Dispatching Requests to User Threads

Previous Next

By default, Ice uses a thread from one of its thread pools to dispatch requests and to execute callbacks registered to run upon completion of asynchronous invocations.

For regular calls, Ice uses threads from its server thread pool (or object adapter thread pool, if one is configured) for dispatching requests, and it uses threads from its client thread pool to execute AMI callbacks. For bidir calls, the roles are reversed: Ice uses threads from its client thread pool for dispatching requests and threads from the server thread pool (or object adapter thread pool) for AMI callbacks.

This simple behavior is suitable for most applications. There are however situations you may want to execute dispatches or AMI callbacks in a particular thread. For example, in a server, you might need to update a database that does not permit concurrent access from different threads or, in a client, you might need to update a user interface with the results of an invocation. (Many UI frameworks require all UI updates to be made by a specific thread.)

Ice allows you to register a *dispatcher* to control which thread(s) execute dispatches and AMI callbacks. The dispatcher API is specific to each language mapping.

On this page:

Dispatcher Interception Point

A dispatcher intercepts a call (request dispatch or AMI callback) and can select in which thread to execute this call. An Ice communicator gives each such call to the registered dispatcher before this call unmarshals parameters and before this call locates the target servant (for request dispatches). The call given to the dispatcher takes no parameters, returns nothing, and never throws any exception back to the dispatcher.

Registering the Dispatcher in InitializationData

You configure the dispatcher used by an Ice communicator by setting the data member `dispatcher` of its `InitializationData`:

C++11

```
int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties(argc, argv);
    initData.dispatcher = dispatcherFunction;

    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

The `dispatcher` data member's type is `std::function<void(std::function<void()>, const std::shared_ptr<Ice::Connection>&)>`.

C++98

```
class MyDispatcher : public Ice::Dispatcher /*, ... */
{
    // ...
};

int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties(argc, argv);
    initData.dispatcher = new MyDispatcher;
    Ice::CommunicatorHolder ich(argc, argv, initData);
    ...
}
```

The `Ice::Dispatcher` abstract base class has the following interface:

```
namespace Ice
{
    class Dispatcher : public virtual IceUtil::Shared
    {
    public:
        virtual void dispatch(const DispatcherCallPtr&, const ConnectionPtr&) = 0;
    };

    typedef IceUtil::Handle<Dispatcher> DispatcherPtr;
}
```

The `DispatcherCall` instance encapsulates all the details of the call. It is another abstract base class with the following interface:

```
namespace Ice
{
    class DispatcherCall : public virtual IceUtil::Shared
    {
    public:
        virtual ~DispatcherCall() { }

        virtual void run() = 0;
    };

    typedef IceUtil::Handle<DispatcherCall> DispatcherCallPtr;
}
```

C#

C

```
public class Server
{
    public static void Main(string[] args)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = (System.Action call, Ice.Connection connection) => { ... };
        using(Ice.Communicator communicator = Ice.Util.initialize(ref args, initData))
        {
            // ...
        }
    }
}
```

The dispatcher is a delegate with type `System.Action<System.Action, Ice.Connection>`.

Java

```
public class Server
{
    public static void main(String[] args)
    {
        com.zeroc.Ice.InitializationData initData = new com.zeroc.Ice.InitializationData();
        initData.dispatcher = (runnable, connection) -> { ... };

        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args, initData))
        {
            // ...
        }
        catch(LocalException ex)
        {
            // ...
        }

        // ...
    }
}
```

The `dispatcher` is an object (typically a lambda) that implements the functional interface `java.util.function.BiConsumer<Runnable, com.zeroc.Ice.Connection>`.

Java Compat

```
public class MyDispatcher implements Ice.Dispatcher
{
    // ...
}

public class Server
{
    public static void main(String[] args)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = new MyDispatcher();
        try(Ice.Communicator communicator = Ice.Util.initialize(args, initData))
        {
           // ...
        }
        // ...
    }
}
```

The `dispatcher` is an object that implements the functional interface `Ice.Dispatcher`:

```
public interface Dispatcher
{
    void dispatch(Runnable runnable, Ice.Connection con);
}
```

ObjC

