Instituto Politécnico Nacional

Escuela Superior de Cómputo

Desarrollo de Sistemas Distribuidos

Prof. Ephrain Herrera Salgado

Práctica: Reserva de libros con sincronización
Sincronización en Sistemas Distribuidos

Grupo: 4CM1

Alumno: Naranjo Ferrara Guillermo

Enlace de video:
https://youtu.be/rFmYUT1n7EY

Fecha:22 Enero 2021

Práctica 5: Sincronización en sistemas distribuidos

Objetivo de la Práctica Que el alumno aplique los algoritmos de sincronización de reloj en un sistema distribuido para mantener coherencia en el tiempo.

Tecnologías a aplicar: Sockets, RMI, POO, Protocolos de comunicación, Bases de Datos, algoritmo de Christian.

Actividades

A partir de la práctica 4, desarrollará una aplicación para repartir libros de un sistema de apartado virtual en una biblioteca a través de dos servidores. Los servidores deberán tener sus relojes sincronizados. De acuerdo a los siguientes requerimientos:

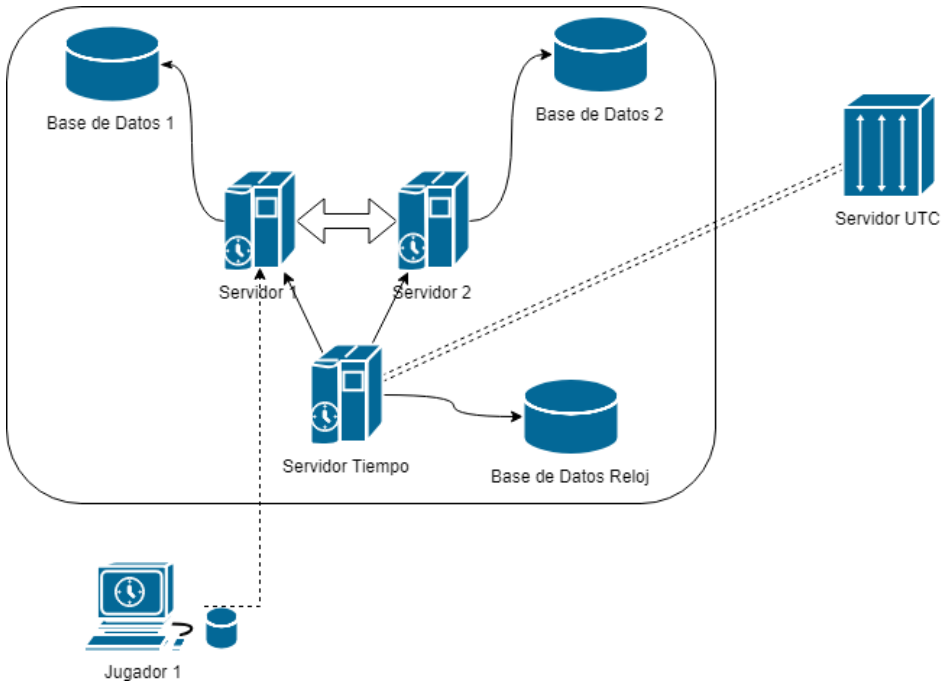


Figura 1. Sincronización usando el algoritmo de Cristian

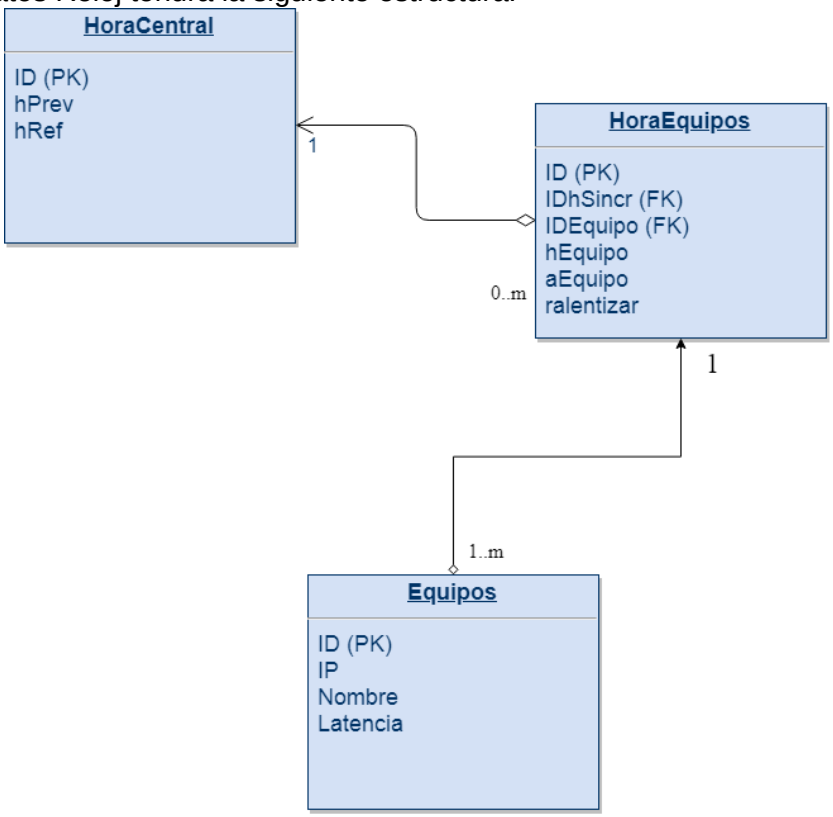
Requerimientos funcionales

- Los servidores son los únicos conectados y con acceso a las bases de datos.
- Dentro de la interfaz de cada servidor habrá un botón de “Reiniciar” y un Canvas para poner una imagen, inicialmente vacío.
- En cada computadora Servidor y Usuario está el reloj de la práctica 1.
- En cada computadora Usuario hay un botón de “Pedir libro”. Éste manda una petición al servidor correspondiente, con la cual dicho servidor envía la información de un libro al azar a ese Usuario.
