The Information Bus®– An Architecture for Extensible Distributed Systems

Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen

Teknekron Software Systems, Inc.
530 Lytton Avenue, Suite 301
Palo Alto, California 94301
{boki, pfluegl, alexs, skeen}@tss.com

Abstract

Research can rarely be performed on large-scale, distributed systems at the level of thousands of workstations. In this paper, we describe the motivating constraints, design principles, and architecture for an extensible, distributed system operating in such an environment. The constraints include continuous operation, dynamic system evolution, and integration with extant systems. The *Information Bus*, our solution, is a novel synthesis of four design principles: core communication protocols have minimal semantics, objects are self-describing, types can be dynamically defined, and communication is anonymous. The current implementation provides both flexibility and high performance, and has been proven in several commercial environments, including integrated circuit fabrication plants and brokerage/trading floors.

1 Introduction

In the 1990s, distributed computing has truly moved out of the laboratory and into the marketplace. This transition has illuminated new problems, and in this paper we present our experience in bringing large-scale, distributed computing to mission-critical applications. We draw from two commercial application areas: integrated circuit (IC) fabrication plants and brokerage/trading floors. The system we describe in this paper has been installed in over one hundred fifty production sites and on more than ten thousand workstations. We have had a unique opportunity to observe distributed computing within the constraints of commercial installations and to draw important lessons.

This paper concentrates on the problems posed by a "24 by 7" commercial environment, in which a distributed system must remain operational twenty-four hours a day, seven days a week. Such a system must tolerate software and hardware crashes; it must continue running even during scheduled maintenance periods or hardware upgrades; and it must be able to evolve and scale gracefully without affecting existing services. This environment is crucially important to our customers as they move toward real-time decision support and event-driven processing in their commercial applications.

One class of customers manufactures integrated circuit chips. An IC factory represents such an enormous investment in capital that it must run twenty-four hours a day. Any down time may result in a huge financial penalty from both lost revenue and wasted materials. Despite the "24 by 7" processing requirement, improvements to software and hardware need to be made frequently.

Another class of customers is investment banks, brokers, and funds managers that operate large securities trading floors. Such trading floors are very data-intensive environments and require that data be disseminated in a timely fashion to those who need it. A one-minute delay can mean thousands of dollars in lost profits. Since securities trading is a highly competitive business, it is advantageous to use the latest software and hardware. Upgrades are frequent and extensive. The system, therefore, must be designed to allow seamless integration of new services without affecting existing services.

In the systems that we have built and installed, dynamic system evolution has been the greatest challenge. The sheer size of these systems, which can consist of thousands of workstations, requires novel solutions to problems of system evolution and maintenance. Solving these problems in a large-scale, "24 by 7" environment leads to more than just quantitative differences in how systems are built—these solutions lead to fundamentally new ways of organizing systems.

The contributions of this paper are two-fold. One is the description of a set of system design principles that were crucial in satisfying the stringent requirements of "24 by 7" environments. The other is the demonstration of the usefulness and validity of these principles by discussing a body of software out in the field. This body of software consists of several tools and modules that use a novel communications infrastructure known as the Information Bus. All of the software components work together to provide a complete distributed system environment.

This paper is organized as follows. Section 2 provides a detailed description of the problem domain and summarizes the requirements for a solution. Section 3 outlines the Information Bus architecture in detail, states principles that drove our design, and discusses some aspects of the implementation. Section 4 describes the notion of adapters, which mediate between old applications and in the Information Bus. Section 5 describes other software components that use the Information Bus and provide a complete application environment. This section also provides an example to illustrate the system. Section 6 presents related work. Section 7 summarizes the paper and discusses open issues. The Appendix discusses the performance characteristics of the Information Bus.

2 Background

An IC fabrication plant represents a huge capital investment. This investment, therefore, is cost-effective only if it can remain operational twenty-four hours a day. To bring down an entire plant in order to upgrade a key software component, such as the "Work-In-Process" tracking system, would result in lost revenue and wasted material. There is no opportunity to "reboot" the entire system. We state this requirement as R1:

> **R1** *Continuous operation.* It is unacceptable to bring down the system for upgrades or maintenance.

Despite the need for continuous operation, frequent changes in hardware and software must also be supported. New applications and new versions of existing applications need to be brought on-line. Business requirements and factory models change, and such changes need to be reflected in the application behavior. For example, new equipment types could be introduced into the factory. We state this requirement as R2:

> **R2** *Dynamic system evolution.* The system must be capable of adapting to changes in application architecture and in the type of information exchanged. It should also support the dynamic integration of new services and information.

In the systems that we have built and installed, this requirement has posed the greatest challenge. The sheer size of these systems, typically ranging from one hundred to a thousand workstations, makes changes expensive or even impossible, unless change is planned from the beginning.

Businesses often have huge outlays in existing hardware, software, and data. To be accepted by the business community, a new system must be capable of leveraging existing technology; an organization will not throw away the product of an earlier costly investment. We state this requirement as R3:

> **R3** *Legacy systems.* New software must be able to interact smoothly with existing software, regardless of the age of that software.

Other important requirements are fault-tolerance, scalability, and performance. The system must be fault-tolerant; in particular it must not have a single point of failure. The system must scale in terms of both hardware and data. Finally, our installations must meet stringent performance standards. In this paper, we focus on requirements R1, R2, and R3 because they represent "real-world" constraints that have been less studied in research settings.

