Amanda::Application::Abstract

NAME

Amanda::Application::Abstract - A base class for Amanda applications

SYNOPSIS

```
    package amMyAppl;
    use base qw(Amanda::Application::Abstract)

    sub supports_message_line { return 1; }
    ...

    sub inner_backup {
      my ($self, $fdout) = @_;
      ...
    }
    ...

    package main;
    amMyAppl::->run();
```

DESCRIPTION

`Amanda::Application::Abstract` handles much of the common housekeeping needed to implement Amanda's Application API, so that practical applications can be implemented in few lines of code by overriding only a few necessary methods.

`Amanda::Application::Abstract` is itself a subclass of `Amanda::Application`, but the latter cannot be instantiated until after the command line has been parsed (at least the `config` option needs to be passed to its constructor). Therefore, `Amanda::Application::Abstract` supplies *class* methods for the preliminary work of declaring the supported options, parsing the command line, and finally calling `new` to get an instance of the class. Then the actual operations of the application are carried out by instance methods, as you would expect.

In perl, class methods are inheritable and overridable just as instance methods are, and the syntax `$class->SUPER::method(...)` works, analogously to its familiar use in an instance method. So, the pre-instantiation behavior of an application (declaring command options, etc.) can be tailored by simply overriding the necessary class methods, just as its later operations can be supplied by overriding instance methods.

FUNDAMENTAL CLASS METHODS

run

```
    appclass::->run()
```

Run the application: grab the subcommand from `ARGV[0]`, be sure it is a known one, set up for option parsing, parse the options, construct an instance passing all the supplied options (which will include the config name, needed by the `Amanda::Application` constructor), and finally `do()` the subcommand, where `do` is inherited from `Amanda::Script_App` by way of `Amanda::Application`.

new

```
    $class->new(\%opthash)
```

A typical application need not call this; it is called by `run` after the command line options have been declared and the command line has been parsed. The hash reference contains option values as stored by `GetOptions`. A certain convention is assumed: if there is an option/property named *o*, its value will be stored in *opthash* with key exactly *o*, in exactly the way `GetOptions` normally would, *unless* the option-declaring method had planted a code reference there in order to more strictly validate the option. In that case, of course *$opthash{o}* is the code reference, and the final value will be stored with key `opt_`*o* instead. Because of that convention, entries in *opthash* with an `opt_` prefix will otherwise be ignored;

avoid declaring options/properties whose actual names start with `opt_`.

CLASS METHODS IMPLEMENTING VARIOUS QUOTING RULES

dquote

```
    $class->dquote($s)
```

The purpose of this method is to quote a string in exactly the way that `Text::ParseWords::shellquote` unquotes things, because `shellquote` is what `GetOptionsFromString` claims to use. However, the Text::ParseWords POD does not spell out the rules to use. O'Reilly's Perl Cookbook says (1.15) that "Quotation marks and backslashes are the only characters that have meaning backslashed. Any other use of a backslash will be left in the output string." ... which seems to be true, looking at the code, and is true of ' when ' is the quote mark, and true of " when " is the quote mark. That makes it actually *not* like any common Unix shell, and also not like `Amanda::Util::quote_string`, so it needs its own dedicated implementation here to be sure to get it right.

gtar_escape_quote

```
    $class->gtar_escape_quote($s)
```

This method quotes a string using the same rules documented for GNU `tar` in `--quoting-style=escape` mode, as used on the index stream. (Note: the GNU `tar` documentation, as of April 2016 anyway, says that space characters also get escaped, though GNU `tar` has never actually done that, and neither does this method.)

gtar_escape_unquote

```
    $class->gtar_escape_unquote($s)
```

The inverse of `gtar_escape_quote`.

CLASS METHODS FOR VALIDATING OPTION/PROPERTY VALUES

```
    ..._property_setter(\%opthash)
```

For a common type ... return a sub that can be placed in `%opthash` to validate and store a value of that type. The anonymous sub accepts the parameters described at "User-defined subroutines to handle options" for `Getopt::Long`, and will store the validated, converted option value in *%opthash* at a key derived by prefixing `opt_` to the option name. For example, a `declare_common_options` method that defines a boolean property *foo* may contain the snippet:

```
    $refopthash->{'foo'} = $class->boolean_property_setter($refopthash);
```

At present, properties with scalar or hash values can be supported (the setter sub can tell by the number of parameters passed to it by `Getopt::Long`), but not multiple-valued ones accumulating into an array (which can't be distinguished from the scalar case just by looking at what comes from `Getopt::Long`).

boolean_property_setter

Allows the same boolean literals described in `amanda.conf(5)`: `1`, `y`, `yes`, `t`, `true`, `on` or `0`, `n`, `no`, `f`, `false`, `off`, case-insensitively.

CLASS METHODS FOR DECLARING ALLOWED OPTIONS

```
    declare_..._options(\%opthash, \@optspecs)
```

Each of these is a class method that is passed two references: a hash reference *\%opthash* and a list reference *\@optspecs*. It should push onto *\@optspecs* the declarations of whatever options are valid for the corresponding subcommand, using the syntax described in Getopt::Long. It does not need to touch *\%opthash*, but may store a code

reference at the name of any option, to apply stricter validation when the option is parsed. In that case, the code reference should ultimately store the validated value into *\%opthash* at a key derived by prefixing `opt_` to the option name.

For each valid *subcommand*, there must be a `declare_`*subcommand*`_options` class method, which will be called by `run` just before trying to parse the command line.

declare_common_options

Declare the options common to every subcommand. This is called by every other `declare_`*foo*`_options` method, and, if not overridden itself, simply declares the `config`, `host`, `disk`, `device` options that (almost?) every subcommand ought to support.

A subclass that has properties of its own should probably declare them here rather than in more-specific `declare_`*foo*`_options` methods, as whatever properties are set in the Amanda configuration could be passed along for any subcommand; Amanda won't know which subcommands need them and which don't.

declare_support_options

Declare options for the `support` subcommand. Only the `common` ones.

declare_backup_options

Declare options for the `backup` subcommand. Unless overridden, this simply declares the `common` ones plus `message`, `index`, `level`, `record`, and `state-stream`.

declare_restore_options

Declare options for the `restore` subcommand. Unless overridden, this simply declares the `common` ones plus `message`, `index`, `level`, `dar`, and `recover-dump-state-file`.

declare_index_options

Declare options for the `index` subcommand. Unless overridden, this simply declares the `common` ones plus `message`, `index`, `level`.

declare_estimate_options

Declare options for the `estimate` subcommand. Unless overridden, this simply declares the `common` ones plus `message` and `level`. If `$class->supports('multi_estimate')` then `level` will be declared to allow multiple uses.

declare_selfcheck_options

Declare options for the `selfcheck` subcommand. Unless overridden, this simply declares the `common` ones plus `message`, `level`, and `record`.

declare_validate_options

Declare options for the `validate` subcommand. Unless overridden, this declares no options at all.

CLASS METHODS SUPPORTING `support` SUBCOMMAND

These class methods establish the default capabilities advertised by the `support` subcommand. A particular application can supply *class* methods to override them. Most are boolean and an absent one is treated as returning `false`. Therefore, the only ones that need to be supplied here are the non-booleans, and the booleans with non-`false` defaults.

recover_mode

Should return either `undef` or `'SMB'`. Default if not overridden is `undef`.

data_path

Should return a list of `'AMANDA'`, `'DIRECTTCP'`, or both. Default if not overridden is `'AMANDA'`.

recover_path

Should return `'CWD'` or `'REMOTE'`. Default if not overridden is `CWD`.

supports

```
    $class->supports(name)
```

Returns `false` if there is no `supports_`*name*`()` class method, otherwise calls it and returns its result.

INSTANCE METHODS SUPPORTING `support` SUBCOMMAND

max_level

Returns the maximum `level` supported, Default if not overridden is zero, indicating no support for incremental backup. An instance method because the value could depend on state only known after instantiation.

