The Coyote Project:
Framework for Multi-party E-Commerce[1]

Asit Dan, Daniel Dias, Thao Nguyen, Marty Sachs, Hidayatullah Shaikh,
Richard King and Sastry Duri

IBM Research Division, T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract: The Internet provides the opportunity for quickly setting up deals between businesses for promoting each other's products, and to jointly offer new services. Specification and enforcement of such deals stretch traditional transaction processing concepts in several directions since they involve independent businesses with their own internal processes. First, the greater variability in response time in business to business interaction creates a need for asynchronous and event-driven processing, in which correct handling of reissued and cancelled requests is critical. Second, a new transaction processing paradigm is required that supports different views of a **unit of business** for all participants, i.e., service providers as well as end consumers. Between any two interacting parties, there may be several related interactions dispersed in time, creating a **long running conversation**. This paper describes our approach (Coyote) to solving these problems including use of a service contract for specifying the rules of interaction across businesses, and directly generating code for enforcement of the contract. We finally describe the architecture and a prototype of a system which implements the Coyote concepts.

1. Introduction

The ubiquity in network connectivity promises electronic commerce in every walk of life, e.g., shopping, entertainment, education, etc. The client merely connects via an access device to the network, i.e., to a long running service application, and performs business transactions, e.g., creating and modifying a travel plan, checking the status of a purchase request, accepting discount offers, etc. Many such commerce activities may actually span multiple autonomous business organizations. For example, when a client makes a purchase request for an item, he may receive a *business deal* in the form of

---

[1] Proc. 7th Delos Workshop on Electronic Commerce, Crete, Greece, Sept. 21-23, 1998, *Lecture Notes in Computer Science*, Vol. 1513, p. 873, Springer-Verlag, 1998.

receiving a discount on another product, perhaps from a different business organization. Figure 1.1 illustrates one of many such possible deals:

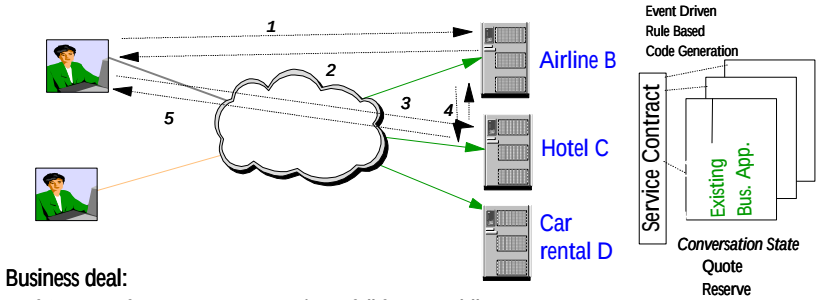

Figure 1.1: Multi-party electronic commerce

If a customer from company A purchases full fare on Airline B, then the customer receives a 20% discount at Hotel C, half of which is reimbursed to Hotel C by Airline B; The customer also receives a 10% discount at Car Rental D, half of which is reimbursed to Car Rental D by Airline B. During check-in at Hotel C and/or Car Rental D the customer must confirm that he/she travelled on Airline B; if there is a timeout during confirmation, successful confirmation is assumed.

It is easy to imagine a wide variety of business deals across multiple business organizations with different terms and conditions, in order to promote each other's products and to provide an improved set of services to a customer. For example, a university enrolment may entitle a student to a discount in a book store; a Broadway ticket may offer a discount at a restaurant, etc. In such cases, there may be concomitant payment between the businesses, transparent to the client. Even between a single selling organization and a single buying organization, many different deals are possible, e.g., *an airline ticket purchased from the Airline A by the customers from the organization B is refundable until 24 hours prior to the date of arrival.* To facilitate spontaneous electronic commerce, deals need to be constructed and implemented quickly, reflecting the dynamic nature of the marketplace. Deals may also promote customer loyalty and ensure guaranteed services across organizations.

Lack of a suitable application development platform, however, hinders quick implementation of such deals on a larger scale. Taken in its entirety, each such business deal is a ***distributed long running application spanning multiple autonomous***

business organizations [1,2]. Specification and enforcement of such deals stretch traditional strict transaction processing concepts in several directions. First, transactions spanning multiple independent organizations may need to address enforcement of pairwise legal agreements and **service contracts**. Second, a new transaction processing paradigm is required that supports different views of a **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**. Achieving global data consistency across the businesses participating in a contract is not possible, both because of the length of time which may be involved in a conversation (which precludes locking the internal resources of a business for the duration of the conversation) and because each business has its own internal processes which generally cannot be synchronized with the processes of other businesses, precluding use of a two-phase commit process across the businesses. Instead, persistent records of business actions must to be kept at each business to aid in recovery, auditing, etc. 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.

We illustrate shortly that conventional technologies do not address the requirements of the above types of applications. As a result, currently it requires a long time to set up electronic business operations across organizations. To alleviate some of these problems, standard Electronic Data Interchange (EDI) messages representing service requests and replies have been defined (e.g., Purchase order (PO), Request for Quote (RFQ)) [3]. Some of these are unique to specific industry segments; others have industry-specific variants. In addition, further agreements (referred to as Implementation Conventions, and complex trading partner agreements) are needed before such EDI messages can be exchanged across business applications. Finally, note that building long running, stateful distributed applications, taking into account various state dependent exception handling, and rollback/compensation of previous operations, etc., is always a challenging task. Erroneous (and, in the worst case, malicious) implementation of application components by a business partner can adversely affect the businesses of other partners.

