

Network Applications: Open *and* Shut

Stephen Mohr, Omicron Consulting

Introduction

The rapid emergence of Internet-enabled and web-delivered applications has given rise to a new form of distributed computing: the network application, sometimes known as the Internet application. An exciting development, the network application model presents programmers with a dilemma. How should designers balance the open standards of the Internet with the efficiency and performance of platform-proprietary technologies? Some will slavishly stick to open standards, while others will stick with the native techniques they have worked so hard to master. Each is partially correct. Open standards and the closed, shut standards of a particular platform have their proper place, and the two can coexist harmoniously.

This presentation examines the challenges of network applications, the roles of open and shut technologies, and provides guidance for selecting the right balance between the two. It builds on my presentation "Building Cooperative Network Applications" at the 1999 Wrox Developer's Conference, and my book "Designing Distributed Applications" (Wrox Press, ISBN 1861002270). The feedback from these made it clear to me that there is considerable confusion among developers as to when to use technologies that are tightly tied to a given platform, and when to use open protocols and standards. For some this is a matter of religion. Programmers tend to move from one extreme or another. I'm going to show there is a role for both and that the two can even coexist within a single application.

We'll start by defining what I consider a network application to be, and then describe the unique challenges faced by network applications. Next, you'll be given an idea of the possible current technologies/solutions that exist in the distributed applications space, followed by some architectural guidance. I'll make the case for using proprietary methods within borders of platform and management that you control, and using open, platform neutral methods when you need to cross those borders. I'll present a layered model for designing network applications and thinking about implementation technologies.



Stephen Mohr

Stephen Mohr is a software systems architect with Omicron Consulting, Philadelphia, USA. He has more than ten years' experience working with a variety of platforms and component technologies. His research interests include distributed computing and artificial intelligence. Stephen holds BS and MS degrees in computer science from Rensselaer Polytechnic Institute.

Network Applications

Network applications are those applications and systems that require the services of a computer network for their proper functioning. They are inherently distributed applications. Although they could theoretically be deployed on a single machine, there is no practical advantage in doing so. They enlist remote services in the accomplishment of tasks, as these services are needed. Network applications go beyond the client-server model and are nearer to the n-tier model of application programming. A key distinction, however, is the transient nature of the services they enlist and the ad hoc and dynamic way in which these services are enlisted. They are characterized by the cooperative use of resources not under the control of the client requesting those services, and frequently not even under the control of the parties who designed and implemented the client.

The Model

Network applications consist of three distinct parts: clients, server-hosted services, and exchanges between services. Clients provide the user interface and some minimal, local capabilities. Network applications typically use more servers than you may be accustomed to in conventional client-server or even n-tier practice. Some will be outside your control and you will obviously have no say in their hosting. Even though a service might be well suited to co-location with another service implementation, the fact that it is controlled by another organization will require that you access their server. You can use them, but you have no influence over their administration, maintenance, or evolution. Servers span all sorts of dissimilar platforms.

Exchanges consist of the transfer of structured data on an ad hoc basis. Since network applications face higher latency and less central control than local applications, long transactions – of the order of minutes or hours – may be required. Exchanges must bridge not only dissimilar platforms, but also different domains of security and trust.

Clients

HTML presentation is quickly becoming the model for client interfaces. Network application clients use this visual metaphor for orchestrating services on behalf of users and for the presentation of results and options. The hardware hosting such clients is breaking open. In addition to browsers on PCs, we will soon need to account for Personal Digital Assistants, cell phones, television set-top boxes and games consoles. These devices offer differing capabilities for display and input, as well as having differing bandwidth and processing power. While this results in certain differences in presentation, the basic model of pages with hyperlinks holds for all.

Servers

Network application servers are multi-tier in nature. Unlike the dedicated servers in the client-server model, some servers will be managed with little or no knowledge of all the uses of their services. The hosting platforms don't need to know – and shouldn't care – about the nature of the applications that use their services. They may offer differing technologies for doing distributed computing, and it is the task of network applications and their architects to overcome these differences.

The Challenges of Network Applications

Network applications face challenges unknown to local applications or even many client-server applications. They are prone to connectivity outages due to the wide span of the network and the number of cooperating nodes. The location of needed services (whose implementations may come, go, and evolve) is something that must be dealt with continually. Unlike the n-tier

applications of even a few years ago, network applications are expected to cross different hardware platforms and operating systems. Security for such widespread and cooperatively managed applications is a continuing challenge. The question of what node or nodes will maintain the overall state of an application, and how this will be done to everyone's satisfaction, also poses difficulties to be overcome.