The typical customer environment consists of a distributed collection of independent processors, *nodes*, that communicate with each other by passing messages over the network. Nodes and the network may fail, and it is assumed that these failures are fail-stop [Schneider83] and not Byzantine [Lamport82].[1] The network may lose, delay, and duplicate messages, or deliver messages out of order. Link failures may cause the network to partition into subnetworks that are unable to communicate with each other. We assume that nodes eventually recover from crashes. For any pair of nodes, there will eventually be a time when they can communicate directly with each other after each crash.

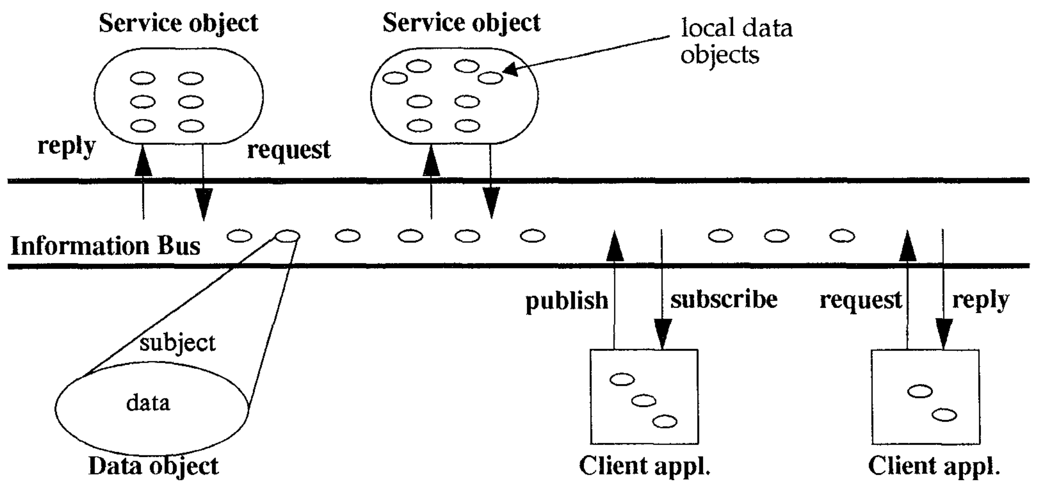
3 Information Bus Architecture

The requirements of a "24 by 7" environment dictated numerous design decisions that ultimately resulted in the Information Bus that we have today. We have distilled those decisions into several design principles, which are highlighted as they become apparent in this section.

Because it is impossible to anticipate or dictate the semantics of future applications, it is inadvisable to hard-code application semantics into the core communications software—for performance reasons and because there is no "right" answer for every application [Cheriton93]. For example, complex ordering semantics on message delivery are not supported directly. Atomic transactions are also not supported: in our experience most applications do not need the strong consistency requirements of transactions. Instead, we provide tools and higher-level services to cover the range of requirements. This allows us to keep the communications software efficient, while still allowing us to adapt to the specific needs of each class of customers. The following principle is motivated by requirements R2 and R3.

1. Failures in our customer environments closely approximate fail-stop behavior; furthermore, protecting against the rare Byzantine failure is generally too costly.

FIGURE 1. Model of computation



P1 *Minimal core semantics.* The core communication system and tools can make few assumptions about the semantics of application programs.

Our model of computation consists of objects, both data objects and service objects, and two styles of distributed communication: traditional request/reply (using remote procedure call [Birrell84]) and publish/subscribe, the hallmark of the Information Bus architecture. This is depicted in Figure 1. Publish/subscribe supports an event-driven communication style and permits data to be disseminated on the bus, whereas request/reply supports a demand-driven communication style reminiscent of client/server architectures. For each communication style, there are different levels of *quality of service*, which reflect different design trade-offs between performance and fault-tolerance. The next section elaborates on these mechanisms.

An *object* is an instance of a class, and each class is an implementation of a type[2]. Our system model distinguishes between two different kinds of objects: *service objects* that control access to system resources and *data objects* that contain information and that can easily be transmitted. A *service object* encapsulates and controls access to resources such as databases or devices and its local data objects. Service objects typically contain extensive state and may be fault-tolerant. Because they tend to be large-grained, they are not easily marshalled into a wire format and transmitted. Instead of migrating to another node, they are invoked where they reside, using a form of remote procedure call. Examples of service objects include network file systems, database systems, print services, and name services.

A *data object*, on the other hand, can be easily copied, marshalled, and transmitted. Such objects are at the granularity of typical C++ objects or database records. They abstract and encapsulate application-level concepts such as documents, bank accounts, CAD designs, and employee records. They run the gamut from abstracting simple data records to defining complex behaviors, such as "recipes" for controlling IC processing equipment.

Each data object is labelled with a *subject* string. Subjects are hierarchically structured, as illustrated by the following well-formed subject "fab5.cc.litho8.thick." This subject might translate to plant "fab5," cell controller, lithography station "litho8," and wafer thickness. Subjects are chosen by applications or users.

The second principle, P2, is motivated by requirements R2 and R3, and together with data abstraction, allows applications to adapt automatically to changes in an object's implementation and data representation.

P2 *Self-describing objects.* Objects, both service and data objects, are "self-describing." Each supports a *meta-object protocol* [Kiczales91], allowing queries about its type, attribute names, attribute types, and operation signatures.

P2 enables our systems and applications to support introspective access to their services, operations, and attributes. In traditional environments, introspection is used to develop program analysis tools, such as class browsers and debuggers. In the Information Bus environment, introspection is used by applications to adapt their behaviors to change. This is key to building systems that can adapt to change at run-time.

