INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

REDES DE COMPUTADORAS

PROF. AXEL ERNESTO MORENO CERVANTES

GRUPO: 3CM7

PRÁCTICA 3 – CHAT MULTICAST NO BLOQUEANTE

ALEJANDRO DE JESÚS ZEPEDA FLORES

15 de junio de 2020

Prácticas: https://drive.google.com/drive/folders/12lUUJImgU72oXJfukVcbrB8OjSsMSkb2?usp=sharing

Práctica 3: https://drive.google.com/open?id=1fq7fYLCbXQpww6fS1ziLMeFKn5D6O\_nl

OBJETIVOS

Implementar un servicio de Chat que permita a los usuarios:

• Enviar y recibir mensajes públicos o privados.

• Enviar y recibir archivos vía multicast.

• Enviar emojis o gifs.

Con la variante de que la comunicación se debe realizar utilizando sockets no bloqueantes.

INTRODUCCIÓN

Las operaciones de entrada/salida son por lo general bloqueantes y esto también ocurre cuando se utilizan con sockets. Esto implica que cuando se realiza alguna de estas operaciones sobre un socket, el proceso pasa al estado dormido, esperando que se satisfaga alguna condición que permita que se complete la operación. Así:

* Si realizamos operaciones de entrada (read, recv, recvfrom) sobre un socket TCP y no hay datos disponibles en el buffer de recepción del socket, el proceso pasa a estado dormido hasta que lleguen datos.
* Si realizamos operaciones de salida (write, send, sendto) sobre un socket TCP, cuando nosotros realizamos esta llamada, el kernel copia los datos del buffer de la aplicación en el buffer de envío del socket, si no queda espacio en este buffer, el proceso se bloquea hasta que haya suficiente espacio.

En muchas ocasiones es recomendable emplear algún mecanismo que nos permita realizar estas operaciones de forma no bloqueante y así poder realizar otras tareas en vez de esperar a que los datos estén disponibles. En esta práctica vamos a ver algunos de los mecanismos que existen para poder realizar operaciones de entrada/salidas no bloqueantes sobre sockets, en concreto, JAVA NIO.

Java NIO (New IO) es una API IO alternativa para Java (de Java 1.4), lo que significa una alternativa a las API estándar Java IO y Java Networking. Java NIO ofrece una forma diferente de trabajar con IO que las API estándar de IO.

* **Canales y bufferes:** En la API IO estándar, trabaja con secuencias de bytes y secuencias de caracteres. En NIO trabajas con canales y buffers. Los datos siempre se leen desde un canal a un búfer, o se escriben desde un búfer a un canal.
* **Non-blocking IO:** Java NIO le permite hacer IO sin bloqueo. Por ejemplo, un hilo puede pedirle a un canal que lea datos en un búfer. Mientras el canal lee datos en el búfer, el hilo puede hacer otra cosa. Una vez que los datos se leen en el búfer, el hilo puede continuar procesándolo. Lo mismo es cierto para escribir datos en canales.
* **Selectores:** Un selector es un objeto que puede monitorear múltiples canales para eventos (como: conexión abierta, datos recibidos, etc.). Por lo tanto, un solo hilo puede monitorear múltiples canales de datos.

DESARROLLO

Iniciamos con el condigo del Servidor, como primer paso debemos implementar una Interfaz de red. Una interfaz de red puede considerarse como un punto en el cual su computadora se conecta a la red. No es necesariamente una pieza de hardware, sino que también se puede implementar en un software. Por ejemplo, una interfaz de bucle invertido que se utiliza con fines de prueba.

