Snap – System Snapshotter

Masters Project Paper
Mohammed Morsi
Syracuse University, 1 / 2008

Table of Contents

I. System Overview....3
1 Project Description....3
2. Background....4
3. Target Users....5
4. Requirements....7
II. System Layout....8
1. Primary Interface....8
2. Exceptions and Errors....9
3. Configuration System....10
4. Callback system....12
5. Plugin Interface....13
6. Backend....16
6.1 Record files....17
6.2 Entity Classes....18
III. Example Implementations....20
1. A Package System Plugin....20
2. A sample graphical frontend....21
IV. Testing....24
1. Insufficient Permissions....24
2. Configuration verification....24
3. Plugin loading and instantiation....25
4. Backup and Restore operations verification....26
5. Generating and Parsing Packages and Files Record Files....26
V. Future Work....27
VI. Conclusion....29
Appendix A. Source Code....30
1. Directory Layout....30
2. Manually Building and Installing the Source....31
About the Author....32

I. System Overview

1 Project Description

Snap is a system backup and restoration utility for Linux distributions that uses the underlying package management system to record and reconstruct software and system configuration. The core library scans and stores the name of all software installed on the user's system as well as new or modified files which the package system cannot track on its own. The result is a snapshot of the system that is then fed into the restore operation which installs and updates all software and dependencies before copying the configuration files back. The backend library may be used by any application with superuser privileges, and two utilities, one text based and one graphical (GTK2), are provided for the end-user.

Snap employs a package management system interface that is completely abstracted from any specific system. Currently a plugin is written to interface with the Yum Package Management System, specifically when running on Fedora Linux. The plugin architecture and interface is available and well documented, thus any developer can add support for any particular package system running on any distribution. Package management system plugins are responsible for performing the core grunt work of the backup and retrieve operations in addition to any necessary system maintenance operations. In addition to the package system plugin interface, a simple callback system is employed by the Snap library to inform any client of the internal operations. Due to the cross-platform compatibility of Python, Snap can potentially be used with any interface, along with any software and file backup / restoration system a host operating system provides. Though for the purposes of this project, we will be focusing solely on Linux.

Initially, we will explore the motivations behind and intended audience/functionality of Snap. We dive into the design specifications next, and detail the general system layout and workings, in addition to the plugin and callback interface specifications. Test cases are then written and discussed, in addition to any potential pitfalls and edge cases to check. We will end by discussing future work that can be done to extend and simply the Snap system.

2. Background

There have been many attempts at system backup and restoration utilities. Possibly most notable are the 'System Restore' application for Windows, the 'Backup' program for Mac OSX and rsync and dump / restore for Linux. Unfortunately many of these utilities are limited in scope. System Restore for example will only backup and restore certain critical files and registry settings, so that a user may revert his computer to a working state upon system corruption. Any personal files and installed software will be lost. Other utilities which provide more robust file system backup and restoration functionality, lack the ability to interface with advanced operating system management systems such as the install / uninstall software services in Windows and the package management systems in Linux. Finally system administrators who need to manage several machines running different operating systems and environment have no unified, simple to use, cross-platform utility and must rely on several disparate means to accomplish system backup / restoration tasks.

This project aims to provide a library and set of utilities to mitigate these problems. Snap is a cross-platform backup and restoration system which may be extended to interface with any O.S. and system environment. Snap intends to provide simple, easy to use interfaces which client applications, sysadmins, or even personal computer users can use to take and restore system snapshots. Pushing many of the backup and restoration activities onto the underlying package management system is a natural solution that allows a user to only locally save the a minimal data needed to record what needs to be restored from a remote location (namely the package repositories).

3. Target Users

Snap provides a simple administrative interface which any application can include and invoke to perform the most abstract system backup and restoration operations. End users come into play as administrators which will need to use the tool, possibly regularly, to restore and backup. Developers need to interface with Snap in order to extend the support for package management system and front ends.

User: Client Software

User Interaction: A client application or library includes the Snap library and invokes the interface, optionally passing in configuration options, to backup and restore software and files. For example, to perform a simple software and file backup operation, a client would

import snap

snapbase = Snap.SnapBase()

snapbase.options.handlepackages = true

snapbase.options.handlefiles = true

snapbase.options.include = '/etc/httpd:/var/local/mysql:/etc/openldap'

snapbase.restore()

User Constraints:

- Since Snap is written in python, the client will need to be as well, else the Python C interface must be employed. Using that snap can be ported to virtually any language.
- Snap needs to enforced permission verification to ensure the effective user has sufficient privileges to access / modify the requested files and software

Required Functionality:

- Simple, abstract interface which to invoke Snap operations
- Able to simply specify any number of options to alter the behavior of Snap
- Callbacks which to subscribe to, in order to be informed of meaningful events.

User: System Administrator

User Interaction: Invokes Snap functionality through a command line or graphical tool. An example command to restore the system from a specified snap file would be,

snaptool –restore –files --packages –snapfile /home/mmorsi/snap-01.10.2008-00.26.56.tgz

Required Functionality:

- Meaningful, human readable output, notifying the user of operations, instructions, errors, and output.
- A single, unified data set container, in this case a snapshot file (snapfile), to be input / output by the system

- Able to preset the behavior of snap by modifying a configuration file, loaded every time the utility is executed
- Able to simply specify any number of options to alter the behavior of Snap

Additional Useful Functionality:

- Ability to select target host which to store / retrieve the snapfile
- Inclusion of a ‘snap partition’ mechanism which to write snapfiles and automatically read / restore on system corruption
- Built in tools to schedule automated backup / restoration tasks (currently trivial to do with the Cron daemon), possibly with incremental features so as to minimize repetitious data transfer.

User: Plugin Developer

User Interaction: Implements plugins to add support for specific package management systems, operation systems and distributions, and frontends

Required Functionality:

- Abstract, well documented interfaces which to expand in order to implement required functionality
- Robust message and status passing system to inform and receive notifications of meaningful events and results.
- Simple configuration system which to specify plugin selection and options

4. Requirements

Based on analyzing the target environment and aforementioned users, the following overall requirements were derived for the Snap library

- Provide a simple, cross-platform abstract interface which to take and restore snapshots.
- Support a simple and extendable configuration and option parsing system that supports the following options: help / version (strictly metadata), mode of operation (backup / restore), verbosity, input / output file location, files and directories which to include / exclude in processing
- Provide an abstract, well-documented plugin interface to add support for additional package systems. This plugin interface should allow specific implementations to specify exact ways which to record and reconstruct installed software and perform system maintenance operations necessary for the flawless flow of operations.
- Provide simple package system plugin configuration/selection and deployment mechanism. To preserve sanity/integrity from the client and the package system, the Snap proxy must catch and handle all exceptions raised by the plugin, and translate them into any appropriate pre-defined Snap Exceptions (if applicable.)
- Ensure software and filesystem sanity is preserved during a failure with any package management system plugin
- Support a messaging and status system via callbacks and custom snap exceptions. Messages should be clear and only employed during meaningful events which the client may need to know about and respond to. A client should not be forced to handle any events which are not needed. Exceptions should be well documented and specific to the Snap system so as to be uniquely captured and handled by a client.
- Restrict unprivileged users to privileged operations. Since this utility works on overall system maintenance operations, it will be a hard-enforced requirement that a superuser enabled process is accessing the library.

