

# A Web-Based Survivability Control Architecture

Kevin Sullivan and Avneesh Saxena

University of Virginia Department of Computer Science

Charlottesville VA 22903, USA

[sullivan@virginia.edu](mailto:sullivan@virginia.edu); [avneesh@cs.virginia.edu](mailto:avneesh@cs.virginia.edu)

## 1 INTRODUCTION

We have begun to investigate a new architectural model for survivability monitoring and control of complex distributed information systems, such as those that underpin critical civic and other infrastructures. The model is a synthesis of two streams of thought. The first was presented in “Information Survivability Control Systems [4].” That work suggests an approach to dynamic survivability management based on decentralized, hierarchical, adaptive control systems superimposed upon information systems to detect and respond to complex faults—e.g., coordinated security attacks—to help ensure continued satisfaction of survivability requirements.

The second idea, presented in “Microcosms: A Web Server in Every Software Component [3],” involves potentially every software component of a complex application embedding web servers to provide secure, world-wide access to component-level meta-data and services *without components having to conform to demanding design rules or implementation strategies*. Services could include controlled access by authenticated parties to component state and reflective meta-data, including runtime and design history; open implementation control of implementation strategies; component-specific programming tools; pointers to markets for replacement parts; links to collaborating components, etc. This idea transcends the notion of component as specialized computer to embody one of component as rich computational world within which such a computer operates and is supported.

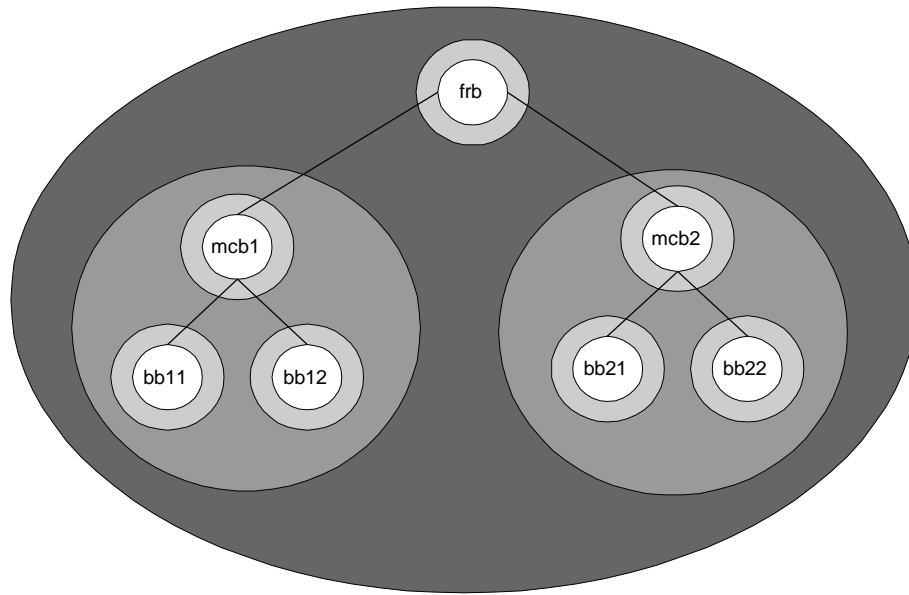
Augmenting components with embedded server implementations achieves a level of independence in component architecture and implementation that competing approaches do not achieve. The approach is almost completely orthogonal to and compatible with any native architectural design, such as the use of DCOM, CORBA or SOAP architectural styles. The approach does not require components to conform to demanding, standardized design rules. Each component need only have one method, such as *GetURL()*, by which a handle to its arbitrarily rich web-based interface is obtained. This independence characteristic makes the approach potentially valuable for secure, global access to the inevitably heterogeneous and legacy elements of many such systems.

The synthesis is the idea that such web-enabled components might provide an architectural basis for survivability control systems for heterogeneous, distributed infrastructure systems. This idea is not mature, but appears to warrant investigation. For this workshop our position is that *it is an intriguing idea with features that make it worth investigating*. We review the two concepts, the synthesis and a proof-of-concept system.

