# Upgrading your Application from Ice 3.5

The subsections below provide additional information about upgrading to Ice 3.6, including administrative procedures for the supported platforms.

On this page:



## Timeout changes

In previous Ice versions, the timeout set for a connection also served as the timeout for all invocations on that connection, such that an invocation timeout would cause Ice to close the connection and consequently report a timeout exception for all pending invocations on that connection.

With the addition of invocation timeouts, Ice now provides a much cleaner separation between two distinct features:

- Connection timeouts
  These timeouts should be used as a fail-safe strategy for handling unrecoverable network errors in a timely fashion.

- Invocation timeouts
  You can now safely abort an invocation that's taking too long to complete without affecting other invocations pending on the same connection.

Together with the new heartbeat functionality offered by Ice's Active Connection Management facility, applications now have much more control over their connections.

Existing Ice applications could configure connection timeouts in several ways, such as by setting Ice.Override.Timeout, or with the `-t` option in endpoints, or by calling `ice_timeout` on proxies. Ice still supports all of these options, and they all configure connection timeouts just like in previous Ice versions. However, you should carefully review these settings to see that they match the true purpose of connection timeouts. Specifically, connection timeouts should normally be chosen based on the speed of the network on which a connection takes place; a small timeout value is fine for a relatively fast network, but a larger value may be necessary for slower connections. Ice enforces connection timeouts when performing network operations such as connection establishment, reading, writing, and closing; setting a timeout allows Ice (and therefore the application) to detect a network problem within a well-defined time period without waiting for low-level network protocols to detect and report the issue.

We recommend that every application enable connection timeouts, and as a result they are now enabled by default as defined by the new property `Ice.Default.Timeout`.

The timeout semantics in previous Ice versions made it difficult for developers to employ invocation timeouts correctly. Using a small timeout value in order to detect network issues reasonably soon risked the possibility that some invocations might time out prematurely. These conflicting goals may have caused developers to modify their application designs to avoid long-running invocations. With the introduction of a separate invocation timeout feature, it's now possible to set connection timeouts appropriately for network errors while using invocation timeouts only for those operations that require them. Unlike connection timeouts, where setting different timeout values causes Ice to open separate connections, invocation timeouts of various values can be freely mixed on the same connection.

The default invocation timeout in Ice 3.6 is "infinite", meaning invocations do not time out by default. You can change the default setting using the new property `Ice.Default.InvocationTimeout`. For individual proxies, the method `ice_invocationTimeout` returns a new proxy configured with the desired invocation timeout. If you need to continue using the timeout semantics of previous Ice versions, set `Ice.Default.InvocationTimeout` to `-2` or call `ice_invocationTimeout` with the value `-2` on a given proxy.

Note that retry semantics differ between connection and invocation timeouts. Ice still performs automatic retries if possible for connection timeouts, but does not retry invocations that fail due to an invocation timeout.

# ACM changes

The Active Connection Management (ACM) facility now offers additional control over its behavior, along with a new automatic heartbeat feature. Client-side ACM was enabled by default in prior Ice versions, which means an existing Ice application most likely uses ACM unless the application explicitly disabled it.

There are several ACM changes that may affect an existing application:

- The `Ice.ACM.Client` and `Ice.ACM.Server` properties have been deprecated and replaced by `Ice.ACM.Client.Timeout` and `Ice.ACM.Server.Timeout`, respectively.
- With the new heartbeat feature, Ice automatically sends heartbeat messages at regular intervals. You can use this feature in place of a dedicated background thread that was commonly used in previous versions of Ice for keeping a session or a bidirectional connection active.
- It's no longer necessary to disable ACM in Glacier2 clients as long as you enable ACM heartbeats.
- Server-side ACM is now enabled by default. Like with the previous versions, the default server-side configuration doesn't close idle connections. However, it now enables heartbeats while incoming requests are pending. This ensures the client doesn't close the connection prematurely while there are long invocations pending.

# IceGrid and Glacier2 sessions

The ACM changes in Ice 3.6 offer new implementation strategies for applications that create Glacier2 or IceGrid sessions. Since a session is tightly bound to a connection, we strongly recommended in prior releases that you disable ACM altogether to avoid the risk of ACM prematurely closing a connection and consequently terminating a session. As of Ice 3.6, it's no longer necessary to disable ACM in these situations. Furthermore, the addition of ACM heartbeats means you can remove existing code that creates a background thread just to keep a session alive.

