AMI in Python with Futures

On this page:

Basic Asynchronous API in Python

Consider the following simple Slice definition:

Slice

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

Asynchronous Proxy Methods in Python

In addition to the synchronous proxy method, the Python mapping generates the following asynchronous proxy method:

Python

```
def getNameAsync(self, number, context=None)
```

As you can see, the `getName` operation generates a `getNameAsync` method, which optionally accepts a per-invocation context. `getNameAsync` sends (or queues) an invocation of `getName`, and does not block the calling thread. It returns an instance of `Ice.InvocationFuture` that you can use in a number of ways, including blocking to obtain the result, configuring an action to be executed when the result becomes available, and canceling the invocation.

Here's an example that calls `getNameAsync`:

Python

```
e = EmployeePrx.checkedCast(...)
f = e.getNameAsync(99)

# Continue to do other things here...

name = f.result()
```

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

An asynchronous proxy method uses the same parameter mapping as for synchronous operations; the only difference is that the result (if any) is returned via an `InvocationFuture`. For example, consider the following operation:

Slice

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

The generated code looks like this:

Python

```
def opAsync(self, inp1, inp2, context=None)
```

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

Python

```
p.opAsync(42, "value for inp2").add_done_callback(lambda future: ret, outp1, outp2 = future.result())
```

As with the synchronous mapping, an operation that returns multiple values supplies its result as a tuple. The completion callback, in this case a lambda function, receives the future as its argument and extracts the values from the result tuple.

Asynchronous Exception Semantics in Python

If an invocation raises an exception, the exception can be obtained from the future in several ways:

- Call `result` on the future; `result` raises the exception directly
- Call `exception` on the future; `exception` returns the exception object

The exception is provided by the future, 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 future (instead of being present twice, once where the `opAsync` method is called, and again where the future 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.

`Future` Classes in Python

Ice provides two future classes: `Ice.Future` and `Ice.InvocationFuture`. Asynchronous proxy invocations return an instance of `InvocationFuture`, which is a subclass of `Future`. The API for `Ice.Future` is similar to that of Python's `asyncio.Future` and `concurrent.futures.Future` classes, while `InvocationFuture` adds some Ice-specific methods that clients may find useful.

Python

```
class Future(...):
    def cancel(self)
    def cancelled(self)
    def running(self)
    def done(self)

    def add_done_callback(self, fn)

    def result(self, timeout=None)
    def exception(self, timeout=None)

    def set_result(self, result)
    def set_exception(self, ex)

    def completed(result)
    completed = staticmethod(completed)

class InvocationFuture(Future):
    def add_done_callback_async(self, fn)

    def is_sent(self)
    def is_sent_synchronously(self)
    def add_sent_callback(self, fn)
    def add_sent_callback_async(self, fn)
    def sent(self, timeout=None)
    def set_sent(self, sentSynchronously)

    def communicator(self)
    def connection(self)
    def proxy(self)
    def operation(self)
```

The `Future` methods have the following semantics:

- `cancel(self)`
  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. `cancel` is a local operation and has no effect on the server. A canceled invocation is considered to be completed, meaning `done` return s true, and the result of the invocation is an `Ice.InvocationCanceledException`.

- `cancelled(self)`
  This method returns frue if the invocation was cancelled via a call to `cancel`, or false otherwise.

- `running(self)`
  This method returns true if the invocation has not yet completed or been cancelled, or false otherwise.

- `done(self)`
  This method returns true if the invocation has completed (either successfully or exceptionally) or has been cancelled, or false otherwise.

- `add_done_callback(self, fn)`
  This method registers a callback to be executed when the invocation completes, either successfully or exceptionally. The callback function receives the future as its only argument. If the invocation is already completed at the time `add_done_callback` is called, the callback method is invoked recursively from the calling thread, otherwise the callback method is invoked in the thread that completes the invocation.

- `result(self, timeout=None)`
  This method returns the result of the invocation. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to complete and raises `Ice.TimeoutException` if the timeout expires without completion. If no timeout is provided, the method blocks indefinitely. If the invocation completes with an exception, the method raises the exception directly. For a Slice operation declared with a `void` return type, the method returns `None` upon successful completion.

- `exception(self, timeout=None)`
  This method returns the exception that completed the invocation, or `None` if the invocation completed successfully. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to complete and raises `Ice.TimeoutException` if the timeout expires without completion. If no timeout is provided, the method blocks indefinitely.

- `set_result(self, result)`
  This method completes the invocation successfully using the given result. Calling this method has no effect if the invocation is already completed.

- `set_exception(self, ex)`
  This method completes the invocation exceptionally using the given exception. Calling this method has no effect if the invocation is already completed.

- `completed(result)`
  This static convenience method returns an instance of `Ice.Future` that is already completed successfully with the given result.

The `InvocationFuture` methods have the following semantics:

- `add_done_callback_async(self, fn)`
  This method's semantics differ from that of `add_done_callback` in the situation where the future is already completed. When you call `add_done_callback_async` and the future is already completed, the callback will be invoked by an Ice thread (or by a dispatcher if one is configured).

- `is_sent(self)`
  When you call an asynchronous proxy method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `is_sent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued or an exception occurred before the request could be sent, `is_sent` returns false.

- `is_sent_synchronously(self)`
  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `is_sent_synchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- `add_sent_callback(self, fn)`
  This method registers a callback to be executed when the invocation has been sent. The callback function receives two arguments: the future object and a boolean indicating whether the invocation was sent synchronously. If the invocation is already sent at the time `add_sent_callback` is called, the callback method is invoked recursively from the calling thread. Otherwise, the callback method is invoked by an Ice thread (or by a dispatcher if one is configured).

