

Tutorial de uso de ZeroC-ICE en JAVA

Basado en el proyecto: Query_dist

[Query_dist usando el patrón de diseño Publisher and Suscribe / ZeroC-ICE](#)

El propósito de este tutorial es proporcionar un enfoque más claro para el uso de la herramienta ICE, de modo que sea más comprensible y vaya un paso más allá del clásico "Hello World" que se encuentra en la documentación de ZeroC-ICE. Este documento no pretende reemplazar la documentación original, sino ser un complemento para lograr un entendimiento más claro y accesible del middleware ICE.

Dado que nos basaremos en el proyecto mencionado anteriormente, necesitaremos configurar un entorno para el mismo.

Requerimientos:

- Java 11
- Gradle 8.6
- ZeroC-ICE 3.7

Compilación:

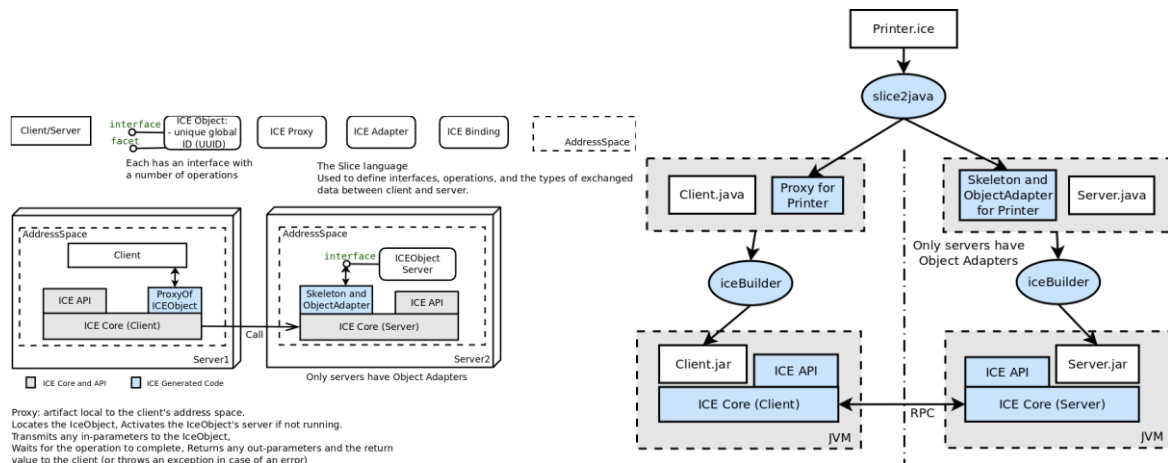
Muchos de los errores que pueden surgir al seguir este tutorial y/o el tutorial oficial de la página se deben a que no se está utilizando las versiones adecuadas. Por lo tanto, es importante verificar las versiones en su entorno y asegurarse de que coincidan con los requisitos mencionados anteriormente.

En el tutorial "Hello World" de la página se explica bien, pero surge un problema al compilar, ya que el tutorial no tiene en cuenta la versión con la que estamos trabajando. El único inconveniente es que el Class-Path en el archivo build.gradle está un poco desactualizado. Por lo tanto, modificaremos el archivo la sentencia jar/manifest/attributes:

```
"Class-Path": configurations.runtimeClasspath.resolve().collect {  
it.toURI() }.join(' ').
```

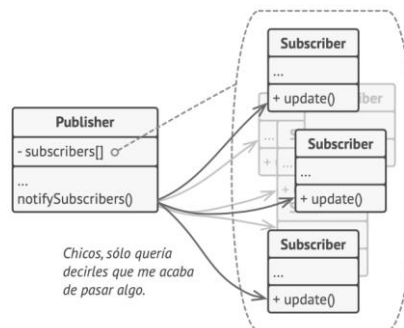
Con esto, el proyecto debería compilar correctamente.

La siguiente imagen muestra la estructura de ICE y del proyecto "Hello World":



Patrón Observer:

Para comprender mejor el proyecto, primero expliquemos cómo funciona el patrón Observer, que es un buen ejemplo dado que al implementarlo podemos entender mejor el funcionamiento de ICE.



No entraremos en detalles sobre el patrón, pero lo importante es entender que se trata de un patrón de diseño de comportamiento que permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. Generalmente, el Publisher es el servidor y el Subscriber es el cliente. Sin embargo, como la aplicación devuelve información al servidor, usaremos un Callback haciendo que el Cliente funcione como Publisher para el Servidor.

Retomando:

Una vez que todo esté configurado, compilado y hayamos comprendido el "Hello World" de la página oficial, podremos seguir con este tutorial.

