Live Distributed Objects: Enabling the Active Web

Krzysztof Ostrowski, Ken Birman, Danny Dolev

Abstract

Distributed computing has been slow to benefit from the productivity revolution that has transformed the desktop. We still treat the Web as a separate technology space: programmers develop network applications very differently from local ones. "Live Distributed Objects" promise to change this dynamic by enabling an active web, setting the stage for an exciting new style of distributed applications. The key technologies are an exceptionally scalable multicast infrastructure and a clean integration with the runtime environment.

Introduction

Developers who work with modern component integration and productivity tools can create desktop applications faster and more easily than ever before. These tools promote a style of development in which the language, runtime environment, debugger and profiler create a seamless whole.

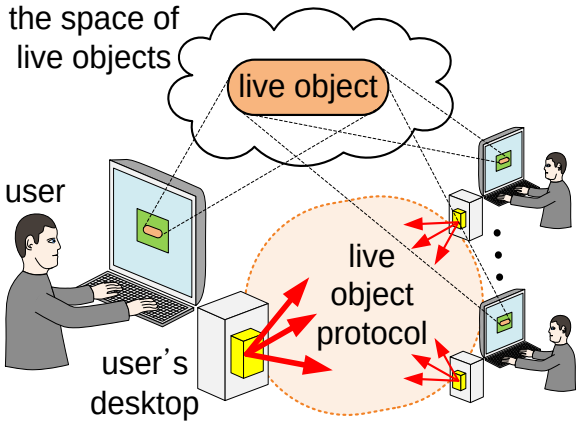

Figure 1. When multiple users across the Internet share a live distributed object, their workstations instantiate local representatives that cooperate to implement whatever abstraction makes the object "live". For example, if the live object is a room in an online game, the object's state could be replicated among the users in this room, and updates would automatically be propagated.

Architects of distributed systems face a more difficult challenge, especially when designing applications that replicate data or services to achieve scalability, guarantee fast response times or to put content close to the end user. The available technologies, such as publish-subscribe message buses or multicast toolkits, are often implemented as free-standing proprietary libraries and are integrated with operating systems and modern component-oriented development tools only in superficial ways.

At Cornell, our team has been exploring ways to bridge this gap. We've created *Live Distributed Objects* ("live objects," for short) that have the "look and feel" of ordinary objects in managed

environments such as .NET or J2EE. Unlike the objects already supported by such environments, live objects need not reside at a single location. A live object can be understood as a distributed mechanism whereby a group of software components can communicate with each other, share data, exchange events, or coordinate actions in a decentralized, peer-to-peer fashion (Figure 1). A live object can represent, for example, a streaming video, a news channel, a collaboratively edited document, a replicated variable, or a fault-tolerant service. The approach is intended to scale. Eventually, we hope to support Internet deployments that might have, e.g., tens of millions of IPTV channels, new forms of collaboration and gaming environments, and new forms of self-managed applications that could literally span the globe.

From the programmer's perspective, live objects replace earlier technologies such as multicast, group communication, publish-subscribe, or state machine replication. These technologies are often proprietary and fit poorly into the modern component-oriented style of development. In contrast, live objects fit easily into environments such as .NET and J2EE, and can leverage their powerful type systems and "management" features. Development and debugging tools work in a natural way.

Live objects are also easy to use. End-users can browse through virtual folders containing thousands of live objects such as chat sessions, enter rooms in a multiplayer game or shared documents, just as they browse through local folders, and can place shortcuts to live objects on their desktop. Applications can access live objects much as they access files. Eventually, it should be possible to drag and drop live objects to create live documents -- web pages, slide shows, and so forth.

What makes live objects so exciting is that they are a gateway to a completely new vision for the Web: they enable a Web that, for the first time, is truly *active*.

Live Distributed Objects enable a Web that, for the first time, is truly active.