- `add_sent_callback_async(self, fn)`
  This method's semantics differ from that of `add_sent_callback` in the situation where the invocation is already sent. When you call `add_sent_callback_async` and the invocation is already sent, the callback will be invoked by an Ice thread (or by a dispatcher if one is configured).

- `sent(self, timeout=None)`
  This method waits for the invocation to be sent and returns a boolean indicating whether the invocation was sent synchronously. If an optional timeout is provided, the method will block for up to the given timeout waiting for the invocation to be sent and raises `Ice.TimeoutException` if the timeout expires beforehand. If no timeout is provided, the method blocks indefinitely. If the invocation completes with an exception, the method raises the exception directly.

- `set_sent(self, sentSynchronously)`
  This method marks the invocation as sent, and the boolean argument indicates whether it was sent synchronously.

- `communicator(self)`
  This method returns the communicator that sent the invocation.

- `connection(self)`
  This method returns the connection that was used for the invocation. Note that, for typical asynchronous proxy invocations, this method returns a nil value because the possibility of automatic retries means the connection that is currently in use could change unexpectedly. The `getConnection` method only returns a non-nil value when the `AsyncResult` object is obtained by calling `begin_flushBatchRequests` on a `Connection` object.

- `proxy(self)`
  This method returns the proxy that was used to call the asynchronous proxy method, or `None` if the future was not obtained via an asynchronous proxy invocation.

- `operation(self)`
  This method returns the name of the operation.

Python 3.5 Features

Ice's future types provide some additional features when using Python 3.5 or later.

`asyncio` Integration

The `Ice.wrap_future` function wraps an Ice future object with an instance of `asyncio.Future`. The function accepts an `Ice.Future` object and returns an `asyncio.Future` object. Since `Ice.Future` objects support use in multi-threaded applications, `wrap_future` ensures that the resulting `asyncio.Future` object is completed in a thread-safe manner.

Awaitable Objects

`Ice.Future` is an *awaitable* object, meaning an instance can be used as the target of the `await` keyword. Note however that your chosen event loop implementation must also support `Ice.Future` objects. For example, attempting to call `await` on an `Ice.Future` while using the `asyncio` event loop will result in an error because `asyncio`'s event loop doesn't support "foreign" future types.

One situation where `Ice.Future` objects can be awaited is in a servant dispatch method that is implemented as a coroutine.

Polling for Completion in Python

The `InvocationFuture` methods allow you to poll for call completion. Polling is 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:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0
while not file.eof():
    bytes = file.read(chunkSize)  # Read a chunk
    ft.send(offset, bytes)        # Send the chunk
    offset += len(bytes.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:

Python

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0

results = []
numRequests = 5

while not file.eof():
    bytes = file.read(chunkSize) # Read a chunk

    # Send up to numRequests + 1 chunks asynchronously.
    f = ft.sendAsync(offset, bytes)
    offset += len(bytes)
    # Wait until this request has been passed to the transport.
    f.sent()
    results.append(f)

    # Once there are more than numRequests, wait for the least
    # recent one to complete.
    while len(results) > numRequests:
        f = results[0]
        del results[0]
        f.result()

# Wait for any remaining requests to complete.
while len(results) > 0:
    f = results[0]
    del results[0]
    f.result()
```

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 Python

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 proxy method on a oneway proxy for an operation that returns values or raises a user exception, the method throws `TwowayOnlyException`.

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

Flow Control in Python

Asynchronous method invocations never block the thread that calls the `begin_` 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. (In that case, `InvocationFuture.is_sent_synchronously` returns true.) 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. (In that case, `InvocationFuture.is_sent_synchronously` returns false.)

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.

You can supply a sent callback to be notified when the request was successfully sent:

Python

```
def sentCallback(future, sentSynchronously):
   # The request was sent, send another!

proxy = ...

future = proxy.doSomethingAsync()
future.add_sent_callback(sentCallback)
```

The `add_sent_callback` method has the following semantics:

- If the Ice run time was able to pass the entire request to the local transport immediately, the action will be invoked from the current thread and the `sentSynchronously` argument will be true.
- If Ice wasn't able to write the entire request without blocking, the action will eventually be invoked from an Ice thread pool thread and the `sentSynchronously` argument will be false.

Asynchronous Batch Requests in Python

You can invoke operations via batch 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 proxy method on a oneway proxy for an operation that returns values or raises a user exception, the method throws `TwowayOnlyException`.

The future returned for a batch oneway invocation is always completed and indicates the successful queuing of the batch invocation. The future completes exceptionally if an error occurs before the request is queued.

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 an asynchronous version of this method so you can flush batch requests asynchronously.

`ice_flushBatchRequestsAsync` is a proxy method that flushes any batch requests queued by that proxy, without blocking the calling thread.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `InvocationFuture.connection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

Concurrency Semantics for AMI in Python

For the `InvocationFuture` returned by an asynchronous proxy method, the Ice run time invokes `set_result` or `set_exception` from an Ice thread pool thread. When you register an action with `add_done_callback`, the thread in which your action executes depends on the completion status of the future. If the future is already complete at the time you call `add_done_callback`, the callback function will be invoked immediately in the calling thread. If the future is not yet complete when you call `add_done_callback`, the action will eventually execute in an Ice thread pool thread.

The semantics are slightly different when you register an action with `add_done_callback_async`: the action is always executed in an Ice thread pool thread regardless of the completion status of the future at the time of the call.

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

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

See Also

- Python Mapping for Operations
- Request Contexts
- Batched Invocations