Agility and Experimentation: Practical Techniques for Resolving Architectural Tradeoffs

T.C. Nicholas Graham[1], Rick Kazman[2] and Chris Walmsley
Namzak Labs
16A-303 Bagot St., Kingston, Ontario
Canada K7K 5W7
{graham,kazman,walmsley}@namzak.com

Abstract

This paper outlines our experiences with making architectural tradeoffs between performance, availability, security, and usability, in light of stringent cost and time-to-market constraints, in an industrial web-conferencing system. We highlight the difficulties in anticipating future architectural requirements and tradeoffs and the value of using agility *and* experiments *as a tool for mitigating architectural risks in situations when up front pen-and-paper analysis is simply impossible.*

1. Introduction

For the past five years Namzak Labs has developed architectures for a family of web-based remote collaboration systems, collectively called *WebArrow* [16[15]. The *WebArrow* platform offers a set of pluggable commercial web conferencing technologies, available in multiple languages, in multiple countries, and in multiple configurations. These components must meet stringent usability, performance, integrability, availability and security requirements. *WebArrow*'s users range from individuals to Fortune 500 companies.

Web-conferencing systems are necessarily complex and demanding: they must work on a wide variety of platforms, the details of which are not under control of the architect; they must be reliable and provide low latency response times, particularly for real-time functionality such as voice over IP (VoIP) and screen sharing; they must provide high security, but provide this over an unknown network topology and an unknown set of firewalls and firewall policies; they must be easily modified and easily integrated into a wide variety of desktop environments and applications; and they must be highly usable and easily learned by users with widely varying skills. Furthermore, the market for remote collaboration is highly competitive and rapidly evolving, and so new features in such systems must be brought to market rapidly and delivered at a reasonable cost. Adding to these challenges is the fact that many of the above-mentioned goals trade off against each other. Typically security (in the form of encryption) comes at the expense of performance. Modifiability comes at the expense of time-to-market. Availability and performance often come at the expense of modifiability and cost.

Finally, traditional requirements elicitation techniques—questionnaires, surveys, observation, focus groups, and so forth—are of limited help in this domain. In many cases, because web conferencing systems are unprecedented, end users and integrators do not know what it is that they want. In addition the space of possible uses of a web-collaboration system is too large, particularly since we focus on integration with third party products, rather than selling the web-conferencing system as a standalone product. As Augustine *et al.* have pointed out [3], "Even seemingly minor changes can produce unanticipated effects, as systems become more complex and their components more interdependent". But the current market will not accept failure to adapt to changing requirements and market conditions, so responding to these changes is a business necessity [9,12].

As a consequence, even if we could collect and analyze all relevant data and requirements, our stringent time-to-market constraints would prevent us from doing so. Trying to support all possible uses is intractable, and the users themselves cannot envision all possible potential future uses. This results in a classic "agility versus discipline" problem [5,14]. On the one hand we want to provide new capabilities quickly, and respond to customer needs rapidly. On the other hand, long-term survival necessitates that we design for extensibility and modifiability, emphasizing a simple conceptual model. We are

[1] Also: School of Computing, Queen's University, Kingston, Canada.
[2] Also: Department of Information Technology Management, University of Hawaii, and Software Engineering Institute, Carnegie Mellon University.

faced with the problem of determining the “sweet spot” between these opposing forces.

In our environment it is virtually impossible to do purely top-down architectural design; there are too many considerations to weigh at once and it is too hard to predict all of the relevant technological barriers. For example, we have had cases where we discovered that a vendor-provided API did not work as specified, or that an API exposing a critical function was simply missing. In such cases, these problems rippled up to the architecture, where a workaround needed to be fashioned. To address the complexity of this domain, we found that in practice we needed to concurrently work both top-down (designing architectural structures that meet our quality attribute goals) and bottom-up (determining a host of implementation-specific and environment-specific settings and constraints).

Finding the sweet spot within the enormous architectural design space of such systems is not feasible by reflection and analysis alone, despite the well-known successes of analytic techniques [7]; there are too many unknowns and too many uncontrollable factors—such as third-party hardware and software in a multitude of versions—over which we have no control. These technological and environmental constraints can outweigh design principles. To compensate for the difficulty in analyzing architectural tradeoffs with any precision, we have adopted an *agile* architecture discipline combined with a rigorous program of *experiments* aimed at answering specific tradeoff questions. These experiments have proven invaluable in resolving tradeoffs, by helping to turn unknown architectural parameters into constants or ranges.

The remainder of this paper will introduce the architecture of *WebArrow*, and will present three specific architectural design problems that we have faced. We will show how agility and experimentation helped us to locate satisfactory solutions from the enormous landscape of potential solutions.

