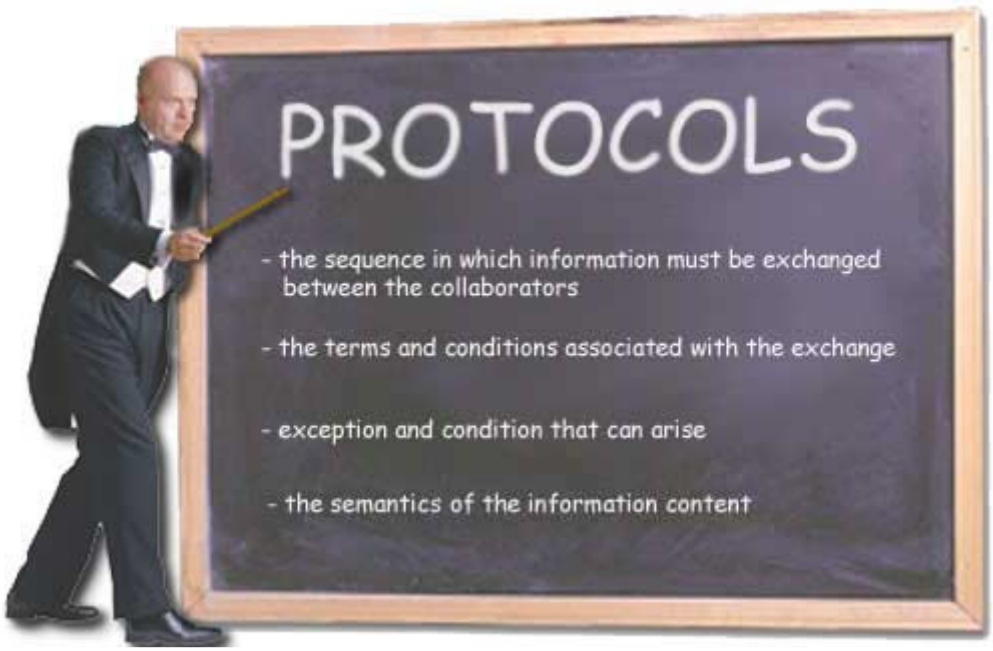


ObjectQ

Reference Guide

Information Design, Inc.
145 Durham Road, Suite 11
Madison, CT 06443
USA
Phone: 203-245-0772 x6212
Fax: 203-245-1885
Email: sales@idi-middleware.com

Contents

Figures

Tables

1 Introduction

The ObjectQ product provides a platform designed to make sending and receiving messages over a heterogeneous network transparent to both the sending and receiving applications. The combination of ObjectQ and a queue-based messaging tool may form the standard communication middleware for a whole enterprise.

Today's applications require constant interaction with other applications. Large resources are expended to define, develop, and maintain interfaces in an application. Often, interfaces between applications are tightly coupled; new features and functionality require intense coordination. One application may not be able to upgrade or change a feature without waiting for a second application to provide some enhancement.

ObjectQ is an off-the-shelf communication middleware product that provides communication between different applications and hides the low-level details of message transmission and message format from the actual application software. ObjectQ components form a middleware back plane that application developers can use as a set of standard services, plugging in elements of code as needed.

1.1 *What is ObjectQ?*

ObjectQ consists of a group of object-oriented C++ classes that provides a framework for building distributed applications. It is based on the premise that there are service providers with public interfaces that can be accessed by any application needing those services.

At its most fundamental level:

- ObjectQ provides a standardized message transport that makes an application independent of its underlying transport mechanism.
- ObjectQ gives applications the capability of gaining access to distributed objects through a transparent messaging layer.
- ObjectQ provides a standardized message format to permit de-coupling of client and server code.
- Services that have been converted to ObjectQ define their objects in a Management Information Base (MIB) available in table form on the World Wide Web.
- These tables put the service provided by an application into a succinct, easily understood form.

1.2 *Benefits of ObjectQ*

These are the most immediate benefits of ObjectQ:

- Reduces requirements, development, and testing effort. Using ObjectQ de-couples the development of front-end and back-end processes. There is less negotiation needed at the requirements stage since the middleware interface is an easily defined quantity.
- Shortens development and testing cycle times. Because of the object-oriented paradigm, code is more easily reusable. Developers don't have to reinvent the wheel each release and testers can focus on what's new in the product.
- Future growth is easier. Since applications are freed from worrying about issues inside other applications, planning for an existing service migration is simplified. Issues that are local to an application, such as capacity increases or hardware changes can be addressed locally.
- Supports communication modes other than client-server, e.g., peer-to-peer.
- Expertise and knowledge of applications is more easily transferred between projects since each application contains at least some subset of a shared common language.

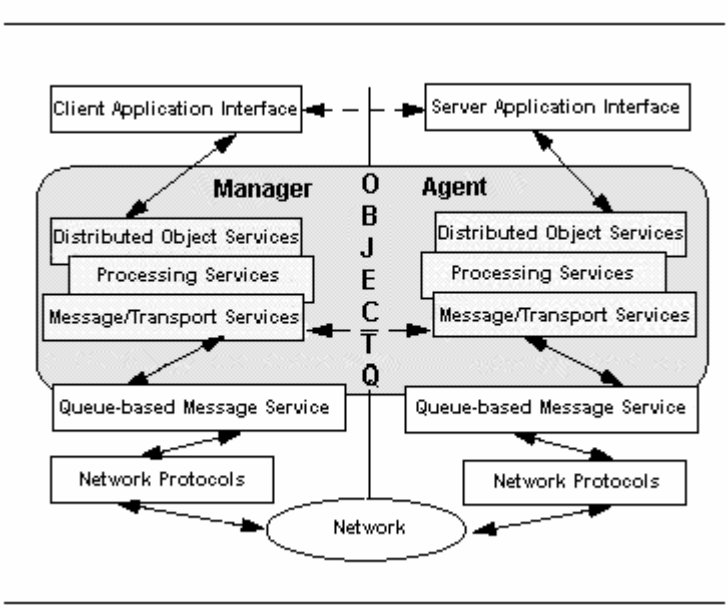
1.3 *ObjectQ on the World Wide Web*

ObjectQ's home page can be found at: http://www.idi-middleware.com/objectq/

Supporting information like manual pages, a copy of this guide, sample code, training schedules, a FAQ, and the software itself can also be found here.

2 ObjectQ Architectural Model

ObjectQ is a layered product designed to connect client processes and server processes. Clients send requests to servers that provide access to the distributed objects such as customer names, customer 800 numbers, equipment, calling services, facilities, and forwarding numbers. Each object can change its characteristics or state based on its attributes.



• Figure 1 - ObjectQ architecture and functional component layer

In order to execute a defined behavior for a distributed object, a client (using a manager) sends a request message to the server process (agent) which owns that instance of the distributed object.

The message protocol is based on the Common Management Information Protocol (CMIP). The data within messages for UNIX-to-UNIX communications is transported as attribute-value pairs; for UNIX-to-MVS communications, the data within messages is transported in a fixed-format APPC[1]-style transaction. A queue-based messaging product transports the messages over the network.

The architectural model (Figure 1) demonstrates the layering over networking protocols that ObjectQ provides. Also, ObjectQ classes can be thought of as being grouped into functional layers, each of which provides some distinct aspect of processing.

The shaded areas of (Figure 1) illustrate these ObjectQ functional layers:

[1] Advanced Peer to Peer Communications. The fixed-format used inside the message is defined through tables. See the UNIX-to-MVS Transport Mechanisms section below.

1) Distributed Object Services. A set of services that supports location independent access to distributed objects. The interactions supported include:
   - Create an instance of an object.
   - Delete one or more instances of an object.
   - Get attribute(s) of one or more objects.
   - Modify attribute(s) of one or more objects.
   - Subscribe to be notified if a specified event occurs.
   - Notify subscriber of a subscribed-to event.
   - Perform some action on one or more objects.

2) Processing Services. A set of processing services used in building an application. The services provided are:
   - Dispatch each message received to the appropriate object for processing.
   - Manage transport objects.

3) Message Manipulation and Transport Services**.** A set of services that facilitates the communication between a client and a server (a manager and an agent). The services provided are:
   - Transmission of messages between clients and servers, independent of the location of the clients and servers and of the underlying transport mechanism used to send and receive messages.
   - Construction of messages and extraction of fields from a message. Provide messaging using a transaction-based format or a CMIP format. Handle differences between machine architectures.
   - Manage service names, e.g., domain, class, attribute, etc., so uniqueness of individual names is assured.
   - Manage attribute definitions associated with distributed objects.
   - Administration messaging.

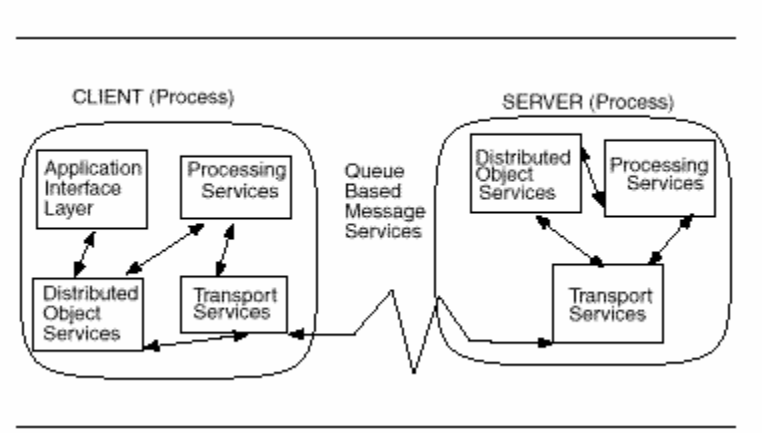
4) Queue Based Message Services**.** A vendor-provided queue-based messaging product supports inter-process communication over a network protocol, e.g., TCP/IP. ObjectQ's Transport class in the Message Manipulation and Transport Services layer extends into this layer to provide interaction with the selected platform.

(Table 1) lists the ObjectQ-relevant functional components and the DSAP-provided classes that make up the functional component. The relationship between the functional components during a simple client-server interaction is pictured in (Figure 2). Some other types of interactions (for example, clients with hybrid servers, clients requesting event notification, etc.) are described in Chapter 3, Communication Scenarios.

• Table 1 - Functional component responsibilities

Functional Component	Class	Responsibility
Distributed Object Services	Event Filter	Get/set event filter ID
		Get/set event filter attributes
	Generic Manager	Fully-functioning manager; can be used as base class for application-specific managers
	Instance Filter	Get/set instance filter ID
		Get/set instance filter attributes
	Manager/Agent	Create, delete, get, set objects and object attributes
		Perform operation on object(s)
		Subscribe to event on object(s)
		Notify subscriber of event occurrence on object(s)
		Service provider defined operations
	Manager Resource	Get/set manager parameters
Processing Services	Dispatcher	Add/delete service request/response, event, and administration message handlers to processes
		Timeout processing
		Dispatch message for processing
	Transport Manager	Add/delete transport object to available transports
		Get transport object
Message Manipulation Services	Attribute	Get/set attribute name value pair
	Message	Get/Set message components
Message Transport Services	Audit	Block of information regarding each hop that an envelope takes
	Audit Resource	Specifies which fields will be used in the audit data
	Envelope	Get/set transport parameters
		Get/add message object
	Resource	Get/set transmission parameters
	Selector	Get/set message selection parameters
	Transport	Send message, reply to messages, forward messages and receive messages

See Chapter 2, Introduction to ObjectQ Classes, for a discussion of the classes used during these processes. Chapter 3, Communication Scenarios describes how the classes interact.



• Figure 2 - Functional Component Relationships

2.1 *ObjectQ Basic Concepts*

Certain basic concepts must be defined before beginning a discussion of ObjectQ's architecture, classes, and their uses. See the Glossary for more definitions.

2.1.1 Distributed Object

Distributed objects that comprise a service consist of public attributes and methods that are available to any client process. Typically, a distributed object is an application-specific class that is used in the execution of a service.

An attribute of a distributed object is typically some variable or combination of variables, each having a type. A method is a procedure that can be triggered from outside to perform certain functions. A method can change the state of an object, update its attributes, or act on an outside resource to which the object has access.

The only way to access data inside an object is through the object's methods.

2.1.2 Service

A service provides access to an abstract collection of objects which, taken as a whole, provides some set of operations (i.e. methods) acting on or accessing a server's information. In ObjectQ terms, a service manages a class or collection of classes.

Operations that can be performed on a class are:

- Get — get a copy of data
- Set — modify data
- Create an instance
- Delete an instance

- Action— some other specific activity, e.g., subscribe or notify

A service is specified in terms of MIB tables. A client accesses a service by making a request of a manager which formats a message and sends that message to an agent.

2.1.3 MIB Tables

MIB tables are a service provider's specifications in the form of concise tables that provide class definitions and a data dictionary. MIB tables are meant to be used for the design of clients: they describe the operations supported by a service, the data available, the input data required, and the output data returned.

Any service that needs to be made available via ObjectQ must provide MIB tables for presentation on the Web. See Chapter 4, ObjectQ MIB Reference for more about MIB tables.

2.1.4 Domain

A domain is a namespace that is prefixed to service, class and attribute names as they are passed into constructors or otherwise used in ObjectQ-related code. The purpose of the domain name is to assure the uniqueness of names used for services, classes and attributes.

A domain name is used in two ways:

1) As a registered name, where the domain name forms the A.B of the numeric tuple A.B.C.D that is used to identify service, class, and attribute names within ObjectQ processes.
2) In strings used to identify service, class, and attribute names. The domain name always precedes the service, class, or attribute name with a period separator. For example, the attribute xyzId in domain abc is expressed as abc.xyzId.

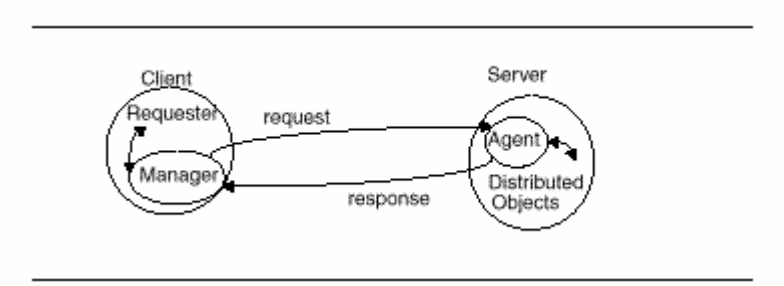
Domain registered names should be obtained from some centralized authority through a registration process. The table containing the domain names and registered names should be used throughout the enterprise and must not be changed.

2.1.5 Manager

A manager exists on a client where it is a surrogate for the service, providing access to methods and data of the distributed objects on the remote server. A manager is not a representation of the distributed object but is in fact a manager of a set of classes and their instances. A manager can perform actions on multiple classes simultaneously and on multiple instances of those classes. A manager uses filters to select the instances affected by the service request. For example, a service request that selects all tickets outstanding on a given circuit can be processed via the manager's `get` method.

A manager provides access to attributes for objects in a generic fashion. This generic form provides a specific list of attributes of interest, which are then retrieved by the agent on behalf of the manager. A mechanism to invoke remote functions is provided by an action method in the manager. These actions, the standard methods (`get`, `set`, etc.), the filters, and attributes

are defined for clients in MIB tables. A simple manager-agent relationship is shown in (Figure 3). ObjectQ provides a generic manager that can be used as the base class for an application-specific manager.



• Figure 3 - Simple client-server (or Manager-Agent) transaction

2.1.6 Agent

An agent exists on a service provider, where it is responsible for interpreting the messages sent by the manager. An agent cannot support more than one service and must handle all requests for that service. Upon receiving a message, an agent may perform the action itself or send the message along to another process designed to handle the specific request.

2.1.7 Attribute

An attribute is an inherent characteristic of an object, as defined in a MIB data dictionary.

Attributes can be read into ObjectQ programs through attribute definition (<*domain-name*>.*adf*) files. These files are ASCII files that contain information for each attribute that can be accessed through a service.

ObjectQ parses an attribute definition file for the process needing ObjectQ services and is used to instantiate an in-memory attribute definition table.

2.1.8 Client Process

In its most simple form, the client process makes requests of a server process.

2.1.9 Server Process

A server process performs the required function(s) and returns information to the client process. A pure server process cannot initiate an action. A server process can provide one or more services. Different forms of servers are described below.

- **Forwarding Server -** A forwarding server takes a request from a client process and passes the request along to a second server, which actually does the work. Typically, the forwarding server is a 'well-known host' that, for load-balancing or data-dependent reasons, is used to send messages to other servers. The second server responds directly to the original client.

- **Hybrid Server -** A hybrid server process receives requests and performs any actions necessary to satisfy requests. In order to satisfy a service request, a hybrid server process can send a service request to another server process, thus functioning as a client process. The second server does the work and responds to the client component of the hybrid server (not to the original client). The hybrid server performs any necessary actions and responds to the original client.
- **Conversational Server**- The conversational server is one that is in conversation with a client and must maintain state information depending on the needs of the client. One example of the use of a conversational server is to provide a database for a game where the database must be in a certain state depending on the client's last move. Through the conversational server mechanism, a specific client is in contact with a single server which provides the service until that client no longer needs it.
- **Subscription Server -** A subscription is a request by a client to be notified when a specified event occurs on one or more instances of a distributed object. When a subscription server receives notification that an event has occurred, it checks a list for matching subscriptions. For each match, an Event Report Request message is sent. Both the subscription request message and the Event Report Request may require the receiver to send a confirmation. The subscription server concept does not contravene the idea that a server cannot initiate an action since there must be an initial subscription request from a client.

2.1.10 Processing Modes

2.1.10.1 Synchronous Processing

In synchronous processing, requests are made to the server by the client. The client blocks, i.e., waits until the request is completed and the response is received, before performing any further processing. The client processes only one request at a time. See also Multi-threaded, below.

2.1.10.2 Asynchronous Processing

Asynchronous processing occurs when a client issues a request but does not block, that is, wait for completion of the request. Instead, after the request is made, other processing continues. If a response from the server is expected, the client will get it later when the server has completed and sent the response. The client has to learn of the existence of the response either by polling for it or receiving an event that indicates that the response is available.

2.1.10.2.1 Polling

If a client process is expecting a response from a server to an asynchronous request, the client process periodically checks to see if a response has been received. The client process chooses how often to check and what the time interval between checks for responses is. Server processes also poll, looking for incoming request messages.

2.1.10.2.2 Event

Instead of explicitly checking for a response to an asynchronous request, a client process may receive an event or signal that indicates that a response has been received. The client provides an event handling routine to process the response.

2.1.10.3 Single-threaded

Single-threaded processes have one thread of execution, i.e., once the execution thread starts, it continues until completion. This means, for example, that a server that is single-threaded can process only one request to completion before it starts processing the next request. Manager and agent objects are single-threaded.

2.1.10.4 Multi-threaded

A multi-threaded process can start more than one execution thread. In a multi-threaded environment, a server process can start an execution thread for each request it receives (within some finite limits specific to the operating environment) and work on more than one request at a time.

2.1.11 Message Transport Mechanisms

For UNIX-to-UNIX communications, the ObjectQ product provides a standard message structure derived from the CMIP standard. The short discussion of CMIS/CMIP and its message structure that follows provides general information about these standards and an understanding of how elements of CMIS/CMIP were used to formulate ObjectQ.

See UNIX-to-MVS Transport Mechanisms later for a brief discussion of transaction-based communications.

2.1.11.1 CMIS/CMIP

CMIP is a protocol defined by the International Standards Organization (ISO) that provides the rules governing the exchange of information and commands affecting managed objects and attributes between applications.

CMIP provides the mechanism for requesting a service and responding to the request.

CMIS defines a set of services that can be invoked by management processes so that remote communication can occur.

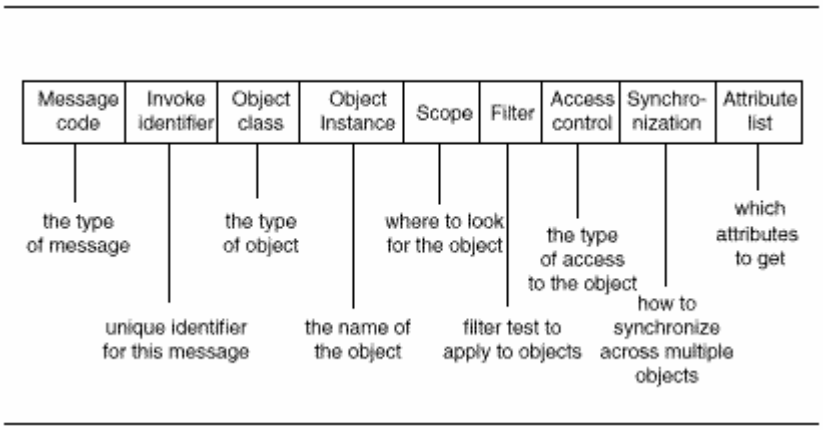
Communications in systems using CMIP consists of message exchange between managers and agents. Systems making requests behave as managers and systems responding to requests behave as agents. A system might act as a manager on one exchange and an agent on another exchange.

2.1.11.2 CMIP Messages

There are two basic types of information transfer exchanges:

- **Notifications** - These are messages that are subscribed to by other systems or applications. An example of this type of message is an alarm event notification. The CMIS service M-EVENT-REPORT is used to send a notification of some event to a remote system.
- **Operations** - These are management commands sent to an agent requesting that the agent system do something:
    - ***M-CREATE*** - create an instance of a managed object, e.g., Trouble Ticket or Equipment.
    - ***M-DELETE*** - destroy an instance of a managed object.
    - ***M-MODIFY*** - change the value of some attributes of the managed object.
    - ***M-GET*** - retrieve requested attributes for one or more managed objects.
    - ***M-SET*** - set one or more attributes of a managed object.
    - ***M-ACTION*** - request a remote system to take some action on the specified managed object.
    - ***M-CANCEL-GET*** - cancel a previously requested get operation.

Each CMIP message or *primitive* has a fixed header portion and a variable section.



• Figure 4 - The M-GET CMIP Message Structure

The message structure is slightly different for each of the different types of message, but a typical message (the M-GET command) is shown in (Figure 4).

Although it looks from (Figure 4) as though the message structure is simple, the individual fields can be very complex. Some fields may be simple integral values, while others may be implemented as linked lists or trees—the CMIP standard does not specify how data should be represented (the *syntax*), but only its meaning (the *semantics*). For example, the attribute list is

defined to be a sequence of attribute identifiers (which may themselves have a complex structure), and the sequence may actually be implemented as a linked list, an array, a set, or as any-thing else (as long as implementers agree on a particular *presentation syntax*, so that each can understand the other's messages).

The fields in (Figure 4) have a close correlation to the fields in the DSAP message which implements the get operation. These fields can be described in this way:

- ***Message code*** identifies which of the seven operations the message relates to.
- ***Invoke identifier*** is a unique identifier which allows a response to be linked, by the manager, to a corresponding request, in the event there is more than one request outstanding.
- ***Object class*** indicates the type of object, and takes one of two forms—either a simple integer (the local form) or an object identifier (the global form). An object identifier is a list of integers that uniquely identify any object in a protocol (e.g., an X.400 body part, a DFR document type, a X.500 attribute type, a public key algorithm, a protocol version, etc.), and represents a path through a naming tree. If one object identifier (OID) is assigned to a particular user, then only that user is allowed to append further numbers to the OID to create new object identifiers. The user may give some of the newly created OIDs to other people, and then they also have the right to create their own new sub-trees in the OID space. The highest levels in this tree belong to ISO and to other standards bodies. This allows a user, for instance, to define his or her own X.400 high-end cyberspace body part, and it won't collide with someone else's self-defined body part. Because many options in OSI protocols are identified by OIDs, it is very easy for implementers to extend the protocols without conflicting with older implementations to which this new OID will be unknown. Many protocols exchange sets of OIDs that identify their implemented subsets and extensions after the connection has been established and automatically determine the largest common subset of all optional protocol features that both may use. Thus, for example, the OID { 1 0 8571 5 1 } identifies the object found by starting at the root of the naming tree, moving down the tree to the subordinate node with label 1 (iso), then moving further down the tree to the subordinate node with label 0 (standard), and so on.
- ***Object instance*** is the name of the object, qualified to unambiguously identify it. In the global form, a complete path to the object in global namespace is given (e.g., US.att.nj.mt.dacs123.28); in the local form, enough of the path is given to identify the object within the current context. The global form is called the distinguished name, and the local form is called the relative distinguished name.
- ***Scope*** restricts the portion of the namespace that is searched for the object, typically either to just the object itself, or to the sub-tree rooted at the object. In this way, it is possible for the requested service to be applied to more than one object by specifying the object instance as an incomplete path (e.g., US.att.nj), and specifying a 'whole sub-tree' scope.

