

Ice Middleware Tests

October 26, 2006

Contents

1	Interfaces	2
1.1	Byte Array	2
1.2	Float Array	4
1.3	Structure	4
2	Client-Server	5
2.1	Client	5
2.2	Server	6
2.3	Configuration	7
2.4	Running	7
2.5	Performance	7
2.5.1	Byte Array Interface	8
2.5.2	Float Array Interface	8
2.5.3	Structure Interface	8
3	Publish-Subscribe	9
3.1	Publisher	9
3.2	Subscriber	10
3.3	Configuration	11
3.4	Running	11
3.5	Performance	11
3.5.1	Byte Array Interface	12
3.5.2	Float Array Interface	12
3.5.3	Structure Interface	12
4	Discussion	12

This report documents Ice (Internet Communications Engine) useability and performance tests done in October 2006 by John Storrs. Ice is an object-oriented middleware package from [ZeroC](http://www.zeroc.com) (<http://www.zeroc.com>), available under both commercial and open source (GPL) licences. It is a successor to [CORBA](http://www.omg.org/gettingstarted/corbafaq.htm) (<http://www.omg.org/gettingstarted/corbafaq.htm>).

The tests were designed and carried out in cooperation with Andrei Maslennikov of the [CASPUR](#)

Inter-University Computing Consortium (<http://www.caspur.it>) in Rome. They were motivated by discussions about middleware for the ITER CODAC and the ITM Code Platform, but also have possible relevance to JET and MAST. Issues raised included performance in client-server and publish-subscribe configurations, programmer ease of use, configurability, and whether a generic middleware API is feasible, independent of particular implementations. The tests provide performance figures for various message sizes, and the test code, summarised in the following sections, casts light on the other issues.

The test hardware, at CASPUR, consists of 5 high performance Linux workstations (bigbeast, storm08, storm09, storm10, storm11). The bigbeast system is dual core Intel bi-processor (4-cpu). It has a 3GHz cpu clock speed, 8GB RAM, and 4 1-Gigabit Ethernet interfaces bonded to a single IP address. The storm systems are 2-cpu, with a slightly lower clock speed and a single 1-Gigabit Ethernet interface connected to one of the 4 bigbeast ports. Maximum possible network throughput to bigbeast is thus 4 times the maximum possible throughput of 1-Gigabit ethernet, seen in tests to be about 117MB/sec.

This document is available in html and pdf versions.

1 Interfaces

Ice defines the Slice language (Specification Language for Ice) for data interface definitions, similar to CORBA IDL and part of WSDL. A Slice interface definition is similar to a C++ class declaration, and can contain type, structure and function declarations etc. It is compiled into equivalent target language class declarations and definitions by an Ice tool, for example slice2cpp for C++. The application programmer then writes a derived class which implements virtual functions to realise desired server behaviour.

This section summarises Slice interface definitions used in the client-server tests, and the derived classes which implement server behaviour. Provision was made for both oneway (with no return value) and twoway (with a return value) function call semantics, to assess the relative costs. Interfaces for the publisher-subscriber tests were similar, except that only oneway calls were used, and 4 interface functions with identical behaviour were provided, to distinguish publishers at the subscriber.

1.1 Byte Array

This is a Slice definition for a variable length array of bytes (byte sequence) and functions which use it to pass data between client and server (or publisher and subscriber). Byte sequence is the simplest Slice type, with the lowest transmission cost because no marshaling, unmarshaling, or byte swapping is required - that has to be handled by the application if required. Unless the data is really byte-oriented (eg image format), it is better to use a float array or structure interface, leaving the dirty work to Ice.

```
module CODAC {
    sequence<byte> ByteSeq;

    interface Messages
    {
        void publishDataMessageOneWay(ByteSeq message);
        int  publishDataMessageTwoWay(ByteSeq message);
        .....
    };
};
```

From this, slice2cpp generates Messages.h and Messages.cpp, containing the declaration and definition of the Messages class among other things. In general it's not necessary to look at the generated files.

