Edge Mashups for Service-Oriented Collaboration

Ken Birman, Jared Cantwell, Daniel Freedman, Qi Huang, Petko Nikolov, and Krzysztof Ostrowski, *Cornell University*

The Live Distributed Objects platform makes it possible to combine hosted content with P2P protocols in a single object-oriented framework.

Networked collaboration tools may be the key to slashing healthcare costs, improving productivity, facilitating disaster response, and enabling a more nimble information-aware military. Better applications could even make possible a world of professional dialog and collaboration without travel.

We term such tools *service-oriented collaboration* (SOC) applications. SOC systems are more and more appealing because of the increasingly rich body of service-hosted content, such as electronic medical health records, data in various kinds of databases, image repositories, patient records, and weather prediction systems. They may also tap into sensors, medical devices, video cameras, microphones, and other real-world data sources.

Many kinds of applications are constructed as *mashups*, in which data from various sources is combined in a single multilayered interactive GUI, and it may seem natural to use mashups to build SOC applications as well. The collaborating team could pull the various data sources it is using into a single interactive application, which would be shared among the users.

But building SOC applications isn't going to be as simple as many believe. Media streams generate high, bursty update rates, and many require low latencies and tight synchronization between collaborating users. Some also require client-to-client security and can't "trust" a Web services platform or any other third party. For example, in many medical scenarios, only the collaborating physicians are permitted to see the communication that occurs; military applications may involve classified information.

These requirements represent serious issues because in today's Web services standards, client-to-client data must be relayed via a hosted service—typically, an enterprise service bus (ESB) using a publish-subscribe model, RSS feeds, a message-queuing (MQ) middleware product like the Java Messaging Service (JMS), and so on. Relaying introduces delay and scalability issues. Moreover, Web services security models focus on client-to-server security. If a server can't be trusted, Web services security offers no help.

Something new is needed: a way to create SOC applications that seamlessly integrate hosted content with the kinds of peer-to-peer (P2P) protocols capable of responding to these needs. Cornell's Live Distributed Objects platform solves this problem, enabling end users to construct mashups that live directly on client platforms and can be operated even without connectivity to the Internet. These edge mashups enable a powerful style of Web-services-supported collaboration (K. Ostrowski, "Programming with Live Distributed Objects," *Proc. 22nd European Conf. Object-Oriented Programming,* LNCS 5142, Springer-Verlag, 2008, pp. 463-489).

TODAY'S WEB SERVICES APPROACH

Let's look more closely at the way today's developers build mashups to support SOC applications. SOC systems focus on interactive scenarios, hence the client will often be running a browser or some other form of GUI. In such cases, it is becoming common for service platforms to export a *minibrowser* component. This is an interactive webpage with embedded script, commonly developed using JavaScript/Ajax, Flash, Silverlight, or a similar technology, and optimized