We now outline the Coyote approach and contrast this approach with earlier models in the next two subsections. In Section 2, we describe the key elements of a service contract. We present the details of the overall application structure and illustrate the associated execution flow in Section 3. Subsequently, we describe the Coyote server environment along with details of the some of the services in Section 4. Finally, a summary and concluding remarks appear in Section 5.

1.1 Coyote Approach

The Coyote (Cover YOurself Transaction Environment) approach recognizes the key features of a multiparty e-commerce application, i.e., *long running, event driven, compensation based business interactions*, and advocates development of service applications as autonomous business processes that interact with other business processes via formally defined rules of engagement. It advocates a clear separation of

states across businesses as well as a separation of *internal and external business processes* within an organization. The rules of external interaction and externally visible states are defined as a ***service contract*** [1,2]. The semantics of such external interaction are well defined, such that there is no ambiguity across businesses as to what each message may mean, what actions are allowed, and what responses are expected at any

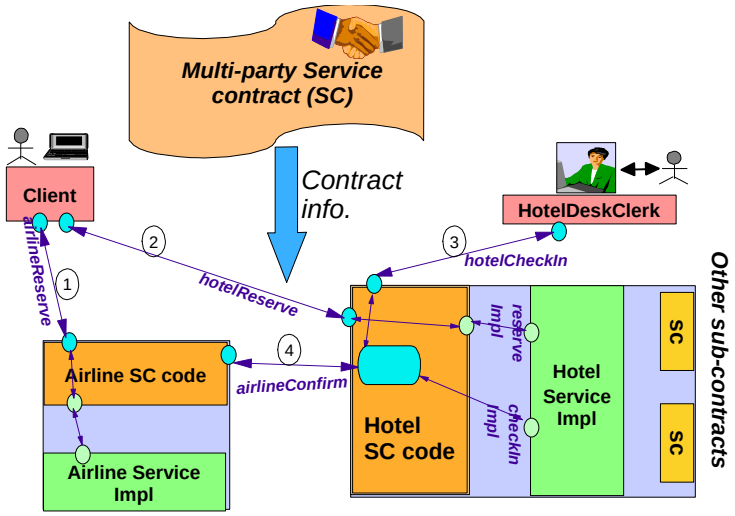

Figure 1.2: Coyote Approach

point during long running interactions (also referred to as *conversations*). A service contract acts both as a guideline for interaction across businesses and also as an enforcement mechanism for guaranteeing proper interaction. Furthermore, at the contract level there is no specification of internal processes of the businesses that may introduce unnecessary dependency or close coupling across businesses. For example, when a customer returns a previously purchased product to the selling organization, there is no need to specify what the selling organization may do with the returned item (e.g., send back to the producer, reshelve the item, etc.).

At the semantic level all messages are assumed to be committed and, hence, cannot be rolled back. In contrast, traditional distributed business applications embrace a distributed transaction model, where each application component runs under a transaction monitor so that the application components can participate in a two-phase commit protocol [4,5]. Note that such traditional models of distributed application development not only introduce close dependencies across software systems in different businesses, but are also inadequate to model long running interactions. Under the Coyote approach the business interactions are modelled after common business practice, i.e., cancellation and/or compensation of previous requests.

Another important aspect of the Coyote approach is to provide an application development and execution environment for *long running, event-driven compensation-*

*based business interaction*s. The overall business application is built from code segments that capture internal business processes and the service contract(s). Figure 1.2 illustrates the development of a multiparty e-commerce application for the earlier mentioned business deal. There are three parties (i.e., client, Hotel and Airline) participating in this deal. (The Car rental agency is not shown in the Figure.) The client for this deal can be a travel agency that makes reservations on behalf of its customers.

To implement this deal the client, Airline and Hotel companies generate deal-specific code from this joint deal specification. The generated code on each side also

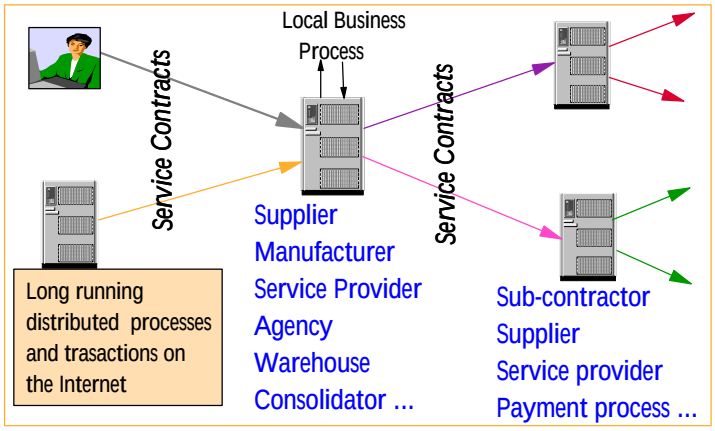

Figure 1.3: Generic Business service application model

connects to the corresponding internal processes. As part of this deal, the client makes a reservation on both the Airline and Hotel companies (actions 1 & 2). As illustrated, each reservation operation has two components: the contract specific code maintaining the state of the interaction and invocation of the reservation service implementation in the Hotel and Airline companies, respectively. At a later point, the customer flies via the Airline, and subsequently checks in at the Hotel (action 3). The Hotel, in turn, as specified in the contract, confirms with the Airline (action 4) that the end-client has indeed travelled via the Airline participating in this deal.