INSTANCE METHODS SUPPORTING LOCAL PERSISTENT STATE

These methods establish a conventional location for a local state file, and a default format for its contents. To avoid relying on non-core modules (even JSON isn't a sure thing yet), the default format relies on writing name/value pairs to the file in a form `Getopt::Long` can recover.

By default, a state is a hash with integer keys representing levels, plus one string key, `maxlevel`. The value for `maxlevel` is an integer, and for each integer key *level*, the value is a hashref that contains (redundantly) `'level' =>` *level* plus whatever other name/value pairs the application wants.

local_state_path

```
    ($dirpart, $filepart) = $self->local_state_path()
```

Supplies a reasonable location for storing local state for the subject DLE. If not overridden, returns a *$dirpart* based on `$Amanda::Paths::localstatedir` followed by `Amanda::Util::hexencoded` components based on `config`, `host`, `disk`, and `device` if present, and a *$filepart* based on the application name.

build_state_path

```
    ($dirpart, $filepart) = $self->build_state_path($basedir)
```

Does the actual building of a local state path (as described above for `local_state_path`) given a base directory, so that a subclass can easily override `local_state_path` to use a different base directory, but otherwise build the rest of the path the same way.

read_local_state

```
    $hashref = $self->read_local_state(\@optspecs)
```

Read a local state, if present, returning it in *$hashref*. In the returned state, the value for `maxlevel` represents the maximum level of incremental *backup* that could be requested, namely, one plus the maximum previous level represented in the state. If no stored state is found, a state hash is returned with `maxlevel` of zero and no other data.

Otherwise, the file is parsed for one or more levels of saved state, by repeatedly applying `Getopt::Long` with *\@optspecs* to declare the application's allowed options.

write_local_state

```
    $self->write_local_state(\%levhash)
```

Save the local state represented by *\%levhash*, creating the file and intermediate directories if necessary. `Amanda::Util::safe_overwrite_file` is used to avoid leaving partially-overwritten state.

update_local_state

```
    $self->update_local_state(\%state, $level, \%opthash)
```

Given a *\%state* (which contains `'maxlevel' =>` (*n*+1) as well as *k* `=>` *(opthash for level k)* for *k* in 0..n), and a new *$level* and corresponding *\%opthash* for that level, change `maxlevel` to 1 + *$level*, drop all entries for levels above *$level*, and install the new *\%opthash* at *$level*.

INSTANCE METHODS IMPLEMENTING SUBCOMMANDS

command_support

As long as the application class overrides the methods indicating its support for these capabilities, this default implementation will take care of producing output in the proper format.

METHODS FOR THE backup SUBCOMMAND

check_message_index_options

Check sanity of the (standard) options parsed from the command line for `--message` and `--index`.

check_backup_options

Check sanity of the (standard) options parsed from the command line for a `backup` operation. Override to check any additional properties for the application.

check_level_option

Check sanity of `--level` option(s) parsed from the command line. In order to work in cases that do or don't support `multi_estimate`, this will check either a single value that isn't an array, or every element of a value that's an array reference.

emit_index_entry

```
    $self->emit_index_entry($name)
```

Write *$name* to the index stream. The caller is responsible to make sure that *$name* begins with a `/`, is relative to the *--device*, and ends with a `/` if it is a directory. Otherwise it should be the exact name that the OS would be given to open the file. This method will handle quoting any special characters in the name as needed within the index file. It will use the same quoting rules as GNU `tar` in `--quoting-style=escape` mode.

command_backup

Performs common housekeeping tasks, calling `inner_backup` to do the application's real work. First calls `check_backup_options` to verify the invocation, creates the `index_out` file handle, and calls `inner_backup` passing the fd to use for output.

On return from `inner_backup`, closes the output fd and the index handle, and writes the `sendbackup: size` line based on the size in bytes returned by `inner_backup` (unless the returned size is negative, in which case no `sendbackup` line is written).

inner_backup

```
    $size = $self->inner_backup($outfd)
```

In many cases, the application should only need to override this method to perform a backup. The backup stream should be written to *$outfd*, and the number of bytes written should be returned, as a `Math::BigInt`.

If not overridden, this default implementation writes nothing and returns zero.

shovel

```
    $size = $self->shovel($fdfrom, $fdto)
```

Shovel all the available bytes from *$fdfrom* to *$fdto*, returning the number of bytes shoveled as a `Math::BigInt`.

METHODS FOR THE `restore` SUBCOMMAND

check_restore_options

Check sanity of the (standard) options parsed from the command line for a `restore` operation. Override to check any additional properties for the application.

emit_dar_request

```
    $self->emit_dar_request($offset, $size)
```

A `restore` subcommand that supports Direct Access Recovery can be passed `--dar YES` and `--recover-dump-state-file` *filename*, read the *filename* to recover the backup stream position information saved at backup time, then look up the names requested for restoration to determine what ranges of the backup stream will actually be needed to complete the recovery. It calls `emit_dar_request` once for each contiguous range needed; *$offset* and *$size* are both in units of bytes.

command_restore

Performs common housekeeping tasks, calling `inner_restore` to do the application's real work. First calls `check_restore_options` to verify the invocation. If DAR is supported and requested, opens the recover-state-file for reading and the DAR stream for writing. Changes directory to `Amanda::Util::get_original_cwd()`, or to the value of a `--directory` property if the application declared such an option and it was seen on the command line.

Calls `inner_restore` passing the input stream fileno, the recover-state-file handle (or `undef` if DAR was not requested), and the command-line arguments (indicating objects to be restored); that is, `inner_restore` has a variable-length argument list after the first two.

On return from `inner_restore`, closes the recover-state-file and DAR handles if used.

inner_restore

```
    $self->inner_restore($infd, $dsf, $filetorestore...)
```

Should be overridden to do the actual restoration. Reads stream from *$infd*, restores objects represented by the *$filetorestore* arguments. If the application supports DAR, should check *$dsf*: if it is defined, it is a readable file handle; read the state from it and then call `emit_dar_request` to request the needed backup stream regions to recover the wanted objects.

If not overridden, this default implementation reads, and does, nothing.

METHODS FOR THE `index` SUBCOMMAND

check_index_options

Check sanity of the (standard) options parsed from the command line for an `index` operation. Override to check any

additional properties for the application.

command_index

Performs common housekeeping tasks, calling `inner_index` to do the application's real work. First calls `check_index_options` to verify the invocation.

Calls `inner_index`, which is expected to read from `STDIN`.

On return from `inner_index`, closes the index-out stream.

inner_index

```
    $self->inner_index()
```

Should be overridden to read from `STDIN` and call `emit_index_entry` for each user object represented.

If not overridden, this default implementation reads everything from `STDIN` and emits a single entry for `/`.

METHODS FOR THE `estimate` SUBCOMMAND

check_estimate_options

Check sanity of the (standard) options parsed from the command line for an `estimate` operation. Override to check any additional properties for the application.

command_estimate

Performs common housekeeping tasks, calling `inner_estimate` to do the application's real work. First calls `check_estimate_options` to verify the invocation.

Calls `inner_estimate` once (for each level, in case of `multi_estimate`), which should return a `Math::BigInt` size in bytes. Writes each size to the output stream in the required format, assuming a block size of 1K.

inner_estimate

Takes one parameter (the level) and returns a `Math::BigInt` estimating the backup size in bytes.

If not overridden, this default implementation does nothing but return zero. Hey, it's an estimate.

METHODS FOR THE `selfcheck` SUBCOMMAND

check_selfcheck_options

Check sanity of the (standard) options parsed from the command line for a `selfcheck` operation. Override to check any additional properties for the application. If not overridden, this checks the same things as `check_backup_options`.

command_selfcheck

If not overridden, this simply checks the options and writes a GOOD message.

