Talarian™: Everything You Need To Know About Middleware

A Guide To Selecting A Real-Time Infrastructure

Contents

Executive Summary

In today’s real-time world, businesses need to exchange information instantly, reliably and securely with customers, partners and suppliers around the globe via the Internet. Because this involves complex applications running across multiple platforms using data distributed across the enterprise, forward thinking CIOs and enterprise architects are finding middleware to be the core infrastructure on which to build their applications.

For years, middleware has been used to handle data movement across disparate platforms and between incompatible databases. In addition, middleware has made it easy for developers to handle the differences between platforms and databases by writing applications that connect to the middleware layer and by letting the middleware handle the data translation and transport.

This model has proved itself extensively, and many large sites rely on middleware for truly mission-critical applications. In today's world though, middleware must do more. Not only must it provide a uniform interface to disparate systems, but it also must serve as the fundamental corporate data transport, both within the enterprise and between the enterprise and the Internet. Hence, middleware today needs to be robust enough to handle the message load of the busiest sites in real-time while providing guaranteed delivery of messages throughout the enterprise.

Several types of middleware provide targeted solutions for this problem. Transaction processing monitors, for example, provide data integrity for database transactions. Object request brokers handle the interaction of objects—such as those in C++ and Java—across disparate platforms. And specialized services, such as Java™ Message Service (JMS), help provide tailored solutions to unique problems. The core middleware, which is a superset of all these models, is message-oriented middleware (MOM). By use of messages, MOM enables all forms of communication between programs—be it transaction processing, object sharing, or Java communication—to occur between any two points in the enterprise.

Talarian's SmartSockets® product implements MOM within the constraints of high-speed, real-time message delivery. SmartSockets is also known for its scalability: it can be deployed on a small network of a few nodes or a network of thousands of clients who need to share real-time data broadcasts in a secure environment where mission-critical reliability is a must. SmartSockets is also deployed at numerous sites where the real-time aspect is not paramount but where enterprise system architects want a messaging infrastructure that does not consume large amounts of bandwidth but is still capable of spanning a wide variety of platforms and clients without difficulty.

The rest of this pamphlet discusses the types of middleware available today, the strengths of each, and the best applications for each kind. It then explains Talarian's SmartSockets in depth.

What is Middleware?

As the distributed model of enterprise computing has become more common, the term middleware has acquired numerous meanings that would allow it to be just about any piece of software that sits between systems. Terms such as enterprise application integration (EAI) and extensible mark-up language (XML) often are mistakenly used to describe middleware.

In the strict sense, middleware is transport software that is used to move information from one program to one or more other programs, shielding the developer from dependencies on communication protocols, operating systems, and hardware platforms. Middleware provides the “plumbing” necessary for applications to exchange data, regardless of the environment in which they are running. Transactions, data broadcasts, EAI packages, and XML data often ride on middleware in the enterprise.

The concept of middleware dates back to the 1980s when companies wanted one package to move data between mainframes, databases, and user terminals. Modern middleware extends this concept to the widespread distribution of data in real time across a remarkable variety of servers, clients, and sites.

Middleware as used in this sense tends to be message-oriented. That is, data is sent between systems in messages, which are similar to data packets on the network. These messages have headers that indicate the destination and payloads of varying sizes and formats that contain the actual data. Message-oriented middleware (or MOM) originally appeared in the form of message queues.

When a message was sent to another system, it was stored in a message queue on the destination system. Whenever the destination system needed the data, it looked in the queue for the message. If it was there, the message was retrieved; if not, the system would wait until the data arrived in the queue. This approach proved reliable, but slow. It still is used today in many transaction-oriented environments, where security of transactions and integrity of message delivery is a high priority.
A second model, called publish and subscribe (or pub/sub), evolved from the need to deliver messages in real time, especially to a large number of clients. In the pub/sub model, clients register for certain kinds of messages they are interested in, and a server sends the clients those messages in real time. The emphasis of the pub/sub model is to send data from one server to many clients as fast as possible. Typical applications might be stockbrokers needing the latest prices on certain bonds or equities. These prices typically are sent in real time to all brokers who subscribe to this information.

One company today, Talarian Corp., combines the two models of MOM: its product SmartSockets delivers messages in real time with the reliability and integrity of message queuing. In fact, SmartSockets can be installed as either a pub/sub implementation or a message-queuing package.

Specialized Kinds of Middleware

The previous section briefly introduced the two types of message-oriented middleware. Other types of middleware are commonly found today performing narrow functions.

The middleware market can be broken into five different segments:

1. Transaction processing monitors
2. Remote procedure calls
3. Object request brokers
4. Homegrown middleware solutions
5. Message-oriented middleware (MOM)

Transaction Processing Monitors

Typically, transaction-processing (TP) monitors are not used for general purpose program-to-program communication. Rather, they provide a complete environment for transaction applications that access relational databases.

In TP monitors, clients invoke remote procedures that reside on servers, which also contain a SQL database engine. Procedural statements on the server execute a group of SQL statements (transactions), which either all succeed or all fail as a unit. The applications based on transaction servers are called on-line transaction processing (OLTP). They tend to be mission-critical applications that require a rapid response 100% of the time and tight controls over the security and integrity of the database.

The communication overhead in this approach is kept to a minimum because the exchange typically consists of a single request/reply (as opposed to the multiple SQL statements required in database servers).

TP monitors provide application development tools (such as user interaction and database interfaces), system administration (such as security and tuning), and transaction execution (such as scheduling and load balancing).

X/Open, a vendor-neutral standards group, has done a considerable amount of work toward defining a process model and related services interfaces for distributed processing applications. Most vendors have pledged to support some or most aspects of the X/Open model.

TP monitors should be considered when transactions need to be coordinated and synchronized over multiple databases. TP monitors tend to be heavyweight and expensive, and they require a great deal of expertise to implement properly. Most TP vendors have a large service side to their business.

