

# New Features in Ice 3.6

This page describes notable additions and improvements in Ice 3.6. For a detailed list of the changes in this release, please refer to the [changelog](#) in the source tree. Our [upgrade guide](#) documents the changes that may affect the operation of your applications or have an impact on your source code.

On this page:

- [New Features](#)
  - [JavaScript Mapping and WebSocket Transports](#)
  - [Objective-C Mapping](#)
  - [Multicast Discovery](#)
    - [IceDiscovery](#)
    - [IceLocatorDiscovery](#)
  - [IceGrid Custom Load Balancing](#)
  - [Invocation Timeouts](#)
  - [Invocation Cancellation](#)
  - [Active Connection Management](#)
  - [SSL](#)
- [Other New Features](#)
  - [Remote Logging](#)
  - [Garbage Collection Improvements](#)
  - [Transport Improvements](#)
    - [HTTP Proxy](#)
    - [Source Address](#)
  - [Batch Invocations](#)
  - [Collocated Invocations](#)
  - [Finder Interfaces](#)
  - [Java Interrupts](#)
  - [Java Lambda Functions](#)
  - [Java Buffers](#)
  - [C++ View Types](#)
  - [Linking with C++ Libraries on Windows](#)
  - [C++ Plug-in for Crypt Password Files](#)
  - [Support for -fvisibility=hidden with GCC and clang](#)

## New Features

Ice 3.6 includes a number of significant new features, many of them in response to requests from our commercial users.

### JavaScript Mapping and WebSocket Transports


Ice for JavaScript offers a native and compact implementation of the Ice run time that you can incorporate into your web apps and gain seamless access to back-end Ice services. Our native JavaScript implementation of the Ice run time features a compact footprint, complete support for all Slice data types, compatibility with modern browsers as well as NodeJS, and an intuitive API to get you productive as quickly as possible.

Here's a really simple but complete example to give you an idea of what it's like to make a remote invocation using Ice in JavaScript:

#### JavaScript

```
var communicator = Ice.initialize();
var proxy = communicator.stringToProxy("hello:wss -h www.myhost.com -p 443");
var hello = HelloPrx.uncheckedCast(proxy);
hello.sayHello().then(
    function() { console.log("sayHello done!"); }
).exception(
    function(ex) { console.log("something went wrong!"); }
).finally(
    function() { return communicator.destroy(); }
);
```

Experienced JavaScript developers might recognize our API's use of *promises*, which is a common solution for managing asynchronous activities. Our promise implementation follows existing conventions and offers a tremendous amount of power and convenience for simplifying your application code.

 There have been a few changes to the JavaScript mapping since the preview release, as described in the [upgrade guide](#).

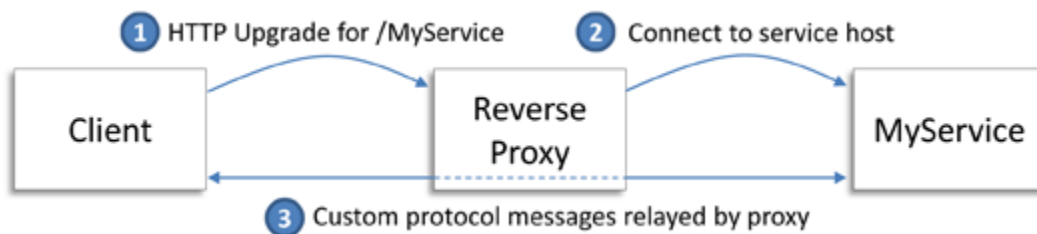
Please refer to the [manual](#) for complete details on the JavaScript mapping.

JavaScript clients running in the browser use the industry-standard WebSocket protocol. They can talk directly to Ice servers listening on WebSocket endpoints. In our example above, the endpoint `wss -h www.myhost.com -p 443` directs the client to use a secure WebSocket connection via the standard HTTPS port. The WebSocket protocol is supported by all Ice language mappings.

