

# Tema 7: Diseño e Implementación de Servicios RPC



---

7.1. El Modelo RPC

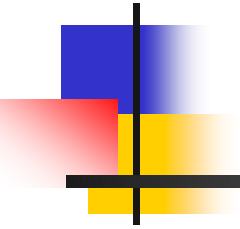
7.2. Apache Thrift

7.3. Caso de Estudio: Diseño e Implementación de Servicios con Apache Thrift

7.4. Otras Tecnologías RPC

7.5. REST vs RPC

## 7.1 El Modelo RPC





# El Modelo RPC (1)

- RPC: Remote Procedure Call (Invocación de Procedimientos Remotos)
  - Un proceso expone su funcionalidad como una serie de operaciones (procedimientos) que pueden ser invocados desde cualquier aplicación en cualquier punto de la red
  - El objetivo que persigue el modelo RPC es que para el programador sea igual de sencillo invocar una operación de una aplicación remota que invocar una operación de una librería local
  - Se basa en generar automáticamente código que se encarga de abstraer al programador de los detalles de crear y parsear mensajes, y de enviarlos/recibirlos por la red
- Primera implementación popular fue Sun RPC (80's)
- Desde entonces ha habido múltiples implementaciones: RMI, Corba, SOAP, Apache Thrift, gRPC, etc.



# El Modelo RPC (2)

## ■ RPC: Modo de Funcionamiento

- El programador de la aplicación servidora
  - Modela la funcionalidad ofrecida a través de un conjunto de operaciones que pueden ser invocadas por los clientes
    - E.g. parte de la funcionalidad de la capa modelo del tema 3
  - Describe las operaciones expuestas en un *fichero de definición* utilizando un lenguaje especial. Para cada operación se indica
    - Nombre de la operación
    - Nombre y tipo de los parámetros de entrada
    - Nombre y tipo de los parámetros de salida o valores de retorno
  - Ejecuta un compilador especial del framework RPC que recibe como entrada el fichero de definición y genera automáticamente el *skeleton*, formado por
    - Un código fuente “plantilla” con un procedimiento vacío por cada operación, o una interfaz con las operaciones
    - Una serie de librerías que se ocuparán de la comunicación con los clientes
  - Implementa las operaciones del código fuente plantilla o la interfaz



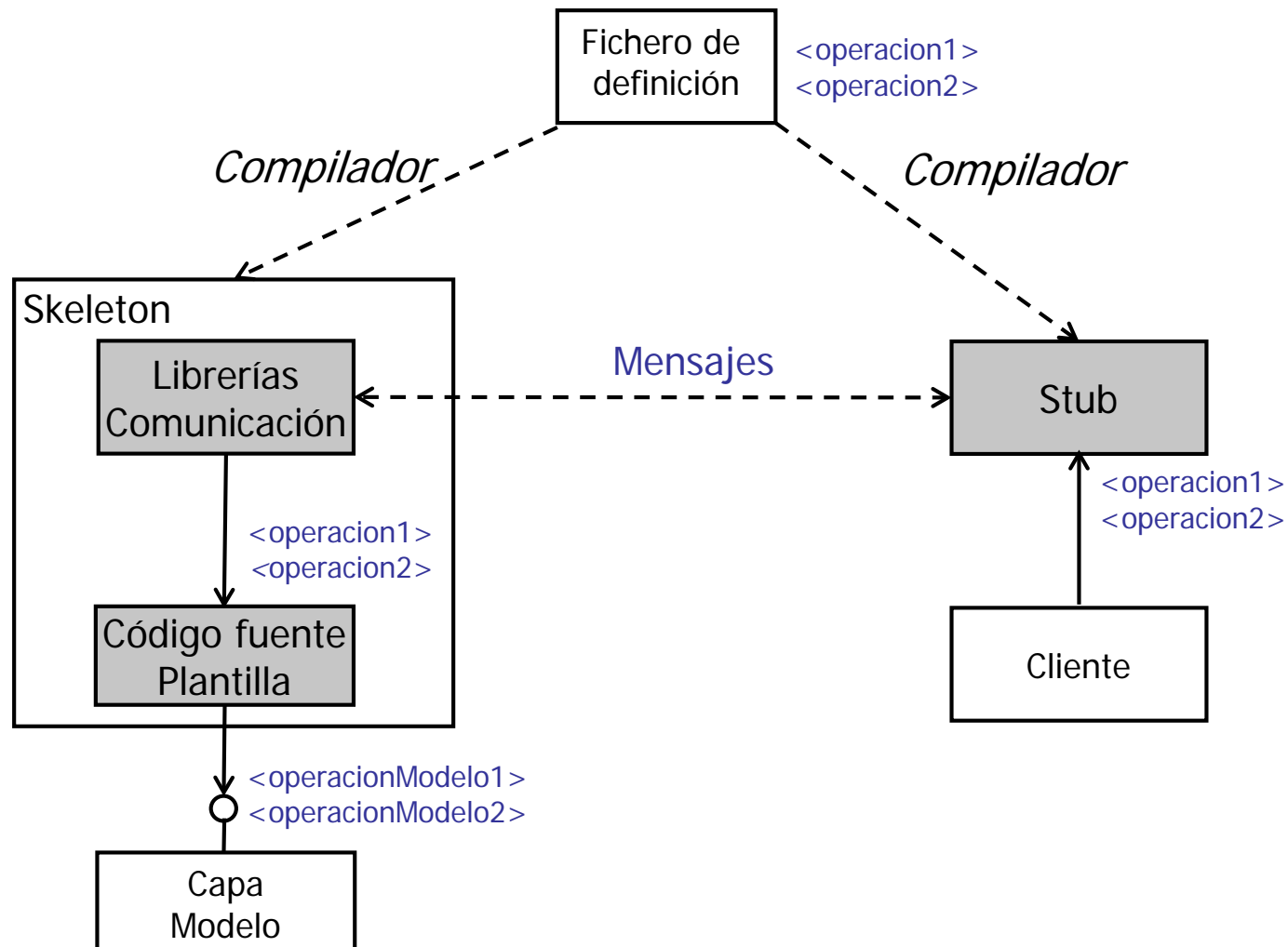
# El Modelo RPC (3)

## ■ RPC: Modo de Funcionamiento

- El programador de la aplicación cliente
  - Ejecuta el compilador especial del framework RPC pasándole como entrada el fichero de definición
    - Genera automáticamente una librería local llamada *stub*
    - El *stub* ofrece al programador del cliente operaciones con la misma firma que las indicadas en el fichero de definición
    - Cada operación del *stub*, al ser invocada, se ocupa de los detalles de invocar la operación correspondiente en el servicio y obtener la respuesta
  - Programa su aplicación cliente normalmente, invocando a la operación correspondiente del *stub* cada vez que necesita invocar una operación remota

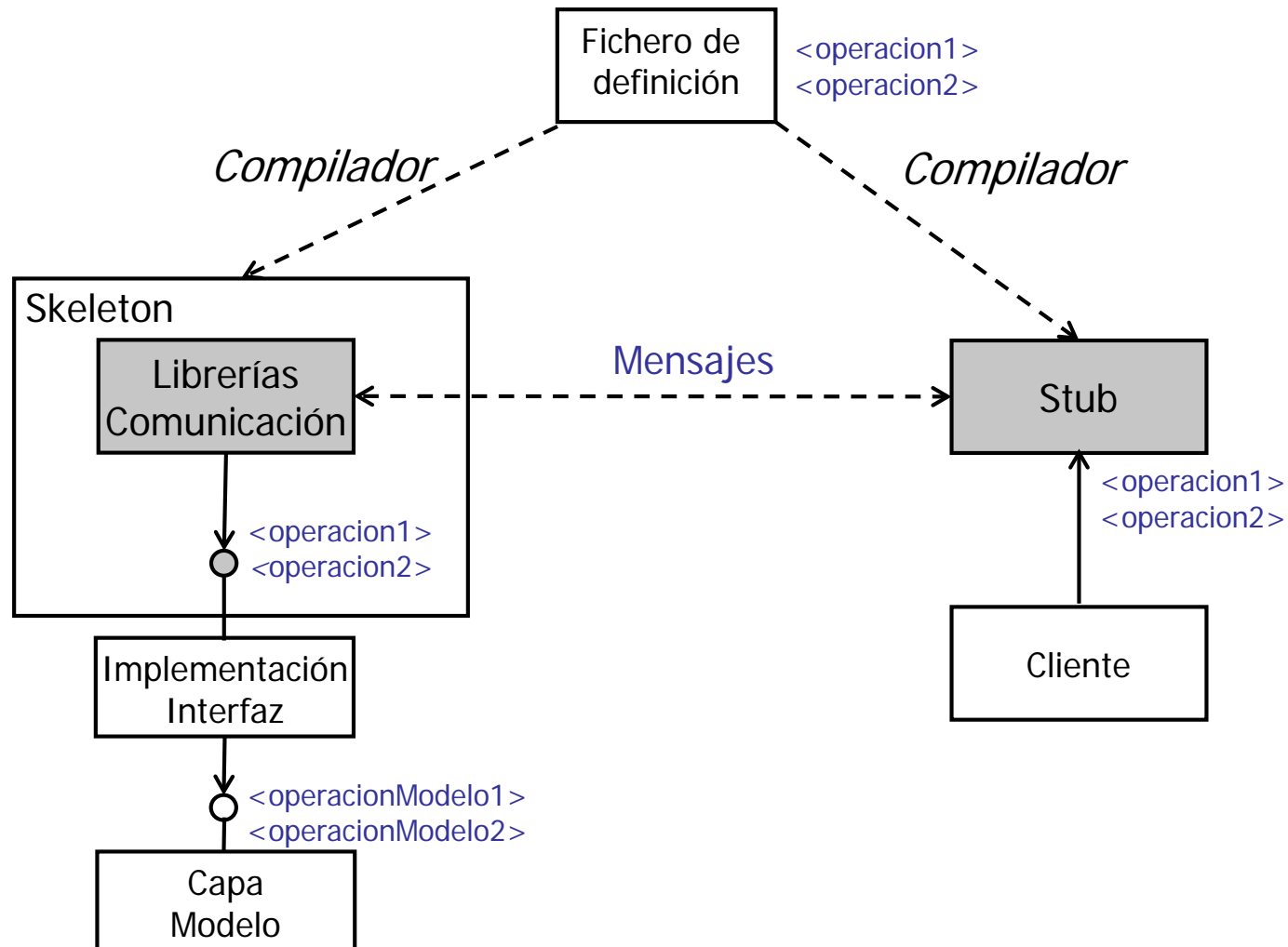
# El Modelo RPC (4)

