Amber: Decoupling User Data from Web Applications

Tej Chajed, Jon Gjengset, Jelle van den Hooff, M. Frans Kaashoek (MIT),
James Mickens* (MSR), Robert Morris, and Nickolai Zeldovich (MIT)

ABSTRACT

User-generated content is becoming increasingly common on the Web, but current web applications isolate their users' data, enabling only restricted sharing and cross-service integration. We believe users should be able to share their data seamlessly between their applications and with other users. To that end, we propose Amber, an architecture that decouples users' data from applications, while providing applications with powerful global queries to find user data. We demonstrate how multi-user applications, such as e-mail, can use these global queries to efficiently collect and monitor relevant data created by other users. Amber puts users in control of which applications they use with their data and with whom it is shared, and enables a new class of applications by removing the artificial partitioning of users' data by application.

1 VISION

The rise of user-generated content is a striking trend on the web. Users store and manipulate e-mail, calendars, spreadsheets, and photos in the cloud. They publish videos, blogs, product reviews, and social updates; they collaborate on documents and communicate via comments in forums. These scenarios contrast with more traditional situations in which users consume content that is generated by the web sites they visit.

In principle, cloud storage should be a perfect match for user-generated data, allowing users to flexibly share their data with an open-ended set of applications and users. Once a user stores photographs in the cloud, she ought to be able to use a photo editor from one vendor, an organizer from another vendor, and a presentation manager from a third. She ought to be able to share those photos selectively with any set of users, the general public, or nobody at all. Similarly, the user should be able to control how her photos are (or are not) embedded in blog posts, social feeds, and documents.

In practice, current web sites create storage silos in which a particular data object is tightly bound to a particular site. Thus, a user must scatter her data across a large number of disjoint sites, each of which treats storage and management of the user's data as its private business. Some sites do allow users to share their site-local data with third-party applications or with other users, but these abilities tend to be limited and site-specific. Thus, traditional web services fail to provide users with the powerful, fine-grained sharing that the cloud can enable.

We envision a web where *users control their own data* — where users choose which applications they use to manipulate their data, and which users to share that data with. In this paper, we propose Amber, a new architecture for web services that provides users with such control. Our proposal is guided by the principle that user data should be cleanly separated from the web applications which process that data. We are motivated by the example of PCs and workstations: users store their data in an application-neutral file system which allows the user to control which software is applied to the data. We have applied this approach to web applications; in the rest of this paper, we describe our preliminary design.

2 APPROACH

Amber completely decouples storage from applications. Each user rents storage space from an Amber service *provider*. Providers participate in Amber's distributed protocols to present a single global namespace for all users' data objects. Web applications, running locally in users' browsers or on separate servers, access users' data by talking to the users' Amber providers. Amber's access controls give users control over who can see their data.

Because Amber organizes data by user, not by application, an application that wishes to combine data from many users needs a way to find relevant data across many providers. To address this, Amber provides a global query system; a query in Amber can in principle cover all data of all users of all providers, access controls permitting. The query language is a much-simplified form of SQL, and considers each user data object as a database row, similar to the approach taken by Dremel [16]. Amber's queries help web applications build shared views aggregated from many users' data, manage concurrent updates to shared data structures, and provide streams of relevant updates.

Amber can be deployed incrementally to make adoption practical. Initially, small-scale applications will benefit from using Amber to build applications without writing server code. As the amount of user data stored in Amber grows, more applications will be written to exploit that data, and there is a potential for network effects to drive adoption.

3 MOTIVATION

We expect Amber to enable classes of web applications that are difficult to build today. For example, a univer-

*Work performed as a Visiting Professor at MIT.

sal messaging application could consult a user's e-mail, chat conversations, and Twitter interactions to assemble a full set of contacts, and display conversations with each contact across all these messaging systems. A calendar application could provide a unified view of events pulled from co-workers' calendars, even though each user might use different calendar software. A third application could warn the user of potential calendar conflicts with events mentioned in e-mail. Today, these kinds of applications require explicit data exports or APIs if users want to use software from different vendors. In an Amber ecosystem, users can give such applications open access to the data that they need.

Hosted application ecosystems such as Google Apps and Microsoft Windows Live support sharing among their own applications and users. External applications can get at users' data through custom APIs or explicit data exports. Amber's advantage over these proprietary ecosystems is that Amber has no barriers between different vendors' applications, or between customers of different providers; it provides a single, unified interface to users' data. A user storing photos in Amber will be able to use applications from multiple vendors to manipulate the photos, as outlined above, and will be able to share the photos with other users. Since the user's data need not move to a new web site when switching applications, users are not tied to any one application. Amber provides the most value if applications adhere to informal or formal standards (*e.g.* JPEG, iCalendar, H.264); such adherence is common, particularly among new vendors wishing to join an existing ecosystem.

