Albert van der Linde

Lic.

Huge-scale Collaborative Systems

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, 2015

Abstract

An increasing number of web applications run totally or partially in the client machines - from collaborative editing tools to multi-user games. This allows to reduce latency when clients are close to each other and to minimize the load on servers. Novel implementation techniques, such as WebRTC, allows to address networking problems that previously prevented these systems to be deployed in practice.

The goal of this work is to develop a system and supporting algorithms to allow clients to efficiently communicate with each other in a peer-to-peer fashion, while tightly integrating with a cloud backend. The proposed approach can be seen as an extension of the cloud based system to support lower latency, better fault tolerance to server failures, and support for disconnected operation. This work will not only explore novel hybrid peer-to-peer-cloud designs but also new technology which we expect will be key in enabling the adoption of such a system design.

Keywords: client-based web applications; peer-to-peer; replication, CRDTs.

RESUMO

Um número crescente de aplicações web executam total ou parcialmente nas máquinas do cliente - a partir de ferramentas de edição colaborativa para jogos multi-usuário. Isto permite reduzir a latência entre clientes quando estão próximos uns dos outros e minimizar a carga nos servidores. Novas técnicas de implementação, tais como WebRTC, permitem abordar problemas de rede que anteriormente impediam esses sistemas a serem implementados na prática.

O objetivo deste trabalho é desenvolver um sistema e algoritmos de apoio para permitir que os clientes comuniquem de forma eficiente uns com os outros, numa forma *peer-to-peer*, enquanto fortemente integrados com um sistema baseado em nuvem. A abordagem proposta pode ser vista como uma extensão do sistema baseado em nuvem para suportar menor latência, melhor tolerância a falhas de falhas de servidor e suporte a operação desconectada. Este trabalho não só irá explorar novos desenhos híbridos peer-to-peer-nuvem, mas também uma nova tecnologia que esperamos que será fundamental para permitir a adopção de uma tal concepção do sistema.

Palavras-chave: aplicações-cliente baseadas em web; *peer-to-peer*; replicação, *CRDTs*.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 Context

Recent changes in web browsers and client devices have enabled programmers to build more powerful and interesting web applications. From collaborative editing tools to multi-user games, an increasing number of web applications run totally, or partially, on client machines. Recent frameworks and developer APIs for browser applications allow for easy development and deployment of such applications. The rise of HTML5 has greatly improved the control a developer has over the browser, including aspects such as multi-threading and local storage, which previously only were available to desktop applications. This creates the opportunity for richer client browser applications to be developed.

Currently, even when web applications run in the client, they typically access information stored in servers and all communication among clients goes through the servers - e.g. all interaction in Google Docs is mediated by a server even when clients are nearby. The growth in browser support for WebRTC[32] makes a good promise for enabling client browser peer-to-peer architectures.

Peer-to-peer systems have gained a lot of attention in research and have been widely adopted by industry. Peer-to-peer has been a medium for file-sharing, streaming media, telephony, and even volunteer computing. Peer-to-peer technologies have also been incorporated into other systems, for example Amazon's Dynamo[7], a NoSQL database that uses DHTs, (discussed in Section 2.1.1) which distributes state over nodes as if it were a hash table. Primitives are similar to any hash table, but providing very efficient lookups in a distributed system.

Though leveraging application logic to the client side in browsers and the use of peer-to-peer architectures are widely explored by academia and industry, there is currently a lack of use of peer-to-peer techniques to improve web applications running in the clients. Using recent browser based technologies enriched with peer-to-peer support, powerful applications can be conceived. Collaborative editing is a good example of an application that can be greatly improved by exploring this design space. Current collaborative editing applications and developer frameworks are based on a client-server model, while currently only a limited number of peer-to-peer applications and frameworks exist that support peer-to-peer support of the browser level.

1.2 Motivation

Real-time collaborative editing tools, where users can edit the same file simultaneously, have been deployed with great client adoption. Existing frameworks that make this possible currently rely on a client-server model. Servers act as the data storage system while also being used for access control and to expose, or resolve, divergence on concurrently edited objects.

Current frameworks for collaborative editing on the web do not allow for direct client to client, i.e., peer-to-peer, communication, and all user interactions are mediated by the central component. The main reasons for this are the following. First, limitations to peer-to-peer networking made it hard to explore peer-to-peer communications in the past. Second, relying on a central component simplified the management of data and notifications to clients. However, a number of recent developments are challenging these problems.

Existing peer-to-peer technologies and the recent development of HTML5, in particular WebRTC, make it possible to design a system that delivers the foundations for collaborative editing applications on the browser exploiting peer-to-peer communications. Improvements on local storage and threading support make possible for objects to be cached or even permanently stored on the client, possibly removing the need for continuous access to the server, as to reduce access latency and bandwith[33], server load or even reducing trust on the service provider[10, 34].

As web based services try to improve latency and availability, typically data becomes geographically distributed among different data centres using partitioning and replication. Centralized architectures resolve state divergence on the server

while the use of CRDTs, Convergent or Commutative Replicated Data Types, allows for automatic reconciliation of state using peer-to-peer systems. Operations on CRDTs immediately execute locally, and, unaffected by network latency, faults, or disconnections, eventually converge towards a common, single, state.

Using peer-to-peer for collaborative editing or other applications that have high user interaction in real-time can greatly reduce the load on the server and reduce client-to-client latency. Ideally, scalability can be greatly increased and user experience improved.

1.3 Main expected Contributions

The work planned, detailed in Section 3.2, builds upon the premise that existing peer-to-peer technologies can be used as a foundation for real-time collaborative editing and that the recent HTML5 APIs allows for such a system to be built for web browsers. As the work focuses on web-development, the idea is to create a framework that supports the creation of applications in the browser without the need, by end users, to install any kind of software or browser plugins. We expect this to be key in the adoption of the framework by the developer community.

The main expected contributions are, condensed in the form of a Javascript web-application development framework, the following:

- A logical peer-to-peer connectivity layer between end users' browsers, supporting the choice between various overlay networks (tree, random graph, DHT).
- A CRDT API on top of the peer-to-peer overlays, enabling real-time concurrent editing of objects in a distributed fashion.
- Integration with existing cloud backends, in a way that the developed solution can be seen as an extension of those systems, enabling the construction of web based online real-time collaborative editing tools.
- Example applications similar to existing frameworks, built on top of previous items, improving client-to-client latency and scalability.

