction involving multiple independent autonomous enterprises or organizations, and
- allow dynamic construction of transactions involving organizational partners who may not have interacted before.

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 [7, 5, 12] for supporting new types of applications, and how network applications can be developed in this new environment.

This paper makes the case for a new style of transaction monitor for supporting long running conversations (i.e., applications). Such applications may be based on various extended transaction models already proposed in the literature [2, 10, 15, 18, 8, 1]. We call the proposed style of transaction monitor COYOTE: for COver YOurself Transaction Environment, to emphasize that when several organizations participate in a business interaction across the Internet, 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.

1.1 Shortcomings of the Traditional ACID Transaction Model for this Environment

Many authors have identified shortcomings of the classical transaction model [2, 10, 15, 18, 8, 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 [13, 12] 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 business interactions, each one of which may span multiple independent enterprises. The 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.

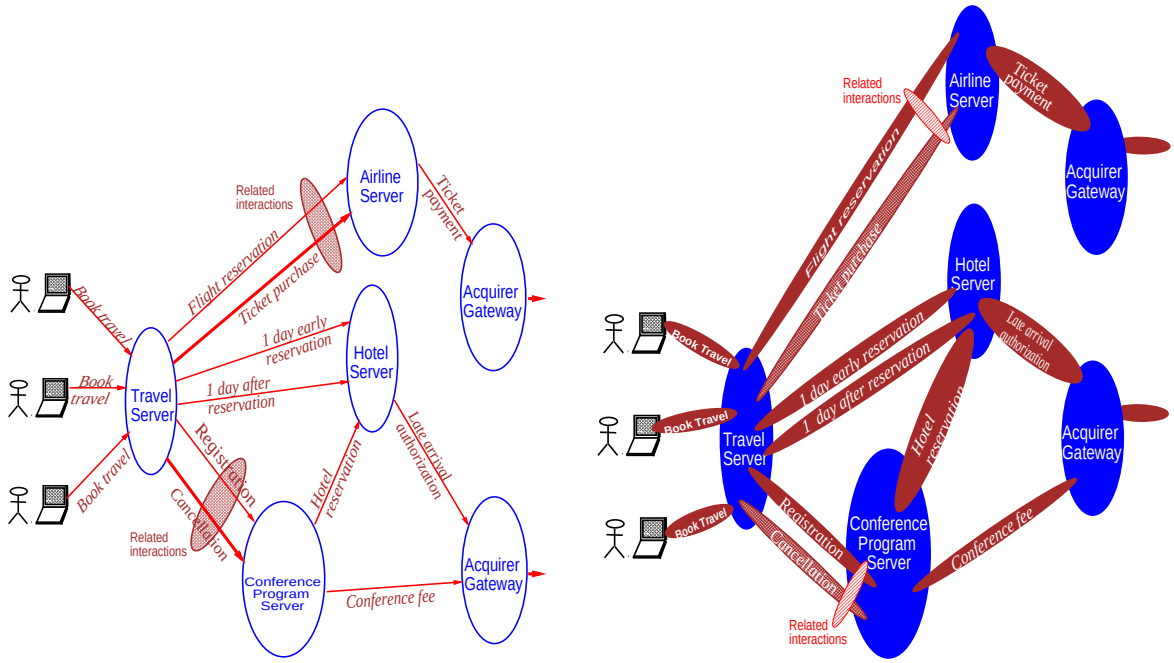
The proposed extended transaction model for addressing the needs of the Internet transaction processing environment does not entirely replace the need for classical ACID transactions. As has been suggested by other works on extended transactions that there may be extended business interactions, i.e., these may not be ACID transactions however their subportions may include ACID transactions. It will be shown that the proposed Coyote transaction monitor can provide this service where needed. We would like to emphasize the contrast in requirements (e.g., in messages and processing) between the traditional transaction processing environment and the new environment. In the traditional environment, the messages and processing primarily dealt with groups of transactionally coordinated ACID actions. In contrast, in the new environment, there may be many more messages and processing dealing with various new actions (e.g., read-only unstructured browsing, asynchronous event handling for compensation actions) and fewer ACID actions. Hence, we are arguing that the design point for an Internet transaction monitor needs to be significantly different from that for classical transaction processing.