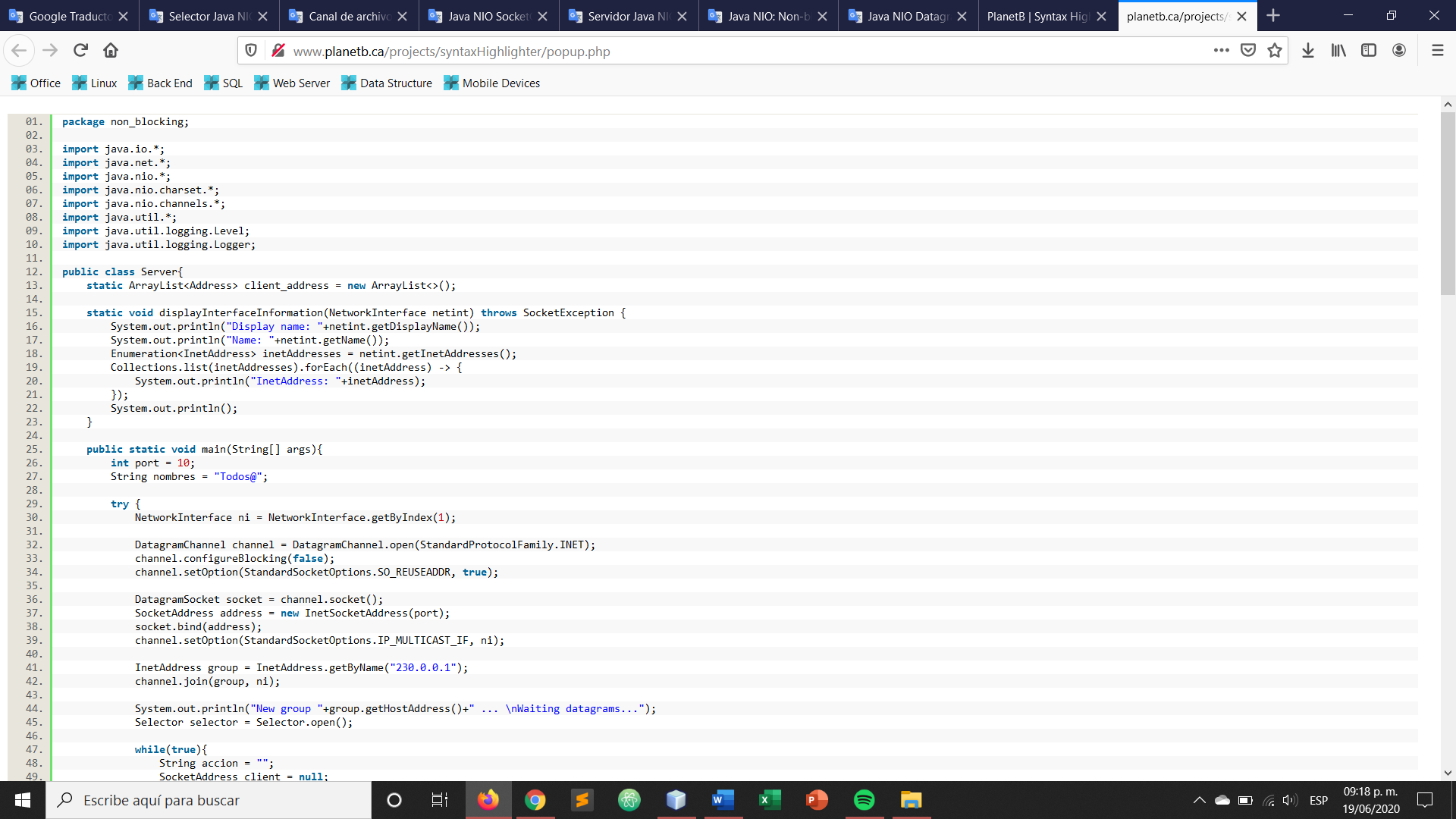
* **Network Interface:** Esta clase representa la interfaz de red, tanto el software como el hardware, su nombre, la lista de direcciones IP asignadas y toda la información relacionada. Se puede usar en casos en los que queremos usar específicamente una interfaz particular para transmitir nuestro paquete en un sistema con múltiples NIC.
* **Datagram Channel:** es un canal que puede enviar y recibir paquetes UDP. Dado que UDP es un protocolo de red sin conexión, no puede simplemente leer y escribir de manera predeterminada Datagram Channel como lo hace desde otros canales. En su lugar, envía y recibe paquetes de datos.
* **Selector:** es un componente que puede examinar una o más instancias de Java NIO Channel y determinar qué canales están listos para, por ejemplo, leer o escribir. De esta manera, un solo hilo puede gestionar múltiples canales y, por lo tanto, múltiples conexiones de red.

