Sistemas Distribuidos – Laboratorio Parte 2

Rev 3

Tras haber realizado la parte 1 del laboratorio, como resultado se debería obtener un servicio plenamente funcional.

En esta segunda parte se tratará de hacer que los servicios se comuniquen entre ellos para llevar a cabo ciertas tareas. Para ello, utilizaremos el mecanismo de comunicación indirecta para descubrir al resto de servicios que necesitemos para realizar nuestras operaciones.

Los objetivos específicos de la práctica son:

- ✓ Familiarizarse con el mecanismo de comunicación indirecta a través de la invocación a métodos remotos utilizando el servicio IceStorm de ZeroC Ice.
- ✓ Fomentar el trabajo en equipos multidisciplinares.
- ✓ Implementar mecanismos de tolerancia a fallos y de prevención de interbloqueos.

Estructura general

El nuevo entorno

Para la realización de esta entrega, se modificarán algunos de los métodos de las interfaces que ya conocéis para incluir algún parámetro adicional que permitirá realizar labores de autenticación.

También se añadirán diferentes mecanismos de invocación indirecta a través de IceStorm que deberán ser implementados para descubrir al resto de servicios e interactuar con ellos cuando sea necesario.

Anunciamiento de servicios

Todos los servicios descritos en el primer entregable deberán ser capaces de enviar y recibir anunciamientos de otros servicios. Esto es muy importante, ya que la única manera que tienen los servicios de comunicarse entre ellos es a través de este descubrimiento.

Interfaz ::IceDrive::Discovery

```
interface Discovery {
    void announceAuthentication(Authentication* service);
    void announceDirectoryService(DirectoryService* service);
    void announceBlobService(BlobService* service);
};
```

Todos los servicios necesitarán tener un *publisher* del *topic* de anunciamientos, así como estar suscritos al mismo para recibirlos.

Cuando un servicio arranque, inmediatamente deberá enviar invocaciones al método "*announceXXXX*" cada 5 segundos periódicamente. Dependiendo del tipo de servicio, enviará uno u otro evento, cuando el servicio recibe una invocación a "*announceXXXX*", deberá guardarlo para poder utilizarlo más adelante. Cuando un microservicio deba cooperar con otro, deberá elegir alguno de los descubiertos. La política de selección se deja al alumno, pudiendo ser round-robin, selección aleatoria, etc. Pero si es importante que cuando una invocación a servicio externo falle

(por ejemplo, por ConnectionError()) se eliminará el proxy de dicho servicio para no volver a ser usado posteriormente.

Resolución diferida

Cuando un servicio recibe una petición que no pueda atender (por ejemplo, una solicitud de *login* de un usuario desconocido) intentará resolverla mediante la colaboración con otras instancias de su mismo tipo de servicio. El mecanismo general será el siguiente:

1) Crear un objeto para almacenar la respuesta de las otras instancias
2) Enviar un evento con la solicitud que se pretende resolver y un proxy al objeto que recibirá la respuesta (creado en el punto 1).
3) Si en 5 segundos no se ha obtenido una respuesta, se cancela la solicitud, se elimina el objeto que almacenaría la respuesta y se responde con el error correspondiente al cliente, o la salida acordada específicamente en cada servicio.
4) Si se recibe una respuesta, se reenvía al cliente.

Los detalles sobre la resolución en diferido se explicarán para cada servicio.

Servicio de autenticación.

El servicio de autenticación podrá tener más de una instancia del programa funcionando a la vez en el sistema. Esto implicará que tanto el "login" de usuarios, su borrado, así como la verificación de los objetos *User* tendrán que hacerse de manera colaborativa entre las instancias:

Método "login"

1. Si el usuario está en la base de datos, se comportará igual que en el entregable 1, comprobando la contraseña y permitiendo la autenticación o no.
2. Si el usuario no existe en la persistencia, el nombre de usuario y contraseña recibidas se enviarán a un *topic de IceStorm* donde todas las instancias del servicio de autenticación (y sólo ellas) estarán suscritas.
3. Cada instancia, cuando reciba esa petición, será la encargada de comprobar si el usuario está en su lista de acceso. Si es así, enviará una invocación remota al servicio que realizó la consulta, respondiendo con el objeto *User* o comunicando la excepción, según si la contraseña es válida o no.
4. Si el servicio no recibe ninguna respuesta en 5 segundos, se considerará que el usuario es desconocido (no está registrado en ninguna instancia del servicio).

