MacOS X Package

Bareos MacOS X Package

We now have our first experimental MacOS X Package for download:

http://download.bareos.org/bareos/beta/13.2/macosx/

The package was built on **MacOS 10.8.4** and will only run on Intel Processors.

We expect that it will probably also work on **MacOS 10.7**.

It is a standard Mac Installer Package and delivered as disk image file (.dmg)

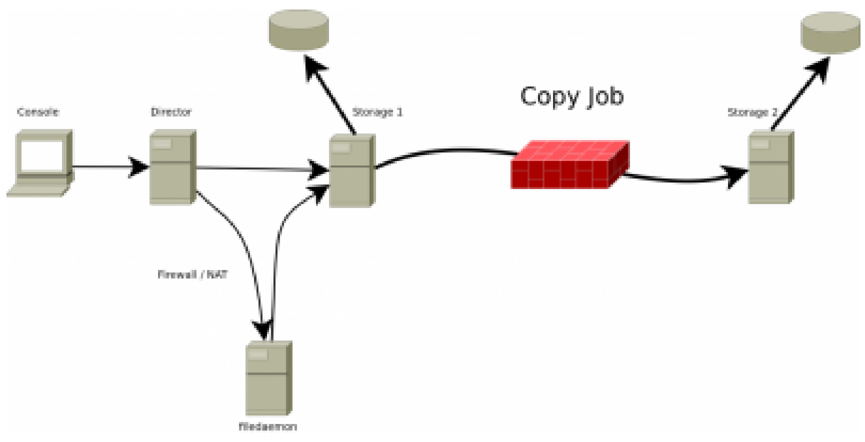
The software is installed to **/usr/local**, the config files reside in **/usr/local/etc**

Job migration/copy between different SDs

Job migration/copy between different SDs

The former copy/migration code was only able to do migrations/copies inside of one storage daemon.

With this new feature now copies and migrations can be made between two different storage daemons over the network. This makes copies and migrations much more usable and creates the possibility to really move copies outside of your datacenter.



Usage

The configuration of the remote replication is straight-forward and is exactly like copying and migration were configured before:

- You need two different storage resources configured in your director (e.g. storage1 and storage2).
- Each storage needs an own device and an individual pool(e.g. pool1, pool2).
- Each pool is linked to its own storage via the **storage** directive in the pool resource.
- To configure the migration from pool1 to pool2, the **Next Pool** directive of **pool1** has to point to **pool2**.
- The copy job itself has to be of type copy/migrate (exactly as already known in copy- and migration jobs)

Please view the following example:

```
#bareos-dir.conf

# Fake fileset for copy jobs
Fileset {
  Name = None
  Include {
    Options {
      signature = MD5
    }
  }
}

# Fake client for copy jobs
Client {
  Name = None
  Address = localhost
  Password = "NoNe"
  Catalog = MyCatalog
}


# Source storage for migration
Storage {
   Name = storage1
   Address = sd1.example.com
   Password = "secret1"
   Device = File1
   Media Type = File
}

# Target storage for migration
Storage {
   Name = storage2
   Address = sd2.example.com
   Password = "secret2"
   Device = File2
   Media Type = File2   # Has to be different than in storage1
}

Pool {
   Name = pool1
   Storage = storage1
   NextPool = File2    # This points to the target storage
}

Pool {
   Name = pool2
   Storage = storage2
}

Job {
   Name = CopyToRemote
   Type = Copy
   Messages = Standard
   Client = None
   FileSet = None
   Selection Type = PoolUncopiedJobs
   Spool Data = Yes
   Pool = pool1
}
```

