LiveTalk: A Framework for Collaborative Browser-Based Replicated-Computation Applications

Matthew Hemmings
University of Victoria
mhemming@uvic.ca

Daniel Ingalls
Y Combinator Research
danhhingalls@gmail.com

Robert Krahn
Y Combinator Research
robert.krahn@cdglabs.org

Rick McGeer
US Ignite
rick.mcgeer@us-ignite.org

Glenn Ricart
US Ignite
glenn.ricart@us-ignite.org

Marko Röder
Y Combinator Research
marko.roeder@cdglabs.org

Ulrike Stege
University of Victoria
ustege@cs.uvic.ca

***Abstract*—In this paper we describe LiveTalk, a framework for Collaborative Browser-based Replicated-Computation applications. LiveTalk permits multiple users separated across the wide area to interact with separate copies of a single application, sharing a single virtual workspace, using very little network bandwidth. LiveTalk features an integrated, browser-based programming environment with native graphics and live evaluation, an integrated, pluggable web server, and a simple messaging service that serves to coordinate activity on shared application sessions, and provides for multiple, mutually-isolated sessions. The first use case for LiveTalk are collaborative big-data visualizations running on thin-client devices such as cellular phones, tablets, and netbooks. These applications form part of a new class of application where the distributed Cloud is leveraged to provide low latency, and high-bandwidth access to geographically disparate users while maintaining the feel of immediacy associated with local computation. The primary motivation of this work is to permit low latency, collaborative applications to be built quickly and easily, while requiring no setup for use by the end-user.**

I. Introduction and Motivation

Cloud-based applications have a substantial number of advantages over fat-client-based desktop applications, one reason many applications have migrated to the web and the Cloud in recent decades. Application state is resident, not on a desktop, but in the Cloud, and is generally immune from local failures such as disk crashes, loss or theft of client equipment, and so on; users can access the application from anywhere there is network connectivity, using any device, including smartphones, tablets, and netbooks; the application runs in a controlled, known environment, eliminating many porting difficulties. For enterprises, thin clients and servers are easier to maintain than fat desktop PCs. Routine Information Technology tasks – data backup, etc. – are handled by the Cloud provider automatically. There is no need for on-site support or maintenance.

Collaboration is inherent in Cloud applications. An excellent example is the Google suite of productivity applications: Google Docs, Google Sheets, and Google Slides. At first glance, these are mere clones of a standard desktop suite, such as Microsoft Office or Open Office. However, a major feature is collaborative editing. Document co-authors can see, in real time, who else is editing the document and the changes that each one is making, which radically reduces the standard revision cycle of modify-then-email.

Cloud platforms and operating systems tend to be more secure than desktop systems, largely due to their heritage – most descend from the more robust and more secure Unix platform. Further, it is far easier to recover a Cloud platform from malware or attack: one simply deletes the infected virtual machine and restores to a pristine, pre-infected image.

Moreover, Cloud-based applications have now become possible, largely due to the HTML5 platform. The web has completed its evolution from being a document-exchange system to a full-fledged programming platform, with every feature available on a traditional desktop system. This includes a rich graphics platform, including 3D graphics.

There are three limitations that have slowed the uptake of Cloud-based applications:

- The need for a ubiquitous Internet connection, and in many cases a very high-speed connection.
- The remote nature of current "Classic Cloud" platforms, where the closest Cloud node is hundreds or thousands of kilometers distant, dramatically reduces the responsiveness of Cloud-based applications. Many applications are "fat data" systems – they require large data transfers between the Cloud node and the web client. For this reason, data-heavy applications such as Geographic Information Systems rely on local fat-client implementations, in order to minimize latency in response to a user request.
- Bandwidth costs, particularly for long-distance transmissions, are not dropping as rapidly as computation costs. What matters is the *ratio* between computation and communications costs, since the former represents our ability to capture, generate, and process data, and the latter our ability to transmit it. This ratio has dropped by two orders of magnitude over the past 20 years[12], which means that networks are becoming increasingly over-stretched. On the side of the application consumer, this disparity appears as per-month bandwidth caps; on the side of the application provider, as direct bandwidth charges; and on the side of the network-provider, as an exponentially increasing demand for bandwidth.
- Privacy and regulatory concerns often dictate that

978-0-9883045-1-2/16 $31.00 © 2016 ITC

application data must be kept in specific political jurisdictions. For example, many universities in the United States have outsourced their office applications and email to Google, using locally-branded versions of GMail and Google Apps for Enterprise. In contrast, universities in the Canadian province of British Columbia are required by local law to keep student data in Canada.

