

UNIVERSIDAD AUTÓNOMA DE MADRID

DEPARTAMENTO DE ING. INFORMÁTICA

Sistemas Informáticos II

Práctica - 1B

DANIEL HERNÁNDEZ LOBATO

Registro de Cambios

Versión ¹	Fecha	Autor	Descripción
1.0	10.12.2024	DHL	Primera versión.

¹La asignación de versiones se realizan mediante 2 números $X.Y$. Cambios en Y indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en X indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivos	3
2. Introducción a RPC	4
2.1. Modern RPC	6
2.2. XML RPC	7
3. Creación del Servidor RPC	8
3.1. Creación de un Nuevo Proyecto	8
3.2. Exportación de la Funcionalidad Como Procedimientos Remotos . . .	10
3.3. Prueba del Servidor RPC	11
3.4. Despliegue del Servidor RPC en la VM	13
4. Creación del Cliente RPC	13
4.1. Creación de un Nuevo Proyecto	14
4.2. Codificación del Cliente RPC	15
4.3. Prueba del Cliente RPC	15
4.4. Despliegue del Cliente RPC en la VM	16
5. Introducción a Colas de Mensajes	17
5.1. RabbitMQ	18
5.2. Pika	19
6. Servicio de Cancelación de Voto Mediante RabbitMQ y Pika	19
7. Cliente de Cancelación de Voto Mediante RabbitMQ y Pika	21
8. Ejecución Cancelación de Voto	24
9. Entrega	25
10. Puntuación de Cada Ejercicio	26

1. Objetivos

Los mecanismos de comunicación en sistemas distribuidos son fundamentales para garantizar la interacción eficiente y coordinada entre los diferentes componentes de un sistema que opera de manera descentralizada. En un entorno distribuido, los procesos, que pueden estar ubicados en máquinas físicas distintas y separadas geográficamente, necesitan intercambiar información de forma confiable, segura y rápida. Para ello, se emplean diversas técnicas y herramientas, como servicios web, llamadas a procedimientos remotos (RPC), colas de mensajes, etc. Estos mecanismos permiten superar desafíos inherentes, como la latencia, la tolerancia a fallos y la heterogeneidad de los sistemas, promoviendo la cohesión y el rendimiento en aplicaciones críticas.

El objetivo de esta práctica es implementar un mecanismo de comunicación entre aplicaciones clientes y servidores, utilizando diversas soluciones ampliamente empleadas en la industria, tales como:

- **RPC (Remote Procedure Call):** un protocolo que permite invocar procedimientos remotos como si fueran locales, con transparencia de datos, ofreciendo mayor integración y menor sobrecarga en ciertos casos.
- **Sistemas de colas de mensajes:** que introducen una capa de asincronía y desacoplamiento entre los componentes, favoreciendo la tolerancia a fallos y la escalabilidad.

En esta segunda práctica nos centraremos en la implementación del RPCs como mecanismo de comunicación y su despliegue en las máquinas virtuales, así como el uso de colas de mensajes para invocar servicios remotos. En particular, primero dividiremos la aplicación votoApp en dos partes (cliente y servidor) que se comunicarán mediante RPCs. La Figura 1 ilustra el funcionamiento de la aplicación de partida votoApp. La Figura 2 ilustra el funcionamiento de la aplicación dividida en dos partes (cliente y servidor) que se comunicarán mediante RPCs.. Segundamente implementaremos un servicio de cancelación de voto mediante colas de mensajes.

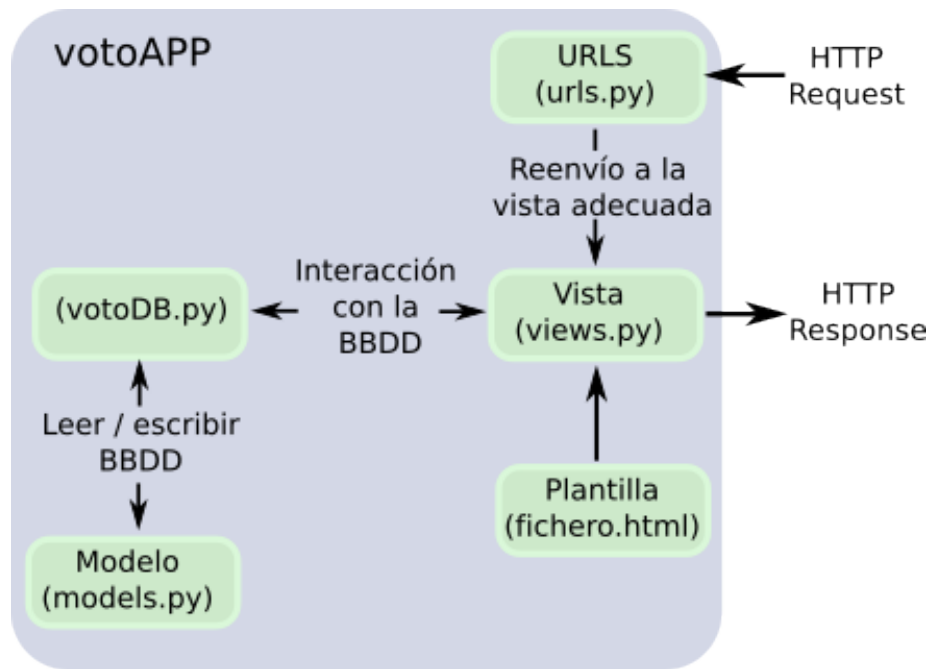


Figura 1: Diagrama de flujo de la aplicación votoApp.