- ***Filter*** gives finer control over the objects selected, by specifying a Boolean expression that is applied to each of the potentially selectable objects—objects satisfying the test will be selected, and others will be rejected. The Boolean expression (of arbitrary complexity) typically checks the object's attributes for existence, or equivalence to some value, or relative to some threshold.
- ***Access control*** may be used for security purposes. Its exact usage is not specified, but it could contain, for example, an authorization key.
- ***Synchronization*** allows for 'two-phase commit' across all the objects selected—it is possible to specify that the requested service should be applied only if it can be successfully applied to all selected objects (atomic), or to as many as possible (best effort).
- ***Attribute list*** is typically a list of name-value pairs, where the name specifies a particular attribute (often in terms of an object identifier), and the value is a particular value associated with that attribute. This is a much more powerful concept than using, say, a C-type structure, because it is possible to add or delete attributes within a managed object without all managing systems having to modify their code (providing, of course, that the code is written defensively).

Each CMIS service has a defined message structure and associated parameters. ObjectQ's message structures, as well as the concepts used in instance identifiers, filters, and attributes follow the CMIP standard to a great extent.

2.1.11.3 ObjectQ Request Messages

Conforming to CMIP and CMIS, an ObjectQ request message for UNIX-to-UNIX communications that accesses a service, i.e., goes from a client to a server, uses these methods:

- ***Get*** is used to retrieve attribute(s) from one or more distributed object classes, e.g., get all 800 numbers mapping to numbers in area code 203 and return the date of creation of the service and who is the contact and the customer address. Instance selection criteria are passed in an instance filter. Attribute selection criteria are passed in an Attribute Identifier List.
- ***Set*** is used to modify or add attribute(s) from one or more distributed object classes, e.g., modify attribute custName to Howard Johnson for custNames having custName HoJo. Instance selection criteria are passed in an instance filter. Attribute types, set operations, and values are contained in an attribute list.
- ***Create*** is used to create a single instance of a distributed object class.
- ***Delete*** is used to destroy one or more instances of a distributed object class, e.g., delete all customers who have not paid their bills in six months. Instance selection criteria are passed in an instance filter.
- ***Action*** is used to carry out some remote procedure on one or more instances of a distributed object class, e.g. do quick test on circuits in office Walnut Creek and return the number of tries attribute; create a subscription to an event occurring

during service-providing activities. Instance selection criteria are passed in an instance filter. Parameters used by the remote procedure are passed in an attribute list. Results of the action are requested via an attribute list.

- ***Event*** used to notify a client that the particular thing it has subscribed to has occurred, e.g., contact name of a customer object has changed. Events are coupled with subscriptions (subscriptions are actions) subscribed to in advance by a client. Attributes of interest or those that have changed are returned in an attribute list.

2.1.11.4 ObjectQ Response Messages

Response messages are usually messages from a server to a client in response to a request; also, an event report message from a server can be considered a response message since the subscription must have been initiated by a client before the subscribed-to event happens.

If a request message is sent with confirm mode enabled, a corresponding response message must be returned to the originator of the request. Get, create and delete messages are always confirmed.

All response messages contain an error field in case process problems are encountered. All response messages (with the exception of delete), return a time and an attribute list. If multiple instances of an object are affected by a request, a separate message is returned to the originator for each instance.

2.1.12 Queue-Based Message Services Concepts

The developer using ObjectQ does not have to know the low-level details of queue-based messaging; ObjectQ's Message Manipulation and Transport Layer masks these details from the developer and the application. However, it's useful for developers to have at least a high-level understanding of this type of messaging before using these classes.

Message queuing allows programs to send messages by directing the messages to a memory- or disk-based queue as an intermediate storage point. The queue stores the messages until the receiving program can process them. Some of the major advantages of this approach are clear:

- Programs can execute independently— they don't have to wait for the message to 'arrive' before continuing (unless they are designed to). That is, messages can be processed asynchronously unless synchronous activity is needed.
- Programs don't have to know where servers or other clients are — the messaging software takes care of locating message destinations based on the service needed.
- Programs can also control the queues, setting up different types of queues as needed, and attaching priorities to messages as needed.
- Clients can send requests to a server's queue even if the server process is not currently running.
- In the event of some network failure, if guaranteed message delivery is requested, messages can be recovered for retransmission from the queues automatically, without any application intervention, once links are re-established.

2.1.12.1 Queuing Bus

Some queue-based message vendors, e.g., BEA MessageQ, use a logical queuing bus as a data highway for transferring messages between processes. Once a process is attached to the queuing bus (ObjectQ takes care of attaching processes), the process can send messages to any other process attached to the same queuing bus (with some possible restrictions), whether that process is on the same physical machine or anywhere on the system. Using the queuing bus means that one application does not have to know the details of where a second application resides on the system; nor does it have to know how to establish contact with the second application.

Queue-based message buses also allow separation of applications into non-interacting domains, for security or other reasons. For example, a test environment can run in parallel but in isolation from the production environment.

2.1.12.2 Message Queues

Message queues are administered through the queue-based messaging vendor's administrative interface.

Once a message is read from a queue, it no longer exists on the queue. Messages are read in either FIFO (first-in/first-out) order or through selection criteria, e.g., priority.

With ObjectQ, the server must attach to a primary named queue for its service to be registered, and will typically block (wait for requests).

The client usually does not use a named queue for sending messages and receiving a reply. Since a client sends a request message, the client does not have to have a known address or be associated with a named queue: the address (or queue) used in the response message to the original request is built into the original request message.

2.2 *UNIX-to-MVS Transport Mechanisms*

ObjectQ supports communications, which use MVS systems as servers. Designers and developers working on these systems must use transaction-based processing and must then decide which communications methodology will be used to send envelopes. These possibilities exist:

1) Transactions can be passed through ObjectQ Direct, an interface that uses IBM's MQSeries as the underlying transport, communicating via TCP/IP directly to the MQSeries process running on the MVS machine.
2) Transactions are sent through an LU6.2 Port Server that acts as a gateway between BEA MessageQ and the LU6.2 protocol 2 . In the context of BEA MessageQ, a port server is a class of the BEA MessageQ application that provides a connection to the BEA MessageQ queuing bus for client applications communicating with platforms that do not have a BEA MessageQ implementation.

3) QmsgBridge, a BEA MessageQ to MQSeries gateway which is not tied to any particular hardware, allows a process using BEA MessageQ to communicate with a process using MQSeries. This third alternative is the preferred long-term solution.

2.2.1 UNIX-to-MVS Transaction-Based Processing

Two message types (transaction receive/response) are used in UNIX-to-MVS communication. These message types contain header data that allows the transport mechanisms and MVS's IMS systems to recognize and process the transaction correctly. In IMS processing, there is no agent software processing all the envelopes for a service. Instead, each transaction is handled by an appropriate transaction handler in a designated IMS region.

Instead of attribute-value pairs being sent back and forth in messages, transaction-based processing requires that fixed-fielded data be transported. Transaction formats are defined by transaction definition (*.xdf*) files.

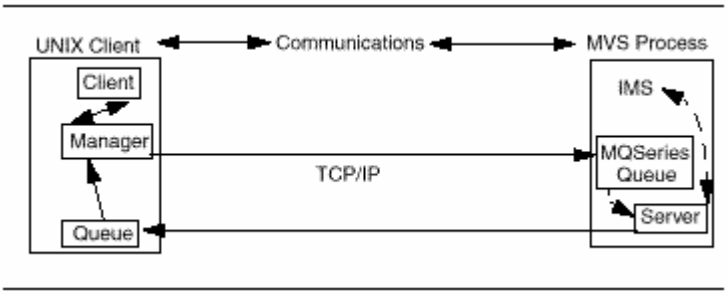
Essentially, these transactions are preformatted data, shaped into the requisite order for processing on MVS.

For more detail on these transaction-based classes, see Chapter 2, Introduction to ObjectQ Classes.

2.2.2 UNIX to MVS Through ObjectQ Direct

ObjectQ Direct uses the MQSeries transport class to provide the necessary transport functionality. This class is a subclass of the ObjectQ transport class.

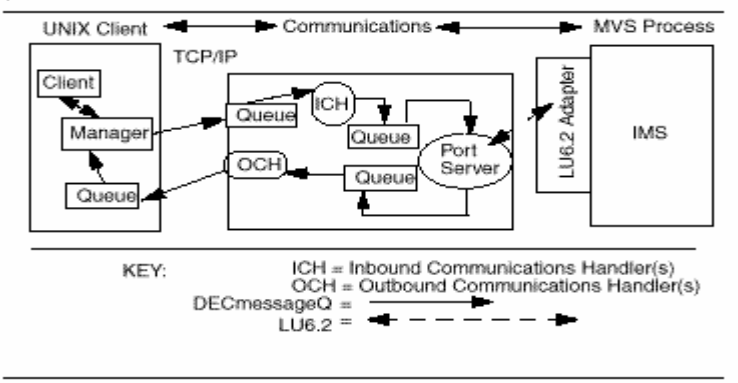
In ObjectQ Direct, the transaction request/response message types must be used and the transactions preformatted on UNIX before they are sent to the IMS sys-tem.



• Figure 5 - UNIX—MVS communication through ObjectQ Direct

This process works the same way as the UNIX-to-UNIX process: the server registers a queue in the name of a service. In this case, the MQSeries queue is used as the destination of envelopes sent to that service.

2.2.3 UNIX-to-MVS Through the LU6.2 Protocol



• Figure 6 - BEA MessageQ—LU6.2 communications

A Communications Handler (a process developed by Digital Equipment Corp according to requirements specified by a customer) can also be used to route messages to the correct queue.

There is one queue (or set of queues) for inbound messages and one queue (or set of queues) for outbound messages, as shown in Figure 6. When the Communications Handler determines that the message is directed to an MVS process, it sends the envelope to a queue associated with the LU6.2 port server, which sends the envelope to the MVS/IMS process.

The LU6.2 port server is off-the-shelf software available from DEC. When the port server is brought up, it establishes sessions with MVS systems.

The inbound Communications Handler(s) and the outbound Communications Handler(s) are each half-duplex with the consequence that the manager handles these transactions asynchronously. This model is scalable: Communications Handlers can be replicated depending on message volume.

From the client application's point of view, while knowledge that the envelope is going to an MVS process is assumed, the transportation method appears to be ordinary BEA MessageQ. No changes to manager code are needed to accommodate the DEC port server or LU6.2 adapter.

2.3 *Administration Messaging*

Administrative messages can be used for any purpose. Generally, administration messages are used for non-service related activities, examples of these activities may be inferred from these command names: `quiesce`, `change-tracelevel`, `reread-datafiles`, `echo`.

Service providers incorporate the ObjectQ administrative manager and agent into their own client and server processes.

Administrative messages are described in their own MIB template, which requires unique naming of each message as well as a listing of the input/output attributes and the conditions under which the message should be confirmed (always, never or mode dependent). A blank Administration Message MIB table can be seen in Chapter 4, ObjectQ MIB Reference.

The only currently existing ObjectQ administrative message is `DSAP.echo` which checks:

- the viability of a service
- the viability of a process reading a specified queue
- the viability of the BEA MessageQ group

3 Introduction to ObjectQ Classes

This chapter begins the discussion of ObjectQ's functional components by decomposing them into their major object-oriented C++ language classes. The general purpose and requirements of each class as well as some interactions among the classes are discussed. This chapter is meant to provide an introduction to the classes. Supplementing the material here are:

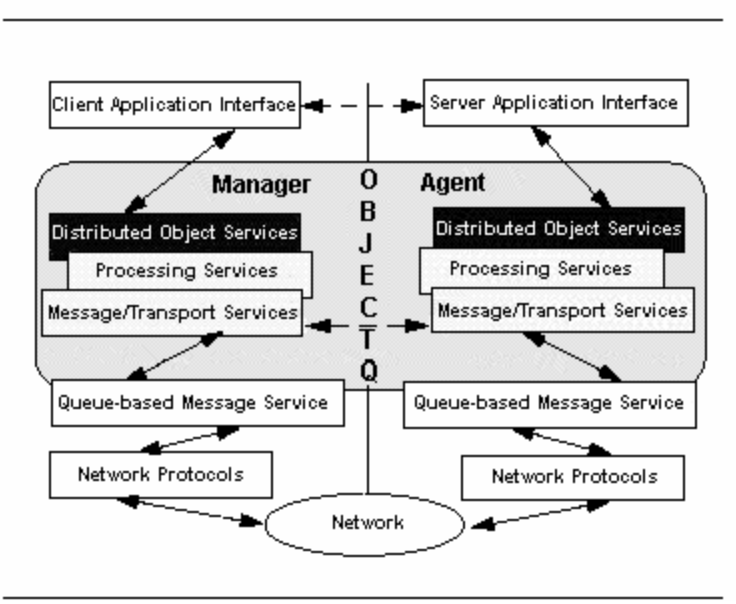
Manual pages for these classes - Can be found on the ObjectQ web page. See Chapter 1, Introduction, for the address.

Class header files **ObjectQ** - header files can be downloaded from the ObjectQ web page.

In this Reference Guide - Chapter 3, Communication Scenarios gives a further view of class usage. Chapter 3 also includes many event trace figures that list the functions used in a class at a given moment in processing.

3.1 *Distributed Object Services*

Distributed Object Services support location independent access to objects. Some examples of distributed objects are individual customer records, 800 number records, and trouble tickets.

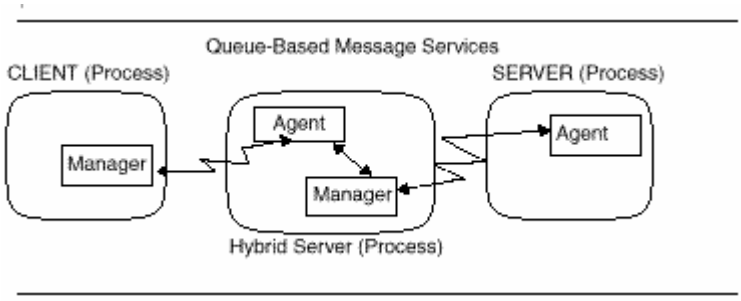


• Figure 7 - Distributed Object Services layer

A distributed object consists of a set of attributes and a set of methods performed by/on the object. Figure 7 shows the Distributed Object Services layer in the general architectural model. The definition of the object is done in terms of MIB tables. These are described in Chapter 4, ObjectQ MIB Reference.

Distributed Object Services are provided by two paired classes, the cpManager class and the cpAgent class. Both classes must be written and made available by the service provider (although a generic manager is provided for a client to use that allows the service provider to abdicate that responsibility — see later).

The cpManager class gives an application access to objects not owned by the application (distributed objects). These distributed objects can exist anywhere on the network. A cpManager class object sends a message to a cpAgent class object which provides the object services. The location of the agent providing the services is transparent to the manager.



• Figure 8 - Hybrid Server with Manager-Agent interaction

An agent which needs to perform an operation on an object it does not own uses a cpManager object, in a recursive fashion. A server process (see Figure 8) that acts as both a server process and a client process is referred to as a 'hybrid' server.

3.2 *cpAgent Class*

The cpAgent class is an abstract base class. A cpAgent class object is the receiver of service requests from the service provider manager. Each cpAgent class object must be paired with a corresponding cpManager class object.

3.2.1 Purpose of cpAgent Class

The cpAgent class is intended to act as the parent of service provider-specific agent classes. Each service provider agent class provides distributed object services on one or more classes of objects.

The cpAgent class object may perform all or part of the operations itself, use a manager object to access other distributed objects, forward the request to another agent object for processing, or implement any other application-specific operations or communications to service the request. When the agent has completed the request, it sends one or more responses back to the manager which made the request.

Each agent must work on only one service request at a time. This simplifies the implementation of service provider agent objects. If an agent becomes a client of another service provider, issuing an asynchronous service request, it is not available for a new service request while it is waiting for a response.

A server process may have more than one agent providing the same service. In a hybrid server, this increases throughput. Multiple agents in a server provide threading without the need for lightweight thread support from the operating system.

3.2.2 Basic cpAgent Class Operations

All agent classes perform these same basic operations:

- Register the service with the dispatcher object.
- Use a message object to extract data from a request message or construct a response message.
- Use an envelope object to package messages and extract messages.
- Use a transport object to send responses or forward messages.
- Use a manager object if access is needed to other distributed objects not managed by the agent.

3.2.3 The requestHandler *Function*

The agent's *requestHandler* function is a virtual function. A *requestHandler* function is used as a destination by the dispatcher (a cpDispatcher class object) whenever some request arrives. A *requestHandler* function must be provided with each service-specific agent object.

3.2.4 The notifyPartial and notifyComplete Functions

These functions must be provided as part of an agent on a hybrid server. On a hybrid server, the cpAgent class object instantiates a cpManager object in order to get data or perform some action on another server asynchronously. When the request made of the second server is complete, the appropriate function is executed.

3.3 *Administration Messages*

By incorporating the ObjectQ administrative agent class into its process, a server process can handle process-level requests. When the administrative agent receives a message of type *cpMSG_ADMIN_REQ*, it may respond by building and replying with a message of type *cpMSG_ADMIN_RSP*.

3.3.1 cpEventFilter Class

The cpEventFilter class is used for subscription requests. It specifies an event being subscribed to. The cpEventFilter class consists of an event filter ID and an optional list of attributes. Event filter IDs are defined as part of a service's MIB tables. Each event filter ID may or may not specify an operation (EQUAL, GREATER THAN, or LESS THAN).

Through the use of save and restore functions, event filters can be saved to persistent storage, where they are represented as octet strings, and later restored.

3.3.2 cpGenericMgr Class

The cpGenericMgr class is a convenient way to access basic cpManager class functionality. As provided, the cpGenericMgr is a fully functioning manager which provides access to the methods of get, set, create, delete, action; event report request message handling is also supported.

The cpGenericMgr can be used as a programming model for an application's own manager, or as the base class for an application-specific manager. In the latter case, only application-specific specializations need to be added to the basic get, set, etc. functionality.

The cpGenericMgr class addresses these cpManager class issues in the described way:

- **States** - three states are maintained the cpGenericMgr:
  - ***cpIDLE***—not processing a request, available for new request
  - ***cpACTIVE***—has begun request processing. At this point, no other requests except createCreateRequest can begin
  - ***cpWAITING***—has sent a request and is waiting for a response. No other **requests can be processing in this state.**
- ***Event Notification*** - the cpGenericMgr's eventHandler function calls utility functions which
  - unpack the envelope
  - extract cpAudit class data from the envelope and add it to the list of audit data
  - execute the notifyEvent function on the requester object.

3.4 *The* subRequest *Function*

In the cpGenericMgr, once the get, set, create, delete, and action requests are built, the *subRequest* function is called. The *subRequest* function does the work of sending the request, either synchronously or asynchronously, and then calling appropriate response handling functions when the reply arrives.

These activities occur in the *subRequest* function:

- Send the request, either to the service, or to the conversational handle.
- Depending on the mode, wait for a response, or terminate processing.
- If a reply is expected and processing is synchronous, call the cpManager's *rcvSynchResponse* function, and then the correct *process. Response* function.
- If a reply is expected and processing is asynchronous, register the message with the dispatcher. When a reply arrives, the correct *process...Response* function is

called.

3.4.1 *The* process...Response *Functions*

The cpGenericMgr class provides basic *process...Response* functions for get, set, create, delete, and action. An application can provide its own functionality or use these functions.

When one of these functions is called, it performs these tasks:

- unpacks the envelope; unpacked messages are added to the cpManager's message list.
- gets the cpAudit data from the envelope and adds it to the list of audit data.
- executes either notifyPartial or notifyComplete depending on whether this envelope is the last envelope, and if processing is asynchronous.
- if processing is asynchronous, unregister the message with the dispatcher.

3.4.2 *cpInstanceFilter Class*

The cpInstanceFilter class is used to specify criteria for object instance specification in request messages. The cpInstanceId class specifies a single object, while the cpInstanceFilter specifies a range.

For example, on a get trouble ticket request, an InstanceFilter object might include a set of work center IDs to specify which trouble ticket objects the get applies to.

The cpInstanceFilter class consists of a filter ID and an optional list of attributes. Filter IDs are defined as part of a service's MIB tables.

Through the use of save and restore functions, instance filters can be saved to persistent storage, where they are represented as octet strings, and later restored.

3.4.3 *cpInstanceId Class*

Each instance of a distributed object must have a unique identifier. This unique identifier must also have a string representation so that it can be flattened and unflattened in messages. See "Flattening and Unflattening" in the cpMessage section below. A base class is provided.

3.4.3.1 Purpose of the cpInstanceId Class

In Table 2-4 the attribute custId is the InstanceId[2] for this class. The function *instanceName* (of the cpInstanceId class) gets and sets the instanceId using a string argument for the InstanceId.

A class hierarchy of cpInstanceId classes is supported. At the top of the tree is the default cpInstanceId class which contains an unstructured string. Any service provider managing a

[2] In Attribute Definition Tables there is no identification of an instanceId. InstanceIds are identified as such in the accessible attributes sections of class MIB tables. See Chapter 4, DSAP MIB Reference for more information on identifying the instanceId.

class of distributed objects needing structured instance identifiers must provide a cpInstanceId class specializing from this default cpInstanceId class.

3.4.4 cpManager Class

The cpManager class is an abstract base class. Service provider manager classes provide the public interface to the service. These service provider manager classes are subclasses of the DSAP cpManager class.

The cpGenericMgr class is provided as a way to access the basic functionality of a manager and also to be used as the base class for an application-specific manager.

3.4.4.1 Purpose of the cpManager Class

If an application needs to perform an operation on one or more distributed objects, such as a user interface object needing data on trouble tickets, it requests the data from a service provider manager which manages that class of distributed objects. The manager sends the request to an agent, somewhere on the network, which has the responsibility for those services. When the agent has completed processing that request, it sends the response back to the manager. The manager then notifies the requester that it has completed the request.

The cpManager class is intended to act as the parent of service provider-specific manager classes. Each service provider manager class provides access to one or more classes of distributed objects as defined in the MIB for that service.

Each manager works on only one request at a time. If an application needs con-current processing of requests sent to the same service, a separate manager must be instantiated for each request.

3.4.4.2 Basic cpManager Class Operations

All manager classes perform these same basic operations:

- Use a message object to construct a request message and extract data from a response message.
- Use an envelope object to package messages and extract messages.
- Use a cpTransportManager class object to get a transport object.
- Use a transport object to send envelopes to agent objects.
- For asynchronous messages, register the message with the dispatcher object when the message is sent and unregister the message when no more responses are expected.
- For asynchronous requests, notify the requesting object when the request has been satisfied.
- For subscriptions, register itself as the event handler with the dispatcher object.

3.4.5 Managers and Functional Units

Each service provider manager class may provide supplementary functional units as convenience functions to make access to distributed objects easier for application builders. A functional unit is a collection of one or more of the basic operations of a manager bundled into a predefined function(s). Some examples of functional units might be *update_ticket*, *close_ticket*, *adjust_billing_record*, *run_test_26*, etc.

