LEGION: Enriching Internet Services with Peer-to-Peer Interactions

Albert Linde, Pedro Fouto, João Leitão, Nuno Preguiça
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa
Santiago Castiñeira, Annette Bieniusa
University of Kaiserslautern

April 11, 2016

Abstract

A large number of web applications are built around direct interactions among users, from collaborative applications and social networks to multi-user games. While these applications are user-centric, they are usually supported by services running on servers that mediate all interactions among clients. Often, users run these applications while being located in close vicinity of each other. Relying on a centralized infrastructure for mediating these 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 that allows client web applications to replicate data from servers, and synchronize these replicas directly among them. Our framework allows the use of extensions to leverage existing web platforms. We have implemented one such extension that interacts and exports the same API as Google Drive Realtime (GDriveRT), also allowing the co-existence of legacy clients accessing GDriveRT directly with enriched clients using our new framework accessing the same shared objects. The results of our experimental evaluation show that, besides supporting client interaction, even when disconnected from the servers, our framework provides much lower latency for update propagation while also decreasing the network traffic load on servers.

1 Introduction

A large number of Web applications are built around direct interactions among users, from collaborative applications and social networks to multi-user games. These applications manage a set of shared objects, with each user reading and writing to 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, which usually consists of multiple objects.

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, the servers become a scalability bottleneck, as all interactions have to be mediated by servers. While such bottlenecks can be partially mitigated through the use of elastic resources available in the cloud, this might be too expensive for small startups and companies launching their web applications. Second, when the servers become unavailable, clients cannot interact anymore, 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 when two clients interact. While this might not be noticeable for applications with low interaction rates, such as social networks, games and collaborative applications rely on interactive response times below 50ms for satisfactory user experience [18].

One alternative that would allow to overcome the drawbacks discussed above is to rely on direct interactions among clients, 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 lowering the latency of interactions among clients. Additionally, it has the potential to reduce the load imposed on the centralized components, which can bring significant benefits when dealing with flash crowds [26] while minimizing the infrastructure cost for web applications operators.

While there has been significant work in the past on peer-to-peer systems, exploring multiple aspects of direct client-to-client communication and interaction models (*e.g.* [22, 14, 31, 5, 35, 21]), this body of work has not being leveraged to improve the operation of web applications, as these usually run on web browsers which restrict the ability to establish direct communication channels among clients. To circumvent this problem, users have to install plugins, an obstacle that makes it difficult to deploy such architectures in practice. However, recently the Web Real Time Communication (WebRTC) initiative [2] has developed simple APIs that enable direct and real-time communication across browsers through a simple JavaScript interface, paving the way for a new generation of web applications that leverage peer-to-peer interactions.

Despite the inherent drawbacks, the use of a centralized infrastructure simplifies the task of circumventing connectivity issues posed by firewalls (and NAT boxes). Currently, the problem of connectivity in such cases can be addressed relying on widely available techniques, such as STUN and TURN [24]. When combined with WebRTC, these create a new ecosystem that empowers web applications running in one browser (or native applications using WebView components) to communicate directly with other application instances. Additionally, HTML5 makes it possible to locally store data that survives between sessions on browsers.

In this work, we present LEGION, a framework that exploit these new features for enriching web applications. In LEGION, each client maintains a local data storage with replicas of a subset of the application (shared) objects. Applications in LEGION experience low latency when accessing objects by instantiating new (local) replicas of these data objects from existing

copies located in other clients. Furthermore, applications can continue accessing these local objects even when servers are not reachable.

LEGION adopts an eventual consistency model, where individual application instances can modify their local replicas without coordination. This allows updates to be performed concurrently on different replicas while modifications are propagated asynchronously. To guarantee that all replicas converge to the same state after all updates have been applied, LEGION relies on CRDTs [29], replicated data types designed to provide eventual convergence of replicas.

Unlike other systems [17, 27, 36, 1] that support objects cached at client side that are synchronized with a server, LEGION clients can also synchronize directly among them, 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 leads to low latency for propagating updates between nearby clients.

In each overlay network, a small number of clients are responsible for synchronizing with the servers (that form a centralized component), uploading updates executed by clients in the network and downloading new updates executed by clients that have not joined the overlay network. Unlike many overlay networks where all clients operate in a similar fashion [22, 14], our system relies on a non-uniform design, where a few nodes are elected to act as bridges between the decentralized infrastructure established between clients and the servers that store data persistently (and serve as access points to legacy clients or clients that are unable to establish direct connections with other clients). This approach has the advantage of reducing the load on the centralized component, which no longer needs to broadcast each update a shared object directly to all active clients (nor track these clients).

LEGION includes support for user defined extensions, to enable the framework to interact and leverage existing centralized web infrastructures. We implemented an extension for API of Google Drive Realtime (GDriveRT) a Google service that is used to support web applications similar to Google Docs. Our extension allows LEGION to not only use GDriveRT infrastructure but also exposes an API fully compatible with the GDriveRT. We support the same data types as GDriveRT, through a similar interface which allows to easily port existing applications to leverage LEGION. Further, this layer provides additional integration of our framework with GDriveRT by storing data as GDriveRT objects over GDriveRT centralized storage layer. In addition, we allow legacy clients (that run an outdated version of an application or whose environment disallows direct connections between clients to be established) to access GDriveRT objects and to interoperate with LEGION-enriched clients that access replicas of the same objects through our framework.

Our evaluation shows that porting existing GDriveRT applications can be achieved by changing only a few lines of code (2 lines in the common case). We also show that, in LEGION, the latency to propagate updates is much lower than when relying on a traditional centralized infrastructure, as in GDriveRT. Since we avoid continuous access to the centralized infrastructure by all clients, we can support a larger number of clients without degrading the latency of the service. Furthermore, clients can continue to interact among them when the server is unreachable. Finally, the network traffic induced on the centralized component is lower when leveraging our

framework, lowering the operational cost of the centralized component.

