U1 TECHNOLOGIES

AmbrosiaMQ:

An Event Management System

Concepts and Capabilities Guide

April 2009

Documentation Revision: 042409

U1 Technologies
204 Tiburon Boulevard
San Rafael, CA 94901

phone: (415) 480-0318
fax: (415) 704-3275
web: http://www.u1.com
e-mail: info@u1.com

Contents

Preface

About this Manual

This manual, the AmbrosiaMQ *Concepts and Capabilities Guide,* introduces the concepts underlying the AmbrosiaMQ event management system. A comprehensive treatment of these concepts provides a useful background for systems analysts, developers, and project managers who wish to learn how to leverage AmbrosiaMQ to develop distributed business applications. The concepts discussed here provide an essential groundwork for application architects and developers who will use these key concepts in the design and development of AmbrosiaMQ business applications. This conceptual groundwork leads naturally into the use of AmbrosiaMQ for building secure, globally deployed Internet/Intranet enterprise class applications.

Documentation Set

The *Administrator's Guide* discusses system configuration and maintenance of AmbrosiaMQ. Consult this manual for instructions on installation procedure, message broker configuration, guaranteed delivery log file administration, use of the Administration Console, and security.

Other documentation for the software includes the Javadoc generated from the com.u1.client package as well as the directory of sample programs with their associated readme files. We encourage you to review these samples before writing your own applications.

Feedback Welcome

Your comments are valuable to us. Your opinions represent the most important input that we receive for the next generation of our documentation. We want to know what we can do to improve this manual. You can send your comments by electronic mail to us at:

AmbrosiaMQsupport@u1.com

Or, you can mail your comments to:

AmbrosiaMQ Technical Publications Feedback
U1 Technologies.
204 Tiburon Blvd
San Rafael, CA 94901

Naming Conventions

This manual denotes Interfaces with the prefix character `I` and Exceptions with the character `E`. In general, this manual adheres to standard Java naming conventions for classes and methods.

Documentation Conventions

The manual uses the following general conventions:

- Italics often designate a reference to other information. Used alone, italics refer to the title of another manual; for example:

  Refer to the *Administrator's Guide* for additional information.

- References to another chapter within this manual start with the chapter number and appear in quotations; for example:

  "Chapter 1: Installation," includes the detailed procedures for installing AmbrosiaMQ.

- References to other sections within a chapter use quotations; for example:

  Refer to "Running the Message Broker as a Windows Service," below, for more information.

Typographical Conventions

The manual uses the following typographical conventions:

- **Bold** emphasis on regular type represents a "button" on a graphical user interface window that performs an action; for example:

  Click **Subjects** to invoke the *Selector* window:

- When describing GUI windows, quotation marks enclosing regular text indicate displayed text; for example:

  "Members" lists all current group members.

  In the above example, "bill," "madeline," and "marshall" are group members of the group "BillsCo."

- Angle brackets (<>) enclose names of keyboard keys or required parameters that are user-specific; for example:

  Enter the name of the new configuration file and press <Enter>.
  Replace "<key>" with your license key.

- Within code examples, angle brackets enclose required actions that are user specific; for example:

<prepare other RMs>

- Terms with their initial letter capitalized indicate Java objects, and appended closed parentheses indicate Java method calls; for example:

  When you publish a Message, it must have a subject. Use `Session.subscribe()` on your `Session` object.

- Courier font represents AmbrosiaMQ and Java classes, methods, arguments, and exceptions; for example:

  To allow subscription to remain in effect, the boolean parameter `forceUnsubscribeAll` of the method `disconnect()` needs to be set to `true`.

1 Introduction

Application developers have learned over the years that common services surrounding an application — resource and network reliability, security, and transactional capabilities — are critical to an application's overall success. These capabilities are especially important when developing enterprise-wide distributed systems that must be fault tolerant and scale to many concurrent users. Most application developers add their greatest value when they focus their time, energy, and resources on business logic rather than on low-level infrastructure. The need for reusable, secure, reliable, and scalable communications services led to the creation of a class of software known as enterprise messaging. Enterprise messaging insulates developers from infrastructure-level programming, and allows them to concentrate their development efforts on business logic and application level features.

AmbrosiaMQ addresses this need, with a communications infrastructure that is proven to be secure, scalable, high performing, efficient, and reliable for the most demanding enterprise applications.

AmbrosiaMQ – The Event Management System of Choice

The AmbrosiaMQ event management system (EMS) facilitates the development of secure, scalable enterprise applications that span a wide variety of network topologies including: LANs, WANs, and the Internet. The AmbrosiaMQ EMS employs an event-based communications model crafted for business applications and provides the seamless exchange of information between highly distributed system elements. AmbrosiaMQ is also commonly referred to as a “Messaging System”; in this document, the terms “Event Management System” and “Messaging System” are synonymous.

AmbrosiaMQ satisfies two basic requirements for an organization’s computing architecture: a short term need for the rapid development and delivery of applications, and a long term need for these applications to be adaptable to changing business requirements. AmbrosiaMQ satisfies both of these requirements by providing a straightforward Application Programming Interface (API), along with a set of core services that applications can leverage.

In addition to these general characteristics, AmbrosiaMQ specifically offers the robust communications functionality you expect by supplying critical network services such as guaranteed message delivery, security, load balancing, and transactional capabilities, all out-of-the box. Until now, the effort required to build applications containing these features presented developers with a choice between investing significant time and resources at the outset, or abandoning these features altogether. With AmbrosiaMQ, business application developers can focus on solving business problems, not developing application infrastructure.

Feature Overview of AmbrosiaMQ

Applications that employ IP networks for reliable connectivity to services need robust communications beyond that which the low-level TCP/IP protocol provides. Messaging products bridge the application and the network, and allow thin clients to offload much of the message processing to external servers (message brokers). This section provides an overview of the wide range of features that AmbrosiaMQ provides to a system. AmbrosiaMQ is ideally suited for the high demands of large-scale systems.

Quality of service

- Reliable delivery – at most once
- Discardable delivery – allows dropping messages for slow clients
- At least once delivery (guaranteed)
- Exactly once delivery
- Message priorities
- Synchronous / asynchronous publishing
- Distributed transaction support with two phase commit
- Message order preservation
- Message expiration
- Virtual Circuit support for communication interruption notification
- Most Recent Value (MRV) (Summer 2007)

Flow Control

- Queuing: client side, broker-to-client, outgoing queue on client
- Full queues: control flow on publisher, subscriber kickoff, or drop messages
- Client side queue size configuration
- Client side queue statistics

Scalability

- Flexible interbroker network architecture
- Global data kept to minimum (subscriptions, active user list).
- Route limits (local, collective, global)
- Multicast like delivery for messages that have more than one subscriber. Single message transmitted on shared route segments.
- Message delivery only to active client (no client side filtering).

Speed

- Efficient wire protocols
- Use of hashing for subject efficiency.
- Fast serialization
- Efficient search algorithms to find subscriptions and ACLs.
- Stream data delivery
- Large result sets
- Static data encoding

Security

- Strong native authentication via a challenge/response protocol in which the passwords are never sent on the wire
- Security database: usernames/passwords, groups, ACLs.
- Encrypted wire protocol (SSL)
- JAAS-based authentication for integration with other authentication mechanisms
- Duplicate connection checker
- Embedded principal ACLs

Fault tolerance

- Redundant configuration servers
- Load balanced connections
- Subscription recovery
- Load balanced subscriptions provide transparent failover capability
- Weighted load balanced Virtual Circuits
- Restart brokers in any order
- Virtual Circuit connection manager

API

- 100% Java client API.
- C and C++ client API
- JMS API

Special Features

- Exposed security API
- JAAS-based authentication

- Point-to-point Round Trip Time monitoring API
- Virtual Circuit service discovery
- Array list subscriptions
- Subscription reference counting
- Subject space API
- Enhanced Selectors including support for regular expressions, date, and time.
- Multiple acceptors/protocols
- Wildcard publishing
- Client heartbeat monitoring
- Centralized log monitoring

Advantages of AmbrosiaMQ

Developers gain a number of advantages by using AmbrosiaMQ. In addition to reducing initial development time, AmbrosiaMQ imparts significant network functionality to applications. Centralized control allows a system administrator to monitor and modify the event management system from a single console. Increased capabilities and enhanced functionality ultimately benefit the end users. In short, AmbrosiaMQ offers the following features:

- Event-Based Communications Infrastructure
- 100% Pure Java Client Interface
- Decoupled Development
- Abstraction of Communications
- Location Transparency
- Modular Architecture
- Administration Services
- Push and Respond
- Quality of Service
- Transactional Services
- Comprehensive Security
- Performance Oriented
- Multiple Wire Protocols
- Intra-process Messaging

Benefits

Event-Based Communications Infrastructure

To more closely model how business systems function, software developers are adopting the publish/subscribe communications model because it supports *event-based computing.* By mirroring the way in which information flows in the business world, applications can approach the real-time processing capability that businesses desire. The event-based computing model automatically notifies all active subscribers that an event has occurred, without requiring those subscribers to continuously poll a central repository for the information.

Here, AmbrosiaMQ avoids the fundamental problem with polling-based implementations: if an application wants to approach "real-time" response to events, it must poll the repository with increasing frequency. This increases server load and network bandwidth requirements. AmbrosiaMQ's event-based approach provides a much better, more efficient solution for all *Internet* applications (as used in this manual, Internet includes intranet, extranet, and any TCP/IP network).

100% Pure Java Interface

For Internet applications, Java is the language of choice. AmbrosiaMQ was designed from the ground up for Java development and earned one of the first certifications as 100% Pure Java. Leveraging Java offers applications cross-platform support and, in the case of a downloaded applet, removes the need to pre-provision client computers with application code. This reduces client administration costs and overhead, while increasing application accessibility.

Decoupled Development

Traditionally, distributed applications required tightly coupled communications, where each application must understand the data formats, semantics, and other low-level details of the other applications with which it communicates. AmbrosiaMQ enables *decoupled development:* the developers of an application need not concern themselves with the specifics of other applications' communications. An application simply publishes event information, at which point it becomes decoupled from the process. All other applications interested in that event automatically receive the published information. Only the *conveyance of the information* — not *what* other applications do with it — matters to the original publishing application.

As additional applications require notification about a particular type of event, they simply subscribe to the subject of that event. Without any modification to the original application, AmbrosiaMQ handles event notification to the additional subscribers. From a developer's perspective, this simplifies the development process when integrating a new application into an existing suite of applications.

Abstraction of Communications

AmbrosiaMQ handles the communications details for distributed computing, including the management of sockets and ports, protocols and semantics, and message transportation.

Unlike many traditional applications, AmbrosiaMQ applications send event information as messages. In the past, applications typically shared information indirectly by using an underlying database to store and retrieve shared data structures. Using AmbrosiaMQ, integrated applications need only agree on a common set of messages and message formats —not the numerous assumptions required to share database tables. Furthermore, there are no shared database issues such as data contention, locking, or read consistency, because the database is no longer used to simulate information flow between applications

Location Transparency

AmbrosiaMQ relies on message brokers to route messages. In this system, the publishing applications need not know the specific destination(s) for their messages. The AmbrosiaMQ Message Broker matches a message's subject against subscriptions submitted by interested client applications. The client can be anywhere on the network. As new subscribers join or leave the system, AmbrosiaMQ delivers messages without any reconfiguration of the publishing application.

Modular Architecture

AmbrosiaMQ implements communications using a hub and spoke topology. Message brokers act as hubs by routing messages to client applications (on the spokes), implementing policies, and supporting central services such as security and administration. This modular architecture can easily accommodate most topologies and growth patterns.

Push and Respond

Traditional applications search for required information by polling. Many applications claim to use push technology; however, the push method is usually implemented using polling and is typically a one-way communication. AmbrosiaMQ implements true proactive push technology and goes beyond just delivering information about events as they occur, to providing applications with the ability *to respond to and act upon* those business events.

Quality of Service

Client applications specify a message's Quality of Service (QOS), or delivery semantic, for a particular subject. AmbrosiaMQ sends messages with either reliable or guaranteed delivery.

With r*eliable delivery* (i.e*,* at-most-once delivery), AmbrosiaMQ will deliver a message to each subscribing client that is currently connected, as long as there are no application or network failures. In some environments, reliable delivery offers the most appropriate QOS. For example, if an application drops a stock quote, the user may not care since another will arrive in seconds. The default QOS is reliable delivery.

If the user needs assurance that a message will be delivered, the application developer can use guaranteed delivery (i.e., at-least-once delivery). Guaranteed delivery assures that AmbrosiaMQ will deliver the message even in the event of an application or network failure.

In addition to these basic QOS semantics, AmbrosiaMQ supports many other special QOS features including discardable delivery, prioritized delivery, load-balanced delivery, and so forth. These capabilities provide the developers with a high degree of flexibility to design their applications according to the business requirements.

Multiple Communications Models

AmbrosiaMQ's Client API, the `com.u1.client` package, offers a variety of communication models: publish/subscribe, solicit/response, and request/reply. These communications can operate either synchronously or asynchronously, depending on whether or not the developer wants a thread to block while waiting for information.

Transactional Services

Several classes of applications require grouping individual tasks into a single, coordinated "unit of work." This grouping of tasks is called a transaction. AmbrosiaMQ supports these applications by acting as a participant or resource manager in a coordinated transaction, though it is not itself a transaction processing (TP) monitor. AmbrosiaMQ coordinates the publication of a group of messages in the transaction such that it sends those messages if and only if the rest of the tasks in the transaction commit. AmbrosiaMQ implements a two-phase commit protocol, which guarantees that it sends either all or none of the messages in the group.

Comprehensive Security

Developers can build information systems that enforce their security policy easily and effectively. AmbrosiaMQ does not require any preexisting software on the client. The comprehensive security subsystem of AmbrosiaMQ allows developers to easily implement their security policies with respect to:

- Secure identification and authentication
- Authorization and access control
- Privacy and integrity protection using encryption
- Integration with an existing authentication infrastructure

Performance Oriented

AmbrosiaMQ strives to be very efficient with regards to network utilization. The primary design principle for AmbrosiaMQ has been to send as little data as possible and only send relevant data. AmbrosiaMQ's dynamic subject-based routing only delivers messages to active subscribers. If a message has to traverse multiple hops to serve more than one client (multiple subscribers), then a single message is sent across the common routing segments. Only at the last broker is the message dispatched to the individual subscribers. In addition, AmbrosiaMQ transparently compresses messages above a programmable threshold using LZ compression. Small messages are automatically batched to improve network efficiency.

Multiple Wire Protocols

Applications can connect to an AmbrosiaMQ message broker using a variety of wire protocols. The protocol choice is transparent to the client process; AmbrosiaMQ supports TCP (plain text), and SSL protocols. Protocol selection is primarily based on efficiency, security, and network topology considerations. The TCP (plain text) protocol is primarily used for internal (intranet) clients and applications. Although, the protocol is plain text, sensitive messages (e.g., authentication) are securely transmitted using strong encryption, but normal messages are transmitted in the clear. The SSL protocol is primarily used by external clients and is suitable for use over the Internet.

