BackStage

LISA V - San Diego
Work in Progress

Frank Kardel

Oktober 91 2/92

BackStage

Frank Kardel
kardel@informatik.uni-erlangen.de

Friedrich-Alexander-Universität
Erlangen-Nürnberg
Martensstraße 1 • D-W8520 Erlangen • Germany

ABSTRACT

Over the years a lot of efforts have been undertaken to solve the backup and archiving problem. Among all available systems only a few address the problems involved in dealing with networks of heterogenous computers (especially different network implementations, security, crash resistance). In the Unix[1] community there are some accepted Backup systems (mainly dump / restore) which unfortunately are not available from all vendors. This mostly leads to local solutions mainly based on the available Unix tools (dump / restore, cpio, tar, and many more).

The great variety of differences in vendor support and the wish to have a uniform backup / archive system leads to the conclusion that a portable, extensible backup / archive system is needed. Experiences with locally designed backup systems have shown that these requirements can not be fulfilled by using standard Unix tools. A project for designing a completely new backup / archive system, tailored for the needs of the University of Erlangen, was started.

The main design goals are: portability, independence from network implementation, extensibility, proper authentication, automated restorations that can be started by the users with minimal operator burden and an improved user interface.

The following paper describes the structure of a combined backup / archive system that is currently being developed.

1. Unix is a Trademark of AT&T Bell Laboratories

1. Introduction

BackStage (from: “The Show must go on”) has been designed to allow easy access to removable mass storage media like tapes and optical disks in a networked environment. Its major goals are the easy handling of different mass storage media for storing software packages in different versions and the efficient handling of incremental backup.

The decision for implementing a whole new backup / archive system stems from the need to handle a constantly increasing number of file servers within the faculty and the increasing software installation base. Since the equipment consists of machines from different vendors running different versions of Unix and other operating systems and only a very limited number of people available for system administration, we are forced to have a consistent network wide backup / archive system. This cannot easily be accomplished with the tools provided by the different Unix versions. Most available backup and archive systems lack at least one of the following:

- proper authentication and security
  (especially problematic in networked backup / archive systems)
- tape handling
  (free, used tapes, tape contents, tape labels)
- concurrent backups
  (writing / retrieving a backup / archive on multiple devices)
- flexible job scheduling
  (user needs a very important restoration done, but the device is being occupied by a total backup of the whole network for another 18 hours)
- operator interface
  (an easy to comprehend interface should exist to find out which media is needed in which device)
- recovery mechanisms for destroyed databases of the system
  (what happens if you lose your tape directory database)
- user controlled restoration
  (usually the operator should not bother with a restoration request except for inserting the media)
- extensibility
  (supporting new mass storage devices and improving functionality should not result in an entire system redesign)

Many of these features are present in most of the mainframe software, but up to now these features are missing from the average Unix system. Even the backup system shipped with Unix System V.4 was not designed with networks in mind.

The decision for building an integrated backup/archive system is rooted in the fact that both tasks require many common actions like media management, device management and retrieval. In our opinion these common tasks need not be duplicated in two different systems.

2. Overview

The *BackStage* system has been broken up into three separate modules, each providing an abstraction for the higher level modules. All modules rely on a common authenticated communication subsystem that allows an abstraction from the network implementations used. A major benefit of authenticated network connections is the ability to check access permissions. Thus it is possible to allow the users to do their own archiving and backup restoration. This greatly relieves some of the burden on the operator personnel.

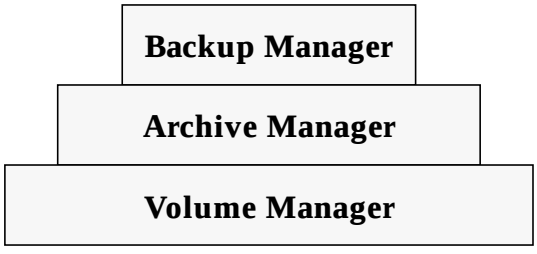
2.1. System Structure

BackStage is layered and consists of three packages. The lowest layer is the *Volume Manager*, which will handle all actions associated with data storage and retrieval on the mass storage media. It also provides the networking mechanisms needed to transfer data between the different hosts and the storage devices. Networking mechanism and device handling modules are designed for high speed data transfers by employing multiple buffer techniques. Another advantage of the *Volume Manager* is that it can do all media maintenance (media replacement, media copy) without introducing these mechanisms into the other modules.

