



The Next Generation of Internet Applications

Table of Contents

| | |
|--|-----------|
| Purpose..... | 2 |
| The Evolution of the Internet..... | 2 |
| <i>In the beginning – Academic data exchange</i> | <i>2</i> |
| <i>The explosion of the Web - Content presentment.....</i> | <i>2</i> |
| <i>The maturing Internet – Applications and interactivity.....</i> | <i>2</i> |
| The Evolution of Applications..... | 3 |
| <i>In the beginning – Character-based interfaces.....</i> | <i>3</i> |
| <i>Applications become mainstream – Graphical user interfaces.....</i> | <i>3</i> |
| <i>Connected applications – Client-server computing.....</i> | <i>3</i> |
| The First Generation of Internet Applications | 4 |
| <i>The birth of the modern thin client</i> | <i>4</i> |
| The pervasiveness of the Web | 4 |
| Centralized application maintenance and hosting | 4 |
| Small client-side footprint | 4 |
| Client-side security | 4 |
| Learning curve of the Win32 API | 4 |
| <i>Problems with Web-based thin clients</i> | <i>5</i> |
| Poor programming model..... | 5 |
| Inability to represent major user interface paradigms..... | 5 |
| Slow response time..... | 5 |
| Poor validation of data entry | 5 |
| Applications cannot be accessed offline..... | 6 |
| Data security may be compromised..... | 6 |
| Higher load on the server | 6 |
| The Next Generation of Internet Applications | 6 |
| <i>A fundamental analysis of thin clients</i> | <i>6</i> |
| <i>Convergence of applications programming and the Internet</i> | <i>7</i> |
| <i>Modern rich client platforms</i> | <i>7</i> |
| Java..... | 8 |
| .NET | 8 |
| What Distinguishes .NET? | 8 |
| <i>Advantages.....</i> | <i>8</i> |
| A simple and productive API and programming language..... | 8 |
| Centralized hosting and management..... | 9 |
| Security..... | 9 |
| Client-side footprint..... | 9 |
| Next Generation | 9 |
| <i>Drawbacks.....</i> | <i>9</i> |
| Weak cross-platform support..... | 9 |
| Installation of the .NET Framework..... | 10 |
| Conclusion..... | 10 |

Purpose

In this paper, we give a brief history of the Internet and a brief history of applications programming and show how the two are converging to form the foundation for the next generation of Internet services and applications.

The Evolution of the Internet

In the beginning – Academic data exchange

The Internet started as a collection of networked computers that shared common ports and protocols that allowed the transfer of data from machine to machine. It was initially used as a collaboration tool for sharing data among researchers, but it grew over time to encompass a number of useful services. As valuable as these services were, the accessibility of the Internet was limited to those with some technical ability, and the majority of applications were targeted for the scientific and engineering community.

The explosion of the Web - Content presentment

With the advent of hypertext, the Internet expanded rapidly. Hypertext was seen as a way to turn the Internet into a simple medium for distributed content presentment, which motivated:

- The development of HTML (HyperText Markup Language)
- The development of HTTP (HyperText Transfer Protocol)
- The assignment of port 80 on all computers for the transport of data over HTTP

The combined impact of these three developments was enormous. HTML provided a global standard for the authoring of content and a simple mechanism for navigating through that content. HTTP provided a global standard for the request and transmission of that data.

However, perhaps the most important feature was that the opening of port 80 allowed all computers to be connected and to communicate with each other. Prior to this, most ports were blocked by corporate firewalls for security purposes. However, since HTTP was considered a safe protocol, port 80 was allowed through the firewall and now allows nearly universal access to any computer connected to the network.

This ubiquitous access to the content, along with its simple navigation mechanism, allowed the Internet to expand rapidly from being a scientific and engineering tool, to a mass medium with rich content that was accessible by the layperson.

The maturing Internet – Applications and interactivity

The success of the Internet as a medium for content distribution caused many to take the next logical step and try to turn the Internet into a platform that could host interactive applications. Initially, the applications were simple, and several extensions to the Web provided the necessary functionality. Cookies, DHTML (Dynamic HTML), and JavaScript were introduced in order to allow applications to provide moderate functionality. More recently a variety of AJAX (Asynchronous JavaScript) libraries have been introduced allowing applications to offer a more interactive experience. However, even with the newest extensions the basic limitations of the Web are still very apparent.