2. Overview of *WebArrow*

WebArrow has been developed by a team of four to fourteen developers over the last five years. The team leads had 5-15 years of development experience, and the teams themselves balanced fresh computing science graduates with more experience developers.

WebArrow's product-line architecture was designed to provide five basic collaborative services—desktop sharing, application sharing, VoIP, file sharing, and chat—along with a large number of supporting services. Some of these supporting services are real-time, augmenting the collaborative experience—such as shared document annotation, remote cut/copy/paste, moderation and “hand-raising”—and some are administrative, supporting billing, auditing, monitoring, and communication and coordination with other systems.

A *WebArrow* session involves two or more users, who might be talking, sharing their desktop or a single application, sharing files or chatting. The Meeting Control Window of a typical session is shown in Figure 1, where Ralph is sharing his desktop (as indicated by the icon) and Nick is talking (as indicated by the icon). The other participants are viewing Ralph's desktop (as indicated by the icon) and all have a chat window available (as indicated by the icon).

Although this is a typical usage of *WebArrow*, it is by no means the only way that it is employed. As stated above, *WebArrow's* focus is on integration—being used as a customer support tool accessed from a web-site, or as a complement to a teleconferencing, video-conferencing or instant messaging system. Integrators may choose to use a different meeting control window, or perhaps no meeting control window at all, and may choose to reveal only a subset of the available collaboration technologies. For example, a teleconferencing system typically does not reveal the VoIP functionality, and a customer-support application typically provides no meeting control options to the customer being supported.

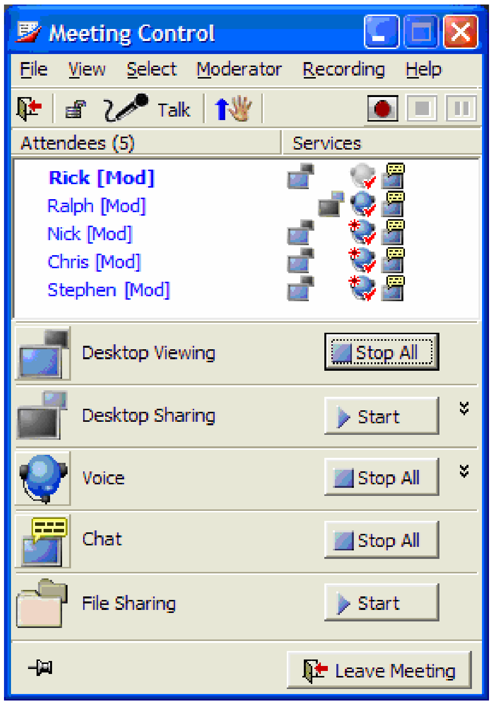

Figure 1: *WebArrow* Meeting Control Window

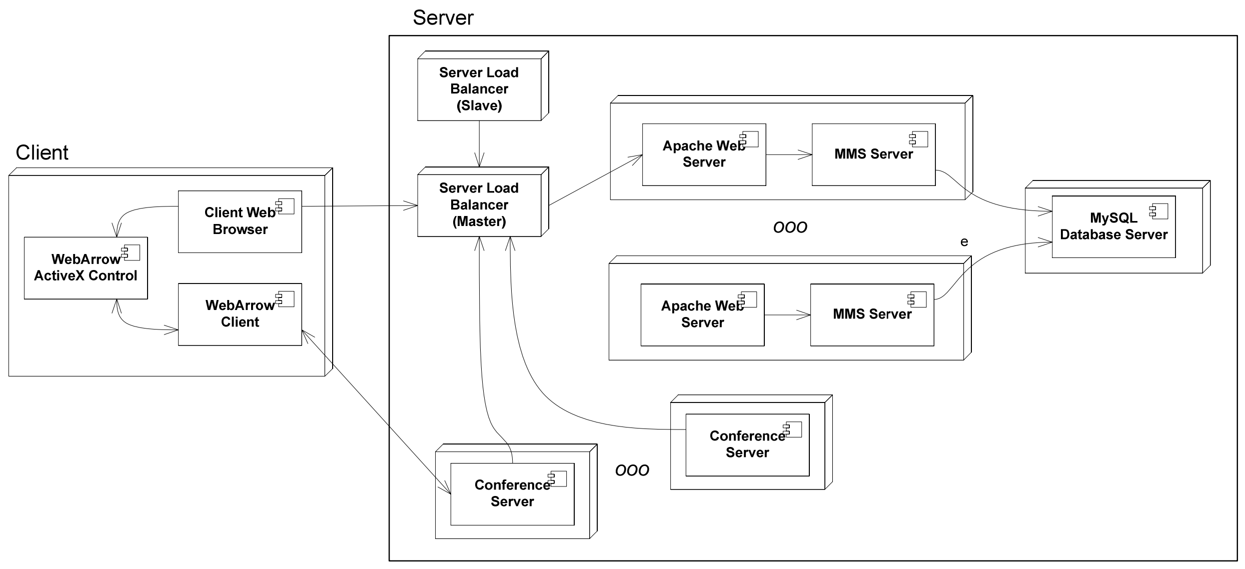

Figure 2: Deployment View of WebArrow Architecture

But in all cases the users and integrators of *WebArrow* want the system to be high performance, highly reliable, secure, and intuitive. These usage scenarios provide the context for the tradeoff challenges discussed in Section 4.

