Epitome2: dedup for the masses

Marco Peereboom
OpenBSD

Abstract

As the proliferation, reliance and importance of rich digital formats have increased over the years, so have demands on data storage capacity. However, backup technologies have not kept up with this trend. The traditional timestamp based Towers of Hanoi backup methodology cannot handle the sheer volume of data and backup windows have been significantly reduced due to the 24 hour online economy. This methodology also backs up the same content repeatedly, even if the content has not changed. This results in a flood of data transfer that can overwhelm networks and other critical resources. Over the same period of time disk technology has progressed in leaps and bounds, both in performance and more importantly, in reliability. In comparison to disks, tape technologies are still comparatively slow and the media tends to deteriorate over time. Additionally, there are the future hardware compatibility issues of trying to match a degraded tape to a working tape drive, not to mention the physical issues such as offline labeling and storage. To work around these issues several new backup and archival paradigms have been developed, however, these are mostly out of reach of the open source community due to cost and licensing issues.

1 Introduction

Epitome is a set of building blocks that enables the creation of networked data backup solutions. It includes several tools to create deduplicated backups – these tools also serve as examples for application writers who wish to make use of the Epitome API. Epitome leverages some ideas from Plan 9's Venti[1] which was the original inspiration for the project.

The goal of the project is multifaceted. It intends to provide a viable alternative to tape backups using magnetic or flash disk storage, whilst being versatile enough to enable more sophisticated scenarios such as archival and Content Addressable Storage (CAS). It also strives to provide an interface that is both simple and "familiar". That said, the initial goal is to create the client/server components required to replace tar with a tool which creates deduplicated backups over the network using a low bandwidth protocol.

This paper will refer to both backup and archives. Industry uses both of these terms loosely, however they are distinctly different. In order to prevent confusion this paper will use the following definitions:

- *Backup:* A copy of primary data that can be used to restore content and/or application state after a data-loss event. This is typically a recurring activity that serves as an insurance policy for business continuity and/or private data sets.
- *Archive:* A collection of data retained for the long-term that defines the record of a business, application and information state. Archives are typically kept for mining, auditing, regulatory and compliance reasons rather than for data recovery.

Additionally, archives typically offer methods to associate rich metadata with the archived content. This is done to describe the content and enable future analysis. Some conceivable scenarios are:

- discontinued application or data format
- historical records
- evidence in a lawsuit

The open source community has made significant progress in technological innovation, however, in the realm of backup and archival storage solutions it has not. Currently such solutions are in the hands of vendors that sell exorbitantly expensive, proprietary systems that do not interoperate with open source software. These proprietary solutions are inaccessible for open source developers and are riddled with encumbering patents. The BSD community has been especially underserved due to lack of vendor interest. Epitome tries to fill this gap by providing the open source community with an alternative backup and archival solution that includes modern features such as:

- data deduplication and compression
- inherent data integrity
- flexible metadata handling

Modern file systems offer some backup and archival features however they are inherently not intended for data-at-rest. Despite implementing advanced and complex features to increase reliability (i.e. CRCs, ECC and parity) and to enable certain disaster recover scenarios (i.e. snapshots and replication), they are not bullet proof. They also do not have the capability to handle rich metadata that is typically associated with an archival solution. Other problems result from licenses and patents - the one truly advanced file system available via open source is really "pseudo open source" due to it being encumbered by

patents and being complex to port to other systems. It serves its purpose and does what it does well; however it is not useful as an open source backup or archival solution.

Inherent to file system backups is a file transfer protocol. The widely used ones, CIFS and NFS, are very chatty and when used in a backup scenario, transfer entire files just like a Towers of Hanoi backup strategy would.

In the past there have been some projects that have not made it beyond the prototype stage, with one positive exception being Plan 9's Venti. However, for a number of different reasons Venti has never really made it out of Plan 9 and is therefore not a viable open source solution either. The Epitome suite tries to pick up where Venti left off whilst providing several new solutions to previously unresolved problems.

Great care was taken to make Epitome as open source friendly as possible. The source code for Epitome is made available under the ISC license and is therefore not encumbered by license deception. All algorithms are simple and based on prior art, thus eliminating patent issues. Despite having complex code paths, the architectural and conceptual ideas behind Epitome are simple enough to be understood by anyone with a casual interest in backup and archival solutions.

