Microcosms: A Web Server in Every *Software* Component

Kevin Sullivan
Department of Computer Science
Thornton Hall University of Virginia
Charlottesville VA 22903 USA.
Tel: +1 (804) 982 2206, FAX: +1 (804) 982 2214
sullivan@virginia.edu

Avneesh Saxena
Department of Computer Science
Thornton Hall University of Virginia,
Charlottesville VA 22903, USA.
Tel: +1 (804) 982 2048, FAX: +1 (804) 982 2214
avneesh@cs.virginia.edu

ABSTRACT
We present a new dimension in software architecture: the systematic embedding of web servers into runtime software components to provide a highly leveraged, scalable, secure mechanism for accessing, monitoring and controlling systems through their runtime architectures. Applications could include remote debugging; distributed management; spider indexing of computations to aid understanding and evolution; survivability control; runtime analysis using web-encoded metadata; web-based open implementations; and aspect-oriented runtime architectures. A simple system suffices to show the feasibility of and to illustrate this idea.

Keywords
Software, architecture, internet, embedded web server

1 INTRODUCTION

We demonstrate that it is possible to open the architectures of running computations to internet-based access by embedding web servers into runtime software components. This idea appears to create interesting new possibilities for software design. The power of the approach springs from at least three fundamental sources.

First, the method immediately permits the enormous power of internet technologies to be used to create rich web-based interfaces to access, monitor, control and augment running computations at the architectural level. Second, the embedding is orthogonal to other aspects of design, which means that it can be applied to many kinds of components. Third, the approach follows the information hiding design principle: server design decisions are hidden within the modules to which they pertain. We have just begun to explore applications. This paper introduces the idea and a simple system that shows that it is feasible, and that gives a flavor of the rather unusual opportunities that it creates.

The rest of this report is organized as follows. Section 2 describes our test-bed system. Section 3 describes how we implemented the approach in our test-bed. Section 4 discusses several web-based interfaces in more detail. Section 5 discusses additional possibilities; and Section 6, related work. Section 7 addresses implementation issues and limitations of our prototype. Section 8 concludes.

2 TESTBED APPLICATION

Our test-bed is a program called *SwitchSet*. It is written in Java [26] and runs under Microsoft Windows NT 4 [18]. A *SwitchSet* computation maintains a set of *Switch* objects. A *Switch* maintains a single bit that is either *on* or *off*.

SwitchSet is based on a Java object of class ***SwitchSet***. The ***SwitchSet*** class defines two important operations in its primary interface: *Add* and *Delete*. *Add* takes a reference to a *Switch* and adds it to the set unless it is already a member, in which case the operation has no effect. *Delete* takes a *Switch* and deletes it if it is a member, and otherwise has no effect. A *Switch* is implemented by a Java object of class ***Switch*** having three key operations, *TurnOn*, *TurnOff* and *GetState*. They act as their names suggest.

To enable interactive manipulation of the set of switches, the *SwitchSet* application presents a graphical interface, as depicted in Figure 1. The *Create* button creates a new *Switch* and calls *Add* to add it to the set. The list-box displays an entry for each *Switch* in the set. The user can select an entry and choose operations to perform on it by "right clicking." If an entry is selected and the *Delete* button is pushed, the *Switch* is deleted from the set. If the *TurnOn* button is pushed, the selected *Switch* is turned on. If *TurnOff* is pushed, the selected *Switch* is turned *off*. *Show State* displays the state of the selected *Switch*.

The *SwitchSet* system abstracts any modular computation and thus provides a good vehicle for testing the feasibility of our approach and for exploring its uses. The highly abstract nature of this system ensures no loss of generality.

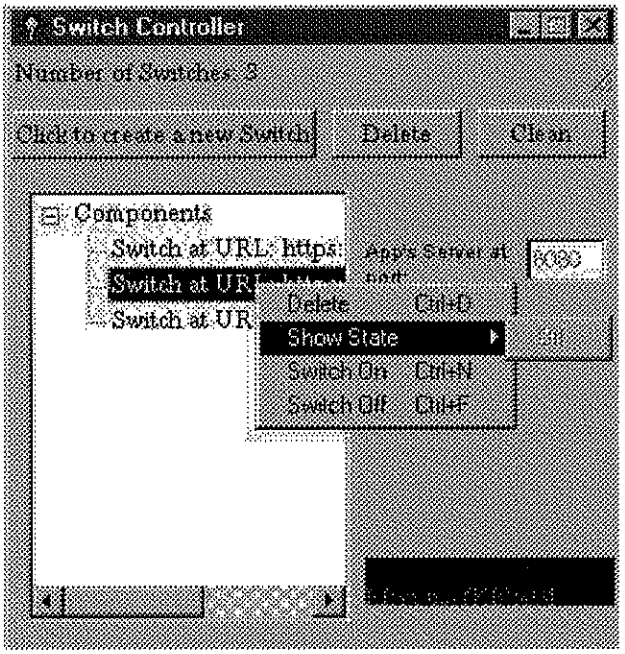

Figure 1 User interface to the *SwitchSet* application

3 SOFTWARE EMBEDDING OF THE INTERNET

We did succeed in weaving the internet into our test-bed system. In this section, we explain how. In the next section, we discuss some capabilities that we have demonstrated based on this novel embedding.

