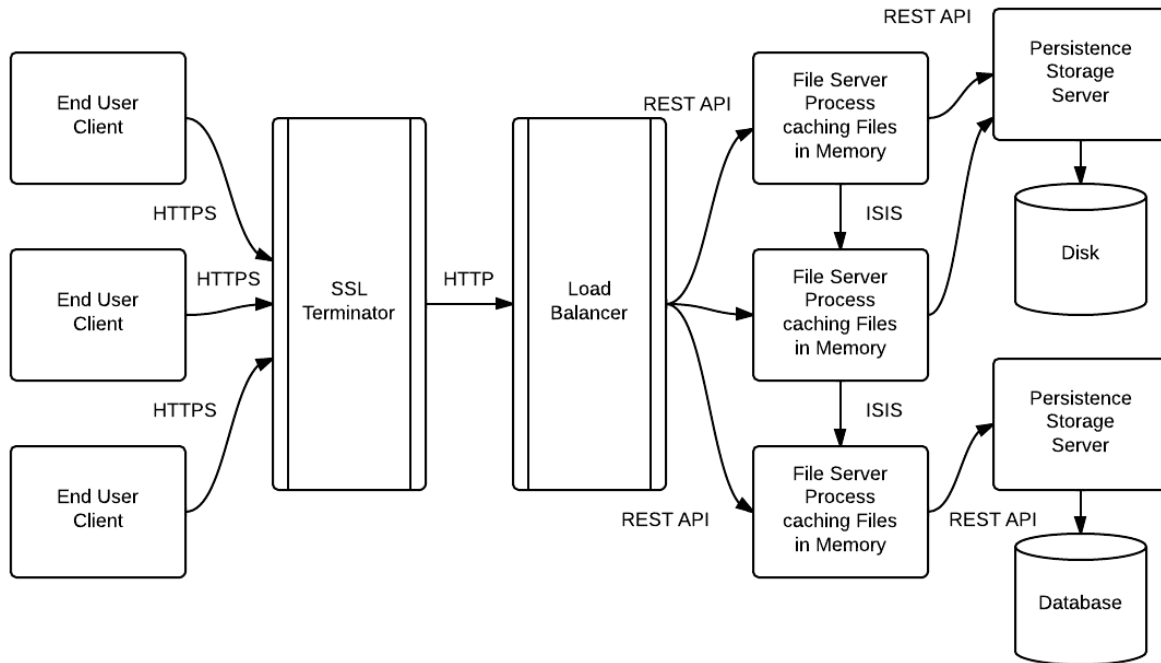


Cloud Based Distributed File Server

- **Architecture**



The main idea of the architecture is to map users to file server groups and thus provide horizontal sharding for scalability. Group of Users always go to the same file server group and the information about the user is shared across the file servers in the group.

We follow a three tiered approach, in the first layer we have the End User Client which access the relevant API. On the Second level we have File Server Process which basically act as in memory caches and hold the latest files in memory. On the Third level we have the Persistence storage to which the data is finally written.

The most recently uploaded and downloaded files are stored in memory of the file servers and are present in each file server of the group. So when the user issues a fetch request for the file its already readily available in memory and reduces the disk I/O's.

The File Server exposes RESTFUL API's for the various services it provides through HTTP. The security is mainly provided by the HTTPS connections. Given the finite time we have not added extra levels of security as would be present in traditional cloud based storage solutions.

The File Server Processes are just service processes and do not host any web pages as such as would be done traditionally. Instead the task of designing UI and displaying the data in the

required form is left to the Desktop application built for the user or a Web Server which ultimately connects to the File Server Processes through the RESTFUL API. This would provide us with a flexible robust architecture where the role of the file server's would only involve serving out the content.

Typical Interaction Flow:

1. User Accesses are typically secured through the use of HTTPS. The end user client makes requests through the RESTFUL API's over HTTPS for each and every file operations.
2. The SSL Terminator provided by the NGINX then converts the HTTPS to HTTP.
3. The SSL Terminator then provides the unencrypted HTTP Content to the Load Balancer, which maintains the file group for a user.
4. The Load Balancer routes the traffic to the required file server processes.
5. The File Server process interacts with the Persistent Storage Server to serve the required files.
6. The Files are then encrypted again at the NGINX component and then returned back to the user.

No	Component	Component	Interface	Function
1	End User Client	SSL Terminator	RESTFUL API over encrypted HTTPS	Provide a Secure Connection for Data Transfer
2	SSL Terminator	Load Balancer	Unencrypted HTTP	Terminate SSL and server the Request to the Load Balancer
3	Load Balancer	File Server	RESTFUL API Unencrypted over HTTP	Distribute Load over the Load Balancer
4	File Server	File Server	ISIS	Communication between File Servers to maintain consistency and replication.