Epitome was developed using the OpenBSD[2] development methodology. This means that the code is only as complex as it needs to be and no more. There are no hooks in the code for future additions - code will be written as required, not as anticipated. In order to be a good OpenBSD citizen the code has to be easily portable and be architecture and endian neutral.

The Epitome protocol is very lightweight and has only a few primitives. The design heavily favors bandwidth reduction over system resource usage. Each primitive consists of 12 bytes and, if required, a payload that is optionally compressed before it is sent over the wire. The protocol has a SCSI-like feel and even reuses some terminology. All network based communication is encrypted using SSL.

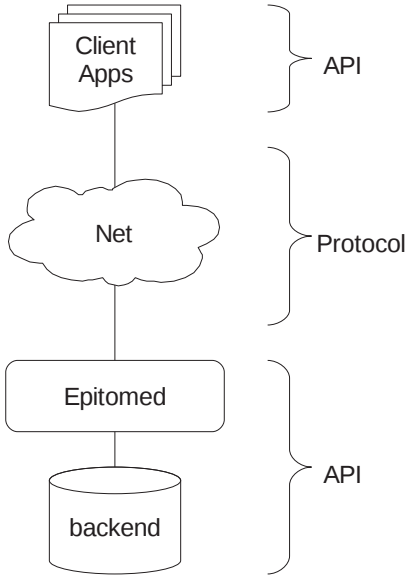
This paper details the design and implementation of the Epitome protocol and the associated applications which make use of its API. It also demonstrates a prototype of a viable backup and archival solution that makes use of magnetic disks instead of traditional tapes. We also offer an insight into the future plans for this project.

2 Background

The Epitome1[3] suite was written as a proof-of-concept data deduplication and compression engine. Whilst it proved to be an interesting solution, its usefulness was limited due to the lack of networking capabilities. Valuable lessons and insights resulted from the development process and guided the design of the Epitome2 suite.

The Epitome2 suite provides a client and server API which implements the protocol and low-level networking capabilities. In order to keep bandwidth usage to a minimum the Epitome protocol requires participation from both the client and server. The idea is that the client only issues commands to the server when needed and utilizes compression whenever possible.

The architectural overview of Epitome2 is as follows:



To keep the API uniform both the client and server code exists inside the same library and make use of the same functions. Depending on the context, functions may or may not be available. The Epitome backend is part of the server-side API and is extensible.

The Epitome suite assumes that the underlying storage devices are reliable and preferably offer some sort of periodic data integrity validation. For example it could be a high-end RAID card providing such services.

When the backend does run into a corrupt chunk it could do several things. For example, it could rename the chunk so that a subsequent write of the same chunk would restore integrity of the overall system. However, this is currently not part of the design. The assumption is that hardware is providing adequate data integrity.

3 epitomed

epitomed is the server-side workhorse of the Epitome suite and is a daemon that listens for incoming network connections. When a valid connection arrives a new process is forked in order to handle the session. Session parameters are negotiated between the client and server via a client request – server response mechanism. Once this step is complete the client can start issuing commands to the server.

Without delving into the details of the protocol, a typical archival session consists of a collection of commands that may result in data being written to the Epitome backend. As the header and payload are received by the server, the payload is uncompressed, if required, and the digest is calculated. Before attempting to write the payload to the backend a check is performed to ensure that a copy of the payload does not already exist. If the payload does not exist it is written to the backend and the digest is returned over the wire to the client. As per the protocol details given in section 5, the client should not issue write requests without first verifying that the digest does not already exist.

An archival retrieval begins by reading the data payload from the backend. Depending on the server configuration and the options specified within the request, the data chunk may be verified and/or decompressed before being sent over the wire. However, if the payload was saved uncompressed it will always be sent uncompressed.

Currently Epitome only supports the zlib[4] compression algorithm. The reason for this choice was due to the simplicity, robustness and speed of the algorithm. It is a well established and robust library with a reasonably simple interface and is light enough to use even on slower hardware.

Currently *epitomed* implements a maildir-like backend. Each chunk is written to a file which has a filename derived from the digest of the uncompressed chunk data. Each chunk has a simple header followed by the raw data which can be optionally compressed. This header field is written using External Data Representation (XDR[5]) to ensure that the data is portable. The header contains the following information:

- compressed size
- uncompressed size
- flags that indicate how the chunk was saved

A maildir-like backend is sub-optimal for a deduplication archive. This is due to several factors including file system block-size and inode starvation. However, it is very simple to implement and has proven robust for mail applications.

