# Merge: a Distributed Collaborative Editor

Guha Balakrishnan, Mandy Korpusik, Amy Ousterhout

May 11, 2014

## 1 Introduction

Merge is a collaborative text editor, which allows multiple users to edit a document simultaneously from different computers, even if they are not co-located. The editor provides the following properties:

1. A user's own edits are applied immediately.
2. There is minimal delay between when user A edits a document and when user B sees A's edits.
3. Conflicts are merged in an intuitive and reasonable way.
4. Editor servers are distributed and fault-tolerant.

## 2 Design

### 2.1 Overview

Merge is a collaborative text editing framework consisting of servers that coordinate edits between users that are typing in separate web browsers (Fig. 1). A user makes edits to the local document in his/her browser, and the local client object sends each edit to a server. Merge supports two types of edits - adding characters and deleting characters. The server also mantains a local document that can be used to initialize new users. In the non-replicated case, there is a single server handling all edits. In the replicated case, servers maintain consistency among one another using a Paxos (or optimized MultiPaxos) scheme. Clients can submit edits to any of the replicas and be assured that all the replicas will eventually have the same version of the document.

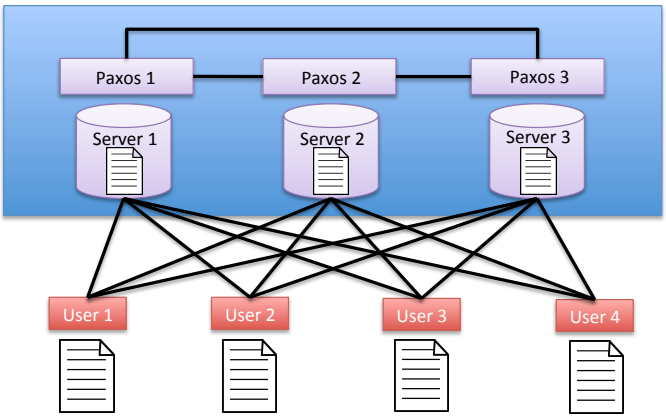


Figure 1: Overview of Merge system, depicting 4 clients and 3 Paxos servers.

### 2.2 Platform

We wrote Merge in JavaScript using Node.js, a platform for building scalable server-side networking applications. We chose to use Node.js because of its event-driven and asynchronous model which is particularly suited for applications in which a server must handle many simultaneous, non-intensive operations at high throughput. Our application fits this use case since a single document may have many collaborators performing simultaneous, rapid edits. Each edit changes only one character of the document, and is therefore not CPU-intensive.

Rather than using HTTP Ajax Get and Post requests for communication between the client and server, we decided to use Socket.io which allows communication through sockets between servers and clients, as well as between servers. Sockets were helpful in our case because we wanted to provide clients with the ability to communicate with any of the server replicas, and we needed servers to communicate with each other in order to implement Paxos and MultiPaxos.

### 2.3 Merging Edits: Operational Transformations

In order to allow users to immediately see the effects of their own edits, we use optimistic concurrency control [1, 3, 4]. Optimistic concurrency control allows clients to immediately modify their own copy of a document, and then merges modifications to different copies of a document later. This approach sacrifices consistency for availability, but is quite reasonable in this context, when conflicting edits are rare; for example, only edits at the same index in the document may conflict. This provides the first property listed in §1 (i.e. the user's edits are applied immediately).

Many existing schemes use optimistic concurrency control to allow multiple users to simultaneously edit a document or file; we chose to implement a scheme similar to that of Jupiter [3], a system that supports shared widgets. As in Jupiter, we use a logically centralized server to manage all updates for a single document and to distribute these updates to all clients; clients communicate only with the centralized server and not with each other. We think this is a reasonable approach, given that most web apps are already served by a centralized server, and centralization significantly simplifies the design compared to distributed approaches [5].

To detect and resolve concurrent updates from different clients, we use operational transformations, as in Jupiter. Operational transformations provide a way of transforming operations so that two clients that apply two operations in different orders can arrive at the same final document. More formally, as described in the Jupiter paper, we must create a function $xform$ such that $xform(c, s) = \{c', s'\}$, and applying $c$ then $s'$ to a document yields the same result as applying $s$ then $c'$. This property guarantees that if a client and server perform operations $c$ and $s$ concurrently, the two can arrive at the same final document after they receive the other's operation, transform it, and apply it.

