Legion: Enriching Internet Services with Peer-to-Peer Interactions[*]

Albert van der Linde[1] Pedro Fouto[1] João Leitão[1] Nuno Preguiça[1]
Santiago Castiñeira[2] Annette Bieniusa[2]
[1]NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa
[2]University of Kaiserslautern

ABSTRACT

Many web applications are built around direct interactions among users, from collaborative applications and social networks to multi-user games. Despite being user-centric, these applications are usually supported by services running on servers that mediate all interactions among clients. When users are in close vicinity of each other, relying on a centralized infrastructure for mediating user interactions leads to unnecessarily high latency while hampering fault-tolerance and scalability.

In this paper, we propose to extend user-centric Internet services with peer-to-peer interactions. We have designed a framework named Legion that enables client web applications to securely replicate data from servers, and synchronize these replicas directly among them. Legion allows for client-side modules, that we dub *adapters*, to leverage existing web platforms for storing data and to assist in Legion operation. Using these adapters, legacy applications accessing directly the web platforms can co-exist with new applications that use our framework, while accessing the same shared objects. Our experimental evaluation shows that, besides supporting direct client interactions, even when disconnected from the servers, Legion provides lower latency for update propagation with decreased network traffic for servers.

Keywords

Web Applications; Peer-to-Peer Systems; CRDTs; Frameworks

1. INTRODUCTION

A large number of web applications mediate interactions among users. Examples are plentiful, from collaborative applications, to social networks, and multi-user games. These applications manage a set of shared objects, the application state, and each user reads and writes on a subset of these objects. For example, in a collaborative text editor, users share the document being edited, while in a multi-user game the users access and modify a shared game state. In these cases, user experience is highly tied with how fast interactions among users occur.

These applications are typically implemented using a centralized infrastructure that maintains the shared state and mediates all interactions among users. This approach has several drawbacks. First, servers become a scalability bottleneck, as all interactions have to be managed by them. The work performed by servers has polynomial growth with the number of clients, as not only there are more clients producing contributions but also each contribution must be disseminated to a larger number of clients. Second, when servers become unavailable, clients become unable to interact, and in many cases, they cannot even access the application. Finally, the latency of interaction among nearby users is unnecessarily high since operations are always routed through servers. This might not be noticeable for applications with low interaction rates such as social networks. However, user experience in games and collaborative applications relies on interactive response times below 50ms [22].

One alternative to overcome these drawbacks is to leverage on direct interactions among clients, thus making the system less dependent on the centralized infrastructure. Besides avoiding the scalability bottleneck and availability issues of typical web application architectures, such an approach can also improve user experience by reducing the latency of interactions among clients. Additionally, it has the potential to lower the load imposed on centralized components, minimizing the infrastructure cost.

While there has been significant work in the design of peer-to-peer systems (*e.g.* [36, 28, 18, 9, 41, 27]), two main reasons prevented its adoption for improving web applications. First, web browsers restricted the ability to establish direct communication channels among clients. Recently, the Web Real Time Communication (WebRTC) initiative [5] has solved this limitation by enabling direct communication between browsers. Second, firewalls (and NAT boxes) restricted connectivity among client nodes. This problem can currently be circumvented by relying on widely available techniques, such as STUN and TURN [30]. Additionally, HTML5 makes it possible for these applications to locally store data that persists across sessions on the same browser. The combination of these techniques has created the opportunity for a new generation of web applications that can leverage peer-to-peer interactions.

In this work, we present Legion, a framework that exploits these new features for enriching web applications. Each client maintains a local data store with replicas of a subset of the shared application objects. We designed Legion to support web applications where groups of, at most, a few hundreds of users, collaborate by manipulating the same set of data objects. Legion adopts an

[*]This work was partially supported by FCT/MCTES: HYRAX project (CMUP-ERI/FIA/0048/2013); NOVA LINCS project (UID/CEC/04516/2013) and the European Union, through project Syncfree (grant agreement n°609551) and LightKone (grant agreement n°732505). Part of the computing resources used for this work were supported by an AWS in Education Research Grant.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052673

eventual consistency model where each client can modify its local replica without coordination, while updates are propagated asynchronously to other replicas. To guarantee that all replicas converge to the same state despite concurrent updates, Legion relies on Conflict-free Replicated Data Types (CRDTs) [34]. CRDTs are replicated data types designed to provide eventual convergence without resorting to strong coordination.

Unlike systems [21, 32, 42, 4] that cache objects at the client, Legion clients can synchronize with the servers and directly among each other, using a peer-to-peer interaction model. To support these interactions, (subsets of) clients form overlay networks to propagate objects and updates among them. This induces low latency for propagating updates and objects between nearby clients.

Unlike uniform overlay networks [28, 18], Legion adopts a non-uniform design where a few selected nodes act as bridges between the client network and the servers that store data persistently. These *active nodes* upload updates executed by clients in the network and download new updates executed by clients that have not joined the overlay (including both legacy clients and clients unable to establish direct connections with other clients). This design reduces the load on the centralized component, which no longer needs to broadcast every update to all clients (nor track these clients).

While leveraging direct client interactions brings significant advantages, it also creates security challenges. We address these challenges by making it impossible for an unauthorized client to access objects or interfere with operations issued by authorized clients. Our design uses lightweight cryptography and builds on the access control mechanism of the central infrastructure to securely distribute keys among clients.

Client-side modules, *adapters*, allow Legion to, instead of using its own standalone servers, leverage existing web infrastructures for storing data and assist in several functions of the framework, including peer discovery, overlay management, and security management. As a showcase, we describe our adapters for Google Drive Realtime (GDriveRT), a Google service for supporting collaborative web applications similar to Google Docs [21]. The GDriveRT adapters allow Legion to: (i) store data in GDriveRT, while exposing an API and data model compatible with GDriveRT; (ii) support interaction between Legion-enriched clients accessing local object replica and legacy clients accessing the same GDriveRT objects directly; and (iii) resort to GDriveRT to assist in establishing initial peer-to-peer connections among clients.

Our evaluation shows that porting existing GDriveRT applications requires changing only a few lines of code (2 lines in the common case), allowing them to start benefiting from direct interactions among clients. We also show that the latency to propagate updates is much lower in Legion when compared with the use of a traditional centralized infrastructure, as in GDriveRT. Additionally, clients can continue to interact when the server becomes (temporarily) unreachable. Updates are stored locally and can be made durable by any active client when the server becomes available, either in the context of the same session or a future session. Since we avoid continuous access to the centralized infrastructure by all clients, the network traffic induced on the centralized component is lower, improving the scalability of the system. We also show that our security mechanisms have minimal overhead.