3.4.5.1 Administration Messages

By incorporating the ObjectQ administrative manager class, a client process can send and receive messages of types *cpMSG_ADMIN_REQ* or *cpMSG_ADMIN_RSP*.

These request/response messages can maintain an application; for example, a manager can be used to make certain that an application resource is present before sending a request which accesses that resource. ObjectQ provides some capability in this area: the DSAP.echo message allows a manager to check the viability of a service through its service name, the viability of a process through its queue name, or the viability of a BEA MessageQ group through its group name.

3.4.5.2 Subscription and Event Notification

Subscription and event notification are handled as follows.

1. An application that needs notification that an event has occurred on one or more instances of a distributed object uses the subscription operation on a service provider's manager.
2. The manager sends the subscription request to its associated agent. The request message uses an event filter ID to specify the event, with an optional set of event attributes, and a filter which specifies the instances of the distributed object.
3. When a subscribed-to event occurs, an event notification message is sent to the manager.
4. An application terminates the subscription by unsubscribing through the manager when it no longer needs event notification.

The specifics on how subscription services are managed, such as what events can be subscribed to, how long a subscription is in effect, etc. are defined by each service provider in its MIB.

3.4.6 cpManagerResource Class

The cpManagerResource class is used to specify cpResource class parameters for message transmission and also to control the behavior of the manager.

3.4.6.1 Purpose of the cpManagerResource Class

All the cpResource class object parameters can be set in the cpManagerRe-source class object. The cpResource class specifies the characteristics of transmission for both cpMessage objects and cpManager class objects. In a manager, the cpManagerResource class can be used to state

the queue at which responses will arrive, also, if a message should be sent synchronously or not.

3.4.6.2 Parameters in the cpManagerResource Class

The parameters listed below can be set in the cpManagerResource object; see the manual page for additional parameters:

- ***Total Timeout*** The timeout value, in seconds, from the time a request is made to a manager until the requester is notified that the request has been satisfied, either partially (if the partial data notify flag is set to cpTRUE) or completely.
- ***Partial Data Notify*** A flag which indicates whether the requester wants to be notified every time a response is received by the manager. This is only meaningful if a large amount of data has been requested — this could result in multiple transmissions of data. If this flag is set to cpTRUE, the manager executes the requester's notifyPartial function after every envelope is received.
- ***Synch Message*** A flag which indicates whether the manager object should process the request synchronously (not return to the calling routine until the request has been satisfied) or asynchronously. If the request is processed asynchronously, the requester is notified when the request has been satisfied by the execution of the notifyPartial or notifyComplete operation on the requester object.
- ***Maximum Messages*** The maximum number of response messages a requester wants. It is added to the request message by the manager and sent to the agent.
- ***Message Packing*** The desired number of response messages in an envelope. It is added to the request message by the manager and sent to the agent.

3.5 *Distributed Object Services Summary*

Table 2 provides a quick reference for these classes and their responsibilities.

• Table 2 - Distributed Object Services Summary

Functional Component	Class	Responsibility
Distributed Object Services	Event Filter	Get/set event filter ID
		Get/set event filter attributes
	Generic Manager	Fully-functioning manager; can be used as base class for application-specific managers
	Instance Filter	Get/set instance filter ID
		Get/set instance filter attributes
	Manager/Agent	Create, delete, get, set objects and object attributes
		Perform operation on object(s)
		Subscribe to event on object(s)

Functional Component	Class	Responsibility
		Notify subscriber of event occurrence on object(s)
		Service provider defined operations
	Manager Resource	Get/set manager parameters

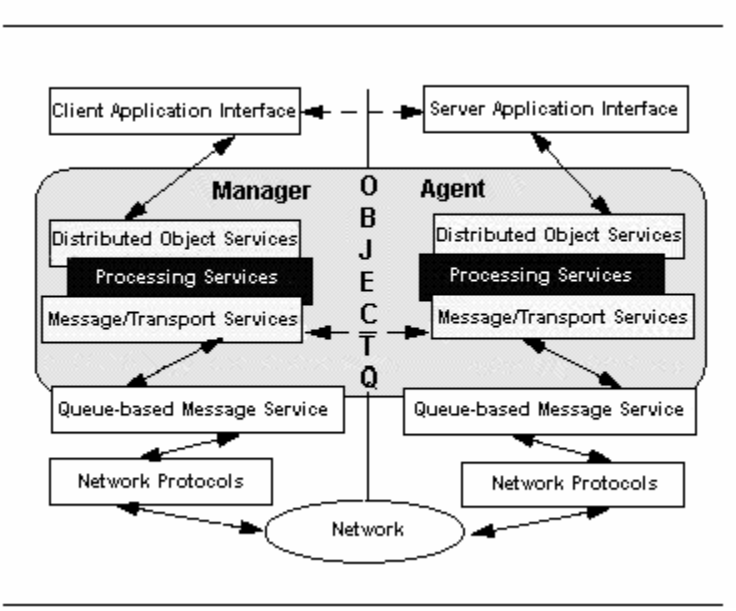
3.5.1 Processing Services

Processing Services are a collection of services that assist in building processes that incorporate the ObjectQ platform. There are two classes provided, the cpDispatcher class and the cpTransportManager class.

The cpDispatch class can be thought of as providing message routing internally, on a client or server. The cpTransportManager class manages transport objects, but does not provide any transport service itself.

3.5.2 cpDispatcher Class

The major function of the cpDispatcher class is to dispatch an incoming envelope to the object that will process the message(s) inside the envelope. There are four types of incoming messages: service requests, responses to requests, administration requests, and event notifications. The dispatcher is designed to handle all four.



• Figure 9 - Processing Services Layer

3.5.3 Basic cpDispatcher Class Functions

A dispatcher sends an envelope to the object that will process it. A dispatcher can dispatch

- a service request to a service provider (an agent object),

- an asynchronous response to a client (through a manager object),
- an administration message to the object which has registered to handle that message,
- or an event notification to a event handler (manager object).

The ***cpDispatcher*** class supports asynchronous client requests in client processes and requests to service providers in server processes.

The ***cpDispatcher*** class registers/unregisters servers, messages, administration handlers, and events. This class also tracks the status of agents as being busy or not busy.

Once an envelope is received, the dispatcher examines the envelope and handles these types of incoming messages:

- ***Request Message -*** Using the service name in the envelope, the dispatcher executes the *requestHandler* function of an available object that has registered to provide that service.
- ***Response Message -*** Using the invokeId in the envelope, the dispatcher executes the *responseHandler* function of the object which has registered that invokeId.
- ***Admin Message -*** For administration request messages, sends the message to the object registered to handle that request; response messages are handled as described above.
- ***Event Report Request Message -*** Using a subscriptionId and the service name, the dispatcher executes the *eventHandler* function of the object which has registered as an event handler for that subscriptionId and service.

3.5.4 Service Provider Support

In support of service providers, the dispatcher maintains a list of service providers that contains the following information:

- Name of service.
- Address of object (that is, the agent) providing the service.
- Status of object providing the service (busy or not busy).

In response to a query of this list, the dispatcher can indicate how many agents for a given service are available. When an actual service is call, if all agents for that service are busy, the dispatcher returns an error when called.

3.5.5 Asynchronous Message Support

In support of asynchronous message processing, the dispatcher maintains a list of outstanding messages. The list contains the following information:

- *InvokeId*— unique request/response identifier.

- *Address of object* waiting for the result.
- *Time-out value* (at the application level).

When the dispatcher object receives a response message, it determines which object should receive the message, based on the invokeId, and executes the response handling function of that object. Each object that receives responses asynchronously must have a response handling function.

3.5.6 Event Notification Support

For event notification processing, the dispatcher maintains a list of event handling objects. The list contains the following information:

- Service name and subscriptionId.
- Address of object providing event notification handling

If the message is an Event Report Request message, the dispatcher gets the service name and subscriptionId from the envelope. It then determines which object(s) should receive the message, and executes the *eventHandler* function of that object. Each object that handles event notification messages must have an *eventHandler* function.

3.5.7 Administration Message Support

For administration request messages, the dispatcher maintains a list of objects which handle specific administration messages. It also contains the address of an object which is the default administrative message handler. The list contains the following information:

- *The name* of the administration command.
- *The address* of the object which will provide the administrative support.

If the message is an Administration Message, the dispatcher gets the name of the command from the envelope. It determines which object should receive the message and executes the *requestHandler* function of that object. Each object that handles administration messages must have a *requestHandler* function.

3.5.8 Timeout Processing Support

The dispatcher also provides timeout processing. When requested, the dispatcher looks through its list of messages awaiting responses and determines if any have timed out, based on the timeout value given when the message was registered.

For each message that has timed out, the dispatcher executes a *timeoutHandler* function on the registered object and removes the message from its list.

3.5.9 cpTransportManager Class

The cpTransportManager class manages a set of transport objects for either a client or server process. The transport manager is responsible for instantiating the appropriate transport objects in a process based on the service or vendor name.

3.5.9.1 Purpose of the cpTransportManager Class

The cpTransportManager class instantiates and manages a set of transport objects. Each service is available via one queue-based message product and there is exactly one transport object for each queue-based message product. The cpTransportManager class makes the choice of queue-based message product transparent to an application.

3.5.9.2 Basic cpTransportManager Operations

In each client and server process, there must be an initial registration with each queue-based message product used by the process. In particular,

- a server process must provide its service name when registering so that it can attach to the named queue which will receive its service requests.
- a client process will generally register for an unnamed queue.

Note that in a messaging system, service requests go to a named queue, not a process. Depending upon the messaging vendor, secondary registrations (which provide additional queue attachments) may be allowed.

When a manager or agent object needs to send a message, it asks the transport manager for a transport object for the given service. An error is returned if the transport object has not been created for the required messaging product.

3.5.10 Processing Services Summary

• Table 3 - Processing Services Summary

Functional Component	Class	Responsibility
Processing Services	Dispatcher	Add/delete service request/response, event, and administration message handlers to processes
		Timeout processing
		Dispatch message for processing
	Transport Manager	Add/delete transport object to available transports
		Get transport object

3.6 *Message Manipulation and Transport Services*

Message Manipulation and Transport Services consist of a set of services that facilitates the communication between a client and a server (a manager and an agent).

These services provide:

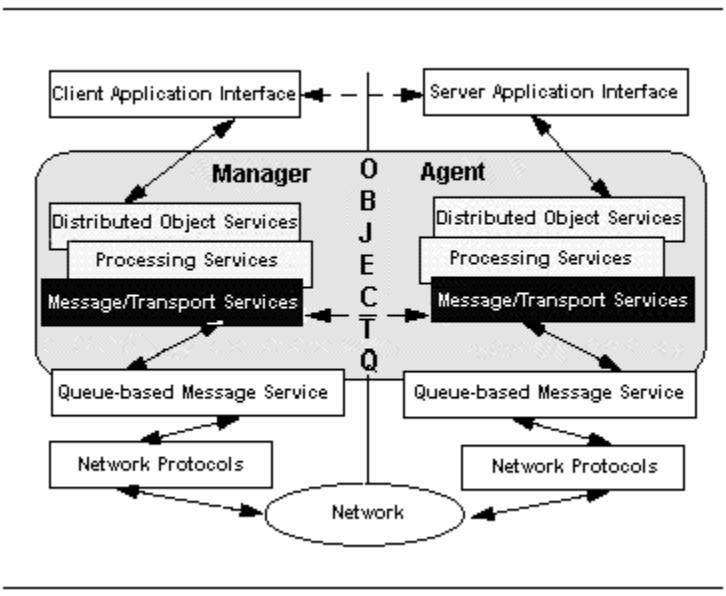
1. Construction of messages and extraction of fields from a message. UNIX-to-UNIX messages are generally in a standard CMIP-like format of attribute-value pairs; UNIX-to-MVS messages are in APPC transaction for-mat. Administration messages are also supported.
2. Transmission of messages between clients and servers, independent of the location of the clients and servers and of the underlying transport mechanism used to send and receive messages.

In Figure 10, notice the thin layer of ObjectQ that works above the queue-based message service layer. This layer is designed to allow transparent access by higher layers to the specific queue-based message product.

3.6.1 Message Manipulation Services

In order to send a message, that message must be placed into an envelope, which contains necessary header information.

Requests are generally one request message in an envelope, though multiple request messages can go in the same envelope[3]. If a response to a request results in multiple messages (each response message contains information on only one instance of the distributed object), one or more of the response messages can be packed into an envelope. It is also possible to send multiple envelopes in response to a single request. The service provider manager must be able to handle the possibly multiple messages and/or envelopes in response to a single request.



[3] Administration messages can only be packaged one message per envelope.

• Figure 10 - Message Manipulation and Transport Services Layer

3.6.2 cpAttribute Class

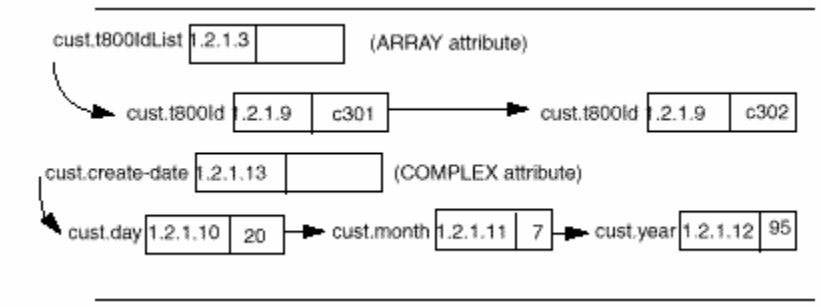
The cpAttribute class is used in cpMessage class objects as the container for data items, so that the data items can be passed between applications.

- An attribute has a name, a type, a value, and an operator.
- An attribute is identified either by a registered name or a string of the form *domainName.attributeName*. When the latter format is used, the data item *zipcode* in the domain *DDD* has the attribute identifier *DDD.zipcode*. If, in the domain *DDD* there are several classes that have a *zipcode* attribute, the attribute in class *contact* can be identified as *DDD.contact.DDD.zipcode*. Registered names are used internally by ObjectQ, and are typically never referred to by application code.

The following attribute data types are supported:

- CHAR — a single character.
- SHORT — a short integer.
- LONG — a long integer.
- OCTETSTRING— an arbitrary sequence of bytes.
- STRING — a character string (not containing nulls).
- COMPLEX— a complex type of attribute which is a composite of attributes of differing types.
- ARRAY — an array of attribute values for a specified attribute.

When an attribute is initialized or set with a value, the type of the value supplied must match the type specified in the MIB. Note that if the types don't match the attribute value is set to *NULL*, and an error code is returned (since constructors do not return values, the member function *createStatus* may be used to determine if the instantiation was successful).



• Figure 11 - Internal Representation of cpAttribute Class Data

The operator is any one of the CMIS-defined types of: REPLACE, ADD_VALUE, REMOVE_VALUE, DEFAULT, or a relational operator of EQUAL, GREATER_THAN, LESS_THAN, GT_EQUAL, LT_EQUAL, or NEQUAL. The default is the relational operator EQUAL. If the type of an attribute is COMPLEX or ARRAY, the cpAttribute class will contain a "sublist" holding these component attributes.

Figure 11 shows how the sublist of component attributes for ARRAY and COMPLEX attributes defined in Table 2-4 works: each attribute's sublist is made of up of the appropriate individual attributes identified by registered name. Note also that each attribute has been flattened into a registered name/value pair.

Through the use of *save* and *restore* functions, attribute and attribute lists can be saved to persistent storage, where they are represented as octet strings, and later restored. This is a helpful feature, if for example, certain attributes or attribute lists need to be re-used frequently.

3.6.3 cpMessage Class

The cpMessage class is a message manipulator. The cpMessage class can be used:

1. To construct CMIP-type request and response messages and to extract data from those messages for UNIX-to-UNIX communications.
2. To construct transaction-based messages and extracts data from those messages for UNIX-to-MVS communications.
3. To construct administration messages and to extract data from those messages for process-level activities.

The cpMessage class is an abstract base class for all message types. There is one pair of request/response cpMessage subclasses for each of these message types:

- CMIP
  - ***Get*** — retrieve data.
  - ***Set***— modify data.
  - ***Create***— create managed objects.
  - ***Delete***— delete managed objects.
  - ***Action***— invoke one of a managed object's member functions.
  - ***Event report*** — for reporting asynchronous events.
- APPC
  - ***Transaction*** — format data for/from IMS processing.

- Admin
    - ***Administration***—handle administration at the process level.

Through the use of *save* and *restore* functions, messages can be saved to persistent storage, where they are represented as octet strings, and later restored.

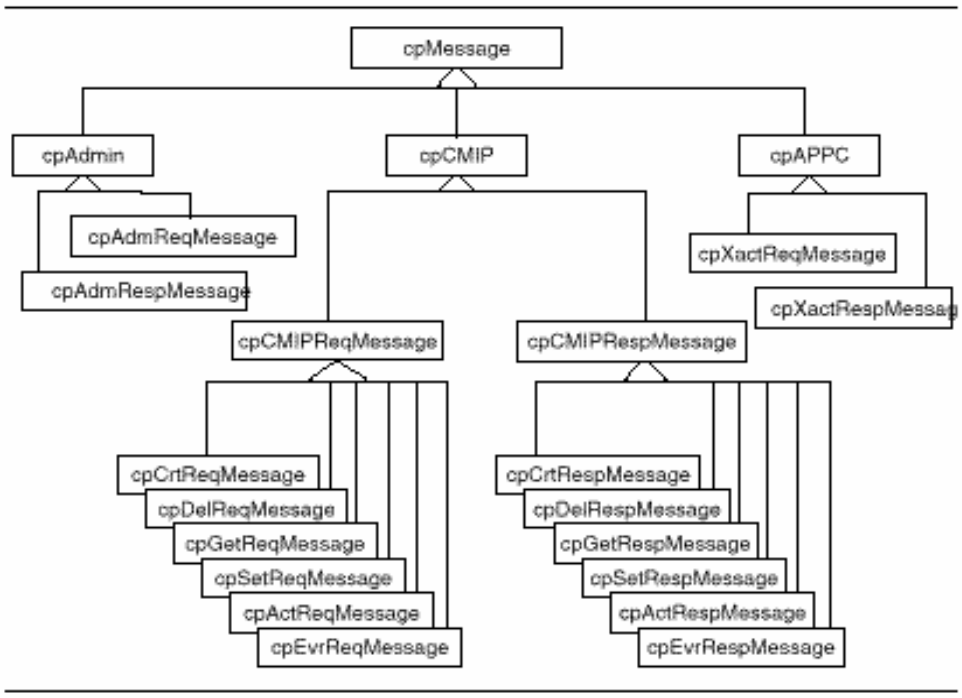
This is a helpful feature, if for example, certain messages need to be reused frequently.

3.6.4 cpMessage Class Organization

For programming convenience, messages are organized as three abstract classes which serve as base classes for all the other types of messages. The root class is cpMessage. In the first inheritance level, the three derived classes are cpAPPC, cpAdmin and cpCMIP.

Both cpAPPC and cpAdmin contain a pair of subclasses for defining request and response messages.

The cpCMIP derived class is further branched into cpCMIPReqMessage and cpCMIPRespMessage, where each comprises of a set of CMIP request and response message subclasses, respectively. Figure 12 illustrates this hierarchy graphically.



• Figure 12 - Message Inheritance Structure

3.6.4.1 cpMessage Class Fields

Messages consist of three parts: the vendor's header (which contains information such as message priority, source and destination), ObjectQ's header (which con-tains information such

as invokeId, and other data which is used to select specific messages, or types of message, from the queue), and the user's data. Table 4 lists the parameters that can be used for each message type. Note that not all parameters are used in each message. Also, request and response formats for each message may be different.

- ***message-type:***
- ***invoke-id:*** uniquely identifying a request/response message pair.
- ***object-class:*** the object class to which this instance belongs.
- ***access:*** for security purposes (actual use undefined at present).
- ***instance-id:*** a string which uniquely identifies one instance of an object in the class.
- ***transaction-id:*** a string of eight characters or less which provides unique identification for an MVS transaction.
- ***link-id***: for multiple linked responses, each usually has an increasing link-id.

• Table 4 - Message Class Parameters and Format

		CreateRsp	DeleteReq	DeleteRsp	SetReq	SetRsp	GetReq	GetRsp	ActionReq	ActionRsp	EvRepReq	EvRepRsp	XactReq	XactRsp	AdminReq	AdminRsp
message-type	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
invoke-id	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
object-class	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔			✔	✔
access	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
instance-id	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔			✔	✔
transaction-id													✔	✔		
link-id		✔		✔		✔		✔		✔						✔
mode					✔				✔		✔					
scope			✔		✔		✔		✔							
instance-filter			✔		✔		✔		✔							

		CreateRsp	DeleteReq	DeleteRsp	SetReq	SetRsp	GetReq	GetRsp	ActionReq	ActionRsp	EvRepReq	EvRepRsp	XactReq	XactRsp	AdminReq	AdminRsp
event-filter									✔							
synchronization			✔		✔		✔		✔							
time		✔		✔		✔		✔		✔	✔	✔				
type									✔	✔	✔	✔				
attributes	✔	✔			✔	✔		✔	✔	✔	✔	✔	✔	✔		✔
attribute-identifiers							✔		✔							
errors		✔		✔		✔		✔		✔		✔				✔
max-responses							✔		✔							
suggested-packing			✔		✔		✔		✔							

- mode - whether the message requires confirmation.
- Scope- the level of the namespace that is searched for an object; see the discussion of Containment Trees in Chapter 4, ObjectQ MIB Reference.
- instance-filter - specifies a set of instances of the class.
- event-filter - specifies what event is being subscribed to.
- Synchronization - how to apply actions that impact multiple objects (best effort, atomic).
- Time - the time of the event.
- type - the type of an action or event report request.
- attributes - a list of attribute name/value pairs.
- attribute-identifiers) - list of attribute identifiers only; specifies which attribute values should be returned.
- error - error list.
- max-messages - maximum number of messages to be sent as a response.
- suggested-packing - desired number of messages to pack in each response

envelope.

3.6.4.2 Flattening and Unflattening

One of the functions of the cpMessage class and its subclasses is to flatten attribute objects when constructing a message for transport and to reconstruct these objects when the message components are accessed.

An ObjectQ message can be thought of as being a wrapper for sending attribute-value pairs between managers and agents. ObjectQ "flattens" these attribute-value pairs into a byte stream which is then "unflattened" by the recipient. This flattening and unflattening is done internally by ObjectQ, and is transparent to the application programmer.

Flattening and unflattening provides a simple form of self-defining message so that synchronization of attribute tables at both ends is not required to receive data.

3.6.5 Transaction Request and Response Messages

These messages are sent in a fixed format APPC-style transaction. Attributes are flattened according to transaction definitions contained in transaction definition files, which relate attributes to fixed formats and positions within the transaction record. These messages are formatted for use by an MVS process.

A transaction consists of an arbitrary number of sub-fields. A transaction is contained within an envelope; a sub-field corresponds to a message within the envelope. It is thus possible to represent a transaction consisting of repetitive segments as a series of messages within an envelope.

3.6.6 Machine Architecture Differences

The other function of the cpMessage class is to handle differences in machine architectures as to byte ordering (endianness). ObjectQ handles this transparently.

3.6.7 Error Handling

The message structure contains two items, which a service-provider populates with pertinent error or debugging information:

- ***return code:*** set to indicate success or failure.
- ***error codes:*** the message class provides an error list containing up to three ancillary error codes; the ancillary error codes can be used to contain service-defined values or one of the CMIS error codes contained in the DSAP.edf error definition file—these error codes should be set as a string, but may be retrieved as either a string or a long.

3.6.8 Message Manipulation Services Summary

• Table 5 - Message Manipulation Services Summary

Functional Component	Class	Responsibility
Message Manipulation Services	Attribute	Get/set attribute name value pair
	Message	Get/Set message components

3.7 *Message Transport Services*

These classes support the Transport Services functional component of the architecture.