File Management:

Representational state transfer (REST) is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. The File Servers provide a RESTFUL API to provide various services to the User. The various services provided by the the File Server we have implemented include:

No	Method	RESTFUL API	Parameters	Output
1	POST	/adduser/{clientId}/{password}	{clientId} = User Name {Password} = Password of the user	Add a user if the user does not Exist or throw a User Exists exception if already exists
2	GET	/getUserFileSystemInfo/{clientId}/{password}	{clientId} = User Name {Password} = Password of the user	List all files on the Users File System
3	GET	/file/{clientId}/{password}/{filename}/{fileowner}	{clientId} = User Name {Password} = Password of the user {filename} = Name of the File or Directory {fileowner} = Owner of the file	Downloading of the File if exists or Exception otherwise
4	POST	/deletefile/{clientId}/{password}/{filepath}	{clientId} = User Name {Password} = Password of the user {filepath} = Path of the file to be deleted	Deletion of the File if exists or a Exception is thrown otherwise
5	POST	/shareFile/{clientId}/{password}/{filename}/{sharedWithUser}	{clientId} = User Name {Password} = Password of the user	File Shared if Successful or Exception

			{filename} = Name of the File or Directory {sharedWithUser} = User with whom the file is to be share	is thrown otherwise
6	POST	/unShareFile/{clientId}/{password}/{filename}/{sharedWithUser}	{clientId} = User Name {Password} = Password of the user {filename} = Name of the File or Directory {sharedWithUser} = User with whom the file is to be share	File unshared if Successful

Versioning:

We are conceptually using logical clocks¹ to support versioning in files and associated meta-data. These version numbers are used to resolve conflicts between operations that happen in this distributed environment. Every file has an version number which is stored as part of the meta data of the file. Here are some of the use cases where we are using version numbers -

a) Suppose a client has downloaded a file x and has a version v1 (which is the latest) cached locally on its client. If this client is logged in some other machine and updates the file on the server, its version number on the server becomes v+1. Now if the clients wants to update the file, it will try to update the file to version number v+1. This update would be rejected since the server recognizes that the client does not have the latest version of the file with it. The client would have to download the latest version of the file and then upload it again.

b) Suppose a server crashes and starts again. It will bootstrap by syncing its in-memory data structures using ISIS replication. It would also be simultaneously applying new updates in order to keep itself up to date. In this case we will use versioning to resolve conflicts.

c) We also use versioning to resolve conflict that arise when checkpointing which the data with the higher version always getting persisted.

In addition to this, add the user metadata like password is also versioned to resolve similar conflicts. The client can refresh its cached files which would get the new versioned file on its machine.

¹ We thank Theodoros Gkountouvas for helping us come up with this.

CheckPointing to Persistent Store

We have a reliable persistent storage which is used for checkpointing all the in memory data stored by the file servers. This persistent storage acts as a reliable store for it which can also be used for bootstrapping the servers in case the systems shuts off. We have a restful service running on these hosts which can interact with the file server. We have not implemented replication at this level since there are many solutions available which can replicate data written on disk (since it is not changed very frequently). A simple example of it would be simple master-slave architecture where everything written to the master is also replicated to the slave server. We are using the traditional unix file system to store our files on disk. A better option would be a distributed and more scalable file system like HDFS.

These are the API's that the persistent restful service exposes -

- 1) BootStrap Server - Whenever a host comes up, it calls this API to bootstrap itself with all the user and their file metadata information. It doesn't load the file content since it called be called by calling the FetchFile API when the request for that file is made. All this metadata is reasonable small and hence this transfer is relatively fast
- 2) FetchFile - File server calls this API when it gets request for a file which hasn't been loaded in its memory. A simple example of this would be the first time a file is loaded by client. In this case this file wouldn't be present in the file server memory and hence it would be loaded off disk. Another case would be when the in memory file limit is reached and one file it evicted in memory (after flushing to disk). In this case if the another request is made for that file, it will be loaded off the disk.
- 3) FlushFile - File server calls this API when it specifically wants to flush a single file to the disk. This will be happen in the case described above - when the file server wants to evict a file from its memory because of memory constraints, it will flush the file to disk
- 4) DoCheckPoint - This API is periodically called to make a checkpoint all the in memory state to disk. This will ensure protection against some catastrophe event which brings down all the file server. This would be an relatively expensive operation and hence this will be done rather infrequently.