In a nutshell, we add an instance variable of a *WebServer* type to any arbitrary class. At object creation time, we initialize such embedded servers. In our implementation, the server initialization routines requires references to their encapsulating objects, domain names, and the names of several directories. We create domain names dynamically by concatenating host names and free TCP port numbers. The servers are initialized to listen for HTTP requests on these ports. The encapsulating objects implement general callback interfaces that servers use to implement their operations (e.g., Get). See Figure 2. These callbacks can perform arbitrary functions, such as the dynamic generation of HTML pages to be returned to web clients.

In our system, the main *SwitchSet* object and each *Switch* object has its own server. When started, the *SwitchSet application* prints the domain name of the server on the standard output. The user can then access the server using a web client such as Microsoft's Internet Explorer [6] or Netscape [20]. Among other things, the *SwitchSet* server can generate HTML pages containing hyperlinks to HTML pages served by the servers within the *Switch* objects.

In more detail, we implemented *Switch* as a Java class that conforms to Microsoft's Component Object Model (COM) standard [10]. Figure 3 presents part of the type definition. *Switch* implements the *WebServerEventHandlers* interface

```
public interface WebServerEventHandlers
{
public void onGet(Object, GetEvent);
public void
onAuthenticate(Object,AuthenticateEvent);
public void onHead (Object, HeadEvent);
public void onPost(Object, PostEvent);
public void onPut(Object, PutEvent);
public void
onRequest(Object,RequestEvent);
public void
onSessionEnd(Object, SessionEndEvent );
public void
onSessionInit(Object, SessionInitEvent);
}
```

Figure 2 *WebServerEventHandlers* Interface

to define server callbacks. These callbacks, or event handlers, provide object-specific implementations for HTTP and other events. In our case, these handlers allow *Switch* objects to create responses and to serve up static and dynamic web pages and to take other actions.

The *rootDir* and *logDir* instance variables are the directory names used to initialize the embedded server. Here they refer to physical directories. Object-specific virtual directories would be better in some cases. The *SourceCode* instance variable is used by one of the callbacks to create an HTML page that presents the source code for the *Switch*. The *webServer* variable is the embedded web server itself. *State* maintains the Boolean *Switch* state. The *clientApplets* variable is used to implement a notification service for applets that are uploaded from the *Switch* to client browsers (discussed later). The *Form* (a dummy object) is needed because the *WebServer* is based on Microsoft's ActiveX®, and every ActiveX® object needs a parent Form. It is not used. The *auxPortNumber* variable stores the TCP port number that is used to create the unique domain name for the object-specific embedded server.

```
public class Switch implements
WebServerEventHandlers {
String rootDir =
"..\\switch\\switchPages\\web";
String SourceCode = "..\\switch";
String logDir =
"..\\switch\\switchLogs";

private WebServer webServer;
private int State;

private Hashtable clientApplets;
private Form form;
private int auxPortNumber;
```

Figure 3 State variables for a *Switch*

```
Package dartwebserver;

// Dual interface IWebServer
/** @com.interface(iid=E0E04097-A0F5-
11D3-9114-00105A17B608, thread=AUTO,
type=DUAL) */

public interface IWebServer extends
IUnknown
{
       /** @com.method(vtoffset=4,
       dispid=1, type=PROPGET,
       name="ClientAuthentication",
       name2="getClientAuthentication",
       addFlagsVtable=4)
      @com.parameters([type=BOOLEAN]
return) */
public boolean getClientAuthentication();

...
}
```

Figure 4 Delegating interface for PowerTCP® web server

Our *WebServer* type is implemented on Dart Technology's library for web development: the PowerTCP® Internet Components [7]. PowerTCP® components are ActiveX® enabled and support the HTTP and secure HTTP (HTTPS) protocols. *WebServer* objects are essentially just wrappers that initialize PowerTCP® server objects. To enable secure communications, the name of the certificate to be used when authenticating can be given. On creation, a *WebServer* instantiates a PowerTCP® server and chooses certificates for authentication.

Accessing member functions of the PowerTCP® web server requires referencing the underlying ActiveX® object, which is implemented as a dynamic link library (DLL). Visual Java provides wizards [17] that allow automatic creation of delegating classes with functions that make native calls to the DLL. See Figure 4. Delegating classes can be used transparently in Java Code to access functions provided by DLLs, avoiding the complexity of writing Component Object Model (COM) code.

An object that exposes a web interface in our style actually implements four interfaces: *native, server management, server callback,* and *web.* We make this idea concrete by explaining how it pertains to the *Switch* type:

- Native interfaces (in our application based on COM) define "normal" component functions:
  - *a) void turnOn ()*: turn the *Switch* on;
  - *b) void turnOff ()*: turn the *Switch* off;
  - c) *String getState ()*: get the state of *Switch.*
- Server management interfaces defines functions that can be used to manipulate the embedded web servers:
  - *a) String getURL ()*: get the embedded server URL;
  - *b) void shutDown ()*: shut down the web server.

