Interconnection of Distributed Components: An Overview of Current Middleware Solutions

Susan D. Urban, Suzanne W. Dietrich, Akash Saxena, and Amy Sundermier
Arizona State University
Department of Computer Science and Engineering
Tempe, AZ  85287-5406
s.urban@asu.edu
dietrich@asu.edu

ABSTRACT
From design and manufacturing to electronic commerce, coordinating business activities in engineering applications requires accessing data and software from distributed sources. The Common Object Request Broker Architecture (CORBA) of the Object Management Group emerged in the 1990’s as a standard for access to distributed software components. Since that time, the standard has matured significantly, providing advanced features for event notification and transaction processing. At the same time, Java-based technology for distributed object computing has also emerged, from Remote Method Invocation to Enterprise JavaBeans, Jini Connection Technology, JavaSpaces, Java Messaging Service, and Java Transaction Service. Sorting through the options available for the use of such tools can be a difficult task. This paper provides an overview of CORBA and Java technology for distributed object computing. A comparison of these different technologies is presented, discussing the similarities and differences, as well as the way in which such tools can be used together for distributed access to the types of software and data components that are needed for the construction of distributed engineering applications. Future directions for the use of such tools are also identified.

1. Introduction
Engineering applications often demonstrate the need for distributed computing solutions. Engineering design and analysis, for example, typically involve many different people, possibly in remote locations, all working together on different parts of the problem and using many different software tools and databases in the process. The output from one tool must often serve as the input to a different tool, creating a workflow of activities between distributed sites. Different databases that contain related data, such as an inventory database and a customer billing database, may need to be updated as part of one coordinated activity. Furthermore, for any given activity there can be a choice of several different tools to use. Manufacturing and planning activities often require communication with devices on the factory floor in addition to just software and database tools. Finally, electronic commerce complicates the picture even further. Businesses must now extend their distributed computing activities outside of the realm of their own companies, creating business-to-business (B2B) solutions where the software and data components of one company must communicate with the software and data components of another company [1].

* This research was supported by NSF Grant No. IIS-9978217.
The development of distributed computing solutions for the above scenarios is a complex task. Often, the software and data components that need to be interconnected were not originally designed for communication within a distributed environment, thus providing no inherent interface for communicating with the outside world. Even if such interfaces exist, low-level network programming for interconnection of distributed sites can be a challenge that application programming experts may not be prepared to address. Interoperability standards and the availability of tools that ease distributed communication is an important aspect in the development of the types of distributed computing solutions that are currently needed in the engineering domain.

Fortunately, there have been significant developments over the last ten years in the creation of middleware solutions that provide standardization for communication with distributed software components as well as layers of abstractions that provide more effective ways of programming in distributed environments. The Common Object Request Broker Architecture (CORBA) [2] of the Object Management Group (OMG) [3] emerged in the 1990’s as a standard for access to distributed software components. Initially, CORBA simply provided a means for developing standard interfaces for software tools so that the services of such tools could be invoked from distributed sites. The strength of CORBA is that it not only provides location transparency but that it is also language independent, allowing the interconnection of software tools that are written in different programming languages. Since the early 1990’s, the CORBA standard has matured significantly, providing advanced features such as event notification and transaction processing, as well as features for communication between different commercial products based on the CORBA standard.

In parallel with the maturity of the CORBA standard, several different solutions for distributed object computing have also emerged based on the use of the Java programming language [4]. These solutions, in some cases, represent alternatives to the use of CORBA and, in other cases, represent solutions that can be coordinated with the use of CORBA. The Java Messaging Service [5] and Java Transaction Service [6], for example, provide functionality that is similar to the event and transaction processing services of CORBA. These services can be especially useful in a purely Java development environment. The Enterprise JavaBeans (EJBs) [7, 8] specification from Sun Microsystems provides a sophisticated software component model. The use of the EJB software component model provides a cleaner separation between business logic and low-level distributed computing issues than that originally provided in the CORBA specification [9]. The OMG has responded with its own component model specification [10, 11], but the EJB model can also be used in coordination with CORBA.
Jini Connection Technology [12] provides a completely different Java-based approach to distributed computing than the CORBA model. Unlike CORBA, Jini does not promote transparency as an advantage of distributed computing. Instead, Jini provides a service that allows distributed components to find each other in a distributed space so that they can then directly communicate. Jini supports event and transaction processing capabilities that are similar to those of CORBA and the Java Messaging and Transaction Services. JavaSpaces is also an important component of the Jini environment, creating a persistent storage space where distributed components can share data [13]. The primary purpose of Jini is to create a flexible distributed computing environment where software and hardware components can enter and leave the environment without human intervention and, at the same time, dynamically find needed services within the environment.

Sorting through the options available for the use of such tools can be a difficult task. This paper provides an overview of the above-mentioned technologies for distributed object computing. Section 2 provides an overview of the Common Object Request Broker Architecture. The Enterprise JavaBeans component model is then presented in Section 3. Section 4 provides an overview of the Java Messaging and Transaction Services, followed by an overview of Jini Connection Technology in Section 5. A comparison of these different technologies is presented in Section 6, discussing the similarities and differences as well as the way in which such tools can be used together for application development. The paper concludes in Section 7 with a discussion of future directions for the use of such tools.

