Choosing Middleware: Why Performance and Scalability do (and do not) Matter

Michi Henning, Chief Scientist, ZeroC, Inc.

Introduction

Apart from Ice, WCF (Windows Communication Foundation) and Java RMI (Remote Method Invocation) are currently seen as major contenders in the middleware space. When it comes to making a choice among them, performance is often taken as the sole evaluation criterion, despite the fact that performance is only one of many factors that influence the choice of middleware. This article provides a performance and scalability comparison of the three middleware platforms, and discusses when performance and scalability matter and when they do not, including their likely impact alongside other factors on the overall cost of a project. Finally, for those applications that indeed require high performance and scalability, the article points out a few techniques you can use to get the biggest bang for your buck.

Contents

ZeroC

Why Bother Measuring Performance?

When it comes to evaluating middleware platforms, the first reaction often is to devise a number of benchmarks and to put the different implementations though their paces. The middleware that "goes fastest" is deemed to be "best."

Unfortunately, this process may well disappoint. Ironically, one reason is that performance may well be irrelevant to the application. While performance is indeed important and can have a large impact on operating cost, using it as a major evaluation criterion makes sense only if the application indeed requires high performance. However, for many applications, middleware performance matters little, either because the overall load is never high enough to make middleware performance an issue, or because bottlenecks are inherent in the work the application has to do, and no amount of improvement in the middleware would significantly improve things overall.

When it comes to evaluating middleware, there are likely to be evaluation criteria that are far more important. Here is a small selection:

- How steep is the learning curve of the middleware? Is it possible for developers to start writing production-quality code without having to go through a long learning process?
- What is the quality of the APIs? Are APIs consistent and easy to learn? Are APIs thread-safe and exception safe? To what degree are they statically type safe? Is memory management automatic or must it be taken care of by the programmer?
- What is the quality of the documentation? Is it easy to find reference material that is clear and comprehensive? Does the documentation provide sufficient tutorial-style material and source code examples for programmers to become productive quickly?
- Does the middleware have a vibrant developer community that actively exchanges experiences and provides peer support?
- Is high-quality professional training available?
- Is the source code available?
- How many operating systems and programming languages are supported by the middleware? Are these relevant to the application? How easy is it to port the middleware to a new platform?
- What level of support is provided by the vendor? Is it possible to obtain priority support and hot-fixes if there is a problem?
- How reliable is the middleware? Are defects that are found in the middleware rectified promptly?
- How complete is the feature set of the middleware? Does the middleware offer features that are irrelevant to the application, but carry a penalty in terms of performance or complexity of the APIs?
- Does the middleware provide just a basic RPC mechanism, or does it include essential services, such as an event service, administrative tools, a persistence service, and so on? How much effort is required to build services that are missing but needed by the application?
- Does the middleware make efficient use of resources such as memory, file descriptors, CPU cycles, and network bandwidth?

ZeroC

- How well does the middleware scale for large applications? Is it possible to scale an application by adding more servers without disturbing an already deployed system?
- What are the licensing conditions? Is it possible to obtain the middleware under GPL? If the GPL is not an option, are the licensing conditions competitive?

These criteria are only a sample; most real-world projects could extend this list with other points, each of which might make the difference between success and failure. Unfortunately, most of these criteria are hard to quantify. It is easy to state that "middleware X moves twice as many bytes per second than middleware Y"—conversely, it is much harder to quantify the impact of poorly-designed and complex APIs on overall development time, testing effort, and eventual defect rate. Moreover, modern middleware has reached a level of complexity where only a handful of experts have the experience to correctly judge the relative merits of various design and feature trade-offs.

In other words, many benchmarking exercises for middleware are undertaken not because they are necessary, but because they are easy. Having gone through the "evaluation process," participants feel that they have exercised due diligence, even though the results may well be irrelevant.

Despite all this, there *are* applications for which performance and scalability are important. This is particularly true for servers that must provide service for thousands or tens of thousands of clients. In that case, the performance of the middleware has direct impact on operating costs in terms of the required network bandwidth and the amount of hardware that is needed to support the load.

So, with the foregoing in mind, the following sections provide a number of performance results for Ice, as well as for WCF (Windows Communication Foundation) and Java RMI (Remote Method Invocation). We chose these platforms because they are often considered as alternatives to Ice, especially since they come bundled with their respective SDKs.

Test Setup

Unless otherwise mentioned, I obtained the test results using two machines:

- Dual-core 2.2GHz Athlon with 2GB of memory running Windows XP Professional SP3.
- Dual-core 2.0GHz Mac Mini with 2GB of memory running Windows Vista Ultimate SP1.

The compiler for the C++ and C# code was Visual Studio 2008, .NET version 3.5. For Java, I used the 1.6.0 JDK. Code was compiled with optimization for speed, where applicable. For the Vista machine, I generated 64-bit code. The Ice tests used version 3.3.0.

The loopback test results were obtained using the dual-core Athlon running Windows XP. For tests using the network, I ran the client on the Windows XP machine, and the server on the Vista machine.

Latency

One fundamental measure of middleware performance is latency. Latency is the amount of time it takes for a client to invoke a twoway operation in a server and to receive the results of the operation. The lowest possible latency for Ice can be achieved with the following definition:

```
// Slice
interface Latency {
    void o(); // Shortest possible operation name
};
```

The operation o has no in-parameters, so no data is sent with the invocation. Similarly, because o does not have out-parameters and has a void return value, no data is returned by the operation.

When a client invokes an operation, the Ice protocol sends the name of the operation and the object identity of the target object to the server. Therefore, we get the shortest possible request size by using a single-letter operation name and by using a single-letter object identity for the target object. In this case, the Ice protocol sends 32 bytes to invoke the operation, and the server returns a reply of 25 bytes.

C++ Latency

In order to judge the performance of Ice with respect to latency, we need to first establish the theoretical minimum latency (without Ice) of a client and server that exchange the same amount of data. Doing this is simple: we can write a C++ client and server that perform the same actions as the Ice run time, but use native sockets instead. For each request, the client writes 32 bytes (corresponding to a request) onto the wire and then reads 14 bytes (corresponding to the Ice protocol header) followed by 11 bytes (corresponding to the payload in the server's reply). The server takes the opposite actions, that is, it first reads 14 bytes (corresponding to the Ice protocol header) followed by 18 bytes (corresponding to the payload in the client's request) and then writes 25 bytes (corresponding to the reply).

In order to eliminate the effects of jitter (variations in the latency of individual invocations), the invocations are performed in a loop that iterates 100,000 times.

When I run the client and server on the 2.2GHz dual-core Athlon over the loopback interface, I get around 18,000 messages per second, so each "message" over native sockets takes about 55µs.

Repeating this experiment with the corresponding Ice for C++ client and server (also over the loopback interface), I get around 10,500 messages per second, so each message takes about 95µs.

You may be appalled by this number, seeing that the latency with Ice over the loopback interface increases by roughly two thirds compared to the native socket case. However, this reflects the fact that the socket client and server do not do anything with the data they send and receive; what they measure is simply the latency of the TCP/IP implementation over the loopback interface. For a real Ice client and server, the additional latency reflects the need to marshal and unmarshal the request and reply and, in the server, to locate the servant and dispatch to the operation implementation. In other words, if we were to add code to the socket client and server to parse the protocol header, ensure that the data received over the wire is internally consistent, invoke an operation implementation, and return results to the client, we would end up with very much the same latency figures as with Ice.

For real client–server applications, it is rare for client and server to run on the same machine. More often, client and server run on different machines and communicate over a network. To get the latency for this (more realistic) case, we can repeat the preceding experiment and run client and

ZeroC

server on different machines that are connected via a gigabit network. The outcome is that both the socket version and the Ice for C++ version run with identical latency, namely 2,300 messages per second (435µs per message).

There are two important points in this simple experiment:

- The latency when communicating over a network is significantly larger than when communicating over the loopback interface.
- Improving Ice to further reduce latency is pointless.

You may be surprised at the second point. However, consider that, even if we were to improve Ice to the point where its latency overhead becomes zero (which is impossible), it would not make an iota of difference: when client and server communicate over a network, the latency incurred by Ice is already so low that it can no longer be noticed.

For the case where client and server run on the same machine, improving latency further would also not yield any noticeable improvement. The results we obtained by comparing the socket version with the Ice version are misleading because neither case is sensible: for real applications, the server does not simply do nothing when it receives an operation invocation, and it normally either receives or returns data. If the server performs any processing inside the operation (or if the client performs any significant processing between operation invocations), we get a different picture. For our empty Ice for C++ test server, we managed to process 10,500 messages per second from a single client. If, per operation, the server spends 10µs (hardly any time at all), we get around 9,500 messages per second; if the server spends 75µs, we get 5,900 messages per second; if the server spends 150µs (still hardly any time at all), we get 4,100 messages per second.

