Storing and Accessing Live Mashup Content in the Cloud

Krzysztof Ostrowski
Cornell University
Ithaca, NY 14853, USA
krzys@cs.cornell.edu

Ken Birman
Cornell University
Ithaca, NY 14853, USA
ken@cs.cornell.edu

ABSTRACT

Today's *Rich Internet Application* (RIA) technologies such as Ajax, Flex, or Silverlight, are designed around the client-server paradigm and cannot easily take advantage of replication, publish-subscribe, or peer-to-peer mechanisms for better scalability or responsiveness. This is particularly true of storage: content is typically persisted in data centers and consumed via web services. We propose a *checkpointed channel* (CC) abstraction as an alternative model for storing and accessing content. CCs are architecture-agnostic: they could be implemented as web services, but also as replicated state machines running over peer-to-peer multicast protocols. They can seamlessly span across the data center boundaries, or live at the edge. They are a more natural way of consuming streaming content. CCs can store hierarchical documents with hyperlinks to other CCs, thus forming a web of interconnected CCs: a live scalable information space. We discuss the advantages of the new abstraction, challenges it poses, and the way it fits within the existing models for RIA development.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction*; E.2 [**Data**]: Data Storage Representations—*Linked representations*; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Data sharing*

General Terms

Design, Languages, Standardization

Keywords

Scalability, Distributed Storage, Rich Internet Application, Cloud Computing, Edge Computing, Peer-to-Peer, Hyperlink

1. INTRODUCTION

Rich Internet Applications (RIAs) and Web 2.0 *mashups* are currently the most visible realizations of the cloud computing concept. In a nutshell, the idea is to run a sophisticated user interface (UI) – directly in the client's browser – through which users can manipulate information stored outside of their personal computers (*in the cloud*; typically on servers in data centers). The core distinguishing features include the ability to *interactively* modify live information stored in the network, *share* it with others in real time, and *combine* content from multiple sources. A number of technologies targeting this model have emerged, such as Ajax, Sun's JavaFx, Adobe Flex and AIR, Microsoft Silverlight, and most recently, Google Wave.

Depite the wide range of available RIA platforms, nearly all existing technologies follow the same pattern of working with data. A typical RIA consists of three key components. First, a rich user interface created in a markup language such as plain HTML, XAML (in Silverlight), or MXML (in Flex), and compiled into an HTML page. Second, a set of SOAP or REST *web services* (WS) at a data center that deliver content on demand (via the *request-response* pattern). Third, scripts in a language such as JavaScript (JS) or ActionScript (AS), embedded in the UI markup language, and running in the client's browser. The scripts provide a link between the UI and the WS backend. Typically, they initiate asynchronous WS calls to fetch content. The response triggers an appropriate callback in the script, and usually takes the form of an XML document. The script deserializes the XML document, and uses DOM or a similar technology to navigate to indidivial UI components (such as fields in a form or cells in a table) to populate them with the received content. The latter is done by calling UI components' setter/getter methods.

The approach just described has several disadvantages. First, it is awkward to use for streaming content or asynchronous update notification from server to the clients. Although there exist technologies that provide this functionality, we are not aware of any widely-adopted and consistent standards. Most RIAs poll for updates synchronously, which is inefficient and non-scalable. The issue stems in part from the fact that the security model in the browser does not allow listening on sockets; hence, even though SOAP WSs support asynchronous callbacks, using them can be problematic; yet to display a *live* content in real-time, updates should be pushed to clients continuously, as asynchronous streams (not unlike videos in Flash).

Second, although scripting logic could be sophisticated, usually it deals with the mundane task of moving data back and forth: calling WSs, calling UI setter and getter methods, etc. As noted above, from a logical perspective, UI components in RIAs as well as their backend WSs produce and consume streams of updates, yet they're forced into a PUT / GET interface, effectively leaving it to RIA programmers to implement the missing streaming behavior manually. Switching to streaming interfaces could reduce the coding burden.

