7th ACM SIGOPS European Workshop. September, 1996. Connemara, Ireland.

The *trading function* in action

Bruce Jacob and Trevor Mudge
Advanced Computer Architecture Lab
EECS Department, University of Michigan
{blj,tnm}@eecs.umich.edu

This paper describes a commercial software and hardware platform for telecommunications and multimedia processing. The software architecture loosely follows the CORBA and ODP standards of distributed computing and supports a number of application types on different hardware configurations. This paper is the result of lessons learned in the process of designing, building, and modifying an industrial telecommunications platform. In particular, the use of the trading function in the design of the system led to such benefits as support for the dynamic evolution of the system, the ability to dynamically add services and data types to a running system, support for heterogeneous systems, and a simple design performing well enough to handle traffic in excess of 40,000 busy-hour calls.

Introduction

This paper describes several benefits of the trading function in the context of a commercial closed system—the Telecom Services Architecture, a commercial architecture that handles the transport and processing of several hundred concurrent IO streams. The software architecture of the system loosely follows the CORBA and ODP standards of distributed computing [OMG93, ITU92] and supports a number of applications on many hardware organizations. The system is organized around the principle of the ODP trading function. The key advantage to the trading function is the flexibility it affords a system; while flexibility is desirable in an academic system, it is vital in a commercial system.

The following are a few of the realities of commercial systems.

- The requirements of the system are constantly being modified, due to decisions in the marketing department, reactions to market trends, or specific customer demands. This activity generally results in changes to the product specification. The changes typically occur more rapidly than the development team can finish implementing the previous changes to the specification.
- Customers frequently want special features that no other customer wants. This places demands on the flexibility, modularity, and generality of the product design. However, too much generality is bad when it results in a system that does not perform well.
- The product is occasionally "blind-sided" with requirements for features that were not foreseen or planned for in the original design. For example, scalability is an obvious future requirement for a product; is it likely that customers will want larger or faster systems in the future. By contrast, support for an Internet connection is non-obvious; several years ago few could have predicted the rapid commercial rush to the Internet and the World-Wide Web.

The *trading function* of the CCITT ODP Recommendation [ITU92, ITU94] is a simple concept that indirectly addresses these concerns. In the trading function, a *trader* acts as a yellow pages directory for services. Server objects that wish to offer services advertise, or *export*, their capabilities to the trader. An advertisement consists of a description of the service and the location of an interface providing the service. Client objects that wish to obtain services send requests to the trader and then *import* information about servers that match their needs. A client request is a simple description of a service desired; the trader matches client requests against the descriptions provided by the server objects.

Though this may seem to be designed with a heterogeneous on-line marketplace in mind, it maps very well to a closed system in which the environment is likely to change. In such a system, the less state spread around the server and client objects (either explicitly or implicitly) the better. In live systems, for instance those handling telephone calls or monitoring medical equipment, when software is upgraded it is unacceptable to restart the system software, let alone restart the operating system or reboot the machine. However, if new data types or services are added to the system it is frequently the case that much of the software needs to be replaced or restarted. This is necessary to let servers know about the new data types and clients know about the new services. Object-oriented designs can mitigate this problem by having self-describing data types, but they seem to represent a smaller portion of industrial software than one would expect.

A system that is based around the trading function naturally tends toward a design where the objects involved know very little about each other, and we have found that the amount of state in the system can be reduced to the following:

- Servers need only know how to reach the trader and the names and interfaces of the services they offer.
- Clients need only know how to reach the trader and the names and interfaces of the services they need,

This is very important: clients do not need to know what servers provide what services or where they are located, servers need only know the location of the trader (as well as the agreed-upon service interfaces, trivial in a closed system), and the trader can be more or less stateless.

The following sections describe our system software and hardware, and illustrate a few of the benefits of a system organized around the trading principle.