We're Not the First

Of course, we are not the first to deal with these issues. The pioneers of distributed computing had to consider these challenges to some degree, although they generally were concerned with centrally managed applications using services in a well-known and prescribed manner.

Socket programming leaves virtually all of these challenges to your own protocol. It provides minimal services for the translation of data formats between platforms, and the Domain Name System (DNS) is the only service for dealing with the naming and location of services.

Remote Procedure Calls (RPC) and component-based technologies like DCOM and CORBA go further. They provide a minimal formal model for all applications that use these technologies. They also deal with data format translation through a canonical format for "on-the-wire" data, thereby avoiding the problem of many-to-many data translations.

LDAP, the Lightweight Directory Access Protocol, arose to integrate directories. Directories provide better naming services than DNS and the like by allowing administrators to describe the properties of networks and services, in addition to their name and location. LDAP, unfortunately, comes very close to being a proprietary protocol. It is specified in terms of programming language-specific bindings, and some "standard" implementations don't interoperate well with others.

Messaging middleware offers something unique: asynchronous interaction between platforms. This is a key enabler for dealing with many of the challenges of network applications. They generally make no provision for, or constraints on, the data passed, leaving those issues to the using applications.

What the Pioneers Left Out

If the pioneers had solved everything, this would be an extremely short presentation. Security is a glaring example of something they missed. Distributed systems are either wide open or rely on proprietary security protocols. The Secure Sockets Layer (SSL) on HTTP comes the closest to being an open security standard. Security will always be a problem for network applications, largely because the inherent environment and characteristics of network applications work counter to the desires and practices of security specialists.

Naming and location runs a close second to security in the race for least-well-addressed challenge of network applications. This task requires more planning than many LAN administrators want to put into their networks and, until recently, most applications were not developed to look at non-local naming sources.

The pioneers also ignored the challenges of the Internet except to the extent that they developed the communications protocols we use on the Web, but this wasn't due to lack of talent. The pioneering generation simply didn't foresee that the Internet would be used for running programs. The pioneering distributed technologies – RPCs, COM, etc. – were designed to be used in predetermined schemes between partners that are closely coordinated.

Architectural Solution

My architectural solution starts with the five principles I enunciated in "Designing Distributed Applications":

- Applications will be built from coarse-grained services that implement a unit of processing larger than components
- Services will be discovered by querying directories
- Services will be provided through the exchange of self-describing data
- Services will be enlisted on a transient basis
- Services must support extension and degrade gracefully

These principles are intended to be sufficiently concise so as to find wide adoption, and they do not make formal and concrete suggestions as to their implementation. That is left for the architects and programmers that use them.

Something that should be provided to architects, however, is a guideline for deciding between the proprietary technologies of a particular platform and open standards and protocols. The key is the boundary outside which you have no control over decision-making. Within that boundary, use native methods. They generally exhibit high performance and more robust capabilities. Outside, the boundary, you must accept the overhead of open methods. In short – use technology that introduces the least amount of overhead you need to get the job done reliably.

Closed Methods

Native methods offer more features – contrast COM and HTTP. Their data representations are binary and are therefore more compact than open forms. Closed methods usually have better integration with the host platform and other applications. On Windows, nearly all services use COM, and many use the IStream interface and its variants to persist objects.

Novice programmers – at least, novices to distributed computing – often have a tendency to use the technique that is receiving the most hype at the moment. Network applications are very challenging to implement, so there is a temptation to look to professed experts and slavishly follow their guidance. Resist this! If you don't need what open methods provide, don't use them. Closed methods are as "good" as open methods. In fact, you can often combine the two approaches in various parts of your network application.

Open Methods

The broad view of open solutions in my architecture runs like this:

- XML for data
- Open communications protocols for exchanges between nodes
- Use the least common denominator approach if you are in doubt

Part of the popularity of XML is due to its simplicity, portability, and expressive power. A relatively simple scheme, XML is nonetheless able to express most forms of data and do so on all common computing platforms. Because it is simply text, it is well suited to transmission via most common Internet protocols. XML and its related standards do have some problems, mainly in the areas of communicating the structure and syntax of XML documents (their vocabularies), versioning vocabularies, and accommodating extensions in a way that is recognizable to all parties.