In summary, we present the design of LEGION, a framework to enrich web applications with local storage and direct peer-to-peer interactions in a transparent way. To achieve this, we make the following contributions:

• a data storage for web client devices, providing CRDTs that can be accessed and modified without coordination with other clients or servers. The use of CRDTs allows to guarantee eventual convergence after applying all updates;

• an overlay network substrate that use WebRTC connections to propagate data directly among clients;

• an extension that integrates LEGION with GDriveRT, by providing a seamless API, storing data in the GDriveRT service, and allowing legacy clients and enriched clients to interoperate while operating over the same data;

• the implementation of a prototype that was experimentally evaluated showing the benefits of the proposed approach in terms of both latency for clients and reduced load on servers.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the design of the system. Section 4 details how to support legacy clients. Section 5 discusses implementation details. Section 6 presents an evaluation of the system, and Section 7 concludes the paper with some final remarks.

2 Related Work

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

Internet services: Internet services are supported by servers typically running in a data center, often as part of some cloud infrastructure. Applications running in user devices access these servers to read and modify the service state. These applications often run in the web browser or as a standalone (mobile) application.

For applications with a large number of users, it is common to offer the service through servers running on multiple data centers, by relying on a geo-replicated data storage subsystem [12, 23, 4, 8, 9]. Some of these storage systems provide variants of weak consistency, such as eventual consistency [12] and causal consistency [23, 4], where different replicas can be concurrently updated by different clients. This approach allows updates to execute without the need for coordination among replicas. As Google Drive Realtime, LEGION adopts an eventual consistency model where updates of each object individually are applied in causal order.

Other storage systems adopt stronger consistency models, such as parallel snapshot isolation [30] and linearizability [9], where concurrent updates are not allowed. These approaches simplify the development of applications, as replicas do not diverge, at the price of requiring coordination among replicas for executing each update. These protocols are prohibitively expensive for high throughput and large numbers of clients.

Replication at the clients: While many web applications use a stateless approach, where data is fetched from the servers whenever necessary, a number of applications store data on

the client side for providing fast response times and allowing operation when the device is disconnected. For example, Google Docs and Google Maps can be used in offline mode and Facebook recently announced offline feed access [3].

Parse [1] provides eventual consistency, by allowing applications to store data in client devices, read, and modify these objects while disconnected. Updates are uploaded to the server when connectivity becomes available, with the last write operation prevailing as the server state. SwiftCloud [36] caches objects in the client machine, allowing updates to execute while disconnected. The system supports a weak form of transactions that enforce causality, while merging concurrent updates using CRDTs. Simba [27] reliably stores data on client machines, allowing applications to select the level of observed consistency: eventual, causal, or serializability. Applications must provide functions for resolving conflicts that may arise when operating under weaker consistency guarantees. Our work extends the functionality of these systems, by allowing clients to synchronize directly with each other, thus reducing the latency of update propagation and allowing collaboration when disconnected from servers.

A large number of data management systems for mobile computing have been proposed [33]. Some of these systems, such as Coda [20] and Rover [19], cache data in client machines for allowing disconnected operation. Clients synchronize with servers, with support for reconciling concurrent updates employing some user-defined mechanism. On contrast, our work focuses on supporting also direct peer-to-peer interactions among clients.

In Bayou [34], clients can hold a database replica and synchronize directly. In each replica, updates are first tentatively executed and later committed after being totally ordered by a primary replica. Bayou introduces the notion of session guarantees, ensuring additional properties, such as *read your writes*. Cimbiosys [28] replicates data in the multiple mobile devices of a user, which are expected to be connected intermittently. Although our work share some of the goals and design decisions with these systems, we also focus on the integration with an existing Internet service, which poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients.

Collaborative applications: Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data in client machines. Etherpad [13] allows clients to collaboratively edit documents. ShareJS [16] and Google Drive Realtime [17] are generic frameworks that are able to manage concurrent modifications to different types of objects. All these systems use a centralized infrastructure to mediate interactions among clients. Furthermore, they rely on operational transformation techniques for guaranteeing eventual convergence of replicas [25, 32]. In contrast, our work relies on CRDTs [29] for guaranteeing eventual convergence and to allow clients to synchronize replicas in a peer-to-peer fashion. Collab [7] uses CRDTs for replication between peers and allows offline initialization and collaboration, but only on a local area network.

Peer-to-Peer systems LEGION follows the peer-to-peer model of direct communication among clients. In particular we resort to the use of decentralized unstructured overlay networks [22, 35, 14] and gossip-based multicast protocols [5, 22, 6]. Both of these aspects of peer-to-peer systems have been the focus of extensive research in the past, and we leverage

and adapt some of the approaches found in the literature. In particular, we resort to an overlay network that is heavily inspired by the HyParView overlay network [22], but while HyParView handles faults by using gossip mechanism to maintain additional information about clients in a system, we have adapted the protocol to leverage the centralized infrastructure effectively minimizing the overhead imposed by the overlay management protocol.

For supporting gossip-based multicast, we resort to a typical push-based gossip strategy that has been used in many systems in the past (*e.g,* [22, 5]). Our gossip strategy is biased for providing a good balance between latency and communication overhead, and fault tolerance, being a variant of the ideas proposed in [6].

3 System Design

In this section we present the system architecture and design of LEGION.

