

2008

Cryptic Backup: a framework for automated compression, encryption, and backup of data

Christopher David Hoff
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Hoff, Christopher David, "Cryptic Backup: a framework for automated compression, encryption, and backup of data" (2008).
Retrospective Theses and Dissertations. 14942.
<https://lib.dr.iastate.edu/rtd/14942>

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

**Cryptic Backup: a framework for automated compression,
encryption, and backup of data**

by

Christopher David Hoff

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Co-majors: Information Assurance; Computer Engineering

Program of Study Committee:
Doug Jacobson, Major Professor
Tien Nguyen
Stephen Gilbert

Iowa State University

Ames, Iowa

2008

Copyright © Christopher David Hoff, 2008. All rights reserved.

UMI Number: 1453076

UMI[®]

UMI Microform 1453076

Copyright 2008 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF FIGURES	iv
LIST OF TABLES	vi
ABSTRACT	vii
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. RELATED TECHNOLOGY	3
<i>Open Source</i>	3
<i>Commercial</i>	5
CHAPTER 3. GOALS & ASSUMPTIONS	9
<i>Goals</i>	9
<i>Assumptions</i>	12
CHAPTER 4. ARCHITECTURE & DESIGN	15
<i>Programming Framework</i>	15
<i>Daemon</i>	23
<i>Client Application</i>	28
<i>Cryptic Backup Suite</i>	32
<i>Alternate Approaches</i>	33
CHAPTER 5. IMPLEMENTATION	39
<i>Daemon</i>	39
<i>Client Application</i>	53
CHAPTER 6. FUTURE WORK	66
<i>Improved Key Management</i>	66

<i>More Robust Change Tracking</i>	67
<i>Compression Suitability Evaluation</i>	68
<i>Safer Backup of Files Located on Mounted Volumes</i>	69
<i>Improved Delete Behavior</i>	70
<i>Bundle Support</i>	71
<i>Implement Watch Lists</i>	72
<i>User Documentation</i>	72
CHAPTER 7. CONCLUSION	75
APPENDIX A: CbackupdProtocol	76
APPENDIX B: Daemon database	77
APPENDIX C: Client application database	80
APPENDIX D: Setup Assistant window execution paths	82
APPENDIX E: Recovery Assistant sheet execution paths	85
APPENDIX F: Backup location interface states	88
REFERENCES	89
ACKNOWLEDGEMENTS	92

LIST OF FIGURES

Figure 4.1: Core Data Overview	20
Figure 4.2: Core Data Managed Object Model	21
Figure 4.3: Daemon Architecture	24
Figure 4.4: Thread-Safe Daemon Database Manipulation	27
Figure 4.5: Client Application Architecture	29
Figure 4.6: Cryptic Backup Suite Architecture	33
Figure 5.1: Setup Assistant — Welcome	54
Figure 5.2: Setup Assistant — Create passphrase	54
Figure 5.3: Setup Assistant — Save passphrase	55
Figure 5.4: Setup Assistant — Load passphrase	55
Figure 5.5: Setup Assistant — Recover settings	56
Figure 5.6: Main Window	57
Figure 5.7: Backup Plan — Name tab	60
Figure 5.8: Backup Plan — What tab	60
Figure 5.9: Backup Plan — When tab	61
Figure 5.10: Backup Plan — Where tab	62
Figure 5.11: Recovery Assistant — All files or a subset	63
Figure 5.12: Recovery Assistant — Restore to original location	63
Figure 5.13: Recovery Assistant — Restore to custom location	64
Figure 5.14: Recovery Assistant — Select files	65

Figure B.1: Daemon Managed Object Model	77
Figure C.1: Client Application Managed Object Model	80
Figure D.1: Setup Assistant execution — primary path	82
Figure D.2: Setup Assistant execution — alternate path — load key file	83
Figure D.3: Setup Assistant execution — alternate path — recall passphrase	84
Figure E.1: Recovery Assistant execution — primary path	85
Figure E.2: Recovery Assistant execution — alternate path — custom location	85
Figure E.3: Recovery Assistant execution — alternate path — specific files	86
Figure E.4: Recovery Assistant execution — alternate path — specific, custom location	87
Figure F.1: The five states of the backup location interface	88

LIST OF TABLES

Table B.1: Daemon database Client entity schema	77
Table B.2: Daemon database Plan entity schema	78
Table B.3: Daemon database Destination entity schema	78
Table B.4: Daemon database Source entity schema	79
Table B.5: Daemon database File entity schema	79
Table C.1: Client application database Plan entity schema	80
Table C.2: Client application database Destination entity schema	81
Table C.3: Client application database Source entity schema	81

ABSTRACT

Historically, creating backups of valuable computer data has been arduous, tedious, and time consuming for the average computer user. The increasing concern of encrypting sensitive information has added another layer of complexity to the process. Existing products, used in conjunction, can solve this problem; but this comes at greater expense and complexity. The challenge is creating a single product that encrypts and backs up data on an automated schedule. Likewise, it must decrypt and restore the data with ease. To specifically address this problem, the Cryptic Backup daemon was developed. The Cryptic Backup daemon schedules and automatically runs backup plans based on user preferences. Files marked for backup are compressed, encrypted, and copied to a user-specified location or remote server. The daemon is also responsible for reversing this process when recovery is needed. As its name implies, the Cryptic Backup daemon offers its services to all users of the system. To demonstrate the power of the daemon, the Cryptic Backup client application was developed. The client application provides a graphical interface, allowing the user to take advantage of the functionality the daemon provides. The result is a Cryptic Backup suite that is highly versatile, reliable, and more than capable of fulfilling the encrypted backup needs of the average computer user.

CHAPTER 1. INTRODUCTION

The number one sin of data backup software is not being able to restore backed up data when the user needs it. This problem typically arises from a failure at one of two key points in the process. The first point of failure is simply the backup application failing to backup the data. This could mean it never successfully backed up the data or that it missed a recently scheduled backup. The second point of failure is something going wrong during the recovery process. Examples of this include failure to locate the backed up data, and data corruption upon restoring the data.

Getting users to backup up their data is difficult. It's fairly easy to convince them that they should backup their data, but getting them to actually do it is another matter altogether. A recent U.S. consumer survey sponsored by Maxtor Corporation provides insight into consumer thought and behavior regarding backing up data. The survey revealed that:

“While most U.S. adults who have personal/professional data and digital information stored on a PC or laptop (89 percent) know they should back up their computer data, not many do it with any regularity [1].”

This statement reveals two important pieces of information. First, the vast majority of U.S. adults store personal or professional data on their computer. This not only implies that the data is valuable and should be backed up, but it also implies that data is sensitive in nature and should be protected, especially if it is backed up to a remote server. The second piece of information gleaned from this statement is the fact that most U.S. adults know they should

backup their data, but don't do it with any regularity. The next logical question is: why don't consumers regularly backup their data? The survey provided one rationale, stating that:

“Despite unaddressed fears of losing valuable information and digital memories, the most frequently-cited reason by those who said they never back up their data is that they are not sure how to do it or that backup is too technical (cited by 44 percent) [1].”

This survey reveals that consumers have identified technical hurdles as the primary reason they don't backup regularly. Of those that know about methods they can use to backup their data, many find them too complicated to be worth the effort. Unfortunately, the extra effort is often not seen as worthwhile until after a catastrophic (financially or emotionally) data loss.

Cryptic Backup aims to be an intuitive, reliable, and secure backup solution. The intuitive nature comes from the user-facing Cryptic Backup client application. It provides a graphical user interface that strives for the “set it and forget it” approach. Once the user has identified what they want backed up, they shouldn't have to think about Cryptic Backup again unless they need to recover data. The workhorse behind the client application is the Cryptic Backup daemon (background system process), which endeavors to reliably and securely backup user data. The daemon should never require user attention; rather it should work in the background, dutifully performing its tasks on behalf of the user. The following chapters outline the journey of Cryptic Backup from this vision to a reality.

CHAPTER 2. RELATED TECHNOLOGY

This chapter presents five technologies that provide functionality similar to the Cryptic Backup suite. Two of these technologies are open source projects, while the other three are commercial, closed source applications. A brief synopsis of each technology is provided along with an explanation of why it or its approach to backup was not directly used within Cryptic Backup.

Open Source

rsync

Rsync is a GNU-licensed (GPL) [2] command line based file copying tool. It is extremely versatile: it can copy files locally, to and from a remote server, or to and from a remote rsync daemon. A large component of rsync is its delta-transfer algorithm in which only the changed portions of a file are transferred. This has the potential to greatly reduce the amount of data that needs to be copied during a synchronize operation; especially with large files that only change a small amount each time. Rsync is widely used for backup and mirroring purposes and can be configured to transfer data over an encrypted channel [3].

On the face of it, it appears rsync could provide a great benefit to Cryptic Backup. However, there are a few significant issues. First, rsync has no facility internally for scheduling and running backups. On Unix- and Linux-like systems, scheduling is typically accomplished using the crontab utility [4]. Cryptic Backup could overcome this shortcoming by also using crontab or developing its own scheduler. Another problem with rsync is the lack of an intuitive graphical interface, or rather a graphical interface at all. Again, Cryptic

Backup could overcome this with its own graphical user interface to serve as a front-end for rsync.

The biggest problem with using rsync is the need for encrypted storage of backups. The advantage of the brilliant rsync delta-transfer algorithm quickly fades when encryption is brought into the mix. This algorithm works by comparing blocks of the local file with blocks of the remote file to determine which portions have changed and need to be transferred. With Cryptic Backup, the remote file is encrypted while the local file is not. Strong encryption means the remote file will look nothing like the local file, but when decrypted, they are identical. For rsync's delta-transfer algorithm to work in this case, the remote file would need to be decrypted, compared, modified as necessary, and re-encrypted. This requires tight interaction with the remote server (such as shell access) or excessive file transfers so the work can be done locally. Cryptic Backup can't assume the user has shell access to the remote server they use for storing their backups. Downloading the encrypted file, doing the work locally, and uploading the modified file is significantly more expensive than Cryptic Backup's method of checking for changes locally and re-uploading a newly encrypted file if changes are detected.

rsyncrypto

Rsyncrypto is a GNU-licensed (GPL) companion utility for rsync. It provides rsync friendly encryption by ensuring that local changes to the plaintext file will result in local changes to the cipher text file [5]. It is meant to be used to encrypt local files in preparation for using rsync to backup or mirror them remotely. In this way, files can be encrypted locally

and remotely with only a small impact on the efficiency of rsync's delta-transfer algorithm when changes are made locally.

Cryptic Backup could have used rsyncrypto for encrypting local files and a delta-transfer algorithm for uploading only the changes. This would have increased the efficiency of Cryptic Backup file transfers, but not without sacrificing some security. Although rsyncrypto uses AES encryption, it is forced to weaken the encryption to ensure localized plaintext changes result in localized cipher text changes. This is an unacceptable compromise for Cryptic Backup.

Commercial

Time Machine

Time Machine is an automatic backup utility built into Mac OS X. It became publicly available in October of 2007 when Apple released Mac OS X 10.5 Leopard. It touts the ability to “go back in time” to recover anything that has been backed up. After the user specifies where they want their backups stored, Time Machine runs a backup hourly, copying changed files to the backup location. By default, Time Machine backs up everything required to rebuild the user's system from scratch. Time Machine saves the hourly backups for the past 24 hours, daily backups for the past month, and weekly backups for everything older than a month. When the backup location runs out of available disk space, Time Machine starts deleting the oldest versions of the backup, always ensuring a full restore can be made if necessary [6].

Since Time Machine is a proprietary, closed-source application, it was never considered as a component of Cryptic Backup. However, it has similar functionality, so the differences between it and Cryptic Backup will be discussed. The greatest advantage Time Machine has over Cryptic Backup is its use of incremental backups. With Time Machine, a user can restore a file as it looked in a previous state. With Cryptic Backup, a user is limited to restoring the most recently backed up version of a file. Time Machine implements incremental backups elegantly using hard links [7]. A limitation of Time Machine is where backups can be stored. It requires a locally or network (via Apple Filing Protocol) connected volume formatted using the HFS+ filesystem. Network connected backup store locations are required to be running Mac OS X Leopard. In contrast, Cryptic Backup is filesystem independent and allows for backups locally or over several file-transfer protocols. Another limitation of Time Machine is that it doesn't create encrypted backups. If the source file is encrypted, it will back it up without a problem, but it doesn't encrypt anything on its own. To summarize, Time Machine creates complete incremental backups, but doesn't use encryption and limits where backups can be stored.

Backup.app

Backup.app is a backup utility made by Apple for Mac OS X [8]. The latest major release occurred in September of 2005 with version 3. The most recent minor release came in March of 2007 with version 3.1.2. Backup.app preceded Time Machine by several years. Now that Apple has released Time Machine, it is unclear if Apple will continue development of Backup.app. It comes with preset backup plans configured to backup music, movies, personal settings, and the user's entire home directory. Custom backup plans can also be

created. Backups are scheduled and automatically run. They can be stored locally, on the user's iDisk [9], or burned to a CD or DVD.

Like Time Machine, Backup.app is a proprietary, closed source application, so it was not considered as a component of Cryptic Backup. However, Backup.app and Cryptic Backup are similar in several ways. They both have the concept of backup plans that run on a user-defined schedule. Both applications allow the user to store their backup locally or on their iDisk. Backup.app also allows for burning to a CD or DVD, while Cryptic Backup allows backups to be stored on Amazon S3 [10] or an SFTP or secure WebDAV server. The biggest difference between the two applications is that Cryptic Backup creates encrypted backups while Backup.app does not.

SuperDuper!

SuperDuper! rounds out the list of related technologies. It is a shareware backup utility created by Shirt Pocket [11]. SuperDuper! refers to its backup plans as scripts. It includes several scripts by default, including a script for backing up all files and a script for backing up just the user's files. Custom scripts can be created to backup specific files. Like backup plans, SuperDuper! scripts can be scheduled to run automatically. Backed up data can be stored on a local volume or a mounted network disk.

SuperDuper! was not considered as a component of Cryptic Backup due to its proprietary, closed source nature. Like Backup.app, SuperDuper! is similar to Cryptic Backup. All three have the concept of backup plans, but SuperDuper! refers to them as scripts. SuperDuper! has two key limitations. First, backups can only be stored locally or on a mounted network share. As previously stated, Cryptic Backup can use Amazon S3, iDisk,

local or network mounted volumes, SFTP, and secure WebDAV to store backups. The second major shortcoming of SuperDuper! is its inability to create encrypted backups. As with Time Machine, it will backup already encrypted files, but it doesn't encrypt anything on its own.

CHAPTER 3. GOALS & ASSUMPTIONS

Before designing Cryptic Backup, goals and assumptions were defined that served as a guide for the design process. This chapter outlines goals for Cryptic Backup as a whole as well as for the two components: the daemon and the client application. Assumptions made before and during the design process are also noted in this chapter.

Goals

As a backup application, the primary goal of Cryptic Backup is reliable data recovery. The last thing a backup application should do is fail when the user needs it most: during recovery. For recovery to be successful, the backup application must correctly execute many steps, both during the backup process and during the recovery process. Data availability and integrity are important sub-goals of reliable data recovery.

Along with the goal of reliable data recovery, Cryptic Backup should strive to protect the backed up data in transit and at rest at its destination. Users may not have total control over the communication channels their data backups travel on, so it is important for Cryptic Backup to respect the possibly sensitive nature of this data and protect it from eavesdropping. Similarly, users may not have total control over the system their backed up data is stored on. Users that are conscious about the importance of offsite backups often purchase storage space from a hosting company. While this storage space may be protected with a username and password or some other authentication mechanism, that doesn't guarantee an unscrupulous employee won't take a peak at the data. Additionally, the threat of an outside attacker

gaining unauthorized access is always present. To allay these concerns, Cryptic Backup should ensure that data at rest is comprehensible only to the owner of the data.

As discussed previously, it is difficult to get users to backup their data. Because of this reality, another goal of Cryptic Backup is to make the process of creating backups as intuitive and painless as possible. It's hard enough to get the average user to backup their data without requiring them to edit obscure configuration files. For this reason and others explained in Chapter 4, Cryptic Backup will be divided into two components: a system-level daemon and user-level graphical client application. The specific goals of these two components are defined in the next two sections.

Daemon

The Cryptic Backup daemon should assume the responsibility of fulfilling most of the previously stated goals. It needs to be responsible for scheduling and automatically running the user-created backup plans. The task of running backup plans has many sub-tasks including compressing, encrypting, and copying the data to the user-selected storage location. Since the daemon could be processing thousands of files in a single backup plan, performance is another important goal.

On the flip side, the daemon should also be responsible for restoring files from a backup at the request of the user. Like the backup process, the restore process has several sub-tasks including copying the data from the storage location, decrypting it, and uncompressing it. This process has the potential to be faster than the initial backup process due to the typical disparity in upload and download connection speeds to the Internet.

However, in this case, the situation is likely more urgent for the user, so performance is again an important goal.

Throughout the backup and restore operations, the daemon should ensure the availability and integrity of the data to the extent that it can. The daemon obviously can't be held accountable if the backed up data becomes unavailable, or if it is corrupted or deleted by an external person or program. To preserve data integrity, the daemon shall handle user data in a consistent and reliable manner.

Another goal the daemon should handle is encryption key management. To fulfill the previously stated goal of protecting user information in transit and at rest, the daemon should use encryption with a user-specific key. This requires a key management system that facilitates the creation, storage, and recovery of encryption keys.

Finally, the Cryptic Backup daemon must offer its services to all users of the system. Perhaps more importantly, it must act on the user's behalf, even while they aren't logged in. A scheduled backup should never be missed simply because the user isn't currently logged in to the system.

Client Application

The primary goal of the Cryptic Backup client application is to provide an interface that allows the user to harness the complex functionality provided by the Cryptic Backup daemon. All interfaces to the client application should be intuitive and graphical in nature. On first run, it should present a graphical interface for creating and saving a key file or for recovering data from a backup. Once fully launched, the client application should display the

backup plans that have been created with details about each one such as if it is enabled and when it will run next.

The client application should also allow the user to create a backup plan by specifying a name, what should be backed up, when it should be backed up, and where it should be backed up to. Conversely, the client application should allow the user to delete a backup plan. Similar to creating a backup plan, the client application should allow the user to edit all the details that define a backup plan, including the name assigned to it. While backup plans are running and after they've completed, the client application needs to provide status information to the user on a per-plan basis. Finally, the client application should allow the user to initiate a backup or recovery procedure on demand. This is an obvious requirement for recovery, but maybe not as much for backup. If the a user has modified a file and wants it backed up immediately, they should have that option without going to great lengths to create a new backup plan or rescheduling a current one.