Though these extensions solved a few of the problems, they were really bandages on top of a platform that was designed for presenting and formatting content, not for hosting applications. Sophisticated applications on the Web have largely failed for two reasons:

1. User interfaces built in HTML cannot represent all the necessary elements for building a good interface to a complex application.
2. The cost of building an application with snippets of HTML and JavaScript becomes prohibitively large with the increasing functionality of the application.

The programming paradigm is simply wrong. It's like trying to mow a lawn with a pair of scissors. This is why the vast majority of ASPs (Application Service Providers) have failed to succeed on the Internet. The cost of building an HTML application increases disproportionately as the application becomes more complex and in general the value of any complex HTML application to the end user is substantially less than the cost of building it.

The Evolution of Applications

In the beginning – Character-based interfaces

Paralleling the development of the Internet was the development of applications programming for the personal computer. Initially, applications were developed using character-based interfaces, and these user interfaces were designed by manipulating character strings on the screen. As primitive as this seems, with the exception of the mouse, this is the state of programming on the Web today. HTML strings are assembled by the application, which are then sent to the browser to form the appearance of a user interface. Though the analogy may appear crude, it is not as far from the truth as one might think.

Applications become mainstream – Graphical user interfaces

With the development of the mouse and the GUI (Graphical User Interface), applications now had the ability to represent complex behavior to the user in a more intuitive manner. The complexity of programming these interfaces necessitated the need for the operating system to support libraries of functions that could be used by all applications, without having to rewrite the interface code every time. On the Windows operating system, these functions became encapsulated into what is now known as the Win32 API (Application Programming Interface).

The pervasiveness of computers that supported GUIs allowed applications to become mainstream. It would be hard to think of a company today that does not use a graphical word processor, spreadsheet, or accounting package.

The significant growth in functionality of GUI-based applications and the tremendous increase in processing power have placed considerable stress on the design of GUI programming interfaces. On the one hand, the interfaces need to be extended to encapsulate the latest advances in functionality, but on the other hand, they need to be backward compatible to support a growing body of older (legacy) applications. This stress has caused the design of all major GUI programming interfaces to become cluttered and more difficult to use. This will become an important factor we discuss later in this document.

Connected applications – Client-server computing

As computer applications have become mainstream, they have also increased the scope of the problems they are trying to solve. Many of these problems are no longer fit the domain of an isolated single user application, and require data to be shared and synchronized between multiple users, many of whom may be using the system concurrently.

These applications became known as client-server applications and were typically implemented with dedicated networks setup by a company's IT (Information Technology) staff. As useful as these applications were, they also had become difficult to manage. Every time a significant change was made to the application, the IT staff would need to visit every client computer and update the software to become compatible with the new server. For a company with hundreds, thousands, or even tens of thousands of clients, this was a logistical nightmare.

The First Generation of Internet Applications

The birth of the modern thin client

A number of factors have converged to cause the rise of thin client applications on the Internet. By thin client, we mean applications that perform the majority of processing on the server, and contain minimal logic on the client. In this section we will characterize these factors and briefly describe their attributes.

The pervasiveness of the Web

Among the chief motivations for switching from traditional client-server computing to Web-based thin clients is due to the pervasiveness of the Web. Networked applications no longer need to set up dedicated networks. Enabling Internet access on each computer is sufficient. Furthermore, because HTML is accessible on virtually any platform (Windows, Unix, Macintosh, etc.), cross-platform support is nearly free.

Centralized application maintenance and hosting

Another huge motivation is one that Sun and Oracle have been pushing for years: centralized application hosting and maintenance. As compared with traditional client-server computing, applications can now be maintained and updated in one place: on the server. Technical staff does not need to visit and update every client machine every time an application is updated. In addition, there is no installation pain. Because the application is centrally hosted, users do not need to understand how to obtain and install an application. They simply bookmark a URL and are done.