2. A type is an abstraction whose behavior is defined by an *interface* that is completely specified by a set of *operations*. Types are organized into a supertype/subtype hierarchy. A class is an implementation of a type. Specifically, a class defines methods that implement the operations defined in a type's interface.

Introspection enables programmers to write generic software that can operate on a wide range of types. For example, consider a "print" utility. Our implementation of this utility can accept any object of any type and produce a text description of the object. It examines the object to determine its type, and then generates appropriate output. In the case of a complex object, the utility will recursively descend into the components of the object. The print utility only needs to understand the fundamental types, such as integer or string, but it can print an object of any type composed of those types.

The third principle, P3, enables new concepts and abstractions to be introduced into the system.

P3 *Dynamic classing.* New classes implementing either existing or new types can be dynamically defined and used to create instances. This is supported for both service and data objects.

P3 enables new types to be defined, on-the-fly. Note that P2 enables existing applications to make use of these new types without re-programming or re-linking.

To support dynamic classing, we have implemented TDL, a small, interpreted language based on CLOS [Keene89]. We have chosen a subset of CLOS that supports a full object model, but that could be supported in a small, efficient run-time environment.

3.1 Publish/Subscribe Communication

To disseminate data objects, data producers generate them, label them with an appropriate subject, and *publish* them on the Information Bus. To receive such objects, data consumers *subscribe* to the same subject. Consumers need not know who produces the objects, and producers need not know who consumes or processes the objects. This property is expressed in principle P4. We call this model *Subject-Based Addressing*™, and it is a variant of a generative communication model [Carriero89]. This principle is motivated by requirements R1 and R2, and it allows applications to tolerate architectural changes on the fly.

P4 *Anonymous communication.* Data objects are sent and received based on a subject, independent of the identities and location of data producers and data consumers.

Subjects can be partially specified or "wildcarded" by the consumer, which permits access to a large collection of data from multiple producers with a single request. The Information Bus itself enforces no policy on the interpretation of subjects. Instead, the system designers and developers have the freedom and responsibility to establish conventions on the use of subjects.

Anonymous communication is a powerful mechanism for adapting to software changes that occur at run-time. A new subscriber can be introduced at any time and will start receiving immediately new objects that are being published under the subjects to which it has subscribed. Similarly, a new publisher can be introduced into the system, and existing subscribers will receive objects from it. Our model of computation does not require a traditional name service like Sun's NIS or Xerox's Clearinghouse [Oppen83].

In a traditional distributed system, whenever new services are added to the system, or a service is being replaced with a new implementation, the name service must be updated with the new information. To use that information, all applications must be aware that the new services exist, must contact the name service to obtain the location of the new service, and then bind to the service. In our model, the new implementation need only use the same subjects as the old implementation; neither publishers nor subscribers must be aware of the change. Subject names can be rebound at *any* time to a new address, a facility that is more general than traditional late-binding.

The semantics of publish/subscribe communication depends on the requirements of the application. The usual semantics we provide is *reliable* message delivery. Under normal operation, if a sender and receiver do not crash and the network does not suffer a long-term partition, then messages are delivered exactly once in the order sent by the same sender; messages from different senders are not ordered. If the sender or receiver crashes, or there is a network partition, then messages will be delivered at most once.

A stronger semantics is *guaranteed* message delivery. In this case, the message is logged to non-volatile storage before it is sent. The message is guaranteed to be delivered at least once, regardless of failures. The publisher will retransmit the message at appropriate times until a reply is received. If there is no failure, then the message will be delivered exactly once. Guaranteed delivery is particularly useful when sending data to a database over an unreliable network.

For local area networks, reliable publication is implemented with Ethernet broadcast. This choice allows the same data to be delivered to a large number of destinations without a performance penalty. Moreover, Ethernet broadcast eliminates the need for a central communication server. Our current implementation uses UDP packets in combination with a retransmission protocol to implement reliable delivery semantics.

In our implementation of subject-based addressing, we use a daemon on every host. Each application registers with its local daemon, and tells the daemon to which subjects it has subscribed. The daemon forwards each message to each application that has subscribed. It uses the subject contained in the message to decide which application receives which message.

Given the high traffic rates, Ethernet broadcast across wide area networks is undesirable. We could use IP multicast [Cheriton85], but unfortunately, commercial implementations are not mature enough for mission-critical use. Therefore, wide area networks require additional communication tools.

Our implementation uses application-level "information routers" to solve the problem posed by wide area networks. To the Information Bus, these routers look like ordinary applications, but they actually integrate multiple instances of the bus. Messages are received by one router using a subscription, transmitted to another router, and then re-published on another bus. The router is intelligent about which messages are sent to which routers: messages are only re-published on buses for which there exists a subscription on that subject; the router can also perform other functions, such as transforming subjects or logging messages to non-volatile storage. Thus, the overall effect is to create the illusion of a single, large bus that is capable of publishing over any network.

3.2 Dynamic Discovery

In a distributed system, it is often necessary for an application to discover the identify of the participants in a protocol. For example, a new client needs to determine the set of servers that serve a subject; a new server needs to determine if any clients have pending requests; a replicated server needs to find the other servers that maintain the replicated data. Specifically, in Xerox's corporate email service, a traditional distributed system, client mail applications find a mail service for posting mail messages by using an expanding ring broadcast technique, a kind of discovery protocol [Xerox88].

