Peer Dropbox Design Document

Index

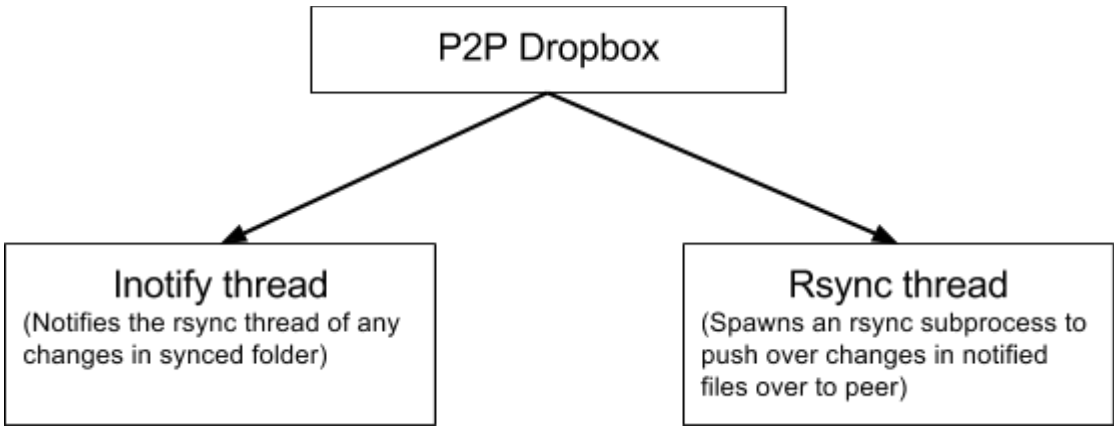
Introduction

The document talks about a linux-based peer-to-peer dropbox application which has the following features:

- Syncing a folder locally or between 2 or more peers automatically
- Bidirectional or unidirectional sync
- Retry mechanism to ensure reliability over syncing failures
- Automatic login, without password
- Automatic service-startup during boot-time, with sync starting automatically when network is available
- CLI to configure [add/remove sync-folders] and show file synchronization status

Basic Design

The design basically involves using two threads which perform separate tasks which will help us synchronize folders. The diagram below gives a summary of the two threads



The two threads are described below:

- **Inotify thread**: Responsible for notifying the main process of any changes in the files contained in the folder to be synchronized. It uses the kernel inotify API, where we

can add watches for directories which will notify us during any of the below mentioned events.

- File/Folder is moved into/created in the directory
- File/Folder is moved out/deleted from the directory
- File/Folder is modified inside a directory

In the event of a folder being added/removed, it also adds/removes the folder from the inotify watch list. The notifications are received by the parent-directory watch in inotify.

It also maintains a list of changed files to be processed by the Rsync thread, which is modified optimally, in the time-interval between rsync-subprocess creation.

- **Rsync thread**: Responsible for periodically (every 5 seconds) processing the list of changed files, and spawning an rsync subprocess for each changed file,which copies over files remotely.

It deals with various possible errors during the rsync process by using a retry mechanism with exponentially increasing timeouts upto a certain level.

It also maintains the file synchronization information to be used by the CLI to display current status.

Design Choices

1. **Use of rsync over scp**: `rsync` copies over changes intelligently ie. it copies over the differences instead of the entire file unlike `scp`. This ensures a much better use of network bandwidth.

2. **Use of inotify**: An alternative could have been to periodically `rsync` over the entire folder to push changes on to the peer. But inotify is much more efficient since we would start an rsync process only when something changes. This is important since we expect the dropbox folder change events to be rather infrequent and not happening all the time.

3. **Periodic polling of file change events by rsync thread**: Instead of processing the file change events immediately as they are added, the rsync thread processes them every 5 seconds. This was done to avoid having to process multiple duplicate notifications when the user makes a series of changes to one particular file.

Detailed Implementation

1. Basic Application Design -

Inotify thread

1. It creates a watch for the source directory and nested directories using `inotify_add_watch` which returns a file descriptor.
2. It then does a `select` on all the watch file descriptors, and gets blocked indefinitely until an inotify event occurs.
3. It updates the `inotify_event` list to be processed by the `rsync` thread whenever an inotify event occurs. It will also add or delete a watch if the inotify event is regarding the addition/deletion of a nested directory.
4. The `inotify_event` list is processed before it is executed by the rsync thread. Processing includes removal of certain redudant events (eg:- 2-3 modification events can be clubbed into one.)

Rsync thread

1. It periodically processes the `inotify_event` list created by the inotify thread. In processing each `inotify_event`, it forks a child process which carries out the rsync process.
2. It maintains a `pending_subprocess` *`linked-list`* for all child processes, which contains the child process PID, retry timeout, and the details about the folder/file being synced.
3. After processing the inotify_event list, it then iterates over the `pending_subprocess` list, polling without blocking, to check if any of the child process has exited. If any child process exits with a non-zero code, it forks another child process for retrying.
4. If the child process has not exited yet, it will poll again in the next iteration after a definite interval of time. Information of ongoing subprocesses is written to the dropbox-status file, which is used for interprocess-communication with the CLI python script to display current status.

We use mutexes to ensure synchronization between the two threads which share the `inotify_event` list. The `inotify_event` list is written to by the inotify thread and read by the rsync thread.

Libraries/Algorithms/Data Structures Used:

- The `pthread` library in C is used to create `POSIX threads`. Mutexes are implemented using in-built functions.
- Interprocess-communication is achieved by using `memory-mapped-files`, using the mmap function in `<sys/mman.h>` library to map the file content to memory with write protection. The same implementation is used in the CLI python script which has read protection enabled. `File-locking` using `<fcntl.h> library` in C is used to deal with race-conditions.
- An `undirected adjacency list` is used to store the watch hierarchy, with the parent-directory storing the watch-descriptors of it's direct children (sub-

directories) and the children storing the watch-descriptor of the parent directory, if present.