In summary, we present the design of Legion, a novel framework to enrich web applications through client side replication and (transparent) direct peer-to-peer interactions. To achieve this, and besides introducing the design of the Legion architecture, we make the following contributions:

- a data storage service for web clients, providing causal consistency and using CRDTs (§ 3.2);
- a topology-aware overlay-network core that uses WebRTC and promotes low-latency links between clients (§ 3.1.2);
- a lightweight security mechanism that protects privacy and integrity of data shared among clients (§ 3.3);
- a set of client adapters that integrate Legion with GDriveRT, storing data in the GDriveRT service, providing a seamless API and support for inter-operation with legacy clients (§ 4);
- the implementation (§ 5) and evaluation (§ 6) of a prototype that demonstrates the benefits of our approach in terms of latency for clients and reduced load on servers.

2. RELATED WORK

Our work has been influenced by prior research in multiple areas.

Internet services: Internet services often run in cloud infrastructures composed by multiple data centers, and rely on a geo-replicated storage system [15, 29, 7, 12, 13] to store application data. Some of these storage systems provide variants of weak consistency, such as eventual consistency [15] and causal consistency [29, 7], where different clients can update different replicas concurrently and without coordination. Similar to Google Drive Realtime, Legion adopts an eventual consistency model where updates to each object are applied in causal order.

Other storage systems adopt stronger consistency models, such as parallel snapshot isolation [35] and linearizability [13], where concurrent (conflicting) updates are not allowed without some form of coordination. Coordination among replicas for executing each update is prohibitively expensive for high throughput and large numbers of clients (manipulating the same set of data objects).

Replication at the clients: While many web applications are stateless, fetching data from servers whenever necessary, a number of applications cache data on the client for providing fast response times and support for disconnected operation. For example, Google Docs and Google Maps can be used in offline mode; Facebook also supports offline feed access [17].

Several systems that replicate data in client machines have been proposed in the past. In the context of mobile computing [38], systems such as Coda [25] and Rover [23] support disconnected operation relying on weak consistency models. Parse [4], SwiftCloud [42] and Simba [32] are recent systems that allow applications to access and modify data during periods of disconnection. While Parse provides only an eventual consistency model, SwiftCloud additionally supports highly available transactions [8] and enforces causality. Simba allows applications to select the level of observed consistency: eventual, causal, or serializability. In contrast to these systems, our works allows clients to synchronize directly with each other, thus reducing the latency of update propagation and allowing collaboration when disconnected from servers.

Bayou [39] and Cimbiosys [33] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among clients [33] or servers [39]). Although our work shares some of the goals and design decisions with these systems, we focus on the integration with existing Internet services. This poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients, namely because most of these services can only act as storage layers (i.e., they do not support performing arbitrary computations).

Collaborative applications: Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data in client machines. Etherpad [16] allows clients to collaboratively edit documents. ShareJS [20] and Google Drive Realtime [21] are generic frameworks that manage data sharing among multiple clients. All these systems use a centralized infrastructure to mediate interactions among clients and rely on operational transformation for guaranteeing eventual convergence of replicas [31, 37]. In contrast, our work relies on CRDTs [34] for guaranteeing eventual convergence while allowing clients to synchronize directly among them. Collab [11] uses browser plugins to allow clients in the same area network to replicate objects using peer-to-peer. Our work uses standard techniques for supporting collaboration over the Internet, requiring no installation by the end user, and allowing interaction with existing Internet services.

Peer-to-Peer systems: Extensive research on decentralized unstructured overlay networks [28, 41, 18] and gossip-based multicast protocols [9, 28, 10] have been produced in the past. Although our design for supporting peer-to-peer communication among clients builds on HyParView overlay network [28], it differs from this system in the way it promotes low latency links among clients and leverages the centralized infrastructure for handling faults efficiently.

3. SYSTEM DESIGN

Legion is a framework for data sharing and communication among web clients. It allows programmers to design web applications where clients access a set of shared objects replicated at the client machines. Web clients can synchronize local replicas directly with each other. For ensuring durability of the application data as well as to assist in other relevant aspects of the systems operation (discussed further ahead), Legion resorts to a set of centralized services. We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of *adapters* with well defined interfaces.

By replicating objects in web clients and synchronizing in a peer-to-peer fashion, Legion reduces dependency and load on the centralized component (as the centralized component is no longer responsible to propagate updates to all clients), and minimizes latency to propagate updates (as they are distributed directly among clients). Furthermore, it allows clients (already running) to continue interacting when connectivity to servers is lost.

Figure 1 illustrates the client-side architecture of Legion with the main components and their dependencies/interactions:

Legion API: This layer exposes the API through which applications interact with our framework.

Communication Module: The communication module exposes two secure communication primitives: point-to-point and point-to-multipoint. Although these primitives are available to the application, we expect applications to interact using shared objects stored in the object store.

Object Store: This module maintains replicas of objects shared among clients, which are grouped in *containers* of related objects. These objects are encoded as CRDTs [34] from a pre-defined (and extensible) library including lists, maps, strings, among others. Web clients use the communication module to propagate and receive updates to keep replicas up-to-date.

Overlay Network Logic: This module establishes a logical network among clients that replicate some (shared) container. This network defines a topology that restricts interactions among clients, meaning that only overlay neighbors maintain (direct) WebRTC connections among them and exchange information directly.

Connection Manager: This module manages connections established by a client. To support direct interactions, clients maintain a set of WebRTC connections among them. (Some) Clients also maintain connections to the central component, as discussed below.

Legion uses two additional components that reside outside of the client domain:

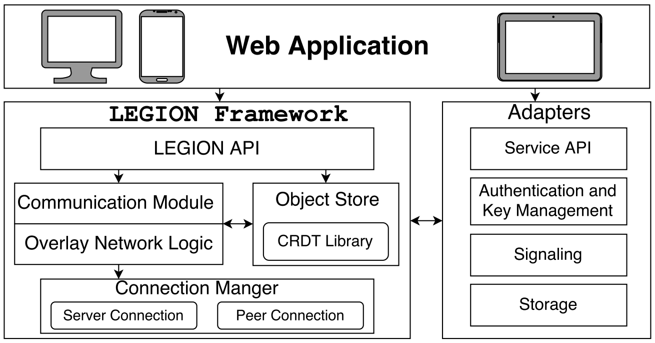


Figure 1: The Legion Architecture.

