university of groningen

faculty of mathematics and natural sciences

Data Replication in Offline Web Applications:

Optimizing Persistence, Synchronization and Conflict Resolution

Master Thesis Computer Science

May 4, 2013

Samuel Esposito - 1597183

Primary supervisor: Prof.dr. M. Aiello

Secondary Supervisor: MSc A.C. Emerencia

Abstract

As HTML5 features like web-storage and offline web-apps become more widely supported by modern browsers, an increasing number of web applications push their logic to the client. The consequence of this decentralization trend is that part of the application data now lives on the client-side. In order to keep this mobile data consistent and meaningful over time, it has to be replicated between all clients in the network. This poses developers building offline web applications for complex challenges: traditional eager data replication strategies are not fit for synchronizing mobile data, because web applications can go offline anytime and generate data while being offline. Fortunately the literature on mobile data replication offers solutions for doing lazy data replication instead and proposes algorithms for automatically detecting and resolving the synchronization conflicts that may occur during lazy replication.

In this thesis we attempt to optimize these replication and reconciliation solutions for use within the context and constraints of offline web applications. Our benchmarks show that the network load of synchronizing conflicting data updates can be reduced and that the memory footprint of recording local data updates can be decreased. In addition we propose an *eager conflict resolution* strategy and explore whether it better fits the constraints of offline web applications than the existing solutions. We provide two different implementations of this strategy and compare them in terms of performance overhead and the average number of conflicts they successfully resolve.

Contents

1. Introduction

With the advent of the HTML5 standard [7, 8, 9] and its support by most modern web browsers a whole new set of features has recently become available to web developers. Three major examples of such features are offline web-apps, web-storage and web-sockets. We explain them in detail below.

Offline web-apps allow the end user to continue using a web application, or part of its functionality, once it is opened, even when an internet connection is no longer available. Resources like HTML, CSS, JavaScript and image files are stored locally by the browser and are kept up to date as they change. When the application is accessed without a network connection, the browser switches over to the local files so that the application continues to work as usual, even when navigating between pages [7, 10].

The web-storage feature complements offline web-apps in that it allows to locally store data objects that would be stored on the server otherwise. Using this feature the state of a web application remains available to the end user when the application is offline and pages are being refreshed [7, 11]. The data stored in the web-storage can be synchronized to the server at a later stage.

Web-sockets provide a single-socket full-duplex data channel allowing web applications to use bi-directional communication with the server [13]. Compared to previous polling and long-polling solutions, web-sockets reduce the network traffic and latency [15]. It is the ideal communication means for synchronizing data with the server.

The combination of the three web technologies described above allows developers to build web applications that have capabilities resembling those of native clients. As a consequence an increasing number of web applications push part or all of their application logic from the server-side to the browser. And with this decentralization much of the application data now gets to live on the client-side and requires synchronization techniques in order to remain consistent and meaningful over time [1].

For this kind of synchronization traditional pessimistic replica control protocols are not suitable, as end users must be able to modify their data while being disconnected from the network [4, 5]. Instead an optimistic replica control strategy is called for, allowing local data writes without locking any resources on the server-side or any other node in the network [1, 3]. In addition, as the literature on mobile data replication points out, local data changes have to be replicated in a lazy manner, because waiting for all nodes to be online at the same time during synchronization is infeasible [2]. One important implication of these architectural constraints is that local data updates are not necessarily serializable any more [4, 6]. This means concurrent data modifications made by different end users in different browsers may very well

conflict with each other. These conflicts have to be detected and resolved at synchronization time in order to keep the mobile data consistent.
In this thesis we attempt to optimize existing data replica management and conflict resolution protocols for use in the context of offline web applications, as such applications impose specific requirements and technical limitations to mobile data replication. To begin with, when a substantial number of nodes is disconnected while mobile data is changed locally, as we have with offline web applications, the odds of generating conflicting updates are much bigger than in a connected system [2]. Therefore the network load caused by synchronizing and resolving conflicting versions of mobile data is an important aspect of the system. Our benchmarks show that this load can be reduced significantly using our proposed *preventive reconciliation* strategy, while only inducing limited network overhead on the synchronization of non-conflicting data updates.
Also when storing data locally in an offline web application one has to deal with the strict memory limitations browsers impose on local storage [12]. Therefore the memory footprint of data versioning in a data replica management system has to be minimal, leaving most of the memory for storing the actual data. In this thesis we propose a *merged browser log* for storing different versions of data objects. Our benchmarks indicate that this solution decreases the memory footprint of data versioning. In addition we show that the log only grows very slowly once it exceeds the original size of the data objects whose updates it records.
Finally in the context of offline web applications one cannot rely on the end user to resolve data conflicts. Resolving conflicts can be a tedious and confusing task and often requires some technical knowledge [45]. In this thesis we explore an *eager conflict resolution* strategy which aims at resolving all conflicts automatically, irrespective of their nature or cause, while maximally preserving the conflicting data versions. We propose two different implementations of this strategy: one based on comparing serialized data objects and one considering individual data attributes in isolation. Both implementations are compared in terms of performance overhead and the average number of conflicts they successfully resolve. Our benchmarks show that the performance of the serialized data implementation does not scale well in the browser. This is undesirable because it might prevent a web application from being responsive to user interaction when a lot of data is updated locally. In addition the serialized data implementation fails to resolve a substantial number of conflicts when local updates change a large portion of the data. The results for the attribute oriented implementation are a lot better. The implementation causes negligible overhead in the browser and resolves the majority of all data conflicts, even when conflicting updates change a large portion of the data.

In the next section we shed more light on existing solutions for mobile data replication, data persistence in the browser and conflict resolution. In addition we go into more detail about our optimizations, which aim at improving the fitness of these solutions for use in the context of offline web applications. After that we explain the concepts and architectural decisions behind our optimized mobile data replication system. In the methods section we discuss the benchmarking setup we use to profile the network load, memory footprint and performance of our system. Finally we present the results our benchmarks yield and discuss them in the light of the current literature on mobile data replication.

2. Related work

In the following subsections we give an overview of the solutions the current literature proposes for mobile data replication, data versioning and persistence in the browser and automated conflict resolution. After that we present some optimizations of these solutions in the context of offline web applications and discuss the improvements in network load, memory footprint and performance we expect them to bring to the data replication process.

2.1 Mobile data replication

As mentioned in the introduction most traditional data replication solutions do not apply to an offline web application setup [4, 5]. Gray *et al.* [2] thoroughly analyze the behavior of eager and lazy replication schemes for a distributed disconnected architecture as offline web applications have. When performing eager replication, all nodes in the network are updated in one atomic transaction. In order to successfully complete, such a transaction has to wait for all nodes to be online, while maintaining a lock on the data being updated. In the context of offline web applications where nodes can be offline for hours these data locks would bring the entire system to a grinding halt in no time, making this approach completely infeasible.

This leaves us lazy replication algorithms as only remaining option, an approach which is not without risks. When any node in the system can modify any data object anytime, updates are not necessarily serializable any more. This means they cannot be mapped to a single timeline without overlap [16]. Overlapping updates may very well result in conflicts, which have to be reconciled on the client-side before the updated data can be stored on the server. It is the merit of Gray *et al.* to show that a simple replication setup using lazy replication creates a scale-up pitfall. When either the number of nodes in a system increases or the nodes are spending more time offline, the conflict rate grows cubically and quickly becomes astronomically high [2]. A long message propagation time needed for broadcasting replica updates makes the situation even worse. When conflicts are not quickly and successfully resolved, a state of system delusion is reached: the distributed database becomes inconsistent and there is no obvious way to repair it [2].

In order to address this issue Gray *et al.* propose a two-tier replication approach, consisting of base nodes that are always connected to each other and mobile nodes that may be offline for some time. Mobile nodes make tentative updates to a local copy of the data. They occasionally connect to the base node cluster to propose these updates. When updates can successfully be re-executed at a single base node, they are persisted on all base nodes using traditional eager replication. After that, the local copy of the data residing on the mobile node is updated to reflect concurrent changes on the base nodes. Finally rejected tentative updates are reconciled by the mobile node owner who generated them. With this scheme

Gray *et al.* achieve a scalable architecture which allows nodes to modify data while being offline, while the data in the system still converges to a stable consistent state, eventually.

2.2 Data versioning and persistence

Based on the two-tier replication architecture described above, Cannon and Wohlstadter develop an interesting framework providing automated data versioning and persistence for offline web applications [1]. The framework allows to mark JavaScript data objects as persistent. These objects are continuously monitored for changes to their data. When a local update occurs, the updated data is automatically persisted together with a log of the data change. The so called browser log thus recorded is optimistically synced [3] with the server by sending over the updated data at regular intervals. Any update conflicts that may occur during the synchronization process are detected by simply comparing timestamps. More precisely the timestamp a local version of the data received during its last synchronization phase is compared to the timestamp of the master version [2] on the server. When these timestamps match, the update is considered non-conflicting and can be applied. When the server version is newer than the local version, a conflict is detected. The conflict is then reported to the client, which handles it in the application layer or presents it to the end user. Finally the local version of the data in the browser is updated with all changes that were saved to the server since the last synchronization session. The data versioning and persistence framework of Cannon and Wohlstadter is a welcome addition to the field of web application development, where generic solutions for data replication are sparse.

2.3 Automated conflict resolution

As we discussed in the introduction of this document it is advisable to automate the reconciliation process instead of depending on the end user for resolving conflicting data updates. Zhiming *et al.* put forward a transaction-level result set propagation model [4], which allows for automatically detecting and resolving conflicts during data synchronization. When performing local updates, the so called read set and result set of a data change are recorded on the client. At synchronization time the local data versions in the read set are compared to the respective versions on the server. When the local versions are up to date, the update is considered serializable, and the corresponding result sets are applied on the server. Thanks to this fine-grained set comparison, data updates can be replicated to the server even when the data in their result sets has known concurrent updates since the client's last synchronization session.

Phatak *et al.* take a different approach to the problem of automated conflict resolution, for which they coin the name multiversion reconciliation [6]. For detecting and resolving conflicts they consider the write set of an update instead of the read set

as Zhiming *et al.* do. And rather than requiring full serializability of data changes, multiple versions of the data are allowed as a starting point for updates. These multiversion updates can be applied to the server as long as they do not overwrite any concurrent updates to data objects in their write set during synchronization. So when using multiversion reconciliation, data updates can be synchronized even when the data in their read sets has changed on the server since the last synchronization session.

2.4 Optimizations for offline web applications

With a solid data replication architecture, data versioning and persistence in the browser and strategies for automated conflict resolution we seem to be all set for replicating mobile data in offline web applications. There are however some conditions and limitations specific to offline web applications that call for optimizations in the existing solutions. These conditions and optimizations are described in detail below.

Preventive reconciliation

In offline web applications a considerable number of nodes frequently has no access to the network. Therefore local data changes typically remain local for some time before they are sent to the server. In addition it takes time for these changes to propagate from the server to all other nodes in the network. Because of this delay between data updates and data replication the conflict rate in offline web applications is higher than in traditional applications, especially when many nodes manipulate the data. In order to keep the distributed database consistent it is important that these conflicts can be detected and resolved in a timely manner [2]. Unfortunately synchronizing conflicting updates using the framework of Cannon and Wohlstadter induces undue network load delaying the successful resolution of conflicts. In their approach a synchronization session starts by sending local changes to the server, which detects conflicting updates and reports them back to the client. The conflicts are resolved on the client, and the amended updates are sent back to the server. The data associated with conflicting updates is thus sent over the network twice, once before the conflicts are detected and once after reconciliation. Because conflicts are more frequent in offline web applications than in traditional connected applications, this network overhead may further delay the replication of data and thus cause even more conflicts [2].

This is where our first optimization comes in. We attempt to reduce the network overhead of synchronizing conflicting updates by using *preventive reconciliation*. In this approach the client starts by sending the server only the metadata needed for detecting conflicting local updates. The server reports back any conflicts, which are then resolved on the client-side. Only when all conflicts are resolved, the actual updated data is sent to the server. We expect that *preventive reconciliation*

decreases the network load in the case of synchronizing conflicting updates, as the actual updated data is sent over the network only once. In addition we expect that sending metadata to the server before sending the actual data only induces minimal overhead to the synchronization process.

Merged browser log

When a mobile data object is updated locally using the framework of Cannon and Wohlstadter, the changes are stored in a browser log. This log is an incremental record of all local data changes since a client's last synchronization session. Because in offline web applications clients typically are disconnected for large periods of time while being fully operational, these incremental logs can consume a lot of memory. This is a problem because local data storage in modern web browsers has a strict size limit [12]. And the more memory needed for storing the browser log, the less remains for storing the actual application data.

In this thesis we propose a *merged browser log* for recording local data updates. Whenever a data object is updated, the changes are merged into a single log entry for this object containing the original values of any previously updated data attributes. Updating data attributes that were changed before does not affect the log entry. As such the size of the merged browser log only grows as long as it does not contain all original attribute values or new attributes are added by local updates. We expect the *merged browser log* to consume less memory than an incremental log when a substantial number of local updates are being made, as every attribute is recorded only once. In addition we expect the log to grow very slowly once it recorded all original attribute values, as only the keys of new attributes are added from then on.

Eager conflict resolution

The approaches to automated conflict resolution described above explicitly distinguish the read set and write set of a local data update. This is very useful for a distributed data processing setup bearing a clear distinction between input and output. In the context of offline web applications where data updates are performed by human beings, it is usually impossible to determine the read set of an update. When for instance a piece of text is changed in a text area, this update can be based upon any source, either internal or external to the web application. So when the write sets of such an update conflict with the version stored on the server, the approach of Zhiming *et al.* cannot be used to resolve the conflict, because one cannot verify whether the read set of the update is unchanged since the client's last synchronization session. Equally the algorithm of Phatak *et al.* does not allow to resolve the conflict, because we know the update's write sets do conflict.

At this point we are left with no option but to rely on the end user for resolving conflicting local updates. As we mentioned earlier however, this is no desirable situation. Resolving conflicts can be a tedious and confusing task and often requires

some technical knowledge [45]. For this reason we explore an alternative reconciliation strategy, which we call *eager conflict resolution*. This strategy aims at resolving all conflicts in an automatic way, so that this process is completely hidden from the end user. Conflicts are resolved by re-executing the updates in one conflicting version of the data on top of the data version it conflicts with. When re-execution fails, the process is reversed: the updates in the last data version are re-executed on top of the first version. The reconciled version thus obtained, has received the updates of both conflicting data versions. We provide two different implementations of this reconciliation process. The first implementation records updates by comparing serialized data objects and re-executes updates by patching these serialized data objects. Our second implementation records updates for individual data attributes and re-executes them by patching these attributes. We expect that recording updates causes only limited processing overhead in the browser, so that a web application remains responsive to user interaction when local updates are being performed. We furthermore hope that for all conflicting updates re-execution succeeds on at least one of the conflicting data versions, so that the changes in these versions are maximally retained in a final reconciled version. We make no specific predictions about any differences in performance between our two conflict resolution implementations.

In this section we presented the current state of research in mobile data replication, data versioning and persistence for the web and automated conflict resolution. In addition we explained how our work seeks to optimize the existing solutions for use in the context of offline web applications. In the next section the concepts and architecture behind these optimizations are discussed in more detail.

3. Concept

In this section we discuss the architecture of our optimized data replication system for offline web applications. We kick off by sketching the overall setup of the system. After that we take a closer look at the architectural decisions behind the optimizations for network load, memory footprint and conflict resolution. For a detailed description of the actual implementation of our data replication system we refer to appendix A.

3.1 System architecture

As discussed in the related work section a two-tier replication architecture is employed in our mobile data replication system for offline web applications. One tier consists of mobile nodes, being the clients that run in the browser and make local updates to mobile data. The second tier contains base nodes, which are the web servers that maintain the master version of the data. When online, mobile nodes initiate a synchronization session with a base node in the second tier in order to propose their local updates (see figure 1).

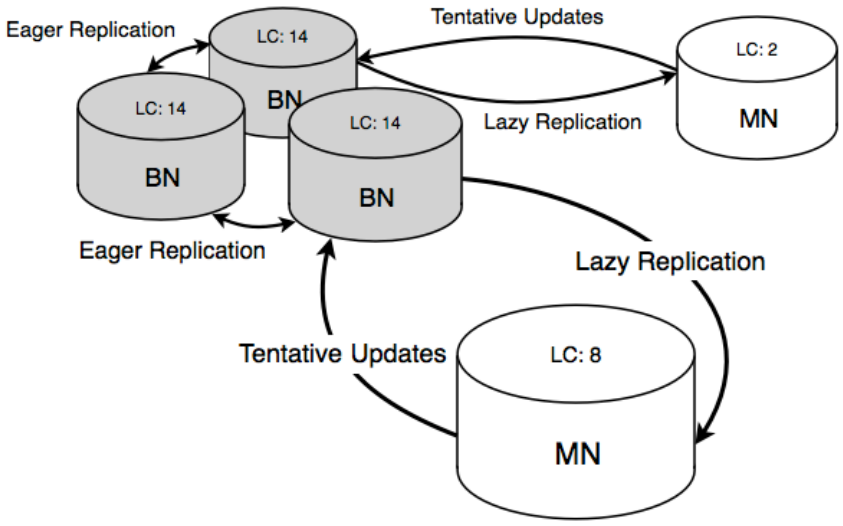


Figure 1: A two-tier replication architecture is used to manage an offline web application's mobile data. When online, mobile nodes (MN) make tentative updates to the base nodes (BN) during a so called synchronization session. When these updates can successfully be applied, the base nodes eagerly replicate them among each other. A Lamport clock (LC) keeps track of each individual synchronization session. This clock is used to determine and lazily replicate the remote updates a mobile node misses while being offline.

The base node checks whether these tentative updates can be safely applied and, if so, eagerly replicates them with all other base nodes. Each synchronization session is identified by a Lamport clock [37], which is assigned by the base nodes. All objects that are successfully updated during such a session are marked with this clock. The base nodes lazily replicate all updates a mobile node missed while being offline by querying all objects bearing a Lamport clock greater than the clock of the mobile node's last successful synchronization session.

3.2 Preventive reconciliation

Before replicating tentative updates, base nodes check whether these can safely be applied. More specifically they verify whether a local data update is generated concurrently with the update that was last applied to the master version of this data. Concurrent updates are said to conflict because they might unintentionally overwrite each others data modifications. To detect such conflicts, every version of a data object is labeled with a vector clock [17]: an array of Lamport clocks indicating the number of updates each mobile node performed on the object. Now all a base node has to do in order to detect a conflict is compare the vector clock of a local data version with the vector clock of the master version of this data. When the local and master vector clock do not succeed one another, the update is considered conflicting (see figure 2).

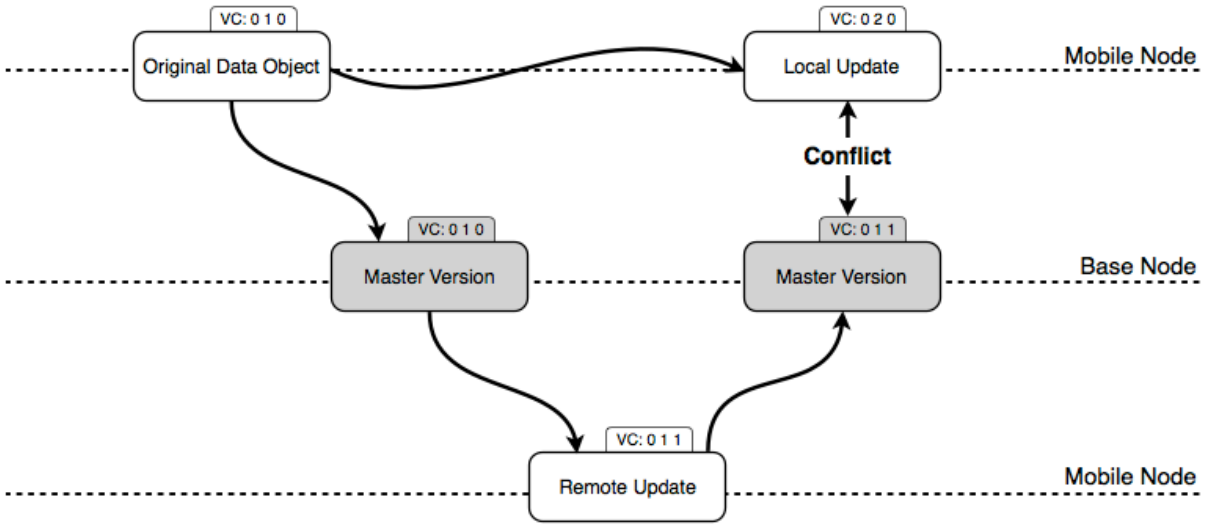


Figure 2: Every version of a data object is labeled with a unique vector clock (VC). When two versions of a data object bear vector clocks that do not succeed one another, these versions are said to conflict. Such a conflict can arise when two mobile nodes update the same data object concurrently.

Now we understand how detecting conflicts works, we can sketch how conflicts are traditionally handled during a synchronization session. The mobile node initiates the session by sending all pending local updates to a base node. The base node compares the updates' vector clocks with those attached to the master version of the affected data. When a conflicting update is detected, this master version is sent back to the mobile node. This node then reconciles the conflicting update using the master

version provided and sends the reconciled update back to the base node for replication (see figure 3). In this setup synchronizing a non-conflicting update takes only one message. Synchronizing a conflicting update however requires sending the updated data twice: once before the conflict is detected and once after it is resolved.

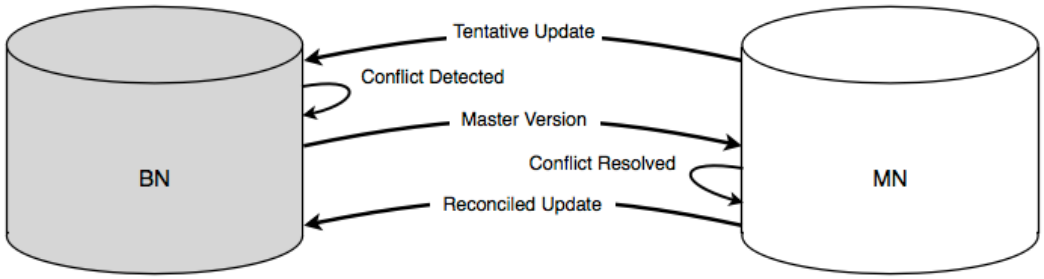


Figure 3: During traditional synchronization tentative updates are sent to a base node (BN) without knowing whether they can safely be applied. When a conflicting update is detected, the master version of the affected data is sent to the mobile node (MN) that generated the update. This node resolves the conflict and sends the reconciled update back to the base node, which applies it. In this approach the data of conflicting updates is sent over the network twice: once before and once after reconciliation.

When using *preventive reconciliation* the mobile node starts the synchronization session by sending only the vector clocks of all locally updated data objects. This information is all the base node needs to detect conflicts. When an conflict is detected the master version of the data is sent back to the mobile node. This node resolves the conflict and sends the actual data of the now reconciled update to the base node. Synchronizing a conflicting update thus requires sending the updated data only once: after the conflict has been resolved. Synchronizing a non-conflicting update requires two messages however: one containing an update's vector clock and one containing the actual updated data.

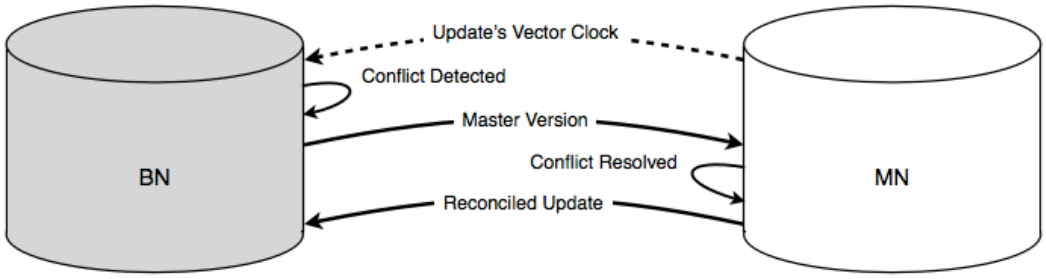


Figure 4: In preventive reconciliation an update's vector clock is sent to a base node (BN) first. This is all information needed for detecting a conflicting update. In the event of a conflict, the update is reconciled by the mobile node (MN) and sent back to the base node. In this approach the data of a conflicting update is sent to the base node only once: after the conflict has been detected and resolved.

3.3 Merged browser log

Local updates to mobile data are recorded in a so called browser log. The purpose of this log is two-fold: it stores all data that requires replication and it contains the updates to be re-executed when resolving a conflict. Traditionally such a browser log is incremental: with every data update a new browser log entry is appended containing the changes performed (see figure 5). In our implementation of an incremental browser log, updates are stored as the diff between the serialized old and new data version. Because offline web applications are fully operational while being disconnected from the network, many local updates can be performed on the data before the application is able to initiate a synchronization session. And because with every update the browser log stores an extra entry, this log can consume a lot of memory.

