Asynchronous Method Dispatch (AMD) in Python

Previous Next

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool. If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server can provide its results to the Ice run time for delivery to the client.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

On this page:

AMD Mapping in Python

Annotating operations with `["amd"]` metadata directives has no effect in the Python mapping. In fact, the mappings for synchronous and asynchronous dispatch are nearly identical, with the only difference being the return type: the operation has asynchronous semantics if you return a future, otherwise the operation has synchronous semantics. The parameter passing rules for in parameters are the same in both cases.

An asynchronous implementation will normally return an instance of `Ice.Future`. However, Ice also accepts any other future type that provides an `add_done_callback` method, such as `asyncio.Future` or `concurrent.futures.Future`. Ice registers its own completion callback with the future so that, upon completion of the invocation, Ice can marshal the results or exception.

> The implementation is responsible for ensuring that all futures complete successfully or exceptionally. Neglecting to complete a future can cause the client's invocation to hang indefinitely.

Consider the following operation:

Slice

```
interface Test
{
    ["amd"] int foo(short s, out long l);
}
```

We can implement operation `foo` as follows:

Python

```
class TestI(Test):
    def foo(s, current=None):
        if s < 5:
            return (1, 2) # Synchronous dispatch
        else:
            f = Ice.Future()
            # Asynchronous dispatch, e.g., queue the request, start a thread, etc.
            # We eventually need to complete this future, such as with
            # f.set_result((1, 2))
            return f
```

Unlike previous versions of the Python mapping, the name of the dispatch method does not use an AMD-specific suffix; the method name is the same regardless of whether you intend to use synchronous or asynchronous dispatch. The implementation can also use both styles, as shown in the example above. If the implementation returns something other than a future, including `None` for an operation that returns no values, Ice assumes the operation has completed successfully and marshals the results immediately.

The `Ice.Future` class accepts a single value as its result. If a Slice operation returns multiple values, including the return value and all out parameters, they must be supplied as a tuple to `set_result`. The semantics are identical to those for synchronous dispatch.

Back to Top ^

AMD Thread Safety in Python

As with the synchronous mapping, you can add the `marshaled-result` metadata to operations that return mutable types in order to avoid potential thread-safety issues. Your asynchronous operation can then return a future whose result is *Op*`MarshaledResult`.

Back to Top ^

AMD Exceptions in Python

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).

> These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

Although we recommend that the future be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the future. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Back to Top ^

AMD Example in Python

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

Slice

```
module Demo
{
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {}

    interface Model
    {
        Grid interpolate(Grid data, float factor)
            throws RangeError;
    }
}
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo.Model` and supplies a definition for the `interpolate` method that creates a `Job` to hold the future and arguments, and adds the `Job` to a queue. The method uses a lock to guard access to the queue:

Python

```
class ModelI(Demo.Model):
    def __init__(self):
        self._mutex = threading.Lock()
        self._jobs = []

    def interpolate(self, data, factor, current=None):
        with self._mutex:
            f = Ice.Future()
            self._jobs.append(Job(f, data, factor))
            return f
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

Python

```
class Job(object):
    def __init__(self, future, grid, factor):
        self._future = future
        self._grid = grid
        self._factor = factor

    def execute(self):
        if not self.interpolateGrid():
            self._future.set_exception(Demo.RangeError())
            return
        self._future.set_result(self._grid)

    def interpolateGrid(self):
        # ...
```

If `interpolateGrid` returns false, then we complete the future using `set_exception` to indicate that a range error has occurred. If interpolation was successful, we send the modified grid back to the client by calling `set_result` on the future.

Back to Top ^

Chaining Asynchronous Invocations in Python

Given that asynchronous proxy invocations return futures, and asynchronous dispatch methods return futures, it becomes quite easy to chain together a sequence of calls under the right circumstances. Specifically, the operations being chained must have the same result types and compatible user exception specifications.

Continuing our example from the previous section, suppose our `Model` implementation delegates its requests to an internal server:

Python

```
class ModelI(Demo.Model):
    def __init__(self, internalModel):
        self._internalModel = internalModel

    def interpolate(self, data, factor, current=None):
        return self._internalModel.interpolateAsync(data, factor)
```

The constructor receives a proxy for the internal model server, and the implementation of `interpolate` simply returns the future created by the asynchronous proxy invocation.

Back to Top ^

Using Coroutines in Python

For applications using Python 3.5 or later, a Slice operation may optionally be implemented as a coroutine, allowing you to use the `await` keyword to suspend processing while waiting for subtasks to complete. Again using our interpolation example, suppose we need to process the data grid in stages:

Python

```
class ModelI(Demo.Model):
    def __init__(self, internalModel):
        self._internalModel = internalModel

    # Implement as a coroutine
    async def interpolate(self, data, factor, current=None):
        # Stage 1
        data = await self._internalModel.interpolateAsync(data, factor)
        # Stage 2 with new factor
        return await self._internalModel.interpolateAsync(data, factor * 2)
```

Ice automatically detects a dispatch method that is implemented as a coroutine and ensures it runs to completion. A coroutine may await on futures; Ice restarts the coroutine when the future completes and passes the future's result.

Back to Top ^

See Also

- User Exceptions
- Run-Time Exceptions
- Asynchronous Method Invocation (AMI) in Python
- The Ice Threading Model

Previous

Next