OMNESYS™
TECHNOLOGIES

The OMNE™

An Overview of the Omnesys Meta Network Environment™

September 1, 2001

ONE BLUE HILL PLAZA, 10TH FLOOR
PEARL RIVER, NEW YORK 10965-8689

WWW.OMNESYS.COM
845 735 5000 VOICE
845 735 5950 FAX
888 565 OMNE TOLL FREE

Table of Contents

The OMNE™ Overview

The OMNE™, which stands for **The Omnesys Meta Network Environment**, is a messaging oriented middleware product. It based on a simple and scalable architecture that is extremely versatile and reliable.

Due to its inherent simplicity, The OMNE™ enables developers to design programs and systems intuitively and quickly. Such programs and systems are called **OmnePresent**™.

The heart of The OMNE™ is its patent pending Content Based Router™ (CBR™) process. This process routes messages between OmnePresent™ programs using the **request/reply** and **anonymous publish/subscribe** paradigms. Simple and versatile, the CBR™ scales effortlessly to support just a few clients as easily as it supports thousands.

Data Processing Paradigms

REQUEST/REPLY: a paradigm in which messages are exchanged between a requesting process and a replying process. The requester (the client) sends a request message to a request handler (a server or service provider). The request handler (the replying process) sends response messages back to the requester..

PUBLISH/SUBSCRIBE: a paradigm in which messages are exchanged between processes that publish messages to an intermediary process (a message distributor) and processes that subscribe, for published messages, to an intermediary process.

ANONYMOUS PUBLISH/SUBSCRIBE: an implementation of the publish/subscribe paradigm such that only the consumers of a published message (the subscribers) can define the criteria by which a subscription to it may be made.

Conventional data distribution systems use the request/reply paradigm in a **peer-to-peer** topology. The OMNE™ uses a **"Hub-and-Spoke"** topology arranged in a CBR™ Tree for both request/reply and anonymous publication/subscription.

Hub-and-Spoke Topology

PEER-TO-PEER TOPOLOGY: Clients connect directly to servers. Every client must be connected to every server from which it wants to receive data. As the number of clients and services increases, the demand on the infrastructure rises exponentially.

HUB AND SPOKE TOPOLOGY: Servers send all available data to a CBR™ or CBR™ tree, and the CBR™ distributes the data to those clients who have an interest in it. The decision to accept or not accept data lies solely with the client. This topology is readily scalable.

A model Hub-and-Spoke configuration is shown in the figure below.

