cosFileTransfer

cosFileTransfer 1.2.2
March 26, 2018

March 26, 2018

1 cosFileTransfer User's Guide

The cosFileTransfer Application is an Erlang implementation of the OMG CORBA FileTransfer Service.

1.1 The cosFileTransfer Application

1.1.1 Content Overview

The cosFileTransfer documentation is divided into three sections:

- PART ONE - The User's Guide
  Description of the cosFileTransfer Application including services and a small tutorial demonstrating the development of a simple service.
- PART TWO - Release Notes
  A concise history of cosFileTransfer.
- PART THREE - The Reference Manual
  A quick reference guide, including a brief description, to all the functions available in cosFileTransfer.

1.1.2 Brief description of the User's Guide

The User's Guide contains the following parts:

- cosFileTransfer overview
- cosFileTransfer installation
- A tutorial example

1.2 Introduction to cosFileTransfer

1.2.1 Overview

The cosFileTransfer application is a FileTransfer Service compliant with the **OMG** Service CosFileTransfer.

Purpose and Dependencies

If a Virtual File System is started as 'FTP', the inets-2.5.4 application, or later, must be installed.

cosFileTransfer is dependent on **Orber**, which provides CORBA functionality in an Erlang environment, and **cosProperty**.

Prerequisites

To fully understand the concepts presented in the documentation, it is recommended that the user is familiar with distributed programming, CORBA, the Orber and cosProperty applications.

Recommended reading includes **CORBA, Fundamentals and Programming - Jon Siegel** and **Open Telecom Platform Documentation Set**. It is also helpful to have read **Concurrent Programming in Erlang**.

1.3 Installing cosFileTransfer

1.3.1 Installation Process

This chapter describes how to install *cosFileTransferApp* in an Erlang Environment.

Preparation

Before starting the installation process for cosFileTransfer, the application `Orber` must be running and `cosProperty` installed by using `cosProperty:install()`. Please note that it is **NOT** necessary to use `cosProperty:install_db()` for running the cosFileTransfer application.

Configuration

When starting the cosFileTransfer application the following configuration parameters can be used:

- **buffert_size** - default is 64000. This option determine how many bytes will be read at a time when transferring files.

1.4 Using the File Transfer Service

1.4.1 Overview

This chapter describes how two File Transfer Service applications interact.

Components

There are several ways the OMG File Transfer Service can be used. Below one scenario is visualized:

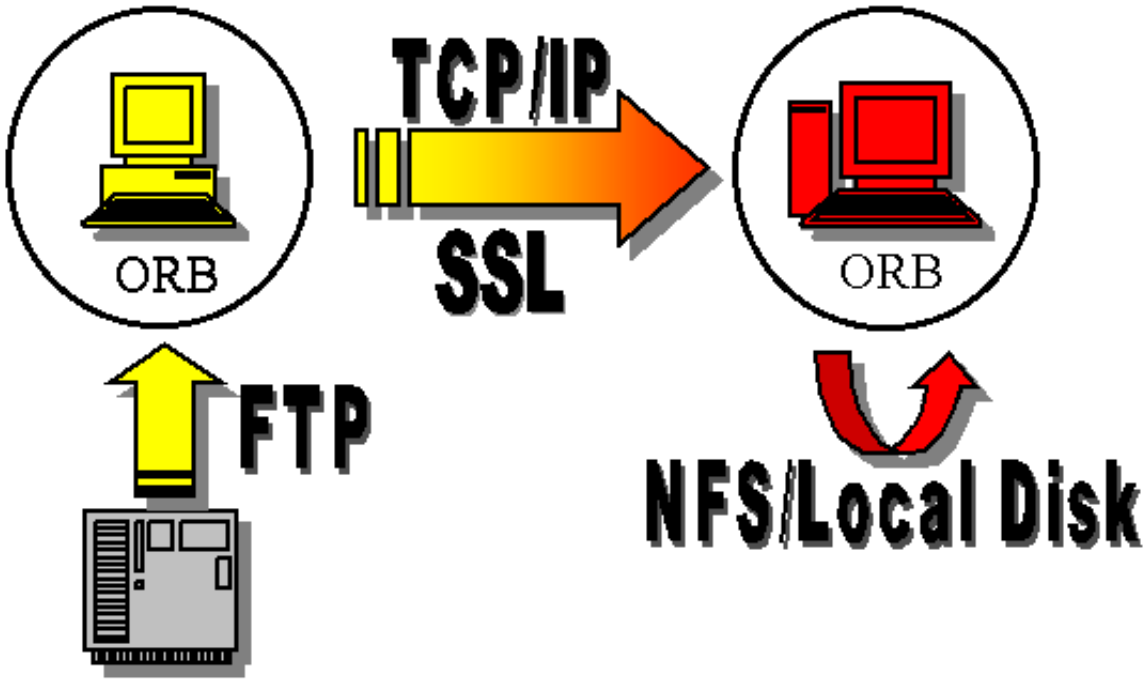

Figure 4.1: Figure 1: The File Transfer Service Components.

- **Source ORB:** this is the ORB we want to transfer a file from/via and it holds an object reference to a *Virtual File System (VFS)* which, in this example, represents an FTP server.
- **Target ORB:** the goal may be, for example, to transfer a new file or append to an existing file placed at the location that this ORB's VFS represents. In this scenario it is the local disk or the NFS.
- **Transport Protocol:** initially the ORB's, i.e., target and source, communicate via normal CORBA requests to determine whether or not they can communicate. If the File Transfer Service's have one, or more, `Transport Protocol` in common the data will be streamed using this protocol. The cosFileTransfer application currently supports TCP/IP and SSL.

Which type of file system the VFS is supposed to represent is determined by the options given when creating it, which is also how one determine which `Transport Protocol` to use. Hence, the source and target VFS described above can be started by invoking, respectively, the following operations:

```
1> SVFS = cosFileTransferApp:create_VFS('FTP', [], Host, 21, [{protocol, tcp}]),
2> TVFS = cosFileTransferApp:create_VFS({'NATIVE', 'cosFileTransferNATIVE_file'},
                                        [], OtherHost, 0, [{protocol, tcp}]),
```

Naturally can any combination of VFS-types be used and it is also possible to use own drivers, i.e., `{'NATIVE', 'MyDriver'}`.

After creating necessary `VFS's` we can login in and perform operations on files and directories residing on each file system.

How To Use SSL

To be able to use SSL as transport protocol a few configuration parameters must be set. The required parameters depend on if Orber is the target or/and the source ORB. However, the SSL_CERT_FILE variable must be defined in both cases.

Setting of a CA certificate file with an option does not work due to weaknesses in the SSLeay package. A work-around in the ssl application is to set the OS environment variable SSL_CERT_FILE before SSL is started. However, then the CA certificate file will be global for all connections (both incoming and outgoing calls).

Configurations when cosFileTransfer is Used as Target

The following three configuration variables can be used to configure cosFileTransfer's SSL target behavior.

- **ssl_server_certfile** which is a path to a file containing a chain of PEM encoded certificates for cosFileTransfer as target.
- **ssl_server_verify** which specifies type of verification: 0 = do not verify peer; 1 = verify peer, verify client once, 2 = verify peer, verify client once, fail if no peer certificate. The default value is 0.
- **ssl_server_depth** which specifies verification depth, i.e. how far in a chain of certificates the verification process shall proceed before the verification is considered successful. The default value is 1.

There also exist a number of API functions for accessing the values of these variables:

- cosFileTransferApp:ssl_server_certfile/0
- cosFileTransferApp:ssl_server_verify/0
- cosFileTransferApp:ssl_server_depth/0

Configurations when cosFileTransfer is used as Source

Below is the list of configuration variables used when cosFileTransfer act as the source application.

- **ssl_client_certfile** which is a path to a file containing a chain of PEM encoded certificates used in outgoing calls.
- **ssl_client_verify** which specifies type of verification: 0 = do not verify peer; 1 = verify peer, verify client once, 2 = verify peer, verify client once, fail if no peer certificate. The default value is 0.

