Pedro Filipe Nunes Durães

Integration of browser-to-browser architectures with third party legacy cloud storage

Relatório intermédio para obtenção do Grau de Mestre em

Engenharia Informática

Orientador:	Nuno Manuel Ribeiro Preguiça, Professor Associado, Universidade Nova de Lisboa
Co-orientador:	João Leitão, Postdoctoral Fellow, Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2016

Abstract

An increasing number of web applications run totally or partially in the client machines - from collaborative editing tools to multi-user games. Avoiding to continuously contact the server allows to reduce latency between clients and to minimize the load on the centralized component. Novel implementation techniques, such as WebRTC, allows to address network problems that previously prevented these systems to be deployed in practice. Legion is a newly developed framework that exploits these mechanisms, allowing client web applications to replicate data from servers, and synchronize these replicas directly among them.

This work aims at extending the current Legion framework in two significative ways: (i) integration of additional legacy storage systems that are often used to support web applications. Furthermore we plan to study how the different data models supported by these storage services and interfaces affect the performance of an integration with Legion; (ii) enrich the Legion framework with causal consistency by leveraging mechanisms provided by the centralized storage services.

Keywords: distributed storage systems; CRDT; Legion; causal consistency.

Resumo

Um crescente número de aplicações *web* é executado totalmente ou parcialmente nas máquinas do cliente - desde ferramentas de edição colaborativa até jogos de vários utilizadores. Ao cortar a execução de operações no servidor é possível reduzir a latência entre clientes e minimizar a carga no servidor. Novas tecnologias, como WebRTC, permitem atacar problemas que anteriormente afectavam a implementação destes sistemas num ambiente de produção. O Legion, é uma *framework* recém desenvolvida que explora este mecanismos, permitindo que aplicações *web* do lado do cliente consigam fazer replicação de dados e sincronização directa entre elas.

Neste trabalho pretende-se estender a *framework* Legion de duas formas significativas: (i) integração com sistemas de armazenamento já existentes que são tipicamente usados para suportar aplicações *Web*. Também pretendemos estudar o impacto que diferentes modelos de dados usados por estes sistemas de armazenamento assim como as suas interfaces têm no desempenho do sistema resultante da integração do Legion com estes sistemas; (ii) enriquecer o Legion para suportar consistência causal alavancando nos mecanismos disponibilizados pelos serviços de armazenamento centralizados.

Palavras-chave: sistemas de storage distribuido; CRDT; Legion; consistencia causal

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Web applications have taken a central role in access to remote services. Traditionally, clients would only act as an end interface of the data from/to the server, not keeping data nor making computation, with the exception of computation related to data presentation. With the grow of processing power and storage capabilities of commodity hardware, such as PCs, Laptops, SmartPhones and Tablets, there has been a tendency to move some of the computation and data to these devices, instead of always relying on the servers.

Peer-to-peer technologies have been around for long, but it has not been used in web applications due to the difficulty of having one client communicating with other clients. When communicating directly among clients, there can be benefits in terms of (i) latency, since devices can be closer to each other than to a server; (ii) scaling, work can be partitioned between peers; (iii) availability, because a service doesn't need to stop if the central server is temporarily down. Recent technology developments in this area, such as WebRTC, STUN and TURN, allowed browser-to-browser communication with no need for native applications or browser plugins.

As an example of this, Legion is a newly created framework which explores these technologies and allows client web applications to replicate data from servers, and synchronize these replicas directly among them. Browser-to-browser communication is useful in web applications that exhibit frequent interchange of information or sharing of contents between clients, such as collaborative editing.

Although replicating data between web clients is a promising approach, personal devices can be unstable compared to the use of a centralized component server because they join and leave a network very often, so data persistency cannot be fully delegated to these devices. There is a need store data periodically in a more stable storage system.

1.2 Motivating Problem and Solution

The processing power and the amount of storage of user devices allow current systems to transfer part of the work and data from a central server (or servers) to these end-point devices. There has been an effort to build tools that facilitate programmers to develop software that takes advantage of direct communication between clients.

In this scenario, communication between browsers is possible with minimal effort from both developer and user, thanks to frameworks like WebRTC that enable direct real time communication between browsers. With WebRTC it is possible to stream data, audio, or video between browsers with no need for plugins or extensions.

WebRTC made possible frameworks like Legion. Legion allows client web applications to replicate data from servers, and synchronize these replicas directly among them. This can have major impact in areas like collaborative editing, where current approaches like Google Docs always use a central server do mediate communication between clients. Legion offers the same API as Google Drive Realtime, thus allowing to easily port existing applications to the new framework.

However, there are some disadvantages in direct browser-to-browser communication. User devices are unstable when compared to a server on a data center, as they can join and leave the system frequently. This makes imperative to include in the system a centralized component where data durability is guaranteed. Part of the goal of this thesis is to study how to integrate well known legacy distributed storage systems with the Legion framework, giving this framework more robustness when it comes to persistence of data and allowing clients that don't support WebRTC to use the framework as an old fashion centralized server approach.

This integration will have the following main challenges. First, the incompatibility between the Legion framework data model and the storage systems data models. To address this challenge, it will be necessary to both extend the Legion data model and to create a mapping between the two models. Second, the need to keep data synchronized between the central storage and the Legion framework, considering that replicas can be modified concurrently. This encompasses the following problems: (i) identifying the updates that have been executed in the central server and in Legion; (ii) propagate the updates across systems efficiently. Besides this, and based on the study of distributed storage systems techniques, an important topic is consistency guarantees in the data propagation. Most well known storage systems offer some sort of consistency policy, whether this is atomic transactions, eventual consistency, or causal consistency. Currently, Legion support causal consistency for each object, but not across objects. In this work, we will study how to bring causal consistency between objects to the client side. This will help peer-to-peer application developers to have an easier time reasoning about data propagation between nodes.