Amber permits a variety of business models, an important practical consideration for adoption from developers and providers. Providers can directly charge users for their services, similar to utility companies or ISPs. Advertisers and ad brokers can subsidize users' payments in exchange for access to some of their data. The success of ad-supported services such as GMail, Facebook, and Twitter suggest that many users do not mind granting access to some of their data in lieu of payment. Some applications will also form financial relationships with users in the form of subscriptions, as many current web services do.

4 CHALLENGES

Realizing Amber's vision will require overcoming many challenges:

- Queries must be inexpensive in the common case even though they notionally consult all Amber providers. This is perhaps the hardest technical challenge Amber faces.
- The authentication and access control system must be expressive (*e.g.*, support useful groups) but not too costly. This is particularly important since Amber queries must only return data to which the querying user has access. A single query may thus involve a very large number of access control decisions.
- Each user can be expected to trust their own provider, and each provider can be trusted to speak for its own users, but no other trust is likely to be warranted. This constrains how data can be allowed to move between providers, and how query execution and access checks can be divided among providers.
- Amber's mechanisms must be hardened against abusively large volumes of data, queries, or query results, either accidental or intentional.
- Amber must cope when some providers are temporarily offline.
- Amber's architecture must be compatible with sensible economic arrangements among users, Amber providers, and application developers.

5 DESIGN

5.1 Providers

Each Amber provider operates a cluster of servers offering access to Amber storage. Each user is a customer of a provider which stores that user's data, executes queries for the user's applications, and handles user authentication. The user might rent service from the provider, or an organization might operate an Amber provider for its employees. Since providers have a financial relationship with their users, the quantity of resources available in the system in principle scales with the number of users.

An Amber application can run in a user's browser (*e.g.* as JavaScript) or on the application vendor's servers. In either case, the application acquires credentials proving that it speaks for the user, and then communicates with Amber providers to fetch objects, create objects, and execute queries. The Amber APIs make provider boundaries mostly transparent to users and applications. For example, access control permitting, an Amber provider is expected to provide access to its users' stored objects to users of other Amber providers.

5.2 Objects

Storage takes the form of immutable objects, each consisting of a set of key/value pairs. For example, an object might correspond to an e-mail message, with key/value pairs indicating content, source, destination, time, etc. Applications can view objects as relational records, and retrieve them with a limited form of SQL-like queries. Objects are named with handles that contain a cryptographic hash of the object's key/value pairs along with a hint indicating which provider holds the object. Objects are immutable to simplify caching and to avoid the complexity of concurrent object writes; updates must take the form of newly created objects. Each object has a mutable access control list (ACL) indicating which users and

groups can read it. This list may only be modified by the object's owner.

5.3 Global Queries

Amber's simple object model places a large burden on applications; an application must collect relevant objects from Amber's many providers, as well as synthesize data mutability and grouping. Amber provides a query system to help with this, and introduces *standing queries* to help it optimize query execution.

A query can filter objects with simple expressions over keys and values, and can sort, group, and aggregate. Standing queries are long-lived queries that an application *installs* ahead of time to indicate data it is interest in.

Amber continuously maintains that query's output (much like a materialized view), causing it to reflect newly created objects. When a provider receives a standing query, it creates inter-provider subscriptions with each other provider. The subscriptions are themselves persistent queries, describing objects that would be relevant to the associated standing query. Whenever a provider creates a new object, it pushes information about the object to peer providers who have registered subscriptions that match this new object.

The application can then issue ad-hoc queries whenever it needs parts of the standing query's output. For example, the following standing query maintains a count of the number of votes on each article in a Reddit-like system:

```
SELECT article_id, COUNT() AS votes
  WHERE type = 'reddit-vote'
  GROUP BY article_id
```

This standing query covers all vote objects created at all providers, thus summing votes from users all over the Internet. The application can later fetch individual vote counts, or the most popular articles, by issuing queries on the output of this standing query.

By having access to standing queries ahead of time, Amber can optimize their execution (*e.g.* by sharing some or all of the work with similar queries from other users). For example, identical queries can be unified, such as if many users request Reddit vote counts. Queries that differ in small ways, *e.g.* just in the user name, can be combined into a single, more general, inter-provider subscription. The provider receiving a subscription may note that many providers have placed similar subscriptions, and unify them to reduce the matching work needed on object creation. We believe these standing queries, and the opportunities they provide for work-sharing between users and applications, are the key to achieving efficiency for Amber's global query system.