Using these classes, clients and servers communicate, independent of location, hardware architecture, and without any knowledge of how a server provides its service.

The Message Transport Service provides a class that interfaces with the queue-based message product. This class is regarded as a "thin layer" on top of queue-based message services layer in Figure 5.

3.7.1 CpAudit/cpAuditResource Class

The cpAudit class contains data retrieved from a specified envelope. The cpAudit class contains data for a hop, which includes a block of sending audit data and receiving audit data.

The DSAP domain (Registered Name: 1.1) contains attribute definitions for a range of fields: the cpAuditResource object is used to specify which fields of those defined in the DSAP domain are to be used in an envelope. The cpAuditResource class object also contains flags, which indicate whether the audit capability is enabled, or not.

The fields defined in the DSAP domain for audit purposes are:

- process id
- user id of the process
- hostname of the system
- service requested
- queue address
- message size
- userdata-1
- userdata-2
- date/time stamps

Service-related audit data cannot be placed directly into the envelope; the transport object, using cpAuditResource data, actually writes to the envelope. However, the string fields userdata-1 and userdata-2 allow service-related data to be made part of the audit.

When audit is enabled, the envelope contains one array attribute, which holds the audit data blocks. Each audit data block consists of those items listed above plus further distinctions described in the cpAuditResource object. When the envelope reaches its next hop, audit

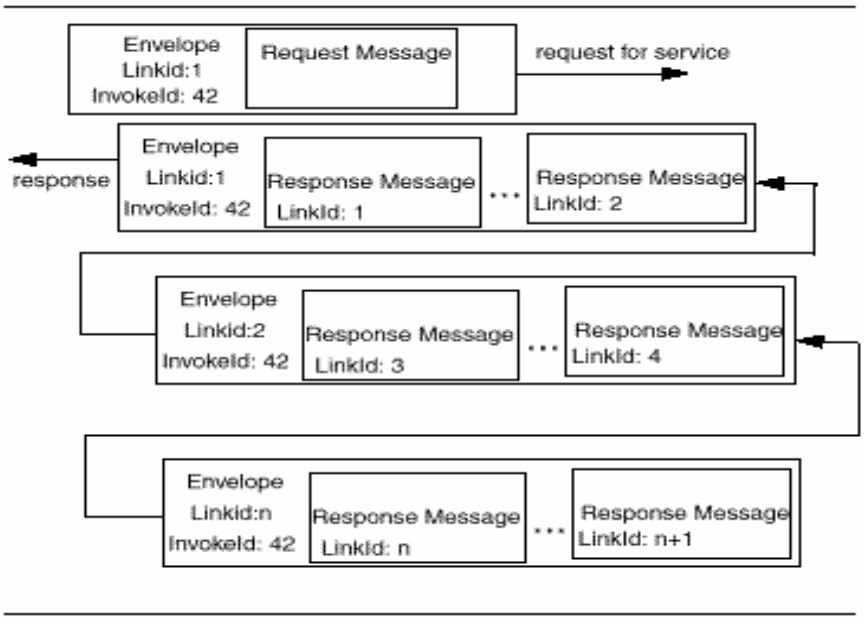
attributes are appended to the existing audit data block. When an envelope returns to its source, audit data will be complete for the entire envelope's stops.

3.7.2 cpEnvelope Class

The cpEnvelope class is the container that is sent between processes using a transport object. It is composed of a header and one or more ObjectQ message class objects.

The cpEnvelope class header contains a unique request identifier (invoke-id), and other data items that can be used to select specific envelopes from the message queue.

The invoke-id, which is created by the transport object when an envelope is sent, must also be used in all response envelopes to the request. In asynchronous processing, the invoke-id is used to match responses with requests.



• Figure 13 - Multiple Envelopes/Message as Part of a Response

An envelope object can contain more than one message only if all the messages are in response to the same request; usually, these response messages are all of the same type (get, set, action, delete).

When multiple messages are part of a response, the linkid field of the message inside the envelope is used to control the order of the messages. In all other cases, there can only be one message in an envelope object.

An envelope object can have an associated resource object, which sets message-specific transmission parameters. If the resource object does not exist, or if the values within it are not set, the values from the default resource object attached to the transport object are used.

Through the use of save and restore functions, envelopes (and their contents) can be saved to persistent storage, where they are represented as octet strings, and can be later restored. This is a helpful feature, if for example, certain envelopes need to be re-used frequently or to provide fault recovery capabilities.

3.7.3 Accessible Transmission Parameters

The use of the cpEnvelope class hides the details of whether transmission parameters are in the queue-based message product's header or in the cpEnvelope class object's header. The following are definitions of transmission parameters than can be set and/or accessed.

- ***Source*** - who sent (or is sending) the message.
- ***Origin*** - who sent the message originally (since it may have been forwarded).
- ***Invoke-id*** - a unique identifier that pairs requests and responses. The invoke-id is set by the transport object when the envelope is sent, and remains set when using the transport object's *reply*, *reply to origin*, and *forward* operations.
- ***Subscription-id*** - unique identifier for each subscription request; can be initialized or obtained.
- ***Service*** - string which specifies service name.
- ***MsgType*** - one of the following supported message types:
    - Create Request/Response
    - Delete Request/Response
    - Set Request/Response
    - Get Request/Response
    - Administration Request/Response
    - Action Request/Response
    - Event Report Request/Response
    - Transaction Request/Response
- ***User-data***- user data that can be used as a criterion for message selection when reading message queue.
- ***Access*** - for security: actual use undefined at present time.
- ***Reply-to*** - a field holding the handle of the queue to which a response must go.
- ***Link-id*** - a short used for sequencing envelopes; note that the message also has a link-id, which is different from this one.
- ***Attribute-list*** - the envelope attribute list is intended to hold information for an audit trail.

3.7.4 cpResource Class

The cpResource class is associated with either an envelope object or a transport object and describes the characteristics of the message transmission.

3.7.4.1 Purpose of the cpResource Class

Transmission parameters set in a resource object associated with an envelope object override the transmission parameters set in a resource object associated with a transport object. In this way, transmission parameters can be set at the global level for all users of a transport object or reset on a per message level, as needed.

3.7.4.2 cpResource Parameters

The following can be set in a Resource object:

- ***Priority***— 2 levels, NORM_PRIORITY (default), and HIGH_PRIORITY.
- ***Assurance***— the assurance level of message delivery.
    - NO_ASSUR - non-persistent (no status indication returned to program) (default).
    - PARTIAL_ASSUR - non-persistent
    - FULL_ASSUR- persistent (guarantees the message will be delivered).
- ***Undeliverable message action -*** action to take if message cannot be delivered.
    - DISCARD - throw the message away.
    - LOG_DISCARD - log the error and throw the message away (default).
    - DEAD_LETTER - log the error and store the message in a dead-letter journal.
    - RETURN_TO_SENDER - return to sender with error.
- ***Send Timeout*** — the timeout, in seconds, specifies the length of time that a send call will wait before returning with a timeout error.
- ***Receive Timeout***— the timeout, in seconds, specifies the length of time that a receive call will wait before returning with a timeout error.

3.7.5 cpSelector Class

The cpSelector class is used to specify how the transport object should read the message queue. The selection criteria that can be set in a cpSelector class object are:

- Primary Queue
- Secondary Queue
- Invoke Id
- Message Type
- User data (8 bytes, free format)
- Service name
- Priority

Limits may be placed on these criteria, however, because of the choice of queue-based message product. Also, message selection may be expensive. See the manual page for the specific product in use.

3.7.6 cpTransport Class

The cpTransport class is an abstract base class that provides virtual transport functions so product-specific transport classes can specialize from the base class.

Users of the cpTransport class use the virtual functions of the cpTransport class to send, receive, and reply to messages.

3.7.6.1 Purpose of the cpTransport Class

The transport manager object manages transport objects, instantiating the appropriate transport object, based on service name. See the Processing Services section above for discussion of the cpTransportManager class.

Users of transport objects request them from the cpTransport class object and use the virtual operations defined for the abstract base class. Each product-specific transport object executes its own method for the requested operation. Each transport object also has an associated resource object which specifies transmission parameters. Default transmission parameters can be overridden by transmission parameters set in the resource object associated with an envelope.

3.7.6.2 Operations Provided by the cpTransport Class

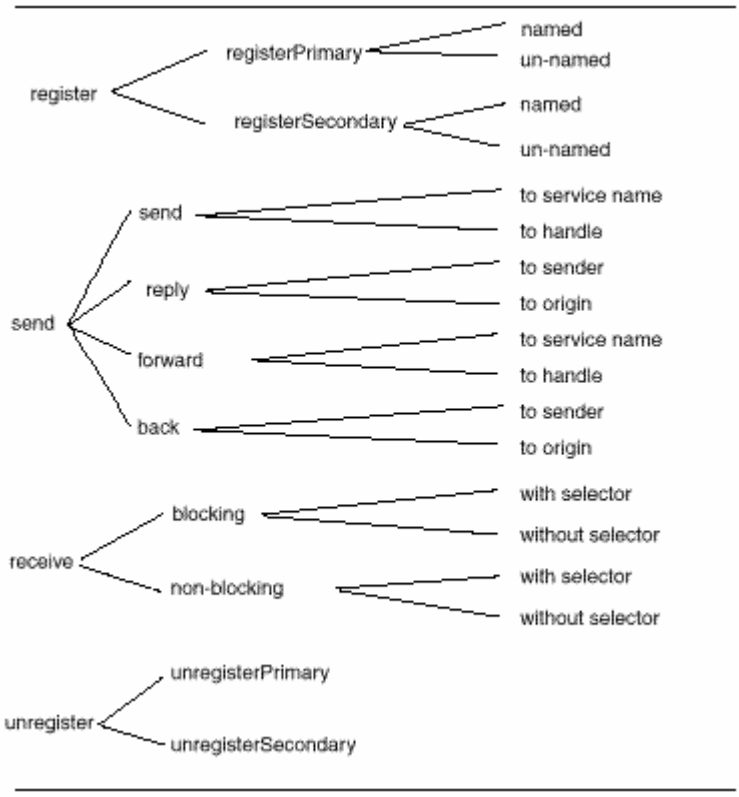
The operations provided by the cpTransport class are:

- ***Primary Register/Unregister*** — connect/disconnect from the queue-based message product's queuing engine and primary queue, either specified (by name) or unspecified (any queue); a service provider should use a service name when registering so the connection is to the queue where service requests are sent.
- ***Secondary Register/Unregister*** — connect/disconnect from secondary queue, either a queue specified by name, or unspecified (any queue); secondary registration is allowed only after primary registration.
- ***Send a message*** to a service — a message goes to the queue where the service is provided; servers providing that service connect to that queue.
- ***Send a message*** by specifying a handle —a handle obtained from a previously received message is used to send a message back to that specific queue.
- ***Receive*** — with either wait or no wait options and optional selection criteria.
- ***Reply***— respond to a request, reply sent to the source field in the request envelope (queue of requester, set by vendor) or, if reply_to field is set in the envelope, to reply_ to queue.
- ***Reply_to_origin***— send response to originator of request (determined as above,

either source or, if set, reply_to field), generally used after a forward.

- ***Forward***— send the request to another server, retaining the original source, invokeId, etc.

These operations are categorized in Figure 14.



• Figure 14 - cpTransport Class Operations

3.7.6.3 User-Defined Selection Criteria

The cpTransport class reads messages from the specified queue either FIFO or based on user-defined selection criteria. The user specifies the queue (that is, the primary queue and possible secondary queue) in the selector. Unless otherwise specified, the receive() is made from the primary queue. The following selection criteria can be set:

- InvokeId
- Service name
- Priority
- Message type
- User defined field
- Queue

3.7.6.4 cpTransport Errors Messages and Functions

All cpTransport class functions return either cpSUCCESS or cpFAIL. If cpFAIL is returned, the *lastRetCode* function indicates the specific error. If *lastRetCode* indicates

cpVENDOR_CODE, the error has been returned by the underlying messaging product— the *lastVendorCode* function will give the vendor-specific error code.

The only exception is that cpSUCCESS is returned from a non-blocking receive if the underlying receive was successful—this can be true even if no messages were read. In this case, *lastRetCode* will return cpNO_MESSAGES.

3.7.6.5 Message Transport Services Summary

• Table 6 - Message Manipulation Services Summary

Functional Component	Class	Responsibility
Message Transport Services	Audit	Block of information regarding each hop that an envelope takes
	Audit Resource	Specifies which fields will be used in the audit data
	Envelope	Get/set transport parameters
		Get/add message object
	Resource	Get/set transmission parameters
	Selector	Get/set message selection parameters
	Transport	Send message, reply to messages, forward messages and receive messages

4 Communication Scenarios

How do the classes used by ObjectQ interact on a typical client-server communication?

Which functions in which classes are used to send and receive messages in common situations? What needs to be running so that ObjectQ's classes will work?

The following sections describe some processing paradigms and how the set of classes defined for the platform can be used to implement them. Refer to Chapter 2, Introduction to ObjectQ Classes for a description of individual classes; in this chapter, only the particular object or function used in the activity is considered.

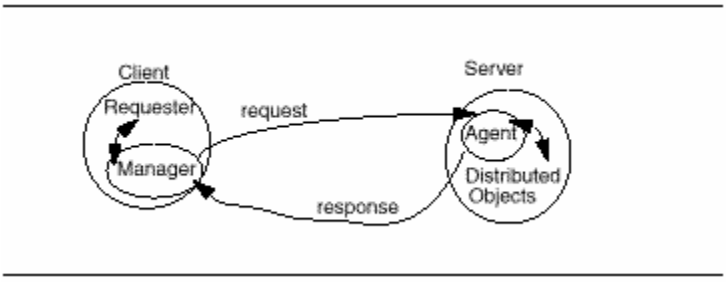
4.1 *About Communications Scenarios*

Two types of information are provided at important stages in the analysis:

1. A figure accompanied by text that describes how ObjectQ works. Note that there are no figures for the Subscription Server scenario.
2. An 'event trace' taking the form of a chart which lists the major classes, objects, or functions used and the arguments passed back and forth; this chart also shows the destination(s) of certain important arguments. The vertical axis represents the passage of time.

4.1.1 Synchronous Client and Single-threaded Server

This is the most simple case: one client sends a request to a server. There is no multiprocessing: each machine has only one (primary) queue. Each time a client sends a request, it waits for the response before sending its next request. The server processes only one request at a time. This simple scenario is expressed as Figure 15.



• Figure 15 - Simple Client-Server (or Manager-Agent) Transaction with Primary Queues.

The processing is divided into these steps:

1. Synchronous Client Initialization
2. Synchronous Client Issues Request
3. Single-threaded Server Initialization

4. Single-threaded Server Processes Request
5. Single-threaded Server Completes Request
6. Synchronous Client Processes Response

4.1.1.1 Synchronous Client Initialization

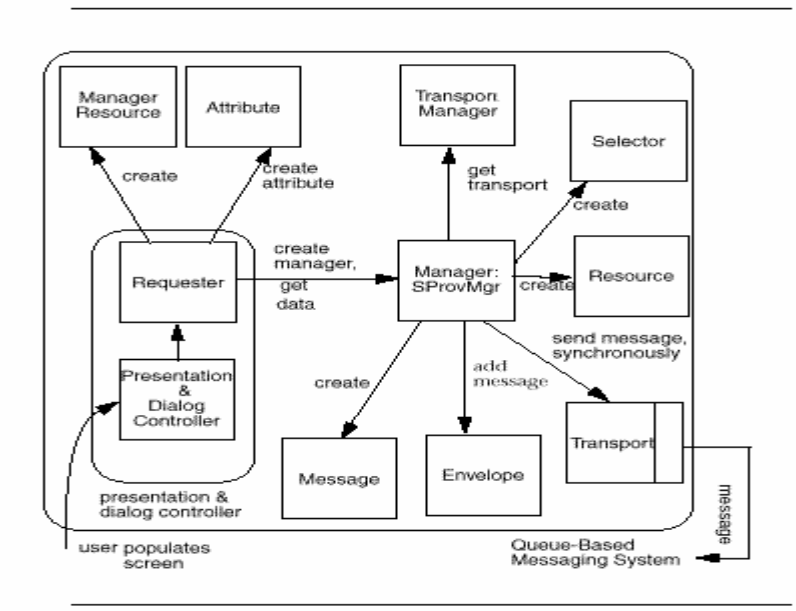
When a client is initialized, the following objects are created:

- cpTransportManager object.
- cpDispatcher object.
- Initialization table objects

The newly-instantiated transport manager creates a transport object for the named queue-based message product, in this case BEA MessageQ. The registerPrimary operation attaches the process to the queuing engine and a queue, assigned by the queue-based message product. Attachments to other queues, if needed, can be handled by executing the registerSecondary operation on the transport object.

4.1.1.2 Synchronous Client Issues Request

A client request (Figure 16) may be initiated by a user selecting something on a screen or from an application that needs information from another application. In the case of a user-interface client, the request is an event to the presentation layer, which passes the request to the dialog controller.



• Figure 16 - Synchronous Client Issues Request

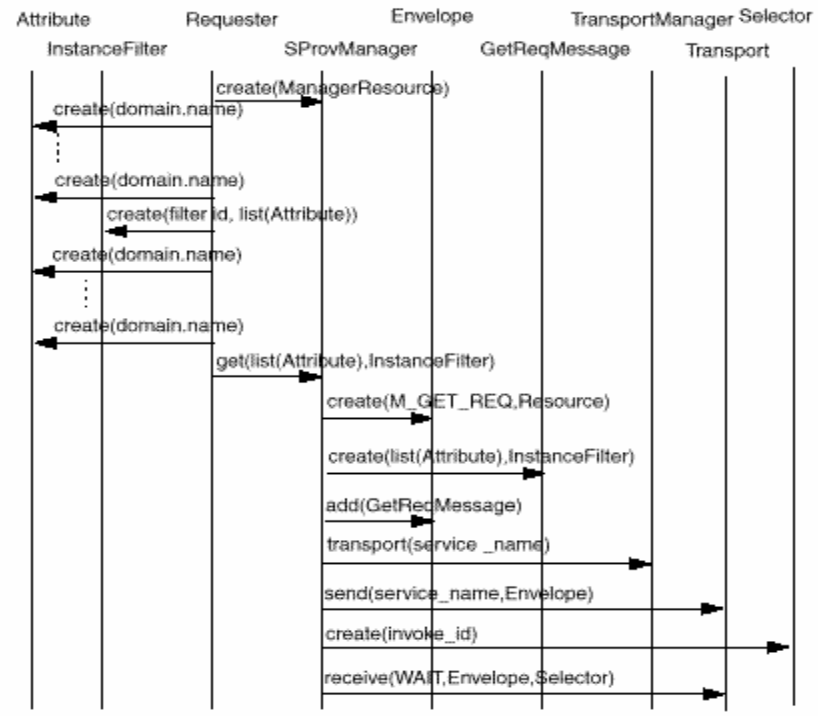
At some point, a member function of an object in the dialog controller (a requester object) is executed which requires some operation on a distributed object.

Figure 17 shows the event trace. In order to simplify the diagram some details, such as the settings in the manager resource object, the resource object, and the selector object, are left out. The manager resource is created by the requester and passed to the manager when it is created. For this scenario, the following are set in the manager resource created by the requester. All other values are default values:

- synchMessage: `TRUE`

The resource object is created by the manager and sets the following parameters:

- Priority: NORM_PRIORITY,
- Assurance: NO_ASSUR
- UMAction: LOG_DISCARD
- SendTimeout: NULL_TIMEOUT
- ReceiveTimeout: NULL_TIMEOUT.



• Figure 17 - Event Trace - Synchronous Client Issues Request

4.1.1.3 Reading Event Trace Figures

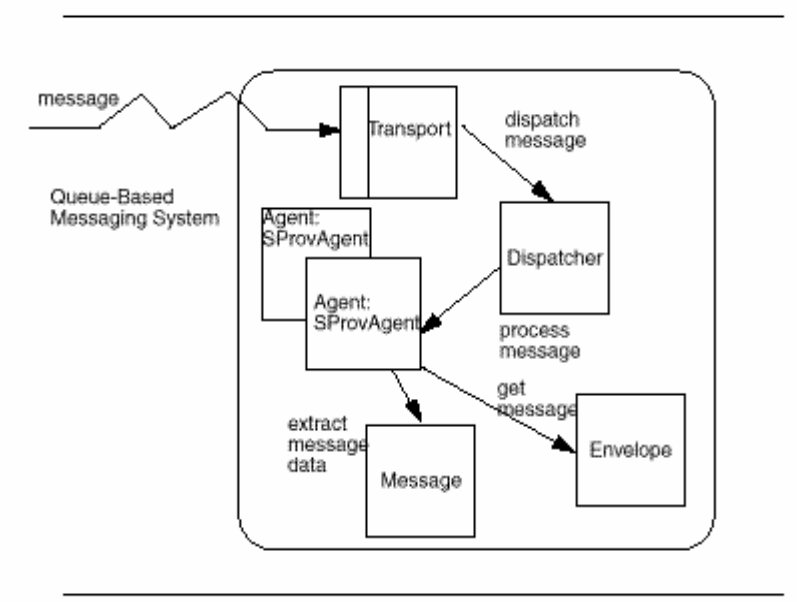
On this and subsequent event trace figures, note the originating object (where each arrow begins) and the recipient object (where each arrow ends). The vertical axis indicates the passage of time. For example, in the event trace above, the requester object first `creates` the manager resource, next the four domain-name pairs (which become the attribute list), and next the instanceFilter. Once these actions are finished, the requester executes the `get` of the

attribute list and the instanceFilter with the service provider manager as the recipient. The service provider manager performs the rest of the actions.

4.1.1.4 Single-threaded Server Initialization

When a server is initialized, the following objects are created:

- cpDispatcher object.
- cpTransportManager object.
- Initialization table objects.
- Service Provider Agent objects, at least one for each service provided by the server process.



• Figure 18 - Single-threaded Server Processes Request

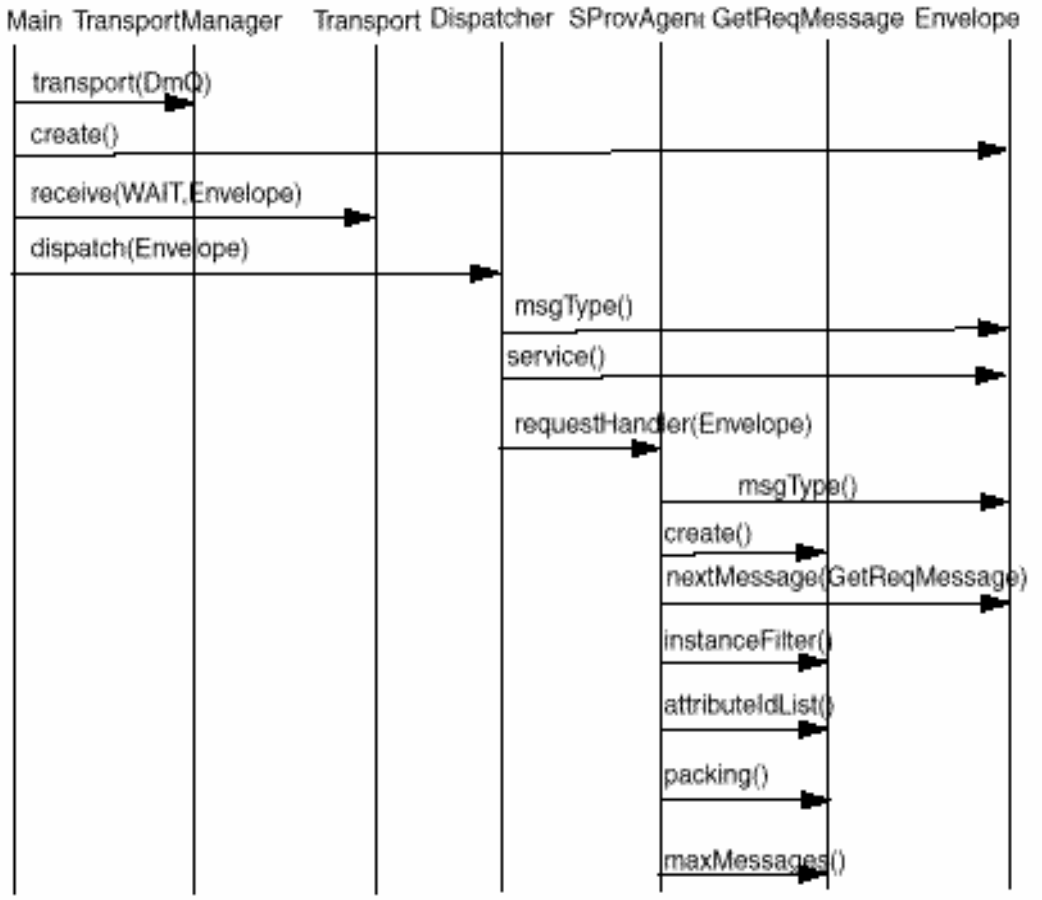
The transport manager creates a transport object for the appropriate queue-based message product, based on service name/queue-based message product mapping, and executes the registerPrimary operation on the transport object, with the given service name. The process is attached to the queuing engine and the queue with the given name. Attachments to other queues, if needed, can be handled by executing the registerSecondary operation on the transport object.