The use of WebSockets isn't useful just for JavaScript clients running in the browser -- it can also be used by regular Ice clients as well. The WebSocket protocol can indeed greatly ease network deployment by allowing the use of the standard HTTP or HTTPS port as well as your existing Web infrastructure. For instance, the client can connect to:

- an Ice server listening directly on port 80 (HTTP) or port 443 (HTTPS),
- an HTTP server (e.g., Nginx) that supports reverse proxying of WebSocket connections,
- an HTTP load balancer (e.g., HAProxy) that supports forwarding WebSocket connections.

Here's how it works when using an HTTP server reverse proxy:



The client connects to the HTTP server and sends an HTTP request to upgrade the existing connection to support WebSocket. The message includes a resource path, such as `/MyService` in the example above, that the HTTP server translates into a target back-end service. The HTTP server establishes its own connection to the service and then relays messages between the client and the service. The service is an Ice server listening on a WS or WSS endpoint.

The WebSocket protocol provides an alternative to opening a port on your corporate firewall. You can instead leverage your existing web infrastructure to allow Ice clients on the other side of your firewall to access your Ice services.

## Objective-C Mapping

The Objective-C mapping from Ice Touch is now included in the Ice distribution.

It currently lacks support for the following features:

- Asynchronous method dispatch (AMD)

Please refer to the [manual](#) for complete details on the Objective-C mapping.

## Multicast Discovery

In an effort to simplify bootstrapping and minimize configuration, we're introducing two new components that facilitate the discovery of Ice servers via UDP multicast.

### IceDiscovery

[IceDiscovery](#) provides a lightweight location service that enables clients to discover objects at run time. Implemented as an Ice plug-in, the service requires only minimal configuration and very few changes to your application code. IceDiscovery is an ideal solution for applications that don't need the additional features that [IceGrid](#) offers, such as server activation and centralized administration, but could still take advantage of the benefits provided by indirect proxies and replication.

Consider the following code example:

**Java**

```
Ice.ObjectPrx proxy = communicator.stringToProxy("hello");
HelloPrx hello = HelloPrxHelper.checkedCast(proxy);
hello.sayHello();
```

Notice here that the code translates the stringified proxy "hello" containing no endpoint information. Ice resolves such proxies using a locator supplied by the IceDiscovery plug-in, all we need to do is configure the client to load the plug-in:

```
# Client configuration
Ice.Plugin.IceDiscovery=IceDiscovery:IceDiscovery.PluginFactory
```

At the proxy's first remote invocation (the call to `checkedCast` in the example above), IceDiscovery issues a multicast query to determine if any servers are available that provide the `hello` object. If more than one server responds, it means the object is replicated and the client selects one of the servers based on additional configuration settings. This discovery process is handled internally by the plug-in and is transparent to clients and servers.



UDP multicast is only used by IceDiscovery to locate available servers. Clients and servers communicate using the transport of your choosing.

In practical terms, using the plug-in means a client no longer needs to hard-code the addressing information of its servers in its source code or configuration files. Assuming the underlying network supports multicast, IceDiscovery enables servers to start, stop, and migrate to different hosts without affecting the ability of clients to find them, and without requiring any administrative changes.