Método "newUser"

1. Si el usuario está en la base de datos, se comportará igual que en el entregable 1, devolviendo la excepción adecuada.
2. Si el usuario no existe en la persistencia, se consultará la existencia del mismo a través del *topic*, donde otras instancias del servicio Authentication recibirán la consulta.
3. Cada instancia, cuando reciba la petición, comprobará si el usuario en cuestión está en su persistencia, y de ser así, responderá notificando sobre ello.
4. Si el servicio recibe alguna respuesta, responderá igual que en el punto 1, dado que el usuario ya existe.
5. Si el servicio no recibe ninguna respuesta en 5s, se creará el usuario en la persistencia local del servicio y se devolverá el objeto "User" correspondiente.

Método “removeUser”

1. Si el usuario está en la base de datos, se comportará igual que en el entregable 1, comprobando la contraseña y permitiendo el borrado o no.
2. Si el usuario no existe en la persistencia, el nombre de usuario y contraseña recibidas se enviarán a un *topic de IceStorm* donde todas las instancias del servicio de autenticación (y sólo ellas) estarán suscritas.
3. Cada instancia, cuando reciba esa petición, será la encargada de comprobar si el usuario está en su lista de acceso. Si es así, enviará una invocación remota al servicio que realizó la consulta, respondiendo con una invocación remota que indicará que el usuario ha sido eliminado, según si la contraseña es válida o no.
4. Si el servicio no recibe ninguna respuesta en 5 segundos, se considerará que el usuario es desconocido (no está registrado en ninguna instancia del servicio).

Método “verifyUser”

1. Si el objeto *User* es de la instancia local, se comportará como en el entregable 1.
2. En caso contrario, se enviará al *topic* para extender esa consulta al resto de instancias.
3. Cada instancia que reciba dicha consulta deberá ignorarla si no reconoce al objeto *User*, y deberá responder con una invocación directa al servicio que envió el mensaje en caso de reconocerlo como propio.
4. Si el servicio no recibe ninguna respuesta en 5 segundos, se considerará que el objeto User no es válido y se responderá en consecuencia.

Interfaz ::IceDrive::AuthenticationQuery.

La interfaz se define de la siguiente manera:

```
interface AuthenticationQuery {
   void login(string username, string password, AuthenticationQueryResponse* response);
   void doesUserExists(string username, AuthenticationQueryResponse* response);
   void removeUser(string username, string password, AuthenticationQueryResponse*
response);
   void verifyUser(User *user, AuthenticationQueryResponse* response);
};
```

Esta es la interfaz a la que se suscribirán todas las instancias del servicio de autenticación.

1) **Autenticar usuario:** el servicio recibe la llamada al método “login” de esta interfaz con las credenciales de usuario. Si no son correctas, se ignorarán. Si el usuario existe, se enviará la respuesta a través del proxy de tipo *AuthenticationQueryResponse* recibido.
2) **Comprobar usuario:** el servicio recibe la llamada al método “doesUserExists” de esta interfaz con el nombre del usuario. Si existe en la persistencia, se responde usando el método adecuado de la interfaz *AuthenticationQueryResponse*. Si no existiera, se ignoraría el mensaje.
3) **Borrar usuario:** el servicio recibe la invocación al método “removeUser” con las credenciales del usuario. Si el usuario existe, se eliminará de la persistencia y se responderá adecuadamente a través de la interfaz *AuthenticationQueryResponse*. Si no existiera, se ignoraría dicha invocación y no se enviaría resuesta alguna.
4) **Verificar usuario:** si el servicio recibe la invocación al método “*verifyUser*” de esta interfaz, deberá comprobar si el objeto *User* le pertenece. Si es así, deberá responder a través de una invocación en el proxy de tipo *AuthenticationQueryResponse* recibido.

Interfaz ::IceDrive::AuthenticationQueryResponse.

```
interface AuthenticationQueryResponse {
    void loginResponse(User* user);
    void userExists();
    void userRemoved();
    void verifyUserResponse(bool result);
};
```

Esta interfaz se utilizará únicamente para recibir las respuestas a las consultas realizadas por el servicio a través del canal de eventos *IceStorm*. Siempre que el servicio reciba un "login" o un "verifyUser" al que no sea capaz de responder por sí mismo, deberá crear un objeto que cumpla con la interfaz descrita y enviarlo a través del *publisher* del topic IceStorm.