The remainder of the paper is organized as follows. In Section 2, we use a motivating example to further explain the *Cover Yourself Transaction Environment (COYOTE)* approach for building applications that use a simple integrity model. We will then explore its relationships to earlier works: the ConTract model [18, 17] and various Workflow/taskflow systems [1, 3, 11, 6, 14, 16]. 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. The details of the compensation services provided by the COYOTE monitor and reliable execution of service requests are provided in Sections 4 and 5, respectively. In Section 6 we provide an overview of how COYOTE environment and services can be used for developing such applications, and comments on the conveniences of this environment to the application developer. Section 7 contains a summary and conclusions of this paper.

2 COYOTE Approach: a Motivating Example

Consider a typical business application: *conference registration* (See Figure 1(a)). The *end-users* need to make complete travel plans associated with attending this conference. They may run local

travel arrangement applications on their desktops/laptops, but are more likely to contact a *Travel Server* that has access to all the business servers which need to participate in reserving this travel plan. As part of the conference registration, the conference organizers provide (via the *Conference Program Server*) 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 arrangement with the hotel (via the *Hotel Server*) 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 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 1: Conference registration example

An end-user may also make separate arrangements 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 application on the underlying systems are as follows. Multiple organizations (e.g., Travel Server, Conference Program Server, Hotel Server, Airline Server, 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. Each organization is responsible for delivering its services. The requested and granted services may be cancelled by (or on-behalf of) the end-user at a later time. The organizations do not need to be in constant communication during the course of this long running application. It is very clear from the earlier discussions that the traditional transaction processing technologies alone can not satisfy these new requirements. Next we describe the COYOTE approach that is appropriate for building such applications.

2.1 COYOTE Approach

In this paper, we advocate the use of a simple integrity model for supporting commercial applications. Under this approach, each organization defines a set of well defined data access interfaces (i.e., transactions) that can be invoked by remote agents to access data maintained by this organization (see Figure 1(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 defined service interface and does not know whether the implementation of the service involves sub-contracted services potentially at other nodes
3. 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.
4. The monitor, that supports server application, provides a persistent application level log and execution of functions 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.

Figure 1(b) illustrates use of 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.

2.1.1 Site Autonomy: Privacy and Implementation Hiding

Site autonomy and privacy are important issues in this new paradigm of ubiquitous connectedness via the web. Each site should have a local view of the overall application, and is responsible for the delivery of its services. No organization gets involved in the business of another organization, and yet, the services provided by each organization, which may in turn depend on the services of other organizations, need to be automated. 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.

2.2 Related Work

Figure 2 illustrates graphically the relationships and overlap amongst ACID transactions, nested transactions, the Coyote transaction concept, and general workflow. As a monitor, COYOTE shares a lot of similarity with the classical TP monitors (i.e., management of registered applications and runtime environment). It also supports a more general transaction concept and various extended transaction models. Sections 3 through 6 will describe various aspects of the COYOTE monitor: the