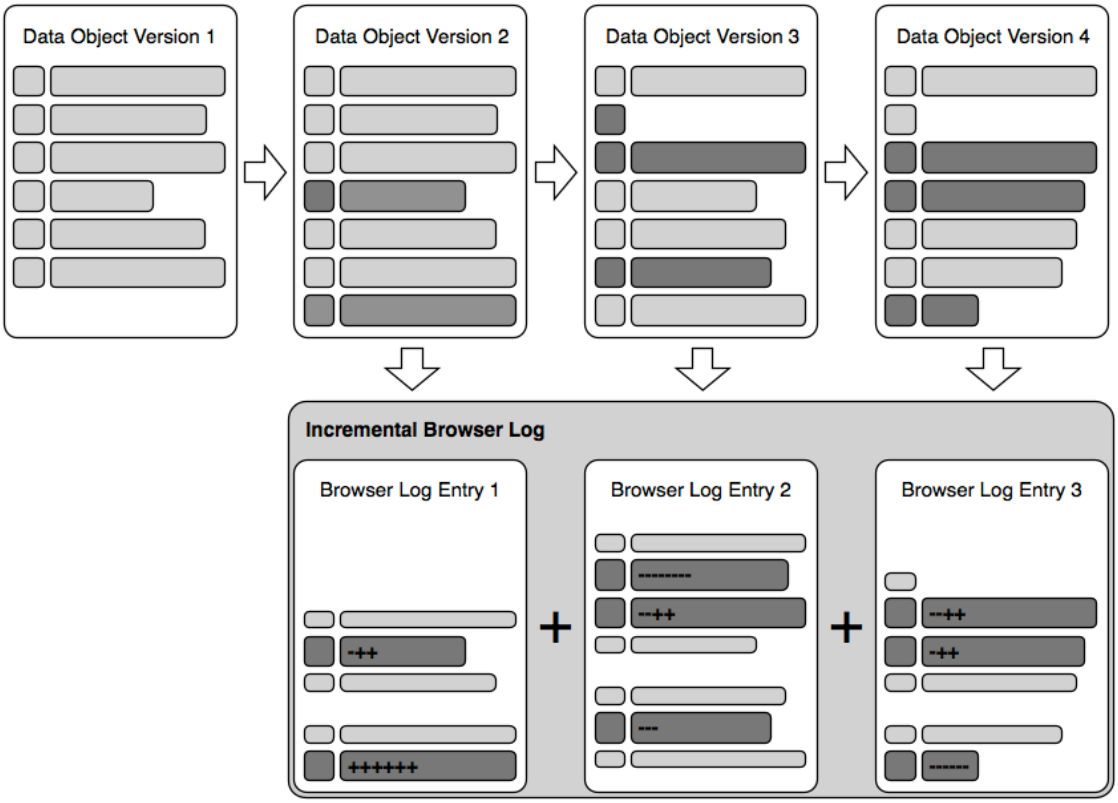


Figure 5: Whenever a local update gives rise to a new version of the data, the performed update is stored in an incremental browser log (dark colored bars represent changed data attributes). Every log entry is a diff between the serialized old an new version of the data. An incremental log has a large memory footprint when an increasing number of updates are being made.

In a *merged browser log* instead of storing all performed data updates separately, updates on the same data object are stored into a single log entry. More specifically for every changed data attribute in an object, its original value since the objects last synchronization session is recorded in this merged log entry. This is enough information for later replication because for each recorded attribute the updated data can be extracted from the data object the log entry belongs to. It also is enough

information for later conflict resolution, because the updates performed between the original and current version of a data object can be derived from the diffs between the recorded original attribute values and the object's current attribute values. When an update is performed on a previously updated data object, only attributes that were not touched before are added to the merged log entry for this object (see figure 6). The original value of the other attributes is already recorded after all. Therefore the *merged browser log* does not grow as fast as the incremental variant. In addition it only grows very slowly once it contains all original values of a data object's attributes, as it only records attribute keys for newly added attributes from then on.

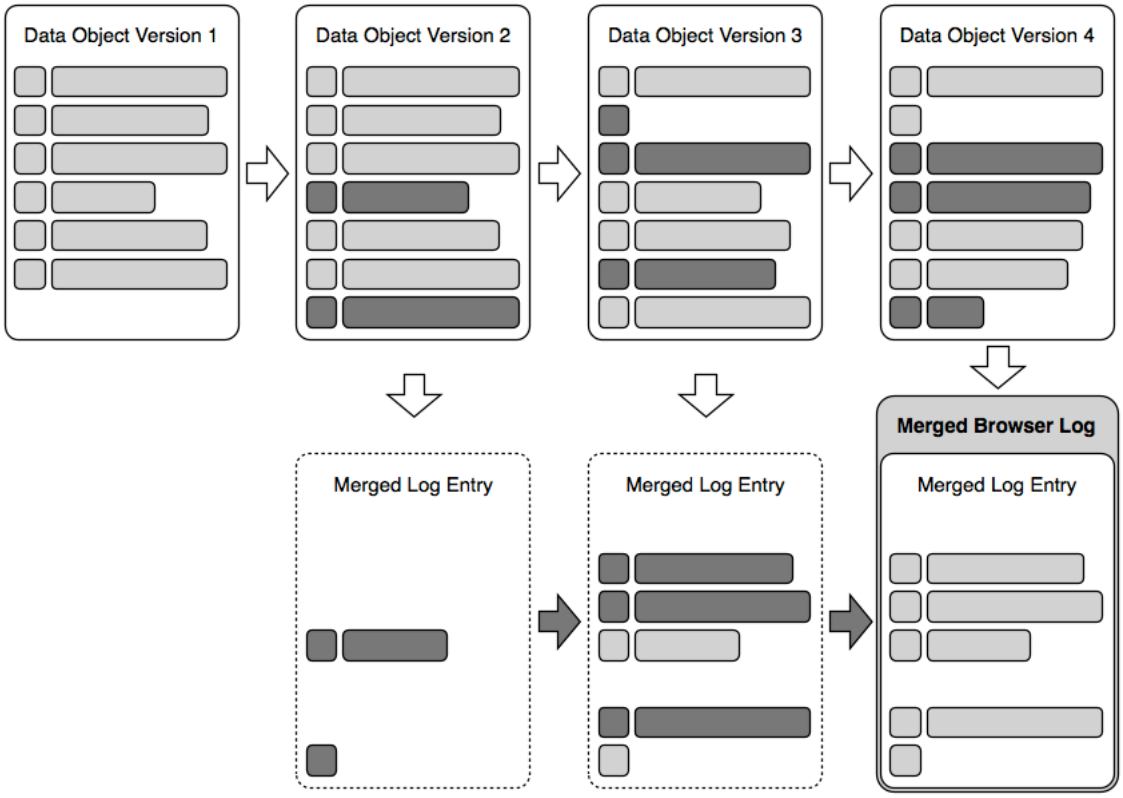


Figure 6: In a merged browser log updates to a data object are merged into a single log entry (dark colored bars represent changed data attributes). This log entry contains the original value of every updated attribute in the object. New updates cause new original values to be recorded in the log entry (the dark colored bars), unless they were recorded previously, until the log entry contains all original values of an object. A merged log entry, and therefore the merged browser log, grows slower than an incremental log.

3.4 Eager conflict resolution

The conflict resolution strategy we propose attempts to reconcile conflicting data versions by creating a new data version that has received all data updates that gave rise to both conflicting versions. This new reconciled data version is generated by for instance re-executing the updates in a conflicting local version of the data on top of the master data version it conflicts with (see figure 7). After this re-execution we know that the resulting data version has received all updates from the master version of the data, as it started off as this master version. We also know it has received the updates that gave rise to the local version of the data, because these are the updates that have been re-executed. Therefore we can consider the resulting data version our desired reconciled version.

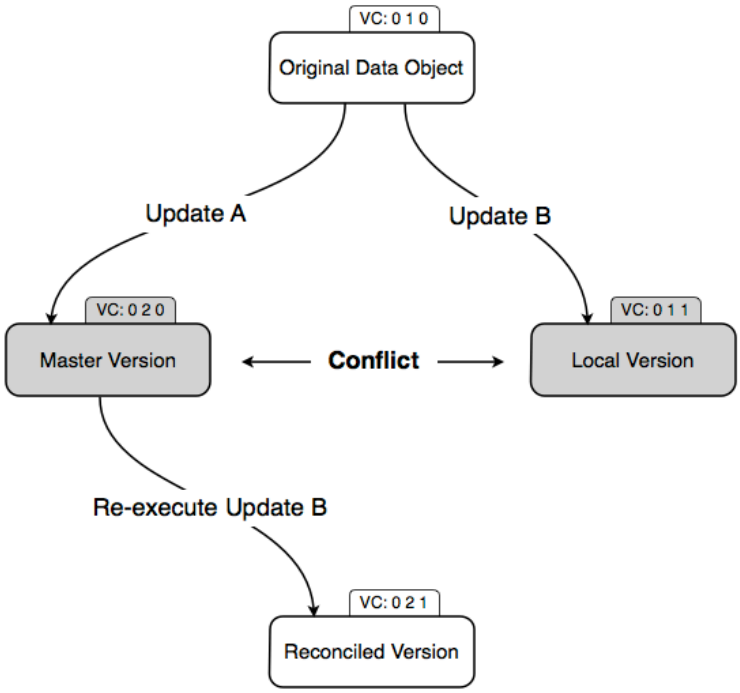


Figure 7: Conflicts are resolved by re-executing the updates that gave rise to a conflicting data version on top of the data version it conflicts with. For instance the updates in a local data version are re-executed on top of the master version of the data. The data version that is the result of this process is considered the reconciled data version, as it has received the updates that gave rise to both conflicting data versions. This reconciled data version can in turn be proposed to a base node.

It can happen that the context of a local update does not match the master version of the data, because this master version diverges too much from the data version the local update was originally applied to. In such case the local update cannot be re-executed on the master version and the reconciliation process is reversed.

In our example this means the updates that gave rise to the master version of the data are re-executed on top of the local version of the data (see figure 8). The resulting version has received all updates in the local data version, as it started off as this version. It also has received all updates in the master version of the data, because these are the updates that were re-executed. Therefore we can again say that the resulting version is our desired reconciled data version.

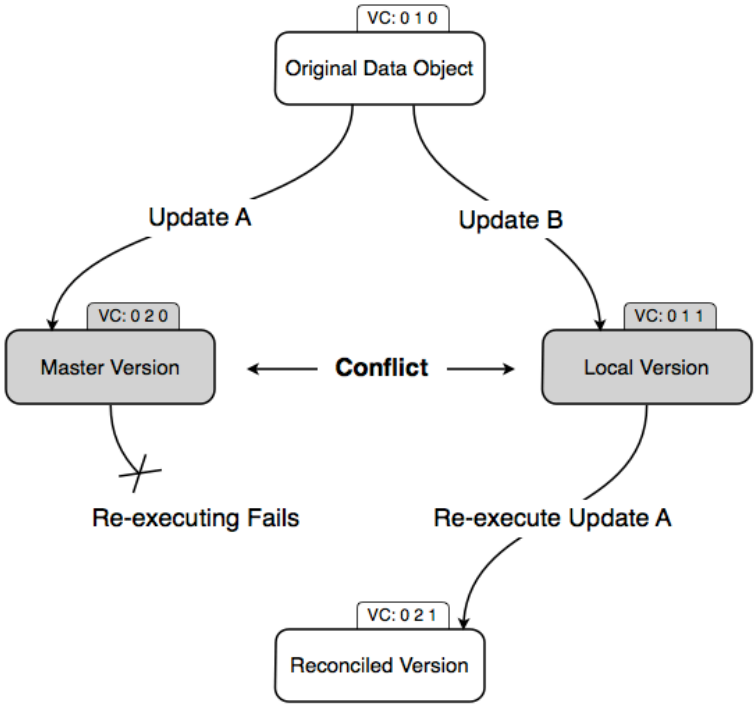


Figure 8: Re-executing a local update on a master data version can fail when the context of the local update does not match the master version. In such case the reconciliation process is reversed. The updates in the master version are then re-executed on top of the local data version. The data version that results from this reversed process can be considered the reconciled data version as well, because it has received the updates from both conflicting data versions too.

We provide two implementations of this update re-execution process, one based on serialized data versions and one focusing on individual data attributes. The reason for this double implementation is that patching serialized data objects, though simple to implement given the right tools, potentially generates mangled data which cannot be deserialized into a valid data object any more. Patching individual attributes solves this deserialization problem, but requires more complex code on the client-side. Comparing both implementations hopefully gives more insight in how frequently a conflict resolution process results in mangled data when using the serialized data implementation and whether the attribute oriented approach resolves this potential problem without increasing the performance overhead or reducing the average number of successfully resolved conflicts.

Upon a local data update the serialized data implementation converts the original and updated data version into a stringified JSON representation [48]. Subsequently using the diff-match-patch library [44] a so-called diff is generated from these string representations, which captures the differences between the original and locally updated data versions. The diff is then stored in the browser log for later conflict resolution. The browser log is thus a collection of diffs representing updates on persisted data objects. When a conflict is detected, all diffs that were collected for the affected data object since the last synchronization session are converted into patches and are successively applied to the conflicting remote version of this data object. This process results in a reconciled data version which has received the remote updates, since it is based on the remote version, and the local updates, through the applied patches.

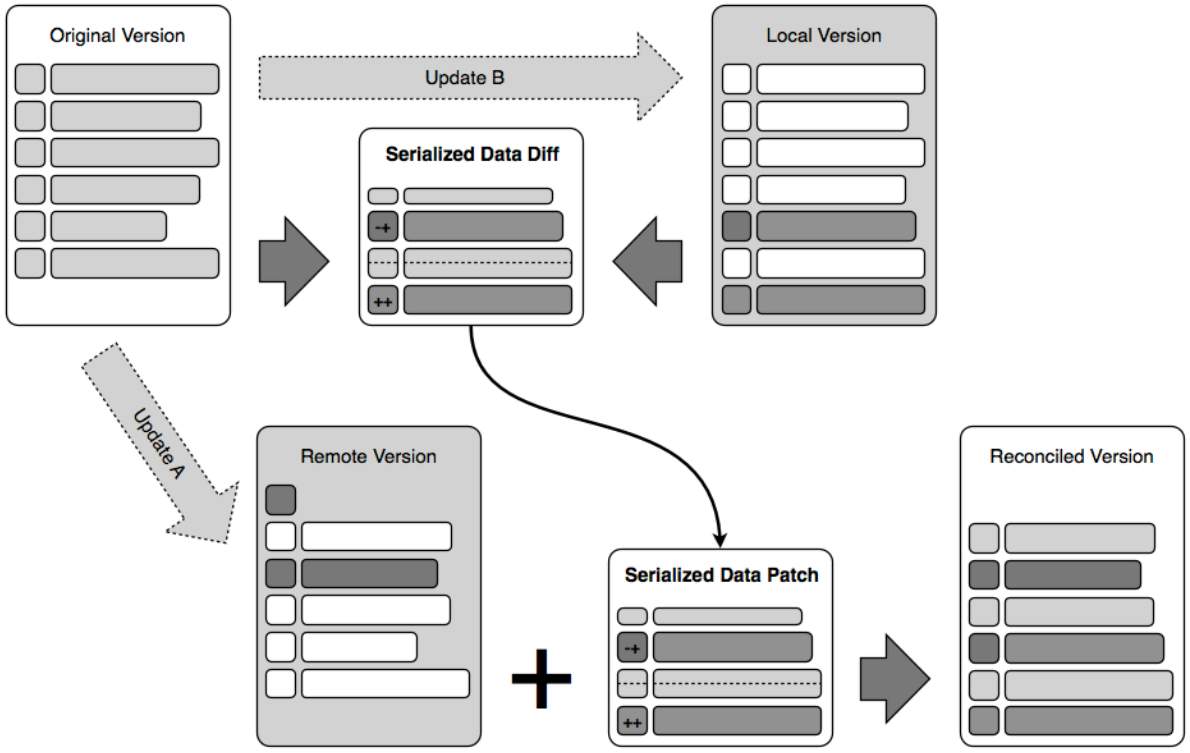


Figure 9: Our first implementation generates a diff between the serialized original and locally updated versions of a data object (dark colored bars represent changed data attributes). When a conflict is detected this diff is converted to a patch that is applied to the serialized remote version of the affected data object in order to generate a reconciled version.

When data objects are locally updated under the attribute oriented implementation, the original values of the affected data attributes are recorded in the browser log. When a conflict is detected, for each changed attribute a diff is generated from its original and current value (see figure 10). The total collection of diffs represents the local updates performed since the last synchronization session. To resolve the conflict, the corresponding attributes are patched in the remote data version. This results in a reconciled data version, which has received the remote updates, as it starts off as the remote version, and the local updates, as these are applied through the attribute patches performed on the remote version.

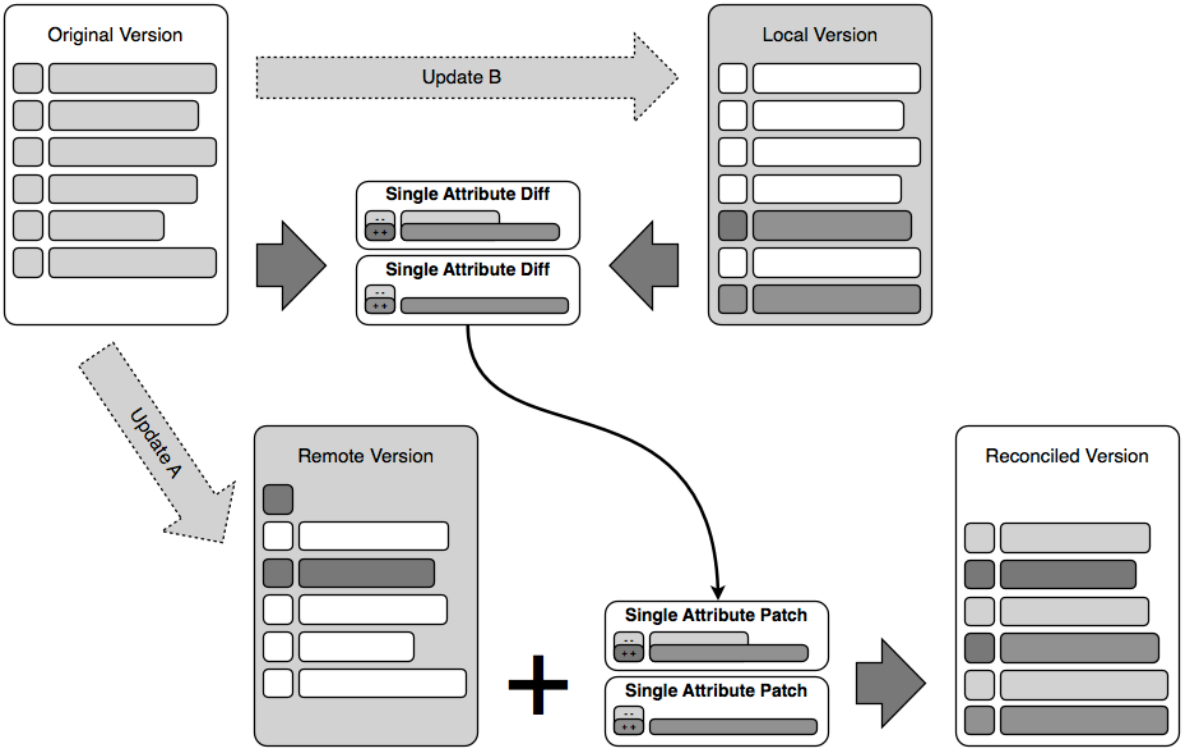


Figure 10: Our second implementation generates a separate diff for each attribute in the local data version that differs from the original data version (dark colored bars). The diffs are then applied as patches to the corresponding attributes in the remote data version in order to obtain a reconciled version.

In this section the concepts behind our mobile data replication system for offline web applications were discussed. We shed light on the overall architecture and explained how the optimizations are realized. In the next section we go into more detail about the methods we use to profile the network load, memory footprint and performance of our system.

4. Methods

There are two methods to find out whether our optimizations yield the improvements in network load, memory footprint and client performance we expect. Either we can determine the time and space complexity of the individual data replication routines we implement, or we collect empirical data while having a dummy web application replicate dummy data. In this project we go for the empirical approach because we believe it gives better insight in how an offline web application in its whole behaves when replicating data. In the subsections below we successively discuss the setup used for profiling *preventive reconciliation*, the *merged browser log* and our *eager conflict resolution* approach. For more details on the actual dummy data used in the benchmarks we refer to appendix B.

4.1 Network load: traditional vs. preventive reconciliation

We expect our *preventive reconciliation* solution to decrease the network load compared to traditional synchronization in the case of resolving conflicts. In addition we know that our approach induces some overhead during synchronization of non-conflicting data versions, say when a new object is created or an existing object is updated without conflict. To fairly evaluate our network load optimization we have to benchmark traditional synchronization and *preventive reconciliation* for both conflicting and non-conflicting cases.

Our benchmarking setup involves a base node, the web server, and a pair of mobile nodes, the web application instances running in the browser. In the benchmark for replicating a newly created data object we have a mobile node create a data object and then measure the time it takes for a traditional synchronization process to replicate this object (see figure 11). More specifically we start a timer before the

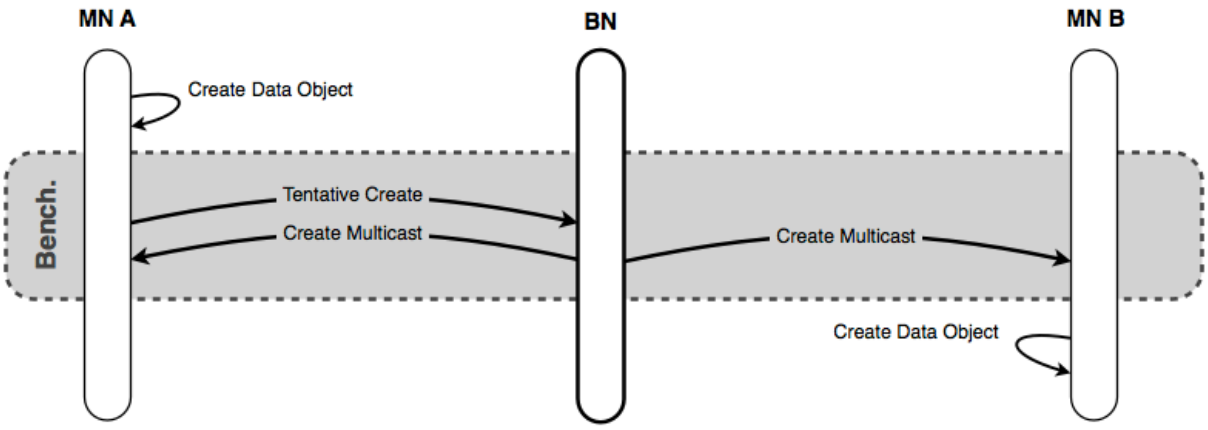


Figure 11: Benchmarking the traditional synchronization of a newly created data object. The first mobile node (MN A) starts by creating a data object. The benchmark (Bench.) starts when this data object is sent to the base node (BN) and it stops when the first as well as the second mobile node (MN B) has received a multicast from the base node.

data object is sent to the base node. The base node then checks for conflicts, stores the data when there are none and sends a message to all mobile nodes. The timer is stopped when all mobile nodes have received this multicast message.
For *preventive reconciliation* a similar process takes place (see figure 12). One of the nodes starts by creating a new data object. The timer starts when the vector clock for this object gets sent to the base node. The base node checks whether it conflicts with the master version of the data and reports the result back to the mobile node. When there are no conflicts, this node in turn sends over the actual data. The base node stores the data and sends out a multicast message. The timer is stopped when every mobile node has received this multicast message.

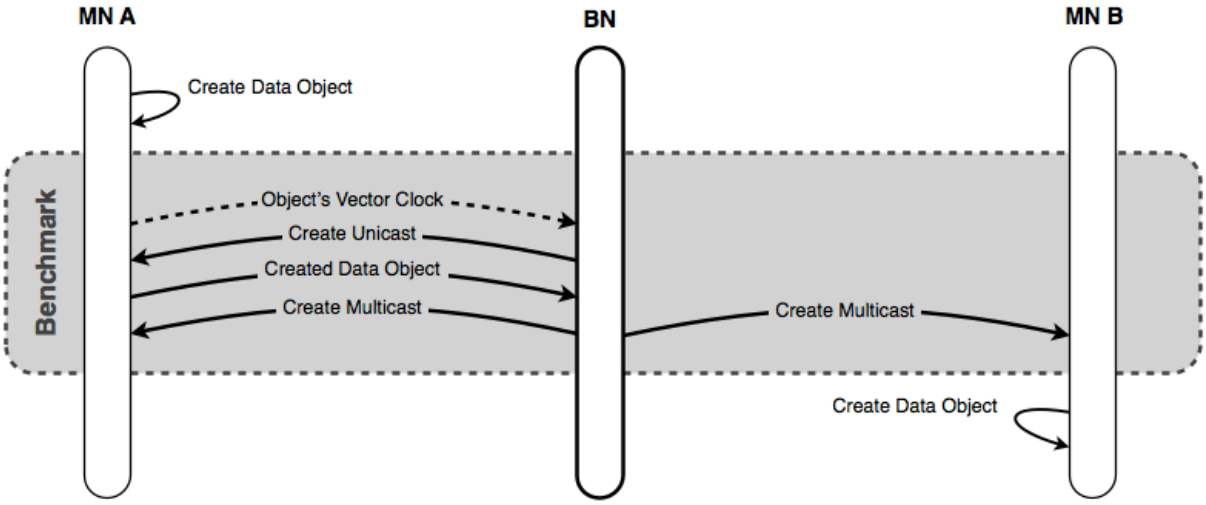


Figure 12: Benchmarking the replication of a newly created data object using preventive reconciliation. The first mobile node (MN A) starts by creating a data object. The benchmark starts when the vector clock for this data object is sent to the base node (BN). The benchmark stops when the first as well as the second mobile node (MN B) has received a multicast from the base node.