2. The Common Object Request Broker Architecture

As described in the introduction, CORBA is a distributed object computing architecture developed by the OMG for the purpose of defining a standard that allows objects to interoperate across networks, regardless of the language in which they are written or the platform on which they are deployed. The OMG, originally formed in 1989, is a consortium composed of close to 800 member companies that have provided input into the development of the standard. The OMG is an open organization, meaning that any company is welcome to join and participate in a consensus-based approach to the development of the standard. The OMG does not produce software but, instead, provides specifications for CORBA and its associated Object Management Architecture [2]. Some of the most well known commercial products based on the CORBA standard are Iona’s OrbixWeb 3.0 [14], Visigenic’s VisiBroker 3.0 [15], and JavaSoft’s Java IDL [16].
2.1 CORBA Basics

CORBA provides a client/server model of communication between software components, where clients request services from remote servers using a well-defined interface that is specified using the Interface Definition Language (IDL) of CORBA. The backbone of CORBA is the Object Request Broker (ORB) that manages the task of locating objects and passing requests, events, and data to and from remote objects. Objects never interact with each other directly. Instead, they go through the ORB, which is an abstraction layer sitting between the objects.

Figure 1 provides a diagram of the ORB architecture as described in [17]. To support communication between the client and the server, the server object provides an IDL file, establishing the interface that a client must use to invoke the services of the server object. The interface defines the attributes of the server object as well as the operation signatures for all object services (i.e., the names of the methods to be called together with the names and types of all parameters and return values). In addition to the development of an IDL file, an object implementation must be developed on the server side to implement the operations as specified in the IDL interface. Object implementations can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.

An IDL compiler compiles the IDL interface of a server object into stubs and skeletons. A stub is used on the client side, encapsulating a remote operation call to make it look like a local operation call. A client can then transparently invoke the implementation of an object service by calling the local stub. This remote operation call is transferred through the ORB to the corresponding skeleton on the server side. The skeleton unwraps the parameters in the remote function call and initiates the call to the actual operation implementation. The skeleton then takes care of transferring output parameters and return values back to the client through the ORB.

The advantage of the CORBA approach to communication is that the client is decoupled from the details of remote method invocations. Remote operations appear as local procedure calls. The ORB is responsible for finding the object implementation using an object reference, transparently activating the object if necessary, delivering the request to the object, and returning any response to the caller. To support the decoupling of applications from implementation details, the CORBA specification defines an abstract ORB Interface as shown in Figure 1. The ORB interface defines a set of standard function calls for accessing the services of an ORB, such as converting object references to strings, creating argument lists for requests, and retrieving a reference to an object for accessing interface and implementation repositories. The ORB interface may be
called by either the client or the object implementation and is mapped to the host programming language. The ORB interface must be supported by any implementation of the ORB specification.

Another major component of the ORB architecture shown in Figure 1 is the Object Adapter (OA). The OA assists the ORB with activating a server object and delivering requests to an object. OAs are the primary ORB service providers for object implementations. The lower level of the ORB architecture is the ORB core, which is the underlying mechanism for providing basic communication between the other components of the ORB architecture. CORBA 2.0 extended ORB communication with the General Inter-ORB Protocol (GIOP) for ORB interoperability, thus allowing ORB-to-ORB interaction. The GIOP can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions as laid out in the CORBA specifications. A specific mapping of the GIOP that runs directly over TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

Advanced features of CORBA include the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). An application programmer can use the DII to issue requests to objects without requiring IDL interface-specific stubs to be linked into the application code. The DSI is the server side’s analogue to
the client side’s DII, allowing an ORB to deliver requests to an object implementation that does not have compile-time knowledge about the type of object that is being implemented.

2.2 The Object Management Architecture
At the next level of abstraction above CORBA, the OMG has specified the Object Management Architecture (OMA) [18, 19]. The primary purpose of the OMA is to support enterprise integration. As shown in Figure 2, the OMA is composed of three main components in addition to the ORB as described in the previous subsection: Application Objects, CORBA facilities, and CORBA services. Application objects will not be standardized by OMG but are left open for vendor competition in the delivery of application-specific, CORBA-based products for customers. CORBA facilities define application-specific frameworks for enterprise development [17, 20]. Horizontal facilities represent application services that can be used in any type of application. Vertical CORBA facilities create standards for specific types of applications such as manufacturing.