## IceLocatorDiscovery

In previous versions of Ice, using IceGrid was the primary way for applications to leverage the benefits of indirect proxies; the only other alternative was to write a custom "locator" service. Indirect proxies help to minimize coupling between clients and servers by eliminating most of the addressing information from application code and configuration files. The one exception has always been the `Ice.Default.Locator` property, which needed to contain at least one endpoint for an IceGrid registry:

```
# Client configuration before IceLocatorDiscovery
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

In a replicated IceGrid deployment consisting of a master replica and one or more slave replicas, the `Ice.Default.Locator` property would normally include endpoints for some or all of the replicas.

The [IceLocatorDiscovery](#) plug-in alleviates the need to statically configure locator endpoints by using UDP multicast to dynamically discover the active locators in a network. Integrating the plug-in simply requires updating client configuration files to install the plug-in and remove the `Ice.Default.Locator` property. No changes are necessary in your application code.

## IceGrid Custom Load Balancing

We've enhanced the IceGrid registry to support [pluggable load balancing strategies](#), making it possible for you to implement application-specific load balancing algorithms or filter the registry's results based on user input.

Two kinds of filters are supported:

- **Replica group filter**  
The registry invokes a replica group filter each time a client requests the endpoints of a replica group or object adapter, as well as for calls to `findAllReplicas`. The registry passes information about the query that the filter can use in its implementation, including the list of object adapters participating in the replica group whose nodes are active at the time of the request. The object adapter list is initially ordered using the load balancing type configured for the replica group; the filter can modify this list however it chooses.
- **Type filter**  
The registry invokes a type filter for each query that a client issues to find a well-known object by type using the operations `findObjectByType`, `findAllObjectsByType`, and `findObjectByTypeOnLeastLoadedNode`. Included in the information passed to the filter is a list of proxies for the matching well-known objects; the filter implementation decides which of these proxies are returned to the client.

Filters are implemented in C++ as regular Ice plug-ins that you install in the IceGrid registry via configuration properties.

## Invocation Timeouts

Timeout semantics have changed for the better with the addition of [invocation timeouts](#). Ice still supports the existing timeout facility, referred to as a connection timeout or endpoint timeout, but now it only affects network activities. The new invocation timeout facility enables you to set an arbitrary timeout per invocation, independent of the connection timeout. Furthermore, when an invocation timeout occurs, it only affects that invocation and the connection remains open. The code below demonstrates the new semantics:

**C++**

```
Ice::ObjectPrx proxy = communicator->stringToProxy(...);
proxy = proxy->ice_timeout(2000);           // Connection timeout: 2sec
proxy = proxy->ice_invocationTimeout(5000); // Invocation timeout: 5sec
try {
    proxy->ice_ping();
} catch (const Ice::InvocationTimeoutException& ex) {
    // server's busy?
} catch (const Ice::TimeoutException& ex) {
    // network problem!
}
```

Review the [upgrade guide](#) to learn how your existing application might be impacted by and benefit from these changes.

## Invocation Cancellation

It's now possible to cancel an asynchronous invocation using the `cancel` method of the `Ice::AsyncResult` object. This method prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. It has no effect on the server. A canceled invocation is considered to be completed and the result of the invocation is an `Ice::InvocationCanceledException`.

## Active Connection Management

We've enhanced the [Active Connection Management](#) facility to give applications more control over ACM behavior. Included in these enhancements is a new "heartbeat" feature that you can use in your applications when you need to ensure that a connection remains open. For example, you can use heartbeats to keep a Glacier2 session alive or prevent a bidirectional connection from being closed prematurely. This feature eliminates the need for an application to implement its own background thread to keep a connection or session alive. We've also added the ability to register a callback with the connection that is notified when the connection closes or receives a heartbeat. The code below shows how to enable heartbeats and install a callback on a connection:

**C++**

```
class CallbackI : public Ice::ConnectionCallback
{
    virtual void heartbeat(const Ice::ConnectionPtr& connection)
    {
        std::cout << "received heartbeat for connection:\n" << connection->toString() << std::endl;
    }

    virtual void closed(const Ice::ConnectionPtr& connection)
    {
        std::cout << "connection closed:\n" << connection->toString() << std::endl;
    }
};

