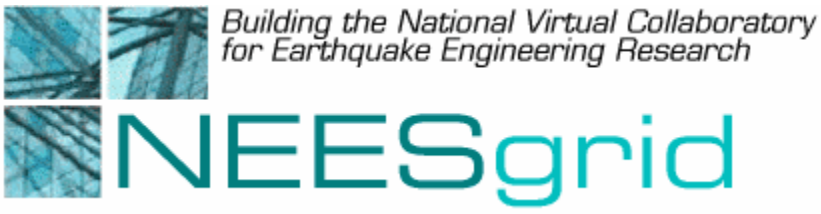

Technical Report NEESgrid-2003-18

www.neesgrid.org

Draft Whitepaper Version: 1.0

Last modified: October 4, 2003

NFMS User Guide

(DRAFT)

Jeff Gaynor, Joe Futrelle[1]

Feedback on this document should be directed to jjg@jqhome.net

[1] National Center for Supercomputing Applications, Champaign, IL 61821

Acknowledgment: This work was supported primarily by the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) Program of the National Science Foundation under Award Number CMS-0117853.

1 Summary

This user guide describes the 2.0 release of NEESgrid File Management Service (NFMS). NFMS provides clients with the ability to locate files independently of how and where they are stored, as well as the ability to negotiate transactions with storage systems. This manual is geared toward users wishing to write an NFMS application and shows how to implement the class NFMSFacade to carry out important tasks, thereby reducing the amount of manual coding done by the user. A guide to performing these tasks manually is also included, along with examples demonstrating both processes and a glossary of terms.

2 Introduction to the NFMS

This document describes the NEESgrid File Management Service (NFMS). NFMS provides clients with the ability to locate files independently of how and where they are stored, as well as the ability to negotiate transactions with storage systems. NFMS is designed to accommodate a variety of storage architectures and data transfer protocols, including GridFTP. Using NFMS, a client can reliably and securely store and retrieve data from a NEESgrid repository without having to keep track of the physical location of the file or use any APIs specific to any particular storage architecture. Because of this, storage architectures can be migrated without having to rewrite NFMS-based applications. NFMS is a connectionless protocol, meaning that an NFMS server can resume a transaction even if it was restarted during the transaction. It is integrated with the NEESgrid Metadata Service, so that metadata about files and the data in them can be managed along with the files.

This document describes the 2.0 release of NFMS, which is a WSDL web service. In later releases, NFMS will be made available as a Grid service using the Open Grid Services Architecture.

2.1 Prerequisites for writing an NFMS application

You should have a functional NFMS installation. NFMS is installed on every NEES-Pop as a standard part of the NEESgrid release. You also must have at least version 1.4.1 of Java in order to get certain security classes needed for the Globus Java CoG. You will also need a version of the CoG installed compatible with the Globus toolkit installed on the NFMS server.

Finally, the Java programming skill required to develop NFMS applications is minimal. You don't need more that a basic idea of how to write classes in Java to use NFMS. The entire API (as outlined in the NFMS Reference) is small. Putting files on the server or retrieving them is a straightforward matter.

3 The easy way: using NFMSFacade

3.1 Introduction.

There is a helper class, **`org.nees.nfms.NFMSFacade`**, which allows an application to do many common tasks with a minimum of coding. Really about all that is needed is a valid credential and the URL to the service.

This class will carry out most all the details needed: It will connect to the service, pass along the correct credentials, initiate the transfer and if so requested, will continue to retry an operation until it succeeds.

We assume that there is an instance of `NFMSFacade` in the variable `_NFMSFacade` in what follows. Also, we omit the obligatory **try ... catch** block that should surround these examples.

