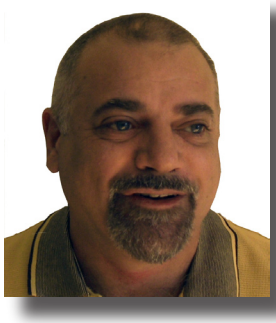
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 20, December 2006



80 Percent?

I have just returned from the Middleware '06 conference. The conference was well attended, with a number of high-quality papers. (Check out the conference proceedings for details.) I also met quite a few Ice users at the conference—I had fun talking with them and seeing Ice make inroads not only in industry, but also in academia.

As the conference papers amply demonstrated, research in the middleware arena is vibrant and active, and it produces useful results. That is a Good Thing™: as I said previously, middleware will continue to increase in importance. In fact, I see middleware as *the* key enabler for continued progress in computing: without ongoing research and improvements for middleware, the "Global Grid" (remember, you heard the term here first!) simply won't happen. But, sadly, very few of the research results make it into mainstream products.

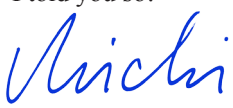
During the many conversations at the conference, I was struck with the disillusionment with web services. After years of trying, developers and researchers are getting increasingly disenchanted with WS complexity and poor performance, not to mention the standards mess and interoperability problems. (See Peter Lacey's satire "The S Stands for Simple" for a sobering history of web services.) Many conference attendees expressed the opinion that the WS bubble will implode in the near future. Personally, I look forward to the release of *WS-Death-Certificate*: some technologies are simply too awful to allow them to exist.

But, for middleware as a whole, this isn't exactly good news. Consider where we currently stand. On the one hand, Java RMI and .NET Remoting are tied to particular platforms and are of no interest for heterogeneous applications. (Besides, .NET Remoting has just been replaced by the Windows Communication Foundation; it goes almost without saying that WCF has a new and incompatible API…) On the other hand, for heterogeneous networking, CORBA is getting rather long in the tooth and, while REST offers some attractive ideas, developers want *technology,* not *philosophy*. And, with this short list, we have just about exhausted the available options. If WS is about to go the way of the Dodo too, that leaves the industry without ubiquitous middleware, despite fifteen years of promises to the contrary.

And *ubiquitous* middleware is what the industry needs. It needs ubiquitous middleware because, without it, universal e-commerce will remain a pipe dream. "B2B everywhere" cannot happen while the market is fragmented among a number of competing, incompatible, and technologically retarded offerings. No amount of de jure standardization will fix this. Here is my prediction of what *will* fix it: one product will capture a majority of the market before the rest of the players wake up. Then, after gaining sufficient experience, we'll write the standards around that product to codify existing best practice (which is what standards should be doing anyway).

"*One* product?" you ask? Yes, I am entirely serious. I firmly believe that *one* middleware product can comfortably serve the needs of 80 percent of applications, without being as complicated and inefficient as web services. So, is Ice that product? Quite possibly: Ice provides everything that is needed to cover the 80-percent bracket. With Ice, you can build efficient, reliable, and industrial-strength e-commerce systems *today* whereas, with web services, you count your blessings if you can exchange anything more complex than an integer among different implementations. No doubt, the industry will be ready to add yet another *WS-Something* standard to address your concerns but, while you wait for that standard to (maybe) fix things, you can use Ice to get on with the job. What do you prefer—a product without a standard that works, or a product with a standard that doesn't?

And, before you know it, you may find that Ice indeed has captured those 80 percent. Please, when that happens, don't be too smug about it. The web services devotees will know that you were right all along without you rubbing salt into their wounds and saying "I told you so!"



Michi Henning
Chief Scientist

Issue Features

Ice for Ruby

In this article, Mark Spruiell introduces Ice for Ruby.

Optimizing Performance of File Transfers

Matthew Newhook describes several techniques that you can use to optimize your applications.

Contents

Ice for Ruby

Mark Spruiell, Senior Software Engineer

Introduction

Dissatisfied with currently available technology, an enterprising developer decides to forge his own path and create something new. It's a familiar story in computing circles, and in fact it's exactly how Ice came to be. In this case, though, I'm referring to Ruby, a language worthy of your attention. In this article, I'll present an overview of the language and introduce the latest addition to the Ice family, Ice for Ruby.

Meet Ruby

Ruby sprang to life in Japan more than a decade ago when Yukihiro Matsumoto, unhappy with existing scripting languages such as Perl and Python, decided to design his own language. His goal was to create a pure object-oriented language that was easy for programmers to learn and use, and judging by Ruby's ever-growing popularity, he has succeeded remarkably well.

Everything is an object in Ruby, including strings, methods, and even numbers. For example, in Java you would obtain the absolute value of an integer by calling a static method and passing the integer as an argument:

```
// Java
int num = Math.abs(-42)
```

In Ruby, you invoke the method directly on the number:

```
# Ruby
num = -42.abs
```

Notice that I've omitted the trailing parentheses when calling the `abs` method. This is an example of Ruby's relaxed syntax, in which parentheses are optional for method invocations.

Matz, as Ruby's creator is known, borrowed heavily from other languages; mostly from Perl, but also from Python, Smalltalk, and Lisp. Many of us have undoubtedly written some completely unreadable Perl code in past lives, and I'm no exception. I'll be the first to admit that it was difficult to resist combining magic symbols like `$_` and complex regular expressions to accomplish a result in one line that would have taken dozens of lines or more in other languages. Although Ruby preserves some of this terseness, it has fallen out of favor among Ruby purists and, as of late, use of such cryptic expressions actually triggers warnings in the interpreter. I've learned my lesson, and Matz apparently has too.

Ruby versus Python

With my background in traditional languages such as C and C++, I was initially repelled by Python's rigid indentation requirements; eventually I overcame that bias and learned to appreciate what Python had to offer. As an Ice user, you probably know that Python is ZeroC's scripting language of choice; however I'm pleased that Python's formatting requirement is one "feature" that Ruby does not emulate. For example, here is a trivial Ruby class definition:

```
# Ruby
class Person
  def initialize(first, last)
    @first = first
    @last = last
  end

  def name()
    "#@first #@last"
  end
end
```

The method `initialize` is the constructor for an instance of class `Person`, and the `@` symbol is how you refer to instance variables. The keyword `end` terminates blocks, methods, classes, and modules. The `name` method demonstrates some convenient string formatting syntax as well as Ruby's convention of using the result of the last statement as the return value of a method. The equivalent Python code is shown below:

```
# Python
class Person:
  def initialize(self, first, last):
    self.first = first
    self.last = last

  def name(self):
    return self.first + " " + self.last
```

They look quite similar, and both are clean and easy to read and understand but, personally, I prefer Ruby's syntax. You may have a different opinion, but that's why we have many choices in programming languages.

Language Highlights

Getting back to our Ruby example, we have defined the `Person` class and now we want to instantiate it. Since classes are objects like everything else in Ruby, we simply invoke the `new` method on the class:

```
p = Person.new("Roger", "Seagraves")
```

If you like, you can also write the statement as follows:

```
p = Person.new "Roger", "Seagraves"
```

This style is a little too loose for my taste (I guess I'm just old-fashioned), so I'll be using parentheses from now on.

Ruby's object-oriented nature gives it a flexibility that you can exploit in a dizzying number of ways. For example, if you prefer a more verbose style in your expressions, you can rename the addition operator:

```
class Fixnum
  alias plus +
end
```

`Fixnum` is the name of Ruby's built-in 32-bit integer class. (It really has only 31 bits, but who's counting?) Declaring the class again does not replace its previous definition; instead, it opens the existing class to be modified or extended. In this case, we have added another way to invoke the + method, so that the following statements are all equivalent:

```
1 + 1
1.+(1)
1.plus(1)
```

Ruby also allows you to add or redefine methods in just one instance of a class:

```
p1 = Person.new("Roger", "Seagraves")
p2 = Person.new("Oliver", "Stone")
class <<p2
  def name()
    "#@last, #@first"
  end
end
puts p1.name()
puts p2.name()
```

The output of this program is shown below:

```
Roger Seagraves
Stone, Oliver
```

All this flexibility might lead you to believe that programming in Ruby is a complete free-for-all, but Ruby does have some interesting naming conventions that might dampen some of that rebellious spirit:

- The names of modules, classes, and constants begin with an upper-case letter.
- The names of methods and local variables begin with a lower-case letter.
- Instance variable names begin with the `@` character and are normally followed by a lower-case letter.
- Class variable names (such as a static variable in C++ and Java) begin with `@@`.
- Global variable names begin with the `$` character.

Ruby enforces some of these conventions, while others are merely recommendations, as determined by the needs of the parser.

In the sections that follow, I'll give a very brief introduction to Ruby's main constructs. If you find yourself intrigued, the "Pickaxe" book is a great way to learn more about the language.

Built-In Types

Ruby's native types include the aforementioned small integer, an arbitrary precision integer, floating point, string, array, and an associative array type called *hash*. As with any untyped language, a Ruby program typically doesn't need to concern itself with types; the type of a variable is determined by its current value, but the variable can be used to hold a value of another type at a later time.

A Ruby array is an indexed collection similar to Python's list type:

```
arr = ['a', 'b', 'c']
arr[10] = 99
```

As you can see, a program can assign values of varying types to arbitrary locations in the array; any gaps are automatically filled with the value `nil`.

