IceGrid Administrative Sessions

Previous

Next

To access IceGrid's administrative facilities from a program, you must first establish an administrative session. Once done, a wide range of services are at your disposal, including the manipulation of IceGrid registries, nodes, and servers; deployment of new components such as well-known objects; and dynamic monitoring of IceGrid events.

Note that, for replicated registries, an administrative session can be established with either the master or a slave registry replica, but a session with a slave replica is restricted to read-only operations.

On this page:

Creating an Administrative Session

The `Registry` interface provides two operations for creating an administrative session:

Slice

```
module IceGrid
{
    exception PermissionDeniedException
    {
        string reason;
    }

    interface Registry
    {
        AdminSession* createAdminSession(string userId, string password)
            throws PermissionDeniedException;

        AdminSession* createAdminSessionFromSecureConnection()
            throws PermissionDeniedException;

        idempotent int getSessionTimeout();

        idempotent int getACMTimeout();
        // ...
    }
}
```

The `createAdminSession` operation expects a username and password and returns a session proxy if the client is allowed to create a session. By default, IceGrid does not allow the creation of administrative sessions. You must define the property `IceGrid.Registry.AdminPermissionsVerifier` with the proxy of a permissions verifier object to enable session creation with `createAdminSession`. The verifier object must implement the interface `Glacier2::PermissionsVerifier`.

The `createAdminSessionFromSecureConnection` operation does not require a username and password because it uses the credentials supplied by an SSL connection to authenticate the client. As with `createAdminSession`, you must configure the proxy of a permissions verifier object before clients can use `createAdminSessionFromSecureConnection` to create a session. In this case, the `IceGrid.Registry.AdminSSLPermissionsVerifier` property specifies the proxy of a verifier object that implements the interface `Glacier2::SSLPermissionsVerifier`.

As an example, the following code demonstrates how to obtain a proxy for the registry and invoke `createAdminSession`:

C++11

```
auto base = communicator->stringToProxy("IceGrid/Registry");
auto registry = Ice::checkedCast<IceGrid::RegistryPrx>(base);
string username = ...;
string password = ...;
shared_ptr<IceGrid::AdminSessionPrx> session;
try
{
    session = registry->createAdminSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

C++98

```
Ice::ObjectPrx base = communicator->stringToProxy("IceGrid/Registry");
IceGrid::RegistryPrx registry = IceGrid::RegistryPrx::checkedCast(base);
string username = ...;
string password = ...;
IceGrid::AdminSessionPrx session;
try
{
    session = registry->createAdminSession(username, password);
}
catch(const IceGrid::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
```

The `AdminSession` interface provides operations for accessing log files and establishing observers. Furthermore, two additional operations are worthy of your attention:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        idempotent void keepAlive();
        idempotent Admin* getAdmin();
        // ...
    }
}
```

If your program uses an administrative session indefinitely, you must prevent the session from expiring. You have two options for keeping a session alive:

1. Call `Registry::getSessionTimeout` and periodically invoke `AdminSession::keepAlive` every *timeout* (or less) seconds
2. Call `Registry::getACMTimeout` and configure ACM settings on the connection

We recommend using the second approach, which you can implement as follows:

C++11

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    auto conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, Ice::nullopt, Ice::ACMHeartbeat::HeartbeatAlways);
}
```

C++98

```
int acmTimeout = registry->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr conn = session->ice_getCachedConnection();
    conn->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

Enabling heartbeats on the connection causes Ice to automatically send a heartbeat message at regular intervals determined by the given timeout value. The server ignores these messages, but they serve the purpose of keeping the session alive.

The `getAdmin` operation returns a proxy for the `IceGrid::Admin` interface, which provides complete access to the registry's settings. For this reason, you must use extreme caution when enabling administrative sessions.