We designed our native and server management interfaces as COM interfaces, for use by the *SwitchSet* application. The *Switch* class is packaged as a DLL using the Visual Java wizards [17] so that any client able to use COM objects can access it. The use of COM is not essential, of course. We could embed servers in C++ or Java objects, in CORBA objects or in many other kinds of components.

- We already described the server callback interface.
- Finally, web interfaces define functions exposed to internet-based clients by embedded web servers.

An important aspect of our approach is that web interfaces are orthogonal to native interfaces. This is the source of its generality: it enables the orthogonal embedding of the Internet into arbitrary software architectures. We designed the web interfaces of our experimental system to show some of the opportunities that our approach creates. We now discuss our specific web interfaces in more detail.

4 PROGRAMMING THE WEB INTERFACE

Our approach enables the exposure of arbitrary web interfaces by runtime components. The approach provides great leverage because it admits the full power of existing and emerging internet technology into virtually any component. We present a few simple examples. We programmed our *Switch* component to allow authenticated users to securely

a) inspect the runtime state of a *Switch;*

b) use its native interface to change its state;

c) view its source code implementation; and

d) view its native and server management interfaces.

To ensure security we made the web interface of *Switch* components accessible only through HTTPS. We left the *SwitchSet* web interface insecure. We implemented security using existing public-key infrastructure. We created our own certificate authority (CA) and configured the servers within our *Switch* components to accept certificates from this CA. Every connecting client is required to authenticate itself to the *Switch* and can in turn ask for authentication from the *Switch.* To demonstrate the creation of fine-grained security domains, we created three classes of users:

a) *Administrators* have full access to the functions that are exposed through the *Switch* web interface;

b) *Developers* may not modify the state of a *Switch;*

c) *Guests* may not modify the state and are prevented from viewing the source code for the *Switch.*

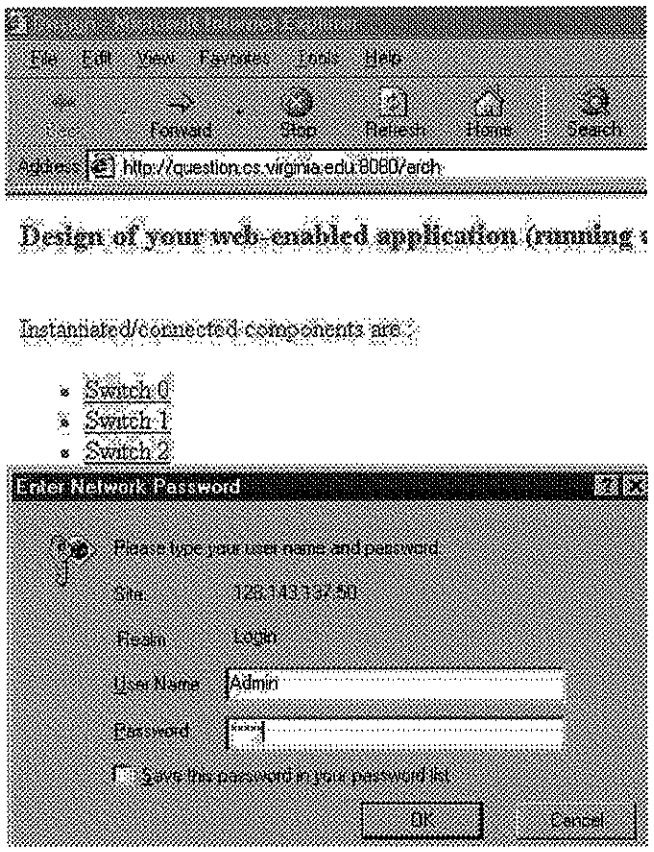

Figure 5 Secure access to a *Switch* within the *SwitchSet*.

Each component has to create its own domain to allow the embedded web server to bind to that particular domain. Dynamic creation and destruction of URLs for servers as enclosing components are created and destroyed supports the creation of unique domains for components.

The Web Interface

The web interfaces of *Switch* and *SwitchSet* support serving of static and dynamic HTML from the embedded servers. Our approach thus creates a mechanism for secure exposure of arbitrary component meta-data on the global internet. This data could include runtime state; specifications in various forms, such as source code; control interfaces; etc.

The *SwitchSet* interface provides services similar to those of a *Switch*. It also provides access to hyper-links to pages served by *Switch* objects. See Figure 5. These hyper-links can be used to connect to the domains of *Switch* components. We thus have a powerful mechanism for browsing running computations at the architectural level.

Moreover, such browsing is authenticated and secure. Our *Switch* uses a password-based authentication scheme running on HTTPS for access control. Clients authenticate themselves by presenting correct certificates. They are then required to enter valid login-name/password combinations before access to the web interfaces is given. See Figure 5.