Assumptions

During the design phase, some assumptions were made. Since assumptions are essentially admitted weaknesses in the design, as few as possible were allowed to pass through. Of those that did, many are addressed in Chapter 6.

An assumption made by the client application is that the daemon is always running and available. The client application requires frequent communication with the daemon and cannot function correctly without it. If the client application can't locate the daemon on startup, it will notify the user and quit.

Both the daemon and client application assume no external program or person has tampered with their respective database files or the backed up data. There is no allowance for data owned or created by Cryptic Backup being deleted, corrupted, or otherwise modified. The result of this occurring is unpredictable.

When running a backup plan, the daemon makes several assumptions. First, it assumes a file needs to be backed up if, and only if, its last modified date has changed. If the daemon can't find a file at a path it previously knew about, it assumes the file has been deleted and removes it from the backup store. The one exception to this is if the daemon determines the file was on a mounted volume that is not currently mounted. The daemon also assumes the user wants the old version of a file in the backup store to be overwritten with the most recent version. When copying files to the backup store, the daemon assumes the remote file system supports large files. Nearly all modern filesystems do, but older filesystems may not have adequate maximum file size limits. For example, the FAT32 filesystem limits files to 4 gigabytes minus 1 byte [12]. Several types of software can easily exceed this limit, such as applications that create or edit video. Modern filesystems typically measure maximum file size limits in exabytes (1 billion gigabytes).

During a restore or recovery operation, the daemon overwrites files as necessary. The user is given the opportunity to select a "clean" restore location, but if they don't, files are overwritten without question.

The most dire assumption made by the Cryptic Backup suite is that in a disaster recovery scenario, the user either has a backup of their key file or can remember the passphrase used to create their key. Without this crucial piece of data, decryption, and thus

recovery cannot happen: there is no backdoor. A scenario where this could lead to trouble is if the user's hard drive or entire computer is stolen or somehow the data on the hard drive becomes unreadable. To get a copy of their data back, the user will want to recover from their backup. But, if their key file only existed on the now unavailable hard drive and they don't remember their passphrase, nothing can be done. In an attempt to mitigate this problem, when the daemon creates a key for a user, it also creates a key file for backup purposes and instructs the client application to force the user to specify where they'd like the key file saved. However, nothing stops the user from only storing this key file on the local hard drive or deleting it manually later. Another complication in this scenario is the loss of the database that contains information mapping local file paths to remote encrypted files with masked filenames. An encrypted copy of this database information is sent to all backup locations for a specific user when one of their backup plans is run, but the user must be able to tell Cryptic Backup where their backups are stored. Without this database information, the daemon cannot recover the user's data. Possible solutions to the problems of a user losing their key file or forgetting their passphrase are discussed in Chapter 6.

CHAPTER 4. ARCHITECTURE & DESIGN

The Cryptic Backup daemon and client application utilize several external frameworks and libraries. These range from frameworks based on open source projects to proprietary APIs and frameworks provided by Apple Inc. The current architecture of the Cryptic Backup suite is a result of several iterations and refinements. Careful consideration was given to the techniques and technologies used, while continually keeping the goals in Chapter 3 in mind.

Programming Framework

The two components of Cryptic Backup are Cocoa applications written in Objective-C using Apple's Mac OS X 10.5 SDK (Leopard). The choice to develop Cryptic Backup on the Mac was driven by two factors. First, an encrypted backup solution for the average (Macintosh) computer user is sorely needed. Second, the APIs provided by Apple and generous third parties for the Mac are impressive in their breadth and depth. In the words of Apple:

“Cocoa is Apple's name for the collection of frameworks, APIs, and accompanying runtimes that make up the development layer of Mac OS X. By developing with the Cocoa frameworks you will be writing applications the same way that Mac OS X itself is written, with complete access to the full power of the operating system, including the signature Mac look and feel. Cocoa is simply the best way to create native Mac applications [13].”

In addition to proprietary Apple APIs, Cryptic Backup relies heavily on the OpenSSL crypto library, libcrypto [14]. This section discusses the main frameworks, libraries, and APIs used in the design and development of the Cryptic Backup suite.

Foundation.framework

The Foundation framework is part of Apple's Mac OS X SDK. It is a library of Objective-C classes that provide the infrastructure for object-based applications [15]. The Foundation framework operates below the graphical layer and, as its name implies, provides common functionality used by every Cocoa application. This includes classes for data structures such as arrays, sets, and dictionaries. Other classes facilitate networking, parsing XML documents, performing affine transforms, and handling Unicode strings. The components of Cryptic Backup make heavy use of the Foundation framework. This is mostly for accomplishing rather mundane tasks, but there are a few interesting classes that perform critical operations for the Cryptic Backup suite. A select few will be highlighted below.

The Cryptic Backup daemon uses the `NSDate` class to assist in determining when a backup plan should run next, based on the last time it ran. `NSDate` is a public subclass of `NSDate`. It performs computations based on the Gregorian calendar. An `NSDate` object instance represents an actual calendar date and time, accounting for time zones. It is precise to one second. Date and time calculations are complicated and hard to get right, especially when considering time zones, leap years, and different calendar formats. For this reason, it was an easy choice to use `NSDate` in Cryptic Backup.

Both components of the Cryptic Backup suite make heavy use of the `NSConnection` class. This class manages communication between objects in different threads or processes. This communication can be local to a single system or to a remote system. In the case of the Cryptic Backup suite, all communication using `NSConnection` is local. The daemon and client application use `NSConnection` to communicate with each other using the distributed objects mechanism. This communication adheres to the protocol in Appendix A, which will be explained in further detail in Chapter 5. Using distributed objects is a very convenient mechanism for communicating between processes. The same results could have been achieved using raw sockets, but such an implementation would have been far less elegant and more susceptible to problems.

The `NSKeyedArchiver` class encodes objects into a format that can later be used to reconstruct or decode the object using the `NSKeyedUnarchiver` class. Any class that wishes to be encoded and decoded must conform to the `NSCoding` protocol. The class is required to implement two methods: one that instructs how to encode the class and another that instructs how to decode the class. In the case of encoding, the class must encode all instance variables that are required to reconstruct the class later. All instance variable classes must also conform to the `NSCoding` protocol so they can encode and decode themselves. This process is continued on down the line as necessary. The Cryptic Backup suite uses the functionality provided by `NSKeyedArchiver` and `NSKeyedUnarchiver` in three key places. First, complex objects that must be sent between the daemon and client application are encoded before being sent and decoded when received. Another use for these classes is when complex objects need to be stored in the database for later retrieval. Finally, the daemon uses these classes when

writing or restoring from a user key file. Because of the cascading nature and tight interdependence of encoding and decoding objects, developing a custom mechanism for this purpose was not desirable or feasible.

A major component of the Foundation framework used by the Cryptic Backup daemon is the `NSOperation` class and associated `NSOperationQueue` class. `NSOperation` is based on `NSThread`, which itself is based on POSIX threads (pthreads) [16]. `NSOperation` takes care of most of the threading details and mitigates the expense of creating pthreads by recycling them behind the scenes. The daemon uses `NSOperations` for the sub-tasks in the backup and restore processes. This involves instantiating an `NSOperation` subclass and adding it to an instance of `NSOperationQueue`. The simplicity of creating `NSOperations`, along with the performance benefits, made using them for the daemon's threading needs an obvious choice.

The `NSProtocolChecker` class is an important security measure employed by the Cryptic Backup daemon. The primary daemon class implements the protocol used by the client application to communicate with it. However, this class has additional public methods exposed that are only meant to be used by other daemon classes. The `NSProtocolChecker` serves as a filter to only allow methods within the defined protocol to be called remotely.

As previously mentioned, one of the daemon's responsibilities is to schedule and run backup plans. This is accomplished using the `NSTimer` class. The daemon sets a timer to go off at the calculated next run time for a plan. When the timer fires, the daemon knows it needs to run that backup plan.

AppKit.framework

The Application framework (usually referred to as AppKit) is also part of Apple's Mac OS X SDK. It is one of the core Cocoa frameworks. AppKit is primarily concerned with the graphical user interface (GUI) of applications. This includes event-handling mechanisms and drawing facilities [17]. Since the Cryptic Backup daemon has no graphical interface, it does not use AppKit. However, the client application makes heavy use of AppKit. It relies on the alerts, animations, buttons, controllers, date pickers, fonts, images, menus, open and save windows, progress indicators, text fields, and tables provided by AppKit to create a functional graphical user interface.

There wasn't much deliberation involved in selecting AppKit. The choice was made when it was decided the Cryptic Backup suite would run exclusively on Mac OS X 10.5. Developing applications on the Mac with a native Mac GUI requires the use of AppKit.

CoreData.framework

Like the previous two frameworks, the Core Data framework is part of Apple's Mac OS X SDK. It provides object graph management and persistence for Cocoa applications [18]. An object graph is a collection of objects, including the relationships between them. Core Data consists of managed objects (NSManagedObject) that live in a managed object context (NSManagedObjectContext) at run-time. Most of the functionality of Core Data is exposed through the NSManagedObjectContext class. It allows for creating, editing, deleting, and fetching objects from the persistent store.

To maintain persistence while the application is not running, Core Data uses a persistent store coordinator (NSPersistentStoreCoordinator). The persistent store coordinator

contains the managed object model definition (NSManagedObjectContext) and provides access to the on-disk file representation of the database. A high-level view of the Core Data system can be seen below in Figure 4.1.

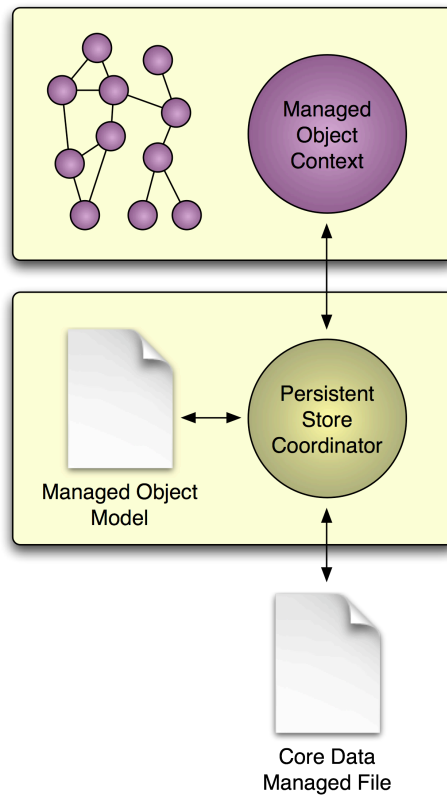


Figure 4.1: Core Data Overview

The managed object model is defined by the programmer. It consists of entities, attributes, and relationships. An example of a managed object data model from a recipes application can be seen in Figure 5.1. Each box in the figure represents an entity. Within each box, attributes and relationships for the entity are defined. A managed object is a unique instance of a specific entity in the database.

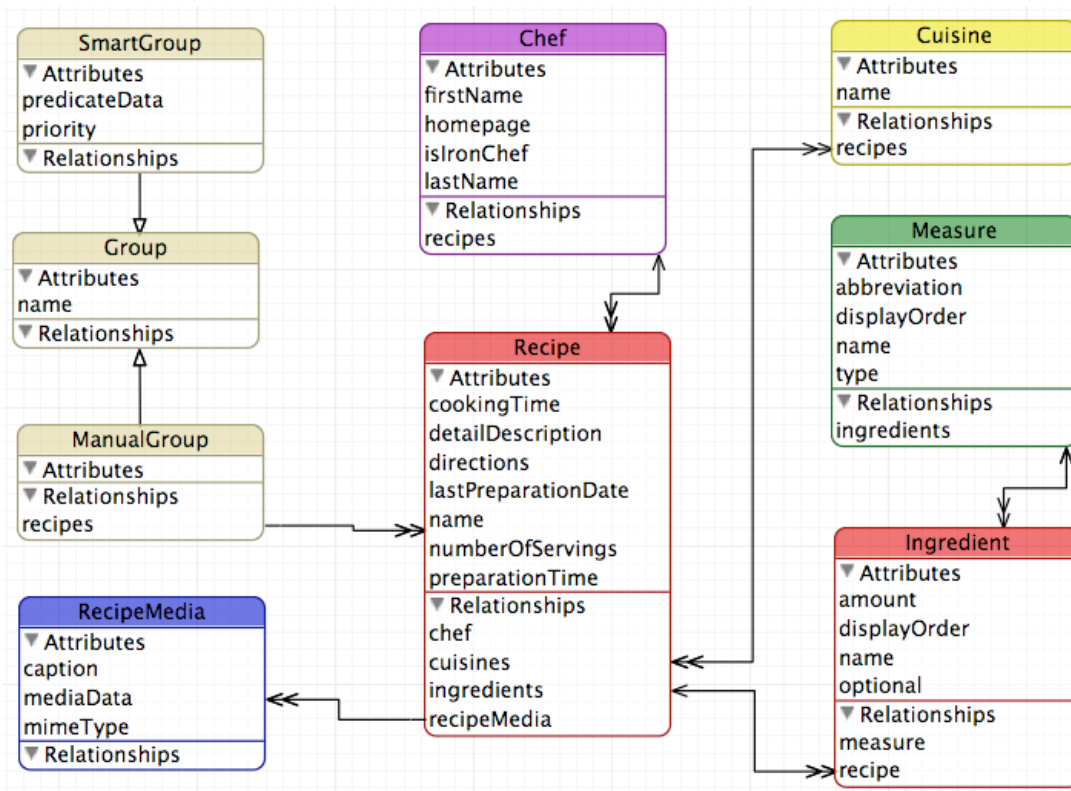


Figure 4.2: Core Data Managed Object Model

Both components of Cryptic Backup use the Core Data framework. The daemon maintains a database containing information about clients, backup plans, backup destinations, files on the local system, and files in backup stores. The client application maintains a database that mirrors portions of the daemon database. Because information in the client application's database is used to populate the user interface elements, having this data local to the client application is necessary for performance reasons.

Core Data was introduced by Apple in Mac OS X 10.4 Tiger. Since then, developers have been learning what Core Data is and what it isn't. Core Data is not a relational database or relational database management system. It also does not remove the need to write code.

Core Data is a fast infrastructure for object graph management and reliable persistence [19].

Because of this, Core Data suits the Cryptic Backup daemon and client application well.

SSCrypto.framework & libcrypto

The SSCrypto framework [20] is an open source project that provides an Objective-C wrapper around the OpenSSL libcrypto and libssl libraries [14]. The Cryptic Backup daemon uses the SSCrypto framework for creating hashes, generating encryption keys, and performing symmetric encryption and decryption operations. The workhorse behind these operations is the OpenSSL libcrypto library. OpenSSL is licensed under an Apache-style license, which essentially means it is free to use in commercial and noncommercial applications, following a few license conditions. Similarly, the SSCrypto framework is licensed under a BSD-style license, which allows for redistribution in source and binary forms, with or without modification. In the case of the SSCrypto framework, slight modifications were made to add support for the SHA-256 hash function [21].

Very little consideration was given to creating a custom implementation of the necessary hash functions and encryption algorithms. This would be a huge undertaking and is far beyond the scope of this work. Using the SSCrypto framework was entirely optional, but it allowed for a more seamless integration with the Cryptic Backup daemon.

Connection.framework

The Connection framework (often referred to as the Connection Kit) is an open source project that aims to make connecting to remote file servers easy, using popular file transfer protocols [22]. Several prominent Mac shareware applications, such as Sandvox [23] and ProfCast [24], use the Connection Kit for their file transfer needs. Currently, the

Connection Kit supports Amazon S3, FTP, SFTP, WebDAV, and secure WebDAV. The Cryptic Backup suite limits the user to using only secure protocols. Since the daemon encrypts the data before transfer anyway, secure transfer of data is not an issue. The problem is with protocols, such as FTP and WebDAV, that send login credentials over unencrypted channels. The Connection Kit is also licensed under a BSD-style license where redistribution in source and binary forms, with or without modification, is permitted, following a few conditions.

Some consideration was given to implementing clients of the necessary file transfer protocols. Ultimately it was decided that the Connection Kit fills this role well.

Daemon

The architecture of the Cryptic Backup daemon is divided into four distinct layers. Layer 0 forms the foundation for the daemon and is where all the frameworks are located. Layer 1 contains the primary daemon class and a controller class for the database. The classes that manage backups and restores make up Layer 2. Layer 3 contains the backup pipeline and the restore pipeline. A diagram of this architecture can be seen in Figure 4.3.

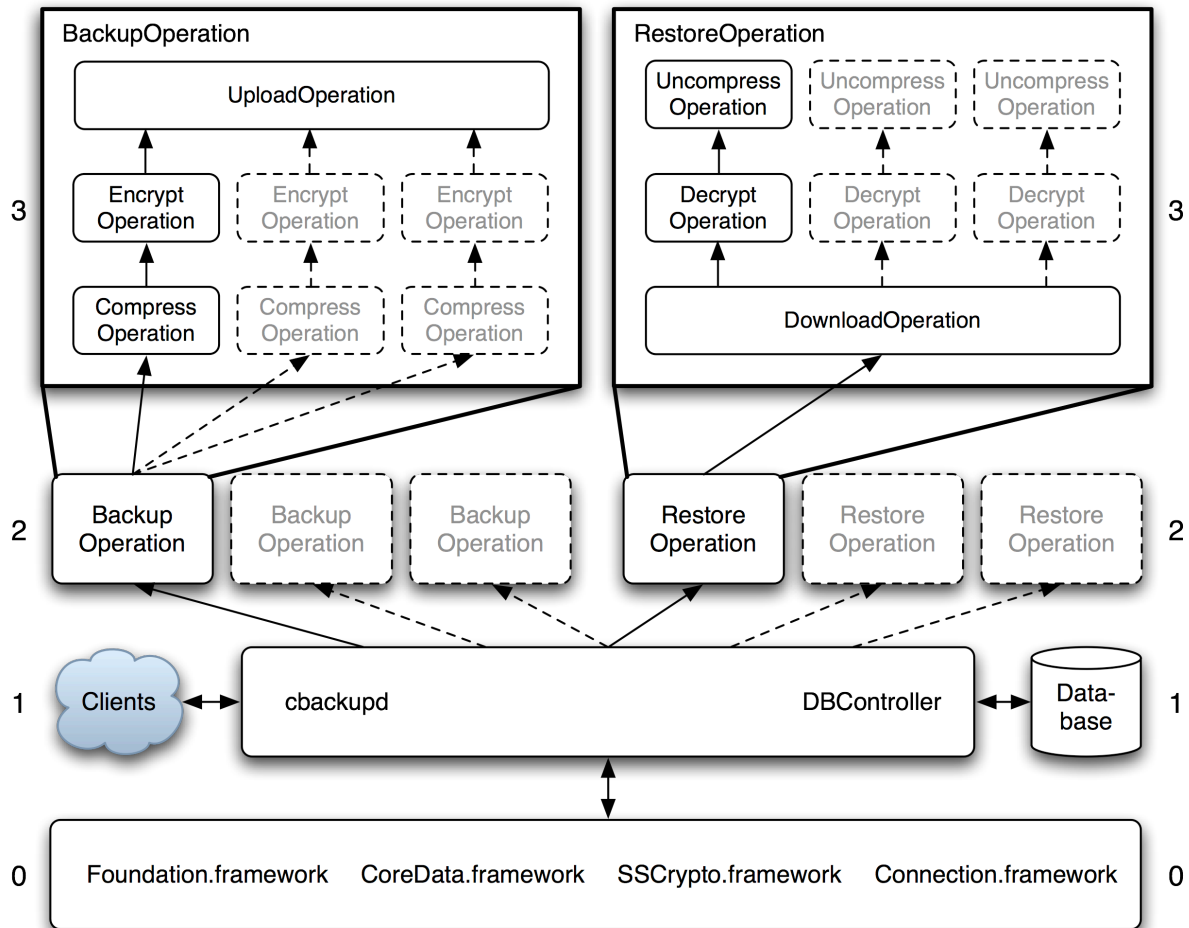


Figure 4.3: Daemon Architecture

Layer 0

Layer 0 is the foundation upon which the Cryptic Backup daemon is built. It contains proprietary Apple frameworks such as Foundation and CoreData that provide most of the basic functionality the daemon needs to function. Additionally, these Apple frameworks provide mechanisms for multithreading, persistent data storage, and distributed objects, which the daemon uses for communicating with clients.