Software design

Fig 1 illustrates the software architecture at a high level of abstraction. It is an example of the *active object model*, using

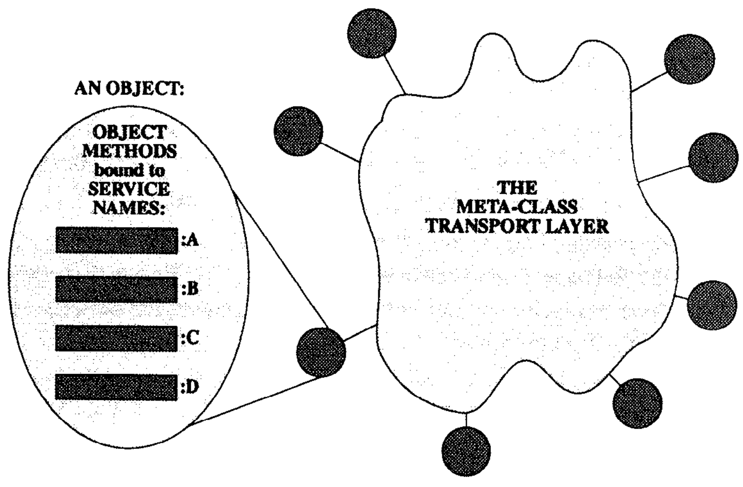
AN OBJECT:
OBJECT METHODS bound to SERVICE NAMES:
:A
:B
:C
:D
THE META-CLASS TRANSPORT LAYER

Figure 1. High level software organization: Object interactions

a popular taxonomy [Chin88]. The system is made up of numerous interacting objects, each of which is a collection of methods bound to service names. The service names are globally visible, and advertised through a central point. Server objects advertise their services through the central point, and client objects obtain the services through this point.

It is a loose implementation of the trading function the ODP Recommendation, not a strict implementation, as the architecture was defined in 1991 before the ODP Trading Function draft was published (in 1994 [ITU94]). In the trading function, exporting servers advertise their services to the trader, who retains the state and acts like a service database to inquiring clients. Importing clients make requests of the trader of the form, "give me a service which behaves like the following ..." The Telecom Services Architecture is a bit different in that since it is a closed system, a client may assume that any service it requests is offered somewhere in the system. Therefore the trader can be stateless, as it does not need to be able to return a message of the form, "no such service exists." Clients post requests to the trader, where they are placed on a prioritized queue from which servers pick up requests they can handle.

This is different from the ODP trading function, where servers advertise first and then wait for client requests to arrive. In the Telecom Services Architecture, the client objects place requests with the trader. Servers poll the trader for work to do; when a server registers itself with the trader it is implicitly ready for work at that moment. This way the trader never has to tell a server to wake up (and potentially block), and clients are never (or, at least *rarely*) in the position of having to deal with defunct servers who leave stale service advertisements with the trader.

The system is comprised of two primary elements:

- *Distributed persistent objects.* Applications in the system are implemented as objects, all derived from a single meta-class which provides method invocation and communication to other objects in the system, transparently to the object itself. An object method executes when another object, or another method within the same object, requests the service bound to that method. A method "wakes up" with a service request in its lap, and performs the service. When finished processing, the method returns a service reply and exits.
- *The trading function—descriptive lookup of object methods.* Objects do not reference each other or each other's methods directly. The methods are bound to global descriptive service names, such as "print" or "fax" or "video." All methods which implement the same service are bound to the same service name. An object requiring a service therefore requests the "print" service or the "fax" service. Requests go to the central *trader,* where they are inserted into a timeout queue similar to the callout table in Unix [Bach86]. Thus, an object needing a particular service only needs to know the name of the service, the interface of the service, and the address of the trader.

Fig 2 illustrates the primary components of the software architecture at a lower level of abstraction. The meta-class

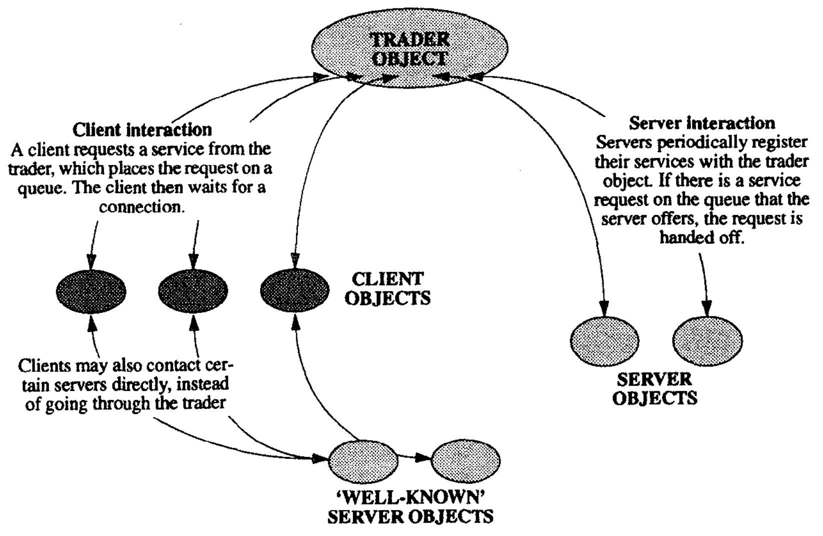
TRADER OBJECT
Client interaction
A client requests a service from the trader, which places the request on a queue. The client then waits for a connection.
Server interaction
Servers periodically register their services with the trader object. If there is a service request on the queue that the server offers, the request is handed off.
CLIENT OBJECTS
Clients may also contact certain servers directly, instead of going through the trader
SERVER OBJECTS
'WELL-KNOWN' SERVER OBJECTS