Remote Procedure Calls (RPCs)

RPCs have been around for a long time. They are one of the earliest forms of interprogram communication, and they operate at a very low level. From a programmer's point of view, RPCs are easy to understand. The code invokes a procedure that is located on a remote system, and the results are returned. Generally, the application components communicate with each other synchronously, meaning they use a request/wait-for-reply model. RPCs work well for smaller, simple applications where communication is primarily point-to-point (rather than one system to many). RPCs do not scale well to large, mission-critical applications, as they leave many crucial details up to the programmer, such as the following:

1. handling network or system failures
2. handling multiple connections
3. portability
4. buffering and flow control
5. synchronization between processes

Due to their synchronous nature, RPCs are not a good choice to use as the building blocks for enterprise-wide applications where high performance and high reliability are needed. The RPC standards have not evolved in any significant way during the last five years, primarily because of the emergence of the Object Request Brokers described in the next section.

Object Request Brokers

Object Request Brokers (ORBS) can be thought of as language-independent, object-oriented RPCs. There are two competing standards for ORBs:

1. CORBA, backed by more than 700 companies from the Object Management Group (OMG)
2. DCOM, backed by Microsoft

(Java's Remote Method Invocation (RMI) could be considered an ORB, although it is useful primarily for facilitating communication between two programs written in Java and does not address other programming languages as do both DCOM and CORBA.)

ORBs are designed for use in projects that require a strict object-oriented approach, where "objects are the only way." Like RPCs, ORBs are generally synchronous and operate in a point-to-point manner. In general, both CORBA and DCOM assume the system has a reliable communications layer, and they do not address the problems involved when this layer is not reliable.

Early on, the OMG recognized that CORBA's request-reply communication was not going to be adequate for building true, enterprise-wide, mission-critical applications. Some of the CORBA vendors added proprietary extensions to their products to address these shortcomings. The OMG specified the CORBA Event Service, a standard set of services layered on top of CORBA, which brought most of the vendor extensions into the CORBA model. In 1998, the OMG approved the Asynchronous Messaging Service. However, this facility is not widely used in CORBA deployments today.

Homegrown Middleware

When companies first encounter the need for a middleware layer, they often have a specific problem to address that requires a modest solution. Rather than invest in a middleware package, smaller firms will allow their own development staff to write a middleware-like solution to solve the particular problem. Although initially workable, this approach tends to lack scalability and flexibility as new problems have to dovetail with the old solution. As a result, supporting the homegrown middleware becomes expensive as it has to be customized and extended constantly—generally by staff members who have never written middleware software before. The final result is an expensive solution that tends to break easily and does not scale well.

Message-Oriented Middleware

In general, MOM products work by passing information in a message from one program to one or more other programs. The information can be passed asynchronously, where the sender does not have to wait for a reply. MOM products, in general, cover more than just passing information; they usually include services for translating data, security, broadcasting data to multiple programs, error recovery, locating resources on the network, cost routing, prioritization of messages and requests, and extensive debugging facilities. Unlike both ORB and RPC products, MOM, in general, does not assume the system has a reliable transport layer underneath. MOM tries to address the problems that surface when the transport layer is unreliable, as occurs when programs must communicate over a WAN or over the Internet.

Two different types of MOM have emerged:

1. message queuing
2. message passing

Message Queuing

In message queuing, program-to-program communications occur via a queue, which is typically a file. It allows programs to send and receive information without having a direct connection established between them. A program simply gives messages to the message queuing service, identifying by name the queue in which it wishes the message to be placed. The message queuing service acts as an intermediary, and the mechanism by which the message is transmitted is completely hidden from the application programs. In large, enterprise-wide applications, queues can be set up to forward the messages to other queues. Message queuing provides safe storage of information and is most appropriate where

applications cannot be connected directly (for example, in mobile computing). However, message queuing tools require considerable configuration to set up correctly and performance can be poor. If access to a queue is lost for any reason, the entire system can be affected.

Message Passing (Publish-Subscribe)

Message passing has proven popular for building large, distributed applications. This approach differs from message queuing in that rather than oblige applications to retrieve the information they request, the information is more efficiently pushed to the interested parties. One increasingly popular flavor of message passing uses a model of communication known as publish-subscribe (pub/sub). In pub/sub, programs subscribe to (register interest in) a subject. Programs also publish (send) messages to the subject. Once a subject has been subscribed to by a program, the program will receive any messages published to that subject in the distributed application. Subjects are defined by the application developer.

In traditional network applications, when two processes must communicate with each other, they need network addresses to begin communicating. If a process wants to send a message to many other processes, it first would need to know the physical network addresses of the other processes and then create a connection to all those processes. This architecture does not scale well because configuration is complicated and tedious. The publish-subscribe communications model provides location transparency, allowing a program to send the message with a subject as the destination property while the middleware routes the message to all programs that have subscribed to that subject.

MOM vendors typically implement publish-subscribe with a set of agents that maintain a real-time database, listing which programs are interested in which subjects. A program publishes a message by connecting with one of the agents (it may or may not be on the same machine) and sending the message to it. The agent then routes the message to the appropriate programs. Often, the pub/sub middleware has greater fault tolerance because the agents can perform dynamic routing of the messages as well as provide hot fail over should any of system fail. Pub/sub is most appropriate for highly distributed applications where fault tolerance and high performance are important. It does not work well in situations where processes may be disconnected from the network for long periods of time.

Java Message Service (JMS)

The JMS is a specification published by JavaSoft, the division of Sun Microsystems responsible for developing and managing the Java language. It specifies a set of programming interfaces by which Java programs can access MOM software. Most MOM vendors today have announced they intend to support JMS in future releases, while a few leading-edge vendors, such as Talarian, already have implemented support for JMS in their products.

The Middleware Market

Due to the incursion of the Internet into enterprise computing and the corresponding move to a distributed-computing model, the middleware market has seen tremendous growth during the past several years. Most analysts expect this growth to continue unabated through the early years of this decade.

Technology Forecast: 2000 from PricewaterhouseCoopers quotes a Dataquest survey of 547 IT shops and their buying plans for all forms of middleware. (See Figure 1.) The type of middleware that the largest number of sites expects to purchase is MOM. During the period from 1998 to 2003, PricewaterhouseCoopers expects the MOM market to grow at a compound annual growth rate (CAGR) of 19.6%

Talarian is recognized in the same report as a key innovator and one of the leading vendors of MOM software. And indeed, since 1988 Talarian has been leading the field in the delivery of real-time MOM infrastructure.