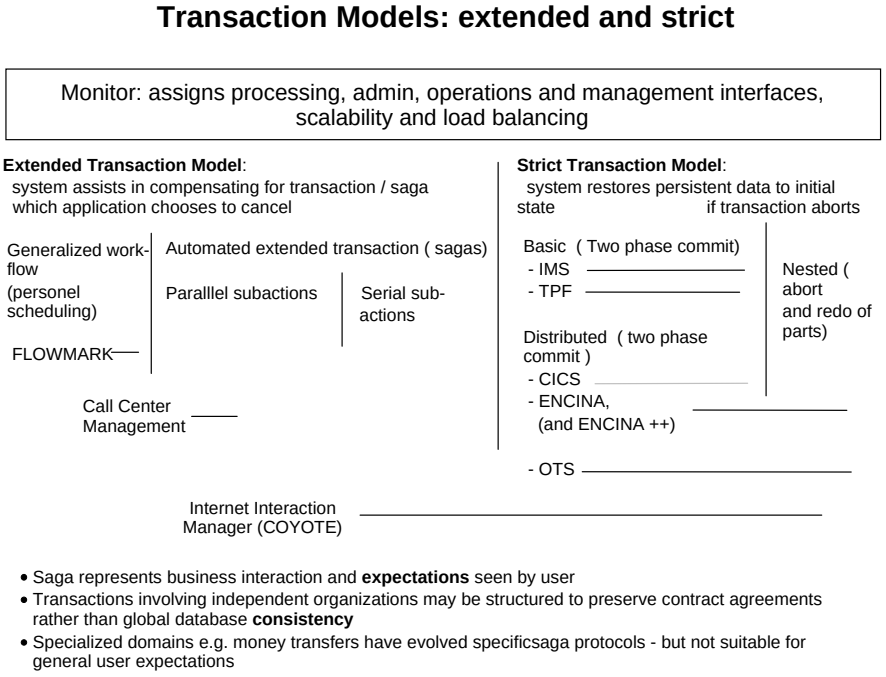


Figure 2: Extended transaction models

essential concepts, compensation and log services, reliable execution and application development. The ConTract model in [18] also describes how long running applications can be written using predefined actions (i.e., *steps*) and well defined control flow (i.e., *scripts*). However, in COYOTE we seek additional supports for pairwise conversations (that are not predefined) between independent organizations. A single service request may execute several *steps* based on a well defined *script*. However, as part of this ongoing conversation, a separate message may result in invocation of a separate *script* that relies on the execution state of the previous *script*. For ease of development of applications that support such conversations the environment also should provide support for form (e.g., HTML) management and processing. The contrasts of COYOTE monitors from workflow systems are also quite clear. The emphasis in COYOTE is on automation and performance needed for Internet based businesses, where as allocation of work to pools of human operators, and maintenance of the workflow pattern in a general database structure tends to be a design point for workflow systems. Of course, the COYOTE applications are likely to be able to include human assistance in typical Coyote supported automation, but only in a very small portion of the traffic through the monitor.

3 Essential Concepts in COYOTE Monitors

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 Requests, 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.

In a typical interaction, a 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 requests: Service requests are the functional requests which flow on a conversation; examples would include making a specific hotel reservation or issuing tickets for a specified itinerary. A service request identifies a function or transaction name which when it arrives at the Coyote transaction manager causes processing of an instance of that named function 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 processing of a particular service function, to cancel a previously issued request for service, to abort an incompletely processed action or a modify action to change what was done in a particular service function. Hence one can think of a sequence of action requests, e.g. (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 function: *making a hotel reservation*.

3.2 The Coyote Monitor

The basic functions of a standard transaction monitor are well known, e.g. CICS, Encina, Tuxedo [4, 12, 7]. We summarize briefly for the purposes of comparing and contrasting special features of the Coyote monitor.

3.3 ACID Transaction Monitors

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

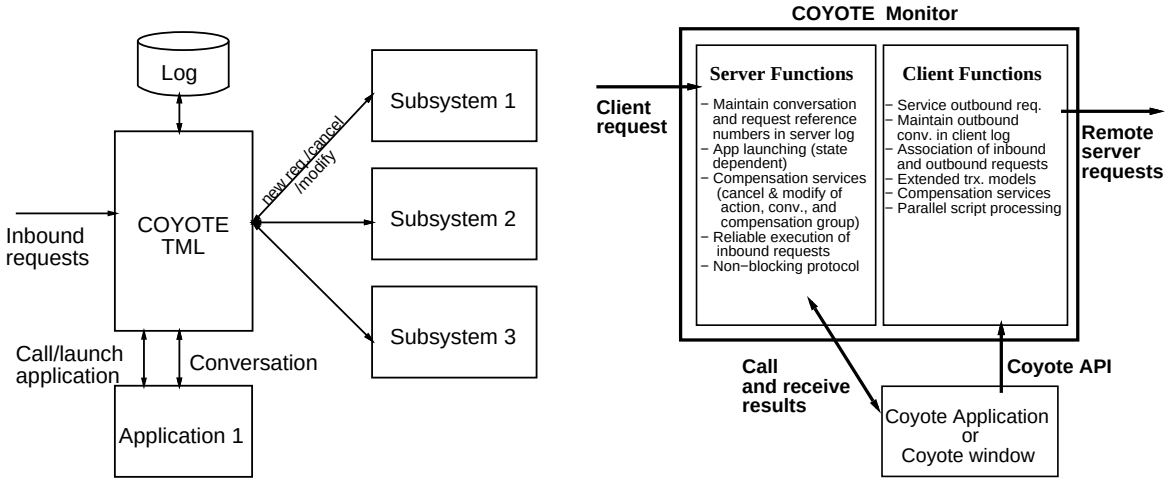
Incoming requests to the transaction monitor appear as messages typically from remote clients. An incoming request is logically queued by the transaction monitor until computing resource is available to execute it. This occurs when some other transaction completes and frees up a process or thread in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