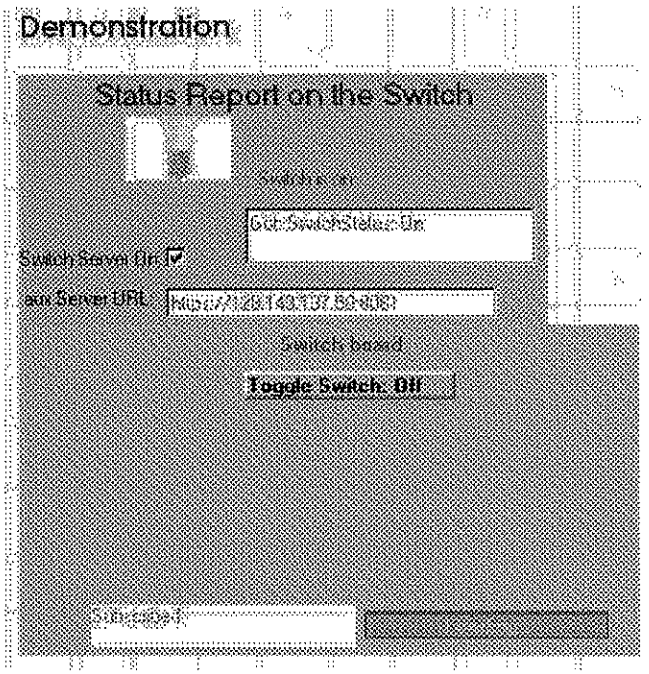

Figure 6 Web-based control Panel for *Switch*

The main web page of a *Switch* component allows access to services based on the identity of the user as specified by the login name. We have implemented a prototype interface that allows users to access or modify the state of the *Switch*, as depicted in Figure 6.

This interface also allows a user to observe changes in the state of the *Switch* as they occur. The interface receives secure notifications from the *Switch* as its state changes. In response, the light bulb icon is set to light or dark. Here again we exploit existing Internet technology to implement a sophisticated Internet-based function at a fairly low cost.

This notification service is implemented on the server *(Switch)* side by having each *Switch* maintain a list of clients that are accessing its web interface. The client (browser) side also implements a secure server that listens for notifications. This server is implemented as a custom ActiveX® component that displays the bulb icons. The other part of the client-side interface is a Java applet that is used to make connections with the parent *Switch* for modifying its state through the *Toggle* button. The applet is uploaded from the *Switch* after the ActiveX® component is uploaded. The applet is given the URL of the server running in the *Switch* component as a parameter. On being loaded, the applet contacts the *Switch* and *subscribes* for notifications. Communication between the applet and the client-side ActiveX® component is implemented using VBScript®. This primitive notification service allows all clients to be made aware of any state changes in the *Switch*. Furthermore, all such communication is authenticated and cryptographically secure.

We have also implemented services that allow clients to view the *Switch* class source code, and to view the COM-interfaces of the *Switch* and *Switch* documentation. These services are implemented by serving static HTML pages.

5 SOFTWARE-EMBEDDING OF THE INTERNET

Our society already relies on complex, distributed software systems. Today, they run everything from automobiles to critical infrastructures: banking and finance; transportation; energy production, transmission and distribution; etc. It has become clear that the software element of many of these systems is close to being out of control. The design, deployment, mapping, analysis, monitoring, control, adaptation and evolution of such software present huge risks and enormous intellectual and technical challenges. As we move into a world of ubiquitous, component-based and invisible computing, these problems will only grow.

Regaining some measure of intellectual control over large and complex software systems requires new concepts in software architecture. We hypothesize that our approach, based on the idea of architectural embedding of the internet into running computation as a means to open them up to secure architecture-level monitoring and control from anywhere on the internet, has the potential to contribute to ameliorating some of these difficulties. In this section, we survey some ways in which our approach might help.

- **Runtime component indexing, search & retrieval**

One of the problems that we face is to know precisely what components are present in a running system. This problem is likely to worsen as systems of systems grow in size and complexity, and as they evolve through module-level replacement initiated by diverse and uncoordinated third party providers. We need powerful, scalable methods to map large, complex, distributed running computations. Our approach allows the use of such things as web crawlers to create architectural maps of running computations.

To validate this concept, we obtained an evaluation copy of the Ultraseek [30] web crawler and applied it to a running *SwitchSet* application. Ultraseek can spider a network at configurable intervals starting at any URL. It builds a database of accessible URLs. Queries can be made to this database. The evaluation version doesn't permit access to sites running HTTPS, so we could not index the web pages of contained *Switch* components. Figure 7 illustrates the results of our experiment: a web page containing links to "live" pages served by components of the running system.

This technique can be applied to mapping global state. The traditional approach to global state acquisition uses ad-hoc techniques. We can now use web crawlers or robots to collect and search state with standard technology. State information served in XML form can support rich global dynamic analysis. Such information might be utilized to detect, debug or prevent deadlocks; to log state transitions, intrusion attempts, load information; etc.

