HostProcControl: Programming Guide

DocumentId:GradSoft-PR-e-22.08.2000-1.0b

July 8, 2001

Contents

1 Project mission

HostProcControl is a CORBA service, which provides API for remote managing of an operating system. Using HostProcControl, you can create CORBA-objects able to give:

1. an access to host file system at the rate of:
   (a) reading and writing of files;
   (b) deleting of files;
   (c) checking of files status;
2. an access to processes at the host at the rate of:
   (a) receiving of the list of process running;
   (b) reading parameters of a single process specified;
   (c) starting processes; and
   (d) killing ones.

You can use HostProcControl with several ORB realizations such as TAO, omniORB and ORBacus for UNIX and with ORBacus only for Windows NT (see *Administration Guide* in the file `AdministrationGuide_eng.pdf` for more details). The package is written on C++ and being distributed in the form of source code.

2 General rules to use

2.1 An overall using procedure

There are three types of HostProcControl CORBA-object:

1. HostControlHome;
2. HostControl;
3. FileReader;
4. FileWriter.

The first is created automatically when server starts, the rest are created via it:

1. HostControl: being created via HostControlHome;

2. FileWriter and FileReader: being created via HostControl.

Thus, an overall procedure of using HostProcControl functionality consists of following steps:

1. ORB initialization, obtaining of HostControlHome initial object reference;
2. Getting instance of HostControlHome;
3. Creating instance of HostControl;
4. Using HostControl; creating, using, and deleting FileReader and FileWriter;
5. Deleting HostControl.

An order of obtaining of initial object reference depends on manner of its representation; the last is described in *Administration Guide*.

2.2 Brief object characterization

1. HostControlHome:
   - Being created automatically when server starts; being got to use via object reference published at the time of server start;
   - Checks client access permissions and creates HostControl object;
2. HostControl:
   - Being created by means of HostControlHome::getHostControl method;
   - Gives an access to processes working at the host; deletes remote files, checks remote file status; creates objects FileReader and FileWriter exposing API for reading and writing of remote files;
   - Being destroied by means of HostControl::destroy().
3. FileReader:
   - Being created by means of HostControl::createFileReader() method;
   - Exposes API for reading of remote files;
   - Being destroied by means of FileReader::close().
4. FileWriter:
   - Being created by means of HostControl::createFileWriter() method;
   - Exposes API for writing of remote files;
   - Being destroied by means of FileWriter::close().

2.3 Access permissions control

All HostProcControl functionality is accessible via object named HostControl. Checking if client access permitted is carried out before the creation of this object. To create HostControl object, client invokes HostControlHome's getHostControl method with two parameters considering to be a complex "client ID" consisting of user name and user password. HostControlHome receives ID and seeks the same in user list. User list is the file at the host being edited by service administrator and being composed of strings like following:

```
John : Johnson
A.E.Belyaev:265-60-43
<another_name>:<another_password>
```

Access permits if ID corresponding has been found, and not permits otherwise.

2.4 Lifetime of objects

The common rule reads as follows: client must not remember to destroy each object created by itself. Nevertheless, you can command to object such as HostControl, FileReader and FileWriter to destroy itself in the case having not been used during some timeout fixed. For this pupose every object has a set_timeout method being invoked with timeout value for the object (in seconds) as a parameter. There are next features of timeout control system now:

1. calling set_timeout(0) means return to default infinite-timeout state, i.e. "zero equal to infinity" in present context;
2. next acts are considered to be "using" of the object:
   (a) creating object;
   (b) calling any object's methods excluding methods close() and destroy();
3. HostControl parent object can not be automatically destructed while at least one of its daughter object exists;
4. destruction of HostControl's daughter object fullfils in fact by parent object and conforms to evident using of HostControl.

3 Objects characterization

3.1 HostControlHome

3.1.1 Overview

This HostControlHome object is created automatically when server starts. This is a "root" object of the service. Depending on server options used, an access to this object may be obtained via three (different) vays (see *Administration Guide* for details):

- by resolving corbaloc IOR of following format:

  ```
  HostControlService=corbaloc::<host>:<port>/HostControlService
  ```

- using NameService, via refering to **HostControlService** object in RootNamingContext
- by resolving stringfied IOR may be placed in file or outtyped on standard output when server starts.

