sib Design Document

May 7, 2014

Contents

List of Figures

1 TODO:

A

- refine documentation
- move common utilities to sib namespace (make the commands static functions in sib.py file)
- release alpha version for testing on my computers/phone
- save/load config details
- more thorough testing
- merging
  - trees

B

- NAT traversal
- profiling
- poll peers for new commits
- notify peers for new commits
- software update mechanism
- storage zlib compression
- make documentation actually accurate
- use first 2 letters of blob hash as folder name
- networking
  - implement file block request
  - implement send commits in range
  - implement matching commits from peer
  - delete blob message
  - implement command and high bandwidth sockets
    - command sockets only used for initializing new high bandwidth sockets
  - packet level encryption
  - better congestion control

C

- automatic port forwarding using (miranda-upnp?, MiniUPnP? PJSIP?)
- realtime sync folders
- GUI
- unit tests
- upload to github?
- public key encryption
- DDNS (namecoin?)
- multiple user download/upload
- caching
- make version for:
  - windows
  - android
  - mac
  - iOS
- installer for windows, linux, IOS, Android and Mac
- send email if peer is completely disconnected from all other peers
- pack large collections of smaller less commonly used files
  - storing many small files is very inefficient
  - design a pack header containing easily read hashes and indexes of all included files
- send a web link to an anonymous user of a single version of a file?
  - storage directory files are stored in folders named with SHA-224 hash of encryption key + salt
- error and consistency checking
  - recovery?
- distributed hash table (DHT) for automating peer connections

Done (in alpha)

- poll working directory for file changes
- auto-update working directory to latest commit
- merging
  - file blobs
- source control

2 Overview

sib is a utility for storing files among any other connected computers. It can be used for off-site file backups, transferring files to remote computers, accessing your files anywhere in the world and sharing files publicly. Computer users will be able to leverage their unused hard disk space and network bandwidth to essentially acquire the aforementioned benefits at no significant cost.

3 Dependencies

- Python 2.7
- pycrypto https://pypi.python.org/pypi/pycrypto/2.6.1
- watchdog for python https://pypi.python.org/pypi/watchdog

4 Getting Started

1. install pip
2. install dependencies
   (a) $pip install pycrypto
   (b) $pip install watchdog
3. import sib (main entry point is sib.py)

5 Concepts

- untrusted remote server (URS)
  - stores files for you
  - assumed to be untrustworthy
  - if any files have sensitive content they should be adequately encrypted before transfer to URS
  - Everything stored on the URS should be verified frequently.
  - All claims by URS (eg. bandwidth, uptime, etc.) should be verified using local experience and peer references

6 Main Features

6.1 File Priorities

- different files or folders can be set to be backed up different amounts and in different ways (optimize latency over reliability, etc.)

6.2 Verification

- users can ask remote computers for hashes plus salts of their files. This can be used to ensure the remote computer is actually storing the desired files
- users can download and upload files at random times to verify remote bandwidth
- remote computers can periodically hash local files to ensure consistency
  - if consistency failures, primary users should be informed asap.

6.3 Expandability

- all operations by utility will have a utility version number associated so utility upgrades can be more easily backwards compatible
- allow plugins for
  - determining how trust is computed
  - custom bandwidth/cache/space optimizations

6.4 Decentralized DNS

- no DNS server is needed. Clients will communicate regularly and if their IP changes they tell their peers. URLs will be given preference if they exist.

6.5 Version Control

- able to recover previous versions of files
- able to merge files (deal with conflicts)
- able to view file history: who modified it when

6.6 Peers

- users are verified locally using certificates?
- can add specific users as trusted computers (friends, family, etc.)
- can share files to specific peers by encrypting with specific keys
- can share files publicly by not encrypting (how are file lists distributed?)
- users can view which peers have which files and how many backups exist
- users can allow access to groups of people

  - voluntary creation of signing keys. eg. keys for:
    * universities
    * governments
    * companies

- trust networks
  - validity of content can be verified through trust networks
  - trust metrics?