Open communications is just a matter of moving data. Sophisticated state management and the like is an application responsibility, so don't get clever with standard protocols. Many

people have promoted techniques for moving binary data peculiar to their applications over HTTP or within XML. These are proprietary methods that should be avoided.

Open *AND* Shut

Unless you are still in school, no one is going to grade your application against a formal standard or theory. Like most of us, you are struggling to deliver working software, preferably using code of which you can be proud. Your application's requirements should drive the technology. Architects and programmers are paid to exercise good judgment and provide the benefit of their experience. Use the proprietary methods for local, small-scale tasks, and reserve open methods for heavy-duty exchanges between platforms.

Open and Closed Examples

The samples presented shortly show a simple COM component used in an HTML client-side script ASP, and then the same component wrapped using the Simple Object Access Protocol (SOAP). These examples are noteworthy for several reasons. First, they show that the open method consists largely of overhead needed to address cross-platform, outside the boundary problems. Next, they show the value of using components to encapsulate the implementation details of open standards on a particular platform. I used the Microsoft SOAP toolkit preview, which offers the ROPE component for building and parsing SOAP messages (`rope.dll` and `soapisapi.dll` are the required DLLs). As a result, the code presented exhibits little or no XML handling code, although SOAP uses XML extensively.

One intricacy is the task of implementing the SOAP service. The COM interface could not be changed as I was deliberately using a legacy component. One of the methods involves passing parameters by reference. This method could not be used in the ASP implementation, as VBScript will not properly pass a variant by reference to a C++ COM component. Consequently, the HTML client using the ASP listener only features one of the methods. Moving to VB for the client and using ISAPI as the listener allowed me to expose and use both methods properly.

Technologies in Layers

It is useful to have a conceptual model for thinking about the challenges and solutions of network applications. I present a seven-layer model (any resemblance to the OSI Seven Layer Network Model is purely coincidental!) that separates the implementing technologies of network applications into layers, according to the complexity of the problem they address. The layers, presented in descending order of sophistication (along with an overview of where various of the technologies discussed in this talk fit into the picture), are:

Layer	Associated Technologies
Systems	RosettaNet LDAP
Applications	
Services	Digital Signatures PKI
Extended Communications	BizTalk Framework SOAP
Simple Communications	SMTP HTTP
Meta Data	XML Schema DTD
Data	ADO Recordset HTML XML

You can use this model to sort out contending open technologies as well as using it to design your own network applications. Starting at the lowest level, we'll now consider each in turn.

Data

This layer refers to the data that leaves one node of the application or system and is exchanged with another application or service. Within COM and current Windows-hosted scripting languages, the variant datatype is one approach to this problem. Its polymorphic nature, combined with type coercion, makes it relatively easy to exchange simple and small amounts of data between COM-based applications. The ADO recordset is another example. It uses types native to the platform, exhibits sufficient flexibility to represent a wide variety of information, and is well supported throughout Windows.

An open solution is something else again. Only a least common denominator approach will work. Text is nearly universal, so it is natural to consider it as a means of passing data around the components of a network application. XML is a fine example of a technology being in the right place at the right time. It has enough formality and rigor to support a wide range of applications, programmers and applications are developing a body of experience working with it, and the considerable market hype behind it has pushed it to near-universal acceptance in a surprisingly short period.

Although XML is self-describing to some degree, a certain amount of metadata must be shared. Some applications get by with linking or embedding XML schemas. Another effort, the idea of central repositories of metadata concerning shared vocabularies, offers uncertain promise and has proved unsuccessful in the past. In theory, it can provide a powerful and accessible way to share and reuse metadata as it is needed. It is far too early, however, to tell whether such repositories will become an accepted part of networking practice or whether the familiar pattern of redundancy and reinvention repeats itself.

Metadata

Metadata provides details of information that describes how XML vocabularies are put together, what items represent, and what constraints are placed on the data represented by a given class of XML document. The W3C has a Metadata Activity of long standing, but it has been slow to

make its way toward issuing a schema recommendation. As a result, vendors have put a multitude of competing schema formats forward. We shall have to beat these back when a formal schema recommendation is finally issued.

