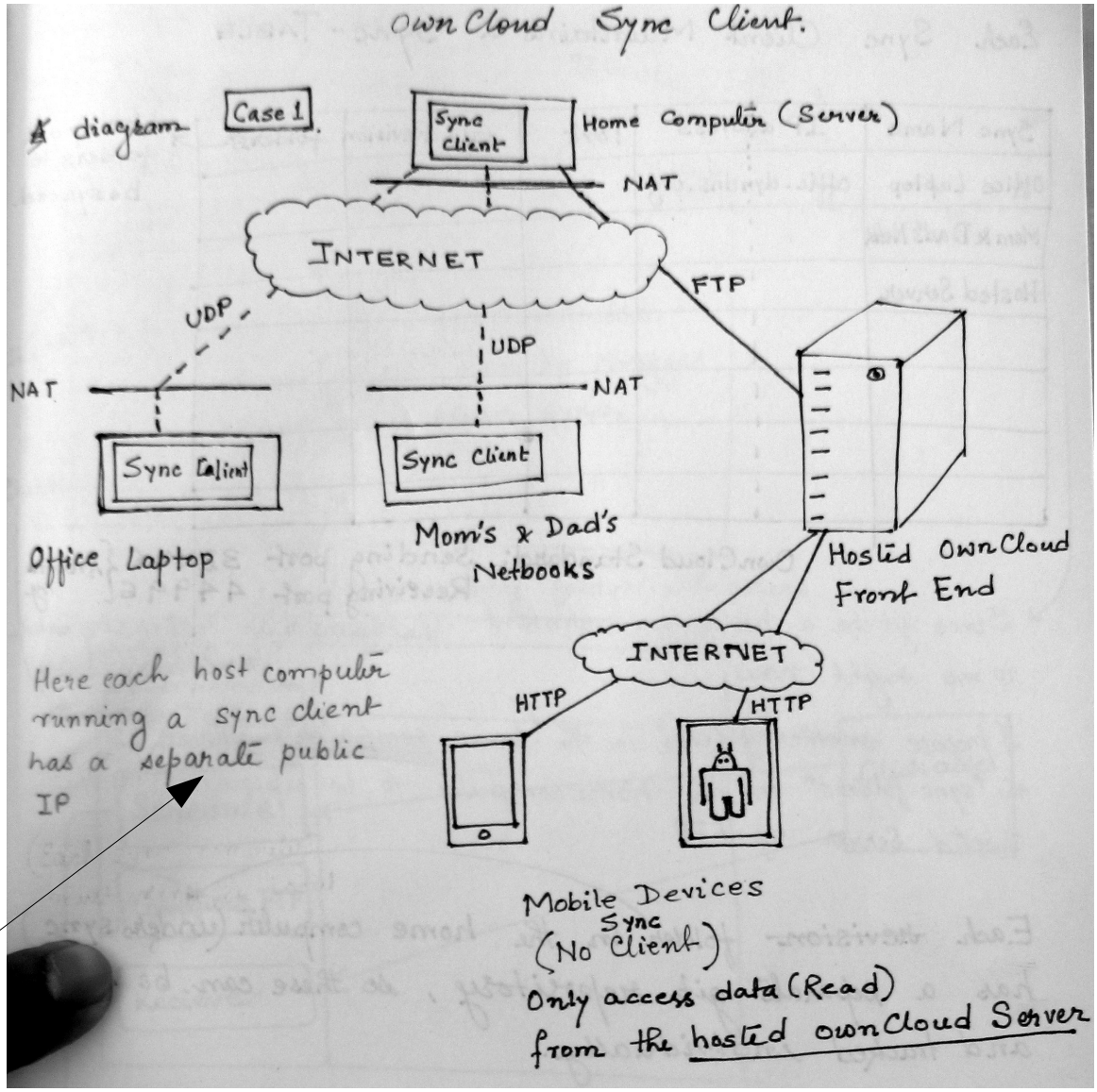
OwnCloud Sync Client

Diagram For Case 1:



Unique to the NAT not the device.