In this paper, we are primarily interested in CORBA services. CORBA services extend the functionality of the ORB by defining system-level services that are needed for application development [21]. These services provide useful middleware capabilities that can be leveraged in the enterprise. Examples of these services include life cycle services (for creating, copying, moving and deleting components across the distributed environment), naming services (for allowing components in a distributed environment to locate other components by name), query services (for querying objects in the environment), event services (for providing messaging capabilities between components), transaction services (for providing transactional capabilities to the participants of a distributed transaction), and concurrency control services (for providing a distributed lock manager that works in conjunction with the transaction manager) [20]. Many of these services have not yet been fully implemented by commercial CORBA tools. Services that have been implemented and are particularly useful to the development of distributed engineering applications are the event and transaction services.
**CORBA Event Service**

The software and data components that participate in a distributed application often have the need to inform other components of the occurrence of specific events. For example, an automatic inventory restocking system may need to know when the quantity of a critical inventory item falls below a certain level. The CORBA event service provides an infrastructure for message disbursement by viewing objects in the environment as *suppliers* and *consumers* of information. Consumers register to be notified about the occurrence of events so that they can perform a specific action in response to events. Suppliers, on the other hand, generate notifications of events.
Figure 3 illustrates the CORBA event model, where suppliers and consumers communicate through the use of an event channel. The event channel is an object that provides a level of decoupling between suppliers and consumers, thus avoiding point-to-point communication between suppliers and consumers and allowing multiple consumers to register for the same event notification. Suppliers in this model are unaware of who receives event notifications. Alone, the event channel is the consumer to the suppliers, and the supplier to the consumers. There is no limit on the number of suppliers. An object may also be a consumer and a supplier to multiple event channels.

The event service specification defines two models for communication, the Push model and the Pull model. The primary difference between the two is the way data is distributed on the channel from suppliers to consumers. In the Push paradigm, a supplier places a notification of an event on the event channel. Consumers that have registered to receive this event under the push model are then notified by the event channel about the occurrence of the event. Under the Pull paradigm, the consumer initiates the communication by periodically polling the event channel for the occurrence of events. The use of the push or pull model depends on the needs of the specific application. Both models can be used together within the same application, although push consumers can only be used with push suppliers and pull consumers can only be used with pull suppliers.
The CORBA Transaction Service

Coordinating activities between multiple, distributed components often requires that the activities must take place in the context of a transaction. Using the traditional database definition, a transaction is a logical unit of work that typically involves the execution of several different operations, possibly requiring multiple changes to a database [22]. A desirable property for most transactions is that the transaction must complete as though it were a single, atomic operation. In other words, all operations within the scope of the transaction must successfully complete. This completion process is known as the commit of a transaction, where all database updates are written to disk. If one operation within the transaction fails, then any changes made by the transaction must be reversed (also known as the rollback process) to leave the database in a consistent state. In a bank teller application, for example, if you are performing a transfer of funds from savings to checking, you would not want the transaction to extract money from your savings account without completing the deposit in your checking account.

A transaction in a distributed environment can be especially complex, involving updates to data in multiple locations. To ensure that all sub-transactions of a distributed transaction complete successfully, a protocol known as two-phase commit is used [22]. Under the two-phase commit process, one site is selected as the coordinator for the process, while all other sites involved in the transaction are participants. The coordinator and the participants then follow a well-established procedure for communicating about whether all sub-transactions commit or all sub-transactions abort.

The Object Transaction Service (OTS) as specified in the CORBA specification defines a set of interfaces that support the two-phase commit process across multiple CORBA objects [2]. The OTS is based on the X/Open Distributed Transaction Processing (DTP) [23, 24, 25] model, which is a standard among most commercial vendors in transaction processing and database domains. Using OTS in CORBA, operations can be invoked on distributed objects as part of a transaction. At the end of the transaction, the two-phase commit process is initiated to determine if all sites involved in the transaction can safely commit their operations. If a rollback is required, then all sites must rollback any changes that were made. The CORBA OTS also supports the capability of passing the transaction context, which is the internal state of all variables that have been modified as part of the transaction as well as system-level information about the transaction. The transaction capabilities of CORBA are a necessary feature for any distributed engineering application that requires updates to data at several locations as part of one logical update activity.
2.3 Recent Developments with the CORBA Standard

The CORBA standard has continued to evolve over the last ten years, with the most recent version of all CORBA specifications available at [26]. The initial release of CORBA 1.1 only addressed the specification of IDL together with various language bindings and interfaces for communicating with the ORB [20]. CORBA 2.0 was released in 1995 to address the need for communication between different vendor ORBs. The forthcoming CORBA 3.0 focuses on Java and Internet integration, messaging and quality of service control, and the development of a component architecture [10, 11]. The Java and Internet integration supports the definition of IDL interfaces for Java objects, thus allowing Java programmers to use the IIOP protocol for remote method invocations. Java servers can also be invoked by CORBA clients using any CORBA-supported programming language. The Internet integration also provides a firewall specification that allows developers to configure firewalls for use with CORBA traffic. An Interoperable Naming Service also defines a format for object references. The messaging and quality of service control feature supports asynchronous messaging and allows clients and objects to control different aspects of method invocation, such as start and end times for time-sensitive invocations.