Intra-Process Messaging (IPM)

AmbrosiaMQ supports local loopback connection IPM (intra-process messaging). This results in a connection that directly maps the client sender to the client listener. There is no broker involved, so the application can publish and handle messages within its own context.

Evolving Roles of AmbrosiaMQ

AmbrosiaMQ was designed to be a general-purpose infrastructure solution for building applications for the Internet/intranet. Its event-based communication model provides powerful tools to coordinate information flow by channeling information between applications in the same manner that an organization processes events. It is best suited for applications carrying a broad variety of critical business information and requiring many-to-many communications.

AmbrosiaMQ assists developers by providing the communication infrastructure beneath applications that manage an organization's business processes: linking mobile workers in a virtual team; developing electronic markets; linking business partners together (for example, with supply chain management); sharing knowledge in service organizations; and optimizing the flow of data within or between functional departments.

Electronic trading and backend interchanges are well suited to make use of an event management system, such as AmbrosiaMQ. These systems are extremely demanding from a performance, security, and reliability standpoint. Middleware often provides a software layer where common solutions and optimizations can be implemented so that all products making use of the middleware inherit the benefits.

AmbrosiaMQ's multicast-like delivery of shared messages is particularly well suited for volume distribution of frequent update of information, such as pricing in an electronic trading system. This mechanism allows an update to be published once by a backend service, but be delivered to potentially hundreds of clients. Message delivery is optimized so that only a single message is sent across transmission segment that is common to multiple subscribers.   This strategy makes the backend services extremely efficient, and greatly reduces the network load by eliminating redundant messages.

2 AmbrosiaMQ Event Management System

Producers create goods. Consumers purchase goods. These basic events represent part of the constant flow of information that businesses must manage. Events impact the entire business process, not just specific functional departments. To reflect the pervasive impact of events, organizations require a comprehensive Event Management System (EMS). An EMS handles the flow of information throughout the organization, delivering critical information to those who need to act on it.

This chapter focuses on the most important aspects of AmbrosiaMQ's EMS: what it is, how it works, and how to manage it to best ensure efficient communication of data between your applications. To help give you an idea of how you can use the AmbrosiaMQ EMS, we have also included a section showing how some very different business situations take advantage of AmbrosiaMQ's capabilities.

How the Event Management System Works

Within an event management system, business applications focus on providing their specific, function-oriented capabilities while the EMS manages the inter-application event flow. A robust EMS must support an application's ability to capture, manage, and communicate the constant flow of business events. AmbrosiaMQ provides a foundation for an EMS by communicating event information between applications and providing services to support these communications.

Developers can link applications to the event management system by using the AmbrosiaMQ client Application Programming Interface (API). Interfacing with AmbrosiaMQ reduces the communications complexity of applications and enables developers to quickly add event-based communications to these applications. Using the AmbrosiaMQ client API, a business application communicates with other client applications or services through a common message broker.

In addition to the client application that provides business logic, this system includes several other components. Figure 2-1 illustrates communications in an EMS between the client applications, the message broker, and such services as Security and Administration services.

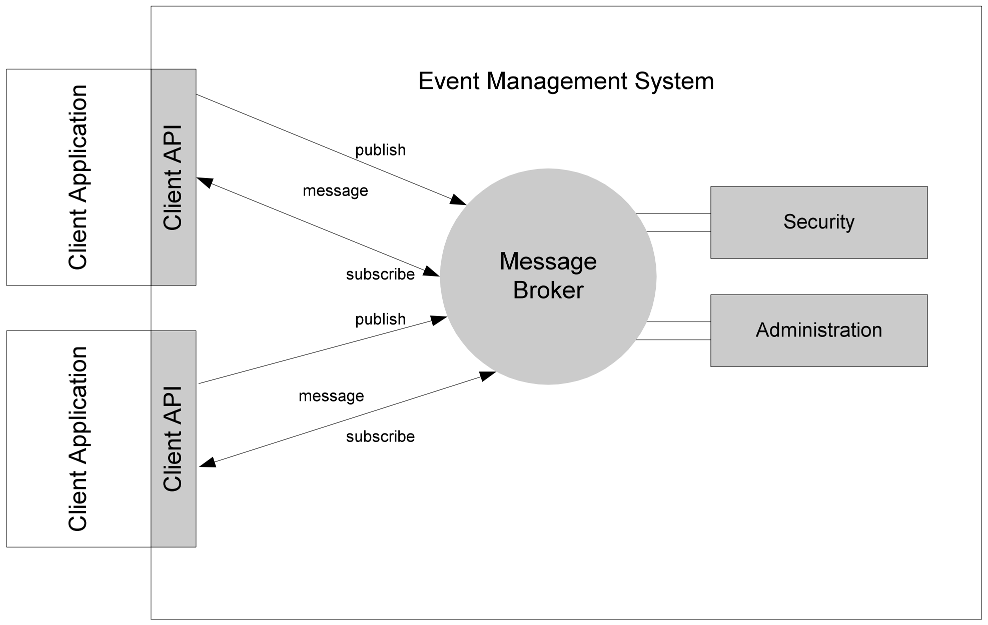


Figure 2-1 The Event Management System

The system can grow in a modular manner by adding client applications, services, and plug-in options. The message broker acts as a hub, supporting multiple concurrent client applications, services, and plug-in options.

Application Components of the EMS

The AmbrosiaMQ Event Management System (EMS) builds on these key components:

- AmbrosiaMQ Message Broker
- AmbrosiaMQ Client Applications
- AmbrosiaMQ Services

AmbrosiaMQ Message Broker

As the heart of the system, the message broker routes event-based information (i.e., messages containing event data) and supports a framework for services. The broker coordinates the flow of events as client applications publish information, delivering it to all other applications that need that information. The broker upholds quality of service policies to ensure guaranteed delivery on selected subjects.

The message broker manages connected client applications, applications' subscriptions, and the overall subject hierarchy used by client applications. Client applications and the message broker communicate through the API specified in the `com.u1.client` package. Please refer to the Javadoc reference for additional details.

AmbrosiaMQ Client Applications

A developer builds a software application to address an organization's business needs. This client application participates in the EMS if the application uses AmbrosiaMQ. The Java application simply imports the AmbrosiaMQ `com.u1.client` package so that the developer can instantiate the appropriate objects and call the relevant methods which establish the client-side communications in the EMS.

A developer's application can take advantage of the AmbrosiaMQ client package in one of two ways: an application uses the package pre-provisioned on a user's local machine, or an applet downloads the specific package classes from its HTTP server. Once on the local machine, the applet uses the client API to connect with the message broker on the server to join the EMS.

For additional details, please refer to the Javadoc description of the `com.u1.client` package.

AmbrosiaMQ Services

The third component of the AmbrosiaMQ system consists of services. From an architectural perspective, each service uses the AmbrosiaMQ architecture to provide value-added services to the overall system. Some of the services provided by the AmbrosiaMQ broker are: authentication, message security enforcement, duplicate client detection and conflict resolution, monitor (queues, clients, performance), and logging. Please refer to the *Administrator's Guide* for more details.

Administration Console

The Broker Administration Console (BAC) complements the core software of the message broker by providing a graphical user interface (GUI) to monitor AmbrosiaMQ-based applications, services, and the AmbrosiaMQ system itself. This interface provides the system administrator with the tools necessary to monitor, diagnose, and correct problems in a variety of areas. A single console can be used to administrate the entire AmbrosiaMQ network. AmbrosiaMQ exposes an Administration API for situations that require integration with an existing system management framework.

Location of the Application Components

One or more AmbrosiaMQ Client Applications may reside on the end user's computer. The end user can also download one or more applets from an HTTP server. In this way, a single user's applications can communicate with each other or with other users' applications.

On or more brokers can reside anywhere on the network. The broker network, known as the Interbroker Network provides a virtual hub, permitting fault-resiliency and efficient load balancing.

Managing Event Information

The EMS components communicate event-based information; this permits applications to parallel the flow of information in a business process. AmbrosiaMQ provides a set of objects to define the relationships between application components and to enable the applications to communicate. The next chapter, “Communication Models,” describes how AmbrosiaMQ enables developers to manage these objects. Figure 2-2 illustrates the conceptual layout of the objects used by AmbrosiaMQ for communications. The following pages explain these objects in general terms. For a definition of these objects in a programming context, please refer to the Javadoc documentation.

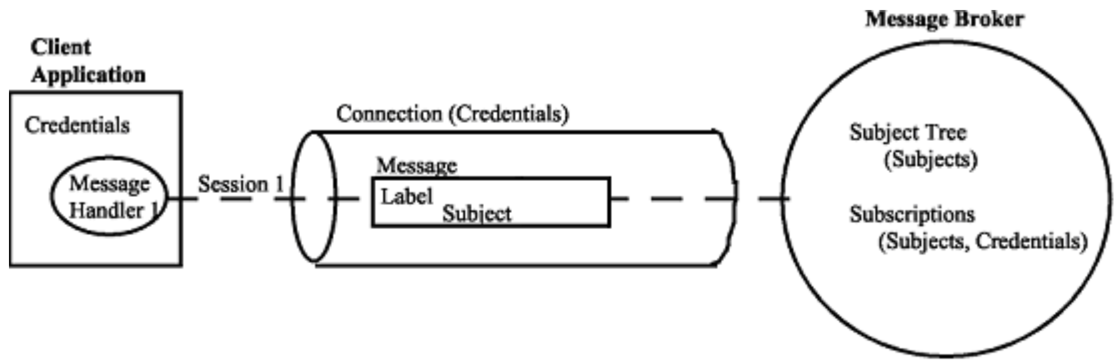


Figure 2-2 Communication-related Objects for EMS Relationships

Connection

The client application and message broker establish a connection for communications. This connection is analogous to a channel through which multiple sessions can flow, carrying a wide variety of messages. Typically, each application opens a single connection to a broker. One or more message handlers may be associated with the connection.

Credentials

The system identifies the end users of AmbrosiaMQ client applications by their credentials, which are based on the user's name and the user's password (AmbrosiaMQ supports other forms of authentication through the pluggable JAAS framework). Credentials serve as the basis for authentication and access control within AmbrosiaMQ’s Security model.

Message

A message carries information about business events. It consists of a subject and a body with the actual information. A client application creates a message, associates a subject with it, populates it with event data, and publishes it to the broker.

Label

A label is associated with a message. It contains routing information and optional message attributes. It can also contain delivery (Quality of Service) attributes.

Message Handler

As part of the client application, the message handler receives and "handles" events forwarded from the broker. The message handler uses a different thread and listens in its own session in a connection with the broker. Every client application must have at least one message handler, the so-called default message handler, as a catch-all to handle all delivered messages, though developers can define a message handler for each specific subject or type of expected event.

Session

Every session represents a single context of communication between the broker and client application. Client applications can create multiple sessions in which to perform work. Each connection has a default session associated with it. Each message handler runs in a separate session.

Subject

A subject identifies a specific category of event-based information. Subjects provide the common link between applications and serve as the key for the broker to route messages. Subjects can be flat or multi-level (hierarchical) with dots to separate the levels. For example, many biologists use a multi-level classification scheme to create specific "subject" names for life forms based on the format:

kingdom.phylum.class.order.family.genus.species.

Around this schema, you could build a subject tree to include a subject such as:

"Animalia.Chordata.Mammalia.Carnivora.Felidae.Felis.catus."

AmbrosiaMQ does not limit the number of subjects, the length of the subject name in a layer, or the number of layers in a particular subject name. However, when using guaranteed messaging the database imposes a 255 character limit on the length of each subject level.

Subject Tree

The set of subjects registered by client applications with the message broker creates a subject tree. Though a subject tree can be flat, it typically builds from one or more root subjects, adding other subjects in levels of parent-child relationships to create a hierarchical naming structure. AmbrosiaMQ does not limit the depth of a subject tree or the number of root subjects.

Each connection uses a subject tree that can overlap with those of other client applications. By referring to subjects within the common subject tree(s), the message broker can route messages between applications.

Subscription

The client applications register subscriptions with the message broker to receive messages published on a particular subject or set of subjects.

Following an Event through the System

The following scenarios describe how events (business information) can travel through the EMS:

- within a retail stock brokerage organization;
- at an oil extraction and refining organization - generating event data and real time feedback without human intervention;
- within a retail inventory tracking system - generating event data and real time feedback between multiple information producers and consumers.

Scenario 1: Basic Event Management

This scenario describes a basic situation: event information traveling through the event management system at a retail stock brokerage organization. The scenario assumes that the account representative has installed the sales order application (or, connected to the intranet and downloaded the sales order applet, along with the AmbrosiaMQ Client API classes).

1. *A business event occurs.* A consumer discovers a hot technology firm and places an order to purchase shares of stock by speaking with an account representative.

2. *A point of contact gathers event-based information.* An account representative completes a form in the sales order application.

3. *The point of contact evaluates the event's context.* The application publishes a request for reply based on information entered in the sales form. The message broker receives the message on the subject "consumer.check" and forwards it to all client applications with active subscriptions for that subject. In this case, the accounting department checks the consumer's accounts and credit line to verify funds can support the stock purchase. The accounting department replies to the request with an approval. Likewise, the compliance department and the risk management departments receive notice of the trade. Both these departments decide the trade meets their criteria. At this point the applet can move the business process forward.

4. *The point of contact notifies the organization.* The sales order application handles the reply (the credit verification and trade approval) by publishing a message on the "equity.buy" subject.

5. *The organization acts on the information.* The trading application on the exchange floor subscribes to the subject of this message. It executes the buy, then publishes information about the purchase to the "equity.buy.price" subject.

6. *The organization propagates information on the event and takes actions for closure.* Several applications receive information on this subject. The client management application adjusts the consumer's records in the

database to reflect the purchase. The accounting application issues an invoice for the stock and logs a receivable. A payroll application adjusts the account representative's commission.

7. *When the environment changes, this system can react without changes to the existing applications.* If the securities firm were to merge with another, their applications could join the EMS by subscribing to the subjects of the above messages. This requires no modifications to the sales order application, the trading system, or any application existing within the EMS.

Scenario 2: Event Management without Human Intervention

Scenario 2 again illustrates event information flowing through the event management system with an added twist: this oil extraction and refining organization's setup demonstrates the generation of event data and real time feedback without human intervention.

1. *A business event occurs.* In a remote stretch of desert, derrick number 9,876 extracts oil from an underground field. Several diagnostic devices measure properties of the oil such as temperature, volume and pressure.

2. *A remote terminal unit publishes event-based information.* Linked to the diagnostic devices, an application on an embedded microprocessor publishes this information on the subject "oil.production.9876" over a satellite link to a message broker located at a control facility. The message broker distributes the information to all subscribers.

3. *The backend service aggregates all information.* A backend application subscribes to "oil.production.*" and receives all production information. The application uses JDBC drivers to insert oil production information into a database on the corporate network for future analysis. This AmbrosiaMQ client application has been implemented as an AmbrosiaMQ service, giving authorized users the convenience of remote administration and configuration with the Administration Console.