As described above, Merge supports two operations: add char and delete char. Therefore, we need three main functions to transform the three possible pairs of operations: two adds, two deletes, and an add and a delete. These functions ensure that conflicts are merged intuitively, to provide property 2 in §1 (i.e. minimal delay between client A seeing client B's edit). For example, if two clients concurrently add a character at the same index, both characters will appear in the final document (in an arbitrary order), but if two clients concurrently delete a character at the same index, the delete will only occur once. All operations are represented as a diff relative to the previous doc; an add is specified by a character and the index at which it should be inserted, while a delete is specified simply by the index at which a character should be deleted. For transforming two delete operations, we use the $xform$ function in the Jupiter paper. For two add operations and an add and a delete operation, we wrote our own functions.

This simple approach for one client and one server is sufficient to support any number of simultaneous clients, if the transform functions are chosen correctly. The server processes operations in the order that they arrive, and transforms each arriving operation against all pending operations. Transformations are applied pairwise, and therefore work with a simple $xform$ function for two operations, regardless of the number of clients. Further details of this approach can be found in the Jupiter paper.

## 2.4 Replication

While one server is sufficient for Merge to work correctly, it is a single point of failure. We therefore replicated the server using a Paxos scheme similar to Lab 3 [2]. Each replica server communicates with other replicas only through its local Paxos instance. Each "value" submitted to the Paxos log is a user's edit. A server replica sweeps through its Paxos log in the background to update its local document.

One significant difference between Lab 3b's implementation and ours is that we cannot assume that clients will send edits serially. In fact, we expect several edits to occur simultaneously as a user types one word because each edit consists of a modification to only a single character. We handle this by making a server only commit an operation to its paxos log once the message previously sent by the same user is known to have been committed. This means that the server must wait for its background process that sweeps the paxos log to execute the previous message. This will increase latency but is needed to ensure correctness.

### 2.4.1 MultiPaxos

We implemented MultiPaxos as an optimization to the Paxos scheme, since MultiPaxos eliminates one roundtrip by skipping the Prepare phase [2]. In Paxos, every replica can send Prepare, Accept and Decide messages. However, in MultiPaxos, only the designated leader may act as a proposer, and all non-leaders only act as acceptors. Since there is one leader, it can Prepare the entire log ahead of time, which means every time a new message arrives, it can skip ahead to the Accept phase. In our implementation, a new leader only sends a Prepare message for the first instance it proposes.

We used the following simple leader election protocol:

- The server with the lowest ID acts as the leader.
- Each server pings all other servers every pingInterval ms, where pingInterval = 1000 ms.
- If a server hasn't received a ping from the server with the lowest ID in 2*pingInterval ms, and it has the second lowest ID, then it appoints itself as the new leader.

When a non-leader server attempts to start a new Paxos instance, it receives an error message. Thus, if a client sends an operation to a non-leader, the server will respond with an error message, and the client will re-send the operation to a different replica.

### 2.4.2 Duplicate Messages

It is possible for a client to send the same message multiple times to replicas. We enforce at-most-once handling of messages by using message generation ids.

#### 2.4.3 Server Crashes

As long as a majority of servers are active, our editor will still work. Clients will recognize that a server is dead and will not send edits to it any longer. Unfortunately, we were not able enforce correct behavior when a server crashes while handling a user edit. Our code only works when a server crashes in between edits. Due to time constraints, we also did not handle the case where a server crashes and reboots.

## 3 Evaluation

### 3.1 Correctness of Operational Transformations

We augmented our client and server designs to enable easy and reproducible testing. First, we added functionality to our server so that it can read in a given test file, send the specified operations to all clients, and then instruct the clients to start once all were ready. Clients apply operations at times specified in the test file. This provides reproducibility of tests. Second, we added a delay to the clients so that all outgoing messages could be held for a specified time before being sent to the servers. This allowed us to force operations to be concurrent, even when the client-server latency was extremely small. Finally, at the end of a test, clients send their current document text as well as a log of all of their received operations back to the servers. The servers then check that all clients and all servers agree on the contents of the document, ensuring eventual consistency.

Using these three features, we thoroughly tested our OT implementation. We tested add operations alone, delete operations alone, and interleaved add and delete operations, for two and three clients. We verified that between these different test cases, we exercise all possible pathways through the transform functions that we implemented, both at the client and the server (code coverage). We also wrote a few tests for specific bugs that we encountered (and fixed) and randomly generated test cases. For the random tests, a Python script generates one thousand operations to be performed over a minute or two, for three clients. Each operation involves an add or a delete, with characters for the add operations chosen randomly. For some tests, the interval between operations was fixed; for other tests, Poisson arrivals were used.

All of the correctness tests for OT pass with two clients. All of the specific case tests for OT also pass with three clients. However, we suspect that there is a bug in our OT implementation for three or more clients, because it does not always pass the longer randomized tests when there are more than two clients editing simultaneously.