With knowledge of the Ice mappings from Slice to C++, the application programmer writes a C++ header file for a MessagesI class derived from the Messages class. The 'I' suffix is just a convention standing for 'Interface'.

```
#include <Messages.h>

using namespace CODAC;

class MessagesI : public Messages
{
public:
    virtual void publishDataMessageOneWay(const ByteSeq&,
                                           const Ice::Current&);
    virtual int publishDataMessageTwoWay(const ByteSeq&,
                                          const Ice::Current&);
    .....
};
```

Then the application programmer implements the required server behaviour in the MessagesI class definition. Member function code is executed on the server when remote calls are made by a client, and can do anything with the message. In the tests we just validate messages with a known pattern to detect data loss or corruption.

```
#include <MessagesI.h>
#include <Ice/Ice.h>

using namespace std;
using namespace CODAC;

void
MessagesI::publishDataMessageOneWay(const ByteSeq& message,
                                     const Ice::Current&)
{
    // validate data
    .....

    // no value returned to client
}

int
MessagesI::publishDataMessageTwoWay(const ByteSeq& message,
                                     const Ice::Current&)
{
    // validate data
    .....

    return EXIT_SUCCESS; // returned to client
}
```

1.2 Float Array

The Slice definition for a variable length array of floats (float sequence), and the associated MessagesI class files, are similar to those shown for byte array.

1.3 Structure

This is a Slice definition for a C-style structure and functions which use it to pass data between client and server (or publisher and subscriber). This interface might be used by a data acquisition system (the client) managing hundreds or thousands of signals. The structure stores signal samples over some time range, with all samples in the range for one signal followed by all samples for the next and so on. Data acquisition is assumed here to be synchronous for all signals, so only one set of time values is stored. This organisation will be convenient for servers which process the data, eg for archiving or display. Note that the definition does not prescribe sequence sizes, just structure. Sizes can change on a call-by-call basis to suit the circumstances of the client.

```
module CODAC {
    sequence<float> FloatSeq;

    struct SignalData {
        string name;
        FloatSeq samples;
    };

    sequence<SignalData> SignalSeq;

    struct DataMessage {
        string name;
        int id;
        int signalCount;
        int sampleCount;
        FloatSeq times;
        SignalSeq signals;
    };

    interface Messages
    {
        void publishDataMessageOneWay(DataMessage message);
        int publishDataMessageTwoWay(DataMessage message);
        .....
    };
};
```

The derived interface class header is essentially the same as the byte array version.

```
#include <Messages.h>

using namespace CODAC;

class MessagesI : public Messages
{
```

```

public:
    virtual void publishDataMessageOneWay(const DataMessage&,
                                          const Ice::Current&);
    virtual int publishDataMessageTwoWay(const DataMessage&,
                                         const Ice::Current&);
    .....
};

```

As is the derived class implementation.

```

#include <Ice/Ice.h>
#include <MessagesI.h>

using namespace std;

void
MessagesI::publishDataMessageOneWay(const DataMessage& message,
                                    const Ice::Current&)
{
    // validate data
    .....

    // no value returned to client
}

int
MessagesI::publishDataMessageTwoWay(const DataMessage& message,
                                    const Ice::Current&)
{
    // validate data
    .....

    return EXIT_SUCCESS; // returned to client
}

```

2 Client-Server

Client-server is the basic Ice mode of use. Given an interface definition as in the previous section, very little application-specific code is required, as can be seen from the following code summaries.

2.1 Client

```

#include <math.h>
#include <Ice/Ice.h>
#include <MessagesI.h>

using namespace std;
using namespace CODAC;

class Client : public Ice::Application

```

```

{
public:
    virtual int run(int, char*[]);
};

int
Client::run(int argc, char* argv[])
{
    // declarations (application)
    .....

    // initialize client (Ice calls)
    .....

    // create message buffer (application)
    .....

    // send messages (application)
    for (int m = 0; m < messageCount; m++) {
        // set message data
        .....

        // call MessagesI member function on server
        //oneway->publishDataMessageOneWay(message);
        result = twoway->publishDataMessageTwoWay(message);

        // record timing
        .....
    }

    // report results (application)
    .....

    return EXIT_SUCCESS;
}

int
main(int argc, char* argv[])
{
    Client app;
    return app.main(argc, argv, "config.client");
}

