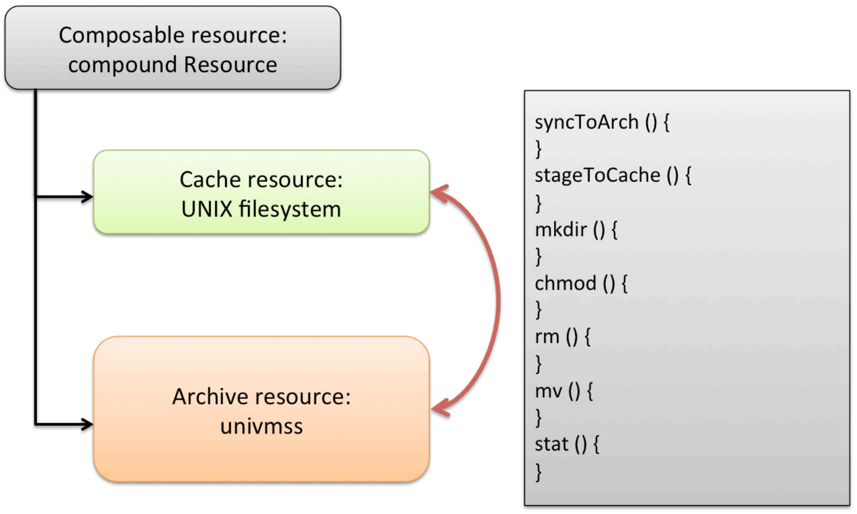
iRODS compound resources

Explanations and guidance for iRODS system administrators.

Context

Compound resources allow to install cache-archive constructs. I.e. you have an iRODS cache resource e.g. of type UNIX file system which communicates with an archive resource that can be of any storage type, even the ones iRODS does not support via a plugin. You will have to setup the two resources, link them in a composable resource fo type compound and you have to provide a script that defines how the cache resource communicates with the archive resource.



univMSSInterface.sh

This tutorial takes you through the steps how to attach an archive resource which in fact is a simple account on another linux machine.

Setting up the resource tree

We need to set up a cache resource, archive resource and group them in a composable resource.

First create the local unix folder that serves as storage for the cache resource:

```
sudo mkdir /archiveCache
sudo chown -R <user who runs iRODS> /archiveCache
```

Now we create the iRODS resources and group them

```
iadmin mkresc  archiveCache unixfilesystem <fqdn>:/archiveCache
iadmin mkresc archiveResc univmss <fqdn>:/home/<USER>/iRODS-archive \
    univMSSInterface_archive.sh

admin mkresc archive compound
iadmin addchildtoresc archive archiveCache cache
iadmin addchildtoresc archive archiveResc archive
```

In both cases *fqdn* is the fully-qualified domain name of the iRODS server. The archive resource is of type *univmss*, the path after the fully-qualified domain name is the path on the other server that should be used as base path for all data coming in from iRODS. We also have to provide the script which will define how interact with the archive resource (see section below **Writing the driver script**).

Check the resource tree:

```
archive:compound
├── archiveResc:univmss
└── archiveCache
```

And check the archive resource:

```
ilsresc -l archiveResc

resource name: archiveResc
id: 12319
zone: aliceZone
type: univmss
class: cache
location: <fqdn>
vault: /home/<USER>/iRODS-archive
free space:
free space time: : Never
status:
info:
comment:
create time: 01492524925: 2017-04-18.16:15:25
modify time: 01493796575: 2017-05-03.09:29:35
children:
context: univMSSInterface_archive.sh
parent: archive
object count: 0
```

You could already put data into the composable resource and the data will enter the cache resource. However, you will find errors in the server log, since iRODS does not know yet how to replicate the data to the archive resource.

Writing the driver script

The driver-script needs to be located in */var/lib/irods/iRODS/server/bin/cmd* (iRODS 4.1) or */var/lib/irods/cmd/exec* (iRODS 4.2). The name for the script needs to match the name in the *context* field of the archive resource.

The driver script defines the following functions:

```
# function for the synchronization of file $1 on local disk resource
# to file $2 in the MSS
syncToArch ()

# function for staging a file $1 from the MSS to file $2 on disk
stageToCache ()

# function to create a new directory $1 in the MSS logical name space
mkdir ()

# function to modify ACLs $2 (octal) in the MSS logical name space
# for a given directory $1
chmod ()

# function to remove a file $1 from the MSS
rm ()

# function to rename a file $1 into $2 in the MSS
mv ()

# function to do a stat on a file $1 stored in the MSS
stat ()
```

You will find and example script that employs the local commands *cp*, *mkdir*, *rm*, etc. */var/lib/irods/iRODS/server/bin/cmd/univMSSInterface_scp.sh*

We will adjust this script to work with a remote linux server.