5.4 Access Control and Groups

The biggest challenges facing Amber's access control system are the ways in which access control interacts with global queries. The most immediate problem is that providers cannot generally trust each other to enforce access control. If a query from provider p_1 to p_2 matches object o at p_2, p_2 must decide whether to send p_1 a copy of o, and thus whether to trust that p_1 will enforce o's ACL. In our design, p_2 sends o to p_1 only if some entry in o's ACL mentions a user of p_1; such an entry implies that o's owner must trust p_1.

When p_2 sends object o to p_1 in response to a query, p_2 must also send access control information to p_1, so that p_1 knows which of its users should see this query result. Relaying this information allows p_1 to match o against multiple standing queries running as *different* users that may have been merged before they were sent to p_2. However, the full ACL may itself contain private information such as readers of controversial political documents. This means that p_2 must send only the subset of the ACL that includes p_1's users to p_1.

Queries place special pressure on Amber's group system: group membership resolution must be cheap enough to run for every object matching a query. In Amber, a group is an ordinary principal, and can appear on an ACL. The group's provider has a list of users who are allowed to act as the group, and will hand out a group credential to an application which can prove it speaks for a user on that list. If an application wishes to gain access to an object by virtue of a group entry in the ACL, it must issue its request with the group credentials, rather than as the user's principal. This restriction requires that applications know why they can access an object (since the relevant group credentials must be used), but allows object access checks to be simple, since they require no searches in a graph of groups.

6 Writing Amber applications

This section provides example implementations of two well-known applications, e-mail and Twitter, to give a better understanding of how applications would be written in the Amber ecosystem.

6.1 E-mail

In an Amber-based e-mail system, each e-mail would consist of a single object containing the e-mail's metadata and contents. To fetch e-mail, a client's e-mail reader would issue a standing query like this to its provider:

```
EQ1: SELECT from, subject, body
     WHERE to = 'sal@amber.mit.edu'
       AND type = 'email'
```

`sal@amber.mit.edu` is the issuing user's e-mail address; `amber.mit.edu` is that user's provider's name. This standing query causes the provider to subscribe to e-mail objects for `sal` from users on all other providers.

Whenever the e-mail reader wants to check for new e-mail, it asks its provider for new rows from the standing query's output.

The sender's e-mail application sends e-mail by creating a new object at its provider with the e-mail contents and metadata. The sending application would also give the recipient access to the e-mail. The recipient's standing query then handles delivering this object to their provider.

The standing query `EQ1` has global scope, and may match e-mail objects created at any of thousands of providers. While it might seem as though the standing queries produced by millions of users' e-mail readers would impose a heavy global load on Amber, the number of queries can in fact be kept quite small with a single optimization: each provider need only register interest once with each other provider for all objects addressed to any of its users. Sam's provider could send a single subscription for `to = *@amber.mit.edu` to all other providers, and match against each individual `EQ1` only when the subscription matches some e-mail. If there are e-mail users at every provider and there are p providers, the total number of merged subscriptions is $p(p-1) = \mathcal{O}(p^2)$. This scales much better in practice than a number of subscriptions proportional to the number of users. Having providers automatically perform this kind of optimization is one of the requirements for allowing Amber to scale to millions of users.

This example demonstrates the power of global queries, and suggests some benefits of separating e-mail data from applications. Any of a user's applications can read a user's e-mail objects if the user allows them to, so a user may opt to use different applications for search, spam detection, mailing list maintenance, etc. Similarly, new applications can combine e-mail with other user data to, for example, extract calendar and travel information. Shared management of e-mail folders among collaborating users (*e.g.* to jointly manage incoming support requests) is also natural, and ACLs can be used to control the sharing.

6.2 Twitter

We have implemented a Twitter-like application on top of Amber. Users send tweets by creating a single tweet object per tweet, and making it publicly accessible. If Sam (`sam@provider1.com`) wants to follow Alice (`alice@provider2.com`), he creates a standing query to match all of her tweets, using the special `.owner` field to refer to the owner of the tweet:

```
TQ1: SELECT content
     WHERE .owner = 'alice@provider2.com'
       AND type = 'tweet'
```

The application indicates that the query operates on public data by running it as a special "public" user. This allows Sam's provider to unify the identical standing queries of all its users who subscribe to Alice; for popular users, this results in significantly fewer subscriptions. The query has global scope, but Sam's provider knows from the `.owner` filter that it only needs to register the inter-provider subscription with Alice's provider. Alice's provider will now send just one copy of each of her tweets to any provider with a user following Alice.

