Connections

ZeroC's Newsletter for the Ice Community

Issue Number 13, April 2006

Community Spirit

A few weeks ago, I was involved in a discussion of what makes software viable. Predictably, the word "standardization" popped up yet again. Clearly, standardization is a good thing. Just think of the success of the Single Unix Specification published by the Open Group, or the importance of the Request for Comments published by the Internet Engineering Task Force. However, not everything is standardized. For examples, look at the Windows world: there are some things that are standardized (such as C#, which is standardized by ECMA) but, by and large, the Windows standard is whatever the latest version of Windows happens to be doing at the time.

What I found interesting in the discussion was that standards were not lauded because they make things so much easier or better (although they often do). Instead, many of the arguments were driven by fear: *"If my vendor starts charging too much…"*, *"I need to protect my investment"*, and *"No-one ever gets fired for buying IBM or Microsoft"* were recurring phrases. There was a strong theme that, without a standard, software is somehow less viable than with a standard. (For a counter-example, again look at the Windows world, which most people would agree is an eminently viable one.)

All this got me to thinking about what makes software viable. Standardization is one contributing factor, but there are others. Technical excellence, ease of use, price, availability, documentation, support quality, feature set, and marketing all contribute to the viability of software. However, none of these factors seems decisive. (I imagine that most of us know of technically excellent products that have disappeared without a trace; likewise, there are successful products that not many people would call technically excellent…)

So, what makes software viable then? To me, there is one overriding factor that contributes more than all the others combined: software community. Software is viable as long as there is an active and vibrant developer community. Proof of this can be found in many places, but particularly in the open source and shareware world: much open source and shareware ended up being very successful, despite having no standardization, no budget, limited platform support, no professional documentation, patchy support, and no marketing whatsoever (at least not initially). Despite that, the software is viable because it meets a real need and because its users care about it enough to keep using and improving it.

In this vein, it is good to see the ever-increasing activity in our user forum and to see you, the developers using Ice, help each other with information, porting tips, bug hunting, architectural issues, and more. Here at ZeroC, we like this very much. For one, it means that we do not have to do all the heavy lifting by ourselves but, more importantly, it means that Ice enjoys an active and growing user community. And that user community is important, more important than any other factor. So, keep it up—everyone is better off for it.

And, speaking of community, November this year will see the 7th International Middleware Conference take place in Melbourne. I am program co-chair of this conference (and I will also present a tutorial on middleware design). I see the conference as an ideal place to strengthen the Ice community and I encourage you to attend. Keep an eye on the conference web site—the call for papers is about to close, but the conference will host a number of workshops with a later closing date where you can present your middleware experiences, participate in discussions with other developers, and learn new tricks of the trade. I want to get to know more about you, the Ice community, and I want to know about the things you do with Ice. So I hope to see you there, maybe over a beer (or three): software and beer are both better when enjoyed in a community!

Michi Henning
Chief Scientist

Issue Features

Integrating Ice with a GUI: Part II

In the second of a four part series on integrating Ice with a GUI Matthew Newhook introduces a number of improvements on the techniques introduced in the first article.

Ice-E 1.1: What's New?

Brent Eagles and Dwayne Boone introduce some of the new features in Ice-E 1.1, and discuss performance improvements included in the release.

Contents

Integrating Ice with a GUI: Part II

Matthew Newhook, Senior Software Engineer

Introduction

The previous article in this series (*Integrating Ice with a GUI*, Issue 12 of *Connections*) outlined a method to send twoway requests without blocking a UI event processing thread. This article expands on that idea with a number of new features, including the addition of a thread to send oneway requests. Before reading this article, I recommending that you familiarize yourself with the previous article. You should also have a look at the FAQs in Issue 2 of *Connections*—they outline various issues you will come across when sending oneway requests.

Sending Multiple Requests in Parallel

The current call queue maintains a strict ordering of requests, that is, it waits for `Call::execute` to return before executing another call. However, we may want a queue that sends multiple calls in parallel. We will look at two possible ways to implement this.

Asynchronous Invocations

One method for attaining more parallelism is to have the call queue use asynchronous method invocation (AMI) instead of synchronous invocation. With asynchronous invocations, the processing of the reply is decoupled from the sending of the request. This means that the call queue will continue to send requests before replies are received for previously-sent requests. (If you are not familiar with asynchronous invocation, you may want to read *Asynchronous Programming* in Issue 4 of *Connections*.)

Consider three queued requests, A, B, and C. A call queue that uses asynchronous invocations guarantees that the three calls are sent in order, that is, A is sent before B, which is sent before C. However, if all three requests are sent to the same server, due to the vagaries of thread scheduling, there is no guarantee as to the order in which the requests will actually be dispatched by the server. In addition, because each request is sent before the reply for the previous request is received, obviously B cannot depend on anything done by A, and C cannot depend on anything done by B. This means that an asynchronous queue implementation is useful only for "stand-alone" requests that do not depend on side effects of earlier requests.

Here is an example implementation of an asynchronous queue. First, we need to add metadata to the Slice operation to instruct the compiler to generate asynchronous stubs:

```
// Slice
interface Hello
{
    ["ami"] nonmutating void sayHello();
    //...
};
```

Next, we need to implement the AMI callback. This requires implementing two methods, `ice_response` (which is called by the Ice run time for a successful reply) and `ice_exception` (which is called by the Ice run time if the operation raises an exception). Our implementation of `ice_exception` will notify the queue when an operation completes with an exception. To inform the application code, the queue can remember the exception and throw it the next time the application adds a message to the call queue. The easiest way to implement this is to pass the queue to the `Call::execute` method, which can then pass the queue to the AMI object for later use:

```
// C++
class AMI_Hello_sayHelloI :
    public AMI_Hello_sayHello
{
public:
    AMI_Hello_sayHelloI(
        const CallQueuePtr& queue) :
        _queue(queue)
    {
    }

    virtual void
    ice_response()
    {
    }

    virtual void
    ice_exception(const Exception& e)
    {
        _queue->error(e);
    }

private:
    const CallQueuePtr _queue;
};
```

The call implementation uses the asynchronous version of the invocation:

```
// C++
void
SayHelloCall::execute(const CallQueuePtr& queue)
{
    _hello->sayHello_async(
        new AMI_Hello_sayHelloI(queue));
}
```

With this code, `execute` does not wait for the result of the `sayHello` invocation to be returned by the server. Unless something has gone wrong and the AMI call blocks due to slow connection establishment or a TCP/IP buffer overflow, `execute` will

return immediately and the results of the `sayHello` operation will be delivered via `ice_response` or `ice_exception` once they are available.