1) **Respuesta a consulta de login:** El servicio que reconozca el nombre de usuario invocará esta respuesta enviando el nombre de usuario y contraseña sobre los que se realizó la consulta y adjuntando un objeto *User* en caso de que las credenciales fueran correctas, o el objeto nulo en caso contrario.
2) **Respuesta a consulta de existencia de usuario:** El servicio que reconozca al usuario proporcionado en la *query* responderá invocando a "userExists".
3) **Respuesta a consulta de eliminación de usuario:** El servicio que reconozca al citado usuario y sus credenciales, lo eliminará y responde usando el método remoto "userRemoved" para indicar que la acción solicitada ha sido realizada.
4) **Respuesta a la consulta de verifyUser:** El servicio que reconozca el objeto *User** enviado invocará este método indicando cuál es el objeto que ha sido verificado y el resultado. En caso de que el servicio reconozca el objeto, pero éste haya dejado de ser válido, el resultado será "falso".

Servicio de directorio.

En esta entrega, el servicio de directorio deberá ser capaz de descubrir y comunicarse con los servicios de "Authentication" y "BlobService" para cooperar con ellos para realizar diversas tareas.

Además, al igual que el resto de servicios podrá ser ejecutado en un entorno con diferentes instancias del servicio "DirectoryService", por lo que también deberá ser capaz de cooperar con ellas.

Interfaz ::IceDrive::DirectoryService. Cambios y casos de uso.

La interfaz se define en slice de la siguiente manera:

```
interface DirectoryService {
    Directory* getRoot(User*user);
};
```

Como se puede observar, en lugar de recibir un "string" con el nombre del usuario, se recibirá un proxy de tipo User. El servicio deberá verificar, al recibir el objeto por primera vez, que dicho objeto pertenece a un Authentication service válido. Para ello, se utilizará un proxy a un servicio de Autenticación que se haya descubierto a través de la interfaz de descubrimiento explicada al principio de este documento.

Dicho proxy de tipo User deberá ser manejado por el servicio para que cada operación realizada por los directorios que pertenecen a dicho usuario requiera de una comprobación "isAlive" sobre el objeto User. De ese modo, si el cliente no actualiza las credenciales periódicamente, no podrá utilizar ninguno de los proxies generados.

Esta interfaz tiene únicamente el método de acceso al servicio, los casos de uso que se deben contemplar son los siguientes:

1) Se solicita un directorio raíz de un usuario que ya lo solicitó previamente: en ese caso, se devolverá el objeto Directory() correspondiente.
2) Si la instancia no dispone del directorio raíz del usuario, intentará una resolución en diferido. Si pasados 5 segundos no se obtiene tampoco un resultado, se creará un nuevo directorio root y se retornará al usuario.

Resolución en diferido de getRoot().

Las siguientes interfaces se usan para implementar la resolución en diferido:

```
interface DirectoryQuery {
   void rootDirectory(User* user, DirectoryQueryResponse* response);
}
```

Cuando una instancia no disponga del directorio root de un usuario concreto, lanzará un evento en el canal asociado a la interfaz DirectoryQuery(). El evento incluye un proxy a un objeto DirectoryQueryResponse() donde la instancia que disponga de dicho directorio deberá enviar el objeto solicitado mediante el uso de una instancia de la siguiente interfaz:

```
interface DirectoryQueryResponse {
   void rootDirectoryResponse(Directory* root);
};
```

De manera que una instancia que disponga del root de un determinado usuario, si recibe un evento preguntando por dicho root, la instancia invocará al método rootDirectoryResponse() con una instancia del directorio root solicitado.

Interfaz ::IceDrive::Directory. Cambios y casos de uso.

Se define la interfaz de la siguiente forma:

```
interface Directory {
 string getPath();
 Directory* getParent() throws RootHasNoParent;
 Strings getChilds();
 Directory* getChild(string childName) throws ChildNotExists;
 Directory* createChild(string childName) throws ChildAlreadyExists;
 void removeChild(string childName) throws ChildNotExists;

 Strings getFiles();
 string getBlobId(string filename) throws FileNotFound;
 void linkFile(string fileName, string blobId) throws FileAlreadyExists, TemporaryUnavailable;
 void unlinkFile(string fileName) throws FileNotFound, TemporaryUnavailable;
};
```

En todo momento la instancia deberá conocer el proxy de objeto “User” para la que fue creada y atender a los métodos sí y sólo sí el objeto sigue estando activo (“isAlive” devuelve “true”).

Dado que para la creación del objeto ya se debió verificar que la instancia de “User” era válida, no será necesario comprobarlo.

Los métodos “linkArchive” y “unlinkArchive” deberán contactar con un servicio de almacenamiento de archivos (“BlobService”) y realizar el enlazado o desenlazado de cada archivo respectivamente. En

caso de no haberse descubierto ninguna instancia de “BlobService” a través del mecanismo de descubrimiento de servicios explicado con anterioridad, se lanzará la excepción “TemporaryUnavailable”.

