Getting Started with the Internet Communications Engine

David Vriezen

April 7, 2014

Contents

1 Introduction

The **Internet Communications Engine**, often simply referred to as **ICE**, is a platform enabling distributed computing. It's both free and open source, and simple to get started with. Distributed computing is useful to delegate a large task to several machines, or even to a single machine better equipped to handle a task. For example, perhaps you might want to assign a task with lots of floating point operations to a machine with specialized hardware designed to execute floating point operations more efficiently, but do the rest of the work on a machine that otherwise better fits your needs.

The **ICE** platform supports C++, Java, .NET, Python, Objective-C, and Javascript, and has partial support for Ruby and PHP. In this tutorial, you will learn how to begin using **ICE**. The examples in this document will be presented in Java, but the examples can be converted to other supported languages fairly trivially.

2 About Ice

ICE claims to use a **client-server model**, but that is somewhat ill-defined. A single machine need not exclusively be a server or a client in ICE. A machine A can connect as a client to a server machine B, and machine B can connect as a client to another server machine C. In fact, any machine can be connected as either a client or a server in a one-to-one, one-to-many, or many-to-many relationship.

2.1 Proxies

A client must have a **proxy** to connect with an ICE object. A proxy represents the ICE object to the client; when the client invokes a method on a proxy, it contacts the ICE object on the server application and invokes the method there.

3 Setting Up ICE

These tutorials assume that your system runs Windows, and is configured with an up-to-date version of Java, as well as Oracle's Eclipse IDE. To install ICE, download the Windows installer from *www.zeroc.com/download.html* and run it. Once completed, be sure to add the ICE compilers to your Windows PATH. At the time of writing, this path is

```
C:\Program Files\ZeroC\Ice-3.5.1\bin
```

4 Slices

Ice uses a **slice** to describe objects and methods. When compiled, a slice generates descriptions for the objects and methods into the supported languages.

This allows for consistency when writing code in different languages, since the code is all generated from the same slice. The slice compiler will translate the slice definition into whatever implementation is correct for a given language.

4.1 Writing a Slice

A slice describes objects and methods in a language that's best described as an amalgam of C++ and Java. A slice is required to have the case-sensitive **.ice** file extension. Below, we will discuss the slice syntax and provide an example of a basic slice.

4.1.1 Modules

A slice file must contain at least one module. A module can contain any legal slice constructs. Use modules to group together related classes and interfaces, to make definitions easy to find, and to avoid naming conflicts in large projects. A module maps to a scoping mechanism when compiled to a language. For example, in C++ a module compiles to a namespace, whereas in Java, it will compile to a package.

An example of module definitions can be seen below.

```
//a module definition
module Foo{
    //a nested module definition
    module Bar{
        ...
    };
    module Baz{
        ...
    };
};
//a module can be reopened later,
//in the same file, or another file.
module Bar{
    ...
};
```

4.1.2 Basic Types, Enumerations, Sequences, and Dictionaries

The Slice syntax supports several basic types, which you should already be familiar with. These types are **bool**, **byte**, **short**, **int**, **long**, **float**, **double**, and **string**. It is important to note that the string type does not provide support for the null string concept.

Enumerations are similar to their usage in C++ and other languages. By default the left-most item in an enumerator evaluates to 0, and then each other value evaluates to one more than the preceding item.

Custom values may be assigned to items as well, or a mix of default and custom may be used.

```
enum Months {January=1, February, May=5, June, July};
```

In the above example, January's value is explicitly assigned the value 1, and so then February is implicitly assigned a value one higher. Then, May is explicitly assigned the value 5, and June and July are 6 and 7, respectively.

Lists of items can be modeled using the **sequence** keyword. Sequences model a variety of collections, including vectors, queues, sets, and trees. How the sequence is used is determined by the application. To declare a sequence of ints, simply write:

```
sequence<int> NumberList;
```

A **dictionary** is a data type which maps a key to a value. Looking up the key will return the corresponding value. While any type is allowed as a value, a key is limited to a string, an enum, any integral numbers, or a struct or sequence containing those types. A dictionary is defined similarly to a sequence:

```
dictionary<integer,string> Months;
```

4.1.3 Structs, Interfaces, and Classes

Objects can be defined in a slice in three different ways.

A **structure** is a user-defined type which contains one or more member variables with arbitrary typing. As an example, we could represent a a line as a series of Cartesian points in a struct like this:

```
//a struct to hold the x and y values of a point
struct Point{
    int x;
    int y;
};
//a sequence of points, used to define a line
sequence<Point> Points;
//a struct containing information about a line
struct Line{
    Points p;
    string label;
};
```

An interface is basically the same as a Java interface. It is only allowed to contain methods, and the methods defined are essentially virtual members, to be defined later in your implementation code. An example interface definition is provided below.

```
interface Foo{
    int Bar(int x);
    //the idempotent method declares that this method can
    //be invoked successively without the result changing
    idempotent void Baz(Point p);
};
```

A class is a structure which can contain both data members and methods. It's syntax is similar to that of structures and interfaces. As expected, a class can implement an interface, or inherit one other class. Examples of bother are provided below:

```
interface Vehicle{
    void go();
    ...
};
class Car implements Vehicle{
    string year;
    string vin;
    ...
};
class Honda extends Car{
    string model;
    ...
};
```

4.1.4 Exceptions

If you'd like to define Exceptions for your program, it's pretty straightforward. Exceptions can include data members, like structs. A simple example of an exception is illustrated below:

```
exception Error{
    string reason;
    int timestamp;
    ...
};
interface Foo{
    void Bar() throws Error;
    ...
};
```

