PEER TO PEER SYSTEM FOR COLLABORATIVE WORK

Using version control and fixed merge pattern to ensure eventual consistency

Jim Norman
Abstract

This paper addresses the possibility of using version control and fixed merge pattern to ensure eventual consistency for collaborative work tools. Where the version control is upheld by a vector clock and used to detect conflicts. Which in turn are solved by rolling back to the last common version before using fixed merge pattern to create a new common version. In the fixed merge pattern a score system is used to lose as little work as possible. This is then tested against a leader node solution for the collaborative work tools. Which uplifts a FIFO consistency to sequential consistency by forcing all updates to go through the leader. The test used is a speed test to see if it is a viable solution or not to use when dealing with weak network connection. Since it would require less messages to be sent over the network compared to a leader node solution.
Acknowledgements

As the author I would like to thank my family who has helped me during hard times. Though I must give more credit to my wife who has gone out of her way in supporting me. I would also like to thank my classmates that I have gotten to know during my studies, since many of them have helped at least at some point with understanding the school work.

Last I would like to thank the teachers who have helped me reach this point where I could make this paper happen. With extra thanks to Ewnetu Bayuh Lakew who has provided me with the help I needed to complete this paper.
## Contents

1 Introduction 1
   1.1 Introduction 1
   1.2 Scope and method 1

2 Building a collaborative work tool 3
   2.1 Weak network connection 3
   2.2 The issue with latency 3
   2.3 What peer to peer system to use 3
      2.3.1 Why peer to peer in the first place? 4
      2.3.2 Connecting the peer nodes 4
   2.4 Consistency 4
      2.4.1 Solutions for sequential or strict consistency 5
      2.4.2 Solutions for eventual consistency 6

3 The collaborative work tools 9
   3.1 The system architecture 9
      3.1.1 The peer logic module 9
      3.1.2 The consistency module 10
   3.2 Unique ID 13

4 Evaluating the collaborative work tools 15
   4.1 The system architecture for the control tool 15
   4.2 Test Environment 15
   4.3 Test design 15
   4.4 Results 16
      4.4.1 Adds 17
      4.4.2 Deletes 17
      4.4.3 Changes 18
      4.4.4 Mixed operations 18
   4.5 Conclusion 19
1 Introduction

1.1 Introduction

There is a huge importance in the ability to do collaborative works in today’s market and several different industries. Even so much that there is a need for collaborative work systems (CWS for short) in order to optimize the productivity when working with several people [5]. So it should come to no surprise as to why there would be a need for different sort of softwares to be used by these industries which is referred to as collaborative work tools. Take "GIT" as an example, which purpose is to assist in software development by allowing the users to share code and then merge it together when the time is right. This is however not the only industry that would be needing some sort of software to work on a distance and "GIT" still requires the users to have some experience handling merges. Furthermore "GIT" requires a centralized node where the users can get the result of said merges [16].

So what if we look at truck drivers that ship timber, where they can start picking up timber in the middle of nowhere before driving to the next location. Limiting their ability to handle any issues that may occur with any software they might need to use. They will not be able to use the same software as the developers since the requirements based on the work they do is different. Though they are both doing collaborative work as the developers produce a new product while the truckers move X goods from A to B, where in both cases they work in a group to get the job done faster.

There for exploring the option of using a decentralized system where eventual consistency is ensured through other means, rather then using a centralized system such as "GIT" where the users must handle their merge issue themselves. Which like "Git" might be able to be presented as an option to use for making a system with strict or sequential consistency [16, 1].

1.2 Scope and method

So the purpose here is to explore the possibility of using a system that is decentralized and follows eventual consistency, when dealing with collaborative work tools. In order to solve the issue the CAP theorem expresses [6, 8]. Now collaborative work is a very broad term since anything that involves more than one person working together, is a sort of collaborative work [5]. But also because defining a good tool or not has no clear answer as different people think different. Not to mention it might not be suitable for every purpose there is. You do not want a hammer when you are trying to sew clothes. Therefore the method for testing this out needs to be focused on something that can be measured but also be fair [23].