The need for an Internet connection is immutable. However, the rise of the distributed Cloud eliminates the other limitations, by bringing the Cloud closer to the user. This increases responsiveness and bandwidth to the user dramatically, effectively to the responsiveness of a program resident to a desktop PC, and eliminates regulatory and jurisdictional questions because application data is now resident in the user's jurisdiction. Moreover, bandwidth costs are primarily long-haul costs on level-2 and level-3 providers; local bandwidth is cheap and plentiful.

This has led to the development of a number of application-specific Edge Clouds. The most familiar of these are content-distribution networks such as Akamai, Coral, CoDeeN, and the Google Global Cache. However, sufficiently popular single applications develop application-specific Edge Clouds interconnected by private networks. For example, the League of Legends game is the leader in e-sports, with the world championship filling 20,000-seat basketball and hockey arenas. Fair competition requires that the game server be under the control of League of Legends, and be equidistant from, and close to, the competing parties – no home-server advantage, and with guaranteed bandwidth to the players (League of Legends is a classic fat-client game, so in practice bandwidth requirements are relatively modest; but quality-of-service is vital). For this reason, Riot Games, the maker and operator of League of Legends, operates its own Edge Cloud of League of Legends servers with a private network between them. Riot's explanation for this is:

> "Currently, ISPs focus primarily on moving large volumes of data in seconds or minutes, which is good for buffered applications like YouTube or Netflix but not so good for real-time games, which need to move very small amounts of data in milliseconds. On top of that, your internet connection might bounce all over the country instead of running directly to where it needs to go, which can impact your network quality and ping whether the game server is across the country or right down the street.
>
> This is why we're in the process of creating our own direct network for League traffic and working with ISPs across the US and Canada to connect players to this network."[33]

For this reason, there have been a number of implementations of Distributed Edge Clouds over the past decade, from PlanetLab[32], to GENI[8], [29], SAVI[22], and G-Lab[31]. These can be made very efficient[6], [7], [4], and in fact a downloadable Edge Cloud which can run in any available VM has been proposed[5]. These are all Platform-as-a-Service (PaaS) Distributed Clouds, which offer some variant of Linux Virtual Machines or Docker Containers as the execution environment.

Platform-as-a-Service environments are common programming environments instantiated on top of Infrastructure-as-a-Service (IaaS) platforms. These offer both higher-level programming abstractions and more facilities for the application developer and greater efficiency in the use of the underlying infrastructure. Commercial examples include Heroku[30], AWS Elastic Beanstalk[43], and Google App Engine[36]. Though some features of commercial PaaS facilities are not relevant for a distributed Cloud environment (e.g., automated scaling), the productivity enhancements certainly are. LiveTalk is a PaaS environment tuned to delivering sophisticated Cloud applications in a web browser. We describe LiveTalk here.

The remainder of this paper is organized as follows. In Section II we describe the architecture and initial implementation of LiveTalk. In Section III we discuss the inter-instance messaging system used to coordinate activities in LiveTalk. In Section IV we describe two example applications that we have built on the LiveTalk framework. In Section V, we discuss conclusions and future work.

II. The LiveTalk Architecture

A typical PaaS environment is a web framework with an automated deployment capability. To a developer, it looks very much like programming Ruby on Rails[40], Django[13] or Node.js[35], with the various deployment scripts and server plumbing taken care of. Client support is generally limited to a thin overlay on HTML, usually a template engine such as Jinja2[17] or Jade[16]. The major features of such frameworks is that database access and manipulation is done transparently through an API, so there appears to be little difference between database manipulation and manipulating a program's data structure, and REST requests are transparently translated into method calls.

LiveTalk differs from most PaaS environments in that it presents a much higher-level client abstraction than raw HTML5. The foundational layer of LiveTalk is the Lively Web[23], [19], [39]. The client abstraction presented by Lively is a graphical user interface based on the Morphic[26] graphical interface, pioneered in Self[42] and used in Squeak[15] and Scratch[25]. Though the Lively UI is rendered using HTML5, this is very much a low-level rendering tool. The actual objects created and manipulated in a Lively Web page (called a "world") are Morphic objects. Morphic is a user interface framework that supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as HTML tags, CSS styles, fonts, and color maps. A primary goal of Morphic is to make it easy to construct and edit interactive graphical objects, both programmatically and