 Published by the IEEE Computer Society 0018-9162/09/$25.00 © 2009 IEEE

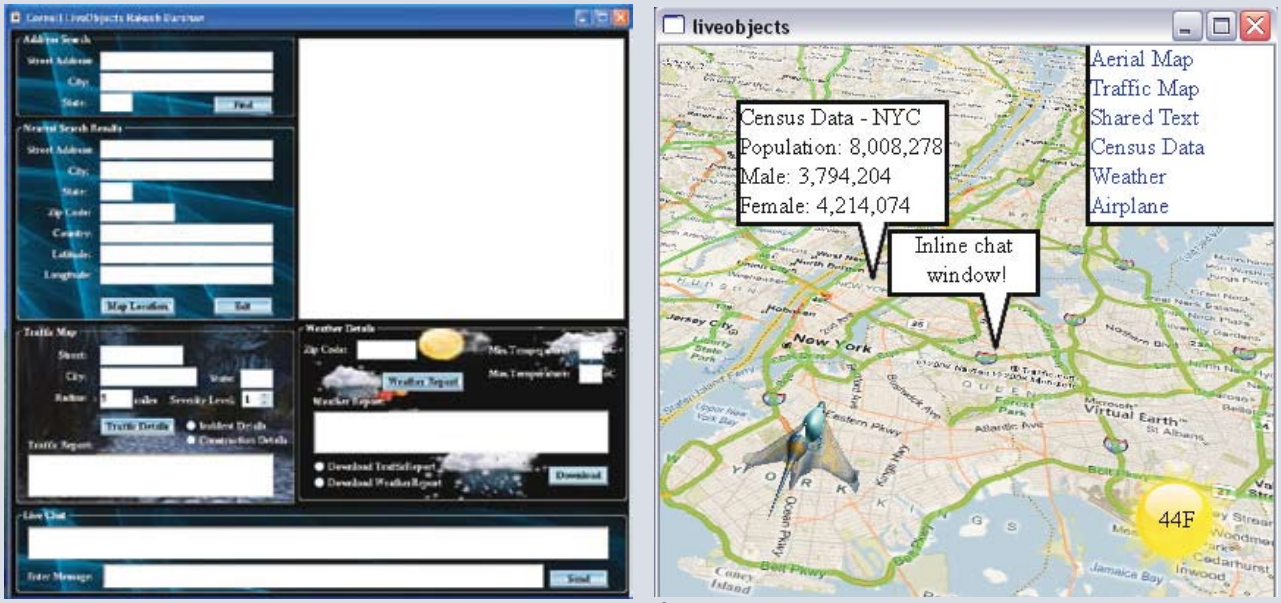

Figure 1. Example service-oriented computing application using (a) a minibrowser-style mashup and (b) Live Distributed Objects.

for some type of content, for example interactive maps from Google Earth or Virtual Earth.

The embedded script is often tightly integrated with back-end services in the data center—services that may not even be directly accessible at a programmatic level. As a result, the only way that new content can be "mashed" into the data available from the service is to have the data center itself compute the mashup.

For example, Google's minibrowsers expose composite images that draw on multiple data sources, presented to the client as selectable layers. If the client pans or zooms the minibrowser window, the data associated with the mashup is also zoomed or panned. Google also offers tools to help end users define new kinds of mashups. When these are used, however, the data is combined by Google's platform, not on the client system. This point will turn out to be important: A mashup built this way won't be functional unless a connection to Google is available, and won't be able to incorporate protocols that run directly between the client machines.

Developers could incorporate these kinds of content and protocols in a second way: by running multiple minibrowser windows in a single webpage. However, they won't talk to one another. Another possibility is to access the data centers at a programmatic level. This, though, is hard because many of the features accessible through minibrowsers are difficult to access, or not available at all, via programmatic APIs.

To illustrate this point, consider Figure 1a, which shows a SOC application constructed using a standard Web services approach, pulling content from the Yahoo maps and weather Web services and assembling it into a webpage as a set of tiled frames. Each frame contains a minibrowser with its own interactive controls and comes from a single content source. To highlight one of the many restrictions: If the user pans or zooms in the map frame, the associated map will shift or zoom, but the other frames remain as they were—the frames are not synchronized.

Implicit in the example is a second, and perhaps even more serious, issue. We noted that SOC applications will need a snappy response, even with substantial numbers of collaborating users. In today's Web services architecture, when one client wants to send an event to some set of other clients, the event needs to be relayed through an ESB, a catch-all term that covers everything from a JMS application to a publish-subscribe product to an RSS feed. The problem is performance and scalability. Bouncing data off a remote server can be slow. If the server is inaccessible, clients won't be able to collaborate even if they have a direct connection to one another.

Additionally, today's ESB solutions scale poorly as users add clients. This is evident in Figure 2. The ESB products evaluated here can operate in durable (logged) mode and nondurable mode, but as we see here, not a single product sustains high throughput as the number of clients scales up.