1.3 Expected Contributions

The planned work for this thesis will be based on the understanding of Legion and the study of well known legacy distributed storage systems. In section 3.1 a more in depth description of the planned work is given. The expected contributions of this work will be:

- Extend Legion to support integration with existing storage systems, such as Antidote, Riak and Cassandra.
- The extension of Legion to support causal consistency across objects, thus providing the same consistency level of some of the storage systems.

1.4 Document Structure

The remainder of this document is organized as follows:

Chapter 2 describes the related work. Existing work is explored in the areas of communication technologies, peer-to-peer systems and distributed storage systems.

Chapter 3 discusses the proposed solution and work plan for the duration of this thesis.

Chapter 2

Related Work

In this chapter, will be presented various aspect that will help with the work to be developed in this thesis. The following sections cover in particular:

- In section 2.1, an overall study of the WebRTC technology is presented.
- In section 2.2, storage systems mechanisms and examples are depicted, as this is one of the most important topics for this thesis.
- In section 2.3, peer-to-peer system mechanisms are explored, as they are widely used, even in storage systems.

2.1 WebRTC

WebRTC[4] is a framework for the web that enables Real Time Communications among browsers. Before the availability of WebRTC, real time communication was either done via native applications or plugins, which demanded large downloads and/or a great effort for both developers and users to install and keep updated. These disadvantages would make web-based applications that resort to direct communication among clients not viable for both operators and users alike. With WebRTC, the final user can have a much better experience on its browser and the developers can benefit from a structured and easy to use API to develop Web applications.

WebRTC includes three main components: audio, video and network. In the network component are mechanisms that to deal with network related practical issues. Also included are components for facilitating the establishment of peer-to-peer connections using ICE / STUN / Turn / RTP-over-TCP as well as support for proxies.

Although WebRTC was design to be used in Peer-to-Peer contexts, it relies on a centralized component for particular interactions:

- Before any connection can be made, WebRTC clients (peers) need to exchange network information (signaling protocol).

- For streaming media connections, peers must also exchange data about the media contents being exchanged, such as video encoding and resolution

- Additionally, as clients often reside behind NAT[1] gateways and firewalls, these may have to be traversed using STUN (Session Traversal Utilities for NAT) or TURN (Traversal Using Relays around NAT) servers.

2.1.1 Signaling

Signaling[25] is the process of coordinating communication. In order for a WebRTC application to set up a "call", its clients need to exchange the following information information: (i) session control messages used to open or close communication; (ii) error messages (iii) media metadata such as codecs and codec settings, bandwidth and media types (iv) key data, used to establish secure connections (v) network data, such as a host's IP address and port as seen by the outside world

This signaling process needs a way for clients to pass messages back and forth. This mechanism is not implemented by the WebRTC APIs, it must be implemented by the application developer. This implementation can be a simple messaging system that uses, for example, a central server.

2.1.2 STUN

NATs provide a device with an IP address for use within a private local network, but this address can't be used externally. Without a public address, there's no way for WebRTC peers to communicate. To get around this problem WebRTC uses STUN.

STUN severs have the task of checking the IP address and port of an incoming request from an application behind a NAT, and send that address back as response. This makes possible for an application to find its IP address and port from a public perspective.

2.1.3 TURN

TURN is used to relay audio/video/data streaming between peers. When direct communication via UDP and TCP fails, TURN servers can be used as a fallback. Unlike STUN servers, that do a simple task and do not require much bandwidth, TURN servers are heavier to support, as they relay data between peers.

[1] Network address translation.

2.2 Storage Systems

Storage is a fundamental aspect for the majority of applications and web applications in particular. Current web applications demand a set of characteristics that fit with their goals, whether this is strong consistency, high availability, global geo-replication of data, or high performance read and write operations.

2.2.1 Data Models

A data model that a storage system uses as internal representation will influence the outcome performance of the system at many levels: (i) querying speed, as the organization of data will influence how fast can the system find it; (ii) scalability, because different amounts of data requires different data structures; (iii) querying functionalities/operations, because how the data is stored will influence the different ways of searching and operating with that information.

2.2.1.1 Relational Model

The relational model[7] is vastly used in traditional databases. It represents data with a collection of relations. Each relation is a table that stores a collection of entities, where each row is a single entity and each column an attribute. Every table has one or more attributes that together make a unique key. This key allows efficient queries by indexing the table.

The relational model allows powerful queries by relying on relational algebra. Commonly used operations between tables include Select, Join, Intersect, Union, Difference and Product.

2.2.1.2 Key-Value Model

Key-Value stores use keys associated with values, making a collection of pairs. Storing and retrieving information from a key-value database can be very efficient, because it can rely on a dictionary to keep track of all the keys, using a hash function to access a position of the dictionary. Although more efficient in certain conditions, this model doesn’t support such powerful querying as the relational model.

The key-value model has been getting more traction recently, as multiple databases and storage systems use it due to its scalability potential.

2.2.1.3 Bigtable Model

Bigtable[6] uses tables as its data model, but not in a relational way. A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map. Data is organized into three dimensions: rows, columns and timestamps. In order to access a specific storage cell, one needs to reference the row key, column key and timestamp. Columns can be grouped into column families, which form the unit of access control. Table updates are executed at a transactional row level.