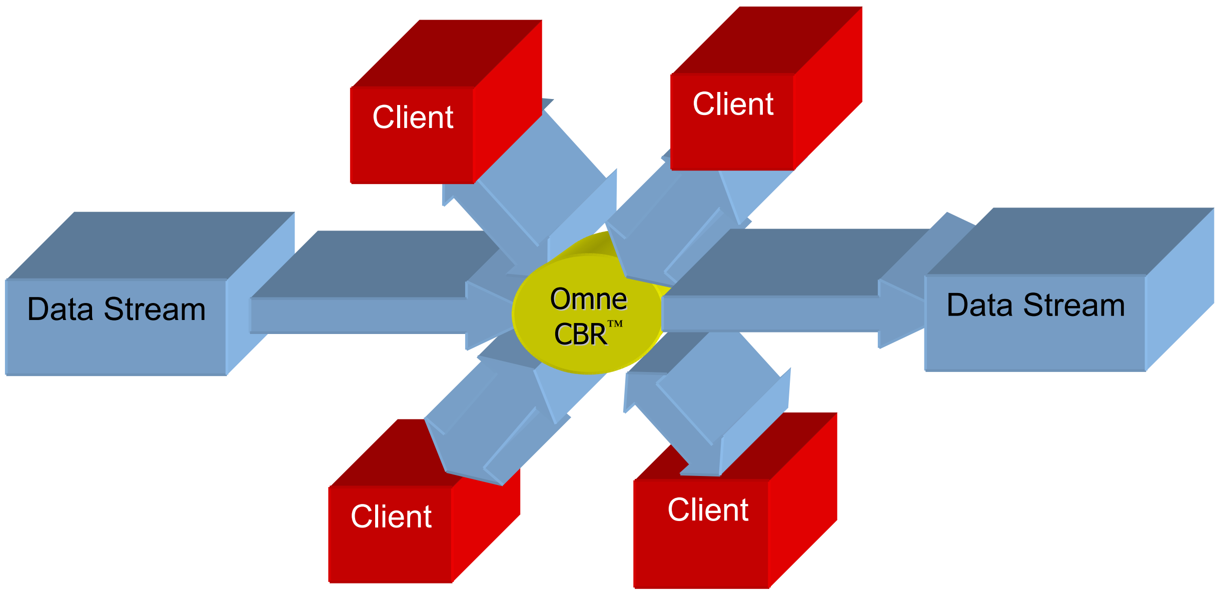

Figure 1 - CBR™ "Hub and Spoke" Topology

Security and Performance

Security is optional to take advantage of performance benefits when operating in unencrypted mode where The OMNE™ is operating in a secure environment or where security is not a concern. However, if inherent security is required, OMNE™ provides Secure Socket Layer compliance. When using SSL, every single message that is sent is encrypted and then decrypted when received. SSL also provides authentication using digital certificates.

Components of The OMNE™

The OMNE™ is comprised of a set of C libraries, associated infrastructure processes, and general purpose utilities. A Java interface, called **OmneVerse**™, is also available.

Libraries - provide methods that enable developers to write distributed application processes that communicate with each other. They are provided to shield developers from the mechanics of application independent functions such as service location, transport, event handling, encryption, automatic reconnection after an outage, message formatting, message parsing, and more.

Associated infrastructure processes - route and log messages, authenticate users, and maintain data and meta data repositories.

General purpose utilities – provide developers and administrators with runtime tools to configure and manage all of the processes of an OmnePresent™ system.

The OMNE™ C Libraries

The core libraries are called the **Kit** and the **API**. The Kit library is organized as a collection of software modules, each with its own responsibilities. The API library, using the Kit library as its foundation, implements the basic communication paradigm of The OMNE™.

Core Libraries

KIT: a library consisting of a set of modules that provide operating system independence for i/o, memory management, and event handling, as well as functional modules such as lists, compression modules, and sequence matching.

API: a library that sits on top of the KIT and provides a set of routines for managing devices.

The API and the Kit are provided on various platforms such as Solaris, Linux, and Windows NT and 2000. Wherever possible (Solaris), they are provided in 64 bit mode. This natively compiled 64 bit library makes available the full 64 bit address space.

There are four sets of libraries built using the API and Kit libraries , the **Piper Library**, the **Data Normalizer Library**, the **In-Memory Hierarchical Database Library**, and the **In-Memory Stack Database Library**.

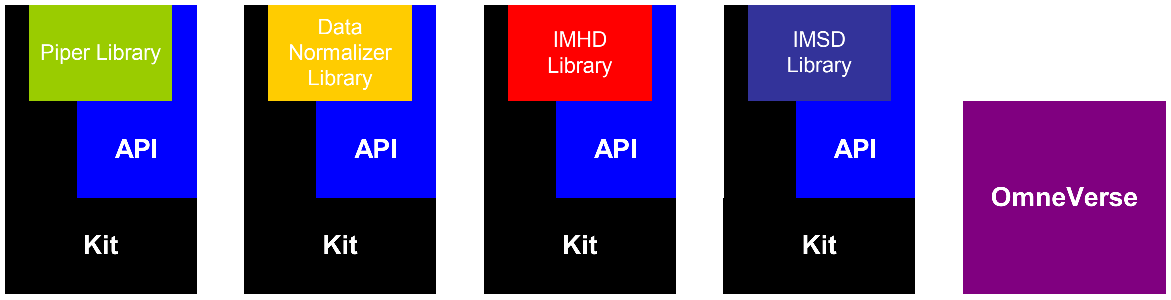

Figure 2 - The OMNE™ Libraries

- **Piper Library:** a thin library for handling the physical connection to third party systems.