6.7 Bandwidth Optimization

- only transmit file deltas (use rsync?)
- transmits parts of files from numerous computers at once to take advantage of multiple users' worth of upload bandwidth

6.8 Cache Optimization

- store commonly accessed files on computers with high bandwidth (maybe even pay for a small amount of space on something like amazon s3)
- store files on remote computers that are near by to minimize latency, but not too close to risk losses from local disasters

6.9 Space Optimization

- files that are shared publicly don't need to be backed up as many times
- incremental backups
- user can specify how much space they want to use, so at some point incremental backups can be deleted
- users can specify if they want to store full backups as well

6.10 Encryption

- able to encrypt with encryptrsync for fast incremental uploads, but able to use stronger encryption if fast incrementals aren't desired
- encrypt local files for security purposes
- encrypt files before they are sent to a remote computer so the remote computer can store them, but not know what the files consist of
- encrypt files with different keys, which can be given out, thereby controlling outside access to said files

- users can access all files using only their password and ip address or url.
- allow re-encrypting files in case master, or peer key is compromised
- varying levels of encryption with high level explanations
  - encryption that likely can't be hacked by:
    - grandma (+0% storage usage and +0% transfer times)
    - ordinary citizens (+1% storage usage and +5% transfer times)
    - large organization (+4% storage usage and +10% transfer times)
    - powerful governments (+8% storage usage and +20% transfer times)
    - powerful governments for the next 25 years (+16% storage usage and +40% transfer times)

7 Secondary Features

- git like version control?
- nice gui
- interface for adding annotation/comments/voting/editing?
- maintain file metadata. eg. file permissions, hardlinks, etc.
- annotation/comments of files, parts of files, and versions of files from peers and certain groups of peers
  - voting/rating on files/comments
- works on desktops and cell phones
- trending content both from friends and public
- distributed search
- distributed recommender system
- anonymity

8 Economy

- By default, users can **elect** to trade their services for other user's services eliminating the need for money in the transactions (very low barrier to entry). Trading HD space is mutually beneficial, as users will gain off-site backups, backups spanned over multiple HD's and increased file transfer speeds (files are spread across many computers).

- users can publish their HD space, max up and download bandwidth, up and download bandwidth reliability, up/downtime. With this information other users can bid to keep their files there.
- users should be able to maintain a bidding profile so pricing can change dynamically
- bidding should be conscious of pricing of industrial options eg. amazon s3
- users can store the following time series of information about remotes
  - upload and download rates
  - file verification successes and failures
  - up/downtime
- users can ask potential peers for a list of their past peers. Users can then contact those peers and download their time series to verify the remotes claims of bandwidth and uptime. Mainly users would only communicate with a number from 1-10 indicating how **truthful** they find the remotes published stats.
- if a remote has lots of peers that trust them then their ratings of other peers can be trusted more (somewhat like pagerank algo?)
- would need some sort of distributed crawler to get trust numbers for many users?
- allow a trial period where dummy data is uploaded to a remote and trust is measured
- allow trust to be computed in many different ways (plugin), one of which includes a user input.

