Programación para las Comunicaciones
Práctica 2

Pedro José Fernández Escámez

3 de junio de 2025

Índice

1. Descripción de la solución

La aplicación desarrollada implementa un sistema cliente-servidor para la distribución de información meteorológica simulada. El sistema consta de múltiples servidores, cada uno de los cuales gestiona un conjunto de variables meteorológicas (como temperatura, humedad, presión), generando valores aleatorios para estas y difundiéndolos periódicamente a la red mediante mensajes UDP broadcast.

Un cliente (o potencialmente varios, aunque la implementación actual se centra en uno) escucha estos mensajes de difusión. El cliente puede procesar los datos recibidos, que pueden estar codificados en formato XML o JSON, y los muestra en consola. Además, el cliente dispone de una interfaz de línea de comandos (shell) que permite al usuario enviar mensajes de control UDP unicast a un servidor específico (identificado por su IP y un puerto de control único) para modificar su comportamiento. Las operaciones de control incluyen cambiar la codificación de los mensajes de difusión del servidor (entre XML y JSON), ajustar la frecuencia de envío de datos, modificar las unidades de una variable específica, activar o desactivar el envío de datos por parte del servidor, y detener el proceso del servidor.

Cada servidor opera en un hilo principal que maneja tanto el envío periódico de datos de difusión como la escucha de mensajes de control en un puerto UDP único y configurable en el momento de su arranque. Al recibir un mensaje de control, el servidor lo procesa, aplica la operación solicitada y envía un mensaje de respuesta unicast al cliente que originó la petición. Los mensajes de control enviados por el cliente y las respuestas enviadas por el servidor utilizan el formato JSON.

La aplicación utiliza la librería GSON para la serialización y deserialización de objetos Java a/desde JSON. Para XML, se utiliza la API DOM de Java, y se implementa la validación de los mensajes XML de difusión recibidos por el cliente mediante un DTD. Todos los mensajes transmitidos y recibidos (tanto por el cliente como por los servidores) se registran en archivos de log para su posterior verificación. El proyecto está gestionado con Maven para facilitar la compilación y la gestión de dependencias.

2. Diagrama de clases

La aplicación se ha estructurado utilizando Maven, con la siguiente organización de paquetes principal dentro de `src/main/java/`:

- `cliente`: Contiene la clase `Client`, responsable de la lógica del cliente, incluyendo la interfaz de usuario (shell), el envío de mensajes de control, y la recepción y procesamiento de mensajes de difusión.
- `servidor`: Contiene la clase `Server`, que implementa la lógica del servidor, incluyendo la generación y difusión de datos, y la recepción y procesamiento de mensajes de control.
- `common`: Agrupa las clases de modelo de datos compartidas entre el cliente y el servidor, como `AbstractMessage`, `DistributionMessage`, `ControlMessage`, `ResponseMessage`, y `WeatherVariable`. También incluye la clase de utilidad `MessageUtils` para la serialización/deserialización XML/JSON y la validación XML.

Los recursos, como el archivo DTD para la validación XML (`distribution_message.dtd`), se ubican en `src/main/resources/`.

Las clases más relevantes implementadas son:

- **Client**: Hilo principal del cliente. Gestiona una interfaz de línea de comandos para que el usuario introduzca órdenes y otro hilo para escuchar continuamente los mensajes de difusión de los servidores. Envía mensajes de control al servidor actualmente seleccionado.

