Building Collaboration Applications That Mix Web Services Hosted Content with P2P Protocols[1]

Ken Birman, Jared Cantwell, Daniel Freedman, Qi Huang, Petko Nikolov, Krzysztof Ostrowski
Dept of Computer Science
Cornell University
Ithaca, NY, USA
{ken, dfreedman, qhuang, krzys}@cs.cornell.edu, {jmc279, pn42}@cornell.edu

The most commonly deployed web service applications employ client-server communication patterns, with clients running remotely and services hosted in data centers. In this paper, we make the case for Service-Oriented Collaboration applications that combine service-hosted data with collaboration features implemented using peer-to-peer protocols. Collaboration features are awkward to support solely based on the existing web services technologies. Indirection through the data center introduces high latencies and limits scalability, and precludes collaboration between clients connected to one-another but lacking connectivity to the data center. Cornell's Live Distributed Objects platform combines web services with direct peer-to-peer communication to eliminate these issues.

I. INTRODUCTION

There is a growing opportunity to use *Service-Oriented Collaboration Applications* in ways that can slash health-care costs, improve productivity, permit more effective search and rescue after a disaster, enable a more nimble information-enabled military, or make possible a world of professional dialog and collaboration without travel. Collaboration applications will need to combine two types of content: traditional web service *hosted content*, such as data from databases, image repositories, patient records, and weather prediction systems, with a variety of *collaboration features*, such as chat windows, white boards, peer-to-peer video and other media streams, and replication/coordination mechanisms.

Existing web service technologies make it easy to build applications in which *all data travels through a data center*. Implementing collaboration features using these technologies is problematic because collaborative applications can generate high, bursty update rates and yet often require low latencies and tight synchronization between collaborating users. One can often achieve better performance using direct client-to-client (also called peer-to-peer, or P2P) communication, but in today's SOA platforms, "side-band" communication is hard to integrate with hosted content. This problem is reflected by a growing number of publications on the integration of web services with peer-to-peer platforms, e.g., [2], [4], [8], [9], [10], [14], [15], [16], [20], [21]. Yet the issue remains unresolved (see Section VII for more details).

Cornell's Live Distributed Objects platform [12] (Live Objects for short) allows even a non-programmer to construct content-rich solutions that blend traditional web services and peer-to-peer technologies, and to share them with others. This is like creating a slide show: drag-and-drop, after which the solution can be shared in a file or via email and opened on other machines. The users are immersed in the resulting collaborative application: they can interact with the application and peers see the results instantly. Updates are applied to all replicas in a consistent manner. Moreover, in contrast to today's web service platforms, P2P communication can coexist with more standard solutions that reach back to the hosted content and trigger updates at the associated data centers. Thus, when an application needs high data rates, low latency, or special security, it can use protocols that bypass the data center to achieve the full performance of the network.

This paper makes the following contributions.

- We describe a new class of Service-Oriented Collaboration applications that integrate service hosted content with peer-to-peer message streams. We analyze two example collaboration applications (search and rescue mission and virtual worlds), identifying shared characteristics. We list the key challenges that these kinds of applications place on their runtime environments.
- We describe a new class of multi-layered mashups and contrast them with more traditional, minibrowser-based approach to building mashups, characteristic of today's web development.
- We discuss the advantages of decoupling transport and information layers as a means of achieving reusability, customizability, ability to rapidly deploy collaboration applications in new environments and adapt them dynamically to the changing needs. We discuss the resulting object-oriented perspective, in which instances of distributed communication protocols are modeled uniformly as objects similar to those in Java, .NET, COM or Smalltalk.
- We present our Live Distributed Objects platform: an example of a technology that fits well with the

[1]This work was supported, in part, by the NSF, AFRL, Intel and Cisco. Qi Huang is a visiting scientist from the School of Computer Sci & Tech; Huazhong University of Sci & Tech, supported by the Chinese NSFC, grant 60731160630.

layered, componentized model derived through our analysis.

- We compare performance of hosted Enterprise Service Bus (ESB) solutions with peer-to-peer communication protocols as an underlying communication substrate for service oriented collaboration. Rather than finding a clear winner, we identify relative strengths of each of the solutions tested. We see this as a further justification for the decoupling of information and transport layers advocated above: instead of a one-size fits all approach, an application can pick and choose among a menu of interchangeable components specialized for different environments.

II. LIMITATIONS OF THE EXISTING MODEL

There are two important reasons why integrating peer-to-peer collaboration with server-hosted content is difficult. The first is not strictly limited to collaboration and peer-to-peer protocols; rather, it is a general weakness of the current web mashup technologies that makes it hard to seamlessly integrate data from several different sources.The web developers' community has slowly converged towards service platforms that export autonomous interactive components to their clients, in the form of what we'll call *minibrowser* interfaces. A minibrowser is an interactive web page with embedded script, developed using AJAX, Silverlight, Caja, or similar technology, optimized for displaying a single type of content, for example interactive maps from Google Earth or Virtual Earth.

The embedded script is often tightly integrated with backend services in the data center, making it awkward to access the underlying services directly from a different script or a standalone client. As a result, the only way such services can be mashed up with other web content is by either having the data center compute the mashup (so that it can be accessed via the minibrowser), or by embedding the entire minibrowser window in a web page. But an embedded minibrowser can't seamlessly blend with the surrounding content; it is like a standalone browser within its own frame, and runs independent of the rest of the page.