Finally, the established approach is incompatible with replication and peer-to-peer protocols, which eliminates many types of scalable architectures that one might wish to use at the backend. Peer-to-peer connectivity could offload the server, reduce latency, or enable

LADIS '09 Big Sky, MT, USA

RIAs to work when clients are partitioned from data centers, but not from one-another (e.g. in military and search and rescue scenarios). The existing collaboration model assumes that all data is persisted at the data center and all updated are routed through a central server.

One reason why the existing RIA frameworks offer poor support for replication is that the *request-response* and PUT / GET patterns do not map in any obvious way to multicast send, receive, and state transfer operations. Another problem is that in the existing model, resources are identified by URIs that function as addresses, from which data can be retrieved. Unlike server-hosted resources, a collaborative peer-to-peer session might not have a URI in the usual sense, and the data that lives in that P2P session does not exist at any particular location; rather, it would exist as a collection of replicas distributed among a dynamically changing set of participants. Our main point is that it would be convenient to be able to treat P2P collaboration sessions as *content* in the same way as content stored in data centers, and embed them as parts of RIAs and mashups.

This motivates our approach, which is to replace the existing pattern of accessing content in RIAs and mashups with a *checkpointed channel* (CC) abstraction described in Section 2.1, and to use this abstraction uniformly at all levels from the UI to the storage backend. We propose to abandon the *client-server, request-response*, PUT / GET style of data access, and treat CCs as the basic unit of storage, and the default way of accessing it. We have implemented this approach as a part of our *live distributed objects* (LO) platform ([1]), and used it in classroom setting. We found CCs to be a natural abstraction (and free of the limitations discussed earlier).

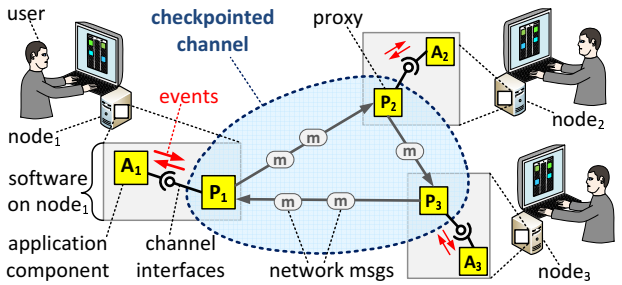

Figure 1: A *checkpointed channel* (CC) spans multiple locations across the network: it consists of a set of communicating *proxies* (here P_1, P_2, and P_3). Internal local states of CC's proxies, and protocols that run between them, are encapsulated in the CC; the latter interacts with application components via *events* exchanged through standardized instances shown on Figure 2.

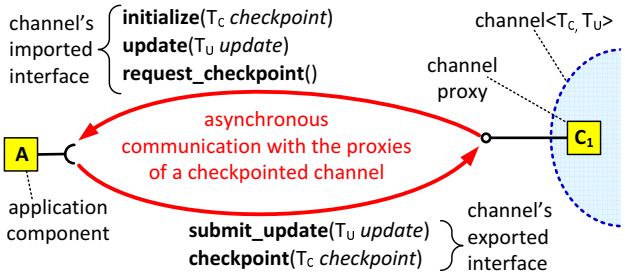

Figure 2: Interfaces exported and imported by a CC are modeled after the APIs exposed by group communication systems.

2. CHANNELS

2.1 Definition, Semantics, and Examples

A *checkpointed channel* (CC) is defined as a set of *proxies*, software components that run on multiple nodes distributed across the network (Figure 1). Each proxy can have a private local state. CC's proxies may communicate with each other, e.g., execute an instance of some distributed protocol; in this case, each proxy would be an instance of the distributed protocol stack. The proxies may also not communicate at all. Proxies interact with local application components via standardized event-based *channel interface* (Figure 2).