```

2.2 Server

```

#include <MessagesI.h>
#include <Ice/Application.h>

using namespace std;

```

```

class Server : public Ice::Application
{
public:
    virtual int run(int, char*[]);
};

int
Server::run(int argc, char* argv[])
{
    // activate server and enter message processing loop (Ice calls)
    // call MessagesI member function when a message arrives
    .....

    // exit on subscriber shutdown (Ice calls)
    return EXIT_SUCCESS;
}

int
main(int argc, char* argv[])
{
    Server app;
    return app.main(argc, argv, "config.server");
}

```

2.3 Configuration

These extracts from the configuration files show how the communication transport and endpoints are defined.

Client:

```
Messages.Proxy=messages:tcp -h 192.168.1.41 -p 10000
```

Server:

```
Messages.Endpoints=tcp 192.168.1.41 -p 10000
```

2.4 Running

The tests were run on bigbeast and storm08 by simply doing:

```
bigbeast: ./server
storm08: ./client
```

Different message sizes were configured by changing constants in MessagesI.h, and oneway or twoway transmission selected by editing Client.cpp, followed by recompilation.

2.5 Performance

With oneway transmission the message is sent and no return value expected; with twoway, a value is returned by the server to the client. Results varied from run to run; representative results are given

here. Little difference was seen between results when the Linux OS on bigbeast was compiled as 32-bit or as 64-bit (only 2 comparisons were done, and taken as indicative). With a message size of 10kB or 100kB, throughput for oneway calls approached or achieved the maximum possible over a single Gigabit Ethernet connection. Twoway calls with a message size of 10kB are comparatively expensive. Test duration was several minutes, with the client sending messages in a continuous loop. Time measurements were made using the client's high resolution cpu timer. 1MB = 1.0e6 bytes.

2.5.1 Byte Array Interface

message type	message size (MB)	cycle time av (ms)	throughput av (MB/s)
oneway	0.01	0.085	117
	0.1	0.85	117
	1.0	11.8	85
	10.0	126	80
twoway	0.01	0.37	27
	0.1	1.57	64
	1.0	14.2	70
	10.0	145	69

2.5.2 Float Array Interface

message type	message size (MB)	cycle time av (ms)	throughput av (MB/s)
oneway	0.01	0.088	114
	0.1	0.91	110
	1.0	10.5	95
	10.0	120	83
twoway	0.01	0.35	28
	0.1	1.45	69
	1.0	13.3	75
	10.0	141	71

2.5.3 Structure Interface

message type	message size (MB)	cycle time av (ms)	throughput av (MB/s)
oneway	0.01	0.086	116

message type	message size (MB)	cycle time av (ms)	throughput av (MB/s)
	0.1	0.85	117
	1.0	10.6	94
	10.0	130	77
twoway	0.01	0.38	27
	0.1	1.8	55
	1.0	15.1	66
	10.0	163	61

3 Publish-Subscribe

Ice provides a publish-subscribe system called IceStorm which uses the IceBox framework. This is all built on basic client-server transactions. Again, very little application-specific code is required, as can be seen from the following code summaries.

3.1 Publisher

```
#include <Ice/Application.h>
#include <IceStorm/IceStorm.h>
#include <math.h>
#include <unistd.h>
#include <MessagesI.h>

using namespace std;
using namespace CODAC;

class Publisher : public Ice::Application
{
public:
    virtual int run(int, char*[]);
};

int
Publisher::run(int argc, char* argv[])
{
    // declarations (application)
    .....

    // initialize publisher and attach to topic (Ice calls)
    .....

    // initialize message buffer (application)
    .....

    // send messages (application)
```

```

for (int m = 0; m < messageCount; m++) {
    // set message data
    .....

    // transmit message using MessagesI member functions
    //errorCount = twoway->publishDataMessageTwoWay(message);
    oneway->publishDataMessageOneWay(message);
    .....

    // record timing
    .....
}

// report results (application)
.....

return EXIT_SUCCESS;
}

int
main(int argc, char* argv[])
{
    Publisher app;
    return app.main(argc, argv, "config.pub");
}