Figure 2. Lower level software organization: Trading function interactions

layer exists as a library in each object, and not as an underlying software layer in the operating system. The layer handles inter-object messaging and the invocation of local methods to support remote object method invocation. Clients objects send service requests to a central trader. The requests wait on a timeout queue until a server object registers a matching service. At this point, the trader hands the request to the registering server object, and the meta-class layer invokes the appropriate object method.

The trader is a potential bottleneck, but not a repository for global state. It holds only the timeout queue: a list of outstanding service requests. The single trader can easily be replaced by several to distribute load and state. Except for the queue, the trader is stateless, so a configuration of multiple traders would not require any communication between them. This was to be the design of the system if it ever grew beyond a hundred nodes.

Hardware configuration

The software can run on virtually any distributed configuration. The configuration of the original system is pictured in Fig 3. It consists of distributed nodes communicating via Ethernet. Each node is a processor with memory, several

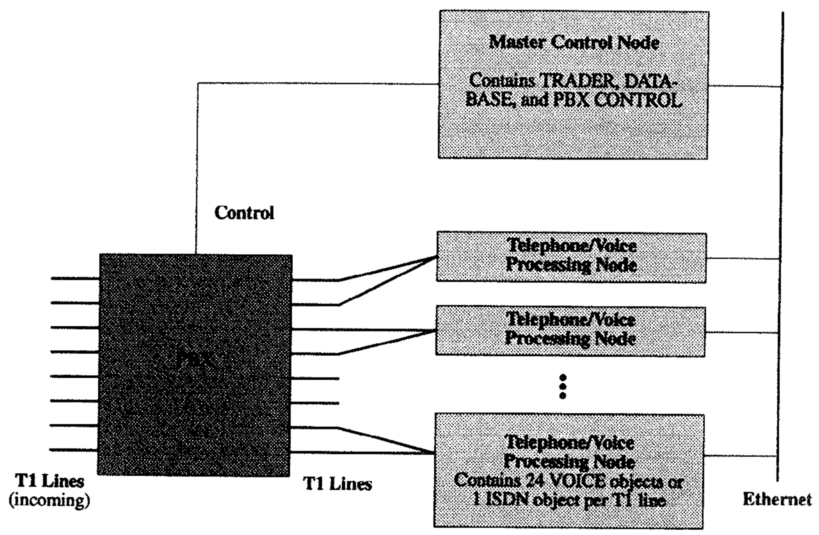
Master Control Node
Contains TRADER, DATA-BASE, and PBX CONTROL
Control
PBX
Telephone/Voice Processing Node
Telephone/Voice Processing Node
Telephone/Voice Processing Node
Contains 24 VOICE objects or 1 ISDN object per T1 line
T1 Lines (incoming)
T1 Lines
Ethernet

Figure 3. General hardware organization

disks, and off-the-shelf telephony hardware. In the original design, each node is an X86 box running Unix. The nodes are also connected via a switching network such as a digital PBX to the PSTN (public switched telephone network). The LAN carries all meta-data and the switching fabric carries the heavy data. The PBX may be dispensed with if the amount of data to be moved around the system is expected to be small or the LAN may be dispensed with if the main switching network has enough bandwidth (for instance, if it is an ATM switch).

