Batched Invocations

Previous

Oneway and datagram invocations are normally sent as individual messages, that is, the Ice run time sends the oneway or datagram invocation to the server immediately, as soon as the client makes the call. If a client sends a number of oneway or datagram invocations in succession, the client-side run time traps into the OS kernel for each message, which is expensive. In addition, each message is sent with its own message header, that is, for *N* message s, the bandwidth for *N* message headers is consumed. In situations where a client sends a number of oneway or datagram invocations, the additional overhead can be considerable.

To avoid the overhead of sending many small messages, you can send oneway and datagram invocations in a batch: instead of being sent as a separate message, a batch invocation is placed into a client-side buffer by the Ice run time. Successive batch invocations are added to the buffer and accumulated on the client side until they are flushed, either explicitly by the client or automatically by the Ice run time.

On this page:

Proxy Methods for Batched Invocations

Several proxy methods support the use of batched invocations. In Slice, these methods would look as follows:

Slice

```
Object* ice_batchOneway();
Object* ice_batchDatagram();
void ice_flushBatchRequests();
```

The `ice_batchOneway` and `ice_batchDatagram` methods create a new proxy configured for batch invocations. Once you obtain a batch proxy, messages sent via that proxy are buffered by the proxy instead of being sent immediately. Once the client has invoked one or more operations on a batch proxy, it can call `ice_flushBatchRequests` to explicitly flush the batched invocations. This causes the batched messages to be sent "in bulk", preceded by a single message header. On the server side, batched messages are dispatched by a single thread, in the order in which they were written into the batch. This means that messages from a single batch cannot appear to be reordered in the server. Moreover, either all messages in a batch are delivered or none of them. (This is true even for batched datagrams.)

Asynchronous versions of `ice_flushBatchRequests` are also available; see the relevant language mapping for more information.

Batched invocations are queued by the proxy on which the request was invoked (this is true for all proxies except fixed proxies). It's important to be aware of this behavior for several reasons:

- Batched invocations queued on a proxy will be lost if that proxy is deallocated prior to being flushed
- Proxy instances maintain separate queues even if they refer to the same target object
- Using proxy factory methods may (or may not) create new proxy instances, which affects how batched invocations are queued. Consider this example:

  C++11

  ```
  communicator = ...;
  auto batch = communicator->stringToProxy("...")->ice_batchOneway(); // Creates a batch oneway proxy
  auto secureBatch = batch->ice_secure(); // Might create a new proxy instance
  batch->ice_ping();
  secureBatch->ice_ping();
  batch->ice_flushBatchRequests(); // Might also flush requests on secureBatch
  ```

  At run time, the `batch` and `secureBatch` variables might refer to the same proxy instance or two separate proxy instances, depending on the original stringified proxy's configuration. As a result, flushing requests using one variable may or may not flush the requests of the other.

Calling `ice_flushBatchRequests` on a proxy behaves like a oneway invocation in that automatic retries take place and an exception is raised if an error occurs while establishing a connection or sending the batch message.

Back to Top ^

Automatically Flushing Batched Invocations

The default behavior of the Ice run time, as governed by the configuration property `Ice.BatchAutoFlushSize`, automatically flushes batched invocations as soon as a batched request causes the accumulated message to exceed the specified limit. When this occurs, the Ice run time immediately flushes the existing batch of requests and begins a new batch with this latest request as its first element.

For batched oneway invocations, the value of `Ice.BatchAutoFlushSize` specifies the maximum message size in kilobytes; the default value is 1MB. In the case of batched datagram invocations, the maximum message size is the smaller of the system's maximum size for datagram packets and the value of `Ice.BatchAutoFlushSize`.

The receiver's setting for `Ice.MessageSizeMax` determines the maximum size that the Ice run time will accept for an incoming protocol message. The sender's setting for `Ice.BatchAutoFlushSize` must not exceed this limit, otherwise the receiver will silently discard the entire batch.

Automatic flushing is enabled by default as a convenience for clients to ensure a batch never exceeds the configured limit. A client can track batch request activity, and even implement its own auto-flush logic, by installing an interceptor.