3. *WebArrow* Architecture

A deployment view of the *WebArrow* architecture is shown in Figure 2. Two of the driving architectural requirements for the *WebArrow* system architecture are ease of scalability and availability. In this architecture clients (web clients and *WebArrow* clients) connect to a farm of MMS (Meeting Management System) servers via a server load balancer (SLB). The SLB is used to distribute incoming HTTP requests amongst a pool of MMS Servers. The SLB can be configured to allocate requests in a round-robin fashion, or based on MMS server load. The SLB monitors the MMS servers, and removes them from the pool in case of failure, and the SLB itself has a backup (slave) in case of failure. The use of the SLB thus promotes availability, but it also promotes scalability, by making it easy to add new servers without impacting any system components beyond the SLB. In addition, all MMS servers are stateless, meaning that any server can take over from any other server in case of a failure.

Conference Servers implement the communication for individual meetings: screen sharing, VoIP, file and chat data. A pool of conference servers is available for scheduling meetings. The MMS Server makes the decision as to what conference server is to be used for a given meeting. MMS Servers load balance conference servers, and remove conference servers from their available pool should conference servers become unavailable. This provides both scalability in the number of meetings that can be hosted, and failover capability should a conference server cease to operate correctly. Finally, conference data (such as account information, schedule information, meeting status and billing data) is stored on a database server. To provide failover capability, a slave database server mirrors the database server in real time.

The architecture shown in Figure 2 is the end result of more than five years of development. In the next section, we will discuss the evolution of the *WebArrow* architecture as business needs, requirements, and tradeoffs in quality attributes changed over time.

4. Architectural Tradeoffs

Any architecture embodies tradeoffs. In *WebArrow* we have many tradeoffs forced upon us by our business model (offering pluggable services, rather than a standalone product), by the realities of the competitive marketplace (the need to keep costs and time-to-market low) and the user expectations that this generates, and by using the Internet with all its vagaries as our underlying communications infrastructure. In this paper we will discuss three of the most important tradeoffs, and how we have reacted to each of them via architectural strategies.

In designing and building an architecture there are three practical strategies for managing change:

1. Plan to build one to throw away, following Brooks' advice [6],
2. Focus on responding to change rather than

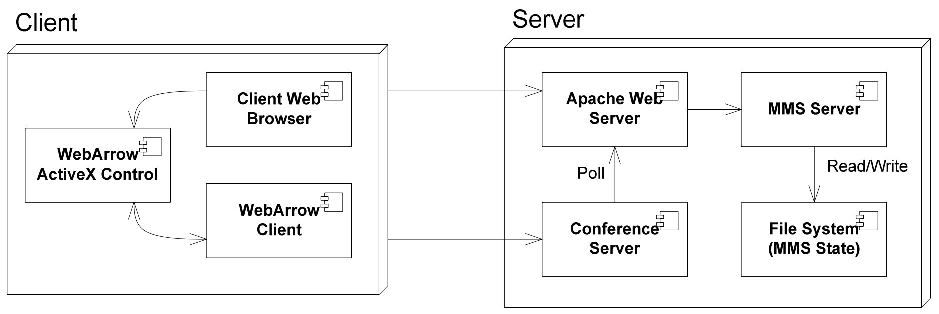

Figure 3: Original WebArrow Architecture

extensive up-front planning, following the precepts of the agile community [1], and

3. Anticipate decisions that are likely to change and modularize them, limiting impact of eventual changes to a small part of the overall system.

Option 1 was simply not possible for our company; we did not have the resources for the luxury of planning to throw one away. Option 3 was similarly impossible—we had to respond to change rapidly and incrementally, and we had to be able to provide a stream of gradually improving releases to our customers.

Option 2 provided a realistic approach allowing us to combine timely releases of the product with quick adaptation to new requirements as they arose. While we did not rigidly follow any codified agile process [1], [2] we did follow their core principals of incremental development, refactoring in the face of new requirements or better understanding of the problem domain, and continuous empirical evaluation. An important distinction is that agile methods normally advocate empirical evaluation to ensure that the process is on track. In our case, *experimentation* was critical for resolving architectural design decisions.

The examples presented in the rest of this section show our experience with this approach in the development of *WebArrow/Conference*.