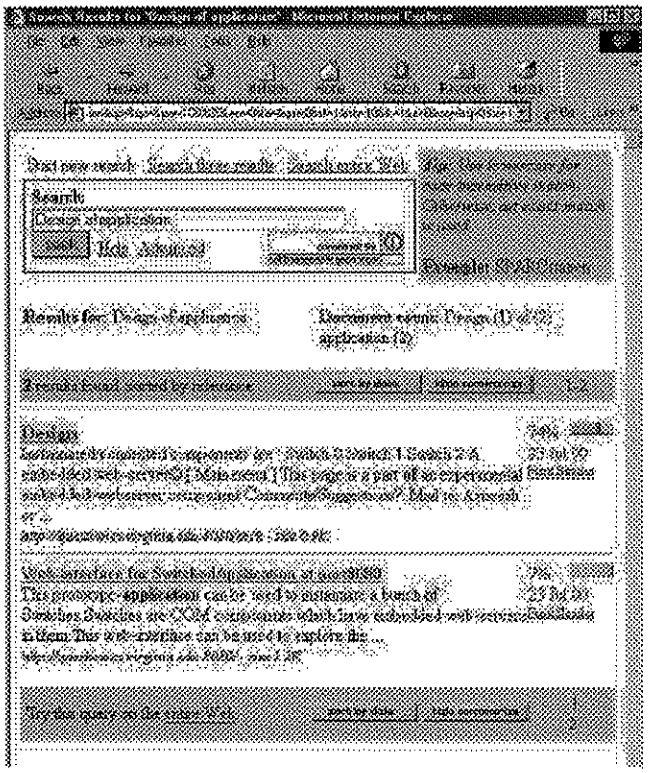

Figure 7 Searching for *SwitchSet* application web pages

A drawback of this approach is its inability to provide globally consistent information about all component states. However, the approach can clearly scale to huge numbers of distributed components. We suspect that information search and retrieval for components of large distributed computations, even without consistency, can provide valuable information and system management capabilities.

- **Distributed Debugging**

Designing and debugging distributed computations is hard, in part because of synchronization issues. Components are often developed and debugged locally and then deployed as parts of a distributed system. However, latent faults can manifest themselves in new execution environments. Thus, components for use in distributed systems should be developed in distributed environments. Many distributed debugging algorithms have been designed to detect *stable* [3] or *unstable properties* [5] that depend on local variables [28].

We hypothesize that embedded servers that expose server-side debugging functions can ease the implementation of such algorithms: remote debugging can increase the chance of catching obscure bugs. A familiar web-based interface can also save learning time. Embedded debugging permits component to be used in diverse systems in tool- and language-independent ways. Control over deployed components might also allow changing of state variables and, hence, some kinds of fault recovery.

Embedded debuggers can be limited to inspecting and modifying local state. Distributed computations, in general, require tools to view and modify consistent, global state, which is known to be NP-complete [4]. We hypothesize that even such limited functionality can be useful for debugging distributed systems that can't be handled by current tools because of the complexity of debugging algorithms. We will develop this idea in future work.

- **Reflection and Introspection**

Reflective [16] systems can process their own state at a meta-level and take corresponding actions. Distributed systems can use reflection to modify their own behavior at the functional level. Modifications can vary from migrating components between machines for load balancing to the dynamic replacement of component implementations for on-the-fly performance improvement.

Our architectural framework can be used to implement the convenient exposure of rich component data to support computational reflection. XML, in particular, is emerging as a dominant standard for component communication. A uniform framework to expose functional-level information to the meta-layer provides easy integration and reuse of meta-level components. Self-describing XML encoding of state can present such a uniform framework. One potential application is for components to carry their own code and for embedded debuggers to act as introspective interpreters.

- **Runtime aspect modules**

Aspect-oriented programming (AOP) [13] is based on the idea that in general there is no modularization of the source code of a software system that isolates all important concerns: any modularization will be cross-cut by some concerns. The problem is that cross-cutting distributes and thus obscures and demodularizes key abstractions, making it harder to design, reason about and change programs.

The problem is inherent in any view of source code as a *linear text.* The problem can be reformulated by stating that there is no linear ordering of important concerns in a complex software system. The approaches to the aspect problem being taken today [15] are based on the idea of representing programs as (what we will call) *multi-texts.* A multi-text is a collection of texts that meet not just at external boundaries but that inter-penetrate each other. A *weaver* tool takes such texts as input and integrates them into linear, and thus tangled, forms. The benefit is that the approach restores a semblance of modularity in design. A system is again presented as a set of text modules: some traditional (e.g., functional modules), others, *aspects,* that inter-penetrate the linear texts (e.g., concurrency control). The extent to which aspect modules can be understood or implemented independently is unclear. At the heart of this difficulty is the subtle shift from text to multi-text as a medium for writing source code modules. How does one read a multi-text?

Our approach suggests a new perspective. The idea that we share with AOP is that, in any modular system, important concerns will cross-cut the dominant modularization. Our perspective departs from current views of AOP in two ways. First, it pertains not to source code but to runtime computations. Second, it is based not on multi-text but on hypertext, and web mechanisms more generally, for integrating physically dispersed but conceptually coherent cross-cutting information back into pseudo-localized forms.

We represent cross-cutting concerns in running systems as *runtime aspect modules* whose embedded web servers use web mechanisms to integrate information that is otherwise distributed over multiple runtime system modules. Figure 5 presents a simple example: an integrated (aspect) view provides access to all of the switches in the system. A more sophisticated example might present all of the performance tuning or intrusion or fault detection interfaces of all system components in a unified, web-based interface.

