# Laboratorio Sistemas Distribuidos 24/25 - Entregable 1

Éste documento es el enunciado del primer entregable de la práctica de Sistemas Distribuidos del curso 2024/2025 en convocatoria ordinaria.

## Introducción

Se quiere implementar un servicio de almacenamiento para diferentes estructuras de datos: listas, conjuntos y diccionarios. El servicio realizará la persistencia de los citados objetos, permitiendo su acceso y modificación de manera interactiva.

El servicio debe implementarse utilizando Python y el *middleware* de comunicación **ZeroC Ice**.

## Requisitos

El servicio permitirá la creación o recuperación de objetos de los 3 tipos citados. Las estructuras de datos dfinidas tendrán las siguientes limitaciones:

- Listas: únicamente permitirá añadir cadenas.
- Conjuntos: únicamente permitirá añadir cadenas.
- Diccionarios: únicamente almacenarán diccionarios con clave de tipo cadena y valor de tipo cadena.

Como se ha citado anteriormente, el servicio debe realizar persistencia de los datos. El formato de dicha persistencia se deja a elección de los alumnos, aunque se recomienda el uso de ficheros **JSON** a través de la libraría `json` incluída en la librería estándar de Python.

La ruta a dichos ficheros de persistencia, o en caso de usar otra tecnología, todos los parámetros necesarios para su configuración deben poder ser definidos en el fichero de configuración del servicio, junto con el resto de parámetros que se puedan configurar del *middleware* de comunicación.

## Definición de interfaces

En el repositorio plantilla se proporciona una versión completa del fichero Slice. En éste enunciado se enumeran las interfaces presentes y su significado:

### `RType`

Es una interfaz de la que heredarán los 3 tipos de objetos que serán implementados (listas, conjuntos y diccionarios). No se crearán objetos que implemente únicamente ésta interfaz, pero permite agrupar ciertas operaciones que deben existir en todos los demás tipos:

```
interface RType {
```

```
    void remove(string item) throws KeyError;
    idempotent int length();
    idempotent bool contains(string item);
    idempotent long hash();
    Iterable* iter();
};
```

- `remove` permitirá eliminar una cadena presente en la estructura de datos. En el caso de los diccionarios, el argumento `item` se refiere a la clave que debe ser eliminada del mismo.
- `length` devolverá un número entero positivo o 0, indicando el número de elementos presentes en la estructura de datos. Es equivalente a llamar al método `len` en Python.
- `contains` devolverá un valor booleano en función de si la cadena está presente en la estructura de datos o no. Al igual que con `remove`, en el caso de los diccionarios hace referencia a las claves del diccionario, no a sus valores. Es equivalente al uso del operador `in` en Python.
- `hash` debe calcular un valor entero en función de los elementos que haya contenidos en la estructura de datos. Dicho valor debe servir para identificar el estado del objeto y no se va a utilizar para comparar diferentes objetos entre ellos.
- `iter` creará un objeto de la interfaz `Iterable` que servirá para iterar a través de los contenidos del objeto. En el caso de las listas, es importante tener en cuenta que el orden importa, mientras que en conjuntos y diccionarios es indiferente.

#### `RDict`, diccionario remoto

Representa a un diccionario. Además de los métodos definidos en su interfaz base (`RType`) explicada más arriba, ofrece las siguientes operaciones:

```
interface RDict extends RType {
    idempotent void setItem(string key, string item);
    idempotent string getItem(string key) throws KeyError;
    string pop(string key) throws KeyError;
};
```

- `setItem` asignará en el diccionario a la clave especificada por `key` el valor `item`. Es el equivalente a la asignación de un valor en un diccionario (`d[key] = item`).
- `getItem` recuperará el valor asociado a la clave especificada. Es el equivalente al selector de Python o al método `dict.get`: `return d[key]`.
- `pop` igual que `getItem`, pero además eliminará la clave y su valor asociado del diccionario. Sería igual que llamar de una misma vez a `getItem` y

`remove`. Es equivalente al método `dict.pop` en Python.

#### `RList`, lista remota

Representa a una lista. Además de los métodos definidos en su interfaz base (`RType`) explicada más arriba, ofrece las siguientes operaciones:

```
interface RList extends RType {
    void append(string item);
    string pop(optional(1) int index) throws IndexError;
    idempotent string getItem(int index) throws IndexError;
};
```

- `append` añade un elemento al final de la lista. Equivalente al método `append` en las listas de Python.
- `pop` recupera un elemento de una posición específica de la lista y lo elimina. Si no se especifica la posición, se devolverá y eliminará el último. Equivalente al método `list.pop` de Python.
- `getItem` recupera un elemento de una posición específica de la lista. Es equivalente al uso de corchetes en Python: `return l[5]`.