In other words, once the server performs any processing on a request, the number of messages per second drops rapidly and end-to-end performance is completely dominated by the actual processing cost; the latency quickly dwindles into insignificance. Our benchmark distorts reality because it measures *only* the latency of the machine's TCP/IP implementation and the efficiency of the Ice run time, and ignores the fact that real applications perform work that—at least for well-designed applications—far outweighs the latency of message exchange.

The preceding figures do *not* mean that an Ice server can handle only 10,500 messages per second over the loopback interface. In fact, with our test setup, both client and server spend a large amount of time twiddling their thumbs: the client does nothing while the server processes a request and, similarly, the server does nothing while it waits for the client to send the next request.

We can double the workload for the server by running two clients concurrently. In this case, for the same server, we get 8,000 messages per second for each client, that is, 16,000 messages per second overall, even though the server is still single-threaded: the additional client helps to ensure that the server does not run out of work. Because I ran these tests on a dual-core machine, further increasing the number of clients (or the number of threads in the server) worsens latency again. This is because, with three active processes, on average, at least two of them can use a CPU while the third process waits for a network I/O to complete; that way, both CPUs are fully utilized, and adding extra processes or threads only adds context-switching overhead without gaining more CPU cycles.

5   **ZeroC**

If I run the client and server on a 2GHz eight-core OS X machine, the server processes up to 35,000 requests per second. (This figure is for a server with two threads and five clients, so each client completes 7,000 requests per second.) This leaves one of the eight cores unused. With a sixth client, the number of messages per second drops to 33,000; similarly, with three threads in the server and five clients, the number of messages also drops. Looking at the CPU utilization for the best case (35,000 messages per second), we find that only 65% of the available CPU time is used. In other words, the bottleneck for this case is no longer the Ice run time, but the loopback adapter: the device driver maxes out at around 35,000 messages per second, and adding more clients or threads in the server only increases context switching overhead.

.NET and Java Latency

Ice for C++ is the speed champion in the suite of Ice implementations because C++ allows us to implement optimizations that safe languages, such as C# or Java, do not permit. Here are the latency figures over loopback and over a gigabit network for Ice for .NET and Ice for Java. For comparison, the table also shows the latency for Ice for C++:

Requests/second	Ice for .NET	Ice for Java	Ice for C++
Loopback	6,900	8,000	10,500
Gigabit network	2,300	2,300	2,300

As you can see, once we make invocations over the network, the different implementations perform within a few percentage points of each other. (As I will show later, on different hardware or with a different operating system, the relative differences can change considerably.)

Throughput

Another main measure of middleware performance is throughput. Throughput determines how much data the middleware can move per unit of time.

Ice for C++ Throughput

Throughput for Byte Sequences

Throughput is heavily dependent on the type of data that is being sent. The most efficient case is the transmission of byte sequences. Because these do not require further processing (such as byte swapping) and because they can be cheaply copied between the application and the Ice run time, byte sequences provide the highest throughput.

To test throughput for byte sequences, we can use the following operation definition:

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

interface Throughput {
    void sendByteSeq(ByteSeq b);
};
```

ZeroC

If we send a sequence of 500,000 bytes via this interface to a server (again, with a single-character object identity), the Ice protocol sends a request of 500,047 bytes and the server returns a reply of 25 bytes.

As for the latency test, we can write a client and server using sockets to establish the basic capabilities of the TCP/IP implementation for the same hardware and OS (2.2 GHz dual-core Athlon, Windows XP). The client and server send and receive the same amount of data as Ice, but do not perform any processing on the data. For this client and server, I can complete 1,000 "invocations" in 2.9 seconds over the loopback interface, which works out to 1.3 Gbit/s. (This figure only counts the actual payload, that is, 1,000 times 500,000 bytes and ignores the protocol overhead of 47 bytes for the request and 25 bytes for the reply.)

Repeating the same experiment with Ice, I get almost the same performance of 1.2 Gbit/s. In other words, the limiting factor is not so much the Ice run time, but the speed of the loopback interface and the rate at which data can be copied.

Note that I obtained these figures using the zero-copy API for byte sequences. If I instead use the standard API, performance is slightly lower at 1.05 Gbit/s. The difference reflects the additional copy of the data on the server side.

Here is an operation that we can use to transfer a byte sequence from server to client:

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

interface Throughput {
    ByteSeq recvByteSeq();
};
```

As it turns out, for receiving data, Ice is not as fast: on the same hardware, we can receive 960Mbit/s. The difference is due to the fact that, for return values, the Ice run time performs an extra data copy; the test result reflects the cost of this copy.

Note that, depending on the hardware, this difference may be larger. For example, on an eight-core OS X machine, the same test produces 3Gbit/s for sending and 1Gbit/s for receiving. In other words, the test is sensitive to the specifics of the hardware and the implementation of the TCP/IP loopback interface, as well as the optimizations applied by the compiler.

As I mentioned for the latency test, client and server rarely run on the same machine. We can get a more realistic idea of throughput by running client and server on different machines over a gigabit Ethernet. In that case, throughput is 750Mbit/s for sending and 650Mbit/s for receiving. This is not quite at the limit of the amount of data we can move over a gigabit network. As for the latency demo (but more so in this case), client and server spend large amounts of time twiddling their thumbs, waiting for I/O to complete. We can push the network to its limit by running two clients and two threads in the server. That way, the server can service a request from one client while waiting for data to be written to the other client, which increases the chance that the server will not run out of work to do. With this setup, I measure 940Mbit/s (470Mbit/s for each client) for sending, and 740Mbit/s (370Mbit/s for each client) for receiving.

ZeroC

In other words, for byte sequences, Ice can quite easily saturate a gigabit network for transmitting data and come close to saturation for receiving data.

Throughput for Structured Data

How relevant are the test results for the preceding byte sequence tests? If you happen to be interested in transferring large amounts of binary data, obviously, these tests are relevant. However, most applications do not exchange binary data, but structured data, so it makes sense to look at a few scenarios that involve more complex data.

Here is a definition that transfers a sequence of structures containing fixed-length members:

```
// Slice
struct Fixed
{
    int i;
    int j;
    double d;
};
sequence<Fixed> FixedSeq;

interface Throughput
{
    void sendFixedSeq(FixedSeq seq);
    FixedSeq recvFixedSeq();
};
```

Each structure contains sixteen bytes of data. Over a gigabit network, for a sequence containing 50,000 structures, Ice for C++ manages 500Mbit/s and 420Mbit/s for send and receive, respectively. Note that the difference between send and receive here is proportionately smaller than for byte sequences. This is because the more complex data requires more complex marshaling and unmarshaling actions: instead of performing a simple block copy of the data (as is possible for byte sequences), the run time marshals and unmarshals each structure member individually. In other words, marshaling and unmarshaling of complex data is more CPU-intensive than marshaling and unmarshaling of byte sequences; the higher CPU overhead therefore reduces the relative impact of the extra data copy when receiving data.

We can stress the Ice run time in a different way by replacing the integers of the previous test with a string:

```
// Slice
struct StringDouble
{
    string s;
    double d;
};
sequence<StringDouble> StringDoubleSeq;
```

Sending sequences of 50,000 such structures containing the string "hello" and the value 3.14 over a gigabit network, Ice manages 250Mbit/s for send and 135Mbit/s for receive. This is a fair bit slower

ZeroC

than the test result for fixed-length sequences. The difference is due to the need to allocate and copy the data for each string member. These initializations require a considerable amount of time. In contrast, if we repeat the test with string members containing 100 bytes each, throughput increases to 500Mbit/s and 300Mbit/s, reflecting the decreased cost of string initialization.

.NET and Java Throughput

Here are throughput figures for Ice for .NET and Ice for Java over the loopback interface. For reference, the table also summarizes the preceding results for Ice for C++:

Throughput (loopback)	Ice for .NET	Ice for Java	Ice for C++
Byte seq (send)	630Mbit/s	800Mbit/s	1,200Mbit/s
Byte seq (recv)	610Mbit/s	720Mbit/s	960Mbit/s
Fixed seq (send)	380Mbit/s	140Mbit/s	620Mbit/s
Fixed seq (recv)	300Mbit/s	110Mbit/s	530Mbit/s
Variable seq (send)	68Mbit/s	65Mbit/s	190Mbit/s
Variable seq (recv)	62Mbit/s	70Mbit/s	150Mbit/s

Note the sharp performance drop-off for variable-length structures containing a 5-byte string; the large number of small memory allocations has significant cost. If we run the same test over the network, we get rather surprising results:

Throughput (gigabit network)	Ice for .NET	Ice for Java	Ice for C++
Byte seq (send)	520Mbit/s	660Mbit/s	740Mbit/s
Byte seq (recv)	410Mbit/s	590Mbit/s	655Mbit/s
Fixed seq (send)	300Mbit/s	250Mbit/s	525Mbit/s
Fixed seq (recv)	205Mbit/s	150Mbit/s	470Mbit/s
Variable seq (send)	55Mbit/s	180Mbit/s	255Mbit/s
Variable seq (recv)	55Mbit/s	75Mbit/s	145Mbit/s

Note that, over the network, throughput is actually higher in some cases than over the loopback adapter. Possible causes are the implementation of the loopback adapter or garbage collection activity that takes place while the test runs. Another anomaly arises for Ice for Java when sending variable-length structures. Note that, for this case, Ice for Java not only performs significantly better over the network than over the loopback adapter, but also performs significantly better than Ice for C#.