In the Information Bus, the discovery protocol is in the form of two publications. One participant publishes "Who's out there?" under a subject. The other participants publish "I am" and other information describing their state, if they serve the subject in question. Section 3.3 provides a specific example of this exchange. This approach preserves P4 (anonymous communication). The subject alone is enough for one participant to make contact with its cohorts.

The publish/subscribe communication model is well-suited to supporting a discovery protocol. Since publication does not require any boot-strapping or name resolution, it can be the first step in a protocol. We are effectively using the network itself as a name service. A subject is mapped to a specific set of servers by allowing the servers to choose themselves. The "Who's out there?" publication can contain service-specific information, so further refinements are possible when selecting servers.

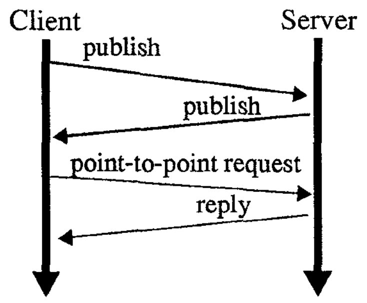
3.3 Remote Method Invocation

Remote method invocation (RMI), or remote procedure call, is the second means of performing distributed computations. This paired request/reply satisfies the demand-driven style of interaction. Clients invoke a method on a remote server object without regard to that server object's location, the server object executes the method, and the server replies to the client. Servers are named with subjects.

Standard RMI provide *exactly-once* semantics under normal operation and *at-most-once* semantics in the presence of failures. Customer-specific requirements such as *exactly-once* semantics, which guarantees that the method will be executed exactly once, even in the presence of failures, can be built on the a layer above standard RMI.

There are two parts to RMI: discovering the server object for a client, and establishing a connection to that server over which requests and replies will flow. The discovery algorithm in our implementation employs publish/subscribe communication as described in Section 3.2. In this algorithm, the client searches for all servers by publishing a query message on a subject specific to that service. The servers receive this message, and then they publish their point-to-point address to all clients on the same subject. Finally, the client invokes a service request on a server object using the point-to-point address. The point-to-point address can refer to any simple, connection mechanism, such as a TCP/IP connection [Postel81]. Figure 2 illustrates this protocol.

FIGURE 2. RMI Protocol



More than one server can respond to requests on a subject. Several server objects can be used to provide load balancing or fault-tolerance. Our system allows an application to choose between several different policies. The servers can decide among themselves which one will respond to a request from the client. Alternatively, the client can receive every response from all of the servers and then decide which server the client wants to use.

4 Adapters

The Information Bus must allow for interaction with existing systems, as dictated by requirement R3. To integrate existing applications into the Information Bus we use software modules called *adapters*. These adapters convert information from the data objects of the Information Bus into data understood by the applications, and vice versa. Adapters must live in two worlds at once, translating communication mechanisms and data schemas. Adapters often require P1 in order to be feasible.

Adapters are essential for integrating the Information Bus into a commercial environment. In the factory floor example, our customer already had a Work In Progress (WIP) system with its own data schemas. We designed an adapter that allows the existing WIP software to communicate with the Information Bus. This achievement demonstrates the flexibility of the Information Bus model: the existing WIP system is written in Cobol, and there is only a primitive terminal interface. The adapter must act as a virtual user to the terminal interface.

The Object Repository is an example of a sophisticated adapter that integrates a commercially available relational database system into the Information Bus architecture. The Object Repository maps Information Bus objects into data base relations for storage or retrieval. This mapping is driven by the meta-data of each object. Besides satisfying requirement R3 as an adapter, the design of the repository also supports dynamic system evolution, which satisfies requirement R2. Users may work freely in the object model without concerning themselves with the relational data model[3] [Codd70]. Using P2, the repository can automatically adapt the relational model to the type structure of the data objects.

The repository behaves as a kind of schema converter from objects to database tables, and vice versa. Users are thus insulated from any changes implementors may wish to make to the database representation of objects. For example, our conversion algorithm decomposes a complex object into one or more database tables and reconstructs a complex object from one or more database tables to answer a query from a user. This conversion respects the type hierarchy, enabling queries to return all objects that satisfy a constraint, including objects that are instances of a subtype. Old queries will still work even as new subtypes are introduced, which helps to satisfy R2. This operation can be fully automated; only the type information is necessary to do the transformation. When the repository needs to store an instance of a previously unknown type, it is capable of generating one or more new database tables to represent the new type.

3. Our object model differs significantly from the relational data model in the following way. A database table is a flat structure composed of simple data types and has little semantics, while an object may contain other objects, may have subtypes or supertypes, and may have methods to manipulate instances of the type.

The repository may be configured in any number of ways, depending on the application. For example, it may be configured as a capture server that captures all objects for a given set of subjects and inserts those objects automatically into the repository under those subjects; it may also be configured as a query server to receive requests from clients and return replies.

5 Example Application

In the previous section, we discussed the Information Bus architecture primarily in terms of an abstract object model and two communication styles, and we espoused several principles of system design. To make the architecture more concrete, we present an example that shows how the various components fit together into a single application and that illustrates how an application can adapt to changes in the environment. In particular, we show how the principles are applied in the context of the example, and discuss some software components that have been built on top of the Information Bus and that are installed at customer sites.