The second layer, the *Archive Manager*, allows the storage of different versions of a set of files. These versions can be stored in different locations in order to achieve redundancy in case the mass storage media fails or gets lost. The *Archive Manager* relies on the *Volume Manager* for its services.

The third layer is the *Backup Manager*, which uses the *Archive Manager* for the backup of all participating nodes in the network.

Each component has two interfaces. The first is the programming interface in form of a library, which allows access to the services of each component. The second interface is a set of Unix tools, based on the programming library, to allow easy integration of *BackStage* into the Unix environment (such as shell scripts).



2.2. General Mechanisms

Two fundamental mechanisms are used throughout *BackStage*. These are *transactions* and *authentication*. Transactions provide a consistent failure model, while authentication allows for mainly unsupervised secure operation.

2.2.1 Transactions

BackStage uses a transaction based client-server model. This mechanism is used to provide consistent failure handling. Since *BackStage* is used within a network of computers, any system component, such as the *Volume Manager*, the *Archive Manager* or the *Backup Manager* can fail for several reasons. These failures arise from programming errors, network partition, resource exhaustion or operating system crashes. Since high reliability and error tracking are the key issues in any backup / archive system we chose the transaction based approach to provide crash resistent operation. Any long lasting operation, such as a *write-file* transaction on the *Archive Manager*, will create a transaction within the daemon and return a *TID* (*T*ransaction *ID*entification) to the client. The *TID* allows the location of a particular service (transaction) even if a server crash has occurred in the mean time. *TIDs* not only allow the location of a service, but also limit the number of orphaned transactions by including an expiration time stamp.
The *TID* thus has a threefold use:

- naming a transaction
- locating the server for the transaction
- control orphaned transactions by expiration

2.2.2 Authentication

Authentication is one of the most neglected issues in Backup / Archive systems currently used in the Unix community. Probably the fact, that almost all data has to be stored on mass media and thus needing a “super-user” is the reason for missing or inadequate authentication mechanisms. It is likely, that most Backup / Archive mechanisms added later rely on the Berkeley *rcmd* authentication when being used in networked environments (*rdump*). The *rcmd* authentication and the fact, that the “super-user” is needed to perform Backup, pose a severe security problem. The “super-user” is still needed for Backup on a particular host, but this does not mean granting “super-user” access from other hosts because of using a centralized Backup / Archive system. Proper authentication allows a Backup / Archive system controlled access to particular hosts. Any host may choose to write out Backup / Archive data in encrypted form, which allows hosts with sensitive data to use a centralized

Backup / Archive system. This may not hold for high security applications, but might provide an extra level of protection, provided a decent encryption algorithm is used.

Basically authentication is performed, whenever a network connection is established. At this time a telnet option negotiating type protocol is used to find out which kinds of authentication procedures should be done. The server will select one or more authentication mechanisms offered by the client. After that the authentication data for each of the selected mechanisms will be exchanged. Each successful exchange leads to an authentication entry on the server side stating the authenticated entity and the authentication mechanism. This data structure can then be used for authorization purposes such as granting access to certain transactions or data via access control lists.

Currently three authentication mechanisms are planned: ANONYMOUS, UNIX, KERBEROS. Due to the modular design it should be possible to extend this list, should more and better authentication mechanisms arise.

Good authentication mechanisms allow for an exchange of a session key (*kerberos*). This feature can be used to encrypt the used network communication paths, if needed.

3. Volume Manager

The *Volume Manager* is responsible for all media handling actions. It does all the bookkeeping pertaining to files and their respective media. In order to be able to write data of (almost) arbitrary length onto almost any media the *Volume Manager* creates the abstraction of v-files being stored on a volume.