This approach allows the system to scale, while still offering some control over the data. System scalability can be guaranteed by allowing concurrent access to different rows, since Bigtable offers transactions per row.

2.2.2 Distributed Storage Systems Techniques

Each storage system has to address multiple aspects related with the storage and manipulation of data. These aspects include data partitioning when considering systems that are materialized by multiple machines, replication to ensure data availability and good performance, data versioning, membership control, failure handling, scalability, and overall system performance.

Each storage system tackles these aspects using different solutions to best fit a specific working environment or particular needs of the system operating on top of them.

2.2.2.1 Consistency Guarantees

There is a well known trade-of between performance and consistency in replicated systems, and in particular in geo-replicated systems[9]. In a nutshell, the greater the consistency guarantees offered by a system, the less performant that system becomes. The main consistency models are presented below:

Strong Consistency With Strong Consistency models strive to have all updates serialized in the same order across all replicas. While this can benefit the programmer by making it easier to reason about the state of a system, offering such guarantees makes the system less performant, and can even halt the system, when replicas cannot be contacted, for instance when there is a network partition.

Snapshot Isolation Snapshot isolation[15] guarantees that all reads made in a transaction will see a consistent snapshot of the system taken in the start of that transaction. This transaction will be successful if the updated values have not been changed since the beginning of the transaction, when the snapshot was taken.

Eventual Consistency Eventual consistency[3] guarantees that all updates will reach every replica eventually and that the state of all replicas will eventually converge. This usually means that update operations (i.e. operations that modify the state of the system) can usually be executed locally at one replica without resorting to coordination(in the critical path of operation execution). This implies that operations return their replies to clients before the operation becomes known by all replicas in the system.

This allows clients that issue read operations, to observe old, or divergent versions (from other clients reading the same content) of the data depending on which replica processes the client read operation.

Causal Consistency Causal consistency is a consistency model, whose guarantees lie somewhere in between strong consistency and eventual consistency. While it allows clients to observe divergent values, contrary to eventual consistency, it guarantees that clients observe values that respect causal relationships between them[2, 14]. A causal dependency is formed, for example, when a node performs a read followed later by a write, even on a different variable, the first operation is said to be causally ordered before the second, because the value stored by the write may have been dependent upon the result of the read.

In practice, respecting the causal relationship on these objects implies that a client should not be able to observe the effect of the write operation discussed above without being able to observe the same (as a latter version) value read by the client that issued that write.

2.2.2.2 Partitioning

In distributed storage systems, multiple nodes materialize the state of the database. This is done to achieve both load balancing and scalability (due to fault tolerance it is also crucial to replicate the same data across multiple nodes). Bellow are presented the main methods to do this:

Consistent Hashing Consistent Hashing[11] is about using a hash function to map each data item to a location(node) in the system. The range of values of the hash function can form a ring by overlapping the highest hash value with the lowest value. Each node in the system is assigned a random value within this range. Searching an entry in such system can be very efficient, because it only requires to apply the hash function to find the item location. Keeping this structure has maintenance costs, because the system needs to update the active nodes as they leave or join.

Directory Based lookups [17] Directory-based approaches can place replicas in arbitrary locations but require a lookup to the directory to find the replica location. This may involve one or more round-trips in the network. This technique has less maintenance costs then consistent hashing, as node joins and leaves are not as heavy to update. Finding an item may take more time as the system scales.

2.2.2.3 Replication

For fault tolerance and to guarantee the persistence of data[17], each item in the system is replicated across N storage nodes in most distributed storage systems.

The number of individual replicas N, controls a trade-off between performance and fault tolerance. With higher values of N, each data item is replicated across more nodes, so the system becomes more tolerant to individual node failures. With lower values of N, the system becomes more performant, as write operations have to be propagated to less machines. This trade-off has been explored in the past[9], leading to the proposal of multiple replication protocols, such as quorum systems[21] among others [1, 19].

To determine how to distribute the load of N replicas across all nodes, one can use the following techniques:

Master/Slave Replication In master/slave replication, each master node can have multiple slaves. The role of the master node is to receive the updates and replicate the acquired data to all of the saves that it is connected to. The slave nodes are used to answers read calls.

Publish/Subscribe The publish/subscribe replication is characterized by listeners subscribing to channels, with publishers sending data to these channels that connect to subscribers. In the particular case of replication, nodes with an interest on a specific source of data (another node) will subscribe to his channel, receiving the updates from it.

Neighbor Replication Neighbor Replication keeps copies of each item in the $N-1$ neighbors of the node responsible for that key. While this allows to keep tight control on the replication degree, by triggering the creation of new replicas when neighbors change, it has a high maintenance cost, since replicas must be created and destroyed with every change in the network.

Virtual Servers In a system working with Neighbor Replication, one can use Virtual Servers to improve load balancing of the system.

Using this approach, each physical node presents itself as multiple distinct identities to the system. Each identity represents a virtual node maintained by that server. On the other hand, this may amplify the effect of churn[2], since the departure of a single node leads to the simultaneous failure of multiple virtual nodes.

Multi-Publication Multi-Publication stores N replicas of each data item in different and deterministically correlated positions. While this offers very good load balancing properties, it requires a monitoring scheme to detect departure/failure of nodes.

Resilient Load-Balancing This policy uses groups to manage the load balancing of the systems. Each group has a collection of data that is replicated between group nodes. When a new node enters the system, it joins the group with fewest members. Load balancing of data is kept by splitting and merging groups when it reaches a certain unbalanced threshold.