Back to Top ^

Accessing Log Files Remotely

IceGrid's `AdminSession` interface provides operations for remotely accessing the log files of a registry, node, or server:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        // ...
        FileIterator* openServerLog(string id, string path, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openServerStdErr(string id, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openServerStdOut(string id, int count)
            throws FileNotAvailableException, ServerNotExistException,
                   NodeUnreachableException, DeploymentException;
        FileIterator* openNodeStdErr(string name, int count)
            throws FileNotAvailableException, NodeNotExistException,
                   NodeUnreachableException;
        FileIterator* openNodeStdOut(string name, int count)
            throws FileNotAvailableException, NodeNotExistException,
                   NodeUnreachableException;
        FileIterator* openRegistryStdErr(string name, int count)
            throws FileNotAvailableException,
                   RegistryNotExistException,
                   RegistryUnreachableException;
        FileIterator * openRegistryStdOut(string name, int count)
            throws FileNotAvailableException,
                   RegistryNotExistException,
                   RegistryUnreachableException;
    }
}
```

In order to access the text of a program's standard output or standard error log, you must configure it using the `Ice.StdOut` and `Ice.StdErr` properties, respectively. For registries and nodes, you must define these properties explicitly but, for servers, the node defines these properties automatically if the property `IceGrid.Node.Output` is defined, causing the server's output to be logged in individual files.

If `IceGrid.Node.Output` is *not* defined, the following rules apply:

- If the node is started from a console or shell, servers share the node's `stdout` and `stderr`. If `Ice.StdOut` and/or `Ice.StdErr` properties are defined for the node, the servers' output is redirected to the specified files as well.
- If the node is started as a Unix daemon and `--noclose` is not used, the servers' output is lost, except if `Ice.StdOut` and/or `Ice.StdErr` properties are set for the node, in which case the servers' output is redirected to the specified files.
- If the node is started as a Windows service, the servers' output is lost even if `Ice.StdOut` and/or `Ice.StdErr` are set.

Log messages from the node itself are sent to `stderr` unless you set `Ice.UseSyslog` (for Unix). If the node is started as a Windows service, its log messages always are sent to the Windows event log.

In the case of `openServerLog`, the value of the `path` argument must resolve to the same file as one of the server's log descriptors. This security measure prevents a client from opening an arbitrary file on the server's host.

All of the operations accept a `count` argument and return a proxy to a `FileIterator` object. The `count` argument determines where to start reading the log file: if the value is negative, the iterator is positioned at the beginning of the file, otherwise the iterator is positioned to return the last `count` lines of text.

The `FileIterator` interface is quite simple:

Slice

```
module IceGrid
{
    interface FileIterator
    {
        bool read(int size, out Ice::StringSeq lines)
            throws FileNotAvailableException;
        void destroy();
    }
}
```

A client may invoke the `read` operation as many times as necessary. The `size` argument specifies the maximum number of bytes that `read` can return; the client must not use a size that would cause the reply to exceed the client's configured maximum message size.

If this is the client's first call to `read`, the `lines` argument holds whatever text was available from the iterator's initial position, and the iterator is repositioned in preparation for the next call to `read`. The operation returns false to indicate that more text is available and true if all available text has been read.

Line termination characters are removed from the contents of `lines`. When displaying the text, you must be aware that the first and last elements of the sequence can be partial lines. For example, the last line of the sequence might be incomplete if the limit specified by `size` is reached. The next call to `read` returns the remainder of that line as the first element in the sequence.

As an example, the C++ code below displays the contents of a log file and waits for new text to become available:

C++11

```
shared_ptr<IceGrid::FileIteratorPrx> iter = ...;
while(true)
{
    Ice::StringSeq lines;
    bool end = iter->read(10000, lines);
    if(!lines.empty())
    {
        // The first line might be a continuation from
        // the previous call to read.
        cout << lines[0];
        for(const auto& p : lines)
        {
            cout << endl << p << flush;
        }
    }
    if(end)
    {
        sleep(1);
    }
}
```

C++98

```
IceGrid::FileIteratorPrx iter = ...;
while(true)
{
    Ice::StringSeq lines;
    bool end = iter->read(10000, lines);
    if(!lines.empty())
    {
        // The first line might be a continuation from
        // the previous call to read.
        cout << lines[0];
        for(Ice::StringSeq::const_iterator p = ++lines.begin(); p != lines.end(); ++p)
        {
            cout << endl << *p << flush;
        }
    }
    if(end)
    {
        sleep(1);
    }
}
```

Notice that the loop includes a delay in case `read` returns true, which prevents the client from entering a busy loop when no data is currently available.

The client should call `destroy` when the iterator object is no longer required. At the time the client's session terminates, IceGrid reclaims any iterators that were not explicitly destroyed.

If the client waits for new data, it must take steps to prevent the administrative session from expiring.

With these operations, an administrative client can retrieve any text file on a system where an IceGrid node is running. While it's common for this text file to contain the output of an Ice logger, it could contain other unrelated outputs. This text file is presented as a sequence of strings, with no particular structure for these strings.

The `Logger` admin facet provides another, and often better, way to retrieve the log messages sent to the logger of a server, node or registry. With the `Logger` facet, you retrieve log messages–and only log messages–as typed structures, whether or not the logger's output is stored in a text file or stored at all. With the `AdminSession` file operations presented above, you can retrieve the log files of a server even when this server is not running; conversely, with the `Logger` admin facet, the target server must be running since this `Logger` facet is hosted in that server. Finally, the programming style espoused by the `AdminSession` file operations is a pull model: the administrative client calls `read` from time to time on the `FileIterator` object provided by IceGrid. With the `Logger` admin facet, the administrative client uses a push model: it registers a remote logger object with the target server, node and registry, and this remote logger receives new log messages as soon as they are generated.

Back to Top ^

Dynamic Monitoring in IceGrid

IceGrid allows an application to monitor relevant state changes by registering callback objects. (The IceGrid GUI tool uses these callback interfaces for its implementation.) The callback interfaces are useful to, for example, automatically generate an email notification when a node goes down or some other state change of interest occurs.

Observer Interfaces

IceGrid offers a callback interface for each major component of the IceGrid architecture:

Slice