The `Ice::Connection` interface provides several new operations, including the ability to obtain and modify a connection's current ACM settings. If you use the Glacier2 helper classes, they call the `Connection` operations to tailor the ACM timeout and heartbeat based on the router's ACM configuration. Applications that manually create a Glacier2 session can configure ACM like this:

**C++**

```
Glacier2::RouterPrx router = ...
// Create a session...
int acmTimeout = router->getACMTimeout();
if(acmTimeout > 0)
{
    Ice::ConnectionPtr connection = router->ice_getCachedConnection();
    connection->setACM(acmTimeout, IceUtil::None, Ice::HeartbeatAlways);
}
```

The new operation `Glacier2::Router::getACMTimeout` returns the router's server-side ACM timeout. In this example, the client calls `setACM` on its connection to the router, passing this same timeout value to ensure consistency between client and server. The client also enables automatic heartbeats so that the connection remains active and prevents the router's server-side ACM from closing the connection.

The ACM improvements include changes to the ACM configuration properties. You can use these properties to achieve the same result as the code above, however the properties can potentially affect other connections as well.

Finally, another new operation on the `Connection` interface lets you specify a callback object that will be notified when the connection receives a heartbeat message, and when the connection closes. This feature can be especially useful for session-based applications that need to closely monitor their connections.

# Glacier2 compatibility for helper classes

The Glacier2 helper classes in Ice 3.6 now depend on the ACM heartbeat features described above to keep a session alive. This functionality requires a Glacier2 router that also uses Ice 3.6 or later. If you're upgrading a client to Ice 3.6, we strongly recommend upgrading the Glacier2 router to 3.6 as well. Using an earlier version of Glacier2 will require your application to manually keep the session alive.

# SSL changes

IceSSL for C++ has been overhauled to make use of platform-native SSL APIs where possible:

- IceSSL on Windows now uses SChannel
- IceSSL on OS X now uses Secure Transport
- Linux platforms continue to use OpenSSL as in previous releases

As a result, there have been a number of changes to the IceSSL configuration properties and its C++ API. In the IceSSL property reference, you'll see platform differences marked with **C++ using Windows**, **C++ using OS X**, and **C++ using OpenSSL**, respectively.

We discuss the affected platforms below, along with other general SSL changes.

These changes also affect applications that use Ice for Python, Ice for Ruby, and Ice for PHP because these language mappings are based on Ice for C++.

## SSLv3 disabled by default

To improve security, IceSSL now disables the SSLv3 protocol by default. In other words, only TLS protocols are enabled by default.

Although we do not recommend it, you can enable SSLv3 using the following settings:

```
# Enable only SSLv3
IceSSL.Protocols=SSL3

# Enable SSLv3 and TLS
IceSSL.Protocols=SSL3, TLS1_0, TLS1_1, TLS1_2

# OS X only: Enables SSLv3 and TLS
IceSSL.ProtocolVersionMin = "SSLv3"
```

Refer to IceSSL.* for more information on these settings.

## Certificate verification

The default value of the `IceSSL.VerifyDepthMax` property is now three (it was previously two). This allows certificate chains of three certificates, such as a chain consisting of a peer certificate, a CA certificate, and a Root CA certificate.

The certificate chain provided in `IceSSL::ConnectionInfo` should now always include the root certificate if the chain is successfully verified.

We also added a `verified` member to `IceSSL::ConnectionInfo`, which indicates whether or not the peer certificate was successfully verified. For a client that sets `IceSSL.VerifyPeer=0`, this member allows the client to check the verification status of the server's certificate. For server connections, the member should always be true since servers always reject invalid client certificates.

## SSL changes on OS X

By using Secure Transport, IceSSL on OS X has now become very similar to IceSSL in Ice Touch. For example, you can use OS X keychains, take advantage of the system's default graphical password prompt, and use many of the same configuration properties.