While an array can only be indexed using an integer, a hash accepts any type of value as a key:

```
h = { "abc" => "def", 17 => "ghi" }
h[false] = [1, 2, 3]
```

True to its Perl heritage, Ruby also supports regular expressions as a native language feature, which makes it trivial to incorporate them into your own code:

```
class Person
  def match(expr)
    @first =~ expr || @last =~ expr
  end
end

p = Person.new("Caleb", "Shaw")
p.match(/[Cc]al/)
```

The value returned by `match` in this example is `0`, representing the position at which the expression matched the string `"Caleb"`. In Ruby, only the values `false` and `nil` cause a boolean expression to fail, therefore the following test executes as expected even when `match` returns `0`:

```
if p.match(/[[Cc]al/)
  puts "Found a match!"
end
```

Modules

A module serves several purposes in Ruby. The most common use case, which should be familiar to Python programmers, is encapsulating methods and classes in a unique namespace:

```
module Ice
  def initialize()
    # ...
  end
end
```

At this point you might be thinking that the code above looks surprisingly like a procedural Python program, and you'd be right. In Ruby's object-oriented world, however, this example actually declares a module object and defines a method in it named `initialize`. You can invoke this method using the `::` scope resolution operator:

```
Ice::initialize()
```

Modules serve another important purpose in Ruby: they provide a convenient way to incorporate functionality into a class without needing to use inheritance. Although Ruby classes are restricted to single inheritance, modules give you the equivalent of unlimited multiple inheritance. Consider the following example:

```
module Comparable
  def <(other)
    self.<=>(other) < 0
  end
  def >(other)
    self.<=>(other) > 0
  end
  # ...
end

class Person
  include Comparable
  def <=>(other)
    # ...
  end
end
```

We have defined the module `Comparable` that supplies a number of comparison methods. Each method assumes that the receiving object has implemented the `<=>` method, which, by convention, must return `-1`, `0`, or `1` to indicate its order compared with another object. It may seem strange at first to see methods that refer to `self` defined in a module—after all, `self` is normally used in classes. However, the module is intended to be used as a *mix-in* by other classes. As you can see, the `Person` class includes the module, thereby adding all of the methods in the module to its own definition. To comply with `Comparable`'s protocol, `Person` must implement only `<=>` to get a number of useful comparison methods for free, without sacrificing its ability to inherit from another class.

Duck Typing

Given the power of modules shown in the previous section, it's no surprise that an object's type has less importance in Ruby than it does in more statically-typed languages. You can still test an object to determine whether it is an instance of a class:

```
if (p1.kind_of?(Person) && p2.kind_of?(Person))
  cmp = p1 < p2 # Compare two people
end
```

However, an object's class (as well as its super-classes) is really just one source of methods you can invoke. The object's class might also include modules that define methods and, as we saw earlier, the object itself may have been extended to have additional methods. As a result, what really matters in many cases is whether an object supports the capability you need, and not whether it is an instance of a particular class. We could rewrite the code above as simply:

```
cmp = p1 < p2   # Equivalent to p1.<(p2)
```

The interpreter will raise an exception in this case if `p1` fails to implement the `<` method. If you want to verify that `p1` implements `<`, use the `respond_to?` method:

```
if (p1.respond_to?(:<))
  cmp = p1 < p2
end
```

In Ruby terminology, this is known as *duck typing*: if it walks like a duck, and talks like a duck, then Ruby treats it like a duck, regardless of whether it actually is a duck. Once you get used to the idea, you'll find it's a very powerful concept.

Blocks and Iterators

Speaking of powerful concepts, check this out:

```
arr = [1, 2, 3]
arr.each { |i| puts i }
```

When executed, this code produces the following output:

```
1
2
3
```

We have declared an array and invoked its `each` method. The argument to `each` is a segment of code enclosed in braces, called a *block* in Ruby. This code is not evaluated immediately; the block is treated as (you guessed it) an object that `each` invokes for every element of the array, and the variable `i` within the vertical bars represents the block's parameter list. The `each` method is known as an iterator because it iterates an arbitrary block over a collection.

Another example of an iterator is the `upto` method, which is often used in place of a traditional `for` statement:

```
0.upto(arr.length-1) { |i| puts arr[i] }
```

This code invokes `upto` on the integer zero and passes the maximum value as an argument.

Blocks are used throughout Ruby's libraries as a very convenient way of defining callbacks and handling transactions. For example, the code below opens a file, reads some text, and closes the file:

```
f = File.open("log.txt", "r")
line = f.gets()
# ...
f.close()
```

We could write this another way using a block, but this time we'll use the `do/end` keywords instead of braces:

```
File.open("log.txt", "r") do |f|
  line = f.gets()
  # ...
end
```

The `open` method detects the presence of this block and changes its behavior: instead of simply returning the opened file as in the previous example, `open` now opens the file, executes our block, and closes the file, freeing us from having to remember to close the file ourselves. As an added bonus, `open` also ensures that the file is closed in case our block raises an exception.

Exceptions

Exceptions in Ruby have the same semantics as in other mainstream languages, although the keywords can be a bit confusing at first. Whereas C++ and Java use the keywords `try`, `catch`, and `throw` for exception handling, Ruby's equivalents are `begin`, `rescue`, and `raise`:

```
begin
  raise "Oops!"   # Throws a runtime error
rescue RuntimeError => ex
  puts ex
end
```

The confusion arises from the fact that the keywords `catch` and `throw` are also used in Ruby, but have nothing to do with exceptions.

Ruby supports the equivalent of Java's `finally` clause, albeit with a different name of course:

```
begin
  raise "Not again!"
rescue => ex # Equivalent to rescue StandardError
  # Handle exception
ensure
  # Clean up
end
```

Ruby guarantees that the `ensure` clause will be executed regardless of whether an exception is raised; this provides the application with an opportunity to take action such as cleaning up resources.

Threads

Ruby's `Thread` class is an example of an API that relies on the block concept we saw earlier:

```
t = Thread.new("http://www.zeroc.com", 80) do
|url, port|
  conn = Net::HTTP.new(url, port)
  page = conn.fetch("/", nil)
end
t.join()
```

The `Thread` constructor requires the caller to define a block; any arguments passed to the constructor are transferred to the block's parameter list. This use of blocks conveniently eliminates the need to define a new class for each activity that you want performed in a separate thread.

The main disadvantage of Ruby's current thread implementation is its use of "green", or emulated, threads. The interpreter runs in a single native thread and performs its own task switching among the threads created by the application. This strategy is portable, and it works fine for many situations, but it may not be appropriate for applications that make a lot of system calls that can block: when the interpreter's one and only thread blocks, *all* Ruby threads block. I'll have more to say about this later.

Run Time Library

I have only touched on a few of Ruby's built-in classes; many others are available that supply such core functionality as signals, processes, date/time, file I/O, and more. In addition, Ruby's robust set of class libraries provides a wide range of services, including wrappers for database systems, math facilities, mutexes and monitors, XML parsing, and a suite of network protocol classes such as HTTP and SMTP. One particularly useful library is WEBrick, which implements a standard HTTP server and includes support for servlets and CGI. For example, here is all you need to serve static pages from a local filesystem using WEBrick:

```
require 'webrick'
include WEBrick
server = HTTPServer.new(
  :Port => 80,
  :DocumentRoot => File.join(Dir.pwd, "/html")
)
trap("INT") { server.shutdown() }
server.start()
```

There is plenty more you can do with WEBrick, as you'll see later in the article.

Ruby on Rails

I would be remiss in my duties if I did not at least mention Ruby on Rails, which is largely responsible for the tremendous growth that Ruby has been experiencing. The Rails framework's success lies in its ability to dramatically simplify the task of creating database-backed Web applications without complicated configuration files or even much programming. If you are in a situation where you need to offer Web access to database CRUD actions (create, read, update, delete), you should investigate what Rails can do for you.

The Birth of a New Mapping

The development of a new Ice language mapping requires varying degrees of effort. The compatibility of Slice and the target language is certainly a factor in determining how much work will be needed, but more significant is the implementation strategy for the Ice run time. There are essentially two choices: implement the Ice run time from the ground up in the target language, or try to

use an existing Ice run time as the foundation of the new product. We strive to do the latter whenever possible (as long as doing so wouldn't negatively impact the product) because our maintenance duties are reduced when we can reuse an existing, well-tested run time. The advantages of a native implementation would have to be very compelling for us to consider attempting such a project.

At the time of this writing, ZeroC has only three native implementations of the Ice run time: C++, Java, and C#. All of the remaining language mappings are layered on top of one of these three products. Since the interpreters for scripting languages are typically written in C for maximum portability, the only sensible choice for the foundation of such a language mapping project is Ice for C++.

The disadvantage of this layered approach is that the script programmer must be aware of the external dependencies (such as shared libraries) that are necessary to use Ice. It's a small price to pay, however, because it's really the only way ZeroC could justify supporting the languages that we already do. Some of the Ice internals are pretty complex and rely heavily on low-level APIs for sockets and other system calls. Even if a scripting language provided access to all of the facilities we would need, and even if they worked reliably (a big if), it would still be a huge undertaking to reimplement the Ice run time, test it, and maintain it. If we had to do that for every new language mapping, it's likely that the *only* language mappings we'd support would still be C++, Java, and C#.

Layering a new scripting language implementation on the C++ run time not only saves an enormous amount of time (which gets the product into your hands that much faster), but also results in a more reliable product from the start. Such a project consists of designing the mapping from Slice to the target language, writing the translator, and implementing a thin integration layer between the interpreter and Ice for C++. The integration layer is often the most challenging, since we need to get intimately familiar with the interpreter's extension API. This can be a somewhat frightening experience, as was the case with PHP, or a relatively pleasant one.

Overall, implementing the Ruby mapping was quite straightforward. The interpreter's C API seems to be generally well designed and easy to use, which made the development process much easier. There was really only one issue we had to face, but it was a significant one: Ruby's lack of support for native threads. It's not the fact that the interpreter uses green threads, but that the C API is not thread safe. As you know, Ice uses threads extensively, so this was a serious limitation. As a result, we didn't even try to implement support for writing an Ice server in Ruby, and it made us give up on asynchronous invocations. Essentially, any functionality that required the C++ Ice run time to call back into Ruby's C API from an arbitrary thread had to be abandoned. What remained was support for synchronous outgoing invocations, which was all the project's sponsor needed anyway. A future version of Ruby is planned that supports native threads, at which point it will become feasible for us to consider adding the missing functionality to Ice for Ruby.