3.1 Architecture

LEGION allows programmers to design web applications that benefit from a hybrid communication model where clients can interact among them directly, and maintain replicas of data objects with the (relevant) application state. This helps reducing the dependency on the centralized component, minimizing latency to propagate updates, as updates are distributed directly among clients, while also minimizing the load on the server, as the centralized component is no longer responsible to propagate updates to all clients. Furthermore, it allows clients to continue interacting when the connectivity to the servers is lost. End users can use LEGION-based applications without installing any kind of software or browser plugins, making the use of the framework completely transparent and non-intrusive.

LEGION has been designed to allow simple integration with different web service infrastructures. For supporting a new service, it is necessary to create a small layer that: (i) provides methods for allowing LEGION to read and write data in the service; (ii) exposes a service-specific API to applications running in clients. Besides using the existing web services for data storage, thus ensuring the durability of application data, LEGION can also store information to ease establishing direct communication channels among clients.

Figure 1 illustrates the architecture of LEGION in the client side, identifying the main components and their dependencies/interactions:

- **LEGION API:** This layer exposes the API through which web applications interact with our framework.
- **Communication API:** The communication API exposes two communication primitives, one point-to-point and another point-to-multipoint. Although these primitives are available to the application, we expect applications to communicate using shared objects stored in the object store. This implies that, in most cases, these communication primitives will only be used internally by LEGION.

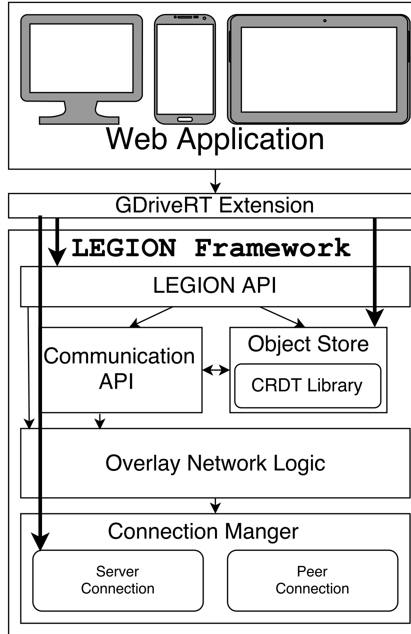


Figure 1: The LEGION Architecture.

• **Object Store:** This module maintains replicas of shared objects that encode relevant application state. Sets of related objects are grouped in a container. This component resorts to the *communication API* to propagate and receive updates to these objects in order to keep replicas up-to-date. Internally, LEGION encodes all objects in the form of CRDTs [29]. LEGION provides an extensible CRDT Library that includes lists, maps and strings.

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

• **Connection Manager:** This module manages the connections established by a client. In order to exchange information directly, clients maintain a set of WebRTC connections among them. (Some) Clients additionally maintain connections to central components, as discussed next.

In addition to the modules discussed above, that reside on the client side, LEGION also resorts to two additional third-party components that reside outside of the client domain. First, one or more *centralized infrastructures* used for several purposes, being the most relevant: i) user authentication; ii) guarantee durability of the application state; iii) allow server-mediated interactions with legacy clients, or clients that run in environments that do not support WebRTC; and iv) assist clients that leverage LEGION to initially join the system. Second, a collection of STUN and TURN [24] servers which are used by our framework to circumvent firewalls and

NAT boxes when establishing direct connection among clients[1].

For the centralized infrastructure, LEGION provides a basic server implemented in Node.JS [10]. Clients can also rely on existing Internet services, which must be accessed through a service-specific extension. We have implemented an extension for supporting integration with GDriveRT. The **GDriveRT extension** provides functions that allow LEGION to read and store data in GDriveRT service and optionally, it can also be used by LEGION to exchange control information to establish direct WebRTC connections among clients. Additionally, this layer allows applications to access shared objects using an interface similar to GDriveRT interface (usually it is possible to port an application that uses the GDriveRT API by modifying only two lines of JavaScript code).

In the remaining of this section we discuss in more detail the design of each of the modules that compose the LEGION framework, focusing on the use of the framework with the GDriveRT extension.

3.2 Communication Modules

3.2.1 Communication API

The communication API provides the interface to exchange messages directly between clients. This module also implements these communication primitives by leveraging the overlay network(s) provided by the *Overlay Management Logic* module (whose design is discussed in Section 3.2.2).

This module offers two main communication primitives. A multicast primitive that allows a client to disseminate a message to all clients that replicate a given object container. In LEGION, each container has a multicast group that clients join when they start replicating an object of the container. This communication primitive operates through a biased push-gossip protocol (similar to the one presented in [22]).

Additionally, this module also provides a point-to-point communication primitive, which is internally implemented on top of our multicast primitive, albeit minimizing the number of messages transmitted by each client. While this implementation is notably inefficient, we have left to future work improving the performance of this primitive, as it currently is not used by LEGION.

3.2.2 Overlay Network Logic

LEGION maintains an overlay network which provides a (partial) membership service to clients using our framework, and defines the interaction patterns among clients (*i.e,* which clients communicate directly). While we allow an application to access multiple containers, we maintain

[1]While LEGION supports both these services, we only use STUN servers which indeed allows to establish direct connections among clients. TURN is used to relay traffic between clients, that are unable to connect directly, through a third party server however, for such clients we instead resort to the centralized infrastructure.

an independent overlay network for each accessed container, which is used for supporting the multicast group associated with that content.

We use an overlay network inspired by the design of the HyParView protocol [22] although with a different mechanism for fault tolerance. This decision is related with the fact that in LEGION we can leverage the centralized infrastructure to simplify the management of the overlay network topology in face of faults. At the overlay level, each client maintains a set of $\mathcal{K}$ randomly selected neighbors ($\mathcal{K}$ is a system parameter). Similar to HyParView, neighboring relationships are symmetric, meaning that if a node a is a neighbor of b, then b is also a neighbor of a. Therefore, our overlay networks are $k-$random undirected graphs.