Platform Effects

The differences in the results for the loopback adapter and the network are a sterling example of the dangers of benchmarking: it is very difficult to create test scenarios that accurately reflect the behavior of real applications. It is hard or impossible to control all the variables, and the results are highly sensitive to a particular platform and the test data. For example, on a different operating system, these figures may change considerably due to the implementation of the loopback adapter.

To illustrate this point, I ran the throughput send tests with Ice for C++ and Ice for Java on a dual-core 2GHz Mac Mini in a dual-boot configuration with OS X and Windows Vista. Note that these tests in each case run on absolutely identical hardware. Here are the results for the loopback adapter:

ZeroC

	C++ byte seq.	C++ fixed seq.	Java byte seq.	Java fixed seq.
Windows Vista	1,800Mbit/s	900Mbit/s	1,250Mbit/s	620Mbit/s
OS X	2,850Mbit/s	740Mbit/s	2,400Mbit/s	145Mbit/s

Note the very significant performance differences, exceeding a factor of 4 for Ice for Java with fixed-length structures. We cannot even say that one operating system provides better performance than the other. For example, for byte sequences, OS X outperforms Vista, but for sequences containing fixed-length structures, we see the opposite.

The figures show that the benchmark measures not only the performance of Ice, but measures all sorts of things, including the performance of the loopback device driver and TCP/IP implementation, the quality of system library and virtual machine implementations, and how well the compiler optimizes code. For example, for the receive test, it matters whether the compiler applies NRVO (named return value optimization), which improves the byte sequence receive results by 20%.[1] When you consider that these test results were obtained by compiling and running identical source code for Ice, it becomes obvious that the platform has a very large impact on throughput.

At the very least, if you want to make a purchasing decision based on benchmarks, you must ensure that you are running the tests on the same hardware that will be used for your application, and that you are using the same OS, compiler, and system library versions for the benchmarks as for the application. Otherwise, all the testing effort may well be wasted.

CPU Consumption

Another important measure of middleware performance is CPU consumption. Simply put, the more CPU cycles are spent by the middleware in order to marshal and unmarshal data, the fewer CPU cycles are available for the application, and the longer it takes for a message to make its way through the middleware into the application code. This is particularly important for threaded servers because each CPU cycle spent in the middleware in one thread is not available to perform application work in a different thread.

Minimizing CPU consumption requires a number of implementation techniques, such as using efficient data structures, minimizing the number of thread context switches, or selectively inlining performance-critical code; overall, the goal is to reduce the processing path through the middleware during invocation and dispatch as much as possible.

CPU consumption does not make much sense in absolute terms. Rather, what matters is the percentage of time spent in the middleware for a particular workload. The larger the percentage, the fewer CPU cycles are available to the application. I will return to CPU consumption in the context of comparing Ice with WCF and the connection scalability tests.

WCF Performance

WCF is Microsoft's offering for distributed computing. Like Ice, it provides a remote procedure call mechanism with various "bells and whistles", such as asynchronous invocation and multi-threading.

---

[1] Ice 3.3.0 does not take advantage of this optimization with gcc. As of Ice 3.3.1, the optimization takes place.

ZeroC

WCF supports a number of different encodings for its messages. Two of these are of particular interest here: the binary encoding and the SOAP encoding. The binary encoding corresponds to the Ice protocol in that it is intended to efficiently encode structured data; comparison with this encoding allows us to judge the relative efficiency of WCF and Ice. The SOAP encoding exists to provide interoperability with web services; comparison with this encoding allows us to judge the cost imposed by this interoperability.

The two encodings are provided by the .NET `NetTcpBinding` and `WSHttpBinding` classes, respectively.

WCF Latency

We can implement the equivalent of our original latency test in WCF using the following contract:

```
// C#
[System.ServiceModel.ServiceContract]
interface Latency
{
    [System.ServiceModel.OperationContract]
    void o();
}
```

As for the tests using Ice, both client and server are single-threaded.

Over the loopback interface, using the binary encoding, this test runs 7,500 invocations per second. Using the SOAP encoding, the test achieves 2,500 invocations per second. Ice for .NET achieves the same 7,500 invocations as WCF with the binary encoding.

To explain these results, we need to look at the amount of data that is transmitted for each request, summarized below. (We obtained the sizes for WCF using Wireshark. With Ice, the sizes are available via its built-in protocol tracing.)

Latency (loopback)	Ice for .NET	WCF (binary enc.)	WCF (SOAP enc.)
Requests/second	7,500	7,500	2,500
Bytes/request	32 + 25 = 57	65 + 58 = 123	620 + 531 = 1,151

Adding the bytes for both request and reply, the Ice protocol uses 57 bytes, WCF with the binary encoder uses 123 bytes, and WCF with the SOAP encoder uses 1,151 bytes. This helps to explain why Ice and WCF with the binary encoder provide the same performance: even though WCF requires a little more than twice the amount of data, the data is small enough for the test to be dominated by the latency. In contrast, the SOAP encoder transmits roughly ten times the amount of data transmitted by the binary encoder and twenty times the amount of data required by the Ice protocol. However, that alone is not enough to explain the large difference. What plays a more important role is how much work is required to marshal and unmarshal the data.

For a binary encoding, little processing is required. In essence, during unmarshaling, the receiver reads the request header to determine how many bytes of payload to read and uses the operation name to identify the operation to dispatch to. Which object (or service) the request belongs to depends on the protocol. For Ice, the identity of the target object is sent over the wire; for WCF, the

ZeroC

identity of the target service is implicit in its endpoint. The exact details do not matter here—the important point is that very little processing is required to construct and decode the request.

In contrast, the SOAP requests are expensive to marshal and unmarshal: XML is complex to create and parse, so many more CPU cycles are used to encode and decode the on-the-wire representation.

Running the tests a second time over a gigabit network gives us the following results:

Latency (gigabit network)	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Requests/second	2,270	2,200	570

Ice is marginally faster than WCF with a binary encoding, while WCF with SOAP is slower by almost a factor of 4. These results are not surprising: the much larger on-the-wire encoding and the CPU-intensive marshaling and unmarshaling add up to significantly fewer invocations per second.

WCF Throughput

Throughput for Byte Sequences

The equivalent of the Ice throughput tests is the following WCF contract:

```
// C#
[ServiceContract]
interface Test
{
    [OperationContract]
    void sendByteSeq(byte[] seq);

    [OperationContract]
    byte[] recvByteSeq();
}
```

As before, we send and receive sequences of 500,000 bytes. Here are the test results for Ice and WCF with the binary and SOAP encoders over the loopback interface:

Byte sequence (loopback)	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Throughput (send)	630Mbit/s	515Mbit/s	144Mbit/s
Throughput (recv)	610Mbit/s	550Mbit/s	165Mbit/s

While Ice and WCF with the binary encode perform more or less on par, WCF with the SOAP encoder delivers roughly one quarter to one third of the throughput.

Running the test over a gigabit network returns the following results:

Byte sequence (gigabit network)	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Throughput (send)	520Mbit/s	425Mbit/s	140Mbit/s
Throughput (recv)	410Mbit/s	370Mbit/s	130Mbit/s

ZeroC

While Ice and WCF with the binary encoder slow down somewhat compared to the loopback interface, WCF with the SOAP encoder provides essentially the same lower throughput as over loopback. This is testimony to the large CPU overhead of the encoding.

Throughput for Structured Data

As I mentioned previously, the performance of byte sequence transfers is relevant only if you happen to transfer binary data. Far more often, applications exchange structured data. We can run the experiment for structured data by sending sequences of 50,000 elements of structures defined as follows:

```
// C#
[DataContract]
struct Fixed
{
    [DataMember] Int32 i;
    [DataMember] Int32 j;
    [DataMember] Double d;
};
```

The structure members are initialized to zero.

Measuring throughput over the loopback interface yields the following results:

Fixed-length sequence (Loopback)	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Throughput (send)	380Mbit/s	29Mbit/s	21Mbit/s
Throughput (recv)	300Mbit/s	28Mbit/s	21Mbit/s

The surprise here is two-fold: WCF provides only 10% of the throughput of Ice, and there is surprisingly little difference between the binary and the SOAP encoder.

Repeating the experiment over a gigabit network, we get:

Fixed-length sequence (network)	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Throughput (send)	300Mbit/s	31Mbit/s	23Mbit/s
Throughput (recv)	205Mbit/s	29Mbit/s	24Mbit/s

As you would expect, Ice slows down somewhat because data transfer over the network is slower than over the loopback interface. However, the WCF throughput is the same as over the loopback interface, regardless of whether we use the binary or the SOAP encoder.

