IcePatch2 Client Utility Library

IcePatch2 includes a pair of C++ classes that simplify the task of writing your own patch client, along with a Microsoft Foundation Classes (MFC) example that shows how to use these classes. You can find the MFC example in the subdirectory `demo/IcePatch2/MFC` of your Ice distribution.

The remainder of this section discusses the classes. To incorporate them into a custom patch client, your program must include the header file `IcePatch2/ClientUtil.h` and link with the `IcePatch2` library.

On this page:

Performing a Patch

The `Patcher` class encapsulates all of the patching logic required by a client:

C++

```
namespace IcePatch2 {
class Patcher : ... {
public:

    Patcher(const Ice::CommunicatorPtr& communicator,
            const PatcherFeedbackPtr& feedback);

    Patcher(const FileServerPrx& server,
            const PatcherFeedbackPtr& feedback,
            const std::string& dataDir, bool thorough,
            Ice::Int chunkSize, Ice::Int remove);

    bool prepare();

    bool patch(const std::string& dir);

    void finish();
};
typedef IceUtil::Handle<Patcher> PatcherPtr;
}
```

Constructing a Patcher

The constructors provide two ways of configuring a `Patcher` instance. The first form obtains the following IcePatch2 configuration properties from the supplied communicator:

- `IcePatch2.InstanceName`
- `IcePatch2.Endpoints`
- `IcePatch2.Directory`
- `IcePatch2.Thorough`
- `IcePatch2.ChunkSize`
- `IcePatch2.Remove`

The second constructor accepts arguments that correspond to each of these properties.

Both constructors also accept a `PatcherFeedback` object, which allows the client to monitor the progress of the patch.

Executing the Patch

`Patcher` provides three methods that reflect the three stages of a patch:

- `bool prepare()`
  The first stage of a patch includes reading the contents of the checksum file (if present), retrieving the file information from the server, and examining the local data directory to compose the list of files that require updates. The `PatcherFeedback` object is notified incrementally about each local task and has the option of aborting the patch at any time. This method returns true if patch preparation completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `prepare` raises an exception in the form of a `std::string` containing a description of the problem.

- `bool patch(const std::string& dir)`
  The second stage of a patch updates the files in the local data directory. If the `dir` argument is an empty string or `"."`, `patch` updates the entire data directory. Otherwise, `patch` updates only those files whose path names begin with the path in `dir`. For each file requiring an update, `Patcher` downloads its compressed data from the server and writes it to the local data directory. The `PatcherFeedback` object is notified about the progress of each file and, as in the preparation stage, may abort the patch if necessary. This method returns true if patching completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `patch` raises an exception in the form of a `std::string` containing a description of the problem.

- `void finish()`
  The final stage of a patch writes a new checksum file to the local data directory. If an error occurs, `finish` raises an exception in the form of a `std::string` containing a description of the problem.

The code below demonstrates a simple patch client:

C++

```
#include <IcePatch2/ClientUtil.h>
...
Ice::CommunicatorPtr communicator = ...;
IcePatch2::PatcherFeedbackPtr feedback = new MyPatcherFeedbackI;
IcePatch2::PatcherPtr patcher =
    new IcePatch2::Patcher(communicator, feedback);

try {
    bool aborted = !patcher->prepare();
    if(!aborted)
        aborted = !patcher->patch("");
    if(!aborted)
        patcher->finish();
    if(aborted)
        cerr << "Patch aborted" << endl;
} catch(const string& reason) {
    cerr << "Patch error: " << reason << endl;
}
```

For a more sophisticated example, see `demo/IcePatch2/MFC` in your Ice distribution.

Monitoring Patch Progress

The class `PatcherFeedback` is an abstract base class that allows you to monitor the progress of a `Patcher` object. The class declaration is shown below:

C++

```
namespace IcePatch2 {
class PatcherFeedback : ... {
public:

    virtual bool noFileSummary(const std::string& reason) = 0;

    virtual bool checksumStart() = 0;
    virtual bool checksumProgress(const std::string& path) = 0;
    virtual bool checksumEnd() = 0;

    virtual bool fileListStart() = 0;
    virtual bool fileListProgress(Ice::Int percent) = 0;
    virtual bool fileListEnd() = 0;
```

```
    virtual bool patchStart(
        const std::string& path, Ice::Long size,
        Ice::Long updated, Ice::Long total) = 0;
    virtual bool patchProgress(
        Ice::Long pos, Ice::Long size,
        Ice::Long updated, Ice::Long total) = 0;
    virtual bool patchEnd() = 0;
};
typedef IceUtil::Handle<PatcherFeedback> PatcherFeedbackPtr;
}
```

Each of these methods returns a boolean value:

- true allows `Patcher` to continue
- false directs `Patcher` to abort the patch.

The methods are described below.

- `bool noFileSummary(const std::string& reason)`
  Invoked when the local checksum file cannot be found. Returning true initiates a thorough patch, while returning false causes `Patcher::prepare` to return false.

- `bool checksumStart()`
  `bool checksumProgress(const std::string& path)`
  `bool checksumEnd()`
  Invoked by `Patcher::prepare` during a thorough patch. The `checksumProgress` method is invoked as each file's checksum is being computed.

- `bool fileListStart()`
  `bool fileListProgress(Ice::Int percent)`
  `bool fileListEnd()`
  Invoked by `Patcher::prepare` when collecting the list of files to be updated. The `percent` argument to `fileListProgress` indicates how much of the collection process has completed so far.

- `bool patchStart(const std::string& path, Ice::Long size, Ice::Long updated, Ice::Long total)`
  `bool patchProgress(Ice::Long pos, Ice::Long size, Ice::Long updated, Ice::Long total)`
  `bool patchEnd()`
  For each file that requires updating, `Patcher::patch` invokes `patchStart` to indicate the beginning of the patch, `patchProgress` one or more times as chunks of the file are downloaded and written, and finally `patchEnd` to signal the completion of the file's patch. The `path` argument supplies the path name of the file, and `size` provides the file's compressed size. The `pos` argument denotes the number of bytes written so far, while `updated` and `total` represent the cumulative number of bytes updated so far and the total number of bytes to be updated, respectively, of the entire patch operation.

See Also

- IcePatch2 Properties