```
module IceGrid
{
    interface NodeObserver
    {
        void nodeInit(NodeDynamicInfoSeq nodes);
        void nodeUp(NodeDynamicInfo node);
        void nodeDown(string name);
        void updateServer(string node, ServerDynamicInfo updatedInfo);
        void updateAdapter(string node, AdapterDynamicInfo updatedInfo);
    }

    interface ApplicationObserver
    {
        void applicationInit(int serial, ApplicationInfoSeq applications);
        void applicationAdded(int serial, ApplicationInfo desc);
        void applicationRemoved(int serial, string name);
        void applicationUpdated(int serial, ApplicationUpdateInfo desc);
    }

    interface AdapterObserver
    {
        void adapterInit(AdapterInfoSeq adpts);
        void adapterAdded(AdapterInfo info);
        void adapterUpdated(AdapterInfo info);
        void adapterRemoved(string id);
    }

    interface ObjectObserver
    {
        void objectInit(ObjectInfoSeq objects);
        void objectAdded(ObjectInfo info);
        void objectUpdated(ObjectInfo info);
        void objectRemoved(Ice::Identity id);
    }

    interface RegistryObserver
    {
        void registryInit(RegistryInfoSeq registries);
        void registryUp(RegistryInfo node);
        void registryDown(string name);
    }
}
```

The next section describes how to install an observer.

Back to Top ^

Registering Observers

The `AdminSession` interface provides two operations for registering your observers:

Slice

```
module IceGrid
{
    interface AdminSession extends Glacier2::Session
    {
        idempotent void keepAlive();

        idempotent void setObservers(RegistryObserver* registryObs,
                                     NodeObserver* nodeObs,
                                     ApplicationObserver* appObs,
                                     AdapterObserver* adptObs,
                                     ObjectObserver* objObs)
            throws ObserverAlreadyRegisteredException;

        idempotent void setObserversByIdentity(Ice::Identity registryObs,
                                               Ice::Identity nodeObs,
                                               Ice::Identity appObs,
                                               Ice::Identity adptObs,
                                               Ice::Identity objObs)
            throws ObserverAlreadyRegisteredException;
        // ...
    }
}
```

You should invoke `setObservers` and supply proxies when it is possible for the registry to establish a separate connection to the client to deliver its callbacks. If network restrictions such as firewalls prevent such a connection, you should use the `setObserversByIdentity` operation, which creates a bidirectional connection instead.

You can pass a null proxy for any parameter to `setObservers`, or an empty identity for any parameter to `setObserversByIdentity`, if you want to use only some of the observers. In addition, passing a null proxy or an empty identity for an observer cancels a previous registration of that observer. The operations raise `ObserverAlreadyRegisteredException` if you pass a proxy or identity that was registered in a previous call.

Once the observers are registered, operations corresponding to state changes will be invoked on the observers. (See the Slice API Reference for details on the data passed to the observers. You can also look at the source code for the IceGrid GUI implementation in the Ice for Java distribution to see how observers are used by the GUI.)

Finally, remember to take the steps necessary to prevent the administrative session from expiring.

Back to Top ^

See Also

- Registry Replication
- Securing a Glacier2 Router
- Resource Allocation using IceGrid Sessions
- Log Descriptor Element
- icegridadmin Command Line Tool
- Bidirectional Connections
- IceGrid.*
- Logger admin Facet

Previous

Next