4.1. Scalability and Availability

Over the five years of development of *WebArrow/Conference*, we have expended significant effort to increase the scalability and fault tolerance of the system. This work in scalability has been motivated by changing and emerging requirements. The types of changes that we encountered illustrate the impossibility of initially designing an architecture for a commercial system that accounts for all possible evolutions in the system's requirements. The changes that we carried out were supported by a continuing regimen of experimentation. When working with complex systems involving a variety of third party (commercial and, increasingly, open-source) components, there is no practical way of predicting the results of architectural changes on scalability, and so constant experimentation must drive development. There are simply too many unknowns and too many forces not under the designer's control. This leads to a development process guided by top-down requirements and design, but ultimately driven by bottom-up experimentation and refactoring to address emerging and changing requirements. This is a picture of architectural design that is not often described or discussed.

4.1.1 Initial Requirements. Initially, *WebArrow/Conference* was targeted towards local deployment in small installations. In the Japanese market, the product was licensed as a "conference in a box" solution in which an MMS, conference server and Web server are packaged in a single computer hosted at the customer's site. While capable of higher load, this solution was targeted towards customers holding conferences of up to 10 participants. With this deployment, scalability was not a concern since even our initial prototype of *WebArrow* conference could easily handle the required load on inexpensive stock hardware. The possibilities for fault tolerance were limited, since there was no redundancy in hardware or network connectivity. Failover requirements were limited to ensuring that the various services (web server and conference server) automatically restarted after failure. The main initial requirements therefore focused on ease of installation and upgrade, provision of simple off-site monitoring, low expense in deployment, and rapid time to market.

To meet these requirements, we deployed the initial *WebArrow/Conference* architecture (Figure 3). Our small size meant that time to market was critical, and so we extended a prototype version of the tool rather than re-architecting from scratch. The conference sever and *WebArrow* client were programmed in C++; the web component of the system was programmed mainly in Perl with performance-critical components programmed in C, and the conference data was represented in flat files. This prototype architecture met our initial requirements:

- Performance and scalability requirements were not stringent in the "conference in a box" deployment. Failover requirements were met using the standard service architecture facilities of Windows 2000/XP.
- Deployment was inexpensive, since no for-pay COTS were used (e.g., commercial databases or web development frameworks)
- Monitoring support was easily added via a web-based status reporting tool.

4.1.2 Scalability and Failover. Over time, particularly in the North American market, the company moved towards offering web conferencing as a hosted service, where customers subscribe to the use of conferencing facilities without requiring an installation at their own site. Even large enterprise customers and development partners embedding *WebArrow* conferencing within their own products have preferred this model, following the industry trend of outsourcing both software development and software services. This shift had a significant impact on *WebArrow/Conference*'s requirements, in turn impacting the system's architecture. In particular,

- Scalability became a significant issue, as the cost of hosting was no longer borne by individual customers, but instead by the centralized service.
- Failover became highly important, as service users, particularly our development partners, require high availability.
- Security became far more important, as companies perceive risks in entrusting their sensitive business communication to an externally hosted service.
- The cost of COTS used in deployment became less important, as these costs could now be amortized over much larger numbers of users sharing a single facility.

To address these changing requirements, the new architecture of Figure 2 was developed. In this new version, the original architecture was refactored in the following ways:

- The flat files used to represent the MMS state were replaced by a third-party database. The database was set up in a cluster, supporting scalable, fault-tolerant operation.
- mod_perl [14] was used to dramatically improve the performance of the server-side MMS code.
- Multiple MMS servers were deployed in a farm behind a redundant pair of third-party SLBs.
- Conference servers use a load polling mechanism to support availability monitoring and load balancing by the MMS server.

This set of changes was carried out progressively, allowing us to incrementally improve scalability and fault-tolerance while maintaining an uninterrupted service to our client companies.

The primary lessons from this experience are the enormous difficulty of anticipating required changes to a system's architecture during the initial design phase, and the benefits of using an incremental, agile approach to change. While we could have initially architected the system for scalability and fault tolerance, these were not strong requirements at the time, and addressing them would have significantly increased our time to market and development expense (in addition to which we might have been wrong about the ways in which the system needed to grow and change). The approach of progressively and incrementally changing the current architecture was invaluable. Building a new system from scratch (following Brooks' recommendation of "plan to throw one away" [6]) would have led to a long period before any scalability improvements were delivered. Additionally, building a new system would have been risky, as new features introduced in the production system would also have to be migrated to the new system before its deployment, providing a moving target for development. A new system would have had benefits; for example, rather than the cumbersome mod_perl approach, we could have used an enterprise web platform; but on balance, the known risks and costs of developing a new system strongly outweighed the potential benefits.