The backend does not provide any services besides reading and writing the header and data payload. The only exception is when the backend is prepared for use; see section 4.2.

The backend has a driver-like implementation and can therefore be easily replaced by something that is much more sophisticated. Only a handful of functions need to be implemented. The functions that the backend must provide are:

- Open
  - Open the backend for use. Performs minimal verification that the backend is ready to use. This function is called when a valid session is established.
- Close
  - Close the backend. All outstanding I/O is flushed and synchronized. This function is called when a session is torn down.
- Read
  - Read the header and data payload from the backend for the specified digest.
- Write
  - Write the header and data payload to the backend for the specified digest.
- Exists
  - Read the header from the backend for the specified digest.
- Create
  - Prepare the backend for first use. This is a destructive operation and is not intended for runtime use.

3.1 Security Considerations

epitomed was designed with security in mind and currently it supports two modes of operation. If run with an EUID of 0 the daemon will chroot and drop privileges. If desired the daemon can be run with a non-privileged EUID which allows the daemon to run in a non-chroot environment.

All network traffic to and from the daemon is encrypted using SSL provided by Agglomerated SSL[6] - an easy to use wrapper for the standard OpenSSL[7] library. All communication requires valid certificates for the CA and both the client and server.

What may come as a surprise is that data-at-rest is not encrypted. This might sound counter intuitive, however, it is done to allow for hash collisions on identical data, which in turn enables the deduplication algorithm. Since the hashed chunks are used by all clients one could argue that there is a risk of guessing digests. However, with the SHA1 algorithm this is in the order of 2^160 per digest. Enumerating such a large name space is unlikely to be practical at the time of writing. In the future the Epitome suite will allow for the use of other hashing algorithms or much more sophisticated fingerprinting algorithms.

The file system based backend is only as secure as its permissions. Therefore the *epitomed* server administrator needs to ensure that the permissions of the target directory are configured appropriately.

Another area that needs consideration is metadata. In the current version the client is responsible for all metadata handling. The server does not interpret the metadata and therefore the client can safely encrypt it.
Metadata is not size checked by the server and can therefore be arbitrarily big. This is an issue that will be resolved in the future.

3.2 Options

To remain true to the KISS development methodology the *epitomed* server has only a few available options. Currently they are:

- max_chunk_size
  - This designates the maximum data payload size. There is a balance between compressibility and dedupibility; typically the larger the payload the higher the compressibility but lower dedupibility and vice-versa.
- queue_depth
  - This determines how many commands a session can have outstanding. This value must be negotiated with the client.
- allow_uncompressed_reads
  - This allows the server to read data from the backend and uncompress it before it is sent over the wire.
- force_uncompressed_writes
  - With this flag set the *epitomed* server will write all data uncompressed on the backend. This has a direct result on future reads since those will send the data verbatim from the backend to the client. Data payloads will not be compressed on future reads.
- be_type
  - This designates the type of the backend. Currently only “file” is supported.
- be_name
  - This has a different meaning depending on the *be_type*. Since only file is supported at this time this option specifies the target directory for the backend. For example: /var/epitome.

These options are read out of a human readable configuration file that uses the familiar *option=value* format. This file is read once at launch. Even though the settings of an *epitomed* server can change throughout the life-cycle it is recommended to plan accordingly instead.

Note: the backend configuration cannot be changed once it is setup.The configuration file can also be overridden using environment variables.Currently the certificate location is not specified in the configuration file and is a command line option only.

4 Applications

Currently the Epitome suite only ships with one tool, *epitomize*. *epitomize* serves a dual purpose. First and foremost it is an archiving tool intended for the end-user. Secondly, it is an example of how to use the Epitome API. Libepitome is where Most of the code for the entire suite resides in *libepitome*. *epitomize* uses this library in order to generate, transfer, validate and receive commands.

4.1 epitomize

epitomize is a tar-like utility. It strives to reuse as many of tar’s command line options as possible in order to provide a familiar interface. The difference being that *epitomize* interacts with the *epitomed* daemon which results in data being deduped and archived instead of being stored in a local file. The result of this operation is either an “archive token” or a metadata file.

A metadata file contains all of the information needed to reconstruct (a.k.a. rehydrate) the original archive. Once again, this file is written using XDR to ensure portability across different platforms. When using the archive token method the resulting metadata is transferred to the *epitomed* server. The metadata transfer is not limited by the max_chunk_size setting.