```
int
main(int argc, char* argv[])
{
    objc_startCollectorThread();
    id<ICECommunicator> communicator = nil;
    @try
    {
        ICEInitializationData* initData = [ICEInitializationData initializationData];
        initData.dispatcher =
            ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
            {
                // ...
            };
        communicator = [ICEUtil createCommunicator:&argc argv:argv initData:initData];
        // ...
    }
    @catch(ICELocalException* ex)
    {
        // ...
    }

    // ...
}
```

The type of the dispatcher callback must match the following block signature:

```
void(^)(id<ICEDispatcherCall> call, id<ICEConnection> connection)
```

The `ICEDispatcherCall` protocol defines how to execute the call:

```
@protocol ICEDispatcherCall <NSObject>
-(void) run;
@end
```

Python

```
initData = Ice.InitializationData()
initData.dispatcher = lambda call, connection: ...

with Ice.Util.initialize(sys.argv, initData) as communicator:
    # ...
```

The dispatcher is a callable object.

Afterwards, the Ice communicator calls the configured dispatcher whenever it dispatches a request or executes an application-supplied callback upon completion of an asynchronous invocation. The first parameter given to the dispatcher corresponds to the call (dispatch or AMI callback) that the dispatcher must execute. The second parameter is the connection associated with this call (if any). The connection parameter is null in the following situations:

- for collocated calls
- when the call failed before a connection could be associated with this call
- when the call is executed through a C# scheduler or Java executor (see Dispatcher with Future Results later on this page)

A dispatcher must always execute the supplied call. Failure to dispatch a call will cause `Communicator::destroy` to block indefinitely. Furthermore, a dispatcher must not make blocking calls from the dispatch thread, such as synchronous invocations or calls to proxy methods that can potentially block, like `ice_getConnection`. Since these calls use the dispatcher for their own completion, you will get a deadlock if your dispatcher executes all calls on a single thread.

If a dispatcher has resources that must be reclaimed (e.g., joining with a helper thread), it can safely do so after `Communicator::destroy` has completed.

Trivial Dispatcher

You can write a dispatcher that blocks and waits for completion of the supplied call, since the dispatcher is called by a thread in the server-side thread pool (for non-bidir request dispatches) or the client-side thread pool (for non-bidir AMI callbacks). For example:

C++11

```
initData.dispatcher = [](std::function<void()> dispatchCall, const std::shared_ptr<Ice::Connection>&)
                      {
                         dispatchCall(); // Does not throw, blocks until the call completes.
                      };
```

C++98

```
class MyDispatcher : public Ice::Dispatcher
{
public:
    virtual void dispatch(const Ice::DispatcherCallPtr& d, const Ice::ConnectionPtr&)
    {
        d->run(); // Does not throw, blocks until the call completes.
    }
};
```

C#

```
initData.dispatcher = (System.Action call, Ice.Connection connection) =>
{
    call(); // Does not throw, blocks until the call completes.
}
```

Java

```
initData.dispatcher = (runnable, connection) -> { runnable.run(); };
```

Java Compat

```
public class MyDispatcher implements Ice.Dispatcher
{
    public void dispatch(Runnable runnable, Ice.Connection connection)
    {
        // Does not throw, blocks until the call completes.
        runnable.run();
    }
}
```

Objective-C

```
void(^myDispatcher)(id<ICEDispatcherCall>, id<ICEConnection>) =
    ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
    {
        // Does not throw, blocks until the call completes.
        [call run];
    };
```

Python

```
initData.dispatcher = lambda call, connection: call()
```

This implementation ties up a thread in the thread pool for the duration of the call, and is not particularly useful: a communicator with no dispatcher provides the same behavior.

Dispatcher for Graphical Applications

The primary use-case for dispatchers is graphical applications where only one thread is allowed to call UI methods. With such an application, you can register a dispatcher that executes all calls in the UI thread. You can also use the UI thread to make asynchronous invocations, since Ice guarantees asynchronous invocations never block the calling thread.

Here are some examples:

C++11

With Qt

```
//
// Define a custom event type to be used by the dispatcher
//
class DispatchEvent : public QEvent
{
public:

    DispatchEvent(std::function<void()> call) :
        QEvent(QEvent::Type(CUSTOM_EVENT_TYPE)),
        _call(call)
    {
    }

    void dispatch()
    {
        _call();
    }

private:

    std::function<void()> _call;
};