METHODS FOR THE `validate` SUBCOMMAND

command_validate

If not overridden, this simply reads the complete data stream and does nothing with it. (Which is just what would happen if this sub were absent and the invocation fell back to `default_validate`, but providing this sub allows subclasses to refer to it with SUPER and not have to think about whether `command_validate` or `default_validate` is the right thing to call.)

ABOUT THIS PAGE

This page was automatically generated Wed Aug 30 14:58:07 2017 from the Amanda source tree, and documents the most recent development version of Amanda. For documentation specific to the version of Amanda on your system, use the 'perldoc' command.

Name

amgrowingfile — Amanda Application for backup of single append-only file

DESCRIPTION

Amgrowingfile is an Amanda Application API script. It should not be run by users directly. It can backup and restore a single file that is known to change only by appending.

Such a file may represent audit or logging data, a stream of data being acquired from instrumentation, etc. Backup levels are supported, where a backup level greater than zero simply consists of all data written to the file beyond its length in the prior-level backup.

Note

For correctness, this strategy requires that the file be known to grow only at the end. On some operating systems (see the **chattr** command in Linux, for example), an *append-only* flag can be set on the file to enforce this. If the file is ever rewritten from the start (such as in log rotation), the next **amgrowingfile** backup must be forced to level 0.

The *diskdevice* in the disklist (DLE) is the filename **amgrowingfile** will read. When restoring, the FILENAME property (which can be set with the *setproperty* command in amrecover(8)) will be used, or, if it is not set, `amgrowingfile-restored` in the current directory. In other words, **amgrowingfile** will not default to restoring data directly to the original filename, but the FILENAME property can be set to do so, if desired.

When restoring, the file named with the FILENAME property (or `amgrowingfile-restored`) is truncated and rewritten in place if it exists, or created with mode 0600 if it does not.

PROPERTIES

This section lists the properties that control amgrowingfile's functionality. See amanda-applications(7) for information on application properties and how they are configured.

FILENAME
: Used only for restore command, can be a device name or file, the data will be restored to it.

EXAMPLE

```
define application-tool app_amgrowingfile {
  plugin "amgrowingfile"
}
```

A dumptype using this application might look like:

```
define dumptype amgrowingfile {
  global
  program "APPLICATION"
  application "app_amgrowingfile"
}
```

Note that the *program* parameter must be set to "*APPLICATION*" to use the *application* parameter.

SEE ALSO

amanda(8), amanda.conf(5), amanda-applications(7), amrecover(8)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amgrowingzip — Amanda Application for backup of append-only ZIP archive

DESCRIPTION

Amgrowingzip is an Amanda Application API script. It should not be run by users directly. It can backup and restore a single file that is a ZIP archive known to change only by appending members.

Such a file may be used to collect many small files that are created over time, such as data acquired from instruments, write-ahead logs of some databases, etc. Where many small files in a directory could use disk blocks inefficiently, appending them, as they are generated, to a growing ZIP archive can achieve efficient storage along with the ability to extract individual files from the ZIP as needed.

Amgrowingzip itself only deals with the ZIP archive as a unit. When restoring, the entire archive will be restored. Any extraction of individual members from the archive can then be done with ordinary ZIP tools. The growing ZIP file can be large, and amgrowingzip supports backup levels.

Note

For correctness, this application requires that the ZIP archive be known to grow only by new members being appended to it. This is analogous to an append-only operation at the file level, but not the same, so it cannot be enforced by typical *append-only* flags offered by the operating system. The structure of a ZIP archive consists of members of the archive (typically compressed), followed by a directory structure at the end. Appending a member involves seeking to the end of the file, backspacing to the start of the directory structure, and overwriting with the new member(s) followed by the new directory structure. For amgrowingzip, a backup level greater than zero is simply everything that must be written over the file starting at the directory structure offset in the prior-level backup.

Any time the ZIP archive is rebuilt rather than simply appended, the next **amgrowingzip** backup must be forced to level 0. Not all ZIP-related tools truly append a member as described here; some perform any modification of the archive by rebuilding it from scratch. To be usable with amgrowingzip, whatever process adds members to the archive must be known to truly append them. The zipfile module in Python is suitable, for example.

Note

If it is possible for the process that appends members to be active while **amgrowingzip** is taking a backup, the FLOCK property should be set, and the appending process should also be coded to hold an advisory exclusive lock on the archive file during each modification. Otherwise, a failure or unusable backup could result from trying to locate the archive's directory while a write is in progress.

The *diskdevice* in the disklist (DLE) names the ZIP archive **amgrowingzip** will read. When restoring, the FILENAME property (which can be set with the *setproperty* command in amrecover(8)) will be used, or, if it is not set, `amgrowingzip-restored` in the current directory. In other words, **amgrowingzip** will not default to restoring data directly to the original filename, but the FILENAME property can be set to do so, if desired.

When restoring, the file named with the FILENAME property (or `amgrowingzip-restored`) is truncated and rewritten in place if it exists, or created with mode 0600 if it does not.

No locking is used while restoring, regardless of the FLOCK property. There is no sense in allowing any writes to the file being restored before the last increment needed has been applied to it. The best approach is to restore under a temporary name and, only after restoration, move the file to the expected place.

PROPERTIES

This section lists the properties that control amgrowingfile's functionality. See amanda-applications(7) for information on application properties and how they are configured. For the recognized *true*/*false* values for a boolean property, see amanda.conf(5).

FILENAME
: Used only for restore command, can be a device name or file, the data will be restored to it.

FLOCK

Set to a *true* value to require an advisory shared lock before backing up the file, in case the process that appends members to it could be simultaneously active. The appending process must also be coded to take an advisory exclusive lock while writing. If this property is given a *false* value, no locking is used.

EXAMPLE

```
define application-tool app_amgrowingzip {
  plugin "amgrowingzip"
}
```

A dumptype using this application might look like:

```
define dumptype amgrowingzip {
  global
  program "APPLICATION"
  application "app_amgrowingzip"
}
```

Note that the *program* parameter must be set to "*APPLICATION*" to use the *application* parameter.

SEE ALSO

amanda(8), amanda.conf(5), amanda-applications(7), amrecover(8)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amooraw — Amanda Application open and read data

DESCRIPTION

Amooraw is an Amanda Application API script. It should not be run by users directly. It directly reads and writes data in a single directory entry.

Amooraw can backup only one directory entry, it can be a single file, a raw device, anything that amanda can open and read.

It is a reimplementation of amraw(8) illustrating the simpler construction of an Amanda application in object-oriented style (the oo in amooraw) using `Amanda::Application::Abstract`.

Note

Where the original **amraw**, when restoring, allows the destination to be specified with the DIRECTORY property, **amooraw** uses a different property, FILENAME, for that purpose. Also, if that property is not specified, restoration will create a file `amooraw-restored` instead of blithely overwriting the original *diskdevice* named in the DLE. If that is what's wanted, simply set the FILENAME property to match the *diskdevice*.

The *diskdevice* in the disklist (DLE) must be the filename **amooraw** is to open and read.

Restore is done in place: an open is done and the data is written to it. A file owned by root and permission 0600 is created if the directory entry doesn't exist before the restore.

Only full backup is allowed.

PROPERTIES

This section lists the properties that control amooraw's functionality. See amanda-applications(7) for information on application properties and how they are configured.

FILENAME
: Used only for restore command, can be a device name or file, the data will be restored to it.

EXAMPLE

```
define application-tool app_amooraw {
  plugin "amooraw"
}
```

A dumptype using this application might look like:

```
define dumptype amooraw {
  global
  program "APPLICATION"
  application "app_amooraw"
}
```

Note that the *program* parameter must be set to "*APPLICATION*" to use the *application* parameter.