The Case above is the one that the sync client would have to handle. There are two types of devices where a user would want to access his / her data :

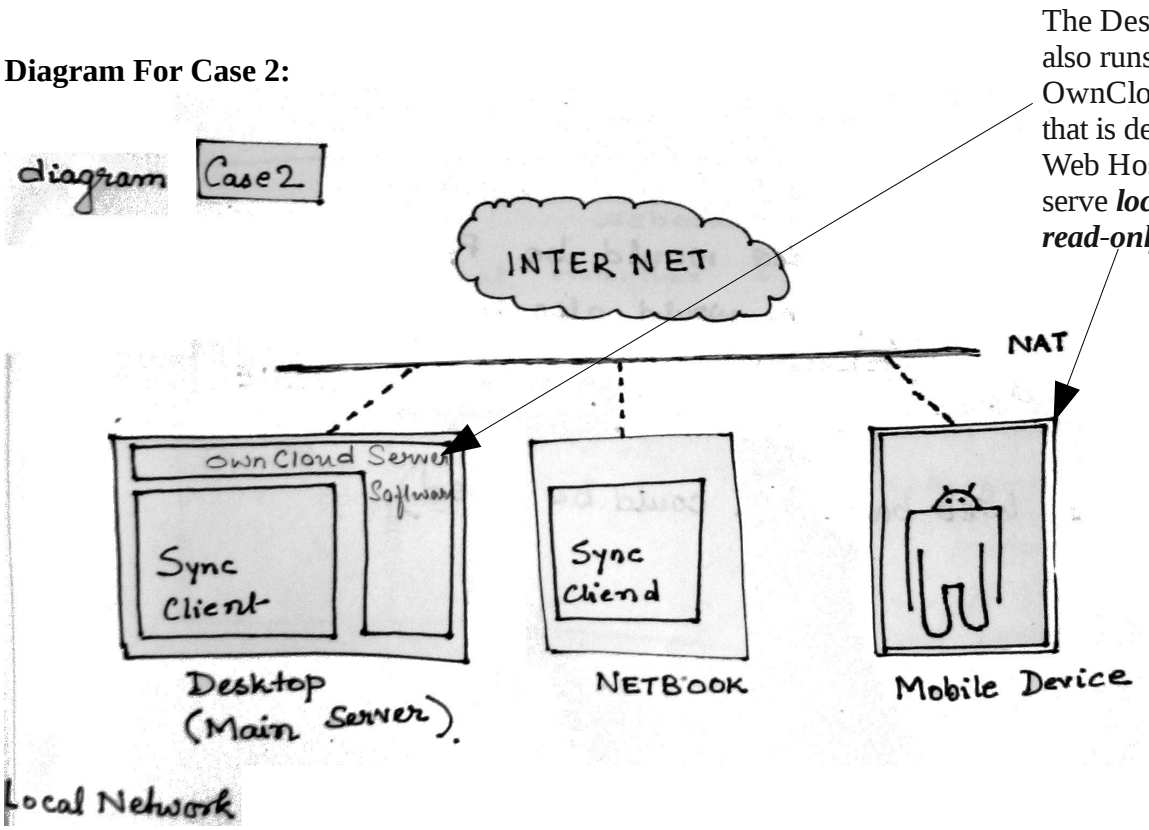
1. A laptop/desktop/netbook where a desktop sync-Client can be run.
2. Mobile devices (mostly with limitted storage, where the user would only want to access the data in his cloud)

In this proposal, i propose a sync – client which allows a user to

1. Sync folders between different computers (hidden behind a NAT or Publically visible) spread across the internet.
2. Sync the main folder (at Home) with a Hosted Server from which the user can only access his data (mobile device use case)

(The protocols used between any two device is indicated in the image above)
Since git in a hosted environment is somewhat impercieveable to me right now , I propose the sync folders in the home computer to be uploaded to the "Hosted server"(see figure above) by FTP.