Figure 1.3 illustrates the use of the Coyote framework in a general business setting, where different organizations provide different services as well as receive services from other business organizations. A single business process may interact with many different business partners, where the interactions are specified via one or more service contracts. The Coyote server in each of the business nodes (which can be suppliers, manufacturers, agencies, warehouse owners, etc.) facilitates development and execution of its overall business processes. The services provided by a business organization can be invoked either directly by an end-user (e.g., through the web) or by a service

application in another business, where allowable interactions are defined via a service contract.

1.2 Related work

Both transaction processing and providing the support for distributed applications, have been active areas of research and development for many years. In the area of transaction processing, the focus has been on extending the ACID transaction model to capture dependencies across many atomic steps of a long running transactional application [6-10]. In contrast to Coyote, the focus in supporting distributed applications has been on middleware for gluing together independently developed application components to build complex but synchronous applications within a single business organization. The CORBA IDL [11] provides the interface of an application component that encapsulates some business logic. The programmer is freed from various system resource management issues (e.g., application scalability, thread allocation, runtime allocation of system nodes to an application component, etc.). However, the programmer has to still understand the overall semantics and make sure the sequence of calls to an application component is meaningful (e.g., that it is a valid sequence of operations on an object). In the current e-commerce context, where different application components are created and managed by different organizations, the programmers in various organizations have to cooperate in creating a meaningful application. Service contracts that formally define not only the interfaces and method signatures, but also the rules of interaction, allow the application development process in various organizations to be decoupled.

Workflow systems provide support for definition and execution of long running applications [12,8,9]. Typically, the execution of an application step may also require the participation of a specific user or a member of a group. The application steps may be developed separately as individual programs and the workflow process defines the sequence in which the steps are to be executed in a Petri-net-like manner. At runtime, the workflow monitor co-ordinates execution of the overall application by maintaining the state of the overall execution flow, maintaining the output data of the previous application steps, selecting the next application step or steps to be invoked, and invoking these application steps by passing appropriate input data. While workflow is an important technology for modelling many business processes within an organization, it may be inadequate for supporting distributed applications capturing multi-party business deals. First, irrespective of the application component programming model, a Coyote-like framework is needed for specification of the service contract, and the generation of the code in each organization that deals with service contracts and, in turn, invokes the corresponding internal business processes. Second, appropriate interaction (i.e., conversation) semantics need to be defined that both decouple the individual internal business processes, and yet allow cancellation of earlier, committed operations.

One implication of the above point is that, at any time in a conversation, a business partner may be allowed to issue many possible service requests, rather than a fixed, small set of operations. Therefore, the set of possible invocation sequences is very large, and a Petri-net like definition may not be appropriate. This is illustrated in the

Petri-net of Figure 1.4, where each node represents the operations performed in a business organization. When transition 1 on Node A fires, it triggers actions 2 on both nodes B and C in parallel. Depending on the responses from nodes B and C, action 3

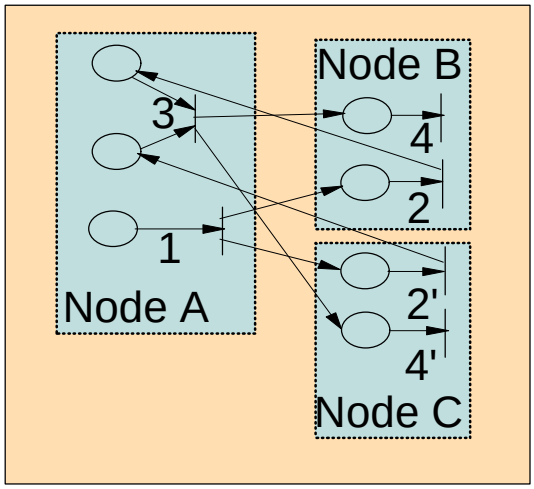

Figure 1.4: Building a Service Flow from Workflow

on node A is triggered. Since there are many possible responses from nodes B and C, other actions on node A may need to be triggered instead of action 3. Action 3 then triggers actions 4 on nodes B and C, and so on. Since the number of possible action sequences is large, representing them in a Petri-net model, and therefore, in a workflow model, becomes complex. Generally speaking, the next service request issued by a partner as well as the set of requests accepted by the other partner depend on the conversation history. In the absence of a global workflow monitor spanning multiple organizations, the conversation history needs to be passed around. (Note that the individual application steps are not aware of the global execution steps.) Therefore, the workflow and other business processing implementation infrastructures need to be extended to manage long running conversational states across businesses as further illustrated in Section 3.

2. Coyote Multi-party Service Contract

As discussed in section 1.1, the contract is a high-level description of the interaction between two or more business partners (parties to the contract). The contract contains two kinds of information. The first is a machine-readable description of the computer-to-computer interactions between the parties that supports the overall application. It concerns those aspects of the application which each party must agree to and that are enforceable by the Coyote system. The second is the usual human-readable legal language that is part of any business-to-business contract and includes those aspects of the agreement which must be enforced by person to person contact rather

than by the computer systems. This project is concerned with the machine-readable section, which we now describe in more detail and which is what we mean when we refer to the contract.