This is why speed testing on a generic collaborative work tool that uses this system is a better option than any sort of user testing. Since it will be more fair to check for speed than have users evaluate it. As users would evaluate the tool not the system [3, 23]. Something that
says little about the actual system on its own. Meanwhile speed is something that is always relevant as any sort of tool that is slow to use, will slow down the worker which in turn may result in less work done. To put it into clearer terms. You do not want to wait five seconds before hitting the nail with the hammer. You just want to hit the nail right away [23].

The generic collaborative work tool in this case would have to have a "collective space" that all workers can access. After all if you do not have something the workers can work together in, it is not a collaborative work. This is why at least two of the workers need to be able to also somehow modify this "collective space" since again it would not be a collaborative work [5]. This leads to the next part, in what way can a worker modify the space? Well it can be made smaller (delete), made larger (add) or have something within it changed (change). Since the goal is to speed test this system it should be a program that includes all three of these, preferably without affecting the results too much. This is why a simple list of elements that every user (worker in this case) can add, remove or modify elements in. Would work well as a generic collaborative work tool. Since it satisfies all the requirements [23, 5, 25].

Finally in order to properly be able to say if the system is slow or not it needs to be tested against something. In this case it is fitting to test it against a competitor which would be the same generic collaborative work tool but using a different system. Which should either be a peer to peer system or a client server system that uses at least Sequential Consistency (can be uplifted to strict consistency if the system uses locks). Since the "collective space" must end up looking the same in the end [8, 23, 1, 14].
2 Building a collaborative work tool

2.1 Weak network connection

This brings up the question about those users that go to places where the network connection is weak. If one were to take Sweden for example even signing up with the company that got the best coverage for mobile network there is still spots where the coverage does not reach (this is without accounting for other factors such as mountains and buildings blocking the signal) [24]. Which highlights the problem even more as 96.41% of the Swedish population had used the internet in the last three month in 2017 [19]. Sweden also ranked second in the highest average internet speed in 2018 at 46 mbps [7]. Hence if the country with the second fastest internet on average whom 96.41% of the population uses the internet with some frequency still got spots where they will have weak or no connection at all. It is fair to assume it is still a relevant issue to address when creating software for any sort of "collective space" on the internet, if the users are not 100% of the time within the high speed internet areas.

2.2 The issue with latency

Any distributed system is affected by the internet connection but for systems that might need its users to see changes in real time, it is an aspect that can play a role in collaborative work tools. Hence it will play a part in the cost of sending "messages" between nodes. So how big is this cost? well the average RTT (round trip delay time) to google across the globe is 100ms (outside of the USA) and for mobile connection the RTT latency can be anywhere between 100-1000ms [9]. For a connection to a satellite the latency is 500 ms or higher [18]. Even with fiber the time is limited by the speed of light so the latency from New York to Sydney can at best be 80ms though in reality it is 160ms [10].

This is why it would be bad if the system would need to send a series of messages for any reason. Since each time a node must wait for a message from another node it increases the delay of whatever you are trying to do. Example: Node A asks Node B about the weather would be better then Node A asking Node B to ask Node C about the weather. Since in the first example it would only get the delay from asking Node B, but in the second example it also gets the delay from Node C. This will be referred to as iterations.

2.3 What peer to peer system to use

For collaborative work tools there is a great need for consistency, speed, fault tolerance as well as availability. Which is why certain peer to peer architectures fit better then others, some even being a poor choice [11, 8].
2.3.1 Why peer to peer in the first place?

The answer to why peer to peer is partial tolerance. A server client solution will go down if the server goes down. Something that will in turn put any work on hold while the system is down, for all essential collaborative work tool. The only way to make sure this do not happen is to make it peer to peer, as the server client solution is dependent on the server. Therefore the only way for the system to not have a single point of failure is to make it into peer to peer, as adding more servers as back-ups would mean it is a peer to peer system with different valued nodes. Some are client nodes and some are server nodes [11, 8].

2.3.2 Connecting the peer nodes

No matter what type of collaborative work tool you are trying to develop, it needs a high fault tolerance and availability. Which means that it will need to be able to recover from almost anything and never be unavailable. As it can be assumed that when the system fails or is down no work is going to be performed. It also must follow some sort of consistency as in the end every worker will need to have the same final product in front of them. Finally it also needs to be fairly fast and have high scalability for its users [23, 8].