The Active Web

To understand the concept of an Active Web, it will be helpful to think about *Second Life* [http://www.secondlife.com], a popular virtual reality environment that has gained millions of users in just a few years. In the words of its inventors, "Second Life is a 3-D virtual world entirely built and owned by its residents… a vast digital continent, teeming with people, entertainment, experiences and opportunity". A typical Second Life scenario might start when a user's avatar walks into a smoke-filled bar, where a seedy collection of creatures are playing poker around a stained oak table. The avatar pulls up a chair, joins the game, and perhaps the user makes (or loses) a fortune – in real money.

Today, an application such as this would be hosted by a big datacenter. But our vision is of a future in which data would flow directly from the applications that generate it (for example, that render an avatar) to the ones that consume it (for example, by displaying the room). Not only can such an approach support much higher data rates with lower latency, but it can also scale better and promotes better security and privacy . Thus, an active web based on live objects could be a world with millions of IPTV streams, new forms of interactive art, live electronic health records that integrate regional medical providers, or banking systems that could trade "live" financial instruments.

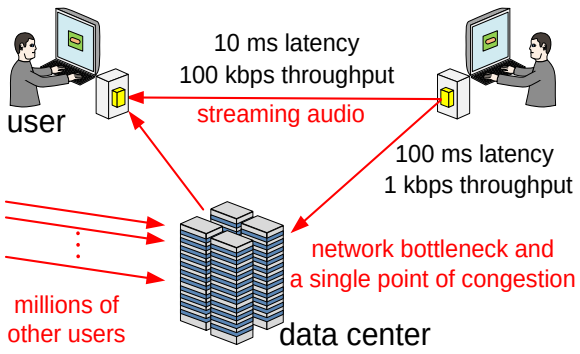

Figure 2. Existing massively multiplayer online games and virtual worlds rely on server farms. These keeps the entire system state and all player interactions occur indirectly, through a shared server. If users communicate directly with one another in a peer-to-peer fashion, server loads are slashed, latency and throughput are potentially improved, and privacy is better protected.

These applications will be tricky to build. Consider electronic health records. Such a system would need to achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario involves decentralized systems linked over networks, integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, etc. Electronic monitoring devices and other sensors running both in the hospital and at home will contribute time-sensitive data, and some therapeutic and drug delivery devices will be remotely controlled.

We want to enable an Active Web supporting these kinds of demanding applications. The Active Web will be a place in which applications can be constructed by composing (gluing together) live objects of various kinds – objects that might embody strong guarantees, such as security, privacy, or fault-tolerance.

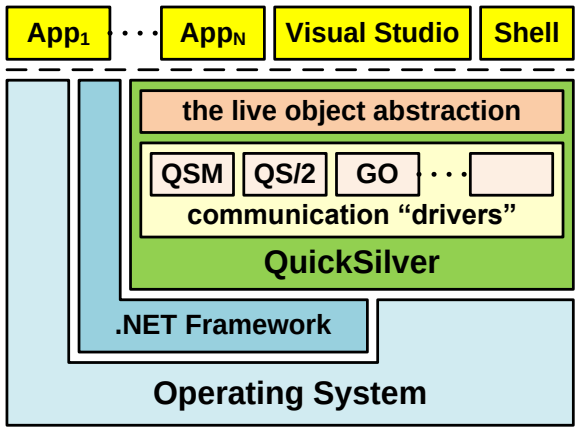

Figure 3. Live objects provided by QuickSilver can be accessed programmatically from applications, from Windows shell like files, or from development environments such as Visual Studio. QuickSilver is composed of two layers: a layer that exposes the virtual objects to applications as a part of the .NET typesystem, and an extensible communication engine with "drivers" for different object types.

Live Distributed Objects in the Quicksilver System

Cornell's Quicksilver system offers a glimpse of

the live object concept in action. Quicksilver is a system built in two layers (Figure 3). One layer extends a system such as .NET to support live objects, and focuses on the embedding of these objects into the .NET Common Language Runtime, and the hooks connecting the objects to the .NET type system and to the Windows shell (the GUI that interprets mouse actions, such as right-clicking on a file). A second layer of Quicksilver provides the scalable and extensible communication infrastructure needed to make the objects "live" and "distributed".