Each agent registers its service with the dispatcher object. When a service is registered with the dispatcher, the dispatcher stores the service name and the address of the agent providing that service. After the initialization is complete, the server process typically goes into an infinite loop, executing the receive operation on the transport object in blocking mode.

4.1.1.4.1 Single-threaded Server Processes Request

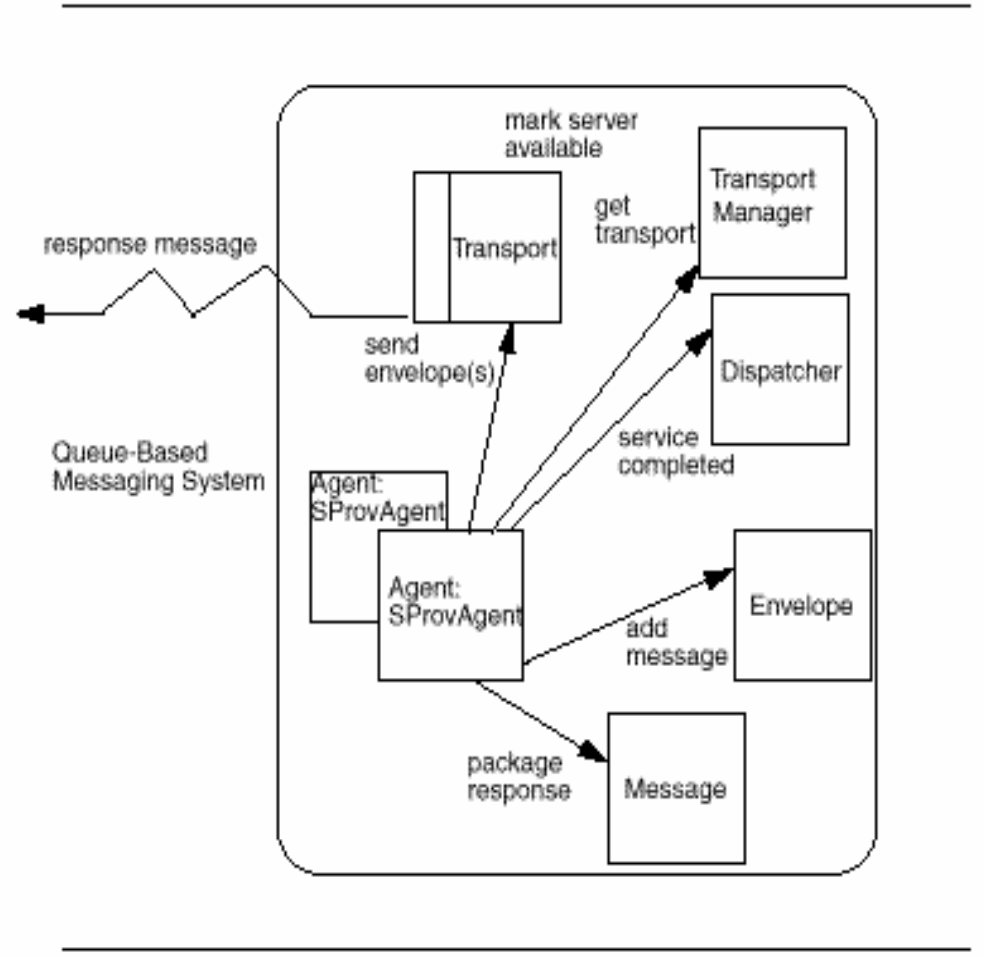
The blocking receive call is executed on the transport object. When a message is received, the dispatch operation of the dispatcher object is executed. The dispatcher then executes the

service agent's requestHandler function. Figure 18 represents the interactions between the classes needed to process the request.



• Figure 19 - Event Trace - Single-threaded Server Processes Request

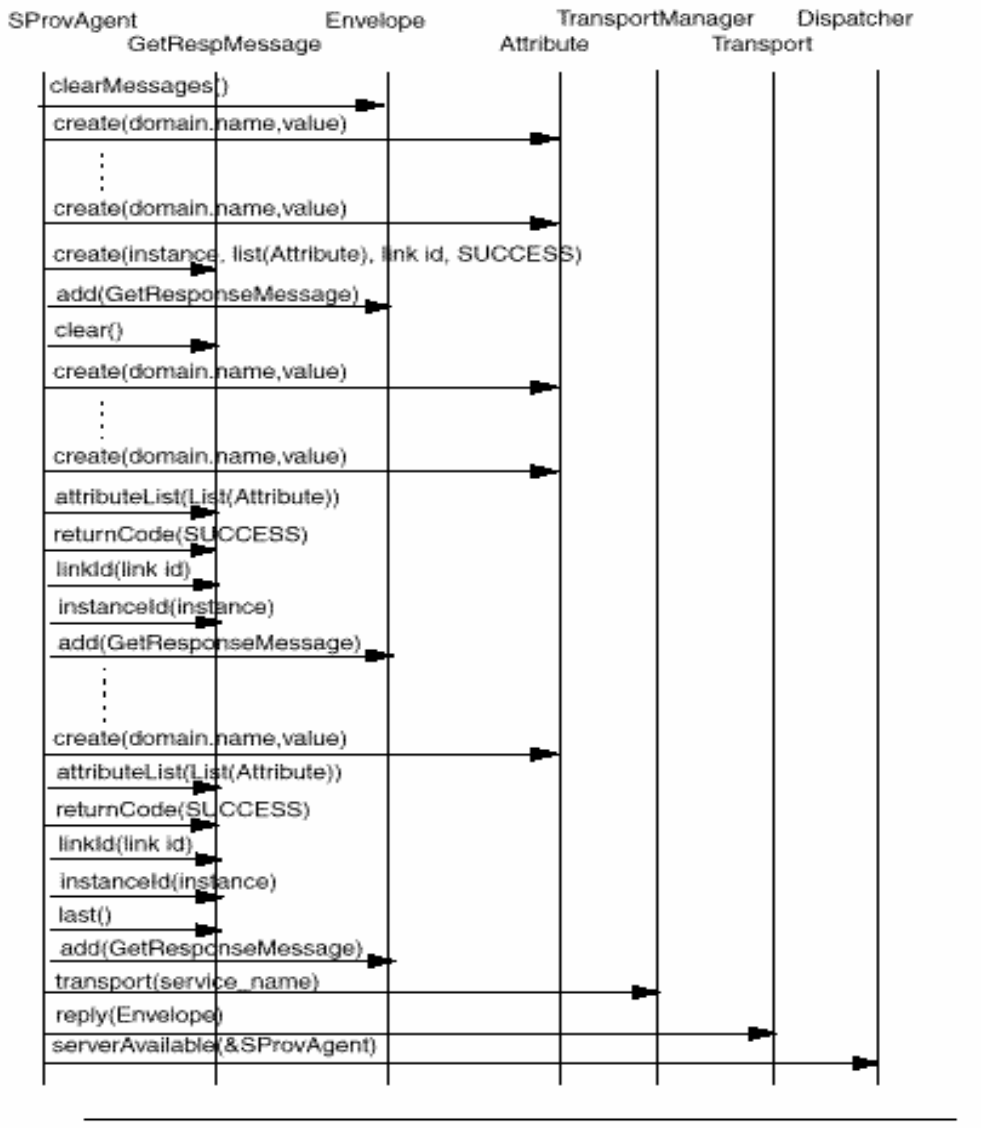
Since the server is single-threaded and provides the service itself, this server process is not available for any new services until it completes the processing of the current request. Figure 19 is the event trace.



• Figure 20 - Single-threaded Server Completes Request

4.1.1.4.2 Single-threaded Server Completes Request

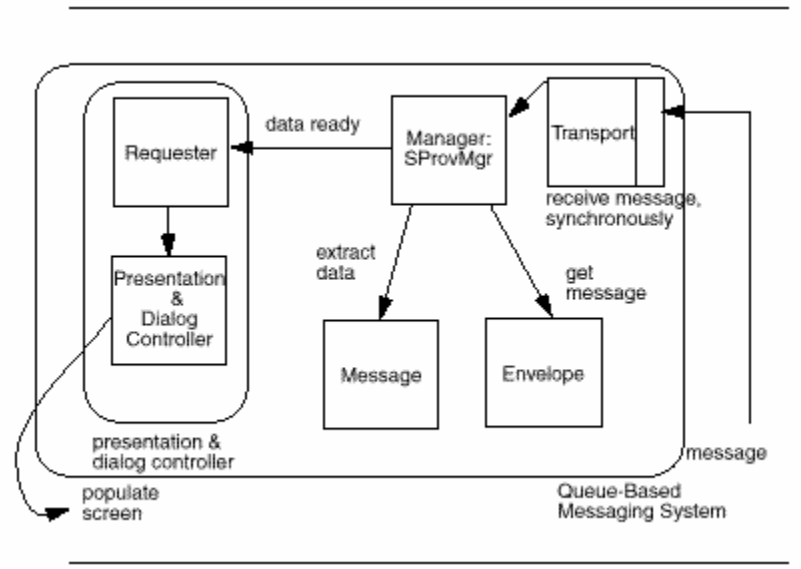
At some point, the agent completes processing of the request. Figure 3-6 shows the interactions between the classes needed to send the response back to the client. The event trace follows in Figure 21.



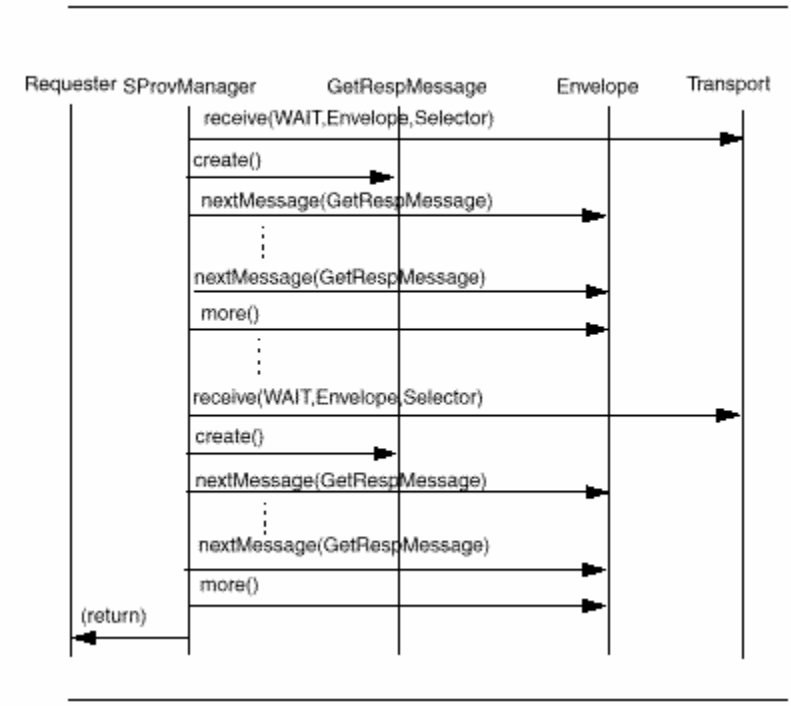
• Figure 21 - Event Trace - Single-threaded Server Completes Request

4.1.1.5 Synchronous Client Processes Response

Since the client's service request is made synchronously, the call to the send operation of the transport object does not return until the response is received. At that point, the manager uses the envelope to extract messages and the message to extract the data. The manager object's function call then returns to the requester where processing is completed.



• Figure 22 - Synchronous Client Processes Response



• Figure 23 - Event Trace - Synchronous Client Processes Response

In this scenario (Figure 22) the SProvManager (service provider manager) has done a blocking receive to the transport object. The event trace starts with the return from the receive operation in the SProvManager. The SprovManager gets all the messages out of the envelope and checks if there are any more envelopes (more()). If there are more envelopes, it repeats the blocking receive and the extraction of messages from the envelope. When there are no more envelopes, it returns control to the requester. Figure 23 contains the sequence of interactions.

4.1.2 Asynchronous Client and Multi-agent Hybrid Server

In this processing paradigm, a client process sends a request but does not wait for the response. An asynchronous client can have multiple outstanding requests resulting in increased throughput since these multiple requests are processed concurrently. This also allows a user interface process to accept input from a user while waiting for a response to a service request. A polling method is used to retrieve the responses when they arrive.

A hybrid server process handles service requests. In processing a service request, the hybrid server uses another server to complete processing. The service requests are sent asynchronously, allowing the hybrid server to be free to handle other service requests while it is waiting for a response. The dispatcher object handles dispatching the incoming messages, either server requests or response messages, to the appropriate object.

A hybrid server can issue synchronous messages to another server, but that scenario is not discussed in this document.

The processing is divided into the following steps:

1. Asynchronous Client Initialization
2. Asynchronous Client Issues Request
3. Multi-agent Hybrid Server Initialization
4. Multi-agent Hybrid Server Processes Request
5. Multi-agent Hybrid Server Issues Requests
6. Multi-agent Hybrid Server Processes Response
7. Multi-agent Hybrid Server Completes Request
8. Asynchronous Client Processes Response

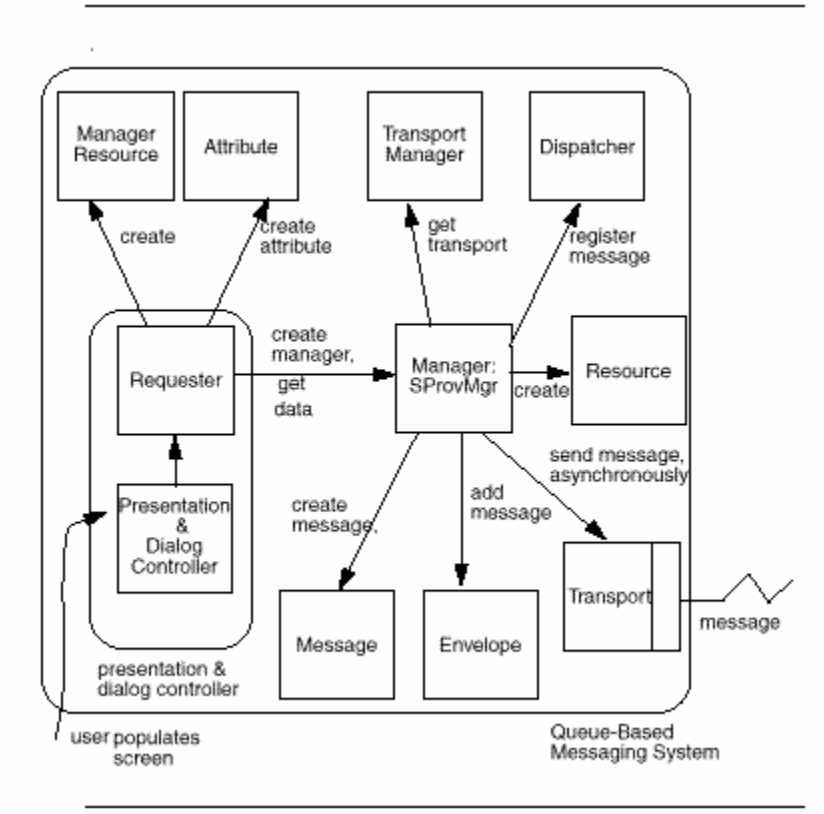
4.1.2.1 Asynchronous Client Initialization

When a client is initialized, the following objects are created:

- cpDispatcher object
- cpTransportManager object
- Initialization table objects

The newly-instantiated transport manager creates a transport object for the named queue-based message product, in this case BEA MessageQ. The registerPrimary operation attaches the process to the queuing engine and a queue, assigned by the queue-based message product. Attachments to other queues, if needed, can be handled by executing the registerSecondary operation on the transport object.

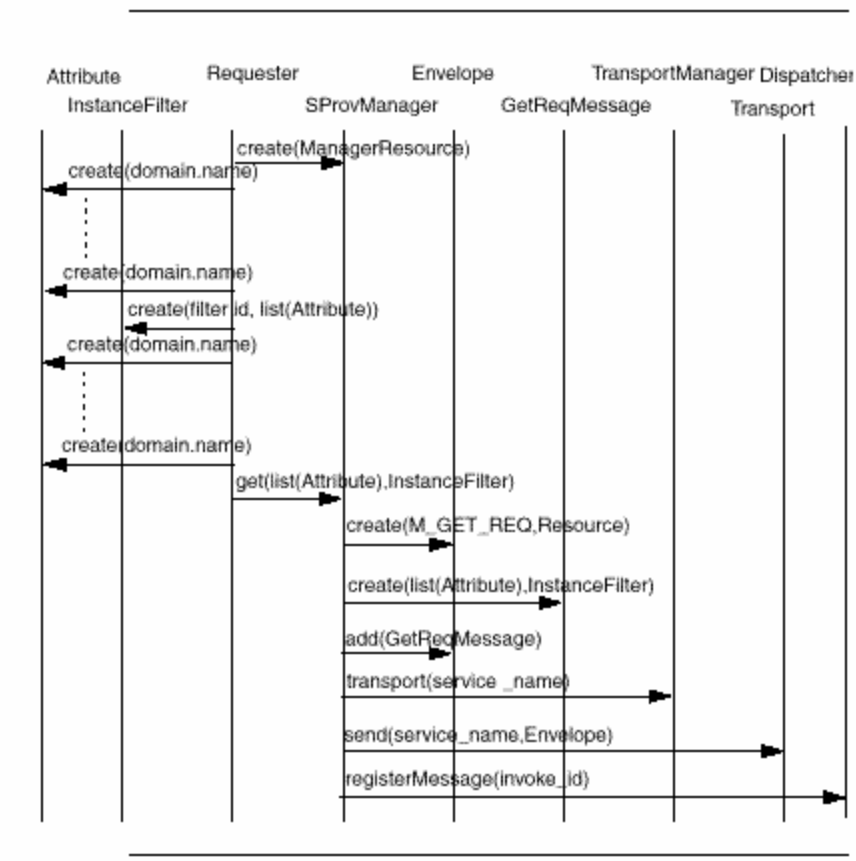
4.1.2.2 Asynchronous Client Issues Request



• Figure 24 - Asynchronous Client Issues Request

As with a synchronous client, processing begins when a user selects something on a screen or an application needs information from another application. This scenario describes an event in a user interface client process. The event occurs in a client presentation layer: a request is passed to the dialog controller.

At some point, a member function of a requester object is executed. In order to process the request, the requester must perform some operation on a distributed object. The following diagram represents the interactions between the classes needed to process the request in the client and send it to the server. Since the message is sent asynchronously, the client will not wait for a response. Some time later, when the response is received, the client process will process the response.



• Figure 25 - Event Trace - Asynchronous Client Issues Request

Figure 25 is an event trace which describes the sequence of messages between the objects.

In order to simplify the diagram some details, such as the settings in the manager resource object and the resource object, are left out. The manager resource is created by the requester and passed to the manager when it is created. For this scenario, default values are used in the manager resource object created by the requester object:

The resource object is created by the manager object and sets the following parameters:

- Priority: NULL_PRIORITY
- Assurance: NO_ASSUR
- UMAction: NULL_ACTION.

4.1.2.3 Multi-agent Hybrid Server Initialization

When a hybrid server is initialized, the following objects are created:

- cpDispatcher object
- cpTransportManager object
- Initialization table objects
- Application specific agent objects, one or more agents for each service provided

by the server process.

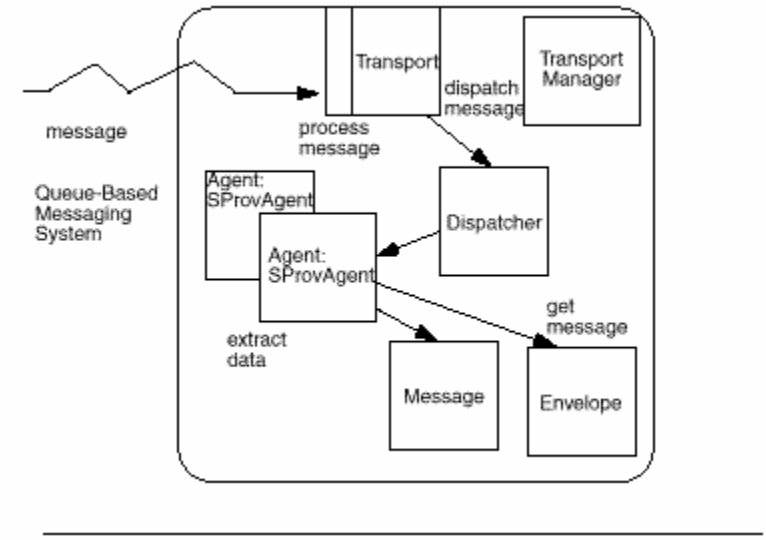
The following initialization are undertaken:

- The transport manager creates a transport object for the appropriate queue-based message product, based on service name/queue-based message product mapping, and executes the registerPrimary operation on the transport object, with the given service name. This will cause the process to be attached to the queuing engine and the queue with the given name. Attachments to other queues, if needed, can be handled by executing the registerSecondary operation on the appropriate transport object.
- Each agent registers its service with the dispatcher.
- When a service is registered with the dispatcher, the dispatcher stores the service name and the address of the agent providing that service.
- After the initialization is complete, the server process goes into an infinite loop, executing the receive operation (in blocking mode) on the transport object in the transport manager. If additional transport objects are added to the transport manager, some method of doing receive operations on each transport object is required.

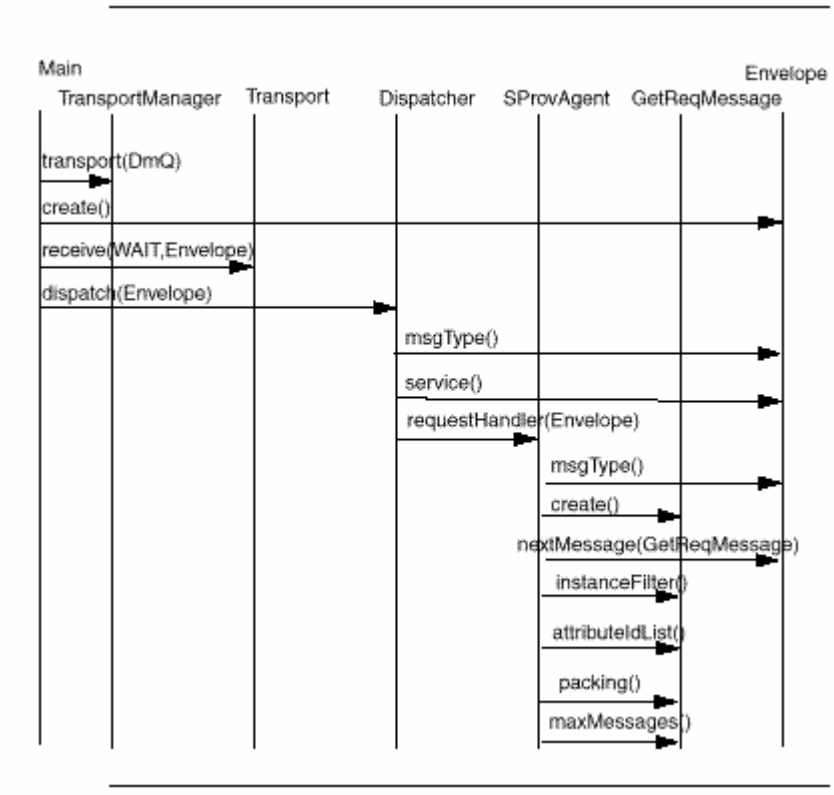
4.1.2.3.1 Multi-agent Hybrid Server Processes Request

The server process executes a blocking receive call on the transport object. When a message is received, the dispatch function of the dispatcher is executed.

