ZING - A Peer-to-Peer Version Control System

Guo Li[1], Pooja Naik[2], Purvi Desai[3]

***Abstract*— ZING is a distributed peer-to-peer version control system which allows users to work collaboratively without setting up a centralized server or using a repository hosting service. With ZING, every node maintains a local log for the user's unstaged data and a global log which maintains the system wide committed data. The global logs across the online nodes are synchronized during every *push* operation. The disconnected nodes can only make local commits; they need to be online in order to *push* or *pull*. ZING uses a modified two phase commit that we explain in this paper to resolve concurrent *push* requests in the system. In order to use ZING for a particular project, all a user needs is a network connection, a setup script and the IP address of any one of the nodes already in the project willing to collaborate.**

I. INTRODUCTION

Efficient communication and collaboration plays a vital role for project management when team members are not always collocated. This makes robust collaboration platforms the need of the day. Also, the number of collaborators on a project may change due to addition or removal of a developer. This makes it important to not only maintain a conflict-free and consistent code base across all the nodes, but also allow rolling back to a previous version of the codebase, tracking the evolutionary changes of the file, and managing simultaneous work on the same file. A central repository server is a single point of failure whose failure pretty much breaks the entire collaboration platform. Additionally, this client-server model has performance and scalability issues as well since the server tends to be the bottleneck in the system. Thus, we propose a version control system with a peer-to-peer design paradigm to mitigate the issues mentioned above.

ZING is a peer-to-peer version control system, that is easy to set up and is designed to support collaboration among users on text intensive projects.

A. MOTIVATION

Some of the widely used collaboration platforms today are Dropbox and Google Drive. While these tools are extremely powerful for sharing data, they lack the feature of version control. Therefore, several version control systems such as SVN, CVS, and Git have gained traction. While these systems have made project collaboration and development a lot more efficient, we can see that under the hood these systems use a centralized point of communication and storage. Designated servers need to be online always to store the data and respond to user requests. For instance, if two users collaborating on Github need to exchange file changes, they need to contact this central repository in the current model. Thus, we wanted to design a collaborative platform that addresses these limitations.

One such communication paradigm that allows users to collaborate among themselves without requiring a designated server is the fully decentralized peer-to-peer paradigm. As opposed to the client server paradigm, in the peer-to-peer model every user is equal in the system in terms of responsibilities and resources. We could think of each behaving as both, client and server, based on the operations being performed. Some of the advantages of peer-to-peer protocol are low maintenance costs, no single point of failure, scalability, and collective ownership [1].

In this paper, we aim to explain the system design and implementation of our peer-to-peer version control system, ZING, that leverages the advantages of the peer-to-peer communication paradigm to do away with the challenges of the client-server architecture. Section II describes ZING's basic system architecture, Section III delves into the details of the communication protocol used, Section IV discusses the algorithms used in the implementation, Section V describes the implementation, Section VI walks through the evaluation, and sections VII, VIII and IX provide closing remarks, observations and note the scope of improvement.

II. ZING'S BASIC SYSTEM MODEL

ZING is a symmetric peer-to-peer version control system. There is no special node in the group. We build ZING over git and use git to handle the low level repository management. On top of git, each node contains a client module which handles all the user inputs, and a server module, which serves requests that come from other nodes.

The high level architecture is shown in Fig. 1

A. Underlying Filesystem

We used the git version control primitives to implement our underlying filesystem. Any working directory that uses ZING for version control would have a global tracking directory or global log,as we call it (that is not visible to the user) nested inside the local directory or local log. Internally, we have 2 git repositories for version tracking, one in the local directory and one in the global directory.

The global repository on every node maintains a consistent global view of the entire system. Thus, every node may have its own version of the local working set of the project. However, when a node wants to push its changes to the entire system, it pushes the changes to its global repository and the global repository of all the other online nodes. On the

[1]Guo Li, gul027@eng.ucsd.edu
[2]Pooja Naik, panaik@eng.ucsd.edu
[3]Purvi Desai, pdesa1@cs.ucsd.edu

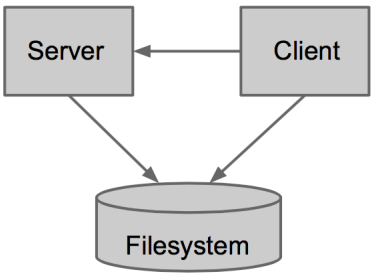

Fig. 1. A node in the ZING system

other hand, when a node wishes to pull the changes that have occurred in the system, it pulls only from the global repository in its local machine.

