Introduction

The rise of networked workstations and fall of the centralized mainframe has been the most dramatic change in the last two decades of information technology. This shift has put more processing power in the hands of the end-user and distributed hardware resources throughout the enterprise. No longer the domain of raised floors and data centers, processing power now resides on desktops, workgroup servers, and minis. This shift first involved hardware; the current challenge is to develop the software infrastructure to make use of these now distributed resources.

As networks of computing resources have become prevalent, the concept of distributing related processing among multiple resources has become increasingly viable and desirable. Over the years, several methods have evolved to enable this distribution, ranging from simplistic data sharing to advanced systems supporting a multitude of services. This paper presents an overview of the means used to enable distribution of computing work, covering core concepts and popular implementations of those concepts. The objective is to educate the audience as to the technologies available and their strengths and weaknesses.

This white paper was developed by senior technical staff at QUOIN, a software development and consulting firm that specializes in distributed objects and business components.

The Common Foundation

Network Communication

Underlying all distributed computing architectures is the notion of communication between computers. Although basic, this shows how close, conceptually, some common distribution architectures are to their underlying communication facilities. The combination of hardware and system-level software that enables computers to communicate is frequently referred to as the transport layer. When several computers are connected to one another through a common transport layer, they can be considered a network of computers.

Much as a piece of information can be wrapped, addressed and sent through the postal service, networks generally operate on packets, which are analogous to the package you might send through the mail. Like the mail package, the network packet has ‘from’ and ‘to’ addresses and contains some information, such as a message. Also like the mail message, the receiver may or may not elect or be compelled to acknowledge receipt of the packet.

If either the mail or network message exceeds certain size limits, it may need to be broken up into separate parts and reassembled upon arrival at its destination. However, these physically separate packets can be treated as single logical packets. The transport layer, addressing semantics, packet sequencing, data formatting and a host of other defined components make up a communications protocol. These pre-defined protocols are what allow computer systems to properly interpret the packets received from other systems.

Synchronous and Asynchronous Transmission

Just like the mail package, the sender’s interest in the subsequent receipt of the packet and actions taken in response to it varies. There are cases where the sender isn’t concerned about when, or perhaps if, the packet arrived at its destination. There are other cases where the sender

wants confirmation that the packet arrived, but doesn't need such confirmation to continue its task. There are also cases where the sender cannot continue until it receives a response from the receiver.

Synchronous modes of operation are those where the sender needs a response from the receiver before it can continue. Modes of operation where the sender doesn't require a response from the receiver, at least not before it can continue, are considered asynchronous. This distinction is generally one of the primary factors in determining a given communication protocol's suitability to a given task.

Clients, Servers and Peers

The terms *Client/Server* and *Peer-to-Peer* have come to be associated with specific attributes of distributed computing. In fact, clients, servers and peers are just roles played by the participants in a communication protocol. These roles can change constantly within a communications session. Note that these participants are actually threads of execution, which may exist on the same system or even within the same process as the thread of execution with which they are communicating.

When a thread of execution opens a communication channel and waits for another thread to contact it, it can generally be considered a *server*. The thread that initiates the communication by contacting the server is generally considered a *client*. *Peer* is a general term used to refer to a thread that is able to act as both a client and a server.

APIs - Application Programming Interfaces

Core communications facilities are generally provided by Operating System (OS) and network requester APIs. These are groups of functions called by a program to accomplish the actual transmission and receipt of bytes of data between systems. In general, these low-level components provide limited abstraction of the underlying communication session, leaving communicants to provide all logical services such as addressing and data conversion.