- A `linked-list` is used to store the rsync-subprocess list as fast event insertion/deletion is required.

2. Command-Line-Interpreter Design -

1. Shell commands `dbconfig, dbstatus, and dropbox-start` have been developed using python scripting to add/remove folders to sync list, show file-syncing status, and start the sync-program respectively.
2. The details of folders being synced in stored in the dropbox.conf file in /etc/ directory which is then read by the dbstatus python program.

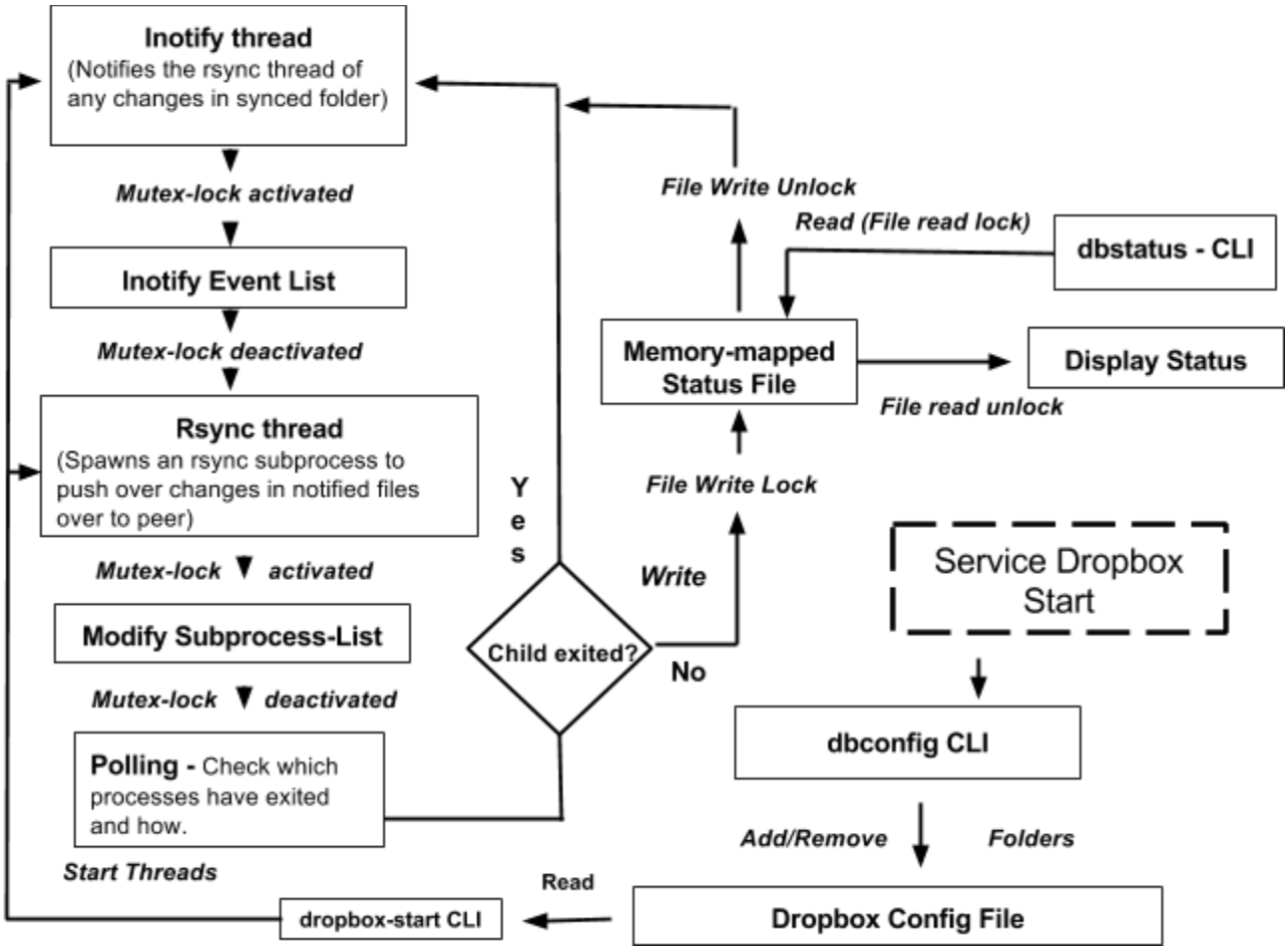
Libraries/Algorithms/Data Structures Used:

- The opt-parse module in python was used to develop the command-line tool, along with os, subprocess and fcntl modules to supplement the application starting process.

3. Service-Design and Packaging -

1. An init.d script is used to enable the application to be started/stopped/restarted by using the `service <service-name> start/stop/restart` shell command.
2. An installation script for placing the different file in appropriate location and resolving dependencies with other packages.

Main Flowchart -



Extra Packages Required:

For the service to work on intra-net networks, such as iitk, both locally as well as with the external networks, an external-ip address needs to be obtained which acts as an identifier for our connection. Logmein-hamachi is included in the installation script which bridges the network between two computers on our local network.

Caveats & Limitations

1. Maximum of 1024 nested directories allowed - This is because the inotify mechanism creates a file descriptor for every watched directory and the per-process file descriptor limit is 1024.

2. Security is compromised. Due to the passwordless login setup, the computer is susceptible to password-less ssh login from the other computer.