2. Introducción a RPC

La llamada a procedimiento remoto (Remote Procedure Call, RPC) es un mecanismo clave en la comunicación entre componentes de sistemas distribuidos, que permite a un programa ejecutar funciones o procedimientos en un sistema remoto como si estuvieran localmente disponibles. Este enfoque abstrae la complejidad de la comunicación subyacente, permitiendo a los desarrolladores centrarse en la lógica del sistema sin preocuparse por detalles de bajo nivel, como la transmisión de datos o la sincronización. Mediante el uso de RPC, los sistemas distribuidos logran una interacción fluida y eficiente entre procesos ubicados en diferentes máquinas, facilitando la creación de aplicaciones escalables y robustas en entornos heterogéneos. Sin embargo, también plantea desafíos, como el manejo de fallos en la red, la seguridad en la comunicación y la necesidad de garantizar la compatibilidad entre los sistemas que interactúan. La Figura 3 muestra el esquema típico de funcionamiento de RPC como mecanismo de comunicación.

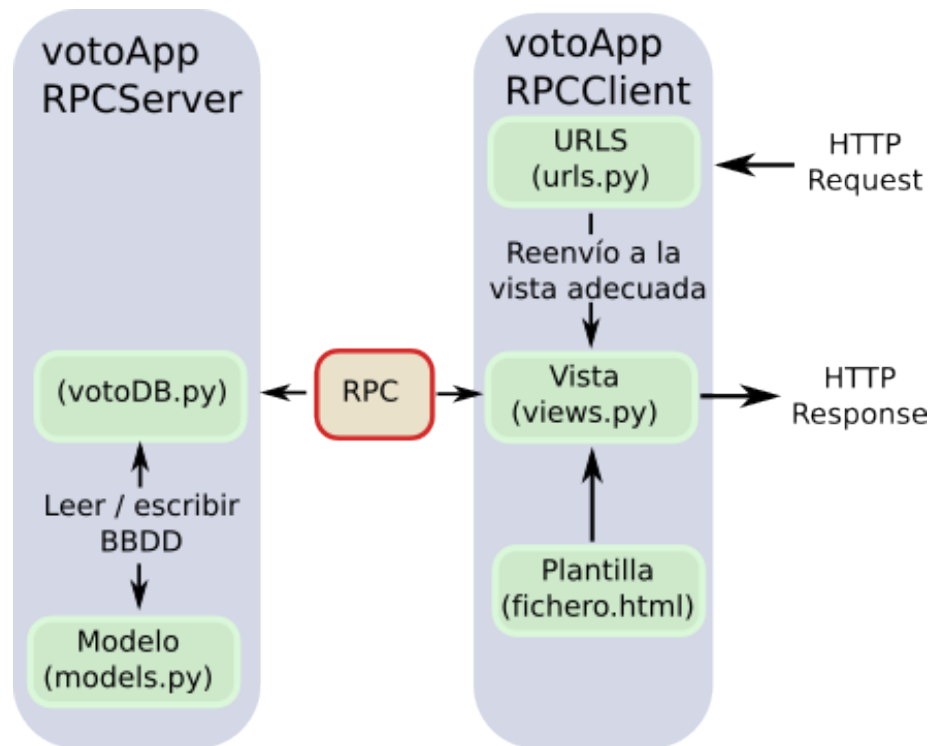


Figura 2: Diagrama de flujo de la aplicación separada en 2 partes comunicadas mediante RPCs.

El mecanismo de llamada a procedimiento remoto (RPC) se compone de varias partes fundamentales que permiten la interacción transparente entre procesos en sistemas distribuidos. En el lado del cliente, se encuentra el stub del cliente, una interfaz que actúa como proxy y encapsula las solicitudes del programa, traduciéndolas en mensajes para su envío al servidor. En el lado del servidor, el stub del servidor recibe estas solicitudes, las deserializa y las pasa al procedimiento correspondiente para su ejecución. Entre estos dos extremos está el módulo de comunicación, encargado de transportar los mensajes mediante protocolos de red, garantizando la fiabilidad y la sincronización. Además, el mecanismo incluye procesos para la serialización y deserialización de datos (*marshaling/unmarshaling*), necesarios para convertir los parámetros de entrada y salida en formatos adecuados para su transmisión. La Figura 4 muestra las componentes típicas en una implementación de RPC como mecanismo