Si observamos el proyecto en git, veremos que el archivo build.gradle tiene algunas diferencias. Esto se debe a que el proyecto implementa dependencias y un plugin que cambia el método de compilación. Esto se hace con el fin de empaquetar las dependencias del proyecto en el archivo .jar para que se pueda ejecutar sin problemas en otros sistemas. A esto se le llama ShadowJar o JarFat.

Por lo tanto, este proyecto se compila con: **gradlew shadowjar**.

Repacemos la construcción del proyecto:

1. **build.gradle:** Aquí se definen las dependencias, la estructura de compilación y la ubicación del archivo .ice.

```
slice {  
    java {  
        files = [file("../Query_dist.ice")]  
    }  
}
```

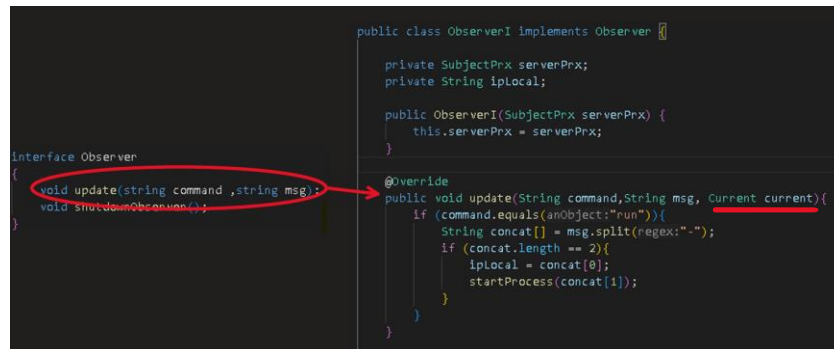
2. **Settings.gradle:** Deben ir los proyectos root y los deseados para el funcionamiento del aplicativo. En este caso, son Client y Server.

```
rootProject.name = 'query_dist'  
include 'client'  
include 'server'
```

3. **.ice (Query_dist.ice):** Todo servicio que sea expuesto tiene que estar en el archivo .ice.

```
module FunctionsPoint  
{  
    interface Observer  
    {  
        void update(string command ,string msg);  
        void shutdownObserver();  
    }  
    interface Subject  
    {  
        void addObserver(Observer* o);  
        void removeObserver(Observer* o);  
        void getTask();  
        void addPartialResult(string fileNameResult);  
    }  
}
```

Estos servicios deben estar implementados en su respectivo proyecto y es importante destacar que, al implementar todos los métodos, estos requieren adicionalmente un parámetro Current por defecto.



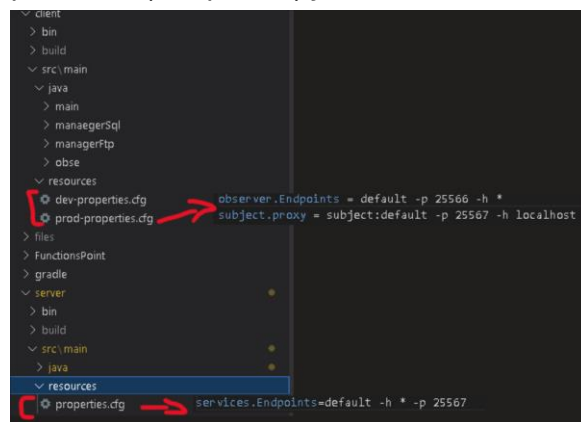
```
interface Observer {
    void update(string command, string msg);
    void shutdownObserver();
}

public class ObserverI implements Observer {
    private SubjectPrx serverPrx;
    private String ipLocal;

    public ObserverI(SubjectPrx serverPrx) {
        this.serverPrx = serverPrx;
    }

    @Override
    public void update(String command, String msg, Current current) {
        if (command.equals(anObject:"run")) {
            String concat[] = msg.split(regex:"-");
            if (concat.length == 2) {
                if (concat[0].length() == 2) {
                    ipLocal = concat[0];
                    startProcess(concat[1]);
                }
            }
        }
    }
}
```

4. **.cfg (Opcional):** En el lado de cada proyecto, se tiene un resources/properties.cfg que se debe configurar para establecer las interfaces provistas (endpoints) y las conexiones con dichas interfaces (proxy).



```
observer.Endpoints = default -p 25566 -h *
subject.proxy = subject:default -p 25567 -h localhost

services.Endpoints=default -h * -p 25567
```

Aunque es opcional, es recomendable hacerlo para tener un poco de orden. Al final, se puede colocar toda la configuración en la misma variable.

5. **Conexiones provistas:** Para gestionar las conexiones provistas, se usa el ObjectAdapter junto con el nombre de la variable que se estableció en el .cfg para configurar el Endpoint.

