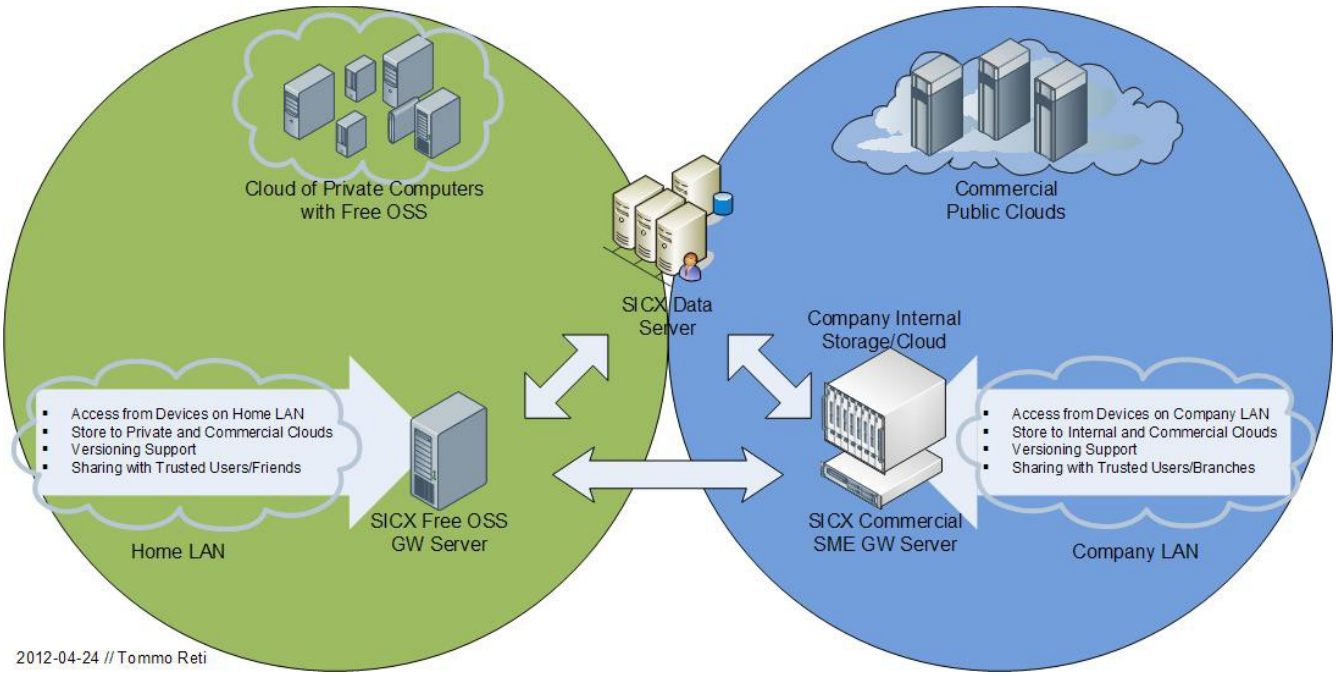
SICX Free Open Source Server/System

The free, consumer- aimed, management system for shared storage.

Introduction, functional overview

The SICX OSS Servers (or peers) will create a shared storage space among trusted friends. An application example would be a folder on your computer where all the saved files are immediately available on all other authorized computers and devices. If the described is the WAN end, the LAN end also creates a shared storage space where the OSS server is the hub and the gateway to the WAN. In other words, as in the commercial server, the SICX server offers a shared volume to the LAN that is accessible by the users and devices on the LAN in order to share files.



The first application for the SICX OSS will be the shared folders as described above. This described solution does not include the Vaadin UI, but comes only as a relatively ordinary- looking filesystem folder. The different SLAs or storing profiles are selected by dropping the files to different folder in the shared volume. Different folders represent different groups of friends/users, different protection levels (e.g., company confidential, public marketing material, only family, only me, etc.), different encryption levels, different remote backup locations, etc. Some folders may even translate files to different codecs (MP4,..) or formats (PDF,..), e.g., for certain customer or platform distribution.