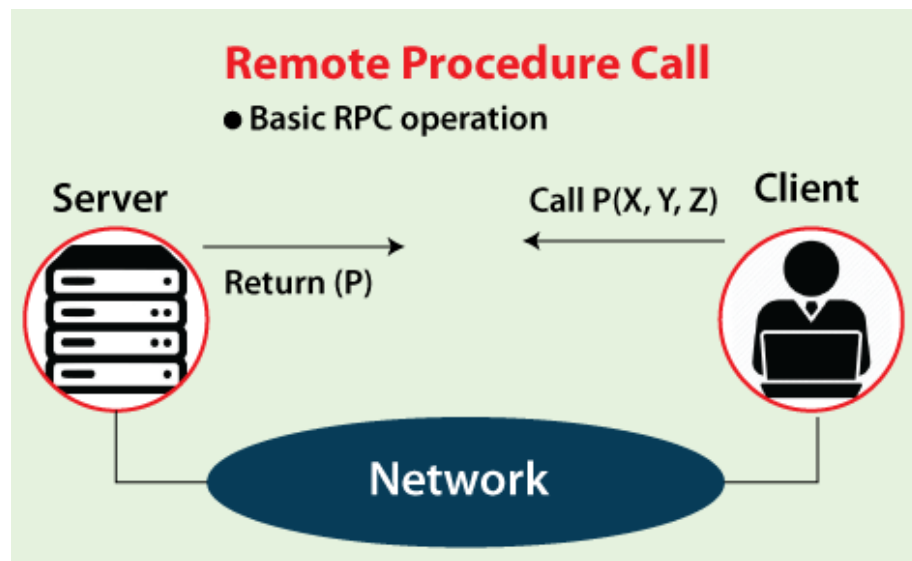


Figura 3: Esquema típico de funcionamiento de una aplicación distribuida basada en RPC. Imagen extraída del enlace.

de comunicación.

2.1. Modern RPC

Modern-RPC es una biblioteca para Django que permite implementar de manera sencilla y eficiente servicios de llamada a procedimiento remoto (RPC) en aplicaciones web. Diseñada para integrar APIs RPC modernas en proyectos Django, esta herramienta facilita la comunicación entre clientes y servidores mediante protocolos como JSON-RPC o XML-RPC, garantizando compatibilidad y simplicidad. Modern-RPC abstrae las complejidades de la comunicación subyacente, proporcionando un marco robusto para definir y gestionar funciones remotas de forma segura, con autenticación integrada y compatibilidad con las características avanzadas de Django, como middleware y decoradores. Esto la convierte en una solución ideal para proyectos que requieren una integración fluida entre clientes distribuidos y servicios back-end en Django. En esta práctica nos centraremos en el uso de XML-RPC como protocolo de comunicación.

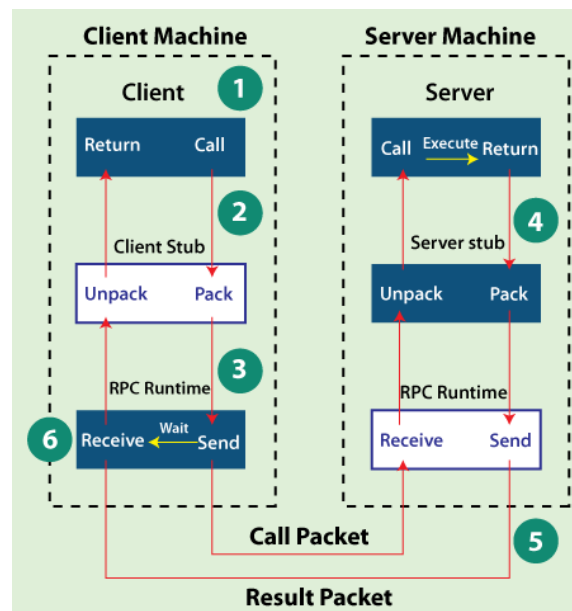


Figura 4: Componentes típicos de RPC para facilitar la interacción entre clientes y servidores. Imagen extraída del enlace.

2.2. XML RPC

XML-RPC es un protocolo de comunicación simple y ampliamente utilizado para la implementación de sistemas distribuidos, que permite la interacción remota entre aplicaciones mediante la transmisión de datos estructurados en formato XML. Este protocolo se basa en el estándar HTTP para el transporte de mensajes, lo que facilita su integración con aplicaciones web y su compatibilidad con diferentes lenguajes de programación y plataformas. En XML-RPC, las solicitudes y respuestas se codifican en XML, describiendo los métodos a invocar y los parámetros asociados, lo que proporciona un formato legible y estandarizado para el intercambio de información. Aunque su diseño es sencillo, XML-RPC es una herramienta poderosa para construir sistemas distribuidos que requieren interoperabilidad y comunicación eficiente, aunque puede ser menos eficiente en comparación con alternativas modernas como JSON-RPC o gRPC debido a la sobrecarga del formato XML.

Listado 1: Ejemplo de petición en XML-RPC


```
1 <?xml version="1.0"?>
2   <methodCall>
3     <methodName>org.wikipedia.intercambioDatos</methodName>
4     <params>
5       <param>
6         <value><i4>360</i4></value>
7       </param>
8       <param>
9         <value><i4>221</i4></value>
10      </param>
11    </params>
12  </methodCall>
```

Listado 2: Ejemplo de respuesta en XML-RPC

```
1 <?xml version="1.0"?>
2   <methodResponse>
3     <params>
4       <param>
5         <value><string>Intercambio datos nro. 360 por
6           ↪ 221</string></value>
7       </param>
8     </params>
9   </methodResponse>
```

3. Creación del Servidor RPC

En esta primera parte de la práctica nos centraremos en crear el servidor RPC usando Modern RPC de Django.