3.2 Creating an NFMSFacade instance

This is quite straightforward, as long as the web-address or URL of NFMS service is known. Typical code would be

```
URL nfmsURL = new URL("http://neespop.myuni.edu:8080/axis/services/nfms");
// use the correct address for your installation.
NFMSFacade _NFMSFacade = new NFMSFacade(nfmsURL);
// Ready for action...
```

3.3 Getting the right credentials

Valid credentials must exist as an instance of `org.ietf.jgss.GSSCredential` in order to use `NFMSFacade`. This means that clients have to use version 3.0 or higher of the Globus Toolkit. The normal mode of operation is to try and load the credentials from the default location specified during installation of Globus. All of the methods in `NFMSFacade` that do not specifically require passing the credential do this. Clients may also specify the credential to be used in all operations and this becomes the default for subsequent operations in `NFMSFacade` (but no place else, it should be emphasized). To query the current credential, issue

```
GSSCredential cred = _NFMSFacade.getCredential();
```

If a different credential is to be used for all subsequent operations in, say, the variable `cred`, issue

```
_NFMSFacade.setCredential(cred);
```

to reset the credential. Again, this does not effect Globus' installation, it merely sets the credential for use in `NFMSFacade.`

3.4 Setting the transfer mode.

Since `NFMSFacade` uses GridFTP to do the transfers, developers should be aware of the fact that, just like regular ftp, there are two ways or modes in which files may be transferred, ***binary*** (sometimes called ***image***) or ***ascii*** (also known as ***text)***. In binary mode, the entire file is treated as if it is binary, *i.e.*, that it is not human-readable and must be faithfully transferred unchanged in the slightest detail. This is in opposition to ascii mode. Ascii (from American Standard Code for Information Interchange) is human readable. In particular, different types of computers represent the end of line markers (plus a few other markers) in different ways. In this mode, whenever one of these markers is found, it is converted accordingly for the computer system. This is fine in many instances, however, in cases where the file is actually computer-readable,

the effect is simply catastrophic and not really readily reversible. Therefore, it is important that the correct mode be chosen for the operation.

The default is binary and that is generally the safest. To change the mode, there are two methods. For examples, to set the mode to binary, an application would invoke

```
_NFMSFacade.setBinaryTransferType();
```

while to put the transfers into ascii mode, a client invokes

```
_NFMSFacade.setASCIITransferType();
```

Notice that neither of these methods requires an argument.

3.5 Retrying operations

There are two modes of operation, ***fail-fast*** and ***retry***. As the name implies, fail-fast mode will try any operation once and promptly exit if there is any problem. This is the default and it is a wise one, especially while developing a client. In retry mode, `NFMSFacade` will continue to redo a failed operation at a given time interval until it succeeds. This is useful in cases where there can be well-understood problems with the operation (*e.g.* certain network operations timing-out, intermittent server connections, etc., etc.)

Here is a typical example for setting the mode to fail-fast.

```
_NFMSFacade.setFaultMode(_NFMSFacade.FAIL_FAST_MODE);
// continue with other operations...
```

Similarly, the call for setting the mode to retry is just

```
_NFMSFacade.setFaultMode(_NFMSFacade.RETRY_MODE);
```

3.6 Uploading

3.6.1 Appending

The various append methods are for the case in which a file possibly already exists in the service and an application needs to add to the tail of it (appending to the beginning of a file is not supported currently). The arguments lists are differentiated chiefly by how much one needs or wants to customize the operation. At its most basic, here is the call to send a file to the service, creating it if it does not exist and appending it to any current version of the file if it does exist. In this case the logical name will be `my file` and the local file is in `c:\documents\junque.txt.`

```
String logicalName = "my file";
java.io.File f = new java.io.File("c:\\documents\\junque.txt");
_NFMSFacade.append(logicalName, f);
```

This loads the default credential and will retry the operation exactly once. If the application wishes to supply the credential, there is a method for that as well.

What if the application wants to be sure that the append fails if the logical name is already in use? This is needed if there is a chance some other application will upload a file with the same logical name. Appending to such a file would render the result useless. (Remember, that the model for NFMS is a file system and it is quite possible to have two users append unrelated files

to the same location with similarly unwanted results). In this case, the correct call would simply be

```
1_NFMSFacade.appendNew(logicalName, f);
```

There is as well an overloaded method that permits passing the credential, if needed.