The role of the "SICX Data Server" is open (see, the pic). It can and will be used in different ways for both type of servers. For example, it can be an index for the P2P and HIP networks in the beginning. Later, we can rebuild and get rid of the bottleneck. At this time, it is more important to get the SICX OSS System up and running. For the future motivation, this P2P Network could be a starting point for Green Credits and Consumer Grid type services in addition to the shared storage. It is worth noticing that this should not be another anonymous free file sharing darknet, so in every design decision we try guiding ourselves out of that.

Future applications could be, as already suggested, different types of data management folders (so called “magic folder”) where the content type is understood and acted upon. The system could also be tied to non-file based use-cases; calendar synchronization, messaging and other activities suitable for peer-to-peer networking. As for possible future developments on the architectural side, the system could act as a gateway for all cloud- or other offsite storage.

Relation to the SICX (commercial) system

The initial version of the FOSS system will be oriented a bit differently than the SICX system, although there are some similarities. Both will act as a gateway for storing data, but the back end is different.

In future development we could see these two interoperate in different ways; the FOSS system creating a storage platform to be offered through SICX, and as SICX being used for accessing additional storage by the FOSS. Individual users could use the system to offer their storage resources for sale, or trade, through SICX. As the shared folders in the FOSS system is maintained by multiple hosts, data storage can be distributed and optimized (everybody does not need to maintain a copy of the actual data of a file). This could even lead to that any storage leased would go noticeable in the file system's accounting, as the space is taken from what has been freed through the distributed maintenance of the user's own shared folders.

The space offered through SICX could be traded into similar offsite storage that the user can use that the FOSS system would transparently move data to. In this way, the FOSS gateway acts as a client for the SICX broker, using 'credits' got from offering its own storage for use. We could also plug in dropbox or similar services which would be used for offsite backups, making it a sort of 'mini' version of the SICX server. It would not have the SLA and other enterprise- relevant features, but could act in a similar manner as a gateway for storing data “somewhere accessible”.

System overview

The system consists of a centralized lookup / registration server, local gateways and clients. The centralized lookup server is run by a trusted third party, while the local gateways are run by the users. The clients are devices accessing the shared folders and are usually within the same LAN as the gateway. In practice, for most installations the gateway and its only client will be on the same machine (e.g., home computer), but this separation allows for local sharing within LANs.

SICX Data/Lookup server

The lookup server acts as a directory for finding other users, much like what the Skype servers do in the Skype network. Users register to it, and after that others can find them and connect. In other words, the lookup server maintains a database of each user (name and other metadata), the user's identity (=public key) and network location (used for connection establishment). The server can also be used for aiding in data synchronization by maintaining changelogs for people that are offline and by acting as a data proxy when users are faced with unfavorable networking conditions (firewalls, NATs etc).

There's actually no reason why there would need to be only one global lookup server. Each user / user group could actually run their own. Users just need to be registered to the same one to be able to find each other. But users could also be registered to multiple ones.

SICX Local GW

The local gateway is a daemon, similar to the dropbox client. It manages the shared folders and provides access to the data (primarily) through a network folder (WebDav / CIFS / FTP etc) interface. Much of the following sections actually describe things that happen in the gateway, as it is the engine of the system.

Clients

A client is any device that accesses the shared folder; opens files, delete and creates new ones etc. The clients access these shared folders using the gateway's network folder interface, and have therefore nothing SICX- specific installed. These can be anything from servers to mobile phones.

Identity management

Each user creates an identity when first adopting the system. An identity is a public key pair, of which the public part is used as the public identity. This is used to authenticate everything the user does. In practice each gateway is tied to an identity, as the logic resides there, and it is the component that communicates with the rest of the network.

Each identity has also a public identifier (a human-readable string) which is assigned by the lookup server (basically the lookup server creates a cert tying the identity key to a identifier). If we imagine using multiple lookup servers, the identifier would naturally be lookup-server specific. Let's say that the format of the identifier is email-like, e.g., *alice@lookup.sicx*.

The identifiers aren't actually really necessary, they only ease the initial introduction of users to each other. We could also imagine having no introductionary features in the lookup server (no 'search for user xx'), but instead have digital 'business cards' that people could mail to each other, import into their local gateways and connect.

Update: The identifiers might be skipped for a more subjective, local, naming system. That is, the public keys are the ones that get passed around, and users just name those contacts however they want in their local “address book”.