MainWindow::MainWindow()
{
    Ice::InitializationData initData;
    initData.properties = Ice::createProperties();
    //
    // The dispatcher implementation creates a new DispatchEvent and adds it to event
    // queue, setting this object as the receiver of the event.
    //
    initData.dispatcher = [this](std::function<void()> dispatchCall, const std::shared_ptr<Ice::Connection>&)
    {
        QApplication::postEvent(this, new DispatchEvent(dispatchCall));
    };
    _communicator = Ice::initialize(initData);
}

//
// Override QObject::event to handle our custom event type and delegate
// to the base class to handle other event types.
//
bool
MainWindow::event(QEvent* event)
{
    if(event->type() == CUSTOM_EVENT_TYPE)
    {
        auto dispatchEvent = static_cast<DispatchEvent*>(event);
        try
        {
            dispatchEvent->dispatch();
        }
        catch(const std::exception& ex)
        {
           ...
        }
        return true;
    }
    return QMainWindow::event(event);
}
```

C++98

With MFC

```
class MyDialog : public CDialog { ... };

class MyDispatcher : public Ice::Dispatcher
{
public:
    MyDispatcher(MyDialog* dialog) : _dialog(dialog)
    {
    }

    virtual void
    dispatch(const Ice::DispatcherCallPtr& call, const Ice::ConnectionPtr&)
    {
        _dialog->PostMessage(WM_AMI_CALLBACK, 0,
                             reinterpret_cast<LPARAM>(new Ice::DispatcherCallPtr(call)));
    }

private:
    MyDialog* _dialog;
};
```

The `MyDispatcher` class simply stores the `CDialog` handle for the UI and calls `PostMessage`, passing the `DispatcherCall` instance. In turn, this causes the UI thread to receive an event and invoke the UI callback method that was registered to respond to `WM_AMI_CALLBACK` events.

The implementation of the callback function calls `run`:

With MFC

```
LRESULT
MyDialog::OnAMICallback(WPARAM, LPARAM lParam)
{
    try
    {
        Ice::DispatcherCallPtr* call = reinterpret_cast<Ice::DispatcherCallPtr*>(lParam);
        (*call)->run();
        delete call;
    }
    catch(const Ice::Exception& ex)
    {
        // ...
    }
    return 0;
}
```

The Ice run time calls `dispatch` once the asynchronous invocation is complete. In turn, this triggers the `OnAMICallback` which calls `run`. Because the operation has completed already, `run` does not block, so the UI remains responsive.

C#

With WPF

```
public partial class MyWindow : Window
{
    private void Window_Loaded(object sender, EventArgs e)
    {
        Ice.InitializationData initData = new Ice.InitializationData();
        initData.dispatcher = (System.Action call, Ice.Connection connection) =>
        {
            Dispatcher.BeginInvoke(DispatcherPriority.Normal, action);
        };
        using(Ice.Communicator communicator = Ice.Util.initialize(initData))
        {
            // ...
        }
     }
}
```

The delegate calls `Dispatcher.BeginInvoke` on the `action` delegate. This causes WPF to queue the actual asynchronous invocation of `action` for later execution by the UI thread. Because the Ice run time does not call your delegate until an asynchronous invocation is complete, when the UI thread executes the corresponding call to the `EndInvoke` method, that call does not block and the UI remains responsive.

The net effect is that you can invoke an operation asynchronously from a UI callback method without the risk of blocking the UI thread. For example:

With WPF

```
public partial class MyWindow : Window
{
    private async void someOp_Click(object sender, RoutedEventArgs e)
    {
        MyIntfPrx p = ...;

        try
        {
            // Call remote operation asynchronously.
            await p.someOpAsync();
            // Update UI...
        }
        catch(System.Exception ex)
        {
            // Update UI...
        }
    }
}
```

We're using Ice's task-based API together with the `async` and `await` keywords to execute asynchronous tasks in a straightforward way. The return value of `someOpAsync` is a `Task` on which we use the `await` keyword to suspend processing until the call completes. Thanks to the dispatcher, processing eventually resumes in the UI thread and we can update the UI as needed. The `csharp/Ice/wpf` demo shows a fully-functional UI client that uses this technique.

Java

With Swing