- one or more *centralized infrastructures* accessed through adapters for: (i) user authentication and key management (*authentication and key management* adapter); (ii) durability of the application state and support for interaction with legacy clients (*storage* adapter); (iii) exposing an API similar to the server API, thus simplifying porting applications to our system (*service API* adapter); (iv) assisting clients to initially join the system (*signaling* adapter);
- a set of STUN [30] servers, used to circumvent firewalls and NAT boxes when establishing connections among clients.

Our prototype includes adapters for GDriveRT and for a simple Node.js [24] server implemented by us. For STUN servers, our prototype relies on Google's public STUN servers.

In the remainder of this section we discuss in more detail the design of each of the modules that compose the Legion framework.

3.1 Communications

3.1.1 Communication Module

The communication module exposes an interface with point-to-point and multicast primitives, allowing a client to send a message to another client or to a group of clients. In Legion, each container has an associated multicast group that clients join when they start replicating an object from the container. Updates to objects in some container are propagated to all clients replicating the container.

Messages are propagated through the overlay network(s) provided by the *Overlay Network Logic* module. The multicast primitive is implemented using a push-gossip protocol (similar to the one presented in [28]).

Messages exchanged among clients are protected using a symmetric cryptographic algorithm, using a key (associated with each container) that is shared among all clients and obtained through the centralized component. Clients need to authenticate towards the centralized component to obtain this key, ensuring that only authorized (and authenticated) clients are able to observe and manipulate the objects of a container. We provide additional details about this mechanism further ahead.

3.1.2 Overlay Network Logic

Legion maintains an independent overlay for each container, defining the communication patterns among clients (*i.e.,* which clients communicate directly). The overlay is used to support the multicast group associated with that container.

Our overlay design is inspired by HyParView [28]. It has a random topology composed by symmetric links. Each client maintains a set of `K` neighbors (where `K` is a system parameter with values typically below 10 for medium scale systems as the ones we target in this paper). Overlay links only change in reaction to external events (clients joining or leaving/failing).

In contrast to HyParView, we have designed our overlay to promote low latency links. As such, each client connects to $\mathtt{K}$ peers, with $\mathtt{K} = \mathtt{K}_n + \mathtt{K}_d$, where $\mathtt{K}_n$ denotes the number of *nearby* neighbors and $\mathtt{K}_d$ denotes the number of *distant* neighbors.

As shown by previous research [26], each client must maintain a small number of distant neighbors when biasing a random overlay topology to ensure global overlay connectivity and yield better dissemination latency while retaining the robustness of gossip-based broadcast mechanisms.

This requires clients to classify potential neighbors as being either nearby or distant. A common mechanism to determine whether a potential neighbor is nearby or distant is to measure the round-trip-time (RTT) to that node [26]. However, in Legion, since clients are typically running in browsers, it is impossible to efficiently measure round trip times between them, since a full WebRTC connection would have to be setup, which has non-negligible overhead due to the associated signaling protocol.

To circumvent this issue we rely on the following strategy that avoids clients to perform active measurements of RTT to other nodes. When a client starts, it measures its RTT to a set of W well-known web servers through the use of an HTTP HEAD request (the web servers employed in this context are given as a configuration parameter of the deployment). The obtained values are then encoded in an ordered tuple which is appended to the identifier of each client. These tuples are then used as coordinates in a virtual Cartesian space of W dimensions. This enables each client to compute a distance function between itself and any other client c given only the identifier of c.

3.1.3 Connection Manager

This module manages all communication channels used by Legion, namely *server connections* to the centralized infrastructure, and *peer connections* to other clients. We now briefly discuss the management of these connections.

Server Connections: A server connection offers a way for Legion clients to interact with the centralized infrastructure. We have defined an abstract connection that must be instantiated by the adapters that provide access to the centralized services. The connection for the Legion Node.js server uses web sockets. For the GDriveRT adapter, connections are established and authenticated for each container (document in GDriveRT). Independently of the employed centralized component, server connections are only kept open by *active clients*.

Peer Connections: A peer connection implements a direct WebRTC connection between two clients [1]. To create these connections, clients have to be able to exchange – out of band – some initial information concerning the type of connection that each endpoint aims to establish and their capacity to do so, which also includes information necessary to circumvent firewalls or NAT boxes using STUN/TURN servers. This initial exchange is known, in the context of WebRTC, as *signaling*.

Legion uses the centralized infrastructure for supporting the execution of the signaling protocol between a client joining the system and its initial overlay neighbors (that have to be active clients *i.e.,* with active server connections). After a client establishes its initial peer connections, it starts to use its overlay neighbors to find new peers. In this case, the signaling protocol required to establish these new peer connections is executed through the overlay network directly. If a client gets isolated and needs to rejoin the overlay, it relies again on the help of the centralized infrastructure. This greatly simplifies fault handling at the overlay management level.

The signaling adapter for GDriveRT stores information on a hidden document associated with the (main) document. Alternatively, clients can use the Legion native Node.js signaling server.

3.2 Object Store

The object store maintains local replicas of shared objects, with related objects grouped in containers. Client applications interact by modifying these shared objects. Legion offers an API that enables an application to create and access objects.

CRDT library: Legion provides an extensible library of data types, which are internally encoded as CRDTs [34]. Objects are exposed to the application through (transparent) object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Strings, Lists, Sets, and Maps. Our library uses Δ-based CRDTs [40], which are very flexible, allowing replicas to synchronize by using deltas with the effects of one or more operations, or the full state. This new type of delta-based CRDTs [6] is specially designed to allow efficient synchronization in epidemic settings, by avoiding, most of the times, a full state synchronization when two replicas connect for the first time. Each data type includes type-specific methods for querying and modifying its internal state, and generic methods to compute and integrate deltas (*i.e.,* differences).

Causal Propagation: This module uses the multicast primitive of the *Communication module* to propagate and receive deltas that encode modifications to the state of local replicas in a way that respects causal order (of operations encoded in these deltas). To achieve this, we use the following approach.

For each container, each client maintains a list of received deltas. The order of deltas in this list respects causal order. A client propagates, to every client it connects to, the deltas in this list respecting their order. The channels established between two clients are FIFO, i.e., deltas are received in the same order they have been sent.

When a client receives a delta from some other client, two cases can occur. First, the delta has been previously received, which can be detected by the fact that the delta timestamp is already reflected in the version vector of the container. In this case, the delta is discarded. Second, the delta is received for the first time. In this case, besides integrating the delta, the delta is added to the end of the lists of deltas to be propagated to other peers.