Whenever we make a new checkpoint, we merge the existing in memory data with the data present in the previous snapshot and store the data into new folder based on time and date. This ensures that we can always load from previous checkpoints in case some catastrophic event occurs. We have made our folder structure on the disk indexed by date to achieve this. We will periodically delete old snapshots to save space.

Replication:

We want all the file servers in the group to have consistent data. This means all the write events that occur in the File Server Group should be ordered. All file servers should see the same sequence of operations in the same order.

We achieve this by using ISIS as the framework for communication between the file servers. All the file servers in a particular groups are mapped to groups in ISIS. When a File Server comes up it knows which group it belongs to. It tries to connect to the Particular group and registers itself with the ORACLE Server for that particular group. Through this way each node present in the group knows about the topology of the other nodes in the file server group.

All events are replicated across all the groups of file servers using the ordered send call of ISIS. This way all servers in the group sees the same set of events in exactly the same way.

Each File server in the group maintains information about the user which is loaded dynamically when the user first logs in. The File server used to serve the request for a given user is chosen randomly by the Load Balancer but because data is replicated in the same way across all file servers in the group, the order in which the server is chosen does not matter.

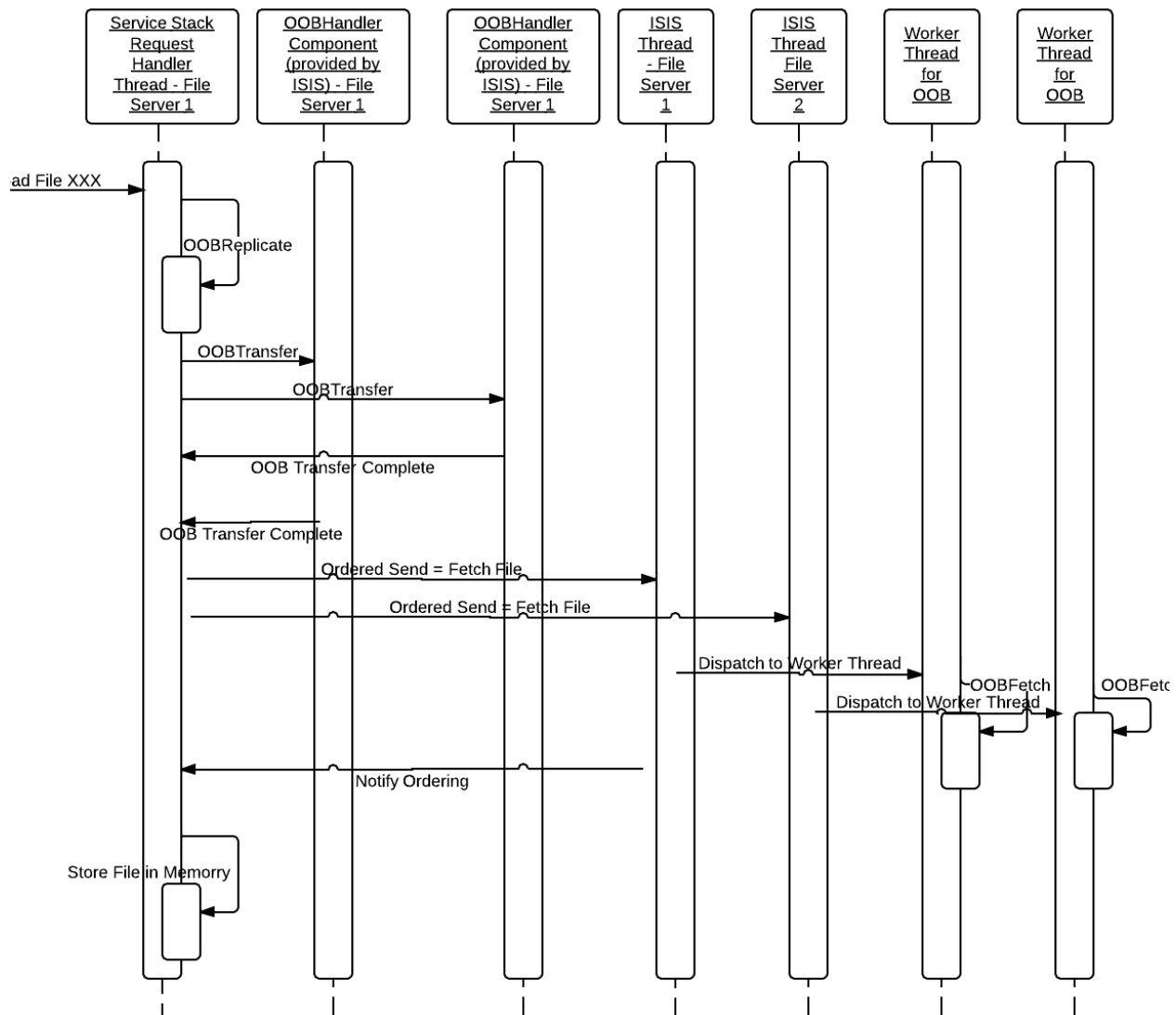
Write Events in the system:

When the amount of data is small, we use the orderedSend call of the Group to exchange information between the nodes. When the data to be transferred is large we use the Out of Band Data Transfer to exchange data between the nodes. This approach makes sure that the ISIS thread servicing the requests from other nodes is not blocked through large data transfers. Below are the events in which the data changes in the system and the approach used to transfer data across nodes.

Number	Event	What Changes ?	Implementation of ISIS Used
1	Adding a New User to the System	Meta Data Change	OrderedSend
2	Downloading a File from the Storage System	Actual file being loaded in Memory	Out of Band Data Transfer using OOBReplicate and OOBFetch
3	Uploading a File to the Storage System	Actually file being loaded in memory	Out of Band Data Transfer using OOBReplicate and OOBFetch

4	Reuploading a already existing file in the Storage System	Actually file being loaded in memory	Out of Band Data Transfer using OOBReplicate and OOBFetch
5	Listing of Files of the User	All the Meta Data of the Files are being Displayed	OrderedSend
6	Changing of the file metadata through sharing of files to another user.	The Meta Data of the file is changed	OrderedSend
7	Changing of the file metadata through un sharing of files to another user.	The Meta Data of the file is changed	OrderedSend
8	Deleting a file from the system	The file will be marked for deletion and will be flushed during the checkpointing. So there is only a meta data change	OrderedSend

The Process of out of band transfer of data is indicated below in the sequence diagram



BootStrapping:

Eventually node crashes or a new node joins the group. In both of these cases the nodes need to come to date with the existing information at other nodes. This process is called bootstrapping wherein when the node comes up it gets the information about the state through other nodes.

When a node comes up, it picks a random node from the available members in the group and sends a bootstrapping request to the picked node. Once a node is picked, bootstrapping consists of a sequence of events. The picked node sends a **BootStrappingBegin** message to the node seeking information indicating that it is now ready to transfer information. Then it sends a sequence of **BootStrappingContinue** messages indicating that there is more data to come. The actual data is transferred using the out of band transfer mechanism similar to the one

explained before. Eventually the Picked node sends a **BootStrappingEnd** message to indicate that all data has been transferred. At this point the bootstrapping process is complete.

But during the bootstrapping process, the ISIS thread continues getting updates from the other parts of the sequences of file operations. As a result the delta of these operations and the original information transferred during the bootstrapping process needs to be ordered.

To solve this problem we note all the bootstrapping data received and store them in a temporary in memory file system, and we store the deltas received during this period in another temporary file system. Each file object has a logical clock associated with it which indicates the ordering of two events on the same file object. We always end up taking the highest event of the file object and discard all older update, additions and deletions to the file objects.

LRU Session Policy:

The amount of space per user in the memory is fixed by an upper bound of 50 MB. When the limit is reached we begin removing the least recently used entry from the file system for the user. In this way we reduce the number of bytes per user in the in memory file system. We also do not cache files which are more than a specified limit since it would consume a lot of resources and fetching the file from the disk would be faster option.

End User Client:

The main function of the client is, to utilize the functionalities exposed by the back end, that allows a user Store, update share and access files from any location. The client is built as a visual C# windows forms application. The different forms establish communication with back end and controls and manages the functionalities and data provided by the back end.

