Adding a Freeze Map to the C++ File System Server

Here we present a C++ implementation of the file system server.

On this page:

Generating the File System Maps in C++

Now that we have selected our key and value types, we can generate the maps as follows:

```
$ slice2freeze -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB::IdentityFileEntryMap,Ice::Identity,\
    FilesystemDB::FileEntry \
    IdentityFileEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
$ slice2freeze -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB::IdentityDirectoryEntryMap,Ice::Identity,\
    FilesystemDB::DirectoryEntry \
    IdentityDirectoryEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map classes are named `IdentityFileEntryMap` and `IdentityDirectoryEntryMap`.

The Server Main Program in C++

The server's `main` program is very simple:

C++

```
#include <FilesystemI.h>
#include <IdentityFileEntryMap.h>
#include <IdentityDirectoryEntryMap.h>
#include <Ice/Application.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;
using namespace FilesystemDB;

class FilesystemApp : public virtual Ice::Application
{
public:

    FilesystemApp(const string& envName)
        : _envName(envName)
    {
    }

    virtual int run(int, char*[])
    {
        shutdownOnInterrupt();

        Ice::ObjectAdapterPtr adapter =
            communicator()->createObjectAdapter("MapFilesystem");

        const Freeze::ConnectionPtr connection(
            Freeze::createConnection(communicator(), _envName));

        const IdentityFileEntryMap fileDB(connection, FileI::filesDB());
        const IdentityDirectoryEntryMap dirDB(
                                  connection,
                                  DirectoryI::directoriesDB());

        adapter->addDefaultServant(new FileI(communicator(), _envName), "file");
        adapter->addDefaultServant(new DirectoryI(communicator(), _envName), "");

        adapter->activate();

        communicator()->waitForShutdown();

        if(interrupted())
            cerr << appName() << ": received signal, shutting down" << endl;

        return 0;
    }

private:

    string _envName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

Let us examine the code in detail. First, we are now including `IdentityFileEntry.h` and `IdentityDirectoryEntry.h`. These header files includes all of the other Freeze (and Ice) header files we need.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

C++

```
    FilesystemApp(const string& envName)
        : _envName(envName) {}
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

The interesting part of `run` are the few lines of code that create the database connection and the two maps that store files and directories, plus the code to add the two default servants:

C++

```
        const Freeze::ConnectionPtr connection(
            Freeze::createConnection(communicator(), _envName));

        const IdentityFileEntryMap fileDB(connection, FileI::filesDB());
        const IdentityDirectoryEntryMap dirDB(
                                    connection,
                                    DirectoryI::directoriesDB());

        adapter->addDefaultServant(new FileI(communicator(), _envName), "file");
        adapter->addDefaultServant(new DirectoryI(communicator(), _envName), "");
```

`run` keeps the database connection open for the duration of the program for performance reasons. As we will see shortly, individual operation implementations will use their own connections; however, it is substantially cheaper to create second (and subsequent connections) than it is to create the first connection.

For the default servants, we use `file` as the category for files. For directories, we use the empty default category.

The Servant Class Definitions in C++

The class definition for `FileI` is very simple:

C++

```
namespace Filesystem {
    class FileI : public File {
    public:
        FileI(const Ice::CommunicatorPtr& communicator,
              const std::string& envName);

        // Slice operations...

        static std::string filesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

The `FileI` class stores the communicator and the environment name. These members are initialized by the constructor. The `filesDB` static member function returns the name of the file map, and the `halt` member function is used to stop the server if it encounters a catastrophic error.

The `DirectoryI` class looks very much the same, also storing the communicator and environment name. The `directoriesDB` static member function returns the name of the directory map.

C++

```
namespace Filesystem {
    class DirectoryI : public Directory {
    public:
        DirectoryI(const Ice::CommunicatorPtr& communicator,
                   const std::string& envName);

        // Slice operations...

        static std::string directoriesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

Implementing `FileI` with a Freeze Map in C++

The `FileI` constructor and the `filesDB` and `halt` member functions have trivial implementations:

C++

```
FileI::FileI(const Ice::CommunicatorPtr& communicator,
             const string& envName)
    : _communicator(communicator), _envName(envName)
{
}

string
FileI::filesDB()
{
    return "files";
}

void
FileI::halt(const Freeze::DatabaseException& ex) const
{
    Ice::Error error(_communicator->getLogger());
    error << "fatal exception: " << ex << "\n*** Aborting application ***";

    abort();
}
```

The Slice operations all follow the same implementation strategy: we create a database connection and the file map and place the body of the operation into an infinite loop:

C++

```
string
FileI::someOperation(/* ... */ const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {

            // Operation implementation here...

        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Each operation creates its own database connection and map for concurrency reasons: the database takes care of all the necessary locking, so there is no need for any other synchronization in the server. If the database detects a deadlock, the code handles the corresponding `DeadlockException` and simply tries again until the operation eventually succeeds; any other database exception indicates that something has gone seriously wrong and terminates the server.