### General Changes

- Password prompt
  In previous releases, OpenSSL would attempt to prompt command-line users for a password if the application failed to define a password or supply a password callback. Now OS X will use its default graphical password prompt in this situation.
- DSA certificates
  DSA certificates are no longer supported on OS X. If you used DSA certificates, you will need to generate RSA equivalents. The `IceSSL.CertFile` property still accepts the same syntax in that it allows you to specify two files (one RSA certificate file and one DSA certificate file), but it will ignore the DSA certificate.
- MD5 signatures
  OS X does not support certificates with MD5 signatures. We recommend using SHA256 instead.

### API Changes

- Since IceSSL on OS X no longer uses OpenSSL, the native C++ class `IceSSL::Plugin` does not support the `setContext` and `getContext` methods.
- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

### Property Changes

- Keychains
  The properties `IceSSL.Keychain` and `IceSSL.KeychainPassword` are now supported on OS X.
- Certificate authorities
  Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.
- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.
- Certificate queries
  The new property `IceSSL.FindCert` lets you query a keychain to find a certificate matching certain criteria.
- Diffie Hellman parameters
  The new property `IceSSL.DHParams` allows you to specify the name of a file containing pre-generated Diffie Hellman parameters. The property `IceSSL.DH.`*`bits`* is no longer supported on OS X.
- Protocol limits
  The `IceSSL.Protocols` property is no longer supported on OS X for specifying the versions of SSL/TLS that a program will accept. Use the new properties `IceSSL.ProtocolVersionMin` and `IceSSL.ProtocolVersionMax` instead.

Refer to deprecated APIs for information about obsolete properties.

## SSL changes on Windows

By using SChannel, IceSSL on Windows has now become very similar to IceSSL for .NET. For example, you can use certificate stores and many of the same configuration properties.

### General Changes

- Password prompt
  In previous releases, OpenSSL would attempt to prompt command-line users for a password if the application failed to define a password or supply a password callback. Windows does not have a default password prompt, so this situation will now result in a run-time exception.
- Anonymous Diffie Hellman (ADH)
  ADH ciphers are no longer supported on Windows.
- Certificate formats
  Windows is able to load a certificate in PEM format if no password is required. Use the PFX (PKCS#12) format for password-protected certificates and keys.

### API Changes

- Since IceSSL on Windows no longer uses OpenSSL, the native C++ class `IceSSL::Plugin` does not support the `setContext` and `getContext` methods.
- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

### Property Changes

- Certificate authorities
  Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.
- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.
- Certificate queries
  The new property `IceSSL.FindCert` lets you query a certificate store to find a certificate matching certain criteria.

Refer to deprecated APIs for information about obsolete properties.

## SSL changes on Linux

### API Changes

- The method `IceSSL::Certificate::verify(const PublicKeyPtr&)` has been deprecated. The new method `IceSSL::Certificate::verify(const CertificatePtr&)` takes its place.

### Property Changes

- Certificate authorities
  Use the new property `IceSSL.CAs` to denote a file containing the certificates of your trusted certificate authorities. If you'd rather use the system's default certificate authorities, enable `IceSSL.UsePlatformCAs` instead.
- Certificates
  Use the `IceSSL.CertFile` property to denote a PKCS12 file containing both a certificate and a private key. The `IceSSL.KeyFile` property is now deprecated and should no longer be used to denote a separate private key file.

Refer to deprecated APIs for information about obsolete properties.

# Collocated Invocation changes

Ice has always supported location transparency, where a proxy invocation should ideally have the same semantics regardless of whether the target object is on a different host, a different process on the same host, or collocated in the current process. For this latter case, a collocated invocation is defined as a proxy invocation on a target object that is hosted by an object adapter in the same process and created using the same communicator as the proxy. In previous versions of Ice, the semantics of collocated invocations differed in several ways from regular "remote" invocations:

- Classes and exceptions were never sliced. Instead, the receiver always received a class or exception as the derived type that was sent by the sender.

- If a collocated invocation threw an exception that was not in an operation's exception specification, the original exception was raised in the client instead of `UnknownUserException`. (This applied to the C++ mapping only.)
- Class factories were ignored.
- Invocation timeouts were ignored.
- If an operation implementation used an in parameter that was passed by reference as a temporary variable, the change affected the value of the in parameter in the caller (instead of modifying a temporary copy of the parameter on the callee side only).
- Asynchronous semantics were not supported for collocated invocations.

