AMI in C-Sharp with Tasks

On this page:

Basic Asynchronous API in C#

Consider the following simple Slice definition:

Slice

```
module Demo
{
    interface Employees
    {
        string getName(int number);
    }
}
```

Asynchronous Proxy Methods in C#

Besides the synchronous proxy methods, `slice2cs` generates the following asynchronous proxy method:

C#

```
public interface EmployeesPrx : Ice.ObjectPrx
{
    System.Threading.Tasks.Task<string>
    getNameAsync(int number,
                 Ice.OptionalContext context = new Ice.OptionalContext(),
                 System.IProgress<bool> progress = null,
                 System.Threading.CancellationToken cancel = new System.Threading.CancellationToken());

    ...
}
```

As you can see, the `getName` operation generates a `getNameAsync` method that accepts several optional parameters:

- a per-invocation context
- a sent callback
- a cancellation token

The `getNameAsync` method sends (or queues) an invocation of `getName`. This method does not block the calling thread. It returns a `Task` that you can use in a number of ways, including blocking to obtain the result, configuring a continuation to be executed when the result becomes available, and polling to check the status of the request.

Here's an example that calls `getNameAsync`:

C#

```
EmployeesPrx e = ...;
Task<string> t = e.getNameAsync(99);

// Continue to do other things here...

string name = t.Result;
```

Because `getNameAsync` does not block, the calling thread can do other things while the operation is in progress.

Asynchronous Mapping for Out Parameters in C#

.NET's standard `Task` API only allows a task to produce one result value. Since a Slice operation could potentially return any number of values, the asynchronous mapping must differ significantly from the synchronous mapping.

The asynchronous mapping depends on how many values an operation returns, including out parameters and a non-`void` return value:

- Zero values
  The corresponding C# method returns an instance of `System.Threading.Tasks.Task`.

- One value
  The corresponding C# method returns an instance of `System.Threading.Tasks.Task<T>` where `T` is the mapped type, regardless of whether the Slice definition of the operation declared it as a return value or as an out parameter. Consider this example:

  Slice

  ```
  interface I
  {
      string op1();
      void op2(out string name);
  }
  ```

  The asynchronous mapping generates corresponding methods with identical signatures:

  C#

  ```
  public interface IPrx : Ice.ObjectPrx
  {
      System.Threading.Tasks.Task<string>
      op1Async(Ice.OptionalContext context = new Ice.OptionalContext(),
               System.IProgress<bool> progress = null,
               System.Threading.CancellationToken cancel = new System.Threading.CancellationToken());

      System.Threading.Tasks.Task<string>
      op2Async(Ice.OptionalContext context = new Ice.OptionalContext(),
               System.IProgress<bool> progress = null,
               System.Threading.CancellationToken cancel = new System.Threading.CancellationToken());

      ...
  }
  ```

- Multiple values
  The Slice-to-C# translator generates an extra structure to hold the results of an operation that returns multiple values. This "result type" resides in the same namespace as the proxy interface and has the name `*Interface_Op*Result`, where `*Interface*` represents the name of the Slice interface that defines the operation `*Op*`. The leading character of the operation name `*Op*` is always capitalized. The values of out parameters are provided in corresponding data members of the same names. If the operation declares a return value, its value is provided in the data member named `returnValue`. If an out parameter is also named `returnValue`, the data member to hold the operation's return value is named `_returnValue` instead. The result type defines a "one-shot" constructor that accepts and assigns a value for each of its data members. The corresponding C# method returns an instance of `System.Threading.Tasks.Task<T>` where `T` is the result type. Consider this example:

Slice

```
interface Example
{
    double op(int inp1, string inp2, out bool outp1, out long outp2);
}
```

The generated code looks like this:

C#

```
public struct Example_OpResult
{
    public Example_OpResult(double returnValue, bool outp1, long outp2) { ... }

    public double returnValue;
    public bool outp1;
    public long outp2;
}

public interface ExamplePrx : Ice.ObjectPrx
{
    System.Threading.Tasks.Task<Example_OpResult>
    opAsync(int inp1, string inp2,
            Ice.OptionalContext context = new Ice.OptionalContext(),
            System.IProgress<bool> progress = null,
            System.Threading.CancellationToken cancel = new System.Threading.CancellationToken());

    ...
}
```

Now let's invoke `opAsync` to demonstrate one way of asynchronously executing an action when the invocation completes:

C#

```
ExamplePrx e = ...;
e.opAsync().ContinueWith((t) =>
    {
        try
        {
            var r = t.Result; // Returns Example_OpResult
            Console.WriteLine("returnValue = {0} outp1 = {1} outp2 = {2}", r.returnValue, r.outp1, r.
outp2);
        }
        catch (System.AggregateException ex)
        {
            // handle exception...
        }
    });
```

Here's a simpler version that uses the `await` keyword:

C#

```
ExamplePrx e = ...;
try
{
    var r = await e.opAsync();
    Console.WriteLine("returnValue = {0} outp1 = {1} outp2 = {2}", r.returnValue, r.outp1, r.outp2);
}
catch (Ice.Exception ex)
{
    // handle exception...
}
```

Asynchronous Exception Semantics in C#

If an invocation raises an exception, the exception can be obtained from the task. For example, calling `Wait` on the task raises a `System.AggregateException` whose `InnerException` property contains the actual exception. The task's `Exception` property also returns the `AggregateException` if the exception has already occurred at the time you access the property.

The exception is provided by the task, even if the actual error condition for the exception was encountered during the call to the `opAsync` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that handles the task (instead of being present twice, once where the `opAsync` method is called, and again where the task is handled).