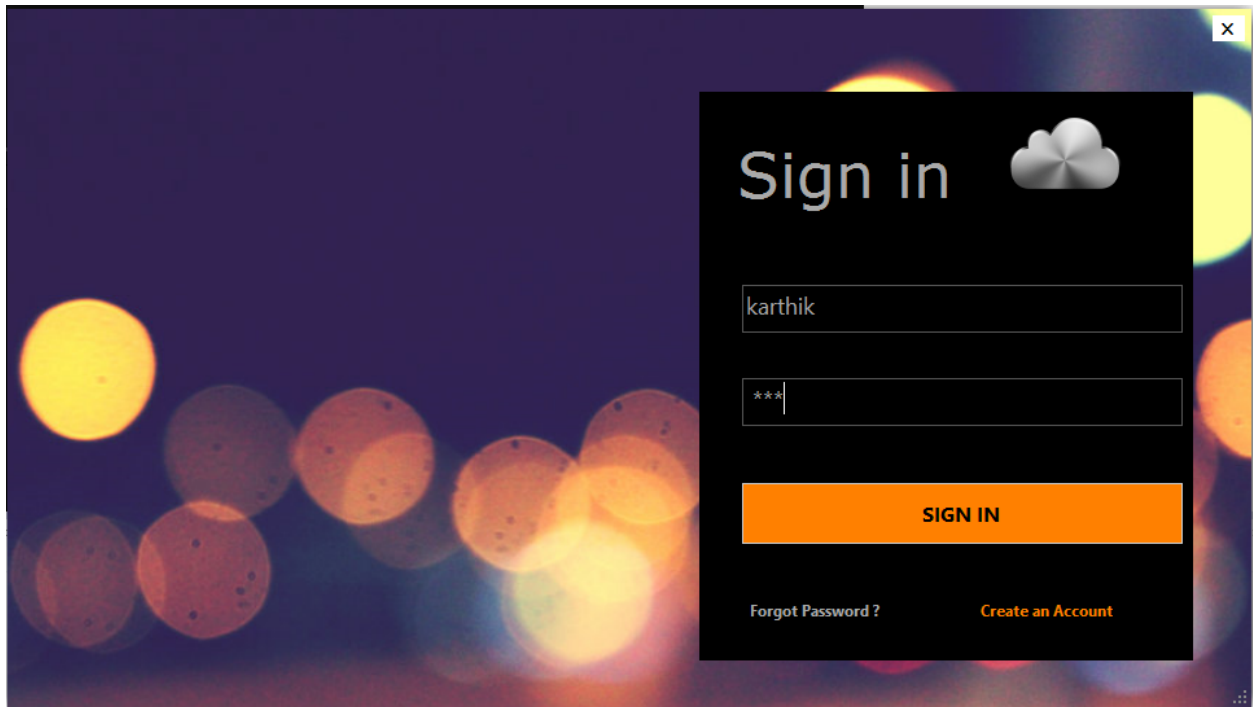
The Clients uses the service stack C# client(JsonServiceClient) to send requests to the RESTFUL API's exposed by the File server.

Client RESTFUL requests -

- 1) **Add User:** POST Request - `"/adduser/{ClientID}/{Password}"`
- 2) **Upload/Update file Request:** POST Request - `"/updatefile/{ClientID}/{Password}"`
- 3) **GetFile List Request:** GET Request - `"/getUserFileSystemInfo/{ClientID}/{Password}"`
- 4) **Download File Request:** GET Request - `"/file/{ClientID}/{Password}/{FileName}/{FileOwner}"`
- 5) **Delete File Request:** POST Request - `"/deletefile/{ClientID}/{Password}/{FileName}"`
- 6) **Share File Request:** POST Request - `"/shareFile/{ClientID}/{Password}/{FileName}/{FileOwner}"`
- 7) **UnShare File Request:** POST Request - `"/unShareFile/{ClientID}/{Password}/{FileName}/{FileOwner}"`

The windows forms UI is developed using MetroFramework's windows Modern UI for .Net WinForms Applications.