To prove that this approach respects causal order, we need to prove that when a delta d is received in a client c_r, all deltas that precede d in the causal order have already been received. This follows from the fact that if delta d has been received from client c_s and we know that client c_s sends deltas in causal order, then c_s has already sent all deltas that precede d in the causal order. By the same reason, the lists of deltas to propagate to other nodes in c_r also respect causal order after adding d to the end of their list. The formal proof for this property can be achieved by induction.

The actual implementation of Legion only keeps a suffix of the list of deltas received. Note that, at the start of every synchronization step, clients exchange their current vector clocks, which allow them, in the general case where their suffix list of deltas is large enough to include the logical time of their peer replicas, to generate deltas for propagation that contain only operations that are not yet reflected in that peer's state.

However, when two clients connect for the first time (or re-connect after a long period of disconnection), it might be impossible (or, at least, inefficient) to compute the adequate delta to send to its peer. In this case the two clients will synchronize their replicas by us-

[1] Our experiments have shown that WebRTC connections can be established even among mobile devices using 3G/4G connectivity when devices use the same carrier.

ing the efficient initial synchronization mechanism supported by Δ-based CRDTs. In this case, if only a delta has been received, it is added to the list of deltas for propagation to other nodes. If it was necessary to synchronize using the full state, then the client needs to execute the same process to synchronize with other clients it is connected to.

3.3 Security Mechanisms

Allowing clients to replicate and synchronize among them a subset of the application state offers the possibility to improve latency and lower the load on central components. However, it also leads to concerns from the perspective of security, in particular regarding data privacy and integrity. In more detail, *privacy* might be compromised by allowing unauthorized users to circumvent the central system component to obtain copies of data objects from other clients; additionally, *integrity* can be compromised by having unauthorized users manipulate application state by propagating their operations to authorized clients.

We assume that an access control list is associated with each data container, and that clients either have full access to a container (being allowed to read and modify all data objects in the container) or no access at all. While more fine-grained access-control policies could easily be established, we find that this discussion is orthogonal to the main contributions of this paper. We also assume that the centralized infrastructure is trusted and provides an authentication mechanism that ensures that only authorized clients can observe and modify data in each container. Finally, we do not address situations where authorized clients perform malicious actions.

Considering these assumptions, Legion resorts to a simple but effective mechanism that operates as follows. The centralized infrastructure generates and maintains, for each container C, a persistent symmetric key K_C within that same container. Due to the authentication mechanism of the centralized infrastructure, only clients with access to a container C can obtain K_C. Every Legion client has to access the infrastructure upon bootstrap, which is required to exchange control information required to establish direct connections to other clients. During this process, clients also obtain the key K_C for the accessed container C.

K_C is used by all Legion clients to encrypt the contents of all messages exchanged directly among clients for container C. This ensures that only clients that have access to the corresponding container (and have authenticated themselves towards the centralized component) can observe the contents and operations issued over that container, addressing data privacy related challenges.

Whenever the access control list of a container is modified to remove some user, the associated symmetric key is invalidated, and a new key for that container is generated by the centralized infrastructure (we associate an increasing version number to each key associated with a given container).

To enable clients to detect when the key is updated in a timely fashion, the centralized component periodically generates a cryptographically signed message (using the asymmetric keys associated with the certificate of the server, used to support SSL connections) containing the current version of the key, and a nonce. This message is sent by the server to active peers, that disseminate the message (without being encrypted) throughout the overlay network.

If a client receives a message encrypted with a different key from the one it knows, either the client or its peer have an old key. When the client has an old key (with a version number smaller than the version number of the key used to encrypt the message), the client contacts the centralized infrastructure to obtain the new key. Otherwise, the issuer of the message has an old key and the client discards the received message and notifies the peer that it is using an old key. This will lead the sender of the message to connect to the central infrastructure (going again through authentication) to update the key before re-transmitting the message.

Note that clients that have lost their rights to access a container are unable to obtain the new key and hence, unable to modify the state of the application directly on the centralized component, send valid updates to their peers, or decrypt new updates.

While there might be a small increase in communication with the centralized infrastructure when a user's access is revoked (as a new key has to be generated and distributed), we believe that removing user permissions in collaborative web applications is not a frequent task. Furthermore, several access revocations can be compressed into a single update of the access control list (requiring only the generation and distribution of a single key).

4. ADAPTERS: GDriveRT

In this section we describe the adapters that can be used to integrate Legion with GDriveRT. In total (see Figure 1) we have implemented 4 distinct adapters with the following purposes: (i) a *storage* adapter enables Legion to outsource storage of application state and to (optionally) support GDriveRT legacy clients; (ii) a *signaling* adapter enables the use of GDriveRT to support signaling for establishing WebRTC connections; (iii) an *authentication and key management* adapter enables Legion to outsource to GDriveRT both user authentication and key management and distribution; and finally (iv) a *service API* adapter that exposes to client applications an interface similar to the GDriveRT API.

To simplify our prototyping, we have implemented these adapters as a single component, that enables the programmer to configure which adapters should be enabled (when an adapter is disabled, the functionality provided by it is delegated to the Legion Node.js server). In this section we discuss the most relevant aspects related with the design and implementation of these adapters which cover the specific challenges that a programmer faces when integrating Legion with an existing Web platform.

4.1 Data model

Our GDriveRT storage adapter supports the same data model as GDriveRT, in which collaboration among users is performed at the level of *documents*. A document contains a set of data objects and is mapped to a Legion container. Each object inside a document is mapped to an object of a similar type in Legion. The adapter transparently performs this mapping.

The associated service API adapter provides applications with an API similar to the GDriveRT API. The main functions of the API include a method to load a document that initializes the Legion framework. This method gives access to a handler for the document, which can be used by the application to read and modify the data objects included in the document state. As discussed, updates executed to the objects of a document's replica are delivered to other document replicas in causal order, i.e., Legion enforces causal consistency for operations over a document.

By exposing the same API of GDriveRT, this adapter enables any web application written in JavaScript that uses the GDriveRT API to be (easily) ported to Legion through the manipulation of a few lines of JavaScript code, namely: (i) adding an include statement to the script file with the code of Legion (and adapters) and (ii) replacing the function call to load a document by the equivalent function of the Legion GDriveRT service API adapter. With the handler for the loaded document, the application can use exactly the same function calls as in GDriveRT.