Data management

Version management and conflict resolution in distributed data management systems is complex. The scheme used In this prototype is a simple one that uses back versions for conflict resolution (instead of merging), and relies on peers maintaining a complete history log.

Shared folder set-up and access configuration

Each shared folder has a reasonably unique identifier (hash of the creator's identity and an UUID), and everything that happens in there (file modifications etc) is packaged into *events* containing a complete description of what happened, signed by the one who did it. The way the folder is shared is by sharing these events between the participants (so they know the directory structure) along with responding to requests for the actual data of the files. Even adding the participants is done using special events. In theory, nobody actually 'owns' the folder; it is up to each one of the participants to decide whose events to take into account or ignore, thus decide who is able to modify the shared folder (as seen by themselves). This means, in theory, that each one can have a different view of the shared folder, but in practice everyone will trust everyone and have a consistent view.

Each participant is required to store the complete event history of the folder. This might be optimized in the future, but it shouldn't be a problem even if not (compare to the changelogs stored by revision control software).

For instance, when creating a shared folder and inviting a friend to share it, what actually happens “under the hood” is that

1. You create a new folder with a new folder ID
2. You create the first event, an “add user” of yourself.
3. You create the second event, an “add user” for your buddy
4. You send over an invitation to a friend to take part in this folder, which includes the folder's ID and its complete event history (which contains the “add user” events).
5. You friend saves this info; the folder id and its event history.

When adding a third or fourth person to the shared folder (by either you or your friend), these addition gets marked as events to the folder, and propagated to all the friends currently sharing the folder. This (the “add user” events) will act as a signal for them to also add those people to their list of people which to sync with. As new users (except the first one) always get added by an existing user, it creates a chain of trust which may be used for different things in the future (such as blocking a set of unwanted people).

Versioning

Each event will have a reasonable unique ID (hash of the one who created it + UUID), a description of what happened and a *parent* event. The events will typically (or perhaps must) contain only a single operation. The visible content of the shared folder will always have a 'head' event; the *current revision* in RCS- terms. Any modification to the folder by the user will result in an event whose parent is that head event (the *head*), and the head will change to the new event.

A simple example:

User A, User B: a newly created shared folder, both's head event ID are null.

User A: event id: 92, action: “add User A”, parent: null. Folder head: 92

User A: event id: 12, action: “add User B”, parent: 92. Folder head: 12

<sync with User B>

User B: Folder head: 12

User A: event id: 23, action: “create file.txt”, parent: 12. Folder head: 23

User A: event id: 54, action: “modify file.txt”, parent: 23. Folder head: 54

<sync with User B>

User B: Folder head: 54 (and sees file.txt in the folder)

User B: event id: 73, action: “rename file.txt => file2.txt”, Folder head: 73

<sync with User A>

User A: Folder head: 73 (and sees file2.txt)

Syncing events:

When syncing events, both will send a list of the most recent events created by each one of the

participants (that is, a list of all the participants in the share, and for each one the most recent event ID). Both check their databases and send over any updates for each one of all the participants.

As these new events may contain “add user”- events, we iterate over all the newly created users as well.

Handling conflicts:

Two or more users may modify the folder simultaneously (or without syncing in between). This will create two lines of events (*event line)* with a common parent. The correct way of dealing with this is by analyzing the event logs, merging and having an authority approve the outcome.

But to simplify things, we will apply all of the events, but in a specific order. When receiving an event line that differs from what we have, we track both to the latest common parent (there will always be one, even if it is the start, “null”, event). After this, we 'reset' the state of the folder to what it was at that point (this is only affects the structure of the folder, and does not involve moving file data). After that we apply the two event lines in order, according to the event IDs simply in the order of the numeric value of the hash (important here is that each user applies, independently, the event lines in the same order). The head of the new state of the folder will be the head of the last applied event line. As there might be more than one branching event line, and within the event lines multiple sub-lines, each of these are applied in the same manner, recursively.

Handling missing parent events:

In case we get an event whose parent we do not have, we will try to fetch that event. This shouldn't be a problem because the one we got the event from must have it also in order for it to have resolved the same issue.

In theory; we might have excluded someone from our view of the shared folder. That is; we don't ask for the excluded persons updates and ignore those if we happen to see any. In this case the parent event of someone we do trust might be missing.