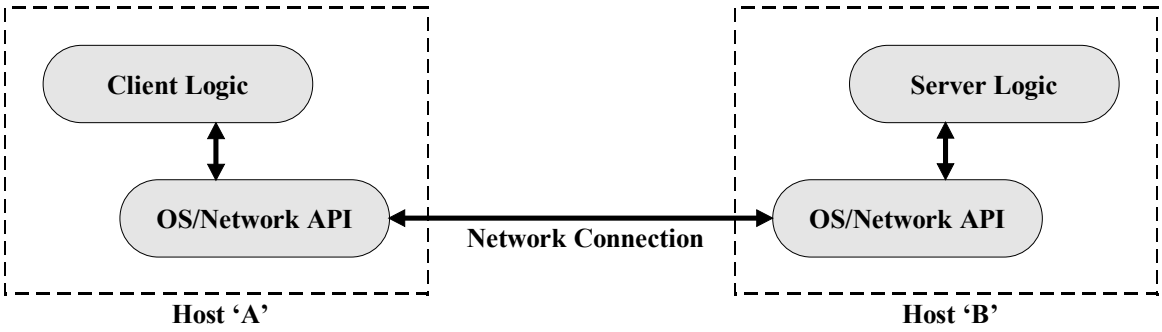

Figure 1: Direct Network Communication

Terminal Interfaces

The oldest form of distributed computing isn't generally recognized as such - logging in to a host system from a dumb terminal or terminal emulator running on a workstation. While not terribly elegant, this method has proven itself very effective.

A number of protocols exist for this type of communication, among them Telnet, rsh and rexec. The concept and implementation are simple; the client acts much like a directly connected terminal, but with some additional facilities allowing it to communicate through a remote connection. Each time a key is pressed, the client sends a packet containing a code identifying the key to the server. The server, in turn, sends back packets containing data to be displayed by the client. While generally limited to textual interfaces, server-based applications are able to use color and extended keys to enhance client interface functionality.

Among the benefits of terminal interfaces is the fact that in many cases they don't require the application to be written using a communication API at all, allowing programs which were written without consideration for distribution to be used remotely without modification.

Messages

Next in the evolution of distributed computing comes the concept of the message, a packet of data that is labeled with the information it contains. This allows an intermediate processing layer at the server to route the message, or the data it contains, to the receiver appropriate to it.

Messaging systems can operate quite naturally in an asynchronous architecture. Because message-based communication is well suited to intermediate routing, these features can be combined to provide a level of abstraction to the communication framework itself. Messages may be deposited into a queue by the server/router, from which they're retrieved and acted upon by one or more logical processors. These processors may not respond to the messages at all or may respond directly to the client. However, to maintain the abstraction, they can send a message back to the server, through another queue, to be routed back to the client.

Message-based architectures are also able to operate synchronously. Generally in this mode, the server/router passes the message to the processor, which passes a response back to the server to be returned to the client. Another hybrid mode is available, however, in which the server behaves asynchronously as described above and only the client behaves synchronously. This combination of behaviors allows the server to gain the efficiency of asynchronous operation while the client benefits from the procedural simplicity and safety of synchronous processing.

This basic architecture of client communication with a server which dispatches messages based on their content will be seen to underlie many of the following distribution models.

RPC - Remote Procedure Call

The concept behind RPCs is simple - to make what appears to be a normal procedure call from within a process and have its execution actually carried out within some other process, possibly on a remote system. Various implementations of RPC protocols have been developed with the common goal of reducing the complexity of communicating between processes through implementation hiding.

The core concept of RPC mechanisms is that of serializing function call data into a sequential stream and reconstructing it on the receiving end of the connection. This behavior takes place synchronously, mirroring the semantics of traditional procedural programming. The RPC client process makes a call to what appears to be a standard function, known as a *stub*. However, rather than executing locally, the parameters passed to the function are packaged and transmitted to a remote execution environment, where they are passed to the real implementation of the

function. Upon completion of the function's execution, its return value is serialized and passed back to the client stub, which returns it to the caller.

This behavior can be built upon the synchronous messages described previously.

Client/Server

As mentioned above, the terms *client* and *server* really refer to generic roles played by the participants in a communication session. However, the term *Client/Server* has become common in its usage describing a higher level, though conceptually similar, architecture. The common interpretation of the term denotes a system where significant processing is done on the *client*, which also submits operations to the *server* for execution. In this type of architecture, synchronous operation is generally assumed, wherein the client waits for confirmation that the operation has been carried out before proceeding.

Database Protocols

The X/Open **Call Level Interface** (CLI) [CLI 96] specification provides an interface to Relational Database Management Systems (RDBMS) using Structured Query Language (SQL) [SQL 92]. Microsoft's **Open Database Connectivity** (ODBC) API [ODBC 96] is the best known implementation of the CLI standard. Sun Microsystems' **Java Database Connectivity** (JDBC) API [JDBC 98] is a new implementation of the CLI standard specifically for Java applications.