The planned framework also aims at providing fundamental security properties, various signalling methods (Section 3.1.1), and easy deployment.

1.4 Document Organization

The remainder of this document is organized as follows:

Chapter 2 describes related work. Existing P2P technologies and web based data storage are addressed. Collaborative editing solutions, and architectures, are also explored.

Chapter 3 describes the work plan for the elaboration of this thesis. It also describes the technological basis over which this work will be developed.

CHAPTER 2

RELATED WORK

This thesis addresses the challenges of real-time collaboration leveraging on peer-to-peer communication directly on the browser. Therefore, various aspects have to be considered. The following sections cover the main aspects of these fields, in particular:
In **Section 2.1** existing peer-to-peer technologies are studied and compared, with special interest on overlay networks.
In **Section 2.2** web based service providers are discussed, in particular addressing the challenge of storing and accessing data.
In **Section 2.3** the field of collaborative editing is explored, with emphasis on real-time collaborative editing.

2.1 Peer-to-peer systems

Peer-to-peer systems typically have a high degree of decentralisation, leveraging the use of resources from the server to the client. In other words, each peer implements both server and client functionality to distribute bandwidth, computation, and storage across all participants of a distributed system[28]. This is achieved by allocating state and tasks among peers with few, if any, dedicated peers.

Nodes are initially introduced to the system and typically little or no manual configuration is needed to maintain connectivity[1]. Participating nodes generally

[1]To consider a network as connected, there should be at least one path from each node to all other nodes.

belong to independent individuals who voluntarily join the system and are not controlled by a single organisation.

Peer-to-peer systems are interesting due to their low barrier to deployment, its organic growth (as more nodes join, more resources are available), resilience to faults/malicious attacks, and the abundance/diversity of systems.

Popular peer-to-peer applications include sharing and distribution of files, streaming media, telephony, and volunteer computing. Peer-to-peer technologies were also used to create a diversity of other applications, for example Amazon Dynamo[7] which is a storage system, which internally heavily relies on DHTs, demonstrating the benefits of leveraging peer-to-peer architectures.

It is important to note that the network topology of the underlying network has a high impact on the performance of peer-to-peer services and applications. For client nodes to be able to cooperate they need to be aware of the underlying network. The typical approach is to create a logical network of nodes on top of the underlying network, called an overlay network.

Thus, in order to develop a peer-to-peer distributed service, it is of paramount relevance to study the mechanisms for creating and managing overlay networks that match the application requirements.

2.1.1 Overlay Networks

An overlay network is a logical network of nodes, built on top of another network. Links between nodes in the overlay network are virtual links, being composed of various links of the underlying network. In peer-to-peer systems, overlay networks are constructed on top of the internet, each link being a connection between two peers.

To achieve an efficient and robust method of delivering data through a peer-to-peer technique an adequate overlay network is needed. When building an application, the programmer must first decide on the overlay to deploy and use, choosing between degrees of centralisation as well as on structured vs unstructured designs.

Degree of centralization: Peer-to-peer networks can be classified by their use of centralised components.

Partly centralized networks leverage system component to dedicated nodes or a central server to control and index available resources. These centralised components are used to coordinate system connections, facilitate the establishment of communication patters, and coordinate node

co-operation.

As an example, when client nodes wants to execute a specific query, only the central component is contacted, which in turn can return the set of nodes that match the query. These systems are relatively simple to build but come with the drawback of a potential single point of failure and bottleneck. Therefore, this design can not be as resilient and scalable as a fully decentralized system. Examples include Napster[21], Bittorrent using trackers[4], BOINC[1], and Skype[2].

Decentralized networks avoid the use of dedicated nodes. All network and communication management is done locally by each participating node, using light coordination mechanisms. This way a singe bottleneck and point of failure is avoided, increasing potential for scalability and resilience. In this type of architecture, nevertheless, a few selected nodes may act as supernodes, as to leverage potential higher CPU or bandwidth available, gaining additional responsibilities such as storing state or even becoming the entry point for new nodes. As queries cannot be sent to and executed by a central component, typically when using unstructured overlays they have to be flooded through the network or routed through the network when using a structured overlay. Example protocols include Gnutella[27], Gossip[3], and Kazaa[18].

Structured vs Unstructured: choosing between structured and unstructured overlays depends mostly on the usefulness of key-based routing algorithms[2] and the amount of churn[3] that the system is expected to be exposed to.

Structured overlays: Each node gets an identifier which determines its position in the overlay structure. Identifiers are chosen in a way that peers are usually uniformly distributed at random over the key space. This allows to create a structure similar to a hash table, named DHT, distributed hast table. This type of overlay graph is typically chosen when efficient (logarithmic complexity) key-based routing is required. Structured overlays typically use more resources to maintain the overlay, but in return get efficient queries at the cost of poor performance when churn is high, in fact, churn is not handled well at all.

[2]Key-based routing is a lookup method, used in conjunction with distributed hash tables, that enables to find the node that has the closest identifier to the key being searched.

[3]Churn is the participant turnover in the network meaning, the amount of nodes joining and leaving the system per unit of time.

Unstructured overlays: There is no particular structure in the network links and queries are usually done by flooding the network. Each peer keeps a local index of its own resources and, in some cases, the resources of its neighbours. The connected peers are designated as a partial view. This is a (small) fraction of all peers in the system with whom that participant can interact directly. Ideally, the size of such partial views should grow logarithmically with the number of participants in the system. Queries are typically disseminated among the connected peers. To ensure that a query returns all possible results, the query must be disseminated to all participants.

Maintaining data: In partially centralized systems data is typically stored at the inserting and downloading nodes. The central component maintains metadata, i.e., an index, on the stored data including where it is located.

In unstructured systems data is also stored on submitting and downloading nodes but to locate data typically the queries are flooded. For faster searches, nodes can distribute metadata amongst neighbours.

In structured overlays distributed state is maintained using distributed hash tables. Primitives are similar to any hash table, and easily implemented when a key-based routing function is available. On high churn it becomes very inefficient to store large amounts of data at peers responsible for the keys, therefore indirection pointers are commonly used, pointing to the node(or nodes) that effectively holds the data.

