IceGrid Troubleshooting

On this page:

Troubleshooting Activation Failures

Server activation failure is usually indicated by the receipt of a `NoEndpointException`. This can happen for a number of reasons, but the most likely cause is an incorrect configuration. For example, an IceGrid node may fail to activate a server because the server's executable file, shared libraries, or classes could not be found. There are several steps you can take in this case:

1. Enable activation tracing in the node by setting the configuration property `IceGrid.Node.Trace.Activator=3`.
2. Examine the tracing output and verify the server's command line and working directory are correct.
3. Relative pathnames specified in a command line may not be correct relative to the node's current working directory. Either replace relative pathnames with absolute pathnames, or restart the node in the proper working directory.
4. Verify that the server is configured with the correct `PATH` or `LD_LIBRARY_PATH` settings for its shared libraries. For a Java server, its `CLASSPATH` may also require changes.

Another cause of activation failure is a server fault during startup. After you have confirmed that the node successfully spawns the server process using the steps above, you should then check for signs of a server fault (e.g., on Unix, look for a `core` file in the node's current working directory).

Troubleshooting Proxy Failures

A client may receive `Ice::NotRegisteredException` if binding fails for an indirect proxy. This exception indicates that the proxy's object identity or object adapter is not known by the IceGrid registry. The following steps may help you discover the cause of the exception:

1. Use `icegridadmin` to verify that the object identity or object adapter identifier is actually registered, and that it matches what is used by the proxy:

```
>>> adapter list
...
>>> object find ::Hello
...
```

2. If the problem persists, review your configuration to ensure that the locator proxy used by the client matches the registry's client endpoints, and that those endpoints are accessible to the client (i.e., are not blocked by a firewall).
3. Finally, enable locator tracing in the client by setting the configuration property `Ice.Trace.Locator=2`, then run the client again to see if any log messages are emitted that may indicate the problem.

Troubleshooting Server Failures

Diagnosing a server failure can be difficult, especially when servers are activated automatically on remote hosts. Here are a few suggestions:

1. If the server is running on a Unix host, check the current working directory of the IceGrid node process for signs of a server failure, such as a `core` file.
2. Judicious use of tracing can help to narrow the search. For example, if the failure occurs as a result of an operation invocation, enable protocol tracing in the Ice run time by setting the configuration property `Ice.Trace.Protocol=1` to discover the object identity and operation name of all requests.
   Of course, the default log output channels (standard out and standard error) will probably be lost if the server is activated automatically, so either start the server manually (see below) or redirect the log output. You can also use the `Ice::Logger` interface to emit your own trace messages.
3. Run the server in a debugger; a server configured for automatic activation can also be started manually if necessary. However, since the IceGrid node did not activate the server, it cannot monitor the server process and therefore will not know when the server terminates. This will prevent subsequent activation unless you clean up the IceGrid state when you have finished debugging and terminated the server. You can do this by starting the server using `icegridadmin`:

```
>>> server start TheServer
```

This will cause the node to activate (and therefore monitor) the server process. If you do not want to leave the server running, you can stop it with the `server stop` command.

4. After the server is activated and is in a quiescent state, attach your debugger to the running server process. This avoids the issues associated with starting the server manually (as described in the previous step), but does not provide as much flexibility in customizing the server's startup environment.

Another cause for a server to fail to activate correctly is if there is a mismatch in the adapter identifiers used by the server for its adapters, and the adapter identifiers specified in the server's deployment descriptor. After starting a server process, the node waits for the server to activate all of its object adapters and report them as ready; if the server does not do this, the node reports a failure once a timeout expires. The timeout is controlled by the setting of the property `IceGrid.Node.WaitTime`. (The default value is 60 seconds.)

You can check the status of each of a server's adapters using `icegridadmin` or the GUI tool. While the node waits for an adapter to be activated by the server, it reports the status of the adapter as "activating". If you experience timeouts before each adapter's status changes to "active", the most likely cause is that the deployment descriptor for the server either mentions more object adapters than are actually created by the server, or that the server uses an identifier for one or more adapters that does not match the corresponding identifier in the deployment descriptor.

Disabling Faulty Servers

You may find it necessary to disable a server that terminates in an error condition. For example, on a Unix platform each server failure might result in the creation of a new (and potentially quite large) core file. This problem is exacerbated when the server is used frequently, in which case repeated cycles of activation and failure can consume a great deal of disk space and threaten the viability of the application as a whole.

As a defensive measure, you can configure an IceGrid node to disable these servers automatically using the `IceGrid.Node.DisableOnFailure` property. In the disabled state, a server cannot be activated on demand. The default value of the property is zero, meaning the node does not disable a server that terminates improperly. A positive value causes the node to temporarily disable a faulty server, with the value representing the number of seconds the server should remain disabled. If the property has a negative value, the server is disabled indefinitely, or until the server is explicitly enabled or started via an administrative action.

You can also manually disable a server at any time using an administrative tool. A manually disabled server remains disabled indefinitely until an administrator enables or starts it. Disabling an *active* server has no effect on the server process; the server is unaware of the change to its status and continues to service requests from connected clients as usual. However, as of Ice 3.5, disabling a server does prevent IceGrid from including the endpoints of the server's object adapters in any subsequent locate requests, and it excludes those object adapters from any replica groups in which they might participate.

Typically, the ultimate goal of disabling a server is to gracefully migrate clients from the faulty server to ones that are behaving correctly. For a client that starts after the server is disabled, migration occurs immediately: the Ice run time in the client issues a locate request on the first proxy invocation, and the result returned by IceGrid excludes any endpoints from the disabled server. For a client that is active at the time the server is disabled, it does not migrate to a different server until its Ice run time issues a new locate request and obtains a new set of endpoints for the target object. The timing of this new locate request depends on several factors:

- If the client has an existing connection to the server, that connection will remain open and active as determined by the configuration settings of the client and server. For example, active connection management could eventually cause the connection to be closed automatically from either end due to inactivity. A subsequent proxy invocation may result in a new locate request.
- The client's use of connection caching also plays an important role. If the client's proxy is configured to cache connections, the client may continue to use the disabled server indefinitely. Connection caching should be disabled to ensure that migration eventually takes place.
- The Ice run time in the client also caches the results of locate requests. Although by default these results do not expire, setting a cache timeout allows you to specify how frequently the Ice run time issues new locate requests. Consider this code:

  C++

  ```
  proxy = proxy->ice_connectionCached(false)->ice_locatorCacheTimeout(20);
  ```

  By disabling connection caching and setting a locator cache timeout, we can ensure that migration occurs within twenty seconds for invocations on this proxy.

See Also

- IceGrid Server Activation
- Locator Semantics for Clients
- icegridadmin Command Line Tool