XML schemas promise to replace DTDs (Document Type Definitions) in a form that is better suited to applications than publishing. They embrace, even demand, namespaces, permitting clear segmentation of data formats and types. Schemas, though, can be thought of as describing a vocabulary's grammar rather than its meaning. For meaning, we shall either have to wait for the evolution and acceptance of something more powerful, for example, the Resource Description Framework (RDF), or accept that a certain amount of metadata must be implicit at the system level.

Simple Communications

It isn't distributed computing if data doesn't move from one node to another in the network. Whether it is between components on a single machine or between applications and services across the public Internet, network applications require communications. This layer addresses the low level issues of moving data between one point and another. It is characterized by the open protocols of internetworking, such as HTTP, SMTP, etc. Reliability is dealt with only insofar as the implementing protocol guarantees packet delivery – for example, TCP. Protocols at this level are stateless and make no promises about message delivery. No provision is made for communicating the information that an application or service needs to recover state information or track workflow through systems.

Extended Communications

Complex applications and systems require the things we've left out of the preceding layer. Having solved the problem of getting data from point A to point B, we need to address the problem of communicating state. Data must arrive, and must arrive exactly once. Having gotten there, the receiving service should be able to determine what workflow process is affected by this data and have enough information to recreate the existing state of the workflow. Network applications also require the concept of long transactions. Losses of connectivity, latency over the public Internet, and human nature in the loop all mean that some steps in a given workflow can require minutes, hours, or days to complete.

Services

Services are the smallest level of cross-border interaction. The overhead of the platform independence approach precludes direct component-to-component interaction. Instead, a service is built of software components and implements one well-focused unit of functionality. A service is accessible across platform and security boundaries.

Web services and protocols are enjoying too much popularity right now, due, in some part, to the naïve thought that port 80 (the default configuration for HTTP) is open in nearly all firewalls. Once mission critical services are made available in network applications, HTTP-based services will be secured in some fashion. Security must be addressed at this level.

Services are stateless and therefore are well suited to high transaction volumes.

Applications

This is the lowest layer that is visible to the human user. Applications are composed of services to accomplish some non-trivial task for a user. Usually, this is a collection of related functions that cannot be adequately addressed by services alone. As services are stateless, some body of code on or near the client contains knowledge of what services are required in what situations, as well as the state of the application.

The application layer adds a navigation and user interface scheme to the features offered by the services enlisted by the application. Application state is usually required at this level. The client portion of the application usually accomplishes this, but dedicated data persistence on one or more servers may be required in complex applications.

Systems

Network distributed systems are built by structuring the flow of activity through multiple applications and services. This is an exciting new area for programmers as it represents the entire business, the integration of which has generally been left to manual processes or ad hoc solutions. Systems require:

- Long transactions
- Semantics for dealing with and specifying concurrency
- Decision points and dynamic activity flow
- Data persistence on dedicated servers

Application integration frameworks address these requirements in several ways. RosettaNet, for example, takes an authoritative stance, prescribing universal workflows for everyone participating in a class of applications. BizTalk Server, Microsoft's implementation of BizTalk Framework, offers tools for specifying workflow but does not mandate their use for all. This is an ad hoc approach. The right balance of formality and flexibility has yet to be determined.

Some Technologies

The seven layers of this model do not directly address implementation. Here are some of the technologies you may wish to consider. You may wish to reference the seven-layer diagram to see where they fit into our model. Be aware that this is not an exhaustive list – you are charged with following technology developments yourself!

HTML

HTML has many, many faults, both in theory and in practice. On the other hand, HTML works and is supported virtually everywhere. It provides a data driven way to specify user interfaces. This makes it ideal for the client-side of network applications. The challenge for network applications using HTML is to stick to standard forms and account for the variety of devices that will seek to be clients of our applications. One worrisome development is the Wireless Application Protocol's WAP Markup Language (WML). An XML vocabulary, it provides a stripped counterpart to HTML for small devices such as cell phones that may also have limited data rates. If third generation cellular and larger displays on cell phones do not displace WML, we may be faced with a split between HTML and WML.

XML and Friends

The reigning champion of data exchange on the Web, XML has many virtues that we've mentioned briefly in this presentation. What may be more important than its simplicity is the wealth of supporting standards (e.g., XSLT, XPath) that XML is built on. Although these are quickly adding complexity to what was a very simple domain, they provide valuable services that may be leveraged to help implement the various layers of the network application model.