We do not yet fully understand the relations between multi-text and the web as media for representing aspects. The power of the web lies largely in its ability to represent arbitrary non-linear views of a text. The modules of a multi-text are hard-wired to represent a single linear form. Combining the aspect idea with the choice of the web and the embedded web server mechanism has attraction as an technology to aid in managing distributed and component-based systems. We will explore this idea in future work.

6 RELATED WORK

We now situate our work in the context of related efforts.

Component models for distributed systems

Several major architectures now support component-based distributed system development. CORBA [21] and DCOM [29] are two that have gained wide acceptance. Both are meant to give operating system and programming language independence, allowing separately implemented systems to interoperate. However, one of the important issues that they don't address is providing a management framework. Our approach is orthogonal to such architectures and thus can be used to extend their capabilities. Our approach can be viewed as a dimension of an architectural style [8] that is largely orthogonal to existing component architectures.

Web-based management

As the Internet continues to expand, the need for standards for managing network elements has become necessary. SNMP [2] is a standard that has been widely accepted and deployed by the industry. An important feature of SNMP is simplicity. However, while vendors have embedded SNMP support in network devices, there has been no uniform way to access, manage and present this information to the user (network administrator). Web-based service [22] and network management [11][14] mechanisms provide intuitive, web-based interfaces to services such as SNPM to alleviate the difficulty of using disparate, unfamiliar tools.

To date, this approach has been limited to managing internet infrastructure and some specific services. There have been related efforts to develop analogous management information bases (MIB) for web servers, E-commerce services etc, and some widely distributed web servers do have web-based interfaces to support server configuration. Our approach is more general: to support web-based access to arbitrary software components. Our approach does not assume any standard underlying management infrastructure like SNMP, but it could use any that emerge. Instead, ours is a general *architectural dimension* for software design.

Embedded and lightweight web servers

Embedded systems have limited resources. A new approach for managing varied hardware devices has been to embed web-servers in them. Many research teams in both industry and academia are developing lightweight hardware servers that can be embedded at low cost. However, the work has concentrated on hardware servers for hardware devices.

One result of such research has been the development of low-cost web servers and protocol implementations with low memory and physical space requirements. Increasingly, such servers are being used to provide powerful web-based management interfaces, e.g., for modems. Our approach is similar in philosophy. The fundamental difference, and our key insight, is that we seek to embed *software* web servers systematically into the *software* components of distributed and other software systems, not just into hardware devices.

Web server components that can be embedded in software are already available in market (e.g., by Dart Technology). However, to the best of our knowledge their use has been limited to single servers used in to expose application-level information, such as configuration functions and, mainly, on-line documentation. Moreover, the idea of exploiting the systematic embedding of web services into software systems at the architectural level does not seem to have appeared previously in the open literature.

SOAP and XML-RPC

It is now being realized that current web-infrastructure is not sufficient to develop and deploy next-generation web applications. One major limitation is the absence of any common protocol for communications among web-enabled applications. Efforts to develop such standards based on the existing web-infrastructure are underway. SOAP [1] and XML-RPC [31] are two efforts to define RPC mechanisms that use XML-based messages that are gaining acceptance.

Such standards support interactions between web sites. These mechanisms really define component models in which web servers (or at least XML-based RPC stubs) are the components and XML-based messages encode procedure calls. These mechanisms enable web servers to wrap application modules to expose their functions on the internet. Our approach is the inverse: application modules wrap web servers. One potential problem with the SOAP approaches is that it forces the designer to adopt a web object model at the highest architectural level, which might be inappropriate. Web communication mechanisms are often costly and unreliable, for example. A benefit of our style is that the designer can have a native architecture appropriate to the task (e.g., DCOM, CORBA, Legion [9]) and yet to benefit from rich component interfaces on the Internet. Of course, components in our style could use SOAP to communicate with each other through their web interfaces; but SOAP need not dominate the architecture.

Open Implementations

Open implementations [12] allow clients to modify module implementations strategies to match performance properties to client needs. Components using this approach should preserve *black-box* implementation as far as possible. Distributed systems have to service varied clients and are prime candidates to be implemented in an open manner. Distributed systems are also reconfigured on account of component failures, system-load etc. The research community has recognized importance of open implementations and guidelines have been proposed for developing components in this style.

Our architecture represents a new mechanism to view and deploy open implementations. An *web-opened computation* can expose its implementation and control over it through its web-interface. Access can be provided and controlled securely. Such data as predicted and actual performance characteristics of components could be presented, for example, which could help clients to make choices about implementation strategy. We are exploring the use of web interfaces to change implementation strategies dynamically. Open implementation combined with embedded servers and on-line, web-based performance analysis mechanisms could significantly aid distributed system management.

7 IMPLEMENTION LIMITATIONS

Embedding web servers into other software components can incur unacceptable performance costs. An embedded device with little memory cannot afford to host components with large memory footprints. A quick assessment of the size of the PowerTCP® web server component shows that each instance has a space cost of about 156,000 bytes.

To date, we have not tried to reduce this cost at all. First, Moore's Law puts an increasing premium on finding useful new ways to use the exploding resources that are becoming available. Second, web servers developed for embedded devices already take no more than a few hundred bytes to implement HTTP. In principle, we can trade function (e.g., security) for space, getting down to this level of overhead. We are now considering building small and portable but extensible componentized web servers to let us apply our approach in a broad range of operating environments.