[2] Churn is the participant turnover in the network (the amount of nodes joining and leaving the system per unit of time)

Most Available Node In this policy, data is placed in the nodes predicted to be the most available in the future with higher probability. Using this technique, fewer items are affected by failures and fewer replicas need to be created again. This introduces savings in data transfer costs, but it creates unbalanced load in the system nodes.

2.2.2.4 Multi-version tracking

Some systems that allow weaker consistency, other than serializability, resort to versioning, where multiple versions of the same data item are maintained at the same time. To distinguish between these, there must exist an unique identifier for each version of an object.

Vector Clocks Each update to a data item is always tagged by a vector clock that uniquely identifies that data item version.

A vector clock is a list of pairs ($node, counter$), where $node$ is the identifier of a node in the system that issued the event/operation being tagged and $counter$ is a monotonically increasing value associated with that node.

With this we can keep separate versions of one data item and we can check if two different versions conflict (meaning that how vector clocks encode concurrent events or divergent versions). To do so, we compare the two vector clocks and verify if either one descends from the other or that there is a conflict caused by two separate updates from the same version(i.e. versions have divergent states of an object).

2.2.2.5 Version Reconciliation and Conflict Resolution

When concurrent (and not coordinated) updates are issued over two replicas of a given data object, these replicas will potentially evolve to divergent states, that at some point must be resolved into a single converged state, ideally are that the effects of all operations that generated the divergences. To take this action there are mainly two aspects to consider: how to do it and when to do it. Multiple techniques address this issue, being the main ones briefly described below:

Last Writer Wins With this simple approach the last update based on some notion of time is the one that is adopted by the system. While this is trivial to implement, using local machine clocks, they are usually not synchronized across nodes, which can lead to incorrect decisions and to lost updates.

Programmatic Merge This technique leverages on the programmer to specify what happens when two versions conflict. Every node is required to either have a piece of code that decides how to merge divergent versions of an object or to show both states to the end user on-line and delegate on the user the decision concerning the final converged state.

Commutative Operations

Another approach is to design the system by only allowing commutative operations to be performed, this means that despite the order in which operations are executed, the final result will be the same, as long as all replicas execute all operations. If we can make every update commutative, then conflict resolution becomes only a matter of ensuring that all replicas executed all operations.

A simple example is a counter that only allows increment operations. Independently of the order of these increments, the final value will be the same across all replicas.

Operation Transformation The goal of this technique is to transform operations issued over the system in an automatic fashion as to ensure that they can be applied in any order and still ensure the correctness of the system. This is specially useful when working on collaborative editing.

CRDTs Convergent or Commutative Replicated Data Types. These are distributed data types and structures with mathematical properties that ensure eventual consistency and state convergence while being very scalable and fault-tolerant.

A CRDT[20] requires no synchronization during the execution of operations to ensure state convergence, an update executes immediately, unaffected by network latency, faults, or network failures. There are two kinds of CRDTs:

CvRDTs, State-based Convergent Replicated Data Types. The successive states of an object should form a monotonic semi-lattice and the replica merge operation computes a least upper bound. In other words, the merging of different states will eventually converge to the same final state at each replica. It allows for weak channel assumptions (i.e. unreliable communication channel). However, sending over the entire state might be inefficient for large objects.

CmRDTs, Operation-based Convergent Replicated Data Types. In the operation-based CRDTs, concurrent operations commute. Operation-based replication requires reliable communication channel with ordering guarantees, such as a causal order between operations.

Both classes of CRDTs are guaranteed (by design) to eventually converge towards a common single state.

CRDTs tend to become inefficient over time, as metadata accumulate and internal data structures can become unbalanced[20]. Garbage collection can be performed using some form of weak synchronization, outside of the critical path of client-level operations.

2.2.2.6 Membership/Failure Handling

It is important in distributed storage systems to keep track of which nodes join and leave the system, so we can maintain the guaranties of replication and correct mapping of data objects to nodes. There are several techniques to achieve this, in particular:

Gossip Protocol This technique relies on periodic communication between pairs of nodes to synchronize their local information concerning system membership.

When a node joins the system, it communicates with a set of existing nodes and these periodically contact random peers, synchronizing their membership list. This keeps going until every node in the system knows the existence of the new node. Similar procedure goes for a leaving node.

Anti Entropy Anti Entropy mechanisms are useful to recover from scenarios where nodes fail. These mechanisms detect inconsistencies between replicas and are used to synchronize data between them. This is a specialized form of gossip protocol.

A fast way to detect inconsistencies and minimize the amount of transferred data is to use Merkle Trees[22]. Merkle Tree is a hash tree where leaves are hashes of its keys and parent nodes are hashes of the values of their children. To check if two replicas are synchronized, we compare the hash value at the roots, if it is the same value they are synchronized, if not, the values of the hashes at each level of the tree are recursively compared until all divergent nodes are located. Synchronization is then performed only over the nodes whose values have diverged.

Sloppy Quorum Usually writes need to be performed in a sub-set of nodes, in order to provide consistency and safe persistence. In scenarios where a temporary failure of one of those nodes happens, such operations could not be executed.

Sloppy Quorum allows to write the updated data item to another node, other then the ones previously selected. This node is now responsible to periodically try to contact the original node and deliver the updates that it had previously missed.