Diagram For Case 2:



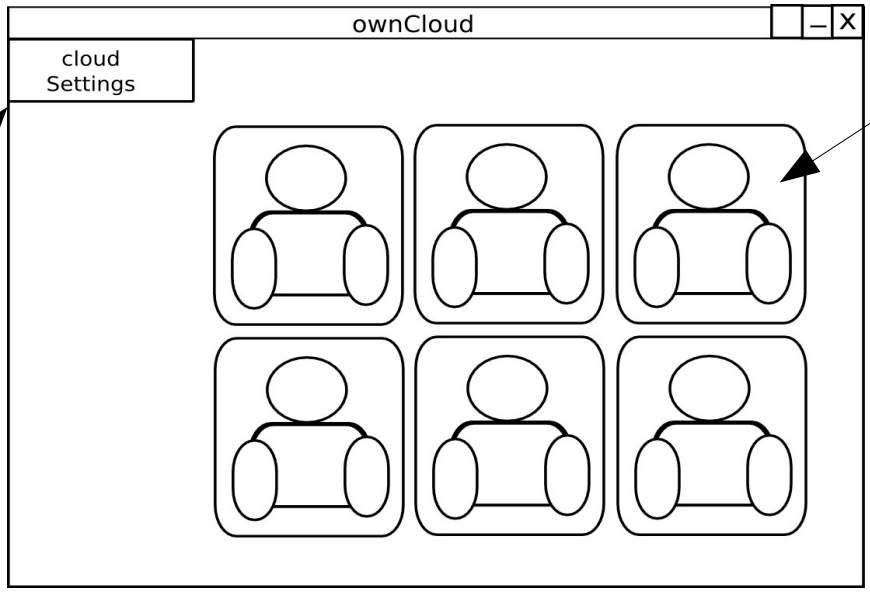
The Desktop machine also runs the same OwnCloud Web Server that is deployed in the Web Hosting Server. To serve ***locally available read-only devices.***

In the Local Network, each device has a common Public IP and a unique Local IP (Sync Clients in the same Local network must support auto discovery)

The Diagram above illustrates the case, where all the devices accessing the cloud are all behind the same NAT, a typical home scenario. In such a case, any client must identify all other clients that are behind the same NAT and use the local IP to sync with them. ***A point to be noted here is that Read - Only devices (mobile devices) have no way (IMHO) of knowing that the Main Server is also behind the same NAT. User of the mobile device needs to enter the local address in the browser of the mobile device to access the locally available server.***

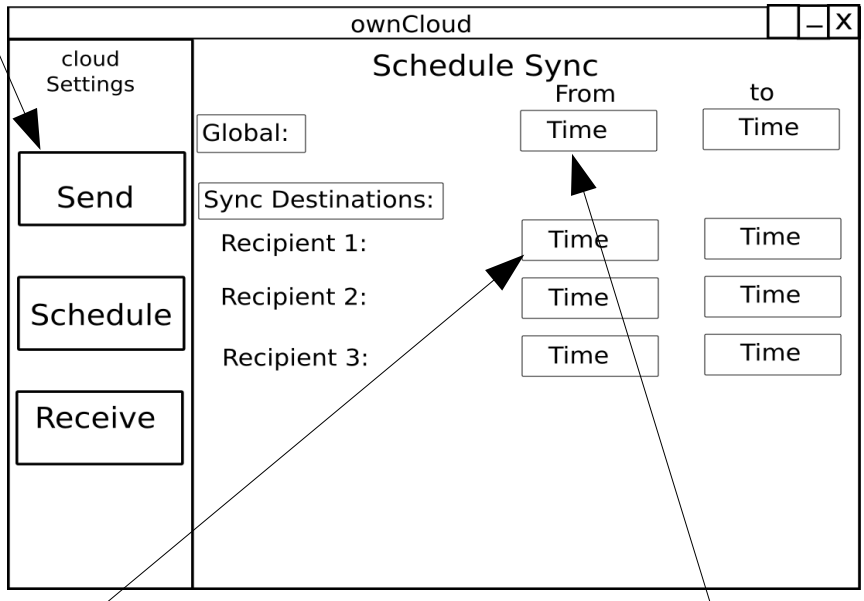
User Interface:
The UI Library of choice would be PyQt / PyKDE , but others like PyGTK / EasyGUI / Tkinter (Python's UI library bundled with the Standard Library) would also be possible candidates.
This is where portability becomes an issue.