9 Processes and Data Structures

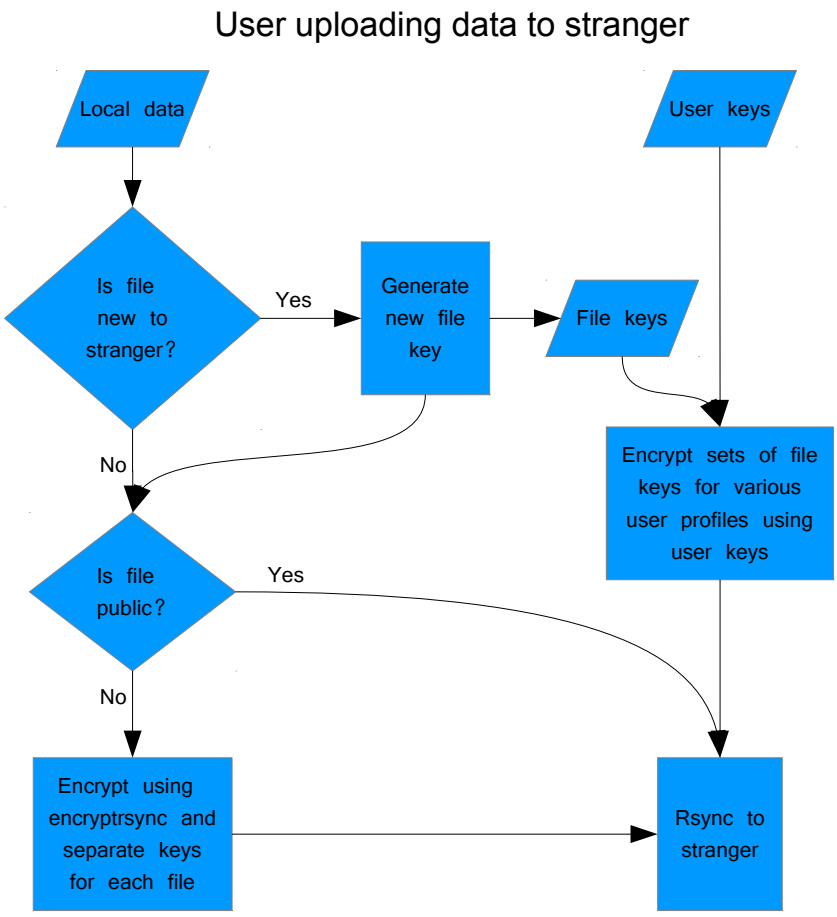

Figure 1: User uploading to stranger flowchart

9.1 User updating files from URS

1. Contact server
2. User sends most recent commit hash
3. Server sends any more recent commit and corresponding tree blobs
4. User reads new commit and tree blobs
5. User calculates list of necessary file blobs and sends this list to URS (in random order)

6. Server begins transferring file blobs to user
7. user applies deltas to file blobs and attains final file versions
8. final file versions are copied to desired user directory for
9. once all files are transferred from URS to user connection is closed

10 Interface Specification

10.1 Trusted Client

- encrypts blobs
- encrypts TOF (table of files)
- performs diffs between blob versions
- transmits blobs and TOF to other computers
- applies deltas to one file so one blob is the most recent file version
- TOFs contain encryption keys for blobs
  - multiple TOFs for multiple users. Some have encryption keys for different subsets of files
- javascript served from client can link to files on remotes
  - encrypted or public
  -

10.2 Untrusted Remote Server

10.3 File Blob Class Structure

Fields

- hash of unencrypted previous version
- diff opcodes to transfer from previous version to this version
  - opcodes: insert, delete, replace, equal, aggregate (signifies this is not a diff, but an aggregate version used to reduce bandwidth for future downloads)
- diff strings corresponding to opcodes

Methods

- write file blob to file

1. serialize
2. compress (zlib?)
3. concatenate version and size bytes
4. compute deterministic obfuscation data if necessary (seeded from SHA-1 hash of current version?)
5. pad for encryption
6. encrypt (AES ECB?)
7. set file name to protocol version, SHA-1 hash of encrypted file

10.4 File Blob File Structure

serialize file blob using JSON

- file name on URS is sib version byte followed by SHA-2 hash of non-encrypted file
- unencrypted file:
  - first byte is version byte
  - next 8 bytes are size of file in bytes
  - next ? bytes are hash codes?
  - any data beyond reported file size are size obfuscation bytes and/or pad bytes

10.5 Commit blob file structure

- all commit names are prepended by '_'

serialize blob using JSON
fields

- protocol version code
- username
- email address
- IP address
- signing key
- date (seconds)
- message
- hash of unencrypted commit tree

10.6 Tree Blob Class Structure

Fields

- protocol version byte
- tree built from python tuples
  - tuples contain:
    - hash of current node
    - string pointing to hash of lower elements
    - string indicating the file or folder name of lower element current node
  - tuples are sorted by current node hashes alphanumerically
  - hash of a current node is a re-hash of all lower elements and names

Methods

- write
  1. traverse tree alphanumerically by breadth first and date modified first and write tuples on separate lines with hashes and names separated by forward slashes
  2. compress (zlib?)
  3. concatenate version and size bytes
  4. compute deterministic obfuscation data if necessary (seeded from SHA-1 hash of current version?)
  5. pad for encryption
  6. encrypt (AES CBC?)
  7. set file name to protocol version, SHA-2 hash of encrypted file
- compute_delta
  - use libdiff from python
  - since tree is stored breath first and date modified first, files in the same directory are stored adjacent. This permits the use of libdiff to compute diffs on the tree data.