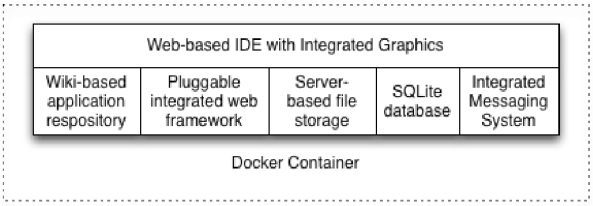

Figure 1. LiveTalk System Architecture

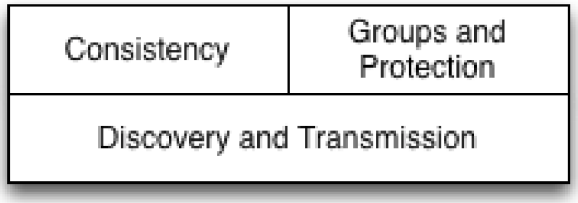

Figure 2. LiveTalk Messaging System Architecture

by direct manipulation.

The easiest way to think about Lively is to consider it a JavaScript-based implementation of Squeak, its immediate predecessor. From Squeak it inherits its fundamental UI and live-evaluation programming semantics (any piece of text in a Lively world can be evaluated as a JavaScript expression); since it is implemented in JavaScript and runs in any browser, any JavaScript library or HTML widget can be incorporated in a Lively world.

Lively comes with an integrated editor, Ace[1], an integrated Wiki for version control, an integrated WebDAV[10] environment for server-side file storage, an integrated SQLite[2] database with client- and server-side APIs, an integrated messaging system which uses socket.io[20] as an underlying transport layer, and an integrated, pluggable Node.js server which both acts to serve the Lively pages themselves and can be used to add server-side code. And just like the client, the server can be programmed through an editor on a Lively page.

The Node.js pluggable server with an integrated file store (WebDAV) and SQLite database would, by themselves, make Lively a competitive web framework. However, the client-side Morphic-based programming environment, integrated Wiki, and messaging system are unequaled in other web development environments. Of particular note is that the WebDAV, database, and messaging systems all have substantial client-side APIs, so for many applications server-side programming is entirely unnecessary; indeed, we have built two-player games with no server-side programming at all.

The architecture for LiveTalk is shown in Figure 1. It should be noted that most of the features of the LiveTalk system are present in its Lively Web base; the contribution for a distributed environment is an enhanced messaging system to support privacy and isolation in collaborative applications and its integration with the GENI Experiment Engine (and, eventually, the PlanetIgnite) Distributed IaaS platform.

Developers writing a LiveTalk application will typically encapsulate the client-facing portion of the system in Lively worlds (Morphic applications written in JavaScript and encapsulated on a web page) that are stored in the Wiki-based application repository. In many cases, theu will write a Node.js subserver as well, or (to avoid fate-sharing) write a separate data server which can be deployed alonside the LiveTalk server, in a companion container. They may also interact directly with the embedded server-side SQLite database through the provided API and/or use the integrated WebDAV client-side API to manipulate server-side files.

III. The LiveTalk Messaging System

The LiveTalk messaging system is based on the Lively Web's integrated Lively2Lively message protocol. The goals of Lively2Lively were to permit:

- Pure message based communication between Lively worlds,
- A transparent network abstraction, with simple put/get semantics on messages,
- Graceful handling of churn and parties leaving and entering systems, and
- Automated handling of messages.

The Lively2Lively system ensures reliable delivery and participant tracking in a worldwide setting through a hierarchy of session trackers, using either a client-side or a server-side API; most applications use the client-side API. Each session starts with a unique, randomly chosen UUID that registers with a session tracker on the Lively server that served the world. Sessions can use the local API to discover other sessions, by URL, user, or UUID, to send messages and receive messages.

The wire protocol for a Lively2Lively message is a JSON object, which **must** have the fields **action** (a string) and **target** (a UUID), which specifies, respectively, the subject and receiver of the message. Optional fields specify the sender, a unique ID for the message, an identifier of a message which prompted this message as a response, if any, and a **data** field, which is a JSON object containing any message data.

A session registers to handle messages of for a specific *action* with the **registerActions** method in the Lively2Lively session tracker. **registerActions** takes a JavaScript dictionary as an argument, where the entry **messageName: function(msg, session)** indicates that the function on the right-hand side is used to handle messages with the *action* **messageName**.

Message routing is handled seamlessly by the Lively servers in a manner similar to SMTP[18]. The server's session tracker determines if it is the designated recipient of the request. If so, it delivers the message. If the recipient is a session tracked by this server, it delivers to that session, otherwise it passes it to a peer for delivery.