3.2.3 Connection Manager

The connection manager module manages all communication channels used by a LEGION client. There are two types of communication channels: at most one connection to the centralized infrastructure (either LEGION server component or GDriveRT infrastructure) that we call *Server Connection* and a set of connections to other LEGION clients, that we dub *Peer Connection*. In the following we briefly discuss how these connections are managed.

Server Connection The native Server Connection provided by LEGION, relies on web sockets to communicate with LEGION Node.JS server. This communication channel is then used by LEGION to perform all necessary operations on the centralized component. When using an extension, LEGION uses the functions provided by the extension to contact the Internet service. For the GDriveRT extension, these connections are established taking into consideration the document being accessed in GDriveRT, and are authenticated through the use of Cookies.

Independently of the employed centralized component, the Server Connections are only kept open by *active clients*. Additionally, when a client joins the system, this connection is leveraged to exchange the information required by WebRTC to enable the client to establish its first peer connection. If a client becomes isolated, *i.e,* loses connection to all its overlay neighbors, the server connection is used to allow such a client to obtain information to rejoin the overlay, which greatly simplifies the handling of faults at the overlay level.

Peer Connection A peer connection materializes a direct WebRTC connection among two clients. To create these connections, clients have to be able to exchange – out of band – some initial information concerning the type of connection that each end-point aims at establishing and their capacity to do so (this includes if the connection should be reliable), as well as information necessary to circumvent firewalls or NAT boxes using the STUN/TURN servers. This initial exchange is known, in the context of WebRTC, as *signaling*.

In LEGION we resort to the centralized infrastructure to perform the singling protocol between a client joining the system and a random active client (*i.e,* one that maintains an active server connection). After a client establishes its first peer connection, it starts to use its overlay

neighbors (initially one) to find new overlay neighbors. In the latter case, the signaling protocol required to establish these new peer connections is exchanged through the overlay network directly. When using our extension for GDriveRT, this initial signaling can be performed relying on information exchanged using an hidden document associated with the (main) document (this hidden file is managed by our extension) or through the use of LEGION native Node.JS signaling server (which option to use is a configurable parameter in our extension).

3.3 Object Store

The object store maintains local replicas of containers. Application clients interact by modifying shared objects. LEGION offers an API that enables an application to create and access objects. LEGION provides an extensible library of data types, which are internally encoded under the form of CRDTs. Objects are exposed to the application through object handlers (which are transparent to the application) that hide the internal CRDT representation.

This module is responsible for using the multicast primitive of the *Communication API* module to propagate operations that modify the state of the objects associated with the application as well as for executing operations received from other clients in a way that respects causality (of these operations).

3.3.1 CRDT Library

The CRDT Library included in LEGION supports the following data types: Strings, Lists, and Maps. These CRDTs provide common methods for manipulating their internal state, which are data type dependent. For instance a List supports methods to *insert* and *remove* elements from the List at some given position, and an additional method *toArray* which returns the internal state of the List as a JavaScript object array (for applications to access the contents of the list). LEGION supports composition of objects by reference, i.e., it is possible to put an object in one (or more) other objects (for instance, associate the same string to multiple keys in a map).

Besides these methods used by applications, each CRDT data type also has the following methods to allow the manipulation of the CRDT itself by the LEGION framework (these methods are only used internally by LEGION and are not exposed to the application). First, an *apply* method, which executes one operation over the CRDT instance. This method is used to execute operations received from other clients;

Second, a *merge* method that allows to merge the state of two (divergent) instances of a given CRDT into a new instance. This method is used when two clients establish a new WebRTC connection and need to synchronize their state. In this case, it might be more efficient to propagate the full object state instead of the list of executed operations.

Finally, a *version* method, which returns a version vector that summarizes the updates executed to the CRDT instance. The container also includes a *version* method that summarizes the updates executed to all objects of the container. These methods are used in the synchronization process and for controlling the execution of operations. These version vectors have one entry

for every client that has modified the state of that object. Because in our system any client can modify each object, in the limit, there might be an entry for each client on the version vector. However, we currently focus on applications that share objects among tens of users, which is also the case for most applications that use GDriveRT which we support with our extension. Due to this, we left the optimization of this aspect for future work.

3.3.2 Causal Propagation

In LEGION, we have implemented operation-based CRDTs [29]. To guarantee the convergence of replicas of these CRDTs, all update operations for an object must execute according to their causal order. Additionally, as we want to provide causal consistency for containers of objects which are manipulated together by an application, we require all such operations to be delivered in casual order on every client that hold local replicas of such objects. To achieve this, we use the following approach:

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

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

To prove that this approach respects causal order, we need to prove that when an operation o is received in a client c_r, all operations that precede o in the causal order have already been received. This follows from the fact that if operation o has been received from client c_s and we know that client c_s sends operations in causal order, then c_s has already sent all operations that precede o in the causal order. By the same reason, the lists of operations to propagate to other nodes in c_r continue respecting causal order after adding o to the end of their list. Due to space limitations we cannot present the complete proof here, however it can be trivially achieved through induction.

The actual implementation of LEGION only keeps a suffix of the list of operations received. Thus, 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 propagate the full list of operations. In this case the two clients will synchronize their replicas by sending the state of objects (and using the *merge* method to merge the two replicas). In this case, a client that has executed a merge will need to synchronize with other clients it connects to also by sending its state (to be merged in the receiving client). Additionally, clients exchange vectors periodically to avoid sending operations that are known by their peers.

3.4 GDriveRT Extension

LEGION resorts to a centralized component to ensure both the durability of application state and to serve as an intermediary for the signaling messages used to establish the initial WebRTC connections for each client. To provide this functionality, LEGION can rely on its own specialized server. However, it might be more convenient to the application operator to rely on an existing centralized component. This is particularly relevant when adapting existing JavaScript web applications to leverage our framework. To showcase this feature of LEGION, we wrote an extension for GDriveRT.