Layer 0 also contains the third-party frameworks SS Crypto and Connection. These frameworks themselves are built on top of several Apple frameworks, but are still foundational to the Cryptic Backup daemon and thus are in Layer 0 from its perspective. As previously stated, the daemon uses SS Crypto as a gateway to the OpenSSL libcrypto library for generating hashes and performing encryption and decryption. The daemon uses SHA-256 hashes [21] and AES-256 encryption [25]. The level of security offered by these algorithms might be considered excessive, but using them made sense when considering the minimal performance impact over hashes with a shorter digest (such as SHA-1) and encryption systems with shorter keys (such as AES-128).

The Connection framework in Layer 0 is used by the daemon to interface with Amazon S3, .Mac (iDisk), SFTP, and secure WebDAV. These, along with local and network mounted volumes, comprise the locations and protocols a user can use to store their encrypted backups. The Connection framework contains an AbstractConnection class that presents a consistent API regardless of the underlying protocol. With the Connection framework handling authentication and file transfers, the daemon simply provides the location, credentials, and the data that needs to be transferred.

Layer 1

Building on top of Layer 0 is Layer 1 with the two core daemon classes. The first is cbackupd, which implements the protocol used to communicate with client applications. The cbackupd class is also responsible for scheduling backup plans and responding when a plan needs to be run or restored. To remain responsive to a possible multitude of clients, cbackupd quickly delegates work to Layer 2.

The second core daemon class is DBController. This class provides methods other parts of the daemon use to interface with the database. Typical actions include adding, removing, and editing database objects. The DBController class also provides fetch capabilities, which is synonymous to a SQL SELECT statement.

Layer 2

Layer 2 of the daemon contains two classes, BackupOperation and RestoreOperation. These are NSOperation subclasses that manage backups and restores, respectively.

Whenever cbackupd in Layer 1 determines a backup or restore needs to be performed, it spawns a BackupOperation or RestoreOperation. The NSOperation implementation in Apple's Foundation framework dynamically determines how many operations should run concurrently based on system resource availability. As the classes in Layer 2 find work to do, they pass it off to Layer 3, and continue to search for more work.

Each instance of BackupOperation requires access to the daemon's database. This means multiple threads need access to the same database concurrently. Core Data is not inherently thread-safe, so this becomes problematic. Referring back to Figure 4.1, the DBController class in Layer 1 contains an instance of the NSManagedObjectContext class linked to the NSPersistentStoreCoordinator class for the daemon's database. The persistent store coordinator is thread-safe. The component that isn't thread-safe is the managed object context. A managed object context can be thought of as an in-memory representation of the database that periodically synchronizes with the persistent store coordinator to permanently commit changes. It's essentially a scratch space where changes can be made and easily discarded if desired. To prevent database corruption, each instance of BackupOperation

needs its own managed object context linked to the shared persistent store coordinator. Before a BackupOperation completes, it saves the changes made in its managed object context. This commits changes to the persistent store. However, the managed object context in DBController is not aware of the changes that were made. Therefore, the BackupOperation must merge the changes made in its local managed object context with DBController's managed object context. By doing this, the managed object context in DBController is now synchronized with the persistent store. A diagram of this interaction can be seen in Figure 4.4.

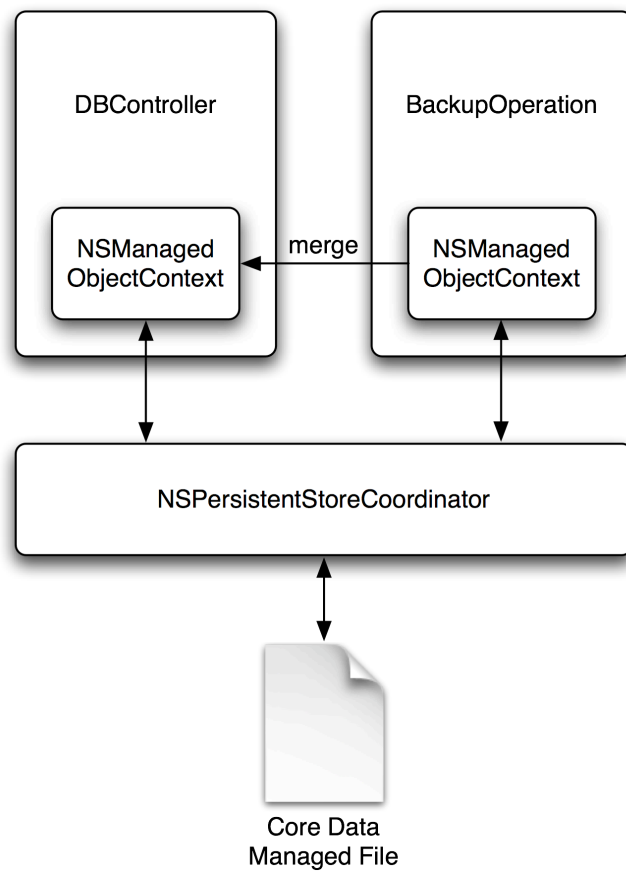


Figure 4.4: Thread-Safe Daemon Database Manipulation

Layer 3

The top layer of the Cryptic Backup daemon stack contains a pipeline for backup operations and a pipeline for restore operations. Much like Layer 2, additional operations are created as needed in Layer 3. For example, when BackupOperation finds a file that needs to be backed up, it creates a CompressOperation and EncryptOperation to prepare the file for uploading. Similarly, when RestoreOperation completes a download, it creates a DecryptOperation and UncompressOperation to complete the processing of the file. The daemon does not set a limit on the number of concurrent operations, however, as discussed in Layer 2, the implementation of NSOperation will prevent too many operations from running concurrently.

Client Application

The architecture of the Cryptic Backup client application also has four layers, but they are very different from the daemon, especially Layers 2 and 3. Layer 0 consists of the frameworks that form the API foundation for the client application. Layer 1 contains the database controller class and a reference object for the daemon. Controlling classes for the client application's graphical interfaces comprise Layer 2. Finally, Layer 3 contains the actual windows, buttons, etc., that make up the client application's graphical interface. A diagram of this architecture can be seen in Figure 4.5.

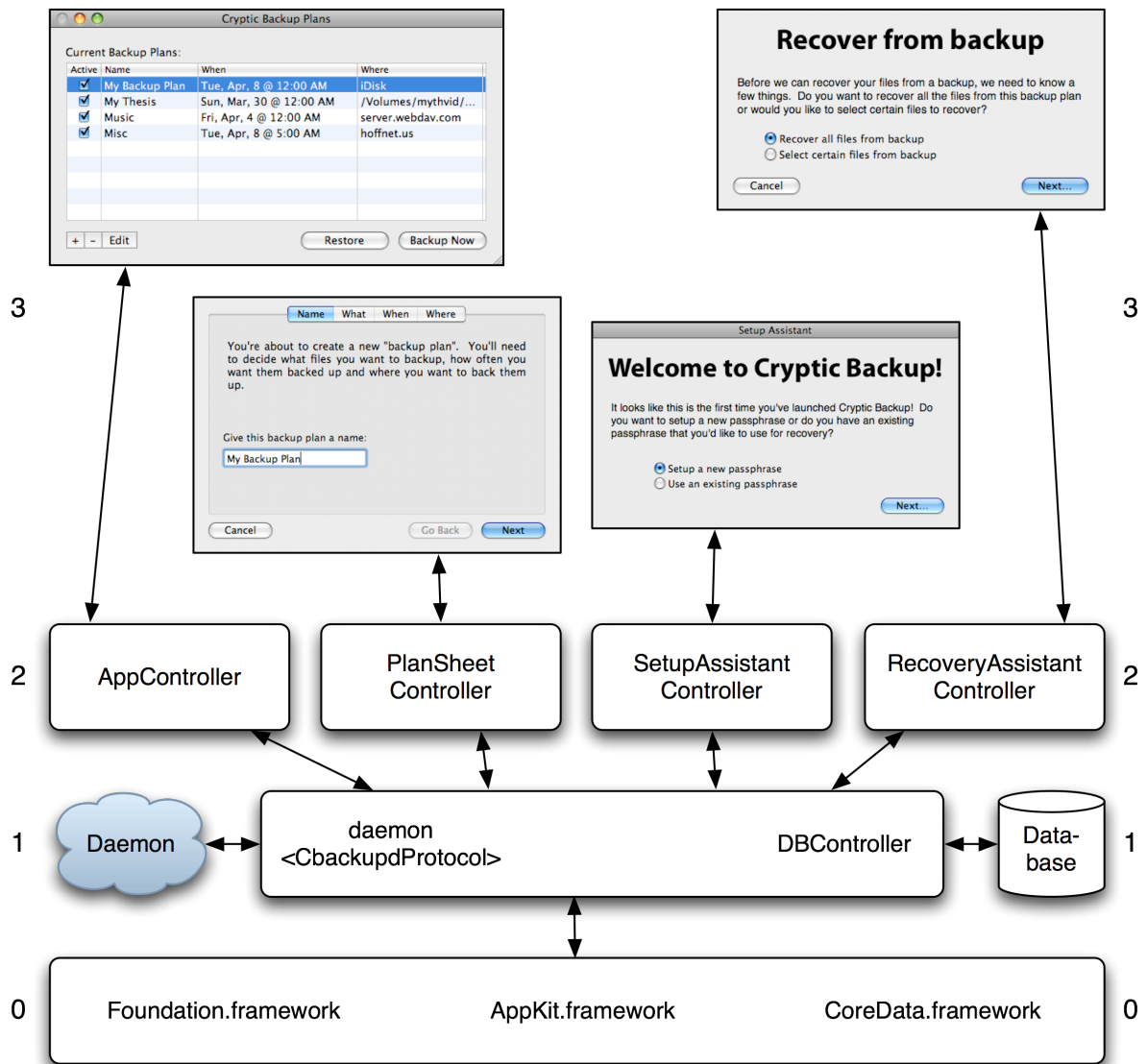


Figure 4.5: Client Application Architecture

Layer 0

Layer 0 forms the foundation for the Cryptic Backup client application. It is composed of proprietary Apple frameworks such as Foundation, AppKit, and CoreData. These frameworks provide most of the basic functionality the client application needs to

function. Additionally, these Apple frameworks provide mechanisms for presenting graphical user interfaces, persistent data storage, and distributed objects, which the client application uses for communicating with the daemon.

Layer 1

Layer 1 contains the two core components of the client application. The first is a shared reference to the daemon. On startup, the client application connects to the daemon and creates a reference that is shared to all classes in the client application. The reference to the daemon adheres to CbackupdProtocol (see Appendix A).

The second core component of the client application is the DBController class. Similar to the DBController class in the daemon's Layer 1, the client application's DBController class provides methods other parts of the application use to interface with the client's local database. This database is a small subset of what the daemon's database stores about the client. Its primary purpose is to serve as a fast and local cache for populating the client application's graphical user interface. Changes made to the client application's local database that need to be communicated to the daemon are sent to the shared daemon object, conforming to CbackupdProtocol.

Layer 2

Layer 2 of the client application's architecture stack contains the classes that control the main graphical interfaces of the client application. In general, they are responsible for configuring the graphical interface, populating it with data, and handling user input. The four classes in this layer are ApplicationController, PlanSheetController, SetupAssistantController, and RecoveryAssistantController. It would have been possible to combine all of these classes

into a single class that controls all the graphical interfaces, but this would have been convoluted. Cocoa design guidelines encourage one controller class for each graphical interface, following the Model-View-Controller design pattern [26].

The `AppController` class controls the client application's primary interface. The interface contains a table of backup plans with four columns of data about each plan. There are buttons to add, remove, and edit backup plans. Two buttons in the lower-right corner allow for on-demand restore and backup of a plan. When a plan is running, a status indicator and message appears in the top-right corner of the window.

The `PlanSheetController` class controls the interface that is presented when the user wants to create a new backup plan or edit an existing one. It has four tabs that step the user through the process of creating a backup plan. Their titles are: Name, What, Where, and When. In plan creation mode, this interface restricts tab navigation to the "Next" and "Back" buttons, as is typical with assistant-style interfaces. In edit mode, the user is free to jump around from tab to tab using the selectors at the top of the tab view.

The `SetupAssistantController` class controls the first interface the user sees. When the client application detects this is the first time the user has run the application, it will present this interface. The `SetupAssistantController` will step the user through creating their key and saving a backup of it. Alternatively, if the user is in a disaster recovery scenario, this interface will step them through recovering their key and database.

The `RecoveryAssistantController` class controls the interface used for recovering all or a portion of a backup. It is launched when the user clicks the "Restore" button on the

client application's primary interface. Once the `RecoveryAssistantController` class has received the necessary user input, it calls on the daemon to execute the restore process.

Layer 3

Layer 3 of the client application is the graphical interface layer. From the user's perspective, this is the client application. The views presented at this layer are managed by the controllers in Layer 2. For more detailed information on the purpose and implementation of these views, refer to Chapter 5.

Cryptic Backup Suite

As previously discussed, the Cryptic Backup suite consists of the daemon and the client application. The daemon is designed to service multiple clients on the system. Because of this, only one instance of the daemon needs to be running at a time. Indeed, this is the behavior expected by the daemon and the client application. It is possible for multiple users to be running their own instance of the client application, and again, this is expected. In order for the daemon to sort out which user is which, each remote call made by a client is accompanied by the user's Unix shortname.

To accomplish backup tasks, the Cryptic Backup suite reaches out to remote servers. This communication is handled by the daemon. Under no circumstances does the client application communicate with a remote server. Additionally, the client application has limited communication with the local file system. Because the daemon runs as a system level process, it has free reign over the file system to read and write user backup data. A high-level diagram of the Cryptic Backup suite can be seen in Figure 4.6.

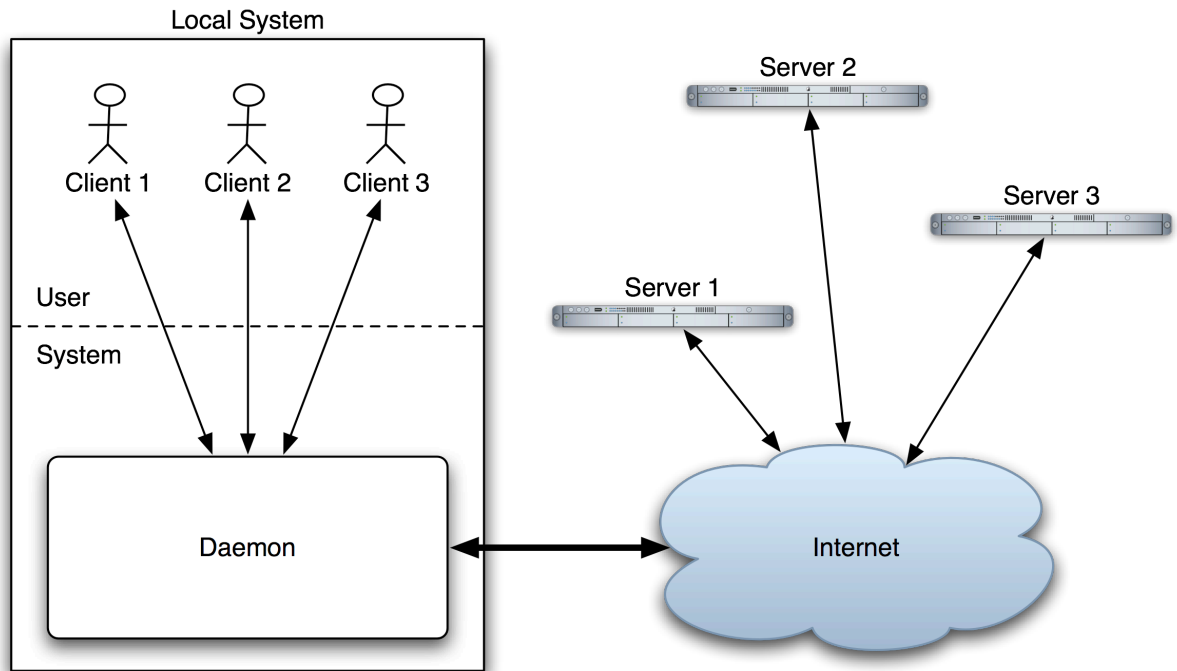


Figure 4.6: Cryptic Backup Suite Architecture

Alternate Approaches

During the design of the Cryptic Backup suite, alternate approaches were considered. This section will present a few of the alternatives and explain why they are not as well suited for the Cryptic Backup suite as the architecture and design already discussed.

Separation of Functionality

The original design of the Cryptic Backup suite wasn't a suite at all; it was a single application designed to perform all the tasks of the current daemon and client application. However, a significant problem with this design was discovered early on. Applications with graphical user interfaces are typically launched by the user and run in the user's windowing environment. The user may leave the application running all the time while they are logged

in. In the case of Cryptic Backup, this would allow backup plans for the user to run as scheduled. The problem with this configuration arises when the user is not logged in, meaning the application is not running. Without the application running, backup plans will not run.

