RFA — Remote File Access

A tool for managing access to files
that are distributed at multiple sites

Oliver Wenzel
GMD – Fokus
Berlin – Germany

Version 1.0

1 Introduction

This software tool called **Remote File Access** (RFA) has been designed to allow access to a set of files in a UNIX™ filesystem that are shared between multiple sites. It has been created within the VERDI 3 project as a result of our own requirements for a distributed document and file management system. Basis for this requirements is the VERDI 3 project which members are located at two different sites. Both sites of the project share a common set of documents and files which need to be accessed and modified during the everyday-project-work. For some time we used the FTAM software to exchange files between the two sites, but after the number of shared files and transfer operations grew, very often questions like

- at which site the up-to-date (master) version of the file is located ?
- is the local version up-to-date ?
- is anyone already editing this file ?
- are there any newly created files ?

appeared. Using a distributed filesystem like NFS was discussed as an alternative solution, but this did not seemed to be practicable because of the very slow communication links between the sites (9600Bps). Therefore, we tried to build a simple tool to handle theses questions. So the RFA tool was born and because the project is working with ISO protocols, we built the tool on top of the ISODE software package using the remote operation compiler *rosy(1)*. Currently the tools only supports to share files between two sites (because this was our primary goal), but it should be possible to extend it to support multiple sites.

2 The Model

2.1 The Data Model

The Data Model is based on the hierarchical structure of the UNIX™ filesystem. The RFA tools allows to share a subtree of the hierarchical filesystem structure between (currently) two sites. The structure of the shared subtree must be identical at both sites. The local root of the shared subtree may be different at both sites. Subject of the operation of the RFA tool are the files and directories within the shared subtree.

2.1.1 File Version-Time

A file within a shared subtree exists at the different sites as different (local) versions of the file. The version-time (state of modification) of a file version at a site is determined by the *modification*

time provided by the UNIX™ system call *stat(2)*. To determine which version of file is up-to-date, the youngest version-time is chosen. This requires that the sites participating in the file sharing use synchronized clocks. This is necessary because the modification time is set automatically by the UNIX system using the local operating system clock. The RFA tool provides an operation to synchronize the clocks where one of the sites is declared to be the *time-master* which provides the time to the *time-slave*.

2.1.2 RFA File Status

Each local version of a file that exists within the shared subtree at a site has a RFA-specific status which is one of:

MASTER the file is the up-to-date version of the file and modifications to the local version of the file are allowed.

SLAVE the file is the read-only version of the file and may be up-to-date but it need not. For each local file with SLAVE status there must be a MASTER version at the remote site. File version with SLAVE status will have file write permissions cleared.

UNREGISTERED the file not shared and has only local significance. It may be modified locally.

This file status allows to determine which site has the up-to-date version of a file and is allowed to do modifications to the file. The site holding a SLAVE version of the file has to request the mastership before it is allowed to do any modifications. Further, it has to be assured that the local version of the file at the new MASTER site is up-to-date. Otherwise the file has to be transferred first. A local version of a file may change its status during its lifetime (e.g. from UNREGISTERED to MASTER) as a result of several RFA operations described later.

2.1.3 File Locking

To avoid that a file is modified by multiple users at the MASTER site or that the mastership is transferred to the remote site while the local version is under modify-access by a local user, a local version of a file can be (soft)-locked. Locking of files is only applicable to file versions with status MASTER or UNREGISTERED.

Setting the lock flag of a directory has the effect, that the *.rfaexec* file is not executed during a RFA *sync* or *rsync* command. This can be used to enable the execution of the *.rfaexec* file for a period of time (e.g. while doing incomplete modifications to the source-code when *.rfaexec* contains commands for re-compilation).

2.1.4 Transferlevel

One mode of operation of the RFA tool is to synchronize sets of local version of files with the remote site. To determine if a file that is mastered remote should be transferred during this synchronization, a transferlevel can be associated with each file:

REQUEST The file is not transferred automatically during synchronization. If a new MASTER file is detected at the remote site, only a local dummy (empty file) is created to indicate the existence of the file.

AUTOMATIC The file is transferred automatically during synchronization if the local version is not up-to-date. If a new MASTER file is detected at the remote site, it is transferred.