Small client-side footprint

Because many users access the Internet through modems or other low speed connections, they are reluctant to spend the time to download larger client-side applications. This has prevented wider adoption of larger clients except where a killer application has a significantly improved experience with a rich client. Instant messaging, MPEG video and audio players, and email/newsreaders are good examples of this.

Client-side security

Traditional client-side applications have the possibility of spreading viruses or exposing private data to untrusted third parties. Since thin clients perform minimal processing on the client computers, they are very passive and pose little security risk to the client.

Learning curve of the Win32 API

Over the years, programming paradigms have changed and GUI functionality has expanded, causing the Win32 API to become bloated, confusing, and difficult to use. For simple applications, HTML is easier to learn and implement.

Because of these factors, the Web-based thin client has become the dominant applications paradigm for the Internet. However, this paradigm also has its associated drawbacks, which we explore in the next section.

Problems with Web-based thin clients

For simple applications, the current thin client paradigm is reasonable. Unfortunately, there are also a number of disadvantages to Web-based thin clients, and as applications increase in functionality, these issues become major problems.

Poor programming model

One of the primary problems is that the Web suffers from an incredibly poor programming model. This was a platform that was designed to present and format content. To ask it to do more by patching together JavaScript and HTML strings to create the illusion of an interactive application is really asking for trouble. It increases the cost of building larger applications, and sophisticated applications become prohibitively expensive to build.

The added complexity of building applications on the Web leads to more fragile implementations. Applications will have bugs that are difficult to fix and may leave security holes that can expose data to unauthorized parties on the server.

In addition, applications are becoming more than just services to people. They are quickly becoming services to other applications. A Web site that validates credit cards is a useful service to another application.

As a result virtually any sophisticated thin client application built today is a cobbled together collection of the legacy web technologies along with a judicious sprinkling of these new web services. In order to build such an application developers must have at a least a limited mastery of between three and six highly disparate technologies. Further, the programming glue required to integrate these web services into the older HTML/JavaScript model is often highly customized and frequently frail. What conceptually should take one or two lines of code now becomes hundreds, or even thousands. It simply is not effective.

Inability to represent major user interface paradigms

Equally problematic is that fact that HTML and JavaScript are simply unable to represent many common user interface paradigms. This limits the functionality and therefore usefulness of a Web application. Necessary elements such as wizards, dialog boxes, and menus are very difficult to represent, and actions such as drag-and-drop or double-clicking are impossible.

Slow response time

Every page requires a round trip to the server to reconstruct the view, which may take several seconds. Simple actions like deleting an item or sorting a column in a table become frustrating to do, and the user experience for the application suffers. The advent of AJAX (Asynchronous JavaScript) has alleviated this problem somewhat but at the cost of greatly increasing complexity and application frailty.

Poor validation of data entry

Most of the data validation must wait until the page is submitted to the server. If client-side validation is implemented, it is either simplistic or susceptible to bugs from DHTML or JavaScript artifacts. The set of controls available for structured data entry is also very limited.

Applications cannot be accessed offline

Because thin clients cannot be accessed offline, this limits the types of applications that can be hosted. Some businesses have mission critical applications that cannot be shut down simply because their Internet provider is having problems.

Furthermore, many applications are still valuable even when not connected to the Internet. Some users may want take their data with them on the road. Some will create the data offline and synchronize the next time they are connected.

Data security may be compromised

Web applications are susceptible to hacking by submitting artificially constructed strings to the server, stealing other client's cookies, etc. which could expose sensitive data to third parties

Higher load on the server

Since the server is performing all the logic for the clients, a more expensive and elaborate server farm is required.

The Next Generation of Internet Applications

A fundamental analysis of thin clients

The emergence of the Web-based thin client was due to the confluence of a number of factors, many of which have nothing to do with the fundamental properties of thin clients. In fact, if we examine the fundamental attributes of thin clients, we only find two predominant properties:

- Small client-side footprint
- Low client-side processing power

Since many users lack high speed Internet access, a small footprint on the client really does have an advantage with the current infrastructure. However, this is only a temporary condition, and as high speed access continues to become more common, this advantage diminishes over time. It is possible that wireless bandwidths will always be low, but in this case, it is unlikely that complex desktop applications would be downloaded to a mobile device anyway.