Briefly, here's a summary of the abstract functionality provided by a live object:

1. Each object has an Internet-wide unique name and is registered in a global distributed "directory" resembling a DNS, where it can be located by the software components that wish to access it.
2. Live objects have "distributed" types. Among other things, type-checking helps verify that the communication "type" beneath an object matches the developer's intent.
3. A live object provides a "natural" interface to its clients. An object representing a video stream would provide methods to send or receive video frames. An object that represents a replicated variable would provide get/set methods, etc.
4. Each object also has object-specific "logic" implementing some abstraction: a room, a medical record, a gossip-based overlay network, etc. This logic is translated into code that is deployed on and executed by the components accessing the live object. The protocols for replicating the object's data among components and for multicasting events among them are provided by type-specific underlying "communication drivers".
5. When a component tries to access a live object, an authentication and join protocol executes. This is used to enforce security, fetch the object's "logic" from the distributed live objects directory, obtain a snapshot of the object's state, initialize it etc. The object's code maintains its state in response to event notifications.

The underlying communications layer is composed of a set of "pluggable" communication substrates, which play a role analogous to that of a device driver. Each live object needs a "communication driver" to replicate its state, propagate events and updates; different objects might use different drivers specialized for different settings (wireless, WAN, LAN) or properties (secure, low latency, etc). QuickSilver allows existing multicast and group communication toolkits to be adapted for use as communication drivers. It also comes with a suite of built-in scalable and high-performance communication drivers for common types of applications, and designed to complement each other:

- **QSM** (Quicksilver Scalable Multicast) is optimized to support large numbers of live objects that may represent streams of events, such as video channels, file backup folders, or stock price update notifications in a trading system. The current version of QSM is designed for enterprise LANs and datacenters. It can support tens of thousands

of event streams, hundreds of users, and transmissions at network speeds. Events are delivered reliably, and the system is robust under stress.

- **QS/2** (QuickSilver 2) is optimized to support live objects with strong reliability properties, such as replicated variables in a banking system or collaboratively edited documents, where strongly consistent replication and transactional updates are needed to preserve data integrity. QS/2 supports a wide range of reliability models, and even custom, user-defined types of reliability. This driver is optimized for use in WAN scenarios.

- **GO** (Gossip Objects) is under development at IRISA and the University of Rennes, in France. It uses the so-called gossip protocols to build live objects that can help systems manage themselves, set their own configuration parameters, and even diagnose and self-repair when failures occur.

We expect this list to grow over time, with protocols to support wireless communication, IPTV, security protocols, and protocols with real-time properties.

How can it scale?

Although brevity precludes a highly technical discussion of scalability, we can still give readers a rough intuition into how we've tackled this problem in QSM. QSM's role is to reliably multicast data in support of higher level "events" related to a number of live objects, such as updates to their state, commands that cause actions to occur, etc. In a typical datacenter scenario, some computers might be using large numbers of live objects, and different live objects could be shared by the same computers. The groups of computers on which these live objects "live" might thus overlap. The world of QSM is one of vast numbers of "overlapping" groups.

Let's make a simplifying assumption (we can remove it later). We'll assume that the overlap is clean and hierarchical. For example, there might be "big" objects A, B, and C directly superimposed, i.e. such that the groups of computers using them overlap perfectly. The hierarchy could also include subsets. For example, perhaps object D lives on half of these computers, and E on the other half. QSM starts by decomposing such a hierarchy of overlapping groups into a set of *regions*, by clustering computers based on their "interest". Each region contains computers that use the same live objects.

QSM now constructs a data dissemination overlay for each region. If possible, IP multicast is employed for this purpose (e.g. each region could be assigned its own IP multicast address). However, since IP multicast isn't always available, QSM can also run on some form of overlay multicast technology. Dissemination is *unreliable*: like a UDP transmission in the Internet, a message might reach none of its destinations, some of them, or (if we are really lucky) all.