To assist administrators, and provide an example and starting point for future frontend developers, two python executables are also provided. 'snaptool' is the command line version, runnable on any OS, and 'gsnap' is a GTK/glade based version for GNOME environments. Snap frontends provided to users must meet the following requirements:

- Provide simple, human readable and writable interface to Snap system
- Allow human users to specify configuration options via a superuser-accessible config file, command line options, and graphical options
- Interface with the snap call back system and display human readable messages for relevant events, accounting for verbosity options set by the user
- Exit gracefully, under no circumstances should the integrity of the running environment or operating system be violated

II. System Layout

The snap library consists of the ‘snap’ subdirectory of the project which gets installed to the system-wide Python library directory up running ‘make install’. It consists of a series of modules defined as the following

- callback.py - defines the base callback interface which frontends extend and use to translate messages
- configmanager.py - defines the configuration interface which to set configuration options and read the config file
- exceptions.py - defines all public exceptions thrown by snap and what they mean
- files.py - internal file processing
- __init__.py - the base snap module, provides the primary public interface
- packages.py - internal package processing
- packagesystemadaptor.py - package system proxy, run package system operations, and monitor, handle their status
- packagesystem.py - base package system interface, which package systems plugins extend and implement
- snapoptions.py – internal options configuration

After installed, any of these modules may be included in a client program simply by running *import snap/modulename*, for example *import snap/callback*

1. Primary Interface

SnapBase, defined in snap/__init__.py, implements Snap’s primary interface’s. It ensures the effect uid is that of the superusers, and will raise exceptions on errors. From the public’s perspective, the primary interfaces consists of the following definition

```
class SnapBase
  options # configuration options which to alter the behavior of Snap, see
  # snap/configmanager::ConfigOptions

  # void and only constructor
  def _init_(self)

  def backup(self, installed_packages = None)
    '''
```

preform the backup operation, recording installed packages and copying new/modified files , tailoring the workflow to specified options

@param installed_packages - optional list of snap.packages.Package to install , else the system will be scanned and all currently installed packages recorded.

@raises SnapPackageSystemError - if an error occurs when backing up the packages

@raises SnapFileSystemError - if an error occurs when backing up the files

'''

def restore(self):

'''

perform the restore operation, restoring packages and files recorded tailoring the workflow to specified options

@raises SnapPackageSystemError - if an error occurs when restoring the packages

@raises SnapFileSystemError - if an error occurs when restoring the files

'''

Clients simply instantiates SnapBase and invokes operations on it by running

import snap

snapbase = snap.SnapBase

snapbase.backup

snapbase.restore

2. Exceptions and Errors

Snap will handle all Python errors generated by any internal operations save for a small subset defined in snap/exceptions.py. Any exception thrown by the primary interface is fatal, and after any cleanup operations have been performed, so the client can safely do the operations required to terminate execution and exit if desired. The following are the possibly exceptions Snap will throw,

class SnapArgError(Exception):

"""An illegal argument to the system was specified or an invalid

option set was detected in ConfigManager.verify_integrity()"""

class SnapFilesystemError(Exception):

"""An error occurred during a filesystem operation"""

class SnapMissingFileError(SnapFilesystemError):

"""A required file was not found"""

class SnapMissingDirError(SnapFilesystemError):

"""A required directory was not found"""

class SnapPackageSystemError(Exception):

"""An error occurred during a packagesystem operation"""

class SnapInsufficientPermissionError(Exception):

"""The user does not have permission to perform the requested operation"""

Due to inheritance, the client must catch the following exceptions at a minimum to safely and gracefully exit.

SnapArgError, SnapInsufficientPermissionError, SnapPackageSystemError, SnapFilesystemError

3. Configuration System

The SnapBase constructor instantiates a ConfigOptions (defined in snap/configmanager.py) object and assigns it to the 'options' member. It is up to clients to set the appropriate options to configure the behavior of Snap. To help, Snap provides a configuration manager class (also defined in snap/configmanager.py) which can be used to automatically parse the configuration file as defined in snap/snapoptions.py (current /etc/snap.conf) and the command line. The configuration manager provides the following public interface

class ConfigManager

def __init__(self, targetconfig):

'''

Single constructor takes one required arg. Automatically parse configuration file, presetting options, and setup the parser to interpret the command line

@param targetconfig - instance of ConfigOptions which to set values retrieved

from the config file and command line on

'''

def parse_cli(self):

'''

parses the command line an set them in the target ConfigOptions

'''

def verify_integrity(self):

'''

verify the integrity of the current option set

@raises - SnapArgError if the options are invalid

'''

Command line options are specified in standard double-dash-prepended format. Equals signs for value based options is optional. The following options are specified via the command line:

```
# --help      - standard help
# --version   - standard version
#
# --backup    - set mode to take snapshot
# --restore   - set mode to restore snapshot
# --v(erbose) - verbose output
# --snapfile  - file which to generate / restore
#             - time of snapshot and extension will be auto appended
# --snaphost  - uri to host to store / retrieve snapshot
# --(no)files - disable/enable file backup / restoration
# --include   - directories to backup
# --exclude   - directions to exclude when backing up
```

The configuration file should be in ini format, with only one, 'main', section. An example can be found in resources/snap.conf. The following option are specified via the config file:

```
# packagesystem=[yum|apt-get|portage|swaret|...] - underlying package management system
# default values for any of the command line options
```

4. Callback system

Callbacks are methods invoked by Snap when a meaningful event occurs. The default callback interface, SnapCallbackBase, defined in snap/callback.py) does nothing an merely exists for clients / frontends to extend. All callbacks are optional, and a SnapCallbackBase derived class can include /exclude those which it needs. All in all, the following callbacks are invoked by snap

warn(self, message) – generic, non-fatal warning with message string

error(self, message) – fatal error, indicates snap will terminate with message string

init_backup(self) – called before the backup process

backup_packages(self) – called before package backup starts

backup_package(self, package) - called upon backing up a package with the snap..packages.Package that was backed up

backup_files(self) - called before file backup starts

backup_file(self, file) - called upon backing up a file with the snap.files.SFile that was backed up

snapfile_created(self, snapfile) – called up creating the snapshot with the snap.files.SnapFile that was created

post_backup(self) – called after the backup process

init_restore(self) - called before the restore process

restore_packages(self) - called before package restore starts

restore_package(self, package) - called upon restoring a package with the snap..packages.Package that was restored

restore_files(self) - called before file restoration starts

restore_file(self, file) - called upon restoring a file with the snap.files.SFile that was restored

snapfile_restored(self, snapfile) - called up restoring the snapshot with the snap.files.SnapFile that was restored

post_backup(self) - called after the restoration process

To register the custom callback implementation with snap, a client simple sets the global callback, snap.callback.snapcallback to an instance of his class. A simple command line tool could invoke Snap functionality while display the smallest meaningful subset of the messages like so,

```
import snap
import snap.callback
import snap.configmanager
class CommandLineCallback(snap.callback.SnapCallbackBase)
    def error(message):
        puts ‘Critical error ‘ + message + ‘ terminating operation’
```

```
    def warn(message):
        puts 'Noncritical warning ' + message
    def snapfile_created(snap, snapfile):
        puts snapfile.snapfile + ' snapfile created'
    def snapfile_restored(snap, snapfile):
        puts snapfile.snapfile + ' snapfile restored'

snap.callback.snapcallback = CommandLineCallback()
snapbase = Snapbase()
configmgr = snap.configmanager.ConfigManager(snapbase.options)
configmgr.parse_cli
configmgr.verify_integrity
if snapbase.options.mode == snapbase.options.RESTORE:
    snapbase.restore()
    return 0
if snapbase.options.mode == snapbase.options.BACKUP:
    snapbase.backup()
    return 0
```

