Getting Started with Glacier2

Previous Next

On this page:

Using Glacier2

Using Glacier2 in a minimal configuration involves the following tasks:

1. Write a configuration file for the router.
2. Write a password file for the router. (Glacier2 also supports other ways to authenticate users.)
3. Decide whether to use the router's internal session manager, or supply your own session manager.
4. Start the router on a host with access to the public and private networks.
5. Modify the client configuration to use the router.
6. Modify the client to create a router session.
7. Ensure that the router session remains active for as long as the client requires it.

> For the sake of example, the router's public address is 5.6.7.8 and its private address is 10.0.0.1.

Back to Top ^

Configuring the Router

The following router configuration properties establish the necessary endpoint and define when a session expires due to inactivity:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.SessionTimeout=60
```

The endpoint defined by `Glacier2.Client.Endpoints` is used by the Ice run time in a client to interact directly with the router. It is also the endpoint where requests from routed proxies are sent. This endpoint is defined on the public network interface because it must be accessible to clients. Furthermore, the endpoint uses a fixed port because clients may be statically configured with a proxy for this endpoint. The port numbers 4063 (for TCP) and 4064 (for SSL) are reserved for Glacier2 by the Internet Assigned Numbers Authority (IANA).

> This sample configuration uses TCP as the endpoint protocol, although in most cases, SSL is preferable.

A client must create a session in order to use a Glacier2 router. Our setting for the `Glacier2.SessionTimeout` property causes the router to destroy sessions that have been idle for at least 60 seconds. It is not mandatory to define a timeout, but it is recommended, otherwise session state might accumulate in the router.

Note that this configuration enables the router to forward requests from clients to servers. Additional configuration is necessary to support callbacks from servers to clients.

You must also decide which authentication scheme (or schemes) to use. A file-based mechanism is available, as are more sophisticated strategies.

If clients access a location service via the router, additional router configuration is typically necessary.

Back to Top ^

Writing a Password File

The router's simplest authentication mechanism uses an access control list in a text file consisting of username and password pairs. Passwords are encoded using the `modular crypt format` (`MCF`).

The general structure of a `MCF` encoded password hash is: `$identifier$content`, where `identifier` denotes the scheme used for hashing, and `content` denotes its contents. Glacier2 supports two types of `MCF` encoded password hashes:

On Windows and macOS:

- `PBKDF2` using `SHA-1`, `SHA-256`, or `SHA-512` as the digest algorithm.

  > PBKDF2 does not have a standard form in the `MCF` specification. In this case Glacier2 uses the same format as passlib.
  >
  > - `$pbkdf2-digest$rounds$salt$` for `SHA-256` and `SHA-512`.
  > - `$pbkdf2$rounds$salt$` for `SHA-1`.

On Linux:

- `Crypt` using `SHA-256`, or `SHA-512` as the digest algorithm.

The property `Glacier2.CryptPasswords` specifies the name of the password file:

```
Glacier2.CryptPasswords=passwords
```

The format of the password file is very simple. Each user name-password pair must reside on a separate line, with whitespace separating the user name from the password. For example, the following password file contains an entry for the user name `test`:

```
test $5$rounds=110000$5rM9XIDChkgEu.S3$ov7yip4NOi1wymAZmamEv1uKPQRB0WzasoJsWMpRT19
```

`icehashpassword` Helper Script

You can use the `icehashpassword` helper script to generate these username-password pairs. This script requires `Python` and `pip` to be installed. To install this script run:

```
> pip install zeroc-icehashpassword
```

You can now use the command `icehashpassword`:

```
> icehashpassword
Password:
$5$rounds=110000$5rM9XIDChkgEu.S3$ov7yip4NOi1wymAZmamEv1uKPQRB0WzasoJsWMpRT19
```

You may also specify several optional parameters:

- `-d MESSAGE_DIGEST_ALGORITHM, --digest=MESSAGE_DIGEST_ALGORITHM`
- `-s SALT_SIZE, --salt=SALT_SIZE`
- `-r ROUNDS, --rounds=ROUNDS`

For example,