The interface between application components and CC proxies consists of five types of events. Event **initialize**(T_C *checkpoint*) is the first event that an application component (A) receives after connecting to a proxy of the channel. The value *checkpoint* of type T_C contained in this event represents the *state* that A should initialize itself with; this is analogous to *state transfer* in group communication. Following **initialize**, component A receives from its channel proxy a sequence of **update**(T_U *update*) events. Each **update** event carries an incremental update of type T_U that A should apply to its local replica of the application state. A can also request updates by issuing event **submit_update**(T_U *update*) to the channel proxy. Its request may not be immediately satisfied; typically, updates will be coordinated and ordered across the channel (we discuss this later). If the request is satisfied, all of the channel's proxies issue **update** events to deliver this update to their local application components. Finally, A might occasionally receive event **request_checkpoint**(); with this, A's local proxy is requesting that A provide a checkpoint of its local state. A responds with **checkpoint**(T_C *checkpoint*). The channel's proxies use this interface to obtain checkpoints necessary to initialize new application components joining the channel. Proxies may not request checkpoints if they maintain replicas of the application state internally (in general, we make no such assumption).

Formally, let T_C be the set of all possible checkpoints (application states), let T_U be the set of all possible updates in this channel, and given $u \in T_U$, let $\mathcal{F}_u : T_C \rightarrow T_C$ be a function that transforms any checkpoint (state) into one that results from applying update u. If A receives from its proxy initial checkpoint $c_0 \in T_C$ followed by a sequence of updates $u_1, \ldots u_n \in T_U$, we say that after all of these updates, A has *reached* state $c_n \triangleq \mathcal{F}_{u_n}(\mathcal{F}_{u_{n-1}}(\ldots \mathcal{F}_{u_1}(c_0)\ldots))$, and if A subsequently receives a checkpoint request, it must report c_n in its **checkpoint** event. It should be noted that when we mention A's application state, we mean a state associated with the given channel. In general, A might be connected to multiple channels; it would then exchange different parts of its state with each of them.

Note there are no explicit acknowledgements in this model. The receipt of an **update** matching an earlier request serves as a positive acknowledgement. There is no need for negative acknowledgements; we assume that channels are reliable: if an update submitted by A can't be accepted, A's local proxy disconnects itself from A. All interaction with a proxy occurs between a pair of **connect** and **disconnect** events. Each connection with a channel's proxy initiates a new interactive session, entirely independent from the past.

Although in general, the semantics of channels one might want to use in practice might vary, our discussion in this paper focuses on one particular class of channels: reliable and totally ordered. We assume that all updates delivered via the **update** event by any of the proxies come from the same totally ordered sequence – $u_1, u_2, \ldots$, and that each proxy delivers to the connected component an initial checkpoint c_n (as defined earlier) followed by a contiguous (finite or infinite) sequence of updates $u_{n+1}, u_{n+2}, u_{n+3} \ldots$, for some n (perhaps different n for different proxies). We further assume that every pair of components that never disconnect from their proxies eventually reaches the same states (in the sense defined earlier), and that all updates they submit are eventually included in $u_1, u_2, \ldots$.

Based on the discussion so far, we can of think of each CC as an entity that has state, much in the same sense as a variable in a pro-

gramming language: each application component A_i that interacts with such channel (through its local proxy C_i) observes (a part of) the same linear sequence of values. The only difference is in the interface: instead of explicit *set* and *get* (or PUT and GET) requests, characteristic of the traditional client-server approach, one now has to think in terms of asynchronous updates and checkpoints.

In the context of RIA and mashups, we are particularly interested in channels that hold structured content. We define an *XML channel* to be a CC in which the checkpoints $c \in T_C$ are well-formed XML documents, and updates $u \in T_U$ are a standardized set of edits that can be performed against such documents (the exact representation of these does not concern us here). By further restricting the valid types of checkpoints, one could distinguish XHTML, RSS, XAML, MXML, and other classes of channels that hold structured content, such as a particular class of Java/.NET objects serialized into XML. RIAs and mashups built in our prototype platform typically involve a hierarchy of XML CCs, in which individual CCs store different parts of a hierarchical document. This is discussed in Section 2.2.