The contract is written in a language from which code can be directly generated. We have selected XML [13] as the contract specification language and are developing a graphical tool that can be used to assist in generating the XML contract. After agreement by all parties, the XML contract can then be turned into code at each of the Coyote servers. In our prototype, the contract is used to generate a service-contract object (SCO) at each party to the contract. During any information exchange between two parties, each party effectively communicates with the SCO at the other party. The SCOs provide interfaces to the application program at each party to the contract. The application program in turn communicates with the individual party's internal processes.

A particular transaction under the contract is a single long-running conversation. We refer to this as a particular instantiation of the contract. An application causes the conversation to be initiated, which results in assignment of a conversation ID and in creation of the necessary SCOs at the various parties. For example, in a contract between a travel agent and an airline, a conversation would be initiated when a traveller asks the travel agent for reservations for a particular trip. All interactions about that trip between the traveller and the travel agent and between the travel agent and the airline would then be associated with that particular conversation, which would remain active until the traveller checks in with the airline for the final segment of the trip (or perhaps longer, if settlement of the travel agent's commission does not take place until later). All activity logging would be labelled with the conversation ID to ensure proper error recovery, and for auditing on a conversation basis. The complete flow of requests and replies through the SCOs to the internal business processes is described in section 3.

As indicated earlier, a contract may involve more than two parties. For example, in the multiparty application described in Section 1.1, travellers would be serviced by a travel agent which has entered into a 3-party contract with the airline and the hotel. The combined discount would be provided to the traveller when checking out of the hotel. In addition to defining the normal set of actions for making and servicing reservations, the contract would also define actions used between the airline and the hotel for confirming that the traveller met the conditions for receiving the combined discount, for settling the shares of the discount contributed by the airline and the hotel, and for settling the travel agent's commission.

In summary, the service contract defines the properties of each party to the contract that must be made visible to the other parties to the contract. These properties include action definitions for each party as a server, communication protocol definitions, security/authentication definitions, and others.

2.1 Elements of multi-party service contract

The primary elements of the contract are shown in Figure 2.1.

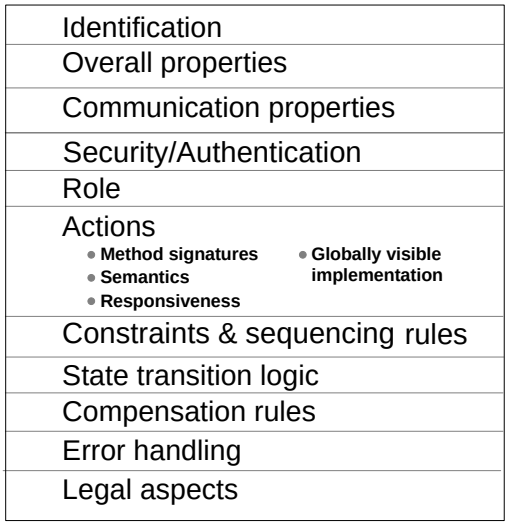

Figure 2.1: Elements of a Service contract

Identification: The identification section assigns a name to the contract and provides the names of each of the parties to the contract.

Overall properties: Overall properties are those attributes of the contract that apply to the contract as a whole and all instantiations of it. Included are the contract duration, permitted number of instantiations, maximum lifetime of a single conversation, default maximum permitted time from a request to receipt of the results of the request, and the name of an outside arbitrator which may mediate resolution of any apparent contract violations.

Communication properties: The communication-properties section provides the information necessary for each party to communicate with all the others. Each party is identified by its name and the information needed for person-to-person communications (mailing address, telephone number, and email address). An authentication code, typically a public-key certificate, is also provided by each party for security and authentication purposes. The communication protocol is described with all necessary parameters such as IP address, service name, port numbers, etc. The message format rules and security mode are also included.

Security/Authentication: The following levels of security are provided (in order from strongest to weakest): nonrepudiation, authentication, encryption, and none. The security mode can be selected for the initial exchange during contract instantiation, all messages, and some messages. The initial and message security modes are specified in the communication properties of the contract and apply to all parties. If the initial

security specification is omitted, the initial security mode is “none”. If the message security specification is omitted, the message security mode is “selective”, in which case, the security specification is included in the definition of each action which requires security above “none”. Non-repudiation both proves who sent a message and prevents later repudiation of the contents of the message. It is based on public-key cryptography. To support non-repudiation, the contract must include the public-key certificate for each party. The corresponding private key for each party, of course, is private to each party and does not appear in the contract. The public-key certificate can be obtained from any recognized certification authority. There is no requirement that all parties to a contract use the same certification authority.

Role: The contract can be formulated in terms of generic roles such as *&airlineco* and *&hotelco*. These roles can then be assigned to specific parties at the time the contract is instantiated as a particular conversation.

Actions: An action is a specific request which a party, acting as a client, can issue to a party acting as a server. Examples of actions are *purchase, reserve_hotel,* and *query_status*. For each party to the contract that can act as a server, the contract specifies an action menu, which is a list of permissible actions and various properties of each action. Action properties include its name, parameter definitions, means by which the server will return its results (e.g. callback), whether the action is cancellable, the name of the cancellation method, and maximum permitted response time.