5. Plugin Interface

Snap package management system plugins must be defined in the directory specified by PACKAGE_SYSTEMS_DIR in snap/snapoptions.py. By default the file name should be the name the developer is giving to the plugin (a combination of package system name and operating system name is a good idea) all in lower case prepended to the text 'packagesystem.py'. As far as the actual implementation, a package system plugin should provide a class whose title is the name of the plugin (camel cased) prepended to the text "PackageSystem", and derives from PackageSystemBase as defined in snap/packagesystem.py. For example a plugin for Portage, Gentoo's package management system would reside in a file named 'portagepackagesystem.py' and would consists of the class definition.

```
import snap.packagesystem
class PortagePackageSystem(snap.packagesystem.PackageSystemBase)
```

If these conventions are followed, the name of the plugin to be used can be set in the snap configuration file, and the appropriate class will be dynamically loaded and executed by the package system proxy (defined in snap/packagesystemadaptor). The proxy class, PackageSystemAdaptor, in itself derives from

PackageSystemBase and in addition to loading the target plugin on instantiation, may be invoked like any particular plugin. The proxy will call the same method on the plugin it previously loaded, while catching all errors and exceptions and translating them into callback messages or a SnapPackageSystemError (defined in snap/exceptions.py), so that a faulty plugin or failed operation can handled by Snap and the client. Since the PackageSystemAdaptor is instantiated and handled by SnapBase, neither the client nor any plugin developer need to interface with it.

The PackageSystemBase class defines the following series of generic operations which plugins must implement in their of fashion. All methods are invoked and thus all must be implemented, though it is up to the developer of the plugin to figure out how to appropriately interface with the target package management system, which may involve library or remote calls, or executing external applications.

```
class PackageSystemBase
    def backup(self, basedir)
    '''
    perform any operations needed to backup the state of the package management system
     @param - basedir – root directory which to store any files in
    '''

    def restore(self, basedir):
    '''
    perform any operations needed to restore the state of the package management system

    @param - basedir - root directory which to copy files from
    '''

    def update_system(self):
    ''
    perform any actions needed to update the current system before any new packages are installed
    '''

    def get_installed_packages(self):
    '''
    Return a list of packages.Package for each installed package in the system.
    '''
```

def install_package(self,packages):

'''

Install the given list of packages.Package'

@param - packages - the list of packages to install

'''

def get_file_status(self,file_name):

''''

Returns True if the file in the given path is "marked", else False.

A marked package is one that is not tracked by the Package Management System or one that has been modified since it was installed'

@param - file_name - path to the file which to check the status of

'''

The package system proxy, package system base class, and individual plugins all operate on lists (eg. Arrays) of instances of the Package object (as defined in snap/packages.py, see section ***6.2*** below). It is not the job of any of these modules to store these lists persistently.

6. Backend

Driving the primary interfaces is a software and file tracking and maintenance system. The backend deals with software in terms of packages, and preserves / reconstructs them via instances of the Package object and a PackageFile (a log of packages) both implemented in the internal snap/packages.py module. Like wise, a file maintained by snap is represented as a SFile as defined in snap/files.py module, along with operations which to record / restore them, generic safe file system operations, and the definition of and operations on the SnapFile entity.

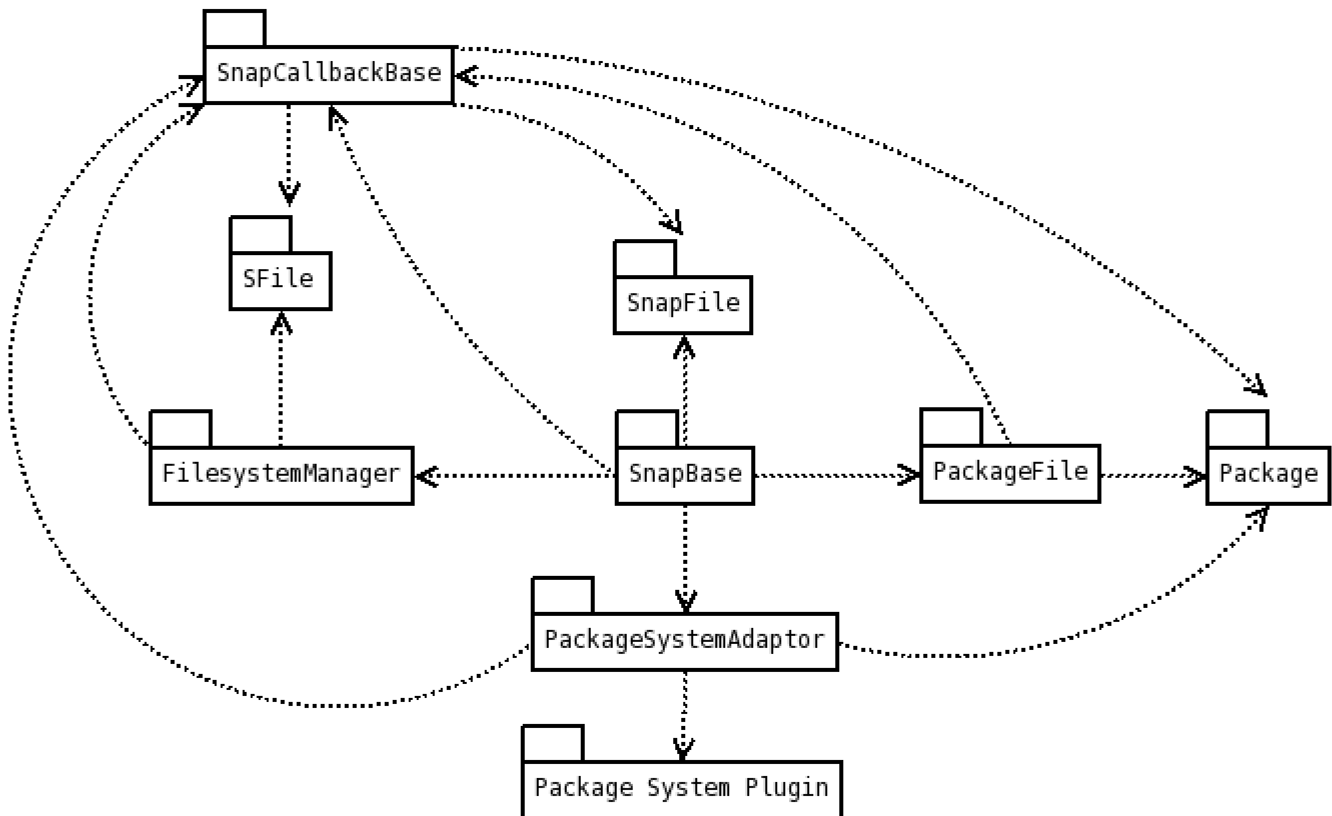

Figure 1. Snap System Snapshotter Library Package Diagram

As we can see in the package diagram, only the top level Snap interface (SnapBase), and the filesystem and package system interfaces have write access to the callback system, which in return depends on several lower level entities to pass along to the client. SnapBase records handles all filesystem and file tracking operations through FileSystem manager, while package system operations are split up between PackageSystemAdaptor which simply deals with lists of packages, and PackageFile that stores the record persistently.