3.1. Creación de un Nuevo Proyecto

Para crear el servidor RPC, el primer paso será crear un nuevo proyecto, partiendo del proyecto P1-base. Por ello, copiaremos el directorio P1-base a P1-rpc-server con

el comando:

```
cp -r P1-base/* P1-rpc-server/*
```

El siguiente paso, será renombrar la aplicación, que en P1-base se llamaba votoApp a votoAppRPCServer. Para ello, ejecutaremos los comandos:

```
cd P1-rpc-server
mv votoApp votoAppRPCServer
find ./ -type f -exec sed -i "s/votoApp/votoAppRPCServer/g" {} \;
```

El siguiente paso será eliminar los ficheros no necesarios en la parte del servidor. En particular, aquellos relacionados con la interfaz web. Para ello ejecutaremos estos comandos:

```
rm votoAppRPCServer/forms.py
rm votoAppRPCServer/views.py
rm -rf votoAppRPCServer/templates
rm votoAppRPCServer/tests_views.py
rm votoAppRPCServer/tests_models.py
```

Por último incluiremos en el fichero settings.py el uso de Modern RPC y los módulos que tendrán los procedimientos remotos. Para ello, actualizaremos la lista de apps instaladas en el fichero settings.py incluyendo una entrada con la cadena 'modernrpc'. También definiremos una lista MODERNRPC_METHODS_MODULES con los módulos que tienen los procedimientos remotos, como se muestra a continuación:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
'votoAppRPCServer.apps.AppConfig',
    "modernrpc",
]
```

```
MODERNRPC_METHODS_MODULES = [
    "votoAppRPCServer.votoDB"
]
```

Ejercicio 1: Ejecute los pasos descritos anteriormente, uno tras otro, para crear el nuevo proyecto. Incluya en la memoria evidencias (capturas de pantalla) de haber realizado estos pasos. Indica por qué no será necesario hacer uso de los formularios Django y las plantillas en la aplicación servidor RPC.

3.2. Exportación de la Funcionalidad Como Procedimientos Remotos

A continuación, para crear el servidor RPC actualizaremos el fichero de mapeo de URLs de la aplicación. Para ello, editaremos el fichero **urls.py** de la aplicación votoAppRPCServer e incluiremos una única entrada, eliminando todas las que se definieron en P1-base:

```
from modernrpc.views import RPCEntryPoint
urlpatterns = [path("rpc/", RPCEntryPoint.as_view(), name="rpc")]
```

Tras esto, exportaremos la funcionalidad para interactuar con la base de datos, definida en votoDB.py como procedimientos remotos. Para ello, editaremos el fichero votoDB.py e introduciremos los siguientes imports:

```
from modernrpc.core import rpc_method
from django.forms.models import model_to_dict
```

A continuación incluiremos la antoación `@rpc_method` encima de cada método definido en votoDB.py, como se ilustra a continuación:

```
@rpc_method
def verificar_censo(censo_data):
    ...
```

Modificaremos el método `registrar_voto` para que devuelva un diccionario con el voto registrado, haciendo uso de `model_to_dict`. Sin embargo, hay que tener en cuenta que `model_to_dict` no incluirá la marca de tiempo del voto, pues no es editable (solo se incluyen campos editables). Por ello, hay que incluirla de forma manual.

```
...
voto_a_devolver = model_to_dict(voto)
voto_a_devolver[ 'marcaTiempo' ] = str(voto.marcaTiempo)
return voto_a_devolver
```

Modificaremos el método `get_votos_from_db` para que devuelva una lista de diccionarios con los votos encontrados, haciendo uso de `model_to_dict`. Para ello es recomendable crear una lista vacía e ir añadiendo a la misma los diccionarios con los votos encontrados. Se devolverá dicha lista. Al igual que se mencionaba más arriba, hay que tener en cuenta que `model_to_dict` no incluirá la marca de tiempo del voto, pues no es editable. Por ello, hay que incluirla de forma manual.

Ejercicio 2: Ejecute los pasos descritos anteriormente, uno tras otro, para exportar la funcionalidad de acceso a la BD como procediminetos remotos. Incluya en la memoria evidencias (capturas de pantalla) de haber realizado estos pasos. Indica razonadamente por qué es necesario hacer uso del método `model_to_dict`.

3.3. Prueba del Servidor RPC

En esta subsección probaremos el correcto funcionamiento del servidor RPC de forma local.

1. Arrancar la primera VM

- Arrancaremos la primera VM en el PC del laboratorio y la configuraremos como servidor de BD. Si no sé creó la base de datos en esa VM, en la práctica anterior, habrá que crearla ahora.

2. Configurar la Base de Datos

- El código asume que existe un fichero llamado **env** en la raíz del proyecto en el que se ha definido la variable `DATABASE_SERVER_URL`:

```
1 # env
2 DATABASE_SERVER_URL =
    ↪ 'postgres://alumnodb:alumnodb@localhost:15432/'
    ↪ voto '
```

3. Ejecutar Migraciones

```
python manage.py makemigrations
python manage.py migrate
```

Nota 1: La base de datos puede que ya exista, si este es el caso os recomendamos borrarla y volver a crearla antes de ejecutar “migrate”.

4. Poblada la base de datos con los datos de censo.