The filesystem also maintains a log file, storing all the pushes that happened in the system, a metadata file, containing all the metadata: the IP addresses of all the nodes in the system, the index number of this node etc.

Fig. 2 shows the basic architecture of the underlying filesystem. From the vantage point of a single node, we can look at the rest of the system as a virtual global repository.

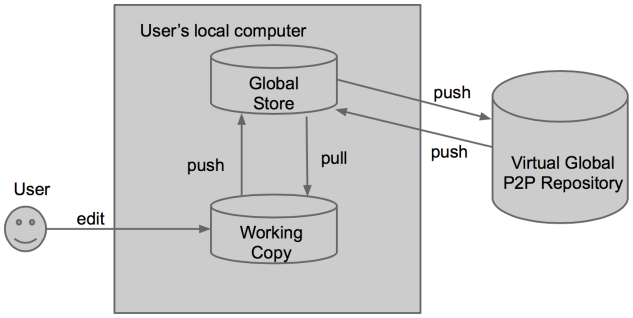

Fig. 2. ZING's Filesystem Architecture

B. Client

Client is the module that interacts with user.

For all the operations we currently support (Init, Clone, Add, Rm, Commit, Status, Log, Pull, Push, Revert), users start the client when they call a command on the terminal. The client process will then process the command, sending messages to the local or remote server as the operation requires.

Operations Init, Add, Rm, Commit, Status, Log and Revert happen locally and are implemented by just calling the underlying git command.

For the Pull operation, we pull the latest changes from the global repository to the local repository, which also does not involve the network.

For the Push operation, a node will generate a patch file for the changes it made, and then broadcast the patch file to all the nodes in the group.

The Clone command is the way how new joins an existing system; it will fetch the latest state of the project and request all the members in the system to update the metadata.

C. Server

Every node maintains a server that listens for requests continuously.

The main job of the server is to process remote pushes, where it receives the pushes from remote nodes, writes them to its globally repository and logs the changes in the log file. A server can go offline since it is hosted on a user's local machine. Every time it comes back online, it will catch up with other nodes to get up-to-date with the current state of the system. When a new node is trying to join the group, servers will receive the request and update the metadata of the system.

The basic communication protocol in our system is a (customized) two phase commit protocol which we explain in more detail in the next section.

III. Communication Protocol

In our peer-to-peer system, all the nodes operate independently and the whole system works collaboratively to make progress. Since all the nodes in the system maintain a single global view of the entire system, we need to ensure consistency of this view across all the nodes in the group. Different events could occur independently and hence, the system should be able to ensure consistency in the face of system-wide concurrent events. Since we are implementing a version control system, we need to guarantee a total ordering on the push operations.

Therefore the system should adopt a protocol that could serialize all the events and maintain a total order across all the nodes. The ABCAST protocol provides such functionality. However, given the design and constraints of our system, we propose another flavor of the two phase commit protocol. The following subsections present the details of the protocol, and in later section we explain why we use this protocol instead of ABCAST.

NOTE: The system indexes all the nodes linearly.

A. Two phase commit

- **Prepare Phase:** For every system-wide event such as a push or an offline node coming online, the node that issues the event will first broadcast a prepare message to all the nodes in the system, including itself. Each node maintains a prepare message queue and appends every incoming prepare message to the end of this queue. A node decides whether its prepare phase succeeded or not by checking whether the position of its prepare message is first in the prepare queue of the *first alive node*, according to the indexes of nodes.
  If multiple nodes issue a prepare message simultaneously, then only one of them will succeed since only one prepare message will reach the first alive node first.

- **Commit Phase:**
  If the prepare request of a particular node succeeds, it will broadcast a commit message, otherwise it will broadcast an abort message. The prepare message in a node's prepare queue will not be removed until the corresponding commit message or abort message arrives. This means that when one node's prepare request succeed, all the following prepare requests will never go through until this node finishes its commit phase.
  When a node receives a push message or an abort message, it will first conduct operation and then clean the corresponding prepare message in the prepare queue.