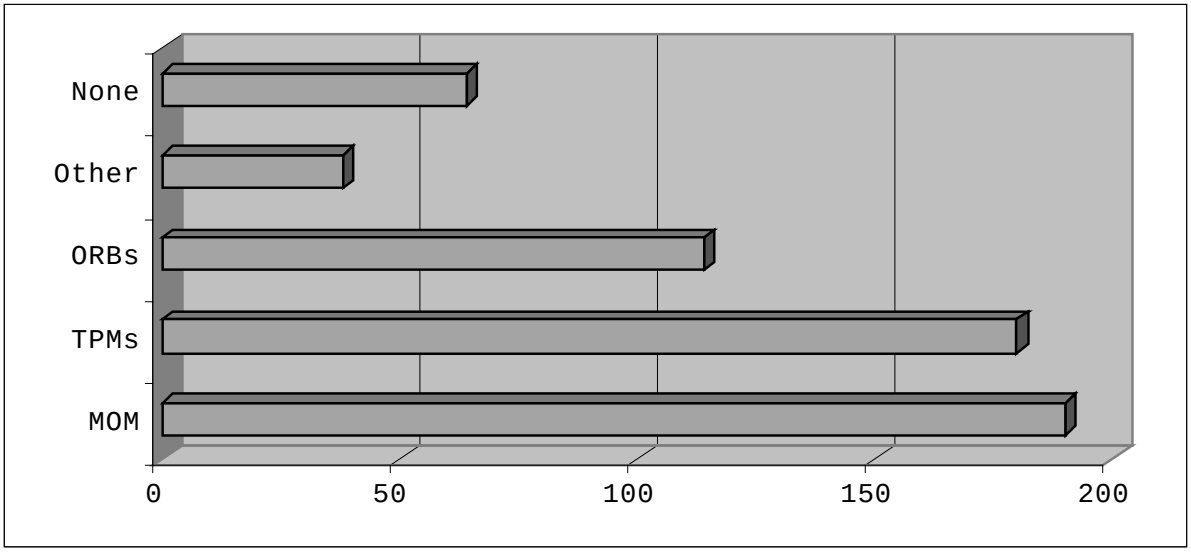

Figure 1. Types of middleware most IT sites are likely to purchase.
(Source Dataquest, quoted by PricewaterhouseCoopers)

Key Features of Modern Middleware

Today’s mission-critical systems make extensive use of distributed computing to share information over large, heterogeneous networks. At the heart of these mission-critical systems is the difficult task of passing information reliably between many applications running on different computers. Today’s IT environment often includes a mix of mainframes, UNIX workstations, and PCs trying to communicate over multiple network protocols.

The need for distributed applications runs the gamut from the classic commercial replication of a database to today's more exotic push technologies where selected information is pushed out over the Internet in real time to interested subscribers. There are increasing amounts of information being made available and ever greater demands for instant access to the information. In many applications, high volumes of information (often passed in messages) must be distributed throughout the network. Examples of such applications include the real-time collection, monitoring, and distribution of data from sources such as factory floors all the way to the data centers where inventory, order entry, and personnel information is kept.

Historically, the communication infrastructure for these distributed applications was built in-house using low-level interprocess communication primitives such as sockets, pipes, shared memory, RPCs, and sometimes even files. Information was typically encoded in a structured packet of information called a message (not to be confused with an e-mail message). These home-brewed solutions were relatively straightforward to develop for very small applications running on LANs. But as the distributed applications grew larger and more heterogeneous, using these primitive mechanisms became less attractive because of the myriad complexities to deal with. In-house solutions are starting to break as the complexity of the network grows and the software is being used on ever more demanding applications.

Challenges of Network Programming

The term network programming is used to define the software development process of building applications that must communicate with one another over a network such as a LAN, WAN or even the Internet. No matter what the nature of the application, establishing program-to-program communication that operates reliably over a network using sockets, shared memory, RPCs or some other interprocess communication mechanism is an arduous task. Network programming can be a software developer's worst nightmare. In projects involving distributed applications, building the interprocess communication is often the most difficult and risky part and often the cause of behind schedule and over budget client/server applications.

With the number of mission-critical distributed systems growing rapidly, a new market has emerged for off-the-shelf middleware.

Before IT managers can identify the right middleware solution, they must understand the difficult challenges of network programming. The following reveals and clarifies those challenges.

Encoding the Information

Information that is passed from one program to one or more other programs must be encoded (formatted) in a manner that all the participating programs understand. Upon receipt of the information, programs must know how to access and decode. If developers are using RPCs or an (ORB), the encoding/decoding rules are defined in an interface definition language. Some other forms of middleware use self-describing messages in which the sending and receiving processes do not need to be tightly coordinated to exchange information. Upon receipt of the

message, the message itself can be queried to understand how it has been encoded.

Data Translation

As information is passed across the network, it often will go from one type of platform (for example, Sun SPARC) to another type (such as Intel x86 PCs). Often, the platforms will not have compatible formats for representing different types of data such as floating point numbers and text strings. The network programmer must translate the different types so they are understood by the receiving processes. Ideally, this translation should be completely transparent and done only one time, at the final destination (receiver makes right).

Multi-Protocol Support

Today's distributed computing environments use many different network protocols, including TCP/IP, IBM's SNA, multicast, Digital's DECnet and many others. Information moving from one program to another across the network may need to be transferred from one protocol to another; this is called context bridging.

Finding Resources

To build a large, distributed application, many resources (for example, processes) must be tracked. For these resources to share information, they often need to know each other's location and how to communicate. The location of the resources changes frequently as new applications and new servers are brought online. The ability to find and track resources is often called naming services.

Flow Control

In a distributed application, programs are sometimes unable to keep up with the flow of information, or they can be busy doing something else when information arrives. This can cause programs and complex applications consisting of multiple programs to get out of sync and potentially use large amounts of system resources, such as memory or CPU. Developing software to handle varying traffic rates can be a complex task.

Portability and Standards Issues

There are few standards for passing information across a network. The few existing standards often conflict with one another. Some of the existing standards include sockets, streams, named pipes, RPCs, and ORBs. Even at this level, there are often multiple, competing standards. For example, there are several different versions of RPCs, including ONC and DCE. Examples of ORBs are OMG's CORBA and Microsoft's COM/OLE. Even Java's RMI could be considered an ORB. Lower-level interprocess communications (IPC) socket primitives include Berkeley UNIX sockets, UNIX System V sockets, and Winsock. Berkeley sockets, which are supported on many different platforms, often have small, subtle differences across different flavors of UNIX operating systems. Winsock is similar to Berkeley sockets, but contains some significant differences.