To illustrate this point, consider Fig. 1 and Fig. 2. The figures are screenshots of web applications, with content from multiple sources mashed-up together. Fig. 1 was constructed using a standard web services approach, pulling content from the Yahoo! maps and weather web services and assembling it into a web page as a set of tiled frames. Each frame is a minibrowser with its own interactive controls, and comes from a single content source. To illustrate 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.

Now consider Fig. 2. Here we see a similar application constructed using Live Objects. In this case, content from different sources is overlaid in the same window and synchronized. We used white backgrounds to highlight the contributions of different sources, but there are no frame boundaries: elements of this mashup (which can include map layers, tables showing buildings or points of interest, icons representing severe weather reports, vehicles or individuals, etc.) co-exist layers within which the end user can easily navigate. Data can come from many kinds of data centers. Our example actually overlays weather from Google on terrain maps from Microsoft's Virtual Earth platform and extracts census data from the US Census Bureau: the lion coexists with the lamb.

The second problem is that with the traditional style of web development, content is assumed to be fetched from a server, either directly over HTTP, or by interacting with a web service. Web pages downloaded by clients' browsers

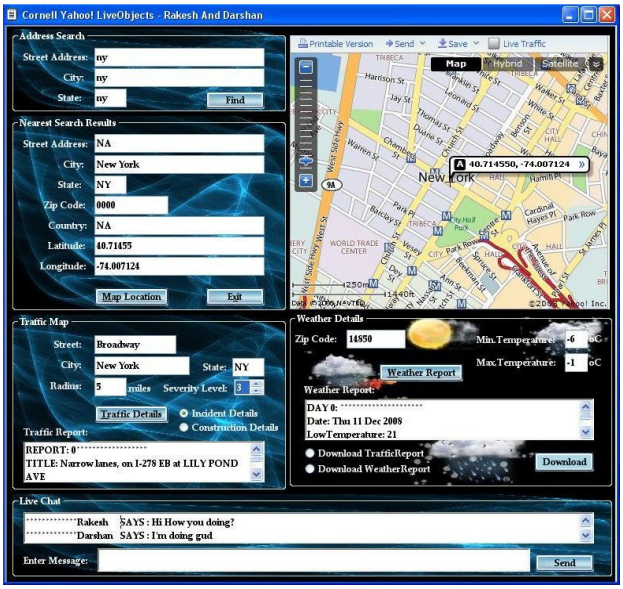

Figure 1. Standard Minibrowser-Style Mashup

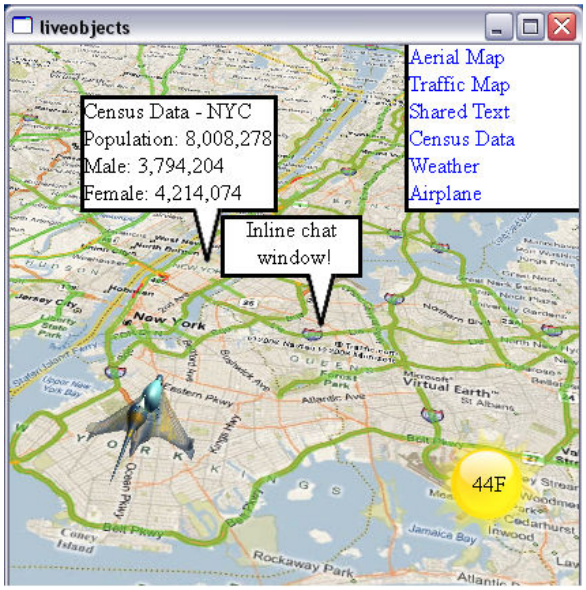

Figure 2. Live Objects Multi-Layered Mashup

contain embedded addresses of specific servers. Client-to-client traffic routes through a data center.

In contrast, Live Objects allow visual content and update events to be communicated using any sort of protocol, including client-server, but also overlay multicast, peer-to-peer replication, even a custom protocol designed by the content provider. As noted earlier, 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.

The above discussion motivates our problem statement:

- Allow web applications to overlay content from multiple sources in a layered fashion, such that the distinct content layers share a single view and remain well synchronized: zooming, rotating, or panning should cause all layers to respond simultaneously, and an update in any of the layers should be reflected in all other layers.
- Allow updates to be carried by the protocol best matched to the setting in which the application is used.

As noted earlier, the solutions discussed here are based on Live Objects. These support drag-and-drop application development. Of course, new types of components must be created for each type of content, but the existing collection of components provides access to several different types of web services hosted content (including all the examples given above). Once constructed, the resulting live application is stored as an XML file. The file can be moved about and even embedded in email. Users that open it find themselves immersed into the application.

Examples of transport protocols optimized for various settings include support for WAN networks with NATs and firewalls (SOLO [6]), low latency (Ricochet [1]), high throughput and very large numbers of nodes (QSM [11]), large numbers of irregularly overlapping multicast groups (Gossip Objects [3]), and strong reliability properties (Properties Framework [13]).

III. SERVICE ORIENTED COLLABORATION

Before saying more about our approach, we analyze an example collaboration application to expose the full range of needs and issues that arise.