Finally, what if the client wishes to version the file? In this case, there is (possibly) a file already in NFMS and the application would like to append something to it, so that the current version is retained as the old version, and the new, appended file becomes the new current version (with the version number appropriately incremented). The correct method then is just

```
_NFMSFacade.appendNewVersion(logicalName, f, cred);
```

where **cred** contains a credential. At this writing, the client must pass a credential, but to have it use the default, just use

```
_NFMSFacade.appendNewVersion(logicalName, f, _NFMSFacade.getCredential());
```

3.6.2 New files

In the case that an application needs to send a file to the service, the simplest call is, assuming that `logicalName` and `f` are as in the previous section

```
_NFMSFacade.put(logicalName, f);
```

This will upload the file, **f**, giving it the `logicalName` stated. If there is a file already in the service with the given `logicalName`, this will replace it silently, *i.e.*, without notice. Again, this is much like any other file system, where copying a file will overwrite a previous, like-named file. There are several overloaded methods to allow for customization. The developer should, however, be aware that all of these eventually just call the method whose signature is

```
_NFMSFacade.put(String, File, UploadOptions, GSSCredentials);
```

If an application needs to ensure that no existing file is overwritten, the appropriate method is

```
_NFMSFacade.putNew(logicalName, f);
```

Finally, if an application needs to replace a file completely, but wishes to retain the old file as a version, then

```
_NFMSFacade.putNewVersion(logicalName, f, cred);
```

where **cred** refers to a credential, is required. The application can use the default credential by invoking

```
_NFMSFacade.putNewVersion(logicalName, f, _NFMSFacade.getCredential());
```

3.6.3 Comment on versioning

Versions of files may be created, but there is currently no easy to recover a previous version of a versioned file. All download methods will retrieve the current version. This will be rectified in a future release of NFMS.

3.7 Downloading

To download a file, an application need to use one of the **get** methods. As with the other groups of methods, these all eventually pass off the call to a single method, whose signature in this case is

```
get(String, File, DownloadOptions, GSSCredential);
```

and the various versions just supply defaults as needed. The simplest usage would be as follows, where a client needs to retrieve the file named **My experimental data** from the service and place into a local file named `c:\documents\my-data.txt`. Such a call (minus the `try ... catch` block) would look like this

```
String logicalName = "My experimental data";
File f = new File("c:\\documents\\my-data.txt");
_NFMSFacade.get(logicalName, f);
```

3.8 Getting and updating information about a file

NFMS keeps track of certain fixed metadata about a file, such as the size in bytes, the date it was created and other items. These are stored in an object of type `org.nees.nfms.gen.FileInfo`. We call these ***file information objects***. There is another type of object that can be stored as well and while this is not managed by NFMS, it is important in its own right for the repository and data browsers. A ***file reference object*** is just a reference to the logical name. Therefore these are objects of type **org.nees.md.Reference**. There are methods in **NFMSFacade** for working with these, but *not* in NFMS, since these are application-centered (as opposed to file-centered).

`NFMSFacade` allows a client to get the file information object. Only logical name is required:

```
String logicalName = "whatever you called it...";
FileInfo = _NFMSFacade.getFileInfo(logiclaName);
// now you can access information about the file.
```

Note that there is no way in this release to update the title through `NFMSFacade`, so a client must talk to NFMS directly for this.

File reference objects can be created them as follows (note that since these are just references and the file must be entered into the repository by uploading it: `NFMSFacade` is perfectly happy to create a reference for something that does not exist.)

```
String logicalName="17.08.2003 16:42:23 shake table 4 results";
String title = "latest results";
File f = new File("/home/foo/temp/sh_tbl4.xsl");
_NFMSFacade.put(logicalName, f);
// now create something to show the users.
Reference ref = _NFMSFacade.createFileObject(logicalName, title);
// code to display this.
```

To look up the file object, issue

```
Reference ref2 = _NFMSFacade.getFileObject(logicalName);
// whatever....
```

So in the above case the application uploads a file then creates a file reference object for it. This is a standard sequence of operations. The title may be arbitrary. This is a different title than the one in the **FileInfo**.