One processing node acts as a dedicated host for the trader, the database, and other singular objects such as the switch controller. The rest of the processing nodes connect to the switch by T1 interfaces (digital telephone lines that carry 24 time-multiplexed individual channels). These processing nodes act as dedicated hosts for the application objects; each individual voice channel attaches to an application object.

An example of control flow. When a call comes in, the PBX sends a notification of the event over a serial control line. For most incoming lines, the appropriate request is the *welcome* service. The switch control object on the Master Control Node (MCN) notes the event and places the appropriate service request on the trader's queue. As soon as a registering object offers the *welcome* service, the trader hands off the service request and places a *transfer call* request onto the queue. The switch control object registers with the trader and picks up the transfer request. The request packet indicates which incoming line the call is on, as well as the application object that is handling the *welcome* service. The switch control object determines the appropriate port numbers and instructs the switch to bridge the call. When the application object detects ringing, indicating a bridged call, it goes off-line (picks up the phone) and begins the service.

Suppose the application object determines the caller wishes to leave a voicemail message. The application object sends the request for a *voicemail* service to the trader. The service request stays on the trader's queue until a registering application object registers the *voicemail* service. When this happens, the application object is given the request and the trader places a *transfer call* request onto the queue for the switch control object.

Decoupled application architecture

The services offered on the Telecom Services Architecture are decoupled in that a logical service is usual handled piecemeal by several different servers on different machines, as described in the previous section. A simple flow diagram for a typical application is shown in Fig 4. Though it seems like a single application to the caller, it is actually

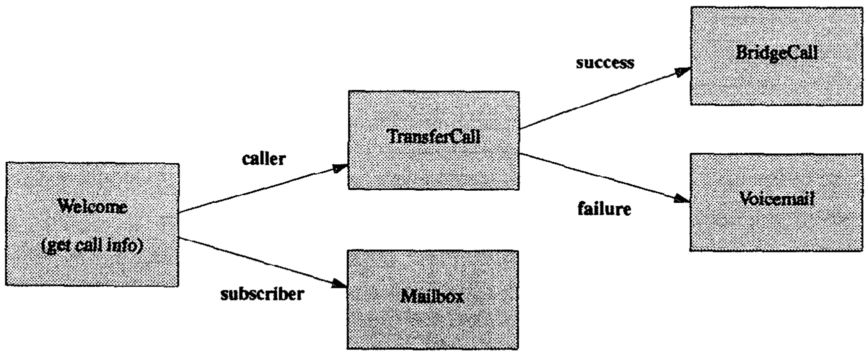

Figure 4. Simple decoupled application

composed of five services, *Welcome*, *TransferCall*, *BridgeCall*, *Voicemail*, and *Mailbox*. In the *welcome* service, the nature of the call is determined: it is either a caller trying to contact a subscriber (for example, someone calling in to the company to reach their spouse), or a subscriber calling in to listen to his/her messages. If it is a caller, the *welcome* service obtains the extension number of the person called and attempts to transfer the call to that person's line. The caller is put on hold at this point. A successful transfer occurs when the party called picks up his own phone, or picks up any other phone in the system and identifies himself to the system. At this point, the *bridgecall* service is invoked, which takes the caller off hold and transfers him to the subscriber. If the subscriber had not picked up the call, (the *failure* case) the caller would have been sent to the *voicemail* service. If the original caller had been a subscriber, the *welcome* service would have obtained his/her user ID, and would have transferred him to the *mailbox* service.

It is important to understand that each of these actions (represented by different boxes in the figure) represents a different portion of the logically monolithic application, but can be handled by a different server entirely. Each box is a potential decoupling point; at any of these points the caller can be put on hold and transferred to a completely different application object residing on a completely different processing node. The caller is unaware of the change. Thus, a phone call can get bounced all around the system in the course of handling the call. This would occur for load-balancing reasons; if a particular node is busy performing many I/O intensive actions, the call could be transferred to another node to handle a voicemail request. It would also occur when a system is upgraded; here, new software is being put on a live system without a disruption in service. As application objects finish their work, they can be told to stop handling more calls and to stop registering themselves with the trader. Once an application object is idle, it can be killed and a newer version restarted in its place. The new object will immediately begin registering with and accepting calls from the trader; from an external perspective there is no perceptible down-time.