Consider a rescue mission coordinator: a police or fire chief coordinating teams who will enter a disaster zone in the wake of a catastrophe to help survivors, control dangerous situations (electrical wires down, chemical leaks, fires, etc.), and move supplies as needed. The coordinator, a non-programmer, would arrive on the scene, build a new collaboration tool, and distribute it to his/her team. Each team member would carry a tablet-style device with wireless communication capabilities. The application built by the coordinator would be installed on each team member's mobile device, and in the offices in mission headquarters.

The coordinator would then deploy teams in the field. Our rescue workers now use the solution to coordinate and prioritize actions, inform each other of the evolving situation, steer clear of hazards, etc. As new events occur, the situational status would evolve, and the team member who causes or observes these status changes would need to report them to the others. For example, removing debris blocking access to a building may enable the team to check it for victims, and fire that breaks out in a chemical storage warehouse may force diversion of resources. As rescue workers capture information, their mobile devices send updates that must be propagated in real-time.

Having defined the scenario, now let's analyze in more detail the requirements it places on our collaboration tool. First, note that, the collaboration tool pulls data from many kinds of sources. It makes far more sense to imagine that weather information, maps, traffic, sensors data, positions of units, buildings, messages and alerts come from a dozen providers than to assume that one organization would be hosting services with everything we need in one place. Data from distinct sources could have different format and one will often need to interface to each using its own protocols and interfaces.

Second, as conditions evolve the team might need to be modify the application, for example adding new types of information, changing the way it is represented, or even modifying the way team members communicate (for example, if reach-back network links fail). Whereas a minibrowser would typically be prebuilt with all the available features in place, our scenario demands a much more flexible kind of tool that can be *redesigned* while in use.

Third, depending on the location and other factors, the best networking protocols and connectivity options may vary. In our rescue scenario, the workers may have to use wireless P2P protocols much of the time, reaching back to hosted services only intermittently when a drone aircraft passes within radio range. More broadly, the right choice of protocol should reflect the operating conditions, and if these change, the platform should be capable of swapping in a different protocol without disrupting the end user. This argues for a decoupling of functionality. Whereas a minibrowser packages it all into one object, better is a design in which the presentation object is distinct from objects representing information sources and objects representing transport protocols. Decoupling makes it possible to dynamically modify or even replace a component with some other (compatible) option when changing conditions require it.

We have posed what may sound like a very specialized problem, but in fact we see this as a good example of a more general kind of need that could arise in many kinds of settings. For example, consider a physician treating a patient with a complex condition, who needs collaboration help from specialists, and who might even be working in a remote location under conditions demanding urgent action. The mixture of patient data, telemetry, image studies, etc., may be just as rich and dynamic as in our search and rescue scenario, and the underlying communication options equally heterogeneous and unpredictable. A minibrowser pre-designed for a wired environment might

perform poorly or fail under such conditions. With Live Objects, if there is a way to solve the problem, there is a way to build the desired mashup.

Throughout the above we noted requirements; for clarity, we now summarize them below. As noted, these needs are seen in many settings. Indeed, we believe them to be typical of most collaboration applications.

- We would like to enable a non-programmer to rapidly develop a new collaborative application by composing together and customizing preexisting components.
- We would like to be able to overlay data from multiple sources, potentially in different formats, obtained using different protocols and inconsistent interfaces.
- We would like to be able to dynamically customize the application at runtime, e.g., by incorporating new data sources or changing the way data is presented, during a mission, and without disrupting system operation.
- We would like to be able to accommodate new types of data sources, new formats or protocols that we may not have anticipated at the time the system was released.
- Data might be published by the individual users, 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 the 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 topology of the network and its characteristics might change over time. The system should be easily reconfigurable.

The requirements outlined above might seem hard to satisfy, but in fact, the solution is surprisingly simple. Our analysis motivates a component-oriented architecture, in which the web services and hosted content are modeled as reusable overlayed information layers backed by customizable transport layers: a graph of components. A collaborative application is a forest: a set of such graphs.

Our vision demands a new kind of collaboration standard, in order to facilitate the side-by-side coexistence of components that might today be implemented as proprietary minibrowsers: if we enable components to talk to one-another, we need to agree on the events and representation that the dialog will employ. The decoupling of functionality into layers also suggests a need for a standardized layering: in the examples above, one can 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 that this decoupling be done using event-based interfaces; a natural way of thinking about components that dates back to Smalltalk.

Thus, rather than having the data center developer offer content through proprietary minibrowser interface, he/she would define an event-based interface between transport and information layers; the visual events delivered by the transport could then be delivered to an information layer responsible for visualizing them. It, in turn, would capture end-user mouse and keyboard events and pass them down, also as events. With this type of event-based decoupling, either layer could easily be replaced with a different one.

In this perspective, distributed peer-to-peer protocols would also be encapsulated within their respective transport layers. Thus, for example, one version of a transport layer could fetch data directly from a server in a data center, whereas a different version might use a peer-to-peer dissemination architecture, a reliable multicast protocol; it could leverage different type of hardware or be optimized for different types of workloads. Provided that the different versions of the transport layer conform to the same standardized event-based interfaces, the application could then switch between them as conditions demand.

In this event-oriented world, end-users interact through Live Objects that transform actions into updates that are communicated in the form of events that are shared via the transport layer. The protocol implemented by the transport layer might replicate the event, deliver it to the tablets of our rescue workers, and report it through the event-based interface back to the information layer at which the event has originated. Thus, the transport layer with the embedded distributed protocol would behave very much like an object in Smalltalk: it would consume events and respond with events. This motivates thinking about communication protocols as objects, and indeed in treating them as objects much as we treat any other kind of object in a language like Java or in a runtime environment like Jini or .NET. Doing so unifies apparently distinct approaches. Just as a remotely hosted form of content such as a map or an image of a raincloud can be modeled as an object, so can network protocols be treated as objects.