This is surprising, considering that the SOAP encoding transmits more data than the binary encoding. Looking at the amount of data that is exchanged between client and server for a single invocation that sends a sequence of 50,000 structures, we find the following if the structures contain small double and integer values:

Small values	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Request size	800,057 bytes	601,538 bytes	2,750,781 bytes

ZeroC

For randomly initialized values, the picture is different:

Random values	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Request size	800,057 bytes	1,167,968 bytes	3,976,802 bytes

The change in size is due to WCF's variable-length encoding of integer and double values: small values require fewer digits and can be represented in fewer bits. In contrast, the Ice protocol uses a fixed-length encoding for integers and doubles, so the request size remains the same.

Despite the considerably larger size of the SOAP-encoded data, the performance of the binary and SOAP encoding are equally poor both over the loopback interface and over the network. This is true for sequence sizes of 25,000 and 100,000 structures as well: requests over the loopback interface and the network end up with the same throughput. This is surprising—intuitively, one would expect the loopback interface to be faster. The reason is the efficiency of the marshaling and unmarshaling code, which I discuss in the following section.

WCF CPU Consumption

We can use a profiler to measure the relative CPU consumption for marshaling and unmarshaling data. As a test case, I used the same sequence of fixed-length structures I used in the preceding section. By profiling client and server separately for the `send` operation, we can get a good picture of the relative cost of marshaling and unmarshaling: because the client sends the data to the server, but the server returns no data to the client, profiling the client measures the cost of marshaling, and profiling the server measures the cost of unmarshaling.

Here are the results for sending a sequence containing 50,000 structures with zero values from client to server. The CPU cycles used by Ice for .NET are normalized to 1 unit; CPU cycles used by WCF are expressed as a multiple of units. (The data was obtained with the ANTS Performance Profiler.)

Fixed-length sequence	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
CPU (send)	1	68	84
CPU (recv)	1	62	236

These figures help to explain the test results we saw in the preceding section. WCF is very much more CPU intensive than Ice for .NET. With the binary encoder, WCF spends more than sixty times the number of CPU cycles. In other words, the preceding test shows similar results for the binary and the SOAP encoder because it is dominated by the time it takes to process each message, which swamps any speed advantage provided by the loopback interface for data transfer.

Also note that the SOAP encoding uses around three times as many CPU cycles for unmarshaling as compared to marshaling. This is due to the complex nature of XML: the parsing effort when decoding the XML as it arrives over the wire is very large.

Java RMI Performance

Java provides RMI (Remote Method Invocation), which allows you to invoke methods on Java objects that are hosted in a remote server, much like Ice and WCF do.

ZeroC

RMI Latency

We can implement the equivalent of our original latency test for RMI using the following definition:

```Java
// Java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Latency extends Remote
{
    void o() throws RemoteException;
}
```

As for the tests using Ice, both client and server are single-threaded.

Over the loopback interface, this test runs 10,600 invocations per second. The table below compares Ice for Java and RMI, including the request and reply sizes:

Latency (loopback)	Ice for Java	RMI
Requests/second	8,000	10,600
Bytes/request	32 + 25 = 57	41 + 22 = 63

Repeating the test over the network, we obtain the following latency figures:

Latency (network)	Ice for Java	RMI
Requests/second	2,300	2,220
Bytes/request	32 + 25 = 57	41 + 22 = 63

While RMI is slightly faster than Ice for Java over the loopback interface, the latencies are essentially identical over the network. This no surprise, seeing that this test measures the speed of the network; the middleware (unless it is very inefficient) is irrelevant.

RMI Throughput

Throughput for Byte Sequences

The equivalent of the Ice throughput test for byte sequences is the following Java interface:

```Java
// Java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Throughput extends Remote
{
    int ByteSeqSize = 500000;

    void sendByteSeq(byte[] seq) throws RemoteException;
    byte[] recvByteSeq() throws RemoteException;
}
```

ZeroC

As before, we send and receive sequences of 500,000 bytes. Here are the tests results for Ice and RMI over the loopback interface:

Byte sequence (loopback)	Ice for Java	RMI
Throughput (send)	825Mbit/s	825Mbit/s
Throughput (recv)	730Mbit/s	800Mbit/s

Running the test over a gigabit network returns the following results:

Byte sequence (network)	Ice for Java	RMI
Throughput (send)	660Mbit/s	283Mbit/s
Throughput (recv)	560Mbit/s	280Mbit/s

As you can see, Ice provides considerably higher throughput for binary data.

Throughput for Structured Data

The following definition allows us to run the throughput experiment for structures with fixed-length members:

```java
// Java
class Fixed implements java.io.Serializable
{
    int i;
    int j;
    double d;
};
```

The structure members are initialized to zero.

Measuring throughput over the loopback interface yields the following results:

Fixed-length sequence (loopback)	Ice for Java	RMI
Throughput (send)	138Mbit/s	48Mbit/s
Throughput (recv)	118Mbit/s	52Mbit/s

Ice outperforms RMI by more than a factor of two here.

Repeating the experiment over a gigabit network, we get:

Fixed-length sequence (network)	Ice for Java	RMI
Throughput (send)	252Mbit/s	66Mbit/s
Throughput (recv)	148Mbit/s	66Mbit/s

ZeroC

As we saw earlier, Ice performs better over the network than over the loopback interface. For RMI, the network is only marginally faster than loopback.

Here are the message sizes for this case:

Fixed-length sequence	Ice for Java	RMI
Request size	800,057 bytes	1,100,146 bytes

The message size difference is not enough to explain the large difference in the throughput results. Looking at the code with a profiler, we find that Ice for Java and RMI consume a comparable number of CPU cycles, so CPU consumption is not the reason for the big difference in performance. However, the profiler shows that RMI spends a large amount of time in a `wait` method. This suggests a quality-of-implementation issue, possibly related to non-optimal send buffer size.

Memory Consumption

With multi-gigabyte main memory commonplace these days, memory consumption is less of an issue than in the past, and the middleware memory footprint is often not an issue. However, for mobile devices, memory consumption is still important, as it is for heavily-loaded servers: higher memory consumption translates directly to increased hardware cost because, for efficient operation, the working set must fit into the available memory.

It is interesting to compare the working set sizes of the different platforms. Here is the server-side working set during the byte sequence throughput test, with Ice for C++ included. (To obtain these figures, I used the procedure outlined in this FAQ.)

Byte sequence	Ice for C++	Ice for Java	Ice for .NET	WCF (binary)	WCF (SOAP)	RMI
Working set	1.2MB	6.5MB	7MB	7.5MB	21MB	7.5MB

Note that Ice for Java, Ice for .NET, WCF, and RMI require roughly the same amount of working memory, except for the WCF server with the SOAP encoder. (The complex XML parsing during unmarshaling explains the higher memory requirements.) It is also interesting to see that Ice for C++ gets by with a fraction of the memory. This largely reflects the fact that managed execution environments, such as .NET and Java, extract a high price in memory consumption.

Impact of Encodings

The choice of the on-the-wire encoding can have a large impact on overall performance. The two main conflicting forces are the size of data, and the amount of effort required to serialize and deserialize to and from the encoding. Ideally, we would like to have an encoding that is as compact as possible, in order to minimize the bandwidth we consume. At the same time, we would like to encode and decode the data such that it takes as few CPU cycles as possible. These two goals are, unfortunately, in conflict with each other.

One of the most efficient encodings in terms of CPU cycles is to encode the data such that it matches its in-memory representation. For example, we can encode a 32-bit integer by simply writing the

ZeroC

data as a 4-byte blob in little-endian representation. That way encoding and decoding are very efficient because they only require a simple copy of the integer to and from the transport buffer. (On a little-endian machine, that is; on a big-endian machine, we need to byte-swap between the big-endian in-memory representation and the little-endian on-the-wire representation.)

However, encoding data in this way also has drawbacks. For example, for small integers that can be represented in a single byte, we marshal three bytes of zeros for each integer. By changing to a variable-length encoding, we could reduce bandwidth. For example, we could use the convention that the top bit of each byte is set if another byte is to follow, and is zero if this byte represents the final byte of an integer. This allows us to encode values in the range 0-127 in a single byte, values in the range 0-16,383 in two bytes, and so on.

Similar ideas can be applied to strings. For example, we could come up with an encoding that compresses strings by using a run-length encoding or similar technique. The problem is that, the more the encoding is biased toward conserving bandwidth, the more expensive it becomes to marshal and unmarshal so, at some point, we reach a trade-off point where the additional CPU cycles required to compress the data cost more than the time it would take to transmit the data in a bulkier encoding.

In addition, the code for more complex encodings is larger and requires more supporting data structures than the code for a simple encoding. In turn, this means that memory consumption and working set size increase with more complex encodings. This is particularly noticeable when the working set size becomes larger than the primary or secondary CPU cache: whenever the working set no longer fits into a cache and spills over from the primary to the secondary cache (or from the secondary cache to main memory), we suffer a large performance hit (often by a factor of ten or more).