Next, QSM builds a peer-to-peer repair structure within each region. This involves many subtle issues, but the basic approach starts by constructing a logical token-ring that links the computers in the region (if a region gets larger than about 25 computers, we break the ring into a tree of smaller rings, linked by a higher-level ring). As the token travels around the ring, a few times per second, we efficiently encode the received message set of each computer. A machine that has a copy of a message some other machine lacks forwards a copy. If an entire region lacks a multicast, the sender re-multicasts it. In our experiments this is very rare; most packet loss involves a single machine that drops a single packet or a few in a row, and can be repaired locally with the help of a nearby peer.

Clustering computers into "regions" enables QSM to handle large numbers of live objects efficiently, by amortizing overhead. While in a traditional system, tens of thousands of live objects would involve tens of thousands of multicast protocols, QSM can use a smaller number of token rings, each of which works for multiple objects at once.

To handle computer crashes, QSM incorporates a hierarchical status-monitoring service structured a bit like the Internet DNS, but designed to track the health of components (live or failed), as well as some additional information used within our protocols. A consensus protocol is employed to ensure that the membership decisions will be consistently reported.

Obviously, we're skipping a lot of details, such as the flow-control mechanism used, data aggregation when multiple small messages are sent to the same group, etc (interested readers can find more information in the technical reports on our web site). But the upshot is that QSM seems to break every performance record we're aware of. We've scaled it to many thousands of groups (live objects) per computer, supported groups with hundreds members, and are able to saturate 100Mbit Ethernet interconnects with inexpensive PCs on the endpoints. The system is stable under stress, and very well tolerates load fluctuations, broadcast storms, or other degenerate behaviors. Moreover, even at the highest loads, overheads are quite low.

Now, all of this reflected a simplifying assumption, namely that groups are hierarchical. But it turns out multiple QSM hierarchies can be superimposed (Fig. 4). We're finding that even very irregular sets of overlapping groups can be "covered" with a surprisingly small number of hierarchies, provided that groups have Zipf-like popularity and traffic levels. For example, studies of financial instruments show that the i'th most popular stock or bond tends to be popular in proportion to $1/i^{\alpha}$ where the exponent, α, can be as large as 2.5 to 3.5. It seems reasonable to assume that in the Active Web, if applications use large numbers of live objects in irregular ways, the same property would hold.

We don't have room here to discuss the QS/2 or GO architectures, and hence will limit ourselves to just a few words on each. QS/2 extends QSM by introducing a novel programming language, interpreted at runtime, that tells the system how to implement a desired reliability model. Using this language we know how to support live objects with stronger semantics (we'll say more about this question below).