The dispatcher then executes the service handling function of the agent providing the requested service. If multiple agents provide the same service, the dispatcher selects an agent which is not busy. If no agent is available to provide service, the dispatcher returns an error. Figure 26 represents the interactions between the classes needed to process the request in the server. Following the figure is an event trace which describes the sequence of messages between the objects.



• Figure 26 - Multi-agent Hybrid Server Processes Request

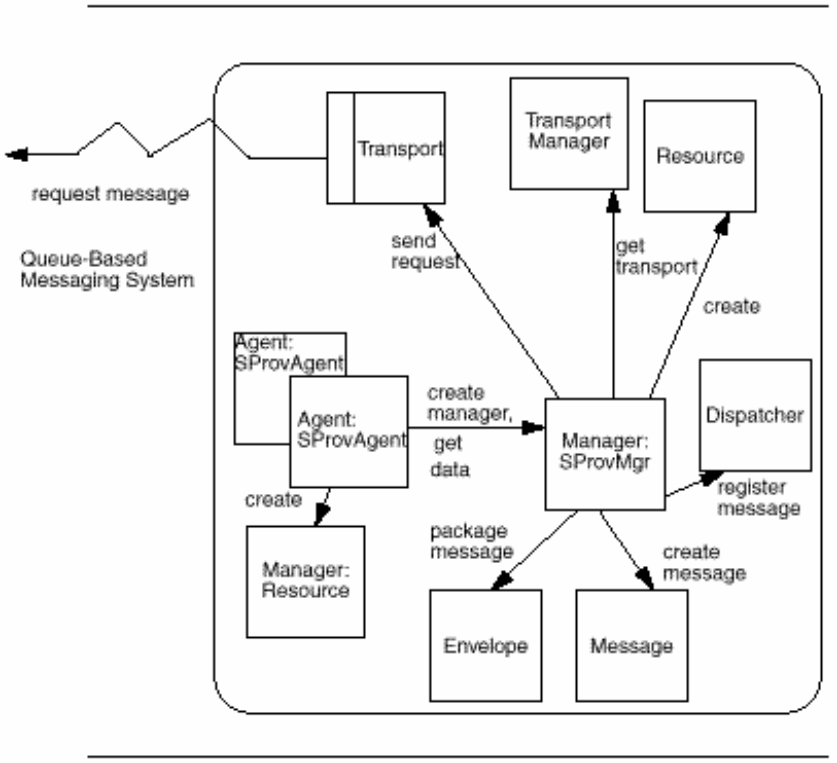


• Figure 27 - Event Trace - Multi-agent Hybrid Server Processes Request

4.1.2.3.2 Multi-agent Hybrid Server Issues Request

When an agent in a hybrid server needs to access operations on a distributed object it does not own, it makes a service request to another server, either synchronously or asynchronously. Using asynchronous messaging increases throughput. The set of class interactions are the same as for an asynchronous client.

The agent behaves like the requester object in that scenario. Below is a picture of the class interactions.

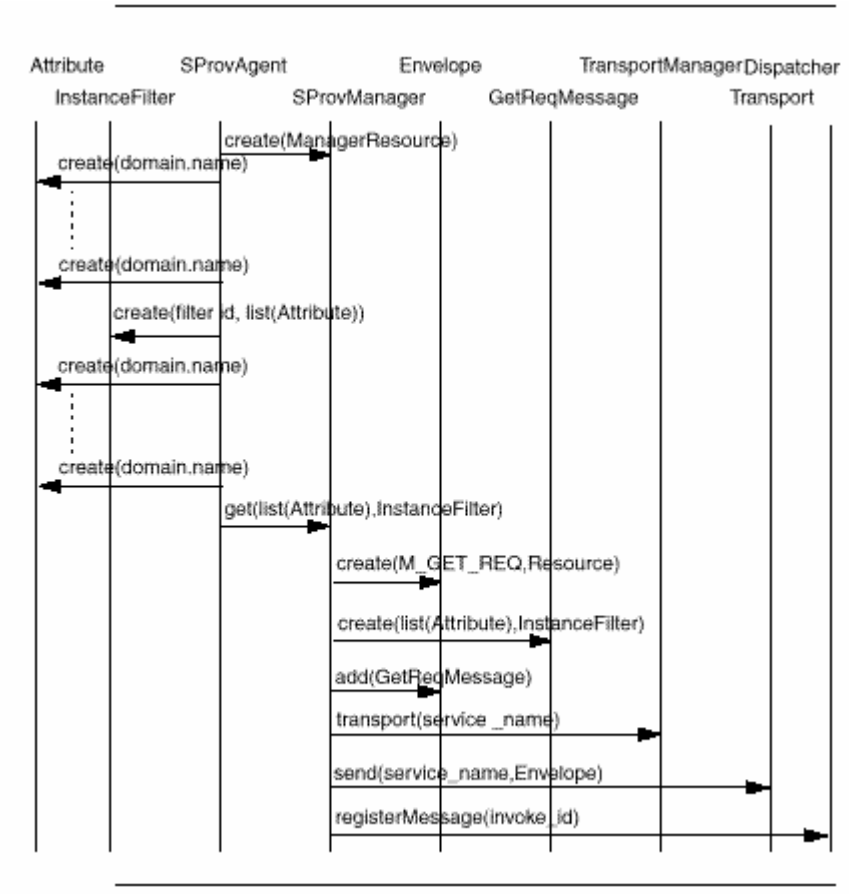


• Figure 28 - Multi-agent Hybrid Server Issues request

The following is the sequence of interactions. In order to simplify the trace some details, such as the settings in the manager resource object and the selector object, are left out. The manager resource is created by the requester and passed to the manager when it is created. For this scenario, the default values are set in the manager resource created by the requester.

The resource object is created by the manager and sets the following parameters:

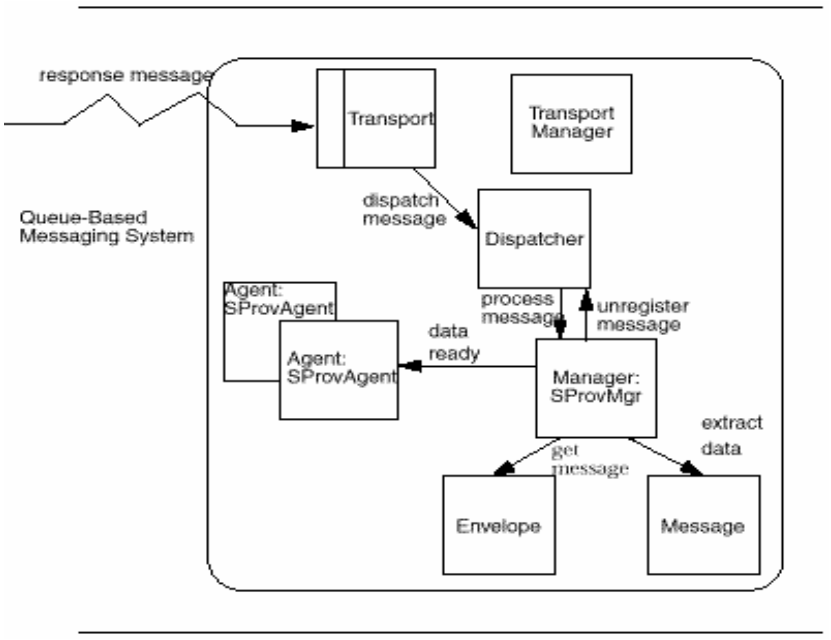
- Priority: NULL_PRIORITY
- Assurance:NO_ASSUR
- UMAction: NULL_ACTION.



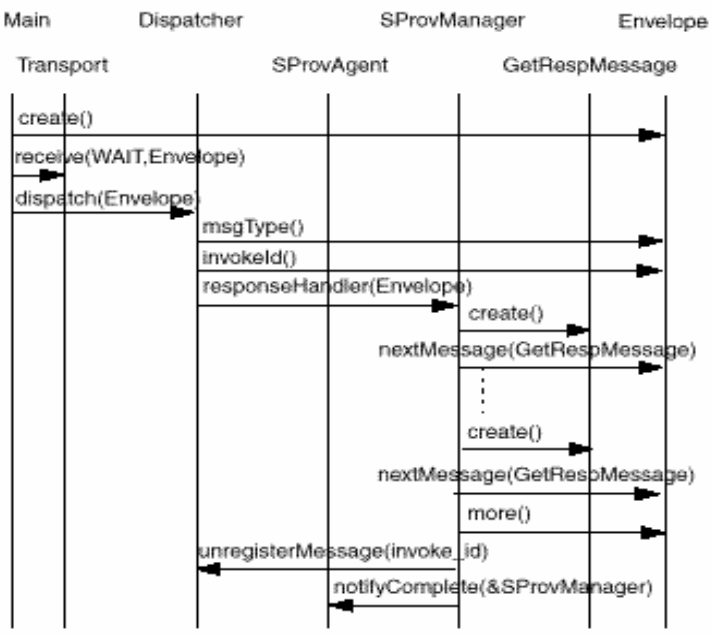
• Figure 29 - Event Trace - Multi-agent Hybrid Server Issues Request

4.1.2.3.3 Multi-agent Hybrid Server Processes Response

A server process, when not processing a request, executes a blocking receive for its incoming messages. When a message is received, it is passed to the dispatcher (that is, the dispatch operation is executed) which executes either a response handling routine on a manager or a service handling routine on an agent. The following assumes the message received is the response to the asynchronous service request. The set of class interactions is the same as for an asynchronous client as described in Figure 34. The agent behaves like the requester in that scenario. Figure 30 is a picture of the class interactions.



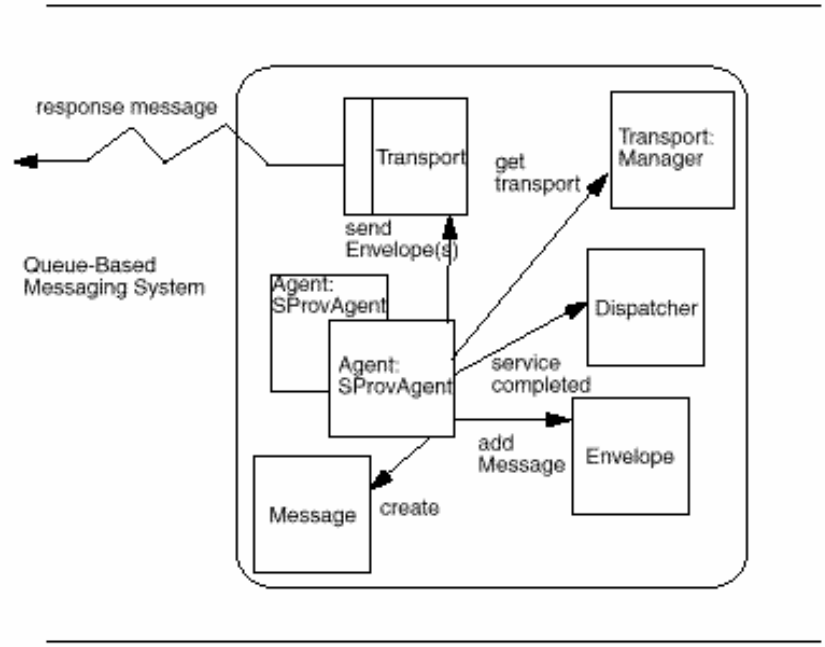
• Figure 30 - Multi-agent Hybrid Server Processes Response



• Figure 31 - Event Trace - Multi-agent Hybrid Server Processes Response

4.1.2.3.4 Multi-agent Hybrid Server Completes Request

At some point, the agent completes the processing of the service request. The set of class interactions is the same as for a single-threaded server. Figure 32 is a picture of the class interactions.



• Figure 32 - Multi-traded Hybrid Server Completes Request

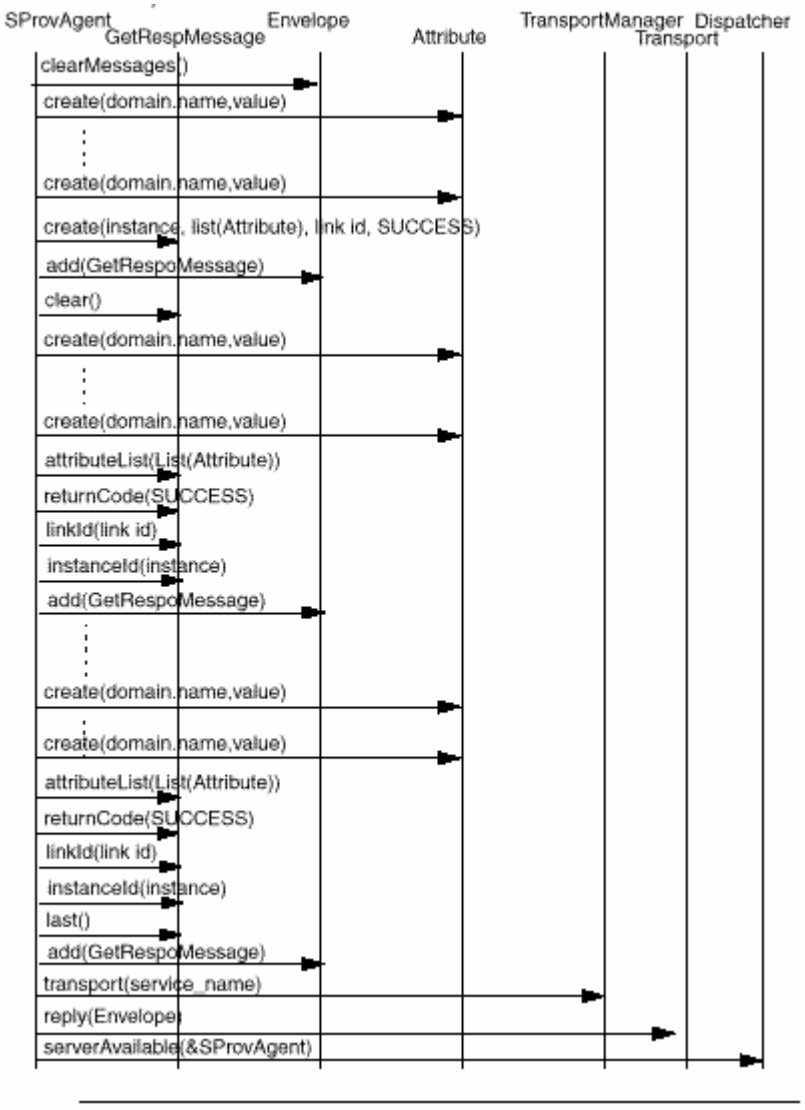
4.1.2.4 Asynchronous Client Processes Response 3-

In clients doing asynchronous messaging, a polling method is used to retrieve messages. Periodically, the client process executes the receive operation on each of the transport objects in the client to determine if there are any messages.

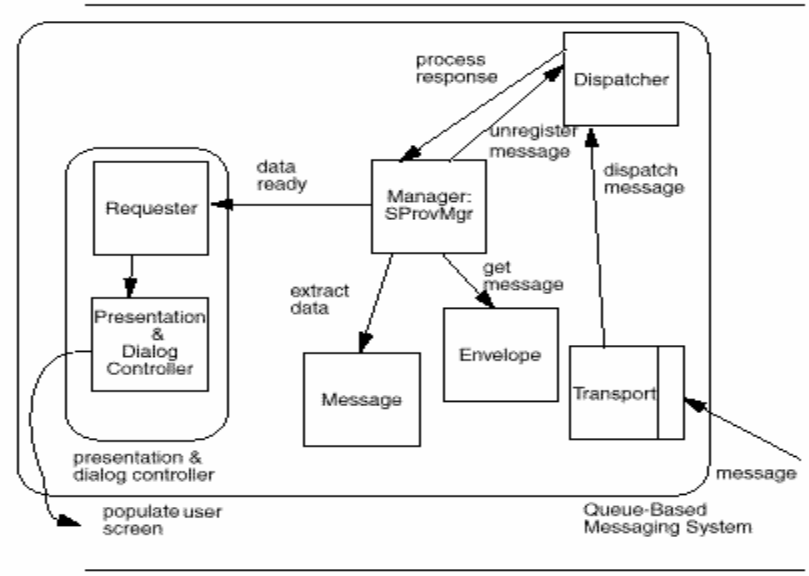
If there are none, other user-interface processing continues. If there are any response messages, the dispatch operation on the dispatcher is executed.

The dispatcher executes the response handling function of the appropriate object.

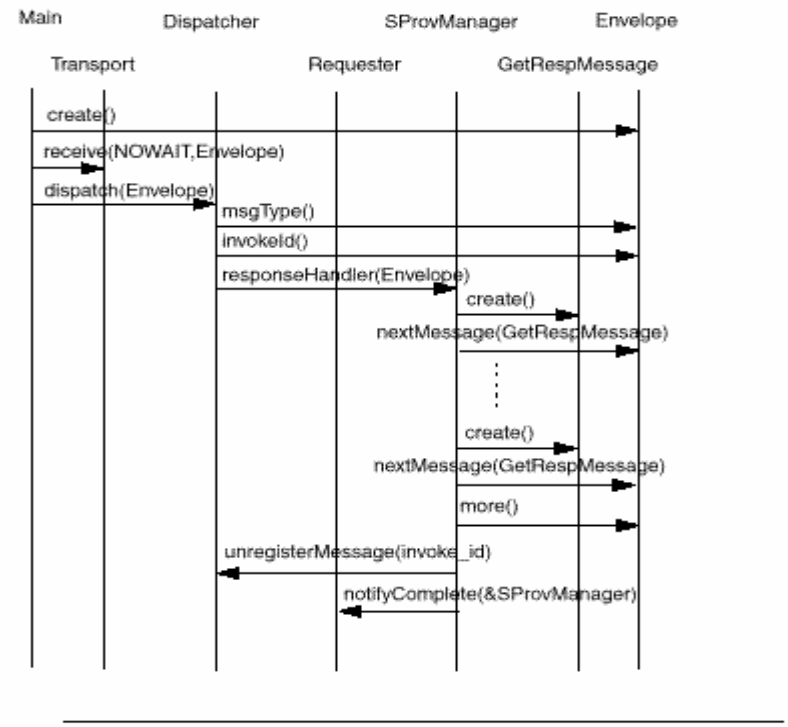
Figure 34 represents the interactions between the classes needed to process the response from the server. Figure 35 is an event trace describing the sequence of messages between the objects.



• Figure 33 - Event Trace - Multi-agent Hybrid Server Completes Request



• Figure 34 - Asynchronous Client Processes Response



• Figure 35 - Event Trace - Asynchronous Client Processes Response

4.1.3 Subscription Server With Event

In this processing paradigm, a client process sends a subscription request to a server process, which stores the subscription information. The subscription is a request to be notified when a

specified event occurs on one or more instances of a distributed object. Some time later, the server learns that an event has occurred and checks its list for matching subscriptions.

For each match, an Event Report Request message is sent. Both the subscription request message and the Event Report Request may require the receiver to send a confirmation. In the following scenario, the subscription request will be confirmed but the Event Report Request message will not be.

The initialization of client and server processes is the same as in the two preceding scenarios. The processing is divided into the following steps.

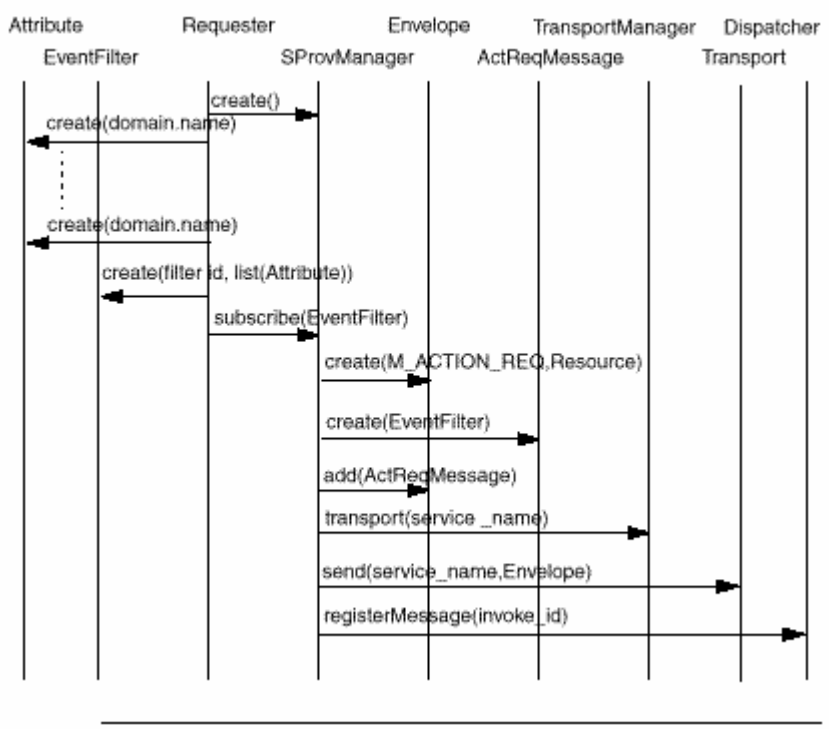
1. Subscription Request
2. Asynchronous Client Issues Subscription Request
3. Multi-agent Server Processes Subscription Request
4. Multi-agent Server Sends Subscription Response
5. Asynchronous Client Processes Subscription Response
6. Event Notification
7. Multi-agent Server Sends Event Notification
8. Asynchronous Client Processes Event Notification
9. Subscription Terminated
10. Asynchronous Client Issues Subscription Termination Request
11. Multi-agent Server Processes Subscription Termination Request

4.1.3.1 Subscription Request

Event notification can be requested from the user interface to provide dynamic screen updates or by any application which needs dynamic updates to drive application processing. The following scenario is a user-interface process requesting a subscription.

4.1.3.2 Asynchronous Client Issues Subscription Request

Processing begins when a user selects a screen, which requires dynamic updates. In order to process the request, the requester creates a service provider manager and executes the subscription operation on the manager. The manager sends the request to an agent, which provides subscription service for the distributed object.