The following are mock-ups of the User Interface:



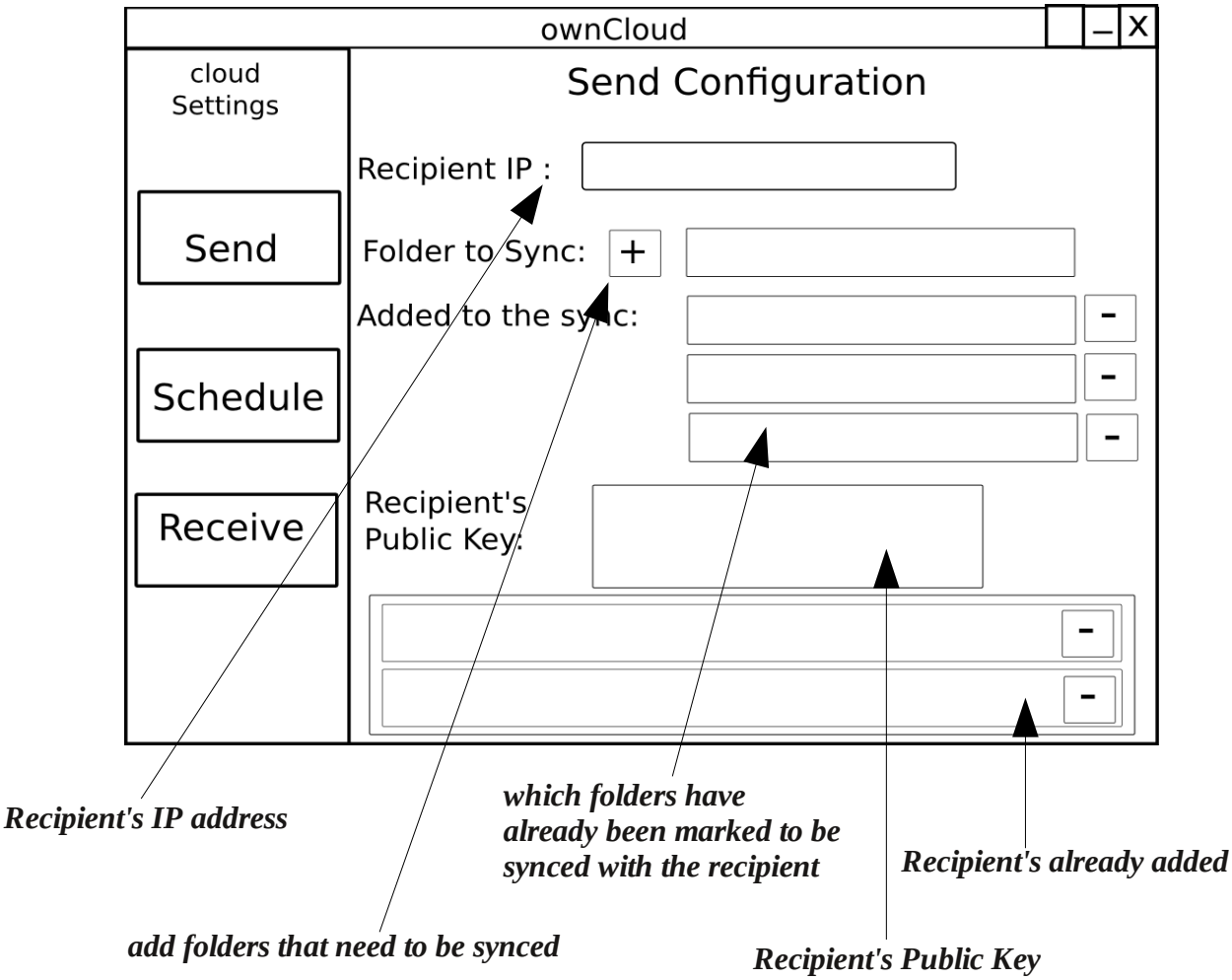
The „cloud settings“ is a button which ***expands to another widget*** shown in next mockup