- La información de petición (IP, hora, libro, etc.) se guardará en la base de datos.
- El botón “Reiniciar”, permitirá reiniciar la sesión del lado del servidor.
- La portada del libro elegido se mostrará (en forma de imagen) solamente en la interfaz gráfica del servidor. En el cliente se mostrará solamente en formato de texto.

- Cuando termina la sesión (se repartieron todos los libros) se le notificará al usuario si quiere salir o reiniciar una nueva sesión.
- Al inicio los relojes tienen horas diferentes elegidas al azar.
- Todos los relojes se pueden modificar manualmente.
- El Servidor de Tiempo tendrá que sincronizar el sistema cada ™ segundos.
- El formato de hora es de 24 hrs.
- Sólo el servidor de tiempo se conectará a un servidor UTC.

Requerimientos no funcionales

- Al pulsar el botón de modificar, el reloj correspondiente se detendrá.
- Tomar en cuenta la latencia al enviar la hora por parte del coordinador.
- El coordinador cada ™ segundos deberá pedir la hora a los demás relojes.
- Para sincronizar se utilizará el algoritmo de Cristian
- Una vez que se calcula la nueva hora de referencia el coordinador envía los ajustes correspondientes.
- Cada vez que se calcule la hora de referencia se deberá crear un nuevo registro en la Base de Datos Reloj.
- La Base de Datos Reloj tendrá la siguiente estructura.



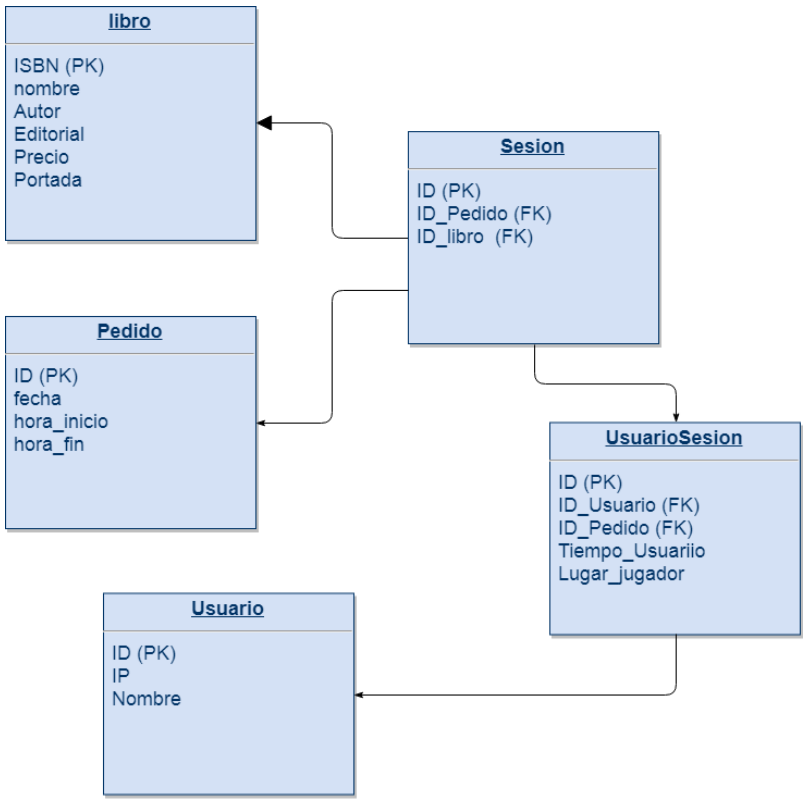


Figura 1. Esquema de la Base de Datos sugerido

Relizar un video con las siguientes características:

Introducción

Un sistema distribuido debe permitir el apropiado uso de los recursos, presentar un buen desempeño y mantener la consistencia de los datos. Consiste en garantizar que los procesos se ejecuten en forma cronológica y a la misma vez respetar el orden de los eventos dentro del sistema. La sincronización de procesos en los sistemas distribuidos resulta más compleja que en los centralizados, debido a que la información y el procesamiento se mantienen en diferentes nodos. Un sistema distribuido debe mantener vistas parciales y consistentes de todos los procesos cooperativos y de cómputo. Tales vistas pueden ser provistas por los mecanismos de sincronización.

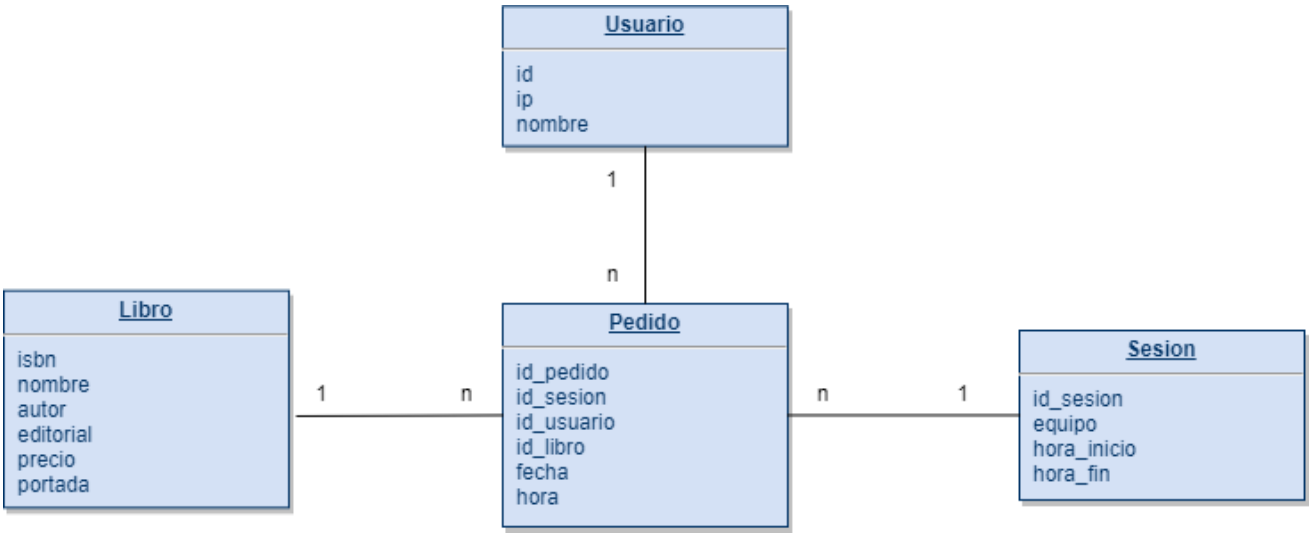
El término sincronización se define como la forma de forzar un orden parcial o total en cualquier conjunto de eventos.

En esta práctica se desarrollará un sistema que presente la sincronización de los eventos por medio de relojes lógicos, y de la misma manera mantenga la consistencia de los datos almacenados en la BD.

La práctica se procederá a desarrollar en el lenguaje Python y como base de datos se usará MySQL con el motor InnoDB. Para mantener la consistencia de los datos y la sincronización de los eventos, se hará uso de sockets y del algoritmo de Christian para la sincronización de los relojes, y se implementarán llamadas a procedimientos remotos, facilidad que nos proporciona la librería Pyro 4 para Python.