The case of updating an existing data object is an extension of the ones described above. We start by measuring the time it takes to replicate a local update using traditional synchronization. One of the mobile nodes creates a data object and sends it to the base node for replication. Once every mobile node has received the base node's multicast message for the creation of this object, the mobile node that created it performs a local update. The timer is started when the updated data gets sent to the base node for replication and it stops when every mobile node has received a multicast message for the update (see figure 13).

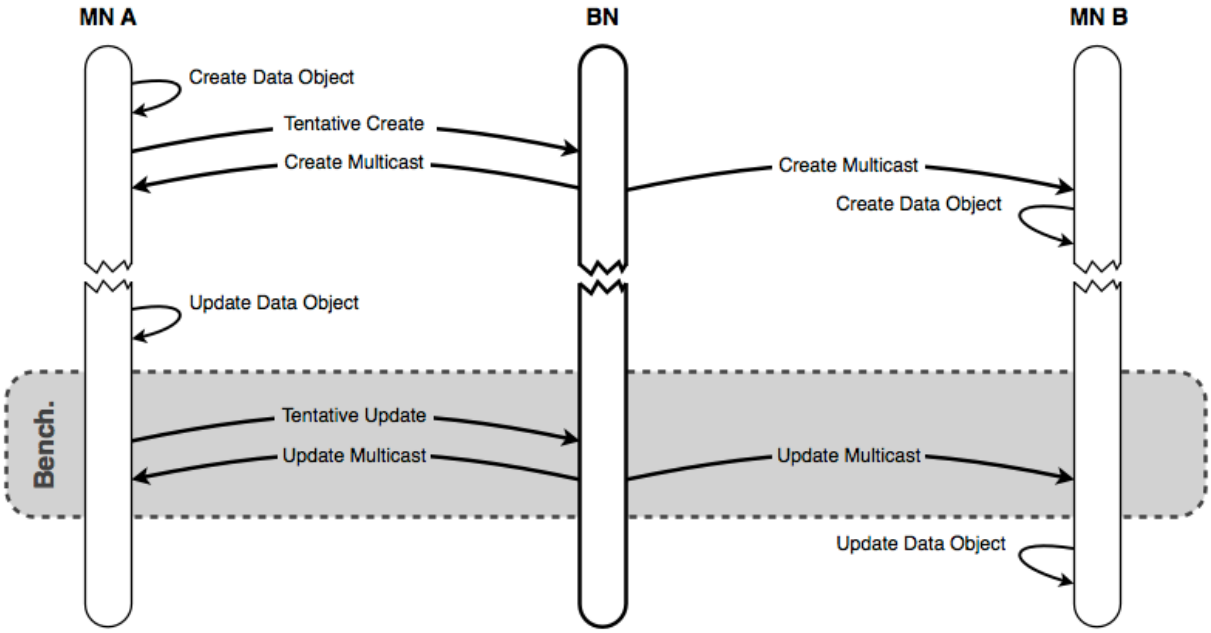


Figure 13: Benchmarking the traditional synchronization of an updated data object. The first mobile node (MN A) starts by creating a data object and synchronizes it to the base node (BN). After the data is stored and a multicast is done, the first node performs a local update to the same object. The benchmark (Bench.) starts when the updated data gets sent to the base node and it stops when the first as well as the second mobile node (MN B) has received the base node's multicast message for this update.

When using *preventive reconciliation* the timer is started when the vector clock of the updated data object is sent to the base node (see figure 14). The base node checks for conflicts and gets back to the mobile node that proposed the update. The mobile node sends the updated data in turn to the base node, which stores it. The timer is stopped when all mobile nodes have received the base node's multicast message for the update.

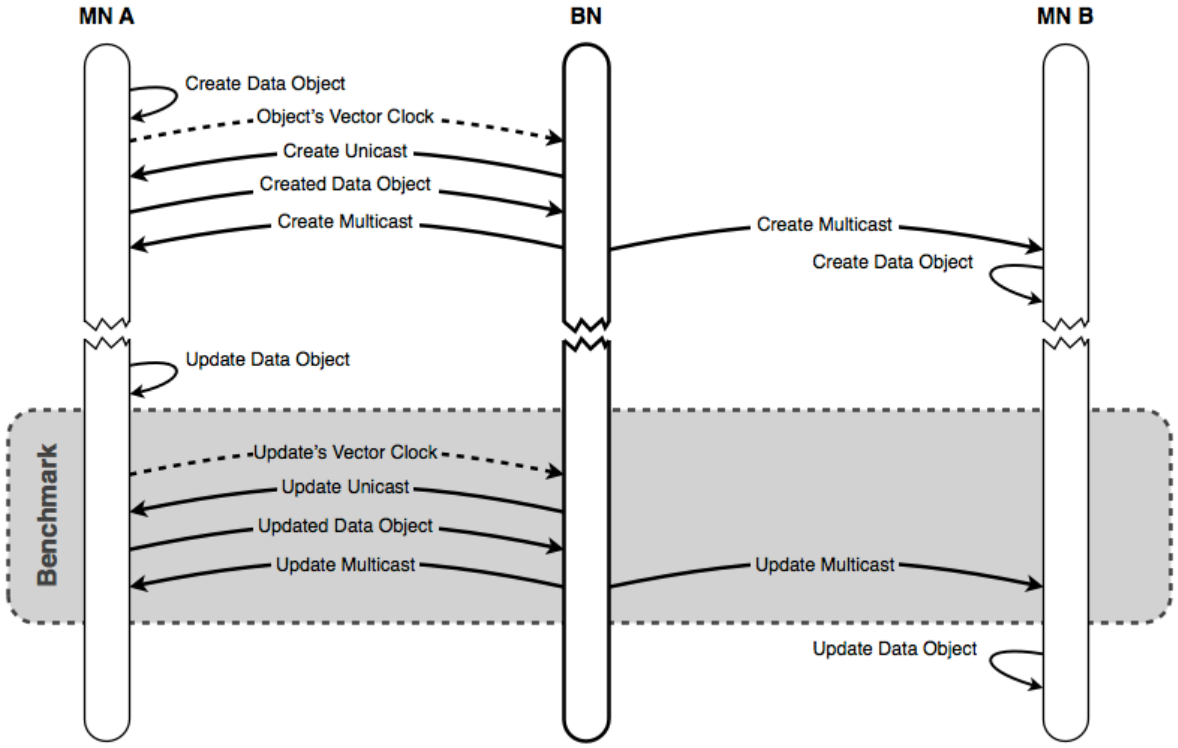


Figure 14: Benchmarking the replication of an updated data object using preventive reconciliation. The first mobile node (MN A) starts by creating a data object and synchronizing it to the base node (BN). The base node stores the data and multicasts it to all mobile nodes. After that the first mobile node performs a local update on the same object. The benchmark starts when the vector clock for this update is sent to the base node. The benchmark stops when the first as well as the second mobile node (MN B) has received the base node's multicast message for this update.

For measuring the duration of detecting and resolving a conflict in a synchronization session, we build upon the cases discussed above. One of the mobile nodes creates a data object, synchronizes it, performs a local update to it and synchronizes this update. The difference with the previous cases is that the multicast message for this update is never received by one of the mobile nodes because it is offline. This offline node in turn updates the same data object. Because this update is based on an outdated data version, it conflicts with the previous update.
In the case of traditional synchronization the timer is started when the offline mobile node comes back online and proposes the conflicting update to a base node. The base node detects the conflict and sends back the master version of the data. The mobile node resolves the conflict and sends the reconciled update to the base node. The timer is stopped when every mobile node has received the base node's multicast message for the reconciled update (see figure 15).

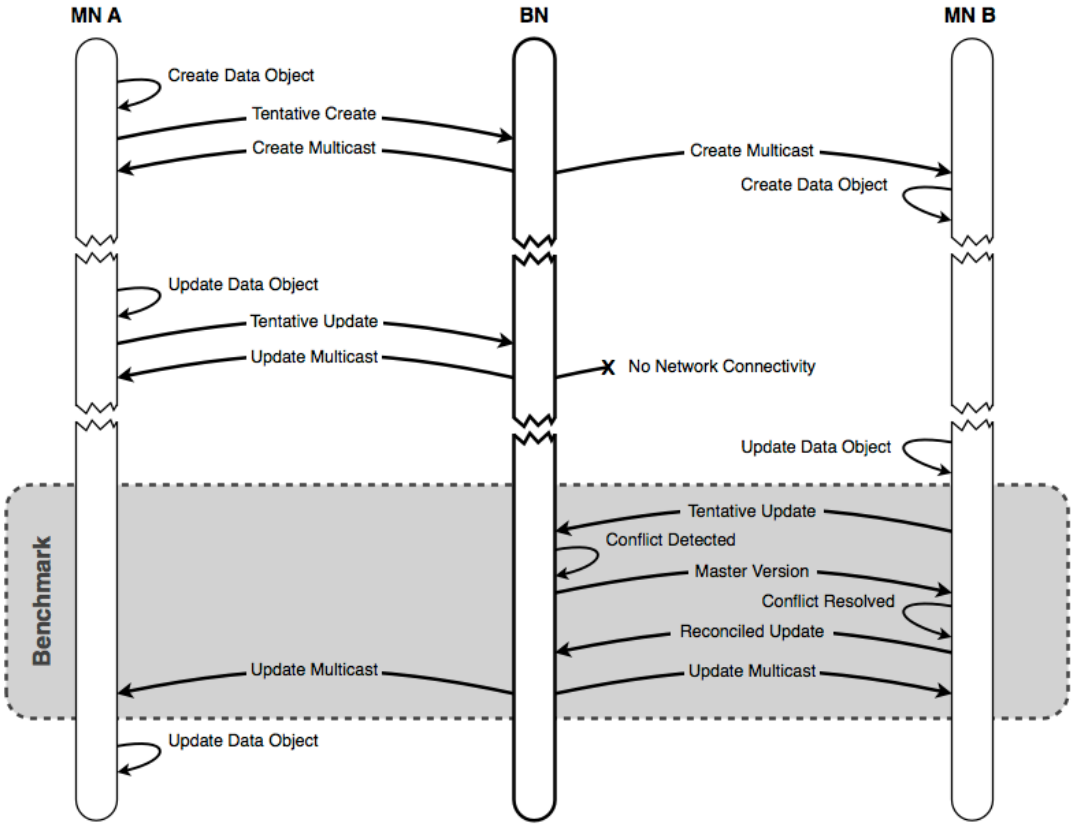


Figure 15: Benchmarking the detection and resolution of a conflict when using traditional synchronization. The first mobile node (MN A) updates a data object and synchronizes the update with a base node (BN). The second mobile node (MN B) misses the multicast for this update, because it is offline, and in turn updates the same data object. The benchmark starts when this second mobile node proposes its update to the base node. The base node detects a conflict and reports it to the second mobile node by sending the master data version. The second mobile node resolves the conflict and sends the reconciled update back to the base node, which sends out multicast messages for the update. The benchmark stops when all mobile nodes have received this message.

In the case of *preventive reconciliation* the timer is started when the offline mobile node comes back online and sends the vector clock for its update to the base node. The base node detects a conflict and reports it to the mobile node by sending the master version of the data. The mobile node resolves the conflict and sends the data of the now reconciled update to the base node. The timer is stopped when all mobile nodes have received the multicast message for the reconciled update.

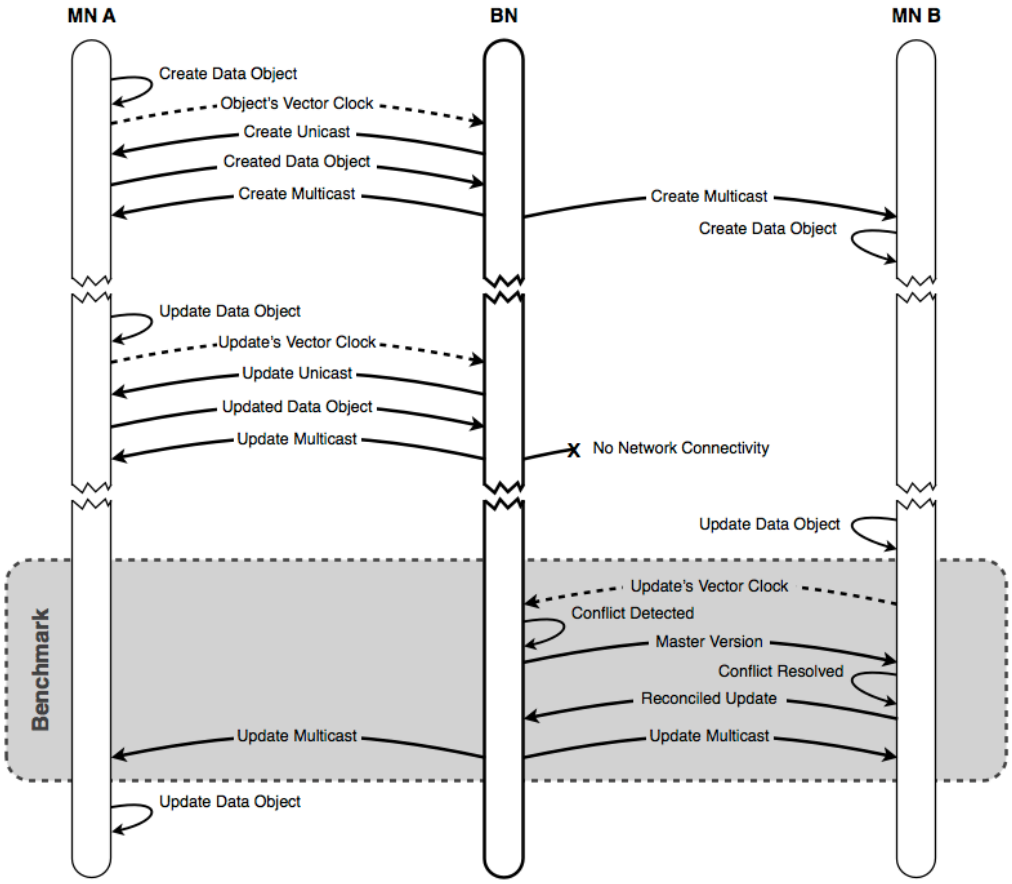


Figure 16: Benchmarking the detection and resolution of a conflict using preventive reconciliation. The first mobile node (MN A) updates a data object and synchronizes it with a base node (BN). The multicast message for this update is never received by the second mobile node (MN B), as it is offline. This node in turn updates the same object. The benchmark starts when it sends the vector clock of this second update to the base node. The base node detects a conflict and sends back the master version of the data. The second mobile node resolves the conflict and sends the data of the reconciled update to the base node. The benchmark ends when both mobile nodes have received the multicast message for the reconciled update.

In this subsection we explained how the network load of traditional synchronization and *preventive reconciliation* are measured. In the next subsection we go into more detail on profiling the memory footprint of an incremental and a *merged browser log*.

4.2 Memory footprint: incremental vs. merged browser log

The memory footprint of the browser log is measured by calculating its size relative to the original size of the data objects that are subject to the recorded updates. In the memory benchmarks random data objects receive different numbers of random updates with varying impact. This is done to gain more insight in how the memory footprint behaves when the number of local updates increases and larger portions of the data are affected by updates. In the benchmark for the incremental browser log for each sequence of updates the size of the log entries recording the performed updates is summed and divided by the original size of the data object they belong to (see figure 17).

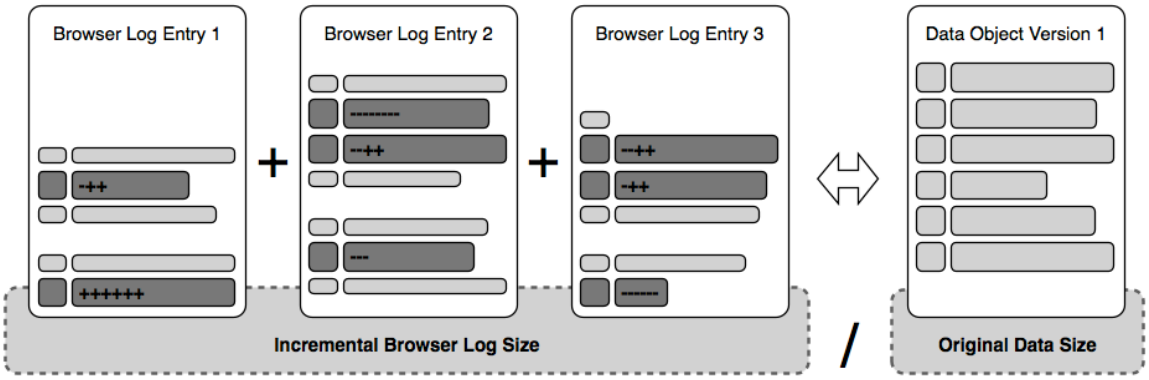


Figure 17: The size of an incremental browser log is measured by calculating the cumulative size of all log entries representing updates on a data object and dividing it by the size this data object had before all updates (dark bars represent updated attributes).

When profiling the memory footprint of the *merged browser log* for each sequence of updates the size of the merged log entry representing the updates is divided by the original size of the data object it belongs to (see figure 18).

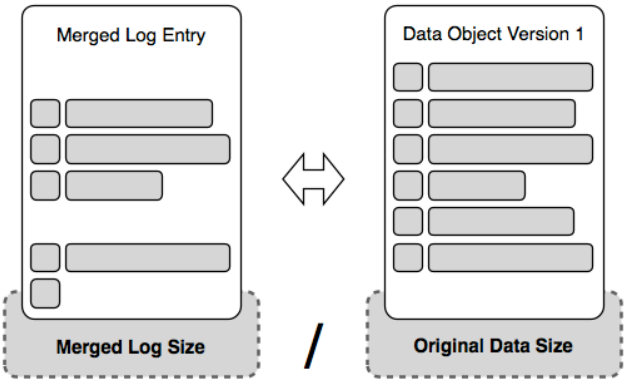


Figure 18: The size of a merged browser log is measured by dividing the size of the merged log entry representing updates on a data object by the size the data object had before the updates (bars represent updated attributes).

Now we know how to measure the memory footprint of our implementation, we continue by explaining how the client performance of our application is profiled.

4.3 Client performance: data versioning and conflict resolution

When it comes to profiling the performance of our *eager conflict resolution* approach, we are concerned about the performance overhead the distinct implementations impose on the browser while recording updates and reconciling conflicting data versions. In addition we are interested in the conflict resolution rate: the average proportion of conflicts that is successfully resolved. We start by discussing the benchmarking setup for performance overhead and continue with our approach to estimating the conflict resolution rate after that.

Performance overhead

Our system performs two main tasks on the client side: recording data changes when objects are updated locally and resolving conflicts during a synchronization session. We use separate benchmark sets for profiling each of these tasks.

To measure the performance overhead of recording local updates when using the serialized data implementation, we record the time it takes to serialize the original and updated version of a random data object, generate a diff of these serialized data versions and store it in the browser log (see figure 19). We do this for updates that change different proportions of a data object in order to get information on how the impact of an update affects the performance overhead.

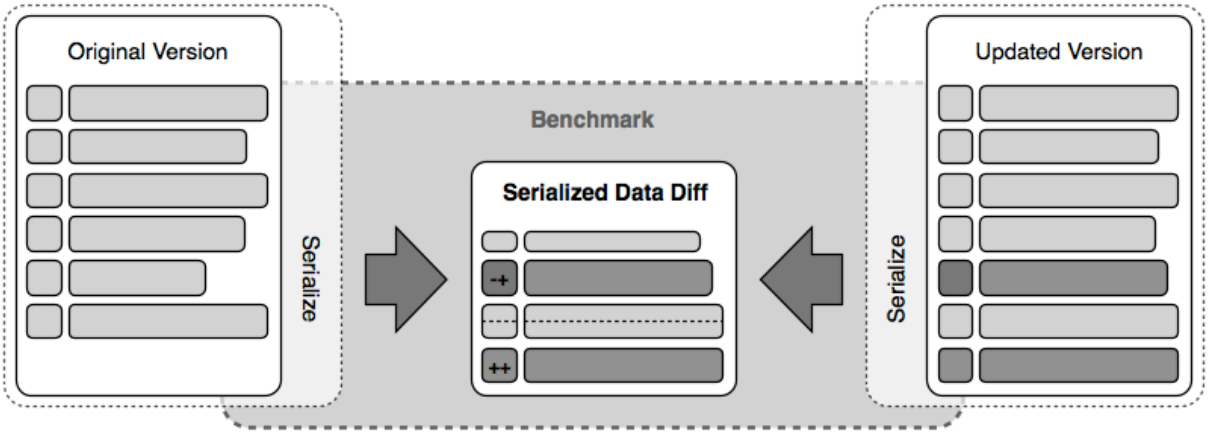


Figure 19: In order to profile the performance of data versioning using our serialized data implementation, we measure the time it takes to serialize the original and updated version of a random data object, generate a diff of these serialized data versions and store it in the browser log (dark colored bars represent changed attributes).

In the case of the attribute oriented implementation we measure the time it takes to iterate over all data attributes and record the original values of the attributes that are affected by an update (see figure 20). This is done for updates of different impact. In the attribute oriented implementation the generation of diffs for each changed data attribute is deferred to the conflict resolution phase.

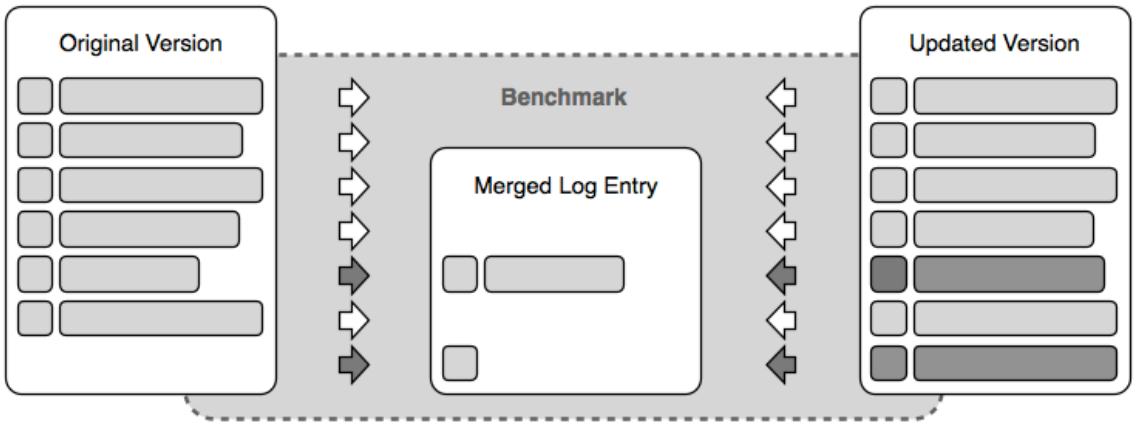


Figure 20: To profile the performance of data versioning using the attribute oriented implementation, we measure the execution time of iterating over all attributes and recording the original value of each attribute that changes between the original and updated versions of a random data object (dark colored bars represent changed attributes).

The performance profile of conflict resolution is captured by measuring the time it takes to reconcile conflicting random updates on random data objects. As discussed before we use updates of different impact to gain insight in how the performance changes when the number of attributes an update affects increases. In the case of the serialized data approach we measure the execution time used for serializing a conflicting data version, applying a patch to this version using the diff stored in the browser log and deserializing it to obtain the reconciled version of a random data object (see figure 21).

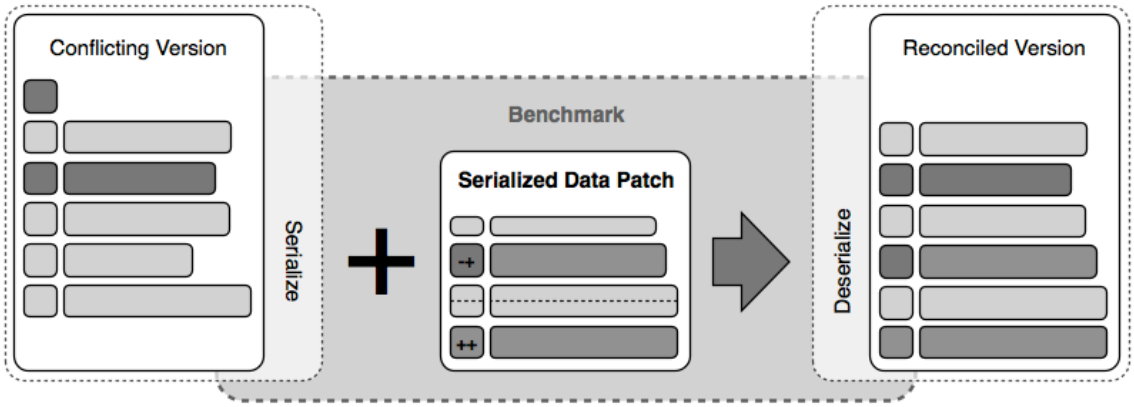


Figure 21: The performance of conflict resolution under the serialized data approach is measured by recording the time it takes to serialize a conflicting data version, patch it using a previously generated diff and deserialize it into the desired reconciled data version (dark colored bars are changed attributes).