Initially, we designed this protocol only for serializing push events. Hence, we use the term *push message* in place of commit message. Later we adopted this approach for several other events in the system, which we will describe later. For the rest of this paper, we continue to use the term push message to imply the commit message in the two phase commit.

B. Prepare queue processing

The prepare queue of each node is a FIFO queue. A new prepare message will only be appended to the end of the queue, and the old prepare message will only be removed from the front of the queue.

In a distributed system, messages to a node may come out of order. A node can receive the prepare message from A before it receives the prepare message from B. However, it could receive the push or abort message for B before receiving the that for A.

Here we enforce a rule that all the pushes or abort message a node receives will only be processed in the order of the prepare message stored in its own prepare queue. In the previous example, if a node receives a push or abort message for prepare B first, it will store this message temporarily in another place. Later when it receives the message for prepare A, it will process these two messages together and remove the prepare messages from A and B from the front of the prepare queue at once. So the prepare queue will always be first in first out.

C. Correctness and Robustness

As we said before, all the events in the system should be serialized and in a total order. Using our two phase commit protocol, if multiple events happen concurrently, only one of them will go through. And at any point in time, there could only be one event going on, since the corresponding prepare message will not be cleaned before it receives a push or abort message. Therefore all the events will be serialized.

The serialization guarantee ensures that the prepare messages for all the successful events in the system will be in a total order. Since an abort message will do nothing but clean the prepare message from the queue, the order of unsuccessful events doesn't matter to our system. Each node processes all the push messages in the order of the prepare message it received, therefore we guarantee that all the successful events in the system will be in a total order.

The logic we use to decide who wins the prepare phase competition is to check whose prepare message reaches the first alive node first. However, this logic breaks if the first alive node goes offline while nodes are issuing prepare requests. For example, if a node's prepare message reaches the first alive node 0 first, and then node 0 goes offline. At the same time another node may broadcast a prepare message and it may reach node 1 first, which is the first alive node in the system at this point in time. This would cause both nodes to falsely consider their respective prepare phases successful.

In order to circumvent this case, we add a check after a node has gathered all the responses of the prepare request to see whether the node it thinks as the first alive node is still alive. If it is not alive, this prepare request fails. This helps prevent the issue described above. Besides, if the first alive node goes offline after the check, since we have already gathered all the responses of the prepare requests, all the following prepare messages will come after this prepare message in the prepare queue of every node in the system. A node come online is a system event that will be serialized with other events, so it will not break our logic of first alive node.

We discuss the case where a node dies while broadcasting the prepare message, or a push or abort message in the discussion section.

IV. ALGORITHMS

ZING add, rm, init, commit, and pull operations happen locally without conflicting with any global events. The main logic of these commands is similar to the corresponding git commands, so we will not explain them here.

The following subsections outline the various scenarios where serialization is mandatory for the overall consistency, and how our system handles it.

A. ZING push

1) Serialization: For every push, the node that issues the push will broadcast the patch file to all the members in the group. All the pushes that every node receives need to be in the same total order to have a consistent trace of versions at every node.

We use the two phase commit protocol mentioned before to ensure this form of ordering, but the system requires more than that. In a version control system, a push could potentially cause a merge conflict that can not be resolved automatically, and the person who causes the conflicts needs to resolve them before finally pushing the changes to the global repository. Basically, there shouldn't be any conflicts globally. For a client-server system, it is easier for the server to resolve the order of different pushes, and report merge conflicts to a client that causes them. In a distributed peer-to-peer system, however, one person could make a push before receiving a push that was in progress. If these pushes are conflicting, every node that receives these pushes will be in a conflicted state.

This implies that the system should not allow simultaneous pushes to happen i.e. if a node's push operation is in progress,

all others should wait until they receive that push before performing a new push operation. In this case, if the new push were to cause a conflict, it would only trigger the conflict locally. This is the reason why we haven't use ABCAST [5] as our communication protocol, since ABCAST only guarantees a total order of messages, without disallowing simultaneous messages to go through.

In ZING, before one node tries to make a push, it first checks whether its prepare queue is empty. If not, it will abort this push operation.

2) View: Another case concerning the push operation is the number of active nodes in the system. Different aliveness status of nodes in the system forms a different *view*. If two different views don't have an intersection, then changes made in different views will bring conflicts to the system.

