VillageShare: A Delay and Fault Tolerant Approach to Remote Village File Sharing

Student: Sarah Jones

Committee: Elizabeth Belding, Amr El Abbadi

Abstract

Introduction

Motivation

culture, economy

Design objectives

File sharing software that
1) contains a network of remote areas
2) can operate with link failures and server failures (fault tolerant)
3) can operate on a bandwidth limited link (delay tolerant)

Owncloud

Open source self-hosted Dropbox-like software
Allows users to upload files to a local server
Allows a user to share files with other users
Some applications such as image and pdf viewers and music and video players
Owncloud consists of a core and apps repositories (primarily)
apps allow to extend owncloud functionality

Because owncloud is intended to be for personal or corporate use (an environment where membership is controlled and behavior expected to be non-malicious), sharing is possible between all users on an owncloud server. In order to avoid spamming of shared files, we would like to restrict who a user can share files with.

My project consists of two apps:
1) Friends - establishes two way friendship between owncloud users; allows for Facebook synchronization; limit sharing so that a user can only share with users they are friends with
2) MultiInstance - creates a network of owncloud instances, so that a user can share files with users on other owncloud instances (and access his/her own files on another instance); uses rsync for synchronization

Previous Work

Internet cost
Internet down == no file access

Friend App

The Friends app protects a user from unsolicited sharing of files by implementing a two-way agreement for file sharing. If a user wishes to share a file with another user, the user visits the Friends app (installed on the owncloud instance by the administrator) and requests a Friendship with a user. The receiving user will then see a request for Friendship in his/her account Friendship app, and can accept or reject the Friendship. Once the Friendship is accepted, the two users can share files with each other.

In order to keep the code as clean and modular as possible, most pieces of the Friends app are in the apps repository. The owncloud core had to be modified so that file sharing could be limited to just Friends; this was added with conditional statements that check whether the Friends app is enabled and if so, executes the new behavior. The new behavior is 1) suggesting only Friends for the autocomplete when sharing a file, 2) checking that the user is a Friend before sharing the file.

Social Networking

In order to help an owncloud user build a list of friends to share files with, we implemented a Facebook integration feature that import Facebook "friends" into owncloud. The Friends app utilizes the Facebook API to fetch the user's Facebook identifier. This requires a user to login to Facebook (requires Internet) and to grant the Owncloud Facebook App access to his/her basic profile information (name, id, friends). Once the user grants this permission, the user's Facebook identifier is stored in the owncloud database. Then the user clicks a button to fetch his/her current friends list; this list is compared to the Facebook identifiers stored in the owncloud database, and any matches result in the automatic creation of owncloud Friendships between the user who just synchronized with Facebook and the users that matched. A user can synchronize as often as he/she wishes.

Previous interviews with and surveys from Internet users in Macha indicate that video and picture uploads to social websites like Facebook and Youtube fail a majority of the time [site David?]. While owncloud can provide an alternative to social media websites for sharing files, it can also facilitate it. owncloud has a built in feature that allows users to share a link through a public url (no login required). A user can then post this url on a social media website (e.g. Twitter, Facebook status); anyone who views that post can click on the link and view the shared file (image, video) through owncloud. The advantage of this is that anyone who views this link in the same village (behind the same satellite link), will be able to download and view the file very quickly; the request will never cross the satellite link and thus the latency will be significantly reduced. This means that an owncloud user will not have to worry about uploading failure of their file to a social media website, and those within the village will not have to worry about the downloading of the file failing.

MultiInstance App

The MultiInstance app seeks connect instances of owncloud together to create a delay and fault tolerant network that will allow a user to share files with users outsides of his/her local community. The basic model of this app is that there is a central server, in this case UCSB, and many non-central servers, in this case one for each each remote village. We assume that the central server will be reliable (e.g. no power issues) and has sufficient resources (e.g. stable Internet at a fast speed, disk backups). Non-central servers are likely to be unreliable (e.g. frequent power outages and brownouts) and connected to the Internet with a slow link (e.g. satellite link), which may also have frequent outages.