Data model: The GDriveRT extension 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 in a LEGION container. Data objects types stored in these documents are similar to the ones supported natively by the LEGION object library. The extension transparently converts the internal representation between (the equivalent) data types used by GDriveRT and LEGION in both directions.

The extension 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 (if not yet initialized). 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 replica are delivered to other document replicas in causal order, i.e., LEGION enforces causal consistency for updates inside a document.

By adopting the same API of GDriveRT, any web application written in JavaScript that uses the GDriveRT API can be (easily) adapted to use 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 ii) replacing the function call to load a document by the equivalent function of the LEGION GDriveRT extension API. With the handler for the loaded document, the application can use exactly the same function calls as in GDriveRT.

LEGION functionality: LEGION can leverage the GDriveRT infrastructure for the following objectives: 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 the gateway between partitioned overlays, for each document, the extension maintains an additional list in GDriveRT with the operations executed in the document. As discussed before, in each overlay, a set of active clients are responsible for uploading operations executed by the clients in the overlay and download and disseminate the new operations from other clients through the overlay. If more than one client executes this process in each overlay, this does not affect correctness, as when an operation is received in a client, if its timestamp is already reflected in the version vector of the replica, it will be discarded. By the same reason that the list of operation in a client respects causal order, it follows that the list maintained in GDriveRT also respects causal order. As a consequence, the operation downloaded from GDriveRT are also disseminated in an order that respects causality.

GDriveRT is also used to reliably keep a snapshot of the CRDT representation of a document, including all CRDT objects. When a client first obtains a replica of a document, it downloads the snapshot and then applies the operations in the list of operations that have not been applied yet. This snapshot is created periodically by clients, when they check that the list of operations has grown over a given threshold. These snapshots are used for efficiency purposes, allowing to create a replica of a document faster in the clients and to discard from the list of operations, the prefix that is reflected in the snapshot.

4 Support for Legacy Applications

While LEGION allows web applications to explore peer-to-peer interactions using the LEGION infrastructure, it also allows legacy applications to continue accessing data using the original GDriveRT interface. For each data object, LEGION keeps two versions: the version manipulated by LEGION and the version manipulated by the 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 the CRDT operations stored in the list of executed operations to the corresponding GDriveRT operations and executing them in the GDriveRT object.

Applying operations executed in a GDriveRT object to the LEGION object is slightly more complex. In this case, 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 list of updates executed by legacy clients is inferred by comparing the current version of the document against the version after the last synchronization (using a diff algorithm). The list of inferred operations is then added to the log of executed operations, guaranteeing that the operations are applied to the 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.

5 Implementation Details

Here we provide a few implementation details of our prototype. The code is publicly available at: `https://github.com/albertlinde/Legion`.

5.1 Overlay networks

To achieve the threshold of $\mathcal{K}$ neighbors we do the following. When a new client wants to access a new container it needs to first join the corresponding overlay, to do that it resorts to the LEGION centralized component to become aware of other clients using LEGION and currently accessing the same container. The new client then uses the LEGION centralized component to exchange signaling information to connect to one such randomly select client.

After establishing this initial peer connection, the new client will establish new peer connections (up to the threshold of $\mathcal{K}$) using its current overlay neighbors to execute the signaling protocol.

5.2 Selection of Active Clients

In our design, we resort to a small subset of clients (that we dub *active clients*) to propagate updates over objects to the centralized infrastructure and also to bias the gossip-based dissemination of messages as to speedup the propagation of updates among clients.

To select these clients we resort to a bully algorithm [15] where initially all clients act as active client, and periodically, every $\mathcal{T} ms$, each client sends to all its overlay neighbors a notification message containing its (locally generated) unique identifier – in our experiments we set $\mathcal{T} = 7000\ ms$. 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. Additionally, and to address the departure or failure of active clients, if a *passive client* does not receive an announcement for more than $3 \times \mathcal{T}\ ms$, it switches its own state back to become an *active client*.

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

5.3 Biased Gossip-based Dissemination

Our gossip protocol is biased to ensure that, when a *passive node* selects the peers to whom it will gossip a message, the *active node* to which it is connected is always selected. Furthermore, we adapt the fanout (*i.e,* the number of nodes to whom a node gossips each message) used by *active nodes* as to ensure that the message is forwarded to all its overlay neighbors with the exception of the one from whom the message was received. Each client only changes the set of peers with whom it gossips, in reaction to changes in its overlay neighbors.

5.4 GDriveRT Extension

When using our extension, operations that resorted to the LEGION centralized component can be executed by leveraging the GDriveRT infrastructure. To achieve such functionality, clients have to be able to determine which other LEGION clients in the system are accessing the same objects (within the scope of a document) as to be able to join the appropriate overlay network. To implement this feature we use an hidden file associated with the main document. This file is leveraged by the new client to exchange the necessary information with a random client already in the system to establish its initial WebRTC connection.

When an application tries to load a document, LEGION attempts to access the requested document at GDriveRT, a step which also serves as authentication for our framework (as the GDriveRT API will implicitly authenticate the user at this point). After authenticating the user, LEGION will access a hidden document containing control objects, managed by our framework (and created by the first client using LEGION that accessed the document). These objects can be used to exchange signaling information for creating peer connections besides carrying information about membership. A copy of the (main) document is then obtained from the decentralized network (or from the GDriveRT infrastructure if there is no other client running that uses LEGION) and, at this point, most clients disconnect from the GDriveRT infrastructure (except active clients).