The call queue itself only needs two changes, namely passing the queue to the `execute` method, and a new `error` method:

```
// C++
void
CallQueue::run()
{
    while(true)
    {
        // ...
        try
        {
            req->execute(this);
        }
        catch(const Exception& e)
        {
            error(e);
        }
    }
}

void
CallQueue::error(const Exception& e)
{
    Lock sync(*this);
    _exception = auto_ptr<Exception>(
        e.ice_clone());
    _destroy = true;
    notify();
}
```

Thread Pool

Another option to attain more parallelism is to use a thread pool. Using a thread pool in the call queue means that the call implementations can use either synchronous or asynchronous invocations and still attain parallelism when calling the server. The call queue maintains a set of worker threads that each wait to remove a call from the queue and then subsequently execute the call. As for the asynchronous implementation, the worker thread notifies the queue if an invocation raises an exception:

```
// C++
class CallQueueWorker : public Thread
{
public:
    CallQueueWorker(const CallQueuePtr&);
    virtual void run();
private:
    CallQueuePtr _queue;
};
typedef Handle<CallQueueWorker>
    CallQueueWorkerPtr;

CallQueueWorker::CallQueueWorker(
    const CallQueuePtr& queue) :
    _queue(queue)
{
}

void
CallQueueWorker::CallQueueWorker::run()
{
    while(true)
    {
        CallPtr req = _queue->next();
        if(!req)
        {
            return;
        }
        try
        {
            req->execute();
        }
        catch(const Exception& e)
        {
            _queue->error(e);
        }
    }
}
```

The worker calls `CallQueue::next` to retrieve the next call to be processed. (In the event that the queue has been destroyed, `next` returns a nil event.) If an invocation raises an exception, the queue is notified via a call to `CallQueue::error`.

The call queue, instead of being a thread itself, now has a list of worker threads. This is managed as follows:

```
// C++
class CallQueue : public Shared,
    public Monitor<Mutex>
{
    // ...
    void join();
    void start();
private:
    friend class CallQueueWorker;
    void error(const Exception&);
    CallPtr next();
    std::list<CallQueueWorkerPtr> _workers;
};

void
CallQueue::start()
{
    Lock sync(*this);
    for(int i = 0; i < 2; ++i)
    {
        CallQueueWorkerPtr worker =
            new CallQueueWorker(this);
        worker->start();
        _workers.push_back(worker);
    }
}
```

```
void
CallQueue::join()
{
    Lock sync(*this);
    list<CallQueueWorkerPtr>::const_iterator p;
    for(p = _workers.begin();
        p != _workers.end();
        ++p)
    {
        (*p)->getThreadControl().join();
    }
    _workers.clear();
}
```

The `start` method starts all worker threads, and `join` waits for the worker threads to terminate. The `join` method calls `clear` on the set of worker threads once it has joined with them; this drops the last reference to each thread and causes its destructor to run.

The `next` method first locks the monitor and then waits for either an event to be added, or the queue to be destroyed. If the queue is not destroyed, it returns the first event from the queue:

```
// C++
CallPtr
CallQueue::next()
{
    Lock sync(*this);
    while(!_destroy && _req.empty())
    {
        wait();
    }
    CallPtr req;
    if(!_destroy)
    {
        req = _req.front();
        _req.pop_front();
    }
    return req;
}
```

With a thread pool, no guarantee for the order in which requests are sent can be made at all. If the application queues up three events A, B, and C, the requests can be dispatched in any order: although the events will be picked up by the threads in the pool in the correct order, the thread scheduler might run the thread that picked up event B and dispatch the event before it schedules the thread that picked up event A. This means that, as for the AMI example, B cannot depend on any side effects of A.

A more flexible implementation of the call queue would be to specify the size of the thread pool when the queue is started. With a pool size of one (as long as the queue does not use AMI), the queue guarantees strict ordering; with a pool size larger than one, the ordering is no longer guaranteed.

You can find a complete example of the AMI and thread pool implementations of the call queue in the source code that accompanies this article.

Oneway Message Queue

Consider an application that regularly sends oneway status messages to a server, as well as twoway messages. If oneway messages are placed into the same queue as twoway messages, the oneway messages may end up being delayed by twoway messages yet to be processed. If the processing time for some twoway is long, this delay may be unacceptable. We can deal with this by dispatching oneway and twoway messages concurrently. An easy way to achieve this is to use separate queues for oneway and twoway calls. (Note that processing concurrent requests in a server requires the server to have as many threads in its thread pool as there are concurrent requests; if more requests arrive than can be concurrently processed, they are delayed and executed as threads for previous requests are returned to the pool.)

Here is how we can use two queues to send oneway and twoway requests concurrently:

```
// C++
twowayQueue = new CallQueue();
twowayQueue->start();
onewayQueue = new CallQueue();
onewayQueue->start();
```

While there is nothing wrong with this code, it ignores an interesting optimization.

Batched Invocations

Ice supports a sending mode known as *batched invocation* for oneway messages. Normally, when a client invokes an operation on a oneway proxy, the Ice run time sends the message to the server immediately. However, sending the message requires a separate write to the network, which is expensive. (Each oneway message also gets its own protocol header, which consumes a bit of bandwidth.) If a client frequently sends small oneway messages, the cost of repeatedly trapping into the kernel and the extra bandwidth can be noticeable.

A solution to this problem is to use batched invocations. Instead of sending each oneway invocation as a separate protocol message, you can group several oneway messages together and send them as a single protocol message. This avoids multiple writes to the network and the single message uses only one protocol header.