4. *A crisis monitoring service alerts managers and remotely adjusts controls.* Another service also receives and reacts to the real time event information. This service checks the information in messages it receives against predefined parameters. If any variables exceed bounds, the service publishes several messages:

   - a message goes to another service, which handles the message by sending an alert to a manager's beeper with relevant information;
   - a message is directed to the source of the out-of-bounds data, instructing it to reduce the production flow to a minimum level until a human can intervene in the process.

5. *An executive information system displays corporate assets in real time.* An applet can subscribe to all oil related messages and portray this data graphically as part of an executive information system. Now, executives can log on from any web browser and enter their user name and password to view the current state of production, instead of waiting to view a monthly report six days after the month's close. This can facilitate

decision-making and permit adjustments to production as market conditions change.

6. *Balancing processes across functional areas.* In this case, the organization has separate groups responsible for extraction, distribution, sales to refineries, and establishment of inter-group transfer prices. Each of these groups actually requires timely information to optimize the organization's success. Each group uses different software systems, which the event management system bridges. With the EMS, transfer prices can reflect real time changes from a number of data sources:

- production from locations with different levels of various elements or compounds;
- available transport capacity and the actual deployment location of transport vehicles;
- availability of pipeline capacity; and
- regional demand from refineries.

Scenario 3: Event Management between Multiple Systems

In this scenario, event information travels through the event management system in a retail inventory tracking system. This particular example demonstrates the generation of event data and real time feedback between multiple information producers and consumers.

1. *A business event occurs.* An irate customer stomps up to an unsuspecting cashier and fumes: "I've been to five different stores! Doesn't anyone sell the blue super widgets?"

2. *Query for real time information.* The cashier smiles and turns to his terminal. From an applet in a web browser, he enters a query for inventory information to find in-stock super widgets from related stores in the area. The applet solicits for data on a subject such as "inventory.in stock.area51.super widgets". The message broker distributes the information to other stores who subscribe to this subject.

3. *Receive replies from other applications.* The applet then receives responses from the other stores. Thus, the cashier can quickly provide a list of nearby stores in the chain that have the required item.

4. *Publish sales data.* The store sells a product and publishes information about this event. Other area stores, the area warehouse, and various corporate applications might subscribe to learn about this type of event.

This event management system may share this same subject tree to communicate events that impact inventory with a number of other applications:

5. *Centralized inventory tracking service.* A backend application subscribes to "inventory.#" and thus receives all events that change the inventory published by stores and warehouse from all areas. The application uses JDBC drivers to insert inventory information into a database on the corporate network.

Other applications can request information for analysis by management. Buyers could subscribe to this subject to determine which types of items are popular. Accounting could use this to track assets. The warehouse could use this to determine which stores need restocking.

6. *An executive information system displays corporate assets in real time.* An applet can subscribe to all inventory messages and portray this data graphically as part of an executive information system. Thus, executives can log on from any web browser and enter their user name and password to see the current levels of merchandise, rather than wait to view a monthly report six days after the month's close. This immediate access to current information can facilitate decision-making and permit adjustments to production as market conditions change.

3 Communication Models

Traditional communication models involve demand-driven request/reply “polling” to access information, which is then transferred over a point-to-point topology. This chapter takes an in-depth look at how AmbrosiaMQ combines the benefits provided by traditional models with the innovations of publish/subscribe “push” technology, offering real time transmission of business events. Application developers will appreciate how AmbrosiaMQ’s hub-and-spoke topology simplifies coordinating client application interactions, as well as providing for easy integration of future applications. On the basic level of individual message transmission, AmbrosiaMQ offers guaranteed delivery of messages sent on important subjects.

Publish/Subscribe – Event-Based Communications

To more closely model how business systems function, AmbrosiaMQ's publish/subscribe communication model supports *event-based* computing. As events occur, the message broker proactively routes published event information to interested client applications. By mirroring the way in which information flows in the business world, applications using AmbrosiaMQ approach the "real-time" processing that businesses require in order to remain competitive.

In the publish/subscribe paradigm, publishers do not know the number of subscribers. Furthermore, the publishers need not know the identity or location of subscribers. Likewise, subscribers do not need to know about publishers. This implies an anonymous communications architecture abstracted via the message broker.

Consider the case of how information about a new hire for a sales department propagates throughout an organization. Each department in the organization does not call up (poll) the human resources (HR) department every fifteen minutes to inquire about the status of new hires for the sales team. The departments wait until the HR staff enter the information about this event and publish this as a message over the intranet. Other authorized departments can express an interest in the hiring of a new employee by subscribing to the subject "employee.hire.sales." Unbeknownst to HR, the subscribers might include a variety of departments: facilities, office management, marketing, information services, payroll, and sales. When HR publishes its "employee.hire.sales" message, the message broker delivers the message to all subscribers to the subject. Upon receiving the message, the facilities department issues a building access card; office management orders a new desk; marketing sends product materials; information services sets up user accounts and passwords; payroll initiates salary payment processes; and sales schedules an accounts review. Note that departments that have not expressed an interest in this subject do not receive the message – there is no scanning or processing of "junk messages." In fact, AmbrosiaMQ's security will only permit departments to subscribe to subjects as authorized by the system administrator.

Using the event-based computing model, the AmbrosiaMQ Message Broker automatically notifies all active subscribers when an event occurs, without requiring subscribers to continuously poll a central repository for the information. As a result, a single message can trigger several different processes (or responses) among the subscribers. As the message about new hires triggers other processes or events, any client in the AmbrosiaMQ system can respond by publishing its own message(s). All client applications built with AmbrosiaMQ can act as both producers and consumers of information.

Publish/Subscribe Process

Client applications publish event-based information as messages with specific subjects to a logically common message broker. The message broker coordinates communication between publishing and subscribing applications by using a set of shared subjects as keys to distributing messages. For each subject, the client may act as a publisher, a subscriber, or both.

Publishing means sending a message to the broker. Any time a client application wants to send or disperse information about an event to other clients of the system, it simply publishes the information as a message to the broker. There may be many publishers on a subject.

Subscribing denotes notifying the message broker of a client application's interest in one or more subjects. The subscription must occur before a message on that subject gets published in order for the client to receive that message. A subscriber can subscribe to one or more subjects; a subject may have zero or more subscribers.

The broker delivers a published message only to those client applications that have registered interested in the message's subject. A client application registers interest in a particular subject by submitting a subscription to the broker. The broker itself enforces no policy on interpreting subjects or messages. System designers and developers must establish conventions on the use of subjects. Please refer to "Chapter 4: Subject Names and Subject Trees."

Request/Reply – Demand-Driven Communications

Most distributed applications today use a request/reply communication model. This model provides a very useful approach to query for particular information. AmbrosiaMQ implements request/reply as a special case of publish/subscribe.

Traditional implementations base request/reply on polling, which most people characterize as “demand-driven” communications. Polling requires an application or process to request event information at automatic intervals. A poll requests information from a database, which can reply with the information, or with a message that the information requested is not available. As long as an application runs, it will continuously issue demands for information.

AmbrosiaMQ’s implementation bases request/reply on its publish/subscribe engine. When a client application publishes a request message on a subject, multiple subscribers registered to that subject may reply. This is unlike a poll-based request, which typically specifies a single information source. Most messages in the AmbrosiaMQ system use a subject for routing, but reply messages do not have a subject. The broker routes these directly to the requestor. The publishing client application accepts the first reply to the request; all other responses are ignored. AmbrosiaMQ extends this basic request/reply model using the method `Session.solicit()`, which permits receipt of more than one reply.

AmbrosiaMQ supports two types of request/reply: *synchronous* and *asynchronous.*

Synchronous

A synchronous request/reply is blocking – the requesting thread of the client application must wait, though other threads in the application may continue processing. This thread of the application will remain suspended as it waits for the reply.

Asynchronous

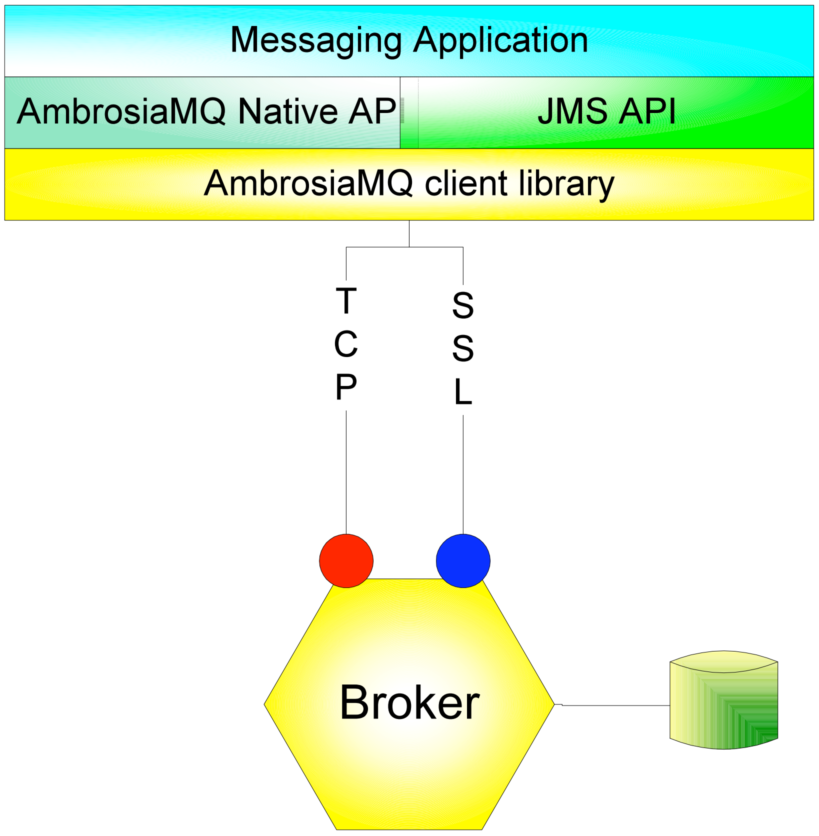
An asynchronous request/reply is non-blocking – the requesting thread of the client application may continue processing while waiting for the reply.

AmbrosiaMQ Interfaces – Connecting Applications

The AmbrosiaMQ Event Management System (EMS) employs an event-based communications infrastructure crafted for business applications that facilitates the exchange of information over the Internet and intranets.

AmbrosiaMQ satisfies two basic requirements for an organization's computing architecture: a short term need for the rapid development and delivery of applications and a long term need for these applications to have an adaptable distributed infrastructure that can meet changing business requirements.

An intuitive interface promotes rapid application development. Multiple interfaces means greater interoperability across heterogeneous environments. Most importantly, this permits organizations to provide legacy system investments with access to the intranet and Internet. This in turn creates new value as business event information flows from formerly isolated data.



Java Client Package

AmbrosiaMQ is provided with a Java interface due to its unique strengths, in particular, the Java language and Virtual Machine's property of cross platform support.

Java Message Service (JMS)

AmbrosiaMQ includes an implementation of a 1.1 compliant JMS API to the AmbrosiaMQ enterprise messaging system. JMS provides an industry standard API for building enterprise messaging applications. This means portability for applications when you change or adopt new JMS provider's messaging system.

JMS includes both a publish-and-subscribe and point-to-point messaging model. AmbrosiaMQ supports the publish-and-subscribe interface, durable subscribers and Queue (point-to-point) functionality.

The AmbrosiaMQ JMS implementation extends the JMS specification in the following areas:

- A message selector can compare JMSDeliveryMode against more than just 'PERSISTENT' and 'NON_PERSISTENT'. This is to support future delivery semantics such as DISCARDABLE delivery for example.
- Access to JMS is obtained by industry standard JNDI. By convention JMS clients find administered objects in a namespace using JNDI.

C/C++ Client Library

AmbrosiaMQ supports C language clients. This client is available for Windows NT, Windows 2000, Windows XP, and Solaris.

The AmbrosiaMQ C/C++ client is a wrapper for the Java client. The C client package calls (and is called by) the Java client. The Java Native Interface (JNI) is used to enable communication between the Java client and the C client. Through JNI, the C/C++ client can invoke methods in the AmbrosiaMQ Java client. The Java client will then perform the required operation and communication with the AmbrosiaMQ Message Broker, returning any results to the C client.

This architecture requires a Java Virtual Machine that implements JNI and that the AmbrosiaMQ Java client software be installed on all hosts whose applications will use the AmbrosiaMQ C client.

The C/C++ client API closely follows the calling interface of the AmbrosiaMQ native Java API.

Quality of Service

AmbrosiaMQ supports many types of publication/delivery semantics: *reliable delivery, discardable delivery, guaranteed delivery, load balanced delivery, and transactional publication*. Each type offers a different Quality Of Service (QOS) and can further be customized via various attributes such as *message priorities, message expiration,* and in the Summer of 2007 *Most Recent Value* (MRV). Each client application can independently configure the type of message delivery for a particular subject. The client application developer should consider business requirements when deciding on the QOS for each subject or message.

Reliable Delivery

Reliable delivery operates in a manner similar to all basic TCP/IP communications. It offers *at most once* delivery of a message to each subscribing client. Reliable delivery means that messages arrive at each subscribing client that is currently connected, except in the event of a network or application failure. If a failure occurs, delivery *may* not occur. AmbrosiaMQ uses reliable delivery by default. The message producer can ensure reliable delivery by publishing messages with a priority level of 2 (reliable priority).

Discardable Delivery

Discardable delivery offer *maybe once* message delivery. Discardable delivery allows AmbrosiaMQ to drop messages in the event that the system encounters a full delivery queue. To avert exceeding the configured queue limits on a message broker, AmbrosiaMQ will automatically drop the oldest discardable messages to make room for new messages. The message producer can select discardable delivery by publishing messages with a Message Label `deliveryMode` option set to `DISCARDABLE`.

Discardable delivery is primarily used for time sensitive messages such as product price updates in an electronic trading system. Use of the discardable delivery feature allows the system to automatically adapt to clients that do not have the network capacity to receive all of the messages that they are subscribed to, by allowing AmbrosiaMQ to drop the oldest discardable messages.

Load Balanced Connections

AmbrosiaMQ implements a load balancing connection mechanism which enables the distribution of connections over any number of available brokers. AmbrosiaMQ supports four modes of load balanced connection as described below:

- LB_LOCAL_SEQUENCIAL
- LB_LOCAL_RANDOM
- LB_SERVER_SEQUENTIAL
- LB_SERVER_RANDOM

The flag word LOCAL implies the first connection is also the final connection. No server load balancing is done.