- **Data Normalizer Library:** a library that provides a convenient frame work for the conversion of data between third party formats and the format most commonly used by The OMNE™.

- **The In-Memory Hierarchical Database Library:** a library that provides a convenient framework for the implementation of an in-memory, hierarchical, and optionally replicated database.

- **The In-Memory Stack Database Library:** a library that provides a convenient framework for the implementation an in-memory, stack database generally used to maintain lists of messages in reverse order of their receipt.

Java Interface

The Java interface to The OMNE™ is called OmneVerse™. The functions of the Kit and the API are provided by Java classes compiled into a single JAR file.

The Kit Library

The Kit library is object-oriented in design and implementation. Each module provides entry point functions for initialization and un-initialization. References to instances are viewed as opaque pointers by the developer and implementation details are hidden from view. All operations are by means of methods operated on instance handles created during a module's initialization.

OS Independence

The Kit library provides a uniform interface to operating system calls like file, socket, time and event operations. This solves the problem of porting to a different platform from the point of view of the developer, and it allows The OMNE™ to provide functionality above and beyond that provided by native system calls.

The concept of a self-delimited *OS Message* is defined which enables atomic reads and writes for file and socket devices (however, raw reading and writing is supported).

The native system calls used by the Kit library (in Windows NT, Windows 2000, Linux, and Solaris) are a conservative set guaranteed to be provided on most platforms.

Modules

The Kit library provides implementations of various data structures and constructs like linked lists, hash functions and tables, dynamic arrays, b-trees, compression and encryption, and memory managers for use by The OMNE™ or by the developer.

Socket Device Interface

The socket device interface integrates various protocols including TCP, UDP, HTTP and SSL. The interface also incorporates a measure of authentication using a handshake protocol that verifies the server to the client. This is implemented as a precautionary measure to determine whether the connection endpoint speaks the correct language and is the correct socket for the desired service.

Namespace Management

The namespace of sockets provided by the kernel (protocol, IP address and port number) is subsumed into an OMNE™ namespace that consists of *domain* and *name*. This is implemented by an OmnePresent™ server process (the Location Broker, discussed later in this document) which manages the maps that translate back and forth between the two schemes. This liberates developers, users, and administrators from always having to know the actual location of connection endpoints.

Versatile Messaging

One of the Kit library modules defines a tagged message format. The field identifiers (tags) are 64K in number, of which a small subset are reserved for use by the API library and by some of The OMNE™ infrastructure processes. The semantics of the other field identifiers are left to interpretation by user (developer) defined OmnePresent™ applications and can mean different things for different applications, as desired. Each message can have more than 2 billion items, each of which is tagged by one of the 64K different values. While 64K tags have proven to be adequate, the design allows for the expansion of field tags to an arbitrary number, if so required.

An arbitrary number of these tagged messages can be marshaled into an *OS Message*. The OMNE™ makes aggressive use of buffer packing to optimize network throughput.

The API Library

The API library consists of a set of routines that provide the developer with the quickest means to write an OmnePresent™ process that interacts with an OMNE™ infrastructure. Basic functions are provided that allow a user to connect to other OmnePresent™ processes, read and write files, and exchange and process messages. Other API library functions build upon these to provide publish/subscribe and request/reply services. The tasks of event handling, connection management, and remote administration are handled by the API, freeing the developer to write hook functions that process messages. The API library calls upon various modules in the Kit library during operation. Both asynchronous and synchronous network operations are supported.

Object Oriented Design

The API library, like the Kit library, is designed with object-oriented principles. The concept of a device is abstracted from timers, files, and sockets. Also, the concept of an I/O object is abstracted from files and sockets. A device open or close function will behave differently depending on the type of device it is operating upon.