Análisis y Diseño

Base de Datos para los libros



Base de datos para los tiempos

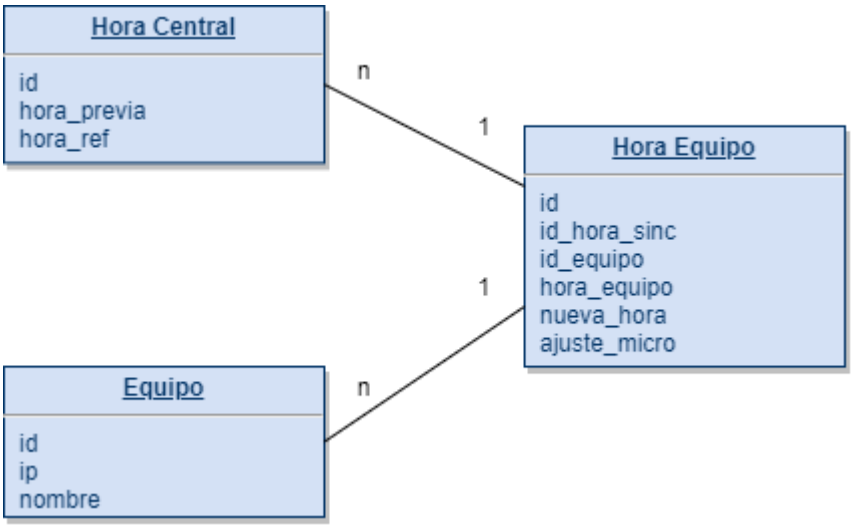
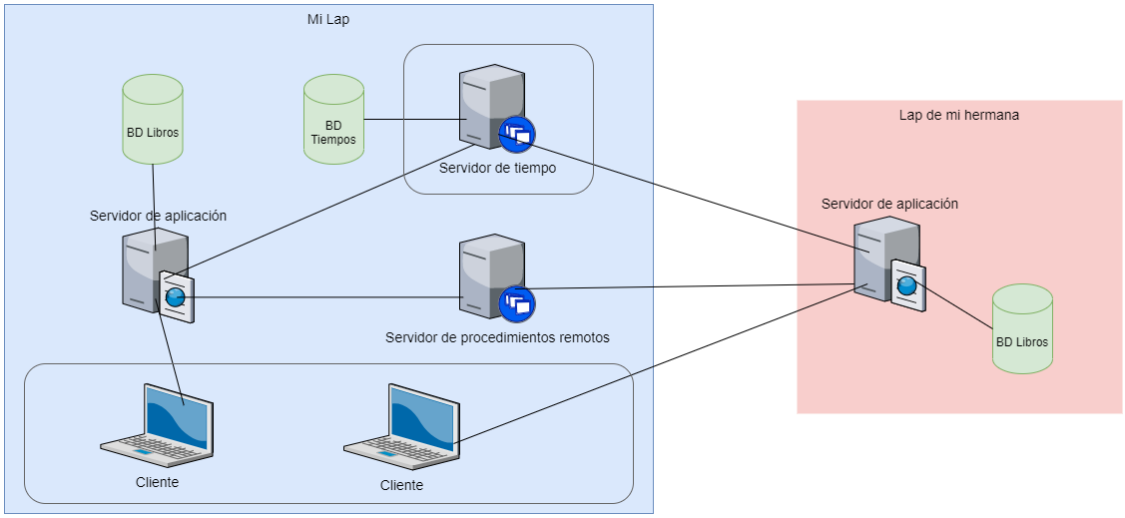


Diagrama de arquitectura

En el siguiente diagrama, se encuentra la disposición de cada uno de los equipos que van a interactuar en el sistema. Aquellos que se encuentran rodeados por un cuadrado con esquinas redondeadas, son los equipos que se tienen no de forma física, sino virtualizada.



Desarrollo de la práctica

Para el desarrollo del proyecto se implementaros las siguientes clases:

- Clock
- ServerTime
- TimeDB_Connection
- Server
- BooksDB_Connection
- Queries
- Client

A continuación se describen los métodos principales de estas clases:

ServerTime

Esta clase es la que se encarga de sincronizar los relojes de los servidores de aplicación de sistema y registrar en la base de datos de los tiempos los ajustes realizados.