SEE ALSO

amanda(8), amanda.conf(5), amanda-applications(7), amrecover(8)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amopaquetree — Amanda Application for backup of a directory tree opaquely

DESCRIPTION

Amopaquetree is an Amanda Application API script. It should not be run by users directly. It can backup and restore a directory tree opaquely (that is, as a whole, with no option to restore just some entries), with efficient increments when large files have only small changes.

Most Amanda applications that backup a directory tree allow selective restoration of only some files or subtrees. Also, the typical applications that support backup levels, such as **amgtar**, operate at the file level: changes, however small, to any file cause the entire file to be included in the incremental backup. These properties are suitable for many backup needs.

Some applications, such as databases and products that incorporate them, may keep files in a directory structure with obscure naming or numbering, where an administrator would rarely be expected to identify or restore selected files, but simply to restore the tree at a consistent moment. For such applications, the individual files can also be large, and modified only in small regions, so that backing up entire changed files would be wasteful.

Amopaquetree is intended for those cases. A directory-list entry (DLE) backed up with amopaquetree can only be restored in full, without selective access to subcomponents. Incremental backups will identify only changed regions within files, rather than backing up entire files when changed, using more storage and computation on the client, but saving network bandwidth, holding disk, and tape space.

The extra storage required on the client is similar to the size of the tree to be backed up. Other Amanda applications supporting incremental backup store only a small client-local state, such as a date or a file list, for each previous backup level. For **amopaquetree**, the client-local state will include one complete copy of the tree for the lowest level backup, plus a tree for each higher level containing only changed files, with links to the lower level for unchanged files. Therefore, *on the client*, storage is required for entire files that have been changed, but this storage is accessed at local speeds. From those copies, the changed regions are efficiently found, and only they are streamed across the network to store on the server.

Note

Most databases and similar applications will have specific steps that must be followed to guarantee the files are consistent before backing them up. Those steps may take the form of a command to create a consistent copy (`svnadmin hotcopy` for subversion, `db2bak` for 389, etc.), or to enter and exit a mode during which a copy may safely be made (`pg_start_backup`/`pg_stop_backup` for PostgreSQL, etc.).

For cases of the first type, that create a consistent copy, an Amanda script to run the necessary command ahead of **amopaquetree** may be all that is needed. The client should have temporary space for that consistent copy, as well as the local state space needed by **amopaquetree** itself. For cases of the second type, entering and exiting a safe-copying mode (which may require additional steps such as copying a write-ahead log), it may be better to create a specialized Amanda application by subclassing `Amanda::Application::AmOpaqueTree`.

IMPLEMENTATION

amopaquetree uses (and, therefore, requires) the widely-available **rsync** tool, both in its `--link-dest` mode to maintain client-local state limited in size to one full copy plus changed files in higher levels, and in its `--only-write-batch` mode to generate the smaller data streams, reflecting changed regions only, to be sent to the server.

PROPERTIES

This section lists the properties that control amopaquetree's functionality. See amanda-applications(7) for information on application properties and how they are configured.

LOCALSTATEDIR
: Amanda has a default location for client local state, but because **amopaquetree** stores a larger-than-typical local state, this property can be used to place the local state storage on a different filesystem. Or, this property can be left at default, and RSYNCSTATESDIR (which defaults to a subdirectory of LOCALSTATEDIR) can be set, to use a different filesystem only for that.

RSYNCEXECUTABLE
: Full path to the **rsync** executable on the client system. If elevated permission will be needed for **rsync** to read the tree being backed up, this can point instead to a wrapper that gains the needed privilege and then executes **rsync**.

RSYNCSTATESDIR
: The directory (defaulting to a subdirectory of LOCALSTATEDIR) for the local copied/linked images of the tree backed up. Because this accounts for the greatest share of local state space, it can be separately redirected to another filesystem using this property.

RSYNCTEMPBATCHDIR
: The directory where **rsync** batch files will be temporarily created. Temporary space must be available for the expected size of a batch; for a level-0 backup, that will be comparable to the size of a compressed archive of the tree being backed up. The default is where Perl's `File::Temp` will put a temporary file.

EXAMPLE

```
define application-tool app_amopaquetree {
  plugin "amopaquetree"
}
```

A dumptype using this application might look like:

```
define dumptype amopaquetree {
  global
  program "APPLICATION"
  application "app_amopaquetree"
}
```

Note that the *program* parameter must be set to "*APPLICATION*" to use the *application* parameter.

SEE ALSO

amanda(8), amanda.conf(5), amanda-applications(7), amrecover(8)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amzfs-holdsend — Amanda Application for backup of ZFS externally-created snapshots

DESCRIPTION

Amzfs-holdsend is an Amanda Application API script. It should not be run by users directly. It can backup and restore ZFS filesystems opaquely (that is, as a whole, with no option to restore just some entries), using **zfs send** *replication streams* based on snapshots created by other means.

ZFS backup approaches compared

ZFS is a sophisticated filesystem manager that lends itself to several backup approaches depending on need (and different approaches can be used on different datasets in the same pool). For two other approaches available in Amanda, see amzfs-sendrecv(8) and amzfs-snapshot(8).

amzfs-snapshot

> A script that can be combined with a conventional file-based archiving application, such as amgtar(8), when the ability to restore individual files will be important. The script creates a new ZFS snapshot, the archiver copies files from the snapshot in the usual way (but without the inefficient side effect of clobbering access times, because the snapshot is read-only), and then the script destroys the snapshot.
>
> Individual files can be recovered in the usual way, though how much advanced metadata gets preserved will depend on what the archiving tool supports. Strictly ZFS-specific features "outside" the filesystem, such as historical snapshots and filesystem properties, are not preserved.

amzfs-sendrecv

> An application that uses **zfs send** to faithfully preserve the contents of a single ZFS filesystem at a single point in time. It creates a new snapshot as of the time it is executed, and can leave the snapshot in place when done so that a future run can be incremental. An incremental run advances the filesystem contents to the just-created new snapshot, but does not preserve intermediate snapshots along the way.

amzfs-holdsend

> *This application*: uses the *replication stream* feature of **zfs send** to preserve a ZFS filesystem or volume and any descendants. This application does not create or destroy snapshots on its own, but selects recent snapshots assumed to be created by other means, such as a scheduler that may take snapshots at regular intervals. Existing historical snapshots are preserved in the backup.

In general, approaches that use a conventional archiver have the advantage that individual files are easily restored when needed, and (typically, at least) a few disadvantages: they may not preserve all advanced metadata (ACLs, security contexts, etc.) supported by the filesystem, identification of changed files for incremental backup may require traversing the whole filesystem making fallible comparisons, and increments may grow larger than necessary by including entire files that have seen but small changes.

This application, as well as **amzfs-sendrecv**, take the complementary approach. While they do not provide for easy recovery of individual files, they faithfully preserve all of the filesystems' contents and metadata, identify all changed objects in an increment without relying on fallible checks, and require only the size of changed data, even with small changes to large files.

In an environment where nothing else is creating snapshots, and there is no desire to preserve anything but the most recent state at each backup, **amzfs-sendrecv** may be a simple solution, as it takes care of creating its own snapshot on each backup run.

If snapshots are being created by other means, such as on a frequent schedule between backups, or if intermediate snapshots between scheduled backups should all be preserved, **amzfs-holdsend** fits the bill. It does not add to or delete from the existing snapshots, but preserves them as they are.

Where supported by the ZFS implementation in use, backup size and CPU utilization can both be reduced for compressed datasets by setting the UNCOMPRESSED property to false.

Multiple datasets in one DLE

ZFS arranges datasets (its generic term for both filesystems and block-device-like volumes) in a tree. The tree structure does not have to mirror the desired layout of filesystem mount points, but can be used to group datasets meant to be similar in other ways, compression settings for example, or datasets to be backed up together.

For **amzfs-holdsend**, the *disk* or *device* given in a disk-list entry names a subtree of datasets to back up. It must always be a ZFS-style dataset name, not a mountpoint, to make clear that it points into the ZFS namespace hierarchy rather than the filesystem namespace, and the backup will include that dataset and any others descended from it in the ZFS tree.