Because of this a "ring solution" would not really cut it, since this would put the nodes in a logical ring where they only know their neighbors. Since this would limit the fault tolerance quite a lot as if one node vanishes with formally quitting the ring it is destroyed. Not to mention it would take n (where n is the number of nodes) iterations to send a message through the system. The solution to that would be a chord system, as it would be able to scale fairly well in large systems. However this greatly reduces the speed for the system, as any message has to go through nlog(n) iterations before reaching all nodes [11, 17]. Something that is not good.

From the other direction we got connecting all the nodes to each other. Which in turn would allow them to send any message directly to any node, giving them just one iteration of messages. Also since they all know each other there will not be any issues reconnecting them if one were to suddenly drop [11, 8]. However this would not give a good scalability though speed wise it will be fairly okay as it can send any message with only 1 iteration before it reaches every node [11, 17].

2.4 Consistency

The consistency used for any collaborative work tool will affect the overall system design. If there is a need for sequential or even strict consistency in the system, the nodes will only be able to perform any updates to the "collective space" while they are online. Meanwhile if it is eventual consistency of some sort the nodes can perform updates on their local version which is then uploaded to the online version when they come online [13, 8].
2.4.1 Solutions for sequential or strict consistency

For a sequential consistency or strict consistency solution, it is fairly similar with one difference. The strict consistency must lock the read area and then agree on an update. While the sequential consistency only needs to agree on which order to update. This is because a sequential consistency only needs to update in the right order while the strict consistency must always have the same version [8, 1, 14].

To ensure this there are a few methods. One being a logical ring which passes a lock between the nodes [11], which is a bad idea since it requires in worst case n message iterations before a node can make an update. But then there is the election algorithm and the use of a leader node [11, 14]. Both of which have to know every other node in order to function. The leader node also will need a method to re-pick the leader which could be the election algorithm [11].

For both of these solutions there is only a need for a two message iteration, when talking about sequential consistency. This is because in both solutions, they need to agree on a common Lamport clock [14]. As otherwise they would stray away from the consistency. For the strict consistency the leader has three iterations and the election algorithm still has two. Since for strict consistency the version always needs to be the same [15].

The sequential consistency with a leader node starts with first message being an update request. Then the second message is the update. While for the election algorithm the first node is an update and then the second one is the vote from every node to decide which update to do. In order to make sure the lamport clock stays the same for every node [14]. For the strict consistency it changes for the leader to be, getting an update request, locking the system and then performing the update. While for the election algorithm it is sending the request, upon getting the request lock down until the election is over. Then send the vote and wait for the other votes before unlocking.

The election algorithm has a weakness though, if a node fails during the election it might cause a lot of delay [11]. But for the leader node algorithm, it only takes longer if the leader fails [11]. Also the system can elect the "best node" to be leader while the election algorithm suffers as a whole if any node is slow. Referring back to latency where mobile connections can have up to 1000ms delay [9]. The election algorithm also has to send $2n$ (n being the number of nodes) messages in total while the leader node only has to send $n + 1$ messages in total unless it is a strict consistency then it is $2n + 1$ in total.
2.4.2 Solutions for eventual consistency

Eventual consistency can only be used when dealing with systems that do not require things to be "correct" or "the exact same copy" at all time [6]. So why use it when dealing with collaborative work tools? Well not every sort of collaborative work is the same and the users may not always be available at the same time. Which either is due to them not being physically present at a device connected to the system, or that the device has issues connecting to the system being used. Some works may also require longer time before a change can be made such as when coding [16, 2, 5]. These factors make it desirable to use a system that allows the users to not constantly be there or have every letter written being sent to every other users at all time. Since it would cause unnecessary delay [15]. That is why eventual consistency may be the preferred instead of another consistency module. Granted of course that each user eventually will end up with the same copy.