Ice 3.6 eliminates all of these differences. Most notably, you can now use the asynchronous invocation API on a collocated servant, and collocated invocations are now supported in Python.

Note however that a few differences in semantics still remain:

- Most collocated invocations are dispatched in the server-side thread pool just like regular invocations; the only exceptions are synchronous twoway collocated invocations with no invocation timeout, which are dispatched in the calling thread.
- The state of the servant's object adapter is ignored: collocated invocations proceed normally even if the servant's adapter is not yet activated or is in the holding state.
- AMI callbacks for asynchronous collocated invocations are dispatched from the servant's thread and not from the client-side thread pool unless AMD is used for the servant dispatch. In this case, the AMI callback is called from the client-side thread pool.
- Invocation timeouts work as usual, but connection timeouts are ignored.

If your application relies on collocated invocations, test it carefully with Ice 3.6 to ensure that it still behaves as expected.

# Batch Invocation changes

We made some significant changes to the way batch invocations work in Ice 3.6. The following table compares the behavior of batch invocations between previous releases and Ice 3.6:

| **Batch invocation behavior in Ice 3.5 and earlier** | **Batch invocation behavior in Ice 3.6** |
|---|---|
| A batch request is queued by the connection associated with the proxy. | A batch request is queued by the proxy used for the invocation. The only exception is for a fixed proxy , in which case batch requests continue to be queued by the connection associated with the proxy. |
| If a proxy was not already associated with a connection, the initial batch request could trigger a connection attempt. | A batch request does not trigger any network activity. |
| Calling `Communicator::flushBatchRequests` flushes all queued requests for all connections. | Calling `Communicator::flushBatchRequests` only flushes queued requests invoked using fixed proxies. |
| Calling `Connection::flushBatchRequests` flushes queued requests for all proxies associated with that connection. | Calling `Connection::flushBatchRequests` only flushes queued requests invoked using fixed proxies associated with that connection. |
| A batch of requests is compressed when at least one of the proxies used to create the requests in this batch has compression enabled. | A batch of requests is compressed when the proxy used to flush this batch has compression enabled. Batched requests flushed through `Communicator::flushBatchRequests` or `Connection::flushBatchRequest` never use compression. |
| Batch requests are not affected by proxy lifecycles because the requests are queued by connections. | Batch requests queued by a proxy are discarded when the proxy is deallocated. |
| Batch requests could be silently lost if an error occurred during a flush. | Calling `ice_flushBatchRequests` on a regular (non-fixed) proxy behaves like a oneway request: failures that occur during network activity trigger automatic retries and can eventually raise an exception. Furthermore, the `sent` callback is invoked for asynchronous calls to `ice_flushBatchRequests`. |

With these changes, Ice's batch invocation facility has become more reliable and behaves more consistently with other invocation modes.

To give you more control over batch requests, we've also added a new `BatchInterceptor` API. You can configure a communicator with a custom batch interceptor in order to implement your own auto-flush algorithms or to receive notification when an auto-flush fails.

# Logger changes

We added a new operation `getPrefix` to the local interface `Logger`.

**Slice**

```
module Ice {
local interface Logger {
...
     string getPrefix();
};
};
```

If you implement your own logger, you will need to update your implementation with a new `getPrefix`. `getPrefix` returns the prefix associated with this logger.

# Crypt Password changes

Both the Glacier2 router and IceGrid registry provide a simple file-based authentication mechanism using a "crypt password" file that contains a list of user name and password-hash pairs. In Ice 3.5 and earlier releases, the Glacier2 router and IceGrid registry use the archaic DES-based Unix Crypt al gorithm to hash the provided password and verify if this hash matches the hash in the password-file.

Ice 3.6 no longer supports this hash format on Windows and OS X. As a result, you need to regenerate your crypt password files on these platforms when upgrading to Ice 3.6.

# C++ changes

## Garbage collection changes

We've made significant changes to the garbage collection facility for cyclic object graphs. If your application uses this facility, note that we've removed the following:

- `Ice.GC.Interval` property
- `Ice.Trace.GC` property
- `Ice::collectGarbage()` function

The new property `Ice.CollectObjects` determines whether garbage collection is enabled by default for Slice class instances that are unmarshaled by the Ice run time. Refer to the garbage collection discussion for more information on using this feature.