For example, suppose there are 5 nodes in the system and only 2 of them are currently online. They may make some pushes and then go offline. Thereafter, if another 2 nodes come online they may make some changes. Later when all the the nodes come online, the system is in a conflicted state.

To prevent this from happening, our push operation enforces a rule that one node can only make a push when there are more than $\frac{n}{2}+1$ nodes online. The 'n' here is the total number of nodes in the system. This ensures that there will always be a node that received all the changes that happened in the system.

B. Offline node comes online

The nodes in our system can come online and go offline at will. Whenever a node goes offline, it will not receive any pushes. This means that a node can miss the pushes that happen when it is disconnected. Next time this node comes online, it needs to catch up with the current state of the system i.e. read the missing data from someone else in the group. But it could also be the case that this node that came online is more up-to-date than the other online nodes in the system. In this case, this node needs to catch the other online nodes up in the system.

The first case is easy to understand. When a node is offline, it cannot receive pushes any more. The nodes that are still online can make pushes that won't reach this inactive node. Our system guarantees that the data is consistent across all the active nodes, so when a node comes alive, it can fetch the missing data from any online node.

The second case could happen when an offline node comes online while the majority of the group was offline. Suppose there are 5 nodes in the system and only 3 of them are online. These 3 nodes could have made several pushes. Now if these 3 nodes go offline and the other 2 come online, they cannot catch up because all the 3 nodes with the latest system state are offline. They also cannot make pushes since they don't form a majority. Now suppose one more node comes online. It will find out that all the nodes that are currently alive have missed some data from a previous view. This node now needs to forward the changes it has to those that are online.

Thus, ZING provides a 2-way catch up mechanism for the system to become consistent and up-to-date once again after an offline node becomes online. We use the phrase *catch up* to refer both operations.

C. Node comes online while a push is in progress

Every time a node comes online, it will catch up with the nodes currently online. After catching up, it will then receive all the new pushes in the system directly. There could be a situation where a node comes online while a push is in progress. In this case, the node could miss this push without even knowing it. The system must therefore serialize the push operation with a node coming online.

The two phase commit protocol now comes into play. When a server comes online, it will broadcast a prepare message and start processing the catching up logic only after the prepare phase returns true. After prepare phase returns true, the node that just came online will, instead of broadcasting push messages, start to catch up first, read the missing data from some node in the system. During this process, the prepare messages of this online event remain in everybody's prepare queue, which means all the other prepare request during this time will return false. The new node will broadcast a push message only after it finished the catching up.

While a node is catching up, it actually disallows any other operation from going through in the system, since the prepare queue of every node is occupied by the prepare message of this come online operation. An optimization we can make is to enable a new online node to receive pushes while doing catch up. This requires the program to allocate a temporary queue for the pushes received during catching up.

D. New node joins the system

The logic of handling a new node joining the system is similar to handling an offline node comes online. This node needs to catch up before actually becoming a part of the system wherein it can participate in all the transactions in the system. We send a prepare message to all the nodes in the system requesting to join the system and then catch up with the system's current state. This is how we serialize this operation with other events.

One difference is that the system needs to assign an index for this new node and everybody needs to add the this new node to their metadata. This index must be recorded as the same value across all replicas. We use a special push message to accomplish this job. After the prepare phase returns successfully and the new node finishes catching up, it broadcasts a special push message which, when received by the other nodes in the view, will trigger them to add the new node information to their metadata. The index of a new node is assigned as the number of nodes currently in the system plus one. Since node joining events are serialized, concurrent joining nodes will not get same index number.

A node cannot issue a push request until its prepare queue is empty. Hence, at the time it can make a push, it should have already received the special push message and appended the new node to its metadata. After that, all the following pushes are be visible to the node that just joined the system.

Another difference from handling a node coming online is that a new node cannot join the system unless there are a majority (n/2 + 1) of nodes online. The reason for this rule is the same with that for push - there will always be one node in the system that contains all the information. At a high level, a new node joining the system will change the state of the whole system just like a push does, and for every operation that could make changes system-wide, we should enforce this majority rule.

V. SYSTEM IMPLEMENTATION

We implement our system using Golang, and use shell script wrappers over the git APIs for our filesystem. We implement all the network communication using GoRPC.