To send batched oneway messages, you need to call either `ice_batchOneway` or `ice_batchDatagram` on the proxy for the target object. Either method returns a new proxy to the same object, but that proxy sends oneway invocations in batches, for example:

```
// C++
HelloPrx hello = ...;
HelloPrx batchHello =
    HelloPrx::uncheckedCast(
        hello->ice_batchOneway());
```

Calling on the `batchHello` proxy now sends a batched oneway message.

```
// C++
batchHello->sayHello();
```

Invoking on a batch oneway proxy buffers the invocation inside the Ice run time instead of sending it immediately. To force all batched invocations that have accumulated to be sent, you need to call `Ice::Communicator::flushBatchRequests`:

```
// C++
batchHello->ice_communicator()
    ->flushBatchRequests();
```

This causes all buffered oneway messages to be sent as a single protocol message. On the server side, batched messages provide additional guarantees:

- A single thread will dispatch all oneway messages contained in the batch.
- The messages in the batch will be dispatched in the order they were buffered on the client side.
- All messages in the batch are delivered, or none.

Note that, for batched datagram invocations, you must take care: if the size of the protocol message exceeds the PDU size of the network, it becomes more and more likely for the UDP message to be lost due to fragmentation. Clearly, the more messages are batched together, the more likely this is to happen. Also note that UDP messages are limited to 65507 bytes (65535 is the maximum size of an IP datagram minus 20 bytes for the IP header and 8 bytes for the UDP header), so you must take care to stay below this limit.

If you send a regular oneway datagram message and exceed the maximum UDP message size, the Ice run time sends a `DatagramLimitException`. However, for batched oneway datagram messages that exceed the maximum size, the Ice run time throws away the entire batch, without raising an exception. (However, the run time logs a warning in this case.)

A Batching Call Queue

We can create a call queue that takes advantage of batching. The basic approach is to send all oneway requests in batches and to flush these requests at regular intervals. We can create a new class `OnewayCallQueue` to do this. Its public interface is almost the same as for `CallQueue`. The only difference is that the constructor takes an `Ice::Communicator` argument that we use to flush the batched messages:

```
// C++
class OnewayCallQueue : public Thread, public
Monitor<Mutex>
{
public:
    OnewayCallQueue(const CommunicatorPtr&);

    void add(const CallPtr&);
    void destroy();
    virtual void run();

private:
    const CommunicatorPtr _communicator;
    bool _destroy;
    std::list<CallPtr> _req;
    std::auto_ptr<Exception> _exception;
};
typedef Handle<OnewayCallQueue>
    OnewayCallQueuePtr;
```

Once again, `add` enqueues a message, and `destroy` instructs the queue to terminate.

Let's take a look at the implementation of the queue. The constructor, `add`, and `destroy` are the same as for the non-batching queue, with the exception of the communicator argument, so let's go straight to the thread implementation itself:

```
// C++
void
OnewayCallQueue::run()
{
    while(true)
    {
        list<CallPtr> req;
        {
            Lock sync(*this);
            while(!_destroy && _req.empty())
            {
                wait();
            }
            if(_destroy)
            {
                return;
            }
            req.splice(req.begin(), _req);
        }
        try
        {
            list<CallPtr>::const_iterator p;
            for(p = req.begin();
                p != req.end(); ++p)
            {
                (*p)->execute();
            }
            _communicator->flushBatchRequests();
        }
        catch(const Exception& e)
        {
            Lock sync(*this);
            _exception = auto_ptr<Ice::Exception>(
```

```
                e.ice_clone());
            _destroy = true;
            break;
        }
    }
}
```

The `run` method executes a loop that, on each iteration, waits to be notified. The notification arrives either because a new item has been added to the queue, or because the queue was destroyed. If the queue is destroyed, `run` returns; otherwise, `run` copies the contents of the queue using `splice`. (`splice` removes all elements from the second argument, and inserts them at the iterator position provided by the first argument; this is guaranteed to take constant time without copying any data and so is highly efficient.) Finally, `run` executes all requests in the queue (which adds them to the current batch) and then calls `flushBatchRequests` to send the batch. Note that the monitor is not locked for the duration of the inner loop, so it is possible to add more requests while the queue is still batching up the previous lot of requests.

To make a oneway call, the application once again creates a call object, and adds it to the oneway call queue. It is the application's responsibility to ensure that the proxy is in fact a oneway batch proxy. (The oneway call queue does not enforce this.) For this example, we can add an `assert` statement to the constructor of `SayHelloCall` to make sure that only a oneway batch proxy can be passed:

```
// C++
class SayHelloCall : public Call
{
public:
    SayHelloCall(const HelloPrx& hello)
      : hello(hello)
    {
        assert(_hello->ice_isBatchOneway());
    }

    void execute()
    {
        _hello->sayHello();
    }
private:
    const HelloPrx _hello;
};
// C++
void
MyWidget::makeRpc()
{
    HelloPrx hello = …;
    _onwayCallQueue->add(new SayHelloCall(hello));
}
```

The above queue works, but is not all that useful. The problem is that it sends each call as soon as it is enqueued (assuming the queue itself is idle). Typically, sending a oneway message and flushing its batch is very fast. As a result, the queue never gives the calls a chance to accumulate and thus no batching occurs. The simplest solution to this problem is to have the queue sleep for a while after flushing each batch to allow calls to accumulate:

```
// C++
list<CallPtr> req;
// ...
list<CallPtr>::const_iterator p;
for(p = req.begin(); p != req.end(); ++p)
{
    (*p)->execute();
}
_communicator->flushBatchRequests();
ThreadControl::sleep(Time::seconds(1));
```

The exact amount of time to sleep is application specific, and should probably be a configuration parameter for a oneway call queue. The only problem with this scheme is that, if the queue is destroyed while its thread is asleep, it will not actually terminate until the sleep has completed. For a one-second timeout this is unlikely to be problematic; however, for longer timeouts, this idea does not work very well. A solution to this problem is to ensure that events are not dequeued for a minimum amount of time. This can be implemented as follows:

```
// C++
void
OnewayCallQueue::run()
{
    //...
    list<CallPtr> req;
    {
        Lock sync(*this);
        Time minEnd = Time::now() +
            Time::seconds(1);
        while(true)
        {
            if(_destroy)
            {
                return;
            }
            Time diff = Time::now() - minEnd;
            if(diff::toSeconds() >= 0)
            {
                if(!_req.empty())
                {
                    break;
                }
                wait();
            }
            else
            {
                wait(diff);
            }
        }
        req.splice(req.begin(), _req);
    }
    // ...
}
```

```
void
OnewayCallQueue::add(const CallPtr& req)
{
    Lock sync(*this);
    _req.push_back(req);
    notify();
}
```

The implementation is somewhat inefficient because each time `add` is called, the thread is woken only to go back to sleep immediately if `diff < 0`. We improve on this by saving `minEnd` in a member variable of the queue and having `add` only call notify if `Time::now() >= minEnd`.