## 2 SURVIVABILITY CONTROL SYSTEMS

The survivability control systems concept addresses the need to keep complex, distributed, often cross-administrative-domain information systems that automate critical infrastructure systems under control in the face of complex runtime disturbances, such as power outages, hardware failures, component design faults, security attacks, and so forth; and, furthermore, that control theory [4] provides a useful vocabulary for thinking about how to keep systems operating as desired, and for structuring information systems to effect such control.

The characteristics of the systems to be controlled imply characteristics of superimposed control systems. Infrastructure systems and the information systems that operate them are huge and distributed; so control must be distributed. Detection, analysis of and response to system-wide phenomena, such as coordinated security attacks, requires some centralization, so control systems with hierarchical structure are needed. Controlled systems change (e.g., as hardware fails and as they evolve in form and function over time), so control systems must be adaptive. Control systems could be used to hijack controlled systems, so they have to be especially secure. Finally, we need to control discrete-state information systems; so we need discrete state control systems [4].



**Figure 1.** Superimposed hierarchical control system

Figure 1 illustrates the hierarchical control concept. White circles represent *native* components of an infrastructure information system: here elements of the United States payment system. The *bb* nodes represent branch banks; *mcb*, money-center banks; and *frb*, the Federal Reserve Bank. The computing elements of such institutions provide services locally and interact with each other. Elements of a hypothetical control system are depicted in shades of gray. Higher level control nodes are in darker shades. The scope of control of each control node is indicated by nesting. Each bank has a local control node to perform local survivability monitoring and control: e.g., to report potential intrusions. In addition, each money-center bank has a control node whose scope is the money-center bank and subordinate branch banks. It communicates with subordinate control nodes, accepting reports from them and giving aggregate system-level information to them. Finally, a Federal Reserve control node has visibility to money-center bank nodes. Each control node could provide an interactive interface to report status to human management, and to enable human-initiated control actions in addition to automated control [4].

### 3 WEB-ENABLED SOFTWARE ARCHITECTURES

The idea of embedding web server implementations systematically into multiple components of distributed systems [3] appears to be new with our work. The concept provides a scalable, secure mechanism that can leverage a broad range of existing Internet technologies to support access, monitoring and control of systems at the runtime module level. Moreover, unlike competing architectures, ours does not require the components of a system to conform to implausibly demanding design rules of a common architectural framework. Our approach requires only that each component provide a mechanism by which clients can ascertain the URL of a top-level page served by the embedded server, which can then provide links to all web-accessible components services

Of course, the idea of associating URLs with objects itself is not new. XML-RPC [6], SOAP [1], Java Beans [5] and Joist [2] are examples. Nor is the concept of embedding web servers in applications new. The Jigsaw web server [7] exposes a web interface by which one configures the server, for example. XML-RPC and SOAP are essentially remote procedure call or messaging facilities that use Internet communication protocols and XML to encode messages. They enable object-oriented component APIs to be exposed on the Internet. Java Beans associate URLs with components that run within and conform to design rules imposed by certain web servers. The Joist web server is embedded in the Python language runtime. It was designed to support web-based debugging of embedded systems. Joist associates URLs with runtime components, treating URLs as pathnames to components. Component meta-data provided to Joist for accessible types enables Joist to treat components according to their types. All components in the same runtime share the same web server implementation.

Despite their many attractions, these related approaches suffer from several difficulties that make them less than ideal for implementing superimposed survivability control systems for heterogeneous, distributed systems. SOAP lacks security and SOAP and XML-RPC assume that Internet protocols will be used as the main “architectural glue” of an application. Java Beans and other server extension mechanisms impose demanding requirements on components: to fit into standard web-managed-object regimes. Joist lacks security. It is language-specific and language-implementation-specific. It requires Joist type descriptions for all accessible component types to be registered with the runtime server. All component designers thus have to “know about” the server, and the server has to “know about” all accessible components. These approaches lack information-hiding and independence properties and are not clearly compatible with native architectural choices. They are thus not well suited to a context of complex, evolving systems built from heterogeneous and legacy elements.