Coordination: In partially centralized systems the central component can trivially achieve coordination.

In unstructured overlays, epidemic techniques are typically used because of their simplicity and robustness to churn. Information tends to propagate slowly though and scaling to large overlays is costly. Spanning trees[4] can increase efficiency but maintaining the tree structure adds maintenance costs.

In structured overlays, key-based routing trees are the basis for potentially large sub-groups within the overlay, enabling fast coordination and good efficiency.

[4] A spanning tree of a graph is a tree connecting all nodes in the graph.

2.1.2 Example peer-to-peer overlay networks

Chord [30] is a distributed lookup protocol that enables peer-to-peer systems to efficiently locate nodes that store a particular data item. It only offers one primitive: given a key, return the nodes responsible for the key. Keys are distributed over the nodes using consistent hashing and replicated over succeeding nodes. Nodes typically store their successor nodes, forming an ordered ring (considering node's identifiers), making it easy to reason about the overlay structure. For fault-tolerance a list of successor nodes is kept and for efficient lookups a finger table, shortcuts to nodes over the graph, is used to jump over nodes in the graph.

Gnutella [27] is a decentralized peer-to-peer file sharing protocol. When a node is bootstrapping to the network, it tries to connect to the nodes it was shipped with, as well as nodes it receives from other clients. It connects to only a small amount of nodes, locally caching the addresses it has not yet tried, discarding the addresses that are invalid.

Queries are issued and flooded from the client to all actively connected nodes, the query is then forwarded to any nodes they know about. Forwarding ends if the request can be answered of the Time-To-Live field ends. The protocol in theory doesn't scale well, as each query increases network traffic exponentially each hop, while being very unreliable as each node is a regular computer user, constantly connecting and disconnecting.

A revised version of the gnutella protocol is a network made of leaf nodes and ultra peers. Each leaf node is connected to a small number of ultra peers, while ultra peers connect to many leaf nodes and ultra peers. Leaf nodes send a table containing hashed keywords to their ultra peers, which merges all received tables. These tables are distributed amongst ultra peer neighbours and used for query routing, by hashing the query keywords and trying to match the tables.

Cyclon [31] is a membership management framework for large P2P overlays. The used membership protocol maintains a fixed length partial view managed through a cyclic strategy. This partial view is updated every T time units by each node though an operation called shuffle. In a shuffle, a node selects the oldest node in its partial view and exchanges some elements of its local partial view with it. When nodes initially join the overlay a random walk is used, ensuring that the in-degree of all nodes remains unchanged. This work achieves an overlay topology with low diameter and low clustering

coefficient with highly symmetric node degrees and high resilience to node failures.

Scamp [12] is a membership management framework for large P2P overlays. The Scamp protocol maintains two views, a PartialView to send gossip messages and an InView from which they receive messages. The PartialView is not of fixed length, it grows to a size logarithmic in scale to the number of nodes in the network without any node being aware of this number. The protocol uses a reactive strategy, in the sense that the partial views are updated when nodes join or leave the system. Periodically nodes send heartbeat messages as to detect and recover from isolation due to failures. Not receiving any heartbeats allows the node to assume that it is isolated, triggering the join mechanism to effectively rejoin the overlay.

HyParView [19], Hybrid Partial View, is a reliable gossip-based broadcast protocol that maintains a small symmetric Active View(managed through a reactive strategy) for broadcasts and a larger Passive View (managed through a cyclic strategy) to recover timely from faults. Both strategies are very similar to Scamp and Cyclon. TCP is used as a reliable transport and to detect failures, being feasible as the Active View is small. Even with a small ActiveView, improving protocol efficiency as less network traffic is required for flooding messages, very good results in reliability are obtained. This work shows the importance of each reactive and cyclic strategies to maintain views of the network, and that the use of a reliable transport mechanism, like TCP, to timely encounter failures, can greatly improve results.

Using a structured network overlay as Chord, a really high network efficiency can be achieved as all request are routed directly to the right nodes. Unstructured network overlays typically have to flood the network, reducing efficiency, but create tolerance to network churn. Declaring some nodes as dedicated to the network, as done by Gnutella, can greatly reduce network traffic. Cyclon, Scamp, and HyParView each show the importance of reactive and cyclic strategies for maintaining partial views as well as the combination of both.

2.2 Data Storage

Web based services store client data on geographically distributed data centers, trying to provide good latency, bandwidth, and availability for interactions with the data. Typically replication and distribution of state across geographically

separated data centres is required to ensure low latency and fault tolerance. A problem arises, formally captured by the CAP theorem[14], which states that it is impossible for a distributed computer system to simultaneously provide all three of the following: **C**onsistency (all nodes see the same data at the same time), **A**vailability (every request receives a response about whether it succeeded or not) and **P**artition tolerance (the system continues to operate despite arbitrary message loss or partial failure of the system or unavailability due to network partitions). Unfortunately, due to how the internet works, partitions due to network or node failures are part of our lives so the question is to ask which to sacrifice, **C**onsistency or **A**vailability?

Strong Consistency. A system is said to provide strong consistency if all accesses to data are seen by all clients in the same order (sequentially). A distributed system providing Strong Consistency will come to a halt if nodes become network-partitioned. It is easy to understand that two nodes cannot decide on a value if they cannot reach one another. **C**onsistency can thus be maintained but the system will sacrifice **A**vailability.

Eventual Consistency is a consistency model used in distributed computing systems to achieve high availability which informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the value of the last update. This allows these systems to, even during network partitions, always serve read and write operations over data. Eventual consistency may not be enough, one example: user A writes to a page and user B answers, due to network latencies user C sees B's answer before A's initial post. This shows that while this consistency model is correct, it can lead to confusion.

Causal Consistency. A system provides causal consistency if potentially causally related operations are seen by every node of the system in an order that respects these causal dependencies. Concurrent writes (i.e. write operations that are not causally related) may be seen in different orders by different nodes. When a node performs a read followed later by a write, even on a different objects, 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. Also, even two write operations performed by the same node are defined to be causally related, in the order they were performed. Intuitively, returning on our previous example, such a system would never show B's updates before A's as they are causally related.