There are other possible approaches for the flushing scheme. For example, the application might want to wait for *n* calls to be queued before flushing. This can be combined with a timer so that, once the first batch event is added, the timer is set; the batch is flushed when the timer expires or once *n* events have been queued, whichever happens first.

A Complete Example

We will now modify the Qt client from the previous article to support the sending of oneway messages. We'll add a toggle button that, while depressed, causes oneway messages to be sent at regular intervals to the server via a oneway call queue.

We'll need to pass the oneway call queue to the hello dialog constructor. In addition, we need a new button that can toggle the sending of oneway events. While the button is depressed, oneway messages are enqueued at regular intervals. The simplest method is to use a `QTimer` for this. We connect the `timeout` signal of the timer to a slot in the dialog also called `timeout` and, for efficiency, we cache the batch oneway proxy:

```
// C++
class HelloDlg : public QDialog
{
    Q_OBJECT

public:

    HelloDlg(const CallQueuePtr&,
        const OnewayCallQueuePtr&,
        const HelloPrx&,
        QWidget *parent = 0);
    // ...
private slots:
    void toggleOneway(bool);
    void timeout();

private:
    const OnewayCallQueue _onewayQueue;
    const HelloPrx _helloOnewayBatch;
};
```

```
HelloDlg::HelloDlg(
    const CallQueuePtr& queue,
    const OnewayCallQueuePtr& onewayQueue,
    const HelloPrx& hello,
    QWidget *parent) :
        QDialog(parent),
        _queue(queue),
        _onewayQueue(onewayQueue),
        _hello(hello),
        _helloOnewayBatch(
            HelloPrx::uncheckedCast(
                hello->ice_batchOneway()))
{
    QPushButton* toggle =
        new QPushButton("Send Oneway");
    toggle->setCheckable(true);
    connect(toggle, SIGNAL(clicked(bool)),
      this, SLOT(toggleOneway(bool)));
    _timer = new QTimer(this);
    _timer->setInterval(100);
    connect(_timer, SIGNAL(timeout()),
            this, SLOT(timeout()));
    // ...
};
```

The timer is started and emits the timeout signal every 100ms until stopped. The implementation of `toggleOneway` is as follows:

```
void
HelloDlg::toggleOneway(bool checked)
{
    if(checked)
    {
        _timer->start();
    }
    else
    {
        _timer->stop();
    }
}
```

When `toggleOneway` is called with a value of `true` (thus the button is depressed), the timer is started, and when called with `false` (the button un-checked), the timer is stopped. The implementation of `timeout`, which is called every time the timer expires, simply enqueues the next call uses the cached `_helloOnewayBatch` proxy:

```
void
HelloDlg::timeout() {
    _onewayQueue->add(
        new SayHelloCall(_helloOnewayBatch));
}
```

The oneway queue then executes and flushes all batch messages every second.

That's it for this issue. Next month's article will look at a more elegant solution to the general problem of invoking a method from a GUI's main thread without the risk of blocking.

Ice-E 1.1: What's New?

Brent Eagles, Senior Software Engineer
Dwayne Boone, Senior Software Engineer

Flexibility, not Contortion—An Alternate Sequence Mapping

In Ice-E 1.0 for C++, Slice sequences were mapped to `std::vector`. While `std::vector` is a perfectly valid data type to use, an application might want to use a different container because it is more appropriate for the problem at hand. Unfortunately, this means that if you want to use data in an Ice-E request that is stored in another container, you have to create a `std::vector` instance of your Slice type and copy the data before making the request. Not only is such "glue" code a little messy, but you also pay a performance penalty—the extra copying of data is not free.

In Ice-E 1.1, by default, sequences still map to `std::vector`, but it is possible to change this mapping to a user-defined class using metadata. Any class can be used as the sequence type as long as it conforms to the following constraints:

- It has a default constructor.
- It has a copy constructor.
- It has a constructor that takes the initial size of the sequence as a parameter.
- It implements `size()` to return the current sequence size.
- It implements `swap()` to swap contents with another class of the same type.
- It has an `iterator` and `const_iterator` and implements `begin()` and `end()`.
- It takes care of its own memory management.

In effect, these constraints form a 'protocol' for allowable containers. For those familiar with the STL, this list will look familiar: the `std::list` and `std::deque` templates satisfy these requirements.

The Slice syntax for specifying an alternate mapping will be familiar to C++ users. For example:

```
// Slice
["cpp:type:std::deque< ::Ice::Byte>"]
    sequence<byte> ByteSeq;
```

The `["cpp:type:std::deque< ::Ice::Byte>"]` metadata will cause `std::deque` to be used for all occurrences of `ByteSeq` in your Slice definitions.