Some P2P systems try to make everything a P2P interaction. But in the examples we've seen, several kinds of content would more naturally be hosted: maps and 3-D images of terrain and buildings, weather information, patient health records, etc. On the other hand, collaboration applications are likely to embody quite a range of P2P event streams: each separate video object, GPS source, sensor, etc, may have its own associated update stream. If one thinks of these as topics in publish-subscribe eventing systems, an application could have many such topics, and the application instance running on a given user's machine could simultaneously display data from several topics. We have previously said that we'd like to think of protocols as objects. It now becomes clear that further precision is needed: the objects aren't merely protocols, but in fact are individual protocol instances. Our system will need to simultaneously support potentially large numbers of transport objects running concurrently in

the end-user's system, in support of a variety of applications and uses.

All of this leads to new challenges. The obvious one was mentioned earlier: today's web services don't support P2P communication. Contemporary web services solutions presume a client-server style of interaction, with data relayed through a message-oriented middleware broker. Even if clients are connected to one-another, if they lose connectivity to the broker, they can't collaborate.

Another serious issue arises if the clients don't trust the data center: sensitive data will need to be encrypted. The problem here is that web services security standards tend to trust the web services platform itself. The standards offer no help at all if we need to provide end-to-end encryption mechanisms while also preventing the hosted services from seeing the keys.

Finally, we encounter debilitating latency and throughput issues: hosted services will be performance-limiting bottlenecks when used in settings with large numbers of clients, as we will see in our experimental section.

We are left with a mixture of good and bad news:

- 👍 Web services standardize client access to hosted services and data: we can easily build some form of multi-framed web page that could host each kind of information in its own minibrowser.
- 👍 When connectivity is adequate, relaying data via a hosted service has many of the benefits of a publish-subscribe architecture, such as robustness as the set of clients changes.
- 👎 The natural way to think of our application is as an object-oriented mashup, but web services provide no support for this kind of client application development.
- 👎 Our solution may perform very poorly, or fail if the hosted services are inaccessible.
- 👎 All data will probably be visible to the hosted services unless the developer uses some sort of non-standard end-to-end cryptography.

IV. USING LIVE OBJECTS FOR COLLABORATION

Cornell's Live Objects platform supports componentized, layered mashup creation and sharing, and overcomes limitations of existing web technologies. We've used it to construct a number of service oriented collaboration applications, some of which are quite sophisticated, including a solution to the search and rescue problem stated in Section 3. The major design aspects 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
  - Display interfaces that visualize folders.
- Mashups of components are represented as a kind of XML web pages; each 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 email, HTTP or other means.
- The application is created by building a forest consisting of graphs of references that are mashed together. At design time, an automated tool lets the developer drag and drop to combine references for individual objects into an XML mashup of references describing a graph of objects.
- The platform type-checks mashups to verify that they compose correctly. For example, a 3-D visualization of an airplane may need to be connected to a source of GPS and other orientation data, which in turn needs to run over a data replication protocol with specific reliability, ordering or security properties.
- When activated on a user's machine, an XML mashup yields a graph of interconnected *proxies.* A proxy is a piece of running code that may render, decode, or transform visual content, encapsulate a protocol stack, and so on. Each sub-component in the XML mashup produces an associated proxy. The hierarchy of proxies reflects the hierarchical structure of the XML mashup.
- 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 by receiving events and reacting to them (e.g. by redisplaying an aircraft).

Our 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, we depart from some of the de-facto stylistic standards that have emerged. For example if one pulls a minibrowser from Google Earth, it expects to interact directly with the end user, and includes embedded JavaScript that handles such interactions. In Live Objects, 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.

Although the term *mashup* may sound static, in the sense of having its components predetermined, this is not necessarily the case. One kind of live object could be a folder including a set of objects, for example extracted from a directory in a file system or pulled from a database in response to a query. When the folder contents change,

the mashup is dynamically updated, as might occur when a rescue worker enters a building or turns a corner.

Thus, Live Objects can easily support applications that dynamically recompute the set of "visible" objects, as a function of location and orientation, and dynamically add or remove them from the mashup. A rescuer would automatically and instantly be shown the avatars of others who are already working at that site, and be able to participate in conference-style or point-to-point dialog with them, through chat objects that run over multicast protocol objects. This model can support a wide variety of collaboration and coordination paradigms.

In summary, the Live Objects platform makes it easy for a non-programmer to create the needed application. The rescue coordinator pulls prebuilt object references from a folder, each corresponding to a desired kind of information. Hosted data, such as weather, terrain maps, etc, would correspond to objects that "point" to a web service over the network. Peer-to-peer objects would implement chat windows, shared white boards, etc. Event interfaces allow such objects to coexist in a shared display window that can pan, zoom, jump to new locations, etc.