4.2 Legion functionality

Our Legion storage adapter can also leverage the GDriveRT infrastructure to: serve as a gateway between partitioned overlays that replicate the same GDriveRT document; and reliably store application state, i.e., documents and associated objects.

For serving as a gateway between partitioned overlays, for each document, the adapter maintains in GDriveRT the list of deltas of the document. As discussed before, in each overlay, a set of active clients is responsible to upload deltas executed by clients in the overlay and to download and disseminate new deltas throughout the overlay. If more than one client executes this process in each overlay, this does not affect correctness, as a delta received in a client is discarded if its timestamp is already reflected in the version vector of the replica. By the same reason that the list of deltas maintained by a client respects causal order, it follows that the list of deltas maintained in GDriveRT respects causal order. As a consequence, deltas downloaded from GDriveRT are also disseminated in an order that respects causality.

4.3 Support for Legacy Applications

While Legion allows web applications to explore peer-to-peer interactions using the Legion framework, it is also possible to allow legacy client applications to continue accessing data using the original GDriveRT interface by enabling a special option on our storage adapter (this support, as we show in § 6, incurs in some overhead).

When supporting legacy clients, for each data object, Legion keeps two versions: the version manipulated by Legion and the version manipulated by legacy applications. The key challenge is to keep both versions synchronized. This process is executed by a Legion client, as follows.

Applying operations executed in Legion clients to the GDriveRT object is a straightforward process that requires converting a delta stored in the list of executed deltas to the corresponding GDriveRT operations and executing them.

Applying operations executed in a GDriveRT object to the Legion object is slightly more complex because it is necessary to infer the executed operations. To this end, the client executing the synchronization process records the version number of the GDriveRT document in which the process is executed. In the next synchronization, the client infers the updates produced by legacy clients by comparing the current version of the document against the version after the last synchronization (using a diff algorithm). The updates are converted into a delta and added to the log of executed deltas, guaranteeing that the deltas are applied to Legion objects.

Both synchronization steps need to be executed by a single client to guarantee an exactly-once transfer of updates from one version to the other. We implement an election mechanism for selecting the client relying on a GDriveRT list. When no client is executing this process, a client willing to do it checks the version number of the document and the current size of the list, and then writes in the list the tuple $< id, n, t >$, with id being the client identifier, n the observed size of the list, and t the time until when the client will be executing the process (for periods in the order of seconds or minutes). The client then reads the following version of the document, which guarantees that its write has been propagated to GDriveRT servers. If the tuple the client has written is in position $n+1$, the client is elected to execute the process. This is correct, as if two clients concurrently try to be elected, the tuple of only one will be in position $n+1$ of the list in the new version of the document.

4.4 Security in GDriveRT

When using GDriveRT, we can leverage on the existing authentication mechanism of GDriveRT to perform access control. The security mechanism presented previously had to be slightly adapted as to ensure compatibility with the authentication and key management adapter due to the fact that GDriveRT only provides storage. GDriveRT exposes no computational capabilities, being therefore unable to generate symmetric keys, nor generate signed messages periodically to speed up the notification of clients of key changes.

To address these challenges we made the following modifications. First, when a new container C is created, the symmetric key K_C associated with that container is created by the first Legion client that accesses the container (after performing authentication).

Additionally, when a client removes a user's access to a container, it also generates a new key for that container. Using the GDriveRT authentication and key management adapter, active clients monitor the key, such that if the key is modified, they disseminate a notification through the overlay network, leading the remaining clients to access the centralized infrastructure to obtain it.

To deal with scenarios where users associated with all active peers have their access to a container revoked, every client periodically contacts the centralized component to verify that the known key is still valid. This step can be performed infrequently because, as soon as a single client becomes aware of a new key, the knowledge that a new key exists is epidemically propagated throughout the overlay network, as that client will start to use the new key to encrypt all messages exchanged with its neighbors, leading them to fetch the new key from the centralized infrastructure.

5. IMPLEMENTATION DETAILS

We now provide a few implementation details of our prototype. The code is available at: `github.com/albertlinde/Legion`.

Overlay networks: To achieve the threshold of `K` neighbors we do the following. Upon joining the system, a client resorts to the centralized component (either the Legion server or another web service accessed through a specialized adapter) to obtain the identifiers of nodes that currently have an open connection to the centralized infrastructure. Using this information, the client establishes a connection to a nearby neighbor and another to a distant neighbor (for these connections the centralized infrastructure is leveraged to perform the WebRTC signaling protocol). Random walks over the existing overlay are then used by the new client to find other nearby and distant neighbors to fill its neighborhood.

To classify peers as being either nearby or distant we resort to the previously described scheme, where we use 4 distinct sites, which are endpoints of Amazon EC2 Web API (scattered throughout 4 different data centers). While different distance functions can be employed over the virtual coordinates associated with each client, in our prototype we use a function that categorizes a client to be *distant* if the difference between at least two coordinates in the 4D virtual space are equal or above 70, and *nearby* otherwise (we have experimentally asserted that this strategy yields adequate results).

Selection of Active Clients: In our design, we use a small subset of clients (that we dub *active clients*) to upload and download updates over objects to and from the centralized infrastructure and to monitor the cryptographic key associated with each container. To select these clients we use a bully algorithm [19] where initially all clients act as an active client, and periodically, every $\mathtt{T}ms$, sends to its nearby overlay neighbors a message containing its unique identifier – in our experiments we set $\mathtt{T} = 7000\ ms$. Whenever a client receives a notification from a neighbor whose identifier is lower than its own, it switches its own state to become a *passive client*, and stops disseminating periodic announcements. To address the departure or failure of active clients, if a *passive client* does not receive an announcement for more than $3 \times \mathtt{T}\ ms$, it switches its own state back to become an *active client* (the factor of 3 is used to avoid triggering this process unnecessarily).

Passive clients disable their connection to the centralized infrastructure. The result of executing this algorithm is that only a small subset of (non-neighboring) clients remain *active clients*.

Security Mechanisms: For the symmetric cryptography algorithm, we used AES operating in block cipher mode, using a key of 128 bits. We use RSA, configured with a key of 2048 bits, for generating and verifying the signature of the messages issued by our Node.js server. Our implementation resorts to the Forge [1] JavaScript library to implement all cryptographic operations.

STUN Service: In our prototype we have resorted to publicly available Google STUN servers. However, this is a configurable aspect in our prototype, and can easily be modified to use privately owned and managed servers if an application operator desires.

Implementation complexity: We have used `Count Lines of Code` [14] and verified that the code for our GDriveRT adapters has 1.768 JavaScript LOC, while the whole implementation of Legion (including the simple server for materializing the centralized component) has 4.639 JavaScript LOC.