A volume may be one of several media types, such as disks or tapes, whereby the physical characteristic of the media must be consistent within a particular volume. All free space management and media locating is done by the *Volume Manager*. V-files have an attribute section, which allows storage of additional information such as access permissions, v-file name, v-file format and the like.

There are two types of attributes.

- The mandatory attributes, like v-file name and ACL (access control list), have to be provided for a file in order for the *Volume Manager* to be able to create and locate the v-file.

- The optional attributes can be used by applications to store additional data like the file type or the encryption method used.

The attribute mechanism allows for easy extension of the system. Attribute information is duplicated on the media and in an on-line database.

3.1. Terminology

The *Volume Manager* defines a certain set of entities that can be used by the *Archive Manager* and the *Backup Manager*. These entities represent a stronger abstraction from the underlying media, thus allowing for consistent extension of device support without changing higher level applications.

- Volume
  A set of v-files that have the same physical characteristics (video tapes, reel tapes, disks, floppy disks)
  It is used to hold a collection of v-files and is referenced by a volume name.

- v-file
  A v-file is stored in a volume. It may be of arbitrary length. If the v-file length exceeds the capacity of the media used by a volume, it will be transparently split to occupy multiple media. A v-file may only be written sequentially. Random access to different file portions is not supported.
  It is possible for a v-file to belong to a *v-file-group*. This is an optional attribute attached to the file when it is created in a volume. Only files belonging to the same file group may reside on the same physical media. This mechanism allows higher level applications to control the placement of files in order to control the degree of media redundancy.

3.2. Transactions

The *Volume Manager* provides several basic operations on volumes:

- Volume creation and deletion
- Volume attribute management
- V-file writing, reading and deletion
- V-file attribute management
- Transaction handling (list, abort)
- Configuration

3.3. Transaction Scheduling

Since the volume daemon will manage all reading and writing of v-files using one or more devices (which may be spread throughout the network), and since many different requests may be waiting for media at the same time, it is necessary to have a flexible scheduling mechanism for the pending transactions. Usually the minimization of media change is the strategy to choose. But there are situations where it might be better to process an urgent request as soon as possible. It is very difficult to devise an optimal policy for handling requests. It is for this reason that a configurable scheduler is being developed. The scheduler will sort the requests according to a partial order. Requests falling into different classes will be processed in strict sequence, while requests within the same class compete for resources and may thus run in parallel provided enough devices

are present. Since the ordering is configurable, it is possible to change the transaction processing order at runtime via a configuration transaction in order to accommodate exceptional conditions. This makes manual intervention using the *vlink* interface possible.

3.4. Interfaces

The *Volume Manager* can be accessed via two applications and the library interface. One application, the *vsh*, provides a very simple interface to the volume operations. The *vsh* is intended for use with shell scripts or for interactive use. The second application, *vlink*, is the operator interface to the *Volume Manager*. *Vlink* is used to display the state of all requests pending within the *Volume Manager* and also the required media for these requests. Its main purpose is to allow for an operator who manages the insertion and removal of tapes without having complete insight into the entire backup and archive system. Another use of *vlink* is the on-line configuration of the *Volume Manager*.

4. Archive Manager

The *Archive Manager* is responsible for managing sets of a-files. These represent the file trees as seen in most operating systems.

4.1. Archive File Tree

Basically, a-files represent a generic view of file system objects. The term file system object refers to all entities found in an actual file system implementation (directories, files, isam-files, symbolic links, etc.). An a-file is the name of a file system object, whose representation may be stored in different versions and incarnations on external media. A new version of an a-file is created whenever the file system object changes its contents. When the type attribute of the file system object changes or the file system object is deleted a new incarnation is created.

An a-file consists of a name and set of attributes. These attributes describe properties of the file system object like the access permissions or the file size. Attributes are not limited to a particular operating system implementation. An *Archive Manager* can thus manage archives from different operating systems. The program that moves file system objects onto external storage via the *Volume Manager* mechanism is called the file-packer. Its task is very operating system dependent, as it has to cope with all file system peculiarities such as special devices, context dependent files, conditional symbolic links and many more. It is unlikely that a common file system standard will be developed in the near future. Therefore the *Archive Manager* uses the attributed file type view to allow for all these extensions, while the file-packer will

cope with specific implementation details. It is the responsibility of the file-packer to save enough information in order to re-create the file system object.
Only a subset of the possible attributes will be held in the archive on-line database. These attributes are used for checking whether a certain file system object has changed and can be listed for selection in the archive retrieval phase.