Using eventual and causal consistency (i.e. not strong consistency) usually comes with a cost: state divergence. To address state divergence, conflict resolution techniques such as the ones discussed in section 2.2.1 must be used.

One way to avoid state divergence, as achieved in Yahoo!'s PNUTS[5], is to funnel all state changing operations through a per record chosen primary site and lazily propagating to replicating nodes. This increases latency and reads can return stale data, but data exposed to users is consistent. The problem of this approach is availability as the primary site is a potential single point of failure.

When multiple nodes can write to the same data, object versioning control has to be done using for example logical clocks or version vectors[17, 24] and conflict resolution techniques have to be applied.

2.2.1 Conflict resolution techniques

Relaxing from a strong consistency model to a weaker model such as causal consistency, minimizes the amount of required of synchronization among replicas at the expense of having to deal with state divergence. To do so one resorts to conflict resolution techniques. Common conflict resolution techniques include:

Last Writer Wins: is the idea that the last write based on a node's system clock will overwrite an older one. Using a single server this is trivial to implement but when clocks are out of synch when writing on multiple nodes, choosing a write between concurrent writes is not trivial at all and can lead to lost updates.

Programatic Merge: letting the programmer decide what to do when conflicts arise. As an example, an application maintaining shopping carts can choose to merge the conflicting versions by returning a single unified cart. This conflict resolution technique requires replicas to be instrumented with a merge procedure, or alternatively, requires replicas to expose diverging states to the client application which then reconciles and writes a new value.

Commutative Operations: If all operations are commutative, conflicts can easily be solved. Independently of the order, when all operations have been received (and applied), the final outcome will be the same. An always incrementing counter, where each operation is uniquely marked by the writing node, is an easy example: independently of the order of operations, the final result will eventually be the same. Commonly used commutative operation techniques are:

OT, Operational Transformation. The idea of OT is to transform the parameters of an operation to the effects of previously executed concurrent operations, so that the outcome is always consistent. As an example, a text document contains "abc" and there are two concurrently editing users. One user inserts 'x' at position 0 and the other deletes 'c' from position 2. If both execute their operation and later receive the operation of the other (due to network latency), the final states diverge to 'xac' and 'xab'. Transforming the operations solves this problem, the delete is transformed to increment one position and the insert can remain the same. Both outcomes become 'xab', independently of the order in which operations are applied.

Operational Transformation has been extensively studied, especially by the concurrent editing community, and many OT algorithms have been proposed. However, it was demonstrated that most OT algorithms proposed for a decentralized OT architecture are incorrect[23]. It is believed that designing data types for commutativity is both cleaner and simpler[29].

CRDT, Convergent or Commutative Replicated Data Types. CRDTs are replicated data types that guarantee eventual consistency while being very scalable and fault-tolerant. An example is a replicated counter, which converges because increment and decrement operations commute. No synchronisation is required to ensure convergence, so updates always execute locally, and immediately, unaffected by network latency, faults, or disconnections. CRDTs can typically divided in two classes:

CvRDT, state-based Convergent Replicated Data Type. In the state-based class, the successive states of an object should form a monotonic semilattice[5] and replica merge computes a least upper bound. In other words, when merging diverging states the end result must be equal. State-based CRDTs require only eventual communication between pairs of replicas.

CmRDT, operation-based Commutative Replicated Data Type. In the operation-based class, concurrent operations commute. Operation-based replication requires reliable broadcast communication with delivery in a well-defined delivery order, such as a causal order between operations.

[5] An idempotent and commutative system that grows only in one direction.

Both classes of CRDTs are guaranteed to eventually converge towards a common, single, state (i.e. when all updates are received by all participating nodes). Practical use of CRDTs shows that they tend to become inefficient over time, as tombstones accumulate and internal data structures become unbalanced[29]. Garbage collection can be performed using a weak form of synchronisation, outside of the critical path of client-level operations.

2.2.2 Example data storage systems

Spanner[6] is a system providing strong consistency which uses the Paxos algorithm as part of its operation to replicate data across hundreds of data centers. It also makes heavy use of hardware-assisted time synchronization using GPS clocks and atomic clocks to ensure global consistency.

One server replica is elected as the Paxos leader for a replica group, that leader is the entry point for all transactional activity for the group. Groups may include read-only replicas, which do not vote in the Paxos algorithm and cannot become group leaders.

Furthermore all transactions in Spanner are globally ordered as they are assigned a hardware assisted commit timestamp. These timestamps are used to provide multi-versioned consistent reads without the need for taking locks. A global safe timestamp is used to ensure that reads at the timestamp can run at any replica and never block behind running transactions.

Spanner thus has very strong consistency and timestamp semantics, providing scalable data storage and synchronous replication.

Dynamo[7] is a highly-available key-value storage system. To achieve high availability, consistency is sacrificed using object versioning and application-assisted conflict resolution, exposing data consistency issues and reconciliation logic to the developers.

Data is partitioned and replicated using consistent hashing and vector clocks are used for object versioning. Dynamo uses a gossip based failure detection and membership protocol. This removes the need for manual configuration creating a completely decentralized system, ensuring that adding and removing nodes can be done without any manual effort. Each node is aware of the data being hosted at its peers. In contrast to other DHT systems, each node actively gossips the full routing table with other nodes in the system.

This model works well in their expected scenario of a couple of hundred of nodes, scaling this design to a higher number of nodes can be troublesome as the routing table increases with the number of nodes in the system.

Gemini and its RedBlue consistency[20] build on the premise that while a system can be leveraged to use eventual consistency for higher performance, strong consistency may be necessary to ensure correctness of the applications.

RedBlue consistency labels operations as red or blue. Blue operations are to be fast (eventually consistent) while red operations are slow(strongly consistent). Blue is used when possible and red when needed. Gemini is a coordination infrastructure implementing RedBlue consistency. Experimental results show that RedBlue consistency provides substantial performance while being able to maintain application invariants, the downside is that operations have to be modified and correctly labelled.