Saving the metadata on the e*pitomed* server has the benefit that a whole archive can be reconstructed on the server-side, however, if it falls into the wrong hands it can allow a malicious person to reconstruct an archive. It also adds to the bandwidth requirement since it travels back and forth between the client and server. Since the *epitomed* server does not delete anything that has been saved it will also lead to increased storage.

It needs to be noted that if the metadata or the archive token are lost then there is no way to reconstruct the original archive.

4.2 Options

The client shares some options with the server and has a few of its own.

The shared options with *epitomed* are:

- max_chunk_size
- queue_depth

Additional options:

- allow_uncompressed_writes
  - If allowed by the server send uncompressed data payloads to the server.

4.3 eprepare

eprepare is a sideband tool that has to be used prior to *epitomed* deployment. Its only intent is to call the Create function of the backend with appropriate parameters in order to initialize the backend (i.e. create directories, setup a database, etc). When using the file backend it creates all directories in the target directory that are used in the maildir-like interface.

It reuses the *epitomed* configuration file for backend specific parameters. Although all options can be overridden via the command line if required.

5 Implementation

The implementation of the Epitome suite resides mostly in a dynamically loaded library. The application writer needs to provide callbacks to perform functions on behalf of the library. The intent is to have libepitome be generic enough so that alternative and third-party applications can be easily developed.

The code is written in C using the Keep It Simple Stupid (KISS) principle. All code is Finite State Machine (FSM) based and every time the queues are evaluated each I/O is pushed to the next state, if possible. This was done to enable full asynchronicity in the code whilst eliminating the need for concurrency. This decision was made in order to avoid timing and other hard to debug issues. For all intents and purposes Epitome is an I/O system and therefore reliability is of the utmost concern. To keep latency as low as possible, techniques such as zero-copy are used where applicable.

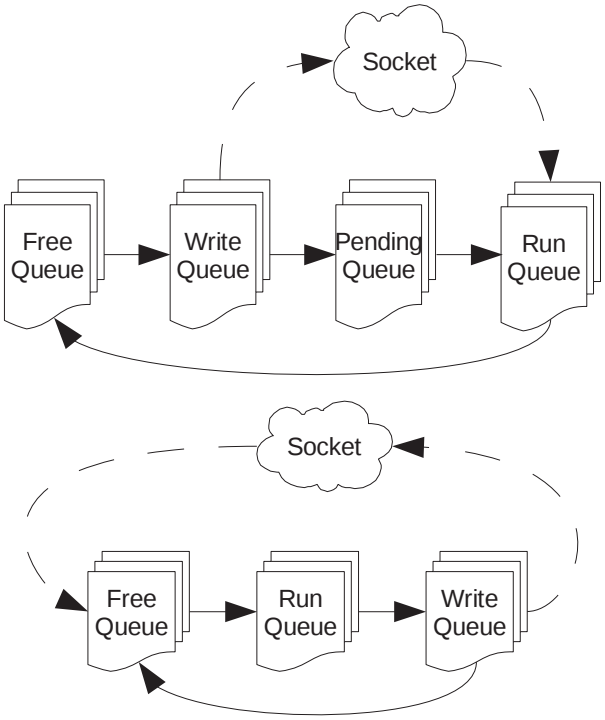
The library has several dependencies and requires the following libraries:

- zlib
- ASSL
- OpenSSL

Additionally the code uses the highly portable queue(3) and tree(3) macros. All queues are implemented using TAILQ and there are several Red Black trees used within the code.

The library also has some debugging features such as memory debugging, memory painting and logging capabilities.

The following diagrams outline the execute-to-wire I/O progression through the queues. Dashed lines indicate network traffic.



5.1 Protocol

The Epitome protocol is what provides all of the nuts and bolts for data transfer and bandwidth mitigation. All commands that travel to the server will be returned to the client with a success or failure indication and a payload, if required. The header in both directions is identical and its meaning is contextual based on the opcode. All headers have a fixed size of 12 bytes and travel the network in traditional network byte order.

Generic header:

Byte 0	Byte 1	Byte 2	Byte3
Version	Opcode	Status	EX status
Tag		Flags	
Size			

The version field indicates which Epitome protocol version is being used. This is intended to prevent future interactions between different protocol revisions which could have a negative impact. Currently only protocol version 1 is allowed and any other version will be rejected.

The opcode field designates the context of the header and

the payload. Depending on the value, individual fields, flags and payload will have a different meaning. The opcode is always an even number for client requests and an odd number for server replies. Both the client and server will reject and terminate the connection if they receive a non-supported command; this includes commands that are invalid in their respective context.

