Batched Invocations

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* messages, 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 in the client-side run time 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.

Automatically Flushing Batched Invocations

The default behavior of the Ice run time, as governed by the configuration property `Ice.BatchAutoFlush`, automatically flushes batched invocations as soon as a batched request causes the accumulated message to exceed the maximum allowable size. 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 maximum message size is established by the property `Ice.MessageSizeMax`, which defaults to 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.MessageSizeMax`.

A client that sends batch requests cannot determine the size of the message that the Ice run time is accumulating for it; automatic flushing is enabled by default as a convenience for clients that unknowingly exceed the maximum message size. A client that requires more deterministic behavior should flush batched requests explicitly at regular intervals.

Flushing Batched Invocations for Communicators and Connections

The `Communicator` and `Connection` interfaces support synchronous and asynchronous versions of `flushBatchRequests`. As you might expect, the `Connection::flushBatchRequests` operation flushes all batch requests queued for a particular connection, and the `Communicator::flushBatchRequests` operation flushes the batch requests of every connection created by a communicator.

The synchronous versions of `flushBatchRequests` block the calling thread until the batch 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 also that the asynchronous version of `Communicator::flushBatchRequests` never raises an exception, even if an error occurs while flushing one of its connections.

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.

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.

Active Connection Management and Batched Invocations

As for oneway invocations, you should disable server-side Active Connection Management (ACM) when using batched invocations over TCP/IP or SSL. With server-side ACM enabled, it is 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).

See Also

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