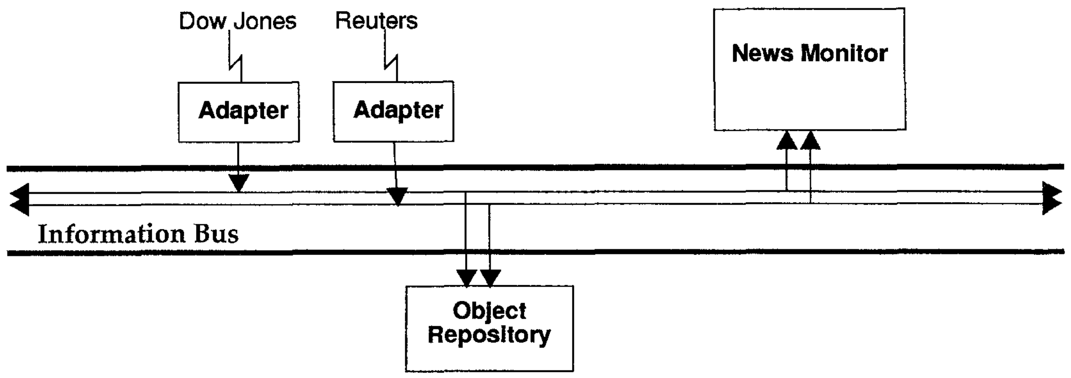
Figure 3 shows an example taken from a financial trading floor, where traders must receive news stories in near real-time. Two news adapters receive news stories from communication feeds connected to outside news services, such as Dow Jones and Reuters. Each raw news service defines its own news format. Each adapter parses the received data into an appropriate vendor-specific subtype of a common Story supertype, and publishes each story on the Information Bus under a subject describing the story's primary topic (for example, "news.equity.gmc" for stories on General Motors). P1 ensures that the raw feeds do not have to support complex semantics.

The figure depicts two applications that consume the Story objects: the Object Repository and the News Monitor. The News Monitor subscribes to and displays all stories of interest to its user. Incoming stories are first displayed in a "headline summary list." This list format is defined by a "view" that specifies a set of named attributes[4] from incoming objects and formatting information. When the user selects a story in the summary list, the entire story is displayed. This is accomplished by using the object's metadata to iterate through all of its attributes and display them, as provided by P2.

The Object Repository subscribes to all news stories and inserts them into a relational database. The repository converts Story objects into a database table format. This conversion is nontrivial because a story is a highly structured object containing other objects such as lists of "industry groups," "sources," and "country codes." Every object

4. "Attributes" of an object are often referred to as "instance variables" or "fields."

FIGURE 3. Brokerage Trading Floor



must be mapped into collections of simple database relations.

5.1 Graphical Application Builder

We needed a simple, general way to access information on the Information Bus and in the database in a pleasing, graphical form. It was not satisfactory to build a single, static solution, since each customer has different needs, and they change frequently. Instead, we built a graphical application builder, designed for applications with a graphical user interface builder. We have used it for several applications, including the News Monitor example and the front-end to a Factory Configuration System, which is the system for storing factory control information.

The application builder is an interpreter-driven, user interface toolkit. It combines the ability to construct sophisticated user interfaces with a simple, object-oriented language. All high-level application behavior is encoded in the interpreted language; only low-level behavior that is common to many applications is actually compiled.

The resulting applications are fully integrated into the Information Bus, providing access to all subjects and services. It is possible to examine the list of available services on the Information Bus by using various name services. Services are self-describing, so users can inspect the interface description for each service. Using that information, a user can quickly construct a basic user interface for any service. This whole process requires only a few minutes, and typically no compilation is involved. Sometimes, a single user interface can be used to access several services, further reducing the amount of work involved.

5.2 Dynamic System Evolution

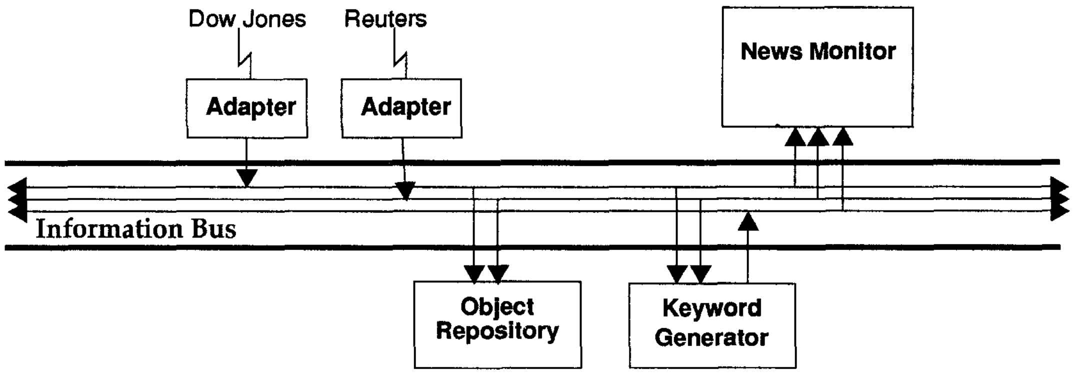
In this section we illustrate how our design principles support the requirement of dynamic system evolution. First, we consider the introduction of a new type into the Information Bus (P3). The Object Repository can dynamically generate extensions to the database schema to accommodate such new types. Such generation may entail creating one or more new database tables for each new type, depending on the particular database representation. When instances of the new type are received, they are dynamically converted to the new database schema. P2 implies that the repository will be able to recognize and process objects that have a new type.