The status byte indicates success or failure. If it indicates failure then the EX status field contains an extended error code which narrows the failure down so that it can be used to determine a recourse.

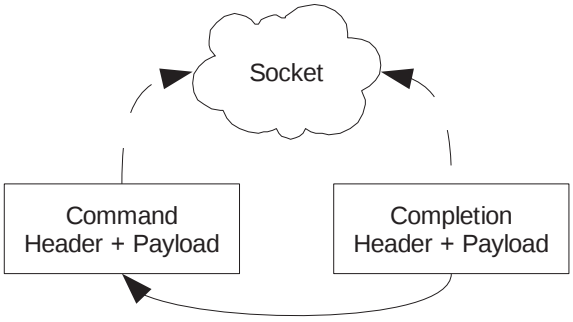
The tag field is a unique identifier for this command. The client is not allowed to issue a duplicate tag to the server. When a command arrives at the server the matching tag is popped off the free-queue. This implies that the client/server tag are always exactly the same.

The flags field qualifies opcode specific hints. See individual primitives for an explanation.

The size field indicates the payload size, if required. This size is used as a hint when streaming commands off the wire into the run queue. Therefore a generic command and completion looks conceptually like the following:

Header	Payload

All commands are initiated by the client and have a completion that is generated by the server based on the result of the client command.



5.2 Primitives

The following sections contain a detailed description of all protocol primitives.

5.2.1 NEG(2)

The NEG command requests a queue depth and max_chunk_size. The server will try to honor the client request but will override it if the client requested values exceed the server settings.

The negotiation process is based on a client request, server dictates mechanism, meaning that the client has to comply with the server limitations. If the client issues commands outside of the negotiated parameters the server will terminate the connection.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x02	N/A	N/A
Requested Queue Depth		N/A	
Requested max_chunk_size			

5.2.2 NEG_REPLY(3)

The NEG_REPLY is the server reply to a NEG command. If possible it honors the client request, if not it returns the overridden values. The client shall honor these values.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x03	N/A	N/A
Negotiated Queue depth		N/A	
Negotiated max_chunk_size			

5.2.3 NOP(10)

The client sends a header plus a 4 byte payload containing a 32 bit unsigned integer. The server shall reply to this command with the same NOP_IP + 1.

The NOP command is used under several scenarios. Typical usage is “alive” or heartbeat monitoring. Other uses include measuring round trip time latency, etc.

epitomize uses a NOP that is designated as the last command of the session to ensure that all commands have been sent and received by the server. It also carries the “last command” designation in the API which then enables the ordered tear down of a completed session.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x10	0x00	0x00
Tag		0x0000	
0x00000004			
NOP_ID			

5.2.4 NOP_REPLY(11)

The server replies to a NOP command with a

NOP_REPLY. The NOP_REPLY returns the NOP_ID that was provided in the NOP command with NOP_ID + 1. The NOP_REPLY has a 4 byte payload that contains a 32 bit unsigned integer.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x11	0x00	0x00
Tag		0x0000	
0x00000004			
NOP_ID + 1			

5.2.5 EXISTS(12)

The EXISTS command is used to determine if a digest exists at the server. It has a 20 byte payload that contains the digest that is being located.

If the flag is set to VERIFY_DIGEST and the digest exists, the server will attempt to uncompress the associated chunk and recalculate the digest value to ensure data integrity.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x12	0x00	0x00
Tag		Depends	
0x14			
Digest[0x00 .. 0x13]			

5.2.6 EXISTS_REPLY(13)

The EXISTS_REPLY command returns the status of an EXISTS command.

If the VERIFY_DIGEST flag was set then the digest is read from the backend, uncompressed and verified against the digest that was provided as the payload of the originating EXISTS command.

1. If the verification succeeds then the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status field and the flags field will report if the chunk resides compressed on the backend and/or if the chunk is designated as metadata.

2. If the verification of the digest fails then the server replies FAILED (0x01) in the status field (byte 2), INVALID_DIGEST(0x03) in the extended status field (byte 3) and the flag will be set to 0x0000.

3. If the digest does not exist the command returns FAILED (0x01) in the status field (byte 2), DOESNT_EXIST (0x02) in the extended status field (byte 3) and the flags will be set to 0x0000.