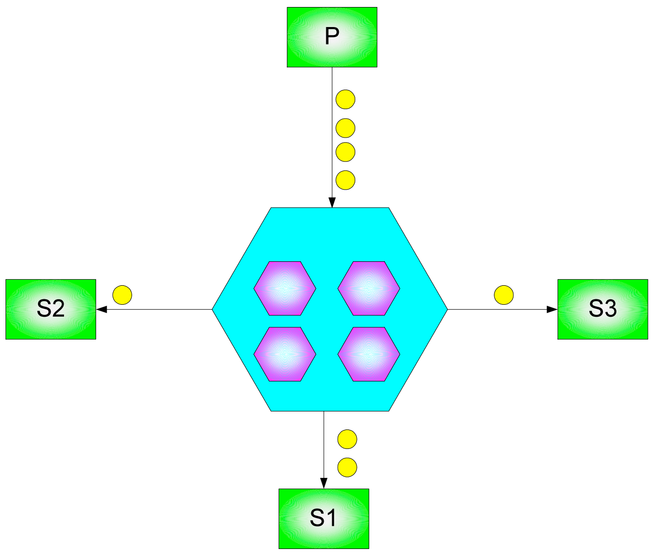
The flag word SERVER means that once the initial connection is made, the broker returns an ordered list, based on the inter-broker load balancing pools. The initial connection is then closed, and a final connection is made by iterating through the broker ordered list until a successful connection is established.

The flag word SEQUENTIAL results in the initial connection being made by trying the client specified list of brokers in order until a successful connection is established.

Finally, the flag word RANDOM means that the initial connection is made by randomly ordering the client specified list of brokers, then iterating through the randomly ordered list until a successful connection is established. . Note that LB_SERVER_RANDOM randomizes the client's list for the initial connection to a broker; the final connection is made by sequentially selecting a broker from the list that the initial broker returns.

Load Balanced Delivery

Load balanced delivery allows a message to be delivered to *one and only one* client that has subscribed to a message subject with the load balancing subscription option set. The load balancing delivery option is a convenient mechanism for building clustered services where it is important that one and only one service handler take action upon a particular message. This mechanism provides the additional benefit of transparent failover; that is, if a participant in the load balancing subscription pool becomes unavailable (loses its AmbrosiaMQ connection), then messages are automatically routed to the remaining participants in the load balancing pool.



Guaranteed Delivery

Guaranteed delivery offers *at least once or exact once* message delivery. AmbrosiaMQ is designed with an optimistic forwarding algorithm that first logs a guaranteed message to a persistent store then forwards the message to subscribing clients. After the client *processes*[1] the message, it sends an acknowledgment back to the message broker to verify receipt. Note that guaranteed delivery means more than simply placing a message in the client's message queue — only after the client acknowledges the message does the system automatically verify that delivery occurred.

In the case of a network or application failure, the message broker re-delivers messages to each client that has not acknowledged receipt. This ensures that the AmbrosiaMQ system will deliver critical information.

Subscribing clients specify guaranteed delivery. For example, if a client application subscribes to a subject assigned with guaranteed delivery, then delivery is guaranteed to the client. Publishing clients need not worry about guaranteed delivery; all synchronous publishes are implicitly guaranteed (to the broker) because they are synchronous. As long as the subscriber uses guaranteed subscriptions, then all messages from the publisher are guaranteed to the subscriber.

Exact once delivery requires that a client explicitly acknowledge guaranteed messages. Once the method `Session.acknowledge(Message)` returns control to a client application, AmbrosiaMQ will not deliver the acknowledged message again.

Transactional Publication

AmbrosiaMQ supports transactional publication of a group of messages. The messages within a transaction (unit of work) are published if and only if the rest of the tasks in the transaction commit successfully. AmbrosiaMQ supports a *two-phase commit protocol* (2PC), which guarantees atomic publication – it sends either all or none of the messages in the unit of work.

Subscription Recovery

AmbrosiaMQ applications can indicate that a subscription should be re-entered after the application disconnects (either normally or abnormally) and subsequently reconnects. This feature can be controlled on a subscription-by-subscription basis and is only relevant for non-guaranteed subscriptions (guaranteed subscriptions need not be re-entered).

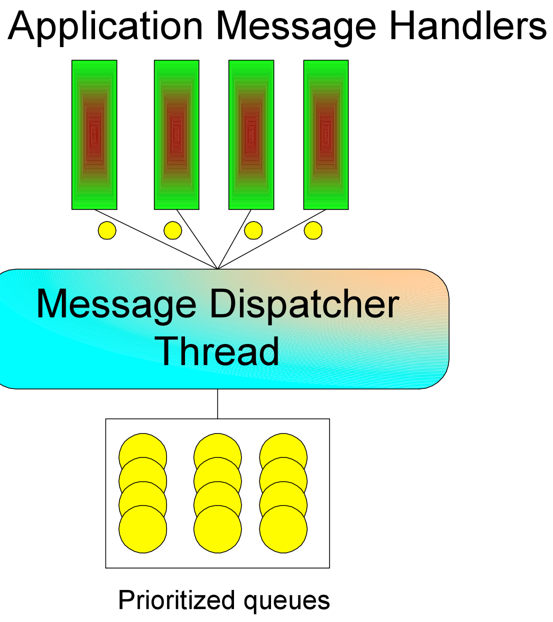
[1] The receiver can acknowledge the message automatically and immediately after receipt, or the application developer can choose to explicitly acknowledge the message at any arbitrary time during the handling of the message.

Customizing Communications

Communication between applications may be additionally customized using the AmbrosiaMQ client API and various system configurations. The primary publication and delivery options are summarized as follows:

Message Priority

AmbrosiaMQ supports message delivery priorities. A message producer determines the priority of a message when it publishes a message. Higher priority messages are delivered before lower priority messages. AmbrosiaMQ guarantees the order of messages from one publisher within the same priority level.



Message Expiration

AmbrosiaMQ supports message expiration. This limits the life of messages within the interbroker network. Message expiration is applicable to reliable and guaranteed messages.

When a message expires, it can either be dropped or it can be sent to a system subject. A broker configuration parameter controls this behavior. When the parameter `ENABLE_MSG_EXPIRATION_NOTIFICATION` is set to `false`, expired messaged will be discarded. When set to `true`, expired messages are published on the system subject:

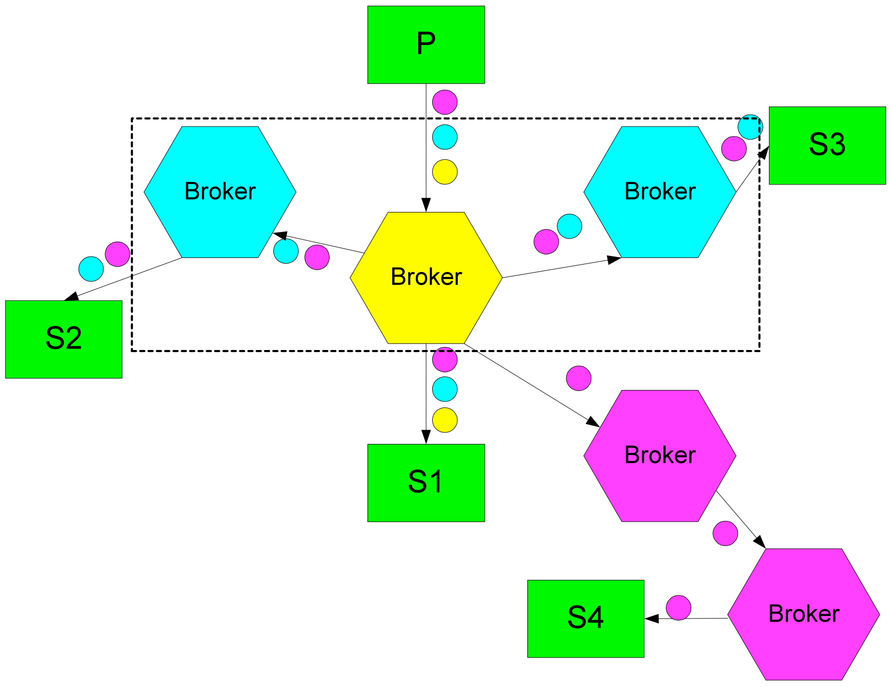
$SYS.broker.expiredMessage.*original_subject_name.original_expire_time*.

The body of the expired message is the body of the original message. To enable this feature, please refer to the *Administrators Guide*, “Chapter 2: Message Broker Configuration.”

A message is considered expired when the clock at the broker machine shows a time greater than the expiration time of a message. If a broker does not consider the message expired and sends it to a client whose clock would claim the message to be expired, the message is delivered to the client anyway. That is, the client's clock has no part in determining whether a delivered message has expired. Note, however, that the client's clock does determine the initial expiration time when a message is published.

Route Limits

AmbrosiaMQ supports a feature called *route limits* that allow a message publisher to control the extent of the delivery for messages they publish. For a subscriber, *route limits* control the extent of the propagation of their subscriptions through the AmbrosiaMQ interbroker system.



There are three *route limits* available in AmbrosiaMQ. These are summarized in the following table:

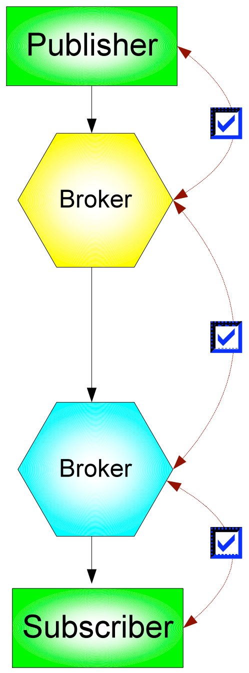
Route Limit	Publisher	Subscriber
Local	Messages only delivered to subscribers that are directly connected to the local broker only.	Subscriptions propagated to local broker only.
Collective	Messages only delivered to subscribers that are directly connected to a broker that is a member of the same collective as the publishing broker.	Subscriptions are propagated to every broker in the collective.
Global	Messages are delivered to any subscriber that is reachable via the interbroker network, including inter-collective deliveries.	Subscriptions are propagated to every broker reachable in the interbroker network. Includes inter-collective subscriptions.

Client applications use the following *labels* to indicate route limits:

- ROUTE_LOCAL
- ROUTE_COLLECTIVE
- ROUTE_GLOBAL

Virtual Circuits

Reliable and guaranteed deliveries satisfy the needs of many applications. However, there are special circumstances where application developers find themselves in a dilemma: reliable delivery is not sufficiently robust but guaranteed delivery is too expensive. For these situations developers can leverage AmbrosiaMQ's Virtual Circuits. Two processes establish a Virtual Circuit (VC) through which they receive notifications about each other's connectivity status (i.e., up or down). Using VCs, an application can simultaneously leverage the speed of reliable delivery and take corrective actions when its peer cannot take delivery of the message. Note that the messages are not actually delivered through the VC.



Applications use publish and subscribe as they would normally; the VC simply provides an asynchronous notification mechanism that signals a change in the peer's connectivity status. AmbrosiaMQ further enhances this unique capability by providing the following additional features for Virtual Circuits.

- Server Discovery — Clients may open a VC with one or more servers using a wildcard subject.
- Connection Manager — AmbrosiaMQ preserves connection parameters and automatically passes it to the VC server upon reestablishing the circuit.

- Weighted load-balanced servers — Clients will be assigned to a VC server from a pool based on the server's weight.
- Round Trip Time — AmbrosiaMQ exposes and API for measuring the roundtrip time between any two communicating parties. The API enables a VC client to determine the round trip time to the VC server. The round trip time is reported as an aggregation of up to three legs of the communication link as follows:
  1. VC client to its local broker;
  2. All the intervening brokers all the way to the local broker of the VC server; and
  3. VC server to its local broker.
- Client Multithreading — AmbrosiaMQ supports parallel startup of VC clients to multiple servers. Thus, the client's handshake with a server completes independent of handshake completion with other VC servers. In other words, if one server is slow starting the other circuits will not be held up.
- Virtual Circuit Duplicate Server Detection — Virtual Circuit clients identify a Virtual Circuit server using a subject pattern. AmbrosiaMQ will automatically disallow more than one Virtual Circuit server from using a specific subject string.

Applications can control VC discovery and load balancing by invoking the method `setLoadBalancedMode` from an instance of the `VirtualCircuitClient` class. The mode may be set to either `ONE_RANDOM` or ALL.

The `ONE_RANDOM` mode will randomly establish a virtual circuit with a single server from the pool of servers that match the wild card server subject. The `ALL` load balancing mode will establish a virtual circuit with every server that matches the wild card server subject. When using the `ALL` mode, VC clients will automatically establish circuits with new servers that are started, even if those servers were not running when the VC client was initialized.

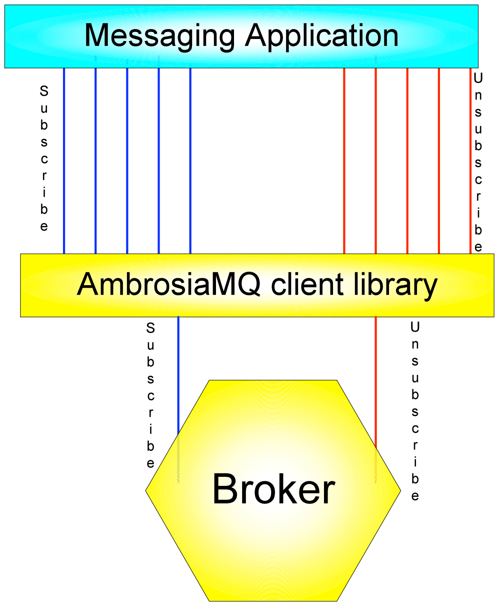
Bulk Subscriptions and Reference Counting

Enterprise-class applications can potentially subscribe to hundreds of subjects. Moreover, depending on the application, the subscription list can change whole-sale rapidly. Consider and example in the financial industry where a user can switch portfolios in an instant. As a result, the trading application that interacts with the user must change its subscription list immediately. This involved unsubscribing from all the subjects relevant to the current portfolio and subscribing to all the subjects relevant to the new portfolio.

AmbrosiaMQ provides two special capabilities to easily and efficiently address these situations:

- Array List Subscription — AmbrosiaMQ exposes an API for subscribing and unsubscribing in bulk. Effectively, the application simply adds and removes subjects from an array. Once completed, a single call to the method `Session.subscribe` (or `Session.unsubscribe`) creates (or cancels) all the subscriptions.

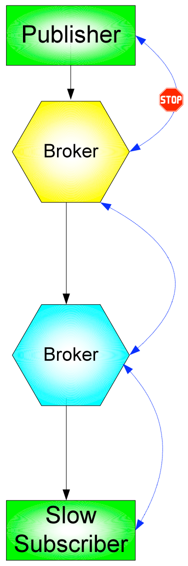
- Subscription Reference Counting — Application developers have the option to let the AmbrosiaMQ client software keep track of the number of subscriptions for a specific subject. Only the initial subscription results in communicating with the broker to enter the subscription. Similarly, only the last unsubscribe will result in the communicating with the broker to cancel the subscription.



Flow Control

AmbrosiaMQ has an automatic flow control system. If a subscriber cannot process messages fast enough, the AmbrosiaMQ client will automatically generate a flow control message to the **original** publisher, asking it to stop sending messages until the subscriber sufficiently empties its queues.

Each subscribing application has two sets of queues **per priority**: one for discardable messages and one for reliable (non-discardable) messages. The application can set the sizes of these queues and set flow control parameters using the `ConnectionProperties` object, which is associated with the `Connection` object. The table below summarizes the available parameters.