Interestingly, with the advent of AJAX and web services the basic footprint of many thin client applications has grown enormously. In many applications the idea that a thin client has a larger footprint than its corresponding rich version is simply no longer true. Thin clients are getting fat.

As for lower processing power on the client, this can actually be a disadvantage for a thin client architecture. Client-side processing power has been shown to be cheap and very abundant. For example, one figure puts Intel as having shipped 1000 times more processing power than Sun did in the year 2000. This ratio has only expanded since that time. Since the majority of Intel's sales were to the desktop, the ratio of processing power at the client dwarfs the processing power at the server by several orders of magnitude.

While browsing the Web, the processors on client machines are essentially idle. Why not put it to use? By distributing the computational load to the clients, the server load can be reduced, saving on equipment and maintenance costs. These savings can then be reinvested in the company or passed on to the end user to provide a lower cost service.

In summary, provided that Internet access speeds continue to rise, there are really no fundamental properties of thin clients that necessitate their use for sophisticated Internet applications several years from

now. Yes, for lightweight applications they still make sense. Content presentation, searching, indexing, etc. will still be built using thin clients, but serious applications have no inherent reasons to stay with the thin client paradigm.

Convergence of applications programming and the Internet

If not the current thin client paradigm, then what is the right solution for larger applications on the Internet? What we are really looking at here is the convergence of the Internet with applications programming, and we can start to build a solution from one of two endpoints.

The first generation of Internet applications presumed the usefulness of the Web and tried to build applications on top of it. As we have seen, this has reached an impasse where application complexity and usefulness are severely limited by the Web architecture.

Since applications are difficult to build, it actually makes more sense to start from the other direction. Take a successful applications paradigm, and then extend it to support the usefulness of the Internet. It turns out this is a far more effective approach.

Client-server computing with rich clients is actually a rather effective paradigm for applications development. However, the specific technologies used to build traditional rich clients have had numerous problems that have prevented their adoption on the Internet. Among the predominant problems are: the lack of centralized hosting and maintenance, poor client-side security, poor cross-platform support, a cumbersome GUI API, and a larger client-side footprint.

These are extremely serious problems. Fortunately, only one of these problems is inherent to rich clients themselves, and the rest are artifacts of the particular technologies used to build them. The one inherent issue is that that of the larger client-side footprint. However, as broadband connections become more common, this becomes less of an issue over time.

What can we gain by moving to a good rich client paradigm? To start off, the majority of features that made Web-based thin clients popular can also be implemented with rich clients, because they were never inherent properties of thin clients to begin with. If the technology is designed properly, the following features are all possible with rich clients:

- Centralized hosting
- Client and server side security
- Cross-platform support
- Good user interface
- Good programmatic interface
- Instantaneous response time
- Good validation of data entry
- Lower server load
- Offline access to the application and data
- Ability to integrate with other client-side applications

Modern rich client platforms

So far we have been talking in the abstract about a modern rich client design. The real question is: are there any technologies that will allow us to implement modern rich clients? In fact, there are. The first technology came from Sun using its Java platform.

Java

Java is an elegant language that caught the eye of many developers because of its simplicity in building applications as well as its cross-platform support. Initially, there was quite a bit of excitement about Java applets and applications that were downloaded to the client to execute. It appeared that a good Internet aware, applications programming environment had finally arrived, and it had the potential to fulfill many of the attributes of a good modern rich client.

Unfortunately, Sun decided to focus on extending Java for server applications, and largely ignored the features that could have made it a successful client-side platform. The reasons for this become clear when you look at their revenue stream. Sun makes the vast majority of its revenue by selling servers. If they could drive the world to a thin client paradigm where all of the processing happened on a Sun server, their revenue would shoot through the roof. This is likely the reason why Sun has been so averse to developing a successful rich client platform.