Benefits of the trading function

The use of the trading function offers advantages that are important in a commercial product, including modularity of design, support for dynamic evolution of a running system, and support for a heterogeneous hardware configuration. The following features are derived directly from the use of distributed objects and the trading function:

- *Dynamic addition of new services and data types*. One can add a new application object to the system, replete with new services and data types, and it will immediately begin working in a running system. The services do not depend on anything except a server object to offer them and a client to request them. They are opaque to the trader and (unlike most RPC mechanisms) do not require any form of global configuration. This enormously simplifies system upgrades.

- *Indirect invocation of object methods.* An object invokes the methods of another object through the trader. This allows application objects (both clients and servers) to know as little as possible about the organization of the system, an advantage when upgrading systems. Further, it makes the request packet the focal point for communication between heterogeneous machine types, and thus simplifies support for a *heterogeneous configuration*. It also gives rise to the next two features: *decoupled application architecture,* and *asynchronous transfer of control.*
- *Modularity through a decoupled application architecture.* An application can easily be broken into smaller pieces, where a different application object handles each piece. In this way, a logically monolithic application can be broken into several pieces, each executed by a different application object on a different machine. Besides simplifying application-building, this allows performance trade-offs such as moving the call to the data instead of moving the data to the call; a logical thread of control (the interaction with a telephone call) can actually pass through many objects on many nodes.
- *Performance though asynchronous transfer of control.* Since services are invoked by handing a request to the trader, one object passes control to another in an asynchronous fashion. It is analogous to one-way RPC; the object handing state off to another need not block until the next object picks up the request packet. The object only blocks until the trader responds, and the request will only remain unserviced in the case of catastrophic failure (a result of it being a closed system). This non-blocking protocol frees application objects from guaranteeing that control is successfully passed and allows them to spend all of their time doing real work.
- *Soft real-time event response.* The system can respond in soft real-time to external events, and routinely handles 40,000 calls per hour. The trader places requests on a timeout queue, tagged with a timeout value and a timeout service; if the request remains on the queue for the length of time specified by the timeout value, the request becomes a different type, specified by the timeout service. This can be anything from a beep to a notification that the system is heavily loaded, to starting a daemon that ascertains whether the system is functioning properly.

Related Work

Two standards of distributed computing exist which outline similar functionality but do not suggest designs. CORBA [OMG93] from the Object Management Group describes the use of objects, including standard definition languages and methods for storing and retrieving the interfaces. CORBA allows objects to invoke methods indirectly through the Object Request Broker, which can translate between client and server protocols. ODP [ITU92, ITU94] from CCITT (now ITU, the International Telecommunication Union) describes the trading function in which servers export service offers to a trader and clients import those offers.

Implemented systems include RPC systems such as the Distributed Computing Environment from OSF [OSF91], the Information Bus [Oki93], and Birrell's Network Objects [Birrell93]. DCE assigns unique ids to service interfaces and requires a client to program for the interface and include the interface description at compile time. A client must know either a server's authentication id or a shared location in the CDS namespace before attempting to bind. The Information Bus allows more anonymous communications: servers "publish" data of various types, and clients "subscribe" to whatever data types they are interested in. A client receives all information on the network of the types to which the client subscribes. Birrell's network objects are similar to other distributed object systems such as Opal [Chase92] and Emerald [Jul88], but simplify the designs by restricting mobility and copy non-stationary objects over the network by "pickling". Objects export their methods under whatever name they desire, so it is possible for a client object to find a server object by a descriptive name.

References

[Bach86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[Birrell93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. "Network objects." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[Chase92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. "Lightweight shared objects in a 64-bit operating system." Technical Report 92-03-09, University of Washington, March 1992.

[Chin88] Roger S. Chin and Samuel T. Chanson. "Distributed object-based programming systems." *ACM Computing Surveys*, 31(3), March 1988.

[ITU92] ITU. *Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing*. International Telecommunication Union, 1992.

[ITU94] ITU. *Draft Recommendation X.9tr: ODP Trading Function*. International Telecommunication Union, 1994.

[Jul88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-grained mobility in the Emerald system." *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Oki93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. "The Information Bus–an architecture for extensible distributed systems." In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[OMG93] OMG. *The Common Object Request Broker: Architecture and Specification, Rev 1.2*. Object Management Group, December 1993. OMG Document Number 93-12-43.

[OSF91] OSF. *DCE Application Development Guide*. Open Software Foundation, 1991.