CLI and the architecture it supports are perhaps the most commonly envisioned usage of client/server computing, allowing applications written using the standard to operate in most cases without regard for the database to which they're connected. The drawback to this lowest-common-denominator approach, of course, is that it does not provide access to some of the more advanced features that differentiate RDBMS products.

The API presented by the specification ranges in appearance from a thinly-veiled messaging interface to an RPC interface. The message-like components of the interface expose a hybrid synchronous/asynchronous mode of operation wherein initial results are returned synchronously while processing may continue asynchronously at the server. This allows the client to continue processing as soon as the server is able to provide an initial set of results; further results are queued by the server and returned to the client as they are requested. The RPC components are used for control purposes and operate in a strictly synchronous manner.

Middleware

Unlike client/server architectures where the client identifies and communicates directly with the server, the concept of 'middleware' assumes a functional layer between the client and server. This layer may provide services to the communicants such as location and alias resolution, authentication and transaction semantics. Other behaviors associated with middleware include time synchronization and translation between data formats.

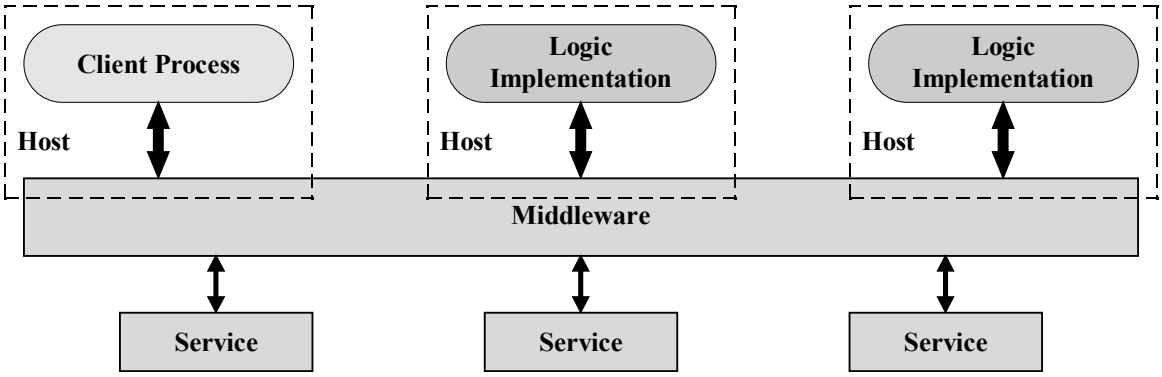

Figure 2: Middleware Architecture

This additional layer allows clients to interact with a generic abstraction of a server rather than with a specific host and/or process. Various services are provided through abstracted layers as well, blurring the distinction between services provided by the middleware and functionality added by servers. These abstractions allow applications to be developed to a standardized API without knowledge of the location or implementation of external functionality. This implementation hiding is one of the middleware model's strengths, although it makes it difficult for the client to determine what performance it can expect from any given logic implementation.

DCE – Distributed Computing Environment

OSF[1] DCE formalizes many of the concepts described here with a group of related specifications [DCE 96]. The DCE RPC specification is among the most widely implemented in the industry, providing consistent behavior across heterogeneous execution environments. The DCE architecture also defines thread, time, authentication and security, directory and naming services.

Because DCE is supported by an industry consortium comprised of many major operating system vendors, its standards enjoy widespread support across major computing platforms. DCE core functionality is included in almost every available variant of UNIX, and as PC operating systems have become more advanced, DCE core service support has become more common in them as well. These standards are based on procedural programming methods in the 'C' programming language, however, limiting their applicability to multilingual and object-oriented deployments.

Reliable Messaging

On their surface, reliable messaging architectures such as IBM's MQSeries and Microsoft's MSMQ appear much like the queued message frameworks described earlier. Beneath this surface, however, they differ greatly in their implementation.