Back to Top ^

Batched Invocations for Fixed Proxies

A *fixed proxy* is a special form of proxy that an application explicitly creates for use with a specific bidirectional connection. Batched invocations on a fixed proxy are not queued by the proxy, as is the case for regular proxies, but rather by the connection associated with the fixed proxy. Automatic flushing continues to work as usual for batched invocations on fixed proxies, and you have three options for manually flushing:

- Calling `ice_flushBatchRequests` on a fixed proxy flushes all batched requests queued by its connection; this includes batched requests from other fixed proxies that share the same connection
- Calling `Connection::flushBatchRequests` flushes all batched requests queued by the target connection
- Calling `Communicator::flushBatchRequests` flushes all batched requests on all connections associated with the target communicator

`Connection::flushBatchRequests` and `Communicator::flushBatchRequests` have no effect on batched requests queued by regular (non-fixed) proxies.

The synchronous versions of `flushBatchRequests` block the calling thread until the batched requests have been successfully written to the local transport. To avoid the risk of blocking, you must use the asynchronous versions instead (assuming they are supported by your chosen language mapping).

Note the following limitations in case a connection error occurs:

- Any requests queued by that connection are lost
- Automatic retries are not attempted
- The proxy method `ice_flushBatchRequests` and `Connection::flushBatchRequests` will raise an exception, but `Communicator::flushBatchRequests` ignores any errors

The `Connection::flushBatchRequests` and `Communicator::flushBatchRequests` methods take an `Ice::CompressBatch` argument to specify under which conditions the batch should be compressed or not. The `Ice::CompressBatch` enumeration is shown below:

slice

```
["cpp:scoped", "objc:scoped"]
local enum CompressBatch
{
    Yes,
    No,
    BasedOnProxy
}
```

You can choose to always or never compress the batch with `Yes` or `No`. The `BasedOnProxy` value specifies to compress based on the proxies used to add requests to the batch. If at least one request was queued with a compressed fixed proxy (a proxy created with `ice_compress(true)` or if `Ice.Override.Compress` is enabled), the batch will be compressed.

Back to Top ^

Considerations for Batched Datagrams

For batched datagram invocations, you need to keep in mind that, if the data for the invocations in a batch substantially exceeds the PDU size of the network, it becomes increasingly likely for an individual UDP packet to get lost due to fragmentation. In turn, loss of even a single packet causes the entire batch to be lost. For this reason, batched datagram invocations are most suitable for simple interfaces with a number of operations that each set an attribute of the target object (or interfaces with similar semantics). Batched oneway invocations do not suffer from this risk because they are sent over stream-oriented transports, so individual packets cannot be lost.

If automatic flushing is enabled, Ice's default behavior uses the smaller of `Ice.BatchAutoFlushSize` and `Ice.UDP.SndSize` to determine the maximum size for a batch datagram message.

Back to Top ^

Compressing Batched Invocations

Batched invocations are more efficient if you also enable compression for the transport: many isolated and small messages are unlikely to compress well, whereas batched messages are likely to provide better compression because the compression algorithm has more data to work with.

> Regardless of whether you used batched messages or not, you should enable compression only on lower-speed links. For high-speed LAN connections, the CPU time spent doing the compression and decompression is typically longer than the time it takes to just transmit the uncompressed data.

Back to Top ^

Active Connection Management and Batched Invocations

As for oneway invocations, server-side Active Connection Management (ACM) can interfere with batched invocations over TCP or TCP-based transports (SSL, WebSocket, etc.). With server-side ACM enabled, it's possible for a server to close the connection at the wrong moment and not process a batch – with no indication being returned to the client that the batch was lost. We recommend that you either disable ACM for the server side, or enable ACM heartbeats in the client to ensure the connection remains active.

Back to Top ^

Batched Invocation Interceptors

Batch invocation interceptors allow you to implement your own auto-flush algorithm or receive notification when an auto-flush fails.