Reverse data channel initialization aka passive clients

Reverse data channel initialization aka passive clients

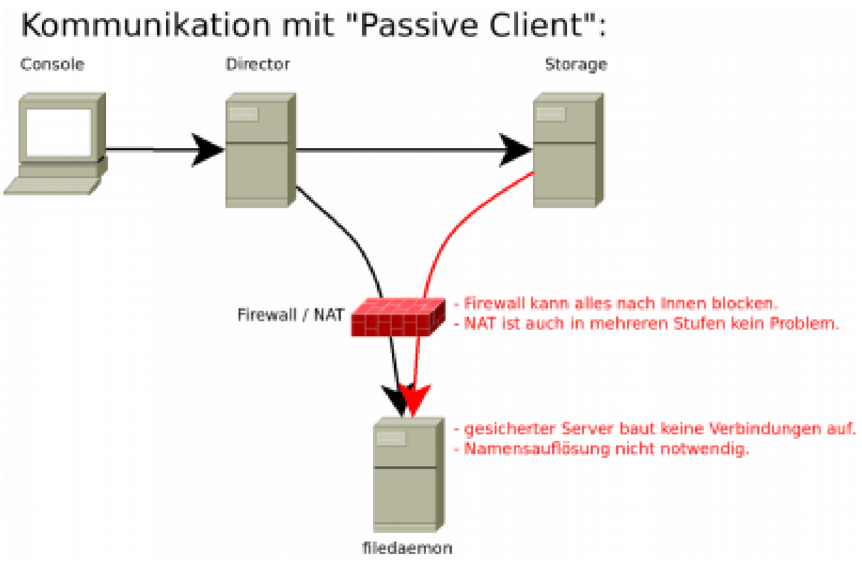
The normal way of initializing the data channel (the channel where the backupdata itself is transported) is done by the file daemon (client) that connects to the storage daemon.

In many setups, this can cause problems, because this means that:

- The client must be able to resolve the name of the storage daemon (Often not true, you have to do tricks with the hosts file)
- The client must be allowed to create a new connection.
- The client must be able to connect to the storage daemon over the network (often difficult over NAT or Firewall)

With this new feature, the initialization of the datachannel is reversed, so that the storage daemon connects to the filedaemon. This solves almost every problem created by Firewalls, NAT-gateways and resolving issues, as

- The storage daemon initiates the connection, and thus can pass thru the same or similar firewallrules that the director already has to access the fileadaemon.
- The client never initiates any connection, thus can be completely firewalled.
- The client never needs any name resolution and is totally independent from any resolving issues.


Kommunikation mit "Passive Client":
Console
Director
Storage
Firewall / NAT
- Firewall kann alles nach Innen blocken.
- NAT ist auch in mehreren Stufen kein Problem.
- gesicherter Server baut keine Verbindungen auf.
- Namensauflösung nicht notwendig.
filedaemon

Usage

To use this new feature, just configure **passive = yes** in the client definition of the director daemon.

```
# bareos-dir.conf

Client {
   Name = PassiveClient
   Password = "secretpassword"
   Passive = yes
   [...]
}
```

Also, you need to set **compatible = no** in the bareos-fd.conf configuration file:

```
# bareos-fd.conf
```

```
FileDaemon {                          # this is me
   Name = -fd
   [...]
   compatible = no

}
```

For further information on this topic, please refer to the Bareos documentation

Security enhancements

"Allowed job command" directive

We implemented a new security layer that allows an administrator to better filter what type of jobs the filedaemon should allow. Until now we had the -b (backup only) and -r (restore only) flags which could be specified at the startup of the filedaemon.

We added a new configuration keyword in the filedaemon config named **allowedjobcommand** which you can define globally for all directors (e.g. by adding it to the global filedaemon resource) or for a specific director when added to the director resource.

You specify all commands you want to be executed by the filedaemon. When you don't specify the option it will be empty which means all commands are allowed.

The following example shows how to use this functionality:

```
Director {
  Name =
  Password =
  Allowed Job Command = "backup"
  Allowed Job Command = "runscript"
}
```

e.g. you specify all commands that are allowed each on a newline with the **allowedjobcommand** keyword.

The following job commands are recognized:

- backup - allow backups to be made
- restore - allow restores to be done
- verify - allow verify jobs to be done
- estimate - allow estimate cmds to be executed
- runscript - allow runscripts to run

We only filter the important commands the filedaemon can perform and not all commands as some commands are part of the above protocols and by disallowing the action the other commands are not invoked at all.

Things like admin jobs are director only and may lead to runscripts on the filedaemon being executed so they are not filtered in the filedaemon as they don't exist there as such.

If you don't use runscripts it would be a good security measure to disable running those e.g. only allow the commands that you really want to be used. Runscripts are particularly a problem as they allow the filedaemon to run arbitrary commands. You may also look into the **allowedscriptdir** keyword to limit the impact of the runscript command.