Login as user who runs iRODS (in most cases *irods*) and make a copy

```
sudo su - irods
cp <path/to>/univMSSInterface.sh <path/to>/univMSSInterface_archive.sh
```

0. Authentication

Usually when we connect to a remote server we will use a password to authenticate. This is not possible in this setup. Instead of a password we will use an RSA-key (public-private key authentication). Please follow this setup to create a public and private key on the iRODS server under the account of the user who runs iRODS and export the public key to the respective account on the remote linux server that serves as archive.

1. syncToArch

We will simply use the *scp* command to transfer data to the other linux server under our user account. Hence the function to copy data over will look like this:

```
export USER="<your username at remote server>"
export ARCHIVEADDRESS="<fully-qualified domain name of the linux server>"
export SCPCOMMAND=/usr/bin/scp # test with `which scp`
export KEY=/var/lib/irods/.ssh/id_rsa # your private key

syncToArch () {
        # <your command or script to copy from cache to MSS> ${1:?} ${2:?}
        # e.g: /usr/local/bin/rfcp ${1:?} rfioServerFoo:${2:?}
        echo "UNIVMSS ${SCPCOMMAND} -i ${KEY} ${1:?} ${USER}@${ARCHIVEADDRESS}:${2:?}"
        "${SCPCOMMAND}" -i "${KEY}" "${1:?}" "${USER}@${ARCHIVEADDRESS}:${2}"
        return
}
```

You can test the function from commandline with:

```
./univMSSInterface_scp.sh syncToArch "test.txt" "/home/<user>/test"
```

This command will use the function to copy data between your iRODS user account and your linux-archive user account.

2. stageToCache

The *stageToCache* function works exactly the same as the function above but then uses the other direction to transfer the data:

```
stageToCache () {
        # <your command to stage from MSS to cache> ${1:?} ${2:?}
        # e.g: /usr/local/bin/rfcp rfioServerFoo:${1:?} ${2:?}
        #op=`which cp`
        #`$op ${1:?} ${2:?}`
        #echo "UNIVMSS $op ${1:?} ${2:?}"
        echo "UNIVMSS ${SCPCOMMAND} -i ${KEY} ${USER}@${ARCHIVEADDRESS}:${1:?} ${2:?}"
        ${SCPCOMMAND} -i ${KEY} ${USER}@${ARCHIVEADDRESS}:${1:?} ${2:?}
        return
}
```

Testing the function:

```
./univMSSInterface_scp.sh syncToArch "/home/<user>/test" "test1.txt"
```

3. mkdir

```
mkdir () {
        # <your command to make a directory in the MSS> ${1:?}
        # e.g.: /usr/local/bin/rfmkdir -p rfioServerFoo:${1:?}
        #ssh remote-host 'mkdir -p foo/bar/qux'
        echo "UNIVMSS ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} mkdir -p ${1:?}"
        ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} "mkdir -p ${1:?}"
        return
}
```

Testing the creation of directories on the remote linux server:

```
./univMSSInterface_archive.sh mkdir "test"
```

Exercise

Test the functions we have just implemented and find out where the data is automatically created.

4. chmod

iRODS sets all data on the remote linux server to mode 600 after copying, so we need to provide it with a function how to do that.

```
chmod () {
        # <your command to modify ACL> ${2:?} ${1:?}
        # e.g: /usr/local/bin/rfchmod ${2:?} rfioServerFoo:${1:?}
        ############
        # LEAVING THE PARAMETERS "OUT OF ORDER" (${2:?} then ${1:?})
        #    because the driver provides them in this order
        # ${2:?} is mode
        # ${1:?} is directory
        ############
        #op=`which chmod`
        #`$op ${2:?} ${1:?}`

        echo "UNIVMSS ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} chmod ${2:?} ${1:?}"
        return
}
```

5. rm

```
rm () {
        # <your command to remove a file from the MSS> ${1:?}
        # e.g: /usr/local/bin/rfrm rfioServerFoo:${1:?}
        #op=`which rm`
        #`$op ${1:?}`
        echo "UNIVMSS ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} rm -rf ${1:?}"
        ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} "rm ${1:?}"
        return
}
```

6. mv

The *mv* command is not only important to enable the iRODS *imv* command, but also to delete data in iRODS (See section on tests below.).

```
mv () {
        # <your command to rename a file in the MSS> ${1:?} ${2:?}
        # e.g: /usr/local/bin/rfrename rfioServerFoo:${1:?} rfioServerFoo:${2:?}
        #op=`which mv`
        #`$op ${1:?} ${2:?}`
        echo "UNIVMSS ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} mv ${1:?} ${2:?}"
        ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} "mv ${1:?} ${2:?}"
        return
}
```