Second, we consider what happens if a new service of a completely different nature is introduced. Consider a Keyword Generator, as illustrated in Figure 4. The Keyword Generator subscribes to stories on major subjects and searches the text of each story for "keywords" that have been designated under several major "categories." For each Story object, a list of keywords is constructed as a named Property[5] object of the Story object and published under the same subject. It also supports an interactive interface that allows clients to browse categories and associated keywords.

When the Keyword Generator comes on-line, the News Monitor will start receiving Property objects on the same subjects on which it is receiving Story objects. According to P4, the News Monitor will be able to receive the new data immediately. Since properties are a general concept in the architecture, it can reasonably be assumed the News Monitor is configured to accept Property objects, to associate them with the objects they reference, and to display them along with the attributes of an object when the object is selected. This capability can be set up using the scripting language of the Graphical Application Builder,

5. The Object Management Group's "Object Services Architecture" is the basis for the nomenclature used here. Accordingly, a "property" is a name-value pair that can be dynamically defined and associated with an object. In this example, the property name is "keywords" and the value is the set of keywords found.

FIGURE 4. Adding a Key Word Generator to the Brokerage Trading Floor.



The interactive interface of the Keyword Generator is an instance of a new service type. Using introspection, the News Monitor can enable the user to interact with this new type: menus listing the operations in the interface can be popped up, and dialogue boxes that are based on the operations' signatures can lead the user through interactions with the new service.

Hence, as soon as the Keyword Generator service comes on-line, the user's world becomes much richer, both in terms of information and of services. As seen by the user, the new service and information is dynamically integrated into the environment. This shows the adaptive flexibility of the overall approach. Note that the example requires only that the News Monitor support Property objects, but it does not require knowledge of how properties are generated, in compliance with P4.

6 Related Work

The Linda system, developed at Yale University, was the first system to support a generative communication model [Carriero89]. In Linda, processes generate *tuples*, which are lists of typed data fields. These generated tuples are stored in *tuple space* and persist until explicitly deleted. Other Linda processes may invoke operations to remove or read a tuple from tuple space. Storing a tuple in tuple space, in effect, is like one process "broadcasting" a tuple to many other processes.

A key difference between Linda and the Information Bus is the data model. Linda tuples are data records, not objects. Moreover, Linda does not support a full meta-object protocol. Self-describing objects have been invaluable in enabling data independence, the creation of generic data manipulation and visualization tools, and achieving the system objective of permitting dynamic integration of new services.

Another key difference between Linda and the Information Bus is the mechanism for accessing data. Linda accesses data based on attribute qualification, just as relational databases do. Though this access mechanism is more powerful than subject-based addressing, we believe that it is more general than most applications require. We have found that subject names are quite adequate for our needs, and they are far easier to implement than attribute qualification. We also argue that subject-based addressing scales more easily, and has better performance, than attribute qualification.

In the ISIS system [Birman89] processes may join *process groups*, and messages can be addressed to every member of a process group. ISIS has focused on various message delivery semantics without regard for application-level semantics. Hence, it does not support a high-level object model.

Usenet [Fair84] is the best known example of a large-scale communications system. A user may post an *article* to a *news group*. Any user who has subscribed to that news group will eventually see the article. Usenet, however, makes no guarantees about message delivery: messages can be lost, duplicated, or delivered out of order. Delivery latency can be very large, on the order of weeks in some cases. On the other hand, Usenet moves an impressive volume of data to a huge number of sites.

Usenet should be viewed as a communication system, whose focus is moving text among humans. News articles are unstructured, and no higher-level object model is supported. It would make an unsuitable communications environment for our customer's applications, given its weak delivery semantics and long latencies.

The Zephyr notification service [DellaFera88], developed at MIT as part of Project Athena [Balkovich85], is used by applications to transport time-sensitive textual information asynchronously to interested clients in a distributed workstation environment. The notice subscription service layer is of particular interest because it most resembles our publish/subscribe communication model. In Zephyr, a client interested in receiving certain classes of messages, registers its interest with the service. The service uses "subscription multicasting" (their term) to compute dynamically the set of subscribers for a particular class of message and sends a copy of the message only to those recipients that have subscribed.

This subscription multicasting mechanism relies heavily on a centralized location database that maps unique Zephyr IDs to information like geographical location and host IP address, and it is not at all clear how well such an implementation would work in a wide-area network. Furthermore, this mechanism is inefficient if the number of interested clients is very large.

7 Conclusion

In this paper, we described the requirements posed by a "24 by 7" commercial environment, such as the factory floor automation system of a semiconductor fabrication plant. The centerpiece of our solution is the Information Bus. The Information Bus has been ported to most desktop and server platforms, and has been installed at more than one hundred fifty sites around the world, running on over ten thousand hosts. We have demonstrated that this architecture is a successful approach to building distributed systems in a commercial setting.

Minimal core semantics (P1), self-describing objects (P2), a dynamic classing system (P3), and anonymous communication (P4) allow applications that use the Information Bus to evolve gracefully over time. P1 prevents applications from being crippled by the communication system. P2 allows new types to be handled at run-time. P3 enables new types to be introduced without recompilation. P4 permits new modules to be transparently introduced into the environment.

The first requirement was continuous availability (R1). Anonymous communication (P4) allows a new service to be introduced into the Information Bus. A new server that implements such a service can transparently take over the function of an obsolete server. The old server can be taken off-line after it has satisfied all of outstanding requests. With this technique, software upgrades can be performed on a live system. In addition, new services can be offered at any time, and existing clients can take advantage of these new services.