4.1.3 Experimentation. The incremental improvement in the scalability and fault-tolerance of *WebArrow/Conference* was guided by significant experimentation. Our experience shows the importance of driving architectural decisions by low-level experiments rather than the "by the book" approach of top-down architectural design followed by coding. The sorts of questions experiments were required to answer were:

- Would moving to a distributed database from local flat files negatively impact feedback time (latency) for users?
- What (if any) scalability improvement would result from using mod_perl versus standard Perl? How difficult would the development and quality assurance effort be to convert to mod_perl?
- How many participants could be hosted by a

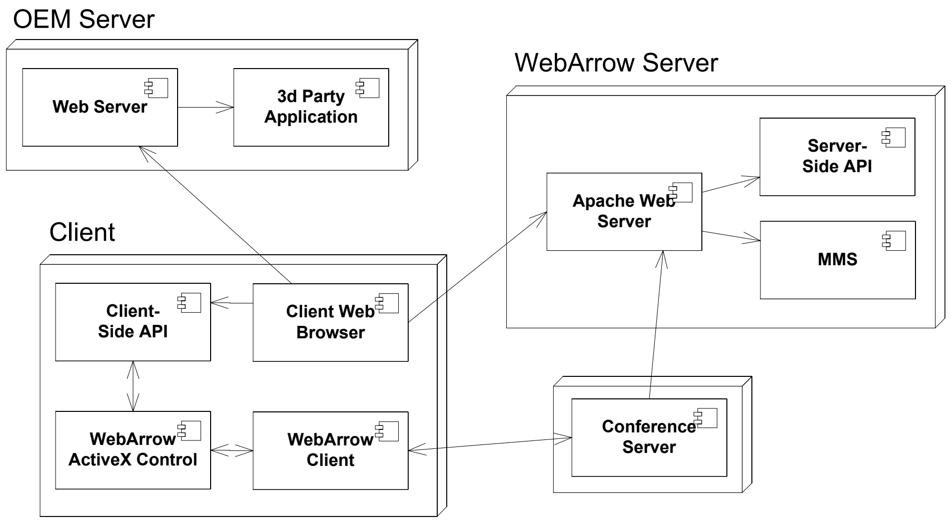

Figure 4: Deployment View of WebArrow's API Architecture

single MMS server?

- What was the correct ratio between database servers and MMS servers?

These sorts of questions are difficult to answer analytically. The answers to these questions rely on behaviour and interaction of third-party components, and on performance characteristics of software for which no standard analytical models exist. Our approach was to build an extensive testing infrastructure, and use it to compare the performance of each incremental modification to the code base. This allowed us to determine the effect of each form of improvement before committing to including it in the final system. The testing infrastructure includes:

- A client simulator that allows tens of thousands of clients to simultaneously interact with an MMS server.
- Instrumentation to measure load on the MMS server and database server with differing numbers of clients.

The lesson from this experience is that experimentation is a critical pre-cursor to making significant architectural decisions. Experimentation must be built into the development process: building experimental infrastructure can be time-consuming, possibly requiring the development of custom tools. Carrying out the experiments and analyzing their results can require significant time. These costs must be recognized in project schedules.

4.2. Third Party Integration API

Our second example of how changes in requirements affected our architecture in unpredictable ways is centered around the evolution of the *WebArrow/Conference* API framework. These APIs are used by third party companies to integrate web conferencing into their own applications.

Figure 4 shows the architecture of a third party application integrated with *WebArrow/Conference*. The core feature of this architecture is that external applications can make API calls to *WebArrow/Conference* to schedule, initiate, join and terminate conferences, and to obtain data (primarily used for billing) on past conferences. The architecture allows third party programs written in any language to make API calls to the MMS using HTTPS, with results returned in XML. This is a lightweight form of web service that does not require API users to master the full framework of SOAP and enveloping. The use of HTTPS for communication provides strong security by encrypting the data in the API calls and allowing the API user to reside behind a firewall.

The API framework is divided into two parts: a server-side API, used for meeting scheduling and reporting, and a client-side API, used for meeting entry and runtime meeting control. An overriding requirement in the design of the API was ease of use. Integration partners vary widely in their technical expertise, from large corporations with sophisticated development departments to small businesses with a single programmer on staff. Uniformly, the ability to perform integrations quickly with minimal learning curve was cited as key to working with Namzak Labs. One integration partner required that a complete integration take no more than one week of local development time. The API architecture allows integration partners to perform integrations using the language of their choice, using their own environment. Connecting to Namzak Labs' infrastructure over the internet relieves them of the time-consuming, highly technical local installation, configuration, and tuning of *WebArrow/Conference*.

The challenge in the design of the API itself was to keep the operations high-level and powerful enough that integration partners had a low learning curve and development cycle, while providing sufficient flexibility so that new integration partners were able to meet unanticipated requirements. This tension between flexibility and ease of use has led to continual refinement of the API in the years since its initial release. We now discuss two examples that illustrate the forms of architectural change that have been required: meeting end callbacks and the client-side library.