For a subtree to be backed up at once by **amzfs-holdsend**, certain conditions must hold. For a full (level 0) backup, there must be at least one snapshot name that is present on every dataset in the subtree. An easy way to ensure that is to use `zfs snapshot -r ...` to create snapshots with identical names on a dataset and all its descendants in one command. However, the stream generation does not depend on that: there is no requirement that the same-named snapshots were created together, or even closely in time; they simply have to exist. If **amzfs-holdsend** will not back up a certain subtree because no common snapshot name can be found, if there is a snapshot name that is present in most of the datasets, the remedy can be as simple as creating snapshots by that name in the datasets lacking it. When **amzfs-holdsend** can identify one or more snapshot names present in the named dataset and all descendants, it will use the latest of those for a level-0 backup.

Note

You could be sly and create some snapshots with matching names everywhere in the subtree, but not always in the same order. When **amzfs-holdsend** picks the "latest" of them, it uses the one that is latest in the topmost dataset, the one that the DLE names.

For an *incremental* backup to be produced, the snapshots used for the next lower backup level must still be present, and must still all have the same name. (If any has been renamed, so must they all.) For a useful backup, there must also be some newer snapshots with their own common name throughout the subtree; again, the "latest" will be used as the new end point for the incremental backup.

Note

You could be sly again and arrange for a later snapshot in some datasets to match the name of a snapshot that came earlier than the prior-level backup snapshot in some other datasets in the same subtree. You won't get away with it, however.

If no new snapshot has been created since the last backup, a (trivial) incremental backup can still be created, from the last-used snapshot to itself. There are warnings from **zfs send** in that case, but it successfully produces a restorable increment with no filesystem-content changes. It will include the latest values of any changed ZFS properties.

Holds on snapshots

After a backup, **amzfs-holdsend** places **zfs hold** tags on the snapshots that must be kept around for future increments. The tags have a recognizable prefix followed by the backup level, and serve two purposes. For this application, they record which backup levels used which snapshots; for ZFS itself, they protect the snapshots from inadvertent deletion.

So that a dataset or subtree can belong to more than one DLE (to back up to more than one media series, or onsite/offsite, etc.), HOLDTAGPREFIX is a property that can be set in a DLE or dumptype. With the different DLEs using different prefixes, the hold tags they generate can coexist.

Note

As **amzfs-holdsend** releases some holds and places others when a backup completes, there can be a moment when it has no holds on any of the snapshots needed for future increments. That should not be a problem, unless many snapshots have been queued up for destruction with **zfs destroy -d**, in which case they could all vanish before a new hold can be placed, making that a risky practice.

Note

When attempts to destroy a filesystem or snapshot are met with "snapshot is busy" complaints from ZFS, the holds are doing their job. When destroying the filesystem or snapshot really is what you want to do, manually-issued **zfs release** commands (after sober reflection) can remove the conflicting holds.

Permissions needed

Because **amzfs-holdsend** only needs to create and release holds and invoke **zfs send**, it will work with just those three ZFS permissions delegated to the Amanda backup user:

```
zfs allow -u amandabackup hold,send,release filesystem
```

Additional permissions are needed for recovery, but **amrecover** is typically run as superuser, so there is no need to grant more permissions to the backup user.

On a platform that lacks **zfs allow** support, it may be necessary to use **pfexec**, **sudo**, or a custom setuid wrapper for the **zfs** command.

Estimate method

The estimate methods CLIENT and CALCSIZE are supported. CALCSIZE is both fast and accurate, but works only on platforms that support **zfs send -nvP** as found in OpenZFS. CLIENT is slower, but works on platforms without that support.

PROPERTIES

This section lists the properties that control amzfs-holdsend's functionality. See amanda-applications(7) for information on application properties and how they are configured.

DEDUP
: Pass the `-D` option to **zfs send**, for platforms that support it to exclude duplicated blocks from the generated stream. See zfs(8) for details.

EMBED
: Pass the `-e` option to **zfs send**, for platforms that support it to generate a smaller stream from a pool that uses the *embedded_data* feature. See zfs(8) for details.

HOLDTAGPREFIX
: All hold tags placed by this application on snapshots will consist of this prefix, a single space, and a number representing the backup level. If one ZFS dataset or subtree is to participate in more than one DLE (for different backup media series, onsite/offsite, etc.), this property can be used to set distinct prefixes for the DLEs, so the holds they create do not interfere. The default is the string "org.amanda holdsend".

LARGE-BLOCK
: Pass the `-L` option to **zfs send**, for platforms that support it to allow blocks larger than 128 kB in the generated stream. See zfs(8) for details.

RAW
: Pass the `-w` option to **zfs send**, for platforms that support it to send data exactly as found on disk, allowing encrypted datasets to be backed up even when their encryption keys are not loaded, and stored securely. See zfs(8) for details.

UNCOMPRESSED
: Defaults to true, because not all ZFS implementations suppport compressed streams (**zfs send -c** where `-c` means 'compressed', as in OpenZFS, not 'contained' as in Solaris). If compressed-stream support is available, setting this property to false allows the backup stream to be generated in the same compressed form used in the dataset, saving both space and CPU cycles, as the stream will not get uncompressed for backup and recompressed when restored. If sending from a pool with the *large_blocks* feature, setting this property false may be ineffective unless the LARGE-BLOCK property is also set true.

ZFSEXECUTABLE
: Full path to the **zfs** executable on the client system.

Properties specific to recovery

The properties in this section apply to recovery only. They can be set using the **setproperty** command within an amrecover(8) session.

DATASET
: The ZFS name under which to recover the data. Because ZFS recovery involves whole filesystems at a time, to reduce the risk of clobbering wanted filesystems by mistake, this property does *not* default to the device named in the DLE; it must be set explicitly with **setproperty** ahead of an **extract** within **amrecover**. There is no prohibition on setting it the same as the device in the DLE if desired, though it is also easy to set it to a different name and rename it after successful recovery.

DESTRUCTIVE
: Controls whether the `-F` is given to **zfs receive** when restoring increments (it will always be used when restoring the level 0 base, regardless of this property). The default is true, meaning descendant datasets and snapshots that disappear in a later increment will be destroyed as that increment is replayed, so the final result matches the original. This property can be set to false, if desired, so that destructions of datasets or snapshots will not be replayed.

EXCLUDEPROPERTY

Takes one or more ZFS dataset property names, which will not be applied from the recovered backup stream (so the values of those properties in the received dataset will be defaulted or inherited in the usual way, as if no value had been saved in the backup stream. Not every platform supports the `-x` to **zfs receive**; use of this property on platforms that do not will cause recovery to fail.

OVERRIDEPROPERTY
: Takes one or more values of the form property=value, which will be applied in the received dataset inplace of values saved in the backup stream. Not every platform supports the `-o` to **zfs receive**; use of this property on platforms that do not will cause recovery to fail.

UNMOUNTED
: Controls whether the `u` is given to **zfs receive** to suppress automatic mounting of datasets received.

EXAMPLE

```
define application-tool app_amzfsholdsend {
  plugin "amzfs-holdsend"
}
```

A dumptype using this application might look like:

```
define dumptype amzfshold {
  global
  program "APPLICATION"
  application "app_amzfsholdsend"
}
```

Note that the *program* parameter must be set to "*APPLICATION*" to use the *application* parameter.

SEE ALSO

amanda(8), amanda.conf(5), amanda-applications(7), amrecover(8)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Amanda::Script::Abstract

NAME

Amanda::Script::Abstract - A base class for Amanda scripts

SYNOPSIS

```
    package amMyScript;
    use base qw(Amanda::Script::Abstract)

    package main;
    amMyScript::->run();
```

DESCRIPTION

`Amanda::Script::Abstract` handles much of the common housekeeping needed to implement Amanda's Script API, so that practical applications can be implemented in few lines of code by overriding only a few necessary methods.