Asynchronous Operation

In a distributed application, a program often needs to send information without waiting for a reply. This implies that the reply will come from one or more of the receiving programs at some indeterminate time in the future. Working in an asynchronous environment makes the job of the network programmer more difficult.

Often, programming mechanisms such as callbacks are used to handle asynchronous behavior. These are functions that are executed when a specific type of event occurs (for example, a reply arriving). Another common way to achieve asynchronous operation is through the use of threads, where a program may have multiple threads sending and receiving over a single connection. This implies the middleware used must be thread-safe.

Configuring Resources/Programs on the Network

In a distributed computing environment, uniquely identifying a machine or program often is done through a complex numbering or naming scheme. For TCP/IP networks, IP addresses of the form xxx.xxx.xxx.xxx are used to uniquely identify a machine. Network programmers must understand and use these arcane facilities when building their applications. Reconfiguring an application when a machine is added or removed from the network can be complex because source code may need to be changed and the entire project may be required to go through another cycle of integration and test.

Dealing with Hardware or Software Failures

One major problem facing today's client/server computing environments is the higher likelihood that some component of the application will fail or partially fail: the network, a machine, or a program on a machine. Recovering from such failures can be difficult. Often, complete recovery is not possible, and the application must gracefully degrade as resources are lost.

Therefore, fault tolerance is an important feature of any solution.
Even detecting something as simple as a network failure is often protocol dependent. TCP/IP was designed to hide errors and try to correct them without any intervention. It does a very poor job of informing the programmer when errors occur. Generally, TCP/IP will let the programmer know only about unrecoverable errors, and it is not unusual for minutes to go by before TCP/IP reveals it has a problem. Without some type of proactive approach, recovery from such problems is almost impossible. This is not acceptable for most of today's mission-critical applications.

Managing Multiple Clients and Servers

In a distributed application, there can be many programs exchanging information over many connections. Often, a program wishes to send a message to multiple receiving programs, all with a single operation. Other times, a program would like to send a message to the “least busy” of a specified set of programs in order to get a quick reply. Coordinating multiple programs running across a heterogeneous network can be a difficult job for a network programmer.

In managing multiple clients and servers, IT professionals have to be careful in the use of network bandwidth to keep all the programs coordinated. A common problem is flooding the network with information whenever some type of change occurs to the distributed system (for example, a program starts or stops). Whenever this happens, a large amount of network chatter can occur, which can use large amounts of network bandwidth and put the system in an inconsistent state until the change has been propagated.

Changing Environment

The network environment on which the distributed application executes changes quite frequently. New computers are brought in to replace old computers or to add more capacity to the enterprise. New network equipment—such as bridges, routers, and switches—is added to make more efficient use of the network. Even interfaces between the software programs change as the scope of the distributed system evolves over time. This dynamic environment can wreak havoc in even the best-designed systems.

Some types of middleware have great difficulty handling these changes. Middleware that depends on an interface definition language, such as ORBs and RPCs, relies heavily on static definitions that will not change. Changes to the interface require changes to the interface definition and the recompilation and relinking of the code. The OMG’s CORBA has defined a dynamic invocation interface (DII) to handle this problem. In practice, few developers use the DII because it is very difficult to understand and program.

Publish-subscribe middleware uses the idea of subjects to handle the frequent reconfiguration of a distributed application. Changes to the interfaces between programs are handled by self-describing messages, whose definition may change on-the-fly.

Debugging and Analysis

Distributing programs across a network can provide large benefits to an organization—supplying real-time information to users whenever and wherever they need it. The payback in productivity and quality realized by the organization can be enormous. Unfortunately, building distributed applications is difficult, and trouble-shooting and debugging problems have the potential to disrupt critical elements of the operation when these applications experience failures.

Effectively debugging a distributed application requires fine-grained visibility into the communication between programs. In general, the primitive interprocess communication (IPC) mechanisms provided by operating systems notoriously lack debugging facilities, making applications that use IPC very difficult to debug. The problem is further exacerbated when processes are distributed across a network in the quest to build scalable applications. In this case, effective debugging tools are a necessity, and their availability reduces development time significantly.

Talarian’s SmartSockets, a robust MOM product that addresses many of the problems described in this section, is the principal Talarian product. The rest of this booklet offers an introduction to SmartSockets.

SmartSockets®: An In-Depth Look

Talarian's SmartSockets is a robust, message-oriented middleware product that offers both message queuing and publish-subscribe communication models. SmartSockets enables programs to communicate quickly, reliably, and securely across LANs, WANs, and the Internet. SmartSockets manages network interfaces, guarantees delivery of messages, handles communication protocols, and deals with recovery after system/network problems. This lets software developers use their skills to handle higher level requirements, rather than to solve the underlying complexities of the network.

SmartSockets greatly reduces risk in software projects where multiple programs must communicate with one another. It speeds development and ensures portability and interoperability. SmartSockets can be used in a broad range of technical applications including real-time command and control, high-performance message passing, high availability solutions, and multi-tier client/server applications. SmartSockets' programming model is built specifically to offer high-speed interprocess communication, scalability, reliability, and fault tolerance.

The following components make up the SmartSockets product:

1. Application Programming Interface—a C/C++ callable library of functions for working with messages, communicating between processes, and monitoring distributed applications.

2. C++ Class Library—an object-oriented layer on top of the standard SmartSockets services.

3. Java Class Library—a version of SmartSockets written natively in Java offering Java programmers access to the publish-subscribe features of SmartSockets.

4. RTserver™ process(es)—a powerful message router that allows applications to use a publish-subscribe communications model; processes register interest in specific subjects, clients publish (send) messages to a subject, and all processes that registered for that subject receive the messages.

5. RTmonitor™ process(es)—a graphical point-and click interface for monitoring, administering, and debugging a distributed application; allows developers to use a visual interface for watching things such as IPC traffic and process information.

6. Ready-to-use message types—predefined message types to get the application developer going quickly. Custom message types can be developed with ease.