4.2. Archive Write Transaction

Archives are written as an entire set. This operation is a single transaction. The set of file system objects belonging to an archive is specified as a list of triples consisting of a-file names, an operation code and a list of attribute names. There are three operation codes:

- *copy*
  transfer file system object to media
- *copy if changed*
  transfer file system object only if at least one of the attributes has changed
- *present*
  do not copy file system object, but mark it as present

An unconditional archiving as achieved with the *copy* operation code, meaning that the file system object must be archived. By using the *copy if changed* operation code, it is possible to archive only those file system objects that have changed in some way. The notion of *changed* is derived from the listed attributes. If the value in the on-line database of the archive does not match the value of the attribute found in the actual file system, a change has occurred. The *present* operation code just marks a file system object as present. This is necessary in order to be able to find out about deleted file system objects. At the end of an archive write transaction all names of a-files that have not been listed will be considered deleted.
Whenever an archive is written, all data stored during this transaction will be associated with a *location*. This location corresponds to a v-file. The reason for this association is motivated by the wish to be able to support redundant data storage. (Nothing is worse than to find out that the needed a-file resides on an unreadable media.) A file system object can be stored in different locations which, in conjunction with the v-file-group mechanism, allows media failure resistance. Furthermore it is possible to delete locations. Deleting locations has the effect of removing redundant locations and old versions of a-files.

4.3. Version Management

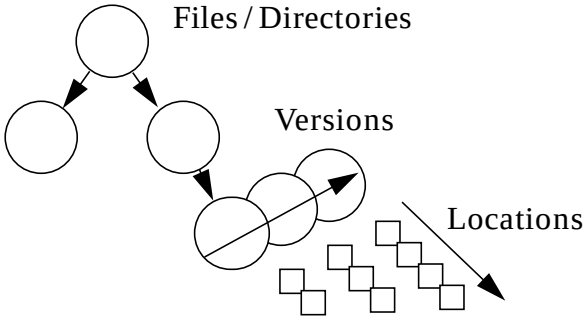
An a-file version consists of two numbers. The first number is the *incarnation*. It is incremented each time the file system object type attribute changes, or when the file system object is deleted (missing from the list of a-file names to be written). The sec-

ond number of the version is the *current_-version* this number is incremented each time, when one of the version-selective a-file attributes changes.
A fully qualified a-file thus consists of a path-name, an incarnation number and a current version.
This version numbering allows for easy history handling of file system objects.

Example:
The following example shows how a single file system object history could look like. Time actually denotes the time the archive was written.

time	**event**	**type**	**version**
1	new	file	1.1
2	changed	file	1.2
3	deleted		2.0
4	new	dir	2.1
5		dir	2.1
6	changed	dir	2.2
8	changed	file	3.1

The on-line-database for the archive allows easy retrieval of attribute information about the file system objects for all versions. The possibility of recognizing deleted file system objects is one major cornerstone for implementing incremental backup policies. Another advantage is that the distinction between total and incremental backup diminishes. The archive presents always the combined view of all write operations. The user has to select whether he would like to see the top version of the archive or also all deleted versions.



4.4. Retrieval

Archive retrieval is also a transaction. The a-files to be retrieved can be located via a lookup mechanism, that allows to scan through the archive contents. It is possible to define an access policy for each archive. For Unix archive one would most likely use the Unix access policy. This policy controls access to the a-file tree by looking at the appropriate file permission attributes. One main issue about automatic file system object restoration is conflict handling. One of the most common cases is the restoration of a subset of the archive. At this point a decision has to be made, where this file tree is to be restored, and what happens when the restored file tree would overwrite existing file system objects. Several restoration conflict handling strategies come into mind:

- common prefix path
- delete common prefix
- rename on conflict

These strategies and their combinations constitute a powerful restoration mechanism that can be used by any user who has access to the appropriate archives. The permission checking policy provides the base for unsupervised retrieval.