Integration With Third Party Libraries

A typical OmnePresent™ application makes use of the OMNE™ event loop, in which various entry points are provided for initialization and message processing. However, in the case of an application using a third party library with its own event loop, it is possible to make the OMNE™ event loop run subservient to the native event loop. A timer is registered with the third party event loop which then services all OMNE™ devices (sockets, files and timers) before returning.

A simple linear back-off algorithm is followed in determining the timer interval, to allow the OMNE™ event loop to adapt to idle or busy conditions efficiently.

Remote Administration

All OmnePresent™ applications can be administered remotely. Device statistics can be monitored, and devices can be added, removed, opened or closed at will. OmnePresent™ applications, by default, open a listener socket that provides this administration service.

Uniform Design

All processes, whether user (developer) defined applications or infrastructure processes, are built using the same set of OMNE™ libraries. This uniformity makes administration simple and maintenance easy.

The Piper Library

This library facilitates the connection between third party systems and OmnePresent™ programs built using the Data Normalizer Library. OmnePresent™ programs built with the Piper Library package third party messages and send them to an OmnePresent™ data normalizer (and vice versa).

The Data Normalizer Library

This library provides the developer with tools needed to build a program that manages the conversion of data between third party formats and the formats most commonly used by OmnePresent™ systems. Additionally this library provides developers and system administrators with the tools to manage data flows to and from multiple third party systems within a single process.

The In-Memory Hierarchical Database Library

The in-memory hierarchical database library provides the developer with tools needed to build an in-memory, hierarchical database. The hierarchy (key definition) of the database and other parameters are configurable. Processes built using this library connect to a CBR™, subscribe based on their hierarchy and export services for lookup and update (add, delete or modify). Some features provided by this library are :

- The scope of the lookups can be specified - defining the key-set for any point of the data tree will retrieve data at that level including a list of the keys at the next level allowing one to walk down the tree without knowing the complete key-set beforehand.
- Selection criteria can be specified to filter the entire database for matching records.

- Multiple occurrences of data values are handled optimally by reference counting.
- A snapshot of the database can be written to a file by a remote administration call to preserve the state between invocations.

Redundant hierarchical databases can be synchronized by a method provided by this library or by methods provided by the CBR™. When using the library's method, updates are written to a file so that a database can be restarted and its state recreated.

The In-Memory Stack Database Library

The in-memory stack database library provides the developer with tools needed to build an in-memory, stack database (an in-memory stack database is generally used to maintain lists of messages in reverse order of their receipt – a trade history database, for example) . Processes built using this library connect to a CBR™, subscribe based on a configurable key, and export services for lookup and update. As each publication message arrives from the CBR™ they locate the key field and its data and pop the message on a stack associated with that data. These processes can be configured to restrict the storage on the stack to a subset of the data in the message.

The OMNE™ Infrastructure Processes

The Content Based Router™

The Content Based Router™ (CBR™) lies at the heart of The OMNE™. The CBR™ and its client processes are arranged in a Hub-and-Spoke topology. All messages between participating clients are routed through the CBR™ (peer to peer communication without using a CBR™ is supported but not generally used).

CBR™ Trees

A CBR™ can be a client of another CBR™ and several CBR™s may be connected in a tree structure. This allows an OmnePresent™ system to scale without redesign as its demand for use and performance expand. As new demands (new users, etc.) arise, CBR™s can be easily added to expand the capacity of the system. The clients of a CBR™ tree in effect belong to a virtual CBR™ that enables them to communicate with each other as if they were all clients of the same CBR™.