Constraints and sequencing rules: Constraints are various conditions which must be satisfied for individual actions. For example, the action *reserve_hotel* might be accompanied by a rule stating the latest time to cancel the reservation. Sequencing rules state the permissible order of actions on a given server. For example, a *cancel_reservation* action cannot be invoked until after the *reserve_hotel* action has been invoked.

State transition logic: When an action is performed, the state associated with the action (and hence the state of the contract) changes. The contract defines additional changes of variables and parameters which take place following the completion or failure of an action. For example, the price of an item may change as the result of an action. Additionally, a single action on one party may result in invocation of actions on other parties to the contract as well as invocation of local actions. In the above multi-party example, a check-in at a hotel causes the hotel to contact the airline involved in the contract for confirmation that the traveller flew on that airline.

Compensation rules: Compensation rules state any conditions relating to the cancellation of previously-invoked actions. As mentioned above, the properties of an action include whether it is cancellable and, if so, the name of the cancellation method. Cancellation groups can be defined in which a group of actions can be cancelled with a single request.

Error handling: Error conditions and methods to be called when they occur are identified. Maximum response times are stated and the number of retries and time-out recovery actions defined. Application-level errors (e.g. no room at the hotel) are identified and recovery actions defined. An outside arbitrator is identified as mentioned above.

Legal aspects: Legal aspects are any and all conditions which are typically defined in a legal contract such as handling of disputes and other exceptional conditions. An example in the context of Coyote is what is to be done if one of the servers is not reachable via the network after a specified amount of time. Another example is rules for premature termination of the contract such as when it is allowed and the penalties which might have to be paid.

3. Coyote Application Development and Flow Through the System

In this section we outline the key aspects of creating a Coyote application, and describe the flow through a Coyote-based system. Illustrated in Figure 3.1 is a Coyote-based node, which gets inbound requests from clients; these clients could be end users or business partners. The client's requests ultimately trigger business logic at this node. The business logic may require calls to business processes or pre-existing services at that node, such as workflow processes, enterprise resource planning (ERP) applications, Enterprise JavaBeans[14] based services, or other applications. Access to such existing applications is provided by so-called connectors, or by encapsulating the applications in objects with methods provided to invoke these applications. In order to provide the requested function, services from subcontracted service providers may also be needed; these are provided by outbound requests to those service providers, as

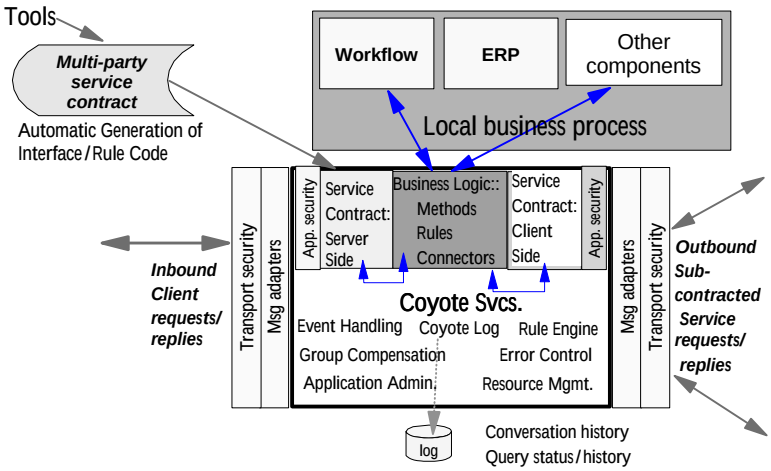

Figure 3.1: Coyote Execution Flow

illustrated in the figure. Both the inbound and outbound requests are specified and enforced by service contracts, as described in Section 2.

There are two main parts to creating a Coyote application: (i) writing a service contract between the parties involved in the application, and (ii) writing the business logic for the actions to be taken in response to requests from clients, as specified in the service contract. The service contract is written in a higher-level language, as outlined in Section 2. A code generator transforms the service contract specification into code which enforces the contract. In an object implementation[2], this code is referred to as the service contract object (SCO). From each service contract, code is generated at each site based on whether that party functions as a server, a client, or both. The service contract objects corresponding to contracts with clients are referred to as Server Side Contract Objects (SSCO), and those corresponding to contracts with service providers (relative to this Coyote server) are referred to as Client Side Contract Objects (CSCO). The contract objects enforce the specifications of the contract, such as checking if an inbound request is allowable, and checking whether its parameters match the request signature defined in the contract; if so, the appropriate method or function in the business logic is invoked, as described in more detail later.

The business logic consists of methods which either execute the required functions at this node or invoke the required services from an existing application or other subsystem. The business logic also uses Coyote services to make outbound calls to remote services.