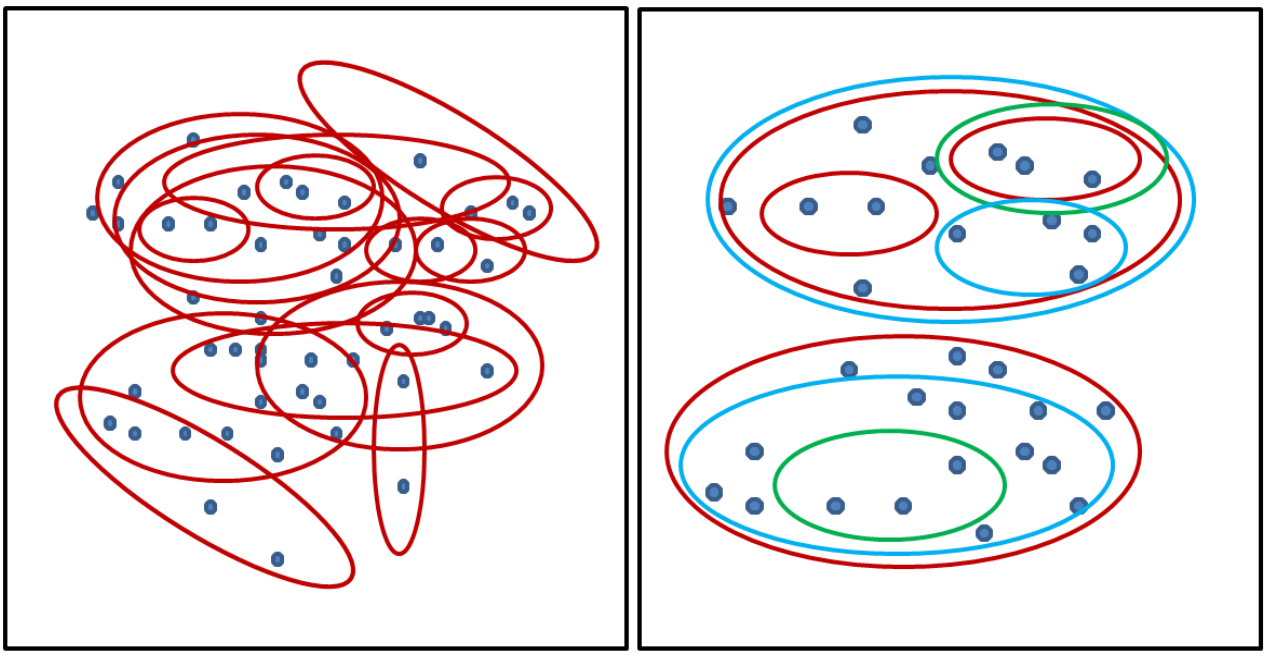

Figure 4: [left] Irregularly overlapping groups of objects arise if live objects are used in an unstructured manner (dots represent computers; oval rings groups of live objects, for example replicas of a shared variable). [right] QSM scales best when groups nest hierarchically, as in these two examples. Our studies suggest that irregularly overlapping groups can often be mapped to the case QSM can handle by superimposing multiple hierarchies.

GO takes live objects in a completely different direction. This subsystem builds a random peer selection layer and then constructs a variety of gossip and epidemic mechanisms over it, presenting these as live objects that can perform tasks such as tracking overall system state, assisting in auto-configuration or repair, and constructing the overlay networks needed for QSM's dissemination layer.

Thus, we have a fast and scalable but limited reliability option in QSM, a robust and self-configuring but relatively slow technology in the case of GO, and a way to support strong properties, through QS/2. The future, of course, will probably bring additional options to extend this list.

What's in a type?

Live objects open the possibility of extending normal type systems to encompass distributed behavioral patterns. A replicated variable that represents an account balance in a banking system might need strong reliability and fault-tolerance properties such as virtual synchrony, while for a similar variable in a monitoring application, weaker reliability properties and scalability of gossip might be a better match. Thus, to make live objects truly useful, we need a way to describe such behaviors as a part of their types.

In QuickSilver, the type of a live object is a tuple, the elements of which specify different "aspects" of the type (much like in aspect-oriented programming). One element is the object's interface (in the usual sense). Another is its "category" (replicated service, replicated variable, event stream, gossip object etc.), which tells which communication protocol driver to use.

Other aspects configure the underlying dissemination substrate, and specify the object's reliability, fault-tolerance, and security properties.

By expressing distributed types this way, we enable a next step in which type information could be used as part of the application design and implementation process. A development environment such as Visual Studio would "understand" the possible distributed behaviors of such a typed object, and could guide the developer through the process of implementing code that will run correctly under the assumption that the communication drivers underlying the live object implement the specified behaviors. Moreover, the runtime system can throw exceptions if mistakes are made, for example if a component designed to work correctly only with live virtually synchronous multicast streams tried to access a live multicast stream that has a weaker QSM or best-effort reliability property.

Type-based programming tools and debugging tools have transformed the experience of building applications for desktop environments. Service oriented architectures (which also revolve around type systems, albeit simple ones) are having a similar impact in networked applications that interact with services hosted in datacenters. Live objects are a natural step in this direction, and bring the benefits of strong typing to the realm of decentralized peer-to-peer applications. They suggest that the active web could be far more than just a veneer over the same old Internet technologies.