```
> python icehashpassword.py -r 25000 -s 32 -d sha256
Password:
$pbkdf2-sha256$25000$pJSSEuKcs9aaE.J8711LyRlD6H0P4Tyn9J7znpNyLsU$Yx7NNLDfwwLeMbZV84X2rBYBnvrXvK/TDIQiIGabQIM
```

Note that `icehashpassword` generates PBKDF2 hashes on Windows and macOS, and Crypt hashes on Linux.

> This authentication scheme is intended for use in simple applications with a few users. Most applications should install their own custom permissions verifier.

Back to Top ^

Starting the Router

The router supports the following command-line options:

```
$ glacier2router -h
Usage: glacier2router [options]
Options:
-h, --help     Show this message.
-v, --version  Display the Ice version.
--nowarn       Suppress warnings.
```

The `--nowarn` option prevents the router from displaying warning messages at startup when it is unable to contact a permissions verifier object or a session manager object specified by its configuration.

Additional command line options are supported, including those that allow the router to run as a Windows service or Unix daemon, and Ice includes a utility to help you install the router as a Windows service.

Assuming our configuration properties are stored in a file named `config`, you can start the router with the following command:

```
$ glacier2router --Ice.Config=config
```

Back to Top ^

Configuring a Glacier2 Client

The following properties configure a client to use a Glacier2 router:

```
Ice.Default.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
Ice.RetryIntervals=-1
```

The `Ice.Default.Router` property defines the router proxy. Its endpoints must match those in `Glacier2.Client.Endpoints`.

Setting `Ice.RetryIntervals` to -1 disables automatic retries, which are not useful for proxies configured to use a Glacier2 router.

Back to Top ^

Glacier2 Object Identities

A Glacier2 router hosts one well-known object. The default identity of this object is `Glacier2/router`, corresponding to the `Glacier2::Router` interface. If an application requires the use of multiple different (that is, not replicated) routers, it is a good idea to assign a unique identity to this object by configuring the routers with different values of the `Glacier2.InstanceName` property, as shown in the following example:

```
Glacier2.InstanceName=PublicRouter
```

This property changes the category of the object identity, which becomes `PublicRouter/router`. The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063
```

One exception to this rule is if you deploy multiple Glacier2 routers as replicas, for example, to gain redundancy or to distribute the message-forwarding load over a number of machines. In that case, all the routers must use the same instance name, and the router clients can use proxies with multiple endpoints, such as:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063:tcp -h 6.10.7.8 -p 4063
```

> A client can discover a router's proxy at run time using the `RouterFinder` interface.

Back to Top ^

Creating a Glacier2 Session

Session management is provided by the `Glacier2::Router` interface:

Slice

```
module Glacier2
{
    exception PermissionDeniedException
    {
        string reason;
    }

    interface Router extends Ice::Router
    {
        Session* createSession(string userId, string password)
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        Session* createSessionFromSecureConnection()
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        idempotent string getCategoryForClient();

        void refreshSession()
            throws SessionNotExistException;

        void destroySession()
            throws SessionNotExistException;

        idempotent long getSessionTimeout();

        idempotent int getACMTimeout();
    }
}
```

The interface defines two operations for creating sessions: `createSession` and `createSessionFromSecureConnection`. The router requires each client to create a session using one of these operations; only after the session is created will the router forward requests on behalf of the client.

The `createSession` operation expects a user name and password and, depending on the router's configuration, returns either a `Session` proxy or nil. When using the default authentication scheme, the given user name and password must match an entry in the router's password file in order to successfully create a session.

The `createSessionFromSecureConnection` operation does not require a user name and password because it authenticates the client using the credentials associated with the client's SSL connection to the router.

To create a session, the client typically obtains the router proxy from the communicator, downcasts the proxy to the `Glacier2::Router` interface, and invokes one of the `create` operations. The sample code below demonstrates how to do it in C++; the code will look very similar in the other language mappings.

C++11

```
auto defaultRouter = communicator->getDefaultRouter();
auto router = Ice::checkedCast<Glacier2::RouterPrx>(defaultRouter);
string username = ...;
string password = ...;
shared_ptr<Glacier2::SessionPrx> session;
try
{
    session = router->createSession(username, password);
}
catch(const Glacier2::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
catch(const Glacier2::CannotCreateSessionException& ex)
{
    cout << "cannot create session:\n" << ex.reason << endl;
}
```

C++98