Property name	**Description**	**Default**
SUB_QUEUE_MEM_POOL_SIZE	Total memory available for reliable messages.	1,300,000 bytes
SUB_DISC_QUEUE_MEM_POOL_SIZE	Total memory available for discardable messages.	1,300,000 bytes
SUB_PER_QUEUE_LIMIT	Per queue maximum in bytes for reliable messages, not to exceed the total memory available	N/A
SUB_DISC_PER_QUEUE_LIMIT	Per queue maximum in bytes, for discardable messages, not to exceed the total memory available	N/A
FLOW_CONTROL_RESTART_THRESHOLD	After message delivery has been stopped due to an overfull queue, the number of bytes that must be available in the queue before restarting in bytes.	1024 bytes
QUEUE_ACTIVATION_THRESHOLD	Minimum number of messages for a queue to be considered "active" for sizing purposes	4

QUEUE_ACTIVATION_DELAY_MILLIS	How often to check active queues to see if they have fallen below the activation threshold	3000 milliseconds
QUEUE_DEACTIVATION_POLL_INTERVAL_MILLIS	How often to check active queues to see if they have fallen below the activation threshold	3000 milliseconds

Publisher applications can control the behavior of the system with respect to flow control as flows:

- A stoppable publisher is a publisher that is willing to stop publishing when any of the subscribers instruct it to do so.
- An unstoppable publisher is a publisher that does not wish to be slowed down by subscriber that cannot process messages fast enough.

A publishing application uses the `String` parameter `PUBLISHER_STOPPABLE` to control whether or not the publisher will be paused when any of its subscriber's queues is full. If a publisher is stopped then a message flow will be paused if any subscribing client reaches or exceeds a queue limit. If the publisher is not stoppable then subscribers will be automatically disconnected when their queue limit is exceeded. The default value for this parameter is "`true`", which means that publisher will be paused when any of its subscriber's queue is full.

A publishing application has one outgoing queue per priority. When an application publishes a message, the data is transmitted to the broker as fast as possible. Typically, the data does not stay on the queue for long. However, when this queue is full (for example when there is a lot of asynchronous publishing), the application will block when it attempts to publish a message until there is space available on the queue.

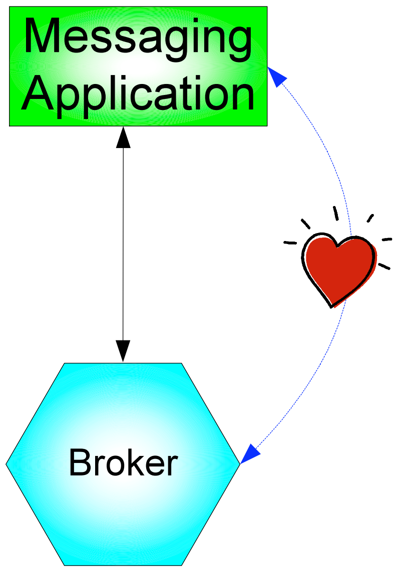
The property that controls the size of this queue is `PUB_QUEUE_MEM_POOL_SIZE` and its default value is 300,000 bytes. Note that unlike subscribe queues, there is no distinction between discardable and reliable queues. In other words, all outgoing messages of the same priority share the same output queue.

Client Queue Size Monitoring

AmbrosiaMQ enables a client to monitor its own queue sizes via a set of APIs. The APIs enable the client to get queue counts, sizes, and limits on for every queue of a specific Message Handler object. In addition, based on policy, a monitoring application can remotely request a client application's queue parameters via a simple request/reply paradigm.

Client Heartbeat Monitoring

An AmbrosiaMQ broker can be configured to monitor its clients' *heartbeat* and close a connection when the client is not responsive. When the broker configuration parameter `ENABLE_HEARTBEATS` is set to true, it instruct the broker to maintain heartbeat with all of its clients. This is a global, per-broker parameter. Note that each connection has its own heartbeat. Thus, one client with three connections will engage the broker in three heartbeat monitoring



Message Content Selection

AmbrosiaMQ is subject-based messaging system and uses subjects for routing. This is very efficient in that the brokers need not examine the content of the message. Once the message is delivered, client applications will examine the content and implement the relevant business logic. AmbrosiaMQ provides a message *Selector* API for evaluating the content of the message.

The Selector language supports three modes as follows:

STRICT_JMS, SQL92 and TYPE_CONV. STRICT_JMS. The Selector constructors and `createSelector` method have an option argument where the mode may be set. Once set on a Selector, the mode is carried through from parsing to evaluation.

- `STRICT_JMS` — This is the basis for all modes. It functions according to the JMS selector syntax as defined by JMS 1.1
- `SQL92` — Adds SQL92 keywords `DATE`, `TIME` and `TIMESTAMP`. These keywords should be followed by a string literal. The resulting SQL Date, Time or Timestamp literal is comparable to other literals or properties of the same type. It also adds non-standard SQL keyword `REGEX`. This is similar to SQL's `LIKE`, but uses regular expression notation.

- `TYPE_CONV` — Adds automatic type conversion of Strings to other types including `Long`, `Double`, `Boolean` and `Date`.

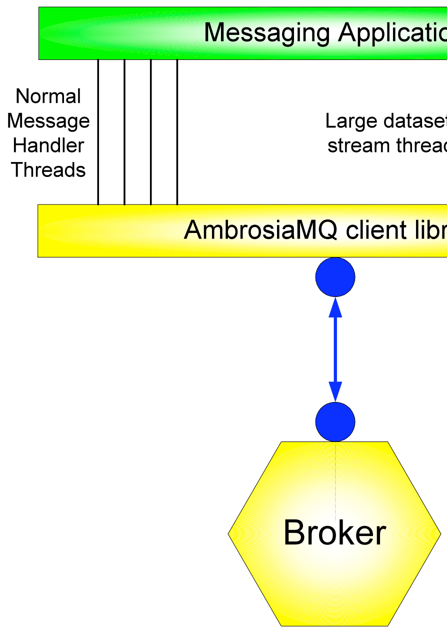
The AmbrosiaMQ Selector language provides the following extensions for additional flexibility:

- Regular expressions for content searching, and
- Support for time zones — this capability is made available as an optional field on String literal to `java.util.date` comparisons. The default date format for a String literal is: yy-MM-dd HH:mm:ss[.sss] [Z]. In addition to the default date formats, Selectors support a SelectorOptions object that allows the date formats to be predefined, using SimpleDateFormat objects, for the following date field types:
  - java.util.Date
  - java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp

Multiple `SimpleDateFormat` objects may be assigned to each type. The first format for a specific type that parses to a non null date value will be used.

Stream Transfer

The stream transfer feature allows the application to download or upload a stream of messages or data from a server. The transfer is done in the background and provides for efficient downloading/uploading of large data sets.


Messaging Application
Normal Message Handler Threads
Large dataset, stream thread
AmbrosiaMQ client library
Broker

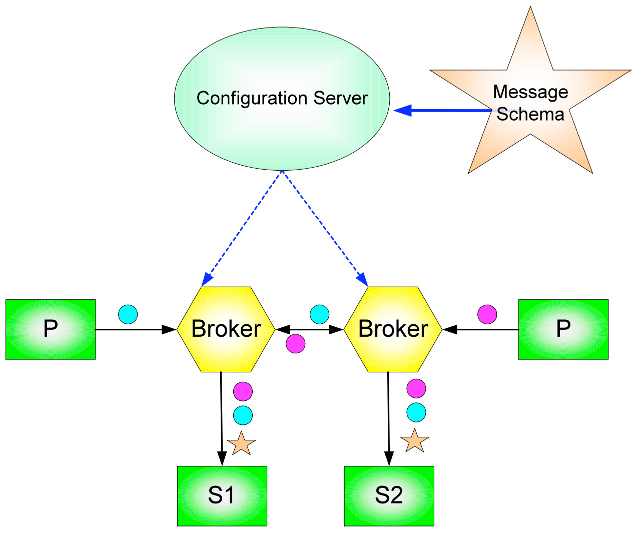
The client side API consists of four methods on a `Connection` object:

downloadMessageStream — Downloads a stream of messages from a server.
uploadMessageStream — Uploads a stream of messages to a server.
downloadDataStream — Download a stream of data from a server.
uploadDataStream — Uploads a stream of data to a server.

The server side is implemented by constructing a MessageStreamServer or DataStreamServer with a subject and a request handler. The request handler is passed StreamRequest objects that can be used to retrieve client information and accept or reject the requests.

Fast Serialization, Dictionaries, and Catalogs

Fast Serialization is based on Java serialization. It has native support for many java objects, including wrappers for primitive types, Collection and Map types, Date, String, BigInteger and BigDecimal. Support for other types can be added by implementing AmbrosiaMQ's `FastSerializable` interface. AmbrosiaMQ automatically uses Fast Serialization. If you wish to disable this, set the Java property `EnableFastSerializer` to `false`.

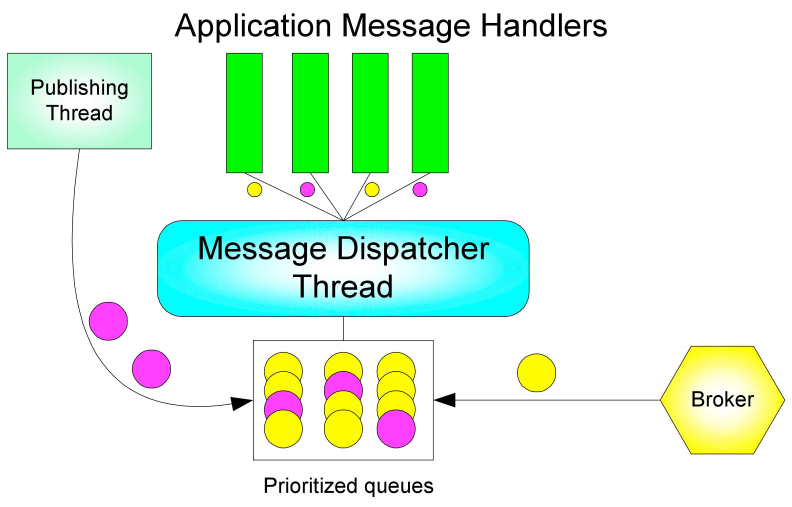


Fast Serialization can leverage a Dictionary. A Dictionary stores commonly used objects so that when an object is serialized only its index is written to the stream. This saves both space and time. The Dictionary must be available (i.e., pre-distributed) to any process that wants to read a message whose contents were serialized using that Dictionary.

A Catalog is a repository for Dictionaries. When an application reads a message, if a Dictionary argument was not given and the object was written with a Dictionary, AmbrosiaMQ will attempt to locate the Dictionary in the catalog.

Intra-Process Messaging (IPM)

AmbrosiaMQ supports local loopback connection IPM (intra-process messaging). This results in a connection that directly maps the client sender to the client listener. There is no broker involved, so the application can publish and handle messages within its own context.



Because there is no broker participating in an IPM connection certain AmbrosiaMQ features are inherently not relevant and are not supported. These include:

Subscribe – An application can subscribe to subjects, but it will have no filtering effect, it just returns as if the subscribe operation succeeded

Unsubscribe - An application can unsubscribe, but it will have no effect, it just returns as if the unsubscribe operation succeeded

ErrorView – Will throw an exception if an attempt is made to construct one.

Transactions – Will throw an exception if an attempt is made to call beginWork on a session.

VirtualCircuits – Will throw an exception if an attempt is made to construct a VirtualCircuitClient or VirtualCircuitServer on an IPM connection

Guaranteed Message – This is not relevant for IPM.

Because there is no subscribe filtering, the default message handler of an IPM connection will receive all messages whose subject is not bound to another handler.

4

Subject Names and Subject Trees

AmbrosiaMQ communicates a business event as a message. AmbrosiaMQ coordinates sending messages between publishers and subscribers using subject-based routing.

Subject-based routing, among other advantages, alleviates the need for supplying specific destinations for your messages, but it also creates a need for consistent and organized subject names.

Following the basic rules of subject name syntax and semantics, you can create and organize subject "trees" that will help you get take advantage of AmbrosiaMQ's routing and subject-based policies to build the most efficient applications. This chapter explains the special features' reserved names and wildcards, so you can accomplish a maximum amount of work with a minimum amount of effort.

Subjects and Subject Trees: Structure

A subject, which is a Java `String`, names a topic of common interest to producers and consumers of information. In AmbrosiaMQ, the broker routes each message based upon its subject. Almost any string of Unicode characters can act as a subject to describe the topic category of a message. AmbrosiaMQ reserves three characters for special use in subject names: the period (.), the asterisk (*), and the pound (#).

Individual subjects serve as elements (i.e., nodes) in a subject tree. Client applications register a set of subjects with the message broker to create subject trees. Though it can be flat (linear), a subject tree typically builds from one or more root subjects, adding other subjects in levels of parent-child relationships to create a hierarchical naming structure. The root subject "" is the root of all the root subjects.

Figure 4-1 illustrates the general structure of a subject tree. The structure of the tree in the figure follows a format with levels of increasing granularity: "country.state.city." Each string in the figure represents a node on the subject tree. Complete subject names aggregate nodes at one or more levels in the subject tree; levels are separated by the period character. Subject names fully specify the path to a specific node from the root of the tree in this format: root.level2.level3. In Figure 4-1, the string "USA" acts as a root node, the first level of a subject name for subjects in this tree. Valid subjects include: "USA", "USA.Alabama" and "USA.Alabama.Montgomery."

[redacted]

Figure 4-1 Subject Tree Geographic Example

Using Subjects

Subjects provide the key to the routing of messages between publishers and subscribers: they provide an anonymous alternative to citing specific destination addresses. The AmbrosiaMQ Message Broker attempts to match a subject on a published message with a list of clients who have subscribed to that subject. The broker delivers the message to each client with a subscription which matches the message's subject.

Thoughtful design of subject names and subject trees can often save time and effort later for routine operations, including:

- subscribing to multiple subjects (see "Using Wildcards with Subjects");
- establishing security policies (see "Inheritance of Security Policies," Chapter 5);
- automatically reacting to messages on a specific subject. An example could be sending an alert to a manager's pager (as in "Scenario 2: Event Management without Human Intervention," Chapter Two).

When designing subject names and subject trees, it is important to remember that the message broker generally does not interpret or attempt to derive meaning from the subject name itself: instead, it only uses the subject name to send related messages to clients who have subscribed to that subject. However, an exception to this rule exists in the case of subjects of the roots "$SYS" and "$ISYS." The message broker itself has registered interest to these special "administrative" subjects, reserved for AmbrosiaMQ's system messages.

Subject Syntax and Semantics

When building an application, the subject tree design plays a crucial role in the application's communication possibilities. This design should account for the following principles of subject name syntax and semantics.

- Subject names are case sensitive (like the Java language). For example, AmbrosiaMQ recognizes "ACCOUNTS" and "Accounts" as two different subject names.
- Subject names can include the space character. One example might be "accounts payable." Spaces are treated just like any other character in the subject name.
- Though not recommended, a subject level may be the empty string. For example, "a..c" is a three level subject name whose middle level is empty.
- For portability reasons, we recommend that subject names not include the null character (Unicode \x0000).