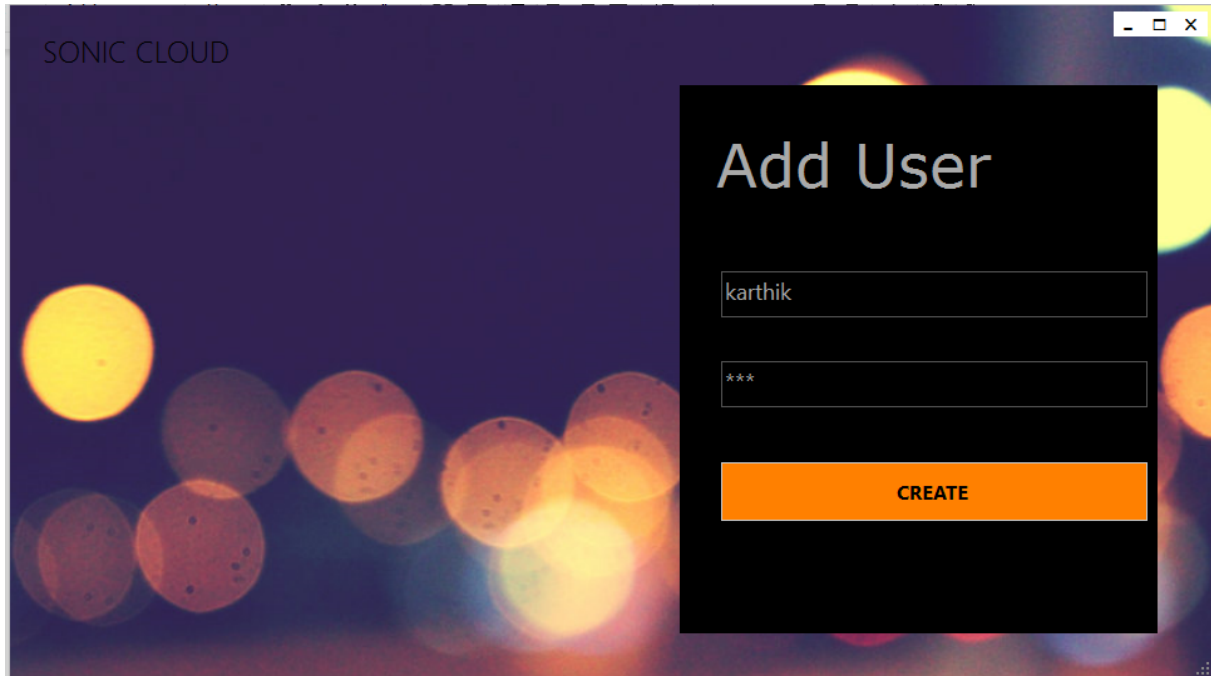
Client forms and operations - Sign Form:



Sign In: the sign in button takes the user-id and password for the respective fields and creates the FUnctionScreen windows form. A new user can use the create new user link at the botton to create a new user account.

New User Form:

Piyush Maheshwari(pm489), Laxman
Prabhu (lp382), Karthikeya Pandit (kkp37)



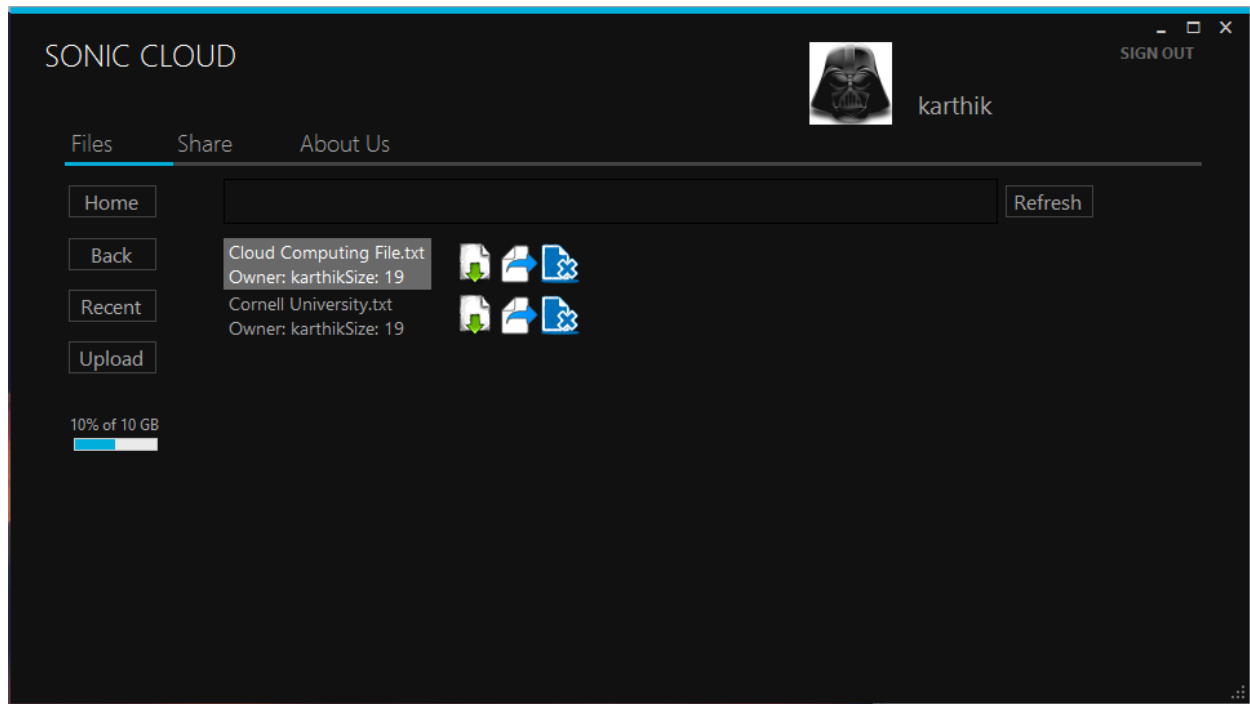
Create : This ensures the new user ID and Passwords are filled and on CREATE button click make a POST request to the file server to add a new user. The RESTFUL POST request for add user is as mentioned earlier.