7. Sample C, C++, and Java Programs—an extensive set of sample C, C++, and Java programs to get you off to a fast start.

8. Documentation—available on-line and in print, a complete bound set of manuals that are filled with examples. Included is an easy-to-read tutorial.

SmartSockets has several capabilities that make building distributed applications easier, while keeping the system up and running. SmartSockets' programming model is built specifically to offer high-speed interprocess communication, scalability, reliability, and fault tolerance. An application developed with SmartSockets can consist of multiple programs working together in a heterogeneous network. The following paragraphs describe the key features of SmartSockets.

High-Speed Interprocess Communication

SmartSockets supports very high-speed message routing and delivery throughout a network. SmartSockets provides access functions so developers can quickly encode, decode, and copy

messages. SmartSockets will transparently determine the shortest route between programs using a shortest-path algorithm. In general, the rate at which information can be passed using SmartSockets IPC is restricted only by the limitations of the physical network.

Publish-Subscribe Services

SmartSockets uses a powerful publish-subscribe communications model in which programs subscribe (register interest) to a subject. Programs also publish (send) messages to the subject. Once a subject has been subscribed to by a client process, the client will receive any messages published to that subject in the application. Subjects are defined by the application developer and can be thought of as a logical message address, providing a virtual connection between client processes. In SmartSockets, these can be specified hierarchically (for example, "/market/stock/NYSE") and can be published and subscribed to using wildcards.

In traditional network applications, when two programs must communicate with each other, they need very specific physical network addresses (for example, in TCP/IP, a node name and port number). If a program wants to send a message to many other programs, it first would need to know the physical network addresses, and then create and maintain a connection to all those programs. This architecture does not scale well, as configuration is complicated and tedious. The SmartSockets publish-subscribe communications model allows a program to simply send the message with a subject as the destination property; then, SmartSockets takes care of routing the message to all programs that are subscribed to that subject.

Once an application developer has written a client using the SmartSockets API, multiple instances of that process can be deployed on the same machine or different machines in the same LAN, WAN or anywhere on the Internet

The API is the same regardless of where the processes reside. Applications therefore can be developed on a single machine and then distributed over a network as needed, all without changing a single line of source code.

Peer-to-Peer Communication

As well as supporting the publish-subscribe model of message routing described earlier, SmartSockets also supports direct peer-to-peer communication between two programs. This is useful in applications where very high-speed communication is required between exactly two programs and no intermediate hops are warranted.

Multicast

When one system needs to broadcast data to many clients, it can choose one of two ways of doing this. The first is to send the same message to each client. This approach is effective, but it consumes a tremendous amount of bandwidth as the same data repeatedly crosses the network. A more efficient method is to use multicast, which is an Internet standard for broadcasts that allows for a more optimized distribution. With multicast, the message is sent to specific servers that then distribute the data to local clients. In this way, enterprise bandwidth is conserved. Historically, multicast has suffered from reliability issues—not every intended client received the message. Recently, however, the reliable multicast protocol (RMP) was designed and implemented to solve this problem. Today, Talarian is one of the first companies to implement multicast with RMP and have it support mission-critical delivery of packets while preserving the bandwidth savings.

Asynchronous Messaging

Rather than block a process waiting for a message to be delivered or for a reply to come back, SmartSockets allows messages to be delivered asynchronously without blocking the mainline logic of the calling program. Sending messages asynchronously allows a program (and the entire distributed application) to operate at higher performance levels because the program can continue its tasks without waiting for a reply to a message it has sent. Also, operating asynchronously allows the system to respond more quickly to external events.

With SmartSockets, it is simple to initiate concurrent operations involving multiple network platforms and to have them run to completion in parallel. SmartSockets' asynchronous functionality allows developers to leverage the inherent parallelism of the network environment.

Synchronous Messaging

Non-blocking RPCs are available within SmartSockets. These RPCs allow a client process to wait for a specified period of time for a specific message. Other messages that arrive while waiting for the RPC to return are buffered and processed once the RPC completes or the time-out expires.

Thread Safe

SmartSockets is completely thread safe, allowing multiple threads to operate simultaneously on a single connection. In multiprocessor environments with operating system kernel thread support, multiple threads can greatly increase performance.

Benefits: Increased scalability, increased productivity, reduced risk, and more productive use of resources.

High Availability and Reliability

In many mission-critical applications, fault tolerance and reliability are important requirements. The systems require continuous operation—24 hours a day, 7 days a week—regardless of hardware or software failures. SmartSockets achieves increased reliability in its communication layer by transparently checking for problems that occur when processes are connecting and sending or receiving messages.

SmartSockets IPC has been designed to handle many different kinds of network failures. In general, the SmartSockets IPC avoids operations that can block (or stall) indefinitely, or puts an upper limit on the amount of time these operations can block, and it periodically checks for potential failure conditions.

Publish-subscribe services are enabled through a SmartSockets program called RTserver. RTserver is a high-speed software message router with which programs can connect to send/receive messages to other programs in the application group. An RTserver may or may not reside on the same machine as the programs that connect to it.

SmartSockets allows programmers to achieve a higher level of availability in a distributed application through the use of software redundancy. Redundancy involves one or more backup (but active) processes for each primary process. For example, to ensure that a specified process continues to run regardless of problems that may occur in the network, one or more backup processes can be run. Backup processes can be receiving the same data as the primary process all along and be ready to take over instantly if the primary process fails.

Quality of Service

To meet the system and network outage recovery requirements, SmartSockets allows the sending program to optionally specify guaranteed message delivery on the messages it sends out. If an outage should occur to the sending program before a message is delivered, the message is automatically recovered and delivered to the receiving application when the system is brought back up. SmartSockets also handles the complex one-to-many case where a message must be guaranteed to multiple receivers.

By using guaranteed message delivery, distributed applications that require high availability can be integrated easily. This simple concept—which is exceedingly difficult to program—allows messages to be reliably delivered between programs in an application, eliminating the need for complex error-recovery code to be written into distributed applications and providing a much more timely mechanism than batch file transfer.

Dynamic Message Routing

SmartSockets handles changes and failures to the network through dynamic message routing. Routing tables in the RTservers are updated in real time as changes occur in the network (such as processes or machines going down). Messages are always sent through the RTserver virtual network using a shortest-path algorithm. This enables a distributed application to continue to function even in the face of component failures.