Accrual Failure Detection [10] The idea of an Accrual Failure Detection is that the failure detection module does not emit a boolean value stating that node is up or down. Instead, the failure detection module emits a value which represents a suspicion level for each of monitored nodes. The basic idea is to express the value of Φ on a scale that is dynamically adjusted to reflect network and load conditions at the monitored nodes.

2.2.3 Storage System examples

In this section, we will present some of the distributed storage systems that are important for this thesis:

Dynamo [8] Dynamo is a distributed storage system used by several services at Amazon, it uses the key-value storage model, the interface supports get and put operations and it guarantees eventual consistency across data replicas.

It uses an always writable policy for updates, with conflict reconciliation on reads. To do so, data is partitioned and replicated using consistent hashing where the hash values are mapped into node identifiers, that are organized in a circular ring formed by their identifiers, this is done by overlapping the highest hash value next to the lowest one.

Replication of each data item is configurable with the number of replicas N. So besides being stored in the responsible node (according to their hash value) each data object is replicated using neighbor replication. Load balancing is achieved by using virtual servers mapped to the same physical node. It is also possible to adjust the number of nodes that are used to serve read or write operations using parameters R and W.

To keep track of multiple versions of the same item, Dynamo uses vector clocks for version tracking and executes version reconciliation when read operations are submitted on an object. This technique favors write performance.

To handle temporary node failures Dynamo uses a sloppy quorum. Upon the need to recover from a failed node, it uses an anti entropy mechanism backed by Merkle trees to calculate differences between replicas.

Membership is maintained in a distributed way using a gossip protocol.

Redis [5] Redis is a non-relational in-memory database that was built to fill the needs of modern web applications, so its focus is mostly on read performance and availability.

It provides eventual consistency of data while supporting partitioning and replication with publish/subscribe and master/slave techniques. Besides the common key-value interface, it also offers Data structures such as strings, lists, sets, sorted sets, and hashes. Hashes are key-value pairs inside a key and are optimal for storing attributes inside an object.

Cassandra [13] Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers.

Cassandra uses tables similarly to Bigtable. Table columns can have colums inside themselves. To access these, Cassandra offers an API with `insert`, `get`, and `delete` methods.

To distribute data across the system nodes, it uses consistent hashing with a ring layout, similar to Dynamo. Replication is handled with three different policies. "Rack Unaware" replicates each item using neighbor replication, "Rack Aware" and "Datacenter Aware" use a system node as a leader to coordinate the ranges of the replicas and spreads the copies of each object across machines that are either on different racks on the same datacenter or in different datacenters. To achieve load balancing it uses resilient load-balancing, based on the analysis of the load information of the ring.

Membership is handled with an anti-entropy gossip protocol and it further relies on accrual failure detection for failure detection and handling.

Riak [12] Riak is a distributed key-value data store system that provides high availability by allowing to chose between strong and eventual consistency. Eventual consistency in Riak uses CRDTs at its core, providing CRDT data types that include counters, sets, flags, registers, and maps.

It uses many of Dynamo's concepts, such as consistent hashing for partitioning data around the replicas ring, neighbor replication with virtual nodes to guarantee data availability, vector clocks for object versioning, a gossip protocol to communicate membership changes and an anti-entropy mechanism based on Merkle Trees to detect differences between replicas.

Besides the common interface methods GET, PUT and, DELETE, it also offers Map-reduce querying, secondary indexes, and full-text search functionalities.

Antidote [23] Antidote is a distributed CRDT key-value store written using the Riak Core module that is intended to provide partitioning, Intra-DC (Data center) and Inter-DC replication, while also supporting atomic write transactions.

Antidote has direct support for read and writing operations over CRDTs, this makes it easier for the programmer to reason about and work with these data structures.

The system architecture is divided in the following layers: (i) across DCs replication layer, is responsible for replicating updates to other DC and include both components need in the sender and the receiver DC. The update propagation is done in a FIFO (First in first out) order; (ii) transaction layer, is responsible for providing transaction support, Snapshot Isolation and causality tracking between objects; (iii) materializer, is in charge of forming the objects from the set of operations returned by the logging layer; (iv) logging layer, is used to provide fast write operations, as it appends an operation to the log before the actual update propagation; (v) replication within a DC, uses strict quorum to perform log reads and writes among N replicas within the DC.

This layered architecture design allows to easily add or remove features to the system as well as rely on different strategies for each of these modules.

2.2.4 Distributed Caching Systems

Traditional caching is about using the main memory to temporarily store the most frequently accessed data, in order to speed up te process of fetching data. Caching in a network context has been used for a long time with the main purpose of improving web access times and reducing traffic and load to the storage to the storage service. With a distributed caching architecture one can achieve better hit ratio, response times, and load balancing[18].

Distributed Caching is becoming more popular since main memory prices have been lowering. Hence, caching is becoming a more relevant layer of the storage service of web applications. Here are some examples of Distributed Caching Systems:

Memcached [24] Memcached is a high-performance, distributed memory object caching system originally intended for use in speeding up dynamic web applications by alleviating the database load.

In Memcached, instead of having a dedicated amount of memory for each node, the whole available memory of the distributed caching service is shared by all the nodes.

Redis Redis can also be used as a caching system if configured to only use the main memory, not dumping periodically to disc.

2.3 Peer-to-Peer Systems

The term “peer-to-peer” (P2P) refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner[16].

Peer-to-peer is an alternative to the client-server model, in its purest form there is no notion of server, as every node is a peer that can act as server or client when needed. There are certain characteristics that can be tuned in peer-to-peer systems to achieve a desired working environment. A key aspect of peer-to-peer systems is overlay networks, logical networks that operate at the application level and that are used to manipulate and control the interactions among the participants of the systems.