- RPC: Modo de Funcionamiento (programa plantilla)



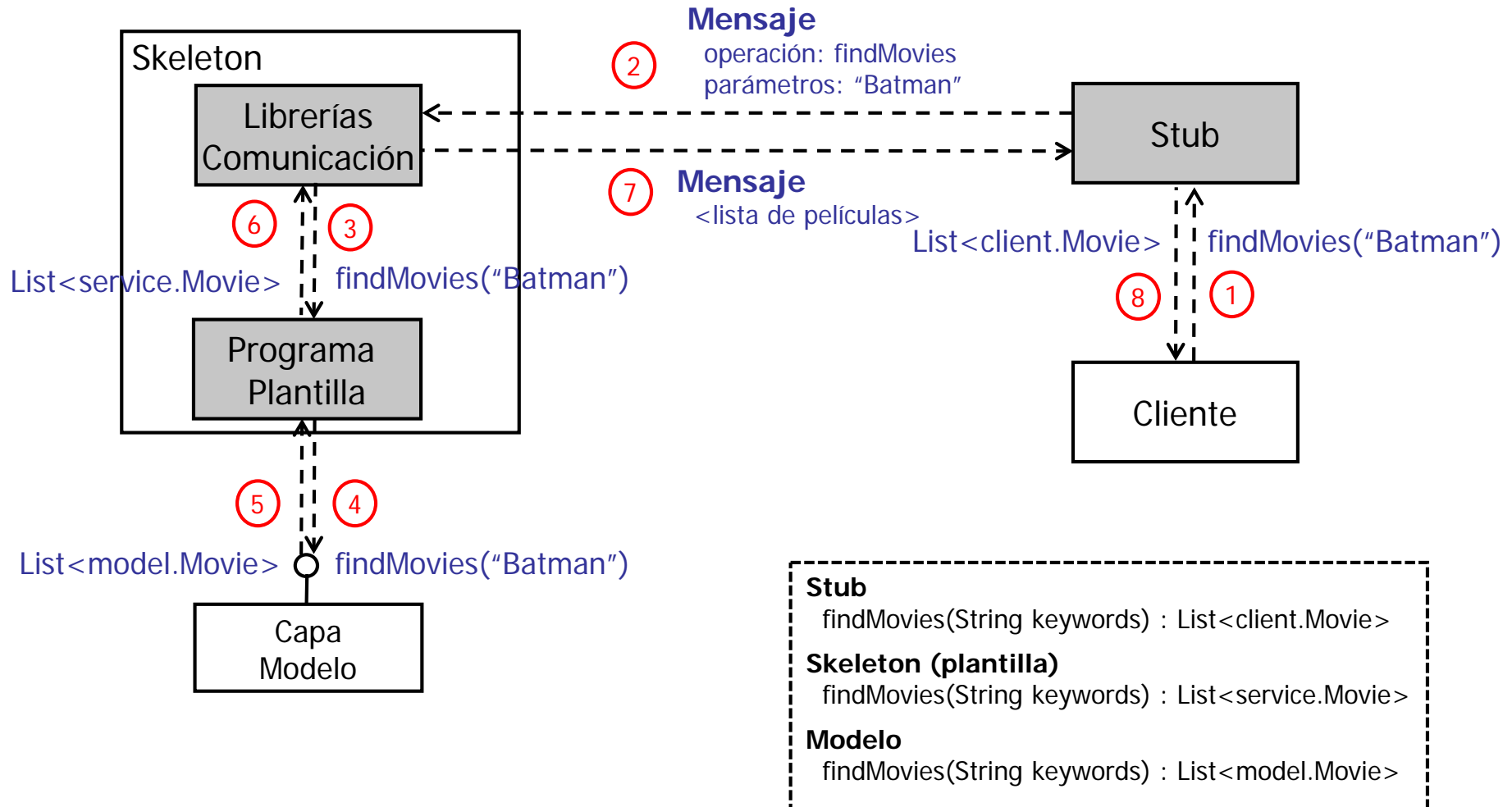
# El Modelo RPC (5)

## RPC: Modo de Funcionamiento (interfaz)



# El Modelo RPC (6)

## RPC: Flujo de una invocación remota (ejemplo)







# El Modelo RPC (7)

## ■ RPC: Flujo de una invocación remota

1. La aplicación cliente invoca una operación del *stub* pasándole los parámetros necesarios
2. El *stub*
  - Genera un mensaje encapsulando el nombre de la operación y los parámetros de entrada recibidos
  - Envía el mensaje por la red al servidor y espera la respuesta
3. El *skeleton*, cuando recibe el mensaje del *stub*, comprueba en el mensaje qué operación se desea invocar, extrae los parámetros de entrada, e invoca la operación correspondiente del código fuente plantilla (o de la implementación de la interfaz)
4. La implementación del código fuente plantilla (o de la interfaz) convierte los parámetros recibidos a los tipos adecuados, por ejemplo, para invocar una operación de una capa Modelo



# El Modelo RPC (8)

## ■ RPC: Flujo de una invocación remota (cont.)

5. La operación de la capa Modelo se ejecuta y devuelve un resultado
6. La implementación del código fuente plantilla (o de la interfaz) convierte el resultado de la capa Modelo a los tipos con los que trabajan las operaciones del código fuente plantilla (o de la interfaz) y lo devuelve
7. El *skeleton*
  - Genera un mensaje encapsulando el resultado recibido
  - Envía el mensaje por la red al cliente
8. El *stub*
  - Al recibir la respuesta del servidor, extrae de la misma el resultado y se lo devuelve al cliente
  - El programa cliente recibe la respuesta a su invocación y continua su ejecución



# El Modelo RPC (9)

---

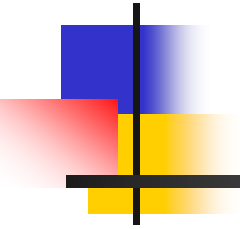
- Existe un compilador del framework RPC para cada lenguaje de programación soportado
- Cada compilador genera los *stubs* y los *skeletons* usando la sintaxis y tipos de datos del lenguaje objetivo
- De esta manera, las aplicaciones cliente y servidora pueden escribirse en lenguajes diferentes



# El Modelo RPC (y 10)

- El Modelo RPC ha sido muy exitoso
  - Uso muy intuitivo para un programador
  - Transparencia de la red para los programadores
- Sin embargo, ha sido criticado
  - En **algunas implementaciones** alto acoplamiento cliente – servidor
    - Cambios en la interfaz obligan a regenerar el stub
      - Ejemplo: renombrar un campo de salida, incluso si el cliente no lo usa
    - El servidor puede ser una aplicación autónoma que no está bajo nuestro control
    - Este problema **no existe en las implementaciones modernas** del paradigma RPC