`Amanda::Script::Abstract` is itself a subclass of `Amanda::Script`, but the latter cannot be instantiated until after the command line has been parsed (at least the *execute-where* argument and `config` options need to be passed to its constructor). Therefore, `Amanda::Script::Abstract` supplies *class* methods for the preliminary work of declaring the supported options, parsing the command line, and finally calling `new` to get an instance of the class. Then the actual operations of the script are carried out by instance methods, as you would expect.

In perl, class methods are inheritable and overridable just as instance methods are, and the syntax `$class->SUPER::method(...)` works, analogously to its familiar use in an instance method. So, the pre-instantiation behavior of a script (declaring command options, etc.) can be tailored by simply overriding the necessary class methods, just as its later operations can be supplied by overriding instance methods.

FUNDAMENTAL CLASS METHODS

run

```
    scriptclass::->run()
```

Run the script: grab the *execute-where* from `ARGV[0]`, be sure it is a known one, set up for option parsing, parse the options, construct an instance passing all the supplied options (which will include the config name, needed by the `Amanda::Script` constructor), and finally `do()` the *execute-where*, where `do` is inherited from `Amanda::Script_App` by way of `Amanda::Script`.

new

```
    $class->new(\%opthash)
```

A typical application need not call this; it is called by `run` after the command line options have been declared and the command line has been parsed. The hash reference contains option values as stored by `GetOptions`. A certain convention is assumed: if there is an option/property named *o*, its value will be stored in *opthash* with key exactly *o*, in exactly the way `GetOptions` normally would, *unless* the option-declaring method had planted a code reference there in order to more strictly validate the option. In that case, of course *$opthash{o}* is the code reference, and the final value will be stored with key `opt_`*o* instead. Because of that convention, entries in *opthash* with an `opt_` prefix will otherwise be ignored; avoid declaring options/properties whose actual names start with `opt_`.

CLASS METHOD FOR DECLARING ALLOWED OPTIONS / PROPERTIES

declare_options

```
    declare_options(\%opthash, \@optspecs)
```

This is a class method that is passed two references: a hash reference *\%opthash* and a list reference *\@optspecs*. It should push onto *\@optspecs* the declarations of whatever options are valid for the script, using the syntax described in Getopt::Long. It does not need to touch *\%opthash*, but may store a code reference at the name of any option, to apply stricter validation when the option is parsed. In that case, the code reference should ultimately store the validated value into *\%opthash* at a key derived by prefixing `opt_` to the option name.

ABOUT THIS PAGE

This page was automatically generated Wed Aug 30 14:58:08 2017 from the Amanda source tree, and documents the most recent development version of Amanda. For documentation specific to the version of Amanda on your system, use the 'perldoc' command.

Name

am389bak — Amanda script for online copy of a 389 Directory Server

DESCRIPTION

am389bak is an Amanda script implementing the Script API. It should not be run by users directly. It uses the **db2bak** command to make a consistent copy of a 389 Directory Server instance.

The name of the directory server instance (the *foo* part, if `slapd-foo` is the directory where its files are found) is given in the INSTANCE property, and the consistent copy will be made into the directory named by *diskdevice* in the disklist (DLE). This is the directory where the estimate and backup application (amopaquetree(8) could be appropriate) will expect to find the data for backup.

The copy is made after *removing all existing content under the directory named as the diskdevice*, all of which happens during PRE-DLE-ESTIMATE, which must be set to be executed on the client:

```
    execute-on  pre-dle-estimate
    execute-where client
```

The script is run as the amanda user, and must be able to run the **db2bak** command as the user running 389. This can be done by setting the DB2BAKEXECUTABLE property not to the path of **db2bak** itself, but to a set-uid *and set-gid* wrapper that obtains the effective user *and group* IDs of the 389 user, copies those to the real IDs (**db2bak** is quite finicky), and finally spawns **db2bak** with the same arguments. The wrapper should take all appropriate precautions to avoid being misused, such as checking the arguments to verify that they match a backup request of an expected 389 instance.

Note

If the wrapper is made both set-uid and set-gid to the user that runs 389, there is no way left (in traditional Unix permissions) to leave the wrapper executable by the Amanda user (except by making it executable to everyone, which misses the point). However, modern filesystems typically support access control lists, so the wrapper can be given an ACL with execute permission granted to the Amanda user specifically.

PROPERTIES

This section lists the properties that control am389bak's functionality. See amanda-scripts(7) for information on the Script API, script configuration.

DB2BAKEXECUTABLE
: Path to the **db2bak** executable, search in $PATH by default. In realistic settings, this will point instead to a set-uid and set-gid wrapper that (after appropriate checks) will execute **db2bak** with the same arguments.

INSTANCE
: Name of the 389 Directory Server instance (the *foo* part, if `slapd-foo` is the directory where its files are found) to be backed up.

EXAMPLE

This example defines a script `make389bak`, using it with an (assumed defined) `app_amopaquetree` application (see amopaquetree(8)) to do incremental backup of the directory data for instance `myinst`.

In amanda.conf(5):

```
  define script "make389bak" {
    plugin "am389bak"
    execute-where client
    execute-on pre-dle-estimate
    property "db2bakexecutable" "/path/to/db2bak-wrapper"
  }
```

In disklist(5):

```
ldaphost myinst /tmp/389bak {
  global
  estimate client
  program "APPLICATION"
  application "app_amopaquetree"
  script {
    "make389bak"
    property "instance" "myinst"
  }
} 1 enet100
```

BUGS

This script does not (yet) make any effort to detect failures.

The **db2bak** utility does so many inconvenient things that there is extra cleanup work needed when it is done. As that work also needs to be done as the 389 user, it is most easily done in the set-uid wrapper, which therefore cannot simply execute **db2bak** after some setup, but must execute it in a subprocess, wait for it, and then do further work.

- If the directory into which the backup is to be made already exists, **db2bak** first renames it to the same name with `.bak` tacked on. There is no documented option to turn off that behavior. Harmless enough, unless *that* directory also already exists, in which case **db2bak** simply fails. So the wrapper may need to remove any such directory so that situation is avoided.
- When **db2bak** creates files in the destination directory, it sets their modes explicitly to disallow any group access, which complicates making the files readable to the Amanda user after the script completes and the backup application needs to read them. A POSIX default ACL on the destination directory will not suffice, because even though the files inherit it, the later explicit denial of group permission masks it off. So the wrapper may need to change permissions or alter ACLs on the files after **db2bak** completes.

The number of tasks that have to be done under an ID other than Amanda's makes a case for adding some general, secure, easily configured way for Amanda applications and scripts to run specific operations with privilege, but that remains future work.

SEE ALSO

amanda(8), amanda.conf(5), amanda-client.conf(5), amanda-scripts(7)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amlibvirtfsfreeze — Amanda script to freeze/thaw filesystems in VMs

DESCRIPTION

amlibvirtfsfreeze is an Amanda script implementing the Script API. It should not be run by users directly. On a machine that hosts virtual machines, it uses libvirt to instruct a virtual machine to freeze one or more filesystems, and then to thaw them after their image files on the host have been backed up.

Freeze and thaw support in libvirt requires that a "guest agent" (for example, qemu-ga(8) for a Linux guest) be installed in the guest OS.

When a filesystem is frozen, any buffered changes are forced to storage so the image can be backed up while consistent at the filesystem level. Activity in the guest that modifies the filesystem may block until the filesystem is thawed, so this technique is best combined with a script (such as amlvmsnapshot(8)) to snapshot the host filesystem, so the VM guest filesystems need only be frozen long enough to create the snapshot on the host.