The relative advantages and disadvantages of our model can be summarized as follows:

- Like other modern web development tools, our platform supports drag-and-drop style of development, permitting fast, easy creation of content-rich mashups.
- The resulting solutions are easy to share.
- By selecting appropriate transport layers, functionality such as coordination between searchers can remain active even if connectivity to the data center is disrupted.
- Streams of video or sensor data can travel directly and won't be delayed by the need to "ricochet" off a remote and potentially inaccessible server.
- New event-based interoperability standards are needed. Lacking them, we could lose access to some of the sophisticated proprietary interactive functionality optimized for proprietary minibrowser-based solutions with an embedded JavaScript.
- Direct peer-to-peer communication can be much harder to use than relaying data through a hosted service that uses an Enterprise Service Bus (ESB) model. Furthermore, the lack of a "one size fits all" publish-subscribe substrate forces the developers to become familiar with and choose between a range of different and incompatible options. An wrong choice of transport could result in degraded QoS, inferior scalability, or even data loss.

V. SECOND LIFE™ SCENARIO

Up to now, we have focused on a small-scale example. But our longer term goal is to support a large-scale next-generation collaboration system similar to Second Life™, a virtual reality immersion system created by Linden Labs. Today's "hosted" Second Life system runs on a data center consisting of a large number of servers storing the state of the virtual world, the locations of all users, etc. Users (represented by avatars) customize the environment, then move about and interact with others. For example, one can create a cybercafé, customize its music, furniture, wall treatments, etc. As other Second Life users enter the room, they can interact with the environment and one-another.

In the Second Life architecture, whenever an avatar moves or performs some action in the virtual world, a request describing this event is passed to the hosting data center and processed by servers running there. When the number of users in a scenario isn't huge, Second Life can keep up using a standard workload partitioning scheme in which different servers handle different portions of the virtual world. However, when loads increase, for example because large numbers of users want to enter the same virtual discotheque, the servers can become overwhelmed and are forced to reject some of the users or reduce their frame-rendering rates and resolution. Under such conditions, Second Life might seem jumpy and unrealistic.

In our lab at Cornell we're using Live Objects to build our own version of Second Life, in which some content will still be hosted, but many kinds of client-to-client communication will flow directly. This form of Live Objects applications poses new but not insurmountable challenges. On the one hand, many aspects of the application can be addressed in the same manner we've outlined for the search and rescue application. One could use Microsoft Virtual Earth, or Google Earth, as a source of 3D textures representing landscapes, buildings, etc. The built-in standards for creating mashups could be used to identify sensors and other data sources, which could then be wrapped as Live Objects and incorporated into live scenes. On top of this, streaming media sources such as video cameras mounted at street level in places such as Tokyo's Ginza can be added to create realistic experience.

The more complex issue is that a search and rescue application can be imagined as a situational state fully replicated across all of its users. In this model, all machines would see all the state updates (even if the user is zoomed into some particular spot within the overall scene). One can contemplate such an approach because the aggregate amount of information might not be that large. In contrast, Second Life conceptually is a whole universe, unbounded in size and hence with different users in very distinct parts of the space. It would make no sense for every user to see every event.

To solve this problem within our Live Objects platform, we built a simple database that can be queried at low cost. Each user sees only the objects within some range, or within line of sight. As a user moves about, the platform recomputes the query result, and then updates the display accordingly. Of course this basic mechanism isn't the whole story, but given the brevity constraints on the current paper, it isn't possible to provide all the details. Instead, let's ask how a solution such as the one we've

mocked up at Cornell contrasts with the more standard version of Second Life: a hosted platform that exports a minibrowser.. Consider a 3D texture representing terrain in some region:

1. In a minibrowser approach, the minibrowser generates the texture from hosted data (say, a map) and displays it. This model makes it difficult (not impossible) to superimpose other content over the texture; generally, we would need to rely on a hosting system's mashup technology to do this. For example, if we wanted to blend weather information from the National Hurricane Center with a Google Map, the Google map service would need to explicitly support this sort of embedding.
2. In our Second Life scenario, the visible portion of the scene – the part of the texture being displayed – will often be controlled by events generated by other Live Objects that share the display window, perhaps under control of users running on machines elsewhere in the network. These remote sources won't fit into the interaction model expected by the minibrowser.
3. The size and shape of the display window and other elements of the runtime environment should be inherited from the hierarchy structure of the object mashup used to create the application. Thus our texture should learn its size and orientation and even the GPS coordinates on which to center from the parent object that hosts it, and similarly until we reach the "root" object hosting the display window. A minibrowser isn't a component: it runs the show.

On the other hand, minibrowsers retain one potential advantage. Since all aspects of the view are optimized to run together, the interaction controls might be far more sophisticated and perform potentially much better than a solution resulting from mashing up together multiple layers developed independently. Furthermore, in many realistic examples event-based interfaces could get fairly complex, and difficult for most developers to work with.

This observation highlights the importance of developing component interface and event standards for the layered architecture we've outlined. The task isn't really all that daunting: the designers of Microsoft's Object Linking and Embedding (OLE) standard faced similar challenges, and today, their OLE interfaces are pervasively used to support thousands of plugins that implement context menus, virtual folders and various namespace extensions, and drag and drop technologies.

Lacking the needed standards, we've compromised: the Live Objects platform supports both options today. In addition to allowing hosted content to be pulled in and exposed via event interfaces, components developed by some of our users *also* use embedded minibrowsers to gain access to a wide range of platforms, including Google, Yahoo, MSN, Flickr, YouTube, and FaceBook.