In order to provide reliable delivery of asynchronous messages, a 'store and forward' model is used wherein a message to be sent by a process is passed synchronously to a middleware layer that stores the message, and any addressing information it may contain, to a persistent storage mechanism before returning control to the sending process. Once the message has been stored in this manner, the middleware can use a variety of methods to try to get the message to its intended recipient, while the sender continues its processing.

[1] The Open Software Foundation (OSF) has been renamed to The Open Group (OG), however DCE is trademarked as "OSF DCE".

The reliability of the architecture comes from the concept that the current holder of the message does not destroy its persistent copy of the message until it has received confirmation from a subsequent receiver that the message has been safely stored by it. Because each link in the communication chain stores the message until it knows it has been forwarded successfully, the original sender can proceed with its processing assured that its message will get through to its destination. Because of the asynchronous nature of this architecture, the sender must request confirmation of receipt of the message (or confirmation must be a specified action upon receipt of the particular message) if it needs to know when the message arrived or other details of its handling.

Distributed Objects

Object distribution architectures build upon the middleware concept by encapsulating data within functional interfaces to objects. Like well-designed procedural APIs, implementation details are hidden from the user of the object. Unlike traditional APIs, however, object architectures limit access to the invocation of methods defined for the object. Furthermore, methods are invoked on the objects indirectly, via references to the objects, eliminating the need for local instances of the objects.

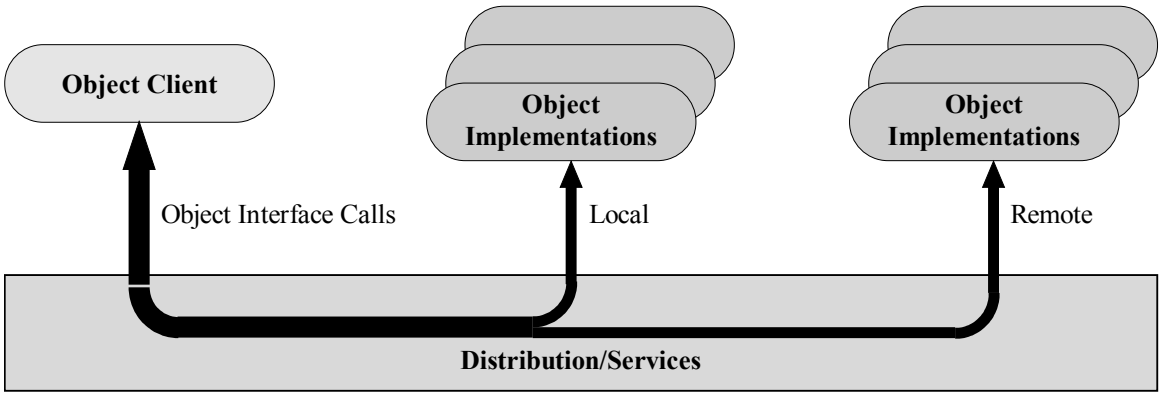

Figure 3: Distributed Object Architecture

This near-complete implementation hiding allows distributed object architectures to support location, platform, and programming language transparency. Such transparency is not without its costs, however, which has prompted the designers of some distributed object architectures to forego some neutrality in exchange for perceived improvements in performance, applicability to specific tasks and/or ease of use.

Java RMI - Remote Method Invocation

Sun's Java, while relatively new to the computing industry, has gained a great deal of acceptance due to its platform neutrality, safety and object-oriented design. Java was designed from the ground up as a complete execution environment, rather than just another programming language, therefore it is able to provide a consistent and abstract interface regardless of the underlying platform.

This platform independence is accomplished through the use of a Java Virtual Machine (JVM) that emulates a computing platform itself. The JVM is provided for each actual combination of hardware and operating system upon which Java is to run. Because Java programs all appear, at

the application level, to be running on the same computing platform, communication between Java applications is made significantly easier.

Java RMI [RMI 97] provides a language-specific architecture allowing Java-to-Java distributed applications to be built easily. The main advantage to using Java RMI when designing a pure Java distributed system is that the Java object model can be taken advantage of whenever possible. Of course, this precludes using Java RMI in multilingual environments. Java's inherent platform independence, however, still allows deployment in heterogeneous environments.