The point at which the trade-off becomes negative depends on the bandwidth of the link between client and server, the relative CPU speeds, and the nature of the data. In other words, no matter what trade-off we choose, it will be wrong at least some of the time.

The design philosophy for the Ice encoding is to optimize performance at the expense of bandwidth, that is, to use as few CPU cycles as possible during marshaling and unmarshaling. For example, Ice marshals integral types and floating-point types in their native little-endian 2's complement and IEEE representation even though small values could be encoded in fewer bytes. The Ice encoding does use a few simple compression tricks (such as marshaling sequence sizes in a single byte for sequences shorter than 255 elements) and, for classes and exceptions, it uses a simple indexing scheme to avoid repeatedly sending the same type identifiers over the wire. However, these compression tricks are implemented with very little CPU overhead and, therefore, effectively reduce bandwidth consumption without unduly increasing CPU cycles.

Some encodings use other tricks to improve efficiency. For example, CORBA's IIOP (Internet Inter-ORB Protocol) uses a "receiver makes it right" scheme that allows the sender to marshal data in the sender's native byte order, instead of choosing a fixed endian-ness for encoded data. The idea is that, if sender and receiver have the same endian-ness, neither of them needs to byte swap. Similarly, IIOP uses a complex padding scheme in an attempt to improve unmarshaling efficiency at the cost of using some extra bandwidth.

In contrast, Ice avoids such peep-hole optimizations. As experiments show, the cost of byte swapping is less than 1% even for pathological cases that require every byte of a message to be swapped with its neighbour, so the extra complexity outweighs the minimal saving in CPU consumption. Neither is the Ice encoding is bloated. For example, IIOP uses considerably more bandwidth for the same data due to its padding rules. Unfortunately, the bandwidth is not well spent because the padding actually increases the number of CPU cycles required during unmarshaling.

In a nutshell, the Ice encoding is both compact on the wire and efficient to marshal and unmarshal, and compression strategies are limited to very simple cases that provide worthwhile savings in bandwidth with a minimal expenditure of CPU cycles.

Compression

To allow Ice to perform efficiently over slow links, it provides a protocol compression feature that compresses requests with a payload of more than 100 bytes using bzip2. Depending on the data in the request, this typically yields a compression factor of around 2 but, for data with more redundancy (such as sparse arrays), can result in much larger compression. Bzip2 is very good at removing redundancy and results in requests that are typically within two or three percent of their theoretical minimum size. This approach is preferable to picking an encoding scheme that not only adds CPU cycles but also is ineffective for many payloads; for example, a scheme to encode small integers in fewer bytes is ineffective if the data happens to contain mostly large integers.

Note that, over fast networks, compression actually slows things down: it is faster to throw the data on the wire uncompressed than to compress and uncompress it. (Bzip2 is an expensive algorithm.) Exactly at what point compression provides a gain depends on the speed of the network, the bit patterns in the data, and the mix of operations that clients invoke. You need to experiment to see whether compression improves performance for your application. (See Mark Spruiell's article in Issue 7 of *Connections*.)

How Small is "Small Enough"?

In general, as long as the encoding is "reasonably" small, the details of how the data is encoded do not matter much. How do we decide what is "reasonable" though? As long as messages are small, just about any encoding will do. This is because, for small messages, the overall performance is dominated by latency: if we send a message at all, we might as well send around a kilobyte of data with it because that takes no longer than sending no data.

However, a larger encoding does mean that we more quickly reach the cross-over point where bandwidth starts to affect end-to-end performance. If, with one encoding, the data still falls below the size where it "hides" inside the inevitable latency but, with another encoding, the same data causes a delay due to bandwidth limitations, we start to notice a performance degradation. Any difference becomes more important over slow links because a bulkier encoding has a proportionately larger impact on overall performance. (End-to-end performance is more likely to be dominated by request size for slow links.)

The WCF SOAP encoding is particularly wasteful in bandwidth. For example, we can re-run our structure test, but with a slightly changed structure definition:

ZeroC

```csharp
// C#
[DataContract]
struct Fixed
{
    [DataMember] Int32 int1Member;
    [DataMember] Int32 int2Member;
    [DataMember] Double doubleMember;
};
```

This is the same definition as the previous one, but with different names for the data members. Running the test for structures with data members initialized to zero, we get the same request size for Ice, almost the same request size for WCF with the binary encoder, and a larger request size for WCF with the SOAP encoder. The results are summarized below. For comparison, the table also shows the earlier results, for the structure with single-letter member names:

	Ice for .NET	WCF (binary encoding)	WCF (SOAP encoding)
Request size (short names)	800,057 bytes	601,538 bytes	2,750,781 bytes
Request size (long names)	800,057 bytes	600,488 bytes	3,404,862 bytes

As you can see, simply renaming the structure members increases the size of the SOAP request by almost 25%. Considering that the payload of the request (in terms of in-memory size) is 800,000 bytes, the encoding overhead for this simple case already is 325%. (The encoding transmits 3.25 times as many bytes in overhead than in payload and 69% percent of the bandwidth is wasted.) Of course, this becomes worse if the structure has a longer name or a large number of data members that are small, such as integers and short strings: the overhead in this case can easily exceed 1,000%, depending on the names of the data members.

Does Size Really Matter?

You may come across the argument that "encoding size does not matter because bandwidth is cheap". For applications that do not need to scale to large numbers of clients and exchange only a small amount of data, the encoding indeed does not matter. However, as the number of clients and the size of the data increases, the encoding overhead becomes very noticeable.

To illustrate how serious this can be, consider an application with clients that receive large amounts of data from a server. There are many applications that fall into this category, such as online games, distributed data analysis tools, and document retrieval applications. Suppose we have a server connected to the Internet via a 50Mbit/s link and that clients request 50kB of data each. Because of framing overhead, 1Mbit/s works out to be 1kB/s of payload, so the 50Mbit/s link allows the server to service 100 clients concurrently each second. Let's be conservative and assume that, for this example, the SOAP encoding is ten times as large as the same payload encoded with Ice. That means that instead of the 100 clients per second we can service with Ice, we can deal with 10 clients per second using SOAP.

To service 100 clients with the SOAP encoding then requires us to increase the speed of the link to 500Mbit/s, increasing the cost of the link into the server by a factor of ten. (In practice, the actual cost is likely to be larger because additional bandwidth becomes more expensive as the link speed increases.) Of course, all this assumes that we can actually handle 500Mbit/s in our server. Given the

very large CPU overhead of SOAP, this is unlikely however. To provision the same service over SOAP, we will have to not only provide a faster link but several times the hardware.

How Fast is "Fast Enough"?

How do we decide whether things are fast enough? To answer this question, we need to look at the overhead added by the middleware for a particular application. Once the time spent in the middleware accounts for only a small fraction of the overall time, say 5-10%, the middleware is no longer the problem. For example, consider a scenario where, for a particular mix of operations, the middleware accounts for 10% of the total processing time, with the remaining 90% of the time spent in application code. Even if we were to double the speed of the middleware (which is unlikely), we would improve performance by only 5% overall. (While 5% are certainly worthwhile, they are unlikely to make or break a project.) On the other hand, if we leave middleware performance the same, but find a way to improve the application performance by only 10% (which is much more realistic), we end up with an overall performance improvement of 9% (nearly twice as much). This example illustrates that it makes sense to improve those parts of the code that account for the bulk of the processing time because that provides the "biggest bang for the buck."

However, there is another consideration we need to take into account: well-designed middleware such as Ice already performs very well. As the preceding tests show, the performance limitations do not arise from the middleware, but from the underlying technology. Assuming an efficient implementation, latency is simply determined by delays in the transport layer and the light speed limit. Similarly, assuming an efficient middleware implementation, bandwidth is determined by the bandwidth of the transport layer, not by processing overhead of the middleware.

All this means is that, once the middleware is "fast enough," further speed improvements can be made only by understanding the limitations of the network and by designing an application to work well within these limitations.

How Slow is "Too Slow"?

Conversely, if the middleware does not make efficient use of the transport layer, the middleware becomes the limiting factor, and no amount of improvement in the application can raise performance beyond the limit established by the middleware.

Apart from the "bandwidth is cheap" fallacy, you may also come across an argument to say that "latency does not matter—after all, messages cannot propagate faster than the speed of light". To see whether this argument makes sense, consider the 570 requests per second we obtained for SOAP messages over a gigabit network. That is around 1.75ms per request; light travels a little over 500km (or a little over 300mi) in that time. As the argument goes, if client and server are separated by more than that distance, latency becomes irrelevant because the light speed limit starts to dominate. This argument is flawed for several reasons:

- Many clients and servers are separated by much shorter distances. For local area or metropolitan networks, the large latency imposed by SOAP remains very noticeable indeed.
- The latency due to the middleware and the latency due to the light speed limit are additive. Light might take 1.75ms seconds to travel from client to server, but that does not mean that

21   **ZeroC**

the latency of the middleware has disappeared. Instead, the middleware adds its own latency so, for the preceding example, we are now looking at 3.5ms of latency instead. The latency of the middleware becomes insignificant only once it drops to, say, around 10% of the total latency. But now we are looking at distances of several thousand kilometers, that is, trans- or intercontinental communication.