4.5. Terminology

The following abstractions are provided by the *Archive Manager*:

- Archive
  Set of a-files
- a-file
  Member of an archive having different versions, each stored in one or more locations
- Location
  logical place, where a version of a file can be stored (usually a *Volume Manager* file)

4.6. Transactions

The *Archive Manager* provides several basic operations on archives:

- Archive creation and deletion
- Archive attribute management
- Archive writing, restoring and deleting
- Transaction handling (list, abort)
- Archive data retrieval (listing)
- Configuration

4.7. File Packing and Unpacking

Whenever an archive is written a file packer is used. This utility cannot be a normal unix tool, since it has to support the transaction semantics of the *Archive Manager*. For this reason it has to generate a detailed packing protocol that allows the *Archive Manager* to update its local database with the new file attributes. The file packer will be capable of encapsulating all standard file types. Today's "archive" programs lack either proper handling of system dependent files such as special files (tar), or are very implementation specific (dump).

The file format of the packer is common between all implementations. It is used to encapsulate file system objects. Each encapsulation has a type attribute allowing to locate the appropriate unpack mechanism. This enables the unpacker to skip unsupported file system objects. Provisions for handling corrupted packer files are also present. The unpacker will be able to unpack all file system objects known to its implementation. It is possible to extend this set by providing additional unpack programs that unpack specific file system objects not previously known to the base implementation of the unpacker. These additional programs handle also file system object conversion. The base implementation of the packer/unpacker can be extended for handling of vendor specific extensions (conditional symbolic links,

context dependent files, contiguous files). These features could also be separate programs, but the detection mechanism for non standard file system objects must be compiled into the file packer.

4.8. Interfaces

The *Archive Manager* has, as well as the library interface, the *ash* application, which allows the viewing of the archive contents and the specification of restoration operations. Due to the inherent authentication mechanism it is possible to let each user manage his own archive. The only burden on the operator would be an occasional request for inserting some media. *Ash* and *vlink* will also be available as X-clients.

5. Backup Manager

With the mechanisms provided by the *Volume Manager* and the *Archive Manager* it is relatively easy to build a *Backup Manager*. The *Backup Manager* will generate the necessary file lists for the *Archive Manager* according to the backup schedules. Overlapping incremental backups are written for data integrity. The v-file-group mechanism of the *Volume Manager* allows the writing of overlapping incremental backups onto different media. Another important issue of the *Backup Manager* is the mapping of the network-wide file tree as created by NFS mounting to the backup archives. A “backup restore ~user” command will invoke the *ash* with the appropriate default settings, so that the user can look through the correct archive, where his/her home directory is archived. Then it is possible to retrieve the needed files by specifying an archive restoration transaction.

The mapping between network-wide file trees and backup archives is usually very complex and will probably demand a fair amount of configuration data.

6. State of Development

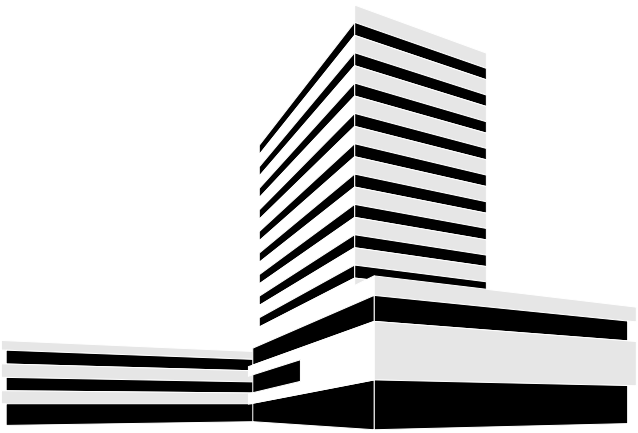
As of July 1991 a prototype implementation of the *Volume Manager* and the *Archive Manager* was underway. The *Backup Manager* was still in the design phase.