Now consider Figure 1b. Here we revisit our mashup application using the Live Objects platform. Content from different sources is overlaid in the same window and synchronized so that each reports data for the same locations. We designed the application to highlight the contribu-

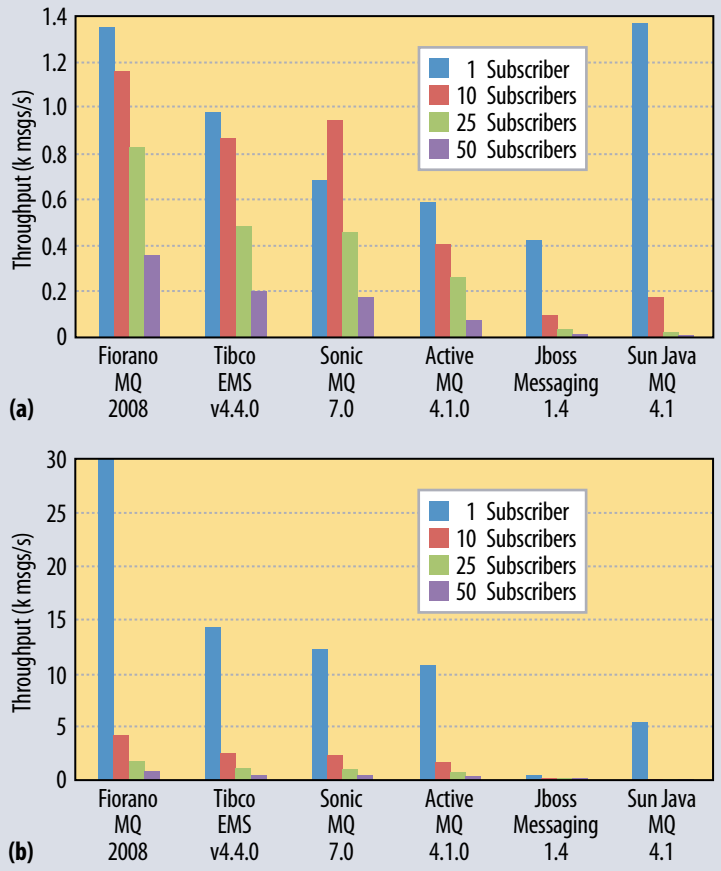

Figure 2. Throughput scalability of enterprise server bus solutions, as a function of the number of subscribers, in (a) durable mode and (b) nondurable mode. Data adapted from Fiorano Software Technologies, "JMS Performance Comparison: Performance for Publish Subscribe Messaging," white paper, Feb. 2008 (www.capitalware.biz/dl/docs/fiorano_jms_performance_comparison.pdf).

tions of different sources, but there are no frame boundaries: Elements of this mashup—which can include maps, 3D terrain features, images of buildings or points of interest, icons representing severe weather reports, vehicles or individuals, and so on—coexist as layers within which the end user can easily navigate.

Data can come from many kinds of data centers. Our example overlays weather from Google, terrain maps from Microsoft's Virtual Earth, census data from the US Census Bureau, and flight information from the US Federal Aviation Administration. Moreover, the mashup is constructed directly on the platform of the users who share the SOC application: These are *edge mashups*, as distinct from Google-style of *hosted mashups*.

Importantly, the Live Objects platform treats every kind of content as an object. Thus, the example seen in Figure 1b isn't limited to hosted content: It includes components that use direct P2P communication protocols. Our platform can support any sort of protocol, including client-server, but also overlay multicast, P2P replication, or even custom protocols designed by the content provider. This makes it possible to achieve extremely high levels of throughput and latency. It also enhances security: The data-center server can't "see" data exchanged directly between peers, and applications can exploit provably secure protocols that create and share cryptographic keys so that only the end-point hosts can access them.