The goal of the object is to check access permissions and to create HostControl object if check-up has been succesfull (see subsection *Access permissions control* for more details 2.3).

3.1.2 Interface

```
interface HostControlHome
{
    HostControl
        getHostControl(in string name, in string password)
            raises(LoginIncorrect);
};
```

3.1.3 Description of the method getHostControl

1. **Functioning:** This method getHostControl checks presence of record composed from name and password values (regarding as "client ID") in user list connected with the server 2.3. If check-out is successful, an example of HostControl object is created. If "client ID" is not recorded in the user list, then LoginIncorrect exception is thrown 4.1. If check-out is impossible (user list not found or can not be opend), then AccessControlFailure exception is thrown 4.2
2. **Parameters:**
   (a) name - represents a user name
   (b) password - represents a user password
3. **Return:** HostControl object

3.1.4 Example

Let we have corbaloc style IOR passed into our client application via option such a following:

```
-ORBInitRef HostControlService=corbaloc::127.0.0.1:1025/HostControlService
```

Using next code presented we get accesss to HostControlHome and try to create HostControl object:

```
   /* variables needed */
   Object_var          obj;
   HostControl_var     hostControl;
   HostControlHome_var hostControlHome;

   /* ORB initialization using options from the command line */
   myORB = ORB_init(argc,argv);

      /* getting access to HostControlHome */
      obj = myORB->resolve_initial_references( "HostControlService" );
      hostControlHome = HostControlHome::_narrow(obj);

   /* getting client ID */
   char name[]="name";
   char password[]="password";

   try {
      /* trying to create HostControl */
      hostControl = hostControlHome->getHostControl(name,password);

   } catch ( const LoginIncorrect& ex ) {
     /* error message may be present here */
     goto quit;
   }
   /* using of HostControl may be present here */

   /* deleting of HostControl */
   hostControl->destroy();

quit:
   myORB->destroy();
```

3.2 HostControl

3.2.1 Overview

This HostControl object is the most powerfull object of the service. It's held the next functionality:

1. process control at the rate of:
   (a) obtaining list of processes exist;
   (b) reading parameters of single process specified;
   (c) starting processes;

(d) killing processes;

2. deleting files, reading status of the files;
3. creating daughter objects FileReader and FileWriter to access to file contents.

3.2.2 Interface

```
interface HostControl
{
  ProcInfoSeq getProcsInfo() raises(HostError);
  ProcInfo getProcInfoByPid(in unsigned long pid) raises(HostError);

  unsigned long bornProcByName(in string name,in StringSeq args) raises(HostError);
  void killProcByName(in short signal, in string name) raises(HostError);
  void killProcByPid(in short signal, in unsigned long pid) raises(HostError);

  FileReader createFileReader(in string path) raises(HostError);
  FileWriter createFileWriter(in string path,in WriteMode mode) raises(HostError);

  boolean statFile(in string path, in char arg) raises(HostError);
  void rmFile(in string path) raises(HostError);

  void set_timeout(in unsigned long timeout);
  void destroy();
};
```

3.2.3 Description of the methods

A. Methods to process control

- getProcInfoByPid

1. **Functioning:** hands over to client information about single process specified.
2. **Parameters:**
   (a) pid = Process ID
3. **Return:** Information about process in structure ProcInfo; ProcInfo interface:

```
struct ProcInfo
{
  unsigned long pid;    // the Process ID
  string        name;   // the name of executable
  unsigned long ppid;   // the PID of the parent
};
```

- getProcsInfo

    1. **Functioning:** hands over to user information about all processes running at the host.
    2. **Parameters:** are absent
    3. **Return:** Process list in the form of sequence of ProcInfo named ProcInfoSeq; ProcInfoSeq interface:

        ```
        typedef sequence<ProcInfo> ProcInfoSeq;
        ```

- bornProcByName

    1. **Functioning:** Starts a process at the host; returns ID of created process or throws HostError when failure occurs.
    2. **Parameters:**
        (a) name = path to program must be run
        (b) args = argument list in the form of StringSeq defined with next interface:

        ```
        typedef sequence<string> StringSeq;
        ```

    3. **Return:** ID of process created

- killProcByName

    1. **Functioning:** kills process found by name or throws HostError exception
    2. **Parameters:**
        (a) signal = a signal to dipatch to process
        (b) name = name of process must be killed
    3. **Return:** nothing

- killProcByPid

    1. **Functioning:** kills process found by number (by process ID) or throws HostError exception
    2. **Parameters:**
        (a) signal = a signal to dipatch to process
        (b) pid = ID of process must be killed
    3. **Return:** nothing

B. Methods to control of the host file system:

- statFile

1. **Functioning:** Check status of file specified.
2. **Parameters:**
   (a) path = the path to the file
   (b) arg = encoded assumptions about the file status
3. **Return:** true, if the real file status agrees with assumption about file status encoded in arg; false otherwize.

The relation between set of arg values and assumptions correcponding is shown in the tables:

for any platform:

arg value	assumption
'e'	file exists
's'	file exists and is not zero size
'c'	file exists and is character special
'd'	file exists and is directory
'f'	file exists and is'nt directory
'r'	file exists and is'nt read protected
'w'	file exists and is'nt write protected
'x'	file exists and may be run

for UNIX extra:

arg value	assumption
'h'	file exists and is symbolic link
'b'	file exists and is block special
'p'	file exists and is named pipe (fifo)
'u'	file exists and can invoke setuid
'g'	file exists and can invoke setgid

All values of arg being not included in this table are not permited; using of them leads to InvalidModeArgument exception will be throwed.

- rmFile(in string path) raises(HostError);

  1. **Functioning:** Removes the file specified.
  2. **Parameters:**
     (a) path = the name of file
  3. **Return:** nothing

- createFileReader

  1. **Functioning:** Creates FileRerader instance for the file specified.
  2. **Parameters:**
     (a) path = the name of file to read

3. **Return:** FileReader object

- createFileWriter

1. **Functioning:** Creates FileWriter instance for the file specified.
2. **Parameters:**
   (a) path = the name of file to write
   (b) mode = parameter fixing precedence rule in the case of file to write already exist:
      i. if the mode being of WriteMode type is equal to WriteMode::enRewrite, then the file will be rewritten;
      ii. otherwize if the mode is equal to WriteMode::enAppend, new information will be appended to the end of that.

   WriteMode interface described as follows:

```
enum WriteMode { enRewrite, enAppend };
```

3. **Return:** FileWriter object

C. Methods to support:

- set_timeout

1. **Functioning:** Sets latency interval for HostControl. Having not been used during more than this interval, HostControl object will be automatically destructed by HostControlHome (see subsection *Lifitime of objects* 2.4 for more details)
2. **Parameters:**
   (a) timeout = latency interval to set
3. **Return:** nothing

- destroy

1. **Functioning:** Destroys HostControl object. (NB:) Client must not remember to destroy each CORBA-object created by itself.
2. **Parameters:** is absent
3. **Return:** nothing

3.2.4 Examples

Let us assume "hostControlHome" is object variable obtained as stated above and "<learner>:<test>" is string in user list;

1. next code outtypes the list of processes working at the host:

```
HostControl_var hostControl;
ProcInfoSeq_var procseq;

/* create HostControl object */
try {
   hostControl = hostControlHome->getHostControl("learner","test");
} catch ( const LoginIncorrect& ex ) {
   /* error message: ... */
   goto quit;
}
  /* obtain the process list */
  procseq = hostControl->getProcsInfo();

/* outtype process list */
for(int i = 0; i< procseq->length() ;  i++) {
    cout << procseq[i].pid  << "  ";
    cout << procseq[i].name << "  ";
    cout << procseq[i].ppid << endl;
}

/* destroy the object */
hostControl->destroy();

quit:
   .....
```