- setInitialHour(). Este método se encarga de recibir las solicitudes de los servidores de aplicación para que se les envíe el tiempo inicial y así el sistema esté sincronizado desde el principio. Al momento de recibir la petición verifica en la BD si el equipo ya ha sido registrado, sino, lo registra en la BD.

```python
def setInitialHour(self):
    receiveSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    receiveSocket.bind(('10.0.0.13', 9000))
    clients = 0
    while clients < 2:
        clientSock = receiveSocket.recvfrom(1024)
        if clientSock[0]:
            #Register node
            self.timeDB.nodeExists(clientSock[1])
            #Send initial hour to application server
            hour = self.clock.returnHour()
            receiveSocket.sendto(hour.encode(), clientSock[1])
            print(clientSock[1])
            print(clientSock[1][0])
            modSock = (clientSock[1][0], 8000)
            print(modSock)
            self.serversAddress.append(modSock)
            clients += 1
```

- alwaysRunning(). Este método se encarga de, cada 10 segundos, invocar al método que solicita a los servidores de aplicación que sincronicen sus relojes.

```python
def alwaysRunning(self):
    while True:
        timesl.sleep(10)
        print("Requesting times")
        #Thread to adjust servers hours
        if len(self.serversAddress) >= 1:
            self.adjustHours(self.serversAddress[0])
        if len(self.serversAddress) == 2:
            self.adjustHours(self.serversAddress[1])
```

- adjustHours(). En este método es el que se encarga directamente de solicitar al servidor que sincronice su reloj. El servidor de tiempo recibe la hora inicial del servidor de aplicación, posteriormente envía su hora para que el servidor se sincronice, y una vez hecho esto, recibe la nueva hora sincronizada por el servidor de aplicación y la diferencia entre la hora anterior y la actual, tras lo cual procede a hacer el registro en la base de datos.

```python
def adjustHours(self, serverAddress):
    adjustSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    #Receving server app time and sending correct time
    adjustSocket.sendto("Send hour".encode(), serverAddress)
    timeSer, serverNode = adjustSocket.recvfrom(1024)
    adjustSocket.sendto(pickle.dumps(self.clock.returnHourTime()), serverAddress)
    timeSer2, serverNode = adjustSocket.recvfrom(1024)
    if timeSer and timeSer2:
        final = pickle.loads(timeSer2)
        print(f'{final["hour"]}, {final["adjust"]}')
        aux = final["adjust"]
        adjust = datetime.combine(date.today(), time(0, 0, 0))+aux
        self.timeDB.setNodeHour(serverNode, pickle.loads(timeSer), final["hour"], adjust.microsecond)
```

TimeDB_Connection

Esta clase es la que establece la conexión con la BD para almacenar los tiempos del sistema. Entre otros métodos como la inserción de un equipo y el registro del tiempo del servidor de tiempo, el más importante es este, que hace los registros de los tiempos y ajustes de sincronización de los servidores de aplicación.

```
def setNodeHour(self, userIP, time, adjust_time, adjust):
    #Getting node
    nodeId_query = "SELECT id from equipo where ip=%s"
    self.cursor.execute(nodeId_query, (userIP[0], ))
    nodeId = self.cursor.fetchall()

    #Insert next hour adjust record
    setNodeHour_query = "INSERT INTO hora_equipo VALUES(%s, %s, %s, %s, %s, %s)"
    node_hour_tuple = (self.nextHourNode, self.nextHour, nodeId[0][0], time, adjust_time, adjust)
    self.cursor.execute(setNodeHour_query, node_hour_tuple)
    self.db.commit()
    self.nextHourNode += 1
```

Server

Esta clase es la que da funcionalidad al servidor de aplicación. Antes de empezar a describir la implementación de los métodos es importante destacar la siguiente línea de código.

```
sincQueries = Pyro4.Proxy("PYRO:biblioteca.queries@10.0.0.13:17000")
```

Esta línea se encarga de establecer la conexión con el servidor donde se encuentran los procedimientos remotos, lo que hace posible la sincronización entre las BDs.

- receiveInitialHour(). Este método es el primero en ejecutarse al iniciarse el servidor, de manera que se pone en contacto inmediato con el servidor central para así sincronizar su tiempo.

```
def receiveInitialHour(self):
    receiveSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    receiveSocket.sendto("Initial time".encode(), ('10.0.0.27', 9000))
    initialHour, serverAddr = receiveSocket.recvfrom(1024)
    return initialHour.decode('utf-8')
```

- adjustHour(). Este método corre en un hilo que siempre se encuentra a la escucha de la solicitud del servidor central para sincronizar su tiempo. Una vez recibe la petición, obtiene su tiempo y pone un timer para medir el tiempo de respuesta del servidor, el cual le regresa su tiempo. Una vez recibido el tiempo correcto se para el timer y se procede a aplicar el algoritmo de Christian para sincronizar el reloj del servidor de aplicación con el de tiempo. Cabe destacar que para poder hacer ajustes certeros se tuvo que hacer modificaciones a la clase Clock de la práctica anterior, a fin de que trabajara con microsegundos.
  Una vez calculado el tiempo corregido, éste, así como la diferencia entre los relojes son pasados ala clase modifyHour() para que proceda a realizar los ajustes.