The popular message queuing products cannot handle changes to the network easily. In general, they are preconfigured to forward messages from a queue to another queue. If any component fails in this path, the message queuing software is stuck until the failed component is functioning correctly.

Multiple RTserver™ Processes

Because of SmartSockets' modular architecture, a distributed application can be built with any number of RTservers, with multiple protocols supported by each. A simple application may use only one RTserver to provide all services. A more complex application may use multiple RTservers to reduce network bandwidth and to add performance, load sharing, modularity, and reliability. Using multiple RTservers reduces network traffic and the number of direct links that must be maintained.

Benefits: Increased application availability, operation 24 hours a day, 7 days a week.

Distributed Application Monitoring, Administration, and Debugging

Debugging and monitoring a distributed application is extremely difficult because of the many variables involved (network, machines, programs) and the primitive tools that are offered by the operating systems. SmartSockets offers both graphical and programming tools to make the monitoring, administration, and debugging of distributed applications much easier. Comprehensive information can be gathered in real time about all facets of a running SmartSockets project, including message traffic, message buffering, memory usage, and CPU usage. It is possible to query each client for the subjects it has subscribed to, the options that have been set for it, the node it is running on, and the architecture of that node. Monitoring can be done interactively through the graphical user interface or programmatically through the programming interface.

Graphical Interface

RTmonitor is a graphical tool designed to monitor, debug, administer, and test an entire SmartSockets application. RTmonitor provides the capability to publish messages and to monitor critical IPC information in real time as it changes, all without having to make a single change to the application and without incurring performance overhead in the application when monitoring is not in use. The RTmonitor GUI is a multi-window development interface that provides snapshots and continuous real-time views of the activities occurring within a process IPC environment.

The ability to monitor a network application executing, with all its IPC activities visible, is invaluable when tracking down elusive problems. The total development time can be cut significantly through the use of RTmonitor alone. SmartSockets' debugging and monitoring capabilities are valuable support tools as well. Applications built with SmartSockets even can be monitored in production. No longer is it necessary to recreate problems or special debug versions of an application to study problems. With RTmonitor, there is no need to ask customers to submit large log files. SmartSockets applications can be studied wherever they are—as soon as they display irregular behavior.

Monitoring Interface

A subset of the SmartSockets robust monitoring API was used to build the RTmonitor graphical interface. From within their program, developers can use the monitoring API to track more than 300 different variables and events. This API provides mechanisms to track when processes start and when they fail, when they subscribe/unsubscribe to a subject, or when a specific message type is published or received. Actions can be taken by a program when a process joins an application or when it terminates.

When a program is linked with the SmartSockets libraries, it is automatically instrumented for monitoring. Calls to the API or clicking on choices in RTmonitor turn it on. By embedding monitoring within the SmartSockets API, developers take advantage of varied information such as CPU usage, memory usage and message queue sizes.

Monitoring can be done either synchronously (polling) or asynchronously (watching). When polling is used, a call is made to the API to collect the specified information and the function does not return until the operation is complete. When watching is used, a call is made to the API to register interest in specified variables or events. The function returns immediately. Whenever the specified event happens, a monitoring message is sent asynchronously to the program. Typically, the program will use callbacks to process the information. Both polling and watching can be used within the same program.

Southwest Airlines Flies with SmartSockets.

In late 2000, Southwest Airlines, the fourth-largest domestic airline in the U.S., deployed Talarian SmartSockets as the real-time infrastructure for its SWIFT (Southwest Integrated Flight-Tracking) system. SWIFT provides flight managers with a real-time status report on flight tracking and fuel usage, passenger loads, crew management data, gate information and weather reports for more than 2,600 daily flights. Says Phil Hyatt, technical project lead for Southwest Airlines, "We needed a system that could be built quickly and could integrate smoothly with SWIFT's publish-and-subscribe architecture in which information is delivered instantly to the correct parties as it changes. What really appealed to us was SmartSockets' complete and easily configurable API. Its simplicity and flexibility allowed us to implement the project in less than a month."

Message Logging

SmartSockets messages can be logged automatically to a file at any point along the way:

1. Messages arriving for a program
2. Messages leaving a program
3. Messages arriving at an RTserver
4. Messages leaving an RTserver

Messages can be logged in ASCII or binary format. ASCII log files can be edited or moved easily from one platform to another. Binary message logging is much faster and uses less disk space.

Message types can be assigned to various logging categories (classes). Logging for a specific class can be dynamically turned on and off as the programs are running.

Message logging allows the developer (or end user) to view the messages sent or received by any process. A developer also can use these message files to set up a controlled test environment in which to test the behavior of a process. This is done by having the process receive its input from the message file instead of from the other processes.

Benefits: Reduced maintenance costs and increased productivity.

Platform and Protocol Independence

Applications built with SmartSockets are guaranteed to be portable across different platforms. For example, an entire application or parts of an application can be moved easily from a PC running Windows NT to a Sun SPARCstation. Because SmartSockets offers a portable C API and C++ and Java classes, source code is immediately portable across different platforms. The IPC portion of a distributed application—often the most platform dependent—need be written only once.

Automatic Data Translation

SmartSockets allows messages to be sent between programs on different types of computers (for example, Intel PCs and Sun SPARC). When a message is sent through a connection, the integers, real numbers, strings, and other primitive data types within the message are converted automatically from the formats of the sending program to the formats of the receiving program; no action is required by the application developer.

No unnecessary data conversions are performed if the sending and receiving programs are running on computers with compatible data types (for example, Sun SPARC and HP PA-RISC).

With SmartSockets' cross-platform and multi-protocol support, developers can distribute processes anywhere over heterogeneous networks and share processing power. Programmers can modify network components and move resources and programs from one location to another with complete transparency. With SmartSockets IT managers can choose the best computing solution for a particular need.

Platform Support

SmartSockets is supported on numerous computing platforms, including the operating systems:

1. Unix (including Solaris, HP-UX, AIX, DEC Unix, Compaq True64, and IRIX)
2. Windows NT and Windows 2000
3. VxWorks
4. IBM OS/390
5. Linux

Client platforms include all the above as well as Windows 95/98/ME, Macintosh, and Java clients.