3.9 A complete example.

We'd like to finish this section off with a complete example using `NFMSFacade`. This will upload a file then download it again, putting it into a new given file and printing out a few details from its file information.

```
import org.nees.nfms.gen.*;
import org.nees.nfms.NFMSFacade;
import java.io.File;
import java.net.URL;

/**
  * A really simple example of using NFMSFacade to upload a file, then
download it again.
  */
public class NFMSFacadeTest{
   public static void main(String[] args){
      if(args.length != 4){
         usage();
         return;
      }
      try{
          // set up the arguments.
          URL nfmsURL = new URL(args[0]);
          File uploadf = new File(args[1]);
          if(!uploadf.exists()){
             System.out.println("Error! The file you specified to"+
                    " upload does not exist.");
             usage();
          } //end-if to check upload file
          String logicalName = args[2];
          File downloadf = new File(args[3]);
          if(downloadf.exists()){
             System.out.println("Error! The target file you specified" +
                    " for download" +
                    " (and over-writing) exists.");
             usage();
          } //end-if to check download file
          // make the facade
          NFMSFacade _NFMSFacade = new NFMSFacade(nfmsURL);
         // upload it
         _NFMSFacade.putNew(logicalName, uploadf);
         // download it to someplace else
         _NFMSFacade.get(logicalName, downloadf);
         // get some information
         FileInfo fileInfo = _NFMSFacade.getFileInfo(logicalName);
         // print a short report about the file.
         System.out.println("Metadata about " + uploadf);
         System.out.println("  logicalName: " + fileInfo.getLogicalName());
         System.out.println("  title: " + fileInfo.getTitle());
         System.out.println("  creation date: " + fileInfo.getCreationDate());
         System.out.println( "  size: " + fileInfo.getSize());
     }catch(Exception oops){
        oops.printStackTrace();
        usage();
      } //end catch block
   } // end method main(String[])

   public static void usage(){
      System.out.println("Usage: This will upload a file from" +
             " uploadff to an NFMS server"+
             " using the logicalName.");
      System.out.println("  It will then download it again" +
```

```
            " to the file downloadf and print"+
            " some of the information about the file.");
      System.out.println("NFMSFacadeTest nfmsURL uploadf" +
            " logicalName downloadf);
   } //end method usage()
} //end NFMSFacadeTest class
```

4 Doing tasks directly

4.1 How is uploading supposed to work?

NFMS runs over a web service, so no active continuous connection exists. Every time a request comes to the server, its state is stored internally until the next request comes through. The *transfer token* (see definition in the glossary) is used to tell the server the transaction whose state is to be recovered. Here are the steps for uploading a file.

- The application creates an instance of NFMS
- The application creates an upload object, then the application sends a request using the `requestUpload` call. This includes the desired logical name for the file, as well as various options (most importantly, the protocol).
- The server responds with a `TransferReturnBean`, containing the transfer token and a URI to the staging area.
- The application sends the file to the given URI using whatever mechanism it has.
- The application notifies the service using the `uploadComplete` call. This includes the transfer token.
- The server copies the file from the staging area to the permanent storage location and removes the file from the staging area.

Details for each of these tasks follow, including code snippets. We assume the service reference in a variable called `_NFMS`.

4.2 How do I connect to the server?

This is done using a couple of helper classes. The complete procedure, which can almost be used as is, reads:

```
import org.nees.nfms.gen.*;
import java.net.URL;
FileManagementService _NFMS;
// Set your URL accordingly:
try{
  URL nfmsURL =
    new URL("http://neespop.ncsa.uiuc.edu:8080/services/axis/nfms");
  FileManagementServiceService service =
      new FileManagementServiceServiceLocator();
  _NFMS = service.getnfms(nfmsURL);
  // good to go...
}catch(Exception oops){
```

```
   oops.printStackTrace(); // or whatever
}
```

A client is ready to start making calls to NFMS. In what follows we will leave out the obligatory `try ... catch` blocks.