LISA V - Fall 1991

BackStage

Frank Kardel

kardel@informatik.uni-erlangen.de



IMMD IV • Martensstraße 1 • D-W8520 Erlangen
Germany

1. Design Goals of *BackStage*

- **Transaction based client/server model**
- **Extensibility**
- **Network protocol/implementation independence**
- **Flexible authentication/authorization**
- **User initiated backup/archive**
- **Support for varying file system semantics**
- **Portability**

2. *BackStage* System Structure

Three subsystems:

- *Volume Manager*
  access to mass storage media
  media maintenance

- *Archive Manager*
  management of *a-file* **trees (***ARCHIVES***)**

- *Backup Manager*
  incremental/total backup management

3. *BackStage* Base Mechanisms

3.1. Transaction Identification (*TID*)

- Service identifier (service . network location)
- ⇒ Transaction (“robust process” - crash resistent)
- ⇒ Data structure (configuration)

3.2. Authentication / Authorization

- **User/Service entity**
- **Multiple protocols**
- **Access control lists**

4. *Volume Manager*

4.1. Semantics / Abstraction

- **Provision of a set of** *VOLUMES*
- *VOLUME*:
  Set of *v-files* **each of (almost) arbitrary length**
- *VOLUME*-**Operations:**
  create, delete volume
  create, delete, read, write *v-file*
  list *VOLUMES*, *VOLUME* **contents**
- **Scheduling of Operations**
  Run-time loadable scheduling strategy

4.2. Components

- *volumed - Volume Manager*
  scheduling, volume database

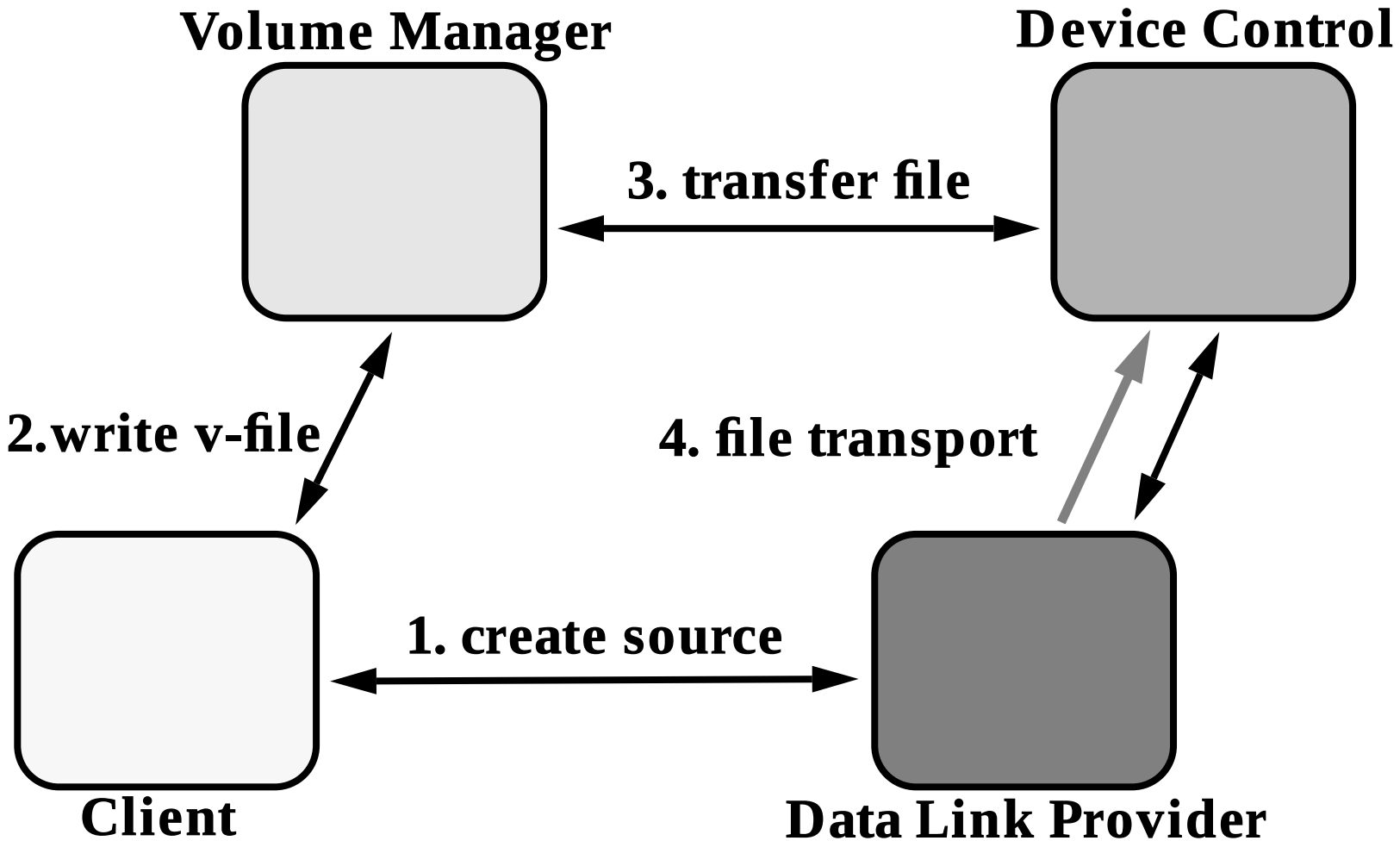
- *devcon - Device Manager*
  actual device access