6.1 Record files

Snap employs a simple XML file schema to record and restore software and files located on a user's system. These files and packages are constructed and parsed in a similar fashion though implemented in disparate modules. snap/packages.py contains the PackageFile class, responsible for generating and parsing the packages record file, provides the following public interface,

```
class PackageFile:
    def __init__(self, filename)
    '''
    class constructor, open and prepare the specified file to be read/written
    '''

    def read(self):
    '''
    parse the contents of the package file, and generate / return a list of snap.packages.Package
    corresponding to the packages parsed.
    '''

    def write(self, packages):
    '''
    write the list of snap.packages.Package to the file
    '''
```

Similar to the PackageFile class is FileManager, located in snap/files.py, which defines the restore & backup methods, respectively corresponding to the read & write methods previously defined. In addition to merely handling the record file, FileManager also takes care of backing up and restoring the actual files from source to destination. The 'target' directory is specified via the constructor (in addition to the record file location, similar to that in the PackageFile constructor), and is used as the destination in the 'backup' operation and the source in the 'restore' operation. If the client has specified any files or directories to include via the configuration system, only those will be traversed, else all files starting from the root directory '/' will be scanned (note only files which the package management system deems as 'new' or 'modified' will actually be copied). Directory structure and file system ownership and permissions will be preserved for each file throughout the entire process.

FileManager also defines static methods to perform generic file system operations. These methods are employed by Snap in order to centralize all file system operations (even though Python is cross platform, some of its libraries and methods within are operating system specific), provide an exception safe proxy to filesystem operations (all errors will be caught and translated into SnapFileSystemError as

define in snap/exceptions.py), and provide a simple failsafe mechanism. If the constant 'FS_FAILSAFE' is set to 'True' in snap/snapoptions.py, Snap will go through all of its normal operations, but ultimately FileManager will not make any changes to the underlying file system. All in all, the FileSystem manager provides the following static methods for use,

def make_dir(target) # create the specified directory

def get_file(path) # create and return a snap.files.SFile for the file on the given path

def get_all_files(startingDirs='/', excludingDirs='', psa=None)

 # return a list of SFile's corresponding to files in one or more directories. Specify a colon separated list of directories to recursively include/exclude and an optional package system adapter instance to use to retrieve file status (else all files found will be returned)

def get_all_sub_directories(startingDir='/')

 # retrieve a list of all subdirectories under the specified one

def copy_file(file, destdir, includedirs=False)

 # copy specified file to the specified directory, preserving directory structure if specified.

6.2 Entity Classes

The lowest level of object in the system, these correspond one-to-one to files and packages retrieve / to-be-restored and the ultimate input / output of the system. It is not the intent of these to be complicated whatsoever, as Snap grows, additional means which plugins can store and retireve additional metadata will be derived. Until then the smallest / simplest subset of the requirements to persistently store packages and files will be recorded. (see *Future Work* below)

SFile, defined in snap/files.py, is used to track every file the system backs up and restores. In backup mode, the list of directories the user specified are scanned and and an instance of SFile is created for each file found. Sfile contains the fullpath of the file, the name (simply the last part of the fullpath, for convenience purposes), and the status as returned by the PackageSystemAdaptor.get_file_status (only those marked as "new" or "modified" get recorded / retrieved).

Package, defined in snap/packages.py, is used to track every package the system records and installs. In backup mode, the client specifies the list of packages or the PackageSystemAdaptor.get_installed_packages is used to return the complete list of all locally installed packages. Currently only the name and version are stored.

Finally, SnapFile is used to construct / read the tar.gz file which Snap outputs to and accepts from the end user. Defined in snap/files.py SnapFile accepts a filename and a working directory to use in its operations. When invoking the 'compress' method, SnapFile archives the contents of the working directory and compresses it, resulting in the final tarball. When invoking 'extract', snap uncompresses the file and unarchives its contents into the working directory. Either way, before operation begins and

after it ends, it is up to Snap to process the input / results.

These three classes are generated and used by Snap's internal mechanisms and are explicitly passed to the client via the callback system. Thus, if the client is handling the *backup_package*, *backup_file*, *restore_package*, *restore_file*, *snapfile_created*, or *snapfile_restored* callbacks, it needs to know how to handle entity object.

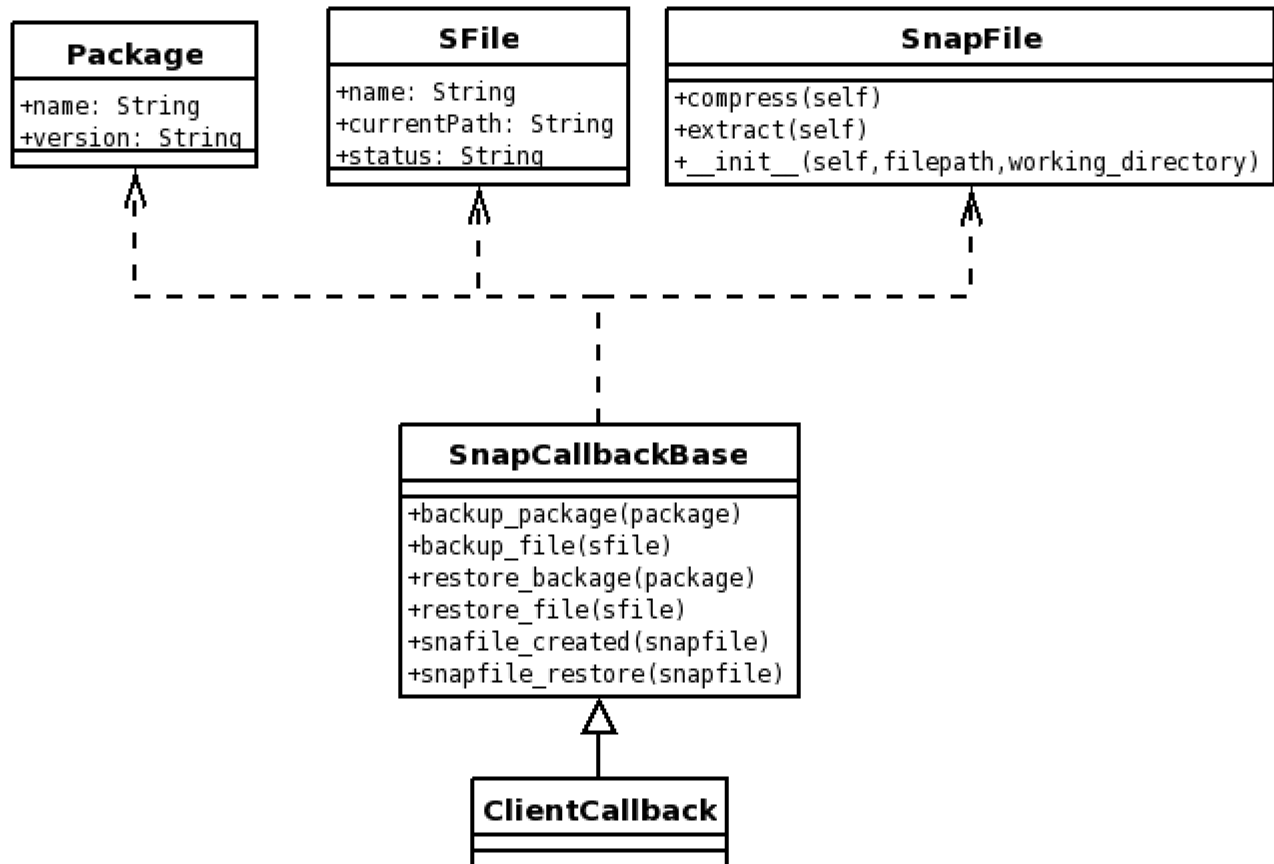

Figure 2. Entity and partial-Callback Class Diagram

III. Example Implementations

The Snap Project includes the Snap library as well as everything needed by a human user to run and effectively use Snap. Currently this includes two frontends, one graphical and one text based, a plugin for the Yum package management system, and various other files and resources needed for everything to work, Everything can be globally installed by running the bin/setup.py script. Here we will look at the Yum plugin and the graphical GTK based frontend.