However with eventual consistency conflicts are bound to happen and because of this there are two approaches to handle them. Either with rollback or with reconciliation [20, 4, 2]. In rollback it returns to a previous version and then handles any conflicts that have occurred [4]. While reconciliation simply tries to agree on a common version based on their current one [2, 4]. Now for a collaborative work tool, this means that reconciliation might lead to unnecessary loss of work as it does not know what has happened before. For example node A and B notice their version is different. Node A has two elements while node B has one. So node B adds node A's element 2, then they decide on a common element 1. However in reality node A has only worked on element 1. But node B on the other hand deleted element 1 causing element 2 to be element 1 and is working on that. So the optimal solution would have been to keep node A's element 1 and then just put node B's element 1 back to 2. But the system will not know that as it does not care for the past. As seen in Figure 1.

![Figure 1: Reconciliation handling conflicts with different numbered nodes](image-url)
This is where rollback comes in. It returns to a previous version and then redoes all the work the nodes have done. So for rollback the issue would look like this. It will see node A and B used to have two elements. Node B deleted element 1 and then worked on element 2. Node A has only work on element 1. It will then ignore node B’s delete of element 1 and put the work of node A there. Then put node B’s work on element 2 as seen in Figure 2. Now reconciliation is faster but it might lead to a bigger loss of work which could be a deal breaker for some tools. This issue only occurs if the “collective space” is dynamic since if the amount of elements can not change this problem will not occur.

![Diagram](image)

**Figure 2:** Rollback handling conflicts with different numbered nodes
3 The collaborative work tools

3.1 The system architecture

As the overall system architecture is a MVC (model-view-controller), as seen in Figure 3. The view and controller is fairly simple. The view consists of a text based UI and the controller just operates as the glue between the view and the model.

![System architecture for both collaborative tools](image)

**Figure 3**: System architecture for both collaborative tools

The biggest part of the system is the model. Which consists of three parts the communication module, the peer logic module and the consistency module. Where the communication module is a simple java RMI that can pass messages to a given node.

3.1.1 The peer logic module

For the peer logic module it handles the functions to create a network, join an existing network, save contact information, reconnect node to an existing node to the network and serve as a middle manager between the consistency module and the communication module. This is because it is completely decentralized and will not need much else, since it implies every node knows every node. Otherwise the system will not be useful for tools that are used where the internet connection might be weak.
3.1.2 The consistency module

In the consistency module all the consistency issues are handled. So it can receive and handle any update given by the peer logic module. It does so by detecting conflicts and if one occurs it performs a rollback to the last common version and then builds a new version after handling the conflicts.

3.1.2.1 Version control and fixed merge patterns

In order to detect conflicts a version control is added to the system. This can be done by adding a vector clock to it. This would also already ensure causal consistency so it only needs to handle the conflict where two nodes or more nodes are performing updates at the exact same time. A vector clock is the same as a Lamport clock but each node has their own clock [14, 22, 4]. When they perform a commit they update their own clock and send a timestamp based on all clocks. This way all nodes will see what updates have been done before this update, and when they order it according to this clock it results in causal consistency [22]. However it can also be used to create unique versions as the vector clock is only identical if they have performed the same updates. If they have not done that it will look different [4, 12, 25].

The issue though is not the use of a vector clock but rather how to handle the versions so they end up being the same after handling any conflicts that might have occurred. Which means to take the casual consistency to eventual getting the exact same version. These conflicts happen when two nodes update at the same time or if an update has been delayed and another node performs an update during this time. Therefore if a conflict were to happen the system would have to rollback to the last common version based on the vector clocks [4]. Then check where these updates take place in the "collective space" then start by handling each position within this space one by one. In this tool those would be the elements position in the list. Therefore the start is at the first element with an update.

Since it is hard for a computer to know which update is worth more to keep and if an update has to be scrapped it should be the less valuable one. Thus each update receives a value and during updates the ones with the highest values are kept. For the sake of this tool only changes have a value and it is only 1.