2.2 The Functional Model

The RFA tools is built using the *Client-Server* model of the ISODE remote operations environment. There is a RFA server (called *ros.rfa)* at each site which is started dynamically by the ISODE *tsapd*. Most of the RFA commands are interpreted by the RFA client (called *rfa*) which issues a remote operation to the remote RFA server. The RFA client does not perform a 1–to-1 mapping of its commands onto the remote operations. It uses none, one or more remote operations to execute a RFA command. The association establishment is done when the first remote operation is invoked. The RFA client can operate in an interactive mode or a command mode.

One of the basic principles of RFA is that modifications to any files are only performed locally by the RFA client. The remote RFA server never modifies any file, so no remote file modifications are possible. The ensure a certain autonomy of the two sites concerning modifications of their local files.

3 The RFA Client

The RFA client (*rfa*) provides an interactive mode and a command mode. The rfa command has the following syntax:

```
rfa [ -u username -p password ] [ -h hostname] [ -q ]
     [ -c rfa-command ]
```

The name of the host (as defined in the *isoentities* database) where the remote site is located can be specified using the *-h* option. Because the RFA client binds to the remote RFA server with a username that must be valid at the remote site, the *-u* and *-p* options can be used to specify the username and the password for the remote site. The *-q* option directs *rfa* to run in quit-mode, where only severe errors are reported and no user interaction is performed. Using the *-c* option, a command can be passed to rfa in command mode. This can be any command described below.

If the rfa is called without the *-c* option, is runs in interactive mode. The *rfa* provides the following commands. A command can be abbreviated with its unambiguous prefix.

3.1 pwd

During interactive mode, *rfa* has a current directory. This directory is relative to the root of the shared subtree. The *pwd* command shows the name of the current working directory relative to the root of the shared subtree. Initially this is the current working directory of the UNIX *sh(1)* where *rfa* has been called if it is within the shared subtree.

3.2 cd

The *cd* command changes the current directory of *rfa* in interactive mode to the directory given as the first argument. No check is made if this directory exists. It is not possible to change the working directory above the root of the shared subtree.

3.3 list

The *list* command provides a listing of the files within the directories given as the command arguments. If no argument is given, the current working directory is listed. The output of the *list* command is similar to the output of the UNIX *ls -lg* command, e.g.

```
-rw-rw-r-- M-A ow       verdi          754 Jul 12 13:59  .sunview
drwxr-xr-x --- ow       verdi          512 Jul 12 14:01  bin
```

The three character field after the file permissions is used to indicated the different rfa-specific states of the file. The first character indicates the status of the file, with `M` for master version, `S` for slave Version and – for unregistered version. The second character is the lock indicator with an `L` indicating that the file is currently locked and an – for unlocked. The third character indicates the transfer mode of the files with `A` for automatic transfer and – for transfer on request.

3.4 rlist

The *rlist* command is similar to the *list* command with the difference that the files in the directory at the remote site are listed.

3.5 lock

The *lock* command is used to get a locked local version of the file(s) that are provided as argument(s) to the command. Only regular files within the shared subtree are valid. It is not possible to lock directories, links, etc. . If the name of the file to be locked is a symbolic link, the file referenced by the link is locked if it is located within the shared subtree.

The *lock* command should be used whenever a file within the shared subtree is modified to ensure that the local version of the file is a) the master version and b) not currently under modify-access by another local user.

If the local version of the files has a SLAVE status, the lock command will try to get the mastership for the file. Therefore, it first verifies that the local version is up-to-date, otherwise it transfers the up-to-date version from the current master, similar to the *getfile* command. After an up-to-date local version of the requested file exists, a *RequestMaster* operation is issued to the remote site to transfer the mastership. If the file is not locked at the remote site, the mastership is granted to the local site which then is the new master. The local lock is set the by setting the `L`(ocked) flag in the local files state.

If the local version of the requested file already is the MASTER version or is an UNREGISTERED file, it is checked if the file is already locked by another local user, where the requested lock is rejected. Otherwise the local lock is set the by setting the `L`(ocked) flag in the local files state.