6. EVALUATION

This section presents an evaluation of Legion with an emphasis on the operation of Legion when using the adapters to inter-operate with the GDriveRT infrastructure (except if specifically stated in our experiments, we ran Legion with all GDriveRT adapters enabled and with support for legacy clients disabled). Our evaluation mainly focus on two complementary aspects. We start with an analysis of our experience in adapting existing GDriveRT applications to leverage Legion. Then, we present an experimental evaluation of our prototype, comparing it to the centralized infrastructure of GDriveRT regarding the following practical aspects: (i) What is the impact on update propagation latency? (ii) What is the impact on application performance? (iii) How does the system behave when the central server becomes (temporarily) unavailable? (iv) What is the impact of using Legion in terms of load imposed on the central component and on individual clients? (v) What is the overhead for supporting seamless integration with legacy clients?

6.1 Designing Applications

In this section, we describe a set of web applications that we have ported to Legion using the GDriveRT adapters.

Google Drive Realtime Playground: The Google Drive Realtime Playground [2] is a web application showcasing all data-types supported by GDriveRT. We ported this application to Legion by changing only 2 lines in the source code (see § 4).

Multi-user Pacman: We adapted a JavaScript version of the popular arcade game Pacman [3] to operate under the GDriveRT API with a multi-player mode. We also added support for multiple passive observers that can watch a game in real time. In our adaptation up to 5 players can play at the same time, one player controlling Pacman (the hero) and the remaining controlling each of the four Ghosts (enemies).

The Pacman client is responsible for computing, and updating the adequate data structures, with the *official* position of each entity. Clients that control Ghosts only manipulate the information regarding the direction in which they are moving. If no player controls a Ghost, its direction is determined by the the original game's AI, running in the client controlling Pacman.

In this game, we employed the following data types provided by the GDriveRT API: (i) a map with 5 entries, one for Pacman and the remaining for each Ghost, where each entry contains the identifier (ID) of the player controlling the character (each user generates its own random ID); (ii) a list of events, that is used as a log for relevant game events, which include players joining/leaving the game, a Ghost being eaten, Pacman being captured, etc. (iii) a list representing the game map, used to maintain a synchronized view of the map between all players. This list is modified, for instance, whenever a *pill* is eaten by Pacman; (iv) a map with 2 entries, one representing the width and the other the height of the map. This information is used to interpret the list that is used to encode the map; (v) a map with 2 entries, one used to represent the state of the game (paused, playing, finished) and the other used to store the previous state (used to find out which state to restore to when taking the game out of pause); finally, (vi) 5 maps, one for each playable character, with the information about each of these entities, for maintaining a synchronized view of their positions (this is only altered when the corresponding entity changes direction, not at every step), directions, and if a ghost is in a vulnerable state.

Along with extending and porting this application to use the GDriveRT API, we also implemented the same game (with all functionality) using Node.js as a centralized server for the game through which the clients connect using web-sockets (this implementation does not leverage Legion). This enables us to investigate the effort in implementing such an interactive application using both alternatives. The Node.js implementation of the game is approximately 2.200 LOC for the client code, and 100 LOC for the server. In contrast, the implementation leveraging the GDriveRT API has approximately 1.620 LOC for the client code, and 40 lines of code for the server side (used to run multiple games in parallel). This shows that an API such as the one provided by GDriveRT and Legion simplifies the task of designing such interactive web applications.

Creating the Legion version (using the GDriveRT adapters) required to change only two lines of code of the GDriveRT version (as described before). From a user perspective, the Legion version runs much smoother, which is also shown by our evaluation presented further ahead.

Spreadsheet: We have also explored an additional application: a collaborative spreadsheet editor. Each spreadsheet represents a grid of uniquely identifiable rows and columns, whose intersection is represented by an editable cell. Each cell can hold numbers, text, or formulas that can be edited by different users.

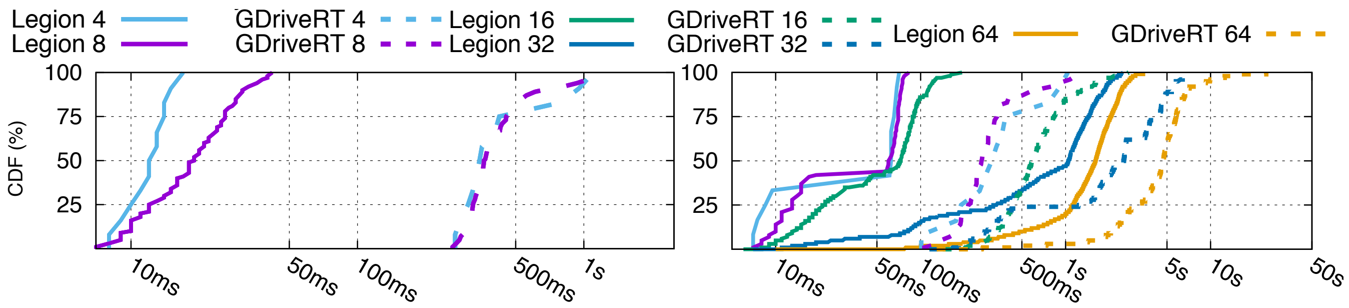
A prototype of the spreadsheet web application was built using AngularJS and supporting online collaboration through GDriveRT. The spreadsheet cells were modeled using a GDriveRT map. Each cell was stored in the map using its unique identifier (row-column) as key. Porting this application to the Legion API only required the change of 2 lines of code (as discussed previously).

Discussion: Our experience with porting these applications to leverage Legion shows that doing so is simple, as the programmer can easily use our GDriveRT adapters. Furthermore, this shows that carefully designing our framework to expose (through adapters) APIs that are similar to existing Web infrastructures is paramount to promote easy adoption of our solutions.

6.2 Experimental evaluation

In our experimental evaluation, we compare Legion, with and without the use of adapters, against GDriveRT, as a representative system that uses a traditional centralized infrastructure.

In our experiments, we have deployed clients in two Amazon EC2 datacenters, located at North Virginia (us-east-1) and Oregon (us-west-2). In each DC, we run clients in 8 m3.xlarge virtual machines with 4 vCPUs of computational power and 15GB of RAM. Unless stated otherwise, clients are equally distributed over both DCs. The average round-trip time measured between two machines in the same DC is 0.3 ms and 83 ms across DCs.