If the VERIFY_DIGEST flag was not set then the backend only determines if the digest exists or not:

1. If the digest exists the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status field (byte 3) and the flags field will report if the chunk resides compressed on the backend and/or if the chunk is designated as metadata.

2. If the digest does not exist the server returns FAILED (0x01) in the status field (byte 2), DOESNT_EXIST (0x02) in the extended status field (byte 3) and the flags will be set to 0x0000.

The EXISTS_REPLY command never contains a payload.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x13	Depends	Depends
Tag		Depends	
0x00000000			

5.2.7 READ(14)

The READ command is used to read a chunk from the server. It has a 20 byte payload that contains the digest of the desired chunk.

The flags field may have the COMPRESSED flag set in order to request compressed data from the server. This is used as a hint only and the server will determine how it sends the data payload.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x14	0x00	0x00
Tag		Depends	
Digest[0x00 .. 0x13]			

5.2.8 READ_REPLY(15)

The READ_REPLY command returns the status and if possible, the chunk data in the payload section.

If the VERIFY_DIGEST flag was set then the digest is read from the backend, uncompressed and verified against the digest that came as the payload of the originating READ command.

1. If the verification succeeds then the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status field and the flags

field will report if the chunk was sent COMPRESSED (0x01) over the wire. The size field will contain the size of the data payload.

2. If the verification of the digest fails then the server replies FAILED (0x01) in the status field (byte 2), INVALID_DIGEST (0x03) in the extended status field (byte 3) and the flag will be set to 0x0000. The size field is set to 0x00000000.

3. If the digest does not exist the command returns FAILED (0x01) in the status field (byte 2), DOESNT_EXIST (0x02) in the extended status field (byte 3) and the flags will be set to 0x0000. The size field is set to 0x00000000.

If allow_uncompressed_reads is set then the server will attempt to uncompress the data from the backend and send it uncompressed over the wire. If it is not set then the server will return what is in the backend verbatim and set the COMPRESSED (0x01) flag accordingly.

Note: if the max_chunk_size is unaligned then the client might be unable to uncompress the payload since it only allocates max_chunk_size + uncompress boundary bytes.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x15	Depends	Depends
Tag		Depends	
On Failure 0x00000000 on Success N			
Payload			

5.2.9 WRITE(16)

The WRITE command sends a data payload to the server. The flags must indicate if the data was COMPRESSED (0x0001) before being sent over the wire. The size field shall be set to the payload size.

Note: a client application should always send an EXIST command before sending a WRITE command in order to minimize bandwidth usage. See the API section for more information.

Byte 0	Byte 1	Byte 2	Byte3
0x01	0x16	0x00	0x00
Tag		Depends	
Payload Size			
Payload			

5.2.10 WRITE_REPLY(17)

The WRITE_REPLY command returns the status of the WRITE command. Upon arrival the data payload is uncompressed, if necessary and the digest is calculated. If the digest does not already exist then the data payload will be saved to the backend.

1. If the digest already exists then the server returns FAILED (0x01) in the status field (byte 2), EXISTS (0x01) in the extended status field (byte 3) and the flags will be set to 0x0000.

2. If the digest does not exist then the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status field and COMPRESSED (0x0001) in the flags if the data was saved compressed to the backend.

Depending on the force_uncompressed_writes setting it will save the data payload to the backend either compressed or uncompressed.

Byte 0	Byte 1	Byte 2	Byte3
Version	0x17	0x00	0x00
Tag		Depends	
0x14			
Digest[0x00 .. 0x13]			

5.2.11 WRITE_MD(18)

The WRITE_MD command sends a metadata payload to the server. The flags must indicate if the data was COMPRESSED (0x0001) before being sent over the wire. The size field shall be set to the payload size. The WRITE_MD command is not bound by max_chunk_size. The flags field must have the METADATA (0x0004) designator. The client must send the uncompressed size as a 4 byte unsigned integer in front of the metadata payload.

Byte 0	Byte 1	Byte 2	Byte3
Version	0x18	0x00	0x00
Tag		Depends	
Size metadata + 4			
Uncompressed size			
Metdata			

5.2.12 WRITE_MD_REPLY(19)

The WRITE_MD_REPLY command returns the status of

the WRITE_MD command. Upon arrival the metadata payload is uncompressed, if necessary and the digest is calculated. This digest is also known as the “backup token”.

If the digest already exists then the command will result in a failure.

1. If the digest already exists then the server returns FAILED (0x01) in the status field (byte 2), EXISTS (0x01) in the extended status field (byte 3) and the flags will be set to METADATA (0x0004).