SOC REQUIREMENTS

To be successful, SOC platforms will need to satisfy a number of what might be termed "client-oriented" requirements:

- SOC systems should enable a non-programmer to rapidly develop a new collaborative application by composing and customizing pre-existing components.
- They should make it possible to overlay data from multiple sources, potentially in different formats, obtained using different protocols and inconsistent interfaces.
- It should be possible to dynamically customize the application at runtime, for example by incorporating new data sources or changing the way data is presented, during a mission, and without disrupting system operation.
- It should be possible to accommodate new types of data sources, new formats, or protocols that we may not have anticipated at the time the system was released.
- Individual users might publish data, and it might be necessary for the users to exchange their data without access to a centralized repository.
- Data may be obtained using different types of network protocols, and the type of physical network or protocols may not be known in advance; it should be possible to rapidly compose the application using whatever communication infrastructure is currently available.
- Users may be mobile or temporarily disconnected, infrastructure may fail, and the network's topology and characteristics might change over time. The system should be easily reconfigurable.

Today's Web services standards are overly focused on the data-center side of the story. Not only are performance, scalability, and security all serious concerns, but the trend toward prebuilt minibrowsers with sophisticated but black-box behavior is making it increasingly more difficult to combine information from multiple sources. SOC applications aren't at odds with Web services, but they do need something new.

USING LIVE OBJECTS FOR SOC

As Figure 1b shows, the Live Objects platform solves these problems. Even a nonprogrammer can build a new SOC application, share it (perhaps via e-mail), and begin to collaborate instantly. Moreover, performance, scalability, and security can all be addressed. The main steps are as follows:

The developer starts by creating or gaining access to a collection of components. Each component is an object that supports live functionality and exposes event-based interfaces by which it interacts with other components. Examples include

- components representing hosted content,
- sensors and actuators,
- renderers that graphically depict events,
- replication protocols,
- synchronization protocols,
- folders containing sets of objects, and
- display interfaces that visualize folders.

Individual components and mashups of components have two representations. When inactive, a component or a mashup is represented as an XML page, describing a "recipe" for obtaining and parameterizing components that will serve as layers of the composed mashup. We call such an XML page a *live object reference*. References can be distributed as files, over HTTP, or through other means.

An end user creates a new SOC application by selecting components and combining them into a new mashup, using drag-and-drop. Our tools automatically combine references for individual objects into an XML mashup of references describing a graph of objects and type-check the graph to verify that the components compose correctly. For example, a 3D visualization of an airplane may need to be connected to a source of GPS and other orientation data, which in turn can only be used over a data replication protocol with specific reliability, ordering, or security properties.

The Live Objects model can support a variety of collaboration and coordination paradigms, including many that the traditional Web services style of client can't offer.

When activated on a user's machine, an XML mashup yields a graph of interconnected proxies. If needed, an object proxy can initialize itself by copying the state from some active proxy (our platform assists with this sort of state transfer). The object proxies then become active ("live"), for example, by relaying events from sensors into a replication channel or receiving events and reacting to them (such as redisplaying an aircraft).

The Live Objects approach shares certain similarities with the existing Web development model, in the sense that it uses hierarchical XML documents to define the content. On the other hand, it departs from some of the de facto stylistic standards that have emerged. For example, earlier we noted that if a developer pulls a minibrowser from Google Earth, that minibrowser will expect to interact directly with the end user, and includes embedded JavaScript that handles such interactions. When working with an application constructed using the Live Objects platform, the same functionality would be represented as a mashup of a component that fetches maps and similar content with a second component that provides the visualization interface. The lower-level component uses event-oriented interfaces. The advantage is that it can also talk to other components, not just the GUI.

Moreover, the Live Objects platform can easily support applications that would remain operational even when connectivity to the Internet is disrupted. For example, a SOC application might include P2P protocols with which rescue workers searching a disaster site could coordinate their actions. Such protocols are remarkably robust and can make progress even if they have intermittent connectivity and no access to Internet data centers. In contrast, a solution constructed using a minibrowser would only be useable so long as a connection back to the host site that provided the minibrowser is available.