## String converter changes

A number of changes have been made to the C++ string conversion API in this release:

- The `stringConverter` and `wstringConverter` members of `Ice::InitializationData` have been removed. Applications should use the process-wide string converter API instead.
- The classes and functions have moved from the `Ice` namespace to the `IceUtil` namespace.
- Overloaded versions of the `nativeToUTF8` and `UTF8ToNative` convenience functions that accepted a communicator argument have been removed.
- The arguments have changed for the `stringToWstring` and `wstringToString` convenience functions.

## OS X with C++

C++ developers on OS X need to be aware of several changes:

- C++11 libraries
  With Ice 3.5, C++11 applications needed to link with a separate set of C++11-specific Ice libraries located in `<Ice installation directory>/lib/c++11`. The Ice 3.6 binary distribution includes a single set of libraries that also support C++11, so you'll need to modify your application's library path to use `<Ice installation directory>/lib` instead.

- Minimum required version
  Ice 3.6 supports OS X 10.9 and 10.10, therefore the C++ libraries in the Ice binary distribution are built with `macosx-min-version=10.9`. Consequently, these libraries require `libc++` and are not compatible with `libstdc++`.

# Java changes

## Java mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string, enumerator data members to the first enumerator in the enumeration, and structure data members to a default-constructed value.

In previous releases, the default constructor initialized these data members to null. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The Java mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure or enumerator prior to an invocation. In Ice 3.6, passing a null value for a structure or enumerator causes Ice to marshal a default-constructed structure or the first enumerator, respectively.

## New Java features

We have added several features that you may wish to incorporate into your Java application:

- Interrupts
- Buffers
- Java 8 and lambdas

# C# changes

## C# mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string and structure data members to a default-constructed value.

In previous releases, the default constructor did not explicitly initialize these data members and so they had the default C# values. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The C# mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure prior to an invocation. In Ice 3.6, passing a null value for a structure causes Ice to marshal a default-constructed structure.

## C# serialization changes

One of the new features in Ice 3.6 is support for .NET serialization for all Slice types (except proxies). Existing applications may be affected by this change because the `Ice.Optional` type now implements `ISerializable` and the serialized format of an optional value is different than in Ice 3.5.

# Python changes

Developers who are migrating existing Ice applications from Ice 3.5 should be aware of a change that affects the Python language mapping. The `Ice.Unset` value now has `False` semantics, making it more convenient to test whether an optional Slice data member has a value:

**Python**

```
# Only valid with Ice 3.6!
if obj.optionalMember:
    # optionalMember has a value
```

With Ice 3.5, the code above would not have the intended behavior because the test would be true even when `optionalMember` is set to `Ice.Unset`. The correct way to write this using Ice 3.5 is shown below:

**Python**

```
# Correct test with Ice 3.5
if obj.optionalMember is not Ice.Unset:
    # optionalMember has a value
```

This code will also have the correct behavior with Ice 3.6, but the new style is easier to read. Also note that the Ice 3.6 semantics mean you need to use caution for optional values that can legally be set to `None`:

**Python**

```
if obj.optionalMember: # Fails for None AND Ice.Unset!
    # optionalMember is not Ice.Unset or None
```

You can distinguish between `Ice.Unset` and `None` as follows:

**Python**

```
if obj.optionalMember is Ice.Unset:
    # optionalMember is unset
elif obj.optionalMember is None:
    # optionalMember is set to None
else:
    # optionalMember is set to a value other than None
```

We recommend that you review for correctness all uses of `Ice.Unset` and tests of optional data members.

# PHP changes

No changes to the PHP mapping in Ice 3.6 affect compatibility for existing applications.

# Ruby changes

No changes to the Ruby mapping in Ice 3.6 affect compatibility for existing applications.

# JavaScript changes

## JavaScript mapping changes

The default constructor generated for Slice structures, exceptions and classes behaves differently for Ice 3.6 than in previous releases. Specifically, the default constructor now initializes string data members to an empty string and structure data members to a default-constructed value.