Riak[16] is a distributed NoSQL key-value data store that supports high availability by giving the possibility between strong and eventual consistency, using quorum read and write requests and multi-version concurrency control with vector clocks. Eventual consistency in Riak uses CRDTs at its core, including counters, sets, flags, registers, and maps. Partitioning and replication is done via consistent hashing using a masterless approach, thus providing fault-tolerance and scalability. The built-in functions determine how replicas distribute the data evenly, making it easy for the developer to scale out to more nodes.

SwiftCloud[33] is an eventual consistency data storage system with low latency that relies on CRDTs to maintain client caches. The main focus of this work is to integrate client and server-side storage. Responsiveness is improved when accessed objects are locally available at the cache, this allows for disconnected operation.

In the presence of infrastructure faults, a client-assisted failover solution allows client execution to resume immediately and seamlessly access consistent snapshots without blocking. Additionally, the system supports mergeable and strongly consistent transactions that target either client or server replicas and provide access to causally-consistent snapshots efficiently.

Systems like Spanner have been designed to provide strong consistency on geographically distributed data-centers. These systems use very complicated algorithms or specialised underlying hardware and are not trivial to deploy. Systems supporting weaker consistency models have been developed, like Dynamo, that support writes on different clients increasing scalability and fault tolerance, but need a way to address state divergence. A system like Gemini that supports both eventual and strong consistency can be used to have the best of both. It can be very difficult to reason in detail on such a system, especially on what has to be strong or what can be eventually consistent. The use of CRDTs, like in Riak and SwiftCloud, can greatly improve latency as all updates can always execute and merging diverging state isn't an issue, as data converges to a single final consistent state.

2.3 Collaborative Editing

A collaborative editor is a piece of software that allows several people to edit files using different client-devices, working together through individual contributions. Collaborative editing can be divided in two types: real-time and non-real-time. In real-time collaborative editing systems users can edit the same file simultaneously while in non-real-time collaborative editing systems editing the same file at the same time is not allowed.

In real-time collaborative editing the main challenge is to figure out how to apply edits from remote users, who produced these edits on versions of the document that possibly never existed locally, and that can potentially conflict with the user's own edits. Users may write on previously decided sub-parts of the document, facilitating merges, or even work together on the same task.

There are several approaches in creating a collaborative editor. The basic needs for such a system are the possibility for concurrent (possibly in real time) editing of objects while preserving user intent. Some approaches include:

Turn taking where one participant at the time 'has the floor'. This approach lacks in concurrency but is easy to comprehend, preserving user intent.

Locking based techniques, where concurrent editing is trivially possible as users work on different objects. Pessimistic locking introduces delays and optimistic locking introduces problems when the lock is denied or when user edits have to be rolled back to a previous state.

Serialization can be used to specify a total order on all operations. Non-optimistic serialisation delays operations until all preceding operations have been processed while in optimistic serialisation, executing operations on arrival is allowed, but there might be the need to undo/redo operations to repair out-of-order executions(as in version control systems).

Commutative operations can be leveraged to address the challenge of collaborative editing system, by using OT or CRDTs (as described in section 2.2.1) a high degree of concurrency can be achieved while capturing and preserving user intent.

A collaborative editor can be designed using a client-server model. The server ensures synchronization between clients, determining how user operations should affect the server's copy and how to propagate these operations to other clients. Though easy to implement, this approach possibly lacks scalability and can detoriate user experience by increasing latency. A more sophisticated solution is one that does not require a server, while avoiding to resort to locking, and supports any number of users.

Though good enough for non-real-time collaborative editing, to provide the basic needs for a real-time collaborative editor it is easy to see that approaches as turn-taking, locking, and serialisation are insufficient. Besides not allowing real-time concurrent editing of the same data, the coordination algorithms of underlying systems can be unnecessarily complicated while scalability and fault tolerance are non trivial to reason about. Using commutative operations is thus widely accepted as the *de facto* solution.

2.3.1 Examples of collaborative editing systems

Etherpad[11] (or Etherpad Lite), is a web-based collaborative real-time editor, allowing authors to simultaneously edit a text document, and see all of the participants' editions in real-time, with the ability to display each author's text in their own color. There is also a chat box in the sidebar to allow direct communication among users. Anyone can create a new collaborative document, known as a "pad". Each pad has its own URL, and anyone who knows this URL can edit the pad and participate in the associated chats.

The software auto-saves the document at regular, short intervals, but participants can permanently save specific versions (checkpoints) at any time. A "time slider" feature allows anyone to explore the history of the pad. Merging

of changes is handled by operational transformation using a Client-Server model.

Dropbox Datastore[8] is an API that allows developers to synchronise structured data easily supporting multiple platforms, offline access, and automatic conflict resolution.

The server doesn't attempt to resolve conflicts. OT-style conflict resolution is done on the client, the server simply serializes requests. Conflict resolution is allowed to be defined by the client, the application created by a developer, by choosing from the following conflict resolution rules: choose remote, choose local, max, min and sum.

A datastore is cached locally once it is opened, allowing for fast access and offline operation. Changes to one datastore are committed independently from another datastore. When data in a datastore is modified, it will automatically synchronize those changes back to Dropbox (i.e., upload local changes and downloading and applying remote modifications).

Google Drive Realtime API[15] is a client-only library that can merely be used in combination with Google servers. The API can be used by developers to implement a real-time application by using its collaborative objects, events and methods. It uses operational transformation to resolve concurrency issues and thus local changes are reflected immediately, while the server transforms changes to the data model so that every collaborator sees the same (final) state. In contrast to Google's own collaborative web applications, such as Google Docs, anonymous users are not permitted and as such using the API requires the end users to have an active Google Drive account.

This API itself is limited to document-based synchronization, such as lists, strings, and key-value maps. It does not specifically support model-based synchronization and complex object graphs. The developer would need to deal with the intricate details of merging model instances while also handling complex situations such as relations to abstract classes or cycles in the object graph.

ShareJS[13] is a server and client library to allow concurrent editing of any kind of content via operational transformation. The server runs on NodeJS and the client works in NodeJS or a web browser. In the local browser, edits are visible immediately. Edits from other people get transformed. ShareJS

currently supports operational transform on plain-text and arbitrary JSON data.

If multiple users submit an operation at the same version on the server, one of the edits is applied directly and the other user's edit is automatically transformed by the server and then applied. In contrast to previous systems ShareJS gives complete control to the developer over both the client and server logic and over the operational transformation protocol.