- **ssl_client_depth** which specifies verification depth, i.e. how far in a chain of certificates the verification process shall proceed before the verification is considered successful. The default value is 1.

There also exist a number of API functions for accessing the values of these variables in the client processes:

- cosFileTransferApp:ssl_client_certfile/0
- cosFileTransferApp:ssl_client_verify/0
- cosFileTransferApp:ssl_client_depth/0

1.5 cosFileTransfer Examples

1.5.1 A tutorial on how to create a simple service

Initiate the application

To use the complete cosFileTransfer application cosProperty must be installed.

How to run everything

Below is a short transcript on how to run cosFileTransfer.

```
%% Start Mnesia and Orber
mnesia:delete_schema([node()]),
mnesia:create_schema([node()]),
orber:install([node()]),
mnesia:start(),
orber:start(),

%% The File Transfer Service depends on the cosProperty
%% application. Hence, we must install cosProperty first.
%% It's NOT necessary to invoke cosProperty:install_db().
cosProperty:install(),

%% Install File Transfer Service in the IFR.
cosFileTransfer:install(),

%% Now start the application and necessary services.
cosFileTransfer:start(),

%% Create two Virtual File Systems respectively representing an FTP-
%% and the local NFS file system.
VFSFTP = cosFileTransferApp:create_VFS('FTP', [], FTPHost, 21),
VFSNATIVE = cosFileTransferApp:
            create_VFS({'NATIVE', 'cosFileTransferNATIVE_file'},
                       [], MyLocalHost, 0),

%% Login to each system.
{FSFTP, DirFTP} = 'CosFileTransfer_VirtualFileSystem':
                  login(VFSFTP, "myId", "myPwd", "myAccount"),
{FSNATIVE, DirNATIVE} = 'CosFileTransfer_VirtualFileSystem':
                        login(VFSNATIVE, "myId", "myPwd", "myAccount"),

%% If we want to copy a file from the NFS to the FTP we must first
%% create a File object which contains its attributes.
Target = 'CosFileTransfer_FileTransferSession':create_file(FSFTP,
                          ["/", "ftp", "incoming", "targetFile"])),

 #'CosFileTransfer_FileWrapper'{the_file = Dir} =
%% Lookup the file we want to copy.
FileWrapper = 'CosFileTransfer_FileTransferSession':get_file(FSNATIVE,
                           ["/", "home", "myId", "sourceFile"]),
Source = FileWrapper#'CosFileTransfer_FileWrapper'.the_file,

%% Now we are ready to transfer the file. Please note that we most
%% call the source Session object.
'CosFileTransfer_FileTransferSession':transfer(FSNATIVE, Source, Target),
```

2 Reference Manual

The cosFileTransfer Application is an Erlang implementation of the OMG CORBA File Transfer Service.

cosFileTransferApp

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

This module contains the functions for starting and stopping the application.

Exports

`install() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation installs the cosFileTransfer application. Note, the `cosProperty` application must be installed prior to invoking this operation.

`uninstall() -> Return`

Types:

`Return = ok | {'EXIT', Reason}`

This operation uninstalls the cosFileTransfer application.

`start() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation starts the cosFileTransfer application.

`stop() -> Return`

Types:

`Return = ok | {error, Reason}`

This operation stops the cosFileTransfer application.

`create_VFS(Type, Content, Host, Port [,Options]) -> Return`

Types:

`Type = 'FTP' | {'NATIVE', 'cosFileTransferNATIVE_file'} | {'NATIVE', MyModule}`
`Content = []`
`Host = string(), e.g. "myHost@myServer" or "012.345.678.910"`
`Port = integer()`
`Options = [Option]`
`Option = {protocol, Protocol} | {connect_timeout, Seconds}`
`Protocol = tcp | ssl`
`Return = VFS | {'EXCEPTION, E}`
`VFS = #objref`

This operation creates a new instance of a Virtual File System. The `Type` parameter determines which type we want the VFS to represent. 'FTP' maps to the `INETS` ftp implementation, while {'NATIVE', 'cosFileTransferNATIVE_file'} uses the `file` module. It is also possible to implement own mappings which are activated by supplying {'NATIVE', MyModule}. The MyModule module must export the same functions and behave in the same way as the INETS ftp module, and an operation named `open(Host, Port)`, which shall return `{ok, Pid}` or `{error, Reason}`.