- **Server**: Hilo principal del servidor. Administra un conjunto de variables meteorológicas, genera datos para ellas periódicamente y los difunde. Escucha en un puerto UDP único y específico (configurado al inicio) los mensajes de control, los procesa y envía respuestas.
- **MessageUtils**: Clase de utilidad con métodos estáticos para serializar objetos Java a cadenas JSON (usando GSON) y a cadenas XML (usando DOM), y para deserializar cadenas JSON y XML a los objetos de mensaje correspondientes. También maneja la validación de XML contra un DTD.
- **AbstractMessage**: Clase base abstracta para todos los tipos de mensajes, conteniendo campos comunes como ID de mensaje, timestamp y formato de codificación.
- **DistributionMessage**: Subclase de `AbstractMessage`, representa los mensajes de difusión enviados por los servidores. Contiene el ID del servidor y una lista de objetos `WeatherVariable`.
- **ControlMessage**: Subclase de `AbstractMessage`, representa los mensajes de control enviados por el cliente a un servidor. Contiene el comando a ejecutar, el ID del servidor destino y parámetros adicionales.
- **ResponseMessage**: Subclase de `AbstractMessage`, representa las respuestas enviadas por un servidor a un cliente tras procesar un mensaje de control. Contiene el estado de la operación y detalles.
- **WeatherVariable**: Clase que encapsula los datos de una variable meteorológica individual: nombre, valor y unidad.

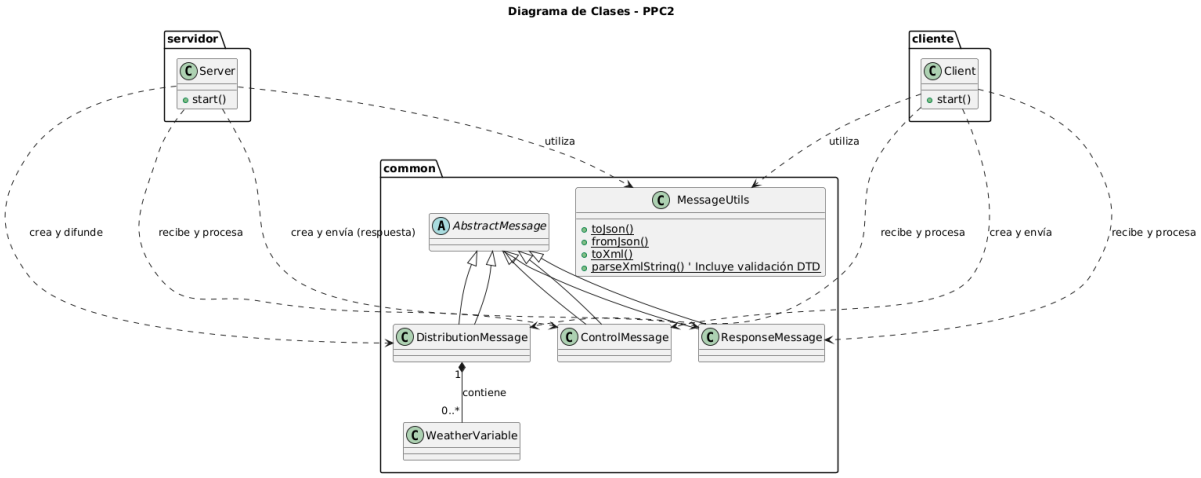

Figura 1: Diagrama de clases de la aplicación.

3. Diagrama de secuencia

3.1. Difusión de información del servidor

El servidor, de forma periódica, crea un objeto `DistributionMessage` con los valores actuales de sus variables meteorológicas. Este objeto se serializa (usando `MessageUtils`) al formato de codificación configurado actualmente para el servidor (JSON o XML). El mensaje serializado, precedido por un indicador de formato ("JSON\n.º "XML\n"), se envía como un paquete UDP broadcast. El cliente escucha estos paquetes, extrae el indicador de formato, y utiliza `MessageUtils` para deserializar el payload al objeto `DistributionMessage` correspondiente, validando el XML si es el caso.