Because there are three different types of operations in this tool, which all do different things. That is why the issues occurring because of them are handled differently. The add for example is something that can always be done, it just becomes a matter of ordering which update has to be done first. Which will be done based on the updates’ values. If they have the same value it uses the nodes’ position in the vector clock, picking what it thinks is the oldest user (can pick newest as well just older usually means more experienced or higher up in the food chain as it is closer to the first node). Changes are handled in a similar fashion just here only one is done, following the same point system as before the winner is the change that is performed. The final operation delete is only performed if there is no change being done, as it is easier to re-delete something then to remake a change.
Also every time a delete or an add is performed, the positions that the other updates have need to be altered accordingly when moving to the next element. Example node A makes an add at position 1 and node B makes a change at the element on position 1. Then the outcome is node A makes an add at position 1 putting the previous element at position 1 at position 2. So node B makes a change at element 2 instead. As seen in Figure 4

![Diagram](image)

**Figure 4:** Rollback handling a conflict with an add and change

Since the tool also allows nodes to return after being disconnected for a while it means that there might be a conflict several versions ago. In order to solve these cases the module counts out the possible paths for the versions to take and then evaluates the value of each path. Example if a node A makes three changes to the same line, it would have the combined value of all three changes. While if a node B just makes an add and then right away deletes it would just be ignored apart from any possible correction that needs to be done to any update between the add and delete. Note that if a node A makes an add and then makes a change to the newly added element. The value of that change is added to the value of that add when calculating the tree.
When putting it into an algorithm it looks like this:

1. Check if conflict has occurred using the vector clock.
2. If no conflict has occurred add new update.
3. Else if an update is out of sync, then wait for the required updates.
4. Else if a conflict exists then rollback to last common version by comparing vector clock for all updates in the conflict.
5. Process each element at a time starting at the first element in the “collective space” that has gotten one or more updates (first as in first position not first to receive an update).
6. Correct to the right element for each update based on adds and deletes that concern them.
7. Take all updates that affect the current element.
8. If there are no updates affecting the current element proceed to the next element.
9. Else compute the update trees (updates that are not in conflict are put together in a tree. Such as two updates made by the same node or where node 1 has received node 2’s update before performing its own)
10. If any tree ends with an add (or add followed by any number of possible changes).
    (a) If it is the only tree with an add perform it first
    (b) Else perform the one with the highest valued tree. If they tie sort based on the nodes position in the vector clock
11. Else If any tree contains a delete.
    (a) Remove the effected updates (updates done before delete by the same tree after the last add made in the tree. No add before the delete equals all updates before the delete.)
    (b) If all trees end in a delete. Perform a delete update on that element based on the nodes position in the vector clock. (Any deletes put the trees value at zero so it has no value to compare with)
    (c) Correct the position for all elements
    (d) Redo the current element.
12. Else if only changes remain then perform the highest valued one. If two or more have the same value then perform them based on vector clock position of the last nodes in each tree
13. Repeat after moving to the next element until there are no more updates.
14. Done
3.2 Unique ID

In order to make sure that the tool can stay decentralized it needs to add a system that can grant unique IDs as well. For this tool, it is done by being given a unique ID from the node which it tried to join the network on. This unique ID is created by taking ID from the existing node and then adding a number to it before giving it away to the joining node. Example is if node A got the ID "1". Then when node B wants to join the existing network and joins on node A it will get the ID "1.X". Where X is based on how many nodes that has joined on node A before. If B is the first node to join on A it is 1, giving it the ID "1.1". If it is the second node to join it is 2 so the ID becomes "1.2" and so on.
4 Evaluating the collaborative work tools

4.1 The system architecture for the control tool

The control system which the collaborative tool will be compared against is using the same overall system architecture but with some minor changes. This collaborative work tool will use sequential consistency and a leader node to enforce it. It has been picked as the control system because it will perform the operations very fast and serve both as a peer to peer system as well as a client server solution. Since the only difference is that a peer to peer system with a leader node can reelect the leader. The reason sequential consistency is picked instead of strict consistency is because strict consistency is only needed if the "collective space" needs to be exactly the same at all times.

The peer logic module therefore does not have to handle rejoining nodes but in return it has to keep track of who is the leader, detect if the leader is dead and reelect a leader. So minor changes to it had to be done. The big difference is with the consistency module. Since for the leader node it only needs to receive an update request and then return an update after making sure it is the leader. In the update it also needs to put a lamport timestamp on it and then tick the clock. Which in turn when a node receives this update, it must sort the update based on the lamport timestamp. This in turn will ensure sequential consistency [14].