Ice::ObjectPrx proxy = communicator->stringToProxy(...);
Ice::ConnectionPtr connection = proxy->ice_getConnection();
connection->setACM(5, Ice::CloseOnInvocationAndIdle, Ice::HeartbeatAlways);
connection->setCallback(new CallbackI());
```

Refer to our [upgrade guide](#) for more information on how the ACM changes can impact existing Ice applications.

## SSL

We've essentially re-written the IceSSL plug-in to make use of a platform's native SSL APIs where possible. On OS X, the plug-in now uses Apple's Secure Transport facility, and on Windows it now uses SChannel. Linux is the only platform on which IceSSL still uses OpenSSL.

By using native SSL APIs instead of OpenSSL for secure communications on OS X and Windows, your applications can more easily take advantage of any hotfixes provided by the operating system vendors when security issues arise.

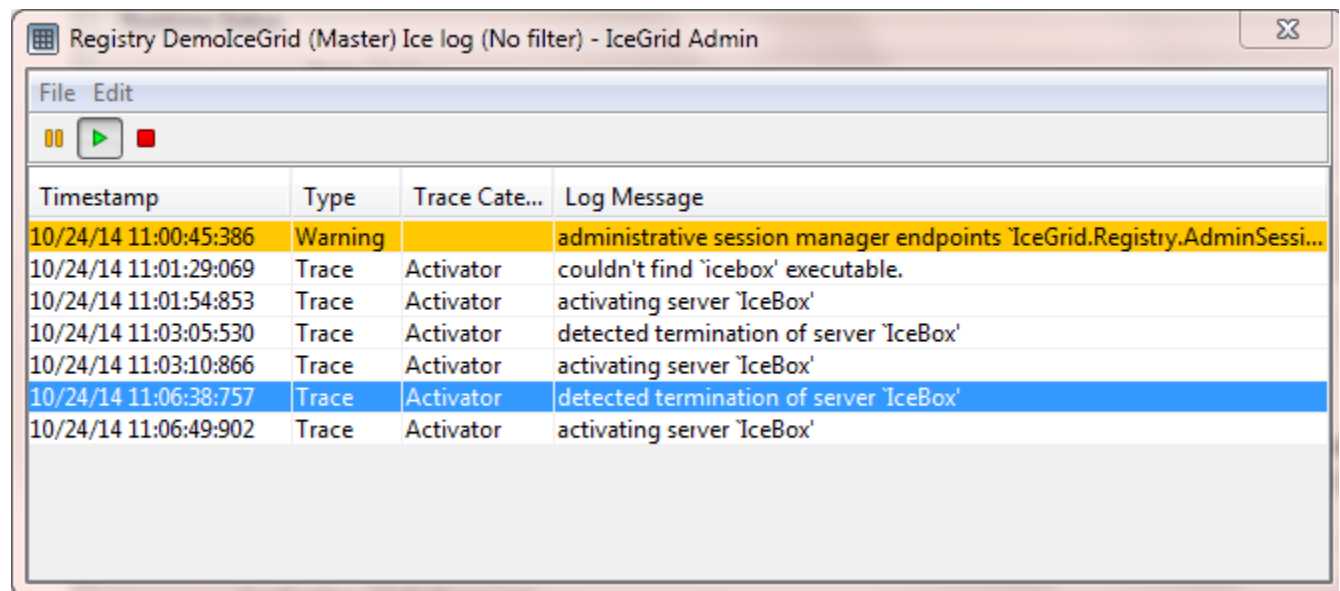
If you use IceSSL, we recommend reviewing the [upgrade guide](#) because there have been some changes to the IceSSL APIs and configuration properties.

## Other New Features

This section describes several more additions and improvements in Ice 3.6 that we'd like to bring to your attention.

### Remote Logging

The Ice administrative facility adds a new [logger facet](#) that allows remote access to a program's log activity. The IceGrid administrative tools now use this feature to let you view the recent log output of Ice servers, IceBox services, and IceGrid nodes and registries.



Timestamp	Type	Trace Cate...	Log Message
10/24/14 11:00:45:386	Warning		administrative session manager endpoints 'IceGrid.Registry.AdminSessi...
10/24/14 11:01:29:069	Trace	Activator	couldn't find 'icebox' executable.
10/24/14 11:01:54:853	Trace	Activator	activating server 'IceBox'
10/24/14 11:03:05:530	Trace	Activator	detected termination of server 'IceBox'
10/24/14 11:03:10:866	Trace	Activator	activating server 'IceBox'
10/24/14 11:06:38:757	Trace	Activator	detected termination of server 'IceBox'
10/24/14 11:06:49:902	Trace	Activator	activating server 'IceBox'

### Garbage Collection Improvements

Ice for C++ has included a [garbage collection](#) facility for many years, but it suffered from one significant flaw: it was impossible to guarantee that object graphs could be collected in a thread-safe manner. In this release, we've improved the implementation and modified the API to ensure it's efficient and thread-safe. The application is now responsible for telling the Ice run time when an object graph is eligible for collection by calling the `Object::ice_collectable(bool)` method. Ice assumes that a graph eligible for collection is immutable, therefore you must not update the structure of the graph after marking it as collectable. The code below demonstrates the new API:

#### C++