Finalmente, la hora corregida y el ajuste realizado es enviado al servidor para su registro en la BD.

```
def adjustHour(self):
    adjustSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    adjustSocket.bind(('10.0.0.13', 8000))
    print("Listening in ")

    while True:
        serverHour, servAdd = adjustSocket.recvfrom(1024)
        if serverHour:
            #Send hour to time server
            hour = self.clock.returnHourTime()
            t0 = timer()
            adjustSocket.sendto(pickle.dumps(hour), servAdd)
            sTime, address = adjustSocket.recvfrom(1024)
            t1 = timer()
            serverTime = pickle.loads(sTime)
            adjust = timedelta(seconds = t1-t0)/2.0
            aux = datetime.combine(date.today(), serverTime) + adjust
            correction_time = aux.time()
            self.modifyHour(hour, correction_time)
            final = {"hour": correction_time, "adjust": adjust}
            adjustSocket.sendto(pickle.dumps(final), servAdd)
```

- modifyHour(). Este método es el que se encarga de sincronizar el reloj del servidor, dependiendo de la corrección de tiempo resultante, de manera que si está atrasado, simplemente se hace el ajuste del reloj a la nueva hora, si está adelantado, se pone a dormir al hilo que se encarga de mantener corriendo el reloj hasta que la hora se sincronice con la del servidor central.

```
def modifyHour(self, oldHour, newHour):
    if (newHour > oldHour):
        print("Hora nueva mayor")
        self.clock.stopClock()
        self.clock.modifyClock(self.lock, newHour)
    else:
        print("Hora vieja mayor")
        self.clock.stopClock()
        micros = oldHour.microsecond/1000000.0
        timesl.sleep(micros)
        self.clock.modifyClock(self.lock)
```

- receiveRequests(). Este método es ejecutado por un hilo que siempre está a la escucha de las peticiones de los clientes en el puerto 7900 del servidor de aplicación. Cuando recibe una petición de un cliente, primeramente le envía la hora del servidor para que se sincronice. Posteriormente realiza varias consultas a la BD para registrar al usuario, si este no lo está y para solicitar un libro. Por último envía el libro al usuario.

```python
def reciveRequests(self):
    self.startNewSession()
    receiveSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    receiveSocket.bind(('10.0.0.13', 7900))

    while True:
        clientSock = receiveSocket.recvfrom(1024)
        if clientSock[0]:
            #Send hour to client
            hour = self.clock.returnHourTime()
            receiveSocket.sendto(pickle.dumps(hour), clientSock[1])
            #DB Queries
            userExists = self.bookDB.userExists(clientSock[1])
            if userExists != None:
                self.syncUploadDB(userExists)
            self.requestBook(clientSock[1], hour)
            #Send book to client
            receiveSocket.sendto(self.bookName.encode(), clientSock[1])
```

- requestBook(). Este método es llamada por el método anterior, pasándole los datos de la IP del usuario que solicitó el libro y la hora en que lo solicitó.
  Primeramente el servidor selecciona un libro aleatoriamente con base en los que se indican disponibles en un arreglo de 0s y 1s. Ahora bien, como este arreglo no está sincronizado entre los servidores, pero sí la BD, una vez seleccionado un libro sin entregar según el arreglo, se consulta a la BD, y si este ya ha sido entregado, se procede a actualizar el arreglo del servidor y se selecciona un nuevo libro aleatoriamente.
  Una vez seleccionado un libro válido, se hace la inserción del registro en la base de datos y se obtienen tanto la imagen como el nombre del libro.
  La imagen, al estar guardada en la BD, se tiene que reconstruir del stream de bytes por medio del método io.BytesIO(), tras lo cual es convertida a una imagen compatible con el canvas de tkinter. La imagen se muestra en el servidor y al usuario es enviado el nombre del libro que se le ha entregado.
  Finalmente se hace la sincronización de las bases de datos.

```python
def requestBook(self, userIP, time):
    #Selecting book randomly
    random.seed()
    book = random.randint(0,14)
    entregados = numpy.where(self.assignedBooks == 0)
    allData = None
    while allData == None:
        if len(entregados[0]) > 0:
            while self.assignedBooks[book] == 1:
                book = random.randint(0,14)
        self.assignedBooks[book] = 1
        allData = self.bookDB.getBook(book, userIP, self.currentSession, time)
    bookData = allData["bookInfo"]
    self.bookName = bookData[0][1]
    imageBytes = bookData[0][-1]
    imageGenerated = Image.open(io.BytesIO(imageBytes))
    self.bookImage = ImageTk.PhotoImage(imageGenerated)
    self.canvas.create_image(70, 30, anchor='center' ,image=self.bookImage)
    self.syncUploadDB(allData["query"])
```