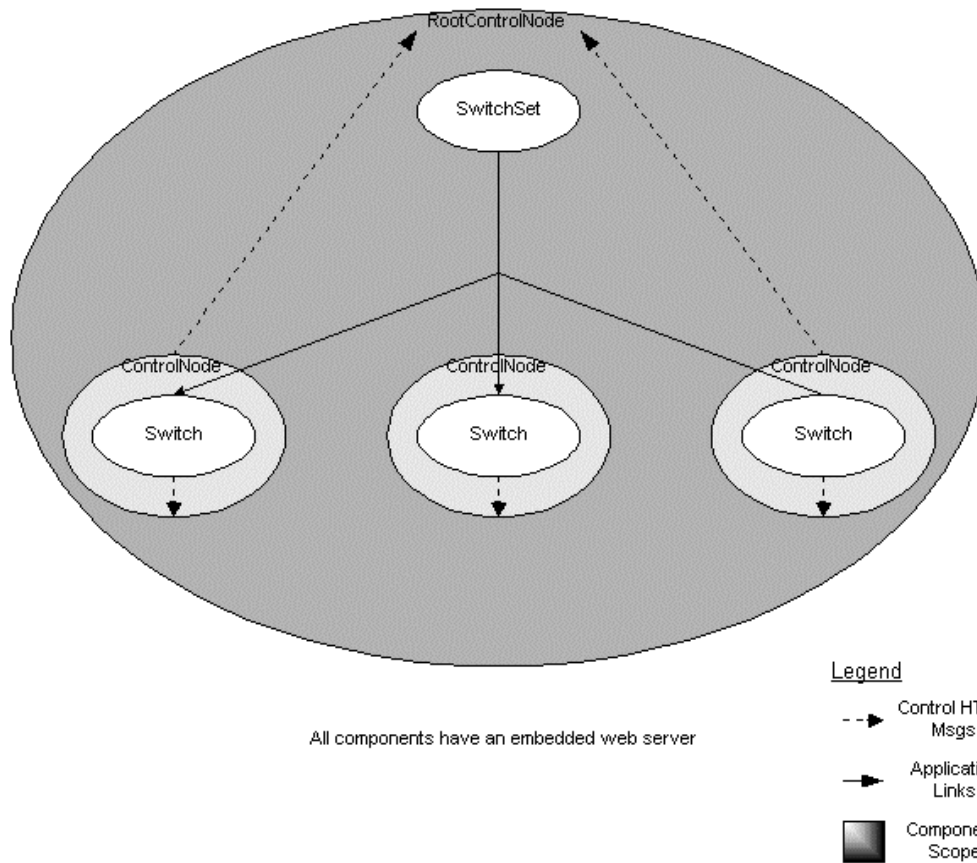
By contrast, our approach does not require that components conform to a complex, common architecture or that they use Internet protocols as their native communication mechanisms. The component-designer is free to implement the system in the best manner, merely ensuring that each component provides access to its server in the form of a single URL. Our approach does not require a single server to know about all components, nor for all component designers to be aware of or to use any common server infrastructure. Our approach also enables each component to specify and implement its own security policy and access control for the information it exposes.

We have implemented a simple proof-of-concept system to demonstrate and support exploration of this idea [3]. In essence, each system object has an instance variable of a *WebServer* type, includes programming code for that server, and exposes a *GetURL* method. We initialize these embedded servers dynamically, assigning URLs to their top-level pages based on host names and free TCP/IP port numbers. We have used this technology to demonstrate several novel and interesting functions. First, components that interact with each other through native mechanisms can support orthogonal web-based interfaces. Second, components can expose arbitrary dynamically constructed information about themselves through their web interfaces. For example, our objects, written in Java, use Java introspection to obtain definitions of their own interfaces, which they make accessible to clients as HTML. Third, web-supported public-key infrastructure can be used for authentication, and additional access control to web pages and services can be implemented with login protocols. Furthermore, HTTPS can be used for secure communications throughout. Fourth, the uploading of applets from components to clients and subsequent secure registration of the applets for notifications from originating components, using HTTPS, enables secure, interactive component monitoring communications. Fifth, browsing of the interconnected components of a running application can be supported through dynamic generation of HTML pages containing hyperlinks that mirror the object-level interconnection topology. Sixth, web crawlers can be used to index the components in a running system by traversing their exposed, linked web pages. Seventh, web-based interfaces can expose “open implementation” style interfaces providing runtime control of the choice of concrete component implementations.