• Figure 36 - Event Trace - Asynchronous Client Issues Subscription Request

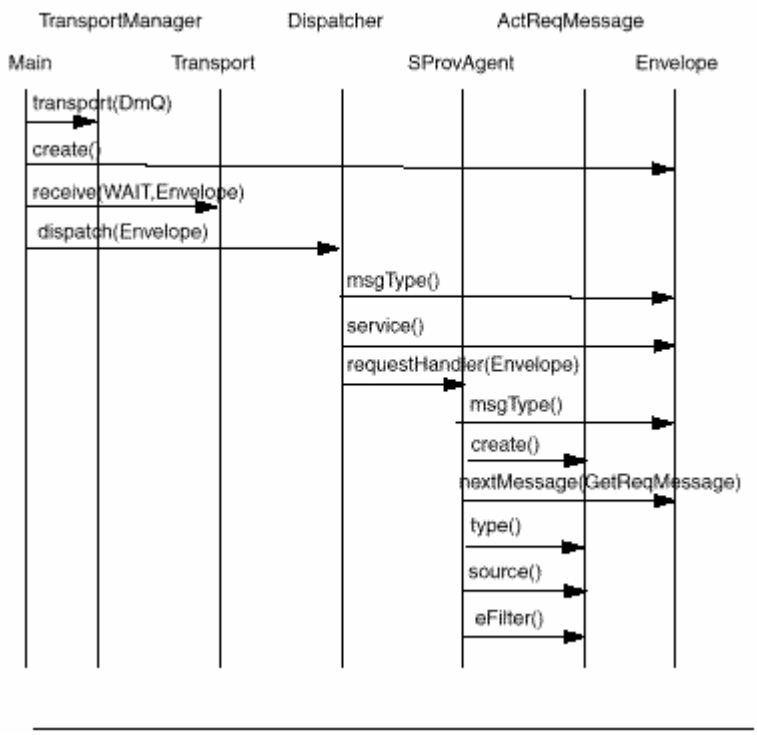
Since the message is being sent asynchronously, the client does not wait for the confirmation. Some time later, when the confirmation is received, the client will process it. The message type used in a subscription message is Action Request message. The response is an Action Response message. The type field in the Action Request message is used to specify that the requested action is subscription.

Figure 36 is an event trace, which describes the sequence of interactions between the objects. Since default settings can be used, no manager resource object is created. The resource object is created by the manager and has the following parameter settings:

- Priority: NULL_PRIORITY
- Assurance: NO_ASSUR
- UMAction:NULL_ACTION.

4.1.3.3 Multi-agent Server Processes Subscription Request

As in the other scenarios, the server process uses the transport object to execute a blocking receive. When a message is received, the dispatch function of the dispatcher is executed. The dispatcher then executes the service handling function of the agent providing the requested service. If multiple agents provide the same service, the dispatcher selects an agent, which is not busy.



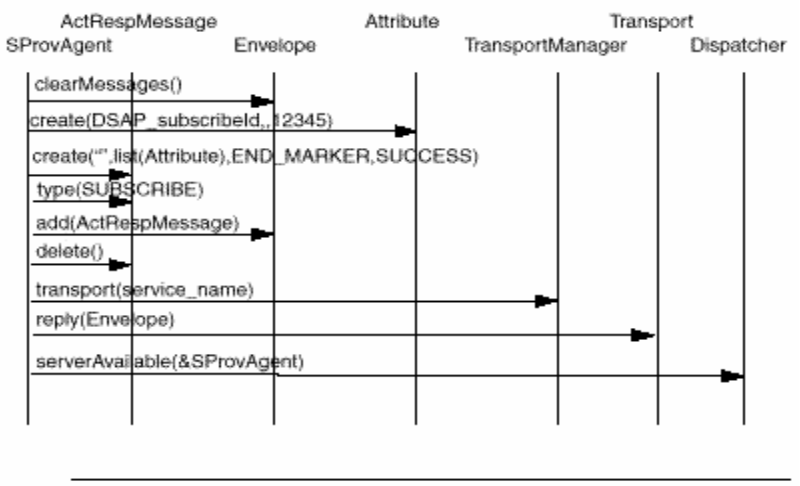
• Figure 37 - Event Trace - Multi-agent Server Processes Service Request

4.1.3.4 Multi-agent Server Sends Subscription Response

If the subscription request indicates confirmation is required, the agent sends a confirmation message to the client, with a message type of Action Response. The subscription service returns a subscriptionId to the client in the confirmation. This subscriptionId is used by the subscriber when terminating the subscription. Figure 38 provides the interactions between the objects.

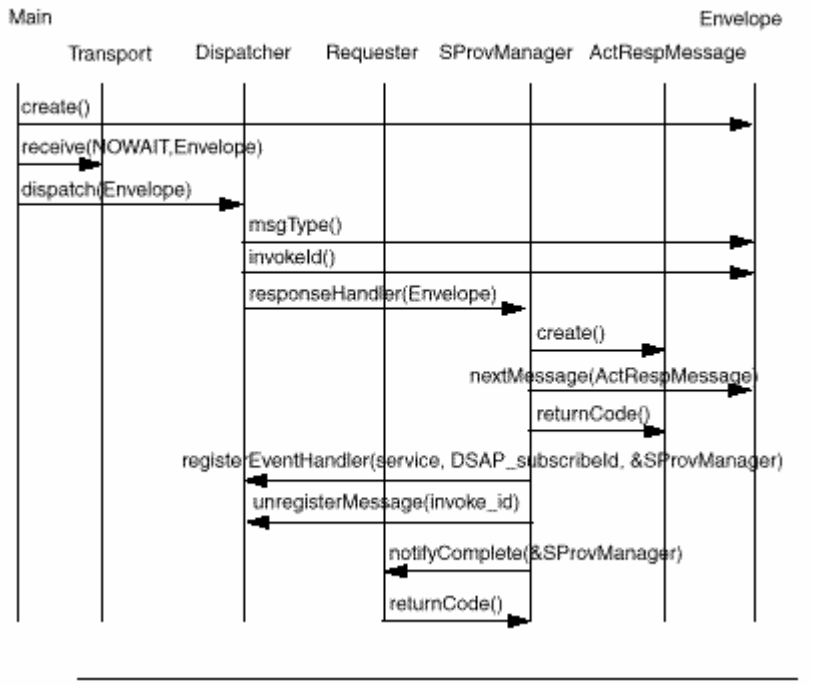
4.1.3.5 Asynchronous Client Handles Subscription Response

As in the previous scenario, the asynchronous client uses a polling method to retrieve messages. Periodically, the client process executes the receive operation on each of the transport objects in the client process and determines if there are any messages. If there are messages, the dispatcher executes a message handling function on the appropriate object. The subscription confirmation is an Action Response message. The dispatcher executes the responseHandler on the appropriate manager, based on invokeId. After the manager receives the subscription confirmation, it must register itself with the dispatcher so that it can receive the Event Report Request messages when they arrive.



• Figure 38 - Event Trace - Multi-agent Server Sends Subscription Response

The manager must also store the subscriptionId in the message to be used later for subscription termination and for registering the eventHandler with the dispatcher. Figure 39 is an event trace describing the sequence of messages between the objects.

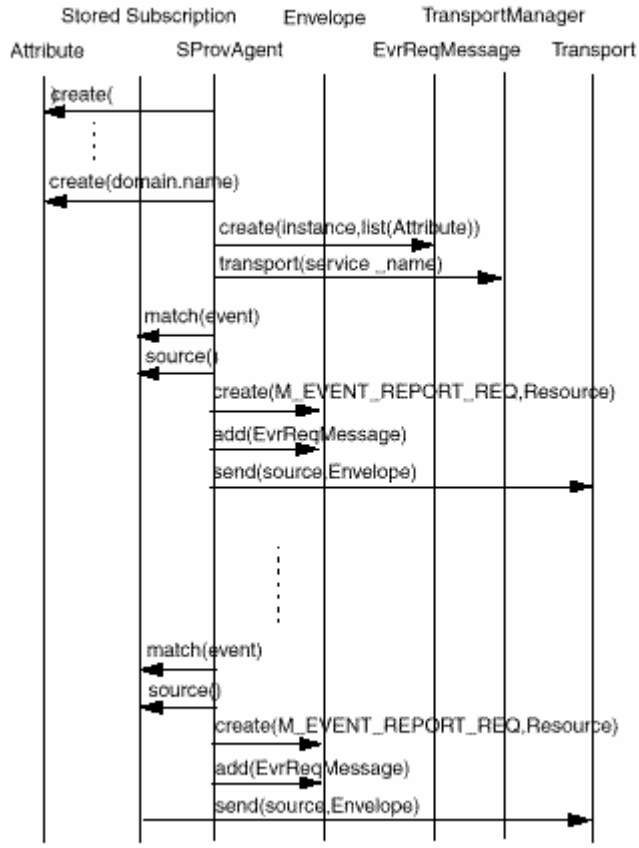


• Figure 39 - Event Trace - Asynchronous Client Processes Subscription Response

4.1.3.6 Event Notification

At some point, an agent providing subscription service is notified that an event has occurred on an object instance(s). The agent determines who has subscribed to this event for this object instance(s) and sends an Event Report Request message to each subscriber. If the subscription

service allows the user to specify the set of attributes to be returned in the Event Report Request message, it may be necessary to create a different message for each subscriber. In the scenario described below, the subscription service does not support this. The Event Report Request message returned to each subscriber has the same set of attributes. Also, in this scenario, the Event Report Request is not confirmed.



• Figure 40 - Multi-agent Server Sends Event Notification

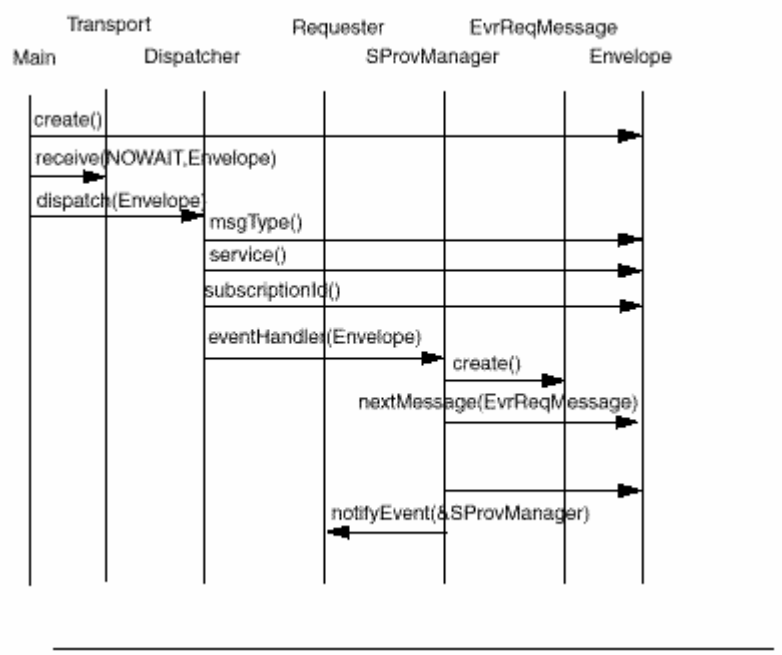
4.1.3.7 Multi-agent Server Sends Event Notification

In this scenario, the agent which sends the event notifications does not permit clients to specify a set of attributes to be returned in the Event Report Request message. Instead, the agent returns a fixed set of attributes in the message.

When the agent is notified of an event on an object, it determines which of its subscriptions matches the event. For each match, the agent creates an Event Report Request message and sends it to the subscriber.

Figure 40 is an event trace which describes the sequence of interactions between the objects. What is not shown is the settings in the resource object in order to simplify the diagram. The resource object is created by the agent and has these parameter settings:

- Priority: NULL_PRIORITY
- Assurance: NO_ASSUR
- UMAction: NULL_ACTION.



• Figure 41 - Event Trace - Asynchronous Client Processes Event Notification

4.1.3.8 Asynchronous Client Processes Event Notification

When an Event Report Request message is received by a client process, the dispatcher determines the service name and subscriptionId in the message and sends the message to the object which has registered an event handler for that service name and subscriptionId. Figure 41 is the event trace which describes the sequence of messages between the classes.

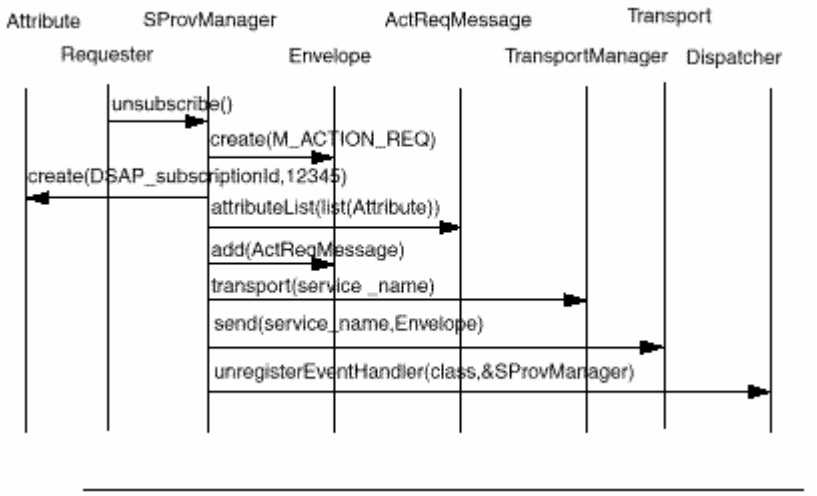
4.1.3.9 Subscription Termination

When a client no longer wants to be notified of events, it must send a message to the server to terminate the subscription. The subscription termination should include the subscriptionId. If a client wants to change subscription, it must first terminate the current subscription and then start the subscription process all over again. As with a subscription request, a client can request that the subscription termination be confirmed. In this scenario, the client will not request confirmation.

4.1.3.10 Asynchronous Client Issues Subscription Termination Request

At some point, the client may wish to terminate the subscription. In the case of a user interface process, if the subscription/event notification is tied to a screen, when the screen is closed, the requester requests that the manager terminate the subscription. The subscriptionId, saved from

the subscription confirmation message, is added as an attribute in the subscription termination message. Figure 42 is the event trace, which describes the sequence of messages between the classes.

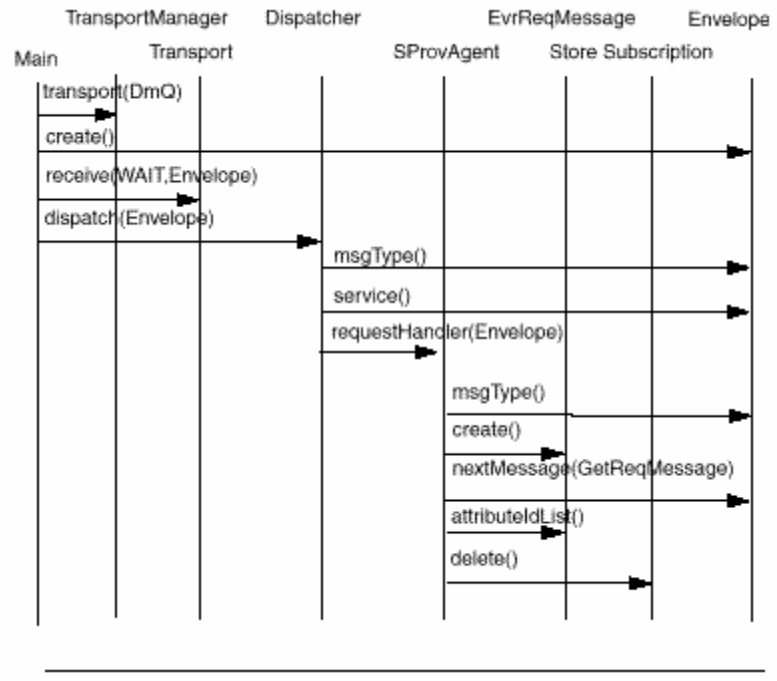


- Figure 42 - Event Trace - Asynchronous Client Issues Subscription Termination Request

4.1.3.11 Server Processes Subscription Termination Request

When the agent who handles the subscription service receives the subscription termination request, it terminates the subscription based on the ObjectQ subscriptionId.

Figure 43 is the event trace, which describes the sequence of messages between the classes.



- Figure 43 - Server Processes Subscription Termination Request

5 ObjectQ MIB Reference

The purpose of this chapter is to provide a reference for the fields in ObjectQ's Management Information Base (MIB) tables.

Once a service is expressed in MIB form, other applications can use the MIB definition, the service's manager, and write client code using ObjectQ classes to access the service. This chapter also contains information about the process of registering the service.

The developer on client system uses ObjectQ classes to request the operations specified in the MIB. The developer of a service uses ObjectQ classes to receive and understand a request and send back a response.

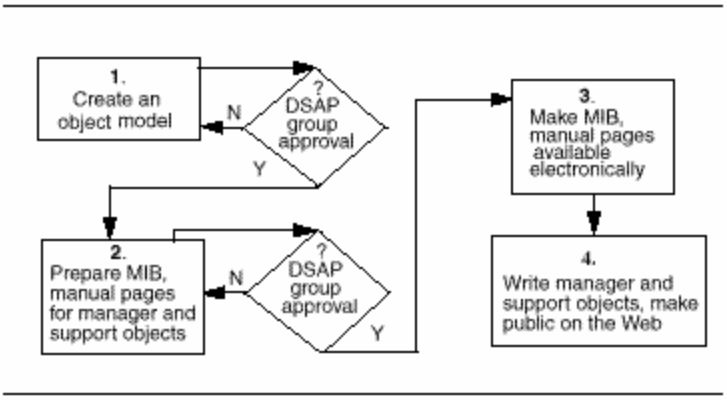
This chapter does not explore issues of object-oriented design, nor does it discuss how to populate MIBs. Examples of MIB usage are on the Web.

5.1 *Making a Service Available*

From a service provider's point of view, what needs to be done to make a service available? What's done with MIB tables once they're filled out? What else is needed in order to make the service available to clients? Figure 44 provides a quick view of the initial steps needed to make a service available.

Initially, a service provider should obtain a registered name for a domain from a central coordinating authority for the enterprise (whose task is to ensure that domain names and registered names are unique, and that MIBs are consistent and non-overlapping).

1. A project develops an object model of its service, then has the object model reviewed and approved by the coordinating authority.
2. A MIB is defined and documented starting from downloaded blank MIB tables. Filled-out MIB tables are submitted for review by the coordinating authority. A project must prepare manual pages for each manager and its support objects, one set for each service provided. Manual pages are also submitted for approval by the coordinating authority: these manual pages can be submitted at the same time or at a different time from the manager MIB tables.
3. Once the MIB tables and manual pages for a service are approved, electronic copies should be made available on the Web.
4. Once the executable manager and support objects are complete, these should also be made public on the Web.



• Figure 44 - Initial Steps in MIB Publication

After these initial stages, prospective clients download the manager code and write interfaces using the MIBs and manual pages provided with the service.

The finished product contains these items:

- manager and support object code librariesthat can be downloaded and used in compilation.
- approved manual pages for the manager and support objects containing information about how to use the manager. Manual pages should go under version control; any changes should go through a change-control process.
- approved versions of MIB tables. MIB tables should go under version control; any changes should go through a change-control process.

5.2 *Management Information Base Tables*

Templates for the MIB tables are in Appendix A, and are obtainable in Word format from the web site. Data that makes up a MIB is contained in five types of tables:

- ***Class Definition Tables*** These tables define the accessible attributes, the actions, and filters for a single class. A template in Word format is available here.
- ***Service Definition Tables*** These tables define a service, which may span a number of classes. One or more containment trees describe the relationships of the classes; other tables list the actions, filters, events, and conversations, which define a service. A template in Word format is available here.
- ***Attribute Data Dictionary*** A single table that specifies the attributes for a domain. The data dictionary contains registered names for the domain and for each attribute in the table, and the type of the attribute. A template in Word format is available here.
- ***Error Table*** A single table that lists the error messages for a service. An identifier number, description, and message type under which each error can occur is listed.

A template in Word format is available here.

- ***Administration Tables*** These are tables that specify a service's administration commands. The name of the command in both string and registered name form, plus the confirmation mode, input/output attribute lists are listed. A template in Word format is available here.

5.2.1 MIBs and ObjectQ Initialization Files

MIB tables form a shorthand way of explaining what a service can provide to a client. These tables have some equivalents in the ObjectQ initialization files (i.e., the files, which provide the running application with attribute, class, error, service names, and administration commands). In particular, the attribute file (<domain>.adf) must bear a likeness to a domain's attribute data dictionary.

Other information in the MIB tables, for example, the instanceIds or the actions listed in the MIB tables, indicates available functionality that is inherent in a service provider's source code and is not provided in any table.

5.2.2 Class Definition MIB Tables

The Class Definition Tables define the accessible attributes, the actions, and filters for a class. The actions and filters described in these tables don't involve other classes.

5.2.2.1 Class Definition Table Fields

- ***Class, Registered Class Name -*** the service provider's name of the class along with the registered name (the `C.D` of the `A.B.C.D` tuple).
- ***Domain, Registered Domain Name -*** the registered name (the `A.B` of any A.B.C.D tuple in the table) of the domain. A registered name for the class is assigned by the coordinating authority.
- ***Derived From*** - which class is this a sub-class of?
- ***Description*** - a short description of the purpose of the class and any important features of the class.
- ***Operations Supported*** - the methods available for a class. Service providers write function(s) for each of these methods, allowing the access to take place. Actions named here are listed under separate tables which indicate the attributes involved on input and output.
- ***Accessible Attributes*** - the attributes that make up the class are listed along with the access allowed for each attribute. Note these issues:
    - attributes in a class can belong to different domains —domains are simply name spaces intended to keep attribute names unique.
    - one attribute (and only one attribute) - may be marked with an asterisk indicating the instanceId for the class.

- ***Get column*** - an ‘O’ in this column indicates that an attribute can be obtained via a get command.
- ***Set column*** - an ‘O’ in either the Replace, Add Value, Remove Value, or Default columns indicates that a value can be changed.
    - ***Replace*** - used for attributes that are not arrays or complex types; specifies replacement of an existing value.
    - ***Add Value, Remove Value*** - used for attributes that are arrays or complex types; specifies addition of a new value or removal of an existing one.
    - ***Default*** - used to specify the attribute's default value.
- ***Create column*** - when a new instance of a class is needed, attributes marked ‘M’ must be provided, those marked ‘O’ are optional.
- ***Defaults/Constraints*** - lists the default values for an attribute, or the ranges of values allowed, or any other limits place on the attribute. Default is designated by D:, constraint by C:.
- ***Comments*** - any other information about the attribute.

5.2.2.2 Class Action Table Fields

Each action listed in the Operations Supported section of the class table should have its own action table. An action is a "method" on the class, and, as such, has input and output parameters.

- ***Action Name/Action Number*** - a name must be supplied that is the same as one listed in the Operations Supported section of the class table. A unique positive number for each action must be provided. This number is used to identify the action in source code.
- ***Description -*** an indication of the activity the action is designed to perform.
- ***Inputs Table***:
    - ***Attribute*** — the domain and name of the attributes that are used in the action as inputs. The attributes are listed as mandatory or optional.
    - ***Default/Constraints*** — default values for the attributes or any limitations (like a range of allowed numbers, for example) that may apply. Default is designated by D:, a limitation or constraint by C:.
- ***Outputs Table*** — the items listed here are the same as the inputs. Notice that if an item is marked ‘M’, it will always be returned as the result of an action.

5.2.2.3 Class Instance Filter Table Fields

Each instance filter for a class must have its own instance filter table.

- ***Filter Number*** - a positive number assigned by the service provider is entered

here. This number is passed between processes and identifies the filter to those processes.

- ***Description*** - an indication of the purpose of the filter.
- ***Table:***
    - ***Attribute*** - the domain and name of the attributes that are used in the filter.
    - ***Op (Operator)*** - can be one of these: =, >, >=, <, <=, !=.
    - ***M/O (Mandatory/Optional)*** - the attributes are listed as mandatory or optional in the filter.
    - ***Defaults/Constraints*** - the default values for the attributes or any limitations that may apply. Default is designated by D:, constraint by C:.
    - ***Comments*** - any other information about the attribute used in the filter.

5.2.2.4 Class Event Filter Table Fields

Each event filter for a class must have its own event filter table.

- ***Filter Number*** - a number assigned by the service provider is entered here. This number is passed between processes and identifies the filter to those processes.
- ***Description*** - an indication of the purpose of the filter.
- ***Table***:
    - ***Attribute*** - the domain and name of the attributes that are used in the filter.
    - ***Op (Operator)*** - can be one of these: =, >, >=, <, <=,!=.
    - ***M/O (Mandatory/Optional)*** - the attributes are listed as mandatory or optional in the filter.
    - ***Defaults/Constraints*** - the default values for the attributes or any limitations that may apply. Default is designated by D:, constraint by C:.
    - **Comments** - any other information about the attribute used in the filter.