```
python manage.py populate
```

5. Probar Localmente la Aplicación

```
python manage.py runserver
```

Accede a <http://127.0.0.1:8000/votoAppRPCServer/rpc> para verificar que la aplicación funcione correctamente. Recordad que en los ordenadores de los laboratorios el puerto 8000 no está disponible y debéis usar otro puerto como pueda ser el 8001 (`python manage.py runserver 8001`). Si todo ha ido bien, en el navegador nos debería aparecer error HTTP 405, método no permitido.

Ejercicio 3: Ejecuta los pasos descritos más arriba e incluye evidencias en la memoria de haberlos llevado a cabo. Ejecuta los test proporcionados con el proyecto y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test (`python manage.py test votoAppRPCServer.tests_rpc_server`). Los test proporcionados deben tomarse como requisitos extras del sistema.

3.4. Despliegue del Servidor RPC en la VM

Siguiendo indicaciones similares a las que se muestran en la práctica 1-A para el despliegue de la aplicación, desplegar la aplicación servidor RPC en la VM número 2, tal y como se indica en el esquema mostrado en la Figura 5. La base de datos estará en la VM número 1. Ambas VMs se ejecutarán en el mismo PC del laboratorio.

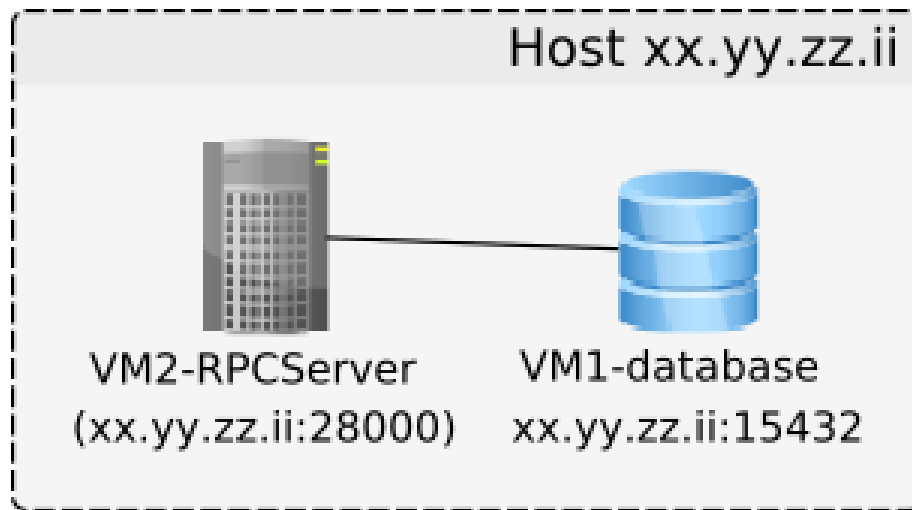


Figura 5: Esquema de despliegue de la aplicación servidor RPC.

Ejercicio 4: Ejecuta los pasos descritos más arriba. Prueba a acceder desde el navegador del PC del laboratorio a la URL `localhost:28000/votoAppRPCServer/rpc` para comprobar que el despliegue de la aplicación es correcto. Incluye evidencias en la memoria en forma de capturas de pantalla mostrando que el acceso a dicha URL es correcto.

4. Creación del Cliente RPC

En esta siguiente parte de la práctica nos centraremos en crear el cliente RPC usando XML-RPC.

4.1. Creación de un Nuevo Proyecto

Para crear el servidor RPC, el primer paso será crear un nuevo proyecto, partiendo del proyecto P1-base. Por ello, copiaremos el directorio P1-base a P1-rpc-client con el comando:

```
cp -r P1-base/* P1-rpc-client/*
```

El siguiente paso, será renombrar la aplicación, que en P1-base se llamaba votoApp a votoAppRPCClient. Para ello, ejecutaremos los comandos:

```
cd P1-rpc-client
mv votoApp votoAppRPCClient
find ./ -type f -exec sed -i "s/votoApp/votoAppRPCClient/g" {} \;
```

El siguiente paso será eliminar los ficheros no necesarios en la parte del servidor. En particular, aquellos relacionados con la base de datos, modelos y migraciones. Para ello ejecutaremos estos comandos:

```
rm -rf votoAppRPCClient/models.py
rm -rf votoAppRPCClient/migrations
rm -rf votoAppRPCClient/management
rm -rf votoAppRPCClient/tests_models.py
```

Por último incluiremos en el fichero env la URL al servidor RPC:

```
RPCAPIBASEURL=http://localhost:28000/votoAppRPCServer/rpc/
```

y en settings.py una variable que la contenga

```
RPCAPIBASEURL = os.environ.get("RPCAPIBASEURL")
```

tras la lectura del fichero env.

Ejercicio 5: Ejecute los pasos descritos anteriormente, uno tras otro, para crear el nuevo proyecto. Incluya en la memoria evidencias (capturas de pantalla) de haber realizado estos pasos. Indica por qué no será necesario hacer uso de los modelos de datos Django en la aplicación cliente RPC.

4.2. Codificación del Cliente RPC

Para que el cliente invoque la funcionalidad remota a través de llamadas RPC, usando XML-RPC como protocolo de comunicación, modificaremos el fichero `votoDB.py` incluyendo los siguientes imports:

```
from django.conf import settings
from xmlrpc.client import ServerProxy
```

A continuación modificaremos cada uno de los métodos en el fichero `votoDB.py` para que invoque el procedimiento remoto correspondiente. A modo de ejemplo, la invocación del procedimiento remoto para registrar el voto sería:

```
with ServerProxy(settings.RPCAPIBASEURL) as proxy:
    return proxy.registrar_voto(voto_dict)
```

Habrás que modificar cada uno de los demás métodos contenidos en el fichero `votoDB.py` de forma similar.

Ejercicio 6: Ejecute los pasos descritos anteriormente, uno tras otro, para crear el nuevo fichero `votoDB.py`, que invoque la funcionalidad de la aplicación como llamadas a procedimiento remoto. Comenta cada función describiendo tanto los argumentos de entrada como los valores devueltos. Incluya en la memoria evidencias (capturas de pantalla) de haber realizado estos pasos. Indica qué tipo de binding, de los tipos vistos en las transparencias de teoría sobre RPC, realiza el cliente RPC codificado.

4.3. Prueba del Cliente RPC

En esta sección llevaremos a cabo los pasos para probar en local el cliente RPC. Se supondrá que el servidor RPC se ha desplegado en la VM como se indicaba en las secciones anteriores.

1. Configurar el acceso al servidor RPC

- El código asume que existe un fichero llamado **env** en la raíz del proyecto en el que se ha definido la variable `RPCAPIBASEURL`:


```
1 # env
2 RPCAPIBASEURL=
    ↪ 'http://localhost:28000/votoAppRPCServer/rpc'
```

2. Probar Localmente la Aplicación

```
python manage.py runserver
```

Accede a <http://127.0.0.1:8000/votoAppRPCClient/> para verificar que la aplicación funcione correctamente. Recordad que en los ordenadores de los laboratorios el puerto 8000 no está disponible y debéis usar otro puerto como pueda ser el 8001 (`python manage.py runserver 8001`).

Ejercicio 7: Ejecuta los pasos descritos más arriba e incluye evidencias en la memoria de haberlos llevado a cabo. Ejecuta los test proporcionados con el proyecto sobre las vistas y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test (`python manage.py test`).

4.4. Desliece del Cliente RPC en la VM

Siguiendo indicaciones similares a las que se muestran en la práctica 1-A para el despliegue de la aplicación, desplegar la aplicación servidor RPC en la VM número 3, tal y como se indica en el esquema mostrado en la Figura 6. La base de datos estará en la VM número 1. La VM 3 se ejecutará en otro PC del laboratorio. Habrá que actualizar la variable `RPCAPIBASEURL` en el fichero `env` del cliente RPC.

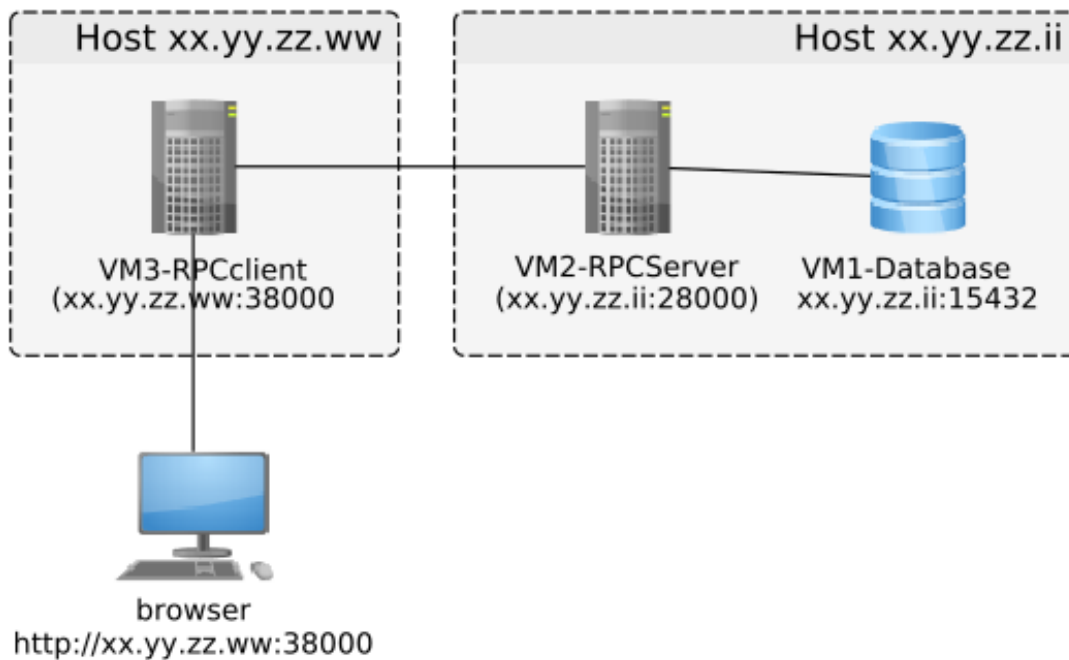


Figura 6: Esquema de despliegue de la aplicación cliente RPC.

Ejercicio 8: Ejecuta los pasos descritos más arriba. Prueba a acceder desde el navegador del PC del laboratorio a la URL localhost:38000/votoAppRPCClient/ para comprobar que el despliegue de la aplicación es correcto. Registra un voto, lístalo (tanto desde la aplicación usando testbd como desde un cliente SQL como DBeaver), y bórralo. Registra el voto tanto desde localhost:38000/votoAppRPCClient/ como desde localhost:38000/votoAppRPCClient/testbd. Incluye evidencias en la memoria en forma de capturas de pantalla.

5. Introducción a Colas de Mensajes

La comunicación mediante colas de mensajes es un paradigma ampliamente utilizado en sistemas distribuidos para garantizar el intercambio eficiente, asíncrono y confiable de datos entre componentes o aplicaciones. Este enfoque utiliza colas intermedias para almacenar temporalmente los mensajes enviados por un productor

hasta que un consumidor esté listo para procesarlos, desacoplando así a los participantes y mejorando la escalabilidad del sistema. Las colas de mensajes permiten gestionar cargas de trabajo variables, implementar patrones de comunicación como publicación-suscripción o punto a punto, y asegurar la persistencia de los mensajes en caso de fallos. Herramientas como RabbitMQ, Apache Kafka o Amazon SQS ofrecen potentes plataformas para implementar este modelo, resolviendo desafíos relacionados con la latencia, la concurrencia y la tolerancia a fallos en entornos distribuidos modernos. En esta parte de la práctica nos vamos a centrar en el uso de RabbitMQ como implementación de comunicación mediante colas de mensajes. La Figura 7 ilustra de forma esquemática la comunicación mediante colas de mensajes.



Figura 7: Esquema típico de funcionamiento de una aplicación distribuida basada en colas de mensajes.

5.1. RabbitMQ

RabbitMQ es una plataforma de mensajería robusta y versátil que implementa el protocolo de mensajería AMQP (Advanced Message Queuing Protocol), diseñada para facilitar la comunicación entre aplicaciones mediante colas de mensajes. Es ampliamente utilizada en sistemas distribuidos gracias a su capacidad para manejar grandes volúmenes de mensajes de manera eficiente y a su soporte para diversos patrones de mensajería, como publicación-suscripción, punto a punto y enrutamiento basado en temas de mensajes. RabbitMQ proporciona características avanzadas como la persistencia de mensajes, gestión de prioridades, tolerancia a fallos mediante clústeres y réplicas, y extensibilidad a través de plugins. Además, es compatible con múltiples lenguajes de programación y entornos, lo que lo convierte en una opción ideal para arquitecturas modernas que requieren comunicación confiable, escalable

y desacoplada entre componentes. En esta práctica usaremos RabbitMQ como implementación para llevar a cabo comunicación con colas de mensajes. RabbitMQ ya está instalado en las VMs de la asignatura.

5.2. Pika

Pika es un paquete de Python ligero y flexible diseñado para interactuar con sistemas de mensajería basados en el protocolo AMQP, como RabbitMQ. Este paquete proporciona herramientas para crear y gestionar conexiones, canales y colas, permitiendo a los desarrolladores implementar patrones de mensajería como publicación-suscripción, enrutamiento o mensajes punto a punto de forma sencilla y eficiente. Pika destaca por su diseño minimalista, lo que le permite integrarse fácilmente en proyectos de diferentes tamaños y niveles de complejidad. Con soporte tanto para operaciones síncronas como asíncronas, Pika es una opción ideal para construir aplicaciones distribuidas que requieren comunicación desacoplada, confiable y escalable, ofreciendo una base sólida para manejar mensajes en entornos de producción. En esta práctica usaremos el paquete de Python Pika para acceder a RabbitMQ.


6. Servicio de Cancelación de Voto Mediante RabbitMQ y Pika

En esta sección elaboraremos un servidor de cancelación de votos que interactuará con los clientes a través de comunicación mediante colas de mensajes. Para construir el servicio, crearemos un fichero llamado `server_mq.py` en el directorio `P1-rpc-server/votoAppRPCServer/` y partiremos del código mostrado a continuación:

Listado 3: Listado del fichero `server_mq.py` de partida.