```
public class Client extends JFrame
{
    public static void main(final String[] args)
    {
        SwingUtilities.invokeLater(
            () -> {
                      try
                      {
                          new Client(args);
                      }
                      catch(com.zeroc.Ice.LocalException e)
                      {
                          JOptionPane.showMessageDialog(
                             null, e.toString(),
                             "Initialization failed",
                             JOptionPane.ERROR_MESSAGE);
                      }
                  });
    }

    Client(String[] args)
    {
        com.zeroc.Ice.InitializationData initData = new com.zeroc.Ice.InitializationData();
        initData.dispatcher = (runnable, connection) -> { SwingUtilities.invokeLater(runnable); };
        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args, initData))
        {
            ...
        }
    }
}
```

The dispatcher simply delays the call to `run` by calling `invokeLater`, passing it the `Runnable` that is provided by the Ice run time. This causes the Swing UI thread to make the call to `run`. Because the Ice run time does not call the dispatcher until the asynchronous invocation is complete, that call to `run` does not block and the UI remains responsive.

The `java/Ice/swing` demo shows a fully-functional UI client that uses this technique.

ObjC

With Cocoa

```
-(void)viewDidLoad
{
    ICEInitializationData* initData = [ICEInitializationData initializationData];
    initData.dispatcher =
        ^(id<ICEDispatcherCall> call, id<ICEConnection> con)
        {
            dispatch_sync(dispatch_get_main_queue(), ^ { [call run]; });
        };

    communicator = [[ICEUtil createCommunicator:initData] retain];

    // ....
}
```

The dispatcher callback calls `dispatch_sync` on the main queue. This queues the actual call for later execution by the main thread. Because the Ice run time does not call the dispatcher callback until an asynchronous operation invocation is complete, when the UI thread executes the corresponding call, that call does not block and the UI remains responsive.

The net effect is that you can invoke an operation asynchronously from a UI callback method without the risk of blocking the UI thread. For example:

With Cocoa

```
-(void)someOp:(id)sender
{
    id<MyIntfPrx> p = ...;
    [p begin_someOp:^{ [self response]; }
          exception:^(ICEException* ex) { [self exception:ex]; }];
}

-(void) response
{
    // Update UI...
}

-(void) exception:(ICEException* ex)
{
    // Update UI...
}
```

The `objective-c/Ice/iOS/hello` demo shows a fully-functional UI client that uses this technique.

Back to Top ^

Dispatcher with Future Results

With C#, Java and Python, asynchronous method invocations return a "future" object on which you can register callbacks to execute upon completion. This future object is a Task in C#, a CompletableFuture in Java and an Ice.InvocationFuture in Python. (We are ignoring here the deprecated AMI APIs for C# and Python that don't return future objects.)

In C# and Java, the thread that executes the callback you supply is determined by the .NET and Java specifications, respectively, so unless you are careful, you typically won't use the configured dispatcher for these callbacks.

It is also possible that the call has already completed by the time you register the callback with the future object, so depending on how you register the callback, the callback could execute synchronously from the calling thread.

If you want to always use the configured dispatcher for these callbacks, regardless of whether or not the call already completed when you register the callback, you need to:

- in C#, register your callback with `ContinueWith` using a custom scheduler provided by *`proxy`*`.ice_scheduler()`. *`proxy`* represents the proxy you used for the asynchronous invocation. You should also not specify the `ExecuteSynchronously` task continuation option or you should use the `RunContinuationsAsynchronously` option to ensure callbacks are always executed asynchronously even if the task completed when `ContinueWith` is called.
- in Java, call an `Async` method on the `CompletableFuture` result (such as `whenCompleteAsync`), and pass *`proxy`*`.ice_executor()` as the second parameter. *`proxy`* represents the proxy you used for the asynchronous invocation.
- in Python, register your callback with `add_done_callback_async` instead of `add_done_callback.`

Back to Top ^

See Also

- Asynchronous Method Invocation (AMI) in C++11
- Asynchronous Method Invocation (AMI) in C++98
- Asynchronous Method Invocation (AMI) in C-Sharp
- Asynchronous Method Invocation (AMI) in Java
- Asynchronous Method Invocation (AMI) in Java Compat
- Asynchronous Method Invocation (AMI) in Objective-C
- Asynchronous Method Invocation (AMI) in Python