Integration partners typically require a reporting mechanism to determine what chargeable services were used in each web-conference so that they can generate a bill for their own (typically external) customers. Our early partners all had similar requirements; each provided a monthly bill to their customers, and so needed API calls allowing them to request usage information for a given billing period. To meet this need, we developed a simple, powerful query language allowing integration partners to request a wide range of usage information over arbitrary time periods.

Problems arose with a new integration partner whose requirements were slightly different. Here, customers were billed via a credit call *each time* they used the system. A bill was presented in the customer's web browser immediately following the call. In the initial implementation of this integration, the API was called each time a web conference was completed to determine the cost of that session. While technically the existing API met this requirement, in practice, as call volumes rose, the load on Namzak Lab's servers became untenable. An alternative architecture was developed, where a much lighter-weight callback mechanism was used to report call termination to the integration partner. This approach required more effort on the part of the integration partner as they had to provide the mechanism for handling the callbacks, but provided the scalability necessary for production use.

A second issue is the library provided to support client-side API integration. As was shown in Figure 2, the *WebArrow/Conference* client is launched via the web. Therefore, it is natural that facilities for client-side runtime control of meetings would be presented as high-level JavaScript routines that could be called from within the integration partner's own web page. More recently, however, we have worked with integration partners whose products are *not* web based, and thus have not been able to use the client-side API without launching an otherwise unnecessary web page (for example, an instant messaging tool augmented with web conferencing). Given the problems of multiple browser versions and the security issues involved in client-side browser programming, this would have been a cumbersome solution. We thus are in the process of creating a version of the client-side library specifically for use in non-web-based applications that replicates the function of the JavaScript library, but is callable from within a standard programming language such as C or Java.

These examples illustrate the difficulties in attempting to predict the ways in which an API will be used. In theory it might be possible to design an API to be so general that any foreseeable future use is possible, by avoiding aggregation of operations and liberally parameterizing operations. However, the cost of this generality is ease of learning and ease of use—such an API would be large and complex—which in turn compromises its adoption.

This experience lends weight to our contention that it is better to architect for flexibility than to attempt to anticipate all possible future changes.

4.3. Changing Security Environment

Security is an area of great interest to users of web conferencing systems. Users worry about the danger of their meetings being intercepted by third parties, about the danger of their computers' being made vulnerable to outside attack due to the installation of the web conferencing software and, in corporate environments, about the danger of employees using an externally hosted web conferencing tool to transmit sensitive information in a way that cannot be easily monitored.

While tools like *WebArrow* must provide air-tight security to their users, the development of *WebArrow* has also faced the challenges of increasing restrictions in the deployment of internet-based applications, both from increasingly secure corporate environments and from the inevitable tightening of security in the Windows operating system. The effect on *WebArrow*'s development of this changing security environment compellingly illustrates the importance of experimentation in architectural design and the value of agility over a more front-heavy architectural design process.

4.3.1 Evolving Distribution Architecture. Over the five year lifetime of the *WebArrow* architecture, changes in the security environment in which *WebArrow* is deployed have had several significant impacts on its architecture. The following chronology illustrates these changes. Initially, *WebArrow* was developed to support remote customer service. The architecture was peer-to-peer, where customers made direct connections to operator machines. This required operator machines to have an external IP address usable by the customer. While initially acceptable, over time, increasing numbers of customers were unwilling to open their firewalls to permit inbound connection to their operators, even just to support the proprietary *WebArrow* protocol. This required us to modify the product to a centralized architecture, where all meeting participants make outbound connections to servers hosted by Namzak Labs. This represented a fundamental architectural change resulting from changes in our clients' perceptions of what was an acceptable security risk. This change from peer-to-peer to centralized operation led to new architectural challenges in fault tolerance (dealing with failure of conference servers) and scalability (ensuring that meetings are allocated to conference servers in a way that ensures timely service). Agility and experimentation was again required to perform architectural changes of this sweeping scope.

4.3.2 Evolving Firewall Technology. Initially, this centralized architecture was successful, as it permitted all meeting participants to connect *out* through their firewalls to the Namzak Labs' conference server. Most environments place fewer restrictions on outbound connections than inbound ones. Over time, however, firewalls have become increasingly sophisticated, restricting outbound connections either by port, by protocol, or by other forms of traffic analysis.

Firewalls are made to keep the external world out; web conferencing requires that the external world be allowed in. This is a fundamental tension in requirements that will never go away. Not surprisingly, firewall manufacturers and corporate IT departments do not reveal detailed information on how their firewalls work, making it difficult to develop techniques that reliably bypass them. Namzak Labs' quality assurance department has assembled a collection of all popular firewalls for testing, in order to reverse engineer the techniques that are used to detect "undesirable" outbound traffic. Over time, these experiments have led to accrued knowledge allowing the creation of a sophisticated firewall traversal library that combines techniques of port scanning, proxy identification and navigation, and protocol emulation. This library is successful in making connections over all but the most restrictive firewalls, but requires continuous work in reaction to new techniques deployed by firewall manufacturers and new configurations deployed by our customers.