- Once communication takes place over large distances, with inevitably large latency, the application must be designed to deal with the latency in other ways, such as by minimizing the number of messages, transferring more data with each message, and by avoiding synchronous round-trip designs, so client and server continue to do useful work while they wait for a message to complete or arrive. In other words, the light speed argument is a red herring because it makes sense only for incorrectly designed applications.

Another point to keep in mind is that a large part of the latency of SOAP is due to CPU overhead. A very common design for distributed applications is to have clients connect to a front-end server that acts as a firewall and connection concentrator. The front-end server usually performs quite simple processing, such as decoding each message, checking security credentials, and forwarding each message to one of a number of back-end servers for further processing. If it takes, say, 80 times the number of CPU cycles to decode each message, the front-end server is not going to scale with the number of clients; we are likely to need tens of front-end servers for the same workload (even assuming that we can handle the increased bandwidth requirements of SOAP).

Overall, applications that exchange SOAP messages are limited to light workloads or, alternatively, can afford to use lots of expensive hardware. Even for moderate loads, SOAP almost certainly falls into the "too slow" category.

Scalability

Much of scalability is a quality-of-implementation issue: if you have efficient middleware, things will automatically scale better than with inefficient middleware, simply because adding more CPUs or bandwidth leaves a larger proportion of the additional resources for the application.

Ice enables scalability in a number of ways.

Scalable Algorithms and Data Structures

Ice uses efficient algorithms and data structures throughout its implementation. This means that there are no artificial and unnecessary bottlenecks that only show up once you push a particular parameter, such as the number of servants, the number of operations per interface, or the number of network connections. In practice, this means that the implementation uses fast-access data structures, such as hash tables, trees, or other custom-designed data structures to avoid costly operations such as linear searches. Similarly, the Ice run time transparently caches performance-critical state to avoid needlessly resending network messages or performing otherwise expensive operations.

No In-Built Arbitrary Limits

Ice does not arbitrarily limit resource consumption. All data structures grow dynamically as demands require, so you will not find arbitrary limits such as "you cannot have more than 1,024 properties" or similar. In general, Ice scales as well as the operating system it runs on: any limits to scalability are

ZeroC

set by the operating system. For example, Ice works well with however many threads, file descriptors, network connections, or memory the operating system will provide.

API Features

Scalability limits are often created by failure of the middleware to anticipate the need for it. Ice provides a number of APIs that exist specifically to facilitate scalability. For example, servant locators exist specifically to allow you to take precise control over the memory versus performance trade-off in a server. How much memory your application consumes as the number of servants increases is entirely in your own hands: if you can afford to spend the memory, you can use it; if you cannot, Ice provides a way that allows you to continue to scale without having to add more memory (albeit at a slower pace).

Threading

A major way to increase scalability is to write multi-threaded clients and servers. Ice was designed for threaded environments from day one. All APIs are thread (and exception) safe, and you never need to protect any of the Ice run time data structures from concurrent access; you only need to worry about the thread safety of your own application code.

Ice scales almost linearly as you add more CPUs: provided a server does not run out of workload, it will perform roughly four times as fast with four CPUs than with a single CPU. Of course, the operating system adds some scheduling and context switching overhead, and a small proportion of the additional CPU cycles is lost to that; however, as rule of thumb, more CPUs directly translates into proportionately better performance. The internal threading and locking algorithms used by Ice were designed specifically to keep overhead to a minimum, so more threads actually benefit the application.

Federation

Ice makes it easy to scale an application by federating it. You can add more servers to an existing application in order to scale to higher workloads without having to tear most of the source code apart. Moreover, using IceGrid, you can implement sophisticated load-balancing schemes to ensure that the workload is distributed effectively over the available hardware. IceGrid itself is replicated in order to remain scalable: IceGrid can act as a location service for tens of thousands of clients without any performance degradation. Similarly, IceStorm, ZeroC's event service, uses federation to scale to hundreds of thousands of clients; increasing scalability is as simple as adding new hardware and editing a few configuration items.

Connection Scalability

An important measure of middleware scalability is connection scalability: how many clients can a server support concurrently before service for some clients degrades unacceptably? One way to address this question is to consider a given workload and assess whether that workload can be sustained as the number of clients increases. For example, assume that, on a single-CPU machine, the server can sustain 1,000 requests per second from 1,000 different clients. If we now replace the server with one that has four CPUs at the same speed (and sufficient memory), ideal middleware should allow the server to support 4,000 requests per second from 4,000 different clients.

ZeroC

Another important point is whether the server can sustain the workload as there are more connected clients, many of which are idle. At first glance, this looks like a strange scenario. After all, if there are lots of clients that don't do anything, they can simply disconnect. For a stateless service, this is true: once a client has received a reply to a request, the client (or the server) can close the connection. (Ice provides active connection management (ACM) to do this automatically.)

However, there are many applications for which the stateless server assumption does not hold. A typical example is a chat application such as MSN Messenger. The server has a large number of connected clients, most of which are idle most of the time. Yet, the idle clients cannot disconnect because, to receive status updates, they must remain connected. (The server cannot initiate connections to clients because the clients are typically behind a firewall that disallows incoming connections, so the server must send status updates on the connections that are opened by the clients.)

Returning to our scalability scenario, suppose we can sustain 4,000 requests per second from 4,000 different clients. The scalability question now becomes "Is it possible to support 4,000 requests per second from 4,000 different clients while there are a total of 20,000, or 50,000 or 100,000 connected clients?" A related question is "What is the largest request load the server can sustain while a very large number of clients are connected to the server?"

Aside from chat applications, the answers to these questions are also important for applications such as connection concentrators and routers (such as Glacier2), bit torrent trackers, and stock trading servers that match bids to offers, among many others.

To test connection scalability, we can use a very simple server that hosts a single object with a single method that does nothing. Clients place load on the server by starting a number of threads; each thread opens a separate connection to the server and invokes the operation once every four seconds. (This simulates the workload that, for example, chat clients place on a chat server—each thread simulates a separate client with a separate connection.) Each client thread records when it completes an invocation or encounters a timeout error and periodically reports how many invocations completed and timed out. (Invocations that take longer than one second are deemed to have timed out, even if they eventually complete successfully.) In addition, clients report the minimum and maximum number of invocations that individual threads completed in the preceding period, and the average number of invocations for the period across all threads. This allows us to monitor whether some threads see service degradation due to errors or because their invocations take longer than normal to complete. In addition, it permits detection of threads that "get stuck" in an invocation for long periods without timing out.

For the tests, we ran the server on an Intel Core 2 Extreme QX6700 quad-core 2.66 GHz machine with 4GB of memory. The operating systems were Red Hat Enterprise Linux 5.2 64-bit and Windows Server 2008 64-bit. For the Java tests, we used JDK 1.6.0_11 64-bit. To load the server, we ran the clients on a number of machines, splitting the number of connections among machines such that individual clients did not encounter bottlenecks.

Connection Scalability with Linux

To run the tests, we changed `/etc/security/limits.conf` to permit 200,000 file descriptors per process because each incoming connection consumes a file descriptor in the server. The CPU and memory consumption figures are as reported by `top`.

To summarize, here are the results for Linux:

Linux scalability	Ice for C++	Ice for Java	RMI
Requests/sec	20,000	20,000	7,500
# connections	80,000	80,000	30,000
CPU usage	15%	25%	30%

The request rate shown in the table is the number of connections divided by 4. The actual rate is a fraction lower because requests are not instantaneous.

Note that, at the workload shown in the table, clients start to get timeouts (requests not completing within one second) with Ice for Java and RMI. RMI fails catastrophically at 31,000 connections due to lack of memory, while Ice for Java continues to scale (albeit with more clients getting slow response to requests as the server's VM does garbage collection passes). Ice for Java had a working set size of 810MB, with total memory of 1.4GB. In contrast, RMI's working set was 3GB, with total memory of 37GB.

The practical limit in Ice for C++ is around 80,000 connections. We increased the number of connections to 90,000, but the speed of accepting connections beyond 80,000 caused unacceptable delays in the clients.

Connection Scalability with Windows

To run the tests, we compiled Ice with an `FD_SETSIZE` of 60,000 (up from the default of 1,024). This was necessary because each connection consumes a file descriptor; the server monitors activity on these descriptors with `select`, which cannot monitor more than `FD_SETSIZE` file descriptors. We obtained CPU usage figures with Windows Server 2008 Resource Monitor.