```
1 # Uses rabbitMQ as the server
2
3 import os
4 import sys
5 import django
```

```
6 import pika
7
8 BASE_DIR = os.path.dirname(
9     os.path.dirname(os.path.abspath(__file__)))
10 sys.path.append(BASE_DIR)
11
12 os.environ.setdefault('DJANGO_SETTINGS_MODULE',
13     'votoSite.settings')
14 django.setup()
15
16 from votoAppRPCServer.models import Censo, Voto
17
18 def main():
19
20     if len(sys.argv) != 3:
21         print("Debe indicar el host y el puerto")
22         exit()
23
24     hostname = sys.argv[ 1 ]
25     port = sys.argv[ 2 ]
26
27     # TODO: completar segun las indicaciones
28
29
30 if __name__ == '__main__':
31
32     main()
```

Código disponible en este enlace: .

El código de dicho servidor debe ser completado para llevar a cabo los siguientes pasos:

1. **Crear una conexión con el servidor RabbitMQ:** Para ello se creará un objeto de tipo `pika.BlockingConnection`, y se invocará sobre él el método `channel()`, para crear un canal de comunicación. Se deberán usar los parámetros

de conexión como nombre de host y n^o de puerto pasados como argumentos. Como credenciales, el nombre de usuario será 'alumnomq' y la contraseña será 'alumnomq'. El servidor RabbitMQ de la VM está configurado para que este usuario ya exista con esa contraseña.

2. **Crear una cola de mensajes:** Usando el canal de comunicación creado, se declarará una cola de mensajes llamada 'voto_cancelacion'.
3. **Crear y registrar función de Callback:** Se creará y se definirá una función de callback que procesará los mensajes recibidos. La función de callback, recibirá un mensaje con el identificador del voto a cancelar y usará el modelo de datos de *Django* para cancelar el voto, modificando el código de respuesta del voto a '111'. La función de callback deberá imprimir información sobre el mensaje recibido y la cancelación satisfactoria o no del voto correspondiente.
4. **Registrar función de callback y comienzo de consumo de mensajes:** Se deberá registrar la función de callback, asociada a la cola 'voto_cancelacion' y se comenzará a consumir mensajes.

Para saber cómo llevar a cabo los pasos descritos anteriormente, consultar el tutorial sobre RabbitMQ y Pika disponible en este enlace. Para pasar un nombre de usuario y una contraseña al crear la conexión, como credenciales, se usará un objeto de tipo PlainCredentials:

```
credentials = pika.PlainCredentials('alumnomq', 'alumnomq')
```

Revisar la documentación disponible en este enlace.

Ejercicio 9: Ejecuta los pasos descritos más arriba. Incluye evidencias en la memoria en forma de capturas de pantalla del código desarrollado.


7. Cliente de Cancelación de Voto Mediante RabbitMQ y Pika

En esta sección elaboraremos un cliente de cancelación de votos que interactuará con los servidores a través de comunicación mediante colas de mensajes. Para

construir el cliente, crearemos un fichero llamado `client_mq.py`. Dicho fichero lo incluiremos en el proyecto `P1-rpc-client`, en la sub-carpeta `cliente_mom`, que deberemos crear previamente. Para crear el fichero Python del cliente mom, partiremos del código mostrado a continuación:

Listado 4: Listado del fichero `client_mq.py` de partida.