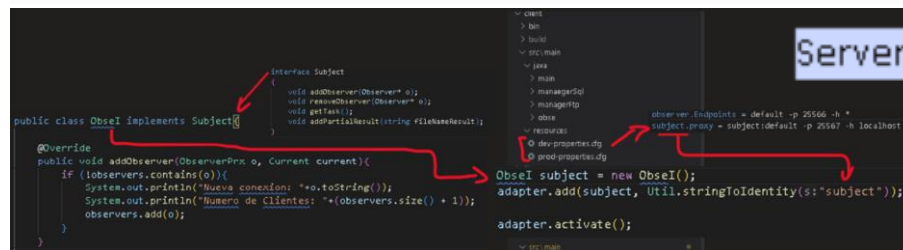


```
try(Communicator communicator = Util.initialize(args, configFile:"properties.cfg")) {
    mainCommunicator = communicator;
    ObjectAdapter adapter = communicator.createObjectAdapter(name:"services");
    services.Endpoints=default -h * -p 25567
}
```

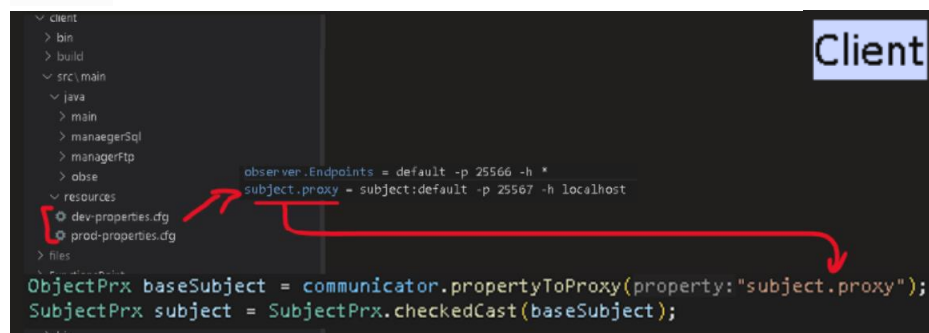
Luego, se inicializa el objeto con la clase que implementamos para que finalmente se agregue al adapter junto con el identificador del lado del cliente. De este modo, la interfaz provista estará lista para que el cliente se conecte

por el proxy (El id subject.proxy va del lado del cliente). Al final, se debe utilizar el método activate(); del adapter para abrir la comunicación.

Se puede usar un mismo adapter para agregar varias conexiones.



6. **Conexiones requeridas:** Para gestionar las conexiones requeridas, se hace la inicialización de la variable ObjectPrx usando el método propertyToProxy(subject.proxy), con el id que colocamos en él .cfg del lado del cliente.



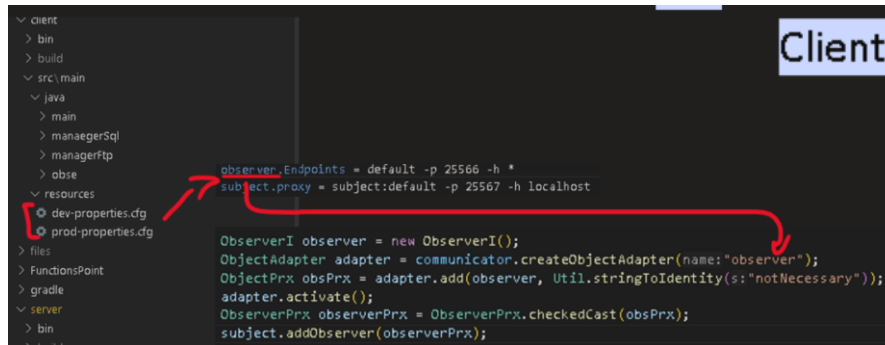
Luego, casteamos este ObjectPrx al objeto SubjectPrx que se generó con el comando slice2java, de tal manera que con este objeto nos podremos comunicar a la interfaz requerida del servidor y poder consumir cualquiera de sus métodos.



En este caso, se puede ver que en el SubjectPrx puede ejecutar el método addObserver() que está del lado del servidor. Pero si somos detallistas, esta función recibe un Observer* como aparece en el archivo .ice. Esto es porque con el '*' se está indicando que reconoce por parámetro el objeto comunicador Observer que tiene arriba que, en este caso, el parámetro comunicador es un ObserverPrx.

Es muy importante que esté arriba primero declarada la interfaz que recibe por parámetro, dado que de esta manera compilará correctamente.