We also note that embedding web functions in components can be implemented in several ways. First, all components

on a machine could share a common server. This approach suffers from security, failure propagation, and information hiding problems. The information hiding issue is critical. A single server (including its programmed extensions) has to "know about" all objects that it serves and all such objects have to know about it, requiring a level of coordination among component designers that is increasingly untenable. A server in each object achieves modularity in a way that decouples design decisions for servers and their programming for different object classes. Such decoupling will be critical when components are designed and produced by independent firms.

Second, each component can have a distinct server process (e.g., Unix process). This approach is costly because processes still are costly. Third, components can have local state but share common code. Our implementation uses shared code DLLs with per-component data separate. Fourth, servers can share state. For example, all objects of the same type can share one copy of its source code.

Ideally, web interfaces can be added transparently to components using our approach. However, such additions can affect native functional and non-functional properties adversely if the embedding is not done with care. For example, executing web-based functions could hurt the performance or security of functions on native interfaces.

The web servers in our prototype use the same threads as their surrounding components. We use the servers in non-blocking mode and our entire application is event-driven. Events are generated by user actions on the *SetSwitches* graphical user interface or by HTTP requests. Our single threaded implementation blocks events until the one that is currently being processed is handled. This can lead to slow response times and unexpected application behavior. In particular, it opens our application to web-based denial of service attacks. Such problems can be resolved to some extent by threading of event handlers and by the use of applicable internet tools and methods. Of course, threading event handlers requires that application objects be designed to be thread-safe, with appropriate concurrency control.

A final limitation of our test-bed program that we want to mention is that it is is not portable. It is built to COM specification, and can only be used on Windows platforms. The *Switch* component is a DLL that limits its execution to the Windows environment. The web pages served by *Switch* components use VBScript® and object HTML tags that are specific to Internet Explorer. The Dart Technology libraries run only under Windows. The lack of portability of our system and its underlying technology is an issue in practice, but not in principle. In future work we will address portability to the extent that it becomes important.

8 CONCLUSION

We have described and demonstrated the feasibility of what appears to be a simple but novel, orthogonal and potentially powerful new dimension in software architectural design. The approach involves the embedding of a web server in virtually any component of virtually any computation. It enables computations to be extended with rich data, and it opens opaque, insular computations to access, monitoring, control from anywhere on the internet. We have shown that sound access control policies can be implemented using available public key infrastructure; and we discussed a variety of ways in which the basic idea could be used: to serve source code, applets for active monitoring, web-based invocation of component functions; and web crawler mapping and searching of running computations. The embedded web server idea appears to create an extremely broad range of valuable new opportunities, and a new way thinking about systems, as having "on-line" architectures.

We will pursue work in several directions. First, we will continue to explore basic capabilities made available by our approach, such as runtime analysis based on web-served component meta-data. Second, we will apply the approach to practical—not just laboratory—applications in order to better understand its potential utility. Third, we will explore using embedded web services as sensors and actuators in survivability control systems [25], in particular.

Finally, we will investigate a major generalization of our approach. The idea of systematically embedding web servers in software component rests on two assumptions. First, computations (rather than, say, source code) are the real first-class citizens in computing systems, so we should be able to converse with them in much richer ways than is possible today. Second, we increasingly find ourselves an era of untold computational riches. We can *afford* design strategies that would have been unthinkable even recently.

Making this *exploding resources* assumption explicit, and combining it with the view of computations and their component parts as being primary, suggests a new question: what else should we be embedding in runtime components? A radical answer is *everything that's relevant.* We imagine that there is potential utility in embedding in running components debuggers; source code and other such design specifications; specialized virtual machines for running encapsulated implementations; compilers; version control mechanisms; scaffolding to support evolution, e.g., state migration and implementation replacement; whole software development environments; systems for on-line interaction with developers, other clients, media sources, markets for upgrades or substitutes; etc.

This view inverts the traditional, deeply held, assumption that computations are best left opaque: with internal states and transitions hidden from clients behind abstract interfaces, and from developers behind compilers. This view also changes how we think about components: not as specialized abstract computing machines, but rather as secure, on-line software *microcosms.* The extent to which such a view is feasible, useful or wise remains to be seen.

ACKNOWLEDGEMENTS

This work was supported by a grant from the National Science Foundation, number CCR-9804078.

REFERENCES

[1] Box D. SOAP: Simple Object Access Protocol. *HTTP Working Group Internet Draft*, September 1999.

[2] Case J, Fedor M, Schoffstall M, Davin C. The Simple Network Management Protocol (SNMP), *RFC—1157*, May 1990.

[3] Chandy K, Lamport L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, vol.3, no.1, Feb. 1985, pp.63-75. USA.

[4] Chase C. and Garg VK. On techniques and their limitations for the global predicate detection problem. *In Proc. of the Workshop on Distributed Algorithms,* pages 303 -- 317, France, Sept. 1995.

[5] Chase C, Garg VK. Detection of Global Predicates: Techniques and their Limitations, *Distributed Computing, (accepted for publication with revisions).*

[6] Crawford Caison C Jr. More Web power with VB6 and IE5. *Visual Basic Programmer's Journal*, vol.9, no.1, Jan. 1999, pp.32-4, 36, 38, 41. Publisher: Fawcette Technical Publications, USA.