```
NodePtr root = new Node();
NodePtr child = new Node();
root->left = child;
child->left = root; // Create a cycle

// Mark the graph as collectable. At this point you should no longer
// modify the structure of the graph referenced by "root".
root->ice_collectable(true);
```

Users of previous Ice versions should refer to the [upgrade guide](#) for details on what has changed.

## Transport Improvements

Aside from adding the WebSocket transports and overhauling our SSL plug-in, we've made a few more enhancements to the Ice transports.

### HTTP Proxy

Outgoing TCP and SSL connections can now be mediated by an HTTP network proxy service that supports HTTP CONNECT tunneling. The new properties `Ice.HTTPProxyHost` and `Ice.HTTPProxyPort` configure the addressing information for the proxy.

### Source Address

Proxy endpoints can now include the `--sourceAddress` option to specify a network interface on which to bind outgoing connections. The `Ice.Default.SourceAddress` property supplies a default value when the endpoint option is not used.

## Batch Invocations

We made some significant changes to the way [batch invocations](#) work in Ice 3.6, making them more reliable and more consistent with other invocation modes. Please review the [upgrade guide](#) for details.

## Collocated Invocations

Collocated invocations have been reimplemented to provide [location transparency](#) semantics that are much more consistent with those of regular remote invocations. For example, you can now make asynchronous invocations on a collocated servant, and the Python language mapping now supports collocated invocations.

If your application relies on collocated invocations, you should review our [upgrade guide](#) for more information.

## Finder Interfaces

Router and locator implementations are now required to implement a *finder* interface that allows a client to obtain a proxy for the service while knowing only its address and port. A finder object has a well-known identity that a client can combine with addressing information to compose its initial proxy. Using the finder proxy, the client can request a proxy for the target service. For example, the IceGrid administrative utility only needs to prompt the user for addressing details but does not need to ask for the identity of the service's well-known object, which may have been customized in the service's configuration.

In addition to [routers](#) and [locators](#), [IceStorm](#) also supports a finder interface for obtaining the topic manager proxy.

## Java Interrupts

Ice for Java now allows an application to safely interrupt a thread that's blocked in a call to the Ice run time. Attempting to use interrupts in previous Ice releases led to undefined and generally undesirable behavior, but in Ice 3.6 the semantics are clearly defined. Consider this example:

### Java

```

HelloPrx helloProxy = ...;
try {
    helloProxy.sayHello();
} catch (Ice.OperationInterruptedException ex) {
    // interrupted by another thread!
}

```

Here we've shown how an application can deal with an interrupt that occurs during a remote invocation. Calls to other blocking Ice APIs, such as `Communicator.waitForShutdown()`, can also be interrupted. Consult the [manual](#) for a complete description of the semantics and a list of the valid interruption points.

## Java Lambda Functions

Ice 3.6 adds support for Java 8, which means you can now use lambda functions for asynchronous callbacks. As an example of how lambda functions can simplify your code, consider these two equivalent invocations:

#### Java

```
// Using an anonymous callback object
p.begin_ice_ping(new Ice.Callback_Object_ice_ping()
{
    @Override
    public void response()
    {
        System.out.println("done!");
    }
    @Override
    public void exception(Ice.LocalException ex)
    {
        System.out.println("error!");
    }
});