Here are the results for Windows:

Windows scalability	Ice for C++	Ice for Java	Ice for .NET	RMI	WCF (binary)	WCF (SOAP)
Requests/sec	675	8,750	7,250	6,875	4,875	8,000
# connections	2,700	35,000	29,000	27,500	19,500	32,000
CPU usage	25%	80%	33%	15%	90%	40%

Naturally, we were disappointed by the poor results with Ice for C++. The root cause is that the server-side run time uses `select` to identify incoming connections that have data ready to read. However, `select` scales very poorly as the number of connections increases. Ice 3.4 will address this problem by using I/O completion ports instead, which do not suffer from this problem. With this change, Ice for C++ will scale as well on Windows as it does on Linux. As a work-around until then, you can increase scalability by deploying more servers or creating several adapters in each server,

ZeroC

providing each adapter with a separate thread pool. At 2,700 connections, the working set of the server is 50MB, with total memory at 2.5GB.

Ice for Java and Ice for .NET scale to comparable levels before clients encounter timeouts. The high CPU usage of Ice for Java (80% for 35,000 connections) compared to Linux (25% for 80,000 connections) can be explained by the JDK's use of `select` on Windows; it spawns a separate selector thread for every 1,023 connections. Conversely, on Linux, the JDK uses `epoll`, which is much more CPU efficient. Ice for .NET also consumes less CPU, due to the use of async I/O to monitor connections.

The RMI figures look good at first glance. However, the server at this point consumes 37GB of virtual memory and requires around 3GB of working set memory and, beyond this point, fails catastrophically due to lack of memory. The likely cause of this is RMI's thread-per-connection model which does not scale well for many connections. In contrast, Ice requires only a modest amount of memory, on par with the Linux figures.

WCF with the binary encoder is limited largely by its CPU consumption. It appears that the version we tested has some quality-of-implementation issue. However, this is of little consolation when it comes to the amount of hardware you need to dedicate to the application. Outside an artificial benchmark situation, in addition to the middleware, the application also consumes CPU cycles. Considering that WCF consumes 90% CPU at 19,500 connections, this means that hardly any CPU remains for the application, so the real scalability limit is likely to be much lower; the only way to increase the limit is to dedicate more hardware to the task.

WCF with the SOAP encoder scaled unexpectedly well. (The figures show the level at which we stopped testing, not the level at which clients encountered timeouts. As it turns out, the WCF clients consume a lot of CPU and we could not get more than 32,000 client threads running on the available hardware.) However, the results are artificial in the sense that the invocations do not transfer any data. It is unlikely that this level of scalability would remain for a representative application load. For example, the high bandwidth requirements of SOAP may well max out the network adapter before any issues arise due to the large number of connections. Similarly, the high CPU overhead of marshaling and unmarshaling SOAP requests may well be the dominant limit.

In general, Windows is not as good a choice for connection scalability as Linux. On equivalent hardware, Ice on Linux scales to more than double the number of connections with lower CPU overhead for both C++ and Java.

For applications using Ice for C++ on Linux, Ice imposes essentially no limitations. In effect, the run time is agnostic to the number of connected clients. It is more likely that bandwidth or CPU cycles end up being the bottleneck for the overall workload. The same is true for Ice for Java, but at a lower overall limit. Scalability is constrained mainly by the cost of garbage collection passes due to the high level of activity in the server; the number of connections has no impact on this. (Double the request rate per connection with half the number of connections incurs the same amount of garbage.)

The excessive memory consumption of RMI is of real concern. At 80,000 connections, Ice for Java requires around 18.5kB of virtual memory and 10.5kB of working set memory per connection. In contrast, at 30,000 connections, RMI requires 1.25MB of virtual memory and 105kB of working set

26

memory per connection. This is larger by a factor of 70 for virtual memory and a factor of 10 for working set memory. These figures directly translate into increased hardware costs as an application needs to scale. (Keep in mind that the test server does not do anything with the data; for a real application, the work performed by the server would increase memory consumption even more.)

Overall, Ice (with the exception of C++ on Windows) scales better in terms of memory consumption, CPU overhead, and connection scalability than either RMI or WCF.

Designing for Performance

There are a number of techniques you can use to get the most out of your Ice clients and servers. The techniques in this section are only a sampling of things you may want to consider. By necessity, the advice given here is fairly general because the effectiveness of various performance improvements varies greatly with the application's workload, the mix of operations it uses, the network, and the hardware clients and servers run on.

The main maxim when considering performance is to optimize the part of the application that is most critical overall. In other words, use a profiler to identify in which operations the application spends most of its time, and then consider how you can improve the performance of these operations. The idea is to focus on the hotspots where you are likely to get the biggest payback for your efforts.

Coarse-Grained Operations

For small messages, performance is completely dominated by latency. For example, for byte sequences, the time per request remains constant for sequences of up to 1,450 bytes; thereafter, the time per request rises linearly with the amount of data, as illustrated by the following graph:
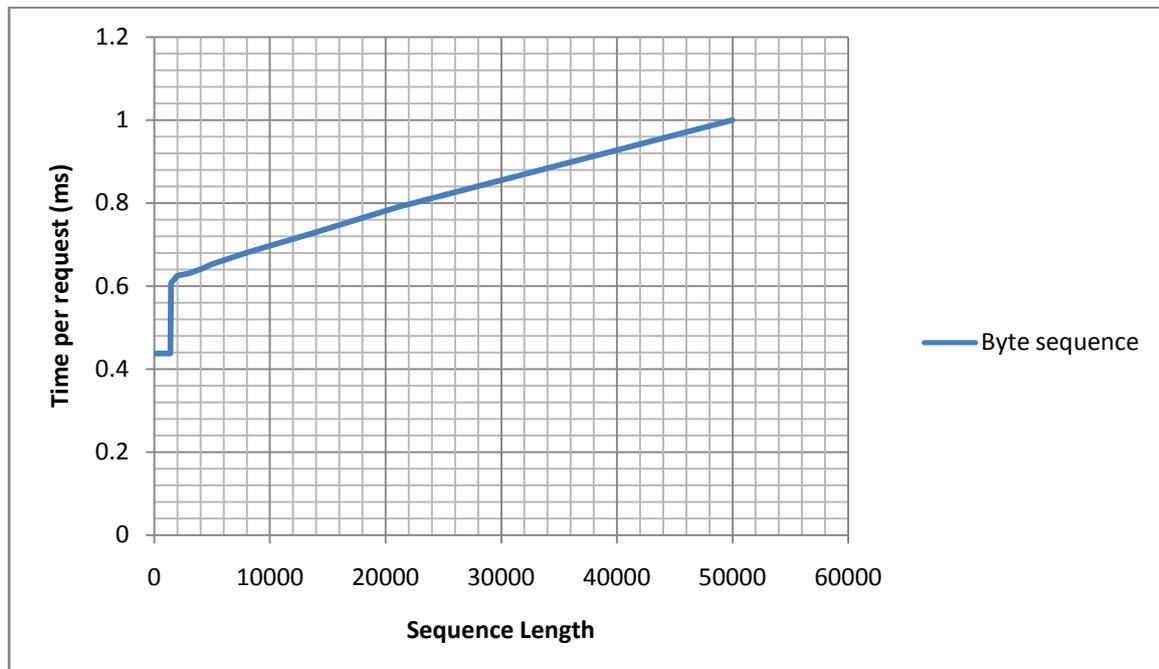
Up to 1,450 bytes, we get a constant request time of around 0.435ms. Once the sequence length crosses that threshold, we see a jump to around 0.6ms. This is no coincidence because, at that point,

ZeroC

we cross the Ethernet PDU size limit. Thereafter, time per request rises linearly with the sequence length.

This means that, if you send any message at all, you might as well send a decent amount of data with it because that will take no longer than not sending any data. It follows that you should avoid defining Slice interfaces that are too fine-grained: fewer operations that pass more data are preferable to more operations that pass little data. This is particularly important for objects that encapsulate many small items of state. For example:

```
// Slice
interface Address {
    string getFirstName();
    void setFirstName(string name);
    string getLastName();
    void setLastName(string name);
    int getHousNumber();
    void setHouseNumber(int number);
    // ...
};
```

Such interfaces can be inefficient because, compared to the cost of sending an RPC, each operation does very little work. As long as there are few clients that call these operations only occasionally, the fine-grained interactions do not matter. However, as the number of clients rises and the load on the server increases, overall throughput will be affected. An alternative way to structure such interfaces is as follows:

```
// Slice
struct Details {
    string firstName;
    string lastName;
    int houseNumber;
    // ...
};

interface Address {
    Details getDetails();
    void setDetails(Details d);

    // Above operations can either replace or
    // supplement fine-grained operations.
    string getFirstName();
    void setFirstName(string name);
    string getLastName();
    void setLastName(string name);
    int getHouseNumber();
    void setHouseNumber(int number);
    // ...
};
```

ZeroC

The `getDetails` and `setDetails` operations pass more state per invocation without incurring additional latency, so they make better use of the network and CPU resources in client and server.

Consider Oneway and Batched Oneway Operations

Any operation that does not have a return value, does not have out-parameters, and does not raise user exceptions can be invoked as a oneway operation. Such operations typically set a value, such as the `setFirstName` operation of the preceding example. If you invoke an operation as a oneway operation, the client-side Ice run time sends a request to the server, but the server does not send a corresponding reply. In addition, the invocation in the client completes as soon as the request has been passed to the local transport; the client regains its thread of control much sooner because it need not wait for the reply from the server.