If no `Options` are supplied the default setting will be used, i.e., tcp and 60 seconds.

The `Content` parameter is currently ignored by must be supplied as an empty list.

`ssl_server_certfile() -> string()`

This function returns a path to a file containing a chain of PEM encoded certificates for the cosFileTransfer as target. This is configured by setting the application variable **ssl_server_certfile**.

`ssl_client_certfile() -> string()`

This function returns a path to a file containing a chain of PEM encoded certificates used in outgoing calls. The default value is configured by setting the application variable **ssl_client_certfile**.

`ssl_server_verify() -> 0 | 1 | 2`

This function returns the type of verification used by SSL during authentication of the other peer for incoming calls. It is configured by setting the application variable **ssl_server_verify**.

`ssl_client_verify() -> 0 | 1 | 2`

This function returns the type of verification used by SSL during authentication of the other peer for outgoing calls. The default value is configured by setting the application variable **ssl_client_verify**.

`ssl_server_depth() -> int()`

This function returns the SSL verification depth for incoming calls. It is configured by setting the application variable **ssl_server_depth**.

`ssl_client_depth() -> int()`

This function returns the SSL verification depth for outgoing calls. The default value is configured by setting the application variable **ssl_client_depth**.

CosFileTransfer_File

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

This module also exports the functions described in:

- CosPropertyService_PropertySetDef in the cosProperty application.

Exports

`'_get_name'(File) -> string()`
Types:
`File = #objref`

This read only attribute represents the target object's associated name.

`'_get_complete_file_name'(File) -> string()`
Types:
`File = #objref`

This read only attribute represents the target object's associated absolute name.

`'_get_parent'(File) -> Directory`
Types:
`File = Directory = #objref`

This read only attribute represents the target object's container. In some cases a `NIL` object will be returned.

`'_get_associated_session'(File) -> FileTransferSession`
Types:
`File = FileTransferSession = #objref`

This read only attribute represents the target object's associated `FileTransferSession`.

CosFileTransfer_Directory

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

This module also exports the functions described in:

- *CosFileTransfer_File*
- CosPropertyService_PropertySetDef in the cosProperty application.

Exports

`list(Directory, Max) -> Return`

Types:

`Directory = #objref`
`Return = {ok, FileList, FileIterator}`
`FileList = [File]`
`File = FileIterator = #objref`

This operation returns a list, of length `Max` or less, containing Object References representing files or directories contained within the target Directory. If the amount of objects found is less than `Max` the returned Iterator will be a `NIL` object.

CosFileTransfer_FileIterator

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

Exports

`next_one(Iterator) -> Return`

Types:

Iterator = #objref
Return = {boolean(), #'CosFileTransfer_FileWrapper'{the_file = File file_type = Type}}
File = #objref
Type = nfile | ndirectory

This operation returns true if a `FileWrapper` exists at the current position and the out parameter contains a valid `File` reference. If false is returned the out parameter is a non-valid `FileWrapper`.

`next_n(Iterator, Max) -> Return`

Types:

Iterator = #objref
Max = unsigned long()
Return = {boolean(), FileList}
FileList = [#'CosFileTransfer_FileWrapper'{the_file = File file_type = Type}]
File = #objref
Type = nfile | ndirectory

This operation returns true if the requested number of `FileWrappers` can be delivered and there are additional `FileWrappers`. If false is returned a list, of length `Max` or less, containing the last valid `FileWrappers` associated with the target object.

`destroy(Iterator) -> ok`

Types:

Iterator = #objref

This operation terminates the target object.

CosFileTransfer_VirtualFileSystem

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

Exports

`'_get_file_system_type'(VFS) -> Return`

Types:

VFS = #objref
Return = 'FTP' | 'NATIVE'

This read only attribute represents the target object's associated file system.