## 7.2 Apache Thrift



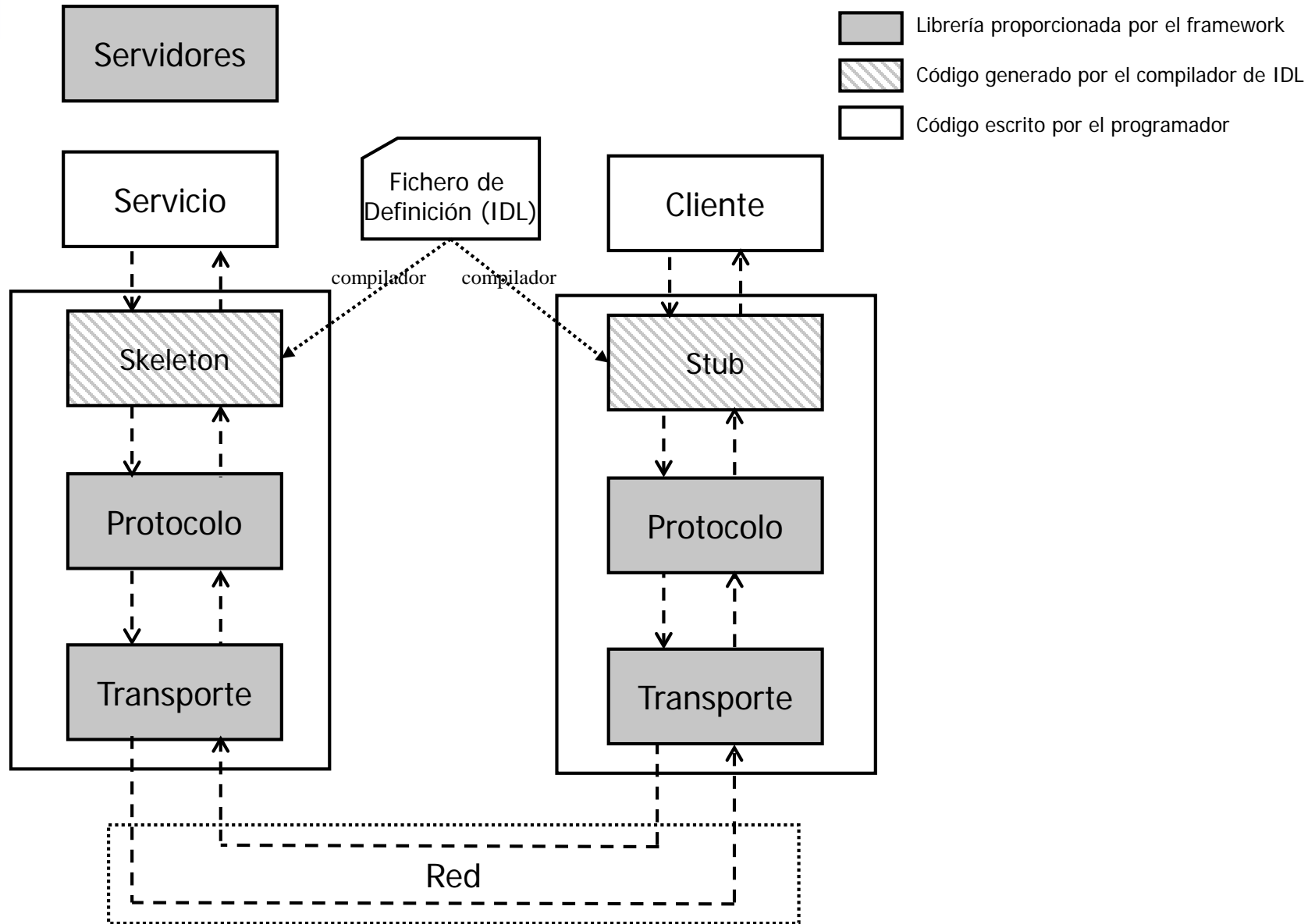


# Introducción (1)

---

- Framework que sigue el enfoque RPC
- Permite diseñar y construir servicios y clientes remotos interoperables entre diferentes lenguajes de programación y plataformas
- Disponible en más de 20 lenguajes
- Desarrollado inicialmente por Facebook
- Es un proyecto Apache *open source*
  - <https://thrift.apache.org>
- Soporta tanto un modelo síncrono como asíncrono
  - Nos centraremos en el síncrono
- Internamente está construido siguiendo una arquitectura en capas
  - Es extensible: es posible proporcionar implementaciones propias para algunas capas

# Arquitectura (1)





# Arquitectura (y 2)

- Cada capa utiliza servicios que le proporcionan la capas inferiores, que pueden estar implementados de diferentes maneras
- Podemos distinguir las siguientes capas
  - Librería de transporte
  - Librería de protocolos de serialización
    - Serialización: proceso para convertir los datos que se envían en una petición/respuesta a un formato concreto (e.g. binario, JSON, XML, etc.)
  - *Skeleton y Stub*
    - **NOTA:** En la terminología de Apache Thrift al *skeleton* se le suele llamar *stub* (igual que en la parte cliente), pero en esta asignatura nos referiremos a él como *skeleton*
  - Código del cliente o servidor
  - Librería de servidores





# Librería de Transporte

---

- Capa encargada de enviar peticiones/respuestas por la red (por ejemplo utilizando sockets TCP o HTTP)
- Todas las implementaciones heredan directa o indirectamente de **`org.apache.thrift.transport.TTransport`**
- Es extensible (pueden proporcionarse implementaciones a medida)



# Librería de Protocolos

---

- Capa encargada de la serialización y deserialización de la información que se envía en cada petición y respuesta
- El framework proporciona diferentes implementaciones de esta capa, que heredan de **`org.apache.thrift.protocol.TProtocol`**
- Ejemplos
  - Protocolo binario (**`TBinaryProtocol`**)
    - Los datos se envían serializados como bytes
  - Protocolo JSON (**`TJSONProtocol`**)
    - Los datos se transmiten como texto en formato JSON
- Es extensible (pueden proporcionarse implementaciones a medida)



# Librería de Servidores

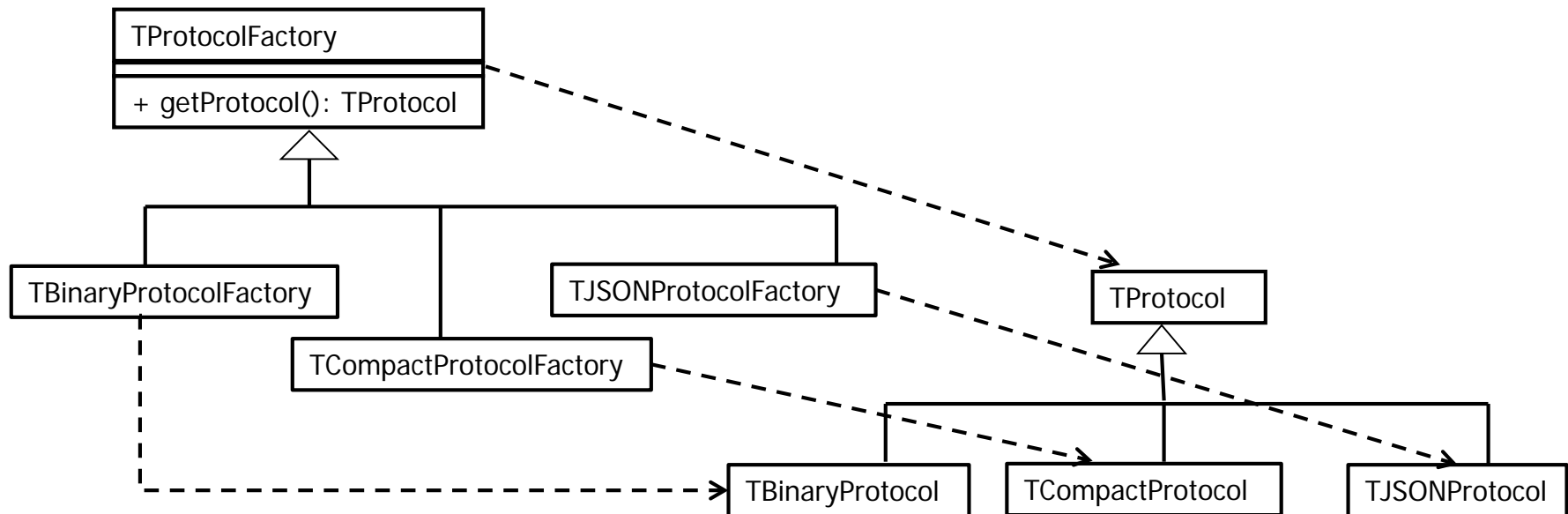
---

- En el contexto de Apache Thrift un servidor es un programa específicamente diseñado para albergar uno o más servicios Apache Thrift
- Esta librería proporciona clases que permiten implementar diferentes tipos de servidores según las necesidades concretas de la aplicación y las capacidades de cada lenguaje
  - Servidores monohilo o multihilo con diferentes modelos de concurrencia
  - En el caso de Java, permite que los servicios se ejecuten dentro de un servidor de aplicaciones, utilizando en ese caso HTTP como transporte
    - El servicio puede acceder a los recursos del servidor de aplicaciones (e.g. Datasources)
    - Usa su modelo de ejecución (multi-thread)

# Protocolo - Uso de Factorías

Los servidores Apache Thrift crean un nuevo objeto **TProtocol** para cada conexión de un cliente aceptada

- Para crear el objeto usan una implementación de **TProtocolFactory**
- Debe proporcionársele una factoría que genere objetos **TProtocol** correspondientes al protocolo deseado
- Los servidores siempre trabajan con la factoría abstracta (**TProtocolFactory**) y la interfaz que representa a los protocolos (**TProtocol**)



- Apache Thrift Interface Definition Language (IDL)
  - Por convención se utilizan ficheros con extensión **.thrift**
- Permite definir
  - Servicios: conjuntos de operaciones relacionadas entre sí
  - Tipos: información intercambiada al invocar las operaciones de los servicios (parámetros, valores de retorno y excepciones)
- Todos los elementos deben tener un nombre
  - El nombre debe ser único dentro de su “contexto”
  - Sensible a mayúsculas/minúsculas
  - No puede ser una de las palabras reservadas del lenguaje (e.g. **i8**, **struct**, **exception**, etc.)
  - Además, deben tenerse en cuenta las palabras reservadas de los lenguajes a los que se vaya a compilar (e.g. si se va a generar código Java, no usar **if**, **else**, **class**, etc.)