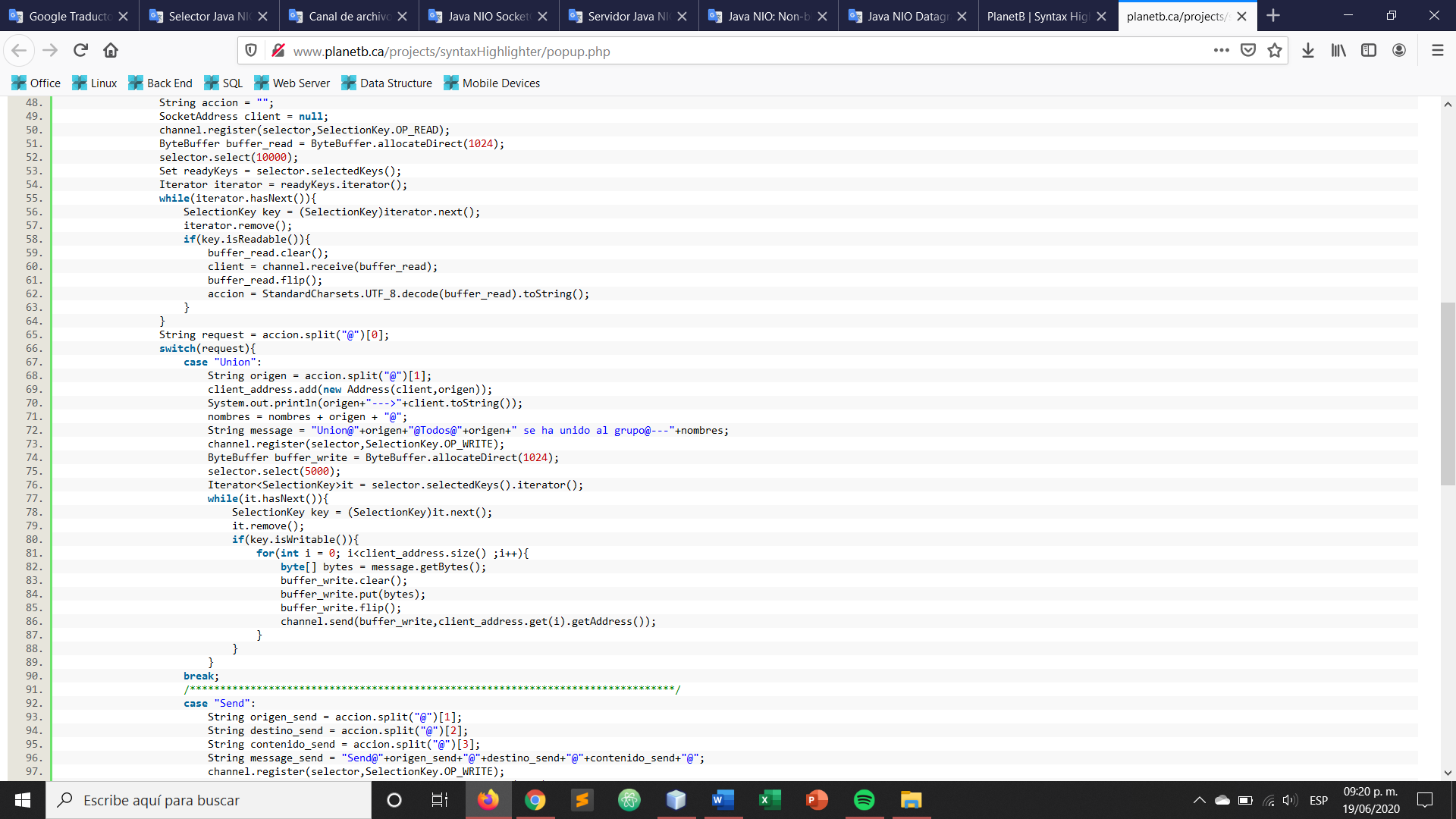
¿Por qué usar un selector?