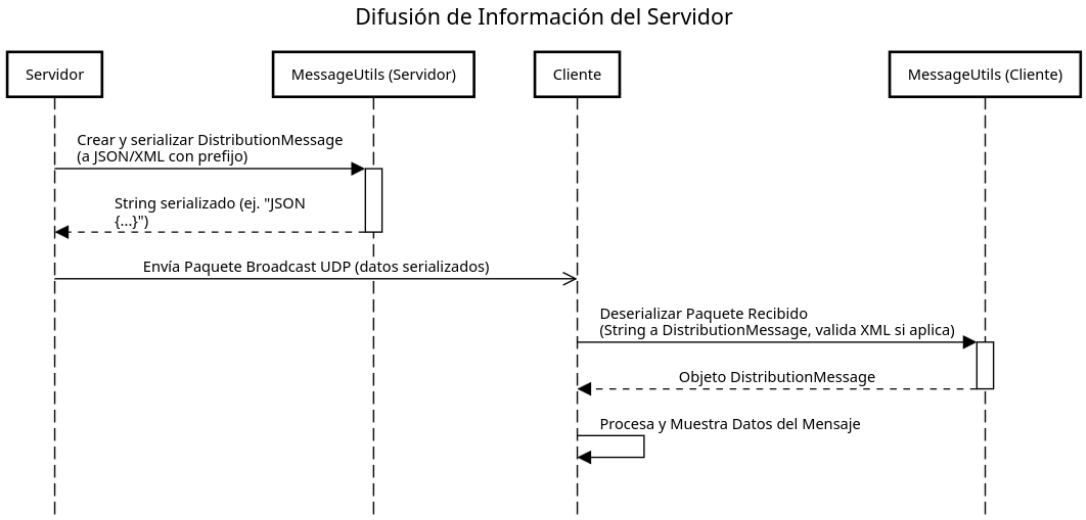

Figura 2: Diagrama de secuencia para la difusión de información del servidor.

3.2. Mensajes de control y respuesta

El usuario introduce un comando en la consola del cliente. La clase `Client` interpreta el comando, crea un objeto `ControlMessage` y lo puebla con los datos necesarios (comando, ID del servidor destino, parámetros). Este objeto se serializa a JSON usando `MessageUtils`. El mensaje JSON se envía como un paquete UDP unicast al servidor y puerto especificados por el comando `TARGET_SERVER`. El servidor correspondiente, que está escuchando en su puerto de control único, recibe el paquete. Deserializa el JSON a un objeto `ControlMessage` usando `MessageUtils`. Procesa el comando, realiza la acción solicitada (e.g., cambiar frecuencia, codificación, etc.), y crea un objeto `ResponseMessage`. Este objeto de respuesta se serializa a JSON y se envía como un paquete UDP unicast de vuelta al cliente (a la dirección IP y puerto desde donde se originó el mensaje de control). El cliente recibe esta respuesta, la deserializa y muestra el estado/detalles al usuario.

4. Mensajes de control implementados

Los mensajes de control son introducidos por el usuario en la consola del cliente. Los comandos implementados son:

- `TARGET_SERVER <ip_address><port>`: Establece el servidor (IP) y puerto de control destino para los siguientes comandos de control.
- `SET_ENCODING <TargetServerID><JSON|XML>`: Solicita al servidor especificado que cambie la codificación de sus futuros mensajes de difusión al formato indicado.
- `SET_FREQUENCY <TargetServerID><milliseconds>`: Solicita al servidor especificado que cambie la frecuencia (periodo en milisegundos) con la que envía sus mensajes de difusión.
- `SET_UNIT <TargetServerID><VariableName><NewUnit>`: Solicita al servidor especificado que cambie las unidades de una de sus variables meteorológicas.
- `ACTIVATE_SERVER <TargetServerID>`: Solicita al servidor especificado que comience (o reanude) el envío de mensajes de difusión. (Corresponde a ‘TOGGLE_SENDING_DATA‘ con parámetro ‘active=true‘).