# IDL - Espacios de Nombres

---

- Por defecto, en la mayoría de lenguajes, el compilador de IDL genera los tipos y servicios en el “contexto global”
  - Los elementos generados podrían colisionar con otros elementos
- Los espacios de nombres permiten definir contextos dentro de los cuáles deben generarse los elementos
  - Su uso es una buena práctica
- Se utiliza la palabra reservada **namespace** seguida del “alcance” (indica a qué lenguaje aplica) y el nombre del espacio de nombres
  - En Java, por ejemplo, sirve para especificar el paquete donde se generarán las clases (el compilador generará además la estructura de directorios oportuna)

```
namespace java es.udc.ws.movies.thrift
```



# IDL - Tipos

---

- Tipos
  - Tipos base
  - Tipos contenedores
  - Tipos definidos por el usuario



# IDL - Tipos Base

- Soporta un conjunto mínimo de tipos base que se encuentran en prácticamente cualquier lenguaje

Palabra clave	Descripción	Tipo Java
<b>binary</b>	Array de bytes	<code>java.nio.ByteBuffer</code>
<b>bool</b>	Booleano: <code>true</code>   <code>false</code>	<code>boolean</code>
<b>double</b>	Punto flotante de doble precisión	<code>double</code>
<b>i8 (byte)</b>	Entero de 8 bits	<code>byte</code>
<b>i16</b>	Entero de 16 bits	<code>short</code>
<b>i32</b>	Entero de 32 bits	<code>int</code>
<b>i64</b>	Entero de 64 bits	<code>long</code>
<b>string</b>	Cadena de caracteres	<code>java.lang.String</code>
<b>void</b>	Para indicar que una operación de un servicio no devuelve nada	<code>void</code>



# IDL - Tipos Contenedores

- Representan estructuras de datos que contienen otros tipos, comunes en muchos lenguajes (listas, *mapas*, conjuntos)
- Se usa la sintaxis `< >` para indicar el tipo contenido (similar a *Generics* en Java)
- Listas

Palabra clave	Descripción	Tipo Java
<code>list&lt;&gt;</code>	Lista (ordenada) de cero o más elementos del tipo indicado	<code>ArrayList&lt;&gt;</code>

- Ejemplos
  - `list<double>`
  - `list<string>`



# IDL – struct (1)

- Permite crear tipos definidos por el usuario que pueden usarse en los mismos lugares que los tipos base y contenedores
- Se definen con la palabra reservada **struct**
  - Tienen un nombre y contienen un conjunto de campos

```
struct User {  
    1: string userName  
    2: i8 age  
}
```

- El compilador genera un tipo por cada **struct**
  - En Java genera una clase cuyo nombre coincide con el de la **struct**, y con un atributo por cada campo con su *getter* y *setter* correspondientes



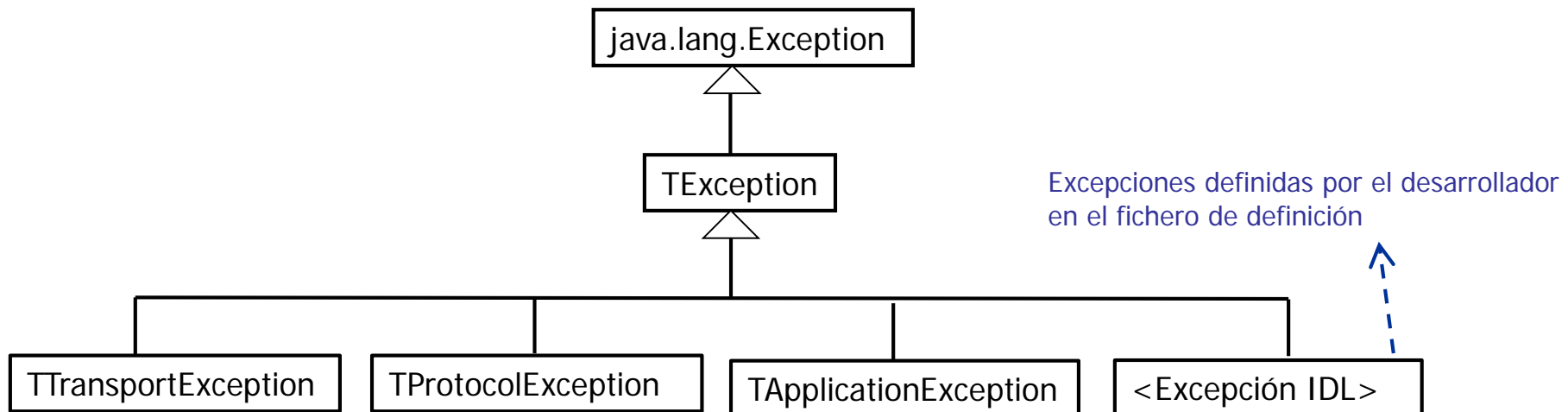
# IDL – struct (y 2)

---

- Cada campo tiene
  - Un identificador o clave
    - Entero positivo de 16 bits
    - Debe ser diferente para cada campo de la lista
    - Es **usado para identificar el campo**
  - Un tipo
  - Un nombre
- Es posible utilizar la coma (,) o punto y coma (;) como separador o “terminador” de cada campo, pero no es necesario

# Modelo de Gestión de Errores

- Apache Thrift adopta el modelo de excepciones como su modelo abstracto para gestionar errores
- Jerarquía de excepciones en Java



# IDL - exception

- Se definen igual que los **struct** pero usando la palabra reservada **exception**
- El compilador de IDL genera código que integra la excepción en la jerarquía de excepciones del lenguaje
  - En el caso de Java, generando una clase que extiende de **org.apache.thrift.TException**, que a su vez extiende de **java.lang.Exception**
- Se pueden declarar como “lanzables” por los métodos de los servicios usando la cláusula **throws**

```
exception InstanceNotFoundException {  
    1: string instanceId  
    2: string instanceType  
}
```



# IDL – Servicios (1)

---

- Un servicio define un conjunto de funciones (o métodos) relacionadas
  - Se declaran con la palabra reservada **service**
- Las funciones tienen
  - Un tipo de retorno: cualquier tipo excepto **exception**
  - Un nombre
  - Una lista de parámetros
    - Se especifican de forma similar a los campos de un **struct**
    - Cada parámetro tiene un identificador, un nombre y un tipo
  - Opcionalmente, pueden declarar una lista de excepciones con la palabra reserva **throws**
    - Se especifican de forma similar a los campos de un **struct**
    - Cada excepción tiene un identificador, un tipo (que debe ser uno de los tipos **exception** declarados) y un nombre
- Entre funciones puede usarse coma, punto y coma o nada

# IDL – Servicios (y 2)

- Ejemplo (fragmento del caso de estudio que se verá a continuación)

```
service ThriftMovieService {  
  
    ...  
    list<ThriftMovieDto> findMovies(1: string keywords)  
  
    i64 buyMovie(1: i64 movieId, 2: string userId, 3: string creditCardNumber)  
        throws (1: ThriftInputValidationException e,  
                2: ThriftInstanceNotFoundException ee)  
    ...  
}
```

- Nota: **ThriftMovieDto** se habrán definido como **struct**, y **ThriftInputValidationException** y **ThriftInstanceNotFoundException** como **exception**



# Evolución – Tipos de Usuario

- Es posible cambiar el nombre de un campo
  - Lo que identifica al campo es **el identificador y el tipo** (es lo que se serializa junto con el valor del campo)
- Usando el nivel de obligatoriedad por defecto, es posible añadir o eliminar campos
  - Si se recibe una petición/respuesta con más campos, estos se ignoran
  - Si se recibe una petición/respuesta con menos campos, a estos se les asigna un valor por defecto (definido en el IDL o el asignado por el lenguaje en cuestión)
  - Si se elimina un campo, su identificador no debe reutilizarse para otro campo
    - Es una buena práctica dejarlo comentado
- Si cambia el tipo de un campo, será ignorado por los programas antiguos (aunque conserve su identificador)





# Evolución - Servicios

## ■ Parámetros

- Es posible cambiar el nombre (siempre que se mantengan **el identificador y el tipo**)
- Es posible añadir o eliminar parámetros
  - Si se recibe una petición sin algún parámetro, se le asigna un valor por defecto (definido en el IDL o el asignado por el lenguaje en cuestión)
  - Si se recibe una petición con parámetros de más, se ignoran

## ■ Funciones

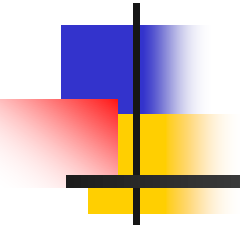
- Añadir una función
  - Los clientes antiguos pueden invocar a los servicios nuevos
  - Los clientes nuevos recibirán una excepción si invocan la nueva función sobre un servicio antiguo
- Eliminar una función
  - Los clientes nuevos pueden invocar a los servicios antiguos
  - Los clientes antiguos recibirán una excepción si invocan la función eliminada sobre un servicio nuevo