AmbrosiaMQ applies the following conditions to the construction and content of a subject tree:

1. There is no limit to the height or the levels of depth (number of period-separated strings) in a subject tree.
2. There is no limit to the length of any particular level name in the tree. However, when using guaranteed messaging the database imposes a 255 character limit on the length of the subject string.
3. There is no limit to the length of the overall string.
4. There may be any number of "root" nodes (that is, any number of subject trees). The virtual root of all root nodes is represented by "" (i.e., the empty set). Users may not publish on the "" root, though they may subscribe to it.
5. AmbrosiaMQ reserves the subject trees rooted at "$SYS" and "$ISYS."

Using Wildcards with Subjects

A wildcard is a lexical marker in a subject expression that can match a set of subjects. This saves writing specific subscriptions for multiple topics, and offers benefits to managers who may need to see information or events across several areas. Client applications can use wildcards when subscribing to a set of subjects, publishing on a subject, or binding a set of subjects to a message handler.

Wildcards in Subscriptions

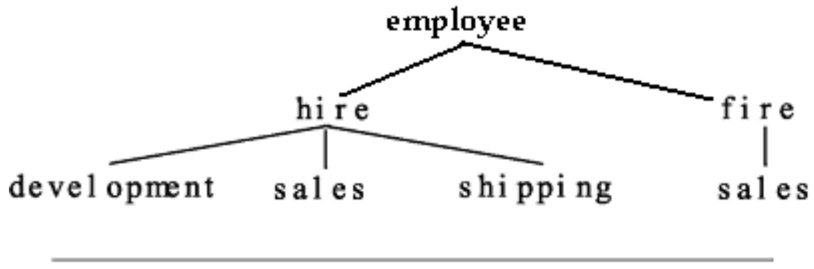


Figure 4-2 Subject Tree for Human Resources

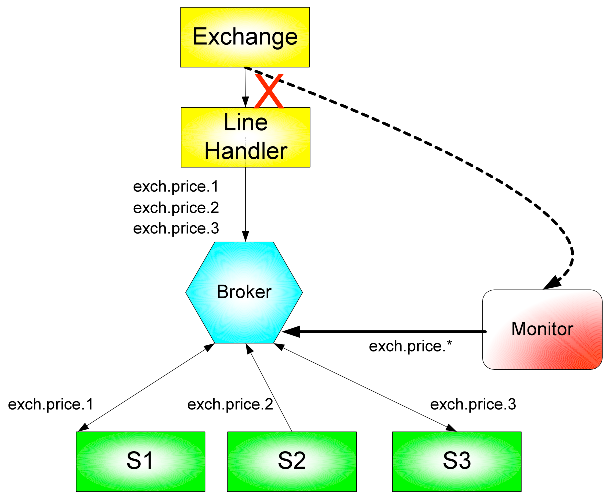
Using the subject tree shown above, a client application could either subscribe to a particular subject, “employee.hire.development,” or use a wildcard to subscribe to the set of subjects “employee.hire.#”.

Client applications can embed wildcards within a subject. Let us assume an application developer, using the subject tree in Figure 4-2, wants the application to track the changes in employees in the sales department. A subscription to the subject expression “employee.*.sales” would specify the subject set including both “hire” and “fire” messages.

AmbrosiaMQ also interprets multiple wildcards. A subject expression can include several asterisks and pounds as long as the use conforms to the guidelines stated above. Examples of valid subjects include using asterisks to specify a number of levels after a certain node such as “employee.*.*,” rather than using the “#” which would match zero or more levels. If Figure 4-2 were the top of a larger tree and a developer wanted to know all the issues relating to staff in the sales department, s/he would use “*.*.sales.#” as the wildcard subject expression.

Wildcards in Publications

AmbrosiaMQ supports a unique feature called wildcard publishing. Wildcard publishing is especially useful for applications that want to broadcast notifications messages to multiple subscribers. Consider a scenario in which three applications have subscribed to three different subjects at a common root as follows. Application 1 subscribers to exch.price.1, application 2 subscribers to exch.price.2 and application 3 subscribers to exch.price.3. Suppose that a monitor application determines that the exchange server is down and wants to notify all the subscribers. The monitor can publish a message on exch.price.*. In this case, AmbrosiaMQ will deliver the message to all subscribers.



Wildcards and Unsubscribe

AmbrosiaMQ uses intelligent wildcard matching for subscribe and unsubscribe requests. An unsubscribe request will only cancel the subscription for the exact subject. For example, suppose a client application enters two subscriptions for the subjects a.b.# and a.b.c.#. If the client unsubscribes from a.b.#, its subscription to a.b.c.# is unaffected (even though a.b.# matches a.b.c# and indeed any subject that is rooted at a.b).

Wildcards in Message Handlers

AmbrosiaMQ supports the ability to have multiple message handlers in an application. An AmbrosiaMQ connection must have at least one message handler, the default message handler. All messages delivered to the client will be received by the default message handler, unless additional message handlers are created and are bound to one or more subjects.

An application uses a bind mechanism, `MessageHandler.bind(subject)`, to control which messages are delivered to a particular message handler. A message handler bound to the root subject name "#" will receive all incoming messages. By rule, AmbrosiaMQ only delivers a message to the default handler if the message was not delivered to any other handler. If a handler is bound to "#", all messages will be delivered to this handler instead of to the default handler. AmbrosiaMQ will deliver a message to more than one message handler if the message's subject matches bindings from multiple handlers.

Subject Space APIs

AmbrosiaMQ allows developers to leverage the capabilities of the subject syntax in their own applications. The Subject Space API provides developers with a general mechanism for managing key/value pairs, where the key is a subject. Application developers can use the full subject syntax including wild card matching. Please note that the Subject Space API is not meant to manipulate message subjects.

5 Security

The vision of the AmbrosiaMQ security model is quite simple: offer a comprehensive security model that can enforce security with minimum application involvement. The two keys in this vision statement are *comprehensive security model* and *minimum application involvement*. We believe that an application's compliance with a security policy is greatly enhanced if the application developer is not responsible for enforcing it.

AmbrosiaMQ provides a complete security solution for Java applications by offering:

- Availability of important security features out-of-the-box, with no additional work required to implement them.
- Application design and implementation that is independent of the security policy.
- Security applied and managed centrally via an administration tool.

Overview of Security Issues

To truly leverage the Internet as a platform for business applications, organizations need to know exactly who is accessing their corporate assets — databases and business rules — over the Internet. Each class of user (employees, customers, partners, and suppliers) requires different levels of access. This implies a need for two very important facets of security: authentication (the determination of the user's identity) and authorization (the definition and control of what the user can and cannot do). Information may travel many places over the network; yet confidential messages must remain private, to prevent others from reading their content and to keep others from secretly tampering with that content.

This first section examines existing approaches to security concerns:

Feature	**Security Issue**
Authentication	Who is the user?
Authorization	What operations can the user perform?
Privacy	Who can see the data?
Integrity	Has the data been changed in transit?

Internet Security

Much has been said and written about Internet security or, more specifically, the inherent lack of security on the Internet. The most popular protocol used for security on the internet is the Secure Socket Layer (SSL) protocol.

Most secure protocols, such as SSL, deal with privacy and integrity protection. Privacy protection ensures that only the intended recipient can view the data. Integrity protection ensures that, if data is maliciously or accidentally changed while in transit, the system will alert the recipient that the contents were changed. Thus warned, the recipient will not rely on the validity of the message.

However, a secure protocol, such as SSL alone, often does not fully address the issue of client authentication. The SSL protocol relies upon the ubiquity of digital certificates for authentication. In many cases, SSL clients do not use a certificate while the SSL server uses a verifiable registered certificate. This result is the SSL client being able to authenticate that the server is registered with a trusted provider, but the SSL server cannot authenticate the client.

In addition, a secure protocol such as SSL does not address anything with regards to access control and mediation of application data.

AmbrosiaMQ provides a high degree of security by providing coverage in the following areas.

- Wire Protocol – (SSL)
- Identification and Authentication
- Pluggable authentication model via JAAS
- Authorization and Access Control for Message Data (User, Group level ACLs)
- Optional per Message Privacy and Integrity Quality Of Protection (QOP) (December 2006)

In summary, no Internet wire protocol deals with all aspects of security in one comprehensive package. AmbrosiaMQ's security features address all of these issues.

AmbrosiaMQ Security Design Advantages

AmbrosiaMQ allows developers to focus on building the application — not on implementing a security policy. Security operates independently of application code through an easy-to-use central administration interface that manages users, groups, Access Control Lists (ACLs), and Quality Of Protections (QOPs). This design permits remote administration for all aspects of security.

If an organization's security policies change, the system administrator can manipulate AmbrosiaMQ's security mechanisms without requiring the application developers to rewrite any application code. By allowing security policies to change with business needs, AmbrosiaMQ provides the flexibility that can extend an application's life.

AmbrosiaMQ achieves this simple yet comprehensive security because it effectively protects *subjects*. By enforcing security policies by subject, AmbrosiaMQ allows developers and system designers to indirectly address security through their design of subject trees. Each type of event information requiring its own security policy must have a unique subject name. Existing applications can immediately take advantage of a new security feature as it becomes available in future versions of AmbrosiaMQ.

The identification and authentication process is the only area of security where the client application must address security. AmbrosiaMQ leaves the application developer responsible for this task only: to solicit and pass to AmbrosiaMQ the user's credentials. Simply stated, this means that the application must solicit from the user enough information so that AmbrosiaMQ can authenticate the user. Beyond this aspect (a standard part of any application), the client application code does not implement or require information for security. Instead, the system administrator uses the Administration Console to set the security policies that AmbrosiaMQ enforces.

AmbrosiaMQ and its security subsystem do not require any pre-existing software on the client. All security functionality used in AmbrosiaMQ is provided through the Java Development Kit (JDK) interfaces such as the JCE. As such, security is built into each and every application or applet that is created with AmbrosiaMQ. Moreover, a developer need not worry about whether the application will run locally or as a downloaded applet. The AmbrosiaMQ security subsystem does not require access to any local resources. The security subsystem uses either part of the Java runtime environment or classes downloaded with the applet.

Identification and Authentication

When the security option is enabled, AmbrosiaMQ requires that each user be authenticated prior to granting the user access to the application level data. Identification is done by providing an user name created and assigned by a system administrator. Authentication is done by having the broker verify the user's credentials (e.g., password) in a secure manner. Out of the box, AmbrosiaMQ supports password-based authentication. Additionally, AmbrosiaMQ can be configured to integrate with any authentication infrastructure that can be exposed as a JAAS provider.

Authentication Overview

AmbrosiaMQ mutually authenticates clients and message brokers to each other. The client authenticates itself to the broker using its password. AmbrosiaMQ's default authentication uses a challenge/response mechanism. During the authentication process, the client's password is not sent over the network, this ensures that the client's password cannot be viewed by eavesdropping network devices. If an SSL connection is being established, the broker conveys its X.509 certificate to the client, informing the client of the broker's public encryption key. The broker is authenticated to the client through AmbrosiaMQ's protocol by proving it knows the client's password without being sent the client's password (challenge/response mechanism). The authentication mechanism also thwarts replay attacks by including client-side and broker-side random numbers in the challenge/response authentication messages.

Subject-based Security

In AmbrosiaMQ, all information flow is based on *subjects,* as demonstrated by the following:

- Developers organize information based on subjects;
- Applications can register their interest in consuming information by subscribing to a subject;
- Applications can produce information by publishing messages to subjects;
- The AmbrosiaMQ message broker routes information from publishers to subscribers based on the subject.

The security subsystem takes advantage of the information flow's dependency on subjects. By protecting the subject, one can precisely and dynamically control the flow of information. AmbrosiaMQ refers to this as *subject-based security.*

AmbrosiaMQ associates a security policy with every subject. The policy determines the following:

1. Who can publish, subscribe, or attempt guaranteed delivery on a subject?

2. Do messages on a subject need to be privacy-protected (encrypted)?
3. Do messages on a subject need to be integrity-protected (encrypted checksum)?

Since subjects themselves are organized in a tree, the security policy of a parent subject can be inherited by some or all of its child subjects that do not have an explicit policy already established (see "Inheritance of Security Policies" later in this chapter). AmbrosiaMQ enforces each security policy automatically and without any application intervention.

Using this subject-based approach, existing AmbrosiaMQ applications will automatically take advantage of new security functions provided in future versions of AmbrosiaMQ.

Authorization and Access Control

AmbrosiaMQ provides the ability to control who can publish, subscribe, or request guaranteed delivery on a particular subject or message format through the use of Access Control Lists (ACLs). The system administrator uses the Administration Console to define both positive and negative rights on the ACLs for specific subjects or message formats (for more details on how to define ACLs, see the *Administrator's Guide).* AmbrosiaMQ automatically uses these ACLs as described below.

The broker performs access mediation on a client's publish and subscribe operations, as well as on guaranteed delivery requests. For example, when the client subscribes to a subject, the broker gets the policy for the subject and checks to verify that the client is permitted to subscribe to the subject. If guaranteed delivery is requested on the subject, an access check is also performed on guaranteed delivery during subscribe time. If either one of these checks fail, the subscribe request is rejected and the client application throws an `EUnauthorizedClient` exception.

When a client publishes a message on a subject, the broker checks to verify that the client is authorized to publish on that subject:

- If the client is not authorized, the publish request is rejected, and the client application throws an `EUnauthorizedClient` exception.
- If the client is authorized, the broker delivers the message to all clients subscribed to the subject of the message.

Access mediation is done entirely on the broker side. However, a client can check for permission to publish, subscribe, or request guaranteed delivery on a specific subject using the `isPublishAllowed()`, `isSubscribeAllowed()`, or `isGuaranteedAllowed()` methods of the `Session` class.

There are three special principals included in the system: PUBLIC, Administrators, and Administrator. Special principals are reserved principal names in AmbrosiaMQ and hence should not be deleted or used for regular user names. The group PUBLIC contains all users defined in the security database (i.e., all users with passwords). This special principle is useful for creating ACLs such that all users would have permission to perform an operation on the subject.

The system administrator uses the Administration Console to add new users, groups, and policies for subjects. Please refer to the *Administrator's Guide* for further details.

Quality Of Protection (QOP)

In a future release AmbrosiaMQ will have the ability to control data protection on messages through *Quality of Protection* (QOP) settings. Using QOPs, the system administrator sets policies on individual subjects to protect message data. QOP options consist of:

1. **None -** message is delivered without special protection;
2. **Integrity -** assures that the message is checked for possible data corruption;
3. **Privacy and Integrity -** assures that the information can only be viewed by the intended recipient(s). All messages set for privacy are also automatically set for integrity.

By default, all messages are delivered without special protection unless the system administrator specifies either **Integrity** or **Privacy and Integrity.**

The **Integrity** feature verifies that the message content upon delivery matches its original published form. Corruption of data can be accidental or intentional. AmbrosiaMQ uses the cryptographic checksum Secure Hash Algorithm (SHA) algorithm to validate the integrity of message content.