Finalmente, le pasaremos por parámetro esta variable con el fin de que el servidor tenga a la mano el Prx con el cual se va a comunicar con el cliente.



```
observer.Endpoints = default -p 25566 -h *
subject.proxy = subject:default -p 25567 -h localhost

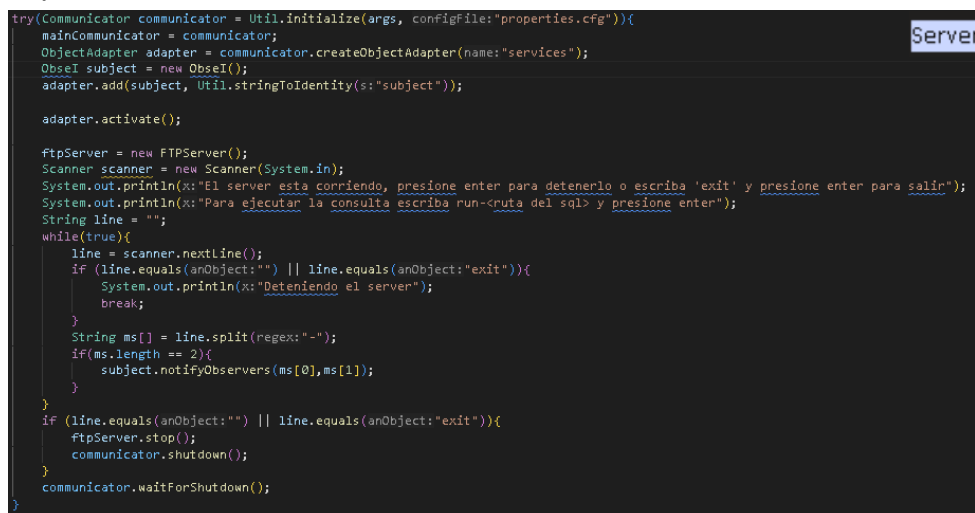
ObserverI observer = new ObserverI();
ObjectAdapter adapter = communicator.createObjectAdapter(name:"observer");
ObjectPrx obsPrx = adapter.add(observer, Util.stringToIdentity(s:"notNecessary"));
adapter.activate();
ObserverPrx observerPrx = ObserverPrx.checkedCast(obsPrx);
subject.addObserver(observerPrx);
```

'notNecessary' solo es una etiqueta para la interfaz provista del cliente, y como su nombre lo indica, no es necesario el id, dado que solo es para que el cliente pueda crear su propio prx para que el servidor se pueda comunicar con el cliente.

Con esto, se completa el ciclo de que el cliente se suscriba al servidor, logrando que el servidor se pueda comunicar con el cliente. Ahora solo falta el callback para que el cliente se pueda comunicar con el servidor cuando sea necesario. Para lograr esto, solo tenemos que guardar el prx subject en el objeto local del observer.

```
ObserverI observer = new ObserverI(subject);
```

Finalmente, es importante llamar el método `waitForShutdown()` del `Communicator` al final del `main` del cliente y del servidor para que no se cierre el proceso de Java.



```
try(Communicator communicator = Util.initialize(args, configFile:"properties.cfg")){
    mainCommunicator = communicator;
    ObjectAdapter adapter = communicator.createObjectAdapter(name:"services");
    ObseI subject = new ObseI();
    adapter.add(subject, Util.stringToIdentity(s:"subject"));

    adapter.activate();

    ftpServer = new FTPServer();
    Scanner scanner = new Scanner(System.in);
    System.out.println(x:"El server esta corriendo, presione enter para detenerlo o escriba 'exit' y presione enter para salir");
    System.out.println(x:"Para ejecutar la consulta escriba run-<ruta del sql> y presione enter");
    String line = "";
    while(true){
        line = scanner.nextLine();
        if (line.equals(anObject:"") || line.equals(anObject:"exit")){
            System.out.println(x:"Deteniendo el server");
            break;
        }
        String ms[] = line.split(regex:"-");
        if(ms.length == 2){
            subject.notifyObservers(ms[0],ms[1]);
        }
    }
    if (line.equals(anObject:"") || line.equals(anObject:"exit")){
        ftpServer.stop();
        communicator.shutdown();
    }
    communicator.waitForShutdown();
}
```

```

try(Communicator communicator = Util.initialize(args, (environment+"-properties.cfg"))
    mainCommunicator = communicator;
    ObjectPrx baseSubject;
    if (host != null) {
        baseSubject = communicator.stringToProxy("subject:default -h " + host + " -p 25567");
    }else{
        baseSubject = communicator.propertyToProxy(property:"subject.proxy");
    }
    SubjectPrx subject = SubjectPrx.checkedCast(baseSubject);

    ObjectAdapter adapter = communicator.createObjectAdapter(name:"observer");
    ObserverI observer = new ObserverI(subject);
    ObjectPrx obsPrx = adapter.add(observer, Util.stringToIdentity(s:"notNecessary"));
    adapter.activate();
    ObserverPrx observerPrx = ObserverPrx.checkedCast(obsPrx);

    if(subject == null){
        throw new Error(message:"Invalid proxy");
    }
    subject.addObserver(observerPrx);

    communicator.waitForShutdown();
}

```

Client

Para finalizar con este tutorial, dejaré un diagrama de secuencia que simplifica el flujo del patrón usado.