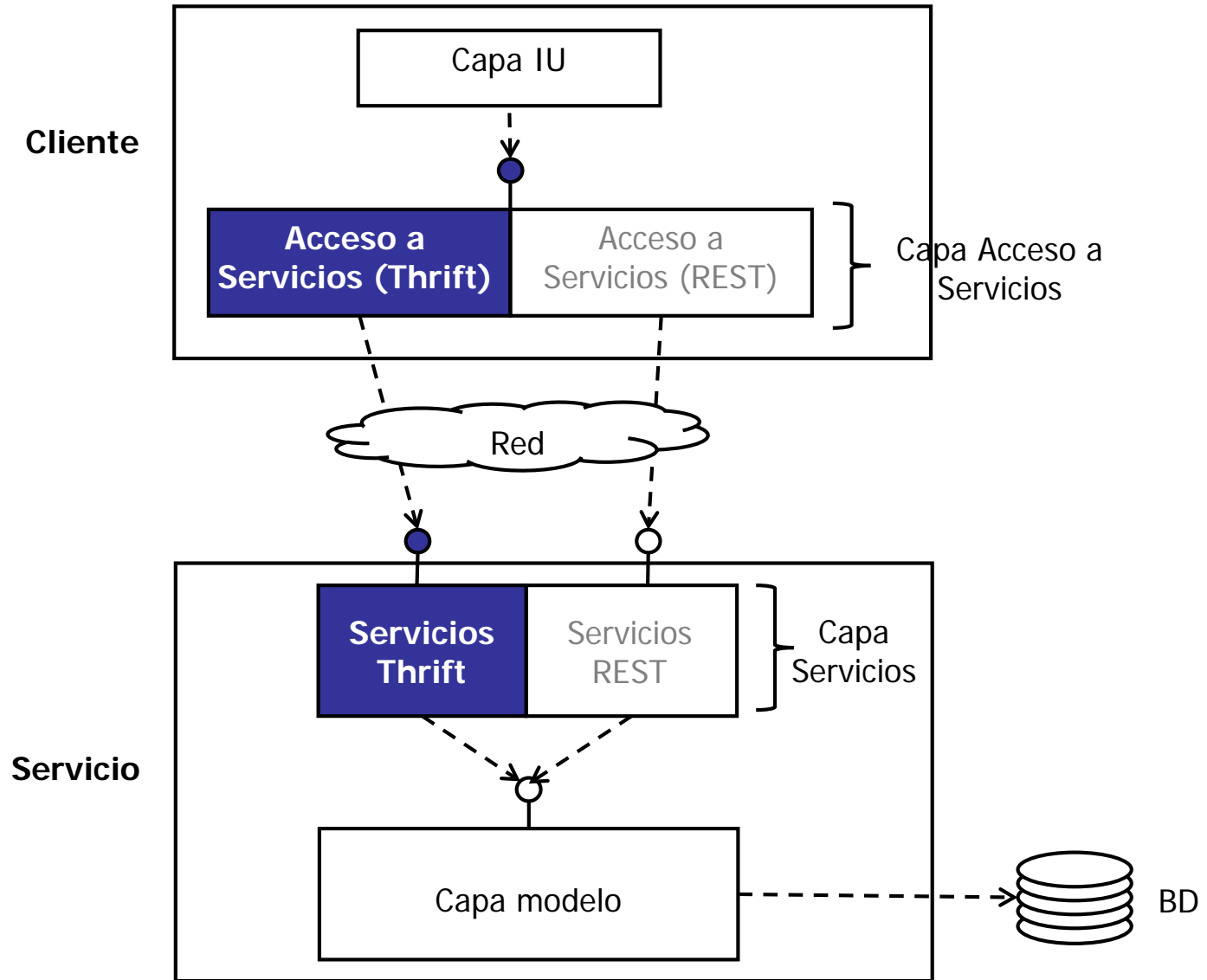
# Compilador

- El compilador de Thrift es un ejecutable que recibe por línea de comandos
  - El nombre del archivo **.thrift**
  - El lenguaje en el que debe generar el *stub* y el *skeleton*
    - Siempre genera ambos (no tiene opción para indicarle que genere solamente el *stub* o el *skeleton*)
- Nosotros lo usaremos a través de un plugin para Maven
  - Lo configuramos para ejecutar el *goal* **compile** durante la fase **generate-sources** (ver el **pom.xml** de **ws-movies/ws-movies-thrift**)
  - Es necesario tener el ejecutable del compilador de Thrift en el PATH del sistema operativo
  - Genera el código Java en **target/generated-sources/thrift**

## 7.3 Caso de Estudio: Diseño e Implementación de un Servicio con Apache Thrift



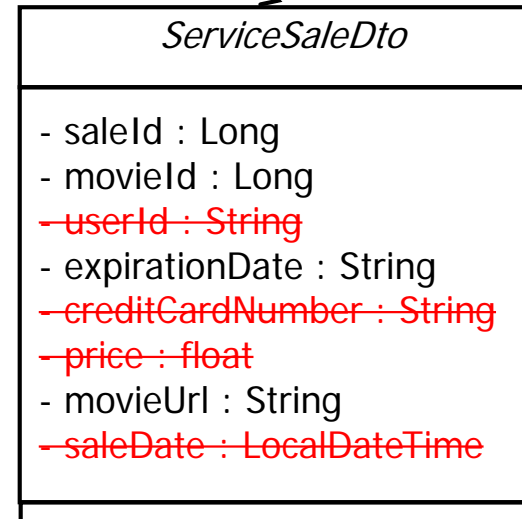
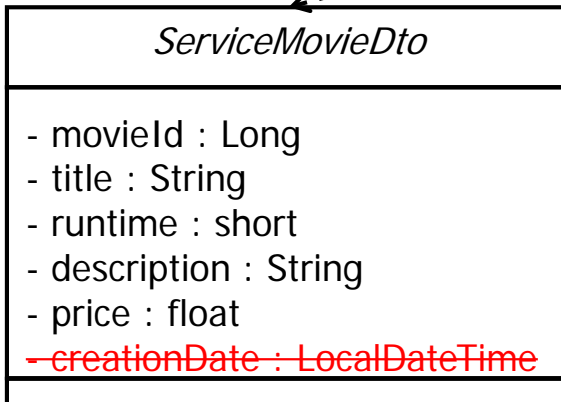
# Arquitectura General de Movies



# Interfaz de la Capa Servicios

## *MovieService (Visión ofrecida por Capa Servicios)*

- + addMovie(movie: ServiceMovieDto) : ServiceMovieDto
- + updateMovie(movie : ServiceMovieDto) : void
- + removeMovie(movieId: Long) : void
- ~~+ findMovie(movieId : Long) : ServiceMovieDto~~
- + findMovies(keywords : String) : List<ServiceMovieDto>
- + buyMovie(movieId : Long, userId : Long, creditCardNumber : String) : Long
- + findSale(saleId : Long) : ServiceSaleDto





# IDL (1)

- *Namespace y structs* correspondientes a los DTOs

```
namespace java es.udc.ws.movies.thrift
```

```
struct ThriftMovieDto {  
    1: i64 movieId;  
    2: string title;  
    3: i16 runtime;  
    4: string description;  
    5: double price  
}
```

```
struct ThriftSaleDto {  
    1: i64 saleId;  
    2: i64 movieId;  
    3: string expirationDate;  
    4: string movieUrl;  
}
```

## ■ Excepciones

```
exception ThriftInputValidationException {  
    1: string message  
}
```

```
exception ThriftInstanceNotFoundException {  
    1: string instanceId  
    2: string instanceType  
}
```

```
exception ThriftSaleExpirationException {  
    1: i64 saleId;  
    2: string expirationDate;  
}
```

```
exception ThriftMovieNotRemovableException {  
    1: i64 movieId;  
}
```

# IDL (y 3)

## ■ Servicio

```
service ThriftMovieService {

    ThriftMovieDto addMovie(1: ThriftMovieDto movieDto)
        throws (1: ThriftInputValidationException e)

    void updateMovie(1: ThriftMovieDto movieDto)
        throws (1: ThriftInputValidationException e,
                2: ThriftInstanceNotFoundException ee)

    void removeMovie(1: i64 movieId)
        throws (1: ThriftInstanceNotFoundException e,
                2: ThriftMovieNotRemovableException ee)

    list<ThriftMovieDto> findMovies(1: string keywords)

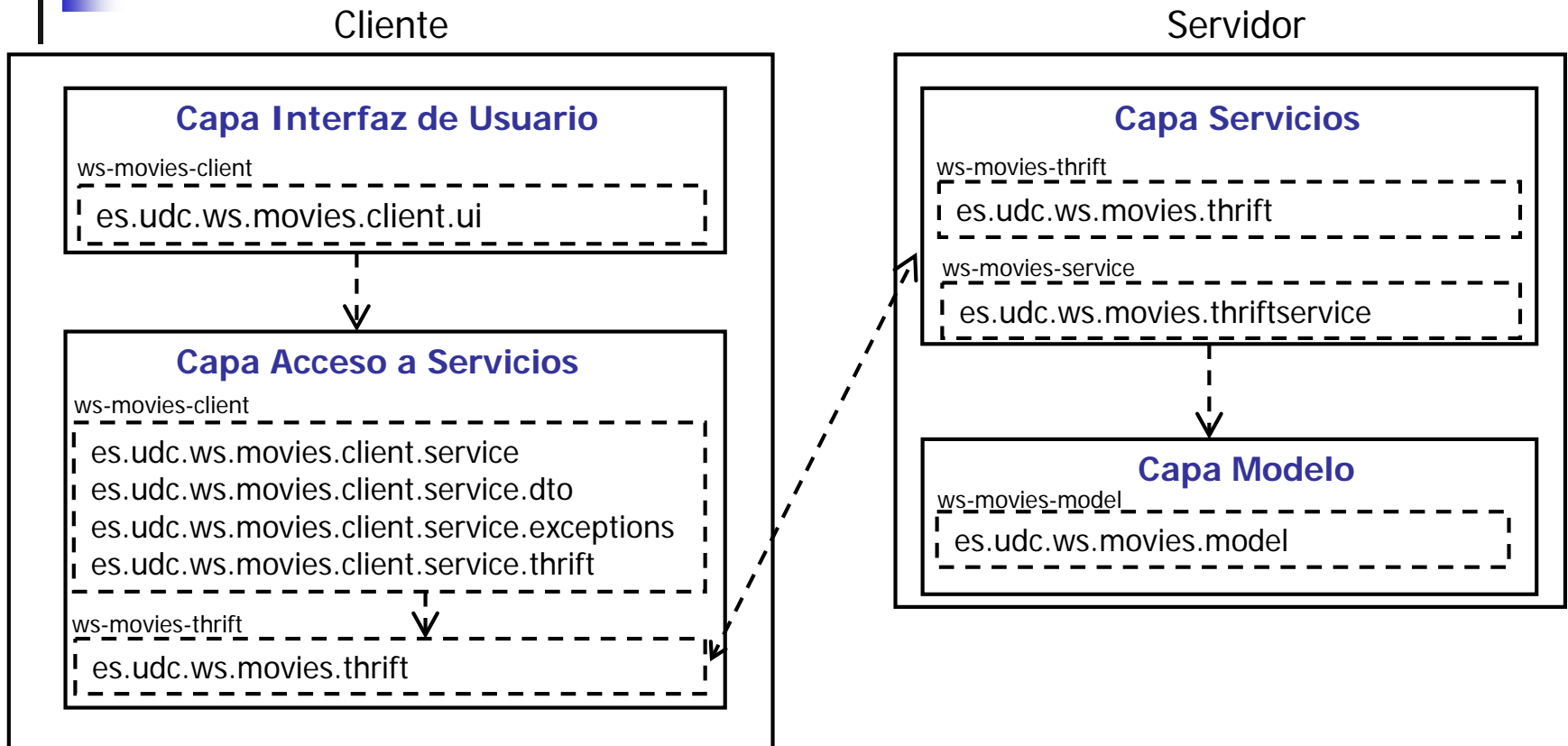
    i64 buyMovie(1: i64 movieId, 2: string userId, 3: string creditCardNumber)
        throws (1: ThriftInputValidationException e,
                2: ThriftInstanceNotFoundException ee)

    ThriftSaleDto findSale(1: i64 saleId)
        throws (1: ThriftInstanceNotFoundException e,
                2: ThriftSaleExpirationException ee)

}
```