In previous releases, the default constructor initialized these data members to null. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The JavaScript mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure or enumerator prior to an invocation. In Ice 3.6, passing a null value for a structure or enumerator causes Ice to marshal a default-constructed structure or the first enumerator, respectively.

## JavaScript packaging changes

The NodeJS packaging has changed from the original Ice for JavaScript 0.1 release, meaning existing JavaScript applications will need to modify their `require` statements. All of the top-level Ice modules (Ice, Glacier2, etc.) are now accessible by including the `ice` package:

**JavaScript**

```
var Ice = require("ice").Ice;
var Glacier2 = require("ice").Glacier2;
// ...
```

Loading the generated code for your own Slice definitions looks similar. Suppose we have the following definitions in `Hello.ice`:

**Slice**

```
module Demo {
interface Hello {
    idempotent void sayHello(int delay);
    void shutdown();
};
};
```

To make the `Demo` module conveniently accessible in our code, we can write:

**JavaScript**

```
var Demo = require("Hello").Demo;
var proxy = Demo.HelloPrx.uncheckedCast(...);
```

Nothing has changed for browser-based JavaScript applications, where loading `Ice.js` adds the Ice definitions to the global window object, and other Ice modules (`Glacier2.js`, etc.) must be loaded individually.

Note that Ice 3.6 adds support for the WebSocket transport to the Ice core, and includes new implementations of the WebSocket transport in Java and C#. This means you no longer need to use Glacier2 as an intermediary if your JavaScript client needs to communicate with a Java or C# server.

# Objective-C changes

The default `init` method and convenience constructor generated for Slice structures, exceptions and classes behave differently for Ice 3.6 than in previous releases. Specifically, these methods now initialize string data members to an empty string, enumerator data members to the first enumerator, and structure data members to a default-constructed value.

In previous releases, the default constructor zero-initialized these data members. Applications that depend on this behavior will require updating.

For situations where the overhead of default-constructing structure data members is undesirable, applications can call the one-shot constructor instead.

The Objective-C mapping has also relaxed its marshaling prerequisites: it is no longer necessary to initialize data members of type structure prior to an invocation. In Ice 3.6, passing a null value for a structure causes Ice to marshal a default-constructed structure.

# Ubuntu packages

Users of Ice 3.5 on Ubuntu had the choice of using Debian's packages or ZeroC's own experimental packages. We called them "experimental" because we expected we might eventually change the packaging structure, and in fact we have changed the structure for Ice 3.6.

Upgrading an existing installation of the ZeroC packages for Ice 3.5 on Ubuntu to Ice 3.6 is relatively straightforward. First add the Ice repository to the system and update the package list:

```
$ sudo apt-add-repository "deb http://zeroc.com/download/Ice/3.6/ubuntu14.04 stable main"
```

```
$ sudo apt-get update
```

To upgrade all of the run-time packages:

```
$ sudo apt-get install zeroc-ice-all-runtime
```

To upgrade all of the development packages:

```
$ sudo apt-get install libzeroc-ice-dev libzeroc-ice-java zeroc-ice-all-dev
```

Refer to our binary distribution page for details on the individual packages.

# Migrating IceGrid databases from Ice 3.5

Ice 3.6 supports the migration of IceGrid databases from Ice 3.3, 3.4 and 3.5. To migrate from earlier Ice versions, you will first need to migrate the databases to the Ice 3.3 format. If you require assistance with such migration, please contact support@zeroc.com.

To migrate, first stop the IceGrid registry you wish to upgrade.

Next, copy the IceGrid database environment to a second location:

```
$ cp -r db recovered.db
```

Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.3, use `db_recover` from Berkeley DB 4.6. For Ice 3.4, use `db_recover` from Berkeley DB 4.8. For Ice 3.5, use `db_recover` from Berkeley DB 5.3. You can verify the version of your `db_recover` tool by running it with the `-V` option:

```
$ db_recover -V
```

Now run the utility on your copy of the database environment:

```
$ db_recover -h recovered.db
```

Change to the location where you will store the database environments for IceGrid 3.6:

```
$ cd <new-location>
```

Next, run the `upgradeicegrid36.py` utility located in the `config` directory of your Ice distribution (or in `/usr/share/Ice-3.6` if using an RPM installation). The first argument is the path to the old database environment. The second argument is the path to the new database environment.