to the function in the incoming request on the allocated thread, 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, independence and durability) can be found in [13, 12]. Treating atomicity as the most essential of these: a primary function of standard transaction monitors is to provide a commit/abort protocol, so that the transactional application can be ABORTED if either a serious error occurs during processing on behalf 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 any resource managers or databases which it has used.

A schematic view of a Coyote monitor is shown in Figure 3. At this level the architectures of a classical ACID transaction monitor and a Coyote monitor are essentially the same.



(a) Coyote monitor schematic (b) Logical structure of Coyote

Figure 3: Coyote monitor

3.4 Novel Aspects of COYOTE Monitors

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 which is sure 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 function registered with the Coyote monitor can have a group of programs associated with it defining, the primary action, and its associated application defined compensation and modify actions.

4. 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.

5. 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 feature of a Coyote monitor is that it shifts the definition of a transactional application away from the concept of 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.

3.5 Coyote Service Request Registration

The COYOTE monitor provides support for issuing, cancelling, reissuing and modifying services requests, where cancellation implies taking some compensation action defined by the service. To provide this support, all service request functions available at a Coyote monitor and used by

applications executing at that monitor must be **registered** with the monitor. Registration always involves defining:

- the name of the service request,
- 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.

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 request, the Coyote monitor is prepared to receive incoming messages requesting that service, and thereupon to start an application program executing on a thread under its control to provide 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 service requests are the services which can be called by applications executing under the Coyote monitor to get at services provided either by other nodes or possibly even at the same node but not processed under the control of the Coyote monitor. For these actions, the registered execution method and the parameters of the IPC, RPC or HTTP requests are used to pass the request for service to the appropriate execution environment.

The function of the Coyote monitor is therefore to use its table of registered inbound service requests to schedule application processing when requests are received, and to intercept all the outgoing service requests made by this started application and forward those requests following information in its table of registered outbound requests.

The fact that a Coyote monitor treats inbound and outbound registrations separately providing different processing and services is illustrated diagrammatically in Figure 3(b).

3.6 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 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, Netscape cookies 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 action requests can now be made on the established conversation. A client will typically first issues an action for a "new" service request. These messages do not have a unique service request number established by server Coyote monitor hence when received, the Coyote monitor must do some processing to determine whether the action request is a duplicate or a valid definition of a new service request for which a unique service request number associated with the conversation will be created. Each action request message includes fields for:

- the client supplied Service Request Number (SRN)
- the server supplied unique SRN ()

Client side unique numbering of requests is optional. For "new" action requests, server supplied SRN is always null on the incoming request (since it has not yet been allocated by the monitor). If the client is not uniquely numbering its service requests, (i.e. client SRN is null or omitted), the Coyote monitor will generate a unique SRN 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 SRN. 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 SRN 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 5. Having established that a valid (non-duplicated) new action request has been received, the Coyote monitor will generate a unique-within-conversation SRN for it and save it in the log. Parameters for the action request and if present the client's SRN 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 SRN, 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, write the allocated server SRN into the reply message and send it back 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 SRN 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 SRN must be supplied in cancel and modify actions and this will be used. The monitor then uses its 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, SRN and action parameters, but also to the original parameters of the "new" service request which started this SRN. 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. 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.

The monitor will assign unique client SRNs to all its outbound requests and will log these SRNs and the outbound action parameter lists in the Coyote log. It will also log reply parameters and any remote server SRNs 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 them. 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. In the case where outbound requests are made to another Coyote monitor, then it is the responsibility of the application to start an appropriate outbound Coyote conversation. In particular the application must decide whether it will do this transparently i.e. passing through the identification of the user responsible for invoking it, or whether to make the sub-request on its own authority as a separate conversation understood by the monitor to be initiated from its conversation.

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 and logged. 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.