- *dlp - Data Link Provider*
  specialized data transfer agent
  Unix - *BackStage* **interface**

- *vlink - Volume Manager Interface*
  operator interface - transaction status display and control

- *vsh* **-** *Volume Manager Shell*
  Unix command interface to *Volume Manager*

4.3. Interaction Structure



5. *Archive Manager*

5.1. Semantics / Abstraction

- **Provision of a set of** *ARCHIVES*

- *ARCHIVE:*
  Set of *a-file* **trees**
  an *a-file* **represents a file system object (like** *directory, file, ISAM-file, symb. link, device, ...)*

- *ARCHIVE*-**Operations:**
  create, delete *ARCHIVE*
  write, retrieve to/from *ARCHIVE*
  list *ARCHIVES, ARCHIVE* **contents**

5.2. *A-file* trees (*ARCHIVES*)

An *a-file* consists of:

- **Pathname**

- **Incarnation.Location**
  incarnation **- file incarnation identifier**
  location **- logical storage identifier**

- **Incarnation identifying attributes**
  change in one attribute leads to a new *incarnation*
  (file size, owner, mode, type)

- **Informational attributes**
  additional information
  (comment, file format, ...)

5.3. *A-file* version sequence

An example for “~/xntpd”

#	Comment	Size	Type	Incarnation.Location
1	initial version	512	dir	1.1
2	removal	-	-	2.0
3	new again	512	dir	2.1
4	it’s a file now	8567	file	3.1
5	no change - but we write it again	8567	file	3.2
6	now it was moved	17	symlink	4.1
7	no change	17	symlink	4.2

5.4. *ARCHIVE* write operation

- *ARCHIVES* **are written by specifying a list of operations and file names**

- **operations are:**

  - *copy*
    copy file unconditionally into archive

  - *copy if changed*
    copy on change of incarnation attributes

  - *present*
    no copy, but file is marked as present

5.5. *ARCHIVE* retrieve operation

- *ARCHIVES* **are retrieved by specifying a list of operations and file names**

- **operations define collision handling/retrieval:**

  - *common prefix*
    on retrieval file name is appended to a prefix path

  - *delete common prefix*
    a common prefix of the file name is deleted

  - *rename*
    if the retrieved file would overwrite an existing, the retrieved file is re-named