Live Objects applications can dynamically recalculate the set of "visible" objects, as a function of location and orientation. Thus an emergency responder could be shown the avatars of others who are already working at that site and participate in conference-style or point-to-point dialog with them.

It should be evident that the model can support a variety of collaboration and coordination paradigms, including many that the traditional Web services style of client can't offer. The ability to combine hosted content with P2P content in a single shared object-oriented paradigm

overcomes all the limitations we cited earlier. High-speed, low-latency P2P protocols can carry event and media streams from client system to client system, seamlessly blending with hosted content drawn from Web-service-based data centers.

NEED FOR NEW STANDARDS

Live objects leverage Web services, but the examples we've given make it clear that the existing Web services standards don't go far enough. The main issue arises when components coexist in a single application. Just as services within a data center need to agree on their common "language" for interaction, and do so using Web services standards, components living within a SOC application running on a client platform will need to agree on the events and representation that the "dialog" between them will employ.

The decoupling of functionality into layers also suggests a need for a standardized layering: The examples above identify at least four: the visualization layer, the linkage layer that talks to the underlying data source, the update generating and interpreting layer, and the transport protocol. We propose using event-based interfaces to perform this decoupling—a natural way of thinking about components that dates back to Smalltalk and is common in modern platforms too, notably Jini.

Also needed are standard ways to express the properties of protocols. Lacking standards, each protocol is just a black box, and the platform can't determine when one can safely be substituted for another. Given standard ways of talking about the guarantees and requirements of a protocol, in a setting with good connectivity, it might be possible to use fast Internet multicast protocols; the same application, running in a setting with poor connectivity, could switch to a slower but more robust option, such as a gossip-based protocol. A solution that exchanges sensitive data could be constrained to only communicate using a secure protocol with an approved end-to-end cryptographic key-management mechanism.

Work on the Live Distributed Objects platform reveals that it can be unexpectedly hard to build high-performance SOC systems using today's Web services standards. These problems are particularly acute when combining data from multiple sources into a new client-side mashup. The core limitations stem from a mixture of issues: scalability and performance problems with ESB components, but also de facto ways of presenting mashups to end users (through proprietary minibrowsers). Live objects solve these problems, making it possible to combine hosted content with P2P protocols in a single object-oriented framework. This opens the door to a wide range of exciting SOC opportunities in settings that range from healthcare to finance to disaster response.

The Live Objects platform can be downloaded, for free, at http://liveobjects.cs.cornell.edu.

Ken Birman *is the N. Rama Rao Professor of Computer Science at Cornell University and heads the Live Distributed Objects project. Contact him at ken@cs.cornell.edu.*

Jared Cantwell *is an MEng degree candidate in the Department of Computer Science at Cornell University. Contact him at jared.cantwell@gmail.com.*

Daniel Freedman *is a postdoctoral researcher in the Department of Computer Science at Cornell University. Contact him at dfreedman@cs.cornell.edu.*

Qi Huang *is a visiting scientist at Cornell University from the School of Computer Science and Technology at Huazhong University of Science and Technology, China. Contact him atqhuang@cs.cornell.edu.*

Petko Nikolov *is an MEng degree candidate in the Department of Computer Science at Cornell University. Contact him at pn42@cornell.edu.*

Krzysztof Ostrowski *is a postdoctoral researcher in the Department of Computer Science at Cornell University. Contact him at krzys@cs.cornell.edu.*

The authors thank Danny Dolev for his many comments and suggestions. This work was supported, in part, by the NSF, the AFRL, the AFOSR, Intel, and Cisco. Qi Huang is supported by the Chinese NSFC, grant 60731160630.

Editor: Simon S.Y. Shim, Dept. of Computer Engineering, San Jose State Univ., San Jose, CA; simon.shim@sjsu.edu