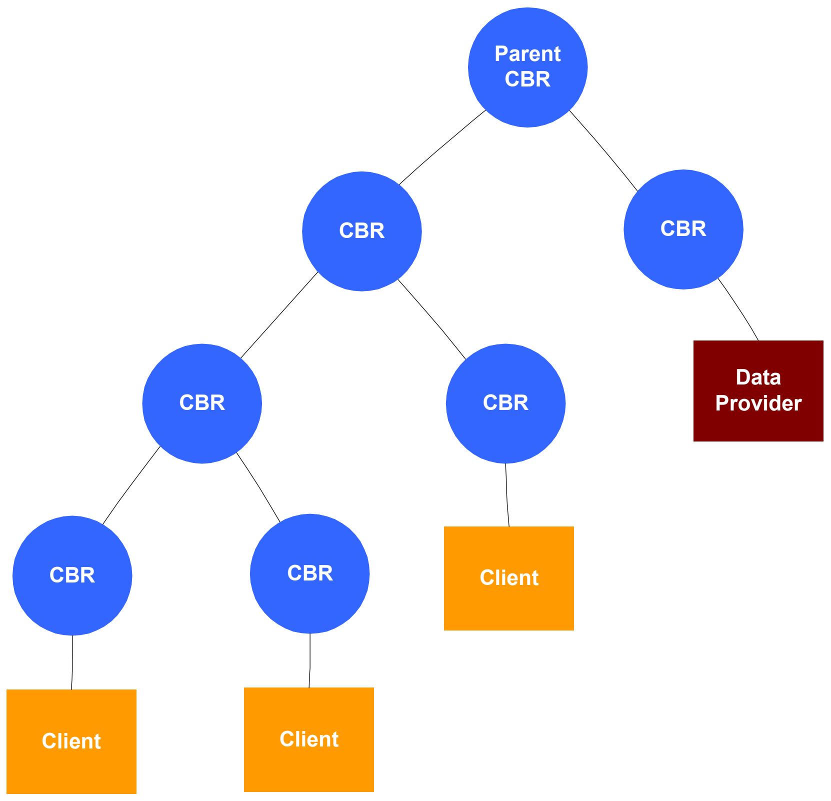

Figure 3 - CBR™ Tree

Subordinate and Peer Modes

CBR™ trees may connect in one of two modes. A subordinate (read only) mode where one tree is subordinate to the other and a peer mode where two trees connect to each other. In the subordinate mode the subordinate (connecting) tree can get any publication from and submit any request to its superior tree but cannot send publications nor respond to requests from its superior tree. In the peer mode each tree can pull data from the other as long as the other is configured to permit it (pull in this case is actually the establishment of criteria so that data is pushed). The subordinate mode is generally used when a single system (a real time ticker plant, for example) feeds data to several similar but separate OmnePresent™ systems (a development system, a UAT system and/or a production system). The peer mode is used when a relatively small subset of data from one tree is useful in another or when the link between two trees is slow (hence the need to restrict the data exchange) or expensive. It can also be used when data exchange between the trees has to be monitored or regulated.

Publish/Subscribe and Request/Reply Paradigms

Both the anonymous publish/subscribe and the request/reply paradigms are supported by the CBR™. A client subscribes to data messages by indicating various fields and/or field/data combinations of interest. The CBR™ scans each message that it receives, compares it with the subscription criteria of each client and forwards it to matching clients. Requests for a service made by a client are forwarded to a client that has registered the service, and the response is returned to the requesting process. Various simple paradigms of service registration (forward to all, round robin, etc.) are implemented. The round robin model allows a simple yet effective way of achieving load balancing.

Scalability

Since all messages are constructed of field/data pairs, this subscription mechanism allows for a complete description of the message itself (the content) without resorting to ad hoc descriptive subjects and headers. The decision to view or not view a message lies solely with the consumer of the message (with help from its CBR™), not with the publisher. The publisher is relieved of the responsibility of having to know its consumers. This greatly eases the deployment and maintenance of distributed systems. When new consumers appear, existing publishers are unaffected.

Reliable and Highly Reliable Modes

The CBR™ and its clients are capable of operating in a highly reliable mode. In this mode, all messages are tagged with a sequence number and stored by the sending process (the client, in the case of the publisher and the subscriber) and the CBR™ in indexed data files. Each message is explicitly acknowledged by the receiver. It then is removed from the files at the sender's convenience. When a connection is recovered after a disruption, the last sequence number acknowledged is exchanged and unacknowledged messages are sent/resent. A CBR™ stores all the messages from various publisher clients in a single repository. Clients that have reconnected after a break are sent only the subset of stored messages that match their subscription criteria.