#### `RSet`, conjunto remoto

Representa a un conjunto. Además de los métodos definidos en su interfaz base (`RType`) explicada más arriba, ofrece las siguientes operaciones:

```
interface RSet extends RType {
    idempotent void add(string item);
    string pop() throws KeyError;
};
```

- `add` añade un elemento al conjunto. Equivalente al método `add` en los `set` de Python.
- `pop` recupera un elemento cualquiera del conjunto y lo elimina.Equivalente al método `set.pop` de Python.

#### `Iterable`

Representa a un iterador remoto sobre cualquiera de los tipos definidos anteriormente. La única forma de obtener un objeto que cumpla ésta interfaz es a través del método `iter` que tienen todos los tipos remotos.

```
interface Iterable {
    string next() throws StopIteration, CancelIteration;
};
```

- `next` devolverá la siguiente cadena del iterador. Cuando se haya iterado por todos los elementos de la estructura de datos, el método lanzará la excepción `StopIteration` definida en el Slice.

Una nota muy importante sobre la implementación es que un iterador deja de ser válido cuando el objeto que está siendo iterado es modificado, ya sea añadiendo nuevos elementos, eliminándolos o modificándolos (en el caso de los diccionarios).

Si con un iterador activo el objeto iterado fuera modificado, el método `next` deberá lanzar la excepción `CancelIteration`.

#### `Factory`

Es la interfaz principal del servicio. Cada instancia del servidor tendrá únicamente un sirviente de ésta interfaz.

```
enum TypeName { RDict, RList, RSet };

interface Factory {
    RType* get(TypeName typeName, optional(1) string identifier);
};
```

Se muestra también el enumerado `TypeName`, que es necesario para las llamadas al método de la factoría.

- `get` devolverá un objeto de uno de los 3 tipos especificados anteriormente, dependiendo del valor que tome el argumento `typeName`. Opcionalmente se podrá especificar un identificador al objeto para poder reutilizarlo más adelante.

Aunque, como se ha mencionado en la definición de la interfaz `RType`, no pueden existir objetos que implementen únicamente esa interfaz, todos los objetos `RSet`, `RList` y `RDict` van a implementar también `RType` por herencia.

Para que un cliente o una prueba pueda utilizar el objeto con su tipo correcto deberá realizar un casting de tipo de la manera habitual en ZeroC Ice (por ejemplo, con `RSetPrx.checkedCast`).

## Entregable

La práctica se deberá realizar y almacenar en un repositorio de Git privado. La entega consistirá en enviar la URL a dicho repositorio, por lo que el alumno deberá asegurarse de que los profesores tengan acceso a dicho repositorio.

El repositorio deberá contener al menos lo siguiente:

- `README.md` con una explicación mínima de cómo configurar y ejecutar el servicio, así como sus dependencias si las hubiera.
- El fichero o ficheros de configuración necesarios para la ejecución.

- Algún sistema de control de dependencias de Python: fichero `pyproject.toml`, `setup.cfg`, `setup.py`, Pipenv, Poetry, `requirements.txt`... Su uso debe estar también explicado en el fichero `README.md`

**Fecha de entrega**

La fecha de entrega será el día X de Y de 2024.

## Repositorio plantilla

En CampusVirtual se compartirá un repositorio plantilla en GitHub que los alumnos podrán utilizar para crear su propio repositorio o para clonarlo y comenzar uno nuevo desde "cero".

Dicho repositorio plantilla contiene todos los requisitos especificados en la sección anterior:

- Fichero `README.md`
- Fichero `pyproject.toml` con la configuración del proyecto y de algunas herramientas de evaluación de código.
- El fichero Slice.
- Un paquete Python llamado `remotetpes` con la utilería necesaria para cargar el Slice.
- Módulos dentro de dicho paquete para cada tipo de datos remoto, con la definición del esqueleto de la clase.
- Un módulo `customset.py` con la implementación de un `set` que sólo admite objetos `str` vista en clase.
- Módulo `remoteset.py` con la implementación del objeto `RSet` completa, por lo que sólo sería necesario implementar `Rlist`, `RDict`, `Iterable` y `Factory`.
- Módulo `server.py` con la implementación de una `Ice.Application` que permite la ejecución del servicio y añadir el objeto factoría existente (en la plantilla, sin implementar sus métodos).
- Módulo `command_handlers.py` con el manejador del comando principal (el equivalente a la función `main`).
- Paquete `tests` con una batería mínima de pruebas unitarias para que sirva de inspiración para realizar más.
- Fichero de configuración necesario para ejecutar el servicio.
- Directorio `.github/workflows` con la definición de algunas "Actions" de la integración continua de GitHub que realizan un análisis estático de tipos y de la calidad del código.

Se recomienda encarecidamente utilizar dicha plantilla y leer bien el `README.md` para entender el funcionamiento, aunque no es obligatorio su uso.