We now outline the flow through a Coyote node. While we describe the flow primarily in the context of an object implementation, a similar flow applies to other implementations. Referring to Figure 3.1, an inbound request first goes through a transport security layer, as specified in the service contract. For example, the Secure Socket Layer (SSL) can be used for transport security. Next, depending on the transport protocol and format (as specified in the service contract), a message adapter layer transforms the request, i.e., extracts and transforms the parameters, and calls the service contract object. For example, the transport could be IIOP (Internet Inter-Orb protocol) [11], which in a CORBA [11] implementation of Coyote would result in using CORBA services to call the SSCO; or the transport could be MQSeries[15], with a message adapter; or it could be EDI (Electronic Data Interchange) [3], on top of HTTP[16], requiring a message adapter. As a specific instance, the transport protocol specified in OBI (Open Buying on the Internet)[17], is EDI over HTTP.

From the message adapter, the SSCO is invoked. First, application-level security, as specified in the service contract, is enforced, including authentication and provisions for non-repudiation [18]. Following this, the aspects of the contract related to the action requested are enforced. This includes determining whether the client is allowed to invoke the requested function, or whether the request is valid at this point in the conversation between the client and this server. For instance, a cancel request is only valid after an order.

Next, the underlying (core) Coyote services, shown in Figure 3.1, are invoked; the request is time-stamped and logged for audit trail purposes, and for supporting queries of the state of a client-server conversation; it is then placed in an event queue. If the request is specified in the contract to be asynchronous, a (synchronous) acknowledgement is sent back to the client that the request has been received. When the request is processed from the event queue, the internal method corresponding to the action in the contract is invoked. This method may invoke existing applications, such as workflow processes, ERP applications or other back-end applications.

In order to satisfy the incoming request, the application may need to make requests to service providers, as illustrated in Figure 3.1. For instance, if the node is an agency (e.g., a travel agency) on the Web, outgoing requests to its service providers (e.g. airline, hotel, etc.) would need to be made. This is done using Coyote services and the CSCO. Rules are attached to the outbound service request, in order to specify what to do when responses come back from the service providers. For instance, an all-or-nothing rule specifies that a specific method at this node is to be called if all the requests to a set of service providers are successful; if any are unsuccessful, cancel requests would be sent to all other service providers that were successful in the initial request, and a specified failure method would be called at this node. Responses from service providers are placed in the event queue, the rules are evaluated and, if a rule fires, the appropriate method is invoked. In this manner, several outbound requests to service providers can be made in parallel or serially. Eventually, one of the responses specified in the service contract is returned to the client, corresponding to the original client request.

4. Coyote Architecture

In this section, we pull the pieces outlined in the previous sections together, and describe the overall Coyote architecture and components. Figure 4.1 illustrates the Coyote server and application components and how they fit together. The top of Figure 4.1 shows the two main elements in building a Coyote application, as outlined in Section 3: (i) the definition of the service contract, and (ii) the development of the business logic. Graphical tools are provided to assist in defining and developing these two elements. A contract template is provided, and the graphical tool assists in generating a specific contract from the template. As illustrated on the left side of Figure 4.1, from the resulting service contract, a code generation tool generates the Client-side and Server-Side Contract Objects (see also Section 3).

The contract and the business logic are registered independently with a Business Services Application (BSA) manager. The BSA manager provides various application and user services for a Coyote system. These services, shown in the middle of Figure 4.1, include application and user administration, start-up and termination, contract instance creation and management, home-base, which supports interaction with an end user, message handling, and the system monitor. They are described in Section 4.1.

The Coyote run-time services, shown at the bottom of Figure 4.1, are the core system services on which the higher-level functions are built. They include event handling, handling of asynchronous requests and replies, conversation management (i.e. correlation of inbound and outbound requests), compensation, thread management, and performance management and are described in Section 4.2.

The persistence and recovery component provides persistence of conversation state and recovery from failures of nodes and components. Conversation persistence is provided by time-stamping and logging incoming and outgoing requests and responses in stable storage. This log is used for providing responses to queries of conversation state, for audit trail purposes, and for re-creation of conversation state. It is replayed during recovery to re-create the states of the open conversations. Recovery also involves interaction with clients and service providers to receive pending responses for

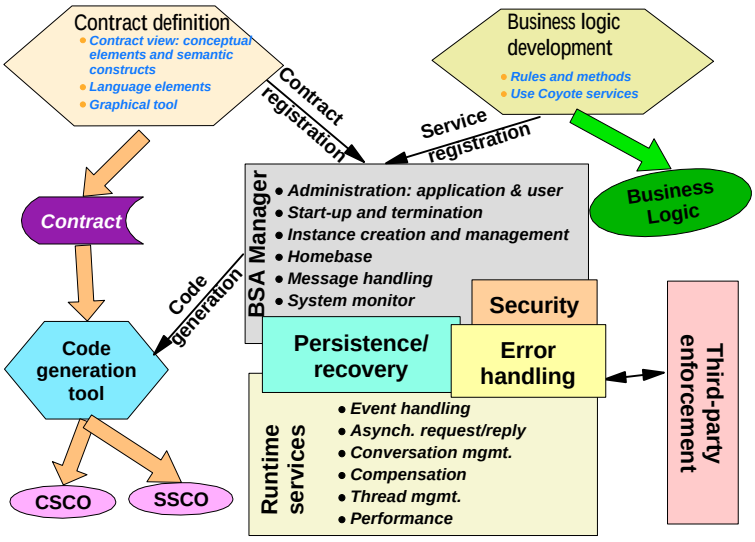

Figure 4.1: Coyote server environment

determining the status of ongoing actions, and to issue any cancellation actions which may be required.