AmbrosiaMQ ensures a message's **Privacy** by using encryption. *Encryption* scrambles the message content before sending it over the wire and restores the original form upon delivery. If the message is intercepted before delivery (i.e., someone attempts to read it as it goes over the wire) the data is not in readable form. AmbrosiaMQ encrypts messages to provide privacy using the Advanced Encryption Standard (AES) algorithm.

For more detailed information about AmbrosiaMQ's encryption, checksum and message digest usage, see "Chapter 5: Security," *Administrator's Guide.*

Inheritance of Security Policies

Since subjects themselves are organized in a hierarchical tree, the security policy of a parent subject can be inherited by some or all of its descendent subjects that do not have explicit policies. Therefore, it is not necessary to have an explicit security policy associated with each and every subject. Every subject has an implicit security policy, which is that of its parent. As an example, consider the following subject tree:

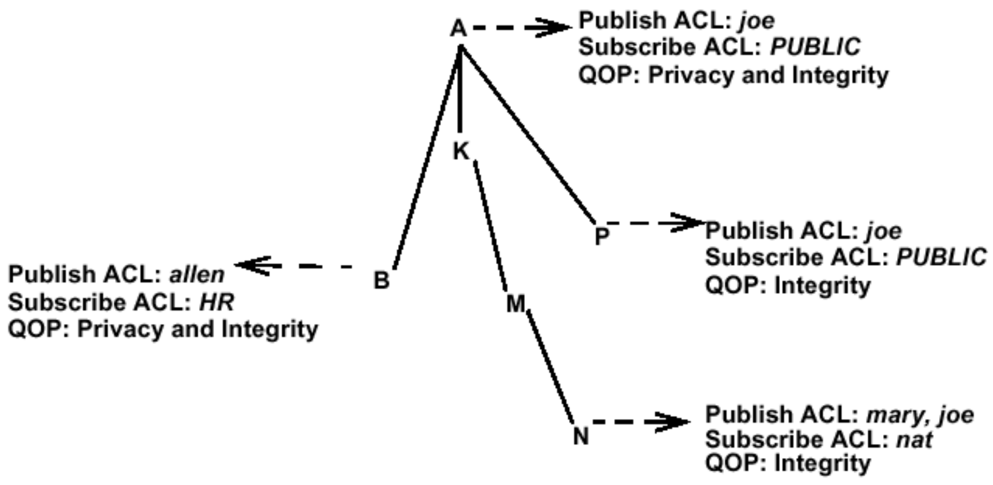


Figure 5-1 Subject Hierarchy with Security Policies

The table below summarizes the Access Control List and Quality Of Protection for each subject in the tree.

Subject	Publishers	Subscribers	Privacy	Integrity	Comments
A	*joe*	*PUBLIC*	Yes	Yes	Explicit policy
A.P	*joe*	*PUBLIC*	No	Yes	Explicit policy
A.K	*joe*	*PUBLIC*	Yes	Yes	Policy through **A**
A.K.M	*joe*	*PUBLIC*	Yes	Yes	Policy through **A.K**
A.K.M.N	*mary, joe*	*nat*	No	Yes	Explicit policy
A.B	*allen*	*HR*	Yes	Yes	Explicit policy

Table 5-1 Access Control and Quality of Protection for Subjects in a Hierarchy

Security Policies and Wildcard Subjects

AmbrosiaMQ does not permit associating an explicit security policy with a wildcard subject (e.g., **A.*,** which represents a two-level hierarchy and includes **A.B, A.K** and **A.P).** The reason is due to possible conflicts between the policy of a wildcard subject and a specific subject. However, AmbrosiaMQ does guarantee correct access mediation when a client subscribes to a wildcard subject.

For example, consider the subject tree in Figure 5-1. **A**.* as a subject does not (and cannot) have a security policy associated with it. Therefore, **A.*** inherits its policy from **A.** As such, any user can subscribe to **A.*** (the subscribe ACL includes PUBLIC). When a message is published on **A.P** or **A.K,** the message broker delivers it to the user who subscribed to **A.*.** However, when a message is published to **A.B,** that message is only delivered to subscribers who are in the HR group. Moreover, if the system administrator changes the subscribe ACL of any subject that matches **A.*** the broker will correctly enforce the ACL at the time of message delivery. Effectively, subscribing to a wildcard subject has the semantics to deliver messages on all subjects that match the wildcard and for which the subscriber has authorization to receive.

Responsibilities

AmbrosiaMQ divides the responsibility of enforcing a security policy across three categories. In increasing order of responsibility, the categories are:

1. Application developer — only responsible for soliciting authentication information from user.
2. System administrator— responsible for maintaining users, groups, and security policies for a subject tree.
3. The AmbrosiaMQ security subsystem — responsible for enforcing all aspects of security including authentication, message protection, and access mediation.

Responsibilities of an Application Developer

The application developer is responsible for soliciting authentication information from the user. The developer then uses this data to establish an authenticated connection with the Message Broker. AmbrosiaMQ provides the `Credentials` class to handle the user ID and password. This class abstracts all authentication models. Typically, before a client connects to the message broker, it creates an instance of the `Credentials` class. Using the `Credentials` class, the application can pass the ID and password of the user to the broker.

Responsibilities of a System Administrator

The system administrator is responsible for managing users, groups, and the security policies of subjects. Managing the security policies of subjects includes maintaining the Access Control List (ACL) and the Quality of Protection (QOP) of those subjects.

Please refer to “Chapter 5: Security Administration”, *Administrator’s Guide* for further details on administering security policies.

Security APIs

AmbrosiaMQ provides a rich administrative GUI for managing users, groups, ACLs, and QOPs. However, there are circumstances in which AmbrosiaMQ’s security must be integrated within a comprehensive security management framework, such as an entitlement provisioning system. For these situations, AmbrosiaMQ exposes a security API in the package `com.u1.client.SecurityAdmin`. Using this API, developers can integrate AmbrosiaMQ’s security administration with other tools.

6 Interbroker Networks

The AmbrosiaMQ EMS is a highly scalable system. A simple small system may be comprised of a single broker with a handful of clients, whereas a complex global system may be comprised of many coordinated message brokers serving thousands of clients. AmbrosiaMQ's key mechanism to building large highly scalable systems is the interbroker network. The interbroker network allows multiple AmbrosiaMQ message brokers to interact together and operate as if they were a single "logical" broker.

With the interbroker architecture, each broker can register itself with other AmbrosiaMQ brokers into a grouping defined as a *collective*. A broker in a collective becomes a composite proxy of subscriptions for all of the clients connected to that broker. Thus, AmbrosiaMQ can consolidate delivery of messages on the interbroker network whenever a broker has more than one subscription for a particular message subject.

For example, ten clients connect with a common broker *(B1)* and are all subscribed to the subject *A.B.C*. When a message *(M1)* is published with subject *A.B.C* by a client connected to some other broker (*B2)* in the interbroker network, only a single copy of the message *M1* is sent on the interbroker network from broker *B2* to broker *B1*. Once broker *B1* has received the message *M1,* the broker will dispatch the message to each of the clients subscribed to the subject *A.B.C.*

This chapter discusses the basic concepts of AmbrosiaMQ's interbroker architecture and provides an overview of the roles that brokers may assume within an interbroker network.

Single Broker Topology

AmbrosiaMQ's design departs from the traditional point-to-point communications architecture to follow a "hub-and-spoke" design: the AmbrosiaMQ Message Broker acts as the hub and AmbrosiaMQ-enabled applications as the spokes. Passing all messages from the applications through the hub has the advantage of centralized administration, security, and routing of messages. For the system administrator, this means all communications between applications can be monitored and maintained from a single location; for developers, each application only needs to interface with the AmbrosiaMQ Message Broker (which loosely couples applications). A comprehensive security model becomes much simpler to arrange when all communications go through a central point. Message routing prevents users from having to screen all messages locally to identify those that they need; also, an application publishing a message does not need to know the address of each subscriber.

Figure 6-1 graphically illustrates the advantages of a hub-and-spoke topology in terms of logical connections between applications.

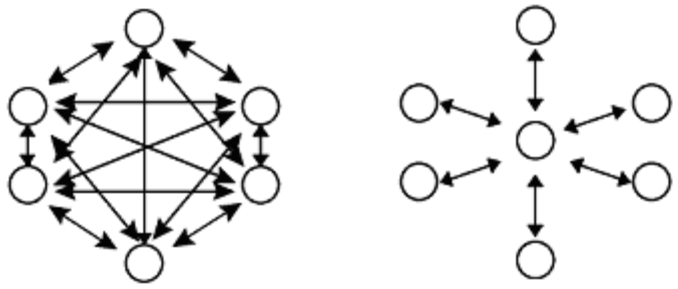


Figure 6-1 Comparing Point-to-point with Hub-and-spoke

In Figure 6-1, with six applications communicating, the system requires fifteen connections with the traditional point-to-point design and only six connections with the hub-and-spoke topology. The point-to-point model requires each application to include code to interface with the five other applications; furthermore, each application requires modification if another application joins the network. In the hub-and-spoke model, each application only needs to include a single set of code for communication with the message broker at the hub. The hub decouples applications, thereby reducing system complexity and increasing system flexibility and adaptability.

Interbroker Topology

The Interbroker architecture allows multiple AmbrosiaMQ Message Brokers to interact with each other and operate as a single logical broker. With the Interbroker architecture, each broker can register with other AmbrosiaMQ Message Brokers. The broker registers as a single proxy to represent all of its application users' publications and subscriptions. This presumes a global name space for users, groups, subjects, and security policies. These entities can be maintained in Relational Database Management System (RDBMS).

Administrators can easily configure and manage multiple, remote brokers using a single Administration Console. AmbrosiaMQ's Interbroker architecture represents a fundamental advance in terms of scalability for AmbrosiaMQ systems. It provides several key benefits:

- scalability to support tens of thousands of concurrent users,
- fault isolation, and
- integration of geographically dispersed locations.

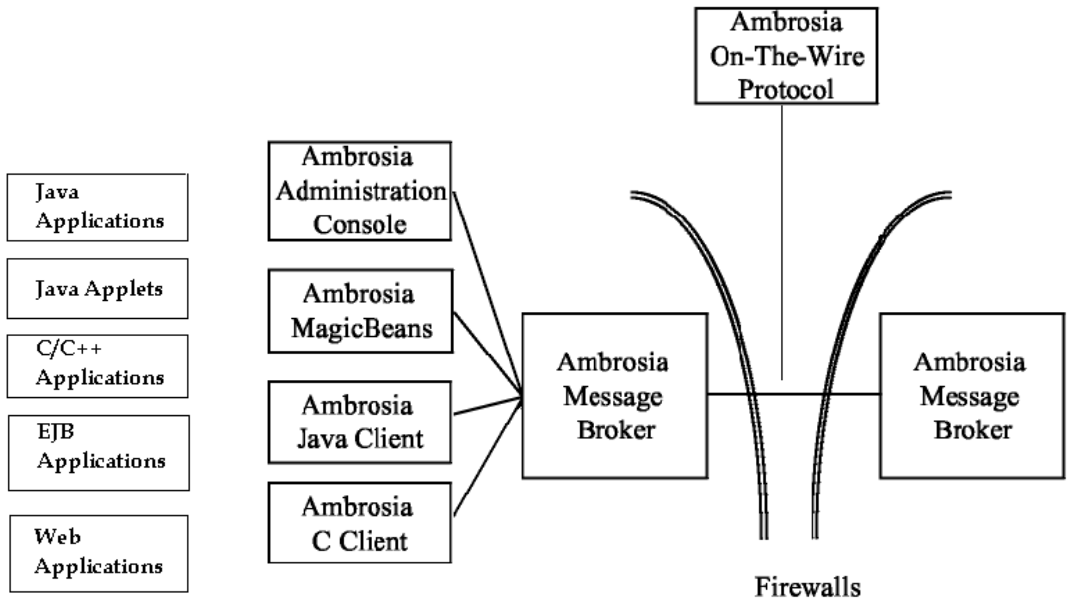


Figure 6-2 Interbroker Bridges Systems

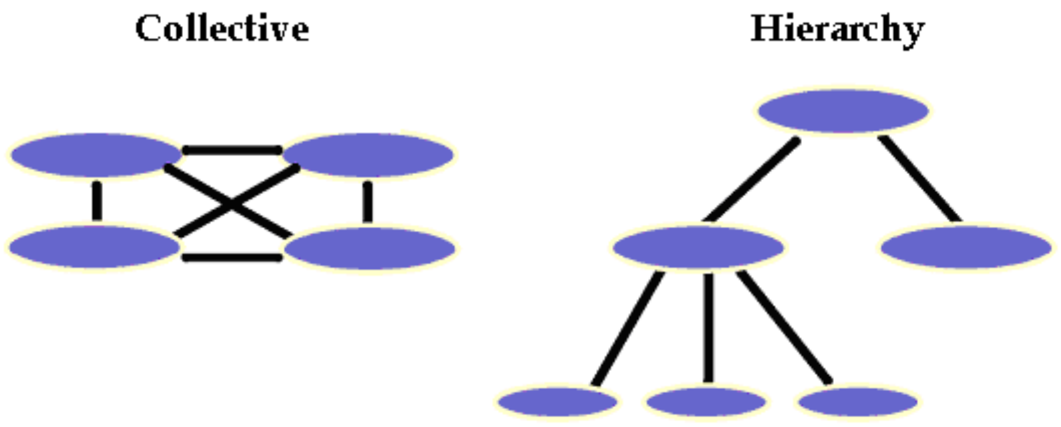


Figure 6-3 Interbroker Architecture: Collective & Collective Hierarchy

The interbroker architecture supports a variety of topologies, which developers can adopt depending on their application requirements. Figure 6-3 shows two basic designs including a simple collective and hierarchy of collectives that form a "tree".

A collective is a fully connected (hyper-connected) collection of brokers. System architects can utilize single brokers, multiple brokers in a collective, or bridge collectives hierarchically into larger structures.

Broker Roles in Interbroker Network

Deployments of AmbrosiaMQ interbroker networks can vary widely in applications. Simple AmbrosiaMQ systems may contain only one or two brokers hosted on a common intranet, whereas a complex system contains several collectives, more than 50 brokers, and crosses a wide variety of network topologies including: LAN, WAN, DMZ's, and the Internet. To address this wide range of possible system deployments, AmbrosiaMQ brokers can be configured to assume various service roles that may be required for a particular system. These roles involve security data distribution, configuration data distribution, access restrictions, and various other specialized services.

Primary Configuration Server

The AmbrosiaMQ EMS supports the dynamic distribution of the interbroker configuration and security data. The broker that acts as the source of the configuration and security data is known as the *configuration server.* The *configuration server* is responsible for providing broker connection points, collective groupings, and load balancing information for any broker that has been assigned to that *configuration server.* In addition, the *configuration server* provides each registered broker with a copy of the security cache, so that a broker can independently authenticate client login requests and enforce security policies.

The primary configuration server maintains a single writeable master repository of security data (users, groups, ACLs and QOPs). Other brokers receive a readable copy of the security repository from the configuration server. The configuration server acts as the transaction manager and is responsible for committing security updates to the security database and for publishing the updates to other brokers. The transaction process uses a combination of AmbrosiaMQ's two-phase commit facilities, guaranteed message delivery, and quality of protection to achieve a highly reliable replication of the security data.