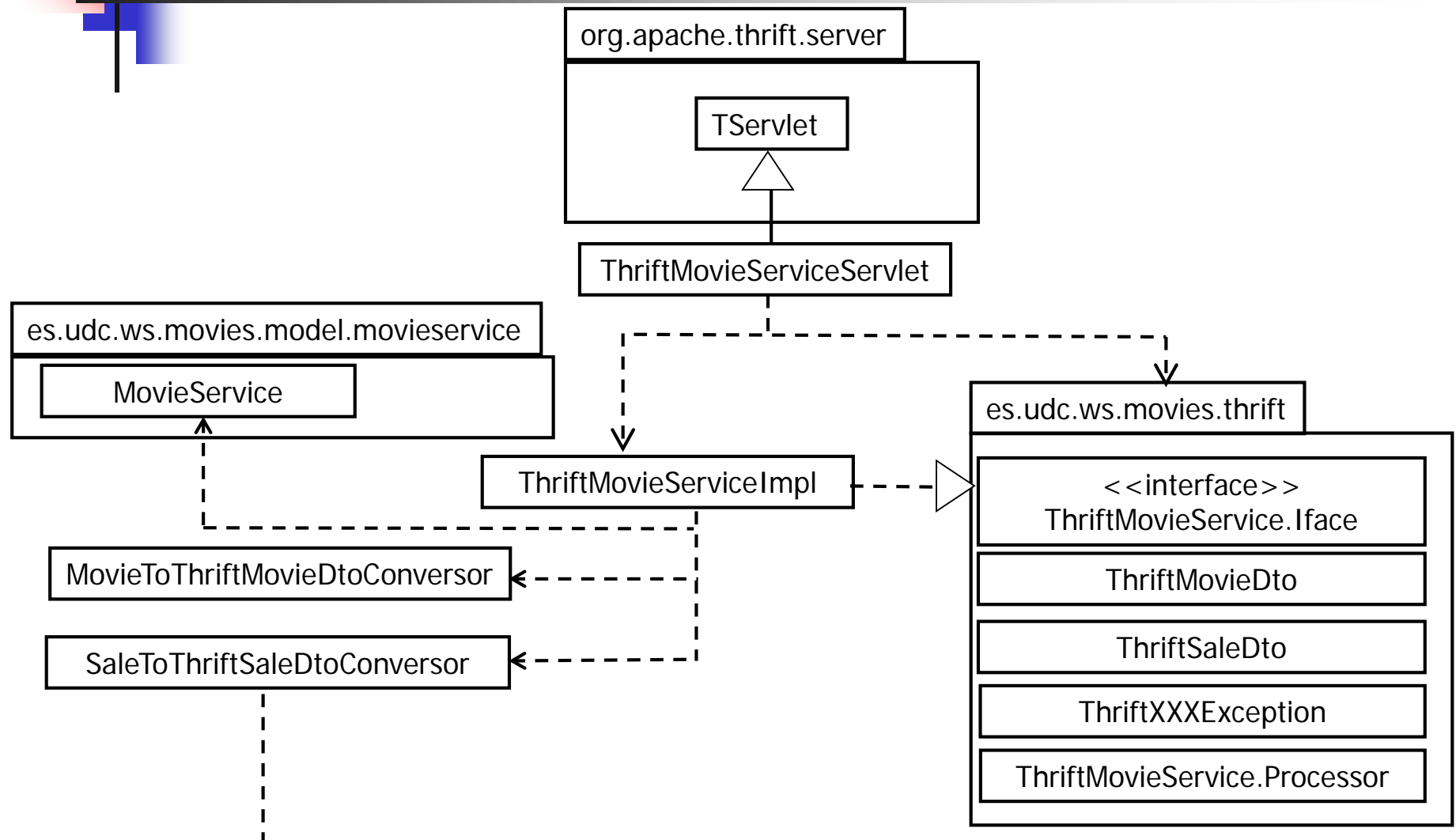
# Principales Paquetes por Capas



- **es.udc.ws.movies.thrift**: contiene las clases e interfaces generadas por el compilador del IDL a Java
  - El fichero de definición está en **ws-movies-thrift/src/main/thrift**
- La implementación del servicio (**ThriftMovieServiceImpl**) está en **es.udc.ws.movies.thriftservice**

# Capa Servicios

## [es.udc.ws.movies.thriftservice]



**MovieToThriftMovieDtoConversor** y **SaleToThriftSaleDtoConversor** realizan conversiones entre los objetos **Movie** y **Sale** (usados por el Modelo), y **ThriftMovieDto** y **ThriftSaleDto** (clases generadas por el compilador del IDL de Thrift a Java)

```
public class ThriftMovieService {  
    public interface Iface {  
        public ThriftMovieDto addMovie(ThriftMovieDto movieDto) throws  
            ThriftInputValidationException, org.apache.thrift.TException;  
  
        public void updateMovie(ThriftMovieDto movieDto) throws  
            ThriftInputValidationException, ThriftInstanceNotFoundException,  
            org.apache.thrift.TException;  
  
        public void removeMovie(long movieId) throws  
            ThriftInstanceNotFoundException, ThriftMovieNotRemovableException,  
            org.apache.thrift.TException;  
  
        public java.util.List<ThriftMovieDto> findMovies(java.lang.String  
            keywords) throws org.apache.thrift.TException;  
  
        public long buyMovie(long movieId, java.lang.String userId,  
            java.lang.String creditCardNumber) throws  
            ThriftInputValidationException, ThriftInstanceNotFoundException,  
            org.apache.thrift.TException;  
  
        public ThriftSaleDto findSale(long saleId) throws  
            ThriftInstanceNotFoundException, ThriftSaleExpirationException,  
            org.apache.thrift.TException;  
    }  
    ...  
}
```

```
package es.udc.ws.movies.thriftservice;
...
import es.udc.ws.movies.thrift.*;
...

public class ThriftMovieServiceImpl implements ThriftMovieService.Iface {

    @Override
    public ThriftMovieDto addMovie(ThriftMovieDto movieDto)
        throws ThriftInputValidationException {

        Movie movie = MovieToThriftMovieDtoConversor.toMovie(movieDto);

        try {

            Movie addedMovie = MovieServiceFactory.getService().addMovie(movie);
            return MovieToThriftMovieDtoConversor.toThriftMovieDto(addedMovie);

        } catch (InputValidationException e) {
            throw new ThriftInputValidationException(e.getMessage());
        }

    }
}
```

```
@Override
public List<ThriftMovieDto> findMovies(String keywords) {

    List<Movie> movies =
        MovieServiceFactory.getService().findMovies(keywords);

    return MovieToThriftMovieDtoConversor.toThriftMovieDtos(movies);
}
```