```

3.2 Subscriber

```

#include <map>
#include <Ice/Application.h>
#include <IceStorm/IceStorm.h>
#include <IceUtil/UUID.h>
#include <MessagesI.h>

using namespace std;
using namespace CODAC;

class Subscriber : public Ice::Application
{
public:
    virtual int run(int, char*[]);
};

int
Subscriber::run(int argc, char* argv[])
{
    // initialize subscriber (Ice calls)
    .....

    // subscribe to topics (Ice calls)

```

```

.....

// set the requested quality of service (Ice calls)
.....

// activate subscriber and enter message processing loop (Ice calls)
// call MessagesI member function when a message arrives
.....

// exit on subscriber shutdown (Ice calls)
return EXIT_SUCCESS;
}

int
main(int argc, char* argv[])
{
    Subscriber app;
    return app.main(argc, argv, "config.sub");
}

```

3.3 Configuration

These extracts from the configuration files show how communication transport and endpoints are defined.

Topic Manager:

```
IceStorm.TopicManager.Endpoints=tcp -h 192.168.1.41 -p 10000
```

Publisher and Subscriber:

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -h 192.168.1.41 -p 10000
```

3.4 Running

The tests were run on bigbeast and storm08-storm11 by doing:

```

bigbeast: icebox --Ice.Config=config.icebox
bigbeast: ./subscriber
storm08: ./publisher 1
storm09: ./publisher 2
storm10: ./publisher 3
storm11: ./publisher 4

```

Different message sizes were configured by changing constants in MessagesI.h, followed by recompilation.

3.5 Performance

Tests were done using oneway calls only for maximum throughput. Note that average throughput here combines the throughput of the 4 publishers to the bonded Gigabit Ethernet interface on bigbeast. The maximum possible is $4 \cdot 117 = 468 \text{MB/s}$. The best throughput shown is here 282MB/s ,

only 60% of that maximum. But the tests recorded were done with a single threaded icebox and subscriber, and only 2 of the 4 cpus were in use as shown by 'top'. When the icebox and subscriber were configured to use 2 threads each, with byte array interface and 100kB message size the throughput was 452MB/s. Test duration was several minutes, with publishers sending messages in a continuous loop. Time measurements were made using the publisher's high resolution cpu timer. 1MB = 1.0e6 bytes.

3.5.1 Byte Array Interface

message size (MB)	average cycle time (ms)	average throughput (MB/s)
0.01	0.22	178
0.1	1.4	282
1.0	22.0	180
10.0	247	160

3.5.2 Float Array Interface

message size (MB)	average cycle time (ms)	average throughput (MB/s)
0.01	0.24	165
0.1	1.6	254
1.0	22.5	179
10.0	225	178

3.5.3 Structure Interface

message size (MB)	average cycle time (ms)	average throughput (MB/s)
0.01	0.25	162
0.1	1.88	213
1.0	23.0	174
10.0	246	162

4 Discussion

Ice is a well engineered and well supported middleware system with good performance. It took only a few days from nothing to write the basic client-server test code, helped by the demos provided with the package. The Slice interface language makes it easy to define and understand complex typed interfaces. The automatically generated mappings from Slice to application programming language are direct and intuitive. Ice runs on Unix and Windows, with language mappings for C++, Java,

C#, VisualBasic, Python and PHP. The C++ mapping enables use of Ice with C, Fortran, etc. Ice has many useful capabilities, including secure transmission, data compression and persistent data storage which have not been explored in these tests. The test results show that Ice throughput can be good to excellent on a high speed LAN, and suggest that it would be limited by the lower network performance on a WAN.

One of the issues which stimulated these tests was the feasibility of a generic middleware API, independent of particular implementations. Because a significant part of middleware systems, the interface specification mechanism, has to be independent of application languages, it stands outside the normal scope of an API. Also, a generic middleware API would either be restrictive or very complex, depending on whether it is the intersection or union of capabilities, in both cases unsatisfactory. The issue concerns the ease with which one middleware system could be replaced by another, and is really about localisation. If middleware interface specifications and function calls are in limited, clearly identified locations, and not scattered through application code, unplugging one system and plugging in another should be straightforward. Localisation, not a generic API, seems to be right way to go here.

In conclusion, Ice is an excellent package which deserves our closer attention.