1. A Package System Plugin

The package system plugins must derive from the plugin base class described in section *2.5* and implement all functions in their own fashion (less an exception be thrown when the package system proxy goes to call that function). The methods defined in PackageSystemBase serve as general abstract package system operations. It is up to the developer to give specific meaning to those operations. packagesystems/yumpackagesystem.py contains the current implementation of the Yum/Fedora plugin for Snap. The local RPM repository is employed as well as the yum executable. Development was started on another Yum/Fedora plugin, one that uses the Yum library, but was discontinued in lieu of other things. It is located in packagesystems/devel-yumpackagesystem.py, but will not be discussed from here on out.

First and foremost, we see the YumPackageSystem class definition and constructor,

```
class YumPackageSystem(PackageSystemBase, yum.YumBase)
    def __init__(self):
        yum.YumBase.__init__(self)
```

By deriving from YumBase, we can easily access some of the useful internal instance methods. Next we see, the implementations of PackageSystemBase.backup and PackageSystemBase.restore. These two simply serve to backup and restore /etc/yum.conf and the repository definitions in /etc/yum.repos.d. Thus when we goto restore software later on, we have all the repositories available that the user had on his original system. The restore function runs the update command to update all the currently installed software before any new installation is attempted. Following this, we see the implementation of PackageSystemBase.get_installed_packages which loops through the list of all installed software in the local RPM database (as given to us by YumBase.rpmdb) and construct instances of snap.package.Package for each one. The implementation of PackageSystemBase.install_packages does the opposite, it takes a list of Package objects and executes yum to install the software. One thing to note, some tweaking of this plugin was done to get software installation working properly with Fedora. Specifically, the kernel module packages are extracted from the list and installed first. This is because, allowing everything to be installed together results in a conflict in kernel dependencies resulting in a failure and nothing being installed. Finally we see the implementation of PackageSystemBase.get_file_status which takes the path to a file and determines if any package provides it and if so if it has not been modified since installation. If either case fails, the file is marked, and Snap ultimately backs it up and restores it.

2. A sample graphical frontend

Snap includes a graphical GTK-based fronted to the backup and restore operations. While tested only in a GNOME environment running on Fedora Linux, Snap should be able to function in any environment that provides the GTK libraries (including Windows after they have been installed). Furthermore, the graphical frontend, dubbed gsnap, relies on a Glade resource file to provide the look / layout for the actual file. Glade is a GTK GUI builder available for Linux and outputs an xml resource file which may be loaded and used in any application by importing the correct libraries. This file can be found in resources/snap.glade. For the purposes of this project, as simple two window design was used, with the output being rendered on the main window, and an options dialog to control the flow of Snap.