For the case of the attribute oriented implementation we measure the time it takes to generate an attribute diff for each original attribute value recorded in the browser log and to apply these diffs to the attributes of a conflicting data version (see figure 22). Again we use updates affecting a different number of attributes.

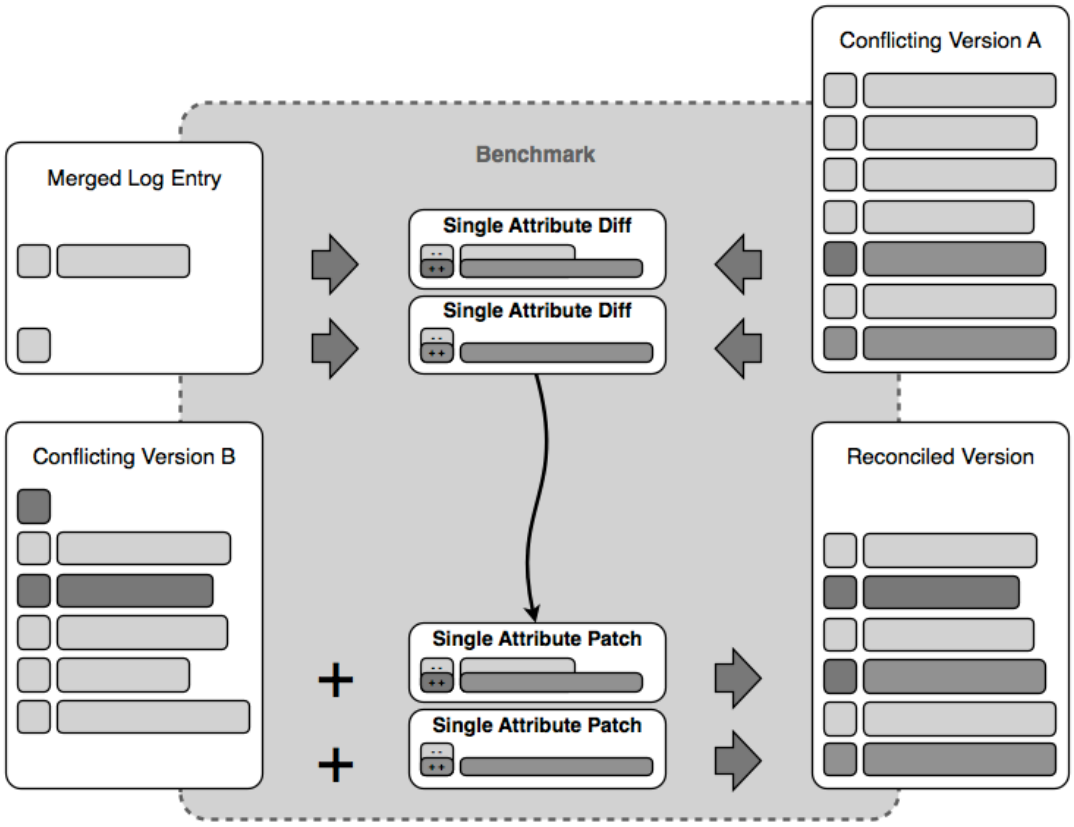


Figure 22: The performance of conflict resolution under the attribute oriented approach is measured by recording the time it takes to generate attribute diffs by comparing each data attribute recorded in the browser log to the data attributes of one conflicting data version (conflicting version A) and to apply these diffs to the attributes of the data version it conflicts with (conflicting version B), in order to obtain the desired reconciled data version (dark colored bars are changed attributes).

Conflict resolution rate

For a realistic estimate of the conflict resolution rate of our serialized data and attribute oriented *eager conflict resolution* implementation, we simulate realistic update conflicts in a benchmark and measure the proportion of conflicts that are successfully resolved. The data objects these updates are applied to, contain random attribute values of different type, among which human readable text, strings and numerical and boolean attributes. The updates used affect a random proportion of these attributes and a random proportion of an attribute's value.

There are two main events that may cause the reconciliation process to fail when using *eager conflict resolution*. The first event occurs when applying the patches of a conflicting update to the serialized data version it conflicts with generates mangled data that does not comply to the serialization format any more. In such case the

resulting data version cannot be deserialized into a valid data object and conflict resolution fails. To find out whether this event is a major cause of reconciliation failures when using the serialized data implementation, in our benchmarks we record the relative amount of reconciliation failures that are due to mangled data. The second event takes place when both conflicting updates modify the original data version to such extend that the contexts these updates were originally applied to mismatch the conflicting data versions. When this happens none of the updates can successfully be re-executed and conflict resolution fails as well. We expect the odds for this event to be connected to the number of data attributes an update affects, as the odds of a context mismatch increases with the amount of change in the data version containing this context. Therefore we include updates of different impact in our benchmarks.

All *eager conflict resolution* benchmarks have the same structure: given two conflicting updates the patch of the first update is applied to the result of the second update, either using serialized data versions or isolated data attributes. When this process fails, it is reverted: the patch of the second update is applied to the result of the first update (see figure 23). The relative number of successes after these two reconciliation attempts is then used as an indicator for the conflict resolution rate of a specific implementation.

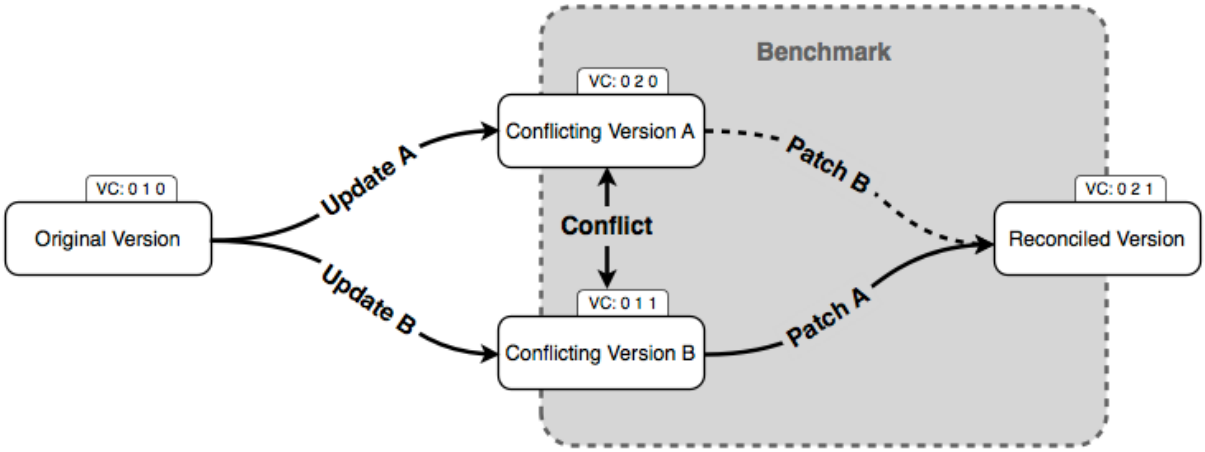


Figure 23: In order to reconcile a conflict, the patch of the first update (patch A) is applied to the result from the second update (conflicting version B). When this fails, the patch of the second update (patch B) is applied to the result of the first update (conflicting version A). The eager conflict resolution benchmarks consist of counting the relative number of successes after these two reconciliation attempts.

In this section we lined out the methods we use for profiling the network usage, memory footprint and client performance of the optimizations we propose for mobile data replication in offline web applications. In the next section we present the results of our benchmarks.

5. Results

Beneath we present the results from benchmarking the network usage, memory footprint and client performance of our optimized data replication system for offline web applications. All benchmarks are run in Google Chrome 24 up to 26 on an iMac with OSX Mountain Lion, a 2.4GHz Intel Core 2 Duo processor, 4GB 667MHz DDR2 SDRAM memory and a 16.8/0.8Mbps internet connection. For an overview of the raw benchmark results we refer to appendix B.

5.1 Preventive reconciliation

The final results from the network load benchmark show us the number of milliseconds it takes to synchronize a newly created, updated or conflicting data object with a base node in the network. When we look at the measurements for a data payload of 210KB (see figure 24), we can see that the number of milliseconds it takes to synchronize a newly created or updated data object is roughly the same in the the case of traditional synchronization and *preventive reconciliation*. The overhead for the latter is at most 1.7% of the total time consumed. The time needed for synchronizing a conflicting update however is reduced with 42% in the case of *preventive reconciliation*.

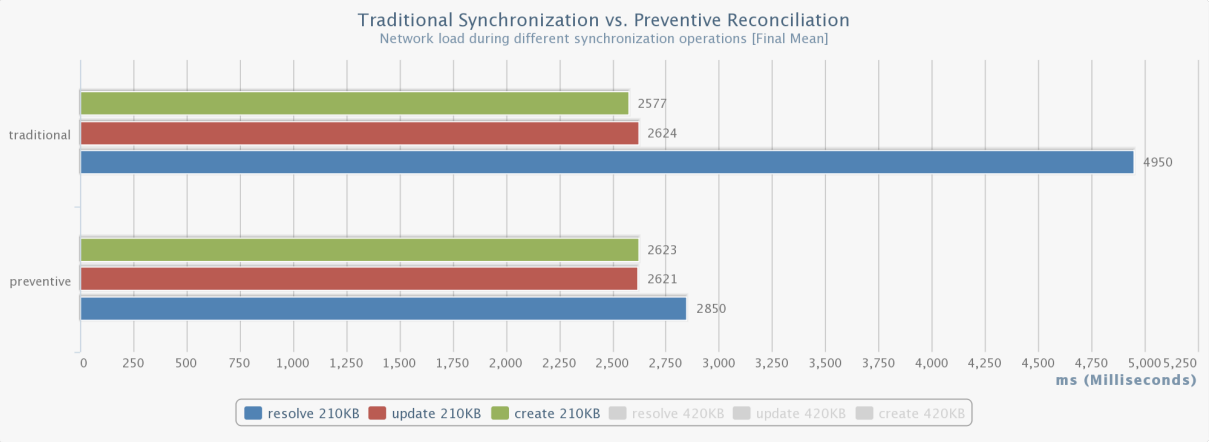


Figure 24: Benchmark results for traditional synchronization versus preventive reconciliation using a payload of 210KB. The results show that preventive reconciliation reduces the network load by 42% in the case of synchronizing a conflicting data object, while inducing only 1.7% overhead in the case of synchronizing a newly created or updated data object.

The measurements for a 420KB payload bear the same pattern. *Preventive reconciliation* induces a performance overhead of at most 0.5% in the case of synchronizing newly created or updated data objects, while it reduces the

synchronization time by 43% in the case of a conflicting update (see figure 25). As such the *preventive reconciliation* optimization behaves as we expected: it reduces the network load when synchronizing conflicting updates while it induces only limited overhead during regular synchronization sessions. The reason for the overhead being so small most likely is the fact that exchanging small messages between client and server using web-sockets has very little overhead compared to methods like HTTP [15, 49].

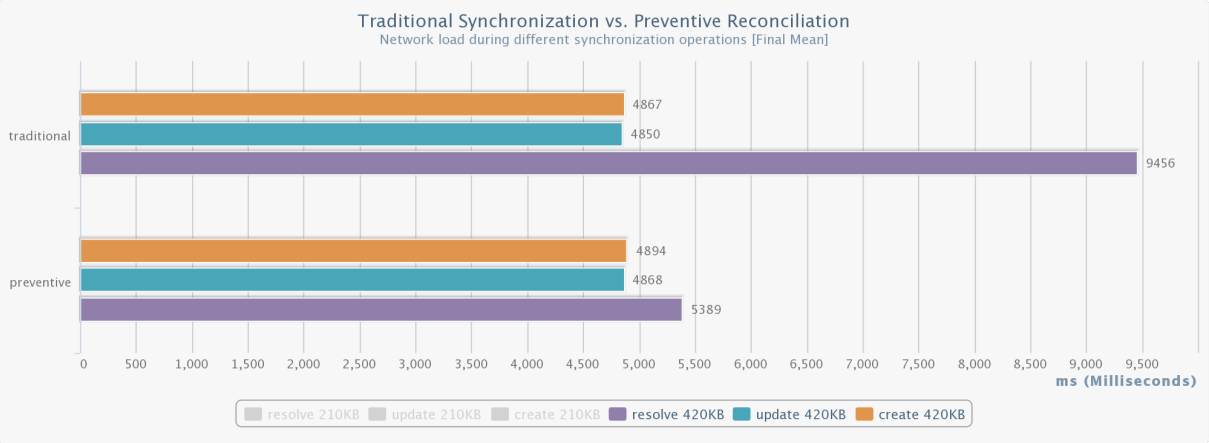


Figure 25: Benchmark results for traditional synchronization versus preventive reconciliation using a payload of 420KB. The results indicate that preventive reconciliation reduces the network load with 43% in the case of synchronizing a conflicting data object, while it induces only 0.5% overhead in the case of synchronizing a newly created or updated data object.

5.2 Merged browser log

The benchmark results for memory usage show us the proportion of memory consumed for recording local updates relative to the original size of the data objects subject to these updates (see figure 26). Below we successively discuss the benchmark results for updates affecting respectively 12%, 25% and 37% of the data objects. In the case of an incremental browser log recording updates that change 12% of a data object the memory used grows linearly with the number of updates applied. With every extra update about the same amount of memory is consumed for recording it. The memory footprint of a *merged browser log* recording the same type of updates seems to grow sub-linear however. It appears that with every increase in the number of updates a smaller amount of extra memory is used. Recording the first update consumes 26% of the memory used for storing the original data object. Recording the second update consumes only 18%, the third 17%, the fourth 11%, the fifth 7% and so on.

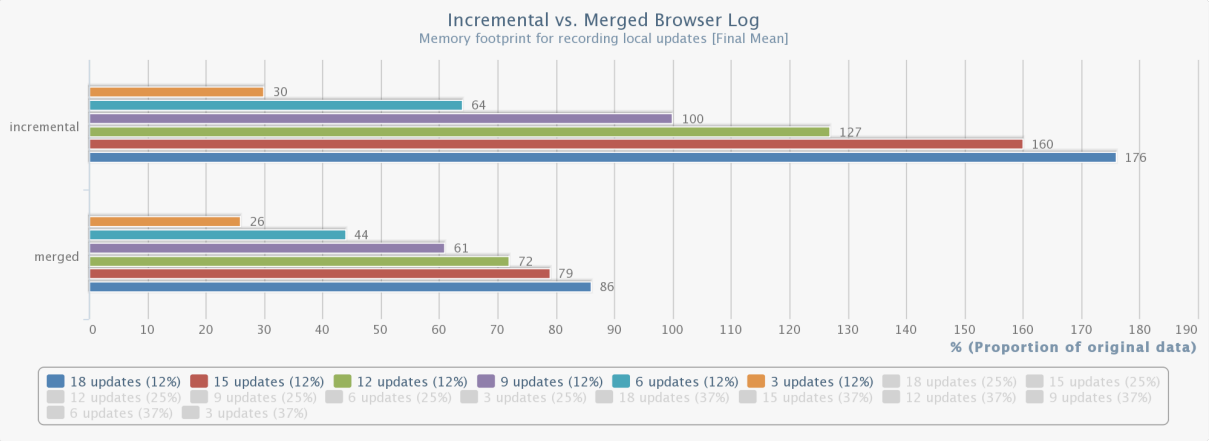


Figure 26: Benchmark results for incremental versus merged browser log recording updates that change 12% of a data object. In the case of an incremental browser log the memory used grows linear with the number of updates recorded. When using a merged log however the extra amount of memory needed for recording an additional update decreases with the number of updates.

These results confirm that the merged browser log consumes less memory than the incremental log. They do not show however what happens when its size exceeds the 100% threshold, which is the point at which we expect the log to contain almost all original attribute values. To find out, we take a look at the results for using updates that affect 25% of a data object's attributes. The memory footprint of the incremental

log still grows linearly with the number of updates (see figure 27). The size of the merged browser log now appears to converge to a value of approximately 100%, which is the original size of the data objects it records changes for.

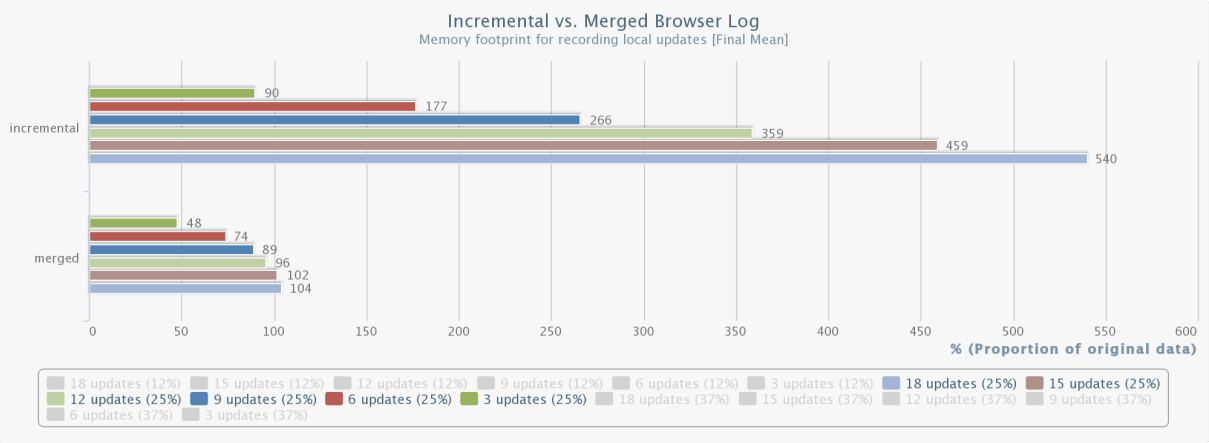


Figure 27: Benchmark results for updates that affect 25% of a data object. The size of the incremental browser log grows linear with the number of updates recorded. The merged browser log's size grows with decreasing steps when the number of updates increases.

In fact it does not really converge, but it only grows slowly with the number of updates from that point on, as we can more clearly see in the benchmark results for updates that affect 37% of the attributes (see figure 28). This slow linear growth is caused by recording the attribute keys of newly added attributes. Recording changed or deleted attributes does not cause an increase of the log any more, as all original attribute values are already recorded.

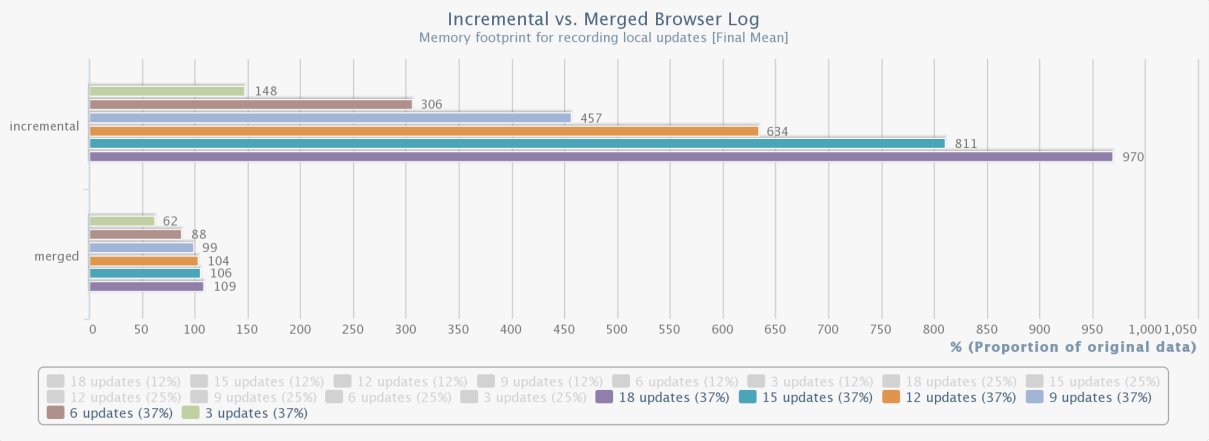


Figure 28: Benchmark results for updates that change 37% of the data objects. The size of the incremental browser log grows linear with the number of updates recorded. The merged browser log's size grows with decreasing steps until it exceeds the 100% threshold. Beyond that point the size shows a slow linear growth.

The memory benchmark results confirm our expectations about the memory footprint of the *merged browser log*. It consumes less memory than an incremental log and grows only slowly once it exceeds the original size of the data objects it records updates for.

5.3 Eager conflict resolution

We have two distinct result sets for our *eager conflict resolution* strategy: one concerning the performance overhead of recording local changes and resolving conflicts in the browser, and another one showing the proportion of conflicts that can successfully be resolved on average. We start by presenting the performance overhead results.

Performance overhead

The performance overhead benchmark results show us the number of milliseconds needed for recording a local update and reconciling conflicting updates using one of our *eager conflict resolution* implementations. The results for recording updates using the serialized data implementation show that the execution time grows linearly (maybe even super-linear) with the impact of the updates applied (see figure 29). The more attributes an update affects, the more time it takes to record and reconcile this update. This connection does not apply to the case of recording a local update with the attribute oriented implementation. There the execution time remains constant regardless the impact of an applied update. This difference does not come as a complete surprise as the attribute oriented implementation defers the generation of diffs between two versions of an attribute's value to the conflict resolution stage. We therefore expect to see more overhead at that point for this implementation variant.

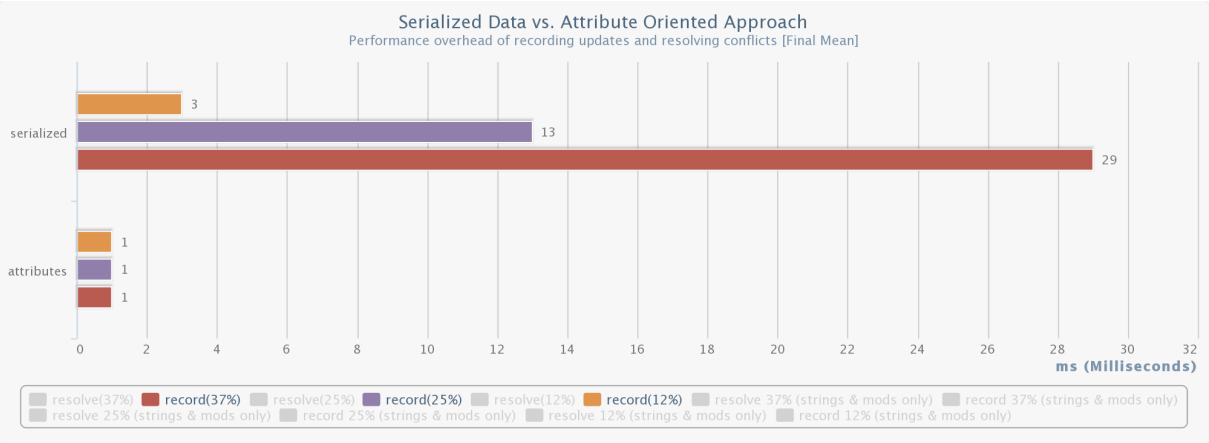


Figure 29: Benchmark results for the performance overhead of recording local updates using the serialized data and attribute oriented implementations. Under the first implementation the execution time grows with the impact of the updates applied. The execution time of the second implementation remains constant.

When we look at the results for reconciling conflicting data objects under the serialized data implementation, we again see the connection between execution time and the number of attributes an update affects (see figure 30). In addition the execution time seems to grow a little slower than in the case of recording updates. The results for the attribute oriented approach are surprising however. Though the execution times now do grow with increasing update impact, they are much smaller than those for the serialized data implementation. This is even more surprising given the fact that they comprise generating diffs and applying patches as opposed to only applying patches as in the serialized data case.