2. next code check if the fusty file "garbage.txt" exist and then removes it:

```
HostControl_var hostControl;

/* create HostControl object */
try {
   hostControl = hostControlHome->getHostControl("learner","test");
} catch ( const LoginIncorrect& ex ) {
   /* error message: ... */
   goto quit;
}

if ( hostControl->statFile("garbage.txt",'f') ) {
   hostControl->rmFile("garbage.txt")
}

/* destruction of the object */
```

```
    hostControl->destroy();

  quit:
    .....
```

3.3 FileReader

This object FileReader provides access to byte sequence type data stored in remote file being opend at the time of creating of the object by means of HostControl::createFileReader 3.2.3.

3.3.1 Interface

```
interface FileReader
{
  OctSeq
      read(in unsigned long nbytes, out unsigned long actual_nbytes)
           raises(HostError);

  boolean eof() raises(HostError);

  void set_timeout(in unsigned long timeout );

  void close() raises(HostError);
};
```

3.3.2 Description of methods

- read
  1. **Functioning:** Reads sequence of byte from remoute file and returns it to client application.
  2. **Parameters:**
     (a) nbytes = number of bytes desired to read
     (b) actual_nbytes = number of bytes actual readed
     (nbytes must be equal to actual_nbytes while end of file not reached)
  3. **Return:** data readed in the form of OctSeq; OctSeq interface:

     ```
     typedef sequence<octet> OctSeq;
     ```

- eof
  1. **Functioning:** Returns true if the end of file will be reached, false otherwize.
  2. **Parameters:** is absent

3. **Return:** true if the end of file will be reached and false otherwize.

- set_timeout

  1. **Functioning:** Set the time in seconds during which FileReader object may exist without using (see subsection *Lifetime of objects* 2.4 for details).
  2. **Parameters:**
     (a) timeout - time in seconds mentioned
  3. **Return:** nothing

- close()

  1. **Functioning:** Close file and destroys FileReader object. (NB:) Client must not remember to destroy each CORBA-object created by itself.
  2. **Parameters:** no parameters
  3. **Return:** nothing

3.3.3 Example

Let us assume that "hostControl" is object variable obtained as stated above; next code copies remote file demo.txt into standard output:

```
/* create FileRedader object */
FileReader_var freader = hostControl->createFileRedader("demo.txt");

OctSeq_var data = new OctSeq;
const int buffer_size = 10000;
data -> length( buffer_size );

char buff[buffer_size+1];
int readed;
do {
       /* read the portion of data from the file */
       data = filereader->read( buffer_size, readed );

   /* send data into standard output */
   memcpy(buff,data->get_buffer(),readed);
   buff[readed+1]='\0';
   cout << buff;
} while ( readed==buffer_size ); // repeat while not eof reached

/* close file and destroy FileReader object */
freader->close();
```

3.4 FileWriter

This object FileWriter wrtes byte sequence type data into remote file being opend at the time of creating of the object by means of HostControl::createFileWriter 3.2.3.

3.4.1 Interface

```
interface FileWriter
{
  void     write(in unsigned long nbytes, in OctSeq bytes) raises(HostError);

  void     set_timeout(in unsigned long timeout);

  void     close() raises(HostError);  \
};
```

3.4.2 Description of methods

- write
  1. **Functioning:** Writes sequence of byte to the file specified
  2. **Parameters:**
     (a) nbytes
     (b) bytes = data to read in the form of byte sequence; OctSeq interface:

     ```
     typedef sequence<octet> OctSeq;
     ```
  3. **Return:** nothing
- set_timeout
  1. **Functioning:** Set the time in seconds during which FileWriter object may exist without using (see subsection *Lifetime of objects* 2.4 for details).
  2. **Parameters:**
     (a) timeout = timeout mentioned
  3. **Return:** nothing
- close()
  1. **Functioning:** Close file and destroys FileWriter object. (NB:) Client must not remember to destroy each CORBA-object created by itself.
  2. **Parameters:** no parameters
  3. **Return:** nothing