A. ZING Server

The server listens on a port for incoming RPC calls.

1) **receiving prepare** The prepare queue is maintained by the server using an array. The RPC function will simply append an incoming prepare message to the end of the array and return true or false by checking whether this message is the first one in the queue.
   Each prepare message is a structure containing the index of the sender, the version number of this prepare message, and the ip address of the sender. Each node maintains an integer version number for every push it makes, so an index with the version number will make every prepare message unique across the system.
2) **receiving push** The structure of a push message contains a prepare structure inside, indicating which prepare message it is associated with. Besides, it also contains a byte array containing the content of the patch file.
   When processing each push message, the RPC function first makes a patch file. It then calls git amend to apply the patch file to the global repository, and writes the whole push structure to the log file. The log file contains an array of push structures the server ever received, and the push are written in the order they are processed, which implies the order of pushes in the log are the same for each node.
   An abort message will simply be a push structure in which the patch array field is empty.
   A special prepare and push message is indicate by a special value of node index and version number fields in the prepare structure. Once a server receive a special prepare message, it will append the IP address field in the prepare structure to the metadata file.

B. ZING Client

We figured our system would be most useful if it adhered to the existing git semantics that users are most used to so as not to affect the speed of user operation significantly. This section describes each of the commands and its design in detail.

1) **zing init:** This command initializes a new repository. It creates the metadata and log files under the .zing directory, sets its own index as 0, since it is the first person in the project. The command maps the index to its IP address, and sets its version to 0.
2) **zing clone:** This command first initialize an empty repository, and then fetching data from the node that are specified by arguments of this command.
3) **zing add:** This command adds a file to be tracked by ZING. We implement this by calling the "git add" command.
4) **zing commit:** This command commits the file, thereby adding it to the node's local log. we implements by calling "git commit".
5) **zing pull:** This command pulls the data that is not in the local git repository from the global repository.
6) **zing push:** This command pushes the data that is not in the global repository from the local repository. Internally, the global repository branches to a separate temporary branch to pull the local changes from the user. Thereafter, it prepares a patch of the changes since the previous commit that will be sent over the network to all the nodes in the system in the subsequent 2 phase commit. We need this temporary branch so that we do not disturb our own global repository in the case where our push is rejected/aborted by the system.
   The client reads the IP address list of all the members in the system from the metadata files, generates patch files using git format-patch command, and sends it to all the members in the system. We implement the broadcast by issuing each RPC function using a go routine.
   The client will finish sending the prepare messages. Regardless of the status of the prepare phase, it makes RPC function calls to its own server (the server running on this node) and lets the server broadcast the push messages or abort messages. The client process then returns without waiting for the broadcasting to finish. The asynchronous nature of this operation here improves the performance.

C. Catching up

When a new node joins the system, or an offline node comes online, it will fetch all the missing data from some other nodes. We use the log file to figure out what to fetch.

As we said, the log file contains an array of push structures in which the order is the same across all the nodes. So a node that issues the catching up operation simply calculates the difference between two log file by subtraction, fetches all the missing push structures, and applies them to its own global repository one by one in the same order.

One problem with this strategy is that the log file needs to increase monotonically. So for roll back, we implement it using a git revert, which will create a new push for the roll back operation.

VI. EVALUATION

Keeping in mind the nature of our system, we deemed it appropriate to compare with collaboration platforms used

by smaller groups. In this section, we describe the results of our evaluations after performing comparisons between ZING, git based client-server model on the same network, and GitHub. Git server-client model is the preferred version control system for small and large groups alike and is used by our system to implement the filesystem as described above. GitHub is a popular web-based git repository hosting service which we used to host our code for this project.

A. Methodology

1) Measuring the performance of ZING for push operations: We measured the performance of ZING for push operations under the following scenarios:
   - Fixed file size with varying number of nodes in the system: In this scenario, we fixed the file size to 10MB. We then performed the push operation for this file size with the number of nodes in the system varying from 2 to 64 and measured the time taken for every trial.
   - Fixed number of nodes in the system with varying file sizes: In this scenario, we fixed the number of nodes in our system to 20. We then performed the push operation with file sizes ranging from 2KB to 64MB and measured the time taken for every trial.
2) Measuring the performance of ZING for pull operations: To measure the performance of ZING for pull operations, we performed a pull operation with a fixed number of nodes in the system with file sizes varying from 2KB to 64MB and measured the time for every trial. We do not need to vary the number of nodes in the system for this measurement since the pull operations are always local, as described in the sections above.