Because the lock command will be applied in the most cases to a local version of the file that is already master, it is not necessary to execute the whole *rfa* program to do the local lock. Because the *rfa* program includes the whole code needed for the execution of the remote operations (ISODE stack, etc.), there may be a significant time delay while loading this huge program. To optimize this, a stand-alone program called *llock* is provided which is able to perform the locking if the local version is the master version. If *llock* could perform the local lock, it exits with exit–code 0. If the file is already locked locally or an error condition occurred it exits with exit-code 1. If the local version of the file is not MASTER, it returns 2 which indicates that the *rfa* command is needed to perform the locking.

3.6 unlock

The *unlock* command is used to release a lock for a local MASTER version of the file(s) provided as command argument(s). If the name of the file to be unlocked is a symbolic link, the file referenced by the link is unlocked if it is located within the shared subtree. There is are no checks made if the file has been locked by the user that performs the unlock operation.

Similar to the *llock* command there exists an *lunlock* command which is capable to perform the release of the (always) local locks to avoid the expensive use of the *rfa* command.

3.7 master

The *master* command can be used to request that the local version of the file(s) provided as argument(s) will become the MASTER version. If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

If the local version of the file is in UNREGISTERED status, its status is set to MASTER if there is no version of the file with MASTER status at the remote site. This is the way how locally created files will be introduced into the set of shared files where the creating site becomes the initial master.

If the local version of the file is in SLAVE status, then the mastership for the file is requested from the remote site similar to the *lock* command. The write access permissions of the new local MASTER version are set.

3.8 slave

The *slave* command can be used to request that the local version of the file(s) provided as argument(s) will become a SLAVE version. If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

If the local version of the file is in UNREGISTERED or MASTER status, its status is set to SLAVE if there is version of the file with MASTER status at the remote site. The write access permissions of the new local SLAVE version are cleared.

3.9 unregister

The *unregister* command can be used to request that the local version of the file(s) provided as argument(s) will become an UNREGISTERED version. If the local version of the file is locked, this results in an error. If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

3.10 setrequest

The *setrequest* command is used to set the transfer mode of the local version of the file(s) provided as argument(s) to REQUEST mode. If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

3.11 setauto

The *setauto* command is used to set the transfer mode of the local version of the file(s) provided as argument(s) to AUTOMATIC mode. If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

3.12 getfile

The *getfile* command is used to request the transfer of the shared file(s) provided as argument(s). If the name of the file is a symbolic link, the file referenced by the link is target of the operation if it is located within the shared subtree.

If the remote version of the file is in MASTER status, a local SLAVE version is created if the file does locally not exist. If a local SLAVE version already exists, it is updated from the remote MASTER version.

If the remote version of the file is in UNREGISTERED status, a local version of the file with status UNREGISTERED is created if the file does locally not exist. If a local version already exists, it is updated from the remote version.

The transfer of the file is performed according the version-time (time of last modification) of the local and remote files. If both the version-times of both files are equal, no transfer has to be performed because the SLAVE version is already up-to-date. If the version-time of the remote MASTER is older that the version-time of the local SLAVE (or UNREGISTERED) version, an inconsistency has been detected and an error is reported. Otherwise, the content of the file has to be transmitted to update the local SLAVE (or UNREGISTERED) version of the file.

The content of the file can be transferred either in uncompressed or in compressed mode, depending on the size of the file. The size limit when compression shall occur can be configured as described in chapter 5. The compression and decompression is done using the UNIX™ *compress(1)* and *uncompress(1)* commands.

Optionally, the old version of the local file will be saved in as a file with the suffix *.bak* if the local file is updated. This option can be configured as described in chapter 5.

3.13 syncdir

The *syncdir* command is used to synchronize the files within the directory/ies provided as argument(s) with their counterparts at the remote site. Here the basic principle is, that the site executing the *syncdir* command is only synchronizing the local site by applying the following actions to the list of files that exists at the **remote** site within the given directory:

- Create a local SLAVE file versions for newly found remote MASTER version file. This is done according to the transfer mode of the file. If the transfer mode is set to REQUEST, only a dummy SLAVE version of file is created which indicates the existence of the file, but does not have its actual content. Otherwise, the content of the file is transferred as described with the *getfile* command.
- Update local SLAVE file version with a remote MASTER version if the transfer mode of the file is AUTOMATIC
- Create local SLAVE sub-directory if a new MASTER sub-directory has been found at the remote site. By this, the shared subtree will grow in size. Directories found at the remote site that are in status UNREGISTERED are **NOT** created locally.
- Create a local symbolic link if a symbolic link has been found at the remote site. This is only done if the file referenced by the link is located within the shared subtree.
- Perform various checks to insure consistency between local and remote sites views of the files within the directory.
- Removes local SLAVE files if there is no corresponding MASTER file at the remote site. If there is a UNREGISTERED file at remote, the local file state is set to UNREGISTERED.
- Execute a existing *.rfaexec* file if any file within the directory has been updated and the directory itself in not LOCKED.