The component architecture of CORBA 3.0 provides an even higher level of abstraction than CORBAservices, thus allowing application developers to more cleanly separate the specification of business logic from low-level programming details involving transaction or security issues. The component architecture of CORBA 3.0 is similar to the Enterprise JavaBeans (EJB) specification from Sun Microsystems [7, 8]. Since the details of EJB’s have been more fully developed than the CORBA 3.0 component model, the next section provides an overview of the EJB component model. In Section 6, we will address the relationship between CORBA and EJB components.

3. The Enterprise JavaBeans Component Model

The Enterprise JavaBeans (EJB) specification from Sun Microsystems provides a server-side software component model for the Java programming language [7, 8]. The EJB component model promotes the vision of separating component services and infrastructure from the business logic of a software component. As outlined in the EJB specification, application server vendors implement environments for deploying application programming interfaces (APIs) to software products, where the API is developed under the EJB component model. Application server vendors employ distributed computing specialists that focus on providing robust infrastructure and services for software components, such as transaction management and security. Developers of EJB components, on the other hand, employ business logic specialists that focus on developing components that implement enterprise business processes. Purchasers of EJB software components can place their EJB components into any application server to
create core business systems. Application developers can then concentrate on building only those custom business components that reflect their unique competitive advantage, while using purchased components for other parts of the application. Separating the responsibilities of the server provider, the component developer, and the application developer helps to simplify the development process for distributed applications.

The EJB component model facilitates an efficient division of responsibilities between different types of programmers by specifying three different entities that participate in the implementation of EJB software components: Objects, Containers, and Servers. These entities are illustrated in Figure 4 and described in more detail below.

![Figure 4: The EJB Component Model](image)

### 3.1 Enterprise JavaBean Objects

EJB objects are developed by application software vendors. For example, the vendor of a product data management system may want to develop an API for the system to make it easier for a customer to integrate the use of the product data management tool into a distributed application environment. The software vendor would then develop an EJB object that implements the API of the system. Developers of distributed applications can purchase the EJB object and use it to integrate the product data management system into a more global application.
In version 1.1 of the EJB Specification [7], there can be two types of EJB objects, namely *Entity Beans* or *Session Beans*. An Entity Bean represents a database or persistent business object in the distributed environment, while a Session Bean provides transient, per client, session-based processing. An application may require the use of both Entity and Session Beans since they represent different aspects of the application. Entity Beans represent the data, which is inherently shared between all users. Session Beans represent the processing of a single task, which is executed on behalf of a single user. A Session Bean may access numerous Entity Beans or other Session Beans during the course of its execution.

Entity Beans are typically fine-grained objects with a persistent representation, such as the individual rows in a relational database table or view. It is also possible for an Entity Bean to represent data from other types of data sources, such as legacy systems. Each Entity Bean has a unique identity that persists across client sessions and an object state that resides in one or more persistent data sources. To help clients locate an Entity Bean with a particular identity, an Entity Bean must provide a primary key and at least one `find` method that returns a reference to a specific object instance based on a primary key lookup. An Entity Bean's persistent state needs to be stored in and retrieved from its data storage. To accomplish this, an Entity Bean must either implement its own data access code or delegate this function to a Container. Containers represent the execution environment of EJB objects and are described in more detail below.

In contrast to Entity Beans, Session Beans represent a process or task that is performed on behalf of the client application. Session Beans may use other EJBs to perform a subtask or to retrieve and store data. A Session Bean indicates whether it is *stateful* or *stateless*. A stateful Session Bean can retain state changes after method invocations. Therefore, the Container must make it appear that the client has connected with the same Session Bean for each method call within the same session so that state changes may be retained between method invocations. A stateless Session Bean does not retain the client’s state changes between method invocations. Therefore, a client may connect to a different stateless Session Bean for each method invocation. Session Beans are typically pooled by the EJB application server, and are allocated to a single client upon request. Session Beans are not shared concurrently between clients, but an instance of a Session Bean will be returned to the resource pool for reallocation when a client has finished with it.

Figure 5 illustrates a conceptual view of client interaction with an EJB component. Each EJB component provides two types of interfaces: the *EJBHome* interface and the *EJBObject* interface. The EJBHome interface represents the life-cycle methods of the component. A client calls the operations in the
EJBHome interface to create or delete instances of objects or to use the find operation to lookup a specific object instance in an Entity Bean. The EJBOBJECT interface defines the signature of the business methods for changing attribute values and carrying out business logic functions that are specific to the EJB component. The EJBOBJECT interface is therefore similar to the functions that are typically provided in a CORBA IDL interface. The EJBHome and EJBOBJECT interfaces define a standard set of methods for clients to use when interacting with an EJB component.

