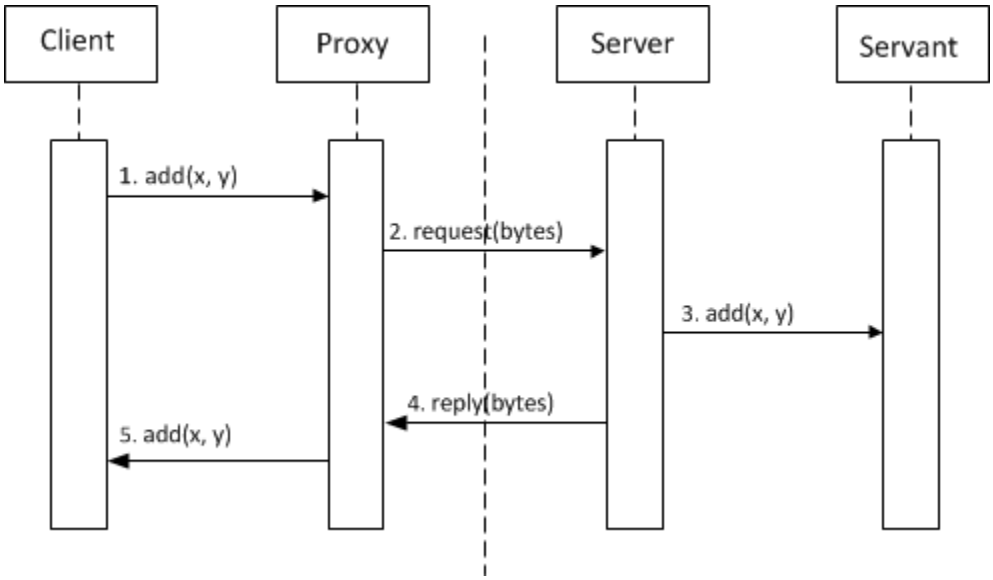
Dynamic Invocation and Dispatch Overview

On this page:

Use Cases for Dynamic Invocation and Dispatch

Ice applications generally use the static invocation model, in which the application invokes a Slice operation by calling a member function on a generated proxy class. In the server, the static dispatch model behaves similarly: the request is dispatched to the servant as a statically-typed call to a member function. Underneath this statically-typed facade, the Ice run times in the client and server are exchanging sequences of bytes representing the encoded request arguments and results. These interactions are illustrated below:



Interactions in a static invocation.

1. The client initiates a call to the Slice operation `add` by calling the member function `add` on a proxy.
2. The generated proxy class marshals the arguments into a sequence of bytes and transmits them to the server.
3. In the server, the generated servant class unmarshals the arguments and calls `add` on the subclass.
4. The servant marshals the results and returns them to the client.
5. Finally, the client's proxy unmarshals the results and returns them to the caller.

The application is blissfully unaware of this low-level machinery, and in the majority of cases that is a distinct advantage. In some situations, however, an application can leverage this machinery to accomplish tasks that are not possible in a statically-typed environment. Ice provides the dynamic invocation and dispatch models for these situations, allowing applications to send and receive requests as encoded sequences of bytes instead of statically-typed arguments.

The dynamic invocation and dispatch models offer several unique advantages to Ice services that forward requests from senders to receivers, such as Glacier2 and IceStorm. For these services, the request arguments are an opaque byte sequence that can be forwarded without the need to unmarshal and remarshal the arguments. Not only is this significantly more efficient than a statically-typed implementation, it also allows intermediaries such as Glacier2 and IceStorm to be ignorant of the Slice types in use by senders and receivers.

Another use case for the dynamic invocation and dispatch models is scripting language integration. The Ice extensions for Python, PHP, and Ruby invoke Slice operations using the dynamic invocation model; the request arguments are encoded using the streaming interfaces.

It may be difficult to resist the temptation of using a feature like dynamic invocation or dispatch, but we recommend that you carefully consider the risks and complexities of such a decision. For example, an application that uses the streaming interface to manually encode and decode request arguments has a high risk of failure if the argument signature of an operation changes. In contrast, this risk is greatly reduced in the static invocation and dispatch models because errors in a strongly-typed language are found early, during compilation. Therefore, we caution you against using this capability except where its advantages significantly outweigh the risks.

Dynamic Invocation using `ice_invoke`

Dynamic invocation is performed using the proxy member function `ice_invoke`, defined in the proxy base class `ObjectPrx`. If we were to define the function in Slice, it would look like this:

Slice

```
sequence<byte> ByteSeq;

bool ice_invoke(
    string operation,
    Ice::OperationMode mode,
    ByteSeq inParams,
    out ByteSeq outParams
);
```

The first argument is the name of the Slice operation.

> This is the Slice name of the operation, not the name as it might be mapped to any particular language. For example, the string `"while"` is the name of the Slice operation `while`, and not `"_cpp_while"` (C++) or `"_while"` (Java).

The second argument is an enumerator from the Slice type `Ice::OperationMode`; the possible values are `Normal` and `Idempotent`. The third argument, `inParams`, represents an encapsulation of the encoded in-parameters of the operation.

A return value of `true` indicates a successful invocation, in which case an encapsulation of the marshaled form of the operation's results (if any) is provided in `outParams`. A return value of `false` signals the occurrence of a user exception whose encapsulated data is provided in `outParams`. The caller must also be prepared to catch local exceptions, which are thrown directly.

Note that the Ice run time currently does not support the use of collocation optimization in dynamic invocations. Attempting to call `ice_invoke` on a proxy that is configured to use collocation optimization raises `CollocationOptimizationException`. See Location Transparency for more information on this optimization and instructions for disabling it.

Dynamic Dispatch using `Blobject`

A server enables dynamic dispatch by creating a subclass of `Blobject` (the name is derived from *blob*, meaning a blob of bytes). The Slice equivalent of `Blobject` is shown below:

Slice

```
sequence<byte> ByteSeq;

interface Blobject {
    bool ice_invoke(ByteSeq inParams, out ByteSeq outParams);
};
```

The `inParams` argument supplies an encapsulation of the encoded in-parameters. The contents of the `outParams` argument depends on the outcome of the invocation: if the operation succeeded, `ice_invoke` must return `true` and place an encapsulation of the encoded results in `outParams`; if a user exception occurred, `ice_invoke` must return `false`, in which case `outParams` contains an encapsulation of the encoded exception. The operation may also raise local exceptions such as `OperationNotExistException`.

The language mappings add a trailing argument of type `Ice::Current` to `ice_invoke`, and this provides the implementation with the name of the operation being dispatched.

Because `Blobject` derives from `Object`, an instance is a regular Ice servant just like instances of the classes generated for user-defined Slice interfaces. The primary difference is that all operation invocations on a `Blobject` instance are dispatched through the `ice_invoke` member function.

If a `Blobject` subclass intends to decode the in-parameters (and not simply forward the request to another object), then the implementation obviously must know the signatures of all operations it supports. How a `Blobject` subclass determines its type information is an implementation detail that is beyond the scope of this manual.

Note that a `Blobject` servant is also useful if you want to create a message forwarding service, such as Glacier2. In this case, there is no need to decode any parameters; instead, the implementation simply forwards each request unchanged to a new destination. You can register a `Blobject` servant as a default servant to easily achieve this.

See Also

- Location Transparency
- The Current Object
- Default Servants
- Streaming Interfaces
- Glacier2
- IceStorm