@Override

```
public ThriftSaleDto findSale(long saleId) throws
    ThriftInstanceNotFoundException, ThriftSaleExpirationException {

    try {

        Sale sale = MovieServiceFactory.getService().findSale(saleId);
        return SaleToThriftSaleDtoConversor.toThriftSaleDto(sale);

    } catch (InstanceNotFoundException e) {
        throw new ThriftInstanceNotFoundException(
            e.getInstanceId().toString(),
            e.getInstanceType().substring(
                e.getInstanceType().lastIndexOf('.') + 1));
    } catch (SaleExpirationException e) {
        throw new ThriftSaleExpirationException(e.getSaleId(),
            e.getExpirationDate().toString());
    }

}

// updateMovie, removeMovie, buyMovie
...
}
```

```
package es.udc.ws.movies.thriftservice;

import es.udc.ws.movies.thrift.ThriftMovieService;
import org.apache.thrift.TProcessor;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocolFactory;
import org.apache.thrift.server.TServlet;

public class ThriftMovieServiceServlet extends TServlet {

    public ThriftMovieServiceServlet() {
        super(createProcessor(), createProtocolFactory());
    }

    private static TProcessor createProcessor() {

        return new ThriftMovieService.Processor<ThriftMovieService.Iface>(
            new ThriftMovieServiceImpl());

    }

    private static TProtocolFactory createProtocolFactory() {
        return new TBinaryProtocol.Factory();
    }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

...

<!-- Thrift service -->
<servlet>
    <servlet-name>ThriftMovieServiceServlet</servlet-name>
    <servlet-class>
        es.udc.ws.movies.thriftservice.ThriftMovieServiceServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>ThriftMovieServiceServlet</servlet-name>
    <url-pattern>/thrift/movieservice</url-pattern>
</servlet-mapping>

...
</web-app>
```





# Comentarios (1)

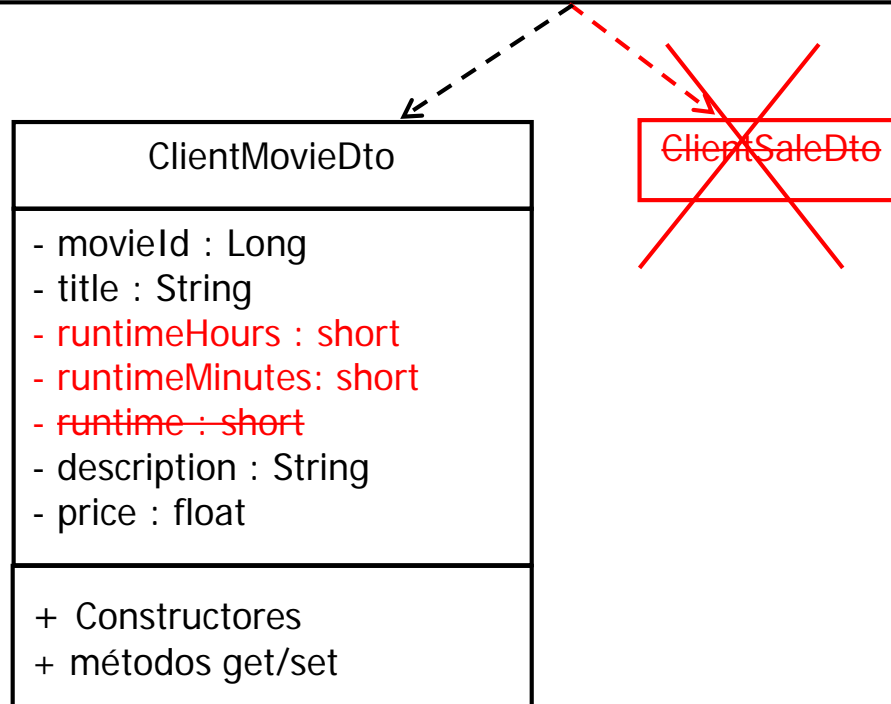
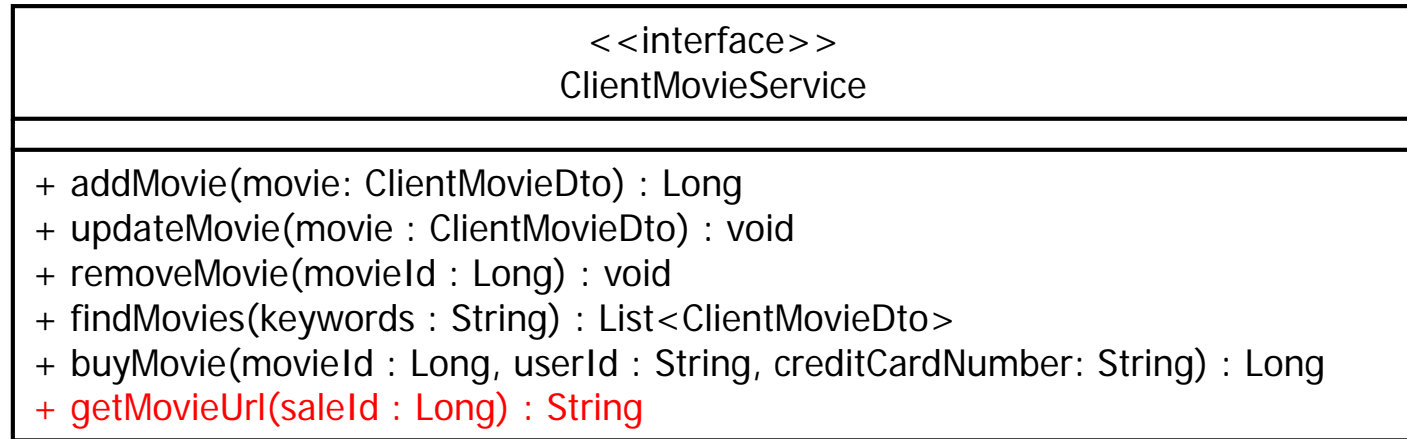
- El compilador genera, entre otras cosas
  - La interfaz del servicio
    - `ThriftMovieService.Iface`
  - Clases para los parámetros y valores de retorno de tipo struct
    - `ThriftMovieDto`
    - `ThriftSaleDto`
  - Clases para las excepciones (extienden a `TException`)
    - `ThriftInstanceNotFoundException`
    - `ThriftInputValidationException`
    - `ThriftSaleExpirationException`
  - Un procesador (`TProcessor`) para el servicio
    - `ThriftMovieService.Processor`
- `ThriftMovieServiceImpl` implementa la interfaz del servicio delegando en las operaciones correspondientes de la capa Modelo
  - Usa los conversores entre DTOs generados por el compilador y entidades de la capa Modelo
  - Convierte las excepciones de la capa Modelo a las excepciones generadas por el compilador