Here is the implementation of the `name` method:

C++

```
string
FileI::name(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            return p->second.name;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

The implementation could hardly be simpler: the default servant uses the identity in the `Current` object to index into the file map. If a record with this identity exists, it returns the name of the file as stored in the `FileEntry` structure in the map. Otherwise, if no such entry exists, it throws `ObjectNotExistException`. This happens if the file existed at some time in the past but has since been destroyed.

The `read` implementation is almost identical. It returns the text that is stored by the `FileEntry`:

C++

```
Lines
FileI::read(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            return p->second.text;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

The `write` implementation updates the file contents and calls `set` on the iterator to update the map with the new contents:

C++

```
void
FileI::write(const Filesystem::Lines& text, const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            FileEntry entry = p->second;
            entry.text = text;
            p.set(entry);
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Finally, the `destroy` implementation for files must update two maps: it needs to remove its own entry in the file map as well as update the `nodes` map in the parent to remove itself from the parent's map of children. This raises a potential problem: if one update succeeds but the other one fails, we end up with an inconsistent file system: either the parent still has an entry to a non-existent file, or the parent lacks an entry to a file that still exists.

To make sure that the two updates happen atomically, `destroy` performs them in a transaction:

C++

```
void
FileI::destroy(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());
    IdentityDirectoryEntryMap dirDB(connection, DirectoryI::directoriesDB());

    for (;;) {
        try {
            Freeze::TransactionHolder txn(connection);

            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            FileEntry entry = p->second;

            IdentityDirectoryEntryMap::iterator pp = dirDB.find(entry.parent);
            if (pp == dirDB.end()) {
                halt(Freeze::DatabaseException(
                       __FILE__, __LINE__,
                       "consistency error: file without parent"));
            }

            DirectoryEntry dirEntry = pp->second;
            dirEntry.nodes.erase(entry.name);
            pp.set(dirEntry);

            fileDB.erase(p);
            txn.commit();
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

As you can see, the code first establishes a transaction and then locates the file in the parent directory's map of nodes. After removing the file from the parent, the code updates the parent's persistent state by calling `set` on the parent iterator and then removes the file from the file map before committing the transaction.

Implementing `DirectoryI` with a Freeze Map in C++

The `DirectoryI::directoriesDB` implementation returns the string `directories`, and the `halt` implementation is the same as for `FileI`, so we do not show them here.

Turning to the constructor, we must cater for two different scenarios:

- The server is started with a database that already contains a number of nodes.
- The server is started for the very first time with an empty database.

This means that the root directory (which must always exist) may or may not be present in the database. Accordingly, the constructor looks for the root directory (with the fixed identity `RootDir`); if the root directory does not exist in the database, it creates it:

C++

```
DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communicator, const string& envName)
    : _communicator(communicator), _envName(envName)
{
    const Freeze::ConnectionPtr connection =
        Freeze::createConnection(_communicator, _envName);
    IdentityDirectoryEntryMap dirDB(connection, directoriesDB());

    for (;;) {
        try {
            Ice::Identity rootId;
            rootId.name = "RootDir";
            IdentityDirectoryEntryMap::const_iterator p = dirDB.find(rootId);
            if (p == dirDB.end()) {
                DirectoryEntry d;
                d.name = "/";
                dirDB.put(make_pair(rootId, d));
            }
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Next, let us examine the implementation of `createDirectory`. Similar to the `FileI::destroy` operation, `createDirectory` must update both the parent's nodes map and create a new entry in the directory map. These updates must happen atomically, so we perform them in a separate transaction:

C++

```
DirectoryPrx
DirectoryI::createDirectory(const string& name, const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection, directoriesDB());

    for (;;) {
        try {
            Freeze::TransactionHolder txn(connection);

            IdentityDirectoryEntryMap::iterator p =
                directoryDB.find(c.id);
            if (p == directoryDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }

            DirectoryEntry entry = p->second;
            if (name.empty() || entry.nodes.find(name) != entry.nodes.end()) {
                throw NameInUse(name);
            }

            DirectoryEntry d;
            d.name = name;
            d.parent = c.id;

            Ice::Identity id;
            id.name = IceUtil::generateUUID();
            DirectoryPrx proxy = DirectoryPrx::uncheckedCast(c.adapter->createProxy(id));

            NodeDesc nd;
            nd.name = name;
            nd.type = DirType;
            nd.proxy = proxy;
            entry.nodes.insert(make_pair(name, nd));

            p.set(entry);
            directoryDB.put(make_pair(id, d));

            txn.commit();

            return proxy;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

After establishing the transaction, the code ensures that the directory does not already contain an entry with the same name and then initializes a new `DirectoryEntry`, setting the name to the name of the new directory, and the parent to its own identity. The identity of the new directory is a UUID, which ensures that all directories have unique identities. In addition, the UUID prevents the accidental rebirth of a file or directory in the future.

The code then initializes a new `NodeDesc` structure with the details of the new directory and, finally, updates its own map of children as well as adding the new directory to the map of directories before committing the transaction.

The `createFile` implementation is almost identical, so we do not show it here. Similarly, the `name` and `destroy` implementations are almost identical to the ones for `FileI`, so let us move to `list`:

C++

```
NodeDescSeq
DirectoryI::list(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection, directoriesDB());

    for (;;) {
        try {
            IdentityDirectoryEntryMap::iterator p = directoryDB.find(c.id);
            if (p == directoryDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            NodeDescSeq result;
            for (StringNodeDescDict::const_iterator q = p->second.nodes.begin();
                 q != p->second.nodes.end(); ++q) {
                result.push_back(q->second);
            }
            return result;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Again, the code is very simple: it iterates over the `nodes` map, adding each `NodeDesc` structure to the returned sequence.

The `find` implementation is even simpler, so we do not show it here.

See Also

- Freeze Maps
- Object Identity and Uniqueness
- The Current Object