Quiet Mode

A CBR™ also helps its clients operate in *quiet mode*. Such a client may subscribe for messages and register services but while in *quiet mode* its publications and service responses are inhibited. This mode is used to ensure that only one of a set of redundant services provides data yet all members of the set update similarly.

The Location Broker

The Location Broker is built using the In-Memory Hierarchical Database Library. It maintains location information about sockets, organized by name. It is one of the few processes that, though it connects to a CBR™ tree, gets its main data directly from a CBR™. A Location Broker replicates using the method provided by the In-Memory Hierarchical Database Library.

The Domain Server

The Domain Server is built using the In-Memory Hierarchical Database Library. It maintains location information on Location Brokers, organized by domain name (it is effectively the Location Broker's location broker). It is one of the few processes that, though it connects to a CBR™ tree, gets its main data directly from a CBR™. A domain server replicates using the method provided by the In-Memory Hierarchical Database Library.

The Field Id Server

The Field Id server is built using the In-Memory Hierarchical Database Library. It is configured with the following hierarchy : application name, user id. It provides a general repository for storing a cross reference for field identifiers and their meanings (usually for display purposes). Data stored at the application level usually corresponds to an application's default meanings for field identifiers. Data stored at the user id level usually corresponds to the meanings for the field identifiers as defined for/by the user associated with the particular user id for the particular application.

The Logger

The Logger is the recipient of messages sent by processes using the logging facilities provided by the Kit. It is one of the few processes that, though it connects to a CBR™ tree, gets its main data directly from a CBR™. It is the only process that by design gets its data using UDP sockets. Upon receipt of a logged message, a logger locates a text field and writes its data to its output file (enhanced with receipt time and date info). It then publishes the message (enhanced similarly) to its CBR™. The logger is part of an isolated CBR™ tree (only loggers connect to this CBR™ tree) to which monitoring processes connect to view commonly logged information in real-time (message rate info, subscription info, etc).

Performance Observations

The CBR™

Reliable Mode

In every day use at one client site, where an OMNE™ is used to receive and distribute real-time market data, we have observed the following :

- The top CBR™ of the market data tree gets between 10,000 and 12,000 messages per second during the NY market open (9:30am – 10:00am, Monday – Friday).
- Each message tends to span more than 150 bytes.
- This CBR™ then sends nearly all (> 95%) of its messages to 1 subordinate CBR™ and to a process that timestamps each message and writes them to a file.
- This CBR™ also sends 10% – 15% of its messages to a subordinate CBR™ that distributes subsets of them to various trading systems and user's screens.
- Effectively, this CBR™ is handling more than 30,000 – 36,000 messages per second routinely.

In tests we have observed that a single CBR™ can handle more than 60,000 messages per second.

Highly Reliable Mode

In tests we have observed that a single CBR™ can handle more than 7,500 messages per second.

A Data Normalizer

In every day use at one client site, where an OMNE™ is used to receive and distribute real-time market data, we have observed that a process built using the Data Normalizer Library gets, translates and sends between 9,000 and 10,000 messages per second to a CBR™ during the NY market open (9:30am – 10:00am, Monday – Friday).

An In-Memory Hierarchical Database

In every day use at one client site, where an OMNE™ is used to receive and distribute real-time market data, we have observed that a process built using the In-Memory Hierarchical Database Library receives between 10,000 and 12,000 messages per second from its CBR™ during the NY market open (9:30am – 10:00am, Monday – Friday). This process maintains real-time information (prices, sizes and volumes) on more than 850,000 primary keys and on more than 1,100,000 secondary keys (financial instruments).

An In-Memory Stack Database

In every day use at one client site, where an OMNE™ is used to receive and distribute real-time market data, we have observed that a process built using the In-Memory Stack Database Library receives between 2,000 and 3,000 messages per second from its CBR™ during the NY market open (9:30am – 10:00am, Monday – Friday). This process maintains real-time information (trade history) on more than 85,000 keys (financial instruments).