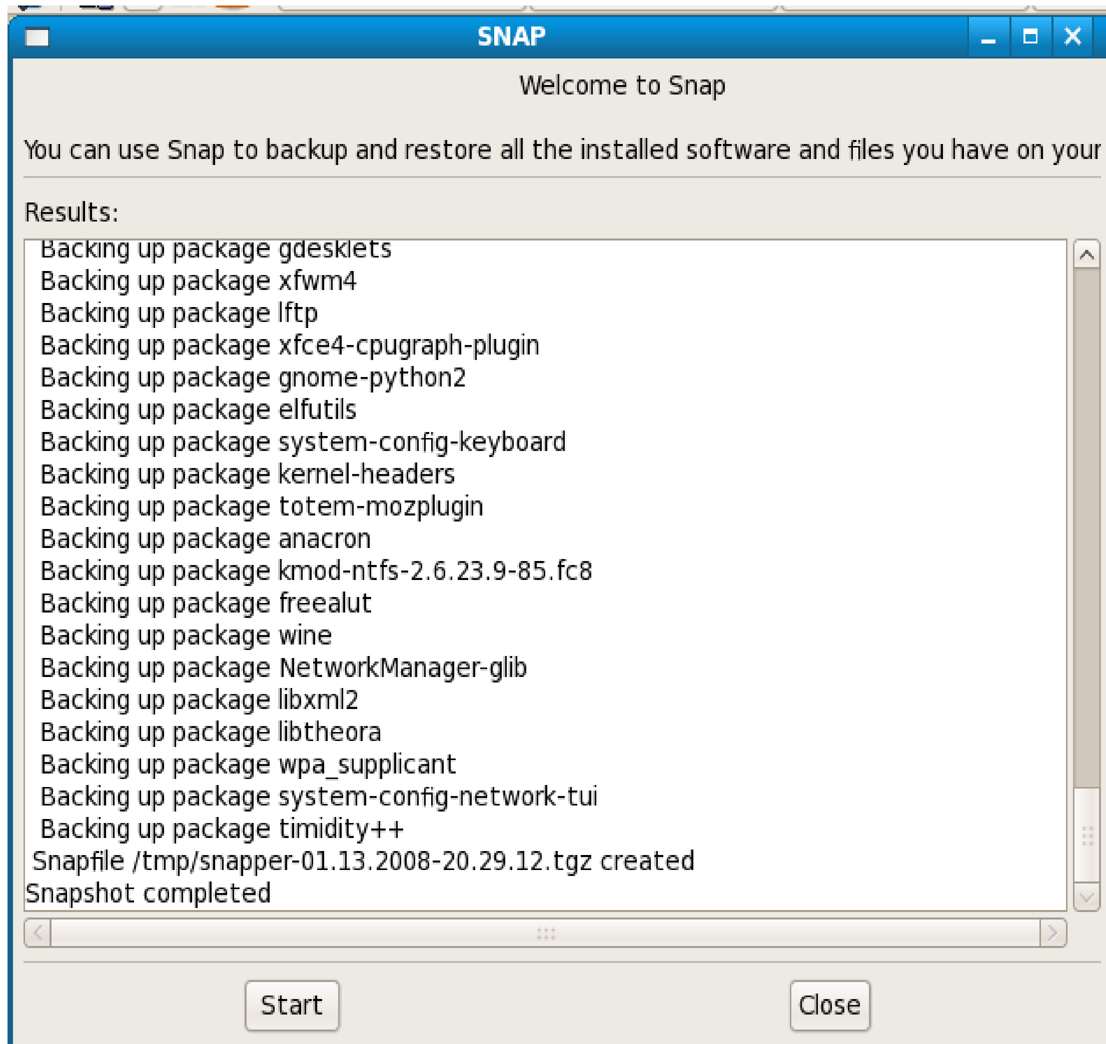

Figure 3 Snap Main Window and Output

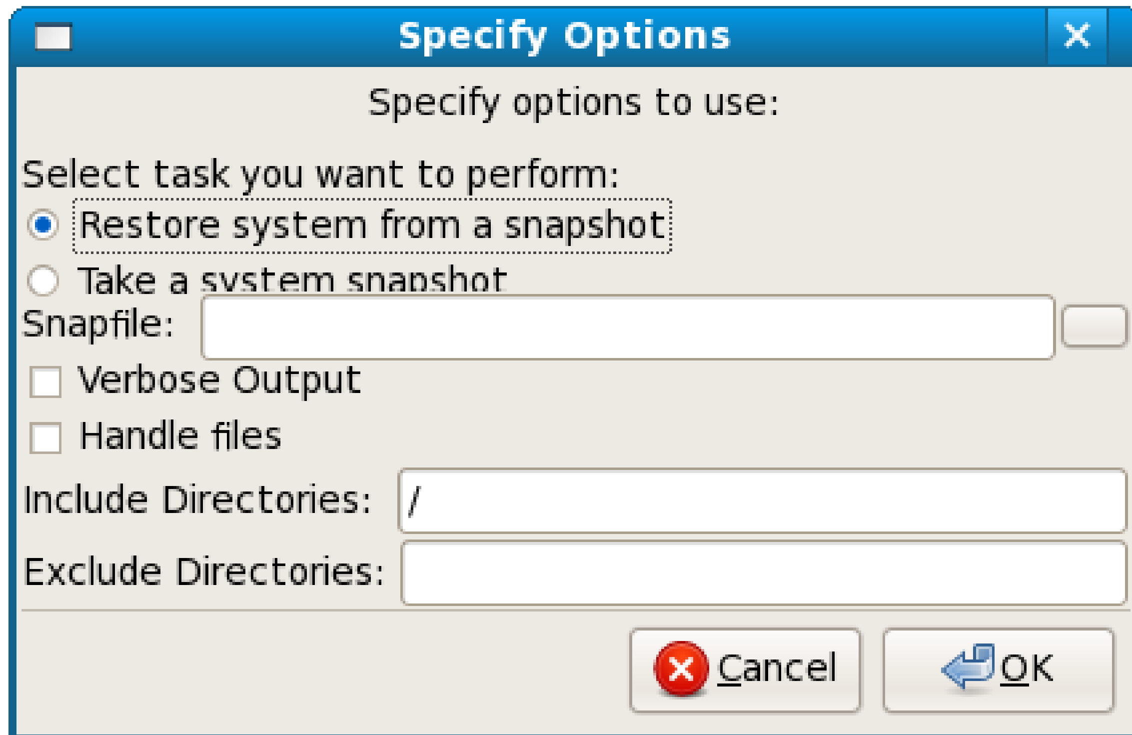

Figure 4. Snap Options Dialog

The main implementation of the graphical frontend is located in bin/gsnap. Initially we see the GSnap class, corresponding to the main window of the application which loads the glade resource and displays it. Upon clicking the 'Start' button, GSnap creates an instance of the dialog window class, GSnapOptionsDialog and displays that. After the dialog window is closed, SnapBase and the GSnapCallback are instantiated before another thread is launched to run the actual backup or restore operation. The alternate thread is used so that the main window can write output to the screen as it arrives via the callback. Any exceptions which snap throws are caught and ignored before the worker thread terminates, Thus the graphical client can be use Snap multiple times without termination.

The callback class simple inserts the text received into the main textbox buffer. This is a very simple. Theoretically a client can do anything with the callback message including internationalizing the text that is displayed, altering the behavior / flow of Snap or further transmitting the messages to a client of another protocol.

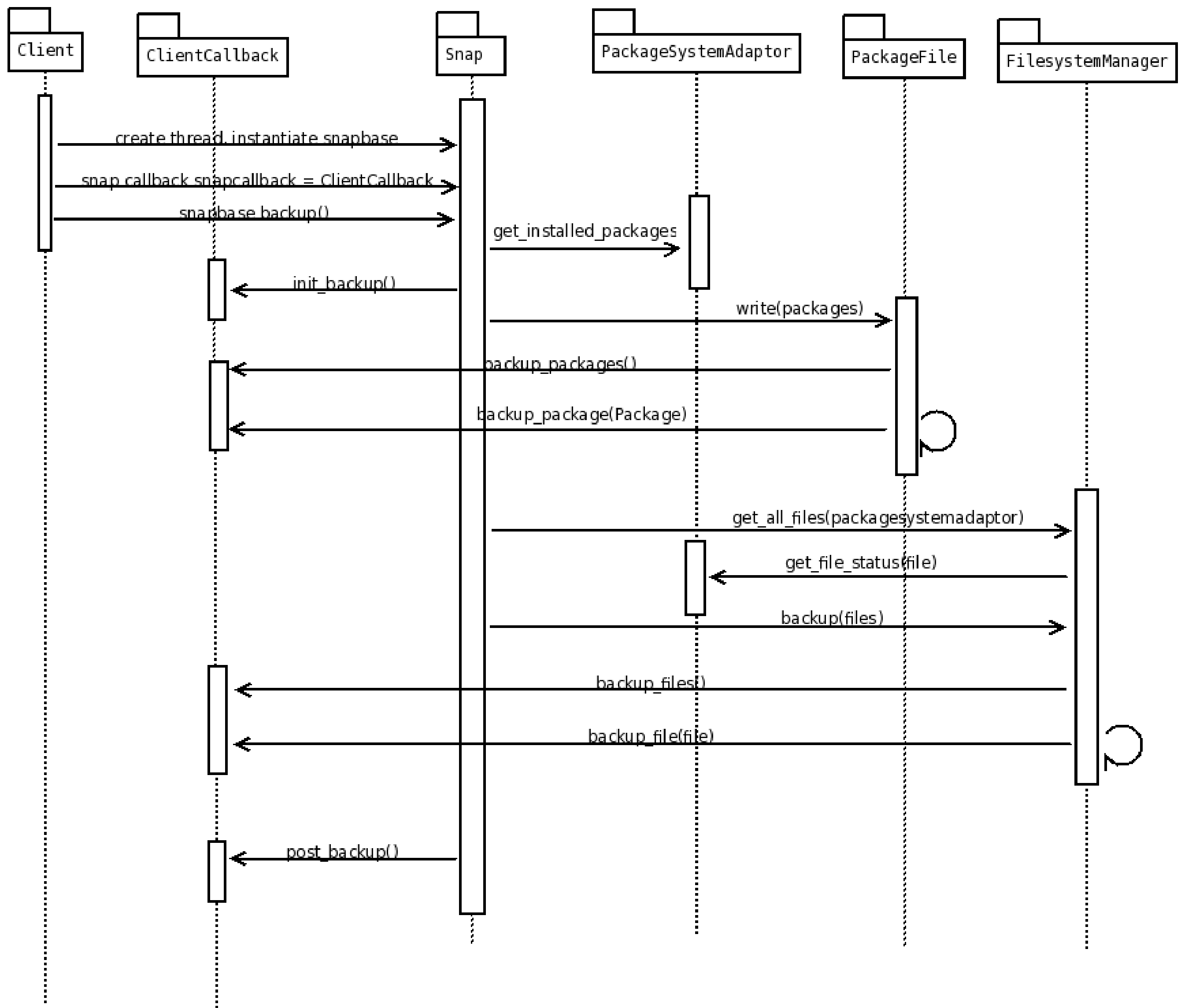

Figure 5. Snap Callback System Sequence Diagram for Backup Operation

Here we can see the Snap's callback system and the interaction with the client in its entirety. All client callbacks operate in the same fashion, including that of the console based tool in bin/snaptool. Though many of the internal Snap modules invoke the callback to relay information a client need only track *SnapBase, SnapCallbackBase, SFile, Package,* and *SnapFile*

IV. Testing

Test cases were developed during the implementation phase to target several critical points in the Snap library. Points which are most vulnerable to failure and needed to be regularly verified were identified and test suites were written to test the encompassing modules. Tests cover all of Snap's core implementation, including some limited plugin verification and excluding frontends. Plugins registered with the testing system are loaded and run, with output being analyzed via simple sanity checks. Fronted developers are encouraged to write their own test cases and fixtures, to ensure all operations succeed as intended, though this is out of scope for testing the core framework.