11 Networking

- servers and clients always listen for commands on ports: 55921, 56921, 57921, 58921, 59921
- once connected new requests to use other ports for further communication (even new sessions) are suggested

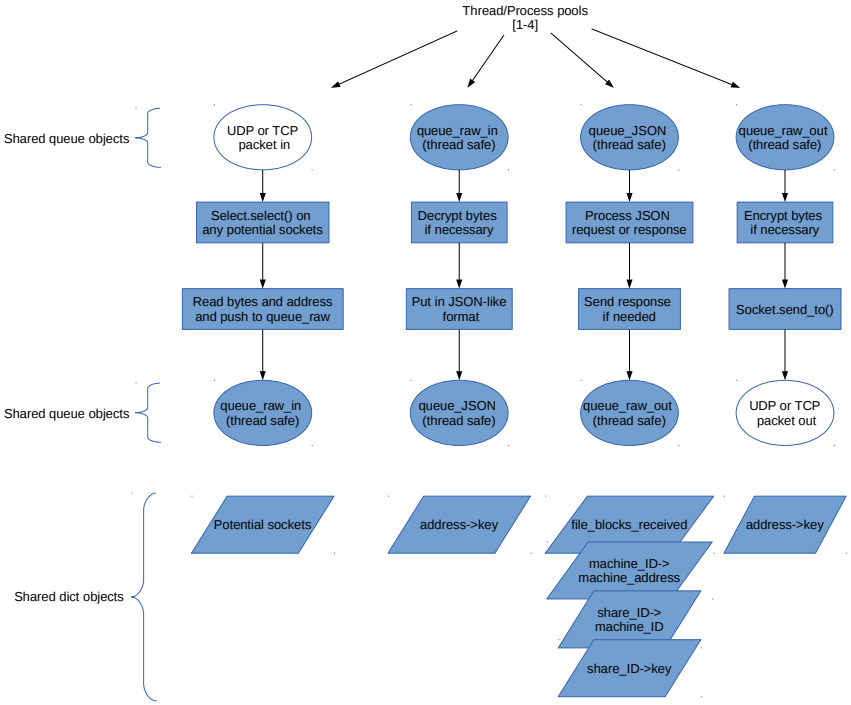

Figure 2: Internal socket data flow

11.1 Socket Protocol

11.1.1 Main message header:

- header is JSONRPC formatted
  - parameters are always arrays (name parameters have too much overhead)
  - first element is protocol version number
  - second element is congestion object
  - further elements are method specific parameters
- bytes after header are non-formatted message specific data

11.1.2 message types:

- authentication (?) (the word 'signature' AES encrypted using group name?) ('authenticate')
- new socket request ('socket request')
- send commits between range ('commits in range')
- file block request (client and server) ('br')
- file block (client and server) ('fb')
- ping ('ping'/'pong')
  - used to test a socket connection, or keep a socket connection alive
- verify file blob
- delete blob
- close connection (?)

11.1.3 send commits between range:

- msg specific args list [time/hash flag, full send or aggregate hash flag, range start, range end]
- time/hash flag:
  - if flag is 'time' then range start and end refer to UTC dates of commit file creations on URS
  - if flag is 'hash' then range start and end refer to commit hashes of alphanumerically sorted list on URS
- full send or aggregate hash flag:

  - 'full' means send all matching commit messages.
  - 'hash' means compute order agnostic hash of matching commit hashes
    * the agnostic hash is simply XORing

- dates in standard time format
- times according to file creation date on URS machine

11.1.4 File transfer block:

- message specific args list [hash, file size, block location, block size, block hash]
- message: file block

11.1.5 File block request:

- message specific args list [file hash, block size, [desired block locations]]
- if desired block locations is empty then send all blocks