Lively2Lively offers the base message delivery layer for LiveTalk, but further functionality is required for a distributed application platform. We see the architecture of a distributed messaging system as similar to the stacked

services in the Internet, where each layer offers functionality building on the layer below. An architectural diagram showing some of the services appears in Figure 2.

Lively2Lively forms the base layer of the messaging system: discovery and reliable transmission, including – in the future – scalability features like multicast and publish/subscribe. The LiveTalk messaging system focusses on the next layers above the discovery and transmission layer: attaching actions to objects and offering protection and privacy for message users and groups.

We begin with the first problem: attaching messages to objects and choosing which recipient on a page is the intended recipient of a message. The choice for the first question is what is the appropriate granularity of a message participant? The choices are: Lively worlds, Lively2Lively sessions, individual morphs, or a new, abstract entity.

Worlds and Lively2Lively sessions are too coarse for the appropriate interface. A world may have multiple morphs participating in various different applications, and the odds of a name conflict are quite high. Since morphs are the general unit in Lively, using a Morph is tempting. However, conversation participants do not need to be tied to a specific graphical object, a single graphical object can have multiple messaging interfaces, and, perhaps most of all, there is no compelling reason to do it. One must remember the First Commandment of Modularity: "Thou Shalt not Overload a Concept Without an Excellent Reason"[1].

Conversations Our choice is therefore a new object, the *Conversation*. We follow the JavaScript convention: a *Conversation* is simply an object, which defines a set of messages to which a Conversation object responds. A Conversation also has a *name*, which defines the set of messages to which it responds.

The Conversation resembles nothing so much as a Java Interface, and faces the same basic constraint: the name of a Conversation must be unique across applications. And thus we chose the Java solution: to use a reversed URL, followed by the user id of the Conversation developer, followed by the name of the Conversation (chosen arbitrarily by the developer). Dots are used to separate the fields, and developers can choose a hierarchical namespace for their own Conversations if they prefer. An example Conversation name is `org.lively-web.www.matt.visualizer`, or `org.lively-web.www.matt.visualizer.pollution`.

The Conversation name defines a set of messages, so each Conversation has a method: `messages` which returns the names of the messages to which a participant in the Conversation responds. The *full name* of a message is the name of the Conversation followed by the name of the message, separated (as always) by a dot. For example, the `showData` message of the converation `org.lively-web.www.matt.visualizer` is `org.lively-web.www.matt.visualizer.showData`.

[1]Like the other commandments of information technology ("Real Operating Systems end in X"; "If It doesn't install with apt-get, walk away"; "Design the API first, then the GUI"; "Java? Ewwwww...") the punishment for disobeying this Commandment is death by a million bugs

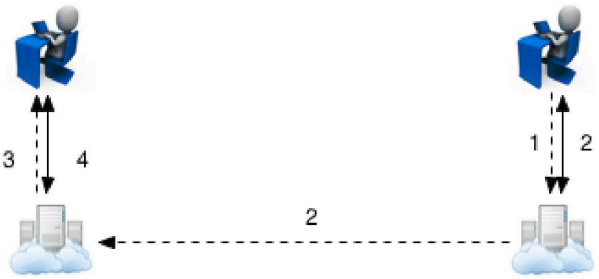

Figure 3. Typical Message and Information Flow

Groups The unit of protection is the *Group*. A group is simply a name, an owner, a set of receivers, a key, and a Conversation. Every user – owner or receiver – is a username and the according UUID. A group's *owner* decides on admissions to the group.

To join a group, an object calls `LiveTalk.joinGroup(conversationName, groupName, messageMapping)`, where `messageMapping` is an object of the form `messageName: function(message)`, declaring the message handlers for each message in a `Conversation`. The return value of `joinGroup` is a Group object, or None if the joining the group failed. The message to join a group is delivered to the group owner (person or agent) which returns the appropriate response.

The key is optional and is used for group security: to prevent non-members from sending messages to the group. When members send a message to the group, they cryptographically signs it using the key. On receipt, the LiveTalk session manager reads the signature, and, if the message is signed appropriately, invoke the registered message handler with the message as an argument.

The resulting, typical information and message flow is shown in Figure 3. Message flow is depicted by dashed arrows and data requests and transfers by solid arrows; the numbers on the arrows represent the time sequence of message flows. One participant in a LiveTalk-enabled application performs an action which manipulates the shared visual space. This will typically, but not always, involve fetching data from the server. The LiveTalk application will simultaneously issue a data request to the server and issue a message to the other participants describing the manipulation of the shared space. When the data request is serviced, the local screen is redrawn in response to the request. Once the message is received by a remote participant, the remote handler will make a duplicate data request to its local server and fetch the data.