To conclude this section, let's look at example architectures that fit the CC abstraction. The simplest type of a CC is one based on a back-end SOAP or REST web service, RSS feed, or other client-server protocol (Figure 3, left). Here, **submit_update** events might translate to SOAP calls or POST requests, whereas **initialize** and **update** might be triggered after synchronous GET, web sevice callbacks (if supported), or RSS notifications. One issue with this scenario is that for many content sources, such as RESTful WSs, RSS and ATOM feeds, there is no obvious way to distinguish between checkpoints and updates. One solution would be for proxies of the channel to repeatedly fetch content and issue **checkpoint** events instead of updates. Another, more expensive, would be for proxies to compare subsequent checkpoints and generate incremental updates; this may be feasible for RSS feeds and other XML sources, where updates might be as simple as inserting items to a collection. In the long run, we believe it necessary to extend the existing Web standards to support asynchronous *checkpoint* / *update* semantics.

Another example of a CC would be an instance of a reliable multicast protocol or a replicate state machine (Figure 3, right). Channel events map in a straightforward manner to multicasts and state transfers. Proxies of the CC that carries data (Q on Figure 3) might need to use an external *membership service* to discover one-another and obtain a consistent view of the membership of their group, but the actual data (checkpoints and updates) would travel directly between the clients' machines. Membership information can also be represented as a channel: checkpoints in this case map to full membership views, and updates to individual joins and leaves. The entire membership infrastructure can thus be implemented using any of the techniques discussed here (centralized, P2P, etc.). To alleviate the issue with NATs and firewalls blocking peer-to-peer traffic, the membership service could act as a STUN or rendezvous server.

The main problem with the "pure" P2P multicast scenario is that the state can be retained for only as long as there exist clients using the CC. Once the last client closes its browser window, all instances of components that held application state and their associated proxies are terminated, at which point all updates are permanently lost.

To prevent this from happening, the service provider could automatically instantiate proxies of the channel on infrastructure servers (Figure 4). The role of these servers would be to ensure that some number of state replicas always stay active. The number of servers could be adjusted based on the channel's fault-tolerance needs and the average duration of the users' interactive sessions.

One can easily imagine combinations of these schemes. For example, instead of using infrastructure servers for persistence, proxies could use reliable multicast when their number grows large, and

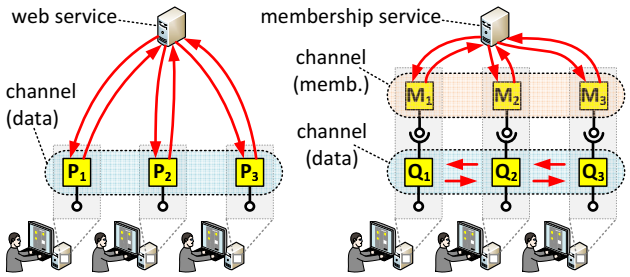

Figure 3: Left: CC based on a web service. Updates translate to SOAP or REST requests, and changes are detected by callbacks or via polling. Right: CC based on a reliable multicast protocol with state transfer. Updates/checkpoints flow directly between channel proxies. An internal *membership* channel (also a CC in our approach) is used to by Q's proxies to achieve consistency.

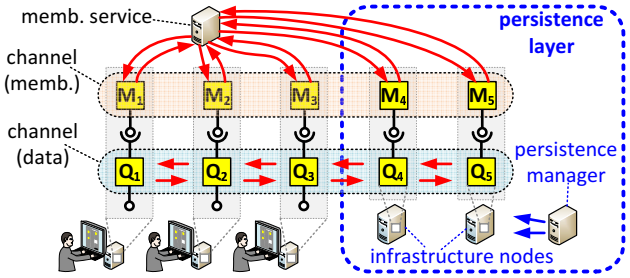

Figure 4: Persistence in peer-to-peer settings could be achieved by automatically joining infrastructure servers to the channel.

fall back to the client-server scheme and route all updates through a centralized service when their number falls below a threshold. The analysis of such scenarios is beyond the scope of this paper.

For use in our target environment, it's important that the CC abstraction be scalable. In past work [11], we proved that strong properties, such as reliable atomic delivery, can be implemented without reliance on a single global membership service, and we proposed a hierarchical architecture that instead uses a large number of (independent) membership services that control portions of the network. Hence at least in theory, it is possible to implement CCs with hundreds of thousands of members.