(a) All clients within the same datacenter

(b) Clients distributed over 2 Datacenters

Figure 2: Latency for the propagation of updates.

Latency: To measure the latency experienced by clients for observing updates, we conduct the following experiment. Each client inserts in a shared map a key-value pair consisting of his identifier and a timestamp. When a client observes an update on this map, it adds to a second map, as a reply, another pair concatenating the originating identifier and the replier's identifier as the key, and as value an additional timestamp. When a client observes a reply to his message, it computes the round-trip time for that reply, with latency being estimated as half of that time. All clients start by writing to the first map at approximately the same time and reply to all identifiers added by other clients. Thus, this simulates a system where the load grows quadratically with the number of clients.

Figure 2 presents the latency observed by all clients for both Legion and GDriveRT. The results show that latency using Legion is much lower than using GDriveRT for any number of clients. The main reason for this is that the propagation of updates does not have to incur a round-trip to the central infrastructure in Legion. Furthermore, for 64 clients, the 95^{th} percentile for GDriveRT is almost an order of magnitude above Legion, suggesting that Legion's peer-to-peer architecture is better suited to handle higher loads than the centralized architecture of GDriveRT.

Multi-Player Pacman Performance: We now show the impact of Legion on the performance of applications in the context of our Multi-player Pacman game.

To that end we conducted an experiment with volunteers, where we had five users playing Pacman (one player controlling Pacman, and four players for each of the ghosts). This experiment was conducted using five machines, in a local area network. Machines were running Ubuntu and clients executed in Firefox.

We focus our experiments in measuring the displacement of entities in relation to their official position. As explained before, each client updates an object with the direction of its movement. The Pacman client computes and updates the official position of each entity periodically. Each client independently updates its interface based on the known direction of movement and the latest official positions. Displacement captures the difference between the position computed by a client and the received official position upon receiving an update. When displacement has large values, users see entities jumping on the game map. In particular we measure: (i) the displacement of Pacman in relation to an eaten pill when an update reporting the pill being eaten is received by a client controlling a Ghost; and (ii) the displacement of Pacman and Ghosts when a client controlling a ghost receives an update for a position. Figure 3 reports the obtained results where the displacement is measured in tiles (the square unit that forms the interface). The board size of Pacman was 19×22 tiles featuring approximately, 59 turning points. Pacman and Ghosts move at approximately 3.33 tiles per second.

Figure 3(a) shows that when using Legion the interface is much more synchronized in relation to the real state of the system, showing that Pacman is visible by other players much closer to the eaten pill than when using the GDriveRT version of the game. Figure 3(b) reinforces these results showing that when using Legion the displacement of entities in the game interface is significantly lower when compared with the game version that only uses GDriveRT, which is unable to send updates to all clients at an adequate rate.

Effect of disconnection: We study the effect of disconnection by measuring the fraction of updates received by a client. In the presented results, clients share a map object, and each client executes one update per second to the map (similar behavior was observed with other supported objects). We simulate a disconnection from the Google servers, by blocking all traffic to the Google domain using *iptables*, 80 seconds after the experiment starts. The disconnection lasts for 100 seconds, after which rules in iptables are removed so that connections can again be re-established.

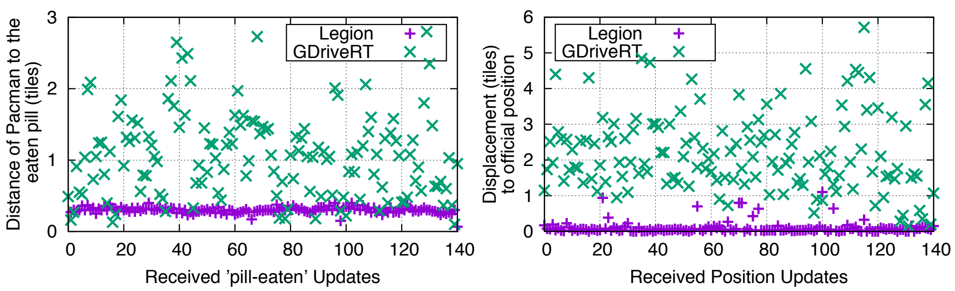
Figure 4 shows, at each moment, the average fraction of updates observed by clients since the start of the experiment (computed by dividing the average number of updates received by the total number of updates executed). As expected, the results show that during the disconnection period, GDriveRT clients no longer receive new updates, as the fraction of updates received decreases over time. When connectivity is re-established, GDriveRT is able to recover. With Legion, as updates are propagated in a peer-to-peer fashion, the fraction of updates received is always close to 100%.

We note however, that while servers remain inaccessible, new clients cannot join the system. However, when leveraging Legion, clients that are active when the server becomes unavailable can continue operating regularly without noticing the server unavailability.

Network load: We now study the network load induced by our approach. To this end, we run experiments where 16 clients share a map object. Each client executes one update per second. The workload is as follows: 20% of updates insert a new key-value pair and 80% replace the value of an existing key selected randomly. The keys and values are strings of respectively 8 and 16 characters. We measure the network traffic by using *iptraf*, an IP network monitor.

In these experiments, we used the following configurations: *Legion w/ Node.js*: that uses our Legion server as backend. *Legion w/ GDriveRT*: that uses GDriveRT documents as backend. *GDriveRT*: that uses the original GDriveRT document as backend.

Figure 5(a) shows the total network load of the setup process, which entails making the necessary connections to the infrastructure and peer-to-peer connections. The incurred load using our own backend server is due to clients requiring to use this component to



(a) Pacman displacement to eaten pills (b) Pacman and Ghosts Displacement

Figure 3: Muti-User Pacman Performance assessment.