3.7 The Coyote Log

As alluded to already in preceding subsections, the Coyote monitor will build and maintain a persistent log which records all incoming action requests, their associated client and server SRNs and their input and reply parameter lists. For outbound service requests, the monitor stores similar

information except that server SRN numbering cannot be relied upon in this case (just as client SRNs were not required on the input side).

This information is maintained in a hierarchy:

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

Logging services are provided for the applications to retrieve this information - the log is presumed to flow over time into some relational database with indexes based on the hierarchy above. However since 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.

3.8 Other Services Provided by the Coyote Monitor

This subsection provides a brief overview of the two other services provided by the Coyote monitor. These services are treated more fully in the two following sections of the paper

Compensation: In the next Sections, we will describe how applications can be structured and how Coyote can maintain appropriate information in its log to enable this compensation service at the Coyote. If a server Coyote at a site fails to provide the desired service the client Coyote at the previous site needs to cancel related activities performed on other servers (via compensating actions). Each interaction need to be identified so that if needed it can subsequently be compensated. Under COYOTE approach, each service completion returns a *server reference number* which are used by the subsequent compensating actions (if any). Note that depending upon the application need, the client Coyote may choose various transactional semantics. For example, it may group only a set of interactions with other servers to be performed atomically, called the *Compensation group*. Any failure in an interaction with another server will result in cancellation of the completed interactions of that group while leaving the other actions by that extended transaction uncompensated.

Reliable execution of requests: Open-ended interactions between any client and server require a confirmation mechanism that ensures that the server has completed its services. A client request may fail for various reasons: the message may be lost, the server may fail to complete its services not only due to failed applications in its own node but also due to failure in other servers from which it receives services. Even when a server completes its services the client may not receive this confirmation due to lost messages. Therefore, only way a client is sure of the completion of its request is when it receives the confirmation from the server. In the absence of any such confirmation message, the client may retry the request or polls the server of its earlier request. The server has to make sure that the client receives the confirmation by receiving an echo of the confirmation message from the client. In any case, the server has to avoid processing duplicate requests. This reliable execution of the request that provides confirmation and avoids repeated execution of the same request is referred to as an well-defined interaction earlier. We will refer to this as *Reliable Execution of Request* or *RER*. Note that a network level reliable message (e.g., as provided by the MQSeries services) cannot guarantee this reliable execution of request.

4 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.

Consider the following single site application (See, Figure 3(a)). The application receives well-defined services from three subsystems, labeled subsystem 1, subsystem 2 and subsystem 3. It also requires support for various extended transactional semantics: i) compensation of a single service request, ii) modification of a previous service request, iii) compensation group services, i.e., either all or none of the specified services of a group are obtained, iv) delayed cancellation of an earlier completed transaction or compensation group or a single service request. A subsystem may not always allow, or it may be impossible to provide, compensation of certain types of services. The subsystem may also allow cancellation of a granted service only for a limited time period after the service is granted. As discussed in Section 2, these services will be useful in building multi-site applications. We will assume that no support is available for two-phase commit across various subsystems. In any case, support for two-phase commit alone can not provide all of the above requirements (e.g., delayed cancellation). However, the above requirements are quite natural in supporting business activities across multiple organizations. Finally, the Coyote should also provide appropriate support to make writing of business applications easier.

An application may request services from various subsystems following the above Extended

Transaction Model (ETM) semantics. All service requests go through the COYOTE transaction monitor and logger. The Coyote is responsible for maintaining the state of the transaction (i.e., application) in a log. The subsystems register with the Coyote a set of services which can be requested by an application via the Coyote. Corresponding to each service, one or more entry points may be registered with the Coyote which are called upon a new, cancel or modify service request. For each new application service request, the Coyote generates an unique reference number and passes this to the subsystem along with other parameters. If the subsystem is successful in completing its service, the reference number along with the results are returned to the application. (For this single site application, we will assume the service request calls are synchronous, and hence, issues in reliable execution are ignored. The issues are explored in detail in Section 5.) The Coyote stores in its log the id of the application, the service and subsystem id, type of request (new, cancel, modify), parameters associated with a new request (to be used in later requests), the results of the service, and the associated reference number. The time-of-the-day information is also stored in the log to verify that delayed cancellation of service (if such an entry point is registered with the Coyote) is not denied by the subsystem.