2.2 Embedding Channels in Applications

In Section 1, we postulated replacing PUT / GET interfaces for data access consistently throughout the RIA, including the UI layer. Accordingly, in our prototype platform all UI components, such as text boxes, panels, and 3D objects, expose interfaces complementary to those exposed by the CCs: they consume events **initialize**, **update**, and **request_checkpoint**, and unless read-only, they issue **submit_update** and **checkpoint**. Thus, in our platform, UI components bind *directly* to their channel proxies, without the need to write any scripting logic to manually move data. This is reflected in the structure of our XML markup language (an analogue to XAML and MXML). Typically, channel specification is passed directly as a parameter of a UI component (compare Figure 5, lines 6-98). When parsing our XML document, the client runtime instantiates all UI components and their channel proxies from their XML descriptions and connects their endpoints to initiate communication between them. Once a channel proxy obtains the initial checkpoint, it issues the **initialize** event, at which point the UI component connected to it is enabled, and can accept further updates or user input. If the connection between the UI an the channel proxy breaks at any

point, the UI element is disabled and the entire process is restarted.

The fact that data consumers bind directly to channels has a notable consequence: each CC stores the complete data set required to display and update the given UI component. This raises a question about *containers* (such as panels, tables, grids, lists, and compound documents) that display UI components embedded in them: should the container channel also carry data associated with elements embedded in it, or should those elements be bound to separate channels? The answer is not obvious. The established practice in RIAs is a mixture of the two: hierarchical content is first shipped to the client as a single HTML document, and then JavaScript (JS) is used to dynamically pull updates to the individual elements of the page, and update them individually via DOM. One could argue that the more dynamic and personalized the content is, the more advantageous it would be for performance reasons to store each unit of data in a separate channel, pull it on demand, and assemble on the client machine based on user's viewing preferences or local session state. Furthermore, different UI components might display data with different update patterns, security, privacy, or reliability properties. It would be desirable to back each component with the type of channel that best matches its characteristics. This is the model we used. In our platform, the container channel (P on Figure 6) stores all information necessary to create and initialize embedded components (B, C, and D) and proxies of their channels (Q_1, R_1, and S_1), but it does not include content to fill the embedded components with; each of these loads its content individually from its private channel.

Our approach could raise concerns. First, it complicates development and deployment: instead of working with a single document, the web designer now commits content into multiple independent CCs. In practice, it may be necessary for developer's changes to be atomic across channels and possible to preview, rollback and audit. Since channels might be heterogeneous, this would require extending distributed commit protocols to also work across channels.

The second concern is scalability: the model presented here leads to a large number of distinct channels, and in the sort of collaborative scenarios, where channels might be physically implemented as instances of reliable multicast, this can incur high overhead. However, note that the users who access a container channel will usually also access the channels of embedded elements, so in practice we'll observe set inclusion and other similarities between sets of users accessing different channels; one can potentially use this to amortize overhead, e.g., via some form of channeling/clustering [13].

At this point, one might pose a question: now that we eliminated the need for scripts to manually fetch data from remote sources and feed it to the UI, what *should* be the primary use of script embedded in the XML markup (if any)? Experiences with our platform suggest two important uses. First, channel management: much effort in web development focuses on delivering personalized content; in our framework, this means fetching data only from channels that match the user's physical location, viewing perspective, profile, etc. Second, distributed coordination: some applications might need to lock a portion of data before changing it, synchronize views across clients, vote, compare, or otherwise aggregate their inputs to determine the course of action. We believe that in collaborative environments, coordination logic is an integral part of the content, and naturally fits as a script embedded in XML markups such as those on Figure 5 much in the same way JS fits in HTML. We're currently working on a scripting language designed with such embedding in mind that can concisely express many types of distributed coordination using a small set of generic language primitives ([10], [11]).

To conclude, we'd like to point to a possible use of CCs as a way of storing personalized user profile and session state. Today's RIAs achieve this via *cookies*: small files stored on user's local machines and attached to client-server requests; servers use these to correlate client's HTTP requests with profiles stored in centralized databases. The problem is that cookies are associated with a particular service provider, and don't work across domains. In a mashup with a large number of components delivering content from different providers, each component tracks user profile and session state separately.