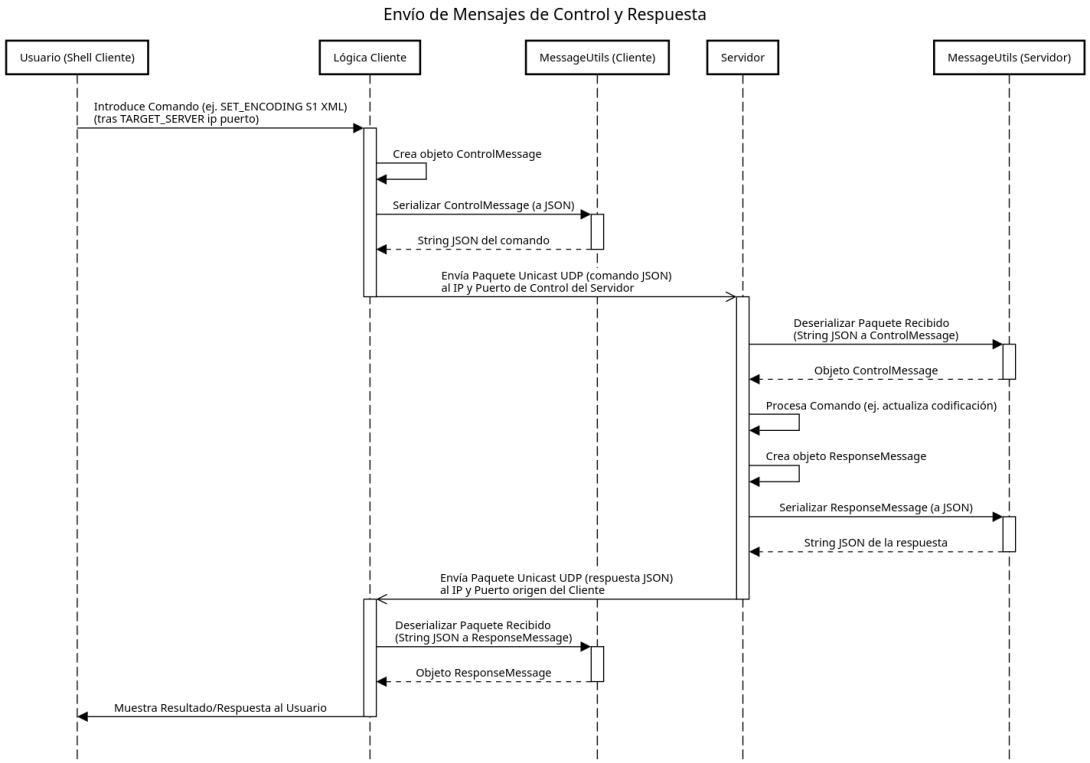

Figura 3: Diagrama de secuencia para los mensajes de control y respuesta.

- `DEACTIVATE_SERVER <TargetServerID>`: Solicita al servidor especificado que detenga (o pause) el envío de mensajes de difusión. (Corresponde a 'TOGGLE_SENDING_DATA' con parámetro 'active=false').
- `STOP_SERVER <TargetServerID>`: Solicita al servidor especificado que detenga su ejecución. (Corresponde a 'STOP_SERVER_PROCESS').
- `HELP`: Muestra una lista de los comandos disponibles en la consola del cliente.
- `EXIT`: Termina la ejecución del cliente.

5. Descripción del protocolo

Formato de Mensajes de Difusión: Los mensajes de difusión enviados por los servidores utilizan un prefijo para indicar el formato del payload. Consisten en la cadena "JSON." o "XML", seguida de un carácter de nueva línea (`\n`), y luego el payload del mensaje serializado en el formato correspondiente.

```
Ejemplo de payload de difusión JSON:
JSON
{"messageType":"DISTRIBUTION","serverId":"S1", ... ,
"variables":[{"name":"temp","value":20.0,"unit":"C"}, ...]}

Ejemplo de payload de difusión XML:
XML
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE distributionMessage SYSTEM "distribution_message.dtd">
<distributionMessage serverId="S1" messageType="DISTRIBUTION" ...>
...</distributionMessage>
```

El DTD (`distribution_message.dtd`) se utiliza para validar los mensajes XML de difusión en el cliente.