Jupiter[22] is a tool that supports multiple users for remote collaboration using operational transformation. The synchronisation protocol is not applied directly between the clients as each client synchronises only with the server. The server is thus used to serialise all operations and disseminate those operations to other clients.

Operations are directly executed at the local client site when generated. They are then propagated to the central server which serialises and transforms operations before executing on the server's copy. Finally the transformed operations are broadcast to all other client sites. When receiving an operation from the server a client may transform this operation if needed, and then execute on the local copy.

SPORC[10] is a cloud-based framework for managing group collaboration services. It uses operational transformation to merge diverging state and, as an example, showcases a collaborative editor that allows multiple users to modify a text document simultaneously via their web browsers and see changes made by others in real-time providing an application similar to Google Drive and EtherPad. However, unlike those services, it does not require users to trust on the centralised server. The purpose of the server is to order and store client generated operations. As the server only sees an encrypted history of the operations all application logic is leveraged to the client.

EtherPad is a completely open-source real-time collaborative word processing tool that can freely be hosted on any server while Google's Realtime API and Dropbox's Datastore require the use of the provider's servers. These systems provide the user with a history view for each document. ShareJS shows that operational transformation is possible over JSON data. Jupiter and SPORC emphasise the importance of a central server to serialise editions, though in SPORC the server doesn't ever see the content of the documents while still being able to provide all functionality.

2.4 Sumary

This chapter discussed previous work in the areas related to the development of this dissertation.

In the peer-to-peer context the need for an overlay network has been described, explaining that different application requirements can require different types of overlays. Overlays can generally be described by degree of centralization and structured vs unstructured.

In the data-storage context we talked about leveraging strong consistency with high availability. Different consistency models have been explored and, in the case of eventual consistency, several techniques for conflict resolution have been described.

In the collaborative-editing context, various commonly used approaches have been explored, describing how concurrency is handled in real-time editing in each of them.

In the next chapter the work plan for the elaboration of this thesis will be described, starting off with the technological basis over which this work will be developed.

CHAPTER 3

PROPOSED WORK

This chapter begins by describing existing technologies that will be leveraged as a basis for performing the work proposed in this document. In the second section the proposed solution is presented and in the final section the work plan is described.

3.1 Existing Technologies

This section explains the technological landscape the work will be built upon. Beginning with an overview on HTML5 and its key features that motivate this work. WebRTC is explored in higher detail.
An overview of developer tools and frameworks are then presented, these can be used to build peer-to-peer and collaborative editing applications.

3.1.1 HTML 5

HTML5 is a core technology markup language of the Internet used for structuring and presenting content in the World Wide Web.

HTML5 brings to the Web video and audio tracks without requiring the use of plugins; programmatic access to a resolution-dependent bitmap canvas, which is useful for rendering graphs, game graphics, or other visual images on the fly; native support for scalable vector graphics (SVG) and math (MathML); features to enable accessibility of rich applications; and more. These features are designed to make it easy to include and handle multimedia and graphical content on the web without having to resort to proprietary plugins and APIs. The APIs and Document

Object Model (DOM) are no longer afterthoughts, but are fundamental parts of the HTML5 specification. For this dissertation, interesting APIs include:

WebWorkers. When using scripts in a web page, the page becomes unresponsive while the script is being processed. A web worker is a JavaScript that runs in the background without affecting performance on the page. All major browsers support web workers. A worker can be created by loading a JavaScript file by using `w = new Worker("file.js")`. Information can be exchanged with the worker by implementing the `w.onmessage` and `w.postMessage` methods. Inside a worker most standard JavaScript functions can be executed, as long as no changes are performed over the parent page. This include manipulating the DOM and using the page's objects. Interestingly, web sockets and file IO can be used inside workers while WebRTC, due to its audio and video streams, is disallowed.

Web Storage, commonly referred to as LocalStorage, is a way for web pages to store data via a put/get key-value interface within the client browser. This data persists on the browser and, unlike cookies, is not sent to remote servers. By default, only 5 megabytes are available, and everything is internally stored in string representations.

WebRTC was created for real-time, plugin-free video, audio, and data communication. Real Time Communication is used by many web services, but requires large downloads, the use of native apps, or plugins. These includes Skype, Facebook (which uses Skype), and Google Hangouts (which use the Google Talk plugin). Downloading, installing, and updating plugins can be complex for both the developer and end user. Overall, it's often difficult to persuade people to install plugins,which impacts the adoption of applications with this requirement. WebRTC was designed to solve these problems.

To acquire and communicate streaming data, WebRTC implements the following APIs: MediaStream, to get access to multimedia data streams, such as from the user's camera and microphone; RTCPeerConnection, for audio or video calling, with facilities for encryption and bandwidth management; RTCDataChannel, for peer-to-peer communication of generic data[9].

WebRTC audio and video engines dynamically adjust bitrate of each stream to match network conditions between peers. When using a DataChannel this is not true, as it is designed to transport arbitrary application data. Similar to WebSockets, the DataChannel API accepts binary and UTF-8 encoded

application data, giving the developer choices on message delivery order and reliability.

Though designed for peer-to-peer applications, in the real world WebRTC needs servers, so each of the following can happen:

- Before any connection can be made, WebRTC clients (peers) need to exchange network information (signalling).
- For streaming media connections, peers must exchange data about media such as video format and resolution.
- As clients often reside behind NAT gateways and firewalls, these may have to be traversed using STUN (Session Traversal Utilities for NAT) or TURN (Traversal Using Relays around NAT) servers.

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

A signaling channel can be any medium that allows messages to go back and forth between clients. This mechanism is not implemented by the WebRTC APIs: it has to be implemented by the developer. It can be as rudimentary as using e-mail or an IM application, can be done via a centralised server, and, theoretically, WebRTC's DataChannels can be used using multiple peers!

Table 3.1: Browser Support

Browser	WebRTC	LocalStorage	WebWorkers	Market Share
Chrome	Yes	Yes	Yes	60.1
Firefox	Yes	Yes	Yes	23.4
Opera	Yes	Yes	Yes	1.6
Safari	No	Yes	Yes	3.7
Internet Explorer	No	Yes	Yes	9.8

Table 3.1 shows the current support from different browsers for various HTML5 APIs and their current market share. Though support for communication between different browsers is currently in development, support by Chrome alone already

constitutes a majority of the users and is therefore the best candidate for the development of this work.