An initial solution to this problem involved creating a daemon-like application for each user to accompany their graphical interface application. However, it was soon realized this design did not scale well. At this point, it was decided there should be a single system-level daemon to service all users. The daemon should be launched at system startup and should schedule and run backup plans in the background, even if the plan's owner was not currently logged in. Each user on the system would use their own instance of the client application at the user-level to interact with the daemon.

After determining the need for separate applications, the next design choice was where to implement the functionality. The reality of users logging out of the system meant that the daemon had to be able to perform backups on schedule, independent of the client application. This placed the SSCrypto and Connection frameworks and their associated functionality in the daemon.

Finally, different approaches were considered for the location of persistent storage. There were three choices: the daemon maintains the only database, each client application maintains its own database (with a very minimal daemon database), or the daemon and each client application maintain a database. Having the daemon maintain the only database was ruled out because populating the graphical interface of a client application would put excessive and unnecessary load on the daemon. The option of having each client application

maintain a database with a minimal daemon database was ruled out for a few reasons. The minimal daemon database would consist of a list of clients and the location of their database file. This might seem workable, but the first problem is if the daemon and client application are attempting to access the shared database simultaneously. This would likely result in database corruption. The second problem is the daemon not being able to load the user's database file. An example of when this could happen is if the user's home directory exists on a remote server and is only mounted when they are logged in. If the user isn't logged in, the daemon wouldn't be able to access their database file and thus wouldn't know about their backup plans. Due to the shortcomings of the first two choices, the final design used option three: the daemon has a single database where it stores information about all the users and their backup plans. Each client application has its own database, used primarily to support its graphical user interface.

Concurrent File Transfers

The first design of the backup pipeline and restore pipeline (Figure 4.3) used an instance of `UploadOperation` and `DownloadOperation` for each file processed, just like `EncryptOperation` and `DecryptOperation` are used in the current design. Within `UploadOperation` and `DownloadOperation`, a connection is made to the remote server. Since dozens or even hundreds of these operations could be running concurrently, this quickly becomes a problem. The net result would be opening dozens or hundreds of simultaneous connections to the same remote server. If the user didn't own the server, this would likely result in their IP address or login credentials being blocked from accessing the server. However, even if it was the user's own server, opening hundreds of connections is

counterproductive. Significant overhead is involved with each connection and opening hundreds of them is likely to overload the remote server.

Another problem with this design is the fact that the limiting factor for most users will be their own connection speed. A single upload consuming 100 percent of the user's upload bandwidth and ten uploads, each consuming 10 percent of the user's upload bandwidth, have the same net effect: all upload bandwidth is being utilized. Because of this and the reasons outlined in the previous paragraph, a single UploadOperation or DownloadOperation, with one connection to the remote server, is shared by each BackupOperation or RestoreOperation, respectively.

Thread-Safe Database Access

As discussed in the daemon's Layer 2 section, multiple threads need access to the database simultaneously. The ultimate solution to that problem is explained in that section and illustrated in Figure 4.4. However, another approach was considered before deciding to use multiple managed object contexts. The Objective-C language provides the `@synchronized()` directive for synchronizing thread execution [27]. This directive is a wrapper around the POSIX mutex system. It could have been used to restrict all database access to a single thread at a time. This would prevent database corruption, but it comes at a significant cost. Continuous database access is required when running backup plans. With plans running simultaneously, the database could become a bottleneck in the process. To mitigate this potential bottleneck, the current design, where database changes are made in multiple contexts and then merged later, was implemented.

Determination of Changed Files

One of the tasks handled by BackupOperation in the Cryptic Backup daemon is finding files that have changed since the last run of the backup plan. The first approach considered for this was to use Apple's filesystem events (FSEvents) API. The mechanism behind this API consists of three parts: kernel code that passes raw filesystem event notifications to user space through a special device, a daemon which filters this stream and sends out notifications, and a persistent database which stores a record of all filesystem changes throughout time [28]. Applications can subscribe to receive FSEvent notifications in real-time while they are running or at any time, they can catch up with what they missed while they weren't running.

FSEvents is a very powerful API, but it has some limitations. Apple made the pragmatic decision that only directory-level changes are stored in the persistent database. So whether one or one hundred files change within a directory, an application is only notified at the granularity of the parent directory in which the changes occurred. Another limitation of FSEvents is in the management of the persistent events database. This database is stored at the root of all HFS+ partitions and requires Mac OS X 10.5 for proper management. This means if a partition is modified by a computer running an earlier version of Mac OS X or another operating system altogether, the persistent database is not accurate. If the partition doesn't use the HFS+ filesystem, the persistent database doesn't exist at all. Clearly, this is a problem with external hard drives that are frequently moved between computers. Changes could be made to the filesystem without being logged in the database. Because of this, Apple

recommends applications periodically perform a full sweep of partitions they care about to ensure that no changes fall through the cracks [28].

With the evident need to develop a system to scan for changed files that need to be backed up, it was decided to use this internal implementation full time and forgo using the FSEvents API. The implementation of this system is explained in further detail in Chapter 5, but in short, it involves recursively scanning directories and searching for any file that has a last modified date after the previously known last modified date for that file.

CHAPTER 5. IMPLEMENTATION

Based on the goals of Cryptic Backup and the architecture and design decisions made, the implementation called for two separate components: the daemon and the client application. This chapter will not serve as an exhaustive explanation of every mundane implementation detail; rather it will highlight the important algorithms that provide vital functionality to Cryptic Backup. This implementation discussion will be divided into two sections: one part for the daemon and another for the client application.

Daemon

The implementation of the Cryptic Backup daemon was directed by the protocol established to frame communication between the client application and the daemon. This protocol is named CbackupdProtocol and can be seen in Appendix A. The protocol defines the messages a client application can send to the daemon. The daemon can respond to these messages with a return value, but it never initiates communication. If a client application wishes to receive status updates of a running backup plan, for example, it is required to periodically poll the daemon for that information. The following subsections explain the implementation of major tasks performed by the daemon.

Client & Key Management

The daemon's database (see Appendix B) contains a Client entity used to keep track of clients that have registered with the daemon. All protocol messages, except for the one to test a connection, require clients to identify themselves with their Unix shortname. If a user is not registered, all messages will be ignored except for the two messages dedicated to

registering new users. The first, and most common, registration message involves the client application sending the user's passphrase to the daemon. The steps are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname and passphrase.
- 2) The daemon generates the user's key using the SHA-256 hash algorithm based on the provided passphrase.
- 3) The daemon creates a new Client entity in its database with the user's shortname and encryption key.

The second message for registering a user involves the client application sending the user's key file data to the daemon. This message is used in a disaster recovery scenario when the user needs to restore their key using the saved key file. The steps are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname and key file data.
- 2) The daemon retrieves the user's encryption key from the provided key file data.
- 3) The daemon creates a new Client entity in its database with the user's shortname and encryption key.

In a disaster recovery scenario, the user also must know how to access one of their previously configured backup locations. Once they enter the connection details, the client application instructs the daemon to retrieve a backup of the user's database from that location. Upon retrieving the database backup, the daemon imports the information into its

own database. It also passes the information to the client application so it can rebuild its local database. The steps are below:

- 1) The client application sends a message to the daemon containing the user's shortname, the type of this backup destination (iDisk, SFTP, etc.), the hostname, the port number, the username, the password, and the root folder.
- 2) The daemon connects to the destination to retrieve the backup of the user's database using the connection details provided by the client application.
- 3) The daemon decrypts, uncompresses, and unarchives the user's database backup.
- 4) The daemon imports the user's database information into its database.
- 5) The daemon packages the database information and sends it to the client application.
- 6) The client application rebuilds its local database with the information provided by the daemon.

The final key management task handled by the daemon is the creation of key file data. This data is generated by request of the client application so the user can save a copy of their key file. The steps are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname.
- 2) The daemon retrieves the user's encryption key from its database.

- 3) The daemon generates the key file data based on the user's encryption key and returns the data to the client application.

Upon receiving the key file data, the client application will save it to a file at a location specified by the user.

Backup Plans

CbackupdProtocol has six messages dedicated to creating, editing, and deleting backup plans. To keep track of backup plan information, the daemon's database contains Plan, Destination, and Source entities. The Plan entity has a one-to-one relationship with the Destination entity and a one-to-many relationship with the Source entity. The database links clients with their backup plans, so the Client entity has a one-to-many relationship with the Plan entity.

Creating a fully defined backup plan requires at least three messages to be sent to the daemon from the client application. The first message creates the Plan entity. The steps are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname, a newly generated Universally Unique Identifier (UUID) for the plan, the plan's name, the time of day the plan should run, how frequently the plan should run, whether the plan should be enabled, and the date the plan was created.
- 2) The daemon creates a new Plan entity populated with the information sent by the client application.
- 3) If the backup plan is enabled, the daemon determines the next time the plan should run and schedules it for that time.

The next message required for defining a backup plan creates the Destination entity. All the components of this message must be sent, but certain parts are ignored based on the destination type. For example, Amazon S3 connections don't require a hostname and port number because they are known and hard-coded. The steps for this message are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname, the plan UUID, a newly generated destination UUID, the type of this destination (iDisk, SFTP, etc.), the hostname, the port number, the username, the password, and the root folder.
- 2) If this plan doesn't yet have a destination, the daemon creates a new Destination entity populated with the information sent by the client application. If this plan does already have a destination, the daemon updates the Destination entity with the new information sent by the client application.

The final message required for defining a backup plan creates the Source entities. A Source entity identifies a file or directory that should be backed up. A backup plan must have one or more sources defined. The steps for this are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname, the plan UUID, and an array of sources for the plan.
- 2) If this plan already has sources defined, the daemon first deletes all of them. The daemon always creates a new Source entity for every source sent by the client application.

At this point, a backup plan is fully created and no further action is required by the client application. However, the protocol allows for the user to edit, delete, and toggle whether a

plan is enabled. The message sent to edit a plan resembles the message sent to create a plan.

The steps for editing a plan are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname, the plan UUID, the plan's name, the time of day the plan should run, and how frequently the plan should run.
- 2) The daemon edits the Plan entity with the information sent by the client application.
- 3) If the backup plan is enabled, the daemon determines the next time the plan should run and schedules it for that time.

If the user chooses to delete a backup plan, the client application sends a message to the daemon attempting to do this. If the backup plan is currently in backup or restore mode, the daemon will refuse. The steps for this are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname and the plan UUID.
- 2) If the plan is currently running, the daemon refuses to delete it and returns a Boolean false to the client application. If the plan is not currently running, the daemon deletes the plan, removes it from its schedule, and returns a Boolean true to the client application.

The final protocol message dedicated to manipulating backup plans instructs the daemon to toggle the enabled status of the backup plan. For various reasons, a user may want to temporarily disable a backup plan so it won't automatically run. Alternatively, the user may want to re-enable a currently disabled plan. When they choose to do this, the client

application sends a message to the daemon instructing it to toggle the enabled status of the backup plan. The steps for this are as follows:

- 1) The client application sends a message to the daemon containing the user's shortname and the plan UUID.
- 2) The daemon toggles the enabled status of the plan in its database and either adds or removes it from its list of scheduled backup plans, depending on the new state.

Another aspect of backup plans the daemon is responsible for is scheduling and automatic backup. There are three parts to this process: calculating the next time a backup plan should run, scheduling it so the daemon will be notified at that time, and responding to the notification. When a user creates a backup plan, they can specify what time of day it should run, which days of the week it should run, and if it should run every week, every other week, or once a month. To determine when the backup plan should run next, the daemon considers the current time and date, the last time and date the plan was run, and the plan's scheduling preferences. Using the result of this complex calculation, a timer is created and set to fire when the backup plan should run next. The timer has a callback function that is automatically called when the timer fires. This function is responsible for starting the backup operation, the process for which is outline in the next subsection.

Creating Backups

A backup plan starts running for one of two reasons. Either its scheduled time has arrived or the user indicated they want to run the backup immediately using the client

application's GUI. Regardless of the reason, after the daemon decides to run a backup plan, the steps it takes to kick it off are the same. They are outlined below:

- 1) If the plan was already running (backing up or restoring), the daemon halts the backup request. If the plan was not already running, the daemon adds it to the list of running plans.
- 2) If the plan has a timer, it is invalidated to guarantee it won't fire during the backup operation.
- 3) The daemon retrieves the user's encryption key and information about the backup plan from its database.
- 4) The daemon initializes a BackupOperation object with the information it needs to run the backup.
- 5) The daemon starts the BackupOperation by adding it to the queue (NSOperationQueue) created for BackupOperations.

After step five, the daemon thread is free to process more incoming messages from client applications. The algorithm to start a backup was designed to do as little work as possible before passing the job off to BackupOperation, which runs on a separate thread.

Before BackupOperation can start uploading files, it needs to determine the files that are new, have changed, or have been deleted. These are the files that will require attention during the backup. New and changed files need to be uploaded. Deleted files need to be deleted from the backup. To keep track of the files that have been backed up, the daemon's database keeps a record of every file with a mapping between its local path and its filename

in the backup store. This record is stored in the File entity, which has a many-to-one relationship with the Plan entity. To find pertinent files, BackupOperation compares the Source entities with the File entities. Items found in Sources, but not in Files, are marked as new and a new File entity is created. Items found in Files, but not in Sources, are marked as deleted and the File entity is deleted. Items found in both Sources and Files are checked to see if their last modified date has changed since the last backup, and if it has, they are marked as changed. As BackupOperation finds files that require attention, it delegates the work to helper operations (threads) and continues searching for more work. An overview of the steps BackupOperation takes are below:

- 1) Create a new NSManagedObjectContext object to use for making changes to the daemon database.
- 2) Create a queue (NSOperationQueue) for each sub-operation (CompressOperation, EncryptOperation, and UploadOperation).
- 3) Create the shared UploadOperation that will be used for all file transfers.
- 4) Locate files that are new, changed, or deleted.
- 5) For each new or changed file found:
 - a. Create a CompressOperation and EncryptOperation.
 - b. Set the EncryptOperation to not start until the CompressOperation has finished.

- c. Queue up the `CompressOperation` and `EncryptOperation` by adding them to their respective queues.
- 6) For each deleted file found:
 - a. Instruct the `UploadOperation` to delete it from the backup store.
 - 7) Wait for all uploads and deletes to complete.
 - 8) Update flags in the File entities ('compressed' and 'lastBackupSuccessful').
 - 9) Merge changes made in the local managed object context with the primary managed object context in the `DBController` class.
 - 10) Extract database information relevant to this user from the daemon database, encrypt it with the user's key, compress it, archive it, and upload it to the backup store.
 - 11) Inform the daemon that this `BackupOperation` is complete so its backup plan can be rescheduled.

`BackupOperation` is by far the most complicated component of the backup process. Its three sub-operations are comparatively simple. Their functionality is briefly described in the next three paragraphs.

`CompressOperation` is the start of the backup pipeline. It is responsible for opening the file and reading it into a data object (`NSData`). This creates a copy of the file in memory. The next action taken is to compress the file using `bzip2` [29] compression. `CompressOperation` then passes the data to `EncryptOperation` and terminates.

EncryptOperation encrypts the data with the user's key using the AES-256 algorithm. After encryption is complete, EncryptOperation adds the data to the upload queue of the shared UploadOperation and terminates.

UploadOperation maintains a connection to the backup store. It uploads data to the backup store as it receives it from EncryptOperation. Since there is little to no advantage in running multiple uploads simultaneously, UploadOperation waits for one upload to complete before starting another. Upcoming uploads are placed in a queue and are prioritized using first-in, first-out. To mask filenames, a UUID is generated to serve as the filename in the backup store. Files uploaded to the backup store are placed in a single directory, regardless of their position in the local directory structure. UploadOperation is also responsible for deleting files from the backup store as necessary.

Restoring Backups

Restore operations only occur on demand. They start as a result of the user requesting to restore from a backup plan using the client application's GUI. Before the restore is started, the daemon needs to know which files the user would like to have restored and where they'd like to save the restored files. The procedure for gathering this information is as follows:

- 1) The client application sends a message to the daemon requesting the list of files backed up by a particular backup plan. This message contains the user's shortname and the plan UUID.
- 2) The daemon retrieves the list of files for this plan from its database and returns the paths to the files to the client application.

- 3) The client application presents this information to the user and allows them to choose specific files to restore or lets them default to restoring them all.
- 4) The client application asks the user if they'd like to set the root of the restore to a particular directory or if they'd like the files to be restored to their original location (overwriting as necessary).
- 5) The client application sends a message to the daemon requesting the restore start. This message contains the user's shortname, the plan UUID, the files to restore, and the root restore directory.
- 6) The daemon begins the restore process based on the information it has received from the client application.

To start the restore process, the daemon follows a series of steps. They are outlined below:

- 1) If the plan was already running (backing up or restoring), the daemon halts the restore request. If the plan was not already running, the daemon adds it to the list of running plans.
- 2) If the plan has a timer, it is invalidated to guarantee it won't fire during the restore operation.
- 3) The daemon retrieves the user's encryption key and information about the plan from its database.
- 4) The daemon initializes a `RestoreOperation` object with the information it needs to run the restore.

- 5) The daemon starts the RestoreOperation by adding it to the queue (NSOperationQueue) created for RestoreOperations.

As with the backup process, the daemon is now free to process more incoming messages from client applications. The RestoreOperation, running on another thread, continues the restore procedure.

RestoreOperation is not as complex as BackupOperation, primarily because it doesn't have to determine which files have changed. RestoreOperation iterates through the list of files it has been instructed to restore and passes the work off to helper operations. The steps it takes are as follows:

- 1) Create a new NSManagedObjectContext object to use for reading the daemon database. The RestoreOperation does not make changes to the database.
- 2) Create a queue (NSOperationQueue) for each sub-operation (DownloadOperation, DecryptOperation, and UncompressOperation).
- 3) Create the shared DownloadOperation that will be used for all file transfers.
- 4) Iterate through the list of files to restore, for each:
 - a. Create a DecryptOperation and UncompressOperation.
 - b. Set the UncompressOperation to not start until the DecryptOperation has finished.
 - c. Queue up the DecryptOperation and UncompressOperation by adding them to their respective queues.

- 5) Wait for all UncompressOperations to complete.
- 6) Inform the daemon that this RestoreOperation is complete so its backup plan can be rescheduled.

Although the restore process is fairly simple, RestoreOperation still relies on three sub-operations to complete its tasks. The functionality of these sub-operations is briefly described in the next three paragraphs.