The implementation of the GDriveRT Extension includes the wrapper interfaces for the native data types provided by LEGION, as well as the code to access the GDriveRT server. We have used `Count Lines of Code` [11] and verified that this extension has $1,251$ LOC in JavaScript. While the whole implementation of LEGION (including the simple server that materializes the centralized component) has $3,602$ LOC of JavaScript.

6 Evaluation

This section presents an evaluation of LEGION with an emphasis on the operation of LEGION when using the extersion to interoperate with the GDriveRT infrastructure. Our evaluation mainly focus on two complementary aspects. We start with an analysis of how complex it is to design application with LEGION in contrast to alternative approaches. Then, we present an experimental evaluation of our prototype, comparing it to the centralized infrastructure of GDriveRT regarding the following practical aspects:

- What is the latency for update propagation?
- How does the system behave when the connection to the central server is (temporarily) lost?
- What is the network load introduced by the peer-to-peer approach?
- How much overhead is induced by the support for seamless integration with legacy clients?

6.1 Designing Applications

In this section, we describe a set of applications that we have ported to LEGION using the GDriveRT extension.

Google Drive Realtime Playground The Google Drive Realtime Playground[2] is a web application that allows to create a document with objects of any of the types supported by GDriveRT. We were able port this applications to LEGION, by changing only 2 lines in the source code (as discussed in Section 3.4).

Multi-user Pacman We have also adapted a version of the popular arcade game Pacman for it to be multiplayer and to operate under the GDriveRT API, which we enriched to also support multiple passive observers that can watch a game in real time. In this adaptation[3], up to 5 players can play at the same time, one player controlling Pacman and the remaining controlling each of the four Ghosts. To manage the multi-player aspect of the game, only the GDriveRT API is used.

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.

Besides extending and porting this application to use the GDriveRT API, we also implemented the same game (with all functionality) using Node.JS to run 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 JavaScript code to materialize 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).

[2] `https://github.com/googledrive/realtime-playground`

[3] The original implementation that runs locally in the browser can be found here: `https://github.com/daleharvey/pacman`

This shows that an API such as the one provided by GDriveRT and LEGION simplifies the task of designing such interactive web applications.

When adapting the game implementation that resorts to the GDriveRT API to leverage LEGION (using the GDriveRT extension), we had only to change two lines of code (as discussed previously). We verified that the game runs much more smoothly (from a user perspective) when leveraging LEGION which we believe happens due to the fact that the latency for propagating user issued commands is much lower.

Spreadsheet We have also explored an additional application: a web-based collaborative spreadsheet editor. Each spreadsheet represent 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 modelled 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 described before).

6.2 Experimental evaluation

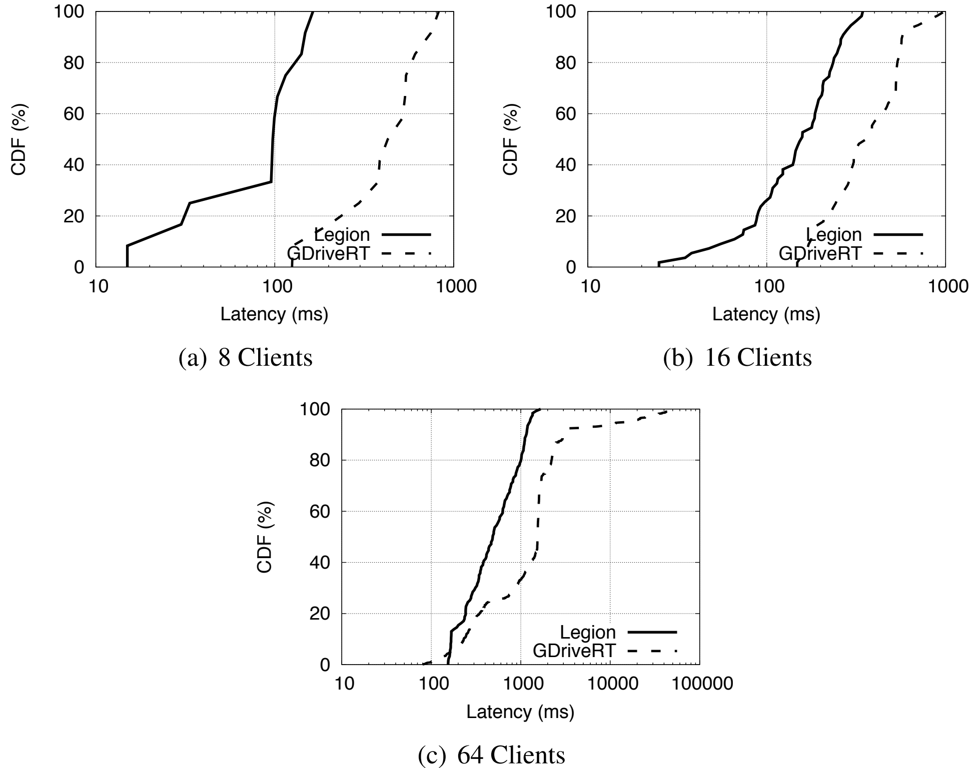
In our experimental evaluation, we compare LEGION using our extension against GDriveRT, as a representative system that uses a traditional centralized infrastructure.

In our experiments, we have deployed clients in two Amazon EC2 datacenters, located at North Virginia (US-East) and Oregon (US-West). In each DC, we run clients in 8 m3.xlarge virtual machines with 4 vCPUs and 8 ECUs of computational power, and 15GB of RAM. In our experiments, half of the clients run in each DC. The average round-trip time measured between two machines in the same DC is 0.6 ms and 80 ms across DCs.