4.2 Test Environment

Both programs have been tested on the same computer, both written in java and ran in windows 10 home operating system. The computer had 16 GB ram and the processor was an Intel(R) core(TM) i7-4790. Also during the test only the consistency modules are used as the rest of the system as it would create unnecessary "noise". But also possible random delays if they were to send messages through a network. Therefore in order to follow good test practices [23, 21] it have to be testing the modules by there own. The tests are also ran as "hot" [21] as the computer might have ran other processes in the background.

4.3 Test design

The tests are designed as followed. Each of the three operation possible to perform by both of the tools, are tested first individually and then in series of three (add, change, delete) referred to as mixed test. Where each operation will be performed X amount of times while a timer is checking how long it takes to perform them. This will be repeated under different circumstances such as number of conflicts occurring, how far back the conflict occurred and how many nodes are sending updates as well as if the updates are out of sync.
Both of the tools must handle the incoming updates and execute them in order to be considered done with them. For the control tool using a leader node this means that it must approve any update requests and then handle the returning update. As in the test it is the leader node that is being tested on, since it has more work to do then a regular node. Which then for the tool being tested, this means that it must rollback the existing update and then re perform the updates after handling the conflict. This is to make sure both tools are performing simulated updates, as these are steps they would normally do.

For the delete operation the space is first filled with adds as otherwise they would be ignored and thus have greater performance. Note also that each update only modifies the first element there is no difference between modifying the first element or an element in the middle.

The reason why only these operation are tested is in order to follow good test practices [23, 21]. It would not be a fair comparison when performing for example joins as the tool using rollback, have a higher first cost when joining a new node as it needs to send all previous updates. While the tool using a leader node would only need to send all current elements in the list when a node is joining. However if the node were to drop and have to rejoin, the first tool will only need the new updates from when it disconnected while the second tool will have to get every element again. So it becomes a question on how many elements vs updates there are.

4.4 Results

The result of each operation is presented down below in the shape of tables where the average value, the mean value, the lowest value, highest value and the difference between the lowest and highest value is presented for the different circumstances during the test runs. In the table the control tool using a Leader node will be referred to as Leader and the tool being tested that uses eventual consistency with rollback will be referred to as Rollback. The tables refers to hold-back as updates that are out of sync and must wait for more updates. Like "update 2" arrives before "update 1" and has to wait. 100 hold-back there for means "update 100" down to "update 2" arrives before "update 1". For the rollback tool tests are added to see how it operates with more then one node (The leader tool is unaffected by more nodes as the algorithm only has one universal clock that only the leader can tick) but also how it operates when being forced to handle conflicts. With number of conflict referring to how many nodes are in conflict and depth in how far back they are in conflict. Note that the number of conflicting nodes are the same number as nodes active in the system.

All test data is in ms (milliseconds) and each table explains how many operations are done during that time. Example the data at table 1 performs 100000 add updates 100 times. So the leader tool then perform 100000 adds on average in 660 ms where the average comes from 100 test runs.
### 4.4.1 Adds

When looking at Table 1 it can clearly be seen that the tool using rollback is a bit slower, especially when more nodes are added or if conflicts occur. The hold-backs also only seem to affect the tool using rollback as well. Now when looking at the hold-backs and the runs with more nodes connected, the extra time taken by the tool is probably due to the vector clock. As the vector clock needs to compare them so many times in order to determine who is next and not. Since the vector clock must find the clock for each user every time it needs to compare two clocks. The tool using a leader node does take a bit longer as well when handling hold-backs, since it needs to check the updates being held back if they are in sync again whenever a update is performed. The conflicts do take more time as well but as it is 100000 of them and the time to perform a single conflict of depth one takes less than one ms. It does not seem to be a lot. Though it is important to remember that getting a single conflict doubles the time it takes to perform them and getting five conflicting nodes increases the time by almost ten times.