DownloadOperation is the starting point for the restore pipeline. As RestoreOperation iterates through the files that need to be restored, it adds them to DownloadOperation's internal queue. Only one file is downloaded at a time. Once a file is completely downloaded, its data is passed to the DecryptOperation previously created for it.

Receiving data from DownloadOperation is DecryptOperation's cue to start decrypting the data. This reverses the process done by EncryptOperation, using AES-256 and the user's key. After decryption is complete, DecryptOperation passes the data to UncompressOperation and terminates.

UncompressOperation reverses the bzip2 compression process previously conducted by CompressOperation. Once the data is uncompressed, it is ready to be written to a file to complete the restore. Before doing this, UncompressOperation ensures all the directories along the file's path exist. If they don't, UncompressOperation creates them. After the file is restored to the desired location, UncompressOperation terminates.

Client Application

The design of Cryptic Backup called for a lightweight client application to serve as a front-end for the workhorse daemon. As such, the implementation of the client application centered on creating a graphical user interface and writing the code to support it. The work of generating keys, creating backups, performing restores, and communicating with remote servers is passed off to the daemon.

Referring back to Figure 4.5, most of the implementation work was done in Layers 2 and 3 of the client application's architecture stack. Obviously, some work went into Layer 1's DBController class, but it was comparatively minimal. The client application's database and DBController class were designed and implemented to support the controllers in Layer 2 and the views in Layer 3. The following subsections focus on the implementation details of the major client application user interfaces.

Setup Assistant Window

The first time the user runs the client application, they are greeted with the setup assistant window. This window steps the user through the process of creating and saving a copy of their key file. Alternatively, in a disaster recovery scenario, it can step them through the process of recreating or recovering their key and restoring their database. The figures below show screenshots of the setup assistant window in various states.

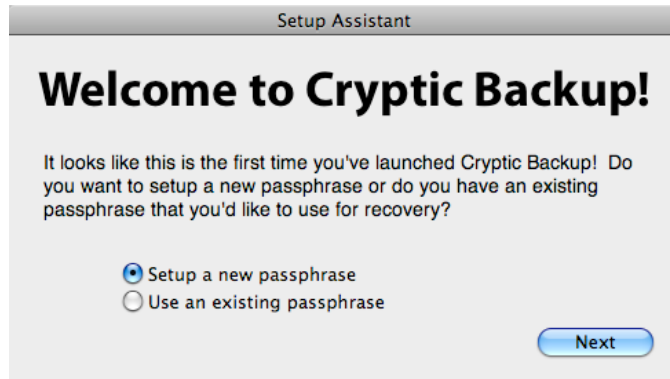


Figure 5.1: Setup Assistant — Welcome

The above Figure 5.1 shows the first interface the user will see. From here they can decide to create a new passphrase or use an existing one for recovery.

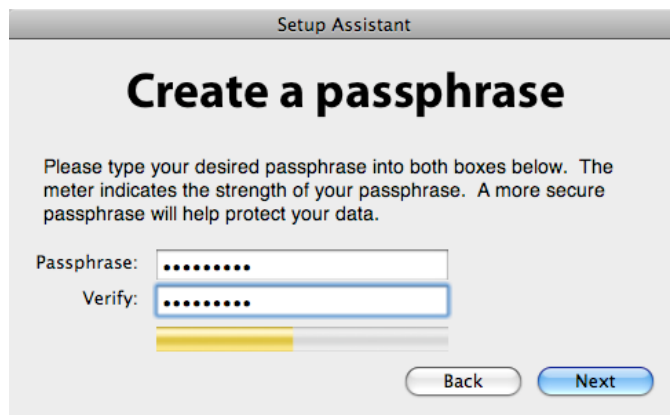


Figure 5.2: Setup Assistant — Create passphrase

Figure 5.2 shows the interface used to create a passphrase. The user must enter the passphrase twice to avoid typo problems. The meter below the text fields estimates the strength of the entered passphrase. In this example, the user has entered a moderately strong passphrase.



Figure 5.3: Setup Assistant — Save passphrase

The interface for saving a copy of the user’s key file is shown in Figure 5.3. The user must specify a location to save their key file using the “Choose location...” button before they are allowed to click the “Done” button. Once the “Done” button has been clicked, the client application registers the user with the daemon and saves the copy of their key file.



Figure 5.4: Setup Assistant — Load passphrase

Figure 5.4 shows the interface presented to the user if the lower radio button was selected in the interface shown in Figure 5.1. This interface allows the user to load their key from a saved key file or recreate their key based on their recollection of their passphrase. If

the user clicks the “Choose key file...” button, they are presented with the standard Mac OS X open file window. After selecting their key file, the user is presented with the interface shown in Figure 5.5. However, if the user instead types in their passphrase and clicks the “Next” button, they are first presented with the interface shown in Figure 5.3 to save a copy of their key file. After specifying the save location, the user is presented with the interface shown in Figure 5.5. For diagrams of the possible execution paths of the Setup Assistant window, see Appendix D.

The image shows a screenshot of a Mac OS X window titled "Setup Assistant". The window's main heading is "Secure backup settings". Below the heading, there is a paragraph of instructions: "Please enter the location and connection details for your backup location. We will attempt to restore your settings from this backup." The form contains several input fields: "Connection type:" with a dropdown menu set to "SFTP"; "Server Address:" with a text field containing "example.com"; "Port:" with a text field containing "22"; "Username:" with a text field containing "choff"; "Password:" with a masked text field containing ten dots; and "Remote Path (optional):" with a text field containing "/home/choff/backups/". At the bottom right of the window, there are two buttons: "Back" and "Done".

Figure 5.5: Setup Assistant — Recover settings

The interface shown in Figure 5.5 is used to gather backup store information from the user. This interface will look different depending on the selected connection type. To see examples of the different states, refer to Appendix F. The information shown on this interface is gathered from the user in a disaster recovery scenario and is used in an attempt to retrieve a backup of the user’s database. Recovering the database is accomplished by instructing the daemon to retrieve a backup of the database it previously created and stored in

the user's backup store. Once the user has entered connection details and clicked the "Done" button, the client application uses a message in the protocol to instruct the daemon to test for connectivity. If the connection test fails, the user is asked to correct any errors they may have made. If the connection test succeeds, the client application registers the user with the daemon and the daemon begins the database retrieval and rebuilding process.

Main Window

After completing the setup assistant, or any launch after the first one, the user is shown the main window. As its name implies, the main window is the primary interface for the client application. A screenshot of the main window is shown in Figure 5.6.

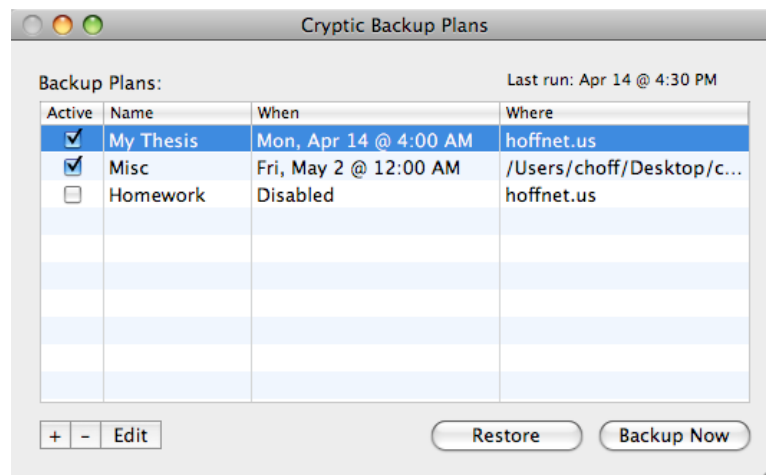


Figure 5.6: Main Window

The dominant feature of the above Figure 5.6 is the table with four columns. This table presents information about the user's backup plans based on what is stored in the client application's database. The first column indicates if the backup plan is enabled. By clicking on the checkbox in this column, the user can disable a backup plan without deleting it.

Disabled backup plans will not run. The next column is the user-assigned name of the backup plan. Column three contains the date and time the backup plan will next run. This date and time is not stored in the database. It is calculated based on information stored in the database. The final column of the table indicates where the backup plan stores its backups. Again, the contents of this column are not explicitly stored in the database. The displayed value is determined based on the connection type of the backup plan's destination. For example, a plan using Amazon S3 will simply display "Amazon S3" while a plan using an SFTP connection will display the hostname of the SFTP server.

In the upper-right corner of the main window, status messages are presented to the user based on the selected plan. A progress indicator informs the user if the plan is currently running. The daemon tracks the status of all running plans and allows client applications to poll for this information using the established protocol. The client application polls the daemon once every second.

The group of three buttons in the lower-left corner deal with creating, deleting, and editing plans. When the user clicks the "+" or "Edit" buttons, the client application presents the backup plan sheet, shown in Figures 5.7-5.10. If the user clicks the "-" button to delete a plan, the client application first checks with the daemon to see if this is allowed. The daemon will refuse to delete plans that are currently being backed up or restored. If the daemon allows the deletion, the plan will be deleted from the daemon's database as well as the client application's database.

The two buttons in the lower-right corner of the main window control the backing up and restoring of plans. When the user clicks the "Restore" button, the client application

presents the recovery assistant sheet, shown in Figures 5.11-5.14. The title of the “Restore” button will change to “Stop Restore” when a restore operation is running for the selected plan. If the user clicks this button, the client application will instruct the daemon to stop the restore process. Already restored files won’t be deleted, but files that haven’t yet been restored, won’t be restored. The final button is the “Backup Now” button. When the user clicks this, the client application instructs the daemon to backup the selected plan now, regardless of its next scheduled time. As with a restore operation, the title of the “Backup Now” button will change to “Stop Backup” when a backup operation is running for the selected plan. If the user clicks this button, the daemon will be instructed to halt the backup. However, changes already made to the backup store will not be reverted.

Backup Plan Sheet

When creating or editing a backup plan, the backup plan sheet is displayed. On Mac OS X, a sheet is different from a window in a few ways. First, as can be seen from the figures below, a sheet doesn’t have a title bar. This is because a sheet is actually attached to another window. It can be attached at any point within the window, but by default it attaches below the window’s title bar, centered horizontally [30].

The backup plan sheet can be presented in either create or edit mode. In create mode, all the fields and controls are set to default and the user is not allowed to use the tab selectors along the top to navigate. Navigation can be achieved using the “Back” and “Next” buttons. In edit mode, the values for the fields and controls are loaded from the client application’s database and the user is free to use the tab selectors for navigation. The figures below show screenshots of the required steps for creating a backup plan.

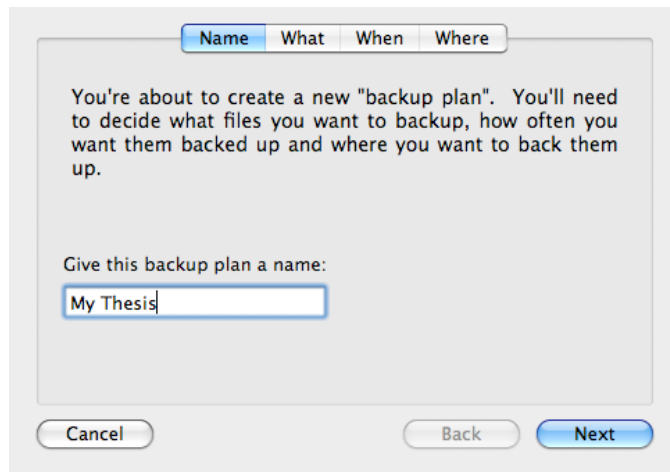


Figure 5.7: Backup Plan — Name tab

The interface shown in Figure 5.7 is the first tab the user is presented with when they decided to create a new backup plan. It explains to them what they are about to do and asks them to give their backup plan a name. In this example, the user has given the name “My Thesis” to their backup plan.

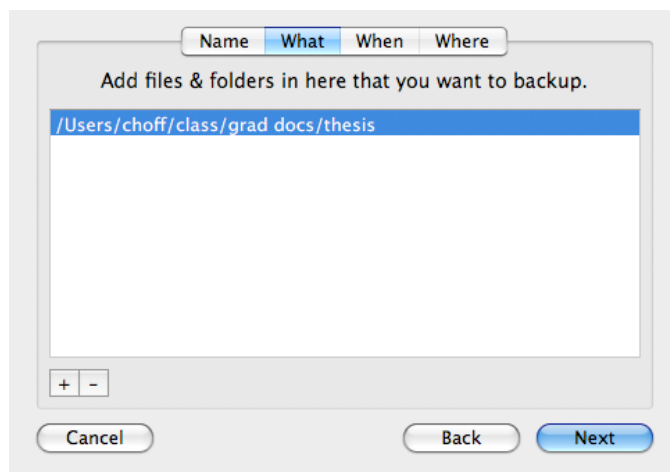


Figure 5.8: Backup Plan — What tab

Figure 5.8 shows the next tab presented to the user. Using this interface, the user identifies the files and folders they'd like backed up with this plan. To add files and folders to the table, the user can drag and drop them from the Finder [31] or click the “+” button to use the Mac OS X open file window to select them. To remove files and folders, the user can drag and release them outside of the table or select them and click the “-” button. In this example, the user has decided to backup only one item. However, since it is a folder, all items within the folder will also be backed up.

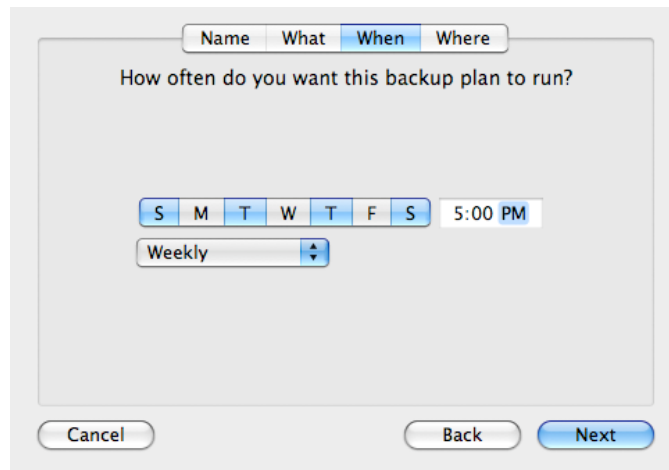
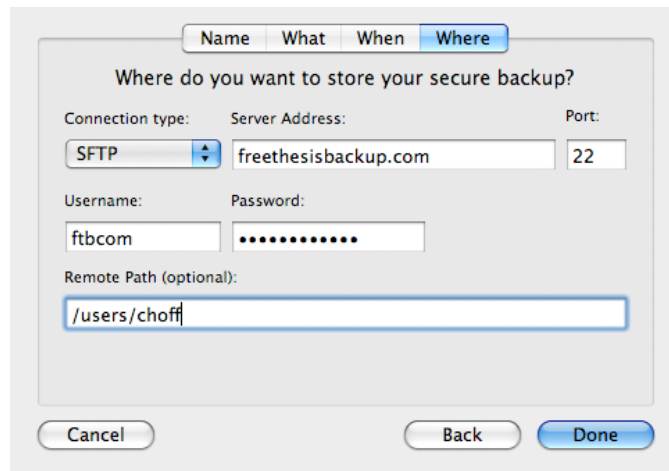


Figure 5.9: Backup Plan — When tab

Figure 5.9 shows the “When” tab in the backup plan interface. This tab is used to specify when the backup plan should run. To specify the days of the week the plan will run, the user selects the desired segments of the Sunday through Saturday picker. Setting the time of day that the backup plan will run, down to the minute, is accomplished using the date picker. Finally, the user can use the frequency selector to specify if they'd like the backup to

run every week, every other week, or every month. In this example, the user has decided to run the backup every week on Sunday, Tuesday, Thursday, and Saturday, at 5:00 PM.



The screenshot shows a configuration window titled "Where do you want to store your secure backup?". At the top, there are four tabs: "Name", "What", "When", and "Where", with "Where" being the active tab. The window contains the following fields and controls:

- Connection type:** A dropdown menu set to "SFTP".
- Server Address:** A text input field containing "freethesisbackup.com".
- Port:** A text input field containing "22".
- Username:** A text input field containing "ftbcom".
- Password:** A password input field with masked characters (dots).
- Remote Path (optional):** A text input field containing "/users/choff".

At the bottom of the window, there are three buttons: "Cancel", "Back", and "Done". The "Done" button is highlighted in blue, indicating it is the primary action.

Figure 5.10: Backup Plan — Where tab

The final tab for creating a backup plan is shown in Figure 5.10. This tab is shown in the SFTP state. Refer to Appendix F to view the five states that the backup location interface can be in. The “Done” button for this interface will not be enabled until the user has entered enough information. How much information is required depends on the connection type. For example, an Amazon S3 connection only requires a username and password. Once the user has entered text into those two fields, the “Done” button will be enabled. As with the database recovery interface shown in Figure 5.5, once the user clicks the “Done” button, the connection details will be tested. If the connection test fails, the user will be asked to check for errors in the information they entered. If the connection test succeeds, the client application will instruct the daemon to create a new backup plan using the information entered by the user.

Recovery Assistant Sheet

To allow the user to recover files from a backup, the recovery assistant sheet is used. This sheet steps the user through the process of selecting the files they'd like to recover and where they'd like them restored to. The figures below show screenshots of the recovery process.

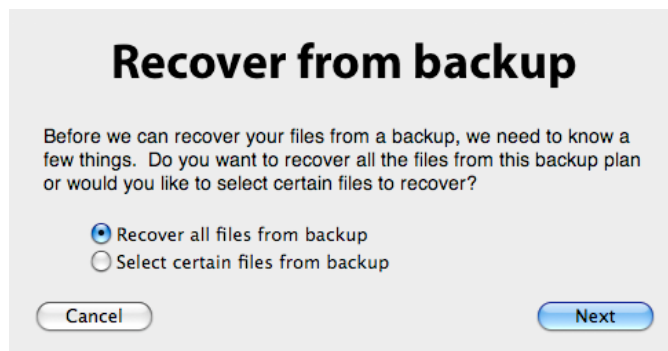


Figure 5.11: Recovery Assistant — All files or a subset

The above Figure 5.11 shows the first interface presented by the recovery assistant. The user is given the choice between recovering all the files from the backup or a subset.