In order to compare our performance with GitHub and a local git client-server model, we did the following experiments:

- Measuring the performance of push and pull operations with GitHub: We cloned a GitHub repository on our local machine. Thereafter, we measured the time taken across multiple trials to perform push and pull operations for file sizes ranging from 2KB to 64MB. Since this is a client-server model, we do not need to vary the number of nodes in the system since there is only a one-way interaction during both push and pull operations.
- Measuring the performance of push and pull operations with a local git server: In order to have a more fair comparison, we decided to compare ZING with a git based version control system that has the client and server on the same network, unlike GitHub. We set up the git server locally on our machine. Thereafter, we measured the time taken across multiple trials to perform push and pull operations for file sizes ranging from 2KB to 64MB just as we did in the experiment described above. Since this is a client-server model, we do not need to vary the number of nodes in the system since there is only a one-way interaction during both push and pull operations.

We also performed an experiment to measure the metadata storage overhead of Zing compared to a local git server for file sizes ranging from 2KB to 64MB.

B. Results

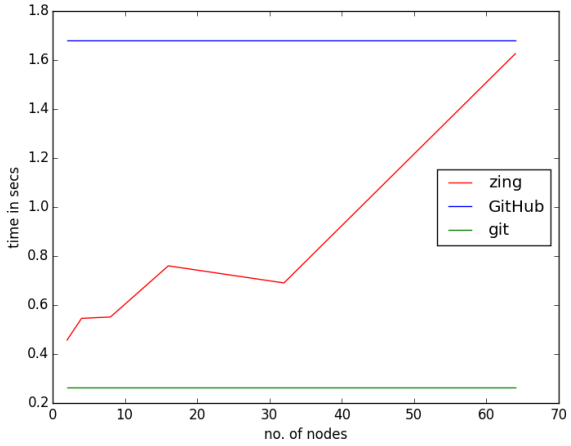

Fig. 3. Push Time in secs v/s no.of Nodes

To measure the impact of the number of nodes on the push time, we plotted the average time taken for a push (measured across 100 trials) by gradually varying the number of nodes (see Fig. 3). We do see an increase in time with the number of nodes which can be attributed to the fact that our pushes involve broadcasting. We expected pushes to take longer in Github due to the remote location of the server. However, it is observed that our system's performance almost matches that of Github at 64 nodes. Network latency can possibly account for the minor inconsistencies observed.

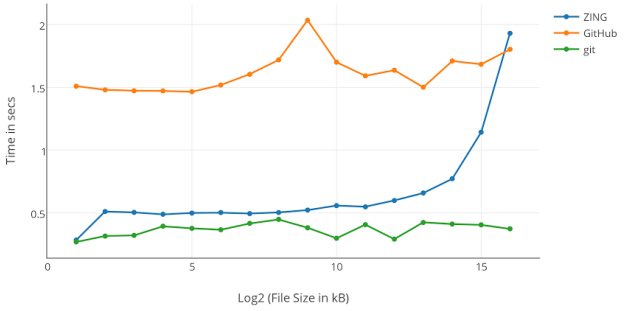

Fig. 4. Push Time in secs v/s Size of File

Fig. 4 shows how the average push time varies with the patch size (in our case, file size because we are pushing newly created files). Our system performs pretty well in comparison to git here, diverging significantly only for files larger than 4MB. However, the performance suffers exponentially thereafter, matching that of GitHub and surpassing it close to 64MB. GitHub does not allow files larger than 100MB

so we did not proceed with the tests past 64MB across all systems.