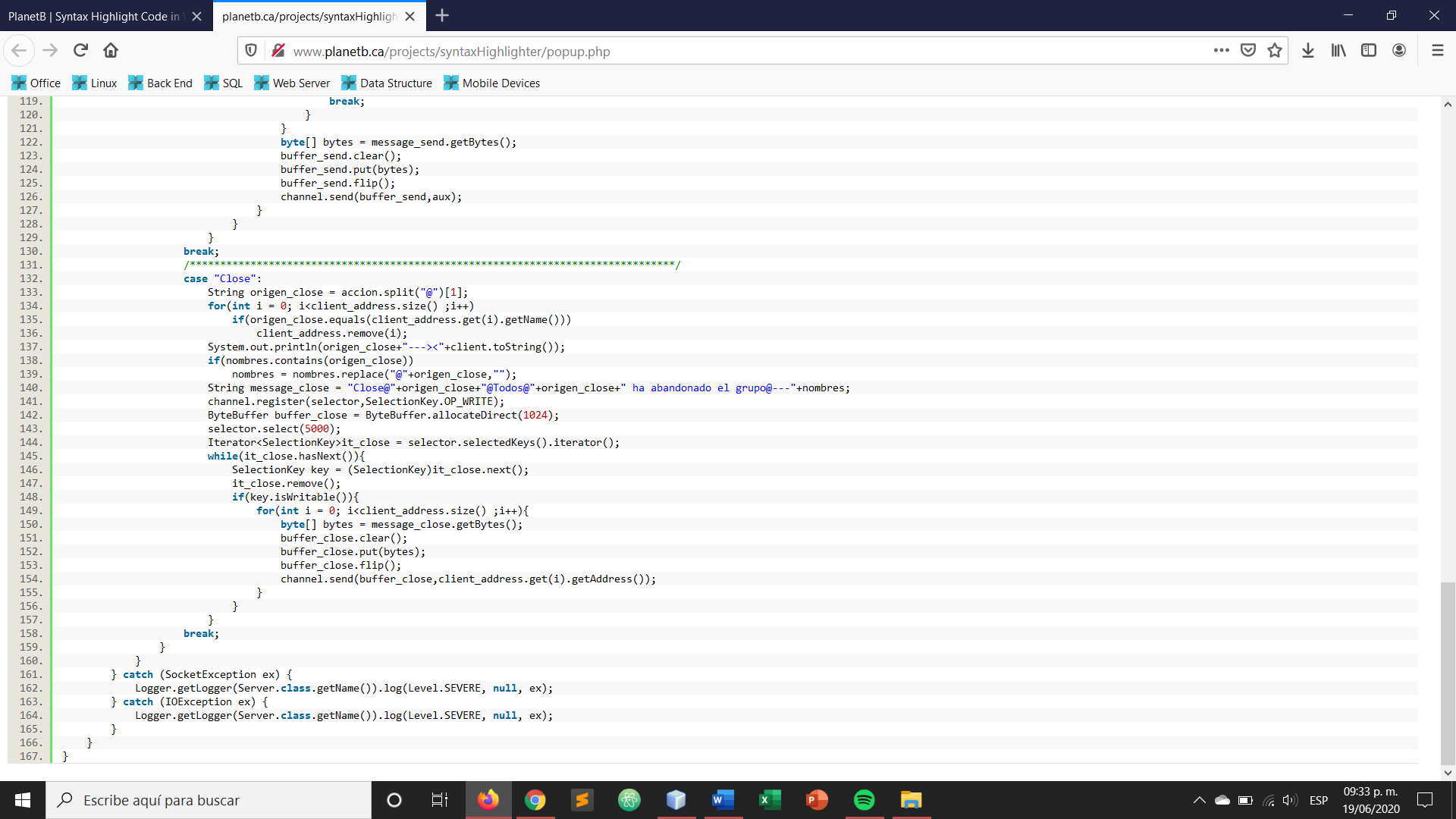
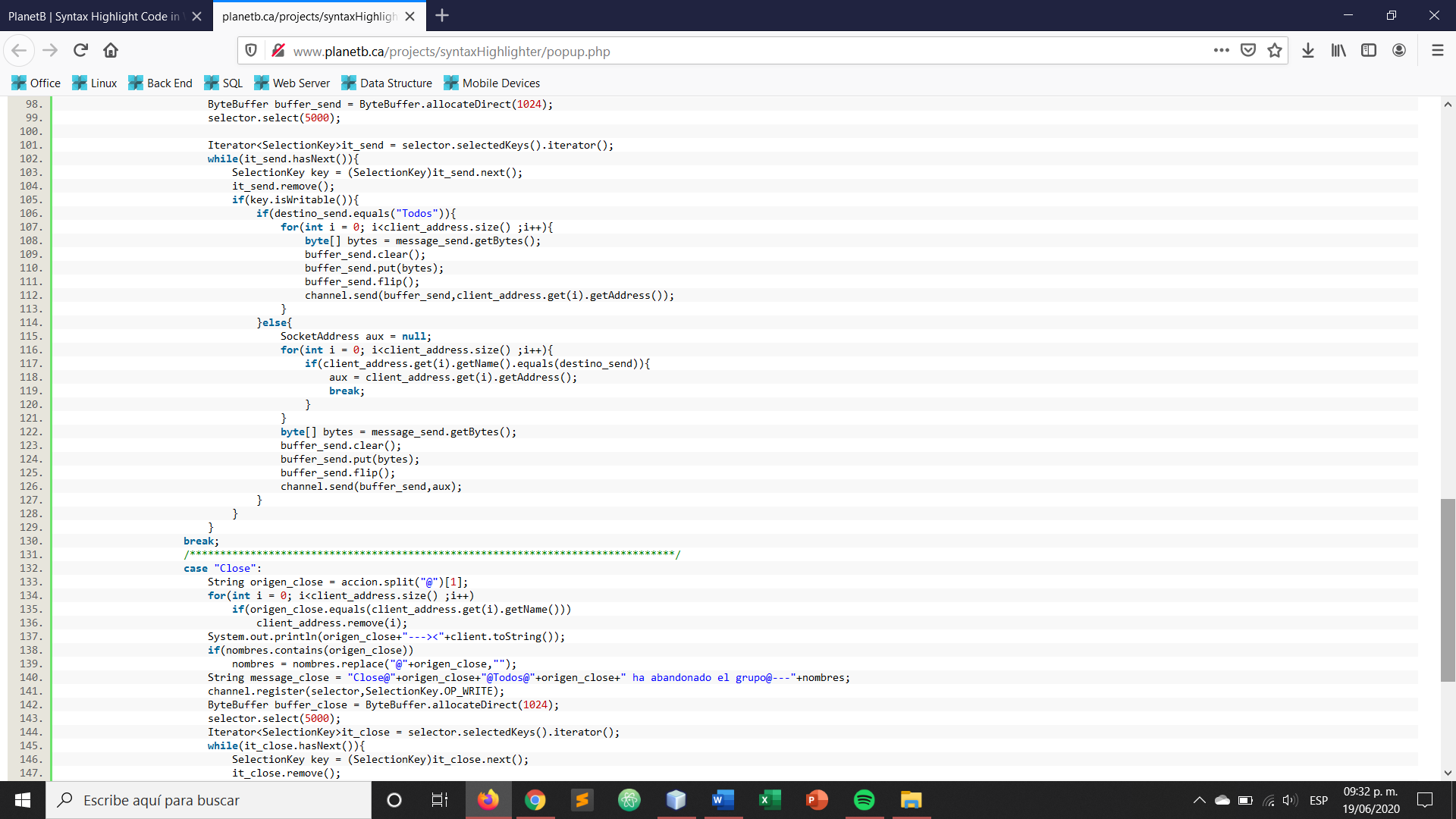
La ventaja de usar un solo hilo para manejar múltiples canales es que necesita menos hilos para manejar los canales. En realidad, puede usar solo un hilo para manejar todos sus canales. Cambiar entre subprocesos es costoso para un sistema operativo, y cada subproceso también consume algunos recursos (memoria) en el sistema operativo. Por lo tanto, cuantos menos hilos uses, mejor.

* **Inet Address:** representa una dirección IP. La clase java.net.InetAddress proporciona métodos para obtener la IP de cualquier nombre de host. Además, InetAddress tiene un mecanismo de caché para almacenar resoluciones de nombre de host exitosas y no exitosas.

A continuación, se muestra el código del servidor.