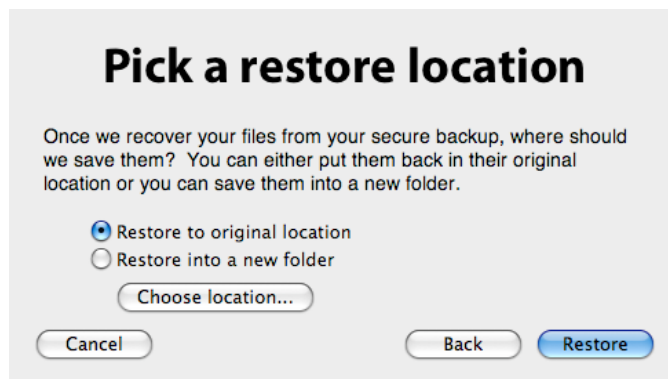


Figure 5.12: Recovery Assistant — Restore to original location

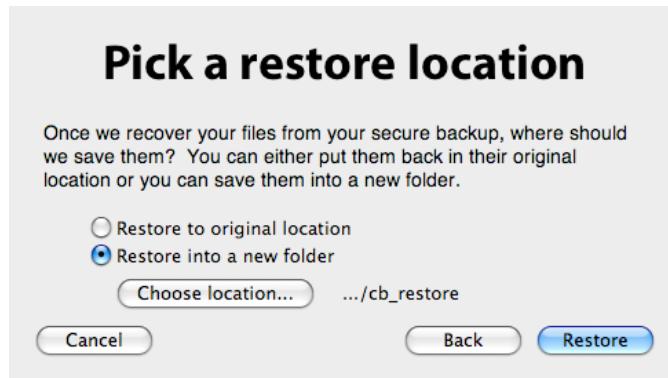


Figure 5.13: Recovery Assistant — Restore to custom location

The interface shown in Figure 5.12 presents the user with a choice to either have their files restored to their original location or to specify a custom location to restore them to. If the user decides that they'd like to pick a custom location, they can select the lower radio button or click the “Choose location...” button. In either case, the client application presents the Mac OS X open file window to let the user specify a custom location. Figure 5.13 shows the interface once the user has selected the location. In this example, the user has chosen to restore their backup to path ending in “cb_restore”. When the user clicks the “Restore” button, the client application sends the daemon the information it needs to start the restore process. For diagrams of the possible execution paths of the Recovery Assistant sheet, see Appendix E.

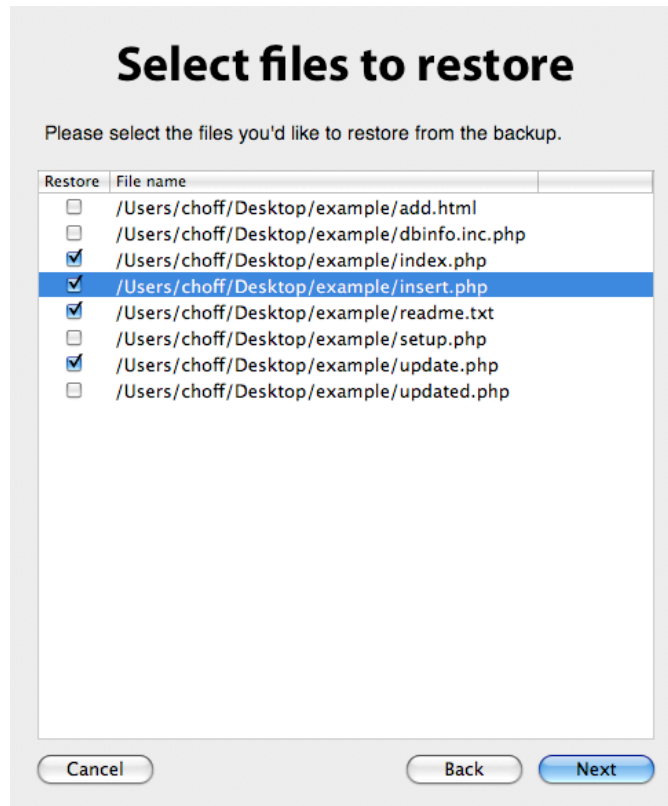


Figure 5.14: Recovery Assistant — Select files

Figure 5.14 shows the interface presented to the user if the lower radio button was selected in the interface shown in Figure 5.11. This interface shows the user all the files in their backup in a two-column table. The first column contains a checkbox where the user can mark the files they'd like restored. The second column contains the local path of the file. The file may or may not exist at this path, but it was there at one point. To change the checkbox for several files at once, the user can select multiple rows and click the checkbox of one of the selected rows. This will set the state of all the selected rows to the new state of the clicked row. Once the user has made their selection and clicked the “Next” button, they are presented with the interface shown in Figure 5.12 to complete the recovery assistant.

CHAPTER 6. FUTURE WORK

The Cryptic Backup suite in its current state has met the goals outlined in Chapter 3. It is a fully functional encrypted backup solution designed with the average computer user in mind. However, there is always room for improvement, and Cryptic Backup is no exception. This chapter discusses six aspects of Cryptic Backup that could use further attention and one new feature. Possible implementation strategies are offered for each. Additionally, this chapter discusses future work required on user documentation if Cryptic Backup was to be sold commercially.

Improved Key Management

As discussed in Chapter 3, Cryptic Backup operates under the assumption that in a disaster recovery scenario, the user either has a copy of their key file or they remember the passphrase they used to create it. This is a rather large assumption, and an admitted weakness in the design. It's possible that the user created their key many years before the occurrence of the disaster they are attempting to recover from and they've since forgotten their passphrase. This is especially likely because Cryptic Backup never needs to prompt the user for their passphrase in day-to-day operation, so it may not be fresh in their mind. It's also likely that the user never bothered to save a copy of their key file on anything other than the hard drive, which is now unreadable. Cryptic Backup forces the user to save a copy of their key file when they create a key and encourages them to store it in a safe place, but it can't force them to do so.

The nightmare scenario of a user not being able to decrypt their backed up data could be avoided if a backdoor was introduced into the cryptosystem, but this would compromise the security of all backups created by Cryptic Backup. A backdoor to decrypting the data is not an acceptable solution. Another possible solution to this problem is setting up a key escrow arrangement. Such arrangements place a key needed to decrypt data in the hand of a trusted third party. The key should be bound to strict access control policies. Only explicitly allowed parties should be granted access. If Cryptic Backup was sold as a commercial application, the owning company could offer key escrow services to its customers. If the customer agrees to place their key in escrow, Cryptic Backup could send a copy of the key to said company. Later, when the customer has misplaced their key and forgotten their passphrase, they could in some way positively identify themselves to the company and retrieve a copy of their key. This might sound like an ideal solution, but several concerns arise from this arrangement. How secure is the customer's key with the company? Who within the company has access to the user's key? How far will the company go to withhold the user's key from law enforcement requests? How can the company verify the identity of a person requesting a copy of a customer's key? These questions and many more need to be answered to establish a viable key escrow arrangement.

More Robust Change Tracking

A portion of Chapter 4 discussed the reasoning behind the chosen method for determining if a file has changed. Chapter 5 detailed the implementation of the design in which files are flagged as changed if their last modified date has changed. However, this is not always adequate. It is possible, although uncommon, to change a file without updating

its last modified date. A relatively simple extension of the current implementation would be to also check if the file's size has changed. If either the last modified date or file size has changed, the file could be flagged for backup. But again, a file could be modified without its size or last modified date changing.

Another approach to solving this problem is performing a byte for byte comparison of the old version of the file with the current version. However, this would be very expensive considering Cryptic Backup would either have to download and decrypt the backed up file or save a copy of the decrypted file locally, for comparison purposes only. At this point, it becomes obvious a cryptographic hash should be used to determine if the file has changed. Before compressing, encrypting, and uploading a file, Cryptic Backup could calculate and store a hash of it in the daemon's database. Later, when trying to determine if the file has changed, the daemon could calculate a hash of the current file and compare it to the stored hash. The downside to this approach is creating a hash is far more expensive than checking the last modified date and file size, especially for large files. The costs and benefits of using hashes over more simple checks needs to be analyzed further before committing to such an approach.

Compression Suitability Evaluation

As briefly discussed in Chapter 4, and in more detail in Chapter 5, `CompressOperation` performs `bzip2` compression on a file before sending it to `EncryptOperation` to be encrypted. Decreasing the size of files speeds up the rest of the backup and restore process. The smaller a file is, the less time it takes to encrypt and upload, and later, download and decrypt. The shortcoming with the current implementation is that it

attempts to compress all files. For already compressed files types, such as JPEG, MP4, and ZIP, this simply wastes time.

To avoid wasting time attempting to compress already compressed files, Cryptic Backup could benefit from the use of a compression suitability evaluation. A compressibility analysis of the file should be fairly inexpensive and could throw out files unlikely to benefit from compression. Failing that, a simple file extension check could go a long way in determining which files to avoid compressing.

Safer Backup of Files Located on Mounted Volumes

Another aspect of Cryptic Backup identified as needing future work is the algorithm used to determine if the volume the file exists on is currently mounted. Checking for this is necessary because the daemon will delete a file from the backup if it discovers it has been deleted from the local system. Without ensuring the file's volume is currently mounted, the daemon would delete a file from the backup that still exists locally. Obviously, this is not the behavior desired by the user. So, if the daemon determines the volume is not mounted, it ignores the file and moves on.

The current algorithm used to determine if a volume is mounted is not flexible enough. The issue is determining what volume the file belongs to and where that volume is mounted. On Mac OS X, volumes are automatically mounted under /Volumes. But, as with most Unix-like operation systems, volumes can be mounted anywhere on the filesystem. Although it wouldn't make much sense, a volume could be mounted at /var/some/deep/path. Cryptic Backup needs a reliable method to determine where a file's volume is mounted. If it knows this, when it discovers the file is missing, it could check to see that the file's volume is

mounted and thus be sure that it is safe to delete the backup of the file. The current algorithm assumes volumes are mounted under /Volumes.

Even this is not foolproof, however. Nothing prevents volume mount points from being reused. Therefore, a volume could appear to be mounted, when in actuality, another volume is mounted at the same location. A possible solution to this problem is creating a zero-byte file at the root of the mounted volume. This file could serve as a unique identifier for the volume. However, this approach comes with its own assumptions: namely that the daemon has write access to the root of the mounted volume and that the file won't be deleted by an external program or person. Clearly, an improved implementation of this algorithm requires more thought and consideration.

Improved Delete Behavior

A stated assumption of the Cryptic Backup daemon is that if it can't find a local file when running a backup plan, it will remove it from the backup store. The implicit assumption here is that the user intentionally deleted the file and so it is safe to remove it from the backup store. However, this might be a very dangerous assumption. If data the user cares about is deleted without their knowledge, and the backup plan containing that data runs, the data could be lost forever. In this scenario, the daemon would delete the data from the backup store. If the user didn't have another copy somewhere, the data would be lost.

One solution to this problem would be to prompt the user before deleting anything from the backup. This is difficult however, considering backups are run in the background by the daemon, which has no graphical interface. Probably the best solution is to allow the user to configure the delete behavior on a per-plan basis. The user could indicate they never

want anything deleted from the backup. Alternatively, the user could take a more aggressive approach and indicate that all files that can't be found locally should be deleted from the backup store.

Bundle Support

In Mac OS X, a bundle is a directory in the file system that groups related resources together in one place [32]. Applications and many document formats are examples of bundles a user will frequently encounter. To the user, these directories appear to be a single file. Double-clicking an application bundle will cause that application to launch. Double-clicking a document bundle will cause it to open with its default application.

In its current implementation, Cryptic Backup treats bundles as they truly are: a directory containing sub-directories and files. However, this behavior can be confusing for a user that doesn't know (and likely doesn't care) what a bundle is. This is especially true when the user attempts to restore specific files from a backup. The list of possible files to restore includes all the components of the bundles in the backup plan. Not only is this confusing to the user, but it is also dangerous. Bundles are not meant to be split up. If the user attempts to restore a document that is actually a bundle, and they neglect to select every file within the document bundle, the restored document is likely to be corrupted and unusable. Because bundles should not be split up, Cryptic Backup should present them to the user as a single entity. To restore an application or document bundle, the user should only have to select the bundle itself, not all the files and directories contained within it. The user should never even see a listing of the files contained in the bundles they've selected for backup.

Implement Watch Lists

A new feature of Cryptic Backup that users may find helpful is the concept of watch lists. A watch list is a group of files or directories that the user wants automatically mirrored to a backup location. Users typically want to do this to ensure that they always have the latest version of a file backed up. This can be accomplished with the current implementation of Cryptic Backup by using the “Backup Now” button in the client application’s main window. Watch lists aim to automate this process so the user doesn’t have to think about it.

Watch lists could be implemented in Cryptic Backup with a special type of backup plan. When the user creates the backup plan, instead of specifying when and how often they want the plan to run, they could specify “Immediately” to indicate this plan should backup file changes immediately, as they happen. The daemon could use Apple’s FSEvents API mentioned in Chapter 4 to automatically receive notifications when the contents of a watched directory have changed. Upon receiving a notification, the daemon could search the contents of the directory for watched files that have changed and proceed to run the backup operation on updated files.

User Documentation

At this point, no documentation has been created for Cryptic Backup aimed specifically at end-users. If Cryptic Backup was ever sold commercially, this documentation, along with a help system, would have to be created. The documentation should start by explaining what Cryptic Backup does. This is important because the term ‘backup application’ is ambiguous. As discussed in Chapter 2, many backup applications exist.

Among these applications, there are several different approaches to backup. Some applications perform incremental backups while others simply synchronize data between two locations.

Since the user may not be familiar with the concept of backup plans, this should be explained, particularly what happens to the backed up data when a plan is edited or deleted. Different scenarios for when recovery may be needed should be presented to the user, ranging from accidentally deleting a file to having their computer stolen. In each scenario, the documentation should advise what should be done before and after the incident. For example, in a scenario where the user's computer has been stolen, the list of tasks that should have been completed before this happened would include having a copy of the key file or recalling the passphrase used to create the key. Additionally, the documentation would include the recommendation to have all the valuable data on the computer backed up as often as necessary to minimize the risk of data loss. A helpful side-note in the documentation could explain what directories the user should backup to ensure all their email messages, documents, web browser bookmarks, and other important information can be restored.

The feature of Cryptic Backup that sets it apart from other backup applications is its integrated encryption. This distinction needs to be made clear and explained to the end-user. The added layer of encryption means the user needs to understand what a passphrase is and how it should be thought of as different from a password. Related to the passphrase is the key file. The user needs to be aware that without their passphrase or a copy of their key file, their backups are unreadable. This point probably can't be stressed too much; it is critical that the user understands the importance of this information.

During the process of creating this documentation, it may be determined that aspects of the Cryptic Backup user interface need to change based on how users conceptualize backups. User testing should be conducted on Cryptic Backup to determine if the interface is intuitive and if the documentation is adequate. The end goal is to get users to backup their data. Previous studies have shown that if this process is too complicated, users will neglect to do it [1].

CHAPTER 7. CONCLUSION

The lofty goals for Cryptic Backup were stated in Chapter 3. To summarize, Cryptic Backup aims to provide reliable data recovery, protect the backed up data in transit and at rest, and make the process of creating backups as intuitive and painless as possible. These goals directed the design and implementation of what became the Cryptic Backup suite. As outlined in Chapter 6, additional work can be done to improve Cryptic Backup. Specifically, more analysis is needed on the usability of the Cryptic Backup client application interface. However, in its current state, Cryptic Backup reached all the goals it set out to. The result is a Cryptic Backup suite that is highly versatile, reliable, and more than capable of fulfilling the encrypted backup needs of the average computer user.

APPENDIX A: CbackupdProtocol

```

@protocol CbackupdProtocol

/* registration and key management */
- (BOOL)doesDaemonKnowMe:(NSString *)shortName;
- (void)registerClient:(NSString *)shortName passphrase:(NSString *)passphrase;
- (void)registerClient:(NSString *)shortName keyFileData:(NSData *)keyData;
- (NSData *)createKeyFileDataForClient:(NSString *)shortName;
- (NSData *)recoverDBForClient:(NSString *)shortName type:(NSString *)type
    hostname:(NSString *)hostname port:(NSString *)port
    username:(NSString *)username password:(NSString *)password
    rootFolder:(NSString *)rootFolder;

/* backup plan management */
- (void)newPlanForClient:(NSString *)shortName planUUID:(NSString *)planUUID
    planName:(NSString *)planName time:(NSDate *)time
    frequency:(NSData *)frequency isEnabled:(BOOL)isEnabled
    dateCreated:(NSDate *)dateCreated;
- (void)editPlanForClient:(NSString *)shortName planUUID:(NSString *)planUUID
    planName:(NSString *)planName time:(NSDate *)time
    frequency:(NSData *)frequency;
- (BOOL)deletePlanForClient:(NSString *)shortName planUUID:(NSString *)planUUID;
- (void)togglePlanEnabledForClient:(NSString *)shortName planUUID:(NSString *)planUUID;
- (void)setDestForClient:(NSString *)shortName planUUID:(NSString *)planUUID
    destUUID:(NSString *)destUUID type:(NSString *)type
    hostname:(NSString *)hostname port:(NSUInteger)port
    username:(NSString *)username password:(NSString *)password
    rootFolder:(NSString *)rootFolder;
- (void)setSourcesForClient:(NSString *)shortName planUUID:(NSString *)planUUID
    sources:(NSArray *)sources;

/* backup and restore operation management */
- (void)backupNowForClient:(NSString *)shortName planUUID:(NSString *)planUUID;
- (NSArray *)getBackedUpFilesForClient:(NSString *)shortName planUUID:(NSString *)planUUID;
- (void)restoreNowForClient:(NSString *)shortName planUUID:(NSString *)planUUID
    paths:(NSArray *)paths rootFolder:(NSString *)rootFolder
    userID:(UInt32)uid groupID:(UInt32)gid;
- (void)stopPlanForClient:(NSString *)shortName planUUID:(NSString *)planUUID;
- (NSDictionary *)getBackupAndRestoreStatusForClient:(NSString *)shortName;

/* test for connectivity */
- (BOOL)testConnectionType:(NSString *)type host:(NSString *)host port:(NSString *)port
    username:(NSString *)username password:(NSString *)password;

@end