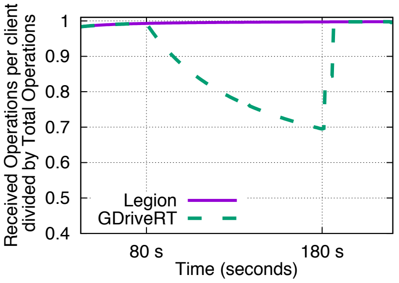
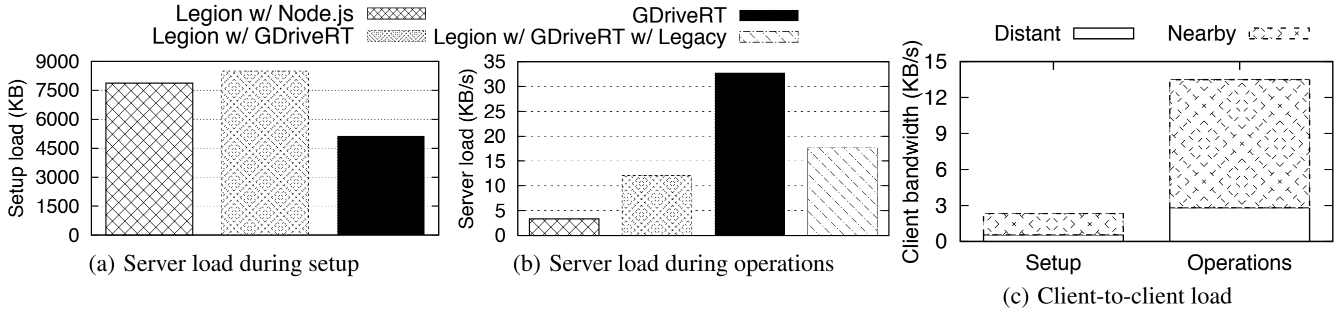


Figure 4: Effect of disconnection



(a) Server load during setup (b) Server load during operations

(c) Client-to-client load

Figure 5: Network load.

connect to each other initially (WebRTC signaling). Legion using GDriveRT as backend has a slightly higher cost due to the overhead of performing signaling through the infrastructure, which is less efficient. In both cases only few clients obtain the initial object and propagate to other clients. Finally, in GDriveRT all clients download the shared data from the infrastructure.

Figure 5(b) shows the network load of the server without considering the initial setup load (computed by adding the traffic of all clients to and from the centralized infrastructure) for all competing alternatives. Results show that the load imposed over the centralized component is much lower when using Legion with GDriveRT as backend than when using only GDriveRT. This is expected, as only a few clients (active clients) interact with the GDriveRT infrastructure, being most interactions propagated directly between clients. Interestingly, the use of our server leads to an even lower load on the centralized component, this happens not only because the signaling mechanism used to establish new WebRTC connections among clients and the process for replica synchronization with the server is more efficient, but also because the data representation used by our backend is significantly more compressed. We run an additional configuration, (*Legion with GDriveRT w/ Legacy*) that uses GDriveRT documents as backend and synchronizes with the original document every 5 seconds. Supporting legacy clients (i.e., synchronizing with the original document) incurs a non negligible overhead. This happens because the mechanism used requires a large number of accesses to the centralized infrastructure as to infer which operations should be carried from legacy clients to the Legion clients and vice versa. However, even with support for legacy clients enabled, Legion induces lower load on the centralized component when compared with GDriveRT.

Figure 5(c) reports the average peer-to-peer communication traffic for each client during the setup of WebRTC connections (*Setup*) and while clients issue and propagate operations (*Operations*). The results show that the traffic of each client is larger than the traffic of each client with the server in GDriveRT (which can be approximated by dividing the server load – in Figure 5(b) – by the number of clients). This happens because our dissemination strategy has inherent redundancy, whereas in GDriveRT there are no redundant transmissions between each client and the centralized infrastructure. However, an average under 14KBps does not represent a huge fraction of available bandwidth nowadays. Furthermore the use of our location aware overlay leads to a network usage pattern where the amount of data sent to distant nodes is significantly lower that that sent to nearby nodes.

7. FINAL REMARKS

In this paper we presented the design of Legion, a framework that allows the development of web applications with seamless support for replication at the client machine leveraging peer-to-peer interactions to propagate operations among clients. Furthermore, we presented the design of adapters that enable Legion to leverage GDriveRT for (potentially) multiple purposes namely, storage backend; WebRTC signaling; authentication and key management; exposing an API akin to that of GDriveRT; and finally, an optional mechanism to support the co-existence of legacy clients.

The evaluation of our prototype shows that latency for update propagation is much lower using Legion when compared with the use of GDriveRT. Furthermore, load to the centralized infrastructure is greatly reduced by leveraging peer-to-peer interactions. Finally, we show that clients are able to interact while servers are temporarily unavailable. As future work, we plan to study how to support applications with thousands of simultaneous users, and to design and implement adapters to integrate Legion with storage services such as Cassandra, Redis, and Antidote.

8. REFERENCES

[1] Forge. github.com/digitalbazaar/forge.

[2] Google Drive Realtime Playground. github.com/googledrive/realtime-playground.

[3] Original implementation of pacman for browsers. github.com/daleharvey/pacman.

[4] Parse. parse.com.

[5] WebRTC. www.webrtc.org/.

[6] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of NETYS'15*, Morocco, 2015.

[7] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of EuroSys'13*, 2013.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proc. of VLDB Endow.*, 7(3), Nov. 2013.

[9] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2), 1999.

[10] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proc. of DSN'07*, UK, June 2007.

[11] S. Castiñeira and A. Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proc. of PaPoC '15 Workshop*, 2015.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB Endow.*, 1(2), 2008.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3), 2013.

[14] A. Danial. Cloc–count lines of code. *Open source*, 2009.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[16] EtherpadFoundation. Etherpad. etherpad.org.

[17] Facebook Inc. Continuing to build news feed for all types of connections. goo.gl/Q06CaL, Dec. 2015.

[18] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Net. Group Comm.* 2001.

[19] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Tran. on Comp.*, C-31(1), Jan 1982.

[20] J. Gentle. ShareJS API. github.com/share/ShareJS.

[21] Google Inc. Google Drive Realtime API. developers.google.com/google-apps/realtime/overview.

[22] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), Aug. 2007.

[23] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. SOSP'95*, 1995.

[24] Joyent Inc. Node. js, 2014.

[25] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TOCS*, 10(1), Feb. 1992.

[26] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE TPDS*, 23(11), Nov 2012.

[27] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Proc. of SRDS'07*. IEEE, 2007.

[28] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07*. IEEE, 2007.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. of SOSP'11*, 2011.

[30] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), RFC 5766. Technical report, IETF, Apr. 2010.

[31] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. UIST'95*, 1995.

[32] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *Proc. of EuroSys '15*, 2015.

[33] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI'09*, 2009.

[34] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proc. of 13th SSS'11*, 2011.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP'11*, 2011.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 2001.

[37] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of Comp. supported cooperative work*, 1998.

[38] D. B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP'95*, 1995.

[40] A. van der Linde, J. Leitão, and N. Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proc. of the PaPoC'16 Workshop*, United Kingdom, 2016. ACM.

[41] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. of Net. & Sys. Manag.*, 13(2), 2005.

[42] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Proc. of Middleware'15*. ACM/IFIP/Usenix, Dec. 2015.