3.2 Proposed solution

As the work focuses on Internet applications and services, the idea is to create a framework that supports the creation of applications in the browser without the need, by end users, to install any kind of software or browser plugins, which is relevant to maximise the adoption of the proposed architecture.

The planned framework also aims at providing fundamental security properties and easy deployment. WebRTC already has basic security measures in place for secure channels. Signalling will be implemented, as it is not provided by the WebRTC API, therefore security aspects will have to be addressed.

The expected framework, roughly depicted in Figure 3.1, will be in the form of a Javascript web-application development framework, consisting of:

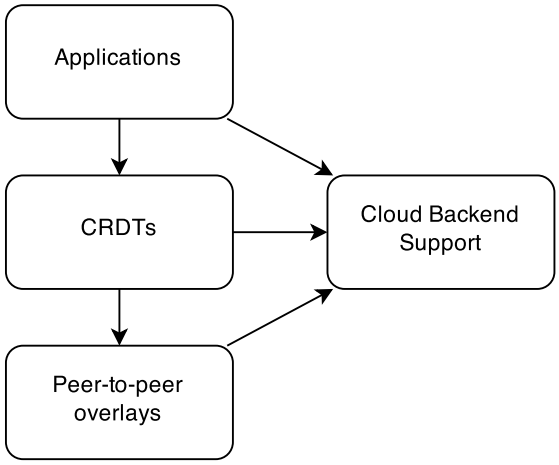

Figure 3.1: Generic overview for the proposed system

- A logical peer-to-peer connectivity layer between end users' browsers, supporting the choice between various overlay networks. As shown in Section 2.1.1, the applicational needs weight heavily in the choice of the overlay network. Therefore the implemented overlay will be built to best provide the requirements of application examples to be chosen in the design phase.

WebRTC will be used as a generic peer-to-peer data transport and WebSockets[1] for signalling. A NodeJS module will be implemented as a signalling server, enabling for easy deployment of a peer-to-peer network.

- A CRDT API, enabling real-time concurrent editing of objects in a distributed fashion. The previously described peer-to-peer overlay module will be used to propagate updates among multiple nodes replicating each object. The WebStorage API will be used to support persistent state, allowing object replicas to be maintained locally even when not being used. In the design phase we will select the CRDTs that will be included in our base system. We expect our system to include a subset of the designs previously proposed in literature[16] and new CRDTs that are specially tailored to functionalities present in the cloud databases, as discussed later in this section.

- Integration with existing cloud backends, in a way that the developed solution can be seen as an extension of those systems, enabling the construction of web based online real-time collaborative editing tools. Web application developers should be able to shift their application, with minimal effort, from using an client-server cloud based approach to our design.

- Example applications, built on top of previous items, decreasing client-to-client latency and improving scalability. As an example, a version of Google Docs where clients communicate in a peer-to-peer fashion instead could be made, reducing client-to-client latency and server load. This can be done using each of the previous items, network overlays for peer-to-peer, CRDTs for real-time concurrent editing and needs integration with the existing Google Drive Realtime API.

Our work could propose to just build a new framework with a new API. However, such approach would make the adoption and test of our work harder. So, we plan to provide a close integration with an existing framework, mapping the existing API to our framework, thus allowing application developer to continue using a familiar API while taking advantage of additional support on the clients with data replication and client-to-client communication and synchronization. In this context, we have not decided which framework will be used, but we are studying using one (or more) of the following:

[1] WebSockets, implemented by all major browsers, allow for a TCP connections to be opened between a browser and a web server, enabling richer client-to-server and even server-to-client communication.

- Redis.io[26], a key-value cache and store. Keys can contain strings, hashes, lists, sets, among others and is therefore often referred to as a data structure server. Our framework could expand Redis.io to better support clients.

- Dropbox Datastore[8], ShareJS[13] or Google Drive Realtime API[15]. All of these are frameworks used by developers to build applications requiring real-time concurrent editing of data by multiple users. Each of these can profit from using the described framework to reduce client-to-client latency and improving scalability.

- PeerJS[25] is a peer-to-peer framework enabling connections between two peers. A good example to show usage for different peer-to-peer network overlays and reducing client-to-client latency.

- Priv.io[34] and Sporc[10] systems use a centralized server for storing confidential data. Leveraging our framework, it would be possible to explore peer-to-peer capabilities to improve perfomance and reduce trust in the centralized component.

The main challenge that we will be addressing in this step is the mapping between the existing system APIs and our system. To this end, we expect to need to implement CRDTs that handle functionalities not available in currently proposed CRDTs.

We plan to evaluate our work in a setting that includes servers running in a cloud platform (Amazon AWS or Microsoft Azure) and client nodes (from PlanetLab). We will conduct a series of micro-benchmarks to evaluate specific functionalities of the system, such as communication overhead and time for executing storage operations. We also expect to measure the end-user perceived latency when executing operations - to this end, we expect to be able to use an application developed for the existing frameworks, such as Redis.io.

3.3 Work Plan

In this section the work plan for the elaboration of the dissertation is described. The work plan is to begin with the design of the system, implement and test each module, and, finally, write the dissertation. The work is to be done roughly in seven months, Table 3.2 depicts the start and end dates as well as the duration of each task.

Table 3.2: Calendar

Task	Start Date	End Date	Weeks
Design	23 February	20 March	4
Implementation	9 March	19 June	15
Phase 1	9 March	17 April	6
Phase 2	6 April	15 May	6
Phase 3	11 May	19 June	6
Final evaluation and Additional Features	22 June	24 July	5
Writing	6 July	24 September	12

Design of the system. The main task is to define system architecture and interactions between system modules. It is expected that some bibliographic refinement is to be done when design challenges are encountered.