Test suites and cases are located in the 'tests/' directory. The standard Python unit testing module, pyunit, is employed to create test suites, each defined in their own module and test one Snap library module, and test cases, located inside the suites, each tests one specific critical point. Tests may be run by invoking *python test/run.py* which will load all the registered test suites and execute the test cases contained within. Additional test suites / cases can be created by simply extending the pyunit test interface, dropping the module in the tests/ directory, and registering it by adding it to test/run.py. All tests utilize the test callback interface, as defined in test/callback.py to catch / handle warnings and errors. The following is a list of critical points located in the system and test cases written to verify their integrity

1. Insufficient Permissions

The snaptest module is used to test the base Snap functionality as defined in the SnapBase class in the library initialization module, snap/__init__.py. Because SnapBase is fairly high level and abstract most operations here are mere callouts to lower level functionality and thus most unit tests are written for those modules. SnapBase does define and enforce the permission checking system to ensure the current process is running with superuser privileges. Because this is critical to the complete operation of Snap, a test case is written to ensure that if a non-superuser is running the application, it is terminated before core processing begins. The test case, defined in the tests/snaptest.py suite, ensures that the current process is not being run by the superuser before it attempts to invoke the backup / restore functionalities to ensure the Snap library exists gracefully with a snap.exceptions.SnapInsufficientPermissionError

2. Configuration verification

Snap requires a sane configuration set to work properly. Since this is completely user controlled, containing built in checks to ensure invalid option settings are not specified before operations begins. We need to ensure these checks are performed and executed properly so that a broken build will not do damage to a system by misinterpreting configuration options.

Currently the following sanity checks are performed by Snap and thus are tested by the test suite defined in test/configtest.py

- Ensure a mode of operation has been specified. If the client does not explicitly set SnabBase.options.mode to either ConfigOptions.BACKUP or ConfigOptions.RESTORE, Snap will throw a snap.exceptions.SnapArgError
- Ensure a target input snapfile is specified during the restore operation. If no output file is specified in backup, it is not a problem as Snap will use the default value /tmp/snapper (appended with the current time to prevent overwrites) as defined in the internal Snap options module snap/snapoptions.py. (the location of this file will be sent to the client via the callback interface). It is bad practice to assume the location of the input file during the restore operation, so Snap enforces the rule where the client must explicitly specify the input file, else a snap.exceptions.SnapArgError be thrown

Any future Snap developer is welcome to additional configuration options checking to the 'verify_options' methods defined in the ConfigManager class in snap/configmanager.py, as well as to the corresponding test module, so long as the added features are sane.

3. Plugin loading and instantiation

Loading an incorrect plugin or failing to instantiate one at all will result in a SnapPackageSystemError. This test suite, defined in test/psatest.py, ensures that an invalid plugin configuration is handled correctly and that all functionality can be successfully invoked. The test suite will attempt to load the package plugin proxy (PackageSystemAdapter as defined in snap/packagesystemadaptor.py) with invalid and valid package plugin names specified, evaluating the output, before calling each PackageSystemBase derived method. The method results are analyzed for basic validity and consistency. No tests performed are complicated or plugin specific, so Snap plugin developers are encouraged to write additional test suites of their own. Plugins that are used are specified via an array defined in the test module, and as additional plugins are developed, this list should grow to accommodate their testing.

Snap PackageSystemAdaptor Test Suite:

```
class PackageSystemAdaptorTest(unittest.TestCase):
    testplugins = [ 'yum' ]

    def testLoadInvalidPlugin(self):
        psa = SnapPackageSystemAdaptor
        self.assertRaises(SnapPackageSystemError, psa, 'foobar')

    def testInvokeInterface(self):
        for testplugin in self.testplugins:
```

```
psa = SnapPackageSystemAdaptor(testplugin)
installedpackages = psa.get_installed_packages()
for installedpkg in installedpackages:
    self.assertEqual(installedpkg.__class__, Package)
status = psa.get_file_status('/etc/passwd')
self.assert_(status == True or status == False)
```

4. Backup and Restore operations verification

This first of these tests, executes backup operation in its entirety, with a select set of inputs of packages and files to backup. The resulting Snapfile is then verified and its contents are parsed to ensure that all the specified targets were recorded. This second test fully runs the restoration operation, using a select snapfile as an input. The Snapfile is then parsed and the test ensures that all contents were restored to their proper locations. Both test suites are defined in test/snaptest.py and make use of the internal packages and files modules to verify output under the assumption the are tested elsewhere.

5. Generating and Parsing Packages and Files Record Files

Record files employed internally by snap are generated and parsed to keep track of critical system data. In order for Snap to operate correctly, these record files must be seamlessly and correctly translated to / from the corresponding lists of packages / files. Tests cases, are written to verify that this is the case for a variety of combinations. Defined in test/recordfilestest.py the package and file tracking systems are tested, using sample files and data found in the test/data directory. Internal Snap classes are used to parse and validate output, with each case covering a target component of the equation.

Test case verifying xml output from a backup operation on a list of files (abbreviated)

```
def testXmlFromFiles(self):
    data = [snap.files.SFile('file1', 'tmp/file1', True), snap.files.SFile('file2', 'tmp/subdir/file2', True) ]
    filemanager = snap.files.FileManager('/tmp/snaprecordfile.xml', '/tmp/snaptest/')
    filemanager.backup(data)
    dfiles,dpaths = [] , []
    parser = self._make_file_parser(dfiles)
    parser.parse('/tmp/snaprecordfile.xml')
    for fle in dfiles:
        dpaths.append(fle.currentpath)
    for file in data:
        if not file.currentpath in dpaths and not os.path.exists('/tmp/snaptest/' + file.currentpath):
            raise Exception # file not copied or no record in xml
```

V. Future Work

Alot can be done to improve and extend Snap. The most obvious is writing package management plugins and frontend for specific windowing systems, but alot of features can be added to the backside as well to further improve an administrators ability to efficiently perform and restore snapshots.

1. Remoting Interface

A wrapper client can be written for Snap providing a remoting interface to the backup / restoration functionality and the configuration / callback systems. Currently there are many remoting solutions, but an emerging standard, especially among 'Desktop' Linux systems, seems to be DBUS. Writing a DBUS wrapper would allow other applications to perform these operations as part of a bigger system maintenance process or in lieu of manual administrator operation.

2. Support for Remote Retrieval / Storage of Snapfiles

Often an administrator wants to schedule periodic backups of a system while storing the snapshot on another system for safety purposes. This would automatically connect to a remote server to store the snapshot upon backup and retrieve it upon restoring the system. Optimally the means which to transfer the file would be encoded in the URI in the config file, for example a URI of 'ftp://someserver.domain/home/someuser' would store / retrieve snapshots via ftp from the '/home/someuser' directory on the 'someserver.domain' server.