This scheme requires Amber providers to maintain a set of inter-provider subscriptions. Each user at a provider requires subscriptions from the providers of all of that users' followers. For popular users, this is likely to be every other provider, while for unpopular users, it can be as many as a subscription per follower if they are all at different providers. Every new object must be matched against the list of inter-provider subscriptions: efficiently matching objects against these subscriptions is important to support applications that generate a large number of subscriptions.

The takeaway from these examples is that Amber's global queries shoulder much of the burden of the global communication in large-scale applications. Queries, if cleverly written, can be executed efficiently despite global scope using a powerful merging optimization we plan to explore further. This Twitter example is representative of a large class of many-to-many communication applications, such as mailing lists and chat groups, which can use a similar design to attain efficiency on Amber.

7 DISCUSSION

This section outlines lessons learned from designing and partially implementing an Amber prototype.

7.1 Applicability

Amber seems most useful when data is naturally associated with individual users, and when most computation (*e.g.* formatting) can be carried out in browsers. In these situations, Amber can act as a shared universal backend service, with all application-specific logic in user browsers. Amber is less natural when a service owns the data (*e.g.* an online store), or when there is data that no single user can be allowed to own (*e.g.* auctions), or when data-intensive computation is required.

7.2 Queries

Amber's global queries are a powerful tool for applications, but are also potentially a source of great cost. We expect two technical challenges in making global queries practical using standing queries: unifying subscriptions to reduce the number of issued subscriptions, and efficiently matching objects against subscriptions. The problem of merging queries for content-based subscription systems has been studied previously [5, 10, 21] and we plan to explore how some of the advanced techniques from this

work might apply to Amber. Content-based subscription systems have also explored approaches to efficiently matching a set of subscriptions [2, 13].

Some instances of query merging are easy (*e.g.* if multiple users issue identical queries), but in most cases, even a single application will issue slightly different queries for each user, such as in the e-mail query example above. Amber must spot situations where many slightly different queries can be unified, perhaps by issuing inter-provider queries that use wild-cards. More general, unified queries may again lead to extraneous objects being sent between providers, and providers will need to carefully balance the costs of query matching and object transmission.

For certain standing queries, remote providers could conceivably perform some of the work, such as report vote counts rather than individual vote objects. However, providers may not wish to perform significant computations for one another, or may not trust other providers to faithfully execute queries. Queries that wish to incrementally maintain the output of standing queries also face tough technical challenges that have been studied extensively in the context of database systems.

A standing query's output reflects both existing objects and (as time passes) newly created objects. Significant work might be required to collect all relevant existing objects when a standing query is first installed. This historical data is often not needed; for example, vote counts on ancient articles are irrelevant for generating today's front page on a Reddit-like site. The query language must allow (or even require) applications to limit how much historical data must be collected for new standing queries.

Amber must send queries and objects between providers, and there are tensions here between efficiency and privacy. Much of the reason why Amber's access control mechanisms and group features are complicated is to avoid leaking sensitive information. Applications will also need to be carefully written so that they do not leak such information. Standing query conditions and objects with poorly thought-through ACLs can easily reveal more information than the user wishes.

7.3 Abuse

We suspect Amber would be subject to at least two kinds of abuse: application-level abuse such as e-mail spam, and provider-level denial-of-service overloads such as excess query generation. Existing techniques can be used for *e.g.* e-mail spam detection. Preventing provider overload requires defenses within Amber itself; rate-limiting may provide some protection.

7.4 Provider Relationships

Amber fundamentally relies on providers to cooperate by satisfying object fetches and queries generated by each other's users. We believe this burden is reasonable: when a provider serves objects owned by its users, it can charge them, much as hosting services charge customers for outgoing traffic [3, 11, 17]. However, some work a provider performs for other providers may not be so obviously attributable to any of its users, for example queries that don't (yet) match any objects. It is an open question how the economics of these cases will work.

8 Related Designs

Amber resembles systems such as W5 [15] and BStore [8], but provides cross-provider queries that allow applications to compute across all users' data on a global scale. User data collation systems such as openPDS [18] are similar in spirit to Amber, but focus on the problem of single-user data collection rather than multi-user data sharing.

Amber's standing queries are similar to continuous queries [4] over the stream of all created objects. The underlying inter-provider subscriptions function similarly to content-based subscription systems such as Siena [7] and Thialfi [1], but benefit from the richer set of features provided by Amber's global queries.

