Dispatching Invocations to User Threads

By default, operation invocations and AMI callbacks are executed by a thread from a thread pool. This behavior is simple and convenient for applications because they need not concern themselves with thread creation and destruction. However, there are situations where it is necessary to respond to operation invocations 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.)

In Ice for C++, Java, .NET, and Objective-C, you can control which thread receives operation invocations and AMI callbacks, so you can ensure that all updates are made by a thread you choose. The implementation techniques vary slightly for each language and are explained in the sections that follow.

On this page:

- C++ Dispatcher API
- Java Dispatcher API
- C# Dispatcher API
- Objective-C Dispatcher API
- Dispatcher Implementation Notes

C++ Dispatcher API

To install a dispatcher, you must instantiate a class that derives from `Ice::Dispatcher` and initialize a communicator with that instance in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

C++

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

int
main(int argc, char* argv[])
{
    Ice::CommunicatorPtr communicator;

    try {
        Ice::InitializationData initData;
        initData.properties = Ice::createProperties(argc, argv);
        initData.dispatcher = new MyDispatcher();
        communicator = Ice::initialize(argc, argv, initData);

        // ...
    } catch (const Ice::Exception& ex) {
        // ...
    }

    // ...
}
```

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

C++

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

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

The Ice run time invokes the `dispatch` method whenever an operation invocation arrives or an AMI invocation completes, passing an instance of `DispatcherCall` and the connection via which the invocation arrived. The job of `dispatch` is to pass the incoming invocation to an operation implementation.

The connection parameter allows you to decide how to dispatch the operation based on the connection via which it was received. This value may be nil if no connection currently exists.

You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

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

C++

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

    virtual void run() = 0;
};

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

Your implementation of `dispatch` is expected to call `run` on the `DispatcherCall` instance (or, more commonly, to cause `run` to be called some time later). When you call `run`, the Ice run time processes the invocation in the thread that calls `run`.

A very simple way to implement `dispatch` would be as follows:

C++

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

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, calls `run` on the `DispatcherCall` instance.

With this simple example, `dispatch` immediately calls `run`, and `run` does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation. In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with MFC:

C++

```
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.

In turn, the implementation of the callback method calls `run`:

C++

```
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 operation invocation is complete. In turn, this causes the `OnAMICallback` to trigger, which calls `run`. Because the operation has completed already, `run` does not block, so the UI remains responsive.

Please see the `MFC` demo in your Ice distribution for a fully-functional UI client that uses this technique.

Java Dispatcher API

To install a dispatcher, you must instantiate a class that implements `Ice.Dispatcher` and initialize a communicator with that instance in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

Java

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

public class Server
{
    public static void
    main(String[] args)
    {
        Ice.Communicator communicator;

        try {
            Ice.InitializationData initData = new Ice.InitializationData();
            initData.properties = Ice.Util.createProperties(args);
            initData.dispatcher = new MyDispatcher();
            communicator = Ice.Util.initialize(args, initData);

            // ...
        } catch (Ice.LocalException & ex) { {
            // ...
        }

        // ...
    }

    // ...
}
```

The `Ice.Dispatcher` interface looks as follows:

Java

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

The Ice run time invokes the `dispatch` method whenever an operation invocation arrives, passing a `Runnable` and the connection via which the invocation arrived. The job of `dispatch` is to pass the incoming invocation to an operation implementation.

The connection parameter allows you to decide how to dispatch the operation based on the connection via which it was received. This value may be null if no connection currently exists.

You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

Your implementation of `dispatch` is expected to call `run` on the `Runnable` instance (or, more commonly, to cause `run` to be called some time later). When you call `run`, the Ice run time processes the invocation in the thread that calls `run`.

A very simple way to implement `dispatch` would be as follows:

Java

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

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, calls `run` on the `Runnable` instance.

With this simple example, `dispatch` immediately calls `run`, and `run` does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation. In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI `response` callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with Swing:

Java

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

    Client(String[] args)
    {
        Ice.Communicator communicator;

        try {
            Ice.InitializationData initData = new Ice.InitializationData();
            initData.dispatcher = new Ice.Dispatcher()
            {
                public void
                dispatch(Runnable runnable, Ice.Connection connection)
                {
                    SwingUtilities.invokeLater(runnable);
                }
            };
            communicator = Ice.Util.initialize(args, initData);
        }
        catch(Throwable ex)
        {
            // ...
        }
        // ...
    }

    // ...
}
```

The `dispatch` method 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 eventually make the call to `run`. Because the Ice run time does not call `dispatch` until the asynchronous invocation is complete, that call to `run` does not block and the UI remains responsive.

Please see the `swing` demo in your Ice distribution for a fully-functional UI client that uses this technique.

C# Dispatcher API

To install a dispatcher, you must initialize a communicator with a delegate of type `Ice.Dispatcher` in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

C#