2. If the digest does not exist then the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status and METADATA (0x0004) in the flags field.

Byte 0	Byte 1	Byte 2	Byte3
Version	0x19	Depends	Depends
Tag		Depends	
0x14			
Digest[0x00 .. 0x13]			

5.2.13 READ_MD(20)

The READ_MD command is used to read a metadata payload from the server. It has a 20 byte payload that specifies the desired backup token.

The flags field may have the COMPRESSED flag set to request compressed data from the server. This is used as a hint only and the server will determine how it sends the metadata payload.

Byte 0	Byte 1	Byte 2	Byte3
Version	0x20	0x00	0x00
Tag		0x0000	
0x14			
Digest[0x00 .. 0x13]			

5.2.14 READ_MD_REPLY(21)

The READ_MD_REPLY command returns the status and if possible the metadata along with the uncompressed size in the payload section.

If the VERIFY_DIGEST flag was set then the metadata is read from the backend, uncompressed and verified against the digest that was provided as the payload of the originating READ_MD command:

1. If the verification succeeds then the server returns OK (0x00) in the status field (byte 2), NONE (0x00) in the extended status field and the flags field will report if the chunk was COMPRESSED (0x01) before being sent over the wire. The flags will contain the METADATA (0x0004) designator. The size field will contain the size of the data payload.

2. If the verification of the digest fails then the server replies FAILED (0x01) in the status field (byte 2), INVALID_DIGEST (0x03) in the extended status field (byte 3) and the flag will be set to METADTA (0x0004). The size field is set to 0x00000000.

3. If the digest does not exist the command returns FAILED (0x01) in the status field (byte 2), DOESNT_EXIST (0x02) in the extended status field (byte 3) and the flags will be set to METADATA (0x0004). The size field is set to 0x00000000.

If allow_uncompressed_reads is set then the server will attempt to uncompress the data from the backend and send it uncompressed over the wire. If it is not set then the server will return what is in the backend verbatim and set the COMPRESSED (0x01) flag accordingly. The server must send the uncompressed size as a 4 byte unsigned integer in front of the metadata payload.

Byte 0	Byte 1	Byte 2	Byte3
Version	0x21	Depends	Depends
Tag		Depends	
Size of metadata + 4			
Uncompressed size			
Metadata			

5.3 Typical Client/Server exchanges

Following are a few examples of client server exchanges under different circumstances.

A fresh backup *without* metadata handling:

EXISTS -> EXISTS_REPLY
WRITE -> WRITE_REPLY
...
EXISTS -> EXISTS_REPLY

WRITE -> WRITE_REPLY
NOP -> NOP_REPLY

The NOP is used to drain the client side queues.

Same backup *after* the initial run:

EXISTS -> EXISTS_REPLY
...
EXISTS -> EXISTS_REPLY
NOP -> NOP_REPLY

Same backup *after* the initial run *with* metadata:

EXISTS -> EXISTS_REPLY
...
EXISTS -> EXISTS_REPLY
WRITE_MD -> WRITE_MD_REPLY
NOP -> NOP_REPLY

Restore operation:

READ -> READ_REPLY
...
READ -> READ_REPLY
NOP -> NOP_REPLY

Restore operation *with* metadata:

READ_MD -> READ_MD_REPLY
READ -> READ_REPLY
...
READ -> READ_REPLY
NOP -> NOP_REPLY

6 Related Work

There is very little related work in the form of open source projects. Besides Venti, only one known project appears to have been sufficiently developed to be useful – lessfs[8] is a "high performance inline data deduplicating" file system that has been developed for the Linux kernel.

All other projects appear to have either stalled or be in an alpha/announce phase.

7 Future Work

The Epitome2 suite is really just the beginning. It enables the creation of many different applications and can be readily extended. The plan is to evolve the protocol and accompanying applications over time. The following sections describe different ideas that are under consideration for future work.

7.1 Applications

The protocol is simple however it facilitates many diverse applications. Some ideas that are under considerations are:

- Deduplicating File System
  - The idea here is to create a backup file system for client applications. For example, one could envision simply copying all relevant files onto such a file system on a daily basis and let it store all changes.
  - An extension to this would be the ability to detect changed files and keep all versions of these files in a read-only directory.
- CDP (Continuous Data Protection)
  - This builds on the previous file system idea, however, this would run on the client machine. As files are opened and closed (with some debouncing heuristics) they are backed up continuously.
- VTL (Virtual Tape Library)
  - This is an interesting idea that requires moving some of the code into the kernel using some form of virtual HBA (such as softraid[9]) which in turn emulates a tape library. This would enable deduplication for backup applications that do not support it natively.