Implementation of the system, this task can be divided into three phases, which will partially overlap, as the implementation of each phase may impact one another. Each of these phases is comprised of implementation of the needed modules and functional prototypes. The phases are as follows:

- Phase one - Implementation of peer-to-peer networks in the browser. Partially centralised overlay networks using structured and unstructured methods are to be implemented and tested.
- Phase two - Implementation of a data storage and interaction layer using CRDTs. Tentatively various CRDTs as depicted in related work will be implemented and tested.
- Phase three - Creation of example applications on top of previous work. To show the correct implementation and advantages of using the created system. This phase includes both integration with existing backends as creating example applications.

Final evaluation and Additional Features. In this task the implemented system will be more thoroughly evaluated as a whole. The knowledge obtained from this task will help us decide what improvements have to be made and how to implement additional features.

Writing, the final task, consisting of writing the dissertation.

Figure 3.2 depicts a Gantt chart presenting a summary of the described schedule, better showing how dates overlap.

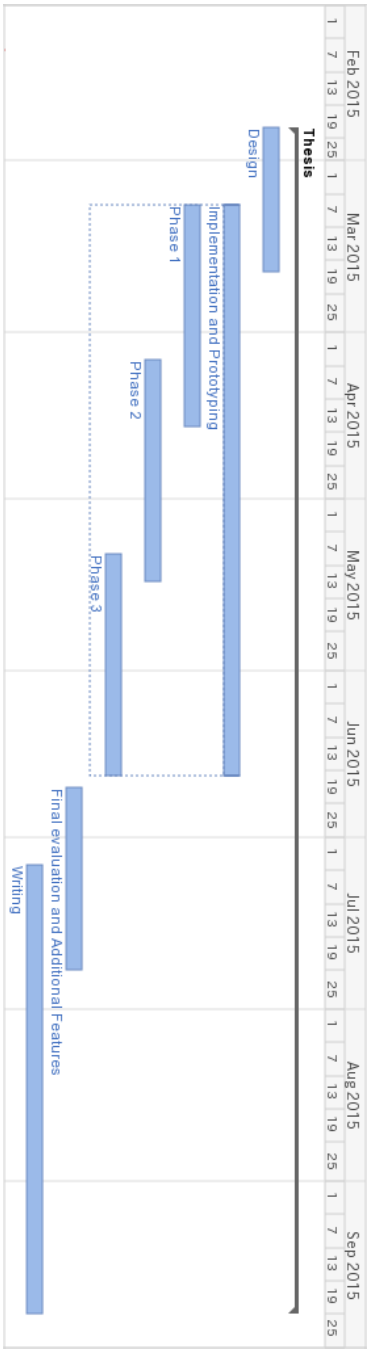

Figure 3.2: Proposed work schedule

It is also predicted that during this work a basis will be provided for scientific research papers that will be submitted to national and international conferences.

BIBLIOGRAPHY

[1] D. P. Anderson. "Boinc: A system for public-resource computing and storage". In: *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE. 2004, pp. 4–10.

[2] S. A. Baset and H. Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol". In: *arXiv preprint cs/0412017* (2004).

[3] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. "Gossip algorithms: Design, analysis and applications". In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 3. IEEE. 2005, pp. 1653–1664.

[4] B. Cohen. *The BitTorrent protocol specification*. 2008.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: http://dx.doi.org/10.14778/1454159.1454167.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[7] 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.

[8] Dropbox. *Datastore API*. 2015. URL: https://www.dropbox.com/developers/datastore.

[9] S. Dutton. *Getting started with WebRTC*. 2015. URL: http://www.html5rocks.com/en/tutorials/webrtc/basics/.

[10] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. "SPORC: Group Collaboration using Untrusted Cloud Resources." In: *OSDI*. Vol. 10. 2010, pp. 337–350.

[11] E. Foundation. *Etherpad*. 2015. URL: http://etherpad.org.

[12] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. "SCAMP: Peer-to-peer lightweight membership service for large-scale group communication". In: *Networked Group Communication*. Springer, 2001, pp. 44–55.

[13] J. Gentle. *ShareJS API*. 2015. URL: https://github.com/share/ShareJS#client-api.

[14] 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.

[15] Google. *Realtime API*. 2015. URL: https://developers.google.com/drive/realtime.

[16] 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.

[17] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[18] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. "Deconstructing the kazaa network". In: *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*. IEEE. 2003, pp. 112–120.

[19] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A membership protocol for reliable gossip-based broadcast". In: *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE. 2007, pp. 419–429.

[20] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary." In: *OSDI*. 2012, pp. 265–278.

[21] L. Napster. "Napster". In: *URL: http://www. napster. com* (2001).

[22] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. "High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 111–120. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215706. URL: http://doi.acm.org/10.1145/215585.215706.

[23] G. Oster, P. Urso, P. Molli, A. Imine, et al. "Proving correctness of transformation functions in collaborative editing systems". In: (2005).

[24] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of mutual inconsistency in distributed systems". In: *Software Engineering, IEEE Transactions on* 3 (1983), pp. 240–247.

[25] PeerJS. *API Reference*. 2015. URL: http://peerjs.com/docs/.

[26] Redis. *API*. 2015. URL: http://redis.io/commands.

[27] M. Ripeanu. "Peer-to-peer architecture case study: Gnutella network". In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. IEEE. 2001, pp. 99–100.

[28] R. Rodrigues and P. Druschel. "Peer-to-peer Systems". In: *Commun. ACM* 53.10 (Oct. 2010), pp. 72–82. ISSN: 0001-0782. DOI: 10.1145/1831407.1831427. URL: http://doi.acm.org/10.1145/1831407.1831427.

[29] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, et al. "A comprehensive study of convergent and commutative replicated data types". In: (2011).

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[31] S. Voulgaris, D. Gavidia, and M. Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217.

[32] WebRTCfortheWeb. *Is WebRTC ready yet?* 2015. URL: http://iswebrtcreadyyet.com.

[33] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. "SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine". In: (Oct. 2013). eprint: 1310.3107. URL: http://arxiv.org/abs/1310.3107.

[34] L. Zhang and A. Mislove. "Building Confederated Web-based Services with Priv.Io". In: *Proceedings of the First ACM Conference on Online Social Networks*. COSN '13. Boston, Massachusetts, USA: ACM, 2013, pp. 189–200. ISBN: 978-1-4503-2084-9. DOI: 10.1145/2512938.2512943. URL: http://doi.acm.org/10.1145/2512938.2512943.