Slice to Ruby

Slice types map quite naturally to Ruby. Rather than repeat what's already described in the Ice Manual, I'll touch on elements of the mapping that provide more information on the Ruby language and demonstrate useful idioms.

Identifiers

Earlier in the article, I described Ruby's quirky naming conventions for identifiers; you need to be familiar with these rules because they can affect the generated code and therefore your application. As an example, consider the following Slice definitions:

```
// Slice
module corp {
  interface backOffice {
    void Reconcile();
  };
};
```

When translated to Ruby, these identifiers are modified as shown below:

```
module Corp
  class BackOffice
    def Reconcile()
      # ...
    end
  end
end
```

As this example demonstrates, a Slice construct that maps to a Ruby module, class, or constant must have an identifier that begins with an upper-case letter, therefore the identifiers `corp` and `backOffice` are modified as shown. For method names such as `Reconcile`, Ruby conventions recommend that they start with a lower-case letter, but the interpreter doesn't enforce that rule, and neither does the Slice mapping.

Primitive Types

Aside from the identifier issue, a Ruby programmer should be very comfortable using the Slice mapping. For example, the table below describes the Ruby mapping for primitive Slice types.

Slice Type	Ruby Type
`bool`	`true` or `false`
`byte`	`Fixnum`
`short`	`Fixnum`
`int`	`Fixnum or Bignum`
`long`	`Fixnum or Bignum`
`float`	`Float`
`double`	`Float`
`string`	`String`

`Fixnum` is Ruby's 31-bit integer type, whose range can represent most of the Slice integer types. If an integer value requires more than 31 bits, Ruby transparently promotes it to the arbitrary-precision type `Bignum`. Consider the following Slice definitions:

```
// Slice
const int I1 = 1073741823;
const int I2 = 1073741824;
```

In Ruby, the constant `I1` is represented by a `Fixnum` object, while `I2`'s value can only be stored in a `Bignum` object. As far as the Ice run time is concerned, an integer's type is irrelevant; what matters is whether the value is compatible with the expected Slice type. Let's continue this example below:

```
// Slice
const long L1 = 1;
const long L2 = 4294967292;

interface IntSender {
  void sendInt(int i);
};
```

It is legal to invoke `sendInt` and pass the values `I1`, `I2`, or `L1`. However, attempting to pass `L2` will prompt the Ice run time to raise an exception because the value exceeds the range of Slice's `int` type.

With respect to strings, Ruby's `String` class represents a string value as an arbitrary array of bytes, therefore it's a good match for the Ice protocol's use of the UTF-8 encoding. Practically speaking, the Ice run time performs no translation while sending or receiving string values, so your strings must be properly-formatted UTF-8 values.

User-Defined Types

User-defined Slice types such as structures, classes, enumerations, sequences, and dictionaries all map easily to Ruby types. The results should not surprise you: structures, classes, and enumerations are represented by Ruby classes, sequences map to arrays, and dictionaries become hashes. The sections that follow present a brief example for each type.

Structures

We'll begin our review with structures, as shown in the example below:

```
// Slice
struct S {
  string str;
  int i;
};
```

The mapping for a structure includes a constructor that accepts values for each of the data members, and definitions for the methods `hash`, `==`, and `inspect`:

```
class S
  def initialize(str='', i=0)
    @str = str
    @i = i
  end
  def hash ...
  def ==(other) ...
  def inspect ...

  attr_accessor :str, :i
end
```

The constructor provides suitable default values for the structure's data members and transfers its arguments to corresponding instance variables. The `hash` method enables an application to use instances of this type as keys in a hash object, and the `==` method provides member-wise equality semantics. Ruby invokes `inspect` on an object to obtain its printable representation, so Ice defines `inspect` to display the object's data members in a nice way.

More interesting is the last statement, which invokes the class method `attr_accessor` (without parentheses) and passes the symbols of the instance variables. This is a convenient way of defining accessor and mutator methods for each of the type's data members, whose private visibility would otherwise make them inaccessible outside the class. Calling `attr_accessor` is equivalent to declaring the following methods:

```
def str(val)
  @str = val
end
def str()
  return @str
end
def i(val)
  @i = val
end
def i()
  return @i
end
```

Since Ruby allows us to omit parentheses when invoking methods, we can obtain the value of a data member in a style that looks very familiar:

```
s = S.new("my string", 5)
puts "str = " + s.str + " and i = " + s.i
```

Furthermore, Ruby automatically translates assignment statements into invocations of a mutator method, so although we could modify the member `i` as shown below:

```
s.i(6)
```

It's much more natural to use an assignment statement and take advantage of Ruby's syntactic sugar:

```
s.i = 6
```

Enumerations

Ruby doesn't have a native enumeration type, so Ice generates a Ruby class that has similar semantics. Consider this example:

```
// Slice
enum Color { red, green, blue };
```

The generated class provides several useful methods:

```
class Color
  include Comparable

  def Color.from_int(val) ...
  def to_s ...
  def to_i ...
  def <=> ...
  def hash ...
  def inspect ...
  def Color.each(&block) ...

  Red = ...
  Blue = ...
  Green = ...
  private_class_method :new
end
```

The first aspect worth mentioning is the inclusion of the standard module `Comparable` which, as we learned earlier, extends the class with a number of comparison methods. The generated class implements the `<=>` method to support this functionality.

The `from_int` method, which allows you to convert an integer into an enumerator, demonstrates the syntax for defining class methods in Ruby.

The methods `to_s` and `to_i` have the conventional names that Ruby classes use for obtaining string and integer representations of an object, respectively. As I described in the previous section on structures, the `hash` method is provided mainly to allow instances to serve as hash keys, while the `inspect` method supplies a user-friendly string that describes the object.

The class method `each` is notable for its use of a Ruby block as a parameter. The leading `&` character indicates that a block is expected, and the interpreter places the block object in the named parameter. This method allows us to iterate over the enumerators as shown below:

```
Color.each { |e| puts e.to_i }
```

Next, a class constant is defined to represent each of the type's enumerators. In this example, you'll notice that the first letter has been changed to upper-case in order to comply with Ruby's identifier semantics for constants.

Finally, the invocation of `private_class_method` changes the visibility of the `new` method so that applications are prevented from calling `Color.new`; the only valid instances of `Color` are those referenced by the constants `Red`, `Green`, and `Blue`.

Sequences

As in most other language mappings, a Slice sequence doesn't generate much code in Ruby because it uses the native array type. There is one exception, however, and that is for the Slice type `sequence<byte>`. Ice can transfer a value of this type much more efficiently when it is stored in a Ruby string rather than an array; therefore, Ice for Ruby always uses a string to hold values of this type.

Ruby's `Array` class offers a lot of useful functionality. For example, let's define a Slice sequence of structures and look at some examples:

```
// Slice
struct S {
  string str;
  int i;
};
sequence<S> SArray;
```

In Ruby, we can create an instance of `SArray` and populate it using the `<<` method:

```
arr = []
arr << S.new("red", 0)
arr << S.new("green", 1)
arr << S.new("blue", 2)
```

Next, we'll search for an element using the `find` iterator:

```
g = arr.find { |s| s.str == "green" }
```

The `find` method invokes our block for each element and stops if the block evaluates to true; the return value is the element at the current position, or `nil` if no match was found.

We can also use a block to selectively remove elements from the array:

```
arr.delete_if { |s| s.i > 0 }
```

I've only scratched the surface of an array's capabilities. If you're new to Ruby, you'll find a lot to like here.

Dictionaries

Let's continue our Slice example from the previous section to demonstrate that hash objects are just as powerful as arrays:

```
// Slice
dictionary<string, S> SMap;
```

The code below constructs an instance of `SMap`:

```
map = {
  "a" => S.new("abc", 0),
  "d" => S.new("def", 1)
}
```

The `each` method is also supported by the `Hash` class, along with variations named `each_key` and `each_value` that do what you'd

expect. In the case of `each`, the block receives two arguments representing the element's key and value:

```
map.each { |k,v| puts "#{k} => #{v.str}" }
```

The output of this statement is shown below:

```
a => abc
d => def
```

Exceptions

The mapping for Slice exceptions looks very similar to that for structures. The only significant difference is the use of inheritance, such that a generated class derives from a specified base class, or from `Ice::UserException` if no base exception is defined. All Ice exceptions ultimately derive from `Ice::Exception`, which derives from Ruby's `StandardError` class.

Typically, an application catches only the exceptions of interest at a particular nesting depth while allowing other exceptions to propagate higher in the call chain:

```
def do_something
  begin
    # ...
  rescue MyUserException => ex
    # handle user exception
  end
end
begin
  do_something()
rescue Ice::Exception => ex
  # General handler
end
```

Classes and Interfaces

Since Ice for Ruby does not support server-side functionality, Slice classes and interfaces serve only two purposes: transferring objects by value and invoking operations via proxies. Let's concentrate on objects-by-value now; I'll discuss proxies in the next section.

In its simplest use case, a Slice class is very much like a structure, except that it also supports inheritance:

```
// Slice
class Vehicle {
  int numPassengers;
};
class Truck extends Vehicle {
  float capacity;
};
```

These are concrete classes because they do not define operations; therefore, you can transfer instances of these classes without any additional effort. Although these definitions don't look much different from structures, the generated Ruby code for a class differs quite a lot from that for a structure. In particular, the generated code makes use of mix-in modules for reasons I'll explain shortly:

```
module Vehicle_mixin
  include ::Ice::Object_mixin
  def ice_ids # ...
  def ice_id # ...
end

class Vehicle
  include Vehicle_mixin
end
```

As you can see, the definition of class `Vehicle` is quite simple because it only needs to include the mix-in module.