While a frozen filesystem will not be 'dirty' at the filesystem level, freezing alone does not ensure that any files in that filesystem represent consistent states of running applications. A guest agent may provide the option to run a freeze-hook script in the guest OS just before freezing and after thawing, and such a script can take application-specific actions to ensure important applications have reached a consistent state and forced it to storage as well. More on freeze hooks will be found in the documentation for the guest agent being used.

The guest virtual machine ("domain" in libvirt parlance) is specified by name in the DOMAIN property, and the MOUNTPOINT property (which can be multivalued) specifies which of that domain's filesystems to act on. The FREEZEORTHAW property specifies the action.

For a thaw action, no MOUNTPOINT is specified, and all filesystems frozen in the guest domain are thawed. For a freeze action, MOUNTPOINT may need to be omitted if the virtualization software or guest OS are too old to support per-filesystem freezing, and in that case, every filesystem mounted on the guest is frozen.

This script can be run in PRE-DLE-ESTIMATE, in the case where one snapshot will be made before estimating and removed after backup. If limited free space in the volume group forces a small snapshot size, or changes to the origin filesystem accumulate quickly, or the Amanda installation in total has a long delay in planning/dumping between the estimate and backup phases, it is possible to execute this script twice, on PRE-DLE-ESTIMATE and PRE-DLE-BACKUP, in conjunction with a snapshot script that removes the snapshot used for estimating and creates another one for backup. In that case, it would also be possible to run this script only for PRE-DLE-BACKUP, as the estimate phase can get the size of the image file accurately enough without needing it frozen.

The phase (or both phases) must be set to be executed on the client:

```
    execute-on  pre-dle-estimate
    #execute-on  pre-dle-estimate, pre-dle-backup
    execute-where client
```

The script is run as the amanda user, and must be able to run the necessary **virsh** command. One approach is to set the VIRSHEXECUTABLE property not to the actual path of the command, but to a set-uid wrapper that takes the same arguments, checks that they represent allowed operations, domains, and filesystems, and then executes the actual command with the same arguments.

Note

A set-uid wrapper obtains its owner's *effective* ID, but may need to set the *real* ID to match it before the **virsh** command will permit the operations.

PROPERTIES

This section lists the properties that control amlibvirtfsfreeze's functionality. See amanda-scripts(7) for information on the Script API, script configuration.

DOMAIN
: The name of the libvirt domain, or guest VM, whose filesystems should be frozen or thawed.

FREEZEORTHAW

One of the values `freeze` or `thaw`.

MOUNTPOINT
: Filesystems in the named domain to be frozen, identified by their mountpoints. This property can have multiple values. It is ignored in a `thaw` operation; all frozen filesystems will be thawed. For some old versions of virtualization infrastructure, the guest agent, or guest OS, it must not be used for `freeze` either, in which case all mounted filesystems will be frozen.

VIRSHEXECUTABLE
: Path to the **virsh** executable, search in $PATH by default. If necessary for authorization, this can be set instead to a set-uid wrapper that (after appropriate checks) will execute **virsh** with the same arguments.

EXAMPLE

This example sets up two script definitions, `domfsfreeze` and `domfsthaw`, both based on this plugin, and a DLE that uses them, along with amlvmsnapshot(8) and an assumed `remote-gtar` dumptype, to freeze the root filesystems of several guest VMs running on a host, then snapshot the host's filesystem (containing the consistent, frozen image files of the guests), thaw the guest filesystems, and finally back up the host from the snapshot.

Note the use of the ORDER option in script definitions. The default (applied to the `lvmsnapshot` script, which leaves it unspecified) is 5000. The explicit 3000 in the `domfsfreeze` definition and 7000 for `domfsthaw` ensure that, when all three scripts are run in a single phase such as PRE-DLE-ESTIMATE, the freezes occur first, then the snapshot, then the thaws.

In a realistic application where **amandad** does not run as the superuser, a privileged wrapper may need to be written and named in the VIRSHEXECUTABLE property as discussed above.

In amanda.conf(5):

```
define script domfsfreeze {
  plugin "amlibvirtfsfreeze"
  execute-where client
  execute-on pre-dle-estimate
  property "freezeorthaw" "freeze"
  property "virshexecutable" "/path/to/virsh-wrapper"
  order 3000 # must execute before lvmsnapshot
}

define script domfsthaw {
  plugin "amlibvirtfsfreeze"
  execute-where client
  execute-on pre-dle-estimate
  property "freezeorthaw" "thaw"
  property "virshexecutable" "/path/to/virsh-wrapper"
  order 7000 # must execute after lvmsnapshot
}

define script "lvmsnapshot" {
  plugin "amlvmsnapshot"
  execute-where client
  execute-on pre-dle-estimate, post-dle-backup
  # ... see amlvmsnapshot.8
}

define dumptype remote-gtar # ...
```

And in disklist(5):

```
example / /tmp/rootsnap {
  remote-gtar
  script {
    "domfsfreeze"
    property "domain" "vm1"
    property "mountpoint" "/"
  }
  script {
    "domfsthaw"
    property "domain" "vm1"
  }
  script {
    "domfsfreeze"
    property "domain" "vm2"
    property "mountpoint" "/"
  }
  script {
    "domfsthaw"
    property "domain" "vm2"
  }
  script {
    "lvmsnapshot"
    property "volumegroup"   "vg_example"
    property "logicalvolume" "lv_root"
    property "snapshotname"  "snap_root"
    property "extents"       "10%ORIGIN"
  }
} 1 enet100
```

BUGS

This script does not (yet) make any effort to detect failures, or to capture and interpret standard output from the **virsh**, which normally produces some. Uncaptured, such output mixes with Amanda's client/server messaging, leading to backup failures. A cheap workaround is to include `dup2(2,1)` in the wrapper that executes **virsh**, so such output goes to standard error and ends up in the client **amandad** log file.

SEE ALSO

amanda(8), amanda.conf(5), amanda-client.conf(5), amanda-scripts(7)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amlvmsnapshot — Amanda script to make an LVM snapshot

DESCRIPTION

amlvmsnapshot is an Amanda script implementing the Script API. It should not be run by users directly. It creates and mounts a snapshot of an LVM filesystem in preparation for backup, then unmounts and removes it.

The logical volume to be snapshotted is specified with the VOLUMEGROUP and LOGICALVOLUME properties, and the SNAPSHOTNAME property will be used to give the created snapshot a name. The EXTENTS property must be set to indicate the size of the snapshot. The mount point for the snapshot will be the directory named by *diskdevice* in the disklist (DLE). This is the directory where the estimate and backup application will expect to find the data for backup.

A snapshot begins with nearly zero space consumed, but grows (up to the specified size) as the origin filesystem accumulates changes made since the snapshot. The EXTENTS property must size the snapshot generously enough for the volume of changes expected in the origin filesystem during the snapshot's lifetime, while fitting in the volume group's available, unused space.

This script can be run in PRE-DLE-ESTIMATE and POST-DLE-BACKUP, in which case one snapshot will be made and mounted before estimating, and unmounted and removed after backup. If limited free space in the volume group forces a small snapshot size, or changes to the origin filesystem accumulate quickly, or the Amanda installation in total has a long delay in planning/dumping between the estimate and backup phases, it may be safer to execute this script four times, on PRE-DLE-ESTIMATE, POST-DLE-ESTIMATE, PRE-DLE-BACKUP, and POST-DLE-BACKUP. In that case, the snapshot used for estimating is unmounted and removed after that phase, and a new snapshot is created later for actual dumping.

The snapshot is made and mounted after *removing all existing content under the directory named as the diskdevice* if it exists, or creating it empty if it does not, which happens during PRE-DLE-ESTIMATE (and PRE-DLE-BACKUP if also executed then). Both (or all four) phases must be set to be executed on the client:

```
    execute-on  pre-dle-estimate, post-dle-backup
    #execute-on  pre-dle-estimate, post-dle-estimate, pre-dle-backup, post-dle-backup
    execute-where client
```

The script is run as the amanda user, and must be able to run the necessary **lvm**, **mount**, and **umount** commands. One approach is to set the LVMEXECUTABLE, MOUNTEXECUTABLE, and UMOUNTEXECUTABLE not to the actual paths of those commands, but to set-uid wrappers that take the same arguments, check that they represent allowed operations and filesystems, and then execute the actual commands with the same arguments.