Each icon is clickable and on clicking opens up the hidden folder which contains the files in a file browser. There is one icon for every sender from whom ther user is receiving files (the folders are synced between the two users). *Also (not shown here) there is an icon for all the personal folders that the user has under sync.*



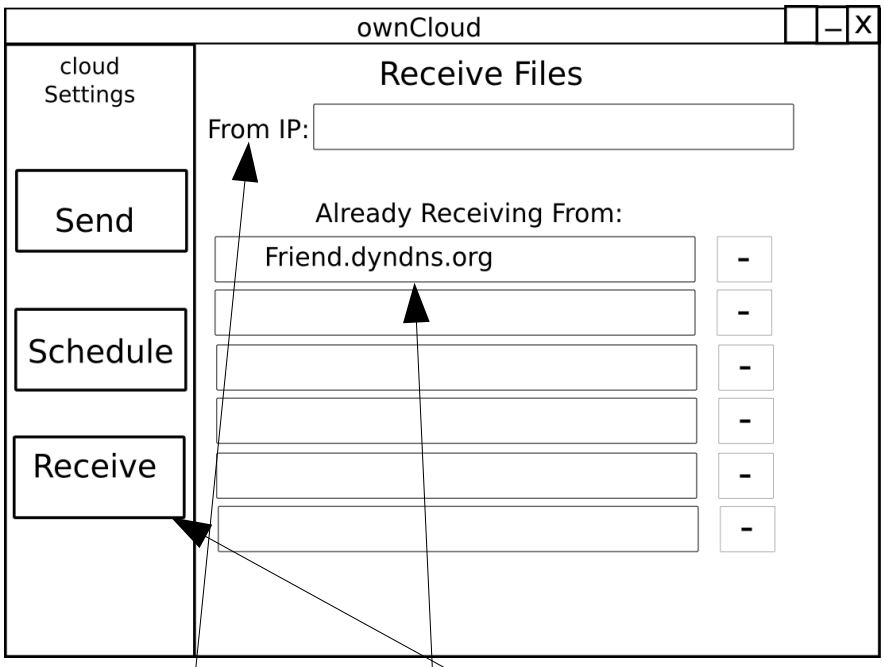
The window above opens up when the "Schedule" button is clicked. A recipient entry is added automatically when a new recipient is added.
If the ***individual recipient sync time*** is provided then it overrides the global settings and if the "From" and "To" entries of a recipient are left blank then they follow the ***Global sync timing.***

When the user wants to share a folder with another user, he clicks on the "Send" button under the "cloud settings" and the following view shows up.



The user needs to specify the ***recipient's IP address (or dyndns.org address)***. He/she also needs to ***add folders that need to be synced***. The UI also shows ***which folders have already been marked to be synced with the recipient***. For added security each sender encrypts the packets with the ***recipient's public key***(If the pulic key is not mentioned, then the communication is unencrypted).

In addition to the above, ***recipients already added*** are shown in a list view for edition and deletion.



When the user wants to recieve a folder from another user, he must add the sender's IP / dyndns.org address to the ***"From IP"*** available under the ***"Receive" button***.

The recipient (user) can also delete ***already configured senders***

Note:

1. Right Now individual files cannot be shared. Whole folders need to be shared.
2. The backend is written in pure Python to ensure portability across operating systems.
3. To ensure each folder can be shared the following technique is employed :
   Each folder (Shown Below) is under a git repository. F1, F2,F3 all have independant git repositories with gitignore set to .git . This is done so that individual folders can be shared and synced separately.

   F1 ----
   \
   |___F2
   |
   |___F3

4. In case the client is terminated, when it restarts, it checks for uncommitted changes in the sync repositories. Each modification / Addition / Deletion is comitted separately so that each commit reflects changes to only one file. Helpful while sending changes to-and-fro.