To help this task be realized, an incremental backup scheme, similar to that which rsync employs, could be implemented. That way when the system is scanned and the snapshot is being generated only new software and files not previously recorded would be stored. This would help ease storage and bandwidth requirements for the snapshots, and a system can be restored to any point in time using the snapshots up to that point. Of course the drawback of this is the reliance now on many files to be present, less the operation might not work. Regardless, if this were an option that the client can specify at some point, the optimum solution can be used for any situation at hand.

3. Support for a Snap Partition

Along the same lines of a remote storage solution, a local one can be employed as well for mere cases of local system corruption may be more cost effective. A partition can be setup on a local disk (preferably different than the one the operating system is running on) and regular incremental backups can be made to it. In the case of a disaster, a disk can be replaced the operating system can be reinstalled, and Snap used to restore the system to its previous working state. All three methods, local filesystem, local partition, and remote storage host can be unified into one snapfile storage system by means of URI specification for simplicity.

4. Support for Incompatible Files

Often as applications change, configuration options once valid become deprecated and then removed / unsupported. Currently Snap does not handle the verification that the files it is copying over will work with the latest software installed by the package management system. This is a challenging task to do right as configuration files vary with each software package they come with. Furthermore many distributions / operating systems alter the behavior of software and format of files for their own purposes. The only reasonable solution is to create an configuration file descriptor plugin system which Snap can use to determine if files are incompatible with software. These can employ timestamping, version checking, file parsing, or any other technique to determine the correct result. Unfortunately due to the large amount of software and system variations out there doing this alone will be hard, and ideally the community of individuals who write the software using the configuration file will help maintain the Snap plugin.

5. Additional Package-System-Specific Metadata

Mentioned in section *6.2*, package system specific metadata support should be added to support the needs of any package management system that requires additional data than what is currently stored. PackageSystemBase can be expanded so as to include methods which to use to retrieve package and file metadata, to store in the record files.

6. Additional Frontend Features

Porting Snap to various frontends such as KDE/Qt, XFCE, X, or that of any other graphical library is a relatively straightforward task following the aforementioned examples. In the future internationalization libraries should be employed and use to translate any text messages into the appropriate locale. Additional features such as graphical package selection and scheduling of automated backups / restorations

7. Transactions and Improved Failsafe Mechanisms

Snap internally provides a basic transaction and failsafe system used to handle and gracefully terminate operations after a critical error. Unfortunately currently there is no rollback mechanism which to revert changes already made before the error occurred. A system that that addresses this, would integrate with the package management system plugins to save and restore a “meta” system state, eg. the original system before the Snapfile restoration is attempted. Should the restoration fail, the package management system plugin would be invoked again to restore the system to that meta state, thus assuring a pristine system state at any point in time.

VI. Conclusion

Snap was built to provide a cross-platform solution to system software / file backup and restoration operations that uses the underlying package management system for simplicity and performance. In this paper we have outlined and discussed the motivations behind the development of the Snap project and discussed the target users and environment. System design is detailed, with all primary and seconds modules the Snap library provides as well as the package plugins and clients / frontends included. Critical points are identified and test suites / cases are written to harden those areas, and finally potential future work on the project is discussed.

This document corresponds to the 0.1 version release of Snap. All present functionality is stable and works as intended, though many established backup/restoration utilities with a long-history include several important features system administrators are looking for. As additional features are added, more package and windowing systems are supported, and as Snap becomes a recognized, reliable system administration and maintenance tool, the version will be incremented until the 1.0 release and beyond.

Appendix A. Source Code

All source code and related files maybe downloaded from the 'snapshotter' project on sourceforge. Provided is SVN access which a developer may utilize to retrieve the latest (potentially unstable) version of the code base. A developer simply needs to download / install SVN if he hasn't already, and run the following command,

```
svn co https://snapshotter.svn.sourceforge.net/svnroot/snapshotter
```

Furthermore, specific versions / builds will be archived and released, able to be accessed through the Sourceforge file management interface,. A client may use this to download tar/gzipped or zipped versions of the codebase, as well as any packages used to automatically install the Snap library and software application system wide. At the present time only a RPM version exists, to be installed on RPM or YUM based systems, though building additional packages is trivial once the file and directory layout is explained below.

All code, documentation (including this one), resources, and all other related files are released under the GNU GPL Version 2 (or later) as detailed here http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

1. Directory Layout

The following is a detailed list of all the directories and files the Snap project provides and their intended meaning / encompassing functionality. (note all files ending in the 'pyc' extension are temporary compiled modules used by Python to speed up processing if the original source has not changed, and thus can be ignored)

- bin/ - Snap client frontends, installed to the /usr/bin/ directory
  - gsnap – GTK frontend
  - setup.py – setup script used in installation
  - snaptool – console frontend
- docs/ - official documentation
  - man1/snap.man - man page, primary details accepted options
- packagesystems/ - package management system plugins go in here, gets installed to /usr/share/snap/packagesystems
  - devel-yumpackagesystem.py – experimental yum package system plugin in development, uses the yum interface as opposed to running the command
  - yumpackagesystem.py – currently the stable yum package system plugin
- resources/ - resources employed by Snap

snap.conf – Snap configuration file, installed to the /etc directory

snap.glade – glade resource file used by GTK frontend, installed to /usr/share/snap/. All other files in this directory are related to glade

snap/ - the Snap library, installed to the system Python library directory, usually /usr/lib/python2.5/site-packages/

callback.py – implementation of the base callback interface

configmanager.py – configuration options and management system

exceptions.py – exceptions thrown by snap

files.py – internal file management

__init__.py – primary Snap interface

packages.py – internal package management

packagesystemadaptor.py – snap package plugin system proxy

packagesystem.py – snap package plugin base class

snapoptions.py – internal snap options, change these if running from a build environment or other non-system-installed location

test/ - test suites, cases, fixtures, and data

data/ - fixtures and data employed by the test system

callback.py – test system callback implementation

configtest.py – configuration system test suite

psatest.py – package system test suite

recordfilestest.py – record files test suite

run.py – main test script

snaptest.py – snap base system test suite

README – general project info

TODO – a list of features that can be worked on

LICENSE – GNU GPL v2

Makefile – make file used in building Snap

2. Manually Building and Installing the Source

Snap may be built and installed simply by using the standard Unix 'make' buildsystem, specifically by running the following commands as a superuser:

make

make install

About the Author

Mohammed Morsi is a software engineer and full time student working on his Masters Degree in Computer Engineering at Syracuse University, in Syracuse, NY. Mohammed has several years of software development experience behind him, and is currently employed as a web developer at Red Hat.

Snap was originally started as a side project, intended to mitigate some of the tedious tasks required in upgrading between versions of the Fedora Linux distribution. Because the open source development model follows the 'release early, release often' principle, many distribution maintainers opt to provide a major release yearly or semiyearly (as in the case of Fedora). To maximize the security and stability on one's system, its advised to keep up with the current releases and thus manually reinstalling all software and migrating all configuration files periodically becomes a major task. Since snap has been written, restoring all software and files involves three steps, 1. run the backup command on the original system, 2. install the new version by whatever means and copy over the snapfile, 2. run the restore command on the new system.

Following the philosophy of open source and free software development, Mohammed encourages active use, modifications, enhancements, and patches for Snap. Any contributions will be accepted through his yahoo email address at movitto@yahoo.com