```

APPENDIX B: Daemon database

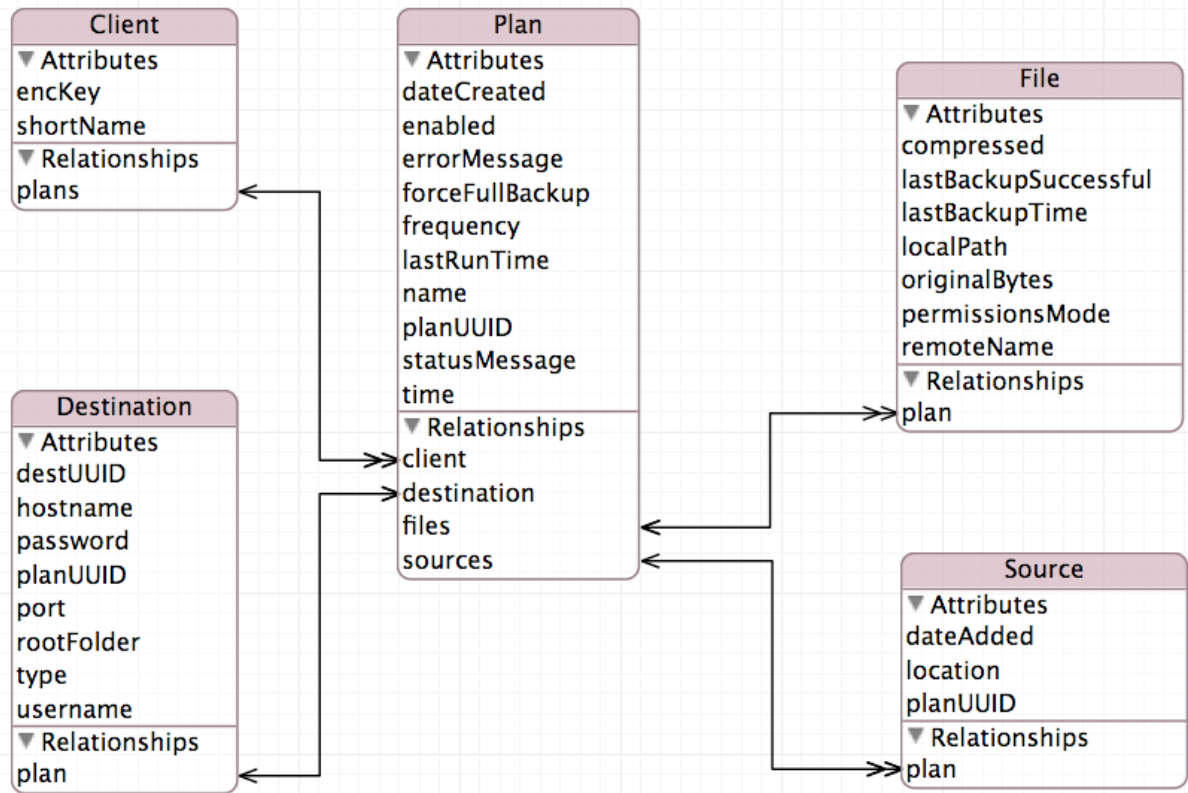


Figure B.1: Daemon Managed Object Model

Table B.1: Daemon database Client entity schema

Property	Type	Description
<code>encKey</code>	Binary	This client's encryption key
<code>shortName</code>	String	This client's Unix shortname
<code>plans</code>	Relationship	Relationship with this client's Plans

Table B.2: Daemon database Plan entity schema

Property	Type	Description
dateCreated	Date	The date this plan was created
enabled	Bool	Is this plan enabled?
errorMessage	String	This plan's most recent error message
forceFullBackup	Bool	Should the next backup by a full one?
frequency	Binary	How often this plan runs
lastRunTime	Date	The date this plan was last run
name	String	The user-assigned name of this plan
planUUID	String	This plan's UUID
statusMessage	String	This plan's most recent status message
time	Date	The time of day this plan runs
client	Relationship	Relationship with this plan's Client
destination	Relationship	Relationship with this plan's Destination
files	Relationship	Relationship with this plan's Files
sources	Relationship	Relationship with this plan's Sources

Table B.3: Daemon database Destination entity schema

Property	Type	Description
destUUID	String	This destination's UUID
hostname	String	This destination's server hostname or IP
password	String	The password required for this destination
planUUID	String	The UUID of this destination's Plan
port	Int 32	The port to connect to this destination on
rootFolder	String	The root folder to store backups in

Table B.3: Daemon database Destination entity schema (continued)

Property	Type	Description
type	String	The type of this destination (SFTP, etc.)
username	String	The username for this destination
plan	Relationship	Relationship with this destination's Plan

Table B.4: Daemon database Source entity schema

Property	Type	Description
dateAdded	Date	The date this source was added
location	String	The path of this source
planUUID	String	The UUID of this source's Plan
plan	Relationship	Relationship with this source's Plan

Table B.5: Daemon database File entity schema

Property	Type	Description
compressed	Bool	Has this file been compressed?
lastBackupSuccessful	Bool	Was the backup of this file successful?
lastBackupTime	Date	The last time this file was backed up
localPath	String	The local path of this file
originalBytes	Int 64	Size in bytes before compression
permissionsMode	Int 16	Unix file permissions (rwx)
remoteName	String	The filename of this file in the backup
plan	Relationship	Relationship with this file's Plan

APPENDIX C: Client application database

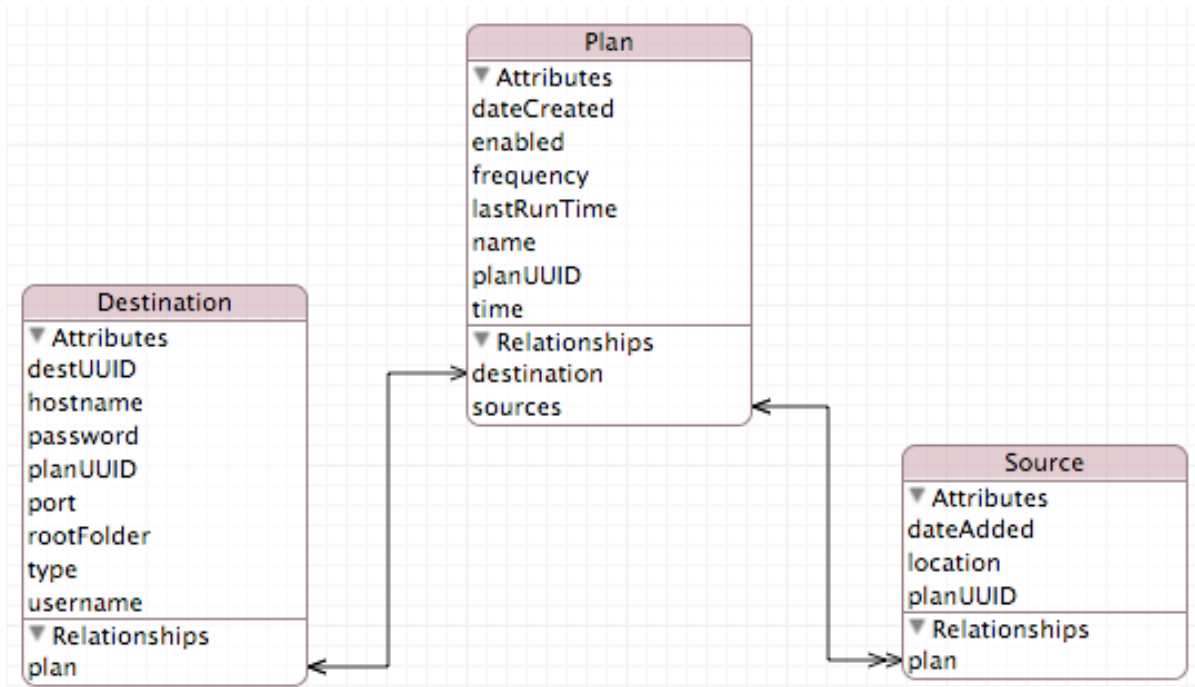


Figure C.1: Client Application Managed Object Model

Table C.1: Client application database Plan entity schema

Property	Type	Description
dateCreated	Date	The date this plan was created
enabled	Bool	Is this plan enabled?
frequency	Binary	How often this plan runs
lastRunTime	Date	The date this plan was last run
name	String	The user-assigned name of this plan
planUUID	String	This plan's UUID
time	Date	The time of day this plan runs
destination	Relationship	Relationship with this plan's Destination
sources	Relationship	Relationship with this plan's Sources

Table C.2: Client application database Destination entity schema

Property	Type	Description
destUUID	String	This destination's UUID
hostname	String	This destination's server hostname or IP
password	String	The password required for this destination
planUUID	String	The UUID of this destination's Plan
port	Int 32	The port to connect to this destination on
rootFolder	String	The root folder to store backups in
type	String	The type of this destination (SFTP, etc.)
username	String	The username for this destination
plan	Relationship	Relationship with this destination's Plan

Table C.3: Client application database Source entity schema

Property	Type	Description
dateAdded	Date	The date this source was added
location	String	The path of this source
planUUID	String	The UUID of this source's Plan
plan	Relationship	Relationship with this source's Plan