7.2 Protocol

Several additions to the protocol are being considered such as:

- DELETE
  - If this primitive is added it will be a non-trivial addition. There are many complexities with reference counting that will need to be solved before it can be implemented. Under certain conditions (external metadata saving, encrypted metadata, etc) this simply cannot be done.
- CLOSE
  - Currently the method to terminate a session is to simply close the socket. This is adequate for current applications but in the future an application will likely need a specific primitive.
- SEARCH
  - The idea here is to allow the user to initiate a metadata search from a client to the server using something like Pinot[10].

Currently the protocol only supports zlib compression and SHA1 digests - this should be expanded to include more efficient (and slower) compression algorithms and different deduplication algorithms. The protocol is designed to run

inline and therefore not every algorithm will be adequate. The infrastructure for this does not currently exist in the code. Protocol wise it comes down to an expanded NEG command in order to allow the client and server to agree on what algorithms to use.

All data is written as sent from the client. This is adequate in most scenarios however it may be necessary to encrypt data-at-rest. Such a scenario will disable many other features and will require modifications to the protocol as a result.

A more radical idea is to use HTTP as the transport mechanism for the current Epitome protocol. There are several reasons why this is under consideration:

- It is well understood
- It can travel through proxies
- It is human readable
- It is becoming the transport of choice in the archiving world

The primitives would need to be translated into generic HTTP requests while some of them would need to become commands that travel over HTTP. Despite the drawbacks and complexities, the idea is attractive for the above mentioned reasons.

7.3 Backend

The current backend code is written with expansion in mind. It has the look and feel of a driver and is therefore relatively simple to expand. Some possibilities are:

- Cloud backend
  - Clustering for horizontal scaling
- Segregation of backends for cloud applications

7.4 Content Addressable Storage

Content Addressable Storage (CAS) is a separate beast altogether, however, the primitives overlap significantly with requirements that a CAS system would need. Bolting on these additional pieces will be relatively easy.

The major use-case for CAS is “regulatory compliance” and “retention”. The complexities of such a system are mostly external to the protocol, for example the policy engine and metadata generation. This would require the protocol to provide a DELETE command along with several other commands to manipulate metadata.

8 Conclusion

Deduplication is very easy to prototype, however it is very hard to move beyond that phase. The networked code was prototyped three times before it performed at an adequate level. Various approaches were tried and failed for different reasons.

It is hoped that this paper and the accompanying code will spur interest and move deduplication out of the vendor-only realm. The library is easy portable to a myriad of different operating systems and should be able to provide the building blocks for other interesting projects. The planned future work will add additional primitives, further enhancing the protocol.

9 Acknowledgments

I’d like to acknowledge several people in the OpenBSD community that made this whole endeavor possible. Joel Sing for his unending patience listening to me yack and providing valuable ideas and insights. He also read through the first drafts and translated them into actual English. Jacek Masiulaniec for working on Epitome1 and making *epitomize* a much more robust tool. Kenneth Westerback for proof reading this paper. Theo de Raadt for his supportive yelling and seemingly unending energy and devotion to keep OpenBSD running allowing me to play in the most intellectually stimulating open source community in the world.

Jacob Yocom-Piatt, you know why mate.

I’d like to acknowledge Clarissa, my lovely wife, who puts up with my open source addiction and even takes the time to read and comment on technical papers such as this.

Finally, my lovely daughter Holland who always manages to brighten my day.

References

[1] Venti http://doc.cat-v.org/plan_9/4th_edition/papers/venti/
[2] OpenBSD http://www.openbsd.org/
[3] Epitome1 http://www.peereboom.us/epitome/
[4] Zlib http://www.zlib.net/
[5] XDR http://www.ietf.org/rfc/rfc1832.txt
[6] ASSL http://www.peereboom.us/assl/html/assl.html
[7] OpenSSL http://www.openssl.org/
[8] lessfs http://www.lessfs.com/
[9] Softraid http://www.openbsd.org/cgi-bin/man.cgi?
query=softraid&apropos=0&sektion=0&manpath=OpenBSD+Current&arch=i386&format=html
[10]Pinot http://pinot.berlios.de/