#### **4 SYNTHESIS: MICROCOSMIC SURVIVABILITY ARCHITECTURES?**

The synthesis is in the idea of using our approach as the basis of an architecture for survivability control systems. The idea is appealing for several reasons. First, it supports the security needed to protect survivability control systems. Second, it accommodates heterogeneity in language, middleware, platform. Third, it is scalable to large, cross-administrative-domain systems. Fourth, it is based on standard, powerful mechanisms for distributed systems. Finally, orthogonality to native architecture means that deploying it in legacy systems might be feasible.

The approach makes a classic tradeoff of design-time redundancy and associated runtime performance penalties for the benefits of architectural orthogonality and implementation independence. This tradeoff is plausible for systems built from large, independently developed components. The cost per web-enabled component is several hundred bytes to several hundred kilobytes, depending on the services provided. There are two basic approaches to managing this cost: (1) embed web services only in selected components; (2) exploit optimization opportunities as they arise. The approach is wholly plausible if it is applied to small numbers of mega-components, e.g., to top-level software components corresponding to such entities as branch banks. It is wholly implausible applied universally to fine-grained components, unless optimization is possible, e.g., by having many components share a single server implementation (as in Java Beans or Joist). Engineering judgment will be required, as usual.



**Figure 2** Architecture of the test-bed Application

To assess the feasibility of the idea, we augmented our simple demonstration application—which substitutes two-state switch objects for banks and a set of switches for the Federal Reserve—with a simple control system. Figure 2 illustrates the system. We designed a *ControlNode* component type, and associated an instance with each application component. Thus, we created a two-level control hierarchy: a control node is associated with the *SwitchSet*. Its scope of control includes the control nodes associated with the individual *Switch* components. The control nodes communicate up and down this hierarchy and with their associated application nodes to exchange information and take actions. Both the application components and the *ControlNode* components embed web servers, which are used to carry out these exchanges. On creation, a switch is made aware of the URL of the web server of its associated control node. Similarly, the control node is made aware of its associated computation node by passing it the URL of the web server embedded in the computation node.

In this simple proof-of-concept system, control nodes merely keep track of the number of times each Switch changes state. On being switched on or off, a switch notifies its control node by sending its server an HTTP request with a header indicating the change. The control node increments a counter and, in turn, sends a notification to its parent control node with a header indicating the current count known to the child control node. Higher-level nodes are thus made aware of the number of times the switches in their scope have changed state. All of these messages are sent securely using SSL. Each component type has a certificate used to authenticate the sender. To demonstrate the flow of information down the control hierarchy, we allow parent control nodes to notify their children to stop sending notifications. Among other things, the *ControlNode* servers could support an interactive interface at each control system level for human participation in the survivability control loop. Our demonstration system is just an “architectural sketch,” but it shows that the approach enables us to combine a set of powerful Internet mechanisms to implement functions that would be critical to any real critical infrastructure survivability control system.

## REFERENCES

- [1] Box D. SOAP: Simple Object Access Protocol. *HTTP Working Group Internet Draft*, September 1999.
- [2] Gorlick MM, Distributed Debugging and Monitoring on \$5 a Day, *Proceedings of California Software Symposium*, Univ. of California, Irvine, California, 1997, pp. 31-39.
- [3] Sullivan KJ, Saxena A, Microcosms: A Web Server in Every Software Component, *Technical Report 2000-20*, University of Virginia Department of Computer Science, July 31, 2000.
- [4] Sullivan KJ, Knight KC, Du X, Geist S, Information Survivability Control Systems, *Proceedings of the 21<sup>st</sup> International Conference on Software Engineering*, May 1999, pp. 184—193.
- [5] Sun Microsystems: Java Technologies, JavaBeans® Specification. [Online Document]. (<http://java.sun.com/products/javabeans/docs/spec.html>).
- [6] Userland Software Inc. XML-RPC Specification. [Online document]. (<http://www.userland.com/spec>).
- [7] World Wide Web Consortium. Jigsaw Web server [Online document]. (<http://www.w3.org/Jigsaw/>)