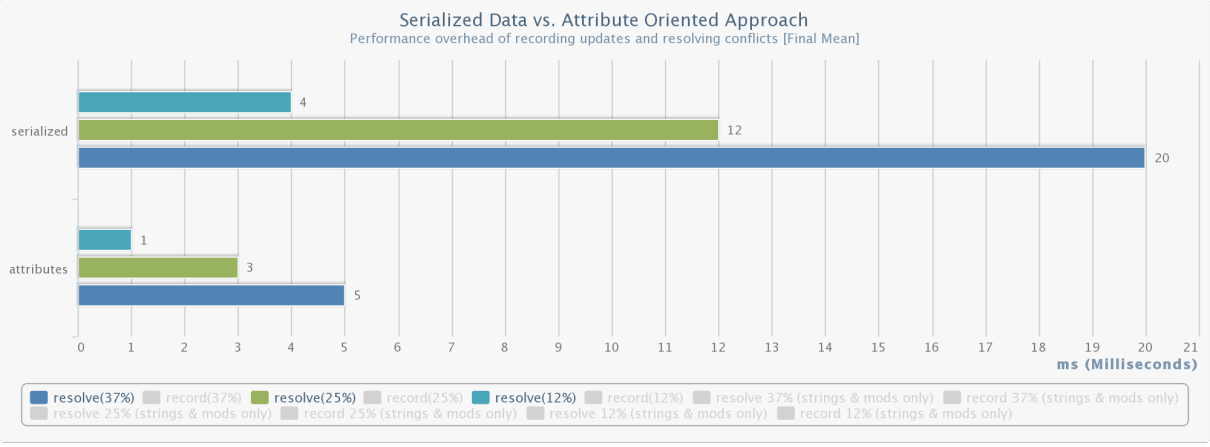


Figure 30: Benchmark results for the performance overhead of resolving conflicting updates using the serialized data and attribute oriented implementations. Both implementations have execution times that grow with increasing impact of the updates applied. The execution times for the attribute oriented implementation are smaller.

Given these benchmark results for the performance overhead of *eager conflict resolution* the question arises: what causes the differences between the two implementations? One explanation could be that recording and resolving numerical, boolean and newly added or deleted attributes is computationally simpler in the attribute oriented approach than in the serialized data approach. The rationale behind this idea is that the attribute oriented implementation treats every attribute in isolation and does not generate diffs and patches for these types of attributes, whereas the serialized data implementation does. To test this hypothesis, we repeat the performance overhead benchmarks using data objects that contain only string and text attributes and updates that only modify existing attributes, but do not add or delete attributes. If the hypothesis holds, the differences in performance overhead should vanish under this setup due to an increase in overhead for the attribute oriented implementation.
When we look at the results of the modified benchmarks for recording local updates we however see that the overall image remains the same. The execution times for the serialized data approach now are even greater, indicating that recording boolean, numerical and added or deleted attributes is easier than recording changed string or

text attributes under this implementation (see figure 31). These results do not support our hypothesis, as they contradict our expectation that recording boolean, numerical and added or deleted attributes is easier for the attribute oriented implementation only.

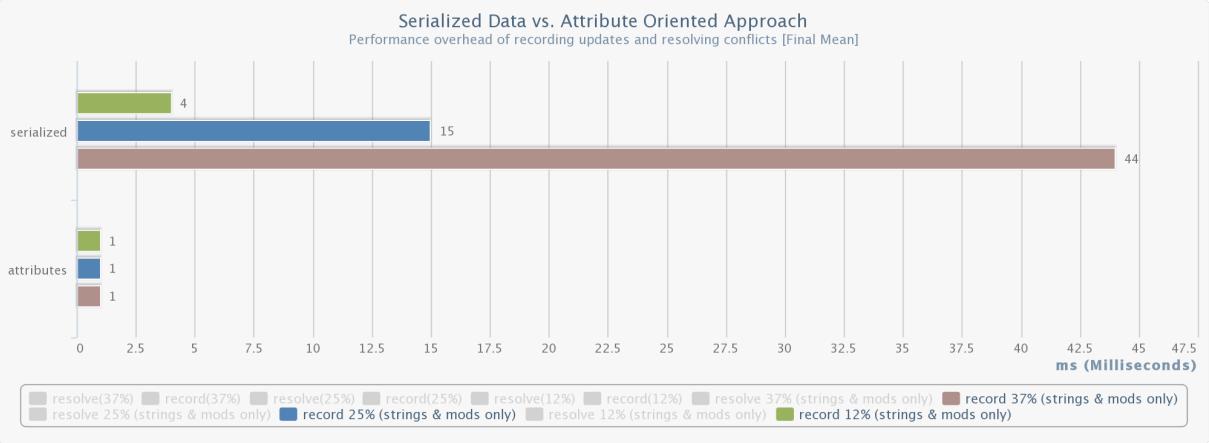


Figure 31: Benchmark results for the performance overhead of recording local updates while using only data objects containing string and text attributes and allowing only updates that change existing attributes. Recording changed string or text attributes is harder than recording boolean, numerical and added or deleted attributes under the serialized data approach.

The results for resolving conflicting updates also show the pattern we observed before. In addition all execution times are greater (see figure 32). This means that for both implementations resolving boolean, numerical and added or deleted attributes is easier than resolving changed string or text attributes. Again this result does not support our hypothesis and we have to reject it. This means that the performance difference between both implementations has nothing to do with the type of the attributes or whether they are newly added, deleted or only changed.

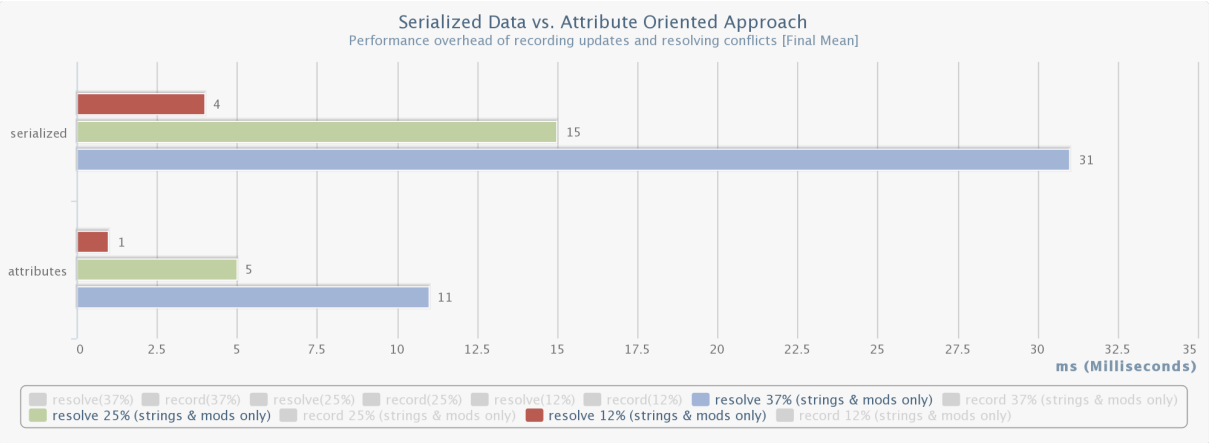


Figure 32: Benchmark results for the performance overhead of resolving conflicting updates while using only data objects containing string and text attributes and allowing only updates that change existing attributes. Resolving data objects with changed string or text attributes is harder than those with boolean, numerical and added or deleted attributes under both implementations.

If not the attribute types or the type of change, then what causes the differences in performance overhead? There is another factor that influences the execution time of our implementation: the time complexity of the algorithm used for generating diffs. We use Myer's algorithm [47] which is shown to have a stochastic time complexity that is quadratic in the length of the generated diff. This explains why the serialized data implementation, which generates one big diff per data object, is much slower than the attribute oriented approach, which generates a set of smaller diffs per data attribute.

Conflict resolution rate

The conflict resolution benchmark results tell us the average rate at which random conflicts are successfully resolved using one of the *eager conflict resolution* implementations. When we look at the results for the serialized data implementation we can see that the success rate decreases quickly when the number of attributes changed by an update increases (see figure 33). This is unfortunate as it means many conflicts cannot be automatically resolved under this implementation. The serialized attribute implementation does remarkably better however. When dealing with conflicting updates that affect 12% to 25% of the original data versions, 99% to 90% of all conflicts are successfully resolved. In the case where 37% of the data attributes are affected by the conflicting updates, still 76% of the conflicts are successfully resolved.

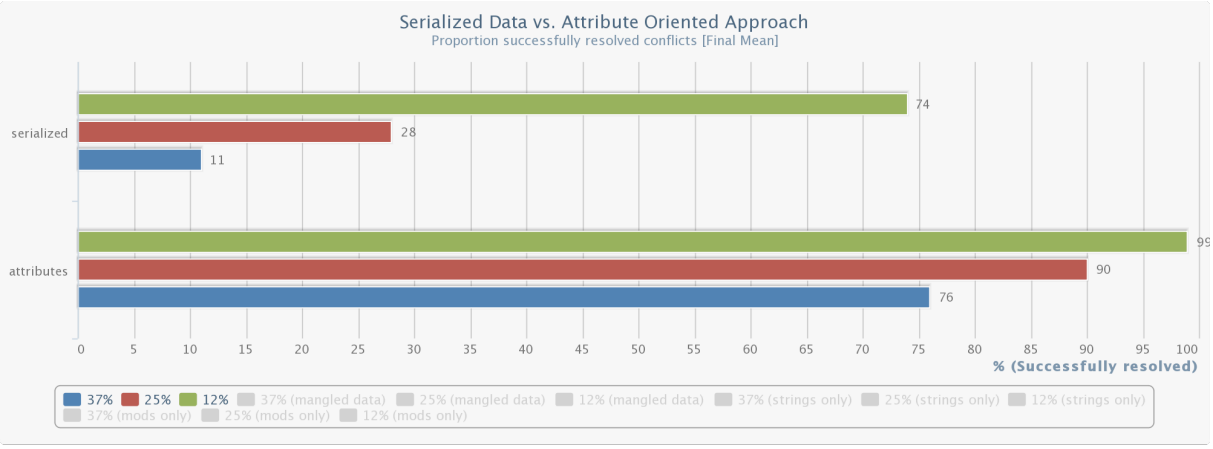


Figure 33: Benchmark results for conflict resolution using the serialized data and attribute oriented implementation respectively. When using the first implementation the success rate drops quickly when the impact of the updates increases. The second implementation performs better.

Now the question arises: is this difference in conflict resolution rate due to the serialized data implementation generating mangled data when reconciling two conflicting data versions? To find out take a look at the recorded relative number of

reconciliation failures that are due to mangled data (see figure 34). The results show that generating mangled data only accounts for 0.4% to 0.8% of all reconciliation failures. This means it is not a major cause of failure and cannot explain the differences in conflict resolution rate between the two implementations.

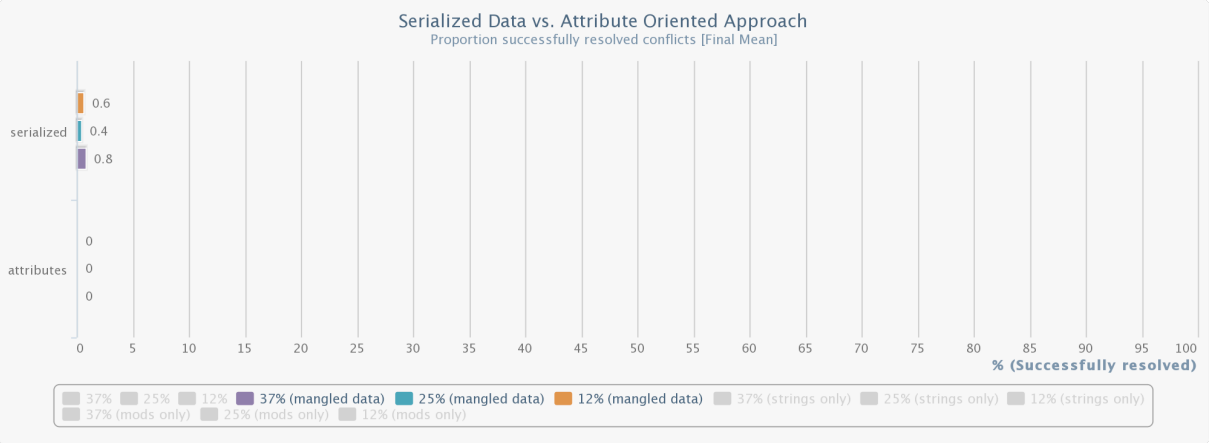


Figure 34: Benchmark results for recording the relative number of reconciliation failures due to generating mangled data. Generating mangled data is no major cause of reconciliation failures.

Another explanation could be that reconciling numerical and boolean attributes is easier and therefore more successful in the attribute oriented approach than for the serialized data approach. The rationale behind this idea is that the attribute oriented implementation treats every attribute in isolation and does not generate diffs and patches for these types of attributes, whereas the serialized data implementation does. To test this hypothesis we repeat the benchmarks for the conflict resolution rate while using data objects that only contain string and text attributes. If the hypothesis holds, the conflict resolution rate differences should vanish under this setup due to a decrease in the success rate of the attribute oriented implementation.

When we look at the results however, the same differences in conflict resolution rate emerge (see figure 35). This means that reconciling conflicting boolean and numerical attributes is just as hard or easy under both implementations. Therefore we reject our hypothesis. The difference between the conflict resolution rate of the two implementations is not caused by the type of the data attributes.

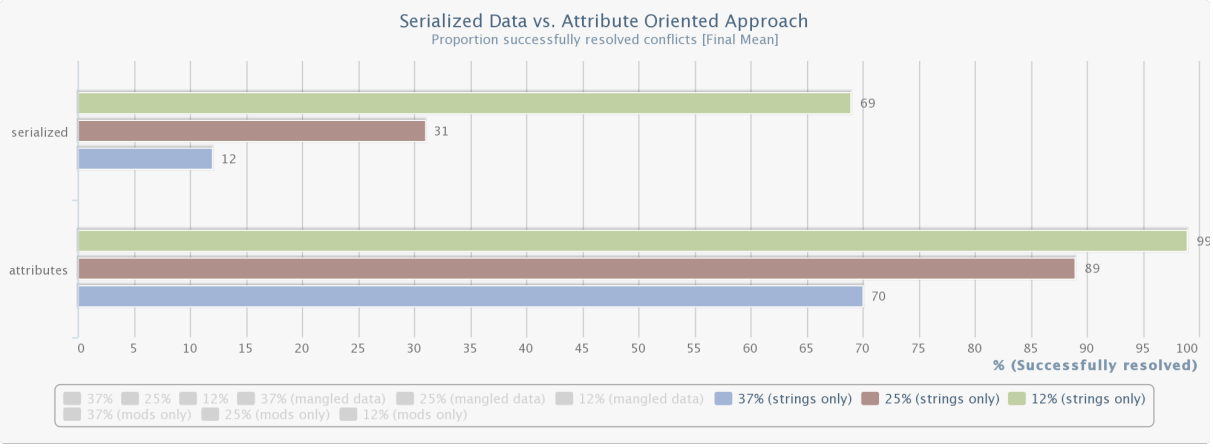


Figure 35: Benchmark results for conflict resolution using data objects containing only string and text attributes. The results are equal to those of the original benchmark setup for measuring the conflict resolution rate.

In the methods section we discuss a second reason for reconciliation failures: context mismatches. It is possible that there are more context mismatches in the serialized data implementation than in the attribute oriented approach because of updates adding and deleting data attributes. The rationale behind this hypothesis is that the serialized data implementation uses a single diff per object. The context of this diff may include attribute values that have been deleted or that were moved when an adjacent attribute was added. In these cases the diff's context does not match the data version it is applied to any more and reconciliation fails. These kind of mismatches do not happen in the attribute oriented approach where every attribute is patched in isolation. To check the validity of this hypothesis we repeat our benchmarks for the conflict resolution rate using updates that only change existing attributes, but do not add or delete attributes. If the hypothesis holds, we should see an increase in the success rate of the serialized data implementation.

The results for the adapted benchmarks now show more similar conflict resolution rates for both implementations (see figure 36). The changes in the outcome are due to both an increase in the success rate of the serialized data implementation and a decrease for the attribute oriented approach. Apparently added and deleted attributes are not only hard to handle for the serialized data implementation but also easy for the attribute oriented approach. This last fact is not a complete surprise, as the attribute oriented implementation does not generate diffs and patches for added or deleted attributes. In all the results support our hypothesis, so we accept it. This means that eager conflict resolution under the serialized data implementation is susceptible to context mismatches when conflicting updates add or delete data attributes.

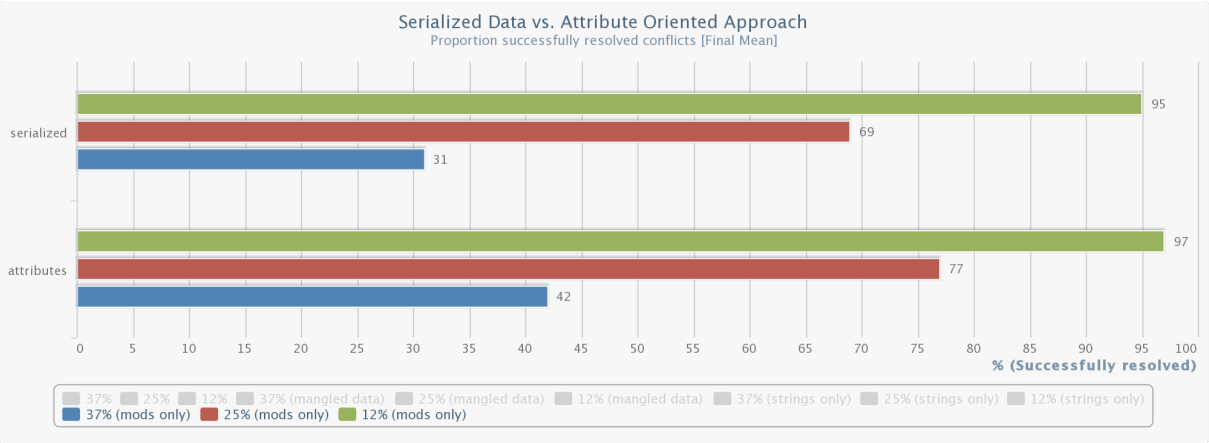


Figure 36: Benchmark results for reconciling updates that only change existing attributes. The success rate for both implementations are quite similar in this case.

In conclusion we can state that the serialized data approach to *eager conflict resolution*, though easy to implement, does not live up to our expectations for performance overhead and conflict resolution rate. Due to the quadratic time complexity of the diff algorithm used, the implementation does not scale well when the size of the data objects and the impact of the updates increases. In addition the reconciliation process is prone to context mismatches caused by updates adding and deleting attributes.
In contrast the attribute oriented implementation performs pretty well. Because it resolves conflicts on the level of attributes, its performance overhead is very limited. In addition it succeeds in resolving the majority of conflicts, even in cases where the conflicting updates modify a large portion of the original data version. We can say that using a more complex reconciliation implementation on the client side does pay off in terms of performance.

In this section we presented the results our benchmarks yield for the network usage, memory footprint and client performance of our data replication system for offline web applications. We contrasted the outcome with our expectations of the implemented optimizations. In addition we explored the causes of unexpected results where they arose. In the next section we discuss the meaning of the benchmark results and our work in general in the light of the current literature on mobile data replication.

6 Discussion and conclusion

In this section we discuss the results of profiling our optimized data replication system in the light of the current literature on mobile data replication. We successively treat *preventive reconciliation*, the *merged browser log* and *eager conflict resolution*. After that we come to a conclusion on the value of our work in the context of offline web application development.

6.1 Preventive reconciliation

As discussed in the related work section of this document, Cannon and Wohlstadter propose an interesting framework for data versioning and persistence in the browser [1]. When applied in the context of offline web applications however, this framework can be optimized in two ways. The first optimization concerns the network load when synchronizing a conflicting update. Because conflicts occur relatively often in offline web applications, it is important that they are resolved in a timely manner in order to keep the distributed database consistent [2]. Unfortunately synchronizing conflicting updates the way Cannon and Wohlstadter's framework does, induces undue network overhead and therefore delay to the reconciliation process. In their approach the client starts a synchronization session by sending the data of a local update to the server. When the server detects a conflict, it reports this back to the client, which resolves the conflict. The client then sends the data of the now reconciled update back to the server for replication. In the end the data of the conflicting update is sent over the network twice. We mitigate this problem using what we call *preventive reconciliation*. In this approach the client starts a synchronization session by sending only metadata to the server, which it uses to detect conflicts. When a conflict is detected, it is resolved by the client. Only then the client sends the data of the tentative update to the server. Eventually the updated data is sent over the network only once.

The benchmark results for network load confirm that *preventive reconciliation* reduces the network overhead by 42%. The overhead of sending an update's metadata at the start of every synchronization session is below 1.7%. Given these results we can state that *preventive reconciliation* allows us to resolve conflicts in a more timely manner without causing any real overhead for regular synchronization sessions.

6.2 Merged browser log

A second optimization to the framework of Cannon and Wohlstadter concerns the memory footprint of recording local updates. In their approach every local update is recorded separately in an incremental browser log which grows linearly with the number of local updates performed. Because offline web applications typically

accumulate many local updates before synchronizing with a server, this increasing browser log size is bound to clash with the strict size limitations browsers impose on their local data storage [12]. We address this problem by introducing a *merged browser log*. This log contains only one entry per data object, regardless the number of updates the object receives. With every update, the original value of an affected attribute is merged into the log entry for this object, unless it is already recorded. The benchmark results for memory usage show that the *merged browser log* has a smaller memory footprint than an incremental log. In addition the *merged browser log* grows only slowly once it exceeds the size of the data objects it records updates for. This is due to the fact that from that point on only attribute keys for newly added attributes are recorded. All original attribute values have already been stored. Therefore the *merged browser log* allows offline web applications to record more local updates before they bump into the size limits of the local storage.

6.3 Eager conflict resolution

In the related work section we present two approaches to automated conflict resolution. The first approach by Zhiming *et al.* compares the read set of conflicting updates to determine whether they can be reconciled [4]. When these read sets are not overwritten by one of the updates, their result sets are considered non-conflicting and can be applied on the server. Unfortunately this conflict resolution scheme is not applicable to offline web applications, as in these applications data modifications are typically performed by human beings. Therefore the read set of an update cannot be defined. In the second approach Phatak *et al.* allow the read set of conflicting updates to be overwritten, as long as their result sets are not changed by the update they conflict with [6]. Unfortunately again, offline web applications only detect conflicts which have a result set that is changed by both updates. These updates can therefore not be resolved using the reconciliation scheme of Phatak *et al*.
This prompts us to devise our own approach to conflict resolution in offline web applications, which we call *eager conflict resolution*. In this approach conflicts are resolved by re-executing the updates in one conflicting data version on top of the data version it conflicts with. When re-execution fails, the process is reversed: the updates in the last data version are re-executed on top of the first data version. The reconciled data version thus obtained has received updates of both conflicting data versions. We provide two implementations of this reconciliation process: one based on serialized data and one that is attribute oriented. The benchmark results show that the serialized data implementation does not perform well for large conflicting data objects containing many changes. In addition the implementation is susceptible to context mismatches when many attributes are added or deleted in conflicting data updates. The attribute oriented implementation yields better results however. Its performance overhead is small and it succeeds in resolving the majority of conflicts, even when conflicting data version are very different.

6.4 Conclusion

Our optimizations for mobile data replication bring many benefits for offline web applications. To begin with we significantly reduce the network overhead of synchronizing conflicting updates without causing any real overhead for regular synchronization sessions. This is important because in offline web applications data conflicts are more common than in regular applications and these conflicts have to be resolved in a timely manner in order to keep the distributed database consistent.
In addition our approach to recording local data modifications has a memory footprint that better fits the limitations modern browsers impose on local storage. Because less memory is consumed while a web application is offline, more local updates can be performed before the limits of local storage are reached and the application is required to start a synchronization session with a server.
Finally we provide a solution for detecting and resolving conflicts that may arise during a synchronization session. Because existing solutions for automated conflict resolution cannot be applied, offline web applications currently either rely on the end user for conflict resolution or have no support for conflict detection and resolution at all. The first case is undesirable because resolving conflicts can be a tedious and confusing task which often requires some technical knowledge. In the second case the recorded updates are applied in the order they are received on the server-side and conflicting updates are simply overwritten. Because there is no notion of conflict detection, it is not possible to store conflicting versions of the data for later recovery. When using our solution for automated conflict resolution the changes in two conflicting data versions are maximally preserved in a reconciled data version, which is then stored on the server. Note this does not mean that our reconciliation process never causes any data loss. It is advisable to always back up conflicting data versions on the server, even after a conflict is successfully resolved.

References

[1] Cannon, B. & Wohlstadter, E. (2010) Automated object persistence for JavaScript. Proceedings of the 19th international conference on World wide web. Raleigh, North Carolina, USA, ACM. pp. 191–200.

[2] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, pages 173–182, New York, NY, USA, 1996. ACM.

[3] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. ACM Trans. Comput. Syst., 10(1):3–25, 1992.

[4] D. Zhiming, M. Xiaofeng and W. Shan, (2002) "A transactional asynchronous replication scheme for mobile database systems", Journal of Computer Science and Technology, 17(4), 389 – 396.

[5] Martin Peters, Christopher Brink, Martin Hirsch, and Sabine Sachweh. 2011. A client centric replication model for mobile environments based on RESTful resources. In Proceedings of the Workshop on Posters and Demos Track (PDT '11). ACM, New York, NY, USA, , Article 22 , 2 pages. DOI=10.1145/2088960.2088982 http://doi.acm.org/10.1145/2088960.2088982

[6] S.H. Phatak and B.R. Badrinath, Multiversion reconciliation for mobile databases, in: Proceedings of the 15th International Conference on Data Engineering (March 1999) pp. 582–589.

[7] M. Pilgrim, HTML5: Up and Running. O'Reilly Media, 2010.