   The need to separately commit each file mentioned in point 4 is explained below:
   If a user wants to revert back to an ***earlier revision of a file***, reverting back modifies ***only that file*** and not the others. But if more than one files are comitted together then ***reverting back*** to that revision ***changes other files in that commit as well, this is not desirable.***

5. Git is only used as a revision control mechanism , git pull / push are not used to sync files. "git bundle" is done on the folder under sync, by the sender between the sender's master and the destination's revision (available with the sender in the sync-Table). This diff is then transmitted.

Each Sync Client Maintains a Sync-Table

Sync Name	**Public IP**	**Local IP**	**Port** (on which to send data)	**GIT Revision**	**Folders to Sync**	**Public Key**
Office Laptop	Office.dyndns.org	192.168.1.45	44996	12333	/home/user/snc	.....
Mom and Dad Netbook	Home.dyndns.org	171.255.1.1	44996	33442	/data/pics,/home/user/music	.....
Secure PC	Secure.dyndns.org	171.255.1.6	44423	12333	/home	.....
Hosted Server	Ftp.hosting.com				/home	

OwnCloud Standard Sync Ports (Arbitrary):
Sending: 33996 Receiving: 44996
(Can be changed for added security. But if non standard ports are used the sender and receiver must be aware of it.)

Adding Modified or New files to git:

1. Periodically , the sync client,iterates over all Folders (under sync) looking for folders which have .git as a sub-folder.
2. In each folder that has a .git sub-folder , git status shows,if any changes exist.
3. We then iterate over each changed file , adding and commiting them till no changed file exists.

Implementation Details:

The application is implemented in 3 parts.

1. The user interface (following the above mock-ups).
2. The part which
   - monitors the git repos , and periodically creates git bundles to be transmitted to and fro between the sender and receiver.
   - Applies git bundles when new ones are available.

3. The backend which handles sending and receiving git bundles. (***We can use TCP or UDP for the same, and i have experimented with sending files between two computers behind separate NATs using either protocol and it works. I would need suggestions as to which protocol to choose***).

   - UDP (All clients constantly send UDP packets to all entries in the sync-Table to maintain open connections.
     - At scheduled time, suppose person A has shared a folder with person B. Client A first requests B to send the bundle (created using "git bundle"). Resembles **A --"git pull"-->B**

        - A on receiving the bundle pulls the changes to his repo (folder under sync). Then creates a bundle for B , and requests Client B to receive. B then pulls the changes from the bundle sent by A.
          Resembles A--"git push"-->B
          But implemented as **A<--"git pull"--B**
        - This is repeated between every two client.
    - TCP (This is significantly difficult because both the sender and receiver clients need to initiate the connection at the same time . Hence the difficulty in implementing the same in a distributed environment). Using TCP is better as compared to UDP because *some* ISPs shape UDP traffic but not TCP hence packetloss would be minimal). Connection initiation over TCP can be implemented as follows:
        - The initial step till creating the connection uses UDP to ensure both ends are online. The clients then simultaneously try to establish a TCP connection with each other, which initially fails but is established in a few tries.If the clients do not connect within a TimeOut period , a new client pair is chosen.
        - Once they are connected then file transfer is trivial.

    NOTE: Hole Punching can only be achieved between a fixed pair of host:port and destination:port. So sending sockets need to be bound to a port, this prevents us from spawning multiple sender processes to connect to all the recipients at the same time.This makes the transfer slow in case of TCP as transfers would have to be made one client at a time.

    So i propose UDP as the transport layer protocol for communications between clients.

Sending and Receiving UDP data:

1. Each receiving client maintains a constant open "hole punch" by periodically sending UDP packets to the sender's host and port.
2. Each Client maintains a process constantly listening for incoming packets.

    Packet Structure:
    Sender's address : Packet number : Total Packets : Payload (Fixed Size) : Checksum
    (It would be great if someone could suggest some module to send and recieve data reliably over UDP so that this packet creation / error control need not be done manually)

3. Standard error control mechanisms are applied to prevent data corruption , like attaching an MD5 hash to the packet