The static definition of the interbroker configuration is stored in the file *intebroker.cfg* located in the *AmbrosiaMQ/bin* directory. The interbroker configuration may be changed on-the-fly using the IBAdminConsole interbroker admin tool. The *configuration server* is responsible for propagating any on-the-fly changes to the interbroker configuration to all registered brokers.

Backup Configuration Server

A backup configuration server is a read-only replica that acts as a standby for the primary configuration server. When a general purpose broker is unable to contact the primary configuration server, it connects with the backup configuration server and receives configuration and security data from it.

General Purpose Broker

General purpose brokers are primarily used for handling the majority of the message traffic within the AmbrosiaMQ system. These brokers do not have any specialized role in the configuration or security distribution, nor do they have any specialized security restrictions. The *general purpose brokers* are typically deployed on internal intranet topologies. *General purpose brokers* are usually added to collectives to improve scalability.

7 Working with Transactions

AmbrosiaMQ's Event Management System supports application transactional processing by acting as a participant in coordinated transactions. Further, AmbrosiaMQ supports the processing of transactions that may be distributed over two or more applications. With such transactions, it is usually imperative that any data changes be consistent globally; if the data changes in one application, the other applications involved in the transaction must also reflect this change.

This chapter discusses usage of AmbrosiaMQ's transactional processing capability to ensure the consistency of data operations across a distributed system.

Introduction to Transactions

Several classes of applications require grouping individual tasks into a single, coordinated "unit of work". This grouping of tasks is called a transaction. AmbrosiaMQ supports these applications by acting as a participant or resource manager in a coordinated transaction, although it is not itself a transaction processing (TP) monitor.

In a *distributed* transaction, the tasks in the unit of work are performed on physically different computers or by different application programs or services. A distributed transaction involves several steps:

- Prepare - to log or record the information necessary to complete the transaction, either by rollback or commit.
- Roll back - to undo or reverse the tasks performed in the transaction, leaving the data unchanged from its original state.
- Commit - to completely perform all the tasks in the unit of work (i.e., causes delivery of all messages in the transaction).

A distributed transaction relies on a transaction coordinator and one or more resource managers. A resource manager is a process or service (such as a database) that controls resources modified as part of the transaction. A transaction coordinator supervises the transaction. It acts as a manager of all the resource managers, telling them what to do. AmbrosiaMQ can act as a resource manager, whereas an AmbrosiaMQ client application or service must act as the transaction coordinator.

AmbrosiaMQ coordinates the publication of a group of messages in the transaction such that it sends messages if and only if the rest of the tasks in the transaction commit. AmbrosiaMQ supports a *two-phase commit protocol* (2PC), which guarantees atomic publication – it sends either all or none of the messages in the unit of work.

Two Phase Commit

How It Works

The two-phase commit protocol (2PC) guarantees that either every task in a transaction commits or none of them commit. Messages are sent if and only if every part of the transaction was completed successfully.

A transaction can use more than one resource manager. When there are multiple resources involved in a single transaction, the only way to maintain their atomicity, consistency, isolation, and durability (ACID) properties is to have a transaction coordinator which orchestrates the transaction. This coordinator works by first asking all the resource managers to prepare their parts of the transaction.

At this point, each resource manager prepares its part of the transaction. As part of a successful preparation, the resource manager guarantees that it can either: 1) completely commit the transaction; or 2) completely undo or “roll back” the transaction. In terms of an AmbrosiaMQ client application, it will publish either the entire group of messages or none of the messages. When it has finished the preparation, the resource manager notifies the transaction coordinator.

The transaction coordinator waits to receive an indication of successful preparation from each of the resource managers. After each resource manager has indicated a successful prepare, the coordinator issues a commit message which completes the transaction at each resource manager (hence, the global transaction). However, if the coordinator receives a single negative response to the original prepare message, the coordinator issues a rollback message to all of the resource managers.

Recovery

By design, the two-phase commit works even if failures occur. This avoids an inconsistent global state that might occur if one resource manager thinks the transaction committed while another resource manager thinks the transaction rolled back.

Therefore, the coordinator must not only send messages to the resource managers, but also must protect itself from failures. Typically, the coordinator writes the state of the transaction to a local log. AmbrosiaMQ’s Message Broker tracks this information for the application. The coordinator becomes, in effect, a resource manager of the resource managers in this transaction. In the event the coordinator fails, it can recover itself by reading its log. If a resource manager fails, it can recover by requesting the outcome of a particular transaction from the coordinator. A resource manager can prepare a transaction and, due to failures in the system, not know if the transaction committed; this state is known as an in-doubt transaction.

A coordinator and a resource manager must agree on how to name a particular transaction. Typically, the transaction coordinator assigns a *transaction identifier* (XID) to a transaction. In AmbrosiaMQ, the client application passes the XID to the message broker using the `Session.prepareWork()` method. In general, a resource manager uses XIDs to track multiple, simultaneous transactions, and to handle each transaction’s status, prepare, commit, or rollback.

Transaction Operations in AmbrosiaMQ

A transaction executes in the scope of a `Session` object. Within the context of a transaction, the AmbrosiaMQ Message Broker acts as a resource manager for published messages, not as a transaction coordinator. The AmbrosiaMQ client application must coordinate the transaction. The operations affected by being in a `Session` with an open transaction are the `publish()`, `solicit()`, and `reply()` methods. Please refer to the *Client API Reference Manual* for details on the transactional methods in the Session class.

In a typical `Session` object, calling a `publish()`, `solicit()`, or `reply()` for an individual message causes publication to occur at the time of the call. However, once `Session` operations take on transactional semantics, opening a unit of work causes these methods to act differently, since publication will not occur at the time of the call. Instead, the publication waits until the client application calls `endWork()`. This method sends all pending publications to the message broker in a logical unit of work; either all pending publications are published or none of them are published. The client application can cancel the pending publications at any time before the `endWork()` call by using `rollbackWork()`.

Basic Transaction Structure

In a basic transaction in AmbrosiaMQ, a client application begins the transaction, produces messages into that transaction, and then completes the transaction by publishing those messages. If the transaction is cancelled before the publish occurs, AmbrosiaMQ throws an `ETransactionRollbackByBroker` exception and the messages are not published; no data is permanently changed.

A transaction involving multiple resource managers uses the same basic framework with a few additional steps to ensure the transaction commits. A client application begins the transaction, produces messages into that transaction, and prepares the transaction with the message broker. If the prepare was successful, the broker sends all the messages: if the prepare was not successful, the transaction is automatically rolled back (the transaction-specific information at each resource manager is returned to its previous state). If a client application loses its connection before the completion of the transaction, it will receive a request message from the broker upon reconnection, requesting the transaction's status on the subject $SYS.TRANSACTION.IN-DOUBT. The client application instructs the message broker to commit or roll back the in-doubt transaction, depending on the status of the global transaction.

Note that only the `publish()`, `reply()`, and `solicit()` methods add messages to a unit of work. The other method which sends messages to other applications, `request()`, never joins as part of a transaction regardless of the state of the `Session`).

Transactions cannot occur nested within other transactions.

Beginning a Transaction

An application explicitly begins a transaction by calling the `beginWork()` method on the Session object; for example, the `Session` passed into a `IMessageHandler.handleMessage()` routine. If this call succeeds, then all later publishes in this `Session` will be made as part of an atomic publish. The atomic publish occurs when the transaction is committed by a call to `endWork()`.

Rolling Back a Transaction

A `rollbackWork()` call on the `Session` object can be made at any time before the transaction is finally committed. AmbrosiaMQ views this as non-acknowledgment of the message delivery. If a client MessageHandler's Session calls `rollbackWork()`, AmbrosiaMQ will immediately re-deliver the message to the Session's handler, if the Session was associated with a handler.

Committing the Transaction

To commit AmbrosiaMQ's part of the transaction, the application first issues a `prepareWork()` call. This method takes a transaction identifier argument for the current transaction. Since AmbrosiaMQ does not try to interpret this identifier, any string of characters that the application (or its designated coordinator) chooses can be used. The identifier is used only if the AmbrosiaMQ client application which issued the `prepareWork()` call fails before the AmbrosiaMQ system receives the final `endWork()` call. After the `prepareWork()` succeeds and all the other resource managers (i.e., client applications) are also ready, the `endWork()` method is issued, causing the message broker to now send any queued publications. After this is done, the AmbrosiaMQ system "forgets" about the current transaction.

Determining Transaction Status

In the event the AmbrosiaMQ system does not get the `endWork()` call following a previously successful `prepareWork()` call, the broker remembers the queued messages and the transaction identifier associated with them via the `prepareWork()` call which queues the publishes from the unit of work. In this case, AmbrosiaMQ assumes the client crashed and that it will re-connect to the AmbrosiaMQ system when it restarts.

When the client re-connects, the system sends a request message on the designated subject "$SYS.TRANSACTION.IN-DOUBT." This message contains the transaction identifier of the in-doubt transaction. The `Message.readUTF()` method should be used to get the in-doubt transaction ID. The client message handler must determine whether or not the transaction fully committed and return that information to the AmbrosiaMQ system. The client message handler informs the message broker by responding with a message whose body contains either true or false. True indicates the transaction committed, while false indicates that the transaction rolled back. The `Message.writeBoolean()` method must be used to write the status into the message.

If the message broker receives the message containing true (transaction committed), it publishes the queued messages from that particular transaction. If the message broker receives the message containing false (transaction rolled back), it does not publish the queued messages and now treats that transaction as rolled back.

Therefore, all AmbrosiaMQ clients processing transactions using `prepareWork()` should bind to “$SYS.TRANSACTION.IN-DOUBT” and attach a message handler capable of determining the status of transactions.

A
Glossary

This glossary offers a brief overview of commonly used terms to help clarify AmbrosiaMQ's major concepts and characteristics.

Definition of Terms

AmbrosiaMQ Client Application

The client application refers to an application that uses the AmbrosiaMQ client package to communicate with the event management system. The package may already reside, pre-provisioned, on a user's local computer or may arrive with an applet downloaded from an HTTP server. Once on the local machine, an application uses methods in the client API to connect with the message broker and join the AmbrosiaMQ system.

Credentials

This consists of the user's ID (i.e., name or login) and password. Credentials objects identify the end users of AmbrosiaMQ client applications.

Event

A business event can be anything that happens which materially affects your organization. AmbrosiaMQ encapsulates business events as information carried in messages.

Event Management System

The AmbrosiaMQ Event Management System includes the entire communications infrastructure and the information flowing within the system. The infrastructure links several main components: the business application using the AmbrosiaMQ Client API, the AmbrosiaMQ Message Broker, and other client applications.

Interbroker

An optional feature of AmbrosiaMQ that allows you to have more than one broker. Using the Interbroker Configuration tool, you may establish and maintain multiple brokers. Groups of brokers are then organized into "collectives."

Message

Message objects carry event information between AmbrosiaMQ client applications. AmbrosiaMQ messages consist of a subject name and a content body, which is simply a series of bytes. As a result, messages can include any application data. Each message must be published to a specific subject. AmbrosiaMQ does not set any limits on the size of a message. AmbrosiaMQ handles the details of marshalling, unmarshalling, and other low level process details so that developers can focus on the message contents.

Message Broker

As the heart of the system, the broker routes messages and implements services. A broker communicates with AmbrosiaMQ client applications and with other message brokers as well. The message broker maintains lists of connected clients and their current subscriptions. Client applications and the message broker communicate by use of the com.u1.client package.

Message Handler

Message handlers act upon messages arriving at a client. The client application must initially set a default message handler, Subsequently, the client can designate specific message handlers for individual subjects or sets of subjects.

Publish

A publish occurs when an AmbrosiaMQ client application produces information and sends a message with this information to the message broker.

Publish/Subscribe

The publish/subscribe communication model is based on the ability of client applications to publish messages tagged with subject names, which are then delivered only to other clients who hold a subscription (that is, an interest) for that subject. This basic model promotes a many-to-many mapping of publishers to subscribers. It also describes an anonymous communications architecture wherein subscribers do not need to know who published a particular message and publishers do not know the subscribers' specific addresses. A central message broker tracks subscriptions, routes messages, and implements delivery policies. By default, every client application can publish and/or subscribe to a subject.

Quality of Protection

The Quality of Protection (QOP) options for a subject consist of privacy and integrity. Privacy assures that only the intended recipient views a message by using encryption. Integrity uses a cryptographic hashing algorithm to ensure that the information within a message cannot be altered without the recipient knowing about it.

Quality of Service

Quality of Service (QOS) refers to delivery semantics, which are the levels of assurance with which AmbrosiaMQ will deliver a message. AmbrosiaMQ supports two types of delivery semantics: *reliable* and *guaranteed* message delivery. These delivery semantics are further enhanced by various QOS options including *discardable, expirable, Most Recent Value,* and so forth.

Request/Reply

The request/reply model provides a very useful approach to query for particular information. AmbrosiaMQ implements request/reply as a special case of publish/ subscribe. When a publisher sends out a message as a synchronous request, the function will wait until the first response to the request is received; all other responses are ignored.

Session

Every session represents a single context of communication between the broker and client application. Client applications can create multiple sessions in which to perform work.

Subjects

Subjects provide the key to the routing of messages between publishers and subscribers. Subjects provide an anonymous alternative to citing specific destination addresses. Almost any string of Unicode characters can act as a subject to describe the topic category of a message. Subject names consist of one or more levels separated by the period (.) character. AmbrosiaMQ reserves subject names with “$SYS” and “$ISYS” at the first level for internal system use.

Subject Expression

A subject expression can include multiple levels and one or more wildcards, the asterisk (*) or the pound(#). Thus, the subject expression can represent a set of subjects for subscribing or binding. Wildcard characters are interpreted literally in `unsubscribe()`or `unbind()`.

Subject Tree

Subject trees form the basis of message routing in AmbrosiaMQ and thus play an important role in application design. Subject trees are hierarchical strings composed of levels of subject names. Levels are established by using a separator character, the period (.), to partition the subject tree branches. Thus, levels can establish subject branches such as “SOFTWARE.JAVA” or “SOFTWARE.C” to provide groups and specific sub-topics for messages. The system can have an unlimited number of subject trees, and each subject tree can have an unlimited number of levels.

Subscribe

To subscribe, a client application registers interest in a subject or multiple subjects, possibly by using wildcards. Subscribing presents several choices: quality of service, specific message handler, and related features. The developer usually hides the use of message handlers, delivery semantics, and specific subjects from the end user of the application using AmbrosiaMQ for its communications.

Transaction

Several classes of applications require grouping individual tasks into a single, coordinated “unit of work”. This grouping of tasks is called a transaction. In a *distributed* transaction, the tasks in the unit of work are performed on physically different computers or by different application programs or services.

Two-Phase Commit

This protocol enables AmbrosiaMQ to commit a transaction and send an entire set of messages in a unit of work, or to abort and roll back the entire set of messages. Thus, the message broker acts as a resource manager so client applications can participate in transactions.

B
Index