DCOM – Distributed Component Object Model

Microsoft's core object distribution protocol is DCOM [DCOM 98], an extension of Microsoft's COM [COM 95] integration architecture, permitting interaction between objects executing on separate hosts in a network.

COM began as a way to let client programs link to object implementations dynamically; i.e. at run-time, incorporating them into a single address space. The implementations were packaged in Dynamic Link Libraries (DLLs). COM is essentially an integration scheme, adopting the structure of C++ virtual function tables for binary compatibility. These virtual function tables, commonly known as 'vtables', consist of a table of function addresses (in C language terms) or the equivalent. COM interfaces are presented to clients as pointers to vtables, thereby hiding the details of the implementation. This makes COM binaries independently replaceable, as long as they implement the same interfaces as their predecessors.

By taking the path of least resistance and adopting the C++ vtable model of binary integration, COM achieved both replaceable binaries and the efficiency of in-process method invocation, equivalent to a C++ virtual function call. These benefits naturally come at some cost. Because an interface is presented as a pointer to a single vtable, interfaces cannot be defined using multiple inheritance. This would imply multiple vtables and therefore multiple pointers per interface. In addition, because there is no intermediary service to dispatch function calls, any programming language besides C++ must go through some contortions to work with COM.

In order to address the rising need for distribution of objects across multiple hosts (i.e. multiple physical address spaces), Microsoft developed DCOM as an extension to COM. As an extension rather than a separate architecture, DCOM inserts a stub interface between the calling application and the actual implementation of that interface. In this manner the architecture strongly resembles an RPC-based model, although the implementation is still based on a binary integration scheme, rather than a more abstract model.

COM+ is the recently announced successor to COM-based models incorporating a new generation of technology. COM+ extends COM with multiple inheritance, a new runtime, and language extensions that will make it easier to build COM objects in a variety of programming languages. As of this writing, a COM+ specification has not been published, making it difficult to assess its architecture and any tradeoffs that may have been (or will be) made to facilitate its implementation.

CORBA – Common Object Request Broker Architecture

CORBA is a standard maintained by the Object Management Group (OMG) for the distribution of objects across heterogeneous networks. Designed as a platform-neutral infrastructure for inter-object communication, it has gained widespread acceptance.

The OMG is a consortium of more than 760 companies formed to create a standard architecture for distributed object computing. The goal of the OMG[2] is to combine object and distributed systems technologies into an interoperability architecture that supported integration of existing and future computing platforms. The result of that effort is the Object Management Architecture (OMA); CORBA specifies the Object Request Broker (ORB) underlying the OMA. The ORB provides the base architecture [CORBA 97] as well as a number of services [COSS 97] such as security, transactions and messaging.

CORBA allows applications to use a common interface, defined in an Interface Definition Language (IDL), across multiple platforms and development tools. OMG IDL is designed to be platform and language-neutral; data and call format conversions are handled transparently by the ORB. All interfaces to CORBA objects, and the data types used in those interfaces, are specified in IDL. This common definition allows applications to operate on objects without concern for the manner in which the object is implemented.

As viewed by the client, a CORBA object is entirely opaque, in that the object's implementation and location are unknown to the application using it. Generally, a CORBA client will know only how to find or create the objects it needs through interfaces to well known objects such as query mechanisms and factories. It is likely that the client will not know where or how even these well known objects are implemented, but will instead be able to locate them by name only through the CORBA Naming Service [COSS 97].

In the case of any of these CORBA objects the client knows only what the object's public interface is and can therefore access the object's functionality. CORBA also provides some capabilities for runtime object interface identification and invocation through its Interface Repository (IR) and Dynamic Invocation Interface (DII). While these have the potential to allow (almost) complete runtime configuration of access to CORBA objects, in practice there may be very few cases where such capabilities are actually workable due to semantic issues.

Conclusions

Many of the above concepts are shared across distribution architectures, and many architectures are built upon each other.

Some technologies, however old, still offer compelling reasons to use them. Certainly, no mainframe or UNIX system administrator would be willing to give up text-mode terminal interfaces for remote administration over dial-up connections. Similarly, implementing high-volume or low-overhead communications is frequently best done using low-level operating system and network interfaces.