[7] Dartcom Inc. PowerTCP WebServer Tool. [Online document], *(http://www.powertcp.com/).*

[8] Garlan D, Allen R, Ockerbloom J. Exploiting style in architectural design environments. *Sigsoft Software Engineering Notes*, vol.19, no.5, Dec. 1994, pp.175-88. USA.

[9] Grimshaw AS, Wulf WA. Legion - A view from 50,000 feet. *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (Cat. No.TB100069).* IEEE Comput. Soc. Press. 1996, pp.89-99. Los Alamitos, CA, USA.

[10] Jones K. The world of COM (Component Object Model). *Enterprise Middleware,* Feb. 1999, pp.2-8. Publisher: Xephon, UK.

[11] Jong-Tae P, Jong-Wook B. Web-based Intranet/Internet Service Management with QoS Support, *IEICE Transactions on Communications*, Vol.E82-B, No.11, 1808-1816, November 1999.

[12] Kiczales G, Lamping J, Lopes CV, Maeda C, Mendhekar A, Mnrphy G. Open implementation design guidelines. *Proceedings of the 1997 International Conference on Software Engineering,* ICSE 97. ACM. 1997, pp.481-90. New York, NY, USA.

[13] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J. Aspect-oriented programming. *ECOOP '97 - Object-Oriented Programming. 11th European Conference Proceedings.* Springer-Verlag. 1997, pp.220-42. Berlin, Germany.

[14] Leidigh C. Web-based Management of Network devices, *Proceedings of the Embedded Systems Conference* (Spring), Miller Freeman. Vol.1, 1998, 1-17 vol.1. San Francisco, CA, USA.

[15] Lopes C V, Kiczales G. Recent Developments in AspectJ™. *ECOOP'98 Workshop Reader*, Springer-Verlag, LNCS-1543. Copyright 1998 Springer-Verlag.

[16] Maes P. Computational reflection. *The Knowledge Engineering Review*, vol.3, no.1, March 1988, pp.1-19. UK.

[17] Microsoft Corp. Building and importing activeX® controls, *Visual J++ Documentation, MSDN (http://msdn.microsoft.com/).*

[18] Microsoft Corp. Windows NT 4.0. [Online document]. (http://www.microsoft.com/ntserver).

[19] Murugesan S, Deshpande Y, Hansen S, On Exploiting the Internet and Web for Software Development and Distribution, *Proceedings of IEEE International Conference on Networking India and the World,* Ahmedabad, India, December, 1998.

[20] Netscape Communications Corp. Netscape Navigator [Online document], (http://www.netscape.com/).

[21] Object Management Group Inc. (OMG). CORBA 2.3.1/IIOP Specifications. [Online document]. October 1999. (http://www.omg.org/technology/documents/formal/).

[22] Qinzheng K, Chen G, Hussain R Y, A Management framework for Internet Services, *IEEE Network Operations and Management Symposium, Conference Proceedings*, IEEE. Part vol.1, 1998, 21-30 vol.1. New York, NY, USA.

[23] Shaw M, Orna R, An Approach to Preserving Sufficient Correctness in Open Resource Coalitions, *Submitted to 10th International Workshop on Software Specification and Design.*

[24] Sobel JM, Friedman DP, An Introduction to Reflection-Oriented Programming, *Reflection '96*, San Francisco, April 1996.

[25] Sullivan KJ, Knight KC, Du X, Geist S, Information Survivability Control Systems, *Proceedings of the 21st International Conference on Software Engineering,* May 1999, pp. 184—193.

[26] Sun Microsystems Inc., Java Language Specification, 2nd Edition, (http://java.sun.com/aboutJava/).

[27] Takagi H, Matsuoka S, Nakada H, Sekiguchi S, Satoh M, Nagashima U. Ninflet: a migratable parallel objects framework using Java. *Wiley. Concurrency Practice & Experience*, vol.10, no.11-13, Sept.-Nov. 1998, pp.1063-78. UK.

[28] Tarafdar A, Garg VK. Debugging in a distributed world: observation and control. *Proceedings. 1998 IEEE Workshop on Application-Specific Software Engineering and Technology.* ASSET-98 (Cat. No.98EX183). IEEE Comput. Soc. 1998, pp.151-6. Los Alamitos, CA, USA.

[29] Thompson D, Watkins D. Comparisons between CORBA and DCOM: architectures for distributed computing. *Proceedings. Technology of Object-Oriented Languages.* TOOLS 24 (Cat. No.97TB100240). IEEE Comput. Soc. 1998, pp.278-83. Los Alamitos, CA, USA.

[30] Ultraseek, Ultraseek Server, [Online document], *(http://www.ultraseek.com)*.

[31] Userland Software Inc. XML-RPC Specification. [Online document]. *(http://www.userland.com/spec)*.

[32] Welch B, Uhler S. Web enabling applications. *Proceedings of the Fifth Annual Tcl/Tk Workshop.* USENIX Assoc. 1997, pp.189-90. Berkeley, CA, USA.