FunctionScreen Form:

The Function screen contains all the main operation and information of the client. The layout of the form consists of a static user picture, static User ID display and the SIGN OUT button used to sign out and quit application. All the operations and data/information is maintained across 3 tabs. The data is managed among the UserFileSystemMetaData, ClientFileSystem objects and is used to make the required Restful API requests. The Details are as described below:

Files tab:

Piyush Maheshwari(pm489), Laxman
Prabhu (lp382), Karthikeya Pandit (kkp37)



The Controls and the layout is described below:

Home Button: Brings the user back to root directory and displays files in the root directory.

Back Button: Brings the user one step back in the directory structure and displays the files in that directory

Recent Button: Displays a list of recently visited directories by the user

Upload Button: This button is used to upload file to the File System. The application shows the open file dialog that the user can use to file to be uploaded. A UserFile object is created for this file and is Uploaded using the Upload/Update Restful POST API Request. The request is as described earlier. When a file is chosen for upload it is checked if the file exists in the user file system already. If it exists the file is checked for version number conflicts. This ensures that the user does not overwrite existing version with a stale version.

Usage Progress Bar: Shows the storage space used by the user.

Refresh Button: The refresh button sends a POST Restful request to update the UserFileSystemMetaData object and uses this to update the file list and the lists in the Shared tab. The Get File List Request format is as described earlier.

Directory Field: The directory field displays the directory the user is currently browsing. It is also used to indicate the directory into which a file has to be uploaded.

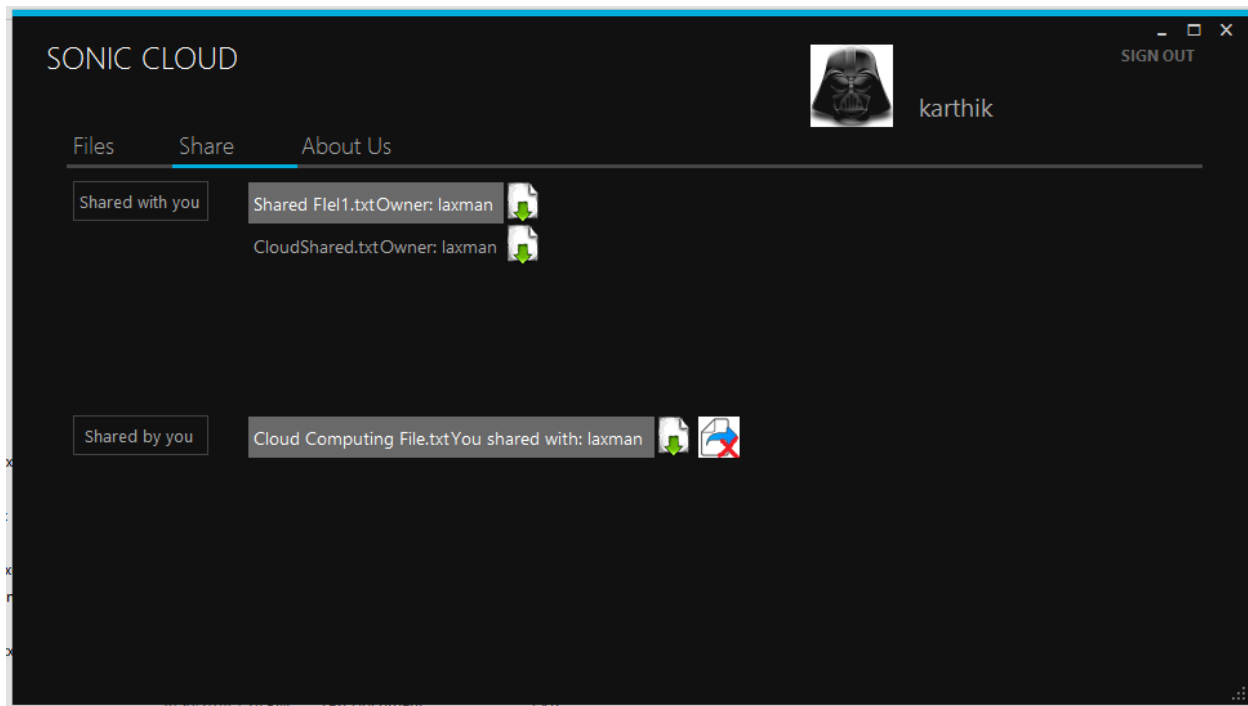
File List: The list of files owned by the user is displayed here. It is populated when the FunctionScreen is launched and is continuously updated to maintain the most updated List of Files. This is achieved by updating UserFileSystemMetaData object using the Get File List request. Each File Listed here allow the user to download, share or delete the file.