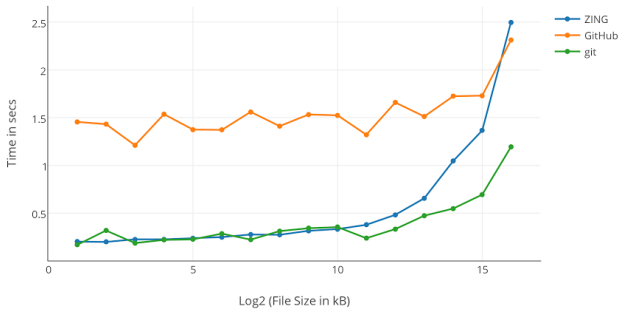

Fig. 5. Pull Time in secs v/s Size of File

ZING performs incredibly well and matches git (see fig. 5) in the case of pulls for files upto 1MB. However, it diverges significantly and becomes exponentially expensive for larger files, surpassing GitHub at 16MB. For our system, pull is just a data transfer from the global log into the local log and the low latency for small files is not surprising. However, for larger file sizes, the disk operations bring in latency. This is reinforced in fig. 6.

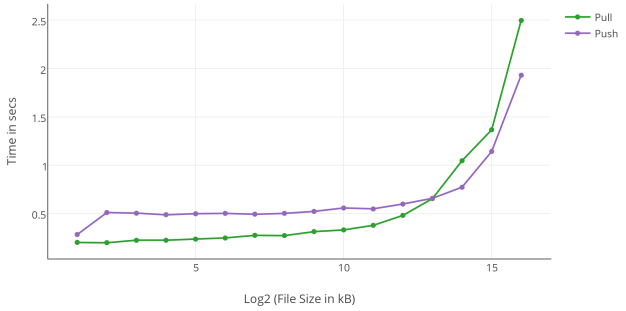

Fig. 6. Push and Pull Comparison by File Size

Due to the high latency observed in the pull in fig. 5 we decided to compare pull and push time by file size (see fig. 6). The resulting plot gave interesting results. For files larger than 8MB, the pushes are faster than pulls. In fig. 5, we see that this is exactly where ZING pulls diverge significantly from git pulls. We attribute this behavior to the fact that at large file sizes, network operations are faster than disk operations.

To compare the storage overhead, we measured the sizes of the .git directory in Git (which wont change with GitHub) and the .zing directory in ZING. The results are described in Table 1 in the Appendix section. The large overhead in ZING is observed because of the patches stored in the global log. Also, it should be noted that the local log and the global log are in fact git repositories and contribute to this overhead, though not in a significant way.

VII. DISCUSSION

When we first started designing ZING, we visualized a system that would be easy to set up and use. ZING can be set up on any machine without requiring any additional tools or libraries. The motivation behind comparing ZING with GitHub was to show how much better our system performs when it comes to small groups working on projects like ours.

The fact that the performance of ZING is significantly degraded when it comes to large files will not affect the normal working of a system because the files in the evaluation will, in reality, be patches and we don't expect a text path to be in the order of Megabytes except when a new node joins or old nodes comes online after an extended time, in which case, we believe, such a delay is permissible and expected. One of our aims was to make the pulls faster than the pushes analogous to reads being faster than writes because the user should be affected by minimum latency in resuming work. Our results reinforce that, keeping in mind the nature of the work users collaborate on, as discussed. Apart from the reasons mentioned above, ZING also has the P2P advantage that all the code resides only on the machines of the collaborators involved in the project. For example, had the students used ZING to work on the labs in CSE 223B, there would be no fears of code being available on public repositories and no measures would have had to be taken by the TA to set up the git server and ensure that it is fault tolerant.

The results also, however, draw attention to some glaring drawbacks of our system. Even as a peer-to-peer system, we realise that ZING is not scalable due to the linear increase latency with the number of nodes. We also require a majority to be online to make a push and this is often not possible, especially as the number of nodes increase. Most importantly,in our current implementation, the global log that is so vital to the functioning of this system will never reduce in size and so, as the project grows, the storage used by ZING will increase, which will add to the space already used by the files users are collaborating on.

Optimistically, since a majority of projects on GitHub have less than 50 contributors [3], we can say that ZING is very relevant in the present text based collaborative work environment.

VIII. FUTURE DIRECTIONS

In this section we highlight the future scope of our project in terms of enhancements.

One of the basic enhancements that our system would benefit from is allowing users to form subgroups within a larger and collaborate within themselves without affecting the overall functioning of the parent group by working on a branch of the parent project. In this way, we can ensure that some users can still collaborate on the work while they do not have majority. We are yet to work out the details of this approach.