Systems aiding in constructing and maintaining distributed web applications have previously been proposed in designs such as Sapphire [22] and Orleans [6]. These systems ease the development of traditional web applications, but do not address the tight coupling of user data to applications that Amber aims to remove.

There is much existing work on distributed databases, distributed query processing, and federation [20, 9, 14, 19]. Since Amber is based on queries, the system can draw on this existing work, particularly for the initial installation of standing queries.

Many web applications share user data by accessing general-purpose storage backends like Dropbox with standardized authentication mechanisms such as OAuth [12]. This approach requires service-dependent integration. The fact that applications tolerate the significant complexity this introduces is a sign of the need for the kind of functionality Amber provides.

9 Acknowledgements

We thank Matei Zaharia and the HotOS reviewers for their feedback on this paper. We also gratefully acknowledge the support of Quanta Computer under the T-Party project, and of the VMWare Academic Program.

10 Conclusions

We have described Amber, an architecture that provides user-centric global storage that is separate from any individual application. This arrangement allows users to flexibly share their data with an open-ended set of applications and users. To allow applications to fetch and aggregate content from users scattered across the Internet, Amber exposes flexible queries with global scope. These powerful queries are both expressive and efficient,

and we have demonstrated how they can implement existing services, as well as enable entirely new classes of applications.

REFERENCES

[1] Atul Adya et al. "Thialfi: A Client Notification Service for Internet-Scale Applications". In: *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*. 2011, pp. 129–142.

[2] Marcos K. Aguilera et al. "Matching Events in a Content-based Subscription System". In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 53–61. URL: http://doi.acm.org/10.1145/301308.301326.

[3] Amazon. *Amazon EC2 Pricing*. URL: https://aws.amazon.com/ec2/pricing (visited on 01/09/2015).

[4] Shivnath Babu and Jennifer Widom. "Continuous Queries over Data Streams". In: *SIGMOD Rec.* 30.3 (Sept. 2001), pp. 109–120.

[5] Sven Bittner and Annika Hinze. "The Arbitrary Boolean Publish/Subscribe Model: Making the Case". In: *Proc. 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 226–237.

[6] Sergey Bykov et al. "Orleans: Cloud Computing for Everyone". In: *Proc. 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 16:1–16:14.

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. "Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service". In: *Proc. 19th ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 219–227.

[8] Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich. "Separating Web Applications from User Data Storage with BStore". In: *Proc. 2010 USENIX Conference on Web Application Development (WebApps '10)*. USENIX. Boston, Massachusetts, 2010.

[9] James C. Corbett et al. "Spanner: Google's Globally-distributed Database". In: *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264.

[10] Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina. "Query Merging: Improving Query Subscription Processing in a Multicast Environment". In: *IEEE Trans. on Knowl. and Data Eng.* 15.1 (Jan. 2003), pp. 174–191.

[11] Google. *Google Compute Engine Pricing*. URL: https://cloud.google.com/compute/pricing (visited on 01/09/2015).

[12] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Fremont, CA, USA: RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6749.txt.

[13] Satyen Kale et al. "Analysis and Algorithms for Content-Based Event Matching". In: *Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05) - Volume 04*. ICDCSW '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 363–369. URL: http://dx.doi.org/10.1109/ICDCSW.2005.40.

[14] Donald Kossmann. "The State of the Art in Distributed Query Processing". In: *ACM Comput. Surv.* 32.4 (Dec. 2000), pp. 422–469.

[15] Maxwell Krohn et al. "A World Wide Web Without Walls". In: *Proc. 6th Workshop on Hot Topics in Networks (HotNets-VI)*. ACM SIGCOMM. Atlanta, Georgia, Nov. 2007.

[16] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proc. 36th Int'l Conf on Very Large Data Bases*. 2010, pp. 330–339.

[17] Microsoft. *Data Transfers Pricing Details*. URL: https://azure.microsoft.com/en-us/pricing/details/data-transfers (visited on 01/09/2015).

[18] Yves-Alexandre de Montjoye et al. "openPDS: Protecting the Privacy of Metadata through SafeAnswers". In: *PLoS ONE* 9.7 (July 2014), e98790.

[19] Amit P. Sheth and James A. Larson. "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases". In: *ACM Comput. Surv.* 22.3 (Sept. 1990), pp. 183–236.

[20] Jeff Shute et al. "F1: A Distributed SQL Database That Scales". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1068–1079.

[21] Sasu Tarkoma. "Chained Forests for Fast Subsumption Matching". In: *Proc. 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 97–102.

[22] Irene Zhang et al. "Customizable and Extensible Deployment for Mobile/Cloud Applications". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 97–112.