To achieve a fully synchronized directory, both sites need to perform a *syncdir* operation for that directory at periodic times. This will be typically done by creating an entry into the systems *crontab*.

3.14 rsyncdir

The *rsyncdir* command is very similar to the *syncdir* command, but performs a recursive synchronization of the whole subtree starting at the directory/ies provided as argument(s). The same actions are performed as within the *syncdir* command with an additional actions:

- If a MASTER directory is detected at the remote site and the transfer mode of this directory is set to AUTOMATIC, the directory is synchronized recursively.
- It does not follow symbolic links pointing to directories

Directories found at the remote site that are in status UNREGISTERED or don't have transfer mode AUTOMATIC are **NOT** synchronized recursively. They have to be updated individually by applying the *syncdir* command.

Using the *rsyncdir* command it is possible to synchronize the whole shared subtree. The configuration which parts of a subtree at a site belongs to that shared subtree is done by setting the MASTER status and the transfer level of the files and directories appropriately.

3.15 timesync

To synchronize the local clocks of the participating sites, the *timesync* command can be used. This command synchronizes the local clock of the site acting as the *time slave* with the clock of the *time master* site. The command can be called at both sites to initiate the synchronization. The configuration of a site to be *time slave* or *time master* is described in chapter 5

3.16 help

The *help* command shows the list of available command and gives a one-line description of each command.

3.17 quit

Obviously, the *quit* commands terminates the *rfa* session.

4 The RFA-EXEC facility

When a directory within the shared subtree is synchronized using the RFA *sync* or *rsync* commands, in most cases there are further actions required if any files have been updated, e.g. issuing a *make* command to re-build the updated software. This can be achieved by RFA's facility to create a file called *.rfaexec* in each directory. This file is executed by *sh(1)* every time the directory is synchronized and minimally one file within the directory has been updated. The file *.rfaexec* can be have executable format (sh, csh, binary) and should have the executed permission set. When RFA executed this file, it passes the names of the files that have been updated as arguments to it.

This facility can be disabled by setting the *rfaexec* tailoring option to *off*. Further, it can be disabled for a single directory by setting the LOCKED status of the directory. This is useful to prevent re-compilations of software during an unfinished development cycle.

For security reasons, the *.rfaexec* file can **NOT** be registered as MASTER or SLAVE file. An automatic transfer of this file may cause a user to execute something during the *sync* command which has been "imported" from the remote site. Therefore, the *.rfaexec* file is always a local unregistered file.

5 RFA Configuration

The configuration of the RFA client (*rfa*) and the RFA server (*ros.rfa)* server can be tailored at start time by a configuration file. This file is called *rfatailor* and can exist either in the ISODE's $(ETCDIR) or the RFA specific tailoring directory which is defined in the *config.h* file of RFA as *RFA_TAILDIR*. This file is read by both the RFA client and server. The file consists of a number of tailoring statements with the syntax

```
tailoringOption     =    tailoringValue
```

Lines with comments start with a "#".

The options that are valid for both the RFA client and server are:

root — the pathname of the root of the shared subtree at the local site

time — the role that the local site plays for time synchronization. Possible values are *master* or *slave*.

The following options are only recognized by the RFA server:

compress — The limit for the filesize in bytes when compression during the transfer operation shall occur.

chmod determines if the write permissions of local SLAVE files shall be removed and write permissions to local MASTER and UNREGISTERED versions shall be set. This requires that user executing the *rfa* command is the owner of the file or the *rfa* program runs under the effective user id of root. Possible values are *on* and *off*.

debuglog enables more detailed logging information to be written to the RFA logfile *rfa.log* in the ISODE logdir. Possible values are *on* and *off*.