You can also define an alternate mapping for specific occurrences of a sequence type, such as an operation parameter or the data member of structure:

```
// Slice
sequence<byte> ByteSeq;
// Uses the default mapping to std::vector.

struct S
{
    // This member uses the default mapping.
    ByteSeq seq1;
    // Modify the mapping for this data member.
    ["cpp:type:std::list< ::Ice::Byte>"]
    ByteSeq seq2;
};

interface I
{
    // Modify the mapping of the return value
    //and the parameter.
    ["cpp:type:list< ::Ice::Byte>"]
    ByteSeq op(["cpp:type:list< ::Ice::Byte>"]
        ByteSeq seq);
};
```

Two additional metadata directives are supported for the input parameters of an operation, `array` and `range`. The `array` directive maps a sequence to a pair of pointers to the element type as `[first, last)`. Consider this example:

```
// Slice
interface I
{
    void op(["cpp:array"] ByteSeq seq);
};
```

The signature of the servant's method is mapped as follows:

```
// C++
void op(std::pair<const Ice::Byte*,
        const Ice::Byte*>, ...)
```

For byte sequences, these pointers refer to memory in the Ice run time's internal buffers. This avoids the overhead of an extra copy on the server side and results in a significant advantage in terms of speed and memory consumption.

The `range` directive maps a sequence to a pair of `const_iterators` that point to the beginning and end of the sequence. For example:

```
// Slice
interface I
{
    void op(["cpp:range"] ByteSeq seq);
};
```

The signature of the servant's method is mapped as follows:

```
// C++
void op(std::pair<ByteSeq::const_iterator,
        ByteSeq::const_iterator>, ...)
```

The `range` directive accepts an optional argument specifying an alternate sequence type, as shown in this example:

```
// Slice
interface I
{
    void op(["cpp:range:std::deque< ::Ice::Byte>"]
            ByteSeq seq);
};
```

The signature of the servant's method is mapped as follows:

```
// C++
void op(std::pair<std::deque<
            ::Ice::Byte>::const_iterator,
        std::deque<
            ::Ice::Byte>::const_iterator>, ...)
```

With these directives, using Ice-E with C++ is more natural than ever. You no longer need to write glue code to deal with the impedance mismatch between your code and an API that uses a type other than vector for sequences.

A Simple Example

Let's consider an example of a class of our own that demonstrates the requirements for the alternate mapping. We'll explore a simple encapsulation of a byte array. A class declaration for this might look as follows:

```
// C++
class MyByteSeq
{
public:
    //
    // Required constructors:
    //  - default
    //  - copy
    //  - initial size
    //
    MyByteSeq();
    MyByteSeq(const MyByteSeq&);
    MyByteSeq(size_t);
    ~MyByteSeq();

    //
    // Required methods:
    //  - size() that returns current
    // sequence size
    //  - swap() that swaps the contents with
    // another class of the same type.
    //
    size_t size() const;
    void swap(MyByteSeq&);

    //
    // Required iterators.
    //
    typedef Ice::Byte* iterator;
    typedef Ice::Byte* const_iterator;
    const_iterator begin() const;
    const_iterator end() const;

    //
    // If you've implemented a copy constructor,
    // you probably want an assignment operator.
    //
    MyByteSeq& operator=(const MyByteSeq&);

    //
    // Not required, but is a feature of the
class.
    //
    bool operator==(const MyByteSeq&) const;

private:

    size_t _size;
    Ice::Byte* _data;
};
```

You can download the implementation of this class from our web site. Here is how we can use this class in a Slice definition:

```
// Slice
//
// The cpp:include metadata tells slice2cppe to
// place an include directives in the appropriate
// spots in the generated code.
//
[["cpp:include:MyByteSeq.h"]]
module XFER
{

//
// Tell the generator to use MyByteSeq whereever
// ByteSeq is specified.
//
["cpp:type:MyByteSeq"] sequence<byte> ByteSeq;

interface BufferTransferSession
{
    //
    // Sends a byte sequence.
    //
    void put(ByteSeq buf);

    //
    // Retrieves a byte sequence.
    //
    ByteSeq get();
};
};
```

If you look at the generated code, you will see `MyByteSeq` being used where `vector<Ice::Byte>` would normally appear. For example, without the metadata, `slice2cppe` would generate code that looks like this:

```
// C++
namespace XFER
{

typedef ::std::vector< ::Ice::Byte> ByteSeq;
...
}
```

With the metadata directive, `slice2cppe` instead generates:

```
// C++
namespace XFER
{

typedef MyByteSeq ByteSeq;
...
}
```

The advantage of this becomes clear if you imagine that `MyByteSequence` is an existing class that is used repeatedly throughout your application and you want to start using Ice-E to introduce distributed capabilities to your system. Consider what client code using the `BufferTransferSession::put()` method might look like without the alternate mapping:

```
// C++
MyBytesSeq seq = ...;
// ...
vector<::Ice::Byte> vs;
MyByteSeq::const_iterator i;
for(i = seq.begin() ; i != seq.end(); ++i)
{
    vs.push_back(*i);
}
transfer->put(vs);
```

Without the alternate mapping, we first have to copy all the bytes from `seq` into a vector, just so we can call `put`; in contrast, with the alternate mapping, we can call `transfer->put(seq)` directly, without having to copy anything.

Similarly, for the server side, without the alternate mapping, things would also be awkward, especially since `MyByteSeq` doesn't implement much in the way of modifiers:

```
// C++
void put(const vector<::Ice::Byte>& seq,
         const Ice::Current&)
{
    MyByteSeq s(seq.size());
    MyByteSeq::iterator current = s.begin();
    vector<::Ice::Byte>::const_iterator i;
    for(i = seq.begin() ; i != seq.end();
        ++i)
    {
        *current = *i;
        ++current;
    }
    // Start using the application specific
    // data.
}
```

With the alternate mapping, we can skip the translation and immediately use the data.

This example is quite basic and might make controlling the mapping seem like a simple convenience. However, for sequences of complex types (or long buffers of simple types), each additional copy eats CPU and memory (particularly for copying of complex types, which typically requires multiple calls to copy constructors).

By the way, in case you are wondering, we've decided that this feature is important enought that we will incorporate it into Ice as of version 3.1.

Performance Improvements

The middleware community recently threw down the gauntlet by challenging Ice-E's performance. Not being ones to ignore a challenge (and finding laurels uncomfortable to rest on), we got to work on making Ice-E even faster.

No Red Lights, Full Speed Ahead—The Blocking Concurrency Model

Concurrency comes at a price. Multiple threads mean context switching and locking overhead. If we need maximum speed at all costs, the software instead needs to focus on doing one thing at one time and doing it as fast as possible. The blocking concurrency model does just that. By avoiding locking and context switching as much as possible, the blocking concurrency model achieves better performance.

