Proceedings of the

Fourteenth ACM Symposium on Operating Systems Principles

December 5–8, 1993
The Grove Park Inn and Country Club
Asheville, NC

Sponsored by
Association for Computing Machinery (ACM)
Special Interest Group on Operating Systems (SIGOPS)

The Association for Computing Machinery
1515 Broadway
New York, N.Y. 10036

ACM ISBN: 0-89791-632-8

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 12114
Church Street Station
New York, N.Y. 10257

Phone: 1-800-342-6626
(U.S.A. and Canada)
1-212-626-0500
(All other countries)
Fax: 1-212-944-1318
E-mail: acmpubs@acm.org

ACM Order Number: 534930

Printed in the U.S.A.

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

SIGOPS '93/12/93/N.C., USA

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.

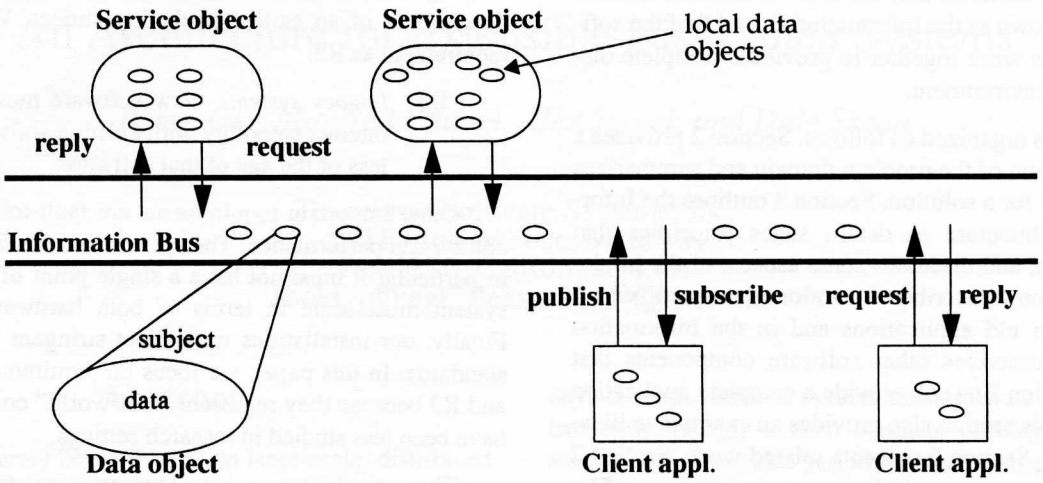
3 Information Bus Architecture

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.