A class becomes abstract once you define an operation. The generated class is no longer sufficient because there is no implementation of the operation; it becomes your responsibility to define a class that supplies the missing implementation. Suppose we modify our definition of `Vehicle` to add an operation:

```
// Slice
class Vehicle {
  int numPassengers;
  void start();
};
```

We have two choices when implementing a class in Ruby: inherit from the generated class, or include the mix-in module:

```
# Approach 1: Inherit from generated class
class VehicleImpl1 < Vehicle
  def start
    # ...
  end
end
# Approach 2: Include the mix-in module
class VehicleImpl2
  include Vehicle_mixin
  def start
    # ...
  end
end
```

As far as the Ice run time is concerned, both approaches are equally correct. In accordance with the duck typing semantics I described earlier, Ice does not check whether an object implements a particular type when you attempt to transfer it by value. All that matters to the Ice run time is that the object complies with the expected protocol, which it can do in either of the ways shown above. As a result, if you prefer to inherit from an unrelated base class, you need only include the mix-in module to satisfy the Ice run time.

Proxies

Defining an operation in a Slice class or interface automatically generates code that allows a Ruby program to invoke that operation remotely. These invocations are made using a proxy, which is a local artifact that represents a remote object. Since the Ice run time is responsible for implementing proxies, we don't need to explore the generated code in any detail. All you really need to know

is that for an abstract Slice class or interface such as `Truck`, Ice generates a separate proxy class named `TruckPrx` through which you invoke operations. You obtain an instance of a proxy class from the Ice run time, either as the result of a remote invocation, or by calling methods on the communicator object such as `stringToProxy` and narrowing the proxy to the desired type:

```
prx = communicator.stringToProxy("...")
truck = TruckPrx::checkedCast(prx)
truck.start()
```

In addition to the operations defined by your Slice interfaces, proxy objects offer a number of other standard methods. The Ruby mapping provides complete support for these methods, and you can find more information in "The Ice Run Time in Detail" chapter of the Ice Manual.

One important aspect of the Ruby mapping for proxies is the semantics of operation parameters. Like Python, Ruby does not support the notion of output arguments; therefore, Ruby and Python both handle them in the same manner. If an operation returns more than one value, the caller receives the values as an array. For example, consider the following operations:

```
// Slice
int getLevel();
void describe(out string text);
string query(string expr, out int numMatches);
```

Both `getLevel` and `describe` return only one value, although in a slightly different way: `getLevel` declares a return value, while `describe` uses an out-parameter. In Ruby, both invocations have the same syntax:

```
lev = proxy.getLevel()
str = proxy.describe()
```

On the other hand, `query` declares both a return value and an out-parameter, so its call syntax is slightly different. If you prefer to receive the values as an array, you can simply write the following:

```
arr = proxy.query("...")
```

The first element in the array is the return value, followed by the out-parameter. Alternatively, you can expand the array into separate variables as shown below:

```
result, matches = proxy.query("...")
```

Although it may seem unusual at first, it's a common technique for returning multiple values in Ruby, and that's what really matters: making remote invocations in Ice is meant to be as natural as calling methods on a language-native object.

Proxy invocations are affected by the threading limitations I mentioned earlier, in that all Ruby threads are blocked until the request completes. You could try to work around this issue by using oneway invocations instead, but that can add considerable complexity to your interfaces and, as this FAQ explains, doesn't guarantee that the main thread won't block.

Using Ice for Ruby

Once you've installed Ice for Ruby and properly configured your environment, you can start using it right away. If you have a Slice file handy, you can start the interactive Ruby interpreter (`irb`) and discover just how easy it is:

```
$ irb
irb> require 'Ice'
irb> Ice::loadSlice('Truck.ice')
irb> comm = Ice::initialize()
irb> p = comm.stringToProxy("truck:tcp -p 10000")
irb> truck = TruckPrx::checkedCast(p)
irb> truck.start()
```

There you have it — in just six commands we have managed to install the Ice extension, load our Slice definitions, initialize a communicator, create a proxy, and invoke an operation. Of course, this example assumes that an appropriate server is active and listening on the specified port.

Having all the capabilities of an Ice client at your disposal in an interactive environment is invaluable, not only for the educational experience, but also during development when you need to quickly test an idea or invoke a remote object. Naturally, Ruby is not only suited to running short scripts, but is fully capable of handling large and complex applications.

Translating Slice

The `loadSlice` method is a convenient way of translating your Slice definitions into Ruby code. Unlike the statically-typed language mappings, where an intermediate translation step is required along with the resulting generated source files, `loadSlice` generates Ruby code dynamically. Immediately after `loadSlice` completes, all of the equivalent Ruby types are installed and ready for use. Of course, if you prefer to generate Ruby code statically, the `slice2rb` compiler is also available and works very much like its counterparts for other languages.

It's often easier to use `loadSlice` at first and then migrate to static translation later if necessary. Moving from one style of translation to another is straightforward. For example, suppose your program currently invokes `loadSlice` as shown below:

```
Ice::loadSlice("--all -I#{ENV['APP_HOME']}/slice
Account.ice")
```

The `--all` option requests that Ruby code be generated for the definitions in `Account.ice` as well as any definitions included by that file. To transition to static code generation, you could use the following command line:

```
$ slice2rb --all -I$APP_HOME/slice Account.ice
```

Alternatively, you can eliminate the `--all` option if you also statically translate the files included by `Account.ice`. In either case, `slice2rb` generates Ruby code into the file `Account.rb`, which your program must load in the usual manner:

```
require 'Account'
```

As always, consult the Ice Manual for more details on Ruby code generation.

Our First Application

Each of the Ice language mappings defines a convenience class named `Application` that encapsulates functionality needed by most Ice programs. In Ruby, we can define a sub-class in just a few lines that replicates our prior interactive session:

```
require "Ice"
Ice::loadSlice("Truck.ice")

class MyApp < Ice::Application
  def run(args)
    comm = Ice::Application::communicator()
    proxy = comm.stringToProxy(
        "truck:tcp -p 10000")
    truck = TruckPrx::checkedCast(p)
    truck.start()
    return 0
  end
end
app = MyApp.new()
exit(app.main(ARGV))
```

`Application` takes care of initializing a communicator, responding appropriately to signals, and cleaning up when the application terminates. Using `Application` certainly is not a requirement, but we do recommend it unless your program has special needs.

Web Services and Ruby

As much as you might like to use Ice for all of your distributed computing needs, there are often times when external forces compel you to employ a different (and dare I say lesser?) technology. In situations where a "Web services" (WS) solution such as REST, XML-RPC, or SOAP is required, you may find that Ruby is an excellent development platform for integrating these technologies with your Ice applications.

A remote invocation using a protocol such as SOAP typically has much higher latency than the same invocation using Ice, so it is advisable to design your WS interfaces to be as coarse-grained as possible. In other words, creating a SOAP/Ice bridge that exports all of your Ice interfaces directly to SOAP clients is often undesirable. A better solution is to design an interface specifically to meet the needs of the WS client, such that completing a WS request might require many Ice invocations behind the scenes.

Ruby's class libraries include support for WS protocols and, as you should expect by now, it's quite easy to get started. Using the WEBrick HTTP server I mentioned earlier, we can build an XML-RPC server that integrates with Ice. As a trivial example, let's create a bridge to an Ice-based search engine:

```
require "webrick"
require "xmlrpc/server"
require "Ice"

Ice::loadSlice("Engine.ice")
comm = Ice::initialize()
prx = comm.stringToProxy("server:tcp -p 8000")
engine = EnginePrx::checkedCast(prx)

servlet = XMLRPC::WEBrickServlet.new()
servlet.add_handler("query") do |text|
  engine.query(text)
end

server = WEBrick::HTTPServer.new(:Port => 80)
server.mount("/RPC2", servlet)
trap("INT") { server.shutdown() }
server.start()
comm.destroy()
```

After instantiating a servlet that speaks XML-RPC, we add a handler for the `query` request. By now, you should recognize the definition of a Ruby block; this one accepts a string representing the search text and returns the results of a proxy invocation on an Ice object. Notice that the block is able to access the local variable `engine` defined in the outer scope. The other notable aspect of this code is the call to `mount`, which associates the servlet with an entry point on the Web server.

In a client, we only need three lines of code to invoke the request via XML-RPC:

```
require "xmlrpc/client"
server = XMLRPC::Client.new(
    "127.0.0.1", "/RPC2", 80)
puts server.call("query", "Camel Club")
```

Constructing a client requires the server's address, mount point, and port number. We invoke the request using the `call` method, whose first parameter identifies the name of the request; any additional parameters are marshaled into XML and included in the request message.

This simple example should give you a hint of what you can do with Ruby. Even if you ultimately decided to deploy a Web services integration project using a different technology, Ruby makes it incredibly easy to create a prototype and get an initial implementation online with minimal effort.

Summary

With its rich language, diverse class libraries, and supportive user community, Ruby is a rising star among modern programming languages. Extending its capabilities with Ice creates a uniquely powerful platform for creating distributed applications. Ruby clearly can't be used in all situations; as in any project you have to pick the most appropriate tool for the job. However, its ease of use and functionality make a compelling case for adding it to your toolbox.

Optimizing Performance of File Transfers

Matthew Newhook, Senior Software Engineer

Introduction

Performance is a tricky subject that is near and dear to our hearts. In this article, I discuss a number of performance enhancing techniques for file transfer that you can adapt to your own projects. As you will see, it is easy to fall into the trap of prematurely chasing the holy grail of performance. The only way to judge the effects of performance improvements is to evaluate them in the context of the entire application; taken in isolation, performance gains can be meaningless and amount to no more than extra development work without gain.