// Using Java 8 lambdas
p.begin_ice_ping(
    () -> System.out.println("done!"),
    (Ice.Exception ex) -> System.out.println("error!"));
```

## Java Buffers

The new metadata tag "java:buffer" can be applied to sequences of certain primitive types, causing the translator to use subclasses of `java.nio.Buffer` instead of the default mapping to a native Java array. The [buffer mapping](#) is especially convenient when an application already has data in a buffer object because it eliminates the need to create and fill a temporary array just to call a Slice operation. The buffer mapping also avoids copying when receiving these sequences by directly referencing Ice's unmarshaling buffer.

Here's a simple example:

#### Slice

```
sequence<byte> Bytes;
sequence<int> Ints;

interface Sender {
    ["java:buffer"] Ints translate(["java:buffer"] Bytes input);
};
```

The `translate` operation maps to the following Java method:

#### Java

```
java.nio.IntBuffer translate(java.nio.ByteBuffer input);
```

On the client side, a call to the `translate` proxy method returns an `IntBuffer` view of the data in the results buffer. On the server side, the implementation receives a `ByteBuffer` wrapper of the relevant portion of the incoming parameter buffer.

## C++ View Types

The new `cpp:type:view-type` metadata directive allows you to safely use "view" C++ types for operation parameters. Such C++ types point to memory held by other objects and don't copy memory. A typical example is the experimental C++14 [string\\_view](#) type. `cpp:type:view-type` can be applied to string, sequence and dictionary parameters, to speed-up your application by reducing the number of copies made during remote invocations. See [the cpp:type and cpp:view-type Metadata Directives](#) for details.

## Linking with C++ Libraries on Windows

You no longer need to list Ice import libraries (such as `IceD.lib`) when linking with Ice C++ libraries on Windows. `pragma` directives in Ice header files provide these import library names to the linker, for example:

### C++

```
// For Ice C++ library
#ifdef _MSC_VER
#   if !defined(ICE_BUILDING_ICE)
#       if defined(_DEBUG) && !defined(ICE_OS_WINRT)
#           pragma comment(lib, "IceD.lib")
#       else
#           pragma comment(lib, "Ice.lib")
#       endif
#   endif
#endif
```

The [Ice Builder for Visual Studio](#) automatically configures the `$(IceLib)` directory that contains these import libraries. `$(IceLib)` depends on the versions of Ice and Visual Studio you're using, and on the target architecture. For example, `$(IceLib)` for a Ice 3.6, Visual Studio 2012 and a x64 target is `$(IceHome)\lib\vc110\x64`.

## C++ Plug-in for Crypt Password Files

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](#) algorithm to hash the provided password and verify if this hash matches the hash in the password-file. This DES Unix Crypt implementation was provided by OpenSSL on all platforms.

Ice 3.6 supports more recent and secure hash formats and algorithms:

- Windows and OS X: [PBKDF2](#) with SHA-1, SHA-256, or SHA-512 as digest algorithm
- Linux: [SHA-256](#) and [SHA-512](#) Crypt, plus the old [DES-based Unix Crypt](#) for compatibility with earlier releases

Windows and OS X no longer rely on OpenSSL for this password hashing, nor for anything else.

In Ice 3.6, we have also moved the implementation of this authentication mechanism to a separate C++ plug-in, the `Glacier2CryptPermissionsVerifier` plug-in. This plug-in is loaded automatically by Glacier2 or IceGrid if you configure a crypt passwords file using one of the existing `CryptPasswords` properties [Glacier2.CryptPasswords](#), [IceGrid.Registry.AdminCryptPasswords](#) and [IceGrid.Registry.CryptPasswords](#). If you don't use the simple authentication mechanism provided by the `CryptPasswords` properties, you don't need to include this plug-in with your application.

## Support for -fvisibility=hidden with GCC and clang

Ice for C++ is now built by default with `-fvisibility=hidden` on Linux (with GCC) and OS X (with clang). You can also build code generated by `slice2cpp` with `-fvisibility=hidden`, and use the `--dll-export` option to export symbols from such generated code.