Given the unreliability of the non-central servers, we depend on the central server for communication between non-central servers. Non-central servers push relevant data, including users and files, to the central server. This way if the link between non-central server A and central server UCSB goes down, non-central server B can request a piece of A's data from UCSB. This also allows the central server to act as a backup for for the non-central servers.

There are two cases where a non-central server may need information from a non-central server. The first is if a user in village A (non-central server) wants to share a file with a user in village B; information about the Friendship between the two users needs to be shared, each other's usernames need to be shared, and the file itself needs to be shared. The second case is where a user travels; when a user from Village A travels to Village B it should be able to login (requiring user information) and view his/her files in a reasonable amount of time.

Add here about usernames@location**

Pushing Data to the Central Server

All non-central servers must push their data to the central server in order to keep the data in the central server updated.
When an event happens on a non-central server (e.g. user creation, user update, friendship creation), the data is queued to be sent to the central server. Currently, each object type (User, Friendship, UserFacebookId) has its own table to store the queued data as well as the timestamp of the event; each object type has a set of one or more fields that make it unique (e.g. a user has a unique id). On a regular interval, the contents of all the Queued<Object> tables are dumped into files. Using rsync, these files are synced to the central server.
The Queued<Object> files are then read into their counterpart tables Received<Object> of the central server. The Received<Objects> are then processed one at a time: if the event timestamp is more current (greater than) the last time the <Object> was update, or it does not yet exists, then the <Object> is updated/created. The central server then dumps an application-level acknowledgement for each Received<Object> that it processed (where the information was used for an update or discarded) into a file which is then synced across (on a regular interval) to the non-central server.
The non-central server will then read the acknowledgement and removal all <Objects> listed in it from the respective Queued<Object> table. If an event <Object> or are acknowledgement manages to get lost due to a bug or a system failure (e.g. power failure) that rsync cannot recover from, the <Object> will be dumped from the Queued<Object> table and synced to the central server until it is successfully processed and an acknowledgement is received.

Acknowledgements

It is not obvious when an acknowledgement should be deleted from the central server. Each bulk acknowledgement is in a file that is synced across from the central server to the non-central server. Overwriting the acknowledgement file with new acknowledgements runs the risk that the acknowledgements will get lost; this could happen if a link goes down or if the processing and

syncing (from the central server to the non-central server) of tasks takes so long that a new set Received<Object>s has been received and processed before the syncing completes. To avoid this, all acknowledgements are written to a file with a name containing the current timestamp. When syncing is about to start the current timestamp is written into a file named last_updated.txt which is also synced to the non-central server. The non-central server has a file named last_read.txt which contains a past timestamp. All acknowledgement between the last_read.txt timestamp and the last_updated.txt timestamp are then processed. The value of last_updated.txt is copied to last_read.txt.
The acknowledgement files will accumulate. At some time interval (e.g. every night at midnight), all acknowledgement files with a filename containing a timestamp less than last_updated.txt will be deleted on the central server. The next sync will delete them on the non-central server.

Pulling Data from the Central Server

It is difficult to know when data for one non-central will be needed on another non-central server; for this reason we request data from the central server when an event happens that requires another non-central server's data.
In this case, an event may be a login attempt (e.g. for a travelling user) or the requesting of a Friendship with a user at of a different server. A request for a user's information (or whatever data the event may require) is queued into the QueuedRequest table. The QueuedRequest table is dumped, synced, received, and processed on the central server like the other Queued<Object>s. When a ReceivedRequest is processed the required data is queued to be pushed to the requesting server and a response to the request is added to the QueuedResponse table. The QueuedResponses are dumped, synced, and then processed on the non-central server; processing includes any database updates like deleting Friendship requests to a nonexistent user as well as deleting the QueuedRequest.
QueuedResponses are deleted from the central server database after they are dumped into a file for syncing.

Pushing Data to a Non-Central Server

In the case of a Request/Response it is the same process as the Push Data to Central Server but with the Non-Central Server and Central Server roles swapped.

Security

Rsync + ssh with private keys

Traffic Control

tc [cite previous VillageShare paper? or something from David?]

Simulation and Results

VirtualBox

Academic Network?

Bandwidth Consumption

Future Work

Conclusion

Acknowledgement

To owncloud.org contributors especially B

References