- syncDownloadDB() y syncUploadDB(). Estas clases son las que se encargan del intercambio de la información para la sincronización entre las bases de datos.
  En la primera, un hilo se encarga de checar cada medio segundo, si el otro servidor ha subido un nuevo query al servidor de procedimientos remotos; en caso de que así sea, todos los queries que se hayan subido, son obtenidos a través de la llamada a getQuery() y se pasan al método synchronizeDB() de la clase BooksDB_Connection() para ejecutar todos los queries que ya se han ejecutado en el otro servidor.
  El segundo método se encarga de subir cada nuevo query realizado en la BD local.

```
def syncDownloadDB(self):
    while(True):
        queries = self.sincQueries.getQuery(self.server)
        for query in queries:
            print("Synchronizing")
            self.bookDB.synchronizeDB(query)
        timesl.sleep(0.5)

def syncUploadDB(self, newQuery):
    self.sincQueries.setQuery(newQuery, self.server)
```

BooksDB_Connection

Esta clase es la que establece la conexión entre el servidor de aplicación y la BD de la librería. Primeramente es necesario mencionar la estructura que siguen todos los métodos que realizan alguna modificación a la BD.

- Especificación de la consulta a realizar
- Creación de la tupla que contiene las variables que se pasan al query.
- Ejecución del query.
- Obtención del resultado.
- Organización de información para sincronización de las BD que será retornado al servidor para que proceda a compartirlo por medio de la llamada a un procedimiento remoto. La información se organiza en un diccionario que cuenta con la siguiente estructura:
  - “type”: El tipo de registro que se ha hecho, ya sea de un nuevo usuario, una nueva sesión, finalización de una sesión o un pedido.
  - “query”: El query formulado anteriormente
  - “values”: La tupla creada con los valores necesarios para hacer la actualización de la BD.

Con esta estructura en mente, a continuación se describen unos de los métodos clave para el funcionamiento de esta clase.

- getBook(). Este método es el núcleo de la repartición de libros y el registro de los pedidos. Recibe como parámetros, la IP del cliente que ha solicitado el libro, la sesión a la que corresponde dicha solicitud y el tiempo en que se realizó.
  Primeramente se verifica que el libro esté disponible para esa sesión, de no ser así se regresa None al servidor, para que este escoja otro aleatoriamente. Si el libro sí se encuentra disponible, se procede a obtener la información del libro y el id correspondiente al usuario que lo solicitó.

Posteriormente se procede a hacer la inserción del pedido en la base de datos, conforme a la estructura mencionada anteriormente, y finalmente se crea el diccionario con la información del query.
Sin embargo, como para este método en específico es necesario retornar información almacenada en la BD, el diccionario que contiene la información del query, será incluido en otro diccionario que tendrá dos índices: “bookInfo”, que contiene el resultado de la consulta del libro, lo que incluye su nombre y su portada; y “query”, que contiene la información descrita en la estructura al comienzo.

```
def getBook(self, book, userIP, session, reqHour):
    isbn = self.books[book]
    #Checking book availability
    checkBook_query = "SELECT COUNT(*) FROM pedido WHERE id_libro=%s AND id_sesion=%s"
    checkBook_tuple = (isbn, session)
    self.cursor.execute(checkBook_query, checkBook_tuple)
    available = self.cursor.fetchall()
    if available[0][0] > 0:
        return None
    #Getting book
    getBook_query = "SELECT * FROM libro WHERE isbn=%s"
    self.cursor.execute(getBook_query, (isbn, ))
    bookInfo = self.cursor.fetchall()

    #Getting user
    userId_query = "SELECT id from usuario where ip=%s"
    self.cursor.execute(userId_query, (userIP[0], ))
    userId = self.cursor.fetchall()

    #Insert request
    setRequest_query = "INSERT INTO pedido VALUES(%s, %s, %s, %s, %s, %s)"
    request_tuple = (self.nextRequest, session, userId[0][0], isbn, date.today(), reqHour)
    self.cursor.execute(setRequest_query, request_tuple)
    self.db.commit()
    self.nextRequest += 1

    newQuery = {
        "type": "book",
        "query": setRequest_query,
        "values": request_tuple
    }
    allData = {
        "bookInfo": bookInfo,
        "query": newQuery
    }
    return allData
```