This list is updated frequently. Please contact your local SmartSockets sales representative for an up-to-date list.

Protocol Driver

One benefit of SmartSockets is that it shields the developer from having to understand underlying network protocols. Although SmartSockets supports most of the popular protocols, the SmartSockets Protocol Driver enables customers to add additional protocols (such as proprietary ones) easily.

Benefits: Protected investment in hardware and software.

Powerful Message Handling

Programs built with SmartSockets communicate via messages. Messages consist of a header and a data part, which holds the information being transferred and is dependent on the message type. The message header contains properties that describe the message, such as destination (subject), message type, priority, level of guarantee, read-only, and other information.

Automatic Message Buffering and Flow Control

Distributed applications typically require a method to establish data-flow between the client processes. A single program often will need to read messages from many different programs. In a SmartSockets application, each program has message queues that transparently buffer the messages when there are variable traffic rates, providing automatic flow control. This method of communication increases performance and greatly simplifies development.

Prioritized Message Queues

All messages in SmartSockets may have a priority attached to them. When a message is delivered to a client process, it is added to the client's message queue in priority order (higher priority messages come before lower priority messages). This forces higher priority messages to be processed before lower priority messages sent at an earlier time.

SmartSockets' message queues support message prioritization, searching, and selection. Messages can be retrieved based on FIFO, LIFO, and other types of priority specifications. Byte streams, RPCs, shared memory, and sockets are, by comparison, primarily FIFO mechanisms.

Reusable and Extensible Message Types

SmartSockets provides more than 150 standard message types that are used by SmartSockets internally and are available to the developer. When a standard message type does not satisfy a specific need, the developer can create a user-defined message type. (This requires only a single API call.) Both standard and user-defined message types are handled in the same manner. Once the message type is created, messages can be constructed, sent, received, processed, and logged through a variety of methods.

Load Balancing

SmartSockets-enabled programs and RTservers can subscribe and unsubscribe as needed by the given application requirements. Dynamic load balancing is designed into SmartSockets using selected algorithms including "least busy" and "round robin." Message delivery can be controlled via a simple round robin or by forwarding messages to the process that is least busy, where "least busy" is defined as the process that has the fewest number of messages unprocessed.

Benefits: Increased performance and reliability.

Scalability

SmartSockets was designed to scale to handle thousands of concurrent users (or processes). Scalability comes from a variety of different features, which are summarized here:

1. SmartSockets has a hierarchical namespace. This means subject names, which are used for a variety of purposes including application partitioning, can be specified hierarchically (for example, "/company/technology/software/talarian" or "/company/transport/auto/ford"), much like many of the popular file systems.

2. The SmartSockets namespace can be arranged hierarchically into any number of levels. (Other middleware products have no namespace, a flat namespace, or a limited number of levels). In large enterprise-wide applications, developers will need to use many levels of the hierarchy to properly partition the namespace.

3. SmartSockets does not require a routing process, or daemon, on every machine where a client process is running. (Other products require at least one routing process on every machine. Some products require multiple processes on each machine. This is not reasonable when deploying to many machines.)

4. Messages are routed dynamically in SmartSockets. Therefore, information sent from one application to another could potentially take a different path every time if parts of the network become unavailable. Message routing can be configured in two ways: manually or using default routing

5. Default routing is calculated using an OSPF (Open Shortest Path First) algorithm that corrects all the problems of the older RIP (Routing Information Protocol) algorithm, which is based on hop counts. Messaging middleware using RIP takes a long time to stabilize when a change occurs (for example, if a machine or process goes down) because hop counts must be propagated throughout. Default routing in SmartSockets uses the fewest number of RTservers possible. When

changes occur in the network, these changes do not need to be rippled all the way through SmartSockets.

6. SmartSockets-enabled processes can fail-over to new RTserver processes running on any machine should the one they are currently connected to fail for any reason.

7. SmartSockets-enabled programs can be set up to connect to any one of a random set of RTservers across a network. This prevents an RTserver from being overloaded with a large number of programs connecting to it.

8. There is no single place in a SmartSockets server cloud that has all the information needed to resolve routes. In this way, SmartSockets works much like the Internet and DNS. Information about routing and the locations of processes is kept in a "virtual distributed real-time database" in the SmartSockets network of processes.

9. SmartSockets' RTservers and programs do not require "root" or "system" privilege to install or run. Most other messaging products require special privileges to install or run. In general, this is not realistic when deploying to thousands of machines. In the case of MQSeries from IBM, changes must be made to the operating system kernel on most supported platforms.

10. SmartSockets messages can be published to a particular subject in the namespace or can be published to a subject with wildcards in it (for example, "/company-/transportation/*").

11. SmartSockets programs can subscribe to a specific subject, multiple subjects, or multiple subjects using a wildcard.

12. All SmartSockets RTserver processes do not need to be connected to one another. SmartSockets will dynamically reconfigure and update its routing tables as RTservers come and go.

13. SmartSockets can perform context bridging, translating messages from one protocol to another as message routing occurs. For example, a message could be sent down one connection via TCP/IP and then be delivered to the receiver via SNA. This facility includes additional protocols that might be added by the customer.

14. SmartSockets uses a "just-in-time, receiver makes right" approach in order to translate the message into the proper format. Even if a message goes through several hops, only the final receiver does the translation and the translation is not done until the receiver is ready to "use" the message. SmartSockets provides all data translation transparently: EBCDIC-ASCII, big endian–little endian, floating-point numbers, and the like.

15. SmartSockets includes graphical development tools that allow you to visualize your entire project and its topology. Further, the application can be monitored in real time to watch message traffic and quickly pinpoint any bottlenecks.

Benefits: Maximum use of existing computer resources and virtually unlimited expansion capability.

SmartSockets at Credit Suisse First Boston. In late 1999, Credit Suisse First Boston (CSFB), one of the world's largest securities-trading firms, adopted SmartSockets as the real-time infrastructure for its high-speed messaging system that links 4,000 equities traders around the globe. SmartSockets powers the Global Application Intercommunication Architecture at CSFB, enabling high-speed messaging among regional offices spread across 30 countries on six continents using more than 500 applications. Says Raymond Mulligan, vice president of Equity Technology at CSFB, "SmartSockets allows us to implement publish-subscribe messaging without having to build an elaborate broadcast infrastructure around it. With clients distributed around the globe, it would be impossible to install ancillary hardware for them all. With SmartSockets, that problem was eliminated."