By default, Ice-E supports the *thread-per-connection* concurrency model. With this model, the invoking thread sends the data across the network then blocks; a *receiver* thread associated with the connection waits for the reply. Once the reply arrives, the receiver thread notifies the invoking thread to unblock it. If other threads invoke an operation while the receiver thread is waiting for a reply to a previous invocation, those threads join a list of threads that are waiting for replies, and the receiver thread notifies the appropriate thread when the reply arrives. (Of course, this does not apply to oneway requests which do not return replies and therefore do not use the receiver thread.)

Ice-E 1.1 adds an additional concurrency model, the *blocking*

Figure 1: Blocking Concurrency Model

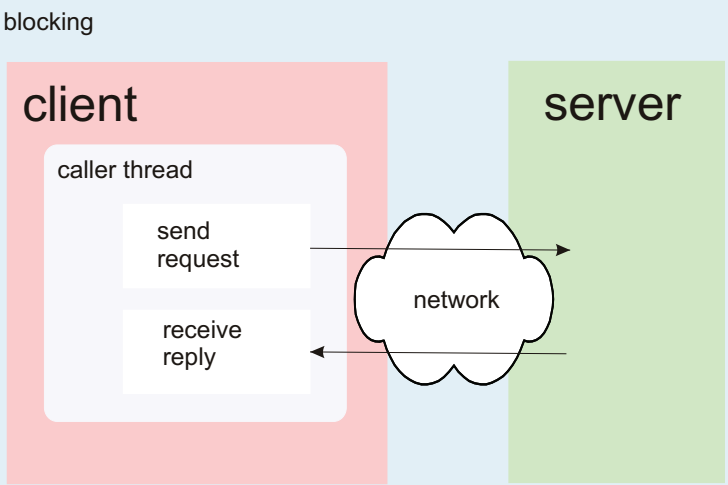
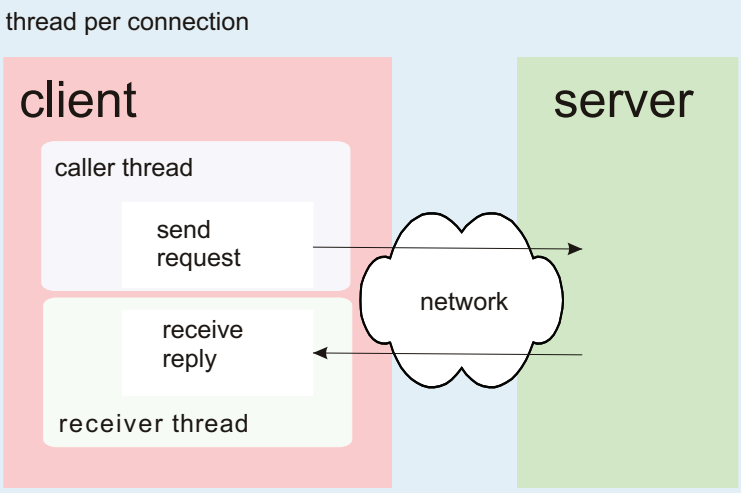


Figure 2: Thread-per-connection Concurrency Model



concurrency model. With this model, the calling thread sends the data across the network and then immediately performs a blocking read for the reply itself. (There is no receiver thread.) The advantage of the blocking model is that fewer steps are necessary to process a twoway request, which results in a significant speed increase. You have the option of building the Ice-E client library with only the blocking concurrency model enabled. This reduces the size of the Ice-E run time because it omits all the code that supports the thread-per-connection model.

As with most optimizations, the blocking concurrency model comes with trade-offs. Requests are multiplexed with the thread-per-connection model: if a server takes a long time respond to one request, during that time, other requests can be sent by the client and processed by the server. In other words, a long-running request does not prevent other requests from completing until the long-running request completes. With the blocking concurrency model, requests are completely serialized end-to-end: if the client sends a request on a connection, no other requests can be sent to the server over the same connection until the previous request completes.

Another limitation of the blocking model is connections cannot be used in bidirectional mode. If a server makes a callback to the client, the callback must be sent over a separate connection. However, this is not an issue unless you want to use callbacks and the server cannot make a connection back to the client (for example, because a firewall disallows incoming connections to the client's machine).

Express Route, No Transfers—Zero-Copy for Simple Types

Allocating and de-allocating memory on the heap is an expensive operation. Unfortunately, remote procedure calls usually require several heap allocations and deallocations during marshaling and unmarshaling. Besides the obvious performance hit, having multiple copies of same data in memory can be a show stopper in a memory-constrained environment. For servers that are implemented on portable or embedded devices, the extra memory cost can be particularly onerous if the servers need to service requests concurrently.

Ice-E 1.1 for C++ has improved performance by implementing an optimization commonly referred to as *zero-copy*. Zero-copy removes the memory allocations and copy operations that are normally performed when processing an Ice-E request. The performance gains are particularly noticeable if an application exchanges many small requests or processes large buffers.

Unfortunately, zero-copy implementations are sensitive to CPU architectures and compilers. Currently, Ice-E supports zero-copy for byte on all platforms. However, for other primitive types, such as int and short, zero-copy is supported only for x86 platforms. A future release will likely add zero-copy to the client side for additional performance gains.

New York to LA—Before and After

So what does this all mean? On the face of it, these changes *ought* to make Ice-E faster and lighter than ever before. Unfortunately, reality doesn't always meet expectations, so we ran extensive performance tests (over and over and over) to make sure we stayed on track and things were indeed getting faster. While we were at it, we also improved the performance of many other parts of the Ice-E code (besides the blocking model and zero-copy) to improve performance for existing Ice-E applications.