```
1 import pika
2 import sys
3
4 def cancelar_voto(hostname, port, id_voto):
5
6     try:
7         # TODO: conectar con rabbitMQ
8     except e:
9         print("Error al conectar al host remoto")
10        exit()
11
12    # TODO: declarar cola y publicar mensaje
13
14    # TODO: cerrar conexion
15
16 def main():
17
18     if len(sys.argv) != 4:
19         print("Debe indicar el host, el numero de puerto,
20             ↪ y el ID del voto a cancelar como un
21             ↪ argumento.")
22         exit()
23
24         cancelar_voto(sys.argv[ 1 ], sys.argv[ 2 ], sys.argv[
25             ↪ 3 ])
26
27 if __name__ == "__main__":
28     main()
```

Código disponible en este enlace: .

El código de dicho cliente debe ser completado para llevar a cabo los siguientes pasos:

1. **Crear una conexión con el servidor RabbitMQ:** este paso se llevará a cabo de forma análoga a en el servidor.
2. **Crear una cola de mensajes:** Usando el canal de comunicación creado, se declarará una cola de mensajes llamada 'voto_cancelacion'. Este paso se llevará a cabo de forma análoga al servidor. Si la cola no existía previamente, se creará en este momento.
3. **Envío de mensaje:** El último argumento se interpretará como un identificador de voto que debe ser cancelado y se depositará ese identificador en la cola de mensajes.

Para saber cómo llevar a cabo los pasos descritos anteriormente, consultar el tutorial sobre RabbitMQ y Pika disponible en este enlace.

Ejercicio 10: Ejecuta los pasos descritos más arriba. Incluye evidencias en la memoria en forma de capturas de pantalla del código desarrollado.

8. Ejecución Cancelación de Voto

Ejercicio 11: En este ejercicio probaremos a ejecutar el código desarrollado en los pasos anteriores.

1. Ejecuta el cliente del servicio de cancelación de votos en el PC del laboratorio. Para ello, escribir `python cliente_mq.py` seguido de la dirección IP, número de puerto de la VM1 para RabbitMQ, e identificador de voto a cancelar. De esta forma el servidor RabbitMQ que se usará será el de la VM1.
2. Como todavía no se ha arrancado el servidor de colas de mensajes, el mensaje enviado se almacenará en la cola. Probar a listar las colas y el número de mensajes en cada cola en la VM1 ejecutando el comando `sudo rabbitmqctl list_queues`. Comprobar que aparece la cola de mensajes asociada al servicio y que contiene un mensaje.
3. Ejecuta el servidor del servicio de cancelación de votos en la VM donde se esté ejecutando el servidor RPC. Debería ser la VM2. Para ello, escribir `python servidor_mq.py` seguido de la dirección IP y número de puerto para RabbitMQ en la VM1. De esta forma usaremos el servidor RabbitMQ que se está ejecutando en la VM1.
4. Comprobar que se ha cancelado el voto correspondiente en la base de datos. Para ello, probad a listar los votos asociados a dicho proceso electoral usando el cliente RPC ejecutándose en la VM3.
5. Probar a listar las colas y el número de mensajes de cada cola, disponibles en la VM1, de nuevo. Comprobar que ya no aparecen mensajes almacenados en la cola.

Incluid en la memoria capturas de pantallas y evidencias de haber llevado a cabo este ejercicio.

9. Entrega

La fecha de entrega de esta práctica se encuentra la ‘Planificación de Prácticas’ disponible en moodle.

La entrega de los resultados de esta práctica se regirá por las normas generales expuestas durante la presentación de la asignatura. El incumplimiento de estas normas conllevará a considerar que la práctica no ha sido entregada en tiempo. Esto implica que se tendrá que volver a enviar correctamente y que se aplicará la penalización por retraso pertinente.

A modo de resumen, para esta práctica se espera encontrar dentro del archivo SI2P1B_<grupo>_pareja>.zip (ejemplo: SI2P1B_2311_1.zip):

- Informe técnico.
- P1-rpc-server con todas las modificaciones que hayan sido necesarias para servidor RPC, incluyendo el servidor de cancelación de voto, server_mq.py.
- P1-rpc-client con todas las modificaciones que hayan sido necesarias para cliente RPC.
- Cliente de cancelación de voto client_mq.py.

10. Puntuación de Cada Ejercicio

Ejercicio	Puntos
1	0.5
2	0.5
3	0.75
4	1.0
5	0.5
6	0.5
7	0.75
8	1.5
9	1.0
10	1.0
11	1.0
Memoria	1.0
Total	10.0

Para aprobar, es necesario que el servicio RPC funcione correctamente para, al menos, registrar votos. En caso de que dicho servicio no funcione correctamente, para la mencionada funcionalidad, la puntuación será el mínimo de la nota calculada con el anterior baremo y 4,9.