APPENDIX D: Setup Assistant window execution paths

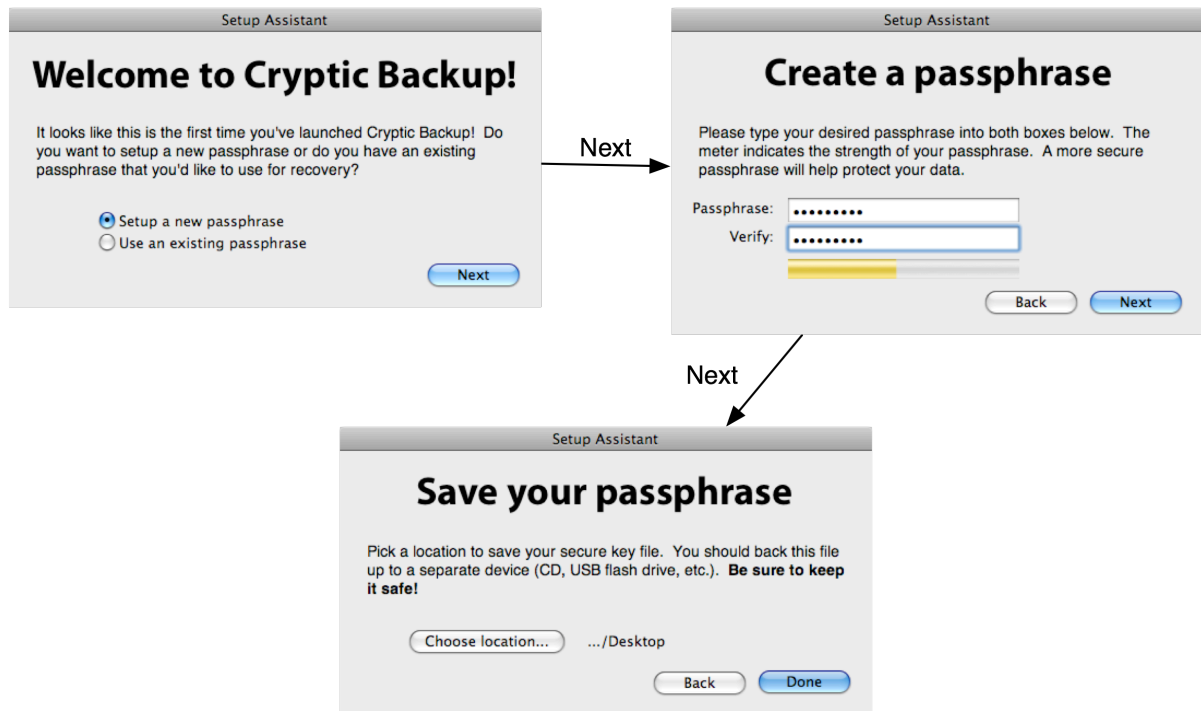


Figure D.1: Setup Assistant execution — primary path

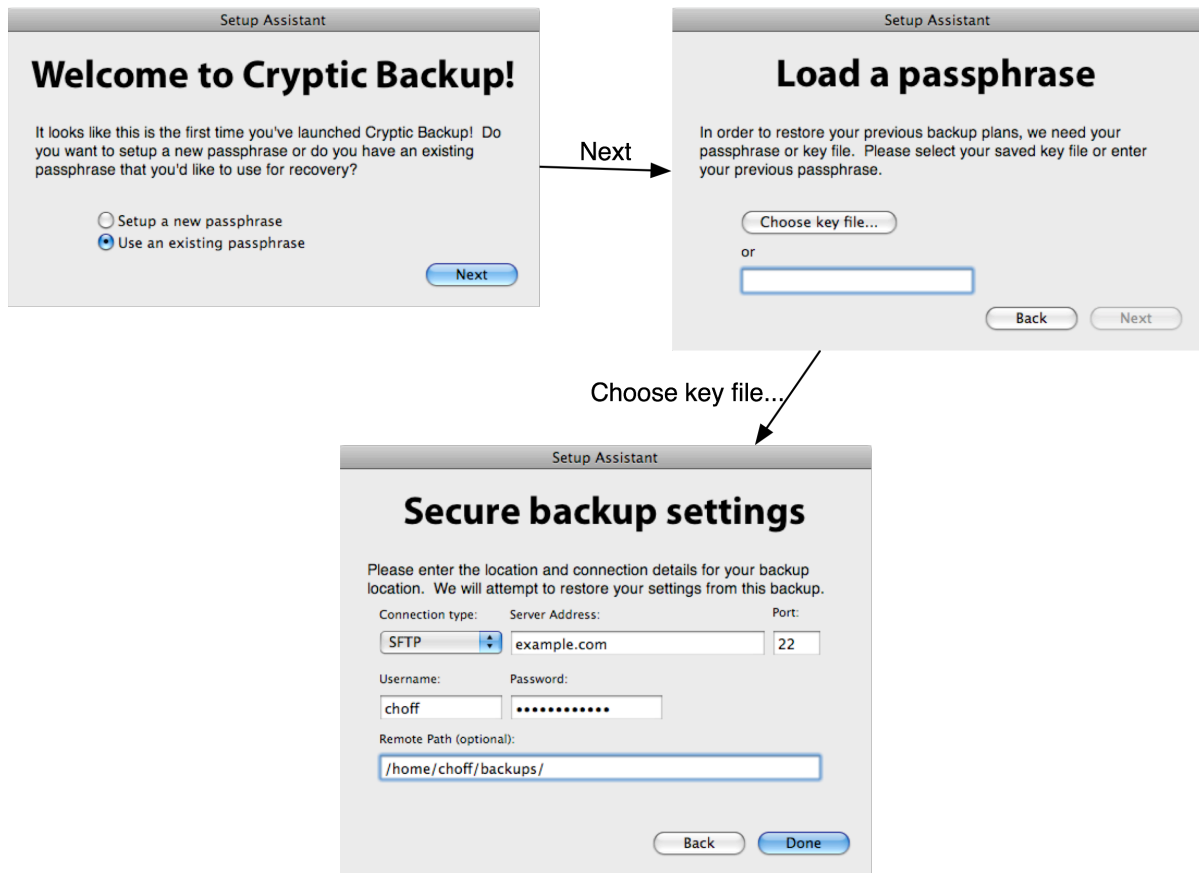


Figure D.2: Setup Assistant execution — alternate path — load key file

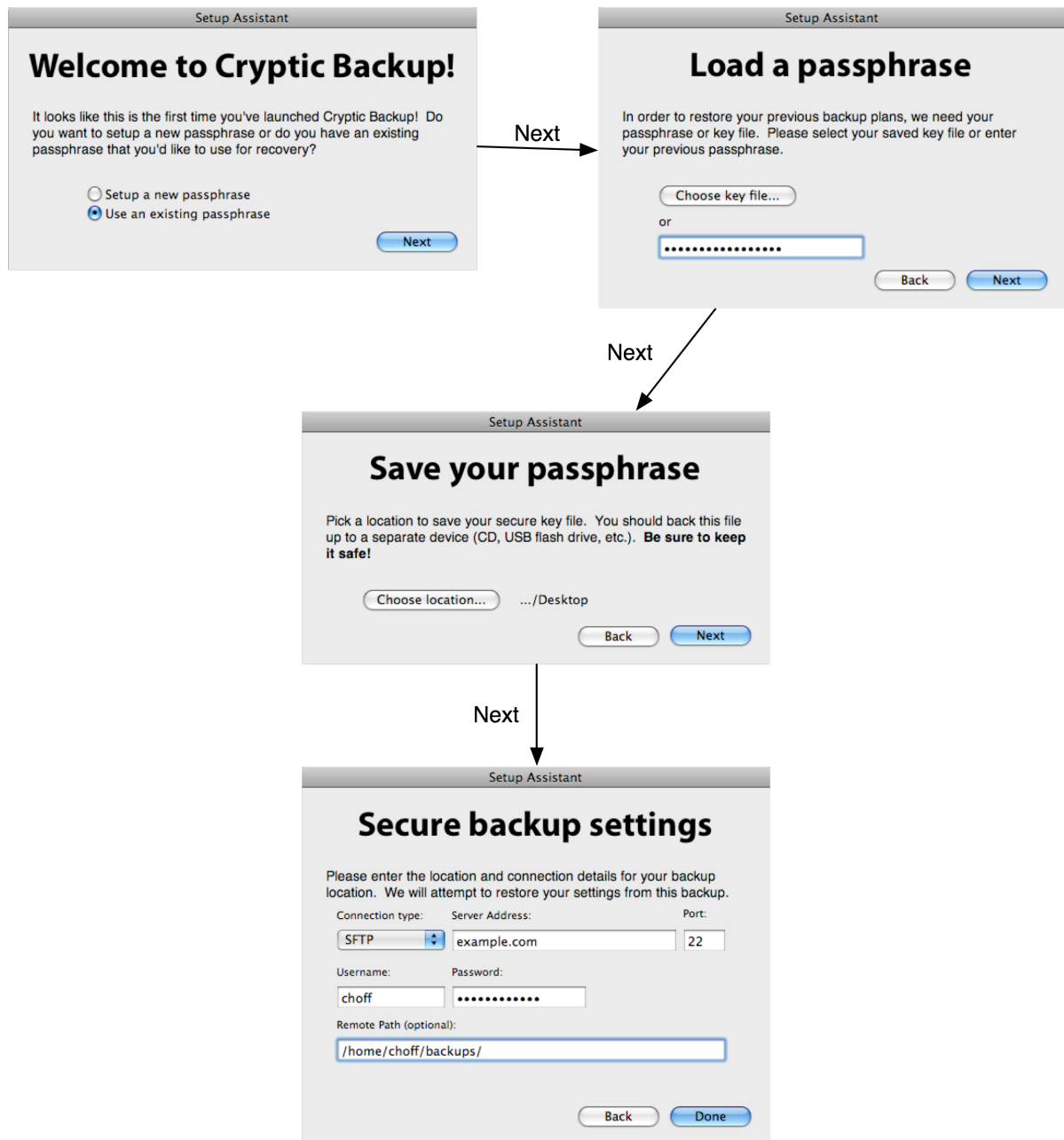


Figure D.3: Setup Assistant execution — alternate path — recall passphrase

APPENDIX E: Recovery Assistant sheet execution paths

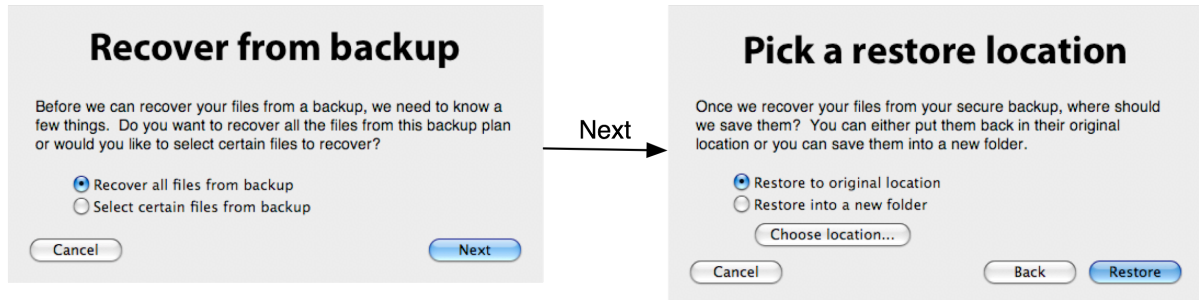


Figure E.1: Recovery Assistant execution — primary path

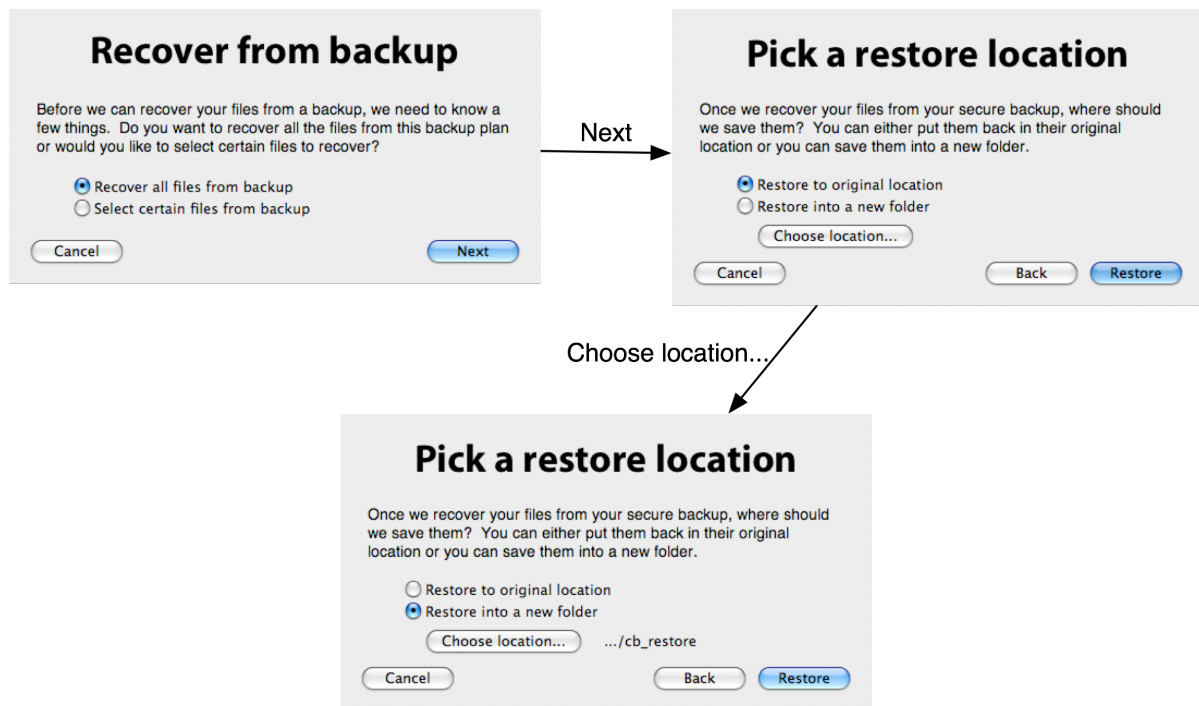


Figure E.2: Recovery Assistant execution — alternate path — custom location

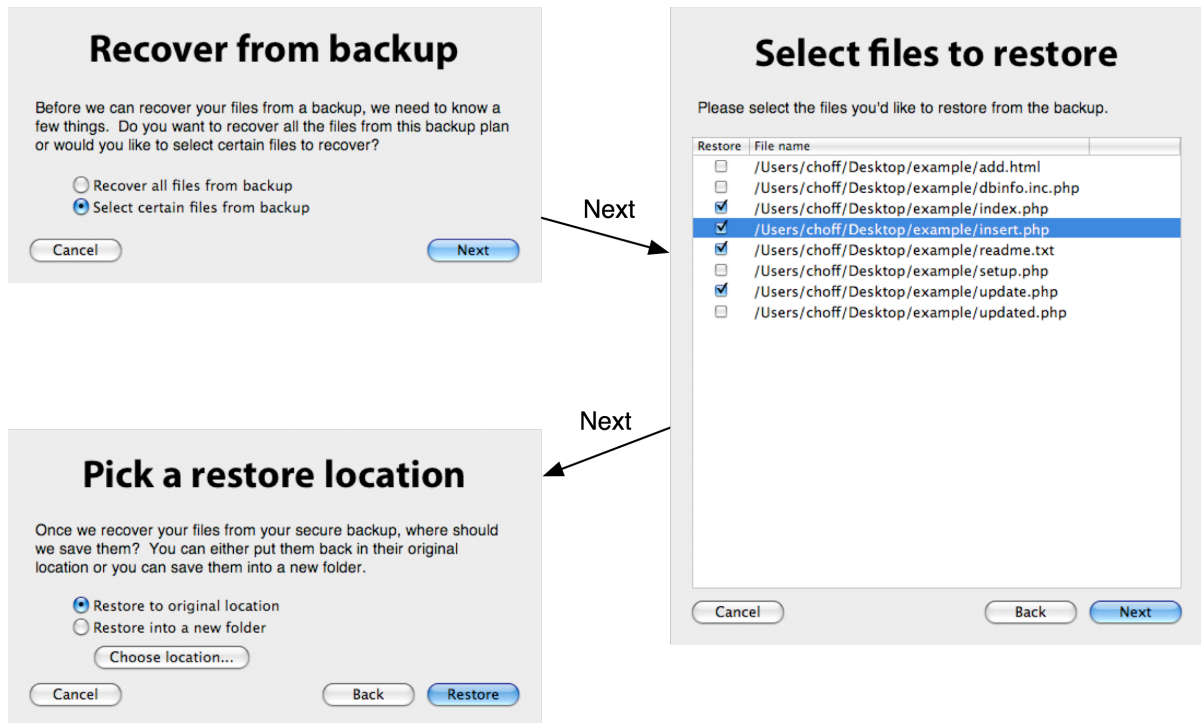


Figure E.3: Recovery Assistant execution — alternate path — specific files

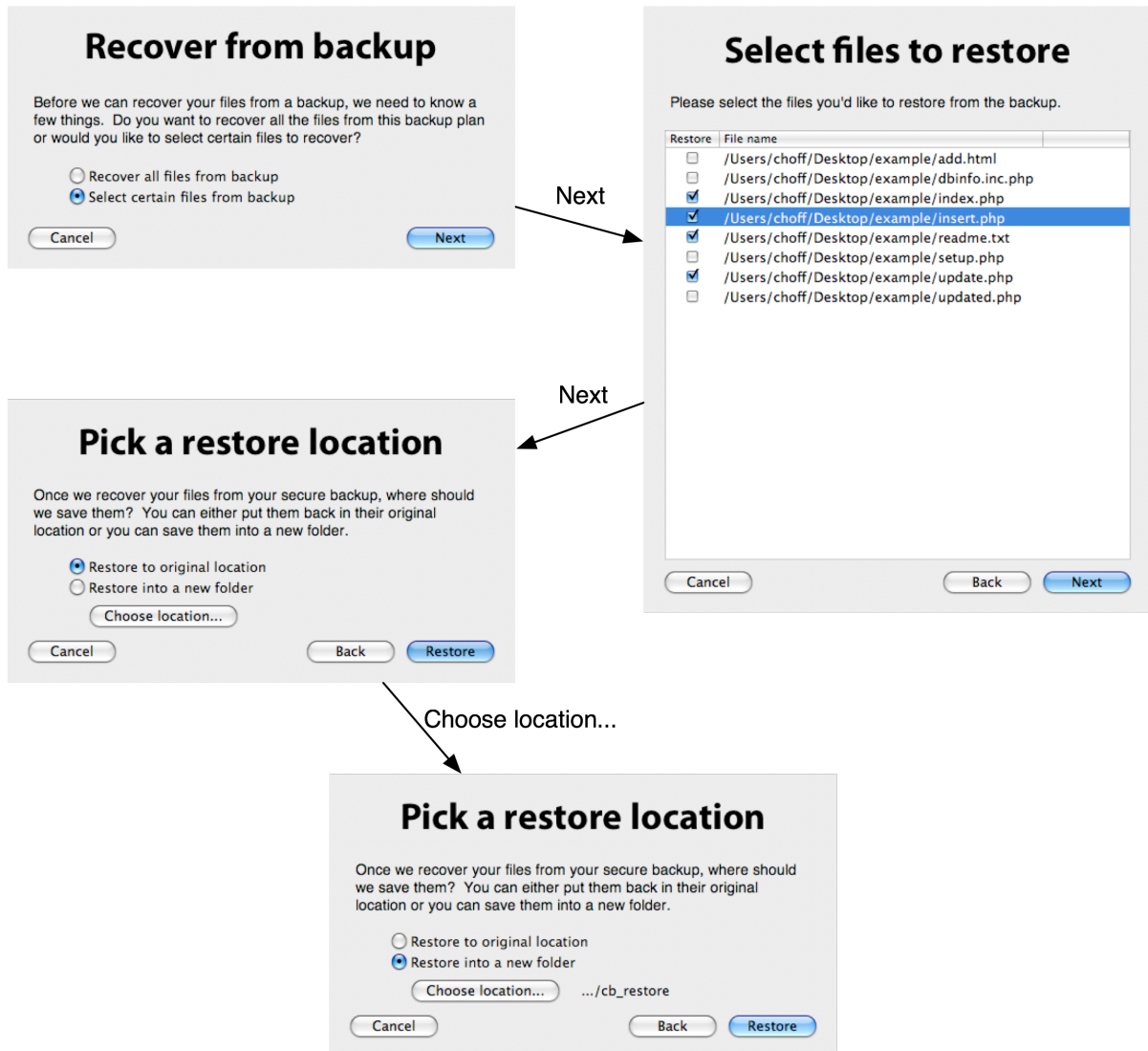


Figure E.4: Recovery Assistant execution — alternate path — specific, custom location

APPENDIX F: Backup location interface states

Where do you want to store your secure backup?

Connection type:
Amazon S3

Username: user Password: password

Bucket (optional):
bucket

Cancel Back Done

Amazon S3

Where do you want to store your secure backup?

Connection type:
iDisk

Cancel Back Done

iDisk

Where do you want to store your secure backup?

Connection type:
Local volume

Choose folder...

Cancel Back Done

Local volume

Where do you want to store your secure backup?

Connection type: SFTP Server Address: example.server.com Port: 22

Username: user Password: password

Remote Path (optional):
/users/me/backup

Cancel Back Done

SFTP

Where do you want to store your secure backup?

Connection type: Secure We... Server Address: example.server.com Port: 443

Username: user Password: password

Remote Path (optional):
/users/me/backup

Cancel Back Done

Secure WebDAV

Figure F.1: The five states of the backup location interface

REFERENCES

- [1] Most computer users walk a digital tightrope, according to national Maxtor survey. (2005). Maxtor Corporation. [Online]. Available: <http://www.harrisinteractive.com/news/newsletters/clientnews/Maxtor2005.pdf>. Retrieved: 27 Mar 2008.
- [2] GNU General Public License. (2007). GNU Project. [Online]. Available: <http://www.gnu.org/copyleft/gpl.html>. Retrieved: 30 Mar 2008.
- [3] rsync man page. (2008). [Online]. Available: <http://www.samba.org/ftp/rsync/rsync.html>. Retrieved: 29 Mar 2008.
- [4] crontab man page. (2004). The Open Group. [Online]. Available: <http://www.opengroup.org/onlinepubs/009695399/utilities/crontab.html>. Retrieved: 30 Mar 2008.
- [5] S. Shemesh. rsyncrypto man page. (2005). [Online]. Available: <http://www.penguin-soft.com/penguin/man/1/rsyncrypto.html>. Retrieved: 29 Mar 2008.
- [6] Time Machine. (2008). Apple Inc. [Online]. Available: <http://www.apple.com/macosx/features/timemachine.html>. Retrieved: 29 Mar 2008.
- [7] J. Siracusa. Time Machine. (2007). Ars Technica. [Online]. Available: <http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/14>. Retrieved: 29 Mar 2008.
- [8] Apple .Mac Backup. (2007). Apple Inc. [Online]. Available: <http://www.mac.com/1/solutions/backup.html>. Retrieved: 29 Mar 2008.
- [9] Apple .Mac iDisk. (2008). Apple Inc. [Online]. Available: <http://www.apple.com/dotmac/idisk.html>. Retrieved: 30 Mar 2008.
- [10] Amazon Simple Storage Service (Amazon S3). (2008). Amazon.com, Inc. [Online]. Available: <http://www.amazon.com/gp/browse.html?node=16427261>. Retrieved: 30 Mar 2008.
- [11] SuperDuper!. (2008). Shirt Pocket. [Online]. Available: <http://www.shirt-pocket.com/SuperDuper/SuperDuperDescription.html>. Retrieved: 29 Mar 2008.
- [12] Working with File Systems. (2005). Microsoft Corporation. [Online]. Available: <http://technet.microsoft.com/en-us/library/bb457112.aspx>. Retrieved: 30 Mar 2008.
- [13] Cocoa. (2008). Apple Inc. [Online]. Available: <http://developer.apple.com/cocoa/>. Retrieved: 27 Mar 2008.

- [14] OpenSSL: Documents, crypto(3). (2007). The OpenSSL Project. [Online]. Available: <http://www.openssl.org/docs/crypto/crypto.html>. Retrieved: 27 Mar 2008.
- [15] Foundation Framework Reference. (2007). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Cocoa/Reference/Foundation/ObjC_classic/index.html. Retrieved: 27 Mar 2008.
- [16] POSIX Threads Programming. (2008). Lawrence Livermore National Laboratory. [Online]. Available: <https://computing.llnl.gov/tutorials/pthreads/>. Retrieved: 28 Mar 2008.
- [17] Application Kit Framework Reference. (2007). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Cocoa/Reference/ApplicationKit/ObjC_classic/index.html. Retrieved: 27 Mar 2008.
- [18] Core Data Framework Reference. (2007). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Cocoa/Reference/CoreData_ObjC/index.html. Retrieved: 27 Mar 2008.
- [19] Core Data Programming Guide: Core Data Basics. (2008). Apple Inc. [Online]. Available: <http://developer.apple.com/documentation/Cocoa/Conceptual/CoreData/Articles/cdBasics.html>. Retrieved: 29 Mar 2008.
- [20] Septicus Software Source Code: SSCrypto.framework. Septicus Software. [Online]. Available: <http://septicus.com/products/opensource/>. Retrieved: 27 Mar 2008.
- [21] Federal Information Processing Standards Publication 180-2: Secure Hash Standard. (2002). National Institute of Standards and Technology. [Online]. Available: <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>. Retrieved: 28 Mar 2008.
- [22] Connection Kit. (2006). Under The Radar Software. [Online]. Available: <http://opensource.utr-software.com/connection/>. Retrieved: 27 Mar 2008.
- [23] Sandvox. (2008). Karelia Software. [Online]. Available: <http://www.karelia.com>. Retrieved: 31 Mar 2008.
- [24] ProfCast. (2008). Humble Daisy. [Online]. Available: <http://www.profcast.com/public/index.php>. Retrieved: 31 Mar 2008.
- [25] Federal Information Processing Standards Publication 197: Advanced Encryption Standard. (2001). National Institute of Standards and Technology. [Online]. Available: <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>. Retrieved: 29 Mar 2008.

- [26] Cocoa Fundamentals Guide: The Model-View-Controller Design Pattern. (2007). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/chapter_5_section_4.html#apple_ref/doc/uid/TP40002974-CH6-SW23. Retrieved: 29 Mar 2008.
- [27] The Objective-C 2.0 Programming Language: Threading. (2008). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/Articles/chapter_11_section_1.html. Retrieved: 29 Mar 2008.
- [28] File System Events Programming Guide: Introduction. (2007). Apple Inc. [Online]. Available: http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/chapter_2_section_1.html. Retrieved: 30 Mar 2008.
- [29] J. Seward. bzip and libbzip2 manual. (2007). [Online]. Available: <http://bzip.org/1.0.5/bzip2-manual-1.0.5.html>. Retrieved: 31 Mar 2008.
- [30] Sheet Programming Topics for Cocoa: About Sheets. (2006). Apple Inc. [Online]. Available: <http://developer.apple.com/documentation/Cocoa/Conceptual/Sheets/Concepts/AboutSheets.html>. Retrieved: 31 Mar 2008.
- [31] Mac OS X Leopard - Finder. (2008). Apple Inc. [Online]. Available: <http://www.apple.com/macosx/features/finder.html>. Retrieved: 31 Mar 2008.
- [32] Bundle Programming Guide: Introduction to Bundle Programming Guide. (2005). Apple Inc. [Online]. Available: <http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/CFBundles.html>. Retrieved: 17 Apr 2008.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to those that were instrumental in the genesis of this project and the completion of this thesis. First I would like to thank my committee members, especially Dr. Jacobson. His encouragement, funding, and feedback made this all possible. Additionally, I'd like to thank two of my colleagues, Brandon Newendorp and Steve Eilers: Brandon for his help in designing the graphical user interface and always being available to bounce ideas off of and Steve for his input, support, and interest in the project. I'd also like to thank my parents, Dave and Pat Hoff, for their unwavering support in all my endeavors and continued expectation of great things from me.

Most of all, I'd like to thank my wife, Jordan, for her steadfast encouragement, kindness, and support throughout my college years. We are about to embark on a new chapter in our lives that will take us far away from home. Jordan, you are the love of my life. Only God knows the experiences we have in front of us, but good or bad, there's no one I'd rather be with than you.