Client

Here is the first interface presented in the FAQ How do I transfer a file with Ice?

```
// Slice
sequence<byte> ByteSeq;
interface FileStore
{
    ByteSeq get(string name);
    void put(string name, ByteSeq bytes);
};
```

This interface transfers the contents of a file in one fell swoop with a single RPC call. If you are on a reliable LAN with plenty of available memory, this simple interface often will do: simply set `Ice.MessageSizeMax` to a large value and let the data fly. However, as explained in the FAQ, there is a down-side to this approach. Whenever the potential data set is very large, it is generally better to segment the data and retrieve it with several RPCs instead of a single one. Real-world interfaces use this technique, as shown in the following snippet from the IcePatch2 service:

```
// Slice
interface FileServer
{
    // ...
    nonmutating Ice::ByteSeq
    getFileCompressed(
        string path, int pos, int num)
        throws FileAccessException;
};
```

The caller already knows which files are available, as well as the number of bytes in each file. The caller transfers the file in chunks by repeatedly calling `getFileCompressed` until it has retrieved all chunks. This avoids the problem of running out of memory if a file is large.

The FAQ goes on to present another interface:

```
// Slice
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 at most `num` bytes (the server may return fewer bytes than requested, for example, if the requested number of bytes would exceed the server's `MessageSizeMax`). The client keeps reading the file in chunks until it receives an empty sequence to indicate end of file.

This interface, however, introduces a performance problem because a single network invocation is split into multiple invocations, so latency increases.

Note that if a file is sufficiently large, there is no getting around sending multiple invocations, especially if either client or server is short on memory For example, to send a 1GB file in a single RPC requires that the client have at minimum 2GB of available memory. Why 2GB and not 1GB? Consider the invocation using the original `FileStore` interface:

```
// C++
FileStorePrx store = ...;
ByteSeq bytes;
// Copy 1GB of data into bytes.
store->put(name, bytes);
```

This clearly occupies 1GB of memory. However, the actual RPC temporarily requires an additional gigabyte of memory because Ice internally creates a marshaling buffer, encodes the protocol header, and then copies the file data into this buffer for transmission over the network. Similarly, the server also temporarily requires an additional gigabyte of memory (unless it uses zero-copy, which I will discuss shortly): 1GB to read the encoded data into a marshaling buffer, plus 1GB for the sequence that is passed to the `put` operation. Note that if your hardware has lots of virtual memory, this approach will be a little slow, but it will work—you need to judge for yourself as to whether this limitation is acceptable in your environment.

Now let's see how a client might use this new `FileStore` interface from the FAQ to read the contents of a file:

```
// C++
string name = ...;
string output = ...;
FileStorePrx store = ...;
int len = 1000*1024;
FILE* fp = fopen(output.c_str(), "wb");
int offset = 0;
```

```
for(;;)
{
    Ice::ByteSeq data = store->read(
        name, offset, len);
    if(data.size() == 0)
    {
        break;
    }
    if(fwrite(&data[0], 1, data.size(), fp)
        != data.size())
    {
        cerr << "error writing: "
            << strerror(errno) << endl;
        break;
    }
    offset += data.size();
}
fclose(fp);
```

This code is quite simple: the client reads data from the store in chunks and writes them directly to the output file. You can try this client along with a simple version of the server by running the `simple` client–server demo contained in the accompanying source code archive.

This implementation suffers from two performance problems. The first, as I mentioned earlier, is network latency. The second and more serious issue is the time spent writing data to the file, during which all network activity ceases. Ideally, we want to exploit parallelism by having the server deliver the next chunk of data while the client is still writing the previous chunk. We can use AMI to achieve this, by sending the request for the next chunk before writing the data to disk. First we must modify the interface to add the AMI metadata directive:

```
// Slice
interface FileStore
{
    ["ami"] ByteSeq read(
        string name, int offset, int num);
};
```

We'll change the client's main loop as follows:

```
// C++
string name = ...;
string output = ...;
FileStorePrx store = ...;
int len = 1000*1024;
int offset = 0;
FileStore_readIPtr cb = new FileStore_readI;
store->read_async(cb, name, offset, len);
Ice::ByteSeq bytes;
for(;;)
{
    cb->getData(bytes);
    if(bytes.empty())
    {
        break;
    }
    offset += bytes.size();
    store->read_async(cb, name, offset, len);
    if(fwrite(&bytes[0], 1, bytes.size(), fp)
        != bytes.size())
    {
        cerr << "error writing: "
            << strerror(errno) << endl;
        break;
    }
}
```

The client starts the reading process by asynchronously sending the read request using the `read_async` call and then enters a read–write loop. Inside the loop, the client calls `getData` on the AMI callback, which blocks until the data is available. Once `getData` returns, the client immediately issues a request for the next chunk of data so that, while it is writing the data to disk, a client-side thread has the opportunity to read the next reply. In theory, this should provide more parallelism and improve the performance of our application. Here is the implementation of the AMI callback.

```
// C++
class FileStore_readI
    : public AMI_FileStore_read,
      public IceUtil::Monitor<IceUtil::Mutex>
{
public:

    FileStore_readI()

        : _done(false)
    {
    }
    // ...
private:

    bool _done;
    auto_ptr<Ice::Exception> _exception;
    Ice::ByteSeq _bytes;
};
typedef IceUtil::Handle<FileStore_readI>
    FileStore_readIPtr;
```

We have three member variables: `_done` is used to wait for pending invocations to complete, while `_exception` and `_bytes` store the results of the invocation. Here is the implementation of `ice_response`:

```
// C++
virtual void
ice_response(const Ice::ByteSeq& bytes)
{
    Lock sync(*this);
    _bytes = bytes;
    _done = true;
    notify();
}
```

This method is called when a thread from the client-side thread pool receives the reply for an asynchronous invocation. The response (a sequence of bytes) is passed as the argument to this method, which saves the response in the `_bytes` member and then

notifies the monitor that a reply has arrived. Here is the implementation of `ice_exception`:

```
// C++
virtual void
ice_exception(const Ice::Exception& ex)
{
    Lock sync(*this);
    _exception.reset(ex.ice_clone());
    _done = true;
    notify();
}
```

This method is called if an invocation results in an exception. We save the results in the `_exception` member variable and then notify the monitor that a reply has arrived. Finally, `getData` waits for the asynchronous invocation to complete:

```
// C++
void
getData(Ice::ByteSeq& bytes)
{
    Lock sync(*this);
    while(!_done)
    {
        wait();
    }
    _done = false;
    if(_exception.get())
    {
        auto_ptr<Ice::Exception> ex = _exception;
        _bytes.clear();
        ex->ice_throw();
    }
    bytes.swap(_bytes);
}
```

The method waits to be notified by `ice_response` or `ice_exception`. If an exception occurred, `getData` rethrows it for the caller to handle. Otherwise, it swaps the byte sequences and returns. (We use `swap` to avoid copying the byte sequence.) Why do we pass the vector by reference instead of returning the vector? Consider this alternative implementation:

```
// C++
Ice::ByteSeq
getData()
{
    // ...
    Ice::ByteSeq bytes
    bytes.swap(_bytes);
    return bytes;
}
```

As this code stands, if the compiler does not implement the Named Return Value Optimization (NRVO), it will cause an additional copy of the vector. We avoid this copy for compilers that do not implement NRVO by passing a reference to the vector instead.

Is there anything we can do to further improve performance? We have three variables to consider:

- The amount of time it takes to marshal and send an AMI request, that is, the amount of time spent in `read_async`.
- The amount of time we block waiting for the reply to an AMI request, that is, the amount of time spent waiting in `getData`.
- The amount of time it takes to write the data to disk, that is, the amount of time spent waiting for `fwrite` to complete.

We cannot reduce the amount of time to taken to send the `read_async` call, and we cannot reduce the amount of time it takes to write to disk. However, we can reduce the amount of time we block waiting for `getData` to return. We can reduce the effects of network latency by having more than one AMI call active. For this demo, I will use two calls but I could have used several. The first call retrieves the current result while the second call retrieves the next chunk of the file. The server is kept busier that way because, as soon as it has sent the reply to the first invocation, the second invocation most likely has arrived already and is waiting to be dispatched. Similar reasoning applies to the client: once the client has read the first reply, the second reply is likely to be available already for reading by the client-side thread pool. Of course, all this assumes that the network really is the bottleneck—if, instead, the bottleneck is the disk, interleaving the calls will not help matters.

The client loop becomes the following:

```
// C++
string name = ...;
string output = ...;
FileStorePrx store = ...;
int len = 1000*1024;
int offset = 0;
FileStore_readIPtr curr, next;
Ice::ByteSeq bytes;
for(;;)
{
    if(!curr)
    {
        curr = new FileStore_readI;
        next = new FileStore_readI;
        store->read_async(
            curr, name, offset, len);
    }
    else
    {
        swap(curr, next);
    }
    store->read_async(
        next, name, offset + len, len);
    curr->getData(bytes);
    if(bytes.empty())
    {
        break;
    }
    if(fwrite(&bytes[0], 1, bytes.size(), fp)
        != bytes.size())
    {
        cerr << "error writing: "
            << strerror(errno) << endl;
        rc = EXIT_FAILURE;
```

```
    }
    offset += bytes.size();
}
```

We have two AMI callbacks, `next` and `curr`, that we swap each time through the loop. The remainder of the loop remains the same, as does the implementation of the AMI callback.

Can we optimize things further? Consider the current implementation of `ice_response`:

```
// C++
virtual void
ice_response(const Ice::ByteSeq& bytes)
{
    Lock sync(*this);
    _bytes = bytes;
    _done = true;
    notify();
}
```

Note that the data is being copied from the buffer `bytes` into the `_bytes` member variable. In addition, the Ice core copies the data from the marshaling buffer into `bytes`. As a result, 1GB of data returned to the application briefly occupies 3GB of memory: 1GB for the marshaling buffer, 1GB for `bytes`, and 1GB for `_bytes`. We can use the zero-copy feature of the C++ mapping to avoid the copy from the marshaling buffer into the vector. Firstly, we have to add the zero-copy metadata to the interface:

```
// Slice
interface FileStore
{
    ["ami", "cpp:array"] ByteSeq read(
        string name, int offset, int num);
};
```

The AMI `ice_response` callback then becomes:

```
// C++
void
ice_response(const pair<const Ice::Byte*,
        const Ice::Byte*>& bytes)
```

The Ice run time passes a pair of `const Ice::Byte*` to the operation. The first pointer points to the start of the sequence and the second pointer points to one past the end. (These are the same semantics as for the `begin` and `end` methods of STL iterators.) These pointers refer directly to locations in the Ice marshaling buffer, so we can avoid making a copy of the data when storing it in the `_bytes` member:

```
// C++
void
ice_response(const pair<const Ice::Byte*,
    const Ice::Byte*>& bytes)
{
    Lock sync(*this);
    Ice::ByteSeq(bytes.first, bytes.second).swap(
        _bytes);
    _done = true;
    notify();
}
```

Instead of using `swap`, we also could have done something like this:

```
// C++
_bytes.resize(bytes.second - bytes.first);
memcpy(&bytes[0], bytes.first,
    bytes.second - bytes.first);
```

However, this technique is less efficient because the `resize` method zero-initializes each element of the vector, whereas the vector constructor does not.

With this change, the client now requires only 2GB of memory to receive a 1GB chunk (1GB for the marshaling buffer and 1GB for the `_bytes` member variable). Is there a way to avoid this extra copy? It actually is possible, such as by writing the file data directly in the `ice_response` callback. However, we would have to be very careful about doing this. For example, the following is incorrect:

```
// C++
void
ice_response(const pair<const Ice::Byte*,
    const Ice::Byte*>& bytes)
{
    Lock sync(*this);
    fwrite(bytes.first, 1,
        bytes.second-bytes.first, _fp);
    _done = true;
    notify();
}
```

Why is this incorrect? Since we have two AMI calls active at any one time, there is no guarantee as to the order in which the callbacks are invoked (unless we take special precautions and make both client and server single–threaded). If we are running with more than one thread, we have to lock around the calls to seek to the correct offset and write the data. (Note that the code is correct: it is legal to seek past the end of a file.)

```
// C++
void
ice_response(const pair<const Ice::Byte*,
    const Ice::Byte*>& bytes)
{
    Lock sync(*this);
    fseek(_fp, _offset, SEEK_SET);
    fwrite(bytes.first, 1,
        bytes.second-bytes.first, _fp);
    _done = true;
    notify();
}
```

Alternatively, we could open the file twice, once in each callback. That way, the lock is no longer necessary because each callback has its own file pointer. However, although this approach avoids an extra copy of the data, it turns out to be slower because it requires a separate seek for each write. Furthermore, the additional context

switches between the three threads (two client-side threads and the main thread calling `read_async`) is costly.

Note that the size of the chunk being transferred also plays a role: increasing the chunk size consumes more memory but requires fewer remote invocations. For sufficiently large chunks, it is better to do the disk writes in the `ice_response` callback instead of copying the data.

Server

Now let's look at the server side. Here is an initial version:

```
// C++
class FileStoreI : public FileStore
{
public:

    Ice::ByteSeq
    read(const string& name, Ice::Int offset,
         Ice::Int num, const Ice::Current&)
    {
        FILE* fp = fopen(name.c_str(), "rb");
        if(fp == 0)
        {
            FileAccessException ex;
            ex.reason = "cannot open `" + name
                    + "' for reading: "
                    + strerror(errno);
            throw ex;
        }

        if(fseek(fp, offset, SEEK_SET) != 0)
        {
            fclose(fp);
            return Ice::ByteSeq();
        }

        Ice::ByteSeq data(num);
        ssize_t r = fread(&data[0], 1, num, fp);
        fclose(fp);
        if(r != num)
        {
            data.resize(r);
        }

        return data;
    }
};
```

This implementation is straightforward. It opens the file, seeks to the correct location, allocates the correct number of bytes, reads the data, and returns the buffer. Again, let's consider the amount of memory required to send 1GB of data. First, the code allocates a 1GB buffer to hold the file data. When the Ice run time returns the data to the caller, it copies the data into a marshaling buffer, which requires an additional 1GB of memory. Depending on your compiler's support for NRVO, the return value may require an additional allocation and copy. We can avoid this overhead for compilers without NRVO by using asynchronous message dispatch. With AMD, we call into the Ice core to send the data instead of returning it as the return value. As usual, we need to add the appropriate metadata to the interface:

```
// Slice
interface FileStore
{
    ["amd"] ByteSeq read(
        string name, int offset, int num);
};
```

Now the implementation becomes:

```
// C++
void
read_async(const AMD_FileStore_readPtr& cb,
           const string& name,
           Ice::Int offset, Ice::Int num,
           const Ice::Current&)
{
    // ...
    Ice::ByteSeq data(num);
    ssize_t r = fread(&data[0], 1, num, fp);
    fclose(fp);
    if(r != num)
    {
        data.resize(r);
    }
    cb->ice_response(data);
}
```

This approach is more efficient if the compiler does not support NRVO because the additional copy is avoided. Now only one copy is required, namely, the one into the marshaling buffer.

We can do a little better still. The creation of the vector requires initializing it with zeros. This is a little costly and can be avoided by using the zero-copy API (otherwise we need to pass the vector as an argument to the AMD callback object). First, we add the relevant meta-data to the interface:

```
// Slice
interface FileStore
{
    ["amd", "cpp:array"] ByteSeq read(
        string name, int offset, int num);
};
```

Now instead of passing a vector to the AMD callback, we pass a pair of `const Ice::Byte*` pointers. As discussed earlier, the first pointer in the pair points to the start of the array, and the second points to one element past the end of the array. Here is the revised implementation:

```
// C++
void
read_async(const AMD_FileStore_readPtr& cb,
           const string& name,
           Ice::Int offset, Ice::Int num,
           const Ice::Current&)
{
```

```
    // ...
    Ice::Byte[] bytes = new Ice::Byte [num];
    ssize_t r = fread(bytes, 1, num, fp);
    fclose(fp);
    pair<const Ice::Byte*, const Ice::Byte*> ret;
    ret.first = bytes;
    ret.second = bytes+r;
    cb->ice_response(ret);
    delete[] bytes;
}
```

Faster Still

We have made significant improvements so far, but a number of performance problems still remain. For each request, the server has to:

- open and close the file
- seek to the correct location in the file
- allocate the read buffer for each request

By making some assumptions about how the client uses the interface, we can provide further performance enhancements. To do so, we'll add a separate `File` interface:

```
// Slice
interface File
{
    ["ami", "amd", "cpp:array"]
    Ice::ByteSeq next();
};

interface FileStore
{
    File* read(string name, int num);
};
```

In order to read the contents of a file, the client calls `FileStore::read` and receives a proxy to a `File` object. The client then repeatedly calls `next` on the `File` object until it receives an empty sequence, which indicates end-of-file. (The server automatically destroys the `File` object when the client reaches EOF.) With this interface, chunks are retrieved sequentially by the client, that is, the client does not specify a file offset. In turn, this allows the `File` object to cache the file handle and avoid opening and closing the file for each chunk. As an added benefit, the implementation of `next` does not need to call `fseek` to explicitly set the file offset each time.

As before, we want to use AMD and the zero-copy API. We first must add the meta-data:

```
// Slice
interface File
{
    ["amd", "cpp:array"] Ice::ByteSeq next();
};
```

An initial implementation of `File` is as follows:

```
// C++
class FileI : public File
{
public:

    FileI(FILE* fp, int num) :
        _fp(fp),
        _num(num),
        _bytes(new Ice::Byte[num])
    {
    }

    ~File()
    {
        delete[] _bytes;
    }

    void
    next_async(const AMD_File_nextPtr& cb,
        const Ice::Current& current)
    {
        pair <const Ice::Byte*, const Ice::Byte*>
            ret(0, 0);
        ssize_t r = fread(_bytes, 1, _num, _fp);
        if(r == 0)
        {
            fclose(_fp);
            current.adapter->remove(current.id);
        }
        else
        {
            ret.first = _bytes;
            ret.second = _bytes + r;
        }
        cb->ice_response(ret);
    }

private:

    FILE* _fp;
    const int _num;
    char* _bytes;
};
```

The server does not support concurrent calls from the same client as a deliberate implementation choice. There is no benefit for a single client in calling the server concurrently, and adding mutex protection to support concurrency would extract a performance penalty. (Note that this still allows *different* clients to use the same server concurrently—only concurrency from within the *same* client is not supported). We'll enforce the restriction by configuring the server to use the thread-per-connection concurrency model.

```
# config.server
Ice.ThreadPerConnection=1
```

The client remains unchanged, using interleaved AMI calls to achieve optimal throughput.

We can add another potential optimization to the server. The current implementation reads data from the disk and sends it to the

AMD callback in the same thread of execution. This can improve throughput even with a single CPU, due to interleaving of disk I/O and network I/O. However, on multi-CPU machines, we can possibly gain a little by sending the reply from a separate work queue:

```
// C++
void
FileI::next_async(const AMD_File_nextPtr& cb,
    const Ice::Current& current)
{
    Ice::Byte* bytes = new Ice::Byte[_num];
    ssize_t r = fread(bytes, 1, _num, _fp);
    if(r == 0)
    {
        fclose(_fp);
        current.adapter->remove(current.id);
    }
    // _sender is the sender thread work-queue
    // that sends r bytes of data to the callback
    // object cb.
    _sender->add(cb, bytes, r);
}
```

Note that this approach requires more memory, as each call to `next_async` must allocate a new byte buffer that is added to the queue for later transmission to the given callback object. Whether or not this actually provides a benefit depends on your hardware and library implementation. You need to run benchmarks to find out for your particular platform. (On single-CPU machines, the technique is almost certainly detrimental due to the extra copy and context switching.) You can find the full implementation in the source code.

It is also worth looking at these changes to examine their impact on the server and client. First, by creating a separate `File` object (instead of allowing the client to specify a file offset with each call to retrieve a chunk), we have made a previously stateless interface stateful. This is often undesirable because it requires the client and server to be in agreement as to the current state of the object. If anything goes wrong during the transfer of the data from the server to the client, the transfer must be terminated, as the client cannot, as the interface stands, reset the state of the `File` object. Second, we have introduced a new object that must be cleaned up in the event that the client misbehaves. For example, the client could create a `File` object but never use it, so the server must have a mechanism to clean up such abandoned objects (see Issue 3 of *Connections*). Certainly, both of these problems can be solved, however, at what cost? The initial implementation was stateless and certainly can be made more efficient by caching more information. As always with optimization efforts, you have to make a judgment call as to whether the performance gain is worth the cost.

Bare Wire

This is as fast as we can get with Ice. However, it is possible to go faster still by using straight sockets for the file transfer, which allows you to avoid the overhead of using Ice for the most performance-critical parts of your application. The key idea is that Ice is used to facilitate the transfer, but not to do the transfer itself. (You can think of this in the same way as using some assembly language in C++ applications to speed up critical pieces of your code.) But why use Ice at all then? Consider the alternative: without Ice, you would not only have to transfer the file with sockets, you would also have to work out some way to organize the file transfer, send the file name, get the number of bytes expected, and so on. All of this adds up to a creating a protocol—which is precisely what we don't want to have to deal with. Consider the following Slice definitions:

```
// Slice
module Demo
{

exception FileAccessException
{
    string reason;
};

interface FileStore
{
    File* read(
        string name, out string ip, out int port)
        throws FileAccessException;
};
};
```

The client calls `read` on the `FileStore` interface with the name of the file. The call returns a source IP address and a source port. The client then connects to this address and port and reads the file data until it has received all of the bytes. Since with straight TCP/IP there is no maximum message size to worry about, we do not need to tell the server the chunk size. The server writes in whatever chunk size it chooses and the client reads in whatever chunk size it chooses—TCP/IP buffering takes care of the rest.

Now consider what the client would look like with this scheme:

```
// C++
FILE* fp = ...;
int fd = connect(...);
vector<char> bytes(len);
while(true)
{
    ssize_t rx = recv(fd, &bytes[0], len, 0);
    if(rx == -1)
    {
        // Error.
    }
    if(rx == 0);
    {
        break;
    }
```

```
    if(fwrite(&bytes[0], 1, rx, fp)
        != static_cast<size_t>(rx))
    {
        // Error
    }
}
```

How does the client know when to stop reading? The apparent answer is when the server closes the connection. However, how does the server know when to close the connection? If the server closes the file descriptor as soon as it has sent all of the data, then any unsent buffered data will be lost. This means that the server and client must agree as to when to close the connection. Again, this looks remarkably like protocol design. Let's modify the interface to deal with this issue:

```
// Slice
module Demo
{

exception TransferException
{
    string reason;
};

interface File
{
    void start(out string ip, out int port,
        out int bytes)
        throws TransferException;
    void destroy();
};

exception FileAccessException
{
    string reason;
};

interface FileStore
{
    File* read(string name)
        throws FileAccessException;
};
};
```

The client first calls `read` on the `FileStore` interface to obtain a proxy to a `File` object, in much the same way as the interface I described in Faster Still. The client then calls `start` on the `File` object to set up the socket connection. This returns the source IP address and the source port, and the total number of bytes that will be transferred. The client then connects to this address and port and starts reading data until all of the bytes have been transferred. At that point, the client calls `destroy` on the `File` object to complete the transfer.

Once again, we could optimize the reading and transmission of the data by using separate reader and writer threads in the server. However, depending on how many CPUs your machine has, this may introduce a performance penalty.

This last interface also introduces state into the server and is significantly more complex than the original stateless solution. Whether or not the optimization makes sense depends on the amount of the performance gain, and most importantly whether your application *requires* this performance gain.

Measuring Results

With performance tuning, the most important thing is benchmarking. Before I go on to that, I want to point out that benchmarks quite often lie. For example, the two most common middleware benchmarks are latency and raw throughput. They measure how much time a single (empty) RPC consumes, and how quickly the middleware can pump data through the network. However, if you are using high-performance middleware such as Ice, for most applications, neither of these things matter in the *slightest*.

If latency is an issue for your application, there is a very good chance that the reason is not poor middleware performance, but incorrect design. In general, interfaces for distributed applications should be coarse grained, that is, they should not have lots of trivial operations that must be called many times to perform a unit of work. Instead, interfaces should have fewer operations that each do a lot of work. As an example, consider an interface that holds a collection of data:

```
// Slice
interface Query
{
    QueryResult next();
};
```

The client calls `getNext` once for each element in the query set. This is a typical example of an inefficient interface. Here is a better version:

```
// Slice
interface Query
{
    QueryResultSeq get(int offset, int len);
};
```

Not only is this interface stateless, it also returns multiple results with a single call and so avoids the latency inherent in the earlier version.

With that out of the way, here is how I tested the various optimizations in this article. In all cases, the client runs on a MacBook 2.0GHz dual-core machine (OS X 10.4.x) with 2GB of memory and a 160GB 5200rpm hard drive, and the server runs on 3.0GHz machine (Fedora Core 4) with 1.25GB of memory and an 80GB 7200rpm hard drive. I ran all tests using an optimized build of Ice 3.1.1.

The two machines were connected with a cross-over cable. I used the same 266MB file for each transfer. For each test, I first ran the server and transferred the file to the client once before taking timings, to allow the file to be cached on the server side first.

To get the timings, I ran each transfer 50 times and computed the average of throughput in megabytes per second. For each test, the machines were otherwise idle.

I conducted the first test over a 100Mbps network. The table below shows the throughput results in MB/sec.

Client/Server	Synchronous	AMD	AMD Zero-Copy
Syncrhonous	8.475	8.690	8.771
AMI	9.720	9.963	10.045
Interleaved AMI	10.813	11.020	11.077

As you can see, the synchronous tests are slowest and the AMI zero-copy and AMD zero-copy tests are fastest, which is the outcome we would expect. The biggest gain comes from using simple AMI and AMD, while moving to interleaved invocations and using the zero-copy API only provides moderate performance improvements. Next, how does the stateful version that uses an explicit `File` object stack up?

Stateful Client/ Server	Single Threaded	Threaded
Single Threaded	11.135	11.127
Threaded	-	11.066

As the numbers show, that version only provides a very modest performance gain over the previous version. Note that the version that sends the replies in a second thread is actually slower than the non-threaded version, due to the fact that the server runs on a machine with a single CPU. (To try the test on your machine, you can control whether the server uses a dedicated sender thread with the `SenderThread` property in the `config.server` file.)

How about the version that uses sockets?

Socket Client/ Server	Single Threaded	Threaded
Single Threaded	11.053	11.072
Threaded	11.212	11.213

Interestingly, the single-threaded version of the socket client is slower than the stateful version above because it reads no data from the network while data is written to disk. Because I ran this test on a dual-core machine, it is slower than the multi-threaded version. The socket version is also not significantly faster than the asynchronous versions that use Ice; however, the source code is significantly more complex and less portable. As with the stateful demo above, you can control whether the client and server use a separate thread for reading and writing the data to the network via configuration properties (see the files `config.client` and `config.server` for details).

For comparison, I re-ran all the tests using a gigabit network. The results are as follows:

Client/Server	Synchronous	AMD	AMD Zero-Copy
Syncrhonous	25.918	37.710	28.501
AMI	32.776	33.265	33.492
Interleaved AMI	33.172	33.978	33.425
AMI Zero-Copy	33.362	33.323	33.690

As before, the asynchronous versions are faster than the synchronous versions. However, the other optimizations now no longer make a difference. Next, the timing for the stateful version:

Stateful Client/ Server	Single Threaded	Threaded
Single Threaded	33.165	33.800
Threaded	-	33.758

And finally the timing information for the socket version:

Socket Client/ Server	Single Threaded	Threaded
Single Threaded	33.430	33.022
Threaded	35.598	33.865

One thing that immediately stands out is that, although this test uses a gigabit network that is supposedly ten times faster than a 100-megabit network, the throughput figures are only three times faster! Furthermore, in contrast to the 100Mbit network, zero-copy and interleaving provide no performance improvement for throughput. (Of course, this ignores the memory benefit provided by zero-copy.) Why is this? As I mentioned earlier, we have to consider the following variables when it comes to performance:

- The amount of time it takes to marshal and send an AMI request, that is, the amount of time spent in `read_async`.
- The amount of time we block waiting for the reply to an AMI request, that is, the amount of time spent waiting in `getData`.
- The amount of time it takes to write the data to disk, that is, the amount of time spent waiting for `fwrite` to complete.

Moving to a gigabit network reduces the amount of time for the first two points. However, the time taken to write to disk is the same. Once that time becomes greater than the amount of time required to transfer the data over the network, reduction in latency no longer matters and the disk becomes the bottleneck! A few simple measurements show the sustained write speed of the disk on the MacBook to be around 33MB/sec. And that is exactly the throughput evident in the results for the gigabit network.

As I said earlier, for many applications, raw throughput is irrelevant (as long as your middleware doesn't do something really stupid, such as sending the data as an XML document). Most applications are not held back by Ice—instead, they are held back by what they do with the data.

For the record, here are the results if I comment out the calls to `fwrite` in the code. (Keep in mind, although these numbers look great, they are irrelevant!) The socket version shows throughput of around 74MB/sec, and the stateful Ice version shows a throughput of around 61MB/sec. With the disk bottleneck removed, since the entire file is in cache on the server side, the network becomes the bottleneck.

Finally, I wanted to see what happens if we send the data over a WAN. For this case, I measured with various transfer block sizes. All of these results are in KB/sec instead of MB/sec because, over a WAN, things are significantly slower. The file size transferred in all cases was 3493161 bytes.

First the results for a transfer block size of 10KB:

Client/Server	Synchronous	AMD	AMD Zero-Copy
Synchronous	44.24	44.16	44.36
AMI	44.36	44.12	44.11
Interleaved AMI	54.50	54.72	54.34
AMI Zero-Copy	54.54	54.67	54.23

Next, for a transfer block size of 100KB:

Client/Server	Synchronous	AMD	AMD Zero-Copy
Synchronous	53.50	53.28	53.67
AMI	53.56	53.44	53.77
Interleaved AMI	56.95	56.79	56.75
AMI Zero-Copy	56.67	56.97	56.88

Next, for a transfer block size of 1000KB:

Client/Server	Synchronous	AMD	AMD Zero-Copy
Synchronous	56.83	56.96	56.55
AMI	56.75	56.74	56.75
Interleaved AMI	56.60	56.83	57.19
AMI Zero-Copy	57.02	57.07	56.95

As you can see, for the 10KB block size, interleaved AMI makes a significant difference. This is as expected because, with such a small transfer size, latency becomes a big issue. By using interleaved AMI, we reduce the negative effects of network latency. As the block size gets larger, network latency is less of an issue because we make fewer remote calls. Zero-copy has no effect at all in these tests because the network is the bottleneck and we have CPU cycles to spare.

The obvious question then is why you would use a small block size over a WAN when you might just as well use a larger block size and save yourself the trouble of interleaved AMI?

If the server handles many thousands of clients, a large block size becomes difficult due to memory limitations. For example, consider a 1MB block size. If there are 1024 clients concurrently transferring files, that adds up to a total 2GB in the server, 1GB for the data buffers, and 1GB for the marshaling buffers. While 2GB of memory for modern-day machines is not such a big deal, consider what happens if the load increases to 5,000 or 10,000 clients! (Of course, if we need to transfer a large number of small files, we cannot get away from the small block size, unless we use more complex algorithms to send multiple files in a single transfer block.)

Conclusion

As dual and multi-core machines become more and more common, designing your applications to take advantage of this extra processing power is important. Fortunately, Ice makes this simple. By using asynchronous method invocations, your applications can easily take advantage of multiple threads without actually having to write complicated threaded code yourself. Furthermore, by using the zero-copy API that Ice provides, it is possible to reduce the memory footprint and increase the performance of bulk data transfers. The tests show that, even for a scenario where it should be very difficult for general-purpose middleware to compete (namely, file transfer), Ice presents essentially no overhead compared to an optimal implementation that uses plain sockets. More importantly, the Ice code is significantly easier to write and maintain. If your application runs over the internet (that is, over a WAN, not a LAN), zero-copy does nothing to increase throughput, but will reduce memory footprint. The most profitable optimization for file transfer is interleaved AMI because it reduces the effects of latency on data transfer.

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: How can I speed up large requests?

For large requests, the network bandwidth is usually the most important factor that limits performance. However, with high-speed (gigabit) networks, bandwidth becomes less of an issue and marshaling overhead and memory management become significant as well. The Ice protocol enables very efficient marshaling and does not require developers to do anything special to achieve optimal performance. However, the memory management of a program can have an impact on overall performance.

The performance of `malloc/free` varies greatly from system to system. Beside the obvious issue of raw hardware speed, performance depends on the quality of implementation of the memory manager. But even high-quality implementations may use trade-offs that can adversely affect the performance of your application. For example, different memory managers use different strategies to allocate blocks of different sizes. For this discussion, we will focus on Linux.

A simple test to measure `malloc` performance is to allocate and free a block in a loop and divide the total time taken by the number of iterations:

```
// C++
const size_t blockSize = 4 * 1024 * 1024;
IceUtil::Time t = IceUtil::Time::now();
int j = 0;
for(j= 0; j < 1000000; ++j)
{
    char* b = (char*)malloc(blockSize);
    free(b);
}
t = IceUtil::Time::now() - t;
cout << t.toMicroSecondsDouble() / j << endl;
```

The following table illustrates the relationship between the time taken by `malloc` and the size of the requested block:

Size (bytes)	Time (microseconds)
240	0.09
1920	0.09
7680	0.08
15360	0.08
61440	0.08
122980	0.09
245760	6.7
512000	6.8
675000	7.4
1000000	7.5
2000000	10.0
4000000	13.0

Notice the large increase in allocation time when going from 122,980 bytes to 245,760 bytes. This increase is due to the default `glibc` behavior. The `glibc malloc` is based on a successful general-purpose memory allocator by Doug Lea known as the Lea allocator. The Lea allocator recycles freed memory, avoids expensive system calls, and vastly improves `malloc/free` performance. However, `glibc`'s default behavior is to treat memory requests above a certain size differently—blocks above a certain size are allocated via `mmap` instead of using recycled blocks. Calls to `mmap` are more expensive than using the recyclable memory already held by the allocator. To make things worse, `malloc` continuously trims unused memory in order to reduce the amount of recyclable memory. This has the benefit of reducing the overall memory footprint of the process, but is costly if an application repeatedly triggers trimming by frequently allocating and freeing large blocks.

The objective then is to configure `malloc` so that it treats allocations up to `Ice.MaxMessageSize` in the most efficient manner possible. The configuration options we will use are `M_MMAP_MAX` and `M_TRIM_THRESHOLD`. Setting `M_MMAP_MAX` to 0 disables allocating large buffers through `mmap`. We also need to alter the default trimming settings. We will disable trimming entirely for our purposes here, but you may want to experiment to see whether there are values that are more appropriate for your application. You configure `malloc` either through the `mallopt` call or via environment variables.

To disable use of `mmap` and trimming, you can make the following two calls to `mallopt` in your code:

```
// C++
mallopt(M_MMAP_MAX, 0);
mallopt(M_TRIM_THRESHOLD, -1);
```

Alternatively, you can set corresponding environment variables:

```
MALLOC_MMAP_MAX_=0
export MALLOC_MMAP_MAX_
MALLOC_TRIM_THRESHOLD_=-1
export MALLOC_TRIM_THRESHOLD_
```

Here are the timings for `malloc` with blocks of different size with the default settings, and with `mmap` and trimming disabled:

Size	Default `malloc` parameters (microseconds)	`mmap` and trimming disabled (microseconds)
240	0.09	0.08
1920	0.08	0.08
15360	0.08	0.08
61440	0.08	0.08
122980	0.09	0.08
245760	6.7	0.08
512000	6.8	0.08
675000	7.4	0.08
1000000	7.5	0.08
2000000	10.0	0.08
4000000	13.0	0.08

For buffers that would normally be trimmed or allocated through `mmap`, the difference is staggering and, for large requests, has a direct impact on throughput performance. Using the Ice throughput test, we can see the impact of tuning on throughput.

	default `malloc()` parameters MB per second	`mmap()` and trimming disabled MB per second
throughput byte sequence 500,000 elements	190	260

System specs: 3 GHz Pentium 4 with HT, 1 GB RAM, CentOS 4.4

With Linux, you can use `mallopt` to configure memory management behavior. If your target system uses an allocator without similar options, you might consider a drop-in `malloc` replacement, such as the Hoard allocator or SmartHeap.

Q: Why do I get an `UnmarshalOutOfBoundsException`?

Consider the following interface:

```
// Slice
interface Example {
    void op(int i, int j);
};
```

When a client invokes operation `op`, it supplies two integer parameters. The Ice run time marshals the parameters into the request that it sends on the wire. The request is preceded by a protocol header that, among other things, tells the server the total request size (including the 14-byte header). The server uses this information to read the appropriate number of bytes: it first reads the 14-byte header and checks the total request size. For example, that size might be 31 bytes. The server then reads the remainder of the request, namely, 31−14=17 bytes and places these 17 bytes into a buffer for subsequent unmarshaling. The buffer keeps track of how many bytes of the payload are available for reading and prevents attempts to read more data than was actually contained in the request.

The 17-byte payload of the request contains 8 bytes for the two integer parameters. (The remainder of the payload contains other details about the request, such as the object identity and request ID, among other things.) Once the server has identified the correct target operation, it dispatches the incoming request to the Slice-generated skeleton, which contains the code to unmarshal the two integer parameters. The skeleton retrieves 8 bytes from the unmarshaling buffer and converts them back into integers, and then passes these two integers to the implementation of operation `op`.

If the unmarshaling code in the skeleton tries to retrieve more data from the buffer than is actually available, the buffer raises an `UnmarshalOutOfBoundsException`. In plain language, the exception means “I expected a certain amount of data for the parameters of an invocation, but there wasn’t as much data available as there should be.” (Similar arguments apply when a client unmarshals the results of an invocation. In that case, the code goes through much the same actions, except that the unmarshaling is done by a proxy instead of a skeleton, and the data is contained in a reply instead of a request.)

So, how can this exception happen? Unless you are using the dynamic invocation or dispatch interfaces, the cause of this exception is invariably a mismatch in the Slice definitions that are used by client and server. For example, suppose that, originally, the preceding interface looked as follows:

```
// Slice
interface Example {   // Earlier version
    void op(int i);
};
```

During development, you decided to add the second parameter, to turn the interface into the two-integer version we saw earlier. Suppose you faithfully updated the server with the new version of the interface but, for some reason, you forgot to update the client. When you run client and server, the client runs with the old interface, but the server runs with the new interface. Of course, this means that the client will send only a single integer when it invokes `op`, but the server expects to receive two integers. This results in an `UnmarshalOutOfBoundsException` on the server side. (If we reverse the situation, such that the client expects two out-parameters, but the server sends only one, you would see the same exception on the client side instead.)

So, in short, `UnmarshalOutOfBoundsException` is invariably caused by mis-matched Slice definitions, unless you are using the dynamic invocation or dispatch interfaces (in which case your code contains the mismatch).

Note that you can easily catch Slice mismatches at run time by adding the `--checksum` option when you compile your Slice definitions. This option creates a dictionary in the generated code that contains a checksum for each Slice type. You can add an operation to the server that returns the dictionary to the client and verify in the client that the checksums for corresponding Slice types are the same; if they differ, you have mismatched Slice definitions. (Please see the Ice Manual for more details on Slice checksums.)