There are two exceptions to this rule:

- if you destroy the communicator and then make an asynchronous invocation, the `opAsync` method throws `CommunicatorDestroyedException` directly.
- a call to an `Async` function can throw `TwowayOnlyException`. An `Async` function throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy.

Using the `await` keyword to invoke an asynchronous proxy method does not raise `AggregateException` but rather raises the inner exception directly. In other words, the exception semantics with `await` are the same as if you had invoked the synchronous version of the proxy method.

Polling for Completion in C#

The asynchronous API allows you to poll for call completion, which can be useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

Slice

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
}
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

C#

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.Length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing — the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

C#

```
using System.Threading;
using System.Threading.Tasks;
...

FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;

var results = new LinkedList<Task>();
const int numRequests = 5;
var sent = new AutoResetEvent(false);
while(!file.eof())
{
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    var task = ft.sendAsync(offset, bs, progress:(ss) => sent.Set());
    offset += bs.Length;

    // Wait until this request has been passed to the transport.
    sent.WaitOne();
    results.AddLast(task);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while(results.Count > numRequests)
    {
        var t = results.First;
        results.RemoveFirst();
        t.Wait();
    }
}

// Wait for any remaining requests to complete.
Task.WaitAll(results.ToArray());
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

Asynchronous Oneway Invocations in C#

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call an asynchronous method on a oneway proxy for an operation that returns values or raises a user exception, the proxy method throws `TwowayOnlyException`.

The task returned for a oneway invocation completes as soon as the request is successfully written to the client-side transport. The task completes with an exception if an error occurs before the request is successfully written.

Flow Control in C#

Asynchronous method invocations never block the thread that calls the asynchronous proxy method. The Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background.

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport. One of the optional arguments to every asynchronous proxy invocation is a `System.IProgress<bool>`. If you provide a delegate, the Ice run time will eventually invoke it when the request has been sent and provide a boolean argument indicating whether the request was sent synchronously. This argument is true if the entire request could be transferred to the local transport in the caller's thread without blocking, otherwise the argument is false. Furthermore, a value of true indicates that Ice is invoking your delegate recursively from the calling thread, whereas a value of false indicates that Ice is invoking the delegate from an Ice thread pool thread.

Here's a simple example to demonstrate the flow control feature:

C#

```
ExamplePrx proxy = ...;
proxy.doSomethingAsync(progress:(sentSynchronously) =>
    {
        if(sentSynchronously)
        {
            // Entire request was accepted by the transport,
            // called recursively from this thread
        }
        else
        {
            // Request was queued but has now been sent,
            // called from a separate thread
        }
    });
```

Using this feature, you can limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

Asynchronous Batch Requests in C#

Applications that send batched requests can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

The proxy method `ice_flushBatchRequestsAsync` flushes any batch requests queued by that proxy. In addition, similar methods are available on the communicator and the `Connection` objects. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Canceling Asynchronous Requests in C#

Every asynchronous proxy method accepts an optional instance of the structure `System.Threading.CancellationToken`. The default value is an empty structure, which is equivalent to passing `CancellationToken.None`. If your application requires the ability to cancel an asynchronous request, you need to create a `CancellationTokenSource` from which you can obtain a token. Cancelling a request is achieved by calling `Cancel` on the source object.

Cancellation prevents a queued invocation from being sent or, if the invocation has already been sent, ignores a reply if the server sends one. Cancellation is a local operation and has no effect on the server. The result of a canceled invocation is an `Ice::InvocationCanceledException`.

Concurrency Semantics for AMI in C#

The default behavior of a call to `ContinueWith` is to execute the continuation in a separate thread from the .NET thread pool. If you're trying to minimize thread context switches, you can pass `TaskContinuationOptions.ExecuteSynchronously` as an additional argument to `ContinueWith`. In this case, the behavior depends on the task's status: if the reply to the proxy invocation has already been received at the time `ContinueWith` is called, the continuation will be invoked by the current thread. If the reply has not yet been received, the continuation will be invoked by an Ice thread pool thread.

If a dispatcher is configured, the Ice thread pool delegates the execution of the continuation to the dispatcher.

The scheduler that runs continuations can be changed by passing a custom scheduler to `ContinueWith`. The Ice thread pool can be used as a task scheduler, and you can obtain this scheduler by calling the `ice_scheduler` proxy method and passing it to `ContinueWith`. With the Ice thread pool scheduler, the continuation is queued to be executed by the Ice thread pool. If you pass the option `TaskContinuationOptions.ExecuteSynchronously` to `ContinueWith`, and the reply has been received at the time you call `ContinueWith`, your thread will execute the continuation. Therefore, if you want to ensure the continuation is always executed by an Ice thread pool thread (or indirectly the dispatcher, if one is configured), you need to call `ContinueWith`:

- with `*proxy*.ice_scheduler()` as your task scheduler
- without `TaskContinuationOptions.ExecuteSynchronously` as continuation option (or, alternatively, override this option with `TaskContinuationOptions.RunContinuationsAsynchronously)`

When using `async` and `await`, the concurrency semantics are determined by the synchronization context in which you're making the proxy invocation. For example, awaiting an asynchronous proxy invocation from the main thread will invoke the continuation from a .NET thread pool thread. Similarly, awaiting an asynchronous proxy invocation from the GUI thread in a graphical application will invoke the continuation from the GUI thread.

Ice configures a synchronization context for its own thread pool threads, so if you happen to await an asynchronous proxy invocation while in an Ice thread pool thread, the continuation will also be invoked by an Ice thread.

Refer to the flow control discussion for information about the concurrency semantics of the sent callback.

See Also

- Request Contexts
- Batched Invocations
- Collocated Invocation and Dispatch