In this example we'll create a new directory `db` in which to store the migrated database environment:

```
$ mkdir db
$ upgradeicegrid36.py <path-to-recovered.db> db
```

Upon completion, the `db` directory contains the migrated IceGrid databases.

By default, the migration utility assumes that the servers deployed with IceGrid also use Ice 3.6. If your servers still use an older Ice version, you need to specify the `--server-version` command-line option when running `upgradeicegrid36.py`:

```
$ upgradeicegrid36.py --server-version 3.5.1 <path-to-recovered.db> db
```

The migration utility will set the `server` descriptor attribute `ice-version` to the specified version and the IceGrid registry will generate configuration files compatible with the given version.

If you are upgrading the master IceGrid registry in a replicated environment and the slaves are still running, you should first restart the master registry in read-only mode using the `--readonly` option, for example:

```
$ icegridregistry --Ice.Config=config.master --readonly
```

Next, you can connect to the master registry with `icegridadmin` or the IceGrid administrative GUI from Ice 3.6 to ensure that the database is correct. If everything looks fine, you can shutdown and restart the master registry without the `--readonly` option.

IceGrid slaves from Ice 3.3, 3.4 or 3.5 won't interoperate with the IceGrid 3.6 master. You can leave them running during the upgrade of the master to not interrupt your applications. Once the master upgrade is done, you should upgrade the IceGrid slaves to Ice 3.6 using the instructions above.

# Migrating IceStorm databases from Ice 3.5

No changes were made to the database schema for IceStorm in this release. Furthermore, Ice 3.5 and Ice 3.6 use the same version of Berkeley DB (Berkeley DB 5.3.x). You can use IceStorm databases created with Ice 3.5 with Ice 3.6 without any transformation.

# Migrating Freeze databases from Ice 3.5

No changes were made that would affect the content of your Freeze databases. Furthermore, Ice 3.5 and Ice 3.6 use the same version of Berkeley DB (Berkeley DB 5.3.x). You can use Freeze databases created with Ice 3.5 with Ice 3.6 without any transformation.

# Migrating Android applications from Ice 3.5

Our recommended development environment for Android applications is Android Studio. Refer to the *Using the Binary Distribution* page appropriate for your platform for instructions on configuring a project in Android Studio.

# Changed APIs

This section describes APIs whose semantics have changed, potentially in ways that are incompatible with previous releases.

The following APIs were changed in Ice 3.6:

- String conversion in C++
  Several changes were made to the string conversion API in C++.

- IceSSL
  The native IceSSL APIs changed on some platforms.

- C++ garbage collection
  The `Ice::collectGarbage` function was removed and replaced by a new garbage collection facility.

# Removed APIs

This section generally describes APIs that were deprecated in a previous release and have now been removed. Your application may no longer compile or operate successfully if it relies on one of these APIs.

The following APIs were removed in Ice 3.6:

- Deprecated API for asynchronous method invocations (AMI)
  This API, which uses proxy operations such as `sayHello_async`, was deprecated in Ice 3.4 and is no longer available. The new API should be used instead. Refer to the appropriate language mapping section for more information.

- `Stats` facility
  This functionality was deprecated in Ice 3.5 and is now provided by the Instrumentation facility and the Metrics facet.

- `Ice.GC.Interval`
  `Ice.Trace.GC`
  `Ice::collectGarbage()`
  Significant changes were made to the C++ garbage collection facility.

- `Ice.MonitorConnections`
  This setting is no longer necessary.

- `IceSSL::Plugin::setContext()`
  `IceSSL::Plugin::getContext()`
  These C++ methods are no longer available on Windows or OS X.

- `Ice::InitializationData::stringConverter`
  `Ice::InitializationData::wstringConverter`
  These C++ data members are no longer available. Use `IceUtil::setProcessStringConverter` and `IceUtil::setProcessWstringConverter` instead.

- `Ice::Router::addProxy()`
  `IceGrid::Admin::writeMessage()`
  `IceStorm::Topic::subscribe()`
  These Slice operations were deprecated in previous Ice releases.

- `IceUtil.Version`
  This Java class was deprecated in previous Ice releases. Use `Ice.Util.stringVersion()` and `Ice.Util.intVersion()` instead.