Benefits of SmartSockets

Building distributed applications can be a very complex and risky venture. For such difficult problems, SmartSockets offers maximum power and flexibility together with superior ease of use. The SmartSockets programming model has several important benefits that are outlined in the sections that follow.

Increased Productivity

The easy-to-understand multiprocess programming model of SmartSockets eliminates the need for network programming skills. A single call to a SmartSockets function is the equivalent of many lines of complex network programming. As a result, the cost of developing a distributed application is significantly reduced.

Reduced Risk

SmartSockets has been used and tested as part of mission-critical systems for years across a variety of industries including financial, telecommunications, process and manufacturing, and aerospace. This track record means proven reliability and reduced risk.

Increased Scalability

As an application grows in terms of the number of processes involved, the amount or rate of message traffic, and the number of machines involved, SmartSockets can grow with it. Because both the client and server processes can be distributed across a network within the SmartSockets programming model, the application can make maximum use of existing computer resources while offering virtually unlimited expansion capability. Applications can be built on a single machine and then deployed over a network without changing even a single line of source code.

Protected Investment in Hardware and Software

SmartSockets allows IT shops to keep the investment in software that they have already developed. By adding calls to the SmartSockets API within already existing software, developers can quickly and easily integrate legacy programs into the distributed application.

Because of SmartSockets' multiplatform support, IT managers will be able to make use of their existing hardware. In addition, if you wish to change hardware vendors, your SmartSockets application will be usable on the new platform. SmartSockets does not require kernel modifications or any other changes to be made to the computer.

Reduced Maintenance Cost

SmartSockets reduces software maintenance costs by virtually eliminating the need for network programming skills. SmartSockets allows a software engineer to write distributed applications without going through the tedious and expensive process of writing low level network software. Furthermore, as updates to the operating system occur, SmartSockets is also updated to mask those operating system changes from the user. Finally, moving all or part of an application to another platform can be done transparently with SmartSockets.

Increased Application Availability

SmartSockets offers several capabilities that greatly increase the availability of a distributed application. SmartSockets offers easily configurable options for responding to network failures, hot failover to parallel processes when a process or machine goes down, guaranteed

message delivery, error callbacks when an exception occurs, and sophisticated real-time monitoring tools to keep an eye on what is happening. All these sophisticated features allow developers to build applications that are robust and reliable.

More Productive Use of Resources

SmartSockets means computers spend less time waiting and more time working. It also gives IT managers the flexibility to match the technology to the business need. For example, a shop may have specific message types that cause control action to be taken by the SmartSockets client processes. Messages associated with other less critical needs can be sent and added to the queue, to be worked on after the higher priority messages have been processed.

Getting Started with SmartSockets

You can get started with SmartSockets by simply using the API (a C library, C++ class library, or Java JAR file) in the parts of your C/C++ or Java programs that need to communicate with other programs in your network. You can also replace the communication sections of your existing programs for a more rapid implementation and faster return on investment. Or, like many SmartSockets customers, you may decide the benefits are too good to be postponed and redesign your existing application to incorporate this exciting new technology.

Example C Programs

To demonstrate how easy SmartSockets is to use, this section contains two heavily commented C programs. The first program simply publishes a message to a subject (it requires only two lines of actual code). The second program subscribes to the subjects, reads the message, and prints out its contents (it is only five lines of code).
Note that all SmartSockets API calls start with "T"
(for Talarian).

```
/* Program 1: send.c */

#include <rtworks/ipc.h>

int main( int argc, char **argv)
{
   /* This one function call will perform the following tasks:

1. Since there was no explicit call to connect to RTserver, the
   process will first establish a connection to RTserver (this
   might including auto-starting RTserver if it is not already
   running).

2. A message of type INFO will be created.

3. The data part of the message will contain the text string
"Hello World".

4. The message will be sent to the RTserver process to be
delivered to the subject "/ss/tutorial/lesson 1"; any process
that has subscribed to this subject will receive the message.

5. The message will then be destroyed. */

TipcSrvMsgWrite("/ss/tutorial/lesson1",
    TipcMtLookupByNum(T_MT_INFO), TRUE, T_IPC_FT_STR,
    "Hello World!", NULL);

   TipcSrvFlush(); /* ensures message is sent immediately */
}
```

```
 /* Program 2: receive.c */

#include <rtworks/ipc.h>
int main(argc, argv)
int argc;  char **argv;
{

   T_IPC_MSG msg;  T_STR MsgText;

/* The following function call performs two tasks:

   1. Since there was no explicit call to connect to  RTserver,
      the
   process will first establish a connection to RTserver (this
   might include auto-starting RTserver if it is not already
   running).
```

2. Inform RTserver to forward any message which has been sent to the subject "ss/tutorial/lesson 1" by any client in the application. */

```
  TipcSrvSubjectSetSubscribe("ss/tutorial/lesson 1", TRUE);

/* This function calls gets the next message from its message
queue; The T_TIMEOUT_FOREVER constant specifies that the function
will block (wait) forever for a message. You can specify an
actual time-out period for this argument; for example, 10.0 would
mean the function would return with a failure code after 10
seconds.  */

    msg = TipcSrvMsgNext(T_TIMEOUT_FOREVER);

/* We now have a message to work with.
   The following call sets the message
   pointer (i.e., the current field) to
   the first field in the data part
   of the message.  */

    TipcMsgSetCurrent(msg, 0);

/* The next call gets the string from
   the data part of the message.*/

    TipcMsgNextStr(msg, &MsgText);

/* Finally, print out information retrieved from the message. */

    TutOut("Message Text = %s\n", MsgText);

}
```

TALARIAN CORPORATE HEADQUARTERS
333 Distel Circle
Los Altos, CA 94022-1404
(800) 883-8050
FAX (800) 883-8057

TALARIAN LIMITED
68 Lombard Street
London EC3V 9LJ
+44 (0) 20 7868 1630
FAX +44 (0) 20 7868 1752

E-Mail info@talarian.com
www.talarian.com

Talarian™
The world works in real-time™

WPMW 1001 1000