The message to the other participants will often route through the network of servers participating in the Conversation; however, this is not a requirement of the protocol. Other implementations, including direct peer-to-peer or routing through a messaging substrate such as a Dis-

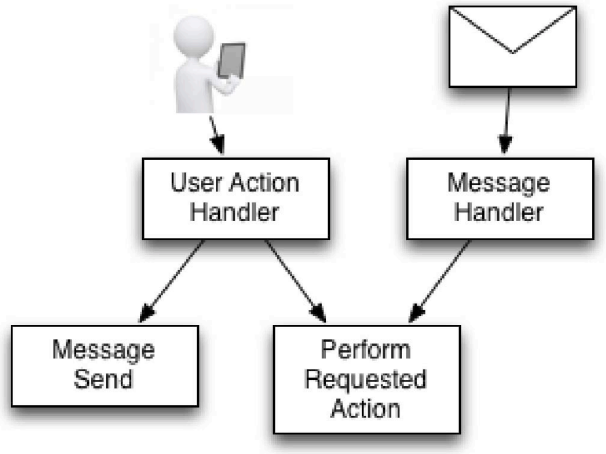

Figure 4. Typical LiveTalk Design Pattern

tributed Hash Tables[28] are possible.

The typical flow above dictates a common design pattern for LiveTalk applications, which is shown in Figure 4. A user action is serviced by an action handler method, which will typically do two things: the first is bundle up a data structure describing the user request and send it off as a message to the group, and the second is to invoke the action the user expects. Upon message receipt, the message handler interprets the message, then invokes the same action on the remote system. To most of the substrate of the application, there is no difference between a user action and an incoming message: only the message-handling/user-facing layer can see a difference.

The result is that inter-user latencies are only visible in two cases: the first is if two users attempt near-simultaneous (roughly, < 1 round trip delay time) and the second is if there is out-of-band communication which has less latency than the messaging substrate.

IV. Example LiveTalk Applications

This section will discuss two collaborative visualizations built in LiveTalk and deployed across the GENI Experiment Engine and SAVI Infrastructures.

A. Pollution Visualization

This visualization, pictured in Figure 5, is constructed of a type where there is a large data set stored locally. The specific dataset is the concentration of 2.5-micron particulate data (PM2.5) on a worldwide 10-km grid[21]. Coarser grids of 25 km, 50 km, and 100 km are obtained from the original grid and stored alongside it. The database stores a series of triples containing the latitude, longitude and scalar value. A snapshot exists for each month from 1997 to 2015, resulting in a database 11 GB in size. Each transaction requires fetching up to 30,000 points from the server, about an 800 KB transfer. The scalar values are displayed as color-coded rectangles on a map.

The data for each interaction is so large that it must be fetched from the local server. See below for an analysis. Collaborative interaction is done using the messaging

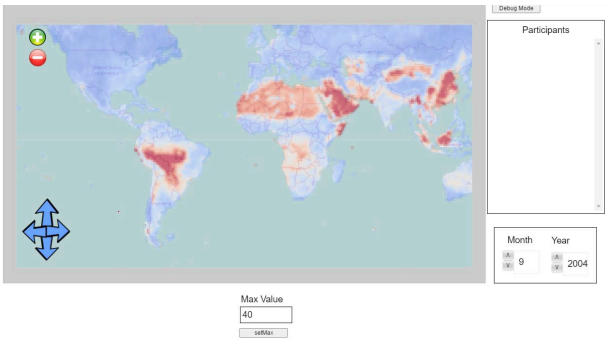

Figure 5. Pollution Visualizer

system described above. The data and message flow follow the flows in Figure 3, and the general architecture of the client system follows that shown in Figure 4. A user interaction with the system selects a map bounding box, desired resolution (10, 25, 50, or 100 km), an opacity for the rectangles, a zoom level, a month, a year, and a maximum value, which scales the color coding. This forms the basis for a bounding box request to the database and sets drawing parameters for the application.

In addition to the data request (not shown) the user interaction generates a message sent to all other Conversation participants giving the relevant data to make a local database request and draw the results on the screen. A sample message is shown in Listing 1

Listing 1. Pollution Message

```
{"action":"changeMap",
"data":
        {"user":"MattH",
        "zoom":"2.2",
        "opacity":"0.5",
        "resolution":10,
        "center":
                {"lon":388.64,
                "lat":23.76},
        "maxVal":"40",
        "month":9,
        "year":2004,
        "monthUpdate":true},
"sender":"client-session:..",
"target":"client-session:...",
"messageId":"client-msg:...",
"messageIndex":17}
```

The wrapping surrounding the action message is the Lively2Lively protocol information, containing who sent the message, and what the message index was on the server. When a user does any manipulation of the map, such as moving the map, zooming in or out, changing the maximum pollution density, changing the date being viewed or the opacity of the points being drawn, a message containing all this information is sent to all other members of the collaborative group in the `changeMap` action. The registered handler for `changeMap` is executed on each listener.