This firewall traversal library could not have been built top-down (since we could never have predicted what mechanisms and policies would be employed by the entire array of firewall manufacturers), but rather required an agile process where new techniques could be added and existing techniques refined as they became better understood. Each encounter with a difficult firewall that is reported by a customer leads to new insights into how the library should function. Experimentation leading to new understanding, leading to refactoring is a critical part of this agility, as otherwise the complexity of the code will quickly become unmanageable.

4.3.3 Evolving Platform. As the market leader, Microsoft's Windows operating system and its Internet Explorer web browser are primary targets for security attacks. In reaction, Microsoft has made security a primary focus in recent years. Microsoft released significant security enhancements as part of its Windows XP service pack 2, and even more extensive enhancements are a key feature of the newly-released Windows Vista operating system.

While welcome to Windows' users, these enhancements have increased the difficulty of web-based delivery of applications. In the case of *WebArrow*, Microsoft's changes have required, once again, significant changes in our architectural approach. Additionally, the way in which Microsoft has released such changes has required extensive experimentation to determine correct architectural choices.

For example, in its Windows XP service pack 2 release, Microsoft severely restricted the function of ActiveX controls, requiring users to go through a two-step validation process before an ActiveX control could be executed. Usability tests and support call logs at Namzak Labs indicated that many users have had difficulty navigating this process—the majority of our end users are technically naive. The result is that applications requiring an ActiveX component, such as *WebArrow*, are significantly more difficult for users to launch. When it was released, the documentation provided by Microsoft for this new feature was minimal, requiring us to perform experiments on beta versions of the service pack to determine its exact behaviour. For example we needed to conduct experiments to understand how the feature interacted with different web browser security settings and different operating system privileges. Numerous companies have moved away from ActiveX delivery of their products, instead using Java or simply streaming an executable directly to their users. The fact that these two approaches allowed applications to obtain the same access to the user's computer as ActiveX, without the onerous validation process, provided a workaround, but one which could easily have been broken by Microsoft's next critical update.

Windows XP service pack 2 turned out to be a dress rehearsal for the upcoming release of Windows Vista. Vista introduces the concept of an execution sandbox, a restricted area in which child processes of Internet Explorer must run. For browser-based applications to gain access to, for example, the main file system, users must respond to a pop-up dialog box requesting an administrator password. This extends the validation process from two steps to three, further increasing the difficulty for a user. In the case of *WebArrow*, this has led to a change in the delivery architecture, moving away from the decision to rely solely on ActiveX controls.

To determine the best delivery architecture, we evaluated several technologies to determine their behaviour under Vista. These included Java applets, Java WebStart, Flash, browser plug-ins, direct execution of a streamed application, and of course, ActiveX. This investigation was performed in the absence of clear documentation (most of which was provided in developer blogs), and therefore was experimental. The experiments were complicated by changing behaviours in each release of Vista.

This example illustrates once again the practical necessity of being agile and basing architectural design on experiments, rather than following a top-down design approach. The properties of the platform so completely influence the ultimate success of the architecture that experimental investigation was critical. Of course, up-front architecture work is still important, but this example once again illustrates the need for an agile process. External events, in this case changes in the security environment, required significant architectural change on our part.

5. Reflections

In the previous section we enumerated a set of three practical options for dealing with architectural change: 1) plan to throw one away, 2) be agile, or 3) modularize decisions that are likely to change.

Given the above discussion it is revealing to consider why option 3 was not practical. We found that it was impossible for us to know, in advance, what significant changes would occur in the architecture, making it difficult to identify the articulation points around which modularization should be performed. Furthermore, we found that there were few predictive tools that were able to guide us. Let us briefly examine why this is the case.

Consider architecting with performance in mind: typical performance prediction models such as Rate Monotonic Analysis [12] or queuing models require as inputs variables representing execution time and blocking time for various threads of execution, as well as a characterization of the arrival rates and distributions of service requests. This information was not available to us. We were unable to predict execution and blocking times until we first built a conference server or a MMS. And, as discussed in Section 4.2, we did not understand the rates or distributions of service requests because our business models were constantly evolving.