5.2.3 Service Definition MIB Tables

Service MIB tables are used to define the way that the activities in individual classes work together. If an action or a filter works on only one class, that action or filter should be defined in class definition tables; however, if the action or filter works on more than one class, that action or filter should be defined as part of the service definition tables.

Central to the issue of working with more than one class is a search path through a service's classes that indicates how some desired objective, i.e., a particular attribute, is reached.

5.2.3.1 Service Containment Trees

Relationships between a service's classes are defined in containment trees.

The first line of the table names the service. At the top left of the table is the name of the service; on the top right of the table is the registered name.

- ***Service, Registered Service Name*** — the service provider's name of the service along with the registered name (the C.D of the A.B.C.D tuple) furnished by the service provider.
- ***Domain, Registered Domain Name*** — the registered name (the A.B of the A.B.C.D tuple) of the domain. The coordinating authority assigns a registered name for the service.
- ***Administration Commands*** — a list of the commands, each of which is defined in an administration table, for this domain.
- ***Definition*** —a short description of the purpose for the service and any important features of the service
- ***Classes*** — a list of the classes, in domainName.className form that are used in the service. Each of these classes has class tables for its own definitions, actions, etc.
- ***Containment Trees*** — each containment tree used in a service. Containment trees fulfil two purposes:
    - Each class in a containment tree describes its relationship to the other classes, that is, a one-to-many, one-to-one relationship.
    - Each class also shows the direction a search for specific attributes takes.

5.2.3.2 Service Instance Filter Table Fields

Each instance filter for a service must have its own instance filter table.

- ***Filter Number*** - a number assigned by the service provider is entered here. This number is passed between processes and identifies the filter to those processes.
- ***Applies to Tree Number(s)*** - which containment tree the filter works against .
- ***Description*** - an indication of the purpose of the filter.
- ***Table***:
    - ***Attribute*** - the domain and name of the attributes that are used in the filter.
    - ***Op (Operator)*** - can be one of the these: =, >, >=, <, <=, !=.
    - ***M/O (Mandatory/Optional)*** - the attributes are listed as mandatory or optional in the filter.

- ***Defaults/Constraints*** - the default values for the attributes or any limitations that may apply. Default is designated by D:, constraint by C:.
- ***Comments*** - any other information about the attribute used in the filter.

5.2.3.3 Service Event Filter Table Fields

Each event filter for a service must have its own event filter table.

- ***Filter Number*** - a number assigned by the service provider is entered here. This number is passed between processes and identifies the filter to those processes.
- ***Applies to Tree Number(s)*** - which containment tree the filter works against .
- ***Description*** - an indication of the purpose of the filter.
- ***Table***:
  - ***Attribute*** - the domain and name of the attributes that are used in the filter.
  - ***Op (Operator)*** - can be one of the these: =, >, >=, <, <=,!=
  - ***M/O (Mandatory/Optional)*** - the attributes are listed as mandatory or optional in the filter.
  - ***Defaults/Constraints*** - the default values for the attributes or any limitations that may apply. Default is designated by D:, constraint by C:.
  - ***Comments*** - any other information about the attribute used in the filter.

5.2.3.4 Service Action Table Fields

Each action provided as part of a service should have its own action table.

- ***Action Name/Action Number*** - a name and unique positive number for each action must be provided. This number is used to identify the action in source code.
- ***Applies to Tree Number(s)*** - which containment tree the action works against .
- ***Description*** - an indication of the activity the action is designed to perform.
- ***Inputs Table*** :
  - ***Attribute*** - the domain and name of the attributes that are used in the action as inputs. The attributes are listed as mandatory or optional. The next column of the table is the place to list the default values for the attributes or any limitations (like a range of allowed numbers, for example) that may apply. Default is designated by D:, limitation or constraint by C:.
- ***Outputs Table*** - the items listed here are the same as the inputs. Notice that if an item is marked ‘M’, it will always be returned as the result of an action.

5.2.3.5 Service Event Report Request Table Fields

Each event report request for a service must have its own event report request table.

- ***Event Filter Number*** - a number assigned by the service provider to identify the event.
- ***Table:***
    - ***Class and Attribute*** - the names of the items that can appear as data in this report request.
    - ***M/O (Mandatory/Optional)*** - the attributes are listed as mandatory (always sent) or optional (sent only on request) in the report**.**
    - ***Defaults/Constraints*** - the default values for the attributes or any limitations that may apply. Default is designated by D:, limitation or constraint by C:.
    - ***Comments*** - any other information about the attribute used in the filter.

5.2.3.6 Service Conversations Table Fields

Providing a conversational service means that a server must maintain state information based on the needs of a client.

The conversations table lists all the messages that can be passed back and forth during a particular conversational service. A service can provide multiple conversational scenarios; each must have its own table listing all the possible messages for that particular conversational activity.

- ***Conversation Number*** - a number assigned by the service provider to identify the conversation service.
- ***Description*** - an indication of the purpose of the conversation scenario.
- ***Table***:
    - ***Message number*** - a unique identifier for each message. These should be listed in the likely order they occur, with the first message being an indication to set up the conversational service (i.e., "start conversation"), and the last indicating its termination ("end conversation").
    - ***Class*** - the domain and name of the class for this message number.
    - ***Tree Number*** - which containment tree the activity works against .
    - ***Get/Set/Create/Delete*** - marked as either Mandatory or Optional for the specific message number.
    - ***Action Number*** - numbers that refer to the service action table.

- ***Comments*** - any other information about the conversational message.

5.2.4 Domain Attribute Data Dictionary

The domain attribute data dictionary is used to identify individual attributes available as part of a domain as to their names, registered names, and types.

5.2.4.1 Domain Data Dictionary Table Fields

Each domain attribute data dictionary identifies the domain and the registered name (A.B) of the domain in the heading area.

In the table itself:

- ***Attribute*** - the name assigned to an attribute. For this attribute data dictionary, names of attributes do not have to be unique since ObjectQ's attribute naming methodology allows the attribute name to be prefixed with the class name that the attribute belongs to. For example, there can be two instances of the name *address* in a single data dictionary, each one belonging to a different class. In ObjectQ code, the class name must be placed in front of the attribute name to assure uniqueness (that is, the attribute would be named *domainName.className.domainName.attributeName*).
- ***Registered name*** - each attribute has a unique registered name assigned to it by the service provider. These registered names are in the form C.D. They are combined with the domain name (A.B) to form a unique registered name for each attribute.
- ***Type*** - these entries are valid:
  - char, short, long, string, octetstring
  - complex — made up of some combination of attributes. The attribute names that make up this item go in the attribute list.
  - array — made up all of the same attribute, named in the attribute list column.
  - refer — the same definition as the referred-to attribute. The attribute name for this item go in the attribute list column.
- ***Comments*** — any other useful information about this attribute.

5.2.4.2 Domain Error Table

Error messages pertaining to a service are defined in a domain error table. Error codes are passed in the errors/error list section of the message header. The errors/error list field is made up of a return code plus up to 3 error codes. The return code should be simply `cpMSUCCESS` or `cpMFAIL` (and not `cpSUCCESS` or `cpFAIL`, which are cpStatus values). See the cpMessage class manual page for a listing of the CMIS error codes that may be used.

5.2.4.3 Domain Error Table Fields

- ***Name*** - for situations, in which CMIS error codes are not appropriate, a service's own error codes can be passed in the message header.
- ***Description*** - the text of a corresponding error message. This description should give a clear explanation of the error, possible causes, and possible corrective actions.
- ***Get/Set/Create/Delete/Action*** - the message type in which the error may occur. Certain error numbers can have meaning for only certain message types and not for others.

5.2.5 Domain Administration Commands Table

Each administration command pertaining to a service is defined in a domain administration commands table.

Administration commands are passed through administration request and response messages. See the cpMessage manual pages for information on the specifics of administration messages.

5.2.5.1 Domain Administration Commands Table Fields

- ***Administration Command Name/Number*** — a name and unique registered name must be assigned to each administration command.
- ***Description*** — a description of what the command does.
- ***Mode*** — must be one of ALWAYS, NEVER, MODE, where ALWAYS indicates that a confirmation (response) is always required; NEVER indicates that confirmation is not expected; MODE indicates that the client/requester is allowed to set the mode field in the request message.
- ***Inputs Table*** —
    - ***Attribute*** — the domain and name of the attributes that are used in the command as inputs. The attributes are listed as mandatory or optional. The next column of the table is the place to list the default values for the attributes or any limitations (like a range of allowed numbers, for example) that may apply. Default is designated by D:, limitation or constraint by C:.
- ***Outputs Table*** — the items listed here are the same as the inputs. Notice that if an item is marked 'M', it will always be returned as the result of an request.

6 ObjectQ Administration

The purpose of this chapter is to provide a description of the administrative files and procedures required by an ObjectQ environment.

6.1 *Tracing and Log Files*

One of the parameters to the *cpInit* function (which must be called by an application before any other ObjectQ function is called) is a numeric trace level. As a result of the value of this parameter, two files may be produced:

- A trace file, named cpTRACEFILE.<pid>, containing debugging trace that will usually only be of interest when ObjectQ's behavior is unexpected.
- A log file, named cpLOGFILE.<pid>, which is used to log events that an application developer or system administrator should be aware of.

where <*pid*> is the process id of the application. These files appear in the directory that the application is running in, unless the environment variable *cpTRACEDIR* is set, in which case they will appear in that directory.

A trace level less than 0 will cause neither of these files to be generated. A trace level of 0 generates a log file, but no trace file. And a trace level between 1 and 5 generates an increasingly verbose trace file (4 is usually the most instructive).

6.2 *The* regdomain *File*

The *regdomain* file will ideally be identical throughout an enterprise, and is a master list of domain names and associated registered names. This file will usually be maintained by a central coordinating authority within the enterprise, whose job is to ensure that all domains and registered names are unique.

The *cpInit* function reads this file from either the current directory (in which case the file must be named *regdomain*), or from the file specified by the *REGDOMAIN* environment variable.

Entries in the file are simple ASCII white-space-separated fields:

- DSAP 1.1
- DSTS 2.1
- ESCAP 3.1
- Etc.

6.3 *MIB Files*

MIB data files are all ASCII files containing white-space-separated fields, and are read in by the *cpInit* function from the directories specified in its parameter list. Note that all files

conforming to the naming convention are read, so that a file will not be hidden by renaming it from, say, *foo.adf* to *old.foo.adf*; rename it to *foo.adf.old* instead.

In all cases a line beginning with the character # is regarded as a comment, and is ignored.

6.4 *Attribute Definition Files*

An attribute definition file has the name *domain.adf*, where *domain* is the name of the domain for which the attributes are being defined. It has four fields:

- ***Name*** - the name of the attribute.
- ***Registered name*** — the associated registered name, which should be unique within the table.
- ***Type*** - one of *CHAR, SHORT, LONG, STRING, OCTETSTRING, COMPLEX, ARRAY* or *REFER*.
- ***Attribute List*** - for *ARRAY*, the full registered name of the attribute which represents elements within the array; for *COMPLEX*, a comma-separated list (with no intervening spaces or newlines) of the full registered name of the attributes that constitute the components of the complex; and for *REFER*, the full registered name of the attribute referred to.

A fragment of the *DSAP.adf* file (where DSAP's registered name is 1.1) looks like:

- date 1.1 COMPLEX 1.1.1.6,1.1.1.2,1.1.1.10
- day 1.2 SHORT
- month 1.6 SHORT
- year 1.10 SHORT

6.5 *Service Definition File*

A service definition file has the name *domain.sdf*, where *domain* is the name of the domain for which the services are being defined. It has four fields:

- ***Name*** - the name of the service.
- ***Queue name*** - no longer used (but must be present).
- ***Registered name*** - the associated registered name, which should be unique within the table.
- ***Vendor*** - no longer used (and need not be present).

The service definition file *Demo.sdf* used by the sample code (where Demo's registered name is 2.1) looks like:

- Sales NAME1 1.1

- Sales1 NAME2 2.1

6.6 *Class Definition Files*

A class definition file has the name *domain.cdf*, where *domain* is the name of the domain for which the classes are being defined. It has two fields:

- ***Name*** — the name of the class.
- ***Registered name*** — the associated registered name, which should be unique within the table.

The class definition file *Demo.cdf* used by the sample code (where Demo's registered name is 2.1) looks like:

- Customer 1.1
- Account 2.1
- Order 3.1
- Bill 4.1

6.7 *Error Definition Files*

An error definition file has the name *domain.edf*, where *domain* is the name of the domain for which the errors are being defined. It has three fields:

- ***Name*** - the name of the error.
- ***Registered name*** - the associated registered name, which should be unique within the table.
- ***Description*** - a quoted text string.

A portion of the error definition file *Demo.edf* used by the sample code (where Demo's registered name is 2.1) looks like:

- 103 1.3 "No such order or no order found"
- 104 1.4 "No such bill or no bill found"
- 105 1.5 "Invalid status code"

Note that the use of numbers for the name is arbitrary — within code, they are still referred to as strings: "*Demo.103*", for example.

6.8 *Administration Command Definition Files*

An administration command definition file has the name *domain.cdf*, where *domain* is the name of the domain for which the commands are being defined. It has two fields:

- ***Name*** - the name of the command.
- ***Registered name*** - the associated registered name, which should be unique within

the table.

The *DSAP.mdf* file (where DSAP's registered name is 1.1) looks like:

- echo 100.1

6.9 *Transaction Definition Files*

A transaction definition file has the name *foo.xdf*, where foo can be any name (in other words, transaction definition files are not associated with a particular domain). It specifies the mapping between attributes and a flat record structure consisting of (potentially) repeated fields.

The first entry in the file should be a transaction name (maximum 8 characters) that will be prepended to the record. Within a transaction, an arbitrary number of sub-fields are defined. Each sub-field specification must be preceded by its name (maximum 8 characters), and contains three fields:

- ***Domain*** - the domain of the attribute being mapped.
- ***Name*** - the name of the attribute being mapped.
- ***Format*** - the format of the field in the record:
  - ***x(n)*** alphabetic data, left justified, space filled of total width n
  - ***u(n)*** unsigned numeric, right justified, zero filled of total width n
  - ***s(n)*** signed numeric, right justified, zero filled of total width n (including the sign)
  - Each of these may be followed by a repetition factor, enclosed in square brackets.

An example transaction definition file looks like:

```
# Transaction name
xact-1 {
# Field name
xact1-a
# First sub-field - alphabetic in the first 16 bytes
tt              day             x(16)
# Second sub-field - alphabetic in the next 16 bytes
tt              month         x(16)
# Third sub-field - alphabetic in the next 16 bytes
tt              year            x(16)
}
```

6.10 *Service Name Resolution Files*

Service name resolution (SNR) data files contain data on one service. The file name must have the format *domainName_serviceName.snr*. The information in the file is divided into sections, one section for the service specific data and one or more sections for location specific information. Each section is bracketed with a *START* and *END* tag. In addition, location sections have a unique location tag for identification. The location tag can be any alphanumeric string.

The data items in the file are in name-value pairs. For each name, the valid value(s) are listed between <>, with a | separating the choices. Optional data items are shown in []. Comment lines begin with a # sign. The format of the file is listed below. Following that is a description of each line in the file:

```
START SERVICE
DSAPRelease: <REL3.1 | REL4.0 | REL4.1.5>
[defaultVendor: <DMQ | MQS | TUXEDO | HTTP>]
[selection: <WAVG>]
[timeout: <dd.hh.mm.ss>]
[maxMessageSize: <max size>]
 [publicKey: <public key for encryption>]
END SERVICE
START LOCATION <location tag>
destName: <queue | queue alias>
[destType: <QUEUE | ALIAS>]
[vendor: <DMQ | MQS | TUXEDO | HTTP>]
[state: <AVAIL | UNAVAIL | QUIESCENT>]
[weight: <positive integer>]
END LOCATION

START LOCATION <location tag2>
...
END LOCATION

etc.
```

Each line in the file is described below:

- ***START SERVICE***: Start of the service data section. This is required.
- ***DSAPRelease***: The DSAP release supported by the service. This is a required field. For releases earlier than release 3.1, use REL3.1; for releases 4.0 through 4.1.4 use REL4.0; for releases 4.1.5 and beyond use REL4.1.5. This is used for inter-release compatibility.
- ***defaultVendor***: The default transport vendor for the service. This is an optional field. The valid values are:
  - *DMQ*: DECmessageQ
  - *MQS*: IBM MQSeries
  - *HTTP*: http

- *TUXEDO*: Tuxedo
- The default value is *DMQ*.

- ***selection***: The selection method to use when selecting a location. This is an optional field. The only current valid value is:
  - WAVG: weighted average
- ***time-out***: This is an optional field. Its format is <days>.<hours>.<minutes>.<seconds>, where each elements is >= 0. If used, it specifies that stored SNR data for this service becomes stale after the specified time interval. The default value is 0.0.0.0. This indicates there is no time-out for this service.
- ***maxMessageSize***: This is the largest envelope that should be sent. This is an optional field. If it is not provided, the maximum envelope size will be based on the maximum size for the default transport vendor.
- ***publicKey***: This is the public key for the service that will be used to encrypt outgoing requests. The key is specified as an unbroken sequence of hexadecimal bytes (for example, 0f1c2dfe10...) representing an X.509 RSAPublicKey.
- ***END SERVICE***: End of the service data section. This is required.
- ***START LOCATION <location_tag>***: Start of a location section. Each location section is required to have a unique tag (unique within the file).
- ***destName***: Delivery point for the service. For queue based messaging products, either queue name or queue alias. This is a required field.
- ***destType***: This specifies whether the destName is a queue name or queue alias. This is an optional field. The valid values are:
  - *QUEUE*: destName is a queue name
  - *ALIAS*: destName is a queue alias
  - The default value is *ALIAS*.
- ***vendor***: Transport vendor for this location. This is an optional field. The valid values are:
  - *DMQ*: DECmessageQ
  - *MQS*: IBM MQSeries
  - *HTTP*: http
  - *TUXEDO*: Tuxedo
  - The default value is *DMQ*.

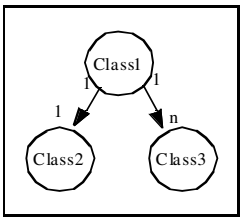
- ***state***: The state of the service at this location. This is an optional field. The valid values are:
  - *AVAIL*: The service is available.
  - *UNAVAIL*: The service is not available.
  - *QUIESCENT*: The service is up but is not accepting new requests.
  - The default value is *AVAIL*.
- ***weight***: The weight assigned to this location. It is used by the weighted average selection method. This field is optional. If no weight is supplied, equal weights will be assumed for all locations. The weight is an integer in the range 1-1000.
- ***END LOCATION <location_tag>***: End of a location section. This line is required, though the location tag is optional.

7 MIB Templates

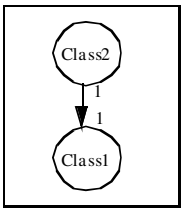
7.1 Domain.Service - *Service Definition:* A.B.C.D

- Service: *ServiceName* Registered Service Name: *C.D*
- Domain: *DomainName* Registered Domain Name: *A.B*
- Definition:
- Classes: *Domain.Class1, Domain.Class2, Domain.Class3, etc.*
- Administration Commands: *Domain.Admin1, Domain.Admin2, etc.*
- Containment Trees:

Tree Number: 1



Tree Number: 2



7.2 Domain.Service - *Service Definition:* A.B.C.D

Instance Filters

- Filter Number:
- Applies to Tree Number(s):
- Description:

Class	Attribute			Default/	Comments

						Constraints	

Op Operator: =, >, >=, <, <=, !=

M/O Mandatory/Optional

7.3 Domain.Service - *Service Definition:* A.B.C.D

Event Filters

- Filter Number:
- Applies to Tree Number(s):
- Description:

Class		Attribute				Default/ Constraints	Comments

Op Operator: =, >, >=, <, <=, !=

M/O Mandatory/Optional

7.4 Domain.Service - *Service Definition:* A.B.C.D

Actions

- **Action Name:** **Action Number:**
- **Applies to Tree(s):**
- **Description:**

Inputs

Class		Attribute			Defaults/ Constraints	Comments
Domain	Name	Domain	Name			

Outputs						
Class		Attribute			Defaults/ Constraints	Comments
Domain	Name	Domain	Name			

M/O Mandatory/Optional

7.5 Domain.Service - *Service Definition:* A.B.C.D

Event Report Request

- Event Filter Number:

Class		Attribute			Defaults/ Constraints	Comments
Domain	Name	Domain	Name			

M/O Mandatory/Optional

7.6 Domain.Service - *Service Definition:* A.B.C.D

Conversations

- Conversation Number:
- Description:

Message Number	Class	Tree Number	Get	Set	Create		Delete	Action Number
	Domain	Name						

Get, Set, Create, Delete: Use X to indicate this message can be used as part of the conversation.

Action: List the action number(s), if any, that can be used as part of the conversation.

7.7 Domain.Class - *Class Definition:* A.B.C.D

- **Class:** *ClassName* **Registered Class Name:** *C.D*
- **Domain:** *DomainName* **Registered Domain Name:** *A.B*
- **Derived From:** *DomainName.ClassName*
- **Description:**
- **Operations Supported:**
  - Get
  - Set
  - Create
  - Delete
  - Actions
    - Action1
    - Action2
    - etc.

7.8 Domain.Class - *Class Definition:* A.B.C.D

Attributes

Accessible Attributes									
Attribute		**Access**						**Defaults/ Constraints**	**Comments**
			Set						
					Remove Value				

- ***Name*** Use * symbol to identify which attribute is the Instance Identifier
- ***Get*** Use O to indicate an attribute is accessible. Use M to indicate an attribute is always returned on every Get request.
- ***Set*** Use O to indicate an attribute can be set. Use M to indicate an attribute is required on all set requests.
- ***Create*** Use O to indicate an attribute can be supplied during a create request. Use M to indicate an attribute is required during a create request.

7.9 Domain.Class - *Class Definition:* A.B.C.D

Actions

- Action Name: Action Number:
- Description:

Inputs				
Attribute			Default/Constraints	Comments
	Name			

Outputs				
Attribute			Default/Constraints	Comments
Domain	Name			

M/O Mandatory/Optional

7.10Domain.Class - *Class Definition:* A.B.C.D

Instance Filters

- Filter Number:
- Description:

Attribute		Op		Default/ Constraints	Comments
Domain	Name				

Op Operator: =, >, >=, <, <=, !=

M/O Mandatory/Optional

7.11Domain.Class - *Class Definition:* A.B.C.D

Event Filters

- Filter Number:
- Description:

Attribute				Default/ Constraints	Comments
Domain	Name				

Op Operator: =, >, >=, <, <=, !=

M/O Mandatory/Optional

7.12Domain - *Error Definition:* A.B

Errors							
Error Name	Registered Name	Description	Get		Create	Delete	Action

7.13Domain - *Administration Command Definition:* A.B

- Administration Command Name: Registered Name:
- Domain Name: Registered Domain Name:
- Description:
- Confirmed: <*ALWAYS|NEVER|MODE*>

Inputs				
Attribute			Default/Constraints	Comments
Domain	Name			

Outputs				
Attribute			Default/Constraints	Comments
Domain	Name			

M/O Mandatory/Optional

7.14Domain - *Attribute Definition:* A.B

Attribute Data Dictionary				
Attribute	**Registered Name**	**Type**	**Attribute List**	**Comments**

Type One of the following types: CHAR, SHORT, LONG, STRING, OCTETSTRING, COMPLEX, ARRAY, REFER

List The list of attributes which completes the definition of an attribute of type COMPLEX, ARRAY or REFER