[8] http://www.w3.org/TR/html5/

[9] http://www.whatwg.org/

[10] http://www.w3.org/TR/offline-webapps/

[11] http://www.w3.org/TR/webstorage/#the-localstorage-attribute

[12] http://www.w3.org/TR/webstorage/#disk-space

[13] http://www.w3.org/TR/websockets/

[14] http://backbonejs.org/

[15] P. Lubbers and F. Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web," SOA World Magazine, Mar. 2010

[16] http://en.wikipedia.org/wiki/Serializability

[17] http://en.wikipedia.org/wiki/Vector_clock

[18] http://www.digital-web.com/articles/scope_in_javascript/

[19] http://coffeescript.org/

[20] https://brendaneich.com/2011/01/harmony-of-my-dreams/

[21] http://faye.jcoglan.com/

[22] http://svn.cometd.com/trunk/bayeux/bayeux.html

[23] http://nodejs.org/

[24] http://www.ruby-lang.org

[25] http://rubyonrails.org/
[26] http://backbonejs.org/
[27] http://en.wikipedia.org/wiki/Model-view-controller
[28] https://github.com/jeromegn/Backbone.localStorage
[29] http://en.wikipedia.org/wiki/Test-driven_development
[30] https://www.relishapp.com/rspec/rspec-rails/docs
[31] https://github.com/guard/guard-rspec#readme
[32] https://github.com/bradphelan/jasminerice#readme
[33] http://sinonjs.org/
[34] https://github.com/netzpirat/guard-jasmine#readme
[35] https://github.com/jonleighton/poltergeist#readme
[36] http://travis-ci.org/
[37] http://en.wikipedia.org/wiki/Lamport_timestamps
[38] http://www.broofa.com/Tools/JSLitmus/
[39] http://benchmarkjs.com/
[40] http://pivotal.github.com/jasmine/
[41] https://github.com/technicalpickles/jeweler
[42] http://www.w3.org/TR/workers/
[43] http://www.w3.org/TR/webdatabase/
[44] http://code.google.com/p/google-diff-match-patch/
[45] http://spinejs.com/mobile/docs/offline
[46] Backbone.js on Rails, thoughtbot, 2012
[47] Myers, E.W. 1986. An O(ND) difference algorithm and its variations. Algorithmica 1, 251–266.
[48] http://json.org/
[49] http://www.w3.org/Protocols/rfc2616/rfc2616.html
[50] http://github.com/krikis/nomad

Appendix A: Implementation

Beneath the languages and technologies used while implementing the mobile data replication system for offline web applications are described. After that we go into depth on the actual implementation of data versioning and our *merged browser log*, the *preventive reconciliation* setup and the *eager conflict resolution* variants. For more information on the overall architecture of the data replication system we refer to the concepts section of this document. For the entire source code of this project we refer to our GitHub repository [50].

1. Languages and technologies

Developing offline web applications involves at least two programming languages, one for the server-side and one for the client-side implementation. Depending on the language used, a wide variety of frameworks and libraries can be chosen from as a solid starting point for our implementation. The decisions we make on these topics are discussed in detail below.

CoffeeScript

When building a data replication system on top of HTML5 technology, obviously a lot of JavaScript source code is involved. JavaScript among other issues has a quirky handling of the 'this' keyword [18], lacks convenient syntax for named classes and inheritance and has semicolons and parenthesis littered all over the place. Handling and debugging hundreds of lines of JavaScript code can therefore become pretty cumbersome. Fortunately there is an alternative for which the name CoffeeScript was coined [19]. This language fixes most of JavaScript's issues and is considered to be more convenient to use, even by the creator of JavaScript himself [20]. In addition CoffeeScript is compiled into plain JavaScript, allowing for seamless integration with existing client and server logic. This motivates our decision to use CoffeeScript for the client-side implementation of our data replication system.

Faye messaging system

As discussed in the introduction of this thesis the best support for bi-directional client-server communication is provided by web-sockets, because they employ a full-duplex channel over a single socket with minimal network overhead. Though web-sockets provide a convenient API for implementing communication protocols from the ground up, we prefer to use the Faye messaging system as it readily provides a complete publish-subscribe solution built top of web-sockets [21]. Faye implements the Bayeux protocol allowing clients to subscribe to a set of channels and publish messages to them asynchronously [22]. In addition it neatly integrates with Node.js [23] and Ruby [24] on the server-side. By choosing an existing solution for the client-server

communication we can focus more on implementing the synchronization and reconciliation logic required for mobile data replication.

Ruby on Rails
When it comes to implementing the server logic that makes use of the Faye messaging system described above, we have a number of options including Node.js and a variety of Ruby web frameworks such as Rails, Sinatra and many others. We choose to base our server implementation on Ruby on Rails [25], because of our previous experience in building Rails apps and the fact that it is the most mature and most widely used web framework built in Ruby. With this decision we tie ourselves to the Ruby language for the server-side implementation.

Backbone.js
Putting all these open source frameworks in place certainly makes implementing mobile data replication simpler already. It would be a shame however not to consider the host of JavaScript frameworks that have recently come into existence, providing a solid architecture for the client-side of modern web applications. Examples of such frameworks are Backbone.js, Spine.js, Ember.js and many others. We choose to use Backbone.js [26], again because it is the most mature and most widely used JavaScript framework. It has a nice MVC-like architecture [27] providing models, collections, routers and views as building blocks, of which the models typically represent the application's data objects. It also uses convenient lifecycle events that are triggered on model *changes* and model *saves*, allowing us to easily hook in our versioning and persistence logic. And finally there exists a localStorage adaptor [28] for Backbone to persist the models' data to the browser localStorage instead of sending it back and forth between client and server. By basing our client-side logic on the Backbone framework and accompanying localStorage adaptor we now only need to implement data versioning, synchronization and conflict resolution.

Jasmine
Because implementing mobile data replication induces considerable complexity to the source code, it can be very hard to track bugs down to their cause and to refactor the code without introducing new bugs. Therefore we decide to adopt test driven development [29] (TDD) at the early stage in the implementation phase. This software development process stimulates us to write a test for every meaningful part of the implementation, thus improving its quality, giving us confidence in its correctness and reducing the number of bugs introduced during refactoring phases. Fortunately the Rails community knows excellent support for test driven Ruby development through a collection of Ruby gems such as RSpec [30], Fabrication and Timecop. Running tests can be fully automated with libraries as Guard-RSpec [31], which smartly decide what tests to run as soon as a portion of the source code is

updated on the hard drive. When it comes to testing JavaScript, the combination of JasmineRice [32], Sinon.js [33] and Rosie.js gives similar TDD support. With Guard-Jasmine [34] and Poltergeist [35] tests are again automatically run in the background on a fast headless browser. Longer running integration and acceptance tests can be delegated to continuous integration servers such as Travis [36], which run all tests as soon as they are pushed to a remote repository. The combined power of these testing libraries makes TDD a painless process with minimal overhead, allowing the developer to focus on building features while the quality of the source code remains guarded.

2. Implementing data versioning

The first functionality we implement in order to support mobile data replication is the data versioning process. Data versioning is the strategy employed to provide each copy of the data with a unique version identifier, which can later on be used for conflict detection. As we discussed previously we chose to use vector clocks for this purpose.

Vector clocks

The CoffeeScript code implementing vector clocks is listed below[1]. It provides functions for comparing vector clocks with each other in order to decide whether they equal (line 11), conflict (line 26) or supersede one another (line 17).

```
1.  class @VectorClock
...
10.   # Does the clock equal the other clock?
11.   equals: (otherVector) ->
12.     @_defineClocksOf(otherVector)
13.     _.all _.properties(@), (clock) =>
14.       @[clock] == (otherVector[clock] || 0)
15.
16.   # Does the clock supersede the other clock?
17.   supersedes: (otherVector) ->
18.     @_defineClocksOf(otherVector)
19.     some_greater = _.some _.properties(@), (clock) =>
20.       @[clock] > (otherVector[clock] || 0)
21.     all_greater_equal = _.all _.properties(@), (clock) =>
22.       @[clock] >= (otherVector[clock] || 0)
23.     some_greater and all_greater_equal
24.
25.   # Does the clock conflict with the other clock?
26.   conflictsWith: (otherVector) ->
27.     @_defineClocksOf(otherVector)
28.     some_greater = _.some _.properties(@), (clock) =>
29.       @[clock] > (otherVector[clock] || 0)
30.     some_less = _.some _.properties(@), (clock) =>
31.       @[clock] < (otherVector[clock] || 0)
32.     some_greater and some_less
```

[1] Note that @ in CoffeeScript stands for 'this' in JavaScript.

A fraction of the Jasmine unit tests that verify the correctness of the `#supersedes` method implemented above is listed below for illustration (see line 54 and further). In the remainder of this document tests that were the product of our TDD process will be omitted.

```
 1. describe 'VectorClock', ->
 2.   beforeEach ->
 3.     @vector = new VectorClock
 4.       some_clock: 1
 5.       other_clock: 2
...
54.   describe '#supersedes', ->
55.     context 'when at least one clock in @vector supersedes
56.              the corresponding clock in @otherVector', ->
57.       beforeEach ->
58.         @otherVector = new VectorClock
59.           some_clock: 1
60.           other_clock: 1
62.       it 'returns true', ->
63.         expect(@vector.supersedes(@otherVector)).toBeTruthy()
...
74.     context 'when no clock in @vector supersedes
75.              the corresponding clock in @otherVector', ->
76.       beforeEach ->
77.         @otherVector = new VectorClock
78.           some_clock: 2
79.           other_clock: 2
81.       it 'returns false', ->
82.         expect(@vector.supersedes(@otherVector)).toBeFalsy()
...
94.     context 'when the vectors conflict', ->
95.       beforeEach ->
96.         @otherVector = new VectorClock
97.           some_clock: 2
98.           other_clock: 1
100.      it 'returns false', ->
101.        expect(@vector.supersedes(@otherVector)).toBeFalsy()
```

Recording the browser log

When the end user makes a change to the data of a Backbone model, the local change has to be recorded and the data version needs to be updated. For this we bind an `#addVersion` callback to the model's *change* event. Because we want the callback to be available to all models instantiated in our web application, we override the Backbone.Model constructor[2]. For this we create an equally named constructor and call the original constructor from it (see line 7 in the code below). After that we bind our callback to the *change* event (line 9). Finally we clone the static properties (line 12) and prototype (line 14) and add the new constructor to it (line 19), so that our Backbone.Model continues to behave as the original one.

```
1.  # Override the Backbone.Model constructor and add the
2.  # on change #addVersion versioning callback to it
3.  Backbone.Model = ((Model) ->
4.    # Define the new constructor
5.    Backbone.Model = (attributes, options) ->
6.      @clientId ||= Nomad.clientId
7.      Model.apply @, arguments
8.      # Bind the versioning callback to the change event
9.      @on 'change', @addVersion, @
10.     return
11.   # Clone static properties
12.   _.extend(Backbone.Model, Model)
13.   # Clone prototype
14.   Backbone.Model:: = ((Prototype) ->
15.     Prototype:: = Model::
16.     new Prototype
17.   )(->)
18.   # Update constructor in prototype
19.   Backbone.Model::constructor = Backbone.Model
20.   Backbone.Model
21. ) Backbone.Model
```

[2] Extending backbone by creating a Backbone.PersistedModel which inherits from Backbone.Model is a viable option too, but would not enforce automated persistence to all models in a web application as we do here.

When the `#addVersion` callback is triggered, one of two different implementations for browser logging is triggered, depending on a global setting. The first implementation creates a *structured content diff* representing the data change at hand and pushes it into an array (see line 29 in the code below), thus building the incremental browser log discussed in the previous section. The second implementation instantiates a `Patcher` object, which updates an existing log entry for the model (see line 33 in the listing below). It provides the merged log described in the previous section. Both browser log implementations are discussed more elaborately below. In addition to updating the browser log, the `#_tickVersion` helper method is called (see line 35). This method updates the model's vector clock by incrementing its local clock value.

```
21.   # update the data version and record the local change
22.   addVersion: (model, options = {}) ->
23.     unless options.skipPatch?
24.       # initialize data version
25.       @initVersioning()
26.       @_versioning.patches ||= []
27.       if @versioning == 'structured_content_diff'
28.         # append structured content diff to incremental browser log
29.         @_versioning.patches.push @_createPatch(@localClock())
30.       else
31.         # update existing merged log entry
32.         patcher = new Patcher(@)
33.         patcher.updatePatches()
34.       # increment local clock
35.       @_tickVersion()
```

Structured content diff

When a *change* event is triggered for a model, Backbone conveniently retains the previous data allowing us to get a diff between two versions of its data. For this a JavaScript library named diff-match-patch [44] is used, which offers robust algorithms for plain text synchronization. We base the diff on a serialized JSON representation of the data. In order to prevent artificial conflicts based on diverging property ordering between two versions, all properties are recursively ordered before the data is serialized (see lines 43 and 44 in the listing below). The ordered data is then passed through the diff-match-patch library to compute a diff (line 48). Finally the diff is converted to a stringified patch and returned for persistence in combination with the version the update was based upon (line 53 and 54).

```
40.   # create a patch based on a structured content diff
41.   _createPatch: (base) ->
42.     # sort properties to prevent artificial conflicts
43.     sorted_previous = @_sortPropertiesIn @previousAttributes()
44.     sorted_attributes = @_sortPropertiesIn @attributes
45.     dmp = new diff_match_patch
46.     dmp.Diff_Timeout = 0
47.     # create a diff between two versions of the model
48.     diff = dmp.diff_main JSON.stringify(sorted_previous),
49.                          JSON.stringify(sorted_attributes)
50.     patch = dmp.patch_make JSON.stringify(sorted_previous),
51.                            diff
52.     # return a persistence friendly version of the diff and the update base
53.     patch_text: dmp.patch_toText(patch)
54.     base: base
```

Merged diff object

In the alternative log implementation instead of recording a list of diffs a single merged diff object is maintained for each model. This is done in the CoffeeScript excerpt below. All of the model's data properties[3] changed by a local update are individually compared to their previous value and the previously recorded changes, if any. When the original value was a nested object, an empty object is recorded in the diff object (see line 28 below). When the difference of the original and current value is of interest rather than the fact that the property changed, the original value is recorded[4] (line 32). Else only the property key itself is recorded (line 36).

```
19.   # update merged diff object to reflect current change
20.   _updatePatchFor: (patch, changed, previous = {}) ->
21.     _.each changed, (value, attribute) =>
22.       previousValue = previous[attribute]
23.       # when the property did not change before
24.       if not _.has(patch, attribute)
25.         # and the original value was a nested object
26.         if _.isObject(previousValue)
27.           # record an empty object
28.           patch[attribute] = {}
29.         # and a data diff is required
30.         else if _.isString(previousValue)
31.           # record the original value
32.           patch[attribute] = previousValue
33.         # and only the fact a change occurred is relevant
34.         else
35.           # just record the property key
36.           patch[attribute] = null
...
```

[3] Property and attribute are used interchangeably in this document.

[4] This can be useful for generating plain-text diffs and patches, number increments or boolean toggles. Currently only plain-text diffs are supported, but this could easily be extended.

When the local update concerns a nested data object, the nested set of changed properties is calculated (see line 42 below) and the recording algorithm goes into recursion (line 45).

```
...
37.       # when recursion is indicated
38.       if _.isObject(patch[attribute]) and
39.          _.isObject(previousValue) and
40.          _.isObject(value)
41.         # calculate the set of changed properties in the nested object
42.         changedAttributes = @_changedAttributes(changed[attribute],
43.                                                 previousValue)
44.         # and update the merged diff object for these changes
45.         @_updatePatchFor(patch[attribute],
46.                          changedAttributes,
47.                          previousValue)
```

Persisting data versioning

As we discussed earlier the persistence of the application data itself is fully handled by a Backbone localStorage adapter. The persistence of vector clocks indicating a model's current data version and the browser log recorded for this model still requires implementation though. For this we simply extend the localStorage adapter, so that it stores a versioning record every time a model is persisted. This is done by adding the `#saveVersioningFor` method to the adapter's `#update` method (see line 71 in the code below).

```
66.     # Model update (it already has a GUID)
67.     update: (model) ->
68.       # Save model data to localStorage
69.       @localStorage().setItem @storageKeyFor(model), JSON.stringify(model)
70.       # Save the model's versioning record
71.       @saveVersioningFor(model)
72.       # Register the model's id with the collection
73.       unless _.include(@records, model.id.toString())
74.         @records.push model.id.toString()
75.       # Save the collection
76.       @save()
77.       model
```

In the `#saveVersioningFor` method the model's versioning record is saved to the browser localStorage (see line 84 in the implementation below).

```
79.     # Persist the model's versioning record
80.     saveVersioningFor: (model) ->
81.       # Initialize versioning if it does not exist
82.       model.initVersioning()
83.       # Store versioning in localStorage
84.       @localStorage().setItem @versioningKeyFor(model),
85.                               JSON.stringify(model._versioning)
```

In addition we extend the adaptor so that it revives the versioning record every time a model is fetched from the browser's localStorage. This is done by adding the `#setVersioning` and `#setAllVersioning` methods to the finder methods of the localStorage adapter (see lines 89, 94 and 95).

```
87.     # Retrieve a model by id.
88.     find: (model) ->
89.       @setVersioning(model)
90.       JSON.parse @localStorage().getItem(@storageKeyFor model)
91.
92.     # Return the array of all models currently in storage.
93.     findAll: (collection) ->
94.       collection?.on 'reset', @setAllVersioning, @
95.       collection?.on 'add', @setVersioning, @
96.       _(@records).chain().map((id) ->
97.         JSON.parse @localStorage().getItem(@storageKeyFor id)
98.       , @).compact().value()
```

The actual reviving is done in the `#setVersioning` method, of which the CoffeeScript code is listed below (line 107 and beyond). The `#setAllVersioning` method simply calls this reviver method for all models (see line 103).

```
100.      # revive the versioning records for all models
101.      setAllVersioning: (collection, options) ->
102.        _.each(collection.models, (model) =>
103.          @setVersioning model
104.        )
105.
106.      # revive the model's versioning record
107.      setVersioning: (model) ->
108.        # parse the serialized versioning record
109.        versioning = JSON.parse @localStorage().
110.                        getItem(@versioningKeyFor model)
111.        # assign versioning record to model
112.        model._versioning = versioning if versioning?
```

3. Implementing preventive reconciliation

For bi-directional client-server communication we use the Faye messaging system which implements a publish-subscribe protocol over web-sockets (see the languages and technologies subsection).

Publish-subscribe

Faye clients can subscribe to channels and publish messages to them. The faye server simply receives published messages and broadcasts them to all clients that subscribed to the channel the message was cast at. Faye clients are typically used for peer-to-peer like communication between browsers. In our implementation however clients communicate to the server only for data synchronization. For this client-server communication the Faye system introduces a special server-side client that runs alongside the server process.

This client subscribes to a set of server channels (see line 11 in the Ruby listing below), to which synchronization messages are sent by mobile network nodes. It then uses an `#on_server_message` callback to process these messages (see line 14).

```
10.   # subscribe server-side client to all server synchronization channels
11.   def subscribe
12.     @client.subscribe('/server/*') do |message|
13.       # hook in the message processing callback
14.       on_server_message(message)
15.     end
16.   end
```

In the browser a global Faye client is instantiated, which is shared by all Backbone collections[5]. Each collection subscribes to a global synchronization channel (see line 36 in the CoffeeScript listing below) for receiving all messages the server broadcasts[6]. In addition collections subscribe to a private channel (see line 40), used for point-to-point client-server communication[7] during conflict resolution.

```
32.   # subscribe collection to synchronization channels
33.   subscribe: ->
34.     # subscribe to multicast channel
35.     global_channel = "/sync/#{@modelName}"
36.     @client.subscribe global_channel, @receive, @
37.     @subscriptions.push(global_channel)
38.     # subscribe to unicast channel
39.     private_channel = "/sync/#{@modelName}/#{@clientId}"
40.     @client.subscribe private_channel, @receive, @
41.     @subscriptions.push(private_channel)
```

[5] The reason for this is that browsers impose a per-host connection limit, which would be saturated by creating a single client for each Backbone collection.

[6] Also called multicast in this document.

[7] Also called unicast in this document.

Early conflict detection

At regular intervals the browser initializes a synchronization session. The first phase of this process is comprised of sending all local versioning data to the server, so that it can detect update conflicts in an early stage. This is implemented by the CoffeeScript below. Newly created and locally changed models are fetched separately using the `#_newModels` (line 17) and `#_dirtyModels` (line 22) helper methods respectively[8]. Their globally unique identifiers (GUIDs) and vector clocks are subsequently collected using the `#_versionDetails` helper method (line 10) and compiled into a synchronization message, which is then sent to the server by the global faye client (line 7).

```
2.  # compile a message for preventive reconciliation and publish it
3.  preSync: ->
4.    message =
5.      new_versions: @_versionDetails(@_newModels())
6.      versions: @_versionDetails(@_dirtyModels())
7.    @fayeClient.publish message
8.
9.  # collect a model's guid and vector clock
10. _versionDetails: (models) ->
11.   _(models).chain().map((model) ->
12.     id: model.id
13.     version: model.version()
14.   ).value()
15.
16. # fetch all models that have never been synced before
17. _newModels: () ->
18.   _(@models).filter (model) ->
19.     model.hasPatches() and not model.isSynced()
20.
21. # fetch all models that have local changes
22. _dirtyModels: () ->
23.   _(@models).filter (model) ->
24.     model.hasPatches() and model.isSynced()
```

[8] This separation is introduced in order to detect conflicts that are the result of a collision between locally generated model GUIDs. The detection and resolution of such conflicts is left undiscussed in the remainder of this thesis.

On the server-side, the GUIDs are used to fetch the master copy of a data object from the database. When such a copy can be found, its version is compared to the version of the local update[9].This is implemented in the Ruby source code shown below. First we check whether the update is obsolete (see line 34). As all synchronization communication is asynchronous and messages can get lost or reordered while transmitted over the network, it can happen that a local data version is synchronized twice. Obsolete updates are simply discarded. Next we check whether the update conflicts with the master version of the data object (line 37). A conflict is detected when the master vector clock supersedes the local vector clock, indicating that there have been updates to the master copy since the client's last synchronization session. In the event of a conflict, the corresponding GUID and the data representing the master copy update are unicast back to the client for reconciliation. The client-side implementation of conflict resolution is discussed later on (see next subsection).

```
28.   # compare the local data version with the version in the master data copy
29.   def check_version(model, version, client_id, results)
30.     object = model.find_by_remote_id(version['id'])
31.     # if the data object already exists on the server
32.     if object
33.       # is the update obsolete? -> discard it
34.       if object.remote_version.obsoletes? version['version'], client_id
35.         false
36.       # does the update conflict? -> report it
37.       elsif object.remote_version.supersedes? version['version']
38.         add_update_for(object, results)
39.         false
40.       # else -> persist the update!
41.       else
42.         [true, object]
43.       end
...
48.   end
```

[9] Note that comparing the master version with a local update's version is equivalent to comparing it to the version the local update is based on. The reason for this is that conflicting entries in a vector clock are never overwritten by creating the version of a local update, as only the local clock is incremented in such case (see previous subsection).