Similarly we had no way of knowing, when we decided to experiment with switching from flat files to a database, whether this would improve or hurt performance. It was not known whether, for our specific workload, the overhead of running a database and retrieving data over the network would drown out the performance improvements that a database normally offers over files. Other unknowns made this profound architectural change difficult to analyze in advance. For example, for a given hardware configuration, what was the maximum load that the database could support? There were many dimensions of this problem: how many servers should we allocate to the database, how powerful should those servers be, with how much memory? Such questions are typically not addressed by analytic models, but are crucial to a commercial operation. As we learned, via experimentation, the database can handle a vast load, but we were not confident of this until we applied representative simulated workloads in a controlled experiment.

However, despite this seemingly bleak situation, when faced with these challenges, we were able to make progress. What allowed us to make these changes successfully in the absence of the ability to predict was:

- A constant program of small experiments aimed at systematically removing unknowns,
- An incremental agile approach to architecture; we never made an ambitious "big bang" rework of the architecture, opting instead for an evolutionary approach, continuously moving towards a desired state via a series of working versions,
- Doing the obvious: encapsulating where possible, and building a clean architecture with good interfaces.

Effectively, we were employing the scientific method—isolating unknowns one at a time and performing experiments to increase our knowledge in a controlled fashion. Juristo and Moreno urge the use of experimentation in software engineering [10, p. 15] (as does Basili [4]), but it is important to recognize that our approach differs from theirs. While Juristo and Moreno view experimentation as being done by researchers and "innovative developers" for publication and eventual use by "routine developers", we have used experiments as a focused tool allowing developers of all stripes to gain answers to their own questions. Experiments of this form are typically too specific to the product and environment to be of interest to the broader research community.

Our approach is consistent with Dybå's advice, that "predictability is a property of small systems", and conversely, the behaviour of large systems is very difficult to predict [8]. Dybå recommends that companies must combine *exploration* and *exploitation*, which in our context means experimentation to determine the best direction, and incremental refinement of existing assets to achieve product requirements.

But even this approach has its limitations – when experimenting, one has to know the right questions to ask. As a result we did not always architect *WebArrow* properly. For example, when we moved from a file-centric to a database-centric approach, the changes were ubiquitous – there was no abstraction or modularization of the file system and hence every module that wrote or read file data needed to be modified.

6. Conclusions

This paper has tried to show that up-front architectural design is difficult, particularly when working in an environment in which there are many variables: unpredictable marketplace forces, constantly changing COTS and open-source products and platforms, and evolving customer requirements. Unfortunately, this description covers much of the modern world of software development.

Clearly we are not opposed to architectural design—the authors of this paper have built their careers on it. What we are arguing for is a view of the architectural life-cycle that leans more towards agility and which advocates a disciplined program of scientific experimentation to shed light on architectural tradeoffs that are not amenable to purely analytic methods and tools. In this way one can still reap the many benefits of software architecture while remaining responsive to the needs of the marketplace.

7. Acknowledgements

WebArrow was developed by a team at Namzak Labs in Kingston, Ontario. We would particularly like to acknowledge the key efforts of Jason Reynolds, Darren Cousineau and Nicholas Eliot.

8. References

[1] P. Abrahamsson, J. Warsta, M.T. Siponen and J. Ronkainen, New Directions on Agile Methods: a Comparative Analysis, in *Proc. ICSE 25,* 244-254, 2003.

[2] AgileManifesto, http://www.agilemanifesto.org, 2006.

[3] S. Augustine, B. Payne, F. Sencindiver and S. Woodcock, Agile Project Management: Steering from the Edges, *CACM.* 48:(12), 85-89, Dec. 2005.

[4] V.R. Basili, The Role of Experimentation in Software Engineering: Past, Current and Future, in *Proc. ICSE 18,* 442-449, 1996.

[5] B. Boehm, R. Turner, *Balancing Agility and Discipline*, Addison-Wesley, 2005.

[6] F. Brooks, *The Mythical Man Month*, Addison-Wesley, 1995.

[7] P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

[8] T. Dybå, Improvisation in Small Software Organizations, *IEEE Software,* 82-87, Sept./Oct. 2000.

[9] J. Highsmith and A. Cockburn, Agile Software Development: the Business of Innovation, *Computer.* 34:(9):120-127, Sept. 2001.

[10] N. Juristo and A.M. Moreno, *Basics of Software Engineering Experimentation,* Springer, 2006.

[11] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, S. G. Woods, "Experience with Performing Architecture Tradeoff Analysis", in *Proc. ICSE 21*, May 1999, 54-63.

[12] M. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzalez Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic, 1993.

[13] P. Manhart and K. Schneider, Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report, in *Proc. ICSE 26,* 378-386, 2004.

[14] mod_perl, http://perl.apache.org/, 2006.

[15] J.R. Nawrocki, B. Walter and A. Wojciechowski, Comparison of CMM Level 2 and eXtreme Programming, in *Proc. 7th International Conference on Software Quality,* 299-297, LNCS, 2002.

[16] *WebArrow*, http://www.*WebArrow*.com, 2006.