C++11

```
namespace Ice
{
    class BatchRequest
    {
    public:
        virtual void enqueue() const = 0;
        virtual int getSize() const = 0;
        virtual const std::string& getOperation() const = 0;
        virtual const std::shared_ptr<Ice::ObjectPrx>& getProxy() const = 0;
    };

    struct InitializationData
    {
        ...
        std::function<void(const Ice::BatchRequest& req, int count, int size)> batchRequestInterceptor;
    };
}
```

C++98

```
namespace Ice
{
    class BatchRequest
    {
    public:
        virtual void enqueue() const = 0;
        virtual int getSize() const = 0;
        virtual const std::string& getOperation() const = 0;
        virtual const Ice::ObjectPrx& getProxy() const = 0;
    };

    class BatchRequestInterceptor : public IceUtil::Shared
    {
    public:
        virtual void enqueue(const BatchRequest&, int, int) = 0;
    };

    struct InitializationData
    {
        ...
        BatchRequestInterceptorPtr batchRequestInterceptor;
    };
}
```

C#

```
namespace Ice
{
    public interface BatchRequest
    {
        void enqueue();
        int getSize();
        string getOperation();
        ObjectPrx getProxy();
    }

    public final class InitializationData
    {
    ...
        public System.Action<BatchRequest, int, int> batchRequestInterceptor;
    }
}
```

Java

```
package com.zeroc.Ice;

public interface BatchRequest
{
    void enqueue();
    int getSize();
    String getOperation();
    ObjectPrx getProxy();
}

@FunctionalInterface
public interface BatchRequestInterceptor
{
    void enqueue(BatchRequest request, int queueBatchRequestCount, int queueBatchRequestSize);
}

public final class InitializationData
{
    ...
    public BatchRequestInterceptor batchRequestInterceptor;
}
```

Java Compat

```
package Ice;

public interface BatchRequest
{
    void enqueue();
    int getSize();
    String getOperation();
    ObjectPrx getProxy();
}
public interface BatchRequestInterceptor
{
    void enqueue(BatchRequest request, int queueBatchRequestCount, int queueBatchRequestSize);
}

public final class InitializationData
{
    ...
    public BatchRequestInterceptor batchRequestInterceptor;
}
```

ObjC

```
@protocol ICEBatchRequest <NSObject>
-(void) enqueue;
-(int) getSize;
-(NSString*) getOperation;
-(id<ICEObjectPrx>) getProxy;
@end

@interface ICEInitializationData : NSObject
...
@property(copy, nonatomic) void(^batchRequestInterceptor)(id<ICEBatchRequest>, int, int);
@end
```

Python

```
class BatchRequest(object):
    def getSize():
        ...
    def getOperation():
        ...

    def getProxy():
        ...

    def enqueue():
        ...

initData = Ice.InitializationData()
initData.batchRequestInterceptor = lambda req, count, size: ...
```

You install an interceptor by setting the `batchRequestInterceptor` member of the `InitializationData` object that the application constructs when initializing a new communicator. The Ice run time invokes the interceptor for each batch request, passing the following arguments:

- `req` - An object representing the batch request being queued
- `count` - The number of requests currently in the queue
- `size` - The number of bytes consumed by the requests currently in the queue

The request represented by `req` is not included in the `count` and `size` figures.

A batch request is not queued until the interceptor calls `BatchRequest::enqueue`. The minimal interceptor implementation is therefore:

C++11

```
initData.batchRequestInterceptor = [](const Ice::BatchRequest& req, int count, int size)
{
    req.enqueue();
};
```

A more sophisticated implementation might use its own logic for automatically flushing queued requests:

C++11

```
int limit = initData.properties->getPropertyAsInt("Ice.BatchAutoFlushSize");
initData.batchRequestInterceptor = [limit](const Ice::BatchRequest& req, int count, int size)
{
    if(size + req.getSize() > limit)
    {
        req.getProxy()->ice_flushBatchRequestsAsync([=](const Ice::Exception& ex) { /* Handle error */ });
    }
    req.enqueue();
};
```

In this example, the implementation consults the existing Ice property `Ice.BatchAutoFlushSize` to determine the limit that triggers an automatic flush. If a flush is necessary, the interceptor can obtain the relevant proxy by calling `getProxy` on the `BatchRequest` object.

Specifying your own exception handler when calling `ice_flushBatchRequestsAsync` gives you the ability to take action if a failure occurs (Ice's default automatic flushing implementation ignores any errors). Aside from logging a message, your options are somewhat limited because it's not possible for the interceptor to force a retry.

For datagram proxies, we strongly recommend using a maximum queue size that is smaller than the network MTU to minimize the risk that datagram fragmentation could cause an entire batch to be lost.

Back to Top ^

See Also

- Oneway Invocations
- Datagram Invocations
- Communicators
- Using Connections
- The Ice Protocol
- Protocol Compression
- Active Connection Management

Previous