### 3.2 Correctness of Paxos/MultiPaxos Replication

In order to test our implementations of Paxos and MultiPaxos, we wrote several automated test cases similar to those in lab 3a: OneProposer, ManyProposers, ManyProposersDiffVals, OutOfOrder, Many, ManyUnreliable, KillLeader, and KillMajority. We used three replicas for each test. For each sequence number, all replicas must agree on the decided value in order to pass the tests. In order to simulate an unreliable network with dropped packets for the unreliable test, Prepare/Decide/Accept messages and replies were randomly not sent approximately 10% of the time. We used KillLeader to test whether MultiPaxos would appoint a new leader and continue to make decisions after the old leader died. In addition, this tests whether Paxos still makes progress after a minority of peers die. We used KillMajority to ensure that Paxos would not make a decision after a majority of peers die.

### 3.3 Performance

We evaluated the latency of client operations (i.e. the amount of time between an operation being submitted by one user and becoming visible to another user) with a varying number of replicas for both Paxos and MultiPaxos replication schemes (Fig. 2). In all cases, we ran the same test (concurrent_both_random_2_clients), which simulates one thousand operations betweem two clients. No replication was used for the case where there was only one replica and, as expected, this case had the lowest mean latency of fewer than 54ms. We used a 50ms delay to simulate the roundtrip time for a message to travel from a client to a server and back. We also used a 5ms delay between replicas to simulate communication between replica servers that are all located within the same datacenter. As expected, MultiPaxos consistently has a lower average latency than Paxos because there are fewer roundtrips. The higher mean latency for MultiPaxos when using three replicas is possibly due to a bottleneck at the leader.

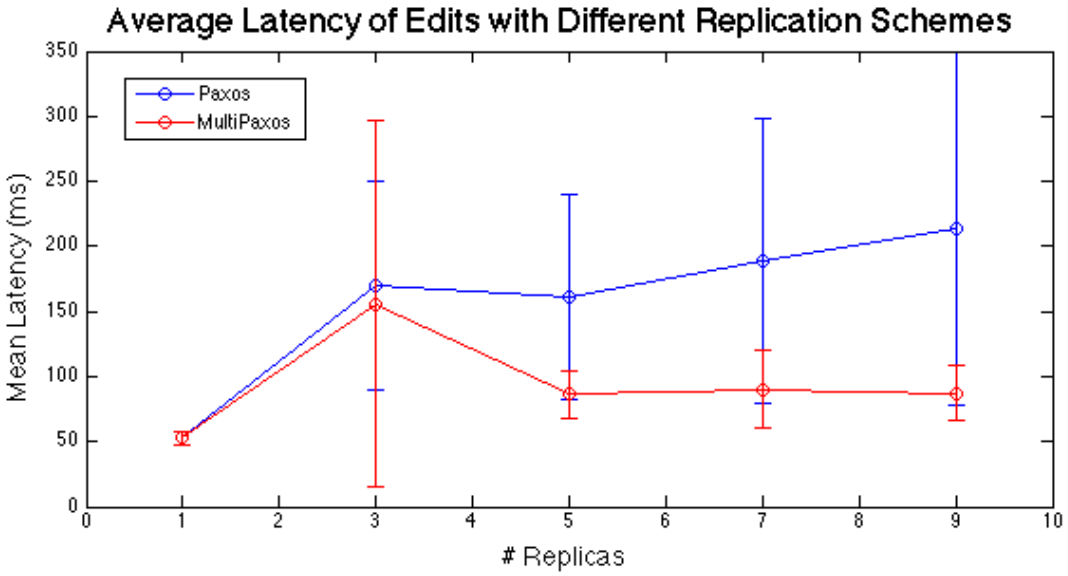


Figure 2: Plot of mean and standard deviation of latencies of edits for Paxos and MultiPaxos replication schemes with various numbers of replicas. The standard deviations of the measurements were all below 145 ms. Latencies greater than 1000ms were considered anomalies and were filtered out before calculating the mean and standard deviation.

## References


[1] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[2] Leslie Lamport. Paxos made simple. Nov. 2001.

[3] David A. Nichols, Pavel Curtis, Michael Dixon, and John 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, pages 111–120, New York, NY, USA, 1995. ACM.

[4] Peter L Reiher, John S Heidemann, David Ratner, Gregory Skinner, and Gerald J Popek. *Resolving file conflicts in the Ficus file system.* UCLA Computer Science Department, 1994.

[5] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. *Proc. of ACM Conference on Computer-Supported Cooperative Work*, pages 59–68, Nov. 1998.