1) Analysis: This application was built to meet the metric of having an action be received, looked up and rendered in 150ms, as this has been established as the

Scenario	Latency	Estimated Bandwidth
Campus	1 ms	1 Gb/s
City	5 ms	1 Gb/s
Continent	50 ms	100 Mb/s
World	250 ms	100 Mb/s

Table I
LATENCY AND BANDWIDTH ESTIMATES

amount of time before users begin to turn away [37], [38], [41]. The Polymaps mapping library was chosen as the basic rendering application, after a series of tests of various mapping libraries demonstrated that this was the highest-performing browser-based mapping library. It was determined that Polymaps [34] was able to render 30,000 points in 100ms, giving a total of 50 ms to send the request to the server, read the data from the database, and send the results over the wire. A special-purpose QuadTree database, described below, was written to fetch the data in 20 ms. A number of network scenarios were calculated, and the results shown in Table I.

"Campus" assumes a server on the same campus where the viewer is located and gigabit bandwidth present. "City" assumes a server within approximately 100 km, but not sharing campus internet with the viewer. "Continent" assumes that there are one or two servers per continent with the viewer connecting to the one with the lower latency, similar in style to Amazon's AWS service. Finally, "World" assumes that there is a single server for global use and that everyone who sends a data request is sending it to the same one. 30,000 points was chosen as the target transaction size, since this suffices to display about a quarter-continent at 10 km resolution and the world at 100-km resolution; the application automatically chooses the finest resolution for a bounding box that will fit into 30,000 points. The wire protocol has about 27 bytes/point, so a size of 30,000 points gives an 810 KB transfer, or a 6.48 Mbit transfer. 1500-byte packets were assumed, with accelerated slowstart and an initial window size of 15 KB.

Computational analysis demonstrated the only feasible scenarios were "Campus" and"City", indicating that each user must have a server within 100 km; since a user could be anywhere, this implies that servers must be everywhere. To approximate this on a continent scale, the GENI and SAVI infrastructures were used. The combination of the GENI and SAVI infrastructures, and LiveTalk result in a destktop-application-like application on a thin client, collaboratively, anywhere.

2) Pollution Quadtree Server: A special-purpose QuadTree server was used to rapidly fetch the data points. Edge servers are resource-poor, and therefore in-memory databases are not possible for a dataset of this size. Further, and geo database creates index sets which are too large for the available disk, so a special-purpose server was written to fetch the dataset from an on-disk quad tree structure, where each leaf cell was a separate file.

To avoid fate-sharing, the quadtree server was not incorporated into the LiveTalk system but was run as a standalone Flask[11] server in a separate container in the same virtual machine as the LiveTalk server.

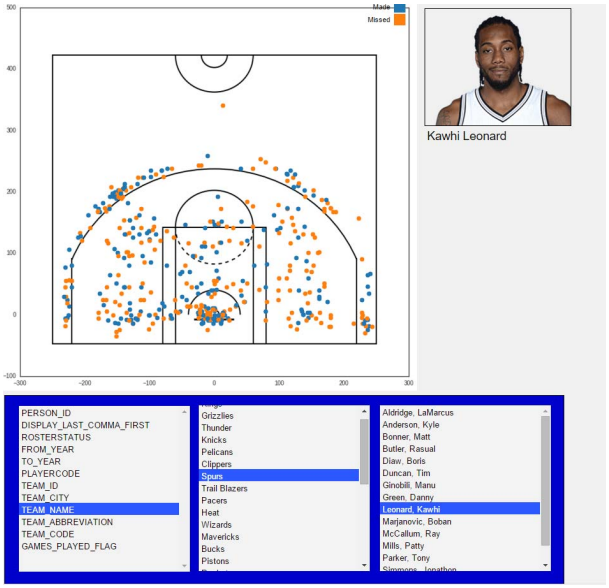

Figure 6. NBA Shot Chart

B. NBA Shot Chart Visualization

The second application constructed on this framework is an example of visualizing sports statistics. In the National Basketball Association (NBA), one measure of determining efficacy of a player is to look at their made and missed shots. With a high enough sample set, it becomes apparent that certain players have hot zones where they shoot from more effectively than others. The hot zones and shot statistics of players are of great interest to basketball aficionados among both the media and the general public.