- `Ice::Object::ice_getHash()`
  This C++ method was deprecated in Ice 3.5.

- `IcePatch2.ChunkSize`
  `IcePatch2.Directory`
  `IcePatch2.Remove`
  `IcePatch2.Thorough`
  These properties were deprecated in previous Ice releases and replaced by properties that use the prefix `IcePatch2Client`.

- `Glacier2.AddSSLContext`
  This property was deprecated in Ice 3.3.1 and replaced by `Glacier2.AddConnectionContext`.

The following components were removed in Ice 3.6:

- Qt SQL database plug-ins for IceGrid and IceStorm, which were deprecated in Ice 3.5.
  Freeze is now the only persistence mechanism for these services.

# Deprecated APIs

This section discusses APIs and components that are now deprecated. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

The following APIs were deprecated in Ice 3.6:

- `Ice.ACM.Client`
  `Ice.ACM.Server`
  Use `Ice.ACM.Client.Timeout` and `Ice.ACM.Server.Timeout` instead. See Active Connection Management for more information.

- `IceSSL.CertAuthFile`
  Use the new property `IceSSL.CAs` to specify the path name of a PEM file containing the Root Certificate Authorities.

- `IceSSL.CertAuthDir` (OpenSSL only)
  Use the new property `IceSSL.CAs` to specify the path name of a directory containing the Root Certificate Authorities.

- `IceSSL.KeyFile`
  Use `IceSSL.CertFile` to configure the IceSSL identity using a PKCS12 file.

- `IceSSL.ImportCert.*` (.NET only)
  This property caused the IceSSL plug-in to import a certificate into a Windows certificate store. Going forward, users will need to install certificates in a store using Windows-provided tools if necessary. However, it is now possible to use trusted CA certificates from a file without loading them into a store. See `IceSSL.CAs` for more information.

- `IceSSL.PersistKeySet` (.NET only)
  Only used by the now-deprecated `IceSSL.ImportCert` property.

- `IceSSL.KeySet` (.NET only)
  Use `IceSSL.CertStoreLocation` instead.

- `IceSSL::Certificate::verify` (C++ only)
  The method `verify(const PublicKeyPtr&)` has been deprecated. The new method `verify(const CertificatePtr&)` takes its place.

- `IceBox.InstanceName`
  `IceBox.ServiceManager.`*`AdapterProperty`*
  These properties are no longer necessary because their functionality is provided by the service manager administrative facet.

- `clr:collection`
  `Ice.CollectionBase`
  `Ice.DictionaryBase`
  The `clr:collection` metadata tag, along with the C# classes `Ice.CollectionBase` and `Ice.DictionaryBase`, were originally provided for backward compatibility with Ice versions prior to 3.3 and are now deprecated. Existing applications should migrate their code to use the standard C# collection types.

- `Ice::StringConverterPlugin` (C++ only)
  The `StringConvererPlugin` base class is deprecated. You should use a regular `Ice::Plugin` to install your string converter(s).

- `Ice::CollocatedOptimizationException`
  This exception is no longer used now that collocated invocations have become much more flexible.

- `Ice.BatchAutoFlush`
  This property controlled whether the Ice run time would automatically flush batch requests for a connection after enough requests had been queued to reach the limit established by `Ice.MessageSizeMax`. We're deprecating this property in favor of a new one, `Ice.BatchAutoFlushSize`, whose value Ice now uses as the limit for automatic flushing. If your application sets `Ice.BatchAutoFlush=0` to disable automatic flushing, you can achieve the same behavior by setting `Ice.BatchAutoFlushSize=0`.

# Visual C++ compiler warnings

The Ice 3.5 header files downgrade the following Visual C++ warnings to level 4:

**C++**

```
#   pragma warning( 4 : 4250 ) // ... : inherits ... via dominance
#   pragma warning( 4 : 4251 ) // class ... needs to have dll-interface to be used by clients of class ...
```

This downgrade affects any file that includes these Ice header files.

Ice 3.6 no longer disables or downgrades any warning in your C++ code. As a result, when upgrading to Ice 3.6, your Ice application may produce compiler warnings that were not reported before. To eliminate these warnings, you can modify your source code, add `pragmas` or disable these warnings in your Visual Studio projects.