We propose to use CCs as containers for user profiles and local session state, replicated and shared among different machines, content providers, and mashup components in the same way multiple users can share a collaboratively edited document. This way, e.g., if the user moves an avatar on a Google map, the location change could be propagated across the CC carrying the user's personal profile to other components that may display weather information from Yahoo! and data from the National Census Bureau, and cause them to update their contents accordingly, so that information presented by different parts of the mashup stays synchronized even though the components displaying it may have come from different providers.

```
01: <?xml version="1.0" encoding="utf-16"?>
02: <Object xsi:type="ReferenceObject" id="e3ea16f4">
03:     <Parameter id="Background Color">
04:         <Value xsi:type="xsd:string">YellowGreen</Value>
05:     </Parameter>
06:     <Parameter id="Channel">
07:         <Value xsi:type="ReferenceObject" id="c71e88e7">
08:             <Parameter id="CheckpointClass">
09:                 <Value xsi:type="ValueClass" id="982130e4" />
10:             </Parameter>
11:             <Parameter id="MessageClass">
12:                 <Value xsi:type="ValueClass" id="982130e4" />
13:             </Parameter>
...                 ...
97:         </Value>
98:     </Parameter>
99: </Object>
```

Figure 5: An example document in our markup language (simplified). Code in lines 2-99 describes a visual component; e.g., a panel. Component type is determined by an id in line 2, and two parameters *Background Color* and *Channel* are specified in lines 3-5 and 6-98, respectively. The latter parameter describes the CC that stores information about all items contained on the panel; for example, it may refer to a built-in channel template, again by specifying the id (line 7), and parameters (lines 8-96).

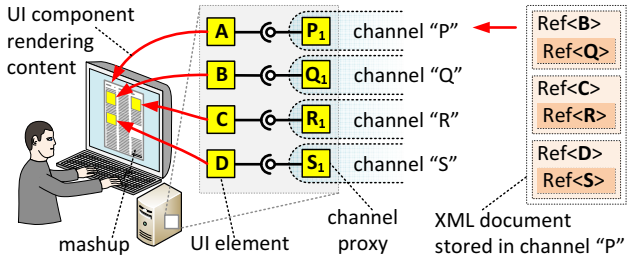

Figure 6: In a document with three embedded elements, there are four proxies that supply content (one for the document, and one for each embedded element) from four different channels. Content stored in channel P takes the form of an XML document with three sections similar to those on Figure 5, allowing the browser to create components B, C, and D with their embedded proxies. Content displayed by B, C, and D is not stored in P, however; it is stored separately in channels Q, R, and S. Sections denoted as *Ref*<*X*> are *references*: XML-serialized instructions for creating UI components, channel proxies, etc.

2.3 Addressing and Linking To Channels

A natural question at this point is: how to identify CCs, and how should a browser on a client machine translate a CC identifier into a running proxy? One approach would be to identify a CC by a URI, just like other resources on the Web, download its code via HTTP, load it into the process, and run it in the same way we run JS scripts and Java applets. The advantage of this solution is its simplicity; a major disadvantage is that it lacks flexibility. In some scenarios, we might prefer to override the name resolution process just described with a more secure version that involves mutual authentication, or construct the channel proxy's distributed protocol stack differently depending on a user's physical location or network characteristics. Another issue is that it creates a dependency on client-server infrastructure. During disconnected operation, one might wish to spontaneously initiate a peer-to-peer collaboration session without having access to the server on which the channel code is stored.

We adopted a different model: CCs themselves are mashups, described in the same XML markup language as the UI (Figure 5). CC specification expressed in XML, called a *reference*, can identify a CC with an identifier and a URL, but it can also describe the CC explicitly as a mashup of components that represent simpler protocol layers, as an instance of some template with parameters, etc. In our model, references play the role analogous to identifiers, names, addresses, or pointers: each reference contains enough information about the CC's protocol stack for the browser to construct its proxy.