However, where ease and cost of deployment are larger factors, standardized architectures are generally a good choice. In this case, the architectures can be divided between asynchronous

[2] Microsoft participates, but does not submit technology to the OMG. Instead, Microsoft promotes its own Windows-centric Distributed Component Object Model (DCOM).

and synchronous, with these most commonly being further divided between procedural and object-oriented methods.

For asynchronous communication, message-based architectures will frequently be most appropriate. In cases where assured delivery[3] is required, reliable store-and-forward messaging middleware is usually the answer. In other cases, the additional overhead of reliable message delivery may not be necessary, especially when the general reliability of current computing platforms is considered.

Where synchronous, procedural programming is used, DCE RPC will generally be a good choice. This proven, widely available framework offers a C language interface that will be reasonably easy to use in C and C++ applications. For systems written in other languages or where DCE RPC support is incomplete, synchronous messaging may be an appropriate solution.

Last, but not least, comes the subject of object-oriented distribution. Certainly, of the three frameworks discussed, CORBA provides the greatest flexibility with its language and platform neutrality. There are, of course, some costs associated with this neutrality, both in deployment and runtime overhead. These costs may be reduced by design trade-offs. Of course, a less flexible alternative may become much more costly if it doesn't support some future requirement.

One of the biggest factors in favor of Microsoft's COM/DCOM solutions is their installed base - virtually every PC running Windows has some level of COM support built in. Most 32-bit versions of Windows have DCOM support as well, providing a compelling argument for its use in Windows-only environments. Nearly all Windows development tools provide fairly easy to use wrappers allowing integration of COM components into applications they generate as well.

The case for using Java RMI is simply that it's so easy to do. Because it only supports Java objects it fits directly into the Java model with minimal impact on development resources. However, since CORBA support is being built into the Java environment, RMI doesn't have the advantage of being assured a larger installed base. Regardless, for pure Java distributed systems, the ease with which RMI-based systems can be deployed is compelling.

Proponents of all of these architectures can be found easily and in great numbers, and it is impossible to state that one alone is best for all distributed systems. It should also be noted that while many of the most widely used implementations of these architectures have been addressed here, there are many more which have not. In particular, reliable messaging is a rapidly growing field.

An extensive comparison of COM and CORBA can be found in [Quoin 98]. Comparisons of, and documentation for, all of the mentioned architectures can be found in a number of locations on the World Wide Web, as well.

Distributed computing is becoming more prevalent every day. The architectures discussed here, and many others, are able to solve a range of problems that could not even be considered a few years ago. We recommend careful identification of present and future needs, as well as current competencies, before deciding to deploy any of them.

[3] Of course, without suitable precautions against component failures, even the most reliable architectures may be exposed to data loss or corruption.

References

[CLI 96] "Information Technology - Database Languages - SQL - Call-Level Interface" ANSI/ISO/IEC 9075-3-1996 (ITI/NCITS October 1996)

[COM 95] "The Component Object Model Specification" (Microsoft Corporation, Digital Equipment Corporation, October 1995)

[CORBA 97] "The Common Object Request Broker: Architecture and Specification, Version 2.1" (Object Management Group, et al, August 1997)

[COSS 97] "CORBAservices: Common Object Services Specification" (Object Management Group, et al, July 1997)

[DCE 96] R. Salz; "DCE 1.2 Contents Overview", Open Group RFC 63.3 (The Open Group, October 1996)

[DCOM 98] N. Brown, C. Kindel; "Distributed Component Object Model Protocol" (Microsoft Corporation, January, 1998)

[JDBC 98] G. Hamilton, R. Cattell; "JDBC™: A Java SQL" (Sun Microsystems, Inc., February 1998)

[ODBC 96] "ODBC 3.0 Programmer's Reference" (Microsoft Corporation, October 1996)

[RMI 97] "Java Remote Method Invocation" (Sun Microsystems, Inc., December 1997)

[SQL 92] "Information Technology - Database Languages - SQL", ISO/IEC 9075:1992 (ISO, November 1992), ANSI X3.135-1992 (ANSI, October 1992)

[Quoin 98] O. Tallman, J. Kain; "COM versus CORBA: A Decision Framework" (Quoin Inc., January 1998)