<table>
<thead>
<tr>
<th>Add operation test</th>
<th>average</th>
<th>median</th>
<th>lowest</th>
<th>highest</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>660</td>
<td>651</td>
<td>645</td>
<td>708</td>
<td>63</td>
</tr>
<tr>
<td>Rollback</td>
<td>738</td>
<td>712</td>
<td>695</td>
<td>1001</td>
<td>306</td>
</tr>
<tr>
<td>Leader with 100 hold-backs</td>
<td>782</td>
<td>806</td>
<td>757</td>
<td>1037</td>
<td>280</td>
</tr>
<tr>
<td>Rollback with 100 hold-backs</td>
<td>1287</td>
<td>1261</td>
<td>1251</td>
<td>1434</td>
<td>183</td>
</tr>
<tr>
<td>Rollback with 10 nodes</td>
<td>1247</td>
<td>1237</td>
<td>1197</td>
<td>1386</td>
<td>189</td>
</tr>
<tr>
<td>Rollback with 2 depth 1 conflicts</td>
<td>1486</td>
<td>1454</td>
<td>1451</td>
<td>2160</td>
<td>709</td>
</tr>
<tr>
<td>Rollback with 5 depth 1 conflicts</td>
<td>6106</td>
<td>6064</td>
<td>5927</td>
<td>6265</td>
<td>338</td>
</tr>
<tr>
<td>Rollback with 2 depth 2 conflicts</td>
<td>2661</td>
<td>2647</td>
<td>2604</td>
<td>3591</td>
<td>987</td>
</tr>
</tbody>
</table>

### 4.4.2 Deletes

Table 2 tells a similar tale as table 1 showing once more that the tool using rollback is only clearly slower when having to handle conflicts or when having to constantly check the vector clock. As having 10 nodes connected almost doubles the time it takes for it to handle a single update. Which is seen in both Table 1 and Table 2.

<table>
<thead>
<tr>
<th>Delete operation test</th>
<th>average</th>
<th>median</th>
<th>lowest</th>
<th>highest</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>665</td>
<td>664</td>
<td>655</td>
<td>750</td>
<td>95</td>
</tr>
<tr>
<td>Rollback</td>
<td>735</td>
<td>730</td>
<td>703</td>
<td>938</td>
<td>235</td>
</tr>
<tr>
<td>Leader with 100 hold-backs</td>
<td>774</td>
<td>770</td>
<td>758</td>
<td>808</td>
<td>50</td>
</tr>
<tr>
<td>Rollback with 100 hold-backs</td>
<td>1389</td>
<td>1392</td>
<td>1342</td>
<td>1545</td>
<td>203</td>
</tr>
<tr>
<td>Rollback with 10 nodes</td>
<td>1273</td>
<td>1243</td>
<td>1227</td>
<td>1434</td>
<td>207</td>
</tr>
<tr>
<td>Rollback with 2 depth 1 conflicts</td>
<td>1667</td>
<td>1676</td>
<td>1610</td>
<td>1742</td>
<td>132</td>
</tr>
<tr>
<td>Rollback with 5 depth 1 conflicts</td>
<td>2672</td>
<td>2703</td>
<td>2640</td>
<td>2875</td>
<td>235</td>
</tr>
<tr>
<td>Rollback with 2 depth 2 conflicts</td>
<td>2359</td>
<td>2356</td>
<td>2326</td>
<td>2422</td>
<td>96</td>
</tr>
</tbody>
</table>
4.4.3 Changes

Like in Table 1 and Table 2, Table 3 also shows an significant increase when dealing with more nodes in the system as well as when having to deal with Hold-backs. Though unlike with the other operations this time the tool using rollback is roughly six times slower at performing a change operation. The conflicts do of course slow down the tool quite a lot here as well.