Following this approach, we can represent hyperlinks as ordinary UI components with embedded CC specifications (just like those on Figure 5); except that instead of activating and connecting to its CC proxy to fetch content, a hyperlink component waits for the user's action. Once clicked, it passes the embedded channel reference to the browser. The UI component that represents the browser window then connects to the channel to retrieve content, much in the same way a regular browser would load content from the HTTP address specified in an ordinary hyperlink and fill its entire window with it.

Following this approach further, we replaced the HTTP protocol with the CC interfaces, and URIs/URLs with CC references. In our platform, instead of typing a URL or clicking on a link to download static content, the user clicks on a CC reference, causing the local browser to create the CC's proxy, connect to it, and start displaying dynamic content stored in the CC. Clicking on hyperlinks embedded on the displayed page would cause the browser to navigate to other channels, as described earlier. Instead of browsing a regular client-server Web, the user browses a Web of hyperlinked CCs.

As mentioned earlier, we have a working implementation of the CC paradigm [1], and students at Cornell have been using it for two years. However, the techniques we described could just as easily be incorporated into Silverlight, or any other modern RIA framework.

3. RELATED WORK

Due to limited space, this section is limited; more complete coverage of related work can be found in the first author's dissertation.

Croquet [12] pioneered the use of replication to store visual Web content; their system was based on two-phase commit. Similarly to most distributed storage architectures developed in the past decade, such as Bayou [4], DDS [7], and Antiquity [15], content in Croquet was stored on server replicas. In contrast, we propose to replicate content directly on the clients nodes and treat clients and infrastructure nodes symmetrically, as channel members. Most researchers believe in storage consolidation in big data centers [14]; we believe that the ever-increasing power of home users' computers, combined with data center scalability limits, will eventually revert this trend, although as several researchers have pointed out [2], there are limits to how much one can store in scenarios with high churn.

Distributed Asynchronous Collections (DAC) [5] pioneered the idea of embedding reliable multicast in a programming language as a general-purpose storage abstraction. Our work is largely inspired by DAC, but poses a different set of technical challenges due to different scenarios (RIAs vs. Java) and the type of content (structured, hyper-linked mashups vs. Java objects), among other factors.

BAST [6] pioneered the use of protocols as components within an object-oriented environment. Our platform uses a black-box approach to composition motivated by intended use in mashups, whereas BAST used a language-centric approach based on inheritance.

The emergence of massively multicore hardware and large computational clusters operating on Web-scale data sets spurred a wave interest in languages and architectures that support streaming and data flow programming [8], [9]. Our proposal to structure Web access and RIA programming around streaming APIs fits this trend.

As pointed our earlier, our approach may lead to a large number of CCs. Few existing replication protocols are designed to scale in this dimension, but several optimization techniques have been proposed recently that can amortize per-channel overhead in publish-subscribe overlays [3], [13]. Solving the problem for reliable multicast may pose a bigger challenge.

4. REFERENCES

[1] Live Distributed Objects. *http://liveobjects.cs.cornell.edu/*.

[2] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. *HOTOS*, 2003.

[3] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub/sub with many topics. problems, algorithms, and evaluation. *PODC*, 2007.

[4] K. Edwards, M. Spreitzer, D. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with bayou. *UIST*, 1997.

[5] P. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. *ECOOP*, 2000.

[6] B. Garbinato and R. Guerraoui. Flexible protocol composition in bast. *ICDCS*, 1998.

[7] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. *OSDI*, 2000.

[8] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. *ASPLOS*, 2008.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. *SIGMOD*, 2008.

[10] K. Ostrowski, K. Birman, and D. Dolev. Programming Live Distributed Objects with Distributed Data Flows. *Cornell University Tech Report*. http://hdl.handle.net/1813/12766.

[11] K. Ostrowski, K. Birman, D. Dolev, and C. Sakoda. Implementing reliable event streams in large systems via distributed data flows and recursive delegation. *DEBS*, 2009.

[12] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet: a collaboration system architecture. *C5*, 2003.

[13] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. *PDCS*, 2005.

[14] A. Veitch, E. Riedel, S. Towers, and J. Wilkes. Towards global storage management and data placement. *HotOS*, 2001.

[15] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. *EuroSys*, 2007.