VI. Performance Evaluation

Central to our argument is the assertion that hosted event notification solutions scale poorly and stand as a barrier to collaboration applications, and that developers will want to combine hosted content with P2P protocols to overcome these problems. In this section we present data to support our claims. Some of the results (Fig. 3, Fig. 4) are drawn from a widely cited industry whitepaper ([7]) and were obtained using a testing methodology and setup developed and published by Sonic Software ([18]). The remainder was produced in our own experiments.

The first graph (Fig. 3), from the industry white paper, analyzes the performance of several commercial Enterprise Service Bus (ESB) products. Shown is the maximum throughput (msgs/sec) for 1024 byte messages. The experiment varies the number of subscribers while using a single publisher that communicates through a single hosted message broker on a single topic. Brokers are configured for message durability: even if a subscriber experiences a transient loss of connectivity, the publisher retains and hence can replay all messages. As the number of subscribers increases, performance degrades sharply.

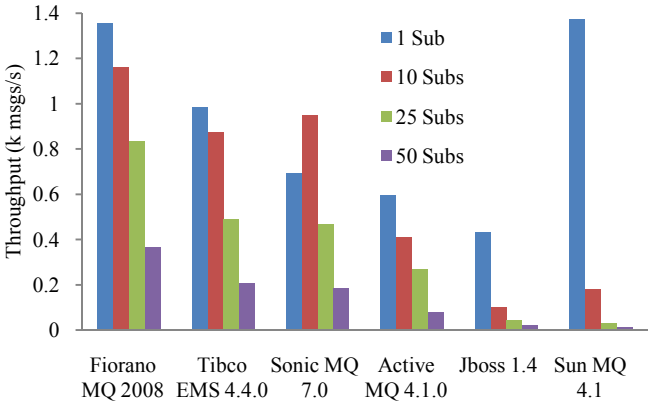

Figure 3. Scalability of Commercial ESBs

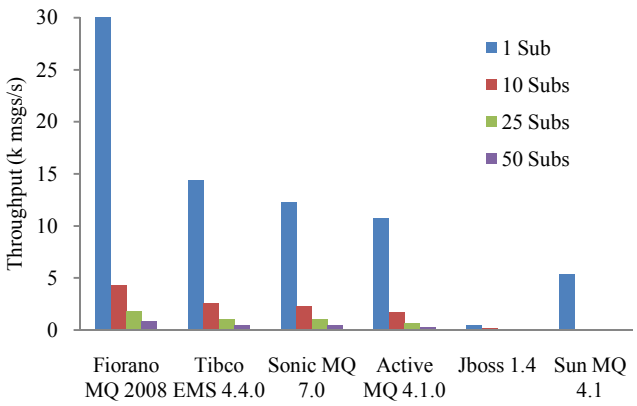

Figure 4. Scalability of Commercial ESBs

Although not shown, latency will also soar because the amount of time the broker needs to spend sending a single message increases linearly with the number of subscribers.

In collaboration applications, durability is often not required. The second graph (Fig. 4) shows throughput in an experiment in which the publisher does not log data. Here, a disconnected subscriber would experience a loss. We find that while the maximum throughput is much higher, the degradation of performance is even more dramatic. One could improve scalability using clustered service structures, but such a step would have no impact on latency, since clients would still need to relay data through the data center. Our point is simply that hosted ESB solutions don't necessarily scale well, and that the client-to-data center communication path could introduce intolerable performance overheads.

Next, we report on some experiments we conducted on our own at Cornell, focusing on scalability of event notification platforms that leverage peer-to-peer techniques for dissemination and recovery. On the first graph (Fig. 5), we compare the maximum throughput of two decentralized reliable multicast protocols, again with 1024-byte messages, a single topic and a single publisher. Unlike in the previous tests, which ran on 1Gbit/sec LANs, these experiments used a 100Mbit/sec LAN; this limits the peak performance to 10,000 messages/second. QSM [11] achieves stable high throughput (saturating the network). JGroups, a popular product, runs at about a fifth that speed, collapsing as the number of subscribers increases. Also, at small loss rates, latency in QSM is at the level of 10-15ms irrespectively of the number of subscribers.

When the number of topics is varied, QSM maintains its high performance. On the second graph (Fig. 6), we report performance for 110 subscribers, but performance for other group sizes is similar. JGroups performance was higher with smaller group sizes, but erodes as the number of topics increases. JGroups failed when we attempted to configure it with more than 256 topics.

Finally, we look at two scalable protocols under conditions of "stress", with a focus on delivery latency (y axis) as a fixed message rate is spread over varying numbers of topics. 64 subscribers each join some number of topics, a publisher sends data at a rate of 1000 messages/second, selecting the topic in which to send at random. Our experimental setup, on Emulab, injects a random 1% message loss rate. In Fig. 7 we see that Ricochet [1], a Cornell-developed protocol for low-latency multicast, maintains steady low-latency delivery (about 10ms; y-axis) as the number of topics increases to 1024 (x-axis). In contrast, latency soars when we repeat this with the industry-standard Scalable Reliable Multicast (SRM), widely used for event notification in their datacenters. As can be seen in the graph, SRM's recovery latency rises linearly in the number of topics, reaching almost 8 seconds with 128 groups.