The second requirement was support for dynamic system evolution (R2). New services and new types can be added to the Information Bus without affecting existing services or types. Self-describing data (P2) ensures that the data model and data types can be substantially enhanced without breaking older software. In many cases, older software can make use of the enhancements in the data objects immediately. This ability implies that applications can provide additional functionality by only changing the data model.

The third requirement was the ability to integrate legacy systems (R3). The Information Bus connects to legacy systems through the use of adapters (Section 4), which mediate between other systems and the Information Bus. The principle of minimal core semantics (P1) aids in the construction of adapters.

Acknowledgments

The authors wish to thank the anonymous referees for their many helpful comments. We thank Mendel Rosenblum for passing on to us the program committee's comments that greatly strengthened the final version of the paper. Our thanks also go to Tommy Joseph, Richard Koo, Kieran Harty, Andrea Wagner, and Brendon Whateley. Finally, we thank TSS management, and Dr. JoMei Chang in particular, for their patience during the lengthy preparation of this paper.

References

[Balkovich85] Balkovich, E., S.R. Lerman, and R.P. Parmele. "Computing In Higher Education: The Athena Experience," *Communications of the ACM 28*, **11** (November 1985), pp. 1214–1224.

[Birman89] Birman, Ken, and Thomas Joseph. "Exploiting Replication in Distributed Systems," *in Distributed Systems*, Mullender, Sape, editor, Addison-Wesley, 1989, pp. 319-365.

[Birrell84] Birrell, Andrew D. and Bruce J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems 2*, 1(February, 1984), pp. 39–59.

[Carriero89] Carriero, Nicholas and David Gelernter. "Linda in Context". *Communications of the ACM 32*, 4 (April, 1989), pp. 444–458.

[Cheriton85] Cheriton, David R. and Steven E. Deering. "Host Groups: a multicast extension for datagram internetworks." In *Proceedings of the 9th Data Communications Symposium, ACM SIGCOMM Computer Communications Review 15*, 4 (September 1985), pp. 172-179.

[Cheriton93] Cheriton, David R. and Dale Skeen. "Understanding the Limitations of Causally and Totally Ordered Communication." In *Proc. of the 14th Symp. on Operating Systems Principles*, Asheville, North Carolina, December 1993.

[Codd70] Codd, E. F. "A Relational Model for Large Shared Data Banks." *Communications of the ACM 13*, **6** (June, 1970).

[DellaFera88] DellaFera, C. Anthony, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Summersfeld. "The Zephyr Notification Service," *Usenix Conference Proceedings*, Dallas, Texas (February 1988).

[Fair84] Erik Fair, "Usenet, Spanning the Globe." *Unix/World*, 1 (November, 1984), pp. 46-49.

[Lamport82] Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), pp. 382-401.

[Keene89] Keene, Sonya. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

[Kiczales91] Kiczales, Gregor, Jim des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Oppen83] Oppen, Derek C. and Y. K. Dalal. "The Clearinghouse: A decentralized agent for locating named objects in a distributed environment." *ACM Tranactions on Office Information Systems 1*, **3** (July 1983), pp. 230-253.

[Postel81] Postel, Jon, "Internet Protocol - DARPA Internet Program Protocol Specification," *RFC 791*, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[Schneider83] Schneider, Fred. "Fail-Stop Processors." *Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference*, March 1983, pp. 66-70.

[Skeen92] Skeen, Dale, "An Information Bus Architecture for Large-Scale, Decision-Support Environments," *Unix Conference Proceedings*, Winter 1992, pp. 183-195.

[Xerox88] Mailing Protocols. Xerox System Integration Standard (May 1988), XNSS 148805.

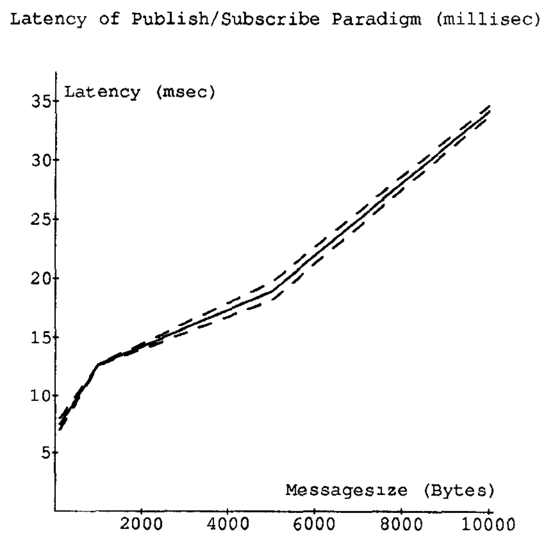
Appendix

For some of our customers, the Information Bus must handle thousands of nodes with thousands of publishers, consumers, clients, and servers. Therefore, performance is crucial. In this section, we measure the performance of the publish/subscribe communication model.

The two factors that most characterize performance are *throughput*, measured in messages per second or bytes per second, and *latency*, measured in seconds. Latency is the average time between the sending of a message and its receipt. In this appendix we present several figures illustrating the performance of the publish/subscribe communication model. The key parameters for performance are the message size and the number of data consumers. Hence, we will plot the throughput and latency versus message size in bytes and explain the effect of the number of consumers.