Servicio de almacenamiento.

Los cambios requeridos en éste servicio respecto al primer entregable se basan, sobre todo, en el descubrimiento y utilización del servicio Authentication para poder verificar los objetos User que se recibirán en los diferentes métodos que así lo necesiten.

Interfaz ::IceDrive::DataTransfer. Casos de uso

La interfaz es la siguiente (sin cambios respecto al entregable 1)

```
interface DataTransfer{
    Bytes read(int size) throws FailedToReadData;
    void close();
};
```

Interfaz ::IceDrive::BlobService. Cambios y casos de uso

Interfaz BlobService:

```
interface BlobService{
    void link(string blobId) throws UnknownBlob;
    void unlink(string blobId) throws UnknownBlob;

    String upload(User* user, DataTransfer* blob) throws FailedToReadData,
TemporaryUnavailable;
    DataTransfer* download(User* user, string blobId) throws UnknownBlob,
TemporaryUnavailable;
};
```

Los métodos “upload” y “download”, que serán invocados por la aplicación cliente, deberán pasar como primer argumento un proxy al objeto User relativo al usuario que realiza dicha acción.

El servicio deberá verificar la autenticidad de dicho proxy a través del método “veryfyUser” del servicio de autenticación. Para acceder a dicho servicio, deberá utilizar el *topic* de Descubrimiento de servicios explicado en apartados anteriores.

Resolución en diferido de download(), upload(), link() y unlink().

Las siguientes interfaces se usan para implementar la resolución en diferido:

```
interface BlobQuery {
    void downloadBlob(string blobId, BlobQueryResponse* response);
    void blobIdExists(string blobId, BlobQueryResponse* response);
    void linkBlob(string blobId, BlobQueryResponse* response);
    void unlinkBlob(string blobId, BlobQueryResponse* response);
}
```

Cuando una instancia no disponga de un blob que se está intentando descargar o actualizar el número de enlaces, se creará un nuevo objeto *BlobQueryResponse()* y se lanzará el evento correspondiente el cual incluirá el proxy del objeto respuesta. También se utilizará el mismo mecanismo cuando el usuario haga una subida de un archivo, para que la instancia del servicio pueda decidir si almacenar el nuevo archivo de forma local o, si ya existe en otra instancia, descartarlo. El objeto respuesta ofrece la siguiente interfaz:

```
interface BlobQueryResponse {
   void downloadBlob(DataTransfer* blob);
   void blobExists();
   void blobLinked();
   void blobUnlinked();
 };
```

De manera que una instancia que disponga del blob por el que se desea realizar la petición, en caso de responder a una solicitud de descarga, creará el objeto *DataTransfer()* y enviará el proxy mediante el método *BlobQueryResponse.downloadBlob()*. En caso de que la solicitud sea actualizar el número de enlaces del blob, confirmará la ejecución de la operación invocando a *BlobQueryResponse().blobLinked()* o *BlobQueryResponse().blobUnlinked()* según corresponda.

Entrega

La fecha de entrega para el primer entregable será el 19 de enero de 2024 y consistirá en un enlace a un repositorio del código en GitHub a través de la plataforma de Campus Virtual. Los profesores de prácticas de la asignatura deben tener acceso a dicho repositorio, que debe ser privado para evitar plagios.

El repositorio deberá incluir todo el código fuente tanto del servicio asignado al alumno como de aquellos programas de prueba que haya podido utilizar, así como código de pruebas unitarias, ficheros de configuración, scripts, etc.

Durante las sesiones de prácticas se proporcionará acceso a los estudiantes a un repositorio plantilla donde se explicará detenidamente los ficheros que serán necesarios para la realización de la entrega.

Documentación

No se solicitará documentación adicional, pero se recomienda encarecidamente incluir en el repositorio tantos ficheros con información sobre el desarrollo. También se consultará el historial de commits en Git.

Defensa

La práctica se corrige a través de un correctos automático que realizará las pruebas pertinentes a los servicios, es habitual que este sistema puntúe a la baja porque es muy sensible a los defectos de forma (por ejemplo, si el ejecutable tiene como shebang “/usr/bin/python3” en lugar de “/usr/bin/env python3”) por lo que la nota inicial podría ser menor que en una corrección manual donde esos defectos son ignorados. Quien desee solicitar una defensa de práctica después de la publicación inicial de notas podrá hacerlo comunicándolo a su profesor de prácticas mediante correo electrónico. Salvo excepciones, la defensa se realizará online y en ella el alumno demostrará manualmente el funcionamiento de su práctica.

El profesorado también podrá solicitar una defensa “manual” cuando estime oportuno.