```
public class Server
{
    public static void Main(string[] args)
    {
        Ice.Communicator communicator = null;

        try {
            Ice.InitializationData initData = new Ice.InitializationData();
            initData.dispatcher = new MyDispatcher().dispatch;
            communicator = Ice.Util.initialize(ref args, initData);
            // ...
        } catch (System.Exception ex) {
            // ...
        }

        // ...
    }

    // ...
}
```

The `Ice.Dispatcher` delegate is defined as follows:

C#

```
public delegate void Dispatcher(System.Action call, Connection con);
```

The Ice run time calls your delegate whenever an operation invocation arrives, passing a `System.Action` delegate and the connection via which the invocation arrived. The job of your delegate is to pass the incoming invocation to an operation implementation.

The connection parameter allows you to decide how to dispatch the operation based on the connection via which it was received. This value may be null if no connection currently exists.

In this example, the delegate calls a method `dispatch` on an instance of a `MyDispatcher` class. You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

Your implementation of `dispatch` is expected to invoke the `call` delegate (or, more commonly, to cause it to be invoked some time later). When you invoke the `call` delegate, the Ice run time processes the invocation in the thread that invokes the delegate.

A very simple way to implement `dispatch` would be as follows:

C#

```
public class MyDispatcher
{
    public void
    dispatch(System.Action call, Ice.Connection con)
    {
        // Does not throw, blocks until op completes.
        call();
    }
};
```

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, invokes the `call` delegate.

With this simple example, `dispatch` immediately invokes the delegate, and that call does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation. In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with WPF:

C#

```
public partial class MyWindow : Window
{
    private void Window_Loaded(object sender, EventArgs e)
    {
        Ice.Communicator communicator = null;

        try
        {
            Ice.InitializationData initData = new Ice.InitializationData();
            initData.dispatcher =
                    delegate(System.Action action, Ice.Connection connection)
            {
                Dispatcher.BeginInvoke(DispatcherPriority.Normal, action);
            };
            communicator = Ice.Util.initialize(initData);
        }
        catch(Ice.LocalException ex)
        {
            // ...
        }
    }

    // ...
}
```

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 invoke your delegate until an asynchronous operation 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:

C#

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

        // Call remote operation asynchronously.
        // Response is processed in UI thread.
        p.begin_someOp().whenCompleted(this.opResponse, this.opException);
    }

    public void opResponse()
    {
        // Update UI...
    }

    public void opException(Ice.Exception ex)
    {
        // Update UI...
    }
}
```

Please see the `wpf` demo in your Ice distribution for a fully-functional UI client that uses this technique.

Objective-C Dispatcher API

To install a dispatcher, you must initialize a communicator with a callback (as an Objective-C block) in the `ICEInitializationData` structure. All invocations that arrive for this communicator are made via the specified callback. For example:

Objective-C

```
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:

Objective-C

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

The Ice run time invokes the dispatcher callback whenever an operation invocation arrives, passing an object implementing the `ICEDispatcherCall` protocol and the connection via which the invocation arrived. The job of your callback implementation is to execute the given call.

The connection parameter allows you to decide how to dispatch the operation based on the connection via which it was received. This value may be nil if no connection currently exists.

You can write the callback such that it blocks and waits for completion of the invocation because the callback is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

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

Objective-C

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

Your callback is expected to call `run` on the `ICEDispatcherCall` instance (or, more commonly, to cause run to be called some time later). When you call `run`, the Ice run time processes the invocation in the thread that calls `run`.

A very simple way to implement the dispatcher callback would be as follows:

Objective-C

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

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls the dispatcher callback which, in turn, invokes the run method on the call.

With this simple example, the dispatcher callback immediately invokes `run`, and `run` does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation. In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to Cocoa or Cocoa Touch main thread:

Objective-C

```
-(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 invoke 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:

Objective-C

```
-(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...
}
```

Please see the Cocoa or iPhone demos in your Ice Touch distribution for fully-functional UI clients that use this technique.

Dispatcher Implementation Notes

An application that uses a custom dispatcher must adhere to the following rules to avoid a deadlock:

- Dispatcher implementations must ensure that all requests are dispatched. Failing to dispatch all requests will cause `Communicator::destroy` to hang indefinitely. 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.
- Never make a blocking invocation from the dispatch thread, such as a synchronous proxy operation or a proxy method that can potentially block, such as `ice_getConnection`. These invocations depend on the dispatcher for their own completion, therefore blocking the dispatch thread will inevitably lead to deadlock.

See Also

- Asynchronous Method Invocation (AMI) in C++
- Asynchronous Method Invocation (AMI) in Java
- Asynchronous Method Invocation (AMI) in C-Sharp
- Asynchronous Method Invocation (AMI) in Objective-C