2.3.1 Degree of Decentralization

Decentralized

In a fully decentralized system, every peer is an equal participant. This way we can avoid bottlenecks and single points of failure, while promoting scalability and resilience. On the other hand, it is hard to keep a global view of such a network, since there is no coordinator node. Usually these networks rely on flooding protocols to propagate changes.

To address this issue, some systems elect supernodes in order to balance load in the network and becoming an entry point for new participants.

Partly Centralized

Partly Centralized systems have a centralized component that stores information about the available resources and acts as a coordinator to the other nodes. This component is responsible for managing node connections.

These systems are simpler to build than fully decentralized ones, but have a potential of bottleneck and single point of failure in the centralized component, where unavailability can potentially render the system impossible to access.

2.3.2 Structured vs Unstructured

This design option is usually based on how useful it is to have a performant exact match querying mechanism and the amount of churn in the network.

Structured Network

Structured networks are specially useful to make efficient queries in the network. Each node has a unique identifier that is used to discover its location among the remaining nodes. Most structured overlay networks rely on consistent hashing to operate, forming a ring array with nodes of the system accordingly to their identifiers, behaving as a distributed hash table (DHT). This structure works similarly to a traditional hash table, where a value can be easily found by its key.

Building and maintaining such structure has costs. The DHT needs to constantly update the list of active nodes, which can be very expensive if network churn is high.

Unstructured Network

In an unstructured network each node keeps its own collection of resources that surround it and updates are propagated using flooding mechanisms. Not having a particular structure makes the network more tolerant to churn, but propagating queries becomes a heavier task as the topology preaches no indication of the location of resources, leading the query to be propagated through a gossip like protocol to all the other participants of the system.

2.4 Summary

This chapter has covered the existing work that will support, influence, and help the development of this thesis.

The main focus is around storage systems and the many techniques they use to tackle different challenges in the design and operation of those distributed systems. The major aspects surround consistency, partitioning, replication, object/update versioning, version reconciliation and membership/failure handling.

This chapter also discussed peer-to-peer systems and some aspects of overlay networks, since they are widely used in many distributed systems and in particular to implement storage systems.

Besides that, the fundamentals of WebRTC, a crucial aspect for the work done on the Legion framework that is also one of the goals of the work to be developed in this thesis, was briefly introduced.

In the next chapter, we discuss the planned work for this thesis, with its major goals and time planning.

Chapter 3

Proposed Work

In this chapter, the work to be done along the duration of this thesis will be presented. The next sections are organized as it follows:

- Section 3.1 presents the proposed solution for the problem explored in 1.2.
- Section 3.2 presents how the work done in this thesis will be evaluated.
- Section 3.3 presents the planned working schedule for the duration of this thesis.

3.1 Proposed Solution

The work to be conducted in the context of this dissertation is highly related to the browser-to-browser communication framework, Legion, there is the need to understand the technologies it uses, such as: (i) WebRTC, for supporting the direct communication technology for browsers; (ii) CRDTs, the conflict-free replicated data types that guarantee eventual convergence of replicas; (iii) a wide knowledge of peer-to-peer networks.

The work to be developed will extend Legion to achieve the following objectives. The first objective is to extend the framework to support additional server side storage alternatives. This can be done by integrating with some well known distributed storage systems. This will allow to persistently store the data shared among clients. Additionally, this can also be useful to allow clients that cannot use WebRTC to interact with clients that leverage Legion. This has also potential to allow legacy clients to continue accessing the same data, while Legion enriched clients can benefit from direct communication with similar clients, all in a (mostly) transparent way for programmers, and without the need for special actions from users.

The discussed distributed storage systems to be integrated with Legion to showcase a wide variety of data models, which will create the opportunity to study, in practice, the implications of different data models in the interaction with Legion (that internally heavily relies on CRDTs). We plan to integrate the following storage systems with Legion:

Riak is a distributed storage system that uses CRDTs at its core to achieve eventual convergence of replicas. It supports the following data types: (i) counters that are allowed to be incremented or decremented; (ii) flags that can be enabled or disabled; (iii) sets as collections of binary values; (iv) registers as binary values; (v) maps as a collection of fields that supports the nesting of multiple data types.

Antidote is a distributed storage system built on top of the Riak core module, and it uses CRDTs to achieve eventual convergence of replicas. Antidote also supports additional functionalities, such as causal and weakly consistent forms of transactions that enforce atomicity among a set of write operations.

Cassandra represents internal data with tables, but not in a SQL way. A table in Cassandra is a distributed multi dimensional map indexed by a key. Operations under a row are atomic per replica and Columns are grouped into sets called column families.

For integrating each one of these storage systems with Legion, we need to address the following two main challenges:

- Make Legion's internal data types and the storage systems' internal data types compatible.

  Legion internally uses CRDTs to materialize objects and propagate state and operations between clients. Riak and Antidote also use CRDTs, although their implementation is different. To be able to propagate updates between Legion and these storage systems, it will be necessary to create an adapter between them. Cassandra represents data internally through tables and keys, so we will need to study how to store CRDTs in Cassandra(i.e., how to provide a mapping of data types both ways, or which, possibly new, CRDTs are needed to represent Cassandra's data types), including how to better encode relevant metadata to support the operations of CRDTs in Legion.

- Guaranteeing synchronization (and correctness) between clients when updates are done directly to the storage system when an update is propagated through the Legion infrastructure.