4.3 How do I request a transfer?

We will show how to request an upload, since requesting a download is all but identical. We assume that you have a connection to the server in the `_NFMS` variable.

```
FileManagementService _NFMS;
// set up a connection to NFMS.
UploadOptions uOptions = new UploadOptions();
uOptions.setProtocol(NFMSConstants. PROTOCOL_GRIDFTP); //set the protocol
uOptions.setNewFile(true); // tells server this is a new file
TransferReturnBean trb = _NFMS.requestUpload("my file", uOptions);
```

Now NFMS is awaiting the file.

4.4 How do I do a grid ftp transfer to NFMS?

If a client must do this directly, it should use the class `org.nees.util.grid.GridFTPFacade`. This presupposes that all the client needs to do is the actual transfer, rather than have the `NFMSFacade` class create all of the options and do it. The only caveat is that `GridFTPFacade` assumes that the application is passing it remote and local file names, rather than a URL and the client must therefore parse the URL into tokens to use this class. See the Javadoc for this class for details, since a full accounting of its usage would exceed the scope of this user guide.

4.5 How do I stop a transfer?

Invoke the `abortXXX` method for downloads or uploads. Example code to abort an upload.

```
  // ... setup is as above to request the upload
  TransferReturnBean trb = _NFMS.requestUpload("my file", uOptions);
  // oops. let's stop this
   _NFMS.abortUpload(trb.getTransferToken());
```

Subsequent attempts to contact the server using this transfer token will throw an exception.

4.6 Tell me about file names.

You are probably wondering why there are three, the title and the logical and physical names. The physical name is designed for server use only. These can actually be viewed by retrieving a `FileInfo` object, hence we will mention it here. The logical name is the handle a client can use to make file recovery easier. The title is needed for things like user interface displays. Here are some examples

Physical name: `file:///nees/nfms/repoHome/4798734567.38976`

Logical name: `c:/documents/experiments/data/st-1-090403.xsl`

Title: `Shake table 1 data for Sept. 4, 2003 (Excel)`

4.7 What protocols do you support?

In the current release, the only protocol supported is grid ftp. Others such as http and plain ftp are planned, but did not make it into this release.

4.8 What do you mean that I shouldn't version all the data from my experiment?

The question is really if a user needs versions of everything. NFMS will cheerily store all versions, but remember that while the total capacity is in theory unlimited, only more recently used files are in the active storage area, the rest being put in long term storage. It is therefore quite possible for a single user to version huge files and slow the system down. If a user *need*s (note operative word) to keep versions of everything, this is fine. All we are saying is that this should not be simply turned on without being aware of the possible ramifications. In particular, NFMS is not intended to be a backup system.

4.9 How do I disconnect from the server?

Since there is not a continuous connection, *i.e.*, every transaction has its state preserved on the server, then stored. If you are not actively making a request to the server, you are not connected. There is no cleanup or shutdown you need to do.

5 Glossary

download This is a transfer of (a) file(s) from the server.

logical name This is a client-given string to identify the file. In the future the application simply refers to this string and the server finds the file. This cannot be changed. (See *title*, below)

physical name This is a server-generated unique identifier for the file. Applications normally never interact with this. This cannot be changed.

staging area This is a location that is accessible via a given protocol to *both* the server and the application. Files are copied to or from this area as dictated by the protocol. *E.g.* for ftp transfers, the user's home directory is used. If the server is receiving the file, it is then copied to it permanent location.

title A user-given string. This is provided so that graphical application have a more easily human readable form. This can be changed at any time. Its initial default value is the same as the logical name.

transfer Sending files to the server or getting them from the server.

transfer token A unique string identifying a transaction.

upload This is a transfer of (a) file(s) to the server.

version A number, starting at 0, that tracks revisions. Subsequent versions have their number incremented automatically.

6 Final thoughts

Be sure to get hold of the reference manual if you are writing applications that invoke NFMS.

Also, there is at this point no mechanism for retrieving a specific version of a file. This will be addressed in a later revision of NFMS.