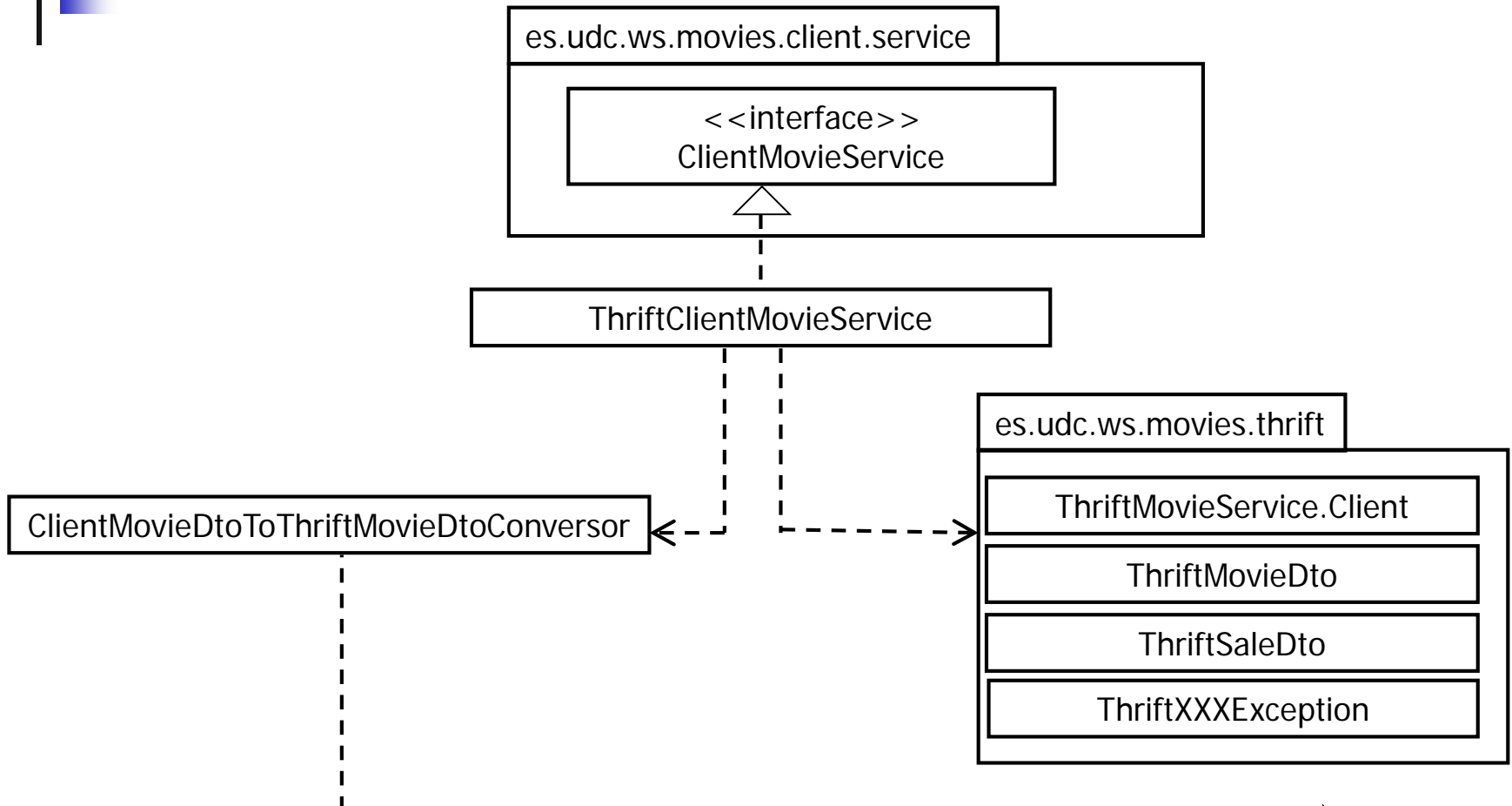


# Comentarios (y 2)


- Para integrar un servidor Thrift en un contenedor de Servlets es necesario crear un Servlet que extienda a la clase **TServlet**
  - Desde el constructor se llama al constructor de la superclase pasándole un *procesador* (**TProcessor**) para el servicio y una factoría para obtener objetos de tipo **TProtocol** acordes al protocolo que se quiera utilizar
    - Para instanciar el procesador es necesario pasarle una instancia de la clase de implementación del servicio (*handler*), en este caso **ThriftMovieServiceImpl**
    - Se utiliza una factoría que genera objetos **TBinaryProtocol**, puesto que deseamos utilizar el protocolo binario
- En el fichero **web.xml** se declara el Servlet y se asocia a la URL deseada
  - El servicio queda asociado a esa URL (será la que haya que indicar desde un cliente cuando quiera invocar el servicio)

# Interfaz de la Capa Acceso a Servicios





**ClientMovieDtoToThriftMovieDtoConversor** realiza conversiones entre objetos de tipo **ClientMovieDto** (usados por el cliente) y **ThriftMovieDto** (clase generada por el compilador del IDL de Thrift a Java)



## es.udc.ws.movies.client.service.thrift.ThriftClientMovieService (1)

### [Capa Acceso a Servicios]

---

```
package es.udc.ws.movies.client.service.thrift;

import es.udc.ws.movies.client.service.ClientMovieService;
import es.udc.ws.movies.client.service.dto.ClientMovieDto;
import es.udc.ws.movies.client.service.exceptions.ClientSaleExpirationException;
import es.udc.ws.movies.thrift.ThriftInputValidationException;
import es.udc.ws.movies.thrift.ThriftInstanceNotFoundException;
import es.udc.ws.movies.thrift.ThriftMovieService;
import es.udc.ws.movies.thrift.ThriftSaleExpirationException;
...
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.THttpClient;
import org.apache.thrift.transport.TTransport;
import org.apache.thrift.transport.TTransportException;
...

public class ThriftClientMovieService implements ClientMovieService {

    private final static String ENDPOINT_ADDRESS_PARAMETER =
        "ThriftClientMovieService.endpointAddress";

    private final static String endpointAddress =
        ConfigurationParametersManager.getParameter(ENDPOINT_ADDRESS_PARAMETER);
```

```
@Override
public Long addMovie(ClientMovieDto movie) throws InputValidationException {

    ThriftMovieService.Client client = getClient();
    TTransport transport = client.getInputProtocol().getTransport();

    try {

        transport.open();

        return client.addMovie(
            ClientMovieDtoToThriftMovieDtoConversor.toThriftMovieDto(movie)).
            getMovieId();

    } catch (ThriftInputValidationException e) {
        throw new InputValidationException(e.getMessage());
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        transport.close();
    }

}
```

```
@Override
public List<ClientMovieDto> findMovies(String keywords) {

    ThriftMovieService.Client client = getClient();
    TTransport transport = client.getInputProtocol().getTransport();

    try {

        transport.open();

        return ClientMovieDtoToThriftMovieDtoConversor.
            toClientMovieDto(client.findMovies(keywords));

    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        transport.close();
    }
}
```

## es.udc.ws.movies.client.service.thrift.ThriftClientMovieService (4)

### [Capa Acceso a Servicios]

@Override

```
public String getMovieUrl(Long saleId) throws InstanceNotFoundException,  
    ClientSaleExpirationException {
```

```
    ThriftMovieService.Client client = getClient();
```

```
    TTransport transport = client.getInputProtocol().getTransport();
```

```
    try {
```

```
        transport.open();
```

```
        return client.findSale(saleId).getMovieUrl();
```

```
    } catch (ThriftInstanceNotFoundException e) {
```

```
        throw new InstanceNotFoundException(e.getInstanceId(),  
            e.getInstanceType());
```

```
    } catch (ThriftSaleExpirationException e) {
```

```
        throw new ClientSaleExpirationException(e.getSaleId(),  
            LocalDateTime.parse(e.getExpirationDate(),  
                DateTimeFormatter.ISO_DATE_TIME));
```

```
    } catch (Exception e) {
```

```
        throw new RuntimeException(e);
```

```
    } finally {
```

```
        transport.close();
```

```
    }
```

```
}
```

```
// updateMovie, removeMovie, buyMovie
```

```
...
```

```
}
```



```
private ThriftMovieService.Client getClient() {  
  
    try {  
  
        TTransport transport = new THttpClient(endpointAddress);  
        TProtocol protocol = new TBinaryProtocol(transport);  
  
        return new ThriftMovieService.Client(protocol);  
  
    } catch (TTransportException e) {  
        throw new RuntimeException(e);  
    }  
  
}
```



# Comentarios (1)

El compilador genera como parte del *stub* la clase **ThriftMovieService.Client**, que tiene un método por cada operación remota ofrecida por el servicio

- Implementa **ThriftMovieService.Iface**
- La implementación de cada método envía una petición apropiada por la red y parsea la respuesta
- Cada vez que el cliente quiera invocar una operación remota es necesario crear una instancia de esa clase (es lo que hace el método **getClient**)
  - Para instanciarla es necesario pasarle un objeto de tipo **TProtocol**, acorde al protocolo que se quiera utilizar
    - Protocolo binario → **TBinaryProtocol**
  - Para instanciar el protocolo es necesario pasarle un objeto de tipo **TTransport**, acorde al transporte que se quiera utilizar
    - Transporte HTTP → **THttpClient**
    - Es necesario indicarle la URL del servicio (que se lee del fichero de configuración)



# Comentarios (y 2)

Una vez creada la instancia de **ThriftMovieService.Client** es necesario

- Invocar al método **open** del objeto de tipo **TTransport** utilizado por el cliente (para abrir una conexión con el servidor)
  - El objeto de tipo **TTransport** se obtiene llamando al método **getInputProtocol** del objeto **Client**, que devuelve el protocolo utilizado para leer datos, y a partir del protocolo se obtiene el transporte
- Invocar la operación deseada
- Invocar al método **close** del objeto de tipo **TTransport** para cerrar la conexión con el servidor
- Cualquier excepción que no sea de lógica de la aplicación se relanza como una **RuntimeException**



## 7.4 Otras Tecnologías RPC

---



# Otras Tecnologías RPC

---

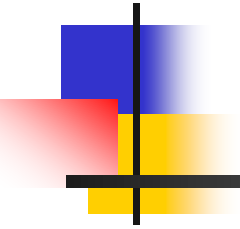
## ■ SOAP

- Es un protocolo (estandarizado por W3C) que permite invocar aplicaciones remotas intercambiando mensajes en formato XML
- Los mensajes SOAP se envían encapsulados en otros protocolos de nivel de aplicación (HTTP es el usado habitualmente)
- Utiliza WSDL como lenguaje de definición de la interfaz del servicio
  - Permite especificar en XML la interfaz de un servicio

## ■ gRPC

- Permite invocar aplicaciones remotas usando un protocolo binario sobre HTTP/2 para el intercambio de mensajes
- Utiliza *Protocol Buffers* como lenguaje de definición de la interfaz del servicio

## 7.5 REST vs RPC





# REST vs RPC (1)

---

## ■ En el modelo RPC

- No existe el concepto de identificador global
  - Normalmente los identificadores son locales a cada servicio
- No se soportan intermediarios transparentes, ya que la semántica de las operaciones es opaca (específica de cada servicio)
  - Aunque algunos frameworks usen HTTP como transporte, suele usarse POST para todo, sea cuál sea la semántica de la operación
- No existe el concepto de hipermedia



# REST vs RPC (2)

- En **algunos** frameworks RPC (sobre todo en las implementaciones más antiguas) el **acoplamiento** del cliente con el servicio es más elevado
  - Casi cualquier cambio en la interfaz del servicio (incluso si se elimina o se añade un campo que nuestro cliente no usa), puede obligar a regenerar los stubs y recompilar
  - Con una librería local, está bajo nuestro control instalar o no una nueva versión, pero los cambios en una aplicación remota y autónoma no están bajo nuestro control
- Con las tecnologías usadas habitualmente con REST, puede conseguirse fácilmente que ciertos cambios no nos afecten
  - Con los frameworks RPC modernos también se puede conseguir (como hemos visto con Apache Thrift)





# REST vs RPC (y 3)

En cuanto a las tecnologías, sin tener en cuenta las diferencias arquitectónicas

- Modelar un servicio con operaciones ad-hoc suele ser muy natural
- El uso de stubs/skeletons es más amigable al programador
  - Con REST manejamos peticiones HTTP y parsing/generación de JSON/XML
    - Aunque es posible utilizar tecnologías de más alto nivel que las vistas en esta asignatura que nos permitan un desarrollo más ágil
- Pero, con el tiempo, el uso de REST se ha hecho más popular en todos los lenguajes de programación