The following options are only recognized by the RFA client. They may additionally be defined on a per-user basis in a file called *.rfarc* in the users home directory.

host the name of the remote host. This name must be defined in the ISODE's *isoentities* database.

user the username that shall be used to authenticate with the remote RFA server. This must be a UNIX^{TM} login name that is valid on the remote site.

password The password corresponding to the user name. This password must be the password for the given login name at the remote site.

backup determines if a backup file with suffix *.bak* shall be created when a local file is updated. Possible values are *on* and *off*.

chgrp determines if the group-id of local SLAVE files shall be set according to the group-id of the remote MASTER file. This requires that user executing the *rfa* command is member of all groups that may appear or the *rfa* program runs under the effective user id of root. Possible values are *on* and *off*.

chown determines if the user-id of local SLAVE files shall be set according to the user-id of the remote MASTER file. This requires that user executing the *rfa* command is the owner of the file or the *rfa* program runs under the effective user id of root. Possible values are *on* and *off*.

transfer the default transfer mode that is set when a file is registered the first time. Possible values are *auto* and *request*

clearsuid determines if the set-uid-on-execution bit of files is cleared if they are transferred to the remote site. Possible values are *on* and *off*. If RFA runs with effective user-id of root and *chown* options is enabled, *clearsuid* should be enabled to avoid that privileged users from one sites are able to gain illegal privileges at the other site.

rmslaves determines if slave files with no remote master file are deleted during the *sync* and *rsync* command. Possible values are *on* and *off*. If a there is a remote UNREGISTERED file, the status of the local SLAVE file is only changed to UNREGISTERED.

rfaexec determines if the *.rfaexec* files are executed during a *sync* or *rsync* command. Possible values are *on* and *off*.

6 Some Words on User-IDs and Permissions

In principle, the RFA client runs with the user-id of the user that invokes the client. The RFA server either runs with the user-id of the user who owns the server program (as performed by *tsapd(8)*), or it runs with the user-id the invoker has supplied with the bind arguments (a username and password valid at the remote server host). In the later case *tsapd* needs to run with user-id of root and ros.rfa needs to be owned by root.

Under certain circumstances, RFA client and server need to do modification to files which are not owned by the user-id that RFA is running with, such as

- changing a files access right (e.g. if it becomes a slave version)
- changing a files owner of group (e.g. if a local slave is updates according to a remote master and the *chown* and *chmod* options are enabled

One way to deal with these problems is to disable these facilities of RFA by setting the *chown*, *chgrp* and *chmod* options to *off*. This will be suitable in an environment where all files within the shared subtree belong to a single user and only this user runs RFA client and server processes.

Otherwise, if different users access the files, minimally the *chmod* option should be enabled to avoid modifications to slave files (remember that locking in rfa is advisory) by making slave files read-only. Further, the *chown* and *chgrp* options should be enabled if the users and groups are the same at both sites to maintain the ownership (and originatorship of modification) across the two sites.

This can be achieved by running the RFA client and server with the effective user-id of root (by setting the set-uid-on-execution flag). The RFA client will set his effective user-id during start-up to his real user-id and switches back to the effective user-id of root only for these sections of code where the root permissions are really required to change the access mode, owner or group of files. Otherwise, the RFA client run with the effective user-id equal to his real user-id.

The RFA server changes his effective user-id during start-up to the user-id provided by the calling client. Like the RFA client, it switches back to the effective user-id of root only for these sections of code where the root permissions are really required to change the access mode of a file.

If the RFA client and server do not run with effective user-id of root, the time synchronization command of RFA can only be used, if the external program *rfatime* of the RFA package is installed to run with set-uid root

7 What is still missing ...

The current version of RFA is a first shot which needs some completion and extensions.

- A command like “remote-diff” could be helpful for resolving inconsistencies.
- The time synchronization only works if the local time of a *time-slave* site can be modified, which may not hold true in each environment.
- There is a problem with the *chown*, *chgrp* and *chmod* facilities of RFA because the enabling of them may cause “permission denied” errors if the RFA does not run with the effective used id of root.
- RFA is only capable to support distribution over two sites, which should be extended to allow a multiple site configuration. This may require the redesign of the RFA model and will introduce a new level of complexity.