7. stat

The *stat* command will be executed to check the remote system status before and after an action. Here we are using the *stat* command on a remote linux system. The main goal of the function is to define, when a file already exists or not. If you are using a different storage system you might have to adjust this function even more. For a gridFTP-stat we refer to the example script here.

```
stat () {
        #op=`which stat`
        #output=`$op ${1:?}`
        # <your command to retrieve stats on the file> ${1:?}
        # e.g: output=`/usr/local/bin/rfstat rfioServerFoo:${1:?}`
        #error=$?
        #if [ $error != 0 ] # if file does not exist or information not available
        #then
        #       return $error
        #fi
        # parse the output.
        # Parameters to retrieve: device ID of device containing file("device"),
        #                         file serial number ("inode"), ACL mode in octal ("mode"),
        #                         number of hard links to the file ("nlink"),
        #                         user id of file ("uid"), group id of file ("gid"),
        #                         device id ("devid"), file size ("size"), last access time
        #                         last modification time ("mtime"), last change time ("ctim
        #                         block size in bytes ("blksize"), number of blocks ("blkcn
        # e.g: device=`echo $output | awk '{print ${3:?}}'`
        # Note 1: if some of these parameters are not relevant, set them to 0.
        # Note 2: the time should have this format: YYYY-MM-dd-hh.mm.ss with:
        #                                           YYYY = 1900 to 2xxxx, MM = 1 to 12, dd
        #                                           hh = 0 to 24, mm = 0 to 59, ss = 0 to 5

        # Get the stat info from the remote server
        output=`ssh -i ${KEY} ${USER}@${ARCHIVEADDRESS} "stat ${1:?}"`
        #echo $output
        error=$?

        if [ $error != 0 ] # if file does not exist or information not available
        then
                return $error
        fi

        device=` echo $output | sed -nr 's/.*\<Device: *(\S*)\>.*/\1/p'`
        inode=`  echo $output | sed -nr 's/.*\<Inode: *(\S*)\>.*/\1/p'`
        mode=`   echo $output | sed -nr 's/.*\<Access: *\(([0-9]*)\/.*/\1/p'`
        nlink=`  echo $output | sed -nr 's/.*\<Links: *([0-9]*)\>.*/\1/p'`
        uid=`    echo $output | sed -nr 's/.*\<Uid: *\( *([0-9]*)\/.*/\1/p'`
        gid=`    echo $output | sed -nr 's/.*\<Gid: *\( *([0-9]*)\/.*/\1/p'`
        devid="0"
        size=`   echo $output | sed -nr 's/.*\<Size: *([0-9]*)\>.*/\1/p'`
        blksize=`echo $output | sed -nr 's/.*\<IO Block: *([0-9]*)\>.*/\1/p'`
        blkcnt=` echo $output | sed -nr 's/.*\<Blocks: *([0-9]*)\>.*/\1/p'`
        atime=`  echo $output | sed -nr 's/.*\<Access: *([0-9]{4,}-[01][0-9]-[0-3][0-9]) *(
        mtime=`  echo $output | sed -nr 's/.*\<Modify: *([0-9]{4,}-[01][0-9]-[0-3][0-9]) *(
        ctime=`  echo $output | sed -nr 's/.*\<Change: *([0-9]{4,}-[01][0-9]-[0-3][0-9]) *(
        echo "$device:$inode:$mode:$nlink:$uid:$gid:$devid:$size:$blksize:$blkcnt:$atime:$m
        return
}
```

Tests

Soft delete and hard delete on the archive resource

- Create some test data:

```
echo "testfile" > test.txt
```

- Upload to iRODS employing the archive resource

```
iput test.txt -R archive

ils TEST
/aliceZone/home/rods:
test.txt
```

Check on the linux server that the data is there:

```
ls /home/<USER>/iRODS-archive/home/rods
test.txt
```

- Soft delete of the test file in iRODS:

```
irm test.txt
ils
```

Now iRODS tells us that the file is gone.

We can check on the remote linux server, the file has been moved to a folder called *trash*:

```
ls /home/<USER>/iRODS-archive/trash/home/rods
test.txt
```

and see that the file is still there.

- Hard delete of the test file If we now do a

  ```
  irmtrash
  ```

  and on the remote linux server a

  ```
  ls /home/<USER>/iRODS-archive/trash/home/rods
  ```

  we see that file was really deleted.

Fine-tuning

We have seen that the put command only finishes when the data is successfully transferred to the archive resource. For large files this might take some time. So we want the iRODS system to switch back to theprompt as soon as the data is stored in the cache resource and later do the transfer to archive.

To do so you need to switch the auto-replication off in the composable resource by setting:

```
iadmin modresc eudat context "auto_repl=off"
```

Now the put-command will finish as sson as the data is written to the cache resource.

To move the data automatically to the archive resource we need to define the *acPostProcForPut*, *acPostProcForRepl* and *acPostProcForCopy* in the iRODS rule set, e.g. */etc/irods/core.re*.

For an example implementation we refer to:

https://github.com/cookie33/irods-compound-resource