Quicksilver Properties Framework

Earlier, we commented that the QS/2 system will extend QSM by allowing some groups to have stronger reliability properties, defined using a high-level declarative *Properties Language*. Using this language, we are able to support a number of important reliability models – "reliability types". Let's look at two of these models and how they might assist the developer of the gambling parlor in our Second Life scenario:

- **Virtual Synchrony** is a powerful distributed computing model in which active programs join *process groups*, within which multicasts are used to disseminate updates and other events. At the moment a process joins, it can initialize itself using a *state transfer* from some active member, and the virtual synchrony platform notifies all members each time membership changes. The power of this model is that it can support consistency guarantees: all the users of a virtual synchrony group see the same thing in the same order. This can be important when multiple users are concurrently taking actions, and yet the "physical world" needs to somehow order them. For example, in some card games, users slap cards onto a stack – first card "wins". Using traditional event notification solutions there can be cases where different users would see the same events in different orders – and in our saloon, it's easy to imagine (virtual) fights erupting in such cases! With virtual synchrony,

the Quicksilver platform would enforce a single, system-wide event ordering, hence all participants see the same events in the same sequence.

- **Transactions** are a stronger execution model. Suppose that a power failure were to cause a few machines to crash simultaneously. Although we didn't make this clear in our discussion of virtual synchrony, that model only provides consistency among live objects that remain operational. If an object crashes and then restarts, saved data from its previous life must be discarded. But suppose that the object represents the bank and real money is changing hands. Transactional live objects support the full one-copy serializability version of the ACID model; with this guarantee, crashed objects can reconstruct a consistent, agreed-upon state after they recover. However, these stronger guarantees come at a (steep) price: performance and scalability won't be nearly as good.

Will the Active Web be the next big thing for the Internet? At Cornell, we're making it happen!

In QS/2, each object can be configured to have the kind of consistency guarantee appropriate to the way it will be used. Different models are expressed using a kind of script that QS/2 "executes" to control the delivery of messages and other events, and to trigger actions such as the forwarding of a message to repair a loss, or the sending of ordering information when a batch of messages must be placed into a total order.

Aiming for the Active Web

Most of the live objects system is now operational in the lab at Cornell. QSM can be downloaded by interested developers. As noted earlier, this communications driver is breaking every performance record we're aware of in the technology space: raw throughput, scalability, tolerance of stress, and robustness against broadcast storms and other forms of oscillatory behavior. The remainder of the system (including QS/2) is running as an experimental prototype and tested under laboratory conditions. Elements still under development include a configuration manager service that can run at Internet Scale, the GO subsystem mentioned above, and many elements of the runtime and debugging/performance-tuning environment.

When completed, early in 2008, we believe that live objects will enable a completely new kind of distributed programming, inspired by the Web, but in which much of the content is dynamic and may be evolving in real-time. Live objects could represent video feeds, streams of media or other content generated by participating computers, telemetry from sensors, etc. By integrating such content into systems like Windows or J2EE in a clean and natural way that leverages the power of type systems and component integration technologies, but offers a

portal to distributed computing, we're hoping to enable a revolution. Live objects could open the door to a new and disruptive generation of Active Web applications that combine high data rates with strong properties, including fault-tolerance, consistency, and security.

Will the Active Web be the next big thing for the Internet? It's too early to know, but at Cornell, we're betting that live objects will make the Active Web a reality.

Learning More

Readers wishing to learn more about our work will find a collection of technical papers at the primary Quicksilver web site, http://www.cs.cornell.edu/projects/quicksilver (this also has a free download link). The team developing the Gossip Objects infrastructure maintains a web site at http://www.irisa.fr/asap. Birman's book, *Reliable Distributed Systems: Technologies, Web Services, and Applications* (Springer-Verlag; 2005), covers many of the protocols and reliability concepts mentioned above. A short article discussing some of the technical issues we face is available as "Exploiting Gossip for Self-Management in Scalable Event Notification Systems." Ken Birman, Anne-Marie Kermarrec, Krzystof Ostrowski, Marin Bertier, Danny Dolev, Robbert Van Renesse. *IEEE Distributed Event Processing Systems and Architecture Workshop (DEPSA-07).* Toronto (June 2007).

Acknowledgements

Our research was supported by grants from AFRL/IF under the "Castor" effort, AFOSR, the TRUST NSF Science and Technology Center, the NSF Cybertrust program and Intel.