The error-handling component handles time-out and other soft errors during run time (while persistence and recovery are focused on “hard” system failures). Error handling

is based on contract-defined rules as described in Section 2. As shown in Figure 4.1, error handling may involve the outside arbitrator in order to perform third-party enforcement when an error is likely to involve violation of the contract terms.

The security component handles security and authentication according to rules specified in the contract. It applies the required security to each outgoing message and tests each incoming message. The security and authentication rules have been discussed earlier (Section 2).

4.1 Coyote Server - BSA Manager

The BSA manager provides administrative services for a Coyote system. In this section we will outline some of the key functions of the BSA Manager.

Service registration: Once a Coyote application has been written it has to be announced to the BSA manager. This is done via a service registration tool. The service registration tool splits the business logic into actions and the user is prompted to associate replies, cancellation methods, etc. with each one of these actions.

Contract registration and automatic code generation: The contract, once written, is to be registered with the BSA managers of all the involved parties. This is done via a contract registration tool. This tool collects information for mapping actions to business logic methods from the administrator and then feeds this information along with the contract and the service registration information to the Automatic Code Generator, which in turn generates the contract objects (SSCO, CSCO or both).

User administration: This basically involves registering the client information with the BSA Manager. This information includes attributes like name, userid, public-key, contact information and privileges. This information is used at runtime for user authentication, callbacks etc.

Service instance creation and management: It is the job of the BSA Manager to create service instances for the clients who are using the services. This is *Server Service Instance Creation* and involves instantiating a BSA object whenever a client wants to start a conversation. When a BSA wants to utilize a remote service the BSA manager instantiates a CSCO object for it.

Startup and termination of the system: Startup involves bringing up dependencies like ORB[11] and Enterprise JavaBeans server and calling the recovery manager which in turn activates the “active conversation” objects. Termination involves gracefully passivating the conversation objects. Startup and termination are done via administrative screens.

Multiprotocol support: The BSA Manager has support for IIOP, MQ and HTTP. A custom protocol handler can easily be plugged in.

Home-base services: This is essentially a proxy for a web browser end-client, in

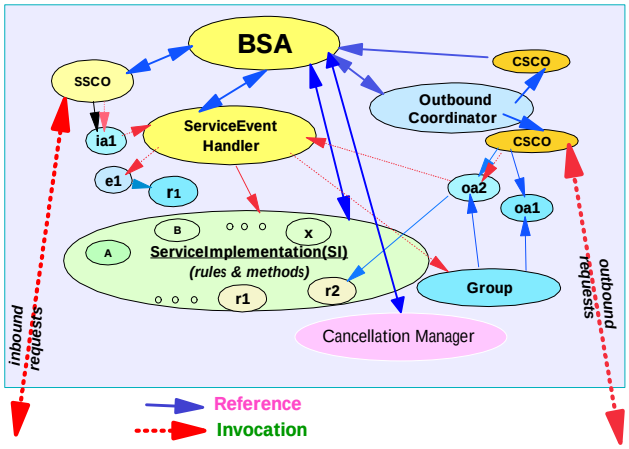

Figure 4.2: Coyote Runtime Services

order to maintain the state of a conversation with an end-client so that a client can reconnect and continue in a conversation where he left off. The BSA manager provides a series of conversation query functions.

System monitoring: The BSA manager has system monitoring functions which detect failures and invoke recovery actions. Tools are available for viewing and querying the trace information.

4.2 Coyote Run-time Services

To invoke services on a Coyote application server, the client (either an end user or an application) first needs to identify and authenticate itself with the BSA manager. Then the client requests the BSA manager to create a new BSA instance to start a new conversation or activate an existing BSA instance to resume a previously started conversation. Each BSA instance, depicted in Figure 4.2, is a version of the Coyote application server that has been customized to a particular client via a service contract. It is created specifically to serve that client and maintains the entire history of the client's conversation with the server until the conversation is terminated. Following are the main components or objects in each BSA instance, as shown in Figure 4.2, as well as the services that each offers:

- The BSA Implementation consists of a **Service Implementation** (SI) object and rule objects (r1 and r2). The SI embodies the business logic of the Coyote application server and contains methods to execute inbound service requests from clients as well as methods to handle responses from remote service providers. The rule objects are used to trigger appropriate methods in the SI to handle different outcomes of the outbound requests to the remote service providers. In addition, there may be application-specific objects (A and B). The BSA implementation is defined by the Coyote application developer. The remaining components discussed below are provided by the Coyote framework or generated from the service contracts.

- **A Service Event Handler** (SEH) is provided by the Coyote framework to support event driven asynchronous request/reply execution. An event is generated for each inbound request from clients or response from external service providers. Each event is attached with rules and delivered to an event queue. The events are processed serially by the SEH which tests each rule and fires an appropriate method in SI to handle the event.

- The **Outbound Coordinator Object** (OCO) is a Coyote framework object that provides many important services to the SI, including:
  - Supplying CSCO references to SI to invoke remote services;
  - mapping outbound requests to inbound requests;
  - creating outbound groups to help SI manage outbound requests;
  - providing a query interface to allow SI to find out state of outbound requests or groups.