Synchronizing the data

Now that all conflicts have been detected (and hopefully resolved) it is time to actually synchronize the local changes to the server. For each model containing local changes, the GUID, data, vector clock and lifecycle timestamps are collected (see line 89 and beyond below) using the `#_dataForSync` helper method and sent to the server. In addition the model's current data version is marked as being synchronized, so that its entry in the browser log, which is needed for conflict resolution, is not modified until the synchronization process successfully completes (line 86).

```
81.   # collect all data that has to be synced to the server
82.   _dataForSync: (models, options = {}) ->
83.     _(models).chain().map((model) ->
84.       json = model.toJSON()
85.       # mark data version as being synced
86.       model.updateSyncingVersions()
87.       delete json.id
88.       # collect data for sync
89.       details =
90.         id: model.id
91.         attributes: json
92.         version: model.version()
93.         created_at: model.createdAt()
94.         updated_at: model.updatedAt()
95.       # mark object as synced
96.       if options.markSynced
97.         model.markAsSynced()
98.       model.save()
99.       details
100.    ).value()
```

When a local update causes no conflict on the server-side, it is saved into the master copy of the data. This is done using the `#set_attributes` method, for which the Ruby code is listed below. First the model's GUID is stored, if it was not set already (line 80). Next the actual data is saved (line 83), together with the vector clock indicating the current data version (line 85).

```
76.   # persist the local update on the master data copy
77.   def set_attributes(object, attributes, last_update = nil)
78.     # persist the GUID
79.     unless object.remote_id.present?
80.       object.update_attribute(:remote_id, attributes['id'])
81.     end
82.     # persist the actual data
83.     object.update_attributes(attributes['attributes'])
84.     # persist the data version and last_update timestamp
85.     object.update_attribute(:remote_version, attributes['version'])
86.     object.update_attribute(:last_update, last_update) if last_update
87.     # persist the lifecycle timestamps
88.     object.update_attribute(:created_at, attributes['created_at'])
89.     object.update_attribute(:updated_at, attributes['updated_at'])
90.   end
```

Lazy replication

Whenever a synchronization message is received from the client, regardless of its message type, the server collects all updates that were saved to the master copy since the client's last synchronization message and includes them into its reply to the client. This is implemented in the `#add_missed_updates` method, for which the Ruby code is listed below. All objects with a `last_update` Lamport timestamp [37] bigger than the timestamp of the last synchronization session are queried (see line 9) and filed for unicast to the client (line 16). Thanks to this piggybacking on synchronization messages local updates get lazily synchronized to all nodes in the network.

```
3.   # collect all updates since the last synchronization phase
4.   def add_missed_updates(model, message)
5.     lamport_clock = message['last_synced']
6.     results = init_results(message)
7.     # query all updates since the timestamp if present
8.     objects = if lamport_clock
9.       model.where(['last_update > ?', lamport_clock])
10.    # else query all models
11.    else
12.      model.all
13.    end
14.    # file all updates for unicast
15.    objects.each do |object|
16.      add_update_for(object, results['unicast'])
17.    end
18.    results
19.  end
```

Pushing updates

Thanks to the use of web-sockets, it is possible to speed up the replication of local updates by pushing them to all clients that are currently online. To this end the global channel all collections subscribe to (see above) is used. Whenever a local change is successfully saved to the server, it is simply multicast to all clients in the network who subscribe to the global channel. The implementation of this multicast is listed below (see line 43).

```
37.   # publish the results of a synchronization session
38.   def publish_results(message, results)
39.     # publish successfully saved changes to all nodes in the network
40.     multicast_channel = "/sync/#{message['model_name']}"
41.     if results['multicast'].andand['create'].present? or
42.        results['multicast'].andand['update'].present?
43.       @client.publish(multicast_channel, results['multicast'])
44.     end
45.     # publish detected conflicts or missed updates to the current client only
46.     if message['client_id'].present?
47.       unicast_channel = "#{multicast_channel}/#{message['client_id']}"
48.       if message['new_versions'].present? or
49.          message['versions'].present? or
50.          results['unicast']['resolve'].present? or
51.          results['unicast']['update'].present? or
52.          results['unicast']['_dbReset']
53.         @client.publish(unicast_channel, results['unicast'])
54.       end
55.     end
56.   end
```

When the client receives such a pushed update, it first compares the update's vector clock to the local vector clock indicating the data version of the associated Backbone model. When the local data precedes the server update[10] (see line 141 in the CoffeeScript below), the model is updated to reflect the server update. This is implemented in the `#_update` helper method. Fist the data is set to the model without recording the change in the browser log (line 201). Then the model's vector clock is updated so that it represents the remote clock increments that accompanied the pushed update (line 203). Finally the model is persisted to the browser's localStorage (line 204). The local copy of the data is now up to date with the server again.

```
136.   # determine which update strategy to use
137.   _updateMethod: (remoteVersion) ->
138.     switch @_checkVersion(remoteVersion)
139.       when 'supersedes' then '_forwardTo'
140.       when 'conflictsWith' then '_rebase'
141.       when 'precedes' then '_update'
 ...
197.   # update the model to reflect the update pushed from the server
198.   _update: (attributes) ->
199.     [version, created_at, updated_at] =
200.       @_extractVersioning(attributes)
201.     @set attributes, skipPatch: true
202.     # update vector clock to contain remote clock updates
203.     @_updateVersionTo(version, updated_at)
204.     @save()
205.     null
```

It can happen that the end user made some local changes that are not synced to the server yet when a pushed update is received. In this event, the local conflicting updates are immediately reconciled on the client-side. The implementation of this use case is dealt with later on (see next subsection).

[10] i.e. there were no local updates on the data since the last synchronization session.

Cleaning the browser log

Because updates are broadcasted to all nodes in the network once they are successfully applied on the server, the client initializing the synchronization process receives them too. We conveniently use these multicast messages as synchronization acknowledgements. Whenever the client receives a pushed update and the local version of the data supersedes (or equals) the received server version (see line 139 below), we know this push confirms that a local update the client itself previously synced was successfully saved to the master copy of the data. Any browser logs representing that or any older update can therefore safely be removed, as they are no longer needed for conflict resolution. This is implemented in the `#_forwardTo` helper method. All log entries that represent an update based on a data version smaller than the server version are discarded (line 114-115).

```
110.   # clean up all recorded changes preceding the received data version
111.   _forwardTo: (attributes) ->
112.     vectorClock = attributes.remote_version
113.     patches = _(@_versioning.patches)
114.     while @hasPatches() and patches.first().base < vectorClock[@clientId]
115.       patches.shift()
116.     @_finishedSyncing(vectorClock)
117.     @save()
118.     null
 ...
136.   # determine which update strategy to use
137.   _updateMethod: (remoteVersion) ->
138.     switch @_checkVersion(remoteVersion)
139.       when 'supersedes' then '_forwardTo'
140.       when 'conflictsWith' then '_rebase'
141.       when 'precedes' then '_update'
```

4. Implementing eager conflict resolution

Whenever the server reports a conflicting update or pushes an update that conflicts with local changes, the client tries to resolve the conflict by rebasing the local update. Changes performed to outdated data are thus re-executed on the new version of the data. We implement two approaches to this reconciliation strategy, one using an incremental log of structured content diffs and one using a single merged diff object.

Structured content diff

The incremental log can be seen as a list of subsequent patches representing changes to the local copy of the data. Local updates that are based on an outdated version of the data are repaired by applying the corresponding patches to the new version of the data. This is implemented in the coffeescript below (lines 173 and 174).

```
167.   # apply recorded patches to new version of the data
168.   _applyPatchesTo: (dummy) ->
169.     # if the incremental log is used
170.     if @versioning == 'structured_content_diff'
171.       patches = _(@_versioning.patches)
172.       # apply all patches to the new data
173.       patches.all (patch) =>
174.         dummy._applyPatch(patch.patch_text)
 ...
```

A patch is applied on the ordered stringified JSON representation of the new data using the diff-match-patch library we discussed before (see line 191 in the code below).

```
181.   # apply a patch to the model
182.   _applyPatch: (patch_text) ->
183.     dmp = new diff_match_patch
184.     dmp.Match_Threshold = 0.3
185.     # deserialize the patch
186.     patch = dmp.patch_fromText(patch_text)
187.     # sort the model data properties to prevent artificial conflicts
188.     sorted_attributes = @_sortPropertiesIn @attributes
189.     json = JSON.stringify(sorted_attributes)
190.     # apply the patch
191.     [new_json, results] = dmp.patch_apply(patch, json)
192.     if false not in results
```

```
193.       patched_attributes = JSON.parse(new_json)
194.       # save the re-executed update
195.       @set patched_attributes
196.       true
197.     else
198.       false
```

Whenever a patch is successfully applied, the resulting data is set on the model (see line 195 above).

When all recorded local changes are re-executed on the new data without failure, the newly generated local update is saved to the model (see line 159), and the local vector clock is updated so that it includes the new data version received from the server (see line 161).

```
148.   # resolve a conflicting update by rebasing it on a new data version
149.   _rebase: (attributes) ->
150.     # remove all obsolete entries from the browser log
151.     @_forwardTo(attributes)
152.     [version, created_at, updated_at] =
153.       @_extractVersioning(attributes)
154.     dummy = new @constructor
155.     dummy.set attributes
156.     # attempt to apply all recorded patches to the new data
157.     if @_applyPatchesTo dummy
158.       # persist the result of re-executing the local update
159.       @set dummy, skipPatch: true
160.       # update the local data version to include the new data version
161.       @_updateVersionTo(version, updated_at)
162.       @save()
163.       return @
 ...
```

Merged diff object

In our alternative approach to conflict resolution each data property is individually patched when rebasing a local update on a new version of the data (see lines 80 and 83 in the CoffeeScript below).

```
77.   # apply the recorded merged diff to the new data version
78.   _applyPatch: (patch, attributesToPatch, currentAttributes) ->
79.     # for each data property
80.     result = _.map patch, (originalValue, attribute) =>
81.       currentValue  = currentAttributes[attribute]
82.       # patch it with the diff between the original and current value
83.       @_patchAttribute(attribute, attributesToPatch,
84.                        originalValue, currentValue)
85.     false not in result
```

This is done using the `#_patchAttribute` helper method, for which the source code is listed below. When the original value of a property is stored and the types of new, updated and original value match, the property can be patched using a diff between its original and current value (line 95). This is discussed in more detail later on for the case of a text property.

```
87.   # patch an attribute given the original and current value
88.   _patchAttribute: (attribute, attributesToPatch,
89.                     originalValue, currentValue) ->
90.     if _.isString(currentValue)
91.       # when a diff can be generated
92.       if _.isString(originalValue) and
93.          _.isString(attributesToPatch[attribute])
94.         # patch attribute with diff between original and current
95.         @_patchString(attribute, attributesToPatch,
96.                       originalValue, currentValue)
97.       else
98.         attributesToPatch[attribute] = currentValue
99.         true
...
```

When the new, updated and original value are objects, the method goes into recursion in order to patch the nested properties (see line 105 below). In all other cases the new value is overwritten with the updated value (see for instance line 112).

```
...
100.     else if _.isObject(currentValue)
101.       objectToPatch = attributesToPatch[attribute]
102.       # when recursion is indicated
103.       if _.isObject(originalValue) and _.isObject(objectToPatch)
104.         # patch the attributes of the nested data object
105.         @_applyPatch(originalValue, objectToPatch, currentValue)
106.       else
107.         attributesToPatch[attribute] = currentValue
108.         true
109.     # when only the change itself is relevant
110.     else
111.       # apply the current value
112.       attributesToPatch[attribute] = currentValue
113.       true
```

Patching a text property using a diff is implemented with the diff-match-patch library we used before. First a diff of the original and current string value is generated (see line 125 below). Then a patch is created which is applied to the property at hand (line 130).

```
120.   # patch a new string value using the diff
121.   # of the original and current value
122.   _patchString: (attribute, attributesToPatch,
123.                  originalValue, currentValue) ->
124.     # calculate the diff and patch
125.     diff = @dmp.diff_main originalValue,
126.                           currentValue
127.     patch = @dmp.patch_make originalValue,
128.                             diff
129.     # apply the patch
130.     [patched_value, results] = @dmp.patch_apply(
131.       patch,
132.       attributesToPatch[attribute]
133.     )
```

```
134.      if false not in results
135.        # set the new value on success
136.        attributesToPatch[attribute] = patched_value
137.        true
138.      else
139.        # keep the current value when patching fails
140.        attributesToPatch[attribute] = currentValue
141.        false
```

When the patch applies successfully, the final result is retained (see line 136 above). Should the patch fail, because the new data version differs too much to match the patch, the current value is retained (see line 140 above) and false is returned. If all locally changed attributes can successfully be patched on the new data version, the local update thus generated is saved to the model (see line 159 below), and the local vector clock is updated to include the new data version it received from the server (see line 161 below).

```
148.    # resolve a conflicting update by rebasing it on a new data version
149.    _rebase: (attributes) ->
150.      # remove all obsolete entries from the browser log
151.      @_forwardTo(attributes)
152.      [version, created_at, updated_at] =
153.        @_extractVersioning(attributes)
154.      dummy = new @constructor
155.      dummy.set attributes
156.      # attempt to apply all recorded patches to the new data
157.      if @_applyPatchesTo dummy
158.        # persist the result of re-executing the local update
159.        @set dummy, skipPatch: true
160.        # update the local data version to include the new data version
161.        @_updateVersionTo(version, updated_at)
162.        @save()
163.        return @
 ...
```

Appendix B: Raw results

Below the raw benchmark results for network load, memory usage and client performance are listed. Each data point is a mean calculated from 10 subsequent measurements on random instances of a certain event (only the network load benchmarks use less measurements per data point). The benchmarks are run until the variation in the mean of all calculated data-points is less than 5% for the last 5 data points of all benchmarks in a suite. As the outcome in some benchmarks is more variable than others, the total number of collected data-points varies between benchmark suites. For a more elaborate explanation on the benchmarking setup used and the meaning of the collected data we refer to the methods and results sections in this document.

1. Network load

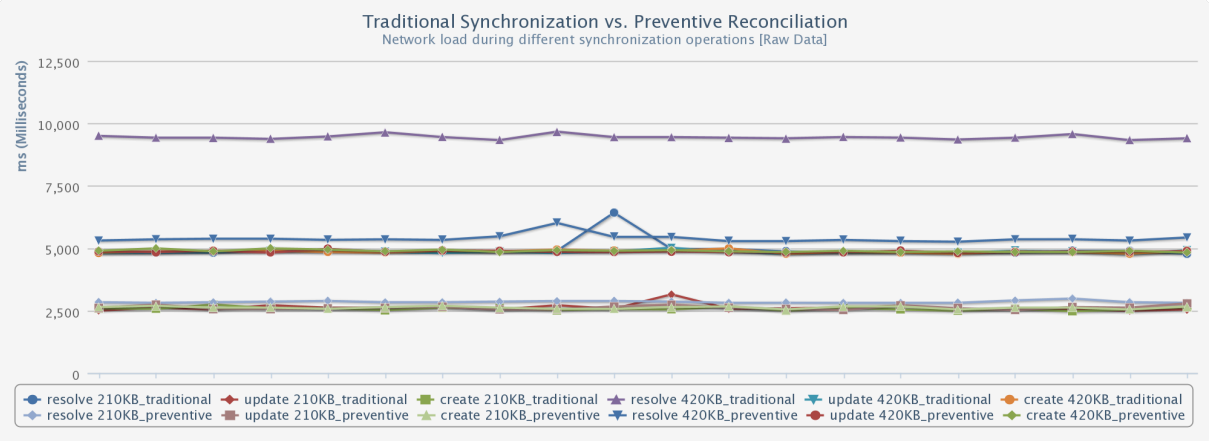
Network load data collected on February 1, 2013 at 18:01. The recorded events are synchronizing a created, updated and conflicting data object. The data object has a payload of either 210KB or 420KB. The synchronization methods used are preventive reconciliation and traditional synchronization. The results are the average amount of milliseconds it takes to synchronize a data object (average over 4 measurements in the 210KB case and 2 measurements in the 420KB case). The events are measured 80 times fore each distinct event-data-method combination (40 in the 420KB setup), which makes 720 runs in total.

Event	*Data*	*Method used*	*Results (ms)*
Create	210KB	Preventive reconciliation	2645, 2688, 2623, 2629, 2582, 2638, 2702, 2611, 2579, 2579, 2636, 2665, 2540, 2656, 2676, 2545, 2623, 2624, 2555, 2656
		Traditional Synchronization	2590, 2581, 2733, 2597, 2592, 2517, 2614, 2599, 2519, 2576, 2557, 2669, 2501, 2610, 2549, 2499, 2569, 2475, 2554, 2639
	420KB	Preventive reconciliation	4908, 5000, 4872, 5004, 4944, 4886, 4937, 4846, 4918, 4890, 4918, 4883, 4851, 4895, 4839, 4869, 4843, 4846, 4891, 4834
		Traditional Synchronization	4814, 4880, 4906, 4899, 4855, 4835, 4902, 4880, 4946, 4907, 4897, 4991, 4784, 4840, 4821, 4776, 4865, 4852, 4784, 4900
Update	210KB	Preventive reconciliation	2608, 2738, 2562, 2561, 2585, 2610, 2653, 2550, 2557, 2662, 2730, 2621, 2573, 2527, 2705, 2603, 2526, 2645, 2618, 2785
		Traditional Synchronization	2509, 2622, 2555, 2723, 2617, 2615, 2653, 2535, 2715, 2564, 3140, 2572, 2598, 2620, 2690, 2557, 2591, 2559, 2503, 2542
	420KB	Preventive reconciliation	4873, 4838, 4892, 4837, 4980, 4859, 4915, 4899, 4857, 4847, 4861, 4844, 4828, 4845, 4900, 4817, 4844, 4877, 4863, 4886
		Traditional Synchronization	4796, 4823, 4858, 4892, 4864, 4858, 4809, 4868, 4807, 4880, 5010, 4846, 4787, 4811, 4814, 4817, 4905, 4875, 4758, 4922

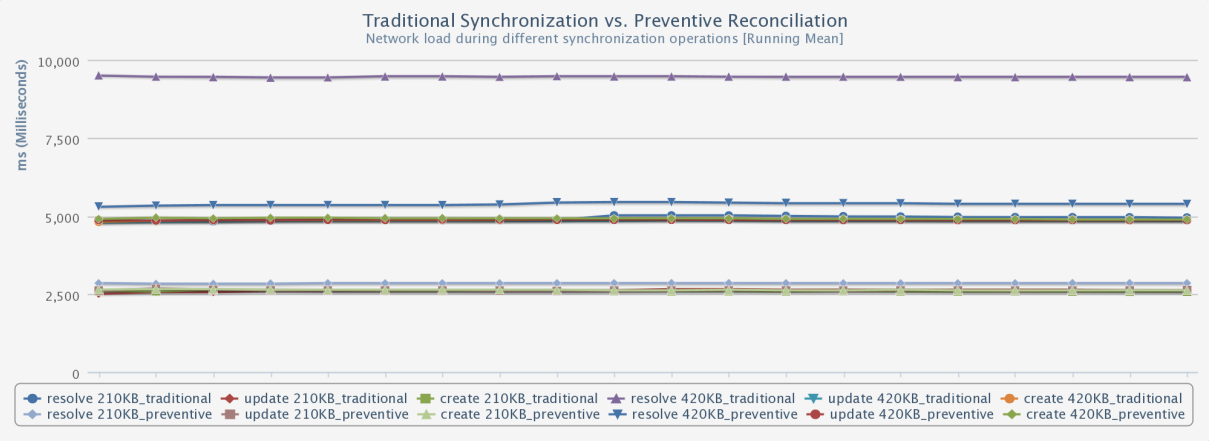
Table continues ...

Event	Data	Method used	Results (ms)
Resolve	210KB	Preventive reconciliation	2843, 2806, 2831, 2864, 2895, 2827, 2833, 2852, 2874, 2892, 2865, 2818, 2824, 2812, 2809, 2822, 2901, 2987, 2840, 2805
		Traditional Synchronization	4845, 4831, 4822, 4894, 4908, 4880, 4874, 4877, 4903, 6426, 4971, 4977, 4880, 4822, 4860, 4850, 4816, 4901, 4896, 4773
	420KB	Preventive reconciliation	5304, 5362, 5374, 5375, 5342, 5347, 5338, 5476, 6022, 5461, 5445, 5292, 5284, 5340, 5294, 5261, 5354, 5368, 5299, 5433
		Traditional Synchronization	9511, 9437, 9426, 9390, 9485, 9643, 9464, 9345, 9665, 9452, 9455, 9427, 9396, 9464, 9438, 9360, 9436, 9576, 9336, 9412

In the chart below the individual data points are plotted as they are successively calculated by the network load benchmark suite. There is not much variability between succeeding measurement of the suite indicating that the network latency is quite stable.



In he chart below a running mean of the above data points is plotted. It smooths the lines of the previous chart as most of the variability between data points is removed by calculating the mean.



2. Memory footprint

Memory usage data collected on May 4, 2013 at 13:11. The recorded events are recording respectively 3, 6, 9, 12, 15 and 18 local updates on a random data object. Each data object has 10 to 30 attributes of which 40% contain human readable text, 20% contain strings, 10% contain numbers and 10% contain boolean values. The updates randomly change respectively 12%, 25% and 37% of the data attributes and their values. The recording methods used are *merged browser log* and incremental browser log. The results are the average size of the resulting browser log proportional to the original size of the data objects subject to the updates. The events are measured 200 times for each event-method combination, which makes 7200 benchmark runs in total.

Event	*Impact*	*Method used*	*Results (%)*
3 updates	12%	Merged log	24, 31, 30, 26, 29, 26, 23, 28, 29, 30, 28, 20, 28, 31, 30, 26, 21, 21, 20, 21
		Incremental log	37, 35, 36, 29, 35, 34, 32, 25, 28, 26, 31, 26, 25, 26, 27, 34, 33, 28, 29, 33
	25%	Merged log	46, 44, 45, 50, 55, 41, 43, 52, 47, 46, 43, 43, 46, 48, 50, 41, 54, 50, 51, 55
		Incremental log	74, 80, 92, 91, 75, 77, 94, 102, 91, 93, 103, 94, 88, 89, 88, 91, 86, 97, 84, 103
	37%	Merged log	55, 61, 60, 60, 63, 64, 63, 60, 69, 67, 52, 68, 61, 58, 71, 62, 61, 52, 63, 72
		Incremental log	141, 151, 152, 161, 180, 152, 122, 162, 147, 157, 149, 141, 143, 144, 143, 138, 133, 155, 135, 159
6 updates	12%	Merged log	47, 56, 45, 40, 43, 42, 47, 51, 45, 43, 32, 34, 41, 37, 48, 52, 45, 39, 43, 45
		Incremental log	64, 58, 68, 68, 61, 73, 65, 71, 61, 70, 65, 58, 65, 70, 68, 65, 59, 58, 50, 69
	25%	Merged log	72, 67, 75, 77, 74, 68, 73, 74, 79, 79, 77, 69, 82, 75, 68, 80, 64, 80, 71, 72
		Incremental log	189, 162, 166, 173, 203, 175, 176, 156, 182, 174, 187, 178, 197, 183, 164, 177, 168, 172, 161, 191
	37%	Merged log	79, 86, 93, 87, 84, 83, 91, 81, 87, 89, 88, 94, 93, 89, 89, 93, 85, 91, 85, 92
		Incremental log	336, 305, 310, 293, 294, 284, 308, 309, 309, 278, 316, 292, 327, 329, 277, 335, 297, 334, 295, 289

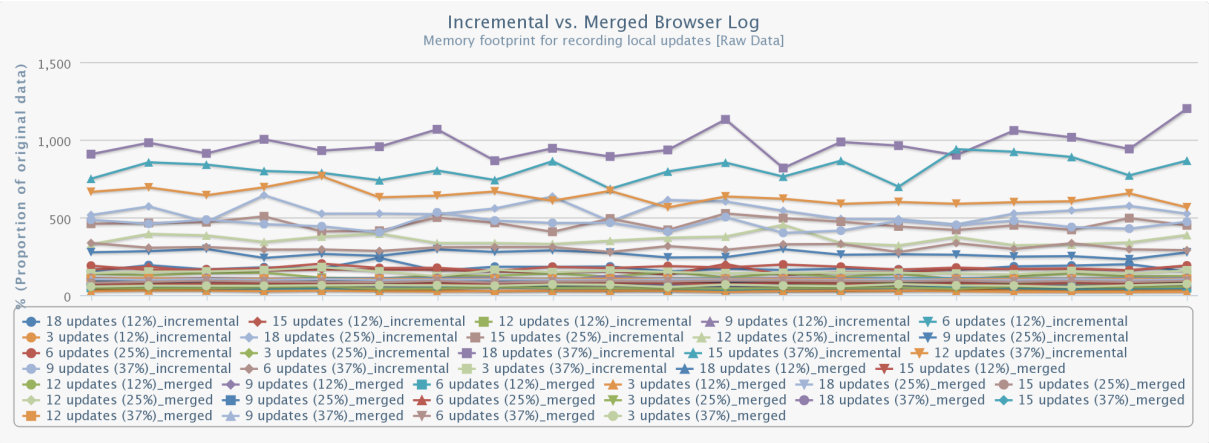
Table continues ...