To serve this interest, beginning in the 2013-14 season, the NBA installed SportVU [24] cameras, 6 per arena, in the catwalks. These cameras each record an image every 40 milliseconds, feed this data into specialized software and then made available to various applications both in and out of the arena. This data is available live during the game and is also available for past games.

An application to visualize this data is shown in in Figure 6. It has varying ways of narrowing down the selections available, but in the final box, a selection of a player's name will display on the chart above their made and missed shots, color-coded. This data is queried live from the NBA stats API [3], and is fetched from the Akamai content distribution network. Technically, it would be feasible to replicate the database at each visualizer site, and in future we will request permission from the NBA to replicate their data for this purpose.

The data returned from a query is a JSON object containing all the shot detail for a requested player. In Listing 2, an example of the shot detail available and what information is contained for a shot. Each shot object has a collection of properties, only three are used for the shot visualizer: x, y, and whether the shot was made or missed. The x, y co-ordinate pairs are fed to the visualization chart

along with the made or missed flag and each is rendered on the shot chart as a scatter plot.

C. Experiments and Results

In this application, the messages sent, as shown in Listing 3, are extremely small. The action of clicking a list box item sends the name of the box clicked and the index of the item to all collaborators. Each of them separately sends a query to the NBA statistics API for their own data and renders it appropriately. Since the NBA has cloud hosted their statistics using Akamai, the localization is handled for us (15 measured GENI sites gave an average speed of 0.083s for request, processing on the server side and download of the data), so for this application, there is only a thin client for distribution to the nearest site to the user, without the large set of data. The visualization in question, using a JavaScript library called Data-Driven Documents (d3) [9], is able to render 600 points in 9 ms, or 67 points/ms, so with a remaining 70ms before hitting the desired threshold, we could conceivably render 4690 shots, more than enough to handle a season for any player on record. At scale, of course, to minimize impact to the NBA statistics servers, the data set could be created and manipulated in the same was as the Pollution Visualizer in Section 5.

Listing 2. Example Shot Data

```
[
"Shot Chart Detail",
"0021500013",
21,
1495,
"Tim Duncan",
1610612759,
"San Antonio Spurs",
1,
9,
59,
"Made Shot",
"Hook Shot",
"2PT Field Goal",
"Restricted Area",
"Center(C)",
"Less Than 8 ft.",
3,
-1,
39,
1,
1
]
```

Listing 3. Shot Chart Message

```
{"action":"setList",
"data":
        {"listname":"List2",
        "idx":4},
"sender":"client-session:...",
"target":"client-session:...",
"messageId":"client-msg:...",
"messageIndex":4}
```

V. Conclusions and Future Work

The LiveTalk System is the first distributed Platform-as-a-Service Cloud, running on the GENI Experiment Engine. It can be extended wherever a Docker VM can be instantiated. It incorporates a pluggable server, integrated client-side Smalltalk-like development system, an integrated messaging framework, database, and server-side file storage. Its unique strength is collaborative visualization applications across the wide area, or, as one observer described it, "Google Docs for Visualization".

LiveTalk is promising but much remains to be done. We have deliberately excluded consistency from the current messaging framework, believing it to be an additional service that some applications will want and others not; it is not free. We intend to explore strategies from continuous interactive media[27] or distributed virtual worlds[14].

We further intend to offer an automated distribution service independent of the existing GEE Ansible-based distribution services, and integrate with a wide-area file system when one becomes available.

All of the software used in this paper is open-source, freely distributable, and stored in public repositories on github.com. The Web front ends for the various applications are available on a number of Lively Web servers; direct references can be found by contacting the authors.

Acknowledgements

We are grateful to our colleagues at the University of Victoria, US Ignite and the Hasso-Plattner Institute for support, many conversations, and much help. Dr. David Lary of the University of Texas, Dallas, provided the pollution data set and worked with us on that application. Andy Bavier, Niky Riga, Aki Nakao, Brecht Vermuelen, Max Ott, Hadi Bannazadeh, and Andi Bergen were very helpful with deployments of the early visualizer stages. This work was partially supported by the GENI Project Office and by MITACS.

References

[1] Ace: The High Performance Code Editor for the Web. https://ace.c9.io/.

[2] G. Allen and M. Owens. *The Definitive Guide to SQLite*. Apress, Berkely, CA, USA, 2nd edition, 2010.

[3] N. B. Association. Nba statistics. http://stats.nba.com, 2015.