Oneway invocations are not as reliable as twoway invocation because, if the connection drops unexpectedly, the client is not informed of the failure. However, oneway invocations are still useful, particularly in LAN environments, where unexpected connection closure usually indicates a serious failure anyway.

You can also use batched oneway invocations. When invoking via a batched oneway proxy, the client sends operation invocations as usual. However, instead of each invocation being sent immediately to the server as a separate message, the invocations are queued in the client-side run time and sent as a single protocol message when the client flushes the queue. This leads to substantial speed improvements. For example, Ice for Java requires around 45 seconds for 100,000 empty twoway invocations, whereas invoking the same 100,000 operations as batched oneway operations requires only 0.3 seconds. Similarly, using oneway invocations for the byte sequence test increases throughput from 660Mbit/s to 950Mbit/s.

Consider Asynchronous Method Invocation (AMI)

A twoway invocation blocks the calling thread until the server returns a reply that indicates success or failure. This means that, if you make many invocations that move only a small amount of data you lose a processing thread for the time it takes for the invocation to make the complete round trip from client to server and back again.

By using AMI, the client regains the thread of control as soon as the invocation has been sent (or, if it cannot be sent immediately, has been queued), allowing the client to use that thread to perform other useful work in the mean time. AMI can be particularly effective for servers that make callbacks on objects provided by clients: the server can continue to use the calling thread to perform other useful work while the AMI invocation is in progress.

AMI can also be useful for large messages. For example, IcePatch2 achieves extremely high throughput (within a small fraction of the theoretical maximum throughput) by transferring large amounts of data in chunks, with each chunk being sent asynchronously. This technique takes advantage of I/O interleaving because it allows the calling thread to prepare and send another request while the data for the previous request is still in transit.

Use Threaded Servers

Using threaded servers is one of the most effective ways to improve overall performance: as long as a server is single-threaded, it can only perform work on behalf of one client at a time and so is

unlikely to scale well as the number of clients increases. Moreover, especially for operations that perform little work, a single-threaded server will spend much of its time waiting for I/O to complete.

The main question is how many threads a server should have. The answer depends on the number of available CPUs, and on the number of requests that are going to be in progress concurrently for a typical workload. For example, if you rarely have more than five clients making requests simultaneously, there is no point in giving eight threads to the server because chances are that the additional threads won't improve performance, and may even decrease performance, due to increased memory footprint and context-switching overhead.

Assuming that there are enough clients to keep the server supplied with work, a good rule of thumb is to have as many server threads as there are CPUs, plus a few extra threads as headroom. The additional threads can take up the slack while other threads perform I/O. This also means that, even for a server that runs on a single CPU, adding a second thread can improve performance due to I/O interleaving. (One thread can use the CPU while the other thread is waiting for I/O to complete.)

Aside from the threading requirements for Ice, you also need to take the needs of your application into account. Depending on the threading architecture, your application may well benefit from additional threads beyond those required to achieve the best possible performance for RPC. The only way to find the optimum number of threads is to monitor your server under a typical workload and vary the number of threads. The idea is to push CPU utilization a high as possible. If you add a thread to the server and find that overall CPU utilization increases, the extra thread does useful work. On the other hand, if you find that CPU utilization stays the same or drops, the extra thread is doing more harm than good.

Consider Using C++

If you know that you must squeeze every last ounce of performance, consider using Ice for C++ instead of Ice for Java or Ice for .NET. Due to optimizations that are impossible with safe languages such as Java and C#, Ice for C++ provides considerably better throughput. For example, you can use the zero-copy API for sequences to improve performance. (Refer to the C++ mapping for [sequences](#) in the Ice manual for more information.)

In addition, working set sizes with Ice for C++ are considerably smaller, which is important if you must create servers that service a large number of concurrent clients. However, keep in mind that even though Ice for C++ provides throughput that is better by a factor of three or four, your application must actually spend a significant fraction of its time marshaling and unmarshaling data. If that is not the case, Ice for C++ is unlikely to provide much of a performance benefit. In other words, you should not let your choice of development language be dictated by performance requirements unless that is truly warranted.

So, How to Choose?

I hope that the foregoing has led you to the conclusion that middleware performance and scalability are only two of many evaluation criteria. As the experiments show, performance is highly sensitive to factors that have nothing to do with the middleware, such as the quality of device drivers and system libraries. Moreover, performance varies widely with the amount and type of data that is exchanged, and synthetic benchmarks can provide only an overall rough indication of the

ZeroC

performance of a real application. This is not to say that middleware performance is not important; it *is* important, but only if you actually have an application that depends on good middleware performance.

WCF with the SOAP encoder performs much worse than the alternatives. However, for infrequent and coarse-grained interactions, that performance may be entirely acceptable, and you may have other overriding reasons for wanting to use SOAP. (Just don't count on being able to scale such a system up, unless you are prepared to spend a lot on hardware.)

If your application must scale to large numbers of clients, keep in mind that network bandwidth, memory, CPU, and connection limits cannot be evaluated in isolation. Depending on the application workload, any one of these factors can become the one that caps scalability. Once the limit is reached, the only (expensive) option is to add more hardware to handle the load.

Currently, Ice, RMI, and WCF are seen as major contenders in the middleware space. (CORBA is rapidly dwindling into insignificance and generally no longer considered as being in the race.) Even though Ice generally provides better performance and scalability than WCF and RMI, your choice of middleware should likely be influenced by additional factors. In order to make a choice, there are three things you can consider that should help to narrow things down quickly:

- Platform and language support
- Services beyond RPC
- Ease of use

First and foremost are platform and language support. Java RMI is OS-agnostic, but not language-agnostic: if you want to use Java RMI, you must use Java at both ends. If that is acceptable, RMI may well be a good choice. WCF, on the other hand, is language-agnostic, but not OS-agnostic: if you want to use WCF, you must use Windows at both ends. Again, if that is acceptable, WCF may well be a good choice. If you anticipate that you will have to use a mix of languages, such as (unmanaged) C++ for the servers for performance reasons, and Java for the clients, you can use neither RMI (because it won't handle C++) nor WCF (because it won't handle Java). Similarly, if you need to run your system on more than one operating system, WCF is not an option. And, if you happen to have genuine high performance requirements, neither is likely to be a good choice.

In contrast, Ice is language-agnostic, OS-agnostic, and provides excellent performance and scalability. This makes Ice an ideal choice for heterogeneous environments that combine a variety of operating systems and programming languages. This is particularly important when you consider how your system might evolve over time. Even if it is perfectly reasonable today to accept that "everything is Java" or "everything runs on Windows", that may not be true a few years later, as your system has to adapt to changing requirements.

You may also want to consider that today's WCF may not be the final word. (Looking at the history of middleware from Microsoft, we find—in rapid succession—DDE, OLE, OLE Automation, COM, ActiveX, DCOM, COM+, .NET Remoting, and, finally, WCF. Who knows what might be the flavor of the day two or three years from now…)

The second important consideration is that middleware is much more than just a remote procedure call mechanism. Real distributed systems need not only basic RPC, but a whole host of services

ZeroC

without which it becomes much more difficult to build applications. For example, IceStorm provides publish–subscribe event distribution. The service scales to hundreds of thousands of clients and is fault tolerant so there is no single point of failure. Another example is IceGrid, which (among many other things), enables replication and fault tolerance, transparent migration of services, location transparency, and central administration and deployment. Yet another example is Freeze, which makes it very easy to add persistence to your applications.

Unless your application is very simple, chances are that you will use at least one of these services (typically IceGrid), and will likely use all three. The degree of performance, scalability, and reliability provided by these services is difficult to achieve, requiring considerable experience and development effort. If you decide to build them yourself, chances are that they will not only be expensive to build, but also be less performant, less scalable, and less reliable. In other words, you should take care to consider the entire package when choosing middleware, instead of focusing on only one aspect, such as performance, and you should consider only those parts of the package that you actually need. (Feature tick lists are a waste of time if they list features that you will not use.)

An often overlooked consideration is ease of use. Ease of use is difficult to quantify but can have major impact on development cost. The number of new things developers have to learn, the number of development tools they need, the elegance (or otherwise) of APIs, and the quality of the documentation play a large role not only for time it takes to develop an application, but also its defect count and the concomitant effort required to reach release quality. We believe that Ice is superior to WCF and RMI in this regard. To get started with Ice, all you need is a text editor and a compiler. And, despite its very extensive feature set, Ice APIs are simple and safe to use: Ice respects the principle of "you should not pay for what you don't use" and leaves developers with more capacity to focus on solving application problems (instead of middleware problems).

Finally, if you do happen to have a genuine requirement for high performance, you should give Ice serious thought. Its performance is outstanding and allows you to get the best out of your network and hardware, regardless of what operating system or language you decide to use.

ZeroC