Compensation of action: The application may cancel an already granted service at a later point via an explicit request to the Coyote. The Coyote may also generate this cancellation request implicitly as part of the transaction semantics. Application passes only the service reference number for this cancellation to the Coyote. The Coyote retrieves all the associated parameters with the original request as well as the returned results from its log. If no such prior service request is found in the log or the service is already cancelled, or the cancellation request is too late, or the service request is still outstanding, it returns appropriate error code to the application. (In Section 5, we will explore how these responsibilities are shared across Coyotes in various sites, i.e., division of client and server Coyote activities.) Otherwise, the entry point corresponding to the cancellation request is called. If the cancellation is denied by the subsystem wrongfully or it fails for other reasons, appropriate error codes are returned to the application. The error conditions are also logged in a separate log for later dispute resolution.

The application may also modify its earlier service requests with new set of parameters. The Coyote again, as above, checks to see if the modify request should be honored, i.e., if it is allowed by the subsystem, if the service is already cancelled, and if the delay is within time limit. If so, the Coyote retrieves parameters associated with the earlier service request, returned results and the reference number, and passes these to the subsystem along with the new set of parameters.

Compensation of conversation: All service requests by the application are associated with a conversation thread, or Extended Transaction (ET). A new conversation or ET at the Coyote is

started by an application using the Coyote interface OPEN_CONVERSATION(). The application id, the user id, and the user conversation number are provided to the Coyote as part of this call. The user id will be used for authentication purpose, and for checking privileges and restrictions in subsequent interactions. The Coyote 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 Coyote conversation number parameter is also passed by this application with all its interactions with the Coyote. The application supplied user conversation number is logged by the Coyote, and used to corelate cascaded cancellation. Its purpose would be clearer in Section 2.

Compensation group: A compensation group is similarly defined by the application by an OPEN_COMPENSATION_GROUP() call to the Coyote. 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 CANCEL_COMPENSATION_GROUP() call to the Coyote. A conversation group is closed by a CLOSE_COMPENSATION_GROUP() call. Note that individual requests in a conversation group can not be cancelled or modified. However, an entire compensation group can be cancelled by a CANCEL_COMPENSATION_GROUP() call to the Coyote.

Query of state: Finally, the application may enquire Coyote about the following:

1. complete state of an ET i.e., set of interactions and their reference numbers, and compensation group numbers.
2. the state of an compensation group, i.e., set of interactions, reference numbers, and state of the group.
3. state of an interaction, i.e., new, cancel and modify requests associated with this reference number.

These information can be used by the application in its logic.

5 COYOTE Services for Reliable Execution

Under COYOTE approach, each node is assumed to be independent. An application requests service from another independent application at the same site or a remote site via the Coyotes in each site.

This is illustrated in Figure 4, where an application in site A receives service from another application in site B via the shaded Coyotes.