Latency: To measure the latency experienced by clients for observing an update, we conduct the following experiment. Each client inserts in a shared list an (init) string containing his identifier and a timestamp. When a client observes such a string, it adds to the list, as a reply, another string that concatenates the original string and an additional timestamp. When a client observes a reply to his original message, it computes the round-trip time for the client that issued the reply, with latency being estimated as half of that time.

All clients start by writing one string at approximately the same time and reply to all (init) strings added by other clients. Thus, this simulates a system where the load increases with the number of clients.

Figure 2 presents the latency observed for both LEGION and GDriveRT. In LEGION, all clients are included in a single overlay, which is used to propagate messages. The results show that the 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 95th percentile



(a) 8 Clients

(b) 16 Clients

(c) 64 Clients

Figure 2: Latency for the propagation of updates.

for GDriveRT is more than $10\times$ worse than with LEGION, suggesting that LEGION's peer-to-peer architecture is better suited to handle higher loads than the centralized architecture of GDriveRT.

One could expect an even lower latency for many messages when using LEGION, as messages are being propagate to clients in the same DC. This does not happen as our overlay network is not biased towards creating connections with peers that are closer. In the future, we intend to study how to achieve this while keeping the properties of the algorithm we used.

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

Figure 3 shows, at each moment, the average fraction of updates observed by clients since the start of the experiments (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

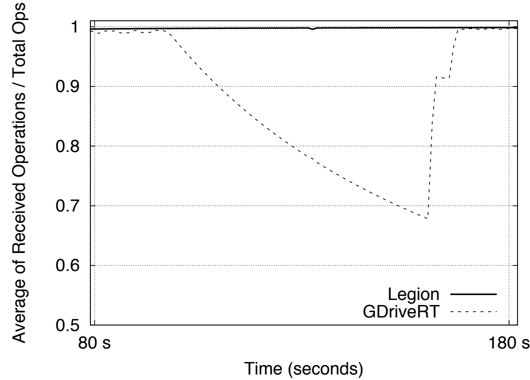


Figure 3: Effect of disconnection

decreases over time. When the connectivity is reestablished, GDriveRT is able to recover. With LEGION, as updates are propagated in a peer-to-peer fashion, the fraction of updates received is always 100%.

Network load: We now study the network load induced by our approach. To this end, we run experiments where 16 clients share a single 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 value is a string of 16 characters and the key has 8 characters. We measure the network traffic by using *iptraf*, an IP network monitor. In these experiments, we run an additional configuration of LEGION (dubbed LEGION *Mixed*) where all signalling is routed through the native LEGION server.

Figure 4(a) shows the total network load of the setup process, which includes making the necessary connections to the GDriveRT infrastructure and peer-to-peer connections. The load is similar across all competing alternatives, which shows that the overhead imposed by the setup of the peer-to-peer component of LEGION is negligible when compared with a centralized architecture.

Figure 4(b) shows 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 computed by dividing the server load – in Figure 4(c) – by the number of clients). This happens because our dissemination strategy resorts to a gossip-based protocol, which has inherent redundancy, whereas in GDriveRT there is no need to propagate redundant information between each client and the centralized infrastructure. However, an average of 35kbps does not represent a huge fraction of available bandwidth nowadays.

Figure 4(c) 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. This is expected, as only a few clients (active clients) interact

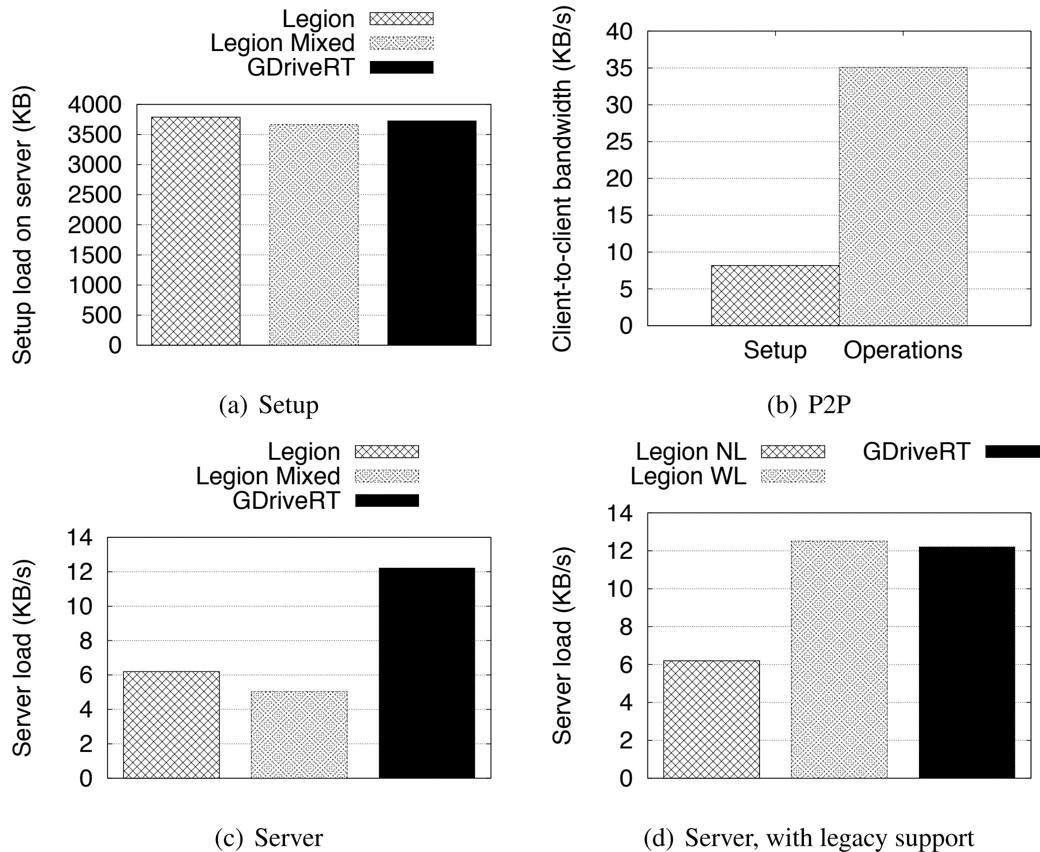


Figure 4: Network load.

with the GDriveRT infrastructure, being most interactions propagated directly between clients. Interestingly, when performing signalling through our LEGION server, the load in the centralized component drops slightly. This happens because the signalling mechanism performed through the GDriveRT API is less efficient.