[4] A. Bavier and R. McGeer. The geni experiment engine. In *The GENI Book*, chapter 11. Springer-Verlag, New York, 2016.

[5] A. Bavier, R. McGeer, and G. Ricart. Planetignite: A self-assembling, lightweight, infrastructure-as-a-service edge cloud. In *In Submission*, 2016.

[6] A. C. Bavier, J. H. Chen, J. Mambretti, R. McGeer, S. McGeer, J. C. Nelson, P. O'Connell, G. Ricart, S. Tredger, and Y. Coady. The GENI experiment engine. In *2014 26th International Teletraffic Congress (ITC), Karlskrona, Sweden, September 9-11, 2014*, pages 1–6, 2014.

[7] A. C. Bavier, J. H. Chen, J. Mambretti, R. McGeer, S. McGeer, J. C. Nelson, P. O'Connell, G. Ricart, S. Tredger, and Y. Coady. The GENI experiment engine. *EAI Endorsed Trans. Ubiquitous Environments*, 2(6):e2, 2015.

[8] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds - Part I.

[9] M. Bostock. Data-driven documents. http://d3js.org/, 2015.
[10] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. Updated by RFC 5689.
[11] Flask. http://flask.pocoo.org/, 2015.
[12] M. Hemmings, R. Krahn, D. Lary, R. McGeer, M. Roeder, and G. Ricart. The ignite distributed collaborative scientific visualization system. In *The GENI Book*, chapter 19. Springer-Verlag, New York, 2016.
[13] A. Holovaty and J. Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.
[14] D. Imbs and M. Raynal. Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, 2012.
[15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 318–326, New York, NY, USA, 1997. ACM.
[16] Jade template engine. http://jade-lang.com.
[17] Jinja2 (the python template engine). http://jinja.pocoo.org/.
[18] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), Oct. 2008. Updated by RFC 7504.
[19] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively wiki a development environment for creating and sharing active web content. In *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, WikiSym '09, pages 9:1–9:10, New York, NY, USA, 2009. ACM.
[20] P. Krill. Socket.io javascript framework ready for real-time apps. *InfoWorld*, June 2014.
[21] D. J. Lary, F. S. Faruque, N. Malakar, A. Moore, B. Roscoe, Z. L. Adams, and Y. Eggelston. Estimating the global abundance of ground level presence of particulate matter (pm2. 5). *Geospatial health*, 8(3):611–630, 2014.
[22] A. Leon-Garcia and H. Bannazadeh. Savi testbed for applications on software-defined infrastructure. In *The GENI Book*, chapter 22. Springer-Verlag, New York, 2016.
[23] Lively. http://www.lively-web.org/.
[24] S. LLC. Sportvu. http://www.stats.com/, 2015.
[25] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.
[26] J. H. Maloney and R. B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM.
[27] M. Mauve. Consistency in replicated continuous interactive media. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 181–190, New York, NY, USA, 2000. ACM.
[28] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.
[29] R. McGeer, M. Berman, C. Elliott, and R. Ricci, editors. *The GENI Book*. Springer-Verlag, New York, 2016.
[30] N. Middleton and R. Schneeman. *Heroku: Up and Running*. O'Reilly Media, Sebastopol, 2013.
[31] P. Mueller and S. Fischer. Europe's mission in next-generation networking with special emphasis on the german-lab project. In *The GENI Book*, chapter 21. Springer-Verlag, New York, 2016.
[32] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *In Proceedings of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI*, 2006.
[33] L. Plunkett. League of legends is building its own network to kill lag [update]. http://bit.ly/1wU6bSM.
[34] Polymaps. http://polymaps.org/, 2015.
[35] G. Rauch. *Smashing Node.js: JavaScript Everywhere*. John Wiley & Sons, 2012.
[36] D. Sanderson. *Programming Google App Engine*. O'Reilly Media, Sebastopol, 2012.
[37] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)*, 16(3):265–285, 1984.
[38] S. B. Shneiderman and C. Plaisant. *Designing the user interface 4 th edition*. Pearson Addison Wesley, 2005.
[39] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform. In *34th Euromicro Conference Software Engineering and Advanced Applications*, pages 293–302. IEEE, 2008.
[40] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Sebastopol, 2006.
[41] N. Tolia, D. G. Andersen, and M. Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer, IEEE*, 39(3):46–52, 2006.
[42] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp Symb. Comput.*, 4(3):187–205, July 1991.
[43] J. van Vliet, F. Paganelli, S. van Wel, and D. Dowd. *Elastic Beanstalk*. O'Reilly Media, Sebastopol, 2011.