All data presented here was collected on our development network of Sun SPARCstation 2s and Sun IPXs with twenty-four to forty-eight megabytes of memory running SunOS 4.1.1. The network was a 10 Megabits/second Ethernet, and it was lightly loaded. Since all monitored publishers/consumers are on the same subnet, information does not need to go through any bridges or routers. All message delivery is reliable but not guaranteed. For any given test run, the message size was constant. For the performance data shown here, publishers and consumers were spread over fifteen nodes.

FIGURE 5. Latency vs. Msg Size



The data for Figure 5 was collected by executing one publisher publishing under a single subject. The information is consumed by fourteen consumers (one consumer per node). It shows that the latency depends on the message size. Although not shown, the latency is independent of the number of consumers. The 99%-confidence interval is presented with dashed lines. The Information Bus has a batch parameter that increases throughput by delaying small messages, and gathering them together. When measuring the latency, the batch parameter was turned off to avoid intentionally delaying the publications. Variances of the data sets used in Figure 5 ranged from 1.1×10^{-4} to 1.7×10^{-2} milliseconds.

FIGURE 6. Throughput - Msgs/Sec vs. Msg Size

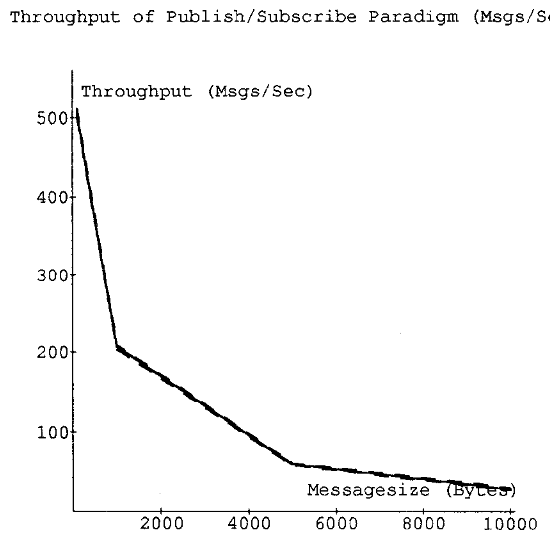


FIGURE 7. Throughput - Bytes/Sec vs. Msg Size

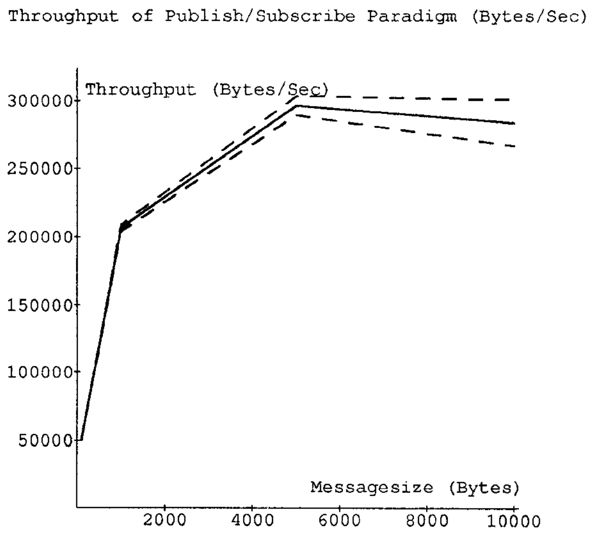
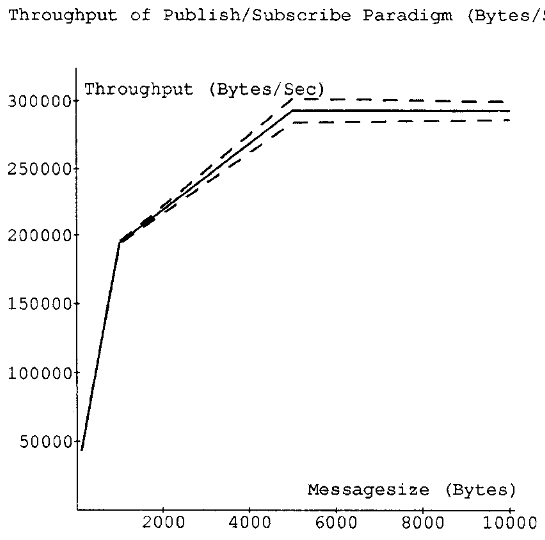


Figure 6 and 7 show the throughput of a network with one publisher publishing under one subject, sending to fourteen consumers. For this test, the batching parameter was turned on. For messages larger than five thousand bytes, the device bandwidth becomes the limiting factor: it is difficult to drive more then 300 Kb/sec through Ethernet with a raw UDP socket, suggesting that the Information Bus represents a low overhead. The slight decrease in throughput and increase in variance between five thousand and ten thousand-byte messages is due to collisions from unrelated network activity.

This set of test cases also verified that the publication rate is independent of the number of subscribers. Therefore, the cumulative throughput over all subscribers is proportional to the number of subscribers. The variances of the data sets used in Figure 6 ranged from 0.25 to 125 messages/second. Figure 7 was plotted based on the same data.

FIGURE 8. Throughput - Effect of the Number of Subjects



The difference between the environment of Figure 7 and Figure 8 is that the publisher published on ten thousand different subjects instead of one, and the fourteen consumers subscribed to all ten thousand subjects. As the data in Figure 8 shows, the number of subjects has an insignificant influence on the throughput. For Figure 8, we collected data in messages/second. These data sets have a variance that ranges from 1.2 to 4.6 messages/second. The time to process each subscription request is not shown in the above figure since these requests are performed once at start-up time.