Remote procedure call performance can be measured in many ways. Two of the most basic types of measurements are *latency* and *throughput*. Latency is the time it takes for a single request to get from the sender to the receiver, and for the reply (if any) to be returned back to the sender. Throughput is a measure of how much data can be processed in a given period of time. Both latency and throughput need to be considered because low latency does not guarantee high throughput, and vice versa. This is obvious when we consider what contributes to latency and what contributes to throughput. In a low-latency system, we must keep call overhead to a minimum, so the number of operations performed per call is as low as possible. In addition, we need to avoid operations that can take a variable amount of time, and operations that need to yield resources to other operations as much as possible. For a high-throughput system, we obviously need to minimize the number of operations that are performed during marshaling and unmarshaling, and access the network as efficiently as possible. We might also use batch requests, compress data, or take advantage of multiple processors. Ice-E strives to find an optimal balance by allowing you to configure the system for low latency or high throughput, depending on application requirements.

The following two sections summarize the results of our performance tests and explain how each test works. For Windows, we tested on a Dell Dimension XPS with a 3.2GHz Pentium 4 CPU with hyper-threading and with 1GB of RAM. For Linux, we tested on a Dell Dimension with a 3.0GHz Pentium 4 CPU and with 2GB of RAM running Fedora Core 4. (By the way, you don't have to take our word for these test results: the data comes from the latency and throughput demos in the Ice-E distributions.)

Latency

As mentioned above, latency is the overall time it takes for a request to complete. We use a simple request with no data to evaluate latency. Since no data is transferred, latency represents the time it takes to perform the following general steps:

- construct a request
- marshal the request
- send a buffer containing the marshaled data through a network interface
- read a buffer containing the marshaled data from the network interface
- unmarshal the request
- find a servant
- dispatch the request
- construct a reply
- marshal the reply
- send the reply containing the marshaled data through a network interface
- read the reply from the network interface
- match the reply to a request
- return from the call

Ice-E 1.1 has greatly improved latency, particularly on Linux with a net improvement of 62%.

Table 1: Twoway Request

Product	Windows		Linux	
	µs/call	change	µs/call	change
IceE 1.0.0	111.6		80.9	
IceE 1.1.0	92.8	16.9%	61.9	44.5%
IceE 1.1.0 blocking	80.7	27.7%	46.2	58.6%

Oneway requests have similarly improved:

Table 2: Oneway Requests

Product	Windows		Linux	
	µs/call	change	µs/call	change
IceE 1.0.0	26.1		14.5	
IceE 1.1.0	14.7	43.9%	8.8	39.1%
IceE 1.1.0 blocking	14.1	45.9%	8.8	39.1%

Table 3: Oneway Batch Requests

Product	Windows		Linux	
	µs/call	change	µs/call	change
IceE 1.0.0	13.807		4.055	
IceE 1.1.0	4.275	69%	2.355	41.9%
IceE 1.1.0 blocking	4.286	69%	2.363	41.7%

For oneway requests, there is little or no difference between the blocking and thread-per-connection models because Ice-E does not wait for replies to oneway requests, so the blocking concurrency model offers no advantage.

A frequently asked question for middleware is how it compares to using raw sockets. Since we wanted to know where we stood, we performed a latency-oriented comparison with raw sockets as well:

Table 4: Raw Socket vs. IceE Blocking

Product	Windows		Linux	
	µs/call	change	µs/call	change
Raw Socket	55.8		34.0	
IceE 1.1.0	85.8	53.8%	46.6	37.0%

Depending on the platform, Ice-E is approximately 37% or 54 % slower than raw sockets when testing for latency. This might seem like quite a gap, but consider what these numbers actually mean. With Windows, a simple read/write of a few bytes of data over a socket takes 55.8 µsec. Ice-E obviously cannot do better than that, but it does take *at least* that long for Ice-E to send the data for a request. So, if we normalize to the baseline of raw sockets, it takes 30 µsec for an Ice-E request on Windows and 12.2 µsec on Linux. In other words, only 35% of the latency figure for Windows, and only 27% for Linux, are attributable to Ice-E. Considering how much extra work Ice-E does for you, that's quite a good deal.

Throughput

We compiled two groups of throughput tests, the normal *send* test and the *receive* test. In the case of the send test, most of the data

transfer occurs during the sending of the request to the server. In the receive test, the majority of data is transferred in the reply. These tests help us examine the effect of removing the receiver thread in the blocking concurrency model when receiving data and to compare it with the time it takes for the calling thread to send the data. In almost all cases, the difference between the blocking concurrency and thread-per-connection models is quite small. This isn't too surprising since these are end-to-end tests, so the cost of the server side handling the request is an overwhelming factor. In other words, a percentage point or two with the blocking concurrency model is still significant. More important is that the lack of the receiver thread in the receive tests doesn't cost anything. On Linux, performance improvements tend to be more dramatic because the Linux loopback adapter is implemented more efficiently than the one for Windows.

The byte throughput test shows good results, particularly on Linux. Zero-copy contributes to the performance gains. Each call sends or receives a byte sequence containing 500,000 elements.

Table 5: Byte Send

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	5.6563		4.6140	
IceE 1.1.0	4.9427	12.6%	2.5953	43.8%
IceE 1.1.0 blocking	4.8387	14.5%	2.5256	45.3%

Table 6: Byte Receive

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	6.5470		7.8279	
IceE 1.1.0	6.4323	1.8%	6.7030	14.4%
IceE 1.1.0 blocking	6.4113	2.1%	6.6721	14.8%

String handling in Ice-E is partly affected by the quality of the compiler's STL implementation. Ice-E shows a nice performance improvement when sending and receiving sequences of strings. Each sequence is 50,000 copies of the string "hello".

Table 7: String Sequence Send

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	28.1300		35.7704	
IceE 1.1.0	27.4793	2.3%	32.4171	9.4%
IceE 1.1.0 blocking	26.8073	4.7%	32.2799	9.8%

Table 8: String Sequence Receive

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	53.7500		54.0287	
IceE 1.1.0	52.1403	3.0%	50.6490	6.3%
IceE 1.1.0 blocking	52.1613	3.0%	50.1119	7.2%

The structure sequence throughput test illustrates improvement in marshalling and unmarshalling performance. Each sequence contains 50,000 elements of the following structure definition:

```
//Slice
struct StringDouble
{
    string s;
    double d;
};
```

Each struct is initialized with the string "hello" and 3.14 for the double value.