The results presented in Figure 4(d), measure the overhead imposed on the centralized component for supporting legacy clients (LEGION *WL*) and compares it with LEGION with this option disables (LEGION *NL*) as well as the load imposed over the centralized component of GDriveRT when clients do not use LEGION. Results show that supporting legacy clients incurs in significative overhead, leading the load imposed on the centralized component to rise to values similar to the centralized architecture. This happens because the mechanism used to support legacy clients 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.

7 Final remarks

In this paper we presented the design of LEGION, a framework that allows to extend web applications, by supporting replication at the client machine and using peer-to-peer interactions to propagate updates among clients. Furthermore, we propose a mechanism to allows the coexistence of legacy clients accessing GDriveRT directly and clients using our new framework. The evaluation of our prototype shows that latency for update propagation is much lower with LEGION when compared with using GDriveRT, and that clients continue to receive updates while disconnected from the servers.

As future work we plan to design and implement extensions to integrate LEGION with storage services such as Cassandra, Riak, and Redis.

References

[1] Parse. http://parse.com.

[2] Webrtc. http://www.webrtc.org/.

[3] `https://goo.gl/Q06CaL`, Dec. 2015.

[4] ALMEIDA, S., LEITÃO, J., AND RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of EuroSys '13* (2013), pp. 85–98.

[5] BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. Bimodal multicast. *ACM TOCS 17*, 2 (1999), 41–88.

[6] CARVALHO, N., PEREIRA, J., OLIVEIRA, R., AND RODRIGUES, L. Emergent structure in unstructured epidemic multicast. In *Proceedings of DSN'07* (Edinburgh, Scotland, UK, June 2007), pp. 481 – 490.

[7] CASTIÑEIRA, S., AND BIENIUSA, A. Collaborative offline web applications using conflict-free replicated data types. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data* (2015), PaPoC '15.

[8] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[9] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Google's globally distributed database. *ACM TOCS 31*, 3 (2013), 8.

[10] DAHL, R. Node. js: Evented i/o for v8 javascript. *URL: https://www. nodejs. org* (2012).

[11] DANIAL, A. Cloc–count lines of code. *Open source* (2009).

[12] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev. 41*, 6 (Oct. 2007), 205–220.

[13] ETHERPADFOUNDATION. Etherpad.

[14] GANESH, A., KERMARREC, A.-M., AND MASSOULIÉ, L. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*. 2001, pp. 44–55.

[15] GARCIA-MOLINA, H. Elections in a distributed computing system. *Computers, IEEE Transactions on C-31*, 1 (Jan 1982), 48–59.

[16] GENTLE, J. Sharejs api.

[17] INC., G. Google drive realtime api. https://developers.google.com/google-apps/realtime/overview.

[18] JAY, C., GLENCROSS, M., AND HUBBOLD, R. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact. 14*, 2 (Aug. 2007).

[19] JOSEPH, A. D., DE LESPINASSE, A. F., TAUBER, J. A., GIFFORD, D. K., AND KAASHOEK, M. F. Rover: A toolkit for mobile information access. In *Proceedings of SOSP '95* (1995), pp. 156–171.

[20] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. *ACM TOCS 10*, 1 (Feb. 1992), 3–25.

[21] LEITÃO, J., PEREIRA, J., AND RODRIGUES, L. Epidemic broadcast trees. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on* (2007), IEEE, pp. 301–310.

[22] LEITÃO, J., PEREIRA, J., AND RODRIGUES, L. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007* (2007), IEEE, pp. 419–429.

[23] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of SOSP '11* (2011), pp. 401–416.

[24] MAHY, R., MATTHEWS, P., AND ROSENBERG, J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), RFC 5766. Tech. rep., IETF, Apr. 2010.

[25] NICHOLS, D. A., CURTIS, P., DIXON, M., AND LAMPING, J. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology* (1995), ACM, pp. 111–120.

[26] PAN, J., HU, H., AND LIU, Y. Human behavior during flash crowd in web surfing. *Physica A: Statistical Mechanics and its Applications 413* (2014), 212 – 219.

[27] PERKINS, D., AGRAWAL, N., ARANYA, A., YU, C., GO, Y., MADHYASTHA, H. V., AND UNGUREANU, C. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *Proceedings EuroSys '15* (2015), pp. 7:1–7:16.

[28] RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. Cimbiosys: A platform for content-based partial replication. In *Proceedings of NSDI'09* (2009), pp. 261–276.

[29] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free Replicated Data Types. In *Proceedings of the 13th SSS'11* (2011), pp. 386–400.

[30] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of SOSP '11* (2011), pp. 385–400.

[31] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[32] SUN, C., AND ELLIS, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer supported cooperative work* (1998), pp. 59–68.

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

[34] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of SOSP '95* (1995), pp. 172–182.

[35] VOULGARIS, S., GAVIDIA, D., AND VAN STEEN, M. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management 13*, 2 (2005), 197–217.

[36] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Proceedings of the Annual ACM/IFIP/USENIX Middleware Conference* (Dec. 2015), ACM/IFIP/Usenix.