<table>
<thead>
<tr>
<th>Change operation test</th>
<th>average</th>
<th>median</th>
<th>lowest</th>
<th>highest</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>100</td>
<td>132</td>
<td>97</td>
<td>132</td>
<td>35</td>
</tr>
<tr>
<td>Rollback</td>
<td>649</td>
<td>580</td>
<td>570</td>
<td>1356</td>
<td>786</td>
</tr>
<tr>
<td>Leader with 100 hold-backs</td>
<td>1132</td>
<td>1083</td>
<td>1077</td>
<td>2940</td>
<td>1863</td>
</tr>
<tr>
<td>Rollback with 100 hold-backs</td>
<td>7580</td>
<td>7534</td>
<td>7391</td>
<td>7732</td>
<td>341</td>
</tr>
<tr>
<td>Rollback with 10 nodes</td>
<td>6405</td>
<td>6440</td>
<td>5858</td>
<td>8884</td>
<td>3026</td>
</tr>
<tr>
<td>Rollback with 2 depth 1 conflicts</td>
<td>1796</td>
<td>1724</td>
<td>1619</td>
<td>2120</td>
<td>501</td>
</tr>
<tr>
<td>Rollback with 5 depth 1 conflicts</td>
<td>7043</td>
<td>6962</td>
<td>6923</td>
<td>7855</td>
<td>932</td>
</tr>
<tr>
<td>Rollback with 2 depth 2 conflicts</td>
<td>2625</td>
<td>2702</td>
<td>2367</td>
<td>3183</td>
<td>816</td>
</tr>
</tbody>
</table>

4.4.4 Mixed operations

As Table 4 shows the final test there is no difference with the clear extra time it takes when having more then one node in the system. Also it is a clear increase with the hold-back as in this case it goes up to 300 of them, since the operations comes in threes. It also shows that handling conflicts that goes up to 15 conflicting updates (5 conflicting nodes at depth 3) takes 250 times longer then handling a single update for the tool using a leader node. Though it also shows that the tool using rollback still takes roughly five times longer to perform a operation when they get mixed. Showing it has less of a difference when handling the different operations then the tool using a leader node. Though it is not a good thing as the tool using a leader node is just faster at handling the operations.

<table>
<thead>
<tr>
<th>Mixed operation test</th>
<th>average</th>
<th>median</th>
<th>lowest</th>
<th>highest</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader</td>
<td>97</td>
<td>95</td>
<td>92</td>
<td>126</td>
<td>34</td>
</tr>
<tr>
<td>Rollback</td>
<td>526</td>
<td>606</td>
<td>479</td>
<td>652</td>
<td>173</td>
</tr>
<tr>
<td>Leader with 300 hold-backs</td>
<td>3238</td>
<td>3219</td>
<td>3175</td>
<td>3380</td>
<td>205</td>
</tr>
<tr>
<td>Rollback with 300 hold-backs</td>
<td>25352</td>
<td>23389</td>
<td>23280</td>
<td>31448</td>
<td>8168</td>
</tr>
<tr>
<td>Rollback with 10 nodes</td>
<td>5035</td>
<td>5099</td>
<td>4773</td>
<td>5537</td>
<td>764</td>
</tr>
<tr>
<td>Rollback with 2 depth 3 conflicts</td>
<td>3097</td>
<td>3037</td>
<td>2962</td>
<td>3783</td>
<td>821</td>
</tr>
<tr>
<td>Rollback with 5 depth 3 conflicts</td>
<td>26190</td>
<td>25443</td>
<td>25407</td>
<td>35370</td>
<td>9963</td>
</tr>
<tr>
<td>Rollback with 2 depth 6 conflicts</td>
<td>8206</td>
<td>8182</td>
<td>8107</td>
<td>8622</td>
<td>515</td>
</tr>
</tbody>
</table>
4.5 Conclusion

The tool using rollback do take longer when performing updates and even more so if it has more nodes in the system due to the use of a vector clock with unique IDs. It also shows that the conflicts slow down the system a fair bit. So as a conclusion it can be said with certainty that it takes longer to run the rollback consistency algorithm than the leader node consistency algorithm. However remember that a leader node consistency algorithm requires two iterations of messages while the rollback consistency algorithm only requires one. So if there were to be a delay of 100 ms on all the messages, one extra message would take as long as performing 250000 non conflicting updates for the tool using rollback. It can also be compared to the 3818 updates when there is 5 conflicting nodes with a conflicting depth of 3, that also can be done in 100 ms. Because of this it might be a valuable tool to use when dealing with weak internet connections, as this way the system might be able to save some time. Granted that the device running the tool is on average able to perform the updates under the message delay time caused by the weak network.
References