To summarize, our experiments confirm that:

- Hosted enterprise service bus architectures can achieve high levels of publish-subscribe performance for small numbers of subscribers, but performance degrades very sharply as the number of subscribers or topics grows.
- The JGroups and SRM platforms, which don't leverage peer-to-peer techniques, scale poorly in the number of subscribers or topics. QSM and Ricochet, where subscribers cooperate, scale well in these dimensions.
- Ricochet achieved the best recovery latency when message loss is an issue (but at relatively high overhead, not shown on these graphs). QSM at small loss rates achieves similar average latency with considerably lower network overheads, but if a packet is lost, it may take several seconds to recover it, making it less appropriate for time-critical applications.

We don't see any single winner here: each of the solutions tested has some advantages that its competitors lack. Indeed, we're currently developing new P2P protocol suite, called SOLO [6]; it builds an overlay multicast tree within which events travel, and is capable of self-organizing in the presence of firewalls, network address translators (NAT) and bottleneck links. A separate project is creating a protocol suite that we call the Properties Framework [13]. The goal is to offer strong

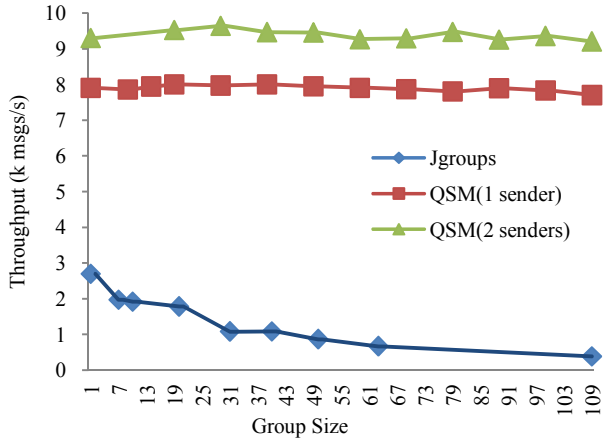

Figure 5. Scalability of QSM and JGroups (throughput for various group sizes)

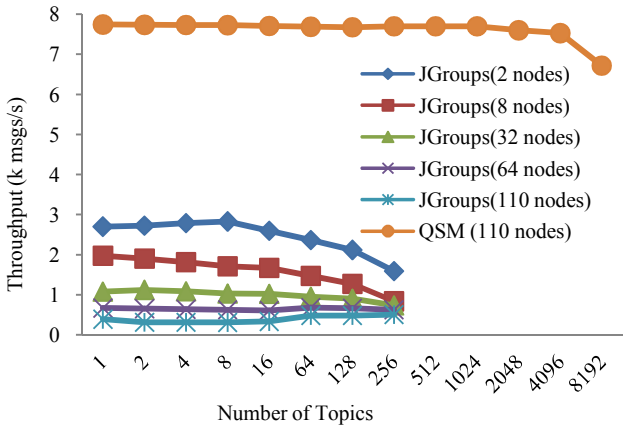

Figure 6. Scalability QSM and JGroups (throughput for various numbers of topics)

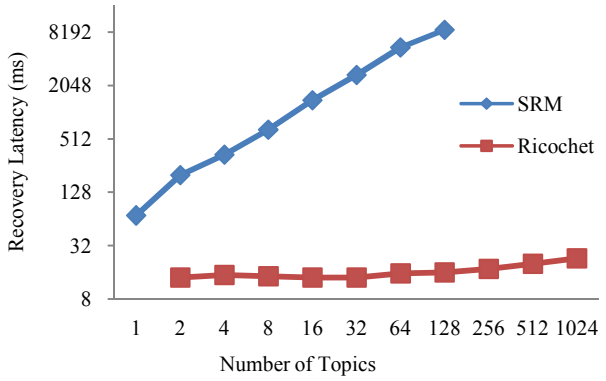

Figure 7. Delivery latency (ms) for SRM and Ricochet with varying numbers of topics.

forms of reliability that can be customized for special needs.

Thus, speed and scalability are only elements of a broader story. Developers will need different solutions for different purposes. By offering a flexible yet structured component mashup environment, Live Objects makes it possible to create applications that mix hosted with P2P content, and that can adapt their behavior, even at runtime, to achieve desired properties in a way matched to the environment.

VII. Prior Work

The idea of integrating web services with peer-to-peer platforms is certainly not new ([2], [4], [8], [9], [10], [14], [15], [16], [20], [21]). The existing work falls roughly into two categories. The first line of research is focused on the use of peer-to-peer technologies, particularly JXTA, as a basis for scalable web service discovery. The second line of research concentrates on the use of replication protocols at the web service backend to achieve fault-tolerance. In both cases, P2P platforms such as JXTA are treated not as means of collaboration or media carrying live content, but rather as a supporting infrastructure at the data center backend. In contrast, our work is focused on blending the content available through P2P and web service protocols; neither technology is subordinate with respect to the other.

Technologies that use peer-to-peer protocols to support live and interactive content have existed earlier; an excellent example of such technology is the Croquet [17] collaboration environment, in which the entire state of a virtual 3D world is stored in a peer-to-peer fashion and updated using a two-phase commit protocol. Other work in this direction includes [19]. However, none of these systems supports the sorts of componentized, layered architectures that we have advocated here. The types of peer-to-peer protocols these systems can leverage, and the types of a traditional hosted content they can blend with their P2P content, are limited. In contrast, our platform is designed from ground up with extensibility in mind; every part of it can be replaced and customized, and different components within a single mashup application can leverage different transport protocols.