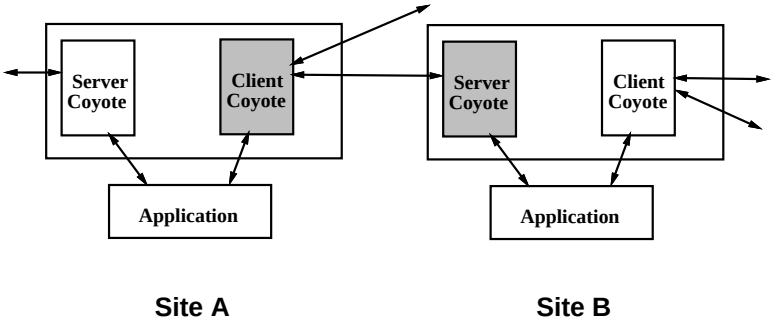


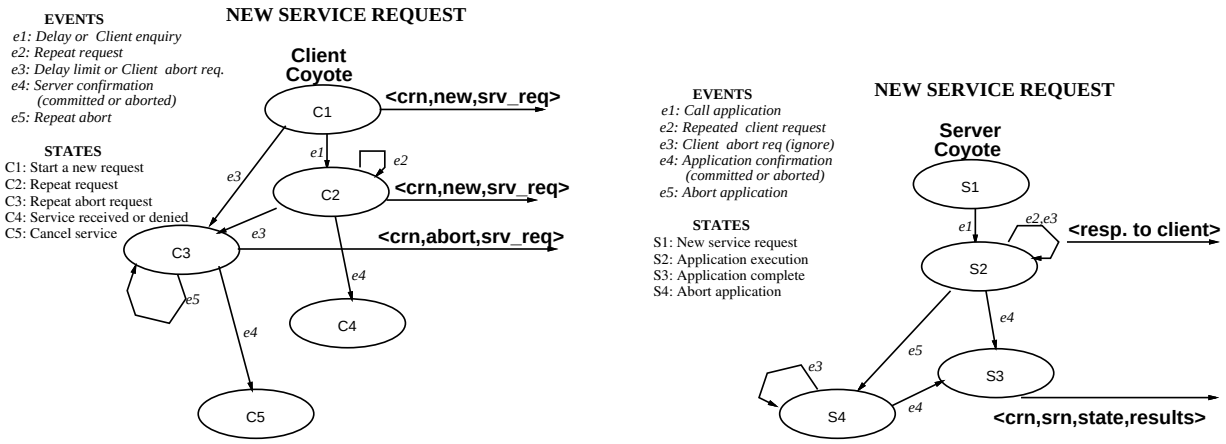
Figure 4: Reliable execution of request via Coyote monitors

Underlying communication support may be reliable, or unreliable. In any case, reliable sending of a request (from the Coyote at the client site to the Coyote at the server site) does not guarantee that the execution of the request will be successful. The client needs a confirmation from the server about the success or failure in providing this service. Figure 5 shows the states of client and server Coyotes in achieving this reliable execution of a new service request. Upon receiving a client request, the client Coyote enters state $C1$. It generates an unique *client reference number* or CRN and sends this service request to the server Coyote along with the CRN. Other parameters sent with this request include client Coyote id, service id, request type (new), and the required parameter list for this service.

To provide an uniform and simple service interface to the application, the client Coyote may maintain a mapping table of all services provided by other servers. The client request is mapped to the appropriate service request by the Coyote. For example, multiple organizations may provide the same service. However, currently a particular site may be chosen by the client site for this service. The application will remain unchanged with the selection of a different service provider. The client Coyote also maintain the types of requests possible for this service, i.e., cancelable, modifiable, etc.

If the server Coyote doesn't respond within some pre-specified time limit, the client Coyote may retry this request. The service request by the application to the Coyote may follow a synchronous or asynchronous model. In the later case, the Coyote returns to the application immediately a CRN which is used by the client for inquiring the state of the request at a later point. The event *e1* represents both these cases. The client Coyote enters state $C2$ and resends the original service request to the server Coyote.

If the delay exceeds an acceptable time limit or if the application sends an explicit abort message, the client Coyote enters state $C3$ and sends repeated abort request (with proper choice of intervals)



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

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

Figure 5: Execution states of a service request in clients and servers

to the server Coyote until it receives confirmation from the server Coyote. The server may not succeed in aborting the application, or it may return a successful completion message. The client Coyote now enters state $C5$, and if the server is successful in completing the requested service, the client Coyote starts a new cycle for compensation of this service. The client Coyote also starts this cycle after receiving a delayed compensation request from the application. Finally, if the service completion message is received while the client Coyote is in state $C2$, it enters state $C4$. The server returns a *server reference number* the state of the service completion (i.e., committed or aborted) and the associated results. The client Coyote logs appropriate information and passes this result to the application, as detailed in Section 5.

Figure 5(b) shows the states of the server Coyote associated with the above execution sequence. Upon receiving a new service request, the server enters state $S1$. It generates an unique server reference number or SRN, calls the appropriate application based on the service map table, and enters state $S2$. Repeated client requests for the same service (determined by checking the CRN associated with this request and looking for an earlier entry with this CRN in the log) are ignored. The request for aborting an application is ignored if the service request can not be aborted by the application. (This feature will be registered with the server Coyote when the entries for this service are registered.) Otherwise, the server Coyote enters state $S4$ and sends an abort request to the application. Optionally, the server Coyote also sends appropriate message to the client Coyote informing the state of the execution to avoid repeated messages from the client Coyote. After

the application completes processing the service request, it returns to the server Coyote either a committed message along with the results associated with this execution, or an aborted message along with detailed reasons for failure. The server Coyote passes this information to the client Coyote, along with the CRN and SRN.