Formato de Mensajes de Control y Respuesta: Los mensajes de control (cliente a servidor) y los mensajes de respuesta (servidor a cliente) se envían como cadenas JSON directamente, sin el prefijo de formato.

```
Ejemplo de payload de control (JSON):
{"messageType":"CONTROL","command":"SET_ENCODING","targetServerId":"S1",
... ,"parameters":{"encoding":"XML"}}

Ejemplo de payload de respuesta (JSON):
{"messageType":"RESPONSE","originalMessageId":"...", "status":"OK",
"details":"Encoding set to XML"}
```

Comunicación con Sockets UDP:

- **Cliente**:
  - Utiliza un `DatagramSocket` vinculado a un puerto fijo (ej. 5000) para escuchar los mensajes de difusión enviados por los servidores.
  - Utiliza otro `DatagramSocket` (vinculado a un puerto efímero asignado por el sistema operativo) para enviar mensajes de control unicast al puerto específico de un servidor y para recibir el mensaje de respuesta unicast de dicho servidor.
- **Servidor**:
  - Utiliza un `DatagramSocket` (vinculado a un puerto efímero) para enviar los mensajes de difusión UDP a la dirección de broadcast.
  - Utiliza otro `DatagramSocket` vinculado a un puerto UDP único y fijo para esa instancia de servidor (ej. S1 en 5001, S2 en 5002) para escuchar los mensajes de control unicast enviados por los clientes. Este mismo socket se utiliza para enviar el mensaje de respuesta unicast de vuelta al cliente.

6. Ejecución de la Práctica

La aplicación se compila y empaqueta utilizando Maven. Consta de dos puntos de entrada principales (clases con método `main`): `servidor.Server` y `cliente.Client`.

6.1. Servidor

Para ejecutar un servidor, se invoca la clase `servidor.Server`. Requiere como argumentos de línea de comandos: el ID del servidor (una cadena, ej. "S1"), el puerto UDP en el que escuchará los mensajes de control (un número, ej. 5001), y opcionalmente, pares de nombre y unidad para las variables meteorológicas que gestionará. Si no se especifican variables, se utilizan algunas por defecto. Se deben gestionar al menos tres variables.

Ejemplo de lanzamiento para un servidor desde la línea de comandos (asumiendo que se ha generado un JAR o se ejecuta vía Maven exec plugin):

```
java -cp <classpath_con_gson_y_clases> servidor.Server
S1 5001 temperature C humidity % pressure hPa
```

O, si se usa el plugin `exec-maven-plugin`:

```
mvn exec:java -Dexec.mainClass="servidor.Server"
-Dexec.args="S1 5001 temperature C humidity % pressure hPa"
```

Para la práctica, se deben lanzar al menos tres instancias de servidor, cada una con un ID y puerto de control únicos:

- Servidor 1: `S1 5001 temp C hum% pres hPa`
- Servidor 2: `S2 5002 temp C rad W/m2 vel_viento m/s`
- Servidor 3: `S3 5003 hum% dir_viento deg rain mm`

Estos lanzamientos se pueden configurar también directamente en un IDE como Eclipse, especificando los argumentos del programa para cada configuración de ejecución.

6.2. Cliente

Para ejecutar el cliente, se invoca la clase `cliente.Client`. No requiere argumentos de línea de comandos para su inicio.

```
java -cp <classpath_con_gson_y_clases> cliente.Client
```

O, vía Maven:

```
mvn exec:java -Dexec.mainClass="cliente.Client"
```

Al iniciarse, el cliente comienza a escuchar mensajes de difusión y presenta una consola interactiva. El usuario deberá usar el comando `TARGET_SERVER <ip><puertoDeControlDelServidor>` para especificar a qué servidor se dirigirán los comandos de control subsecuentes.