# Asynchronous Method Dispatch (AMD) in Python

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 sends a response explicitly using a callback object provided by the Ice run time.

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 (i.e., the callback object and operation arguments) 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.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

On this page:



## Enabling AMD with Metadata in Python

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

**Slice**

```
["amd"] interface I {
    bool isValid();
    float computeRate();
};

interface J {
    ["amd"] void startProcess();
    int endProcess();
};
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

## AMD Mapping in Python

For each AMD operation, the Python mapping emits a dispatch method with the same name as the operation and the suffix `_async`. This method returns `None`. The first parameter is a reference to a callback object, as described below. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

The callback object defines two methods:

- `def ice_response(self, <params>)`
  The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return

type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.

- `def ice_exception(self, ex)`
  The `ice_exception` method allows the server to report an exception.

Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

Suppose we have defined the following operation:

**Slice**

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

The callback interface generated for operation `foo` is shown below:

**Python**

```
class ...
    #
    # Operation signatures.
    #
    # def ice_response(self, _result, l)
    # def ice_exception(self, ex)
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

**Python**

```
def foo_async(self, __cb, s)
```

# 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 callback object 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 callback object. 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.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated and local exceptions may undergo translation.

# 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 {
        ["amd"] 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_async` method that creates a `Job` to hold the callback object 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_async(self, cb, data, factor, current=None):
        self._mutex.acquire()
        try:
            self._jobs.append(Job(cb, data, factor))
        finally:
            self._mutex.release()
```

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, cb, grid, factor):
        self._cb = cb
        self._grid = grid
        self._factor = factor

    def execute(self):
        if not self.interpolateGrid():
            self._cb.ice_exception(Demo.RangeError())
            return
        self._cb.ice_response(self._grid)

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

If `interpolateGrid` returns `False`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

### See Also

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