4.2 Compiling a Slice

To compile a slice for Java, simply open up the Windows command prompt and enter

```
slice2java file.ice
```

It is worth mentioning three helpful options here, **debug** and **output-dir**. If the compiler fails to run as expected, the –debug option can help you to find errors in your slice file. The output-dir option can be used to specify the location where the generated files are placed. This can be useful, depending which platforms you are compiling for. For example, for a simple slice file, where compiling for .NET and Python generated only two or three files, compiling the same file for Java generated over 20 individual files. It is advisable to place these files in their own directory so that they do not clutter up your workspace. Conventionally, this folder is titled **generated**. Lastly, you can also compile more than one slice at a time by listing multiple files. To call these options, use

```
slice2java --output-dir generated --debug file.ice file2.ice file3.ice
```

Several more options are available, and a full explanation of all options can be found by calling

```
slice2java --help
```

For the subsequent examples, it will be assumed that you have compiled the slice distributed with this tutorial to generate the appropriate Java files.

5 Starting with Java

Create a new project in Eclipse called SorterExample. Add the file Ice.jar to your build path. This file can be found in the *lib* folder in the Ice directory in Program Files. Also, include all of the files generated from the slice in a package called **SorterExample**. All the generated files, as well as the SorterExample.ice and the completed Java files which will be mentioned shortly are provided in a zipped archive along with this document.

5.1 Writing the Server

Because there is not yet a concrete implementation of our Sorter object, we must begin by writing one. Create a new file called SorterI.java in the default package. This implementation is called a **servant**. Conventionally, the servant implementation given the same name as the class it's implementing, with an "I" suffix, so the Sorter servant is called SorterI.

SorterI must implement the methods we defined for Sorter, *sort()*, and *reverseSort()*. Eclipse should automatically provide method stubs for you, but if it does not, you can find the method interfaces defined in _SorterOperations.java. As an example, here is an implementation of *sort()*:

```
public class SorterI extends SorterExample._SorterDisp{
    public int[] sort(int[] arr, Current __current) {
        Arrays.sort(arr);
        return arr;
```

```
    }
}
```

You may notice several interesting things about this code. Firstly, *SorterI* extends *_SorterDisp*. This is the skeleton file generated for the Sorter Class. Secondly, you'll notice that the slice compiler has mapped our sequence into an int[], so that Java will recognize it.
Lastly, you'll notice an extra parameter added to the method signature: *__current*. The *Current* object provides information about the currently executing request to the operation, and can be ignored in many cases.

Implementing the server is a little more involved than implementing the sort methods. Create a file called Server.java in the default package. First we must initialize the connection, and then create an ObjectAdapter for the client to connect to. Since this connection could fail, depending on the network connection, this attempt should be enclosed in a try-catch block.

```
public class Server {
    public static void main(String[] args) {
        //This object manages the communications between this application
        //and other clients or servers
        Ice.communicator ic = null;
        try{
        //try to initialize the connect
        ic = Ice.Util.initialize(args);
        //this define the name of the adapter and where to connect
        Ice.ObjectAdapter adapter = ic.createObjectAdapterWithEndpoints(
            "SorterAdapter",
            "default -h localhost -p 10000")
        //this is the object the server provides a proxy to
        Ice.Object object = new SorterI();
        //this adds the object to the adapter
        //and maps it to an identity
        adapter.add(object, ic.stringToIdentity("SorterAdapter");)
        //activate the adapter for connections
        //then wait until the server gets a shutdown signal.
        adapter.activate();
        ic.waitForShutdown();
        } catch (Exception e){
            ...
        }
}
```

5.2 Writing the Client

Create a new file called *Client.java* in the default package. To start with, our client must connect to the server and obtain a proxy object for our sorter.

The code to create a connection and initialize a proxy object looks very similar to that of the server, like this:

```
public class Client{
    public static void main(String[] args){
        //as above
        Ice.Communicator ic = null;
        try{
            //as above
            ic = Ice.Util.initialize(args);
            //this tells the client which adapter to connect to
            //and gets that Object Proxy
            Ice.ObjectPrx base =
                ic.stringToProxy("SorterAdapter:default -h localhost -p 10000);
            //checks and casts the object into the Sorter Proxy type
            SorterExample.SorterPrx sorter =
                SorterExample.SorterPrxHelper.checkedCast(base);
        } catch (Exception e){
            ...
        }
    }
}
```

After we've successfully gotten a proxy, we can invoke methods the proxy, to be executed remotely.

```
int[] toSort = ...;
//the call to SorterPrx.sort invokes the method remotely on the server.
sorter.sort(toSort);
//this call also invokes remotely, but will throw an exception
//because it was never implemented
try{
    sorter.reverseSort(toSort);
} catch (Error e){
    ...
}
```

That's all it takes to get and use a proxy!

6 Notes on Synchronization

By default, ICE uses **dynamically-sized thread pools** to execute tasks. While ICE's libraries themselves are thread-safe, your code may not be. It is up to you to make sure that multiple threads safely access your data structures, using whatever practices are appropriate for your language of choice and your application's needs.

If you need to change the default thread pools size of either the client or the server, it is possible to do so by changing the value **Ice.ThreadPool.Server.Size**, and **Ice.ThreadPool.Server.MaxSize**. It is also possible to change the StackSize and ThreadIdleTime variables similarly.