- getCurrentSession(). Este método es esencial para mantener la consistencia entre las diferentes sesiones abiertas simultáneamente en el sistema. La sesión correspondiente es determinada de acuerdo a la IP del servidor que levanta la sesión, retornando el id de esta última.

```
def getCurrentSession(self, serverIP):
    checkSession_query = "SELECT id_sesion FROM sesion WHERE equipo=%s"
    self.cursor.execute(checkSession_query, (serverIP,))
    result = self.cursor.fetchall()
    return result[0][0]
```

- synchronizeDB(). Este método ejecutar los queries para sincronizar la BD. Recibe como parámetro el diccionario que se mencionó al principio para poder ejecutar las sentencias ya llevadas a cabo por el otro servidor y para mantener consistencia en los ids de las diferentes tablas.

```python
def synchronizeDB(self, query):
    self.cursor.execute(query["query"], query["values"])
    self.db.commit()
    q_type = query["type"]
    if q_type == "user":
        self.nextUser += 1
    elif q_type == "openSession":
        self.nextSession += 1
    elif q_type == "book":
        self.nextRequest += 1
```

Queries

Esta clase es la que contiene los métodos que serán invocados remotamente. A continuación se detalla su funcionamiento.

- Clase Queries. Esta clase es un objeto que se encuentra totalmente disponible para su invocación remota, por medio del comando @Pyro4.expose, es que se puede tener acceso a este. Esta clase cuenta con:
    - 2 listas, que bien podríamos catalogar como colas, pues es el funcionamiento necesario para esta aplicación.
      Cada una de ellas corresponde a cada uno de los servidores, los queries correspondientes del servidor 1, irán a su cola correspondiente y lo mismo para los del servidor 2.
    - Método setQuery(). Es el que, como se mencionó, encola las nuevas peticiones realizadas. Recibe como parámetros el identificador del servidor que está realizando la operación y el query que procede a encolar dependiendo del valor del primer parámetro.
    - Método getQuery(). Se encarga de retornar los queries acumulados publicados por los servidores. Recibe como único parámetro el identificador del servidor, el cual es esencial para un correcto funcionamiento.
      Primero se crea una lista vacía auxiliar. Ahora bien, dependiendo del servidor, dependerá de que cola se copiarán los valores a esta nueva lista. Como el servidor 1, sube sus queries en la cola queriesS1, entonces debe de leer los de la cola queriesS2, ya que son los publicados por el servidor 2, y a la inversa. Una vez copiada la cola a la recién creada, la cola correspondiente es vaciada y se procede a retornar al servidor la cola con los valores copiados.

```python
@Pyro4.expose
@Pyro4.behavior(instance_mode="single")
class Queries(object):
    def __init__(self):
        self.queriesS1 = []
        self.queriesS2 = []

    def getQuery(self, server):
        newQueries = []
        if server == 1:
            if len(self.queriesS2) > 0:
                newQueries = self.queriesS2.copy()
                self.queriesS2.clear()
        else:
            if len(self.queriesS1) > 0:
                newQueries = self.queriesS1.copy()
                self.queriesS1.clear()
        return newQueries

    def setQuery(self, query, server):
        if server == 1:
            self.queriesS1.append(query)
        else:
            self.queriesS2.append(query)
```

- En el main se tiene la inicialización del servidor que mantendrá registro de la clase Queries y permitirá que sea invocada remotamente. El primer parámetro es el Uri con el que se identificará a la clase para que sea invocada. Los posteriores nos indican en que IP y puerto estará escuchando el servidor para poder contestar las llamadas remotas.

```python
def main():
    Pyro4.Daemon.serveSimple(
        {Queries: "biblioteca.queries"},
        host="10.0.0.13",
        port=17000,
        ns=False
    )
```

Consideraciones finales

- Descarga el documento antes de llenarlo.
- Este documento debe ser enviado por cada integrante el equipo en formato PDF.
- Después de llenar el documento, guárdalo como PDF y envíalo a través del tema correspondiente en la plataforma *de aprendizaje virtual correspondiente*.
- Queda estrictamente prohibido cualquier tipo de plagio a otros equipos o grupos de este semestre o anteriores. En caso de incurrir en esta falta, se anulará la asignación correspondiente y se bajarán 2 puntos al (los) equipo (s) involucrados.
- Para el envío del video debe subirlo en alguna plataforma como YOUTUBE o en su ONEDRIVE. No lo suba directamente a CLASSROOM por los inconvenientes que ya hemos comentado en prácticas pasadas debe agregarlo en la portada de este documento.