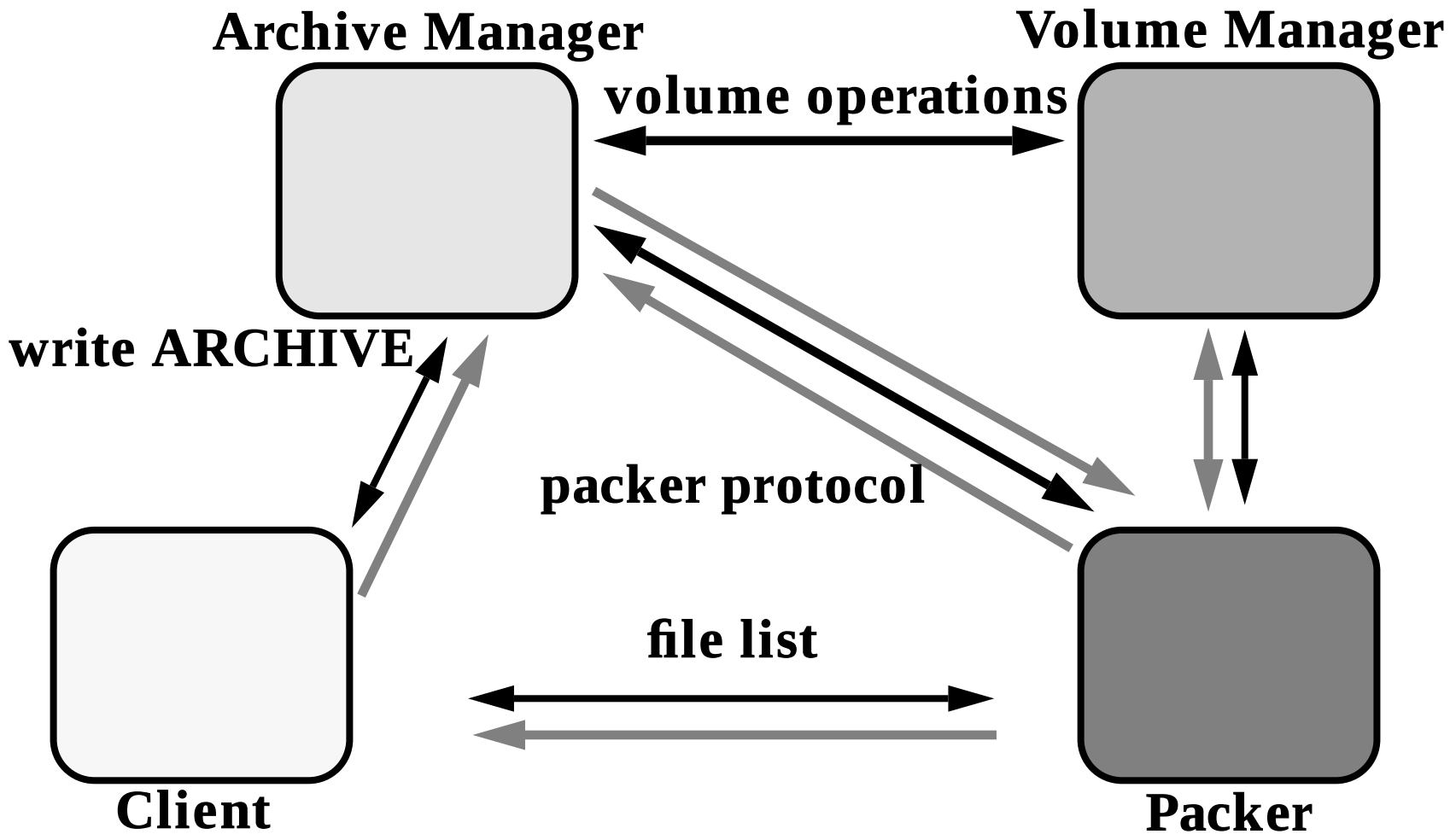
5.6. Components

- *ard - Archive Manager*
  operations, archive database, retrieval control

- *packer - File Packer*
  packing and unpacking of file system objects (encapsulation)

- *ash - Archive Manager Interface*
  user interface
  graphical version available

5.7. Interaction Structure



6. *Backup Manager*

6.1. Semantics / Abstraction

- **Execution of backup sequences for the entire network**
- **Overlapping incremental backups onto *ARCHIVES* throughout the entire network**
- **File system view transformation**

6.2. Components

- *backupd - Backup Manager*
  backup schedules

- *bsh - Backup Manager Interface*
  schedule/status access, restoration (via *ash*)

6.3. Interaction Structure