  The current version of Legion already supports synchronization with Google Drive Realtime. In this case it was simple to make the synchronization between Legion and Google Drive Realtime, because Google Drive Realtime offers a mechanism that allows to access a specific version of an object. Thus, this mechanism could be used to compute changes executed in the storage system that had not been seen by Legion clients. The Realtime API also offers a *update* method, where a new state can be set to all objects (their contents) as one single operation, executing atomically on the server. This method accepts a revision number and internally solves conflicts which could have risen, and thus, facilitated the implementation. In the specific case of Cassandra, Riak, and Antidote such mechanism is not available. To address this problem, the following challenges will be addressed:

1. Propagate updates that were executed on Legion to the storage system. A possible approach would be to keep a list of operations and translate operations between systems.
2. Propagate updates that were executed on the storage system to Legion. From each of the storage systems:
   (i) Riak does not offer a mechanism to detect differences, so it will be necessary to identify them, in order to propagate them. One way of doing this is to keep the last synchronized version and resort to a diff algorithm to find the modifications. This version can be kept either in client or server side. Client side may be more unstable, as clients can crash or depart the system frequently. Another solution worth exploring is to identify the modifications based on a stored summary of metadata in the storage system.
   (ii) Antidote is similar to Riak, but it might be slightly simpler to access internal system information because there are some API that expose the internal state of CRDTs stored in Legion(i.e. relevant with data).
   (iii) Cassandra's data model diverges a lot from Legion's. The challenge will be mapping the data models between systems. A possible solution might be keeping two versions of each object. One with data only, and another one that includes metadata, in order to be able to compute differences between these two versions.

The second objective to be tackled in the context of this dissertation is to bring the causal consistency that can be found in some distributed storage systems to the client side. In particular, causal consistency is available in the Antidote storage system, meaning that no client observes a state of the data store that omits relevant causal dependencies.

In order to have causal consistency, it is necessary to guarantee that an update on an object is not propagated before an update on another object from which the first update is causally dependent. In Antidote, each update includes its causal dependencies. It will be necessary to implement a similar mechanism in Legion. This solution will need the following steps: (i) develop in Legion a similar mechanism for the updates inside the framework; (ii) propagate causal dependency information between Legion and Antidote in an efficient way.

3.2 Evaluation

We plan to evaluate the proposed solutions as follows:

- Regarding the mechanisms to integrate with the different storage systems, we will assess the cost of synchronization by measuring the latency, bandwidth and CPU utilization induced by the synchronization process.
- Regarding the mechanisms for bringing causal consistency to mobile nodes, we will measure the overhead introduced in communication and storage to keep the necessary metadata to enforce causality. We will also measure how this overhead impacts the latency of communication and the CPU usage.

3.3 Work Plan

We now describe the workplan for this work. Table 3.1 shows the start and end dates of the planned tasks, that we detail next:

Phase	Start Date	End Date
Design	February 29	March 28
Implementation 1	March 21	May 9
Antidote Integration	March 21	April 11
Riak Integration	April 4	April 25
Cassandra Integration	April 18	May 9
Evaluation 1	May 2	May 23
Implementation 2	May 23	June 20
Evaluation 2	June 13	July 4
Writing	June 6	September 19

Table 3.1: Schedule

Design of Integration Module This task comprises the design of the modules to integrate Legion with each of the planned storage systems. The task will overlap with the implementation of these modules, as we expect to refine our initial design with feedback from implementation. Additionally, although we expect to be able to re-use some of the design decision from one module to the other, there will be some difference that will need to be addressed.

Implementation of Integration Modules This task will comprise the implementation of the integration modules of Legion with Antidote, Riak and Cassandra. The following tasks may partially overlap over the duration of the work:

- Antidote Integration - Integrate the Antidote storage system with Legion.

- Riak Integration - Integrate the Riak storage system with Legion.
- Cassandra Integration - Integrate the Cassandra storage system with Legion.

As mentioned in 3.1, these goals will have similar challenges related to data model between systems and synchronization.

Evaluation of the Integration Modules is the evaluation of the integration of these systems with Legion, in order to determine how performant is the final solution, and compare between them.

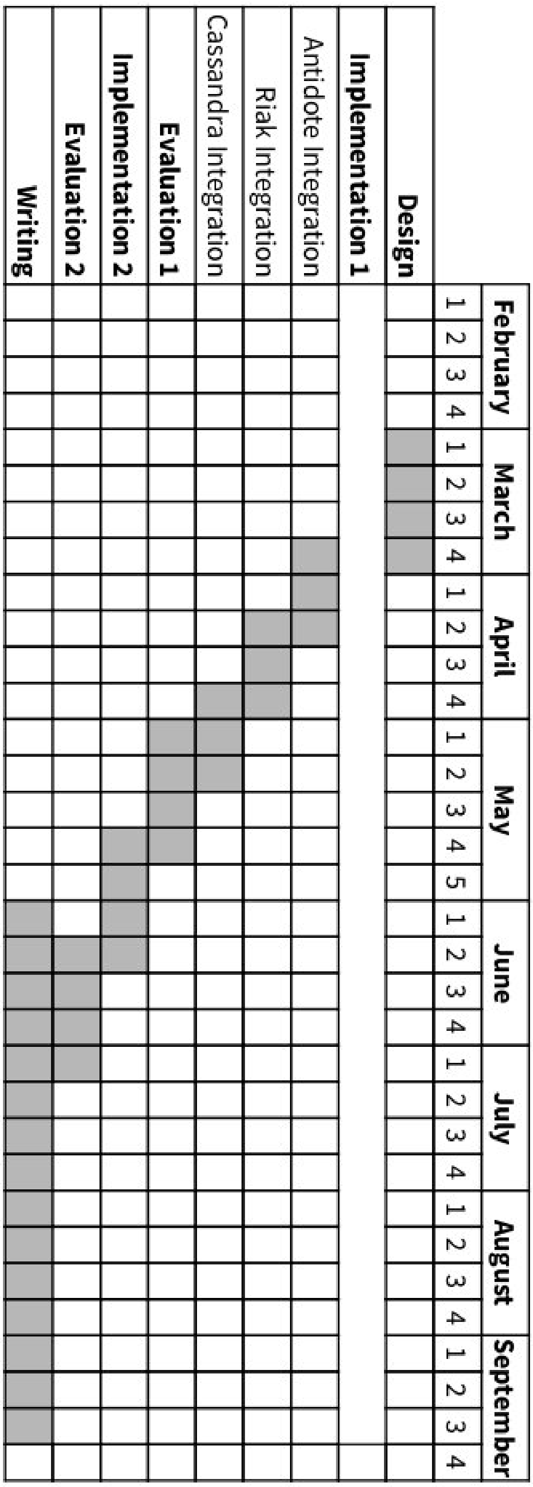
Design for adding causal consistency to Legion This task will design the algorithms that need to be used to support causality across objects in Legion.