```
Ice::RouterPrx defaultRouter = communicator->getDefaultRouter();
Glacier2::RouterPrx router = Glacier2::RouterPrx::checkedCast(defaultRouter);
string username = ...;
string password = ...;
Glacier2::SessionPrx session;
try
{
    session = router->createSession(username, password);
}
catch(const Glacier2::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
catch(const Glacier2::CannotCreateSessionException& ex)
{
    cout << "cannot create session:\n" << ex.reason << endl;
}
```

If the router is configured with a session manager, the `createSession` and `createSessionFromSecureConnection` operations may return a proxy for an object implementing the `Glacier2::Session` interface (or an application-specific derived interface). The client receives a null proxy if no session manager is configured.

A non-nil session proxy returned by a `create` operation must be configured with the router that created it because the session object is only accessible via the router. If the router is configured as the client's default router at the time `createSession` or `createSessionFromSecureConnection` is invoked, as is the case in the example above, then the session proxy is already properly configured and nothing else is required. Otherwise, the client must explicitly configure the session proxy with a router using the `ice_router` proxy method.

If the client wishes to destroy the session explicitly, it must invoke `destroySession` on the router proxy. If a client does not destroy its session, the router destroys it automatically when it expires due to inactivity. A client can obtain the inactivity timeout value by calling `getSessionTimeout` and keep the session alive by periodically calling `refreshSession` if necessary. An easier solution for keeping the session alive is to call `getACMTimeout` and use this value to configure the Active Connection Management behavior of the client's connection to the router:

C++11

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

C++98

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

The client calls `getACMTimeout` to retrieve the router's server-side ACM timeout and passes this value to an invocation of `setACM` on its connection to the router, thereby ensuring the client and router are using a consistent setting. The client also enables automatic heartbeats so that the connection remains active and prevents the router's server-side ACM from closing the connection.

Alternatively, you could configure the client's ACM with the properties `Ice.ACM.Timeout` and `Ice.ACM.Heartbeat`. Note however that these properties affect all connections created by a client. In general, we recommend configuring the connection directly as shown above.

Whether or not you use `refreshSession` or heartbeats to keep the session alive, Glacier2 will invoke `ice_ping` on the session object for each `refreshSession` call or `heartbeat` received. This `ice_ping` check ensures the session is still alive in the session manager. If it fails, Glacier2 destroys the session and the client is notified by the closure of the connection associated to the session.

The `getCategoryForClient` operation is used to implement callbacks over bidirectional connections.

Example

An example of a Glacier2 client is provided in the directory `demo/Glacier2/callback`.

Back to Top ^

Glacier2 Session Expiration

A Glacier2 router may be configured to destroy sessions after a period of inactivity. This feature allows the router, as well as a custom session manager, to reclaim resources acquired during the session, but it requires some coordination between the router and its clients.

Ideally you would select a session timeout that is long enough to accommodate the usage patterns of your clients. For example, a session timeout of thirty seconds is a reasonable choice for a client that invokes an operation on a back-end server once every five seconds. However, that timeout could disrupt a different client that has long periods of inactivity, such as when its invocations are prompted by human interaction.

If you cannot predict with certainty the usage patterns of your clients, we recommend configuring the clients so that they actively prevent their sessions from expiring. The simplest solution is to enable the heartbeat feature of Active Connection Management, wherein the Ice run time automatically sends a heartbeat message at regular intervals. The Glacier2 router interprets a heartbeat message as an indication that the client is still active and wishes its session to remain alive. ACM settings can be configured for all connections created by a communicator, or individually for a particular connection. Use care to ensure the client's ACM timeout and the router's session timeout are compatible.

Ice includes helper classes that simplify the task of creating a session.

Note that if a session times out, the next client invocation raises `ConnectionLostException`. To re-establish the session, the client must explicitly re-create it. If the client uses callbacks, it must also re-create the callback adapter and re-register its callback servants.

Back to Top ^

Glacier2 Session Destruction

A router session is destroyed automatically when the session expires, and when a client explicitly destroys its session. The router also destroys a session if certain connection errors occur while attempting to route a request. These errors are represented by the run-time exceptions `SocketException`, `TimeoutException`, and `ProtocolException`. In other words, if any of these exceptions occur while Glacier2 attempts to establish a connection to the target back-end server, or forward a request to the target back-end server, the router automatically destroys the session.

Back to Top ^

See Also

- Callbacks through Glacier2
- Securing a Glacier2 Router
- Glacier2 Session Management
- Glacier2.*
- Windows Services
- Active Connection Management
- Automatic Retries

- Glacier2 Application Class

Previous

Next