Another enhancement would be the incorporation of fault tolerance in our system. Currently, we only support nodes coming online and going offline voluntarily. However, a node could practically be unreachable at any point in time, either due to a network partition or because the node crashed. We have designed a model that would take care of the above mentioned situation, but we did not get to its implementation due to lack of time. Our design for a fault-tolerant version of ZING is explained with the following scenarios:

1) The first alive node fails: This node is used to enforce a total order on all the system-wise events. The failure of this node may lead to an inconsistent state. We have handled this issue in our current implementation already.
2) A node fails in the middle of broadcasting the prepare message: We would timestamp every prepare message with the time of the receiving node. If a node fails after sending a prepare message to just a few nodes in the system, the nodes that received this prepare message will wait until a certain threshold elapses. Thereafter, these nodes will talk to the other nodes in the system to find out if any of them has received a push message from the sender. Since the sender failed in the middle of sending the prepare messages, these nodes find out that no one in the system received any commit message. Hence, they just drop the prepare message and move on.
3) A node fails in the middle of broadcasting the commit message: Again, we would timestamp every prepare message with the time of the receiving node. If a node fails in the middle of broadcasting the push message, there will be a few nodes that just received the prepare message but did not receive the push message. After a certain threshold has elapsed, these nodes can talk to other nodes in the system to check if they received a commit message from the sender node. If any node has received a push message, it means the sender decided to commit the operation before failed. Hence, all the nodes that received the prepare but did not get the push message assimilate this push message.

We also believe there is scope to reduce the space overhead of the .zing directory and are working to come up with a solution that involves cleaning the global log and using the information in the local log to reconstruct it, if required, and reducing duplication.

Also, currently our rollback is implemented analogous to a push so that our operation version is always increasing for the ease of design and implementation. At present, the log files seem redundant since we can always get the data they record from the global git repository. However, we could use the log file versions recorded to rollback to previous versions of an ongoing project so that we do not need to have an always increasing operation number. Thus, we could implement an operation analogous to git reset that wipes the entries from the system and also from the log.

IX. CONCLUSIONS

A version control system plays a vital role in order to support efficient and accurate collaboration among peers working on a project. Its main functions are to track the evolutionary changes made to the working set of files and to manage changes being concurrently made on those files. In this paper, we have discussed the limitations of current version control systems in the context of the inefficient client-server paradigm for communication, the presence of single point of failure, and scalability issues. As a means to overcome these limitations, we designed and implemented a version control system based on the peer-to-peer communication paradigm. However, we had to face a new set of challenges intrinsic to the peer-to-peer paradigm. The lack of global knowledge which is crucial when pushing and pulling updates and finding was one of the main challenges. Having a consistent global view across all the replicas was another challenge we faced. Our system design overcomes these challenges and proves that the peer-to-peer paradigm can be leveraged to contribute its inherent strong points in the context of a version control system.

REFERENCES

[1] Mukherjee, Patrick. A Fully Decentralized, Peer-to-Peer based Version Control System. Diss. TU Darmstadt, 2011.

[2] Terry, Douglas B., et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. Vol. 29. No. 5. ACM, 1995.

[3] Lima, Antonio, Luca Rossi, and Mirco Musolesi. "Coding Together at Scale: GitHub as a Collaborative Social Network." arXiv preprint arXiv:1407.2535 (2014).

[4] Wolski, Antoni, and Jari Veijalainen. "2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase." Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on. IEEE, 1990.

[5] Birman, Kenneth, and Robert Cooper. "The ISIS project: Real experience with a fault tolerant programming system." Proceedings of the 4th workshop on ACM SIGOPS European workshop. ACM, 1990.

[6] Liskov, Barbara, et al. Replication in the Harp file system. Vol. 25. No. 5. ACM, 1991.

APPENDIX A

TABLE I
STORAGE OVERHEAD

System	File Size												
	0	2KB	4KB	8KB	16KB	32KB	64KB	128KB	256KB	512KB	1MB	2MB	4MB
Git / GitHub	88KB	100KB	112KB	124KB	136KB	148KB	160KB	172KB	184KB	196KB	212KB	232KB	260KB
ZING	186KB	224KB	252KB	284KB	324KB	380KB	468KB	624KB	904K	1.4MB	2.4MB	4.5MB	8.6MB