"Allowed scriptdir" directive

We also implemented an other security enhancement that limits the impact of the runscript command of the filedaemon. You can now configure a **allowedscriptdir** keyword either for all directors (e.g. by adding it to the global filedaemon resource) or for a specific director when added to the director resource.

You specify all directories in which the scripts or commands are located that you allow to be run by the runscript command of the filedaemon. Any program not in one of these paths (or subpaths) cannot be used. The implementation checks if the full path of the script starts with one of the specified paths.

The following example shows how to use this functionality:

```
Director {
  Name =
  Password =
  Allowed Script Dir = "/etc/bareos"
  Allowed Script Dir = "/somewhereelse"
}
```

e.g. you specify all directories in which the scripts/programs can reside.

With the **allowedjobcommand** and **allowedscriptdir** you should be able to work around any concerns your security officer has regarding the security concerns of the Bareos filedaemon being exploited.

Allow for relaxed TLS configuration for console connections.

Until now, the verify_peer flag is hardcoded to **yes** for the console programs.

Now, you can set **TLS Verify Peer = No** in the

- bconsole.conf
- bat.conf and
- bareos-fd.conf

configuration files when using TLS.

```
   # relaxed tls configuration
   # has security implications, attention
   TLS Verify Peer = No
```

Encryption cipher can be chosen now

Until now, the crypto cipher was hardcoded to aes128, while the crypto framework supports much more. Depending on the openssl library version different ciphers are available.

Usage

To chose the desired cipher, configure the **PKI Cipher** option in the filedaemon configuraton and set **compatible** to **no**:

```
FileDaemon {
   Name = client-fd
   # encryption configuration
   PKI Signatures = Yes           # Enable Data Signing
   PKI Encryption = Yes           # Enable Data Encryption
   PKI Keypair = "/etc/bareos/client-
fd.pem"    # Public and Private Keys
   PKI Master Key = "/etc/bareos/master.cert"    # ONLY the Public Key

   # choose encryption cipher
   compatible = no            # PKI Cipher is not bacula compatible
   PKI Cipher = aes128        # specify desired PKI Cipher here
}
```

The available options (and ciphers) are:

- aes128
- aes192
- aes256
- camellia128
- camellia192
- camellia256
- aes128hmacsha1
- aes256hmacsha1
- blowfish

They depend on the version of the openssl library installed.

For decryption of encrypted data, the right decompression algorithm should be automatically chosen.

Implementation of LZ4 and LZ4HC compression

LZ4 is a fast lossless compression algorithm. Details can be found on http://code.google.com/p/lz4

In addition to the available compression algorithms

- **gzip** and
- **lzo** we now also support
- **lz4 compression** with the flavors
  - **lz4**
  - **lz4hc** (lz4 high compression mode) and
- **lzfast**

Usage

To use the lz4 compression feature, you need to do two things:

1. As lz4 compression is not supported by Bacula, you need to set the **compatible = no** in **bareos-fd.conf**
2. Set the desired compression algorithm in the fileset options as usual:

   ```
   FileSet {
     Name = "lz4Set"
     Include {
       Options {
         signature=MD5
         compression=lz4 # or lzfast / lz4hc
       }
       File = /
     }
   }
   ```

The jobreport will show the compression ratio and the used compression algorithm like this:

```
Software Compression: 43.9 % (lz4)
```

Allow cancel by JobId on storage daemon

Sometimes the Director already removed the Job from its running queue but the Storage daemon still thinks it doing a backup (or other Job) and you cannot cancel the Job from within a console anymore.

Allows you to cancel a Storage Daemon Job by JobId.

1. Do a **`status storage`** on the Storage Daemon make sure what Job you want to cancel

2. execute a **`cancel storage= Jobid=`**

This way you can also remove a Job that blocks any other Jobs from running without the need to restart the whole Storage Daemon.

cancel multiple jobs at once

Especially when testing, it would be a very nice feature to be able to cancel multiple jobs at once.

We implemented the following extra options for the cancel cmd to select which jobs to cancel:

- all jobs
- all jobs with created state.
- all jobs with a blocked state.
- all jobs with a waiting state.
- all jobs with a running state.

Usage

```
cancel all
cancel all state=
```