The execution of cancellation and modification of an earlier granted service request are similar. The details are omitted here for the sake of brevity.

6 COYOTE Application Development

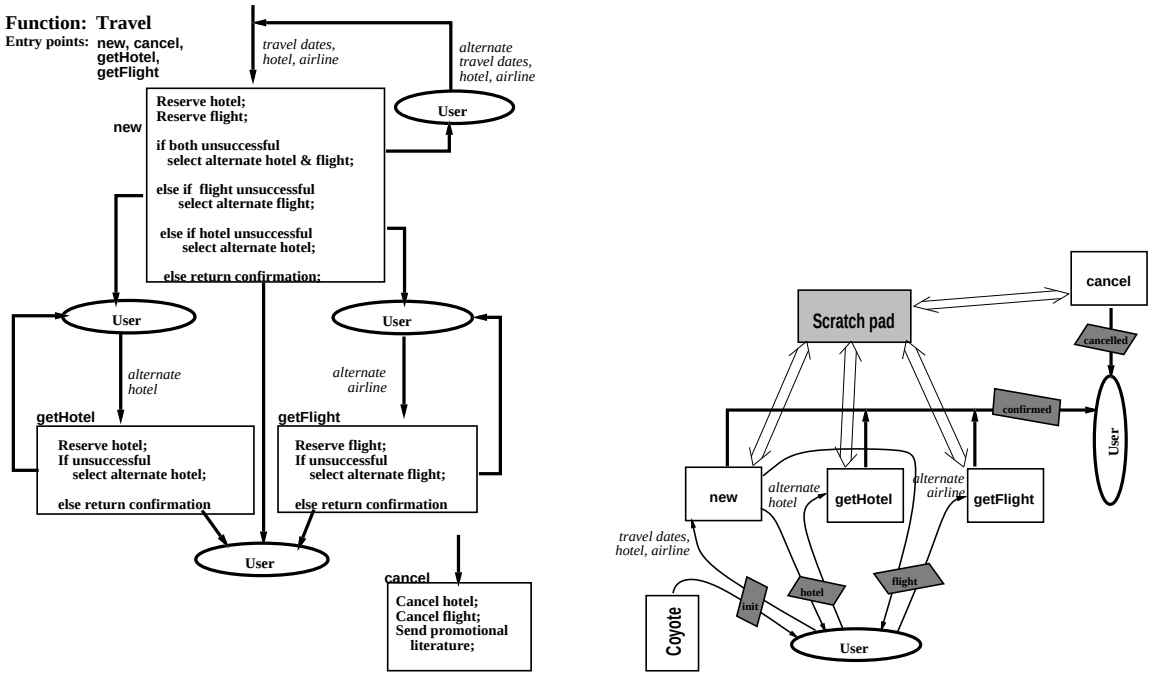
In this Section, we will illustrate the use of Coyote services in writing business applications, and demonstrate how Coyote monitor makes writing complicated applications simpler by factoring out the complicated code for state maintenance, application launching, reliable service execution, form and code management and sharing of data across independent application code segments as Coyote services.

6.1 Travel Booking

Figure 6(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 1). 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.[1] 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 5). 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

[1] 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.



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

Figure 6: Application development

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 6(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 [18] 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 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 are complex, and need to be reliable. Hence, factoring out these code modules as Coyote services is important.

7 Summary and Conclusions

The Coyote system offers a monitor structure suitable for developing long running transactions in an Internet or network centric environment where base services are provided as transactions on other connected processors.

The server running the Coyote monitor typically adds value as a *"middle tier"* server by augmenting and recombining the base transactions. This makes them more useful to new classes of users reachable in the network environment.

Key technical features provided by Coyote 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.

Having these facilities available simplifies the composition of *"middle tier server"* applications. In the absence of these services, development of such applications on a classical transaction or workflow server is significantly more difficult.

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.

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

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

[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] 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.

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

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

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

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

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

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

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

[14] 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.

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

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

[17] 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.

[18] Waechter, H., and A. Reuter, "The ConTract Model", Chapeter 7, pp. 219-263, in [2].