11.2 Determining latest commit from peer

1. A peer sends send a 'commits in range' msg to B peer using a conservative start date and an end date of tomorrow
2. B peer transmits all matching
3. A and B then sort all commits based on hashes
4. A goes through list alphabetically and sends a 'commits in range' msg with aggregate hash flag set.
   (a) sent for every 50 commits
5. if aggregate hash does not match then send 'commits in range' msg with 'full' flag set.
6. continue until list is complete (for start and end of list send '0' and 'z' respectively so the full range of possible commit hashes is covered)
7. store UTC date and peer name to accelerate future commit updates?

what if someone uploads a commit to URS during this check? ans: doesn't matter. at the end of the process the A peer should be able to perform just step 1 again and be completely up to date

11.3 Typical transaction

11.3.1 Peer A updating from peer B

1. perform authentication
2. open new socket for large bandwidth
3. run determine latest commit process
4. peer A selects one or more commits to build
5. peer A requests all relevant blobs that aren't local
6. peer A closes connection

11.3.2 Peer A pushing an update to peer B

1. perform authentication
2. open new socket for large bandwidth
3. peer A asks continually asks peer B for parent hashes of desired commit until all dependencies are found on B
4. knowing the state of relevant commits on peer B, peer A can then send all relevant commits and file blobs to peer B

11.3.3 Peer A cleaning outdated blobs from peer B

1. peer A makes appropriate aggregate commits
2. peer A finds all local parents to aggregate commits and all corresponding blobs
3. peer A uploads aggregate commits to peer B
4. peer A tells peer B to delete information selected in step 2

12 GUI Design

12.1 Local client major features

- served locally
- can navigate local FS and select folders/files to sync
- can manage peers
  - add
  - delete

  - view peers stats
    * peer IDs
    * ping times
    * average bandwidths

- can view commits (in cool physics based directed graph?)
- can make new commits

12.2 Global client major features

- HTML 5 and javascript based
- served from peers
- can access any peer from browser
- tree view of available files

13 Concurrency Issues

- Downloading multiple files from multiple users. Have a non-blocking proxy.get_file() call run in another thread. How to determine get_file() progress/completion? Consider calling get_file_block(). Only measure completion by checking thread.join(). Need a thread pool due to high number of potential block requests
- Downloading entire repository from peer. At the same time, the peer adds a commit to the repository. Will partial files be downloaded? If not, there is no real problem as downloaded orphaned files is only a minor issue.

14 Congestion Control

- data structures
  - shared dict: in_flight
    * key: connect address
    * object: list of packets in flight along with each packet's send time
  - shared dict
    * key: connect address
    * object: list of RTT
- micro TP (μtp) is one solution, but the main downside is a typical increase in latency of 100ms

- better would be to operate at 100% bandwidth and 0 increased latency.
- how to estimate latency reliably?
  - send packet with timestamp, wait for reply
  - shouldn't uplink latency roughly equal downlink?
    - different services will saturate up or down. eg. Skype: up, torrents: down
- how to deal with other algorithms (like μtp)?
  - if we operate at 105 ms latency we will choke them out
  - if we operate at 95 ms we will be choked out
  - operate at 105 ms latency temporarily to estimate bandwidth
  - maintain 100ms latency and aim to send at 1/2 estimated bandwidth

A: packets in flight -packet -send time

B: response packets -response ID -recieve time

managed dict for A -contains managed priority queue -queued objects are packets -packets contain -send time -response ID

managed dict for B -contains managed priority queue -queued objects are packets -packets contain -receive time -response ID

congestion manager object -push sent packets and responses using queues -contains managed dict to track state of connections (addresses are keys) -possible states are: -0:unknown -1:choked -2:available capacity

-private -dict with addresses as keys -dict elements are priority queues of sent packets sorted by send time -dict with addresses as keys -dict elements are priority queues of response packets sorted by receive time -dict with addresses as keys -dict elements are congestion estimation values [latest RTT, min RTT, bytess_in_flight, max bandwidth?]

fuction -process_address(address) """Finds any matches in send and received queues. Updates congestion info for particular address Could be run in process pool. """