Download: Allows the user to download the file to the local file system. It is achieved by using the Download File - GET Restful API request. The file is also saved into the ClientFileSystem object that allows local cacheing of the file. When a download request is made it checks the local cache of the user to find the file. Only if it does not exist locally it is requested from the File Server.

Share: Allows the user to share the file with other existing users. The application requests the client id the user to share the file with. The share file request POST restful request is used to achieve this.

Delete: Allows the user to delete a file from the file server. The delete file request is used to do this.

Share tab:

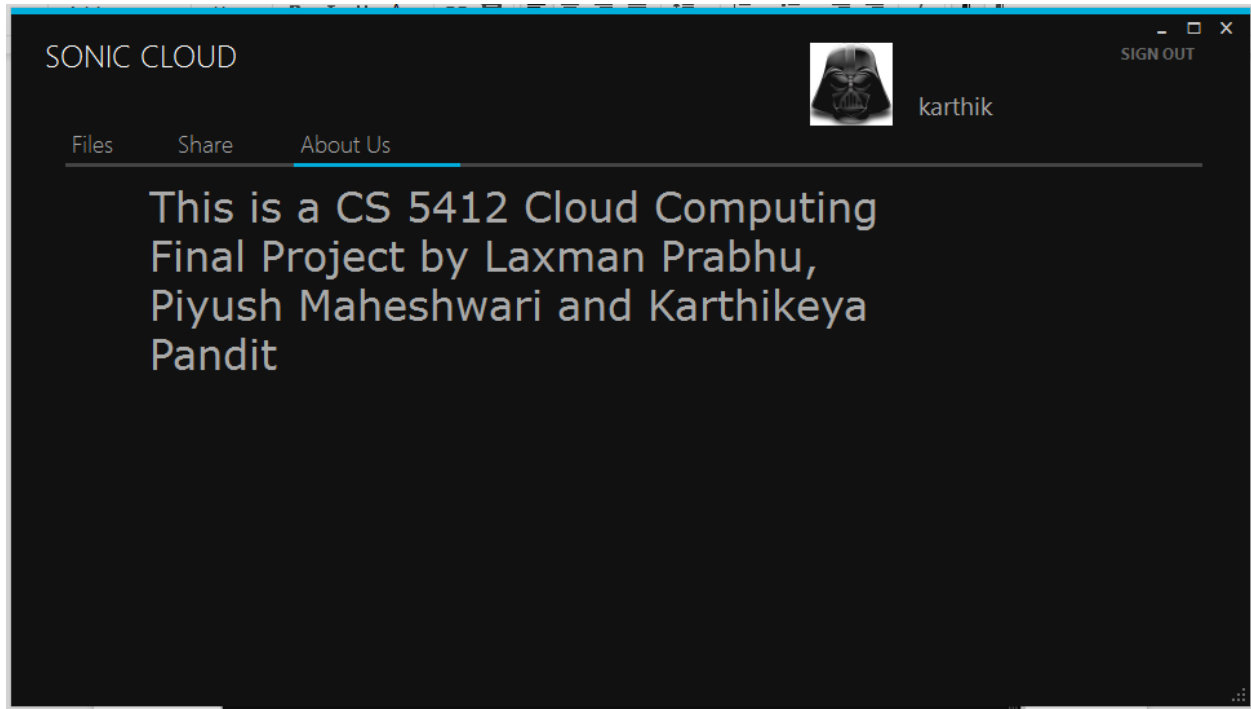


This tab displays two lists: List of Files shared with the user (The Shared with you) and list of files shared by user (Shared by you). The Lists are managed by windows form's DataGridView tool. The file listed as the Shared by you allows the user to download the file to local drive and Unshare the file. The file listed s Shared with you allows the user to only download. Since these are the files that are shared by others with you it cannot unshared by this user. These functions

Piyush Maheshwari(pm489), Laxman
Prabhu (lp382), Karthikeya Pandit (kkp37)

are handled as described in the Files tab section above. The User can update/refresh these lists using the 'Shared With you' and 'Shared by you' buttons

About Us tab:



This tab has static information about the course project and the members of the project team.

Piyush Maheshwari(pm489), Laxman Prabhu (lp382), Karthikeya Pandit (kkp37)