SOAP

SOAP, which relies heavily on XML, is conceptually simple and attractive to developers. SOAP is an XML-based way of providing RPC-like interfaces over Internet protocols like HTTP. A joint effort of Microsoft, DevelopMentor, UserLand, IBM, and Lotus, it is fast on its way to becoming a de facto standard for web-related application architectures. Two cautions are in order,

however. SOAP is a work in progress. It has interoperability problems, and some developers are showing a tendency to advocate adding features to support higher functions (that is, higher layers in our model) to SOAP itself. I believe SOAP is better suited to supporting the Simple Communications layer. Although it relies on other protocols (HTTP for example) at this level, it provides no guarantee of state or reliable delivery. Second, the hype behind SOAP is rushing deployment of implementations, leading to interoperability problems between the major implementations available at present.

BizTalk Framework

BizTalk Framework is Microsoft's vision for an open approach to the Extended Communications layer. It is an XML vocabulary that supports state, long transactions, and routing. While it does not mandate a particular platform or implementation, it is closely tied to Microsoft and may therefore fail to win acceptance in the wider development community. It is worth studying the BizTalk Framework specification, however, as an example of one approach to the problems of the Extended Communications layer.

Digital Signatures and PKI

Cryptographic solutions to authentication and security for network applications necessarily rely on public key infrastructures. This is a large and arcane field, but one that is absolutely essential to securing network applications. While there is general agreement on algorithms and protocols, there is not yet a single solution for handling security in the sort of architecture we envision.

LDAP

Directories that support LDAP offer a powerful mechanism for handling the problems of naming and location. There are also possibilities for conveying or linking to metadata. LDAP straddles the open-shut border, however. I encourage you to use LDAP directories within your borders, but it remains to be seen whether LDAP is a viable solution for crossing organizational boundaries.

Directories and Naming: a Call to Action

Distributed applications inherently require services for naming resources and pointing to the location of their implementation. Unfortunately, this leads to a problem when we try to devise a unified architecture for network applications.

The Problem

Because all distributed technologies need this function, most of them are devising or have devised their own answer to this problem. This leads to conflict, overlap, and massive amounts of reinvention, creating further interoperability issues. This problem cries out for a unified approach and discussion within the developer community.

SOAP Description Language as an Example

SOAP needs to convey metadata that describes the features of a given service. SOAP also needs to point to the location of the service itself. One approach to this problem is the SOAP Description Language (SDL). This XML vocabulary is not part of the SOAP 1.1 specification. It is Microsoft's approach to conveying the metadata and location of a SOAP service. It is useful, but not yet adopted by the SOAP community as a whole. Beyond fracturing SOAP, SDL is an example of reinventing and re-addressing the problem of naming and location within a specific technology.

What's Left?

The technologies addressed so far provide solutions to many of the problems of network applications. There is one glaring omission, though: the System layer of our model is not addressed. The entire issue of workflow has not been solved. This is an exciting area that is still being explored in the products of individual vendors and the proposed standards of web-messaging consortia. One approach to this issue is illustrated in Microsoft BizTalk Server.

BizTalk Server: An Implementation

BizTalk Server's orchestration function is how individual instances of application integration are combined to specify a complex workflow sequence. It is based on graph theory, so it possesses mathematical rigor. A designer uses a Visio-based design tool to draw the desired workflow. The tool has constructs for indicating decision points and concurrency. The designer may also specify ports, points at which a message must be exchanged with an application. When the diagram is complete, the tool outputs an XLANG document. XLANG is an XML vocabulary developed for the tool. BizTalk Server uses the document at runtime to determine what actions to take in response to the arrival of messages. BizTalk Server maintains the state of the overall workflow and uses its data format translation capabilities and standard protocols to move messages between applications.

Summary

I've provided general architectural guidance for a very challenging area of computing. You have likely noticed that this field is still under construction, so there is ample room to express your own creativity. In parting, I pass along the following guidance.

Use the five principles to guide your thinking about the general architecture of your network applications. At any particular point, choose the minimum set of technology needed to accomplish the functions of that point. Proprietary solutions should not be discarded, but rather used wisely within your boundaries of control. Use the layered model I propose to guide your thinking about implementations. As you delve deeper into your architecture, think in terms of services. Keep them focused, test them thoroughly, and then use them to build more complex services and applications. Finally, stay abreast of the progress on open standards development. These are the crucial tools for crossing borders in network applications. As you gain experience, do not hesitate to become involved in the newsgroups and mailing lists that debate these standards!