Implementation of causal consistency In this task, we will modify Legion to integrate the algorithms proposed.

Evaluation of causal consistency In this task we will evaluate the proposed algorithm, as discussed in section 3.2.

Writing In this task we will write the dissertation where we will report the work performed. Additionally, we expect to write a paper reporting the main contributions of this work.

Figure 3.1 depicts a Gantt chart presenting a summary of the described schedule, better showing how dates overlap.



	February				March				April				May					June				July				August				September			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	5	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Design					▨	▨	▨	▨																									
Implementation 1																																	
Antidote Integration								▨	▨	▨																							
Riak Integration										▨	▨	▨																					
Cassandra Integration												▨	▨	▨																			
Evaluation 1													▨	▨	▨	▨																	
Implementation 2																▨	▨	▨	▨														
Evaluation 2																			▨	▨	▨	▨											
Writing																		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	

Figure 3.1: Schedule of activities

Bibliography

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. “Exploiting atomic broadcast in replicated databases”. In: *Euro-Par’97 Parallel Processing*. Springer, 1997, pp. 496–503.

[2] S. Almeida, J. Leitão, and L. Rodrigues. “ChainReaction: a causal+ consistent datastore based on chain replication”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98.

[3] P. Bailis and A. Ghodsi. “Eventual Consistency Today: Limitations, Extensions, and Beyond”. In: *Commun. ACM* 56.5 (May 2013), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2447976.2447992. URL: http://doi.acm.org/10.1145/2447976.2447992.

[4] A. Bergkvist, D Burnett, and C. Jennings. “A. Narayanan," WebRTC 1.0: Real-time Communication Between Browsers”. In: *World Wide Web Consortium WD WD-webrtc-20120821* (2012).

[5] J. L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290858, 9781617290855.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. “Bigtable: A distributed storage system for structured data”. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[7] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[8] 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”. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[9] S. Gilbert and N. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”. In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[10] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. "The phi; accrual failure detector". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on.* 2004, pp. 66–78. DOI: 10.1109/RELDIS.2004.1353004.

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* ACM. 1997, pp. 654–663.

[12] R. Klophaus. "Riak Core: Building Distributed Applications Without Shared State". In: *ACM SIGPLAN Commercial Users of Functional Programming.* CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900176. URL: http://doi.acm.org/10.1145/1900160.1900176.

[13] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[14] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[15] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño Martínez, and J. E. Armendáriz-Iñigo. "Snapshot Isolation and Integrity Constraints in Replicated Databases". In: *ACM Trans. Database Syst.* 34.2 (July 2009), 11:1–11:49. ISSN: 0362-5915. DOI: 10.1145/1538909.1538913. URL: http://doi.acm.org/10.1145/1538909.1538913.

[16] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. *Peer-to-peer computing.* 2002.

[17] J. Paiva and L. Rodrigues. "Policies for Efficient Data Replication in P2P Systems". In: *Parallel and Distributed Systems (ICPADS), 2013 International Conference on.* 2013, pp. 404–411. DOI: 10.1109/ICPADS.2013.63.

[18] S Paul and Z Fei. "Distributed caching with centralized control". In: *Computer Communications* 24.2 (2001), pp. 256 –268. ISSN: 0140-3664. DOI: http://dx.doi.org/10.1016/S0140-3664(00)00322-4. URL: http://www.sciencedirect.com/science/article/pii/S0140366400003224.

[19] F Schenider. "Replication management using the state-machine approach, Distributed Systems". In: *Ed. Sape Mullender,* (1993), pp. 169–198.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[21] D. Skeen. *A quorum-based commit protocol.* Tech. rep. Cornell University, 1982.

[22] M. Szydlo. “Merkle tree traversal in log space and time”. In: *Advances in Cryptology-EUROCRYPT 2004*. Springer. 2004, pp. 541–554.

Webography

[23] *Antidote README*. Accessed: 2016-02-01. URL: https://github.com/SyncFree/antidote.

[24] *Memcached About*. Accessed: 2016-01-26. URL: http://memcached.org/about.

[25] *WebRTC in the real world: STUN, TURN and signaling*. Accessed: 2016-02-08. URL: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.