Prior work on typed component architectures includes a tremendous variety of programming languages and platforms, including early languages such as SmallTalk alongside modern component-based environments such as Java, .NET or COM, specialized component architectures such as MIT's Argus system, flexible protocol composition stacks such as BAST [5], service-oriented architectures such as Juni, and others. None of these, however, has been used in the context of integrating service-hosted and peer-to-peer content. Discussion of component integration systems and their relation to live objects, however, is beyond the scope of this paper. More details can be found in [12].

Finally, much relevant prior work consists of the scripting languages mentioned in the discussion above: JavaScript, Caja, Silverlight, and others. As explained earlier, our belief is that even though these languages are intended for fairly general use, they have evolved to focus on minibrowser situations in which the application lives within a dedicated browser frame, interacts directly with the user, and cannot be mixed with content from other sources in a layered fashion. Live Objects can support minibrowsers as objects, but we've argued that by modeling hosted content at a lower level as components that interact via events and focusing on the multi-layered style of mashups as opposed to the standard tiled model, we gain flexibility.

VIII. Conclusions

To build ambitious collaboration application, the web services community will need ways to combine (to "mash up") content from multiple sources. These include hosted sources that run in data centers and support web services interfaces, but also direct peer-to-peer protocols capable of transporting audio, video, whiteboard data and other content at high data rates, with low latency. A further need is to allow disconnected collaboration, without mandatory reach-back to data centers.

Our review of the performance of enterprise service bus eventing solutions in the standard hosted web services model made it clear that hosted event channels won't have the scalability and latency properties needed by many applications. P2P alternatives often achieve far better scalability, lower latency, and higher throughput. They also have security advantages: the data center doesn't get a chance to see (and save) every event.

The Live Objects platform can seamlessly support applications that require a mixture of data sources, including both hosted and direct P2P event-stream data. Further benefits include an easy to use drag-and-drop programming style that yields applications represented as XML files, which can be shared as files or even via email. Users that open such files find themselves immersed in a media-rich collaborative environment that also offers strong reliability, high performance, impressive scalability and (in the near future) a powerful type-driven security

mechanism. Most important of all, Live Objects are real: the platform is available for free download from Cornell.

REFERENCES

[1] Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, Stefan Pleisch. Ricochet: Lateral Error Correction for Time-Critical Multicast. NSDI 2007.

[2] Farnoush Banaei-Kashani, Ching-Chien Chen, Cyrus Shahabi. WSPDS: Web Services Peer-to-peer Discovery Service. ICOMP 2004.

[3] Ken Birman, Anne-Marie Kermarrec, Krzysztof Ostrowski, Marin Bertier, Danny Dolev, Robbert van Renesse. Exploiting Gossip for Self-Management in Scalable Event Notification Systems. DEPSA 2007.

[4] Jorge Cardoso. Semantic integration of Web Services and Peer-to-Peer networks to achieve fault-tolerance. IEEE GrC 2006.

[5] Benoit Garbinato, Rachid Guerraoui. Flexible Protocol Composition in Bast. ICDCS 1998.

[6] Qi Huang, Ken Birman. Self Organizing Live Objects (SOLO). Submission to DSN 2009; Dec 2008.

[7] JMS Performance Comparison for Publish Subscribe Messaging. Fiorano Software Technologies Pvt. Ltd., February 2008.

[8] Timo Koskela, Janne Julkunen, Jari Korhonen, Meirong Liu, Mika Ylianttila. Leveraging Collaboration of Peer-to-Peer and Web Services. UBICOMM 2008.

[9] Shenghua Liu, Peep Küngas, and Mikhail Matskin. Agent-based web service composition with JADE and JXTA. SWWS 2006.

[10] Federica Mandreoli, Antonio Perdichizzi, and Wilma Penzo. A P2P-based Architecture for SemanticWeb Service Automatic Composition. DEXA 2007.

[11] Krzysztof Ostrowski, Ken Birman, Danny Dolev. QuickSilver Scalable Multicast (QSM). NCA 2008.

[12] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with Live Distributed Objects. ECOOP 2008.

[13] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Chuck Sakoda. Achieving Reliability Through Distributed Data Flows and Recursive Delegation. Submitted to DSN 2009; Dec 2008.

[14] Mike Papazoglou, Bernd Krämer, and Jian Yang. Leveraging Web-Services and Peer-to-Peer Networks. CAiSE 2003.

[15] Changtao Qu and Wolfgang Nejdl. Interacting the Edutella/JXTA Peer-to-Peer Network with Web Services. SAINT 2004.

[16] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. A Scalable and Ontology-based P2P Infrastructure for Semantic Web Services. P2P 2002.

[17] David Smith, Alan Kay, Andreas Raab, David Reed. Croquet:A Collaboration System Architecture. C5'03.

[18] Sonic Performance test suite, available at: http://www.sonicsoftware.com/products/sonicmq/performance_benchmark/index.asp

[19] Egemen Tanin, Aaron Harwood, Hanan Samet, Sarana Nutanong, Minh Tri Truong. A Serverless 3D World. GIS 2004.

[20] Minjun Wang, Geoffrey Fox, and Shrideep Pallickara. A Demonstration of Collaborative Web Services and Peer-to-Peer Grids. ITCC 2004.

[21] Zhenqi Wang, Yuanyuan Hu. A P2P Network Based Architecture for Web Service. WiCom 2007.