`'_get_supported_content_types'(VFS) -> Return`

Types:

VFS = #objref
Return =

This read only attribute represents the target object's supported content types.

`login(VFS, User, Password, Account) -> Return`

Types:

VFS = #objref
User = Password = Account = string()
Return = {FileTransferSession, Directory} | {'EXCEPTION', E}
FileTransferSession = Directory = #objref

This operation creates a new instance of a `FileTransferSession` and a `Directory`. The later represents the current working directory of the returned `FileTransferSession`.

CosFileTransfer_FileTransferSession

Erlang module

To get access to the record definitions for the structures use:
`-include_lib("cosFileTransfer/include/*.hrl").`

Exports

`'_get_protocols_supported'(FTS) -> Return`

Types:

FTS = #objref
Return = [#'CosFileTransfer_ProtocolSupport'{protocol_name=Type, addresses=[Address]}]
Type = Address = string()

This read only attribute returns the protocols supported by the target object.

`set_directory(FTS, Directory) -> Return`

Types:

FTS = Directory = #objref
Return = ok | {'EXCEPTION, E}

Invoking this operation will change the current working directory of the target object's associated file system. If fail to do so the appropriate exception is raised.

`create_file(FTS, FileNameList) -> Return`

Types:

FTS = #objref
FileNameList = [string()]
Return = File | {'EXCEPTION, E}
File = #objref

This operation creates a `File` Object representing a file which may or may not exist. For this operation to be independent of the working directory the supplied `FileNameList` must represent the absolute name.

`create_directory(FTS, FileNameList) -> Return`

Types:

FTS = #objref
FileNameList = [string()]
Return = Directory | {'EXCEPTION, E}
Directory = #objref

This operation creates a new directory in the target objects associated file systems domain. If fail to do so an exception is raised but, if successful, a `Directory` object representing the new directory is returned.

`get_file(FTS, FileNameList) -> Return`

Types:

```
FTS = #objref
FileNameList = [string()]
Return = FileWrapper | {'EXCEPTION, E}
FileWrapper = #'CosFileTransfer_FileWrapper'{the_file = File file_type =
Type}
File = #objref
Type = nfile | ndirectory
```

This operation, creates a `FileWrapper` which represents a file or directory, and should be independent of the working Directory, i.e., a full path name must be supplied. Furthermore, the file or directory represented by the `FileNameList` must exist.

`delete(FTS, File) -> Return`

Types:

```
FTS = File = #objref
Return = ok | {'EXCEPTION', E}
```

This operation removes the file or directory, represented by the `File` object, from the target objects associated file system. If it is a non-empty directory or non-existing file or directory an exception is raised.

`transfer(FTS, SourceFile, DestinationFile) -> Return`

Types:

```
FTS = SourceFile = DestinationFile = #objref
Return = ok | {'EXCEPTION', E}
```

If the target object's and the `DestinationFile's` associated `FileTransferSession's` support the same protocol(s) this operation will copy the file represented by the `SourceFile` from the target object's file system to a file in the destination `FileTransferSession's` file system. The file is represented by the `DestinationFile` object and may not exist. This operation must be invoked on the `FileTransferSession` associated with the `SourceFile` object.

`append(FTS, SourceFile, DestinationFile) -> Return`

Types:

```
FTS = SourceFile = DestinationFile = #objref
Return = ok | {'EXCEPTION', E}
```

This operation behaves almost like the `transfer/3` operation. The difference is that the `DestinationFile` must exist since the `SourceFile` will be appended to the `DestinationFile`.

Currently, it is not possible to use this operation when the target object represents FTP.

`insert(FTS, SourceFile, DestinationFile, Offset) -> Return`

Types:

```
FTS = SourceFile = DestinationFile = #objref
Offset = long()
Return = ok | {'EXCEPTION', E}
```

This operation behaves almost like the `append/3` operation. The difference is that the `SourceFile` will be inserted into the `DestinationFile` `Offset` bytes from the start of the file.

Currently, it is not possible to use this operation when the target object represents FTP.

```
logout(FTS) -> ok
```

Types:

FTS = #objref

This operation terminates the target object and closes the connection to the file system it represents.