3.4.3 Example

Let us assume that "hostControl" is object variable obtained as stated above; next code copies file demo.txt from client to host:

```
/* create FileWriter object*/
FileWriter_var filewriter = hostControl->createFileWriter( "demo.txt", UAKG_enRewrite );
/* create file to read */
FILE* file = fopen("demo.txt","rb");

const int buffer_size = 10000;
OctSeq_var buf = new OctSeq;
buf -> length( buffer_size );

int  readed;
while ( !feof( file ) ) {
    /* read data from the local file */
    readed = fread(buf->data(),1,buffer_size,file);

    /* write data to the remote file */
    filewriter->write(readed,buf);

    if ( ferror( file )) {
        break;
    }
}
/* close files and destroy FileWriter object */
fclose( file );
filewriter->close();
```

4 Description of exceptions may being thrown

4.1 InvalidLogin

This `InvalidLogin` exception arises when creating HostControl object in the case client invoking getHostControl is not registered in the user list (see description of HostControlHome::getHostControl method for details 3.1.3);

- Interface:

  ```
  exception InvalidLogin {};
  ```

4.2 AccessControlFailure

This `AccessControlFailure` exception arises when creating HostControl object in the case access rights control is found to be impossible (for example, user list is not found) (see description of HostControlHome::getHostControl 3.1.3 too);

- Interface:

```
exception AccessControlFailure {};
```

4.3 InvalidModeArgument

This InvalidModeArgument exception arises when invoking HostControl::statFile method for inadmissible value of second parameter "arg" (see description of HostControl::statFile method for details 3.2.3).

- Interface:

```
exception InvalidModeArgument {};
```

4.4 HostError

This is the general-duty exception arising in the case of system error at the host mashine occurs.

- Interface:

```
exception HostError
{
  long   os_errno;   // error number according to POSIX
  string errstr;     // description of system error
  string add_info;   // reserved
};
```

5 Model client application

This model client application demonstrates an order of access to HostProcControl functionality using corbaloc url. Specific rules to adapting of the server see in ORB documentation.

```
using namespace CORBA;
static ORB_var myORB ;

#define NAME_LEN 20
#define PASSWD_LEN 20

int main(int argc,char** argv)
{
   char name[20],password[20];
   myORB = ORB_init(argc,argv);
   Object_var  obj;
```

```
   int retval=1;
   try {
     obj=myORB->string_to_object(
         "corbaloc::x.internal.company.com:1000/HostControlService");
   } catch ( const SystemException& ex ) {
     cerr << argv[0] << ": Can\'t resolve HostControlService reference" << endl;
     goto quit;
   }
   if ( is_nil(obj)  ) {
     cerr << "Object is NULL" << endl;
     goto quit;
   }
   HostControlHome_var   hstCntrHome = HostControlHome::_narrow(obj);
   if ( is_nil(HstCntrHome) ) {
     cout << "Can\'t  narrow object " << endl;
     goto quit;
   }
   cout << "Enter name :" << flush ;
   cin.get_line(name,NAME_LEN);
   cout << "Enter password :" << flush;
   cin.get_line(password,PASSWD_LEN);
   HostControl_var hostControl ;
   try {
      hostControl = HstCntrHome->getHostControl(name,password);
   } catch (const SystemException&  ex ) {
      cerr << "System Exception" << endl;
      goto quit;
   } catch (const IncorrectLogin&  ex ) {
      cerr << "Login incorrect" << endl;
      goto quit;
   }
   cout << "login succesfull" << endl;
   retval=0;
   //
   // do you work here.
   //
   hostControl->destroy();
quit:
   myORB->destroy();
   return retval;
}
```

6 Support

The package HostProcControl is created and supported by GradSoft company, the home page of GradSoft is `http://www.gradsoft.com.ua/`. The current release number of the project is **HostProcControl-1.0**.

7 History of changes

16.03.2001 - current revision;

22.02.2001 - first public revision.