- The **Cancellation Manager** (CM) is a Coyote framework object that offers cancellation services. The SI can use these services to cancel an inbound or outbound requests, outbound groups or an entire conversation.

- The **CSCO** is generated from the service contract between the Coyote application server and its remote service providers. It offers the following services to SI:
  - Time-stamping and logging all outbound requests and replies for audit trail;
  - passing service requests to the remote service providers;
  - responding to queries from the SI on outbound requests;
  - providing time-out services.

- The **SSCO** is generated from the service contract with a client and can differ from one BSA instance to another. It provides many services to the client, particularly:
  - enforcing the terms and conditions of the service contract by performing a variety of checking;
  - time-stamping and logging all requests and replies between the client and the server for audit trail purposes;
  - passing client requests to the SI by generating an event and delivering the event to the SEH;

- providing a query service to allow clients to find out the status of past requests.

5. Summary and Conclusions

In this paper we have described the Coyote concepts for multi-party electronic commerce, and outlined the design of the Coyote framework for defining and implementing inter-business deals on the Internet. A key concept in Coyote is the multi-party service contract, which specifies the rules of engagement between the parties, including the inter-business actions which can be performed, the responsiveness criteria for each action, error handling for time-outs and other conditions, the application level security required for each action, compensation rules for cancelling prior actions, and so on. We have described the creation of a Coyote application, which essentially consists of defining the service contract and business logic corresponding to actions in the contract; graphical tools are provided to assist in both these aspects of application development. From the higher level specification of the service contract(s), contract objects are automatically generated, which enforce the contract and invoke the corresponding business logic. We have outlined the Coyote framework, composed of a Business Services Application (BSA) manager, which is primarily for application and user administration, a run-time environment, which is an event-driven rule-based system, and other components for persistence, recovery, error handling and security.

We have implemented a prototype Coyote system. The prototype has a Java-CORBA based Coyote framework for most of the components outlined earlier, including the BSA manager, the run-time system, simple error handling, and automatic code generation from the service contract specification. Additional work is ongoing on recovery, security and third-party authentication. We are in the process of porting the framework to an Enterprise JavaBeans [14] environment. We have also put together several sample applications, including one similar to the example in Section 1, an OBI-like [17] flow, and tie-in to workflow and other sub-systems.

Our longer term objective in Coyote is that of spontaneous business-to-business electronic commerce. To this end, we are extending the Coyote concepts to registration of service contracts with trading services, negotiation of terms and conditions in the contract, followed by instantiation of the contract objects and tie-in to existing applications. Our Coyote prototype is a first step in this direction.

Acknowledgements: We would like to thank Francis Parr for his active participation at the early stage of the project in defining the concepts, Arun Iyengar for his participation in defining the security and third-party aspects of the Coyote framework, and Nagui Halim for his vision, support and encouragement.

References

1. A. Dan, and F. N. Parr, "*The Coyote Approach for Network Centric Business Service Applications: Conversational Service Transactions, a Monitor, and an Application Style*," ***HPTS Workshop***, Asilomar, CA, Sept. 1997.
2. A. Dan, and F. N. Parr, *"An Object implementation of Network Centric Business Service Applications: (NCBSAs): Conversational Service Transactions, a Service Monitor and an Application Style,"* ***OOPSLA Business Object Workshop***, Atlanta, GA, Sept. 1997.
3. ***Electronic Commerce Resource Guide***, *http://www.premenos.com*
4. *Customer Information Control System/ Enterprise Systems Architecture (**CICS/ESA**), IBM, 1991.*
5. *J. Gray, and A. Reuter ``Transaction Processing: Concepts and Techniques,'' Morgan Kaufmann Publishers, 1993.*
6. *H. Garcia-Molina, and K. Salem, ``SAGAS,'' Proc. of ACM SIGMOD Conf., 1987, pp. 249--259.*
7. *P. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz, ``Specifying and Enforcing Intertask Dependencies'', VLDB 1993, pp. 134-145.*
8. *G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, ``Advanced Transaction Models in Workflow Contexts'' In 12th ICDE, New Orleans, Lousiana, Feb. 1996.*
9. *Y. Breitbart, A. Deacon, H. Schek, A. Sheth and G. Weikum, ``Merging Application Centric and Data Centric Approaches to Support Transaction-Oriented Multi-System Workflows'', ACM SIGMOD Record, 22(3), Sept. 1993.*
10. *A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, K. Ramamritham, ``ASSET: A System for Supporting Extended Transactions'', Proc. ACM SIGMOD 1994, pp. 44-54.*
11. ***Object Management Group (OMG),*** *http://www.omg.org*
12. ***FlowMark***, *SBOF-8427-00, IBM Corp., 1996.*
13. ***XML 1.0 Proposed Recommendation***, *http://www.w3c.org.*
14. ***Enterprise JavaBeans Specification***, *http://www.javasoft.com/products/ejb*
15. ***MQSeries:*** *An Introduction to Messaging and Queuing, IBM publication no. G511-1908*
16. ***HTTP/1.1 Specification***, *http://www.w3c.org*
17. ***The Open Buying on the Internet (OBI) Consortium***, *http://www.openbuy.org/*
18. ***RSA Data Security***, *Frequently Asked Questions About Today's Cryptography 3.0, http://www.rsa.com/rsalabs/newfaq*