.NET

The second technology supporting modern rich clients came from Microsoft. The Microsoft .NET platform was officially launched in February of 2002 and is currently transitioning into its third major version. The subsequent discussion section, 'What distinguishes .NET?' describes some the features of the .NET platform.

What Distinguishes .NET?

Advantages

As much as Microsoft has been maligned, the engineering behind .NET is actually quite good, and over the past 5 to 6 years its use across a variety of application domains has become increasingly prevalent. .NET has taken and improved upon many of the best ideas from Java, and has incorporated numerous improvements in order to address the remaining issues that have prevented rich clients from receiving widespread adoption on the Internet. Many more comprehensive overviews of the .NET platform exist but we give a summary of the issues below:

A simple and productive API and programming language

Previously, the complexities of the Win32 API have scared off many developers from developing on Microsoft's platform. The majority of the Win32 API has now been rewritten into a clean, object-oriented framework that is much easier to use and understand. This is the .NET WinForms technology.

More recently, Microsoft has shipped a next generation UI (User Interface) technology known as WPF. WPF is still in its infancy but shows the promise of being a highly sophisticated yet comparatively approachable technology that will enable applications to be hosted across a variety of platforms.

.NET has also introduced or enhanced several programming languages to operate on top of the .NET framework. Visual Basic and C# are among the most popular. C# shares many of the same features with Java. Previously, developers had to write in C++ which was a far more difficult language to use correctly.

The result of all this is that rich clients written in .NET can have:

- Good user interfaces
- Good programmatic interfaces

- Instantaneous user feedback
- High quality data validation
- Rapid application development

Centralized hosting and management

.NET also allows applications to be hosted on a centralized server. Installation is transparent to the user and .NET automatically takes care of updating the application as new versions become available. Users do not need to worry about downloading the application, finding out where they put the download, and figuring out how to install it.

Security

.NET implements a multi-level trust and security model. Applications are cryptographically signed to prevent accidental or intentional modification. Applications also run in a local sandbox, which allows them save application data to the disk, but are isolated from the file system so they cannot read or write any data they did not create, unless granted permission from the user.

Client-side footprint

Because the .NET Framework is a much cleaner interface than Win32, the applications code tends to be more compact and occupy a smaller footprint on the client. .NET also downloads an application incrementally, which allows users to use an application without committing to a large up-front download. As additional modules are needed in the application, they are retrieved from the server and cached on the client side until a new version is available. This means that the actual bandwidth used by a rich client application may actually be lower than that of a thin client, because in the long term, the rich client is receiving only data from the server, whereas the thin client is receiving both the data and the user interface for every page request.

Next Generation

Microsoft has recently demonstrated another new technology, known as SilverLight, with the intended goal of allowing .NET applications to deploy with **exactly** the same distribution and security requirements as existing thin client applications. This technology promises to completely level the playing field in both of these areas.

Drawbacks

.NET does have some disadvantages. We describe these here.

Weak cross-platform support

The .NET Framework is currently only available on Windows platform. Although the framework is designed to be portable to other platforms, historically, Microsoft has not shown a propensity to do so. The SilverLight technology mentioned above appears to be a significant change to this historical trend. In addition, the reality is that most people need to run a Windows PC anyway for email, word processing, or slide presentations, and many applications are built with some degree of cross application integration where a Windows PC is still a requirement.

Installation of the .NET Framework

Before .NET applications can be run, the .NET Framework needs to be installed on the client computer. Once it is installed, all .NET applications can take advantage of it. The .NET applications themselves are transparently installed, but the .NET framework itself is not.

However, as .NET's market penetration has increased, many machines are now delivered with the .NET framework preinstalled. As significantly, the SilverLight project currently delivers a highly functional subset version of the .NET that has a design goal of loading over standard internet connections in under 20 seconds.

Conclusion

We strongly believe that the current Web-based thin client paradigm is ill suited for the development of sophisticated applications on the Internet. A new paradigm is required, and we believe that the convergence of the Internet with client-server applications programming is inevitable. This convergence will result in the modern rich client, which will replace the Web-based thin client where more sophisticated applications are needed on the Internet. .NET addresses the vast majority of concerns for Internet-based applications and currently has the lead over other technologies in this area.