Note

A set-uid wrapper obtains its owner's *effective* ID, but may need to set the *real* ID to match it before the **lvm**, **mount**, and **umount** commands will permit the operations.

PROPERTIES

This section lists the properties that control amlvmsnapshot's functionality. See amanda-scripts(7) for information on the Script API, script configuration.

EXTENTS
: The size to make the created snapshot. Syntaxes like `10%ORIGIN` or `50%FREE` are possible; see lvcreate(8). There must be at least this much unused space available in the same volume group as the origin filesystem, and this size must be large enough for all the expected modification activity for that filesystem during the snapshot's existence.

LOGICALVOLUME
: The name of the logical volume that will be the origin of the snapshot.

LVMEXECUTABLE
: Path to the **lvm** executable, search in $PATH by default. If necessary for authorization, this can be set instead to a set-uid wrapper that (after appropriate checks) will execute **lvm** with the same arguments.

MOUNTEXECUTABLE
: Path to the **mount** executable, search in $PATH by default. If necessary for authorization, this can be set instead to a set-uid wrapper that (after appropriate checks) will execute **mount** with the same arguments.

MOUNTOPTS

Mount options to be used when mounting the snapshot. Multiple values are accepted. If the filesystem is XFS, two mount options are necessary:

```
    property "mountopts" "nouuid" "norecovery"
```

SNAPSHOTNAME
: The name to be given to the newly-created snapshot.

UMOUNTEXECUTABLE
: Path to the **umount** executable, search in $PATH by default. If necessary for authorization, this can be set instead to a set-uid wrapper that (after appropriate checks) will execute **mount** with the same arguments.

VOLUMEGROUP
: Name of the volume group in which the logical volume to be snapshotted, and in which the snapshot will be created.

EXAMPLE

This example defines a script `lvmsnapshot` and a DLE that backs up a logical volume `lv_root` in volume group `vg_example` (assuming there is already a `remote-gtar` dumptype defined. A single snapshot will be used for both the estimate and backup phases. While in use, the snapshot will be mounted at `/tmp/rootsnap`, which will be created empty if absent, or emptied if present.

A dumptype `remote-gtar-root-tinysnap` dumptype is also created, setting the same VOLUMEGROUP, LOGICALVOLUME, SNAPSHOTNAME, and EXTENTS properties to avoid repeating them in DLEs for several identically-configured clients, and also overriding EXECUTE-ON to use a shorter-lived, smaller snapshot for each phase, in case a client may have insufficient free space in the volume group for all of the modification activity during the longer life of a single snapshot.

In a realistic application where **amandad** does not run as the superuser, privileged wrappers may need to be written and named in the LVMEXECUTABLE, MOUNTEXECUTABLE, and UMOUNTEXECUTABLE properties as discussed above.

In amanda.conf(5):

```
define script "lvmsnapshot" {
  plugin "amlvmsnapshot"
  execute-where client
  execute-on pre-dle-estimate, post-dle-backup
  property    "lvmexecutable" "/path/to/lvm-wrapper"
  property  "mountexecutable" "/path/to/mount-wrapper"
  property "umountexecutable" "/path/to/umount-wrapper"
}

define dumptype remote-gtar-root-tinysnap {
  remote-gtar
  script {
    "lvmsnapshot"
    property "volumegroup"   "vg_example"
    property "logicalvolume" "lv_root"
    property "snapshotname"  "snap_root"
    property "extents"       "2%ORIGIN"
    execute-on pre-dle-estimate, post-dle-estimate, pre-dle-backup, post-dle-backup
  }
}
```

And in disklist(5):

```
example / /tmp/rootsnap {
  remote-gtar
  script {
    "lvmsnapshot"
    property "volumegroup"   "vg_example"
    property "logicalvolume" "lv_root"
    property "snapshotname"  "snap_root"
    property "extents"       "10%ORIGIN"
  }
} 1 enet100

tiny / /tmp/rootsnap remote-gtar-root-tinysnap -1 enet100
```

BUGS

This script does not (yet) make any effort to detect failures, or to capture and interpret standard output from the **lvm**, which normally produces some. Uncaptured, such output mixes with Amanda's client/server messaging, leading to backup failures. A cheap workaround is to include `dup2(2,1)` in the wrapper that executes **lvm**, so such output goes to standard error and ends up in the client **amandad** log file.

Also, this script does not (yet) do anything to prevent **lvm** inheriting file descriptors other than standard input/output

/error that are used in Amanda's script API. That leads to benign warnings from **lvm** about file descriptor leaks. This also has a cheap workaround that can be added to the wrapper, namely to define `LVM_SUPPRESS_FD_WARNINGS` in the environment before executing **lvm**.

SEE ALSO

amanda(8), amanda.conf(5), amanda-client.conf(5), amanda-scripts(7)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.

Name

amsvnmakehotcopy — Amanda script to make a Subversion hotcopy

DESCRIPTION

amsvnmakehotcopy is an Amanda script implementing the Script API. It should not be run by users directly. It uses the `svnadmin hotcopy` command to make a consistent copy of a Subversion repository.

The filesystem path to the Subversion repository to copy is given in the SVNREPOSITORY property, and the consistent copy will be made into the directory named by *diskdevice* in the disklist (DLE). This is the directory where the estimate and backup application (amopaquetree(8) could be appropriate) will expect to find the data for backup.

The copy is made after *removing all existing content under the directory named as the diskdevice*, all of which happens during PRE-DLE-ESTIMATE, which must be set to be executed on the client:

```
    execute-on  pre-dle-estimate
    execute-where client
```

The script is run as the amanda user, and must be able to read the Subversion repository. Possible arrangements include making the repository generally readable, giving it filesystem ACLs granting the access specifically to the amanda user, or setting the SVNADMINEXECUTABLE property not to the path of **svnadmin**, but to a set-uid wrapper that obtains the needed permission and then executes **svnadmin** with the same arguments. If the wrapper approach is chosen, the wrapper should take all appropriate precautions to avoid being misused, such as checking the arguments to verify that they only request a hotcopy of the intended Subversion repository.

PROPERTIES

This section lists the properties that control amsvnmakehotcopy's functionality. See amanda-scripts(7) for information on the Script API, script configuration.

SVNADMINEXECUTABLE
: Path to the **svnadmin** executable, search in $PATH by default. If necessary to gain read access to the repository, this can be set instead to a set-uid wrapper that (after appropriate checks) will execute **svnadmin** with the same arguments.

SVNREPOSITORY
: Filesystem path to the Subversion repository that is to be copied.

EXAMPLE

This example defines a script `svnmakehotcopy`, a dumptype `remote-svn`, and a DLE to backup a particular repository.

In amanda.conf(5):

```
define script "svnmakehotcopy" {
  plugin "amsvnmakehotcopy"
  execute-where client
  execute-on pre-dle-estimate
}

define dumptype remote-svn {
  global
  estimate client
  program "APPLICATION"
  application "app_amopaquetree"
  comment "backup of remote svn repository"
}
```

In disklist(5):

```
 svnhost myrepo /tmp/svnmyrepocopy {
   remote-svn
   script {
       "svnmakehotcopy"
       property "svnrepository" "/var/www/svn/repos/myrepo"
   }
 } 1 enet100
```

BUGS

This script does not (yet) make any effort to detect failures.

SEE ALSO

amanda(8), amanda.conf(5), amanda-client.conf(5), amanda-scripts(7)

The Amanda Wiki: http://wiki.zmanda.com/

AUTHOR

This manual page was written by Chapman Flack.