Event	*Impact*	*Method used*	*Results (%)*
9 updates	12%	Merged log	57, 57, 59, 59, 58, 61, 68, 64, 68, 71, 64, 60, 63, 57, 62, 53, 59, 57, 55, 58
		Incremental log	93, 91, 109, 94, 104, 109, 108, 114, 90, 126, 95, 111, 90, 93, 95, 87, 108, 98, 99, 83
	25%	Merged log	85, 90, 90, 90, 87, 83, 94, 92, 90, 82, 89, 90, 92, 93, 90, 84, 87, 92, 89, 90
		Incremental log	276, 283, 297, 239, 265, 253, 294, 278, 288, 273, 242, 244, 296, 260, 263, 260, 248, 251, 231, 275
	37%	Merged log	103, 98, 102, 98, 102, 97, 99, 100, 99, 95, 101, 100, 98, 95, 102, 96, 97, 102, 96, 101
		Incremental log	485, 459, 486, 457, 443, 404, 532, 480, 464, 466, 408, 502, 401, 415, 478, 451, 479, 438, 428, 472
12 updates	12%	Merged log	71, 69, 68, 71, 76, 73, 72, 69, 69, 68, 73, 67, 61, 74, 77, 78, 79, 75, 75, 73
		Incremental log	138, 128, 142, 144, 112, 107, 126, 125, 141, 118, 150, 114, 141, 117, 138, 103, 121, 137, 120, 124
	25%	Merged log	94, 96, 94, 101, 99, 100, 96, 91, 95, 99, 97, 93, 103, 100, 93, 96, 98, 93, 92, 96
		Incremental log	328, 394, 384, 343, 377, 396, 335, 335, 329, 350, 369, 377, 453, 335, 320, 374, 321, 326, 339, 387
	37%	Merged log	106, 108, 104, 105, 102, 104, 103, 104, 105, 106, 104, 104, 104, 104, 99, 103, 104, 104, 104, 104
		Incremental log	664, 693, 643, 694, 765, 629, 640, 667, 607, 671, 564, 635, 620, 587, 599, 588, 598, 604, 655, 565
15 updates	12%	Merged log	75, 76, 71, 83, 81, 79, 87, 84, 78, 73, 77, 74, 73, 87, 80, 85, 79, 81, 75, 84
		Incremental log	165, 183, 151, 159, 164, 165, 163, 153, 145, 154, 132, 201, 129, 148, 143, 168, 169, 169, 159, 189
	25%	Merged log	100, 99, 100, 105, 104, 100, 102, 100, 103, 101, 105, 97, 100, 101, 101, 106, 100, 105, 103, 99
		Incremental log	459, 464, 469, 507, 410, 414, 499, 465, 408, 493, 420, 526, 496, 473, 442, 419, 449, 420, 496, 449
	37%	Merged log	108, 106, 106, 104, 105, 107, 106, 106, 108, 107, 107, 108, 102, 106, 107, 108, 107, 107, 107, 106
		Incremental log	749, 855, 840, 799, 787, 739, 802, 741, 862, 684, 796, 853, 763, 865, 698, 939, 923, 889, 771, 866

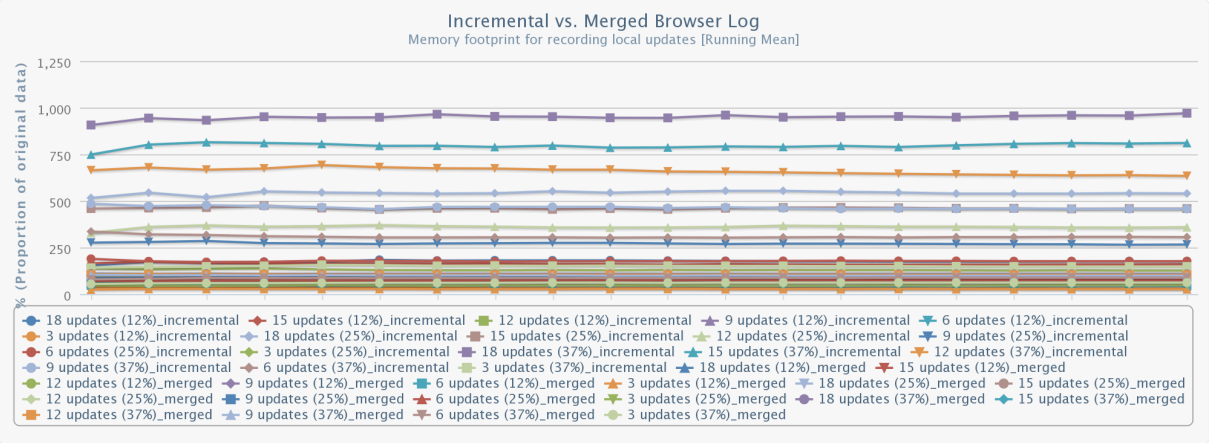
Table continues ...

Event	*Impact*	*Method used*	*Results (%)*
18 updates	12%	Merged log	85, 86, 86, 78, 89, 87, 91, 86, 87, 93, 89, 85, 83, 90, 92, 78, 86, 84, 84, 88
		Incremental log	154, 193, 166, 179, 173, 240, 157, 183, 182, 184, 153, 169, 162, 170, 166, 162, 185, 190, 199, 152
	25%	Merged log	105, 104, 104, 106, 100, 108, 101, 103, 103, 105, 102, 102, 101, 103, 107, 105, 105, 106, 105, 103
		Incremental log	516, 571, 472, 643, 524, 525, 521, 558, 634, 474, 612, 603, 543, 490, 489, 455, 525, 545, 574, 524
	37%	Merged log	111, 107, 111, 107, 107, 109, 109, 110, 106, 109, 109, 108, 109, 109, 109, 109, 108, 108, 110, 110
		Incremental log	907, 981, 912, 1003, 930, 955, 1067, 865, 945, 892, 934, 1131, 818, 986, 962, 901, 1059, 1016, 941, 1201

In the chart below the raw measurement data is plotted for each run of the memory footprint benchmark suite. There is some variability in the measurements which is caused by the randomness in the data objects used and the local updates applied.



The chart below displays the running means for the subsequent measurements of the used benchmark suite. The variability is reduced towards the end as the means converge to a stable value.



3. Client performance

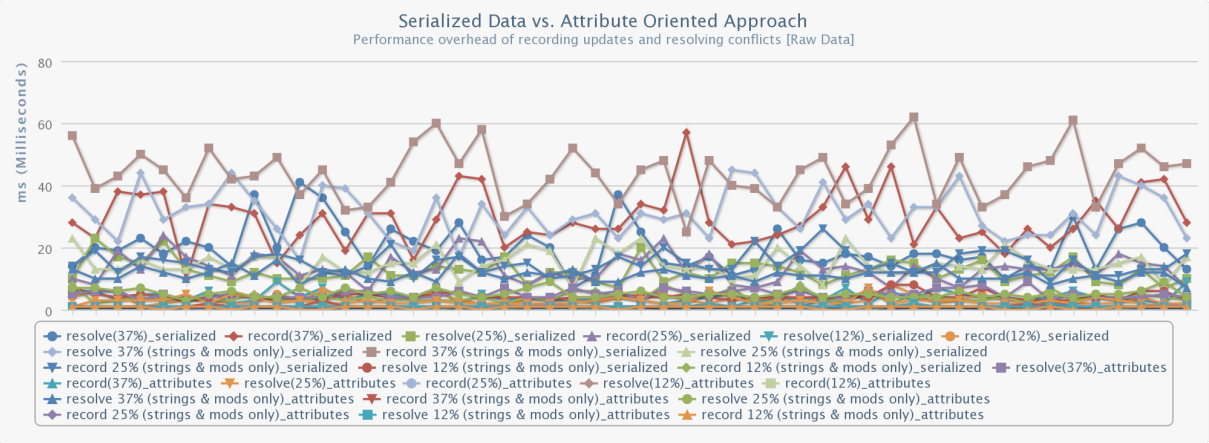
Performance overhead data collected on April 28, 2013 on 23:06. The recorded events are recording a local update and reconciling conflicting versions of a regular data object and recording a local update and reconciling conflicting versions of a data object containing only string and human readable text attributes and witnessing only attribute modifications, but no additions or deletions. The data objects used have 10 to 30 attributes. In the first two events 40% of the attributes contain human readable text, 20% contain strings, 10% contain numbers and 10% contain boolean values. In the third and last event 66% of the attributes contain human readable text and 33% contain strings. Updates of different impact are used: updates randomly change respectively 12%, 25% and 37% of the data attributes and their values. The recording and reconciling methods used are the attribute oriented and serialized data approach. The results are the average number of milliseconds it takes to record a local update or reconcile two conflicting data versions. The events are measured 500 times for each event-impact-method combination, which sums up to 12000 benchmark runs in total.

Event	*Impact*	*Method used*	*Results (ms)*
Record update	12%	Attribute oriented	1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1
		Serialized data	4, 6, 6, 1, 2, 3, 3, 2, 2, 5, 3, 6, 5, 2, 4, 4, 6, 3, 2, 2, 3, 4, 3, 2, 4, 5, 5, 3, 2, 4, 3, 4, 2, 4, 2, 3, 8, 5, 2, 3, 5, 3, 3, 2, 2, 2, 2, 4, 2, 2
	25%	Attribute oriented	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1
		Serialized data	10, 8, 19, 13, 24, 17, 14, 11, 18, 17, 11, 13, 12, 6, 17, 12, 13, 23, 22, 11, 8, 12, 10, 10, 18, 16, 23, 12, 18, 8, 7, 9, 19, 13, 14, 12, 7, 14, 10, 7, 13, 14, 13, 13, 15, 11, 18, 15, 14, 7
	37%	Attribute oriented	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 3, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1
		Serialized data	28, 23, 38, 37, 38, 10, 34, 33, 31, 15, 24, 31, 19, 31, 31, 16, 29, 43, 42, 20, 25, 24, 28, 26, 26, 34, 32, 57, 28, 21, 22, 24, 27, 33, 46, 29, 46, 21, 33, 23, 25, 18, 26, 20, 26, 35, 26, 41, 42, 28

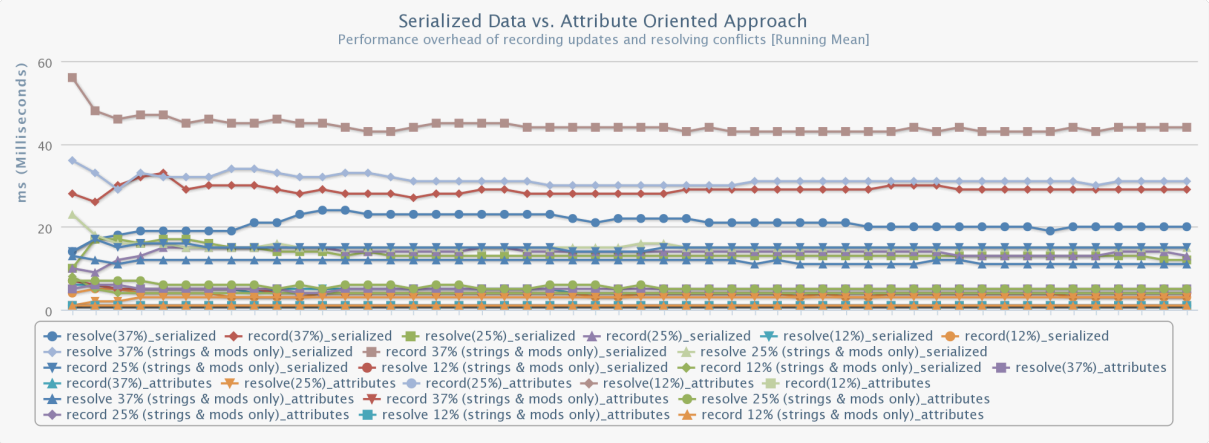
Table continues ...

Event	*Impact*	*Method used*	*Results (ms)*
Reconcile conflict	12%	Attribute oriented	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 2, 1
		Serialized data	6, 6, 3, 5, 5, 3, 6, 2, 3, 9, 4, 7, 3, 5, 5, 4, 7, 2, 5, 3, 4, 3, 3, 3, 4, 3, 2, 3, 4, 5, 3, 3, 4, 3, 7, 2, 6, 3, 7, 5, 4, 4, 3, 4, 6, 4, 3, 3, 2, 9
	25%	Attribute oriented	1, 3, 3, 3, 3, 5, 2, 5, 4, 2, 3, 2, 2, 3, 2, 2, 3, 2, 2, 2, 2, 2, 2, 3, 4, 3, 2, 2, 6, 4, 2, 2, 3, 2, 2, 7, 2, 3, 3, 3, 2, 2, 2, 3, 2, 4, 3, 2, 1, 1
		Serialized data	10, 23, 17, 15, 22, 13, 11, 9, 12, 10, 10, 10, 11, 17, 11, 11, 14, 13, 12, 17, 8, 12, 12, 9, 7, 20, 9, 12, 10, 15, 15, 14, 12, 8, 11, 13, 16, 15, 10, 13, 16, 9, 11, 12, 17, 9, 8, 13, 7, 10
	37%	Attribute oriented	5, 6, 6, 3, 5, 3, 4, 5, 4, 4, 4, 4, 6, 6, 4, 4, 4, 6, 4, 7, 4, 4, 7, 5, 6, 3, 7, 6, 5, 6, 7, 5, 6, 4, 4, 6, 5, 5, 7, 7, 8, 4, 9, 3, 5, 4, 4, 4, 5, 4
		Serialized data	14, 20, 19, 23, 18, 22, 20, 14, 37, 20, 41, 36, 25, 14, 26, 22, 19, 28, 16, 17, 24, 20, 7, 11, 37, 25, 15, 14, 17, 11, 13, 26, 16, 15, 18, 17, 14, 18, 18, 16, 17, 10, 14, 13, 30, 13, 26, 28, 20, 13
Record update (strings & mods only)	12%	Attribute oriented	1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1
		Serialized data	8, 2, 3, 4, 3, 4, 3, 3, 5, 3, 3, 4, 3, 4, 3, 3, 3, 3, 4, 2, 4, 3, 6, 6, 4, 3, 8, 5, 4, 7, 2, 3, 8, 3, 3, 6, 2, 5, 4, 3, 3, 4, 2, 7, 3, 4, 3, 5, 4, 3
	25%	Attribute oriented	1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
		Serialized data	14, 19, 12, 17, 16, 15, 13, 12, 17, 18, 16, 12, 11, 14, 21, 10, 16, 17, 12, 14, 15, 10, 11, 13, 17, 14, 20, 15, 13, 13, 22, 14, 19, 26, 19, 13, 15, 12, 12, 18, 19, 19, 16, 9, 16, 11, 11, 13, 13, 18
	37%	Attribute oriented	1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1
		Serialized data	56, 39, 43, 50, 45, 36, 52, 42, 43, 49, 37, 45, 32, 33, 41, 54, 60, 47, 58, 30, 34, 42, 52, 44, 34, 45, 48, 25, 48, 40, 39, 33, 45, 49, 34, 39, 53, 62, 34, 49, 33, 37, 46, 48, 61, 33, 47, 52, 46, 47
Reconcile conflict (strings & mods only)	12%	Attribute oriented	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1
		Serialized data	7, 5, 4, 5, 4, 3, 4, 5, 4, 3, 4, 3, 5, 5, 4, 4, 5, 6, 4, 4, 4, 3, 4, 4, 5, 4, 4, 5, 4, 3, 4, 4, 4, 4, 4, 4, 8, 8, 4, 4, 7, 4, 3, 3, 3, 5, 3, 6, 6, 3
	25%	Attribute oriented	7, 7, 6, 7, 5, 3, 5, 6, 4, 4, 7, 4, 7, 5, 6, 4, 7, 5, 4, 5, 7, 9, 4, 4, 5, 6, 4, 4, 5, 5, 4, 5, 7, 4, 5, 5, 4, 4, 4, 6, 4, 5, 4, 5, 4, 5, 5, 6, 9, 4
		Serialized data	23, 13, 13, 16, 13, 13, 17, 13, 17, 17, 8, 17, 12, 12, 15, 15, 21, 9, 14, 16, 21, 19, 7, 23, 18, 23, 14, 13, 14, 10, 11, 20, 14, 8, 23, 14, 12, 12, 15, 14, 13, 21, 16, 13, 13, 13, 15, 17, 8, 17
	37%	Attribute oriented	13, 10, 10, 14, 12, 10, 12, 15, 11, 16, 8, 12, 13, 10, 9, 12, 9, 18, 12, 10, 12, 11, 13, 9, 9, 12, 13, 11, 10, 11, 9, 12, 11, 11, 10, 12, 12, 12, 15, 10, 10, 10, 12, 8, 10, 11, 9, 12, 12, 7
		Serialized data	36, 29, 22, 44, 29, 33, 34, 44, 35, 26, 16, 40, 39, 31, 22, 19, 36, 17, 34, 24, 33, 24, 29, 31, 23, 31, 29, 31, 23, 45, 44, 33, 26, 41, 29, 34, 23, 33, 33, 43, 27, 22, 24, 24, 31, 24, 43, 40, 36, 23

In the chart below the raw data points resulting from running the client performance benchmark suite are shown. The variability between measurements is due to the randomness in the data objects used and the local updates applied.



In the chart below we can see the running mean of the recorded data points converge when the number of measurements increases. The difference between the results of the benchmarks handling distinct cases in our suite is more clear than in the chart above.



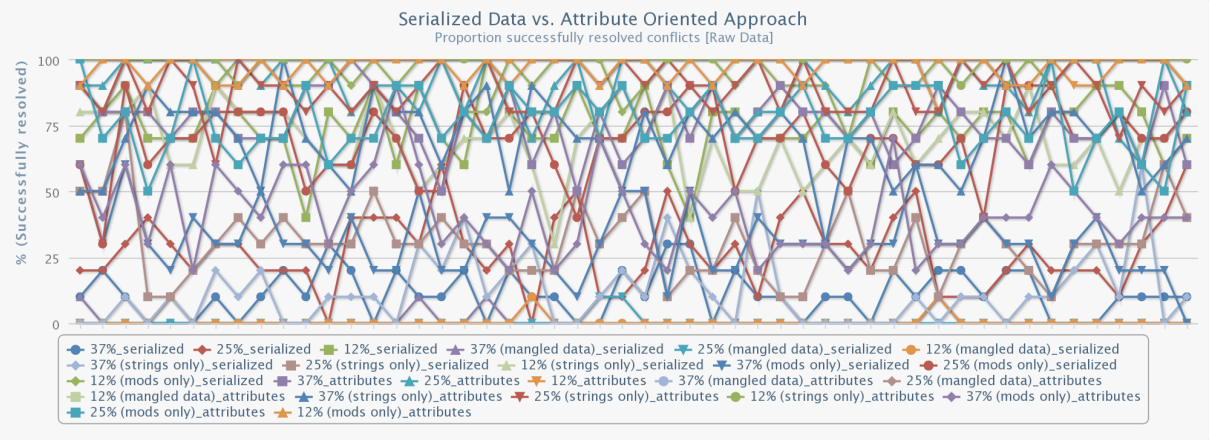
Data on conflict resolution rate collected on April 30, 2013 at 16:38. The recorded events are reconciling random data objects, reconciling random data objects and recording the proportion of reconciliation failures due to mangled data, reconciling data objects containing string and human readable text attributes only and reconciling objects that witnessed only modifications of attributes, but no additions or deletions. The data objects used have 10 to 30 attributes. In the first and last event 40% of the attributes contain human readable text, 20% contain strings, 10% contain numbers and 10% contain boolean values. In the middle event 66% of the attributes contain human readable text and 33% contain strings. The used updates randomly change respectively 12%, 25% and 37% of the data attributes and their values. The reconciliation methods are the attribute oriented and serialized data approaches. The results are the proportion of conflicts that can be successfully resolved using the reconciliation method at hand. Every event is measured 490 times for each event-impact-method combination, which makes 10290 runs in total.

Event	*Impact*	*Method used*	*Results (%)*
Random updates	12%	Attribute oriented	100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 90, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 80, 100, 100, 100, 100, 100, 90, 90, 100, 100, 100, 100
		Serialized data	70, 80, 90, 70, 70, 70, 90, 80, 70, 70, 40, 80, 70, 90, 60, 90, 70, 60, 100, 80, 70, 70, 90, 70, 70, 90, 60, 40, 80, 80, 70, 70, 70, 80, 70, 60, 80, 70, 80, 70, 80, 80, 60, 80, 80, 90, 90, 80, 60, 70
	25%	Attribute oriented	90, 90, 100, 100, 100, 100, 100, 100, 90, 100, 100, 100, 80, 90, 90, 80, 100, 80, 100, 90, 70, 90, 100, 70, 100, 100, 90, 80, 90, 70, 80, 80, 100, 90, 80, 90, 100, 100, 80, 100, 90, 90, 80, 100, 100, 100, 70, 60, 100, 80
		Serialized data	20, 20, 30, 40, 30, 20, 30, 30, 20, 20, 20, 0, 40, 40, 40, 30, 60, 30, 20, 30, 0, 40, 50, 10, 10, 20, 50, 30, 20, 30, 10, 40, 50, 30, 30, 20, 40, 50, 10, 10, 10, 20, 30, 20, 20, 20, 10, 30, 40, 60
	37%	Attribute oriented	90, 80, 80, 80, 70, 80, 80, 70, 70, 70, 90, 90, 100, 90, 80, 70, 50, 80, 70, 90, 60, 80, 50, 80, 60, 70, 90, 70, 90, 70, 80, 90, 80, 80, 70, 80, 60, 70, 90, 80, 70, 70, 60, 80, 70, 70, 80, 60, 90, 60
		Serialized data	10, 20, 10, 0, 0, 0, 10, 0, 10, 20, 10, 30, 20, 0, 20, 10, 10, 20, 0, 20, 10, 10, 0, 0, 20, 10, 30, 30, 0, 20, 10, 10, 0, 10, 10, 0, 20, 10, 20, 20, 10, 20, 20, 0, 0, 0, 10, 10, 10, 10
Random updates (mangled data)	12%	Serialized data	0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

Table continues ...

Event	Impact	Method used	Results (%)
Random updates (mangled data) *cont.*	25%	Serialized data	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
	37%	Serialized data	10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Random updates (strings only)	12%	Attribute oriented	100, 100, 100, 100, 100, 100, 100, 90, 100, 100, 100, 100, 100, 100, 90, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 90, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100
		Serialized data	80, 80, 80, 90, 60, 60, 90, 80, 80, 80, 70, 60, 60, 90, 70, 50, 60, 70, 70, 80, 60, 30, 70, 80, 90, 70, 90, 40, 70, 50, 50, 70, 50, 60, 70, 60, 80, 60, 70, 80, 80, 70, 90, 60, 60, 70, 50, 70, 70, 80
	25%	Attribute oriented	90, 80, 100, 80, 100, 90, 60, 100, 90, 90, 80, 90, 80, 90, 80, 90, 100, 100, 100, 80, 80, 80, 100, 90, 100, 90, 100, 90, 80, 90, 100, 80, 100, 80, 80, 80, 100, 80, 80, 100, 90, 100, 80, 90, 100, 90, 70, 90, 80, 90
		Serialized data	50, 50, 80, 10, 10, 20, 30, 40, 30, 40, 30, 30, 30, 50, 30, 30, 40, 30, 30, 20, 20, 20, 50, 30, 40, 50, 10, 20, 20, 40, 20, 10, 10, 30, 50, 20, 20, 40, 10, 30, 40, 30, 20, 10, 30, 30, 30, 30, 60, 40
	37%	Attribute oriented	50, 50, 70, 90, 80, 80, 80, 70, 50, 100, 70, 60, 50, 90, 80, 80, 60, 80, 90, 50, 90, 80, 70, 70, 50, 80, 60, 80, 70, 80, 70, 80, 70, 30, 70, 70, 50, 60, 60, 50, 70, 100, 70, 80, 80, 70, 80, 50, 60, 70
		Serialized data	0, 0, 10, 0, 10, 0, 20, 10, 20, 0, 0, 10, 10, 10, 0, 30, 20, 40, 10, 20, 30, 0, 0, 10, 20, 10, 40, 20, 10, 0, 50, 10, 0, 0, 0, 0, 0, 10, 0, 10, 10, 0, 10, 10, 20, 30, 10, 60, 0, 10
Random updates (mods only)	12%	Attribute oriented	90, 100, 100, 90, 100, 100, 90, 90, 100, 90, 100, 90, 100, 100, 100, 100, 100, 90, 100, 100, 100, 100, 100, 90, 100, 100, 90, 100, 90, 100, 100, 100, 90, 100, 100, 100, 100, 100, 100, 100, 100, 90, 90, 100, 100, 100, 100, 100, 100, 90
		Serialized data	90, 80, 100, 100, 100, 100, 90, 100, 100, 90, 90, 100, 90, 100, 100, 100, 100, 80, 80, 100, 90, 100, 100, 100, 80, 90, 100, 100, 100, 90, 100, 90, 90, 90, 70, 100, 90, 90, 100, 100, 100, 90, 100, 90, 100, 100, 100, 100, 100, 90
	25%	Attribute oriented	100, 70, 80, 50, 70, 100, 70, 60, 70, 70, 90, 60, 70, 70, 90, 90, 70, 80, 70, 90, 80, 80, 90, 80, 90, 70, 70, 80, 90, 70, 70, 80, 90, 70, 70, 80, 90, 90, 90, 60, 70, 80, 70, 100, 50, 70, 80, 60, 50, 90
		Serialized data	60, 30, 90, 60, 70, 70, 80, 80, 80, 80, 50, 60, 60, 80, 70, 50, 50, 90, 70, 70, 80, 60, 40, 70, 70, 80, 80, 90, 90, 50, 70, 70, 80, 60, 50, 70, 70, 60, 60, 70, 40, 90, 90, 90, 70, 70, 80, 70, 70, 80
	37%	Attribute oriented	60, 40, 60, 30, 60, 20, 60, 50, 40, 60, 60, 30, 50, 60, 90, 60, 30, 40, 30, 20, 50, 20, 30, 70, 50, 30, 20, 50, 40, 50, 20, 30, 30, 30, 20, 30, 70, 20, 30, 30, 40, 40, 40, 60, 50, 40, 30, 40, 40, 40
		Serialized data	60, 30, 60, 30, 20, 40, 30, 30, 50, 30, 30, 20, 40, 20, 20, 50, 20, 20, 40, 40, 30, 20, 10, 30, 50, 50, 10, 50, 20, 20, 40, 30, 30, 30, 20, 30, 30, 60, 30, 30, 40, 30, 30, 10, 30, 40, 20, 20, 20, 0

In the chart below the raw data points as calculated by our reconciliation benchmark suite are displayed. There is a lot of variability due to the randomness in the data objects used and the local updates applied, obscuring the differences in outcome between the distinct benchmarks in our suite.



The chart below shows the running mean for the calculated data points. In this chart one clearly see the differences between the results for the distinct benchmarks in our suite.