Table 9: Struct Sequence Send

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	36.2500		42.2178	
IceE 1.1.0	34.1043	5.9%	39.5455	6.3%
IceE 1.1.0 blocking	33.9530	6.3%	39.4879	6.5%

Table 10: Struct Sequence Receive

Product	Windows		Linux	
	ms/call	change	ms/call	change
IceE 1.0.0	64.1350		66.1004	
IceE 1.1.0	63.7500	0.6%	60.9475	7.8%
IceE 1.1.0 blocking	64.3540	-0.3%	61.6574	6.7%

Overall, we get a range of performance improvement from 7% to nearly 70%, depending on the platform and test. Some tests show no improvement, which is simply an indication that we haven't done any work in that area yet—please feel free to suggest any additional performance tests in our user forum!

The improvements in Ice-E provide better programming convenience, reduced memory footprint, and substantial gains in performance. In the future, we'll be using our hard-won knowledge to find ways to speed things up even more. As Ice-E finds its way into more and more real-world applications, we will no doubt discover use cases that will spur more innovative features.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/vbulletin/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: Why can't my C++ servant class derive from IceUtil::Thread?

C++ servant classes derive from generated skeleton classes. These skeleton classes use `IceInternal::GCShared` as a base class. That class provides reference counting and garbage collection (automatic detection of cyclic dependencies). On the other hand, `IceUtil::Thread` derives from `IceUtil::Shared`, which only provides reference counting, but no garbage collection.

`IceInternal::GCShared` is an internal class that only works in conjunction with the generated skeleton class. On the other hand, `IceUtil::Shared` works with any class, including classes written by Ice users. Because of the differences between these two base classes, it is not possible to derive a C++ servant class from `IceUtil::Thread`.

Instead of deriving a servant class from `IceUtil::Thread`, you can use a helper class that derives from `IceUtil::Thread` and keep that class as a data member in your servant class. The helper class then delegates to a run method of your servant class. Here is an example:

```
// Slice
module M
{
    interface ActiveObject
    {
    };
};

// C++
class ActiveObjectI;
typedef IceUtil::Handle<ActiveObjectI>
ActiveObjectIPtr;

class ActiveObjectI : public M:ActiveObject
{
public:

    ActiveObjectI() :
        _activeObjectThread(
            new ActiveObjectThread(this))
    {
    }

    void start()
    {
        _activeObjectThread->start();
    }

    void run()
    {
        // Your code goes here...
    }

private:

    class ActiveObjectThread :
        public IceUtil::Thread
    {
    public:

        ActiveObjectThread(
          const ActiveObjectIPtr& activeObject) :
            _activeObject(activeObject)
        {
        }

        virtual void run()
        {
            _activeObject->run();

            // Break cyclic dependency.
            _activeObject = 0;
        }

    private:

        ActiveObjectIPtr _activeObject;
    };

    const ThreadPtr _activeObjectThread;
};
```

Q: How do I transfer a file with Ice?

The easiest approach is to simply transfer the entire file in a single RPC. Here are the Slice definitions for a simple file store:

```
// Slice
sequence<byte> ByteSeq;

interface FileStore
{
    ByteSeq get(string name);
    void put(string name, ByteSeq bytes);
};
```

This interface allows clients to read and update the contents of remote files by specifying a file name. Quite often, this approach to file transfer is entirely adequate. Unless the files get quite large (in the tens of megabytes), they can be transferred in a single RPC without problems. If you want to transfer files larger than a mega-

byte, you will need to increase the setting of the `Ice.MessageSizeMax` property though. By default, this property limits the maximum size of Ice messages to 1MB so, to transfer larger files, you will need to increase the default setting. (See also the FAQ in Issue 5 of Connections, which discusses this property.)

If you need to transfer files that are large, you will likely need to use a different approach. The reason is that, once a single RPC transfers too much data, your machine is likely to start thrashing because all of the data for the RPC is buffered in memory before it is made available to the receiving end. In addition (unless you use zero-copy, as explained by Brent Eagles and Dwayne Boone in this issue), during unmarshaling, the Ice run time temporarily requires roughly twice the memory for the data in a request: once to store the data in a transport buffer, and once to make it available to the application as a byte sequence (such as a vector in C++ or a collection in C#).

For larger files, there are various ways to tackle the problem. One of the simplest and most effective is to read the file in chunks and to have the interface mimic the UNIX `read` and `write` system calls:

```
interface FileStore
{
    ByteSeq read(string name, int offset,
            int num);
    void write(string name, int offset,
            ByteSeq bytes);
};
```

The `read` operation requests a number of bytes starting at the specified offset. The operation returns either the number of bytes that were requested, or a sequence containing fewer bytes. In the latter case, the server may have limited the number of bytes it returned to its setting of `MessageSizeMax` (which may be smaller than the client's), so reading fewer bytes than requested does *not* indicate end-of-file (as it does in UNIX). Instead, the client must keep reading until it receives an empty sequence. To retrieve a file in chunks, the client simply keeps reading some number of bytes, starting at offset zero, and adds the number of bytes in the returned sequence to the offset for the next call to `read`, until `read` returns an empty sequence.

The `write` operation writes the byte sequence starting at the specified offset. If the specified number bytes cannot be transferred because the client's setting of `MessageSizeMax` is too low, the operation raises a `MemoryLimitExeption`; if the operation fails because the server's setting of `MessageSizeMax` is too low, the operation raises an `UnknownException`. If you prefer more descriptive exceptions, you can add an additional operation that allows the client to obtain the setting of the largest sequence that a server can handle (which will be smaller than the server's `MessageSizeMax` setting), and make the server throw an exception such as `SizeTooLarge` if the client attempts to transfer too many bytes at once.

Finally, for high throughput applications, you can use a more sophisticated scheme, such as the one used by IcePatch2: instead of transferring one chunk of a file at a time, IcePatch2 uses concurrent calls to avoid idle time while chunks are in transfer. However, that approach is beyond the scope of this FAQ—you can have a look at the IcePatch2 source code if you are interested in seeing how this approach works.