For simplicity, at this point we will be required to request the events of that excluded person as well, as this info is crucial for the conflict resolution. However, we will not apply that person's changes.

For instance, say we trust user B but not user A. User A changes a file, after which user B updates it. To be able to track user B's update in the event line, we need user A's change event (as it is the parent for user B's event). However, we never apply that event, meaning that we never 'see' user A's version. The content of the file will jump from what it was before user A's changes directly to the version created by user B.

Merging folders (optional):

We might consider that two shared folders are merged. This could be done by choosing one of the folder's ID as the new folder's ID (just compare them numerically and choose the larger number) and adding the 'loosing' folder's events to the winning folder. Essentially all the events of the loosing folder will be appended to the winning folder's event history, with the first event's parent changed to the head of the winning folder's log.

This does create problems with events added to the 'loosing' folders log after the merge. We should insert a 'merged-into'- type of an event to the end of the loosing folder's log, and have rules for dealing with those “afterlife” events. Another option would be to just keep the two history logs separate, but operate on the same directory structure.

Event types

The types of events possible are:

create / modify file: creating a new file and modifying a file (however small the modification) is seen as the same event. We simply have a new file, which may overwrite an old one (in case of a modification). Files can be directories. The event will contain the name of the file, its size and hash of contents, date.

Note: currently we have create file, create folder, modify file and rename events. These might be combined into a single 'create entry'- type of event later.

delete file: deleting a file. For simplicity, there's no 'rename', it is handled by delete following a create. As the create will contain the file size and content hash, we can use it to simply move the old file's data to the new file.

add user: adding a user to the share

delete user: deleting the user. This requires some thought still.

As we are only dealing with the events (and not file data) during syncs, we do not have to concern ourself with data merges or keeping backups (for the revoke-scheme). We either have the 'right' version of the file data or we don't. If we don't, we request it from someone who has it.

Locking files and meta data modifications could also be considered.

Data access and synchronization

Access to the file data is based on the checksum and size of its contents, as indicated in the most recently applied create/modify event for that file. If this data is found locally (even though associated with another file) it is used. Otherwise it is requested from other users.

Each user is required to maintain the revisions created by himself of each file, up to a limit (which can be measured in time or space used). This way the user who has created a version of a file can always be used as backup for getting it, in case no one else has it.

Data retrieval can be pre-emptive or lazy (only when the client requests the file). Hybrid models are also possible (get small files and the first x bytes of larger ones to buy some time to initiate the request for the rest..).

As each user will store the complete event log of the folder, it is possible to review the history of each file as well. Whether that file is still available depends on whether the creator (or someone else) is still caching it.

Access interface

The client interface to the shared folder will primarily be as a remote folder. The current state of the folder (based on the head of the event log) will naturally be shown, but we could also consider including the past versions somehow. One option would be to create for each file (e.g., “testfile.doc”) a similarly- named hidden folder (“.testfile.doc”) which contain the versions named according to their event ID (“testfile.doc-23423). These would naturally be read-only, and access might fail in case the version is not available anymore.

Access might fail also for normal files, in case the data cannot be retrieved. Writes to the files should always succeed as we treat each modification as a completely new file.

Connection management

We use HIP for connecting to the other users and to the SICX server. But from the gateway's point of

view, we will use stream-based connections.

In the version we're building, we will always try to maintain an active stream connection to the SICX data server. In future improvements, we should consider an UDP- based method with periodic keep-alive messages to decrease the load on the server.

Data protocol

Something simple, DataStreams if using java. As we probably will not be able to get HIP-based authentication, TLS could be considered.

SICX lookup server functions

The operations the gateway performs with the SICX server are:

- **Register.** On connecting, the gateway will send a registration request. This includes connectivity information (how others can connect to it).
- **Lookup.** In order to sync with the others sharing a folder, the gateway does a lookup for those. The gateway will return the connectivity information needed.
- **Proxy set-up.** In case gateways cannot connect to each other, a gateway can send a request for the SICX server to act as a proxy between them. If the SICX server responds ok, then all data from that point on will be forwarded to the other gateway. The gateway should open a new connection to the lookup after this.
- **Proxy notification.** The SICX server will send to the target of the proxy request a notification that any data following this will be data proxied from another gateway. The gateway should open a new connection to the SICX lookup server after this.
- **Connection request.** The gateway, when unable to connect to someone, may request the lookup to inform the target gateway to try and connect to him. If this fails, the proxy- method is used.