![Diagram of Client Interaction with EJB Components](image)

**Figure 5: Conceptual View of Client Interaction With EJB Components**

### 3.2 Enterprise JavaBeans Container and Server

EJB containers are purchased from an application server vendor. An application server vendor is a company that specializes in the development of software for handling the system-oriented aspects of distributed computing environments. The current EJB architecture assumes that the EJB server provider and the EJB container provider are the same. Therefore, the EJB specification does not define any interface requirements between the EJB container and EJB server, allowing each vendor to implement this interface conveniently for their proprietary system.
An EJB container runs within a server and hosts one or more different types of EJB objects, as shown in Figure 4. Depending on the configuration and resources allocated by the server, a container may host multiple instances of an EJB object. Containers are responsible for providing services to the EJB components, such as remote object access, security, transactions, and persistence. When an EJB is deployed into a container, a deployment descriptor is defined for the EJB that declaratively describes the types of services required by the component. For example, the deployment descriptor contains declarations about access control for a component’s methods. When a client attempts to call the method of a component, the container is responsible for looking up the user's permission in an access control list to verify that the user has the right to invoke that method. Deployment descriptors have a standard format so that EJB components may expect the same services even when the components are deployed in different EJB containers.

An EJB component may choose to have container-managed or bean-managed transactions. If container-managed transactions are selected, then the container takes all responsibility for imposing transaction properties on the component so that, if appropriate, a component can participate in a transaction. When using bean-managed transactions, an application developer must use a third party transaction provider such as the combination of the Java Transaction API (JTA) and the Java Transaction Service (JTS) (see Section 4 below). The developer must write code for the component to manually define transaction boundaries and take care of transaction management tasks.

An EJB may also choose either bean-managed or container-managed persistence. In the case of bean-managed persistence, the bean implements its own data access code. In the case of container-managed persistence, the bean delegates data access to the container. For EJBs under container-managed persistence, the container is responsible for implementing the appropriate data access functions.

3.3 Recent Developments with the Enterprise JavaBeans Specification

Version 2.0 of the EJB specification was released in the summer of 2000 [8]. The new features of the EJB 2.0 specification include enhanced support for container-managed persistence and the introduction of a MessageDrivenBean. The enhancements for persistence provide greater support for storing relationships between Entity Beans within the same container. An EJB query language is part of the enhancement, which will make Entity Beans easier to deploy. The MessageDrivenBean is a new enterprise bean type that integrates with the Java Messaging Service, thus allowing Entity and Session Beans to participate in sending and receiving messages for a more event-based approach to software interconnection. An overview of the Java Messaging Service is presented in the following section.
4. Java Messaging and Transaction Services

The Java programming language has gained significant popularity over the last few years due to the portability of Java code [4]. To support the use of Java in distributed applications, Sun Microsystems has defined additional services for use with the Java programming language that provide alternatives to the use of CORBA for distributed applications that are based purely on the use of the Java. The services that are of particular interest to the engineering community are the Java Messaging Service (JMS) [5] and the Java Transaction Service (JTS) together with the Java Transaction API (JTA) [6].

4.1 The Java Messaging Service (JMS)

JMS is an API for accessing enterprise messaging systems. JMS makes it easy to write business applications that asynchronously send and receive critical business data and events. JMS is gaining in popularity, with more vendors providing implementations of the JMS specification from Sun Microsystems. Many vendors are also joining the list to endorse messaging products that adhere to the JMS specification. The recent EJB 2.0 specification has also adopted JMS as an event service for EJB components [8].

The JMS API is divided into two different specifications: one implements the publish/subscribe model of messaging, and the other implements a point-to-point model of messaging. In the publish/subscribe model, an entity known as a publisher publishes information about the occurrence of events. Subscribers are entities that are interested in obtaining this information from a publisher. The publisher initially creates a topic object that serves as the conduit for messages that are published by the publisher. The publisher and the subscriber register themselves with a topic object that keeps track of the subscribers connected to the topic object as well the publishers to the topic object. Messages are passed to the subscribers whenever the publisher sends out a message on the topic object. Subscribers may optionally specify a query string over their connection to the topic object to retrieve only those messages that meet the criteria specified in the query string.

The point-to-point model is a special case of the publish/subscribe model. Under the JMS messaging model, subscribers to a specific event have a distinct queue object. A publisher that wants to send a message will send the message to the queue object. The queue object will in turn take care of sending the message onwards to the client at the other end. The point-to-point model provides synchronous delivery of messages and is useful when there may be fewer subscribers connecting to the message delivery source.
4.2 Transaction Processing with Java

The Java Transaction Service (JTS) specifies the implementation of a Java transaction manager. This transaction manager supports the Java Transaction API (JTA) that can be used by application servers to build transactional Java applications. Internally, the JTS implements the Java mapping of the OMG OTS. As a result, JTS is fully compatible with CORBA products.

The JTA specifies an architecture for building transactional application servers and defines a set of interfaces for the various components of this architecture. Java application components conduct transactional operations on JTA compliant resources via the JTS, where the JTS acts as a layer over the OTS. Applications can therefore initiate global transactions to include other OTS transaction managers, or participate in global transactions initiated by other OTS compliant transaction managers. Currently JTA has been incorporated into the EJB specification since most implementations using the EJB component model need to make use of JTA compliant transaction processing capabilities. JTA is used to demarcate transaction boundaries programatically by using its scaled down interface to the underlying JTS implementation. Standardizing the transaction service interfaces allows components using JTS or OMG compliant transaction monitors to interact and provide transactional capabilities across the enterprise.

5. Jini Connection Technology

Jini Connection Technology [12, 27] from Sun Microsystems was first announced in 1998 with the vision of providing simple mechanisms that enable software and hardware components to plug together to form an impromptu community -- a community put together without any planning, installation, or human intervention. Independent of operating systems, connectors, or special hardware, components in the Jini environment can simply be connected and start interoperating with other devices on a network.

Jini Connection Technology should be of particular interest to the engineering community. For example, Jini provides the potential for more flexible manufacturing environments, where shop floor configurations can be dynamically altered as different machines exit the environment for repairs or when plans are re-configured to enhance productivity. Furthermore, even though Jini was originally designed for interconnecting devices, Jini is also useful for interconnecting arbitrary software components. In an engineering design environment, for example, Jini could be used to create an environment where different modeling and analysis tools join the environment and advertise their services. Jini provides helpful services such as advanced naming services, remote events, a transaction service, and a unique new service called JavaSpaces [13]. JavaSpaces is based on the tuple spaces research of Gelernter [28] to provide a persistent storage unit for data sharing between the distributed components of the environment.
5.1 The Jini Model of Distributed Computing

In a Jini community, each software and hardware component provides services that other components in the community may use. Jini essentially provides a lookup service that assists components in finding the services of interest. When a component wishes to join the distributed community, the component goes through a process called discovery and join. In this process, the component first locates the distributed community (the discovery process), where it then uploads the interface of the services it can provide (the join process).

The Jini lookup service is a fundamental part of the federation infrastructure for the group of devices, software services, resources, and users that are interconnected by the Jini software infrastructure. The lookup service provides a central registry of services available within the distributed community. Clients can use the lookup service to find an instance of a particular service by providing a template that is used to match against services already registered with the naming service. A match can be made based on the specific data types for the Java programming language implemented by the service as well as the specific attributes attached to the service. Although the collection of service items is stored as a flat structure, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes. The lookup service provides a set of methods to enable incremental exploration of the collection. Once a service has been located by Jini and its reference handle passed back to the requestor, Jini takes itself out of the picture so that all interaction occurs with the service adapter and the client that needs to use the service in question.

Other services provided by Jini are leasing, event notification, and transaction services. The Jini leasing service provide a means for limiting the amount of time that an object can advertise a service. An object can always renew a lease. Setting limits on service advertisement prevents the problem of stale services, where components exit the distributed environment without removing the description of their services. When a lease expires, the description of the service will be removed from the system. Events can be used in the Jini environment to notify others about newly-available services as they come on-line. Events can also be used to send notifications about state changes in objects of interest. The event notification service supports asynchronous notification between objects.

Jini supports the concept of transactions with support for the two-phase commit protocol. Jini, however, only provides the coordination mechanism that can be used by objects to communicate the information necessary for the set of objects to agree on transaction completion. The goal of Jini transactions is to
provide the *minimal* set of protocols and interfaces that allow objects to implement transaction semantics rather than the *maximal* set of interfaces, protocols, and policies that ensure the correctness of any possible transaction semantics.

### 5.2 JavaSpaces

JavaSpaces is a powerful Jini service that facilitates the sharing of information in distributed applications [13]. The Javaspaces model involves persistent object exchange "areas" in which remote processes can coordinate their actions and exchange data. JavaSpaces provides a necessary ubiquitous, cross-platform framework for distributed computing. A JavaSpace can be thought of as a “logical” space that holds various Java objects from participant distributed components that share information by writing objects to the space and reading objects of interest that some other components may have put into the space.

![JavaSpaces Diagram](image)

**Figure 6: The JavaSpaces Approach to Information Sharing**

As shown in Figure 6, a JavaSpace holds *entries*, which are a typed group of Java objects that are written to a JavaSpace service [13]. Entries in a JavaSpace service can be looked up using *templates*, which are entry objects that have some or all of its fields set to specified values that must be matched exactly.
Remaining fields are left as wildcards. An entry can be retrieved from the JavaSpace by either a read or take operation. A read operation returns the entry object if a match was found for the matching template that was used by the read. If no such entry exists, then the operation returns with an indication that no entries were found. A take operation is similar to a read, but if a match is found, the entry is removed from the space. Another facility that JavaSpaces provides is that of notification. An object can request a notification if an entry matching a template that it provides is written to the JavaSpace.

As with other Jini services, an entry cannot exist in a JavaSpace service perpetually. When an entry is written to the space, it must obtain a lease that dictates how long the entry may remain valid in the space. The lease expires in a finite amount of time and objects can renew leases.

All operations that modify a JavaSpace service are performed in a transactionally-secure manner with respect to that space. The Jini transaction services are used to provide a two-phase commit protocol that can be used by objects that wish to perform operations within a transaction scope and span operations that allow multi-operation and/or multi-space updates to complete atomically.

The JavaSpace service addresses the issues of distributed persistence and structures distributed algorithms as a flow of objects. Distributed persistence deals with storage of groups of related objects that can be retrieved based on templates/matching. Designing distributed algorithms as a flow of objects is a departure from traditional method-invocation-style protocols between participants. JavaSpaces encourages these forms of distributed algorithms by basing its protocols on movement of objects in and out of JavaSpaces.

6. Putting It All Into Perspective

The previous sections outlined several different technologies that provide viable alternatives for the development of distributed computing solutions for engineering applications. Many of these technologies provide similar features with slightly different capabilities. Tables 1 and 2, for example, highlight the similarities and differences between the event and transaction processing capabilities of CORBA, Java and Jini technologies. The reality of the situation, however, is that a complete distributed computing solution may require the use of several of these technologies in combination. The work in [29, 30], in particular, describes a study in which different combinations of these technologies were examined for the purpose of developing a declarative, rule-based approach to the interconnection of distributed software components.
The two most divergent technologies are CORBA and Jini Connection Technology. CORBA promotes a *language-independent* environment in which clients *transparently* access the services of server objects. Clients invoke remote procedures as though they were local procedures, while the ORB takes care of the distributed computing details. Jini, on the other hand, promotes a *non-transparent, Java-dependent* approach to distributed computing. Jini simply serves as the initial “locator” service that allows client and server objects to find each other. Once a service is found, all communication occurs directly between the client and server objects. The Jini philosophy is that “the network is not transparent” [27], and that efficient distributed computing solutions cannot be achieved by hiding the network under a layer of abstraction. Jini was also initially targeted at application environments involving the flexible interconnection of hardware devices in addition to just software components.

CORBA does provide a service similar to Jini known as the CORBA trading service [31]. In the CORBA trading service, a trader object is used to advertise the services of other objects and to match the needs of an object against advertised services. The trading service supports the dynamic discovery of services in CORBA environments. Jini has only been available since 1998 compared to the ten year maturity of CORBA, but it will be interesting to see how these two different distributed computing paradigms are used in future distributed applications. A primary disadvantage of Jini for software component integration is that the event and transaction services of Jini do not integrate well with EJBs [29].

CORBA and the EJB component model, on the other hand, are two different but compatible pieces of the puzzle for distributed applications. The two technologies can be used together to create sophisticated solutions that are targeted at the interconnection of commercial off-the-shelf software components.

The initial notion of software components with IDL interfaces in CORBA was a simplistic one, where in-house developers could provide a standard interface for communicating with legacy software. Over the last decade, however, the concept of software components has also evolved. This evolution occurred due to the need to provide a better means for software vendors to package together reusable interfaces to their products, thus making it more appealing for application developers to integrate the use of different software and database tools into distributed applications. Developing a CORBA application, for example, requires the distributed object programmer to use CORBA services for transactions and security by calling standard APIs within the code of the objects. This often takes up valuable time in the development cycle since the application programmer is required to understand and implement code to interface with services as opposed to working on the business logic for the objects.
Table 1: Event Service Comparison

<table>
<thead>
<tr>
<th>Evaluation Category</th>
<th>CORBA Event Service</th>
<th>Java Messaging Service</th>
<th>Jini Event Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging Mode</td>
<td>Supports the pull/push mode of messaging.</td>
<td>Supports the pull/push mode and point-to-point messaging.</td>
<td>Supports point-to-point distributed events.</td>
</tr>
<tr>
<td>Message Data Types</td>
<td>Data types that can be passed on event channels are of a generic data type that must be type cast on receiver side.</td>
<td>Supports multiple types that can be used to pass data in the distributed environment.</td>
<td>Java objects (Rich Objects) can be passed in the distributed environment.</td>
</tr>
<tr>
<td>Quality Of Service</td>
<td>With CORBA 3.0, supports configuration of quality of service parameters, such as delivery constraints.</td>
<td>Supports configuration of various delivery options.</td>
<td>Does not support configuration of quality of service parameters.</td>
</tr>
<tr>
<td>Constraints on Message Objects</td>
<td>Requires objects that are passed on the event channel to be CORBA objects.</td>
<td>Java objects are encapsulated in one of the five data types and are passed in an event.</td>
<td>Java objects may be passed in an event.</td>
</tr>
<tr>
<td>Integration</td>
<td>Does not integrate well with EJBs since events must be structured as CORBA objects.</td>
<td>Integrates with EJBs and is being incorporated into the EJB 2.0 specification.</td>
<td>Does not integrate with EJBs. Only Jini objects may communicate using this mechanism.</td>
</tr>
</tbody>
</table>

The goals of the EJB component model are similar to the goals of the CORBA standard since both standards are concerned with simplifying the task of programming in a distributed object environment. The EJB specification, however, addresses the division of labor between distributed computing specialists and business logic specialists in a more comprehensive manner than the CORBA standard. One of the goals of the EJB specification is to allow the application programmer to spend more time developing business logic while leaving the details of distribution and services to the technology of the application server that supports the deployment and execution of EJB components.
Table 2: Transaction Service Comparison

<table>
<thead>
<tr>
<th>Evaluation Category</th>
<th>CORBA Transaction Service</th>
<th>Java Transaction Service</th>
<th>Jini Transaction Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Control Levels</td>
<td>Provides complete transaction service with two-phase commit and resource management.</td>
<td>Based on the CORBA transaction service specifications.</td>
<td>Only provides a coordination framework (communication) for two phase commit protocol.</td>
</tr>
<tr>
<td>Compatibility With EJBs</td>
<td>Since the EJB transaction service is based on the CORBA specification, CORBA transactions integrate with EJBs when a CORBA-based EJB container is used.</td>
<td>Transaction service targeted for EJBs.</td>
<td>Does not integrate with EJBs.</td>
</tr>
<tr>
<td>Nested Transaction Availability</td>
<td>Nested transactions specified but not yet widely available.</td>
<td>Nested transactions not yet implemented in available versions.</td>
<td>Nested transactions not yet implemented in available versions.</td>
</tr>
<tr>
<td>Control Over Transaction Semantics</td>
<td>Transaction semantics are pre-defined and applications must conform to classical transactional frameworks.</td>
<td>Targeted towards the use of EJBs. Transaction semantics are pre-defined since the specification is based on CORBA OTS.</td>
<td>Transaction semantics are left up to the user. Jini only notifies applications of the stages of the transaction. The action to take it left up to the client.</td>
</tr>
<tr>
<td>Flexibility Of Service As A Building Block To Build Hybrid Transaction Protocol</td>
<td>Provides less flexibility in terms of building transactional systems over the service.</td>
<td>Provides less flexibility in terms of building transactional systems over the service</td>
<td>Provides maximum flexibility for designing a custom transaction environment.</td>
</tr>
<tr>
<td>Transaction Context Propagation</td>
<td>Transaction propagation is possible using explicit or implicit transaction propagation.</td>
<td>Transaction propagation is possible using explicit or implicit transaction propagation.</td>
<td>Transaction context propagation is always explicit.</td>
</tr>
<tr>
<td>Constraints On Transaction Participants</td>
<td>Can be used only with CORBA participants/objects.</td>
<td>Can be used with EJB’s as participants.</td>
<td>Can be used with Jini compliant participants/components.</td>
</tr>
<tr>
<td>Scalability Data</td>
<td>Designed and deployed for scalability and high volume applications.</td>
<td>JTS has been designed to handle enterprise application loads and is a scalable solution.</td>
<td>No scalability data exists to compare with CORBA transaction services that have been deployed in large applications.</td>
</tr>
</tbody>
</table>
Internally, the services provided by the EJB component model are written using CORBA [20]. Furthermore, the transaction processing features of EJB components are based on the CORBA Object Transaction Service, with EJB components providing IDL interfaces for interaction with CORBA frameworks. The CORBA event service, on the other hand, does not integrate well with EJB components since objects passed on a CORBA event channel must be CORBA objects. The more recent CORBA notification service [32], which subsumes the original CORBA event service, adds features that are similar to those supported by JMS, such as quality of service parameters and messaging filtering. But since the EJB 2.0 specification has adopted JMS as its event service, applications that need to use CORBA together with event-based processing over EJB components can use JMS rather than the CORBA event or notification service [29].

Fortunately, the forthcoming CORBA 3.0 component model is based on the EJB specification. The primary difference is that the CORBA component model will allow components to be written in languages other than Java. In fact, using the Java and Internet features of CORBA 3.0, it will soon be possible to connect traditional IDL interfaces with Java objects as well as EJB components and CORBA components. The integration of these different technologies will provide significant flexibility for applications that must integrate a wide range of software component implementations.

7. Summary and Future Directions

This paper has presented a summary and comparison of recent developments in distributed object computing technologies. All of these different technologies are relevant to the construction of middleware solutions that are currently needed in engineering applications. From engineering design and manufacturing to electronic commerce, coordinating activities in many applications often requires accessing data and software from distributed sources. The development of any distributed application in the engineering domain should carefully consider the options outlined in this paper. Jini is the newest technology examined, in comparison to CORBA, which is the most mature. The choice of one tool over the other will ultimately depend on the needs of the specific application to be developed. Although the standards are continually evolving, the use of these tools is typically upward compatible with new versions. Furthermore, as described in the previous section, many of these technologies can be used together to make use of available resources.

Even with all of these different options, there is still work to be done at the semantic level of application integration. The Extensible Markup Language (XML) is emerging as a standard for data exchange [33],
which will certainly have an impact on the manner in which data is formatted for exchange between distributed engineering tools. As described in [34], XML is not a replacement for middleware tools such as those discussed in this paper. XML is, instead, a complimentary technology for the storage and manipulation of text for human-readable documents such as Web pages. The development of distributed applications for engineering environments will increasingly need to investigate the manner in which these two technologies can be used together for the extraction and display of data from engineering tools. Better environments are also needed to support views of the services provided by distributed components and to support more flexible, declarative ways of interconnecting software components [30, 35]. Pending such developments, the technologies outlined in this paper provide significant improvements and exciting solutions for distributed application development.

ACKNOWLEDGMENTS

We would like to thank Ying Jin for her help with the preparation of the figures in this paper.

REFERENCES