Gateway-gateway operations

The gateways should try to establish connections to all members

Invite: Invite a user to a folder. This could also go through the lookup server.

Data retrival: Request a data blob based on the content checksum (hash) and size. We might consider opening a new channel after this (as this one will be blocked by the data transfer), or use chunked data transfers. Actually, the data retrieval request could also include an offset and length, allowing for random access. But the receiver would not be able to verify the data then.

Has-block?: Query asking whether the other has a specific data block. This should be queried before starting the data transfers.

Sync request: Send a user ID and latest event ID created by that user, and expect a list of updates back. This is done for all users that we want updates from. This should be done on each connection.

Sync push: Each time a new event is created on a gateway, it will push this event to all connected peers. When receiving a sync push, unless the gateway has already seen it, it will also push this to its connections.

Also to consider:

Please-store: A request to store a data block. This could be when a file has been changed, and the gateway would like the change to be available from multiple sources.

Forward data request: In case none of the peers directly connected has a piece of data, we could request them to forward the request to their connections in order to find a copy. This data might then be proxied through the intermediate peers.

Local GW design

A daemon that provides network share. We use the milton library for providing the WebDAV interface.

UI:

- The folder configuration: create new, who is it shared with, invite new peers.
    - NoUI approach: Just create a new folder in the root of the webdav path. Have a special folder present for user management (e.g., “/Users”) where dropping a 'business card' of a user would automatically result in that user being added to the share.
- Respond to invites. Set up quotas for shares.
    - NoUI approach: when receiving an invite, an “invite_to_share_xxyy.info” file is created in the root of the webdav. This would contain the invitation (by whom etc). By deleting this file, the invite is rejected. We could have some special folder “Accept invite” in the root of the webdav where dragging the invite- info file would be interpreted as accepting the invite.
- Contact list, search for new users, import users' 'business cards'.
    - NoUI: We display the local user's “business card” (a file containing the public key and other info) at the root of the webdav. The user can copy it from there and share through email or other means. Another special folder “Contacts” would also be created where the user can drop the cards received from other users to create a contact list.
- Local client access configuration (which LAN client can access the webdav interface, authentication on the webdav interface).
    - NoUI: Special, editable, file at root of webdav (“acl.txt”) which would contain user-password combinations.

Architecture

Draft. The current model consists of users, shares and data stores. The users contain the user information, connectivity logic. The shares provide the structure (based on the event log) of a shared folder, while the data stores are relatively simple key-value stores for the file data.

Each file in the share is linked to its contents through the checksum of the data. When serving the file contents, the shares use this checksum to request the data from the data stores. We can therefore use the same data store for multiple shared folders, and possible reduce the space used by those as the data of identical files will only be stored once in the storage (there are some security issues to consider here, however). As the data stores only provide a quite simple interface towards the shared folders, these can be easily exchanged or upgraded without affecting the rest of the system. This is where dropbox or SICX-using modules would be placed.

SICX (commercial) component re-use

Todo: The components that could be re-used, or shared, from the mainline SICX development. The authentication system? The storage access modules?

Future features

SICX Integration

Providing local space for sale.

Seamlessly using external storage providers (including other users' who provide their storage for sale). Files could have in their meta external links, e.g., dropbox://jookos/path/to/file/data or something similar.

Data transfer optimization

Multiple data sources, caching, geo-optimizing.

Real P2P overlay and bittorrent- like swarms.

Rsync-like data transfers, splitting the file into smaller pieces (bittorrent- like).

Data storage optimization

Storing diffs of files instead of simple blobs. Quota management.

Content processing

As Tommo hinted; could be photo- folders that auto-create thumbnails, galleries

Webfolders that automatically publish things on HTTP (also galleries etc media things).

File format conversion folders.

Identity migration

How identities could be moved between places. Can we have the same identity in multiple devices? Would the shared folders have different IDs on each device?

In the current model, this would require atleast a change to the sync protocol, as we rely there on each identity having only a single event line.

Light-weight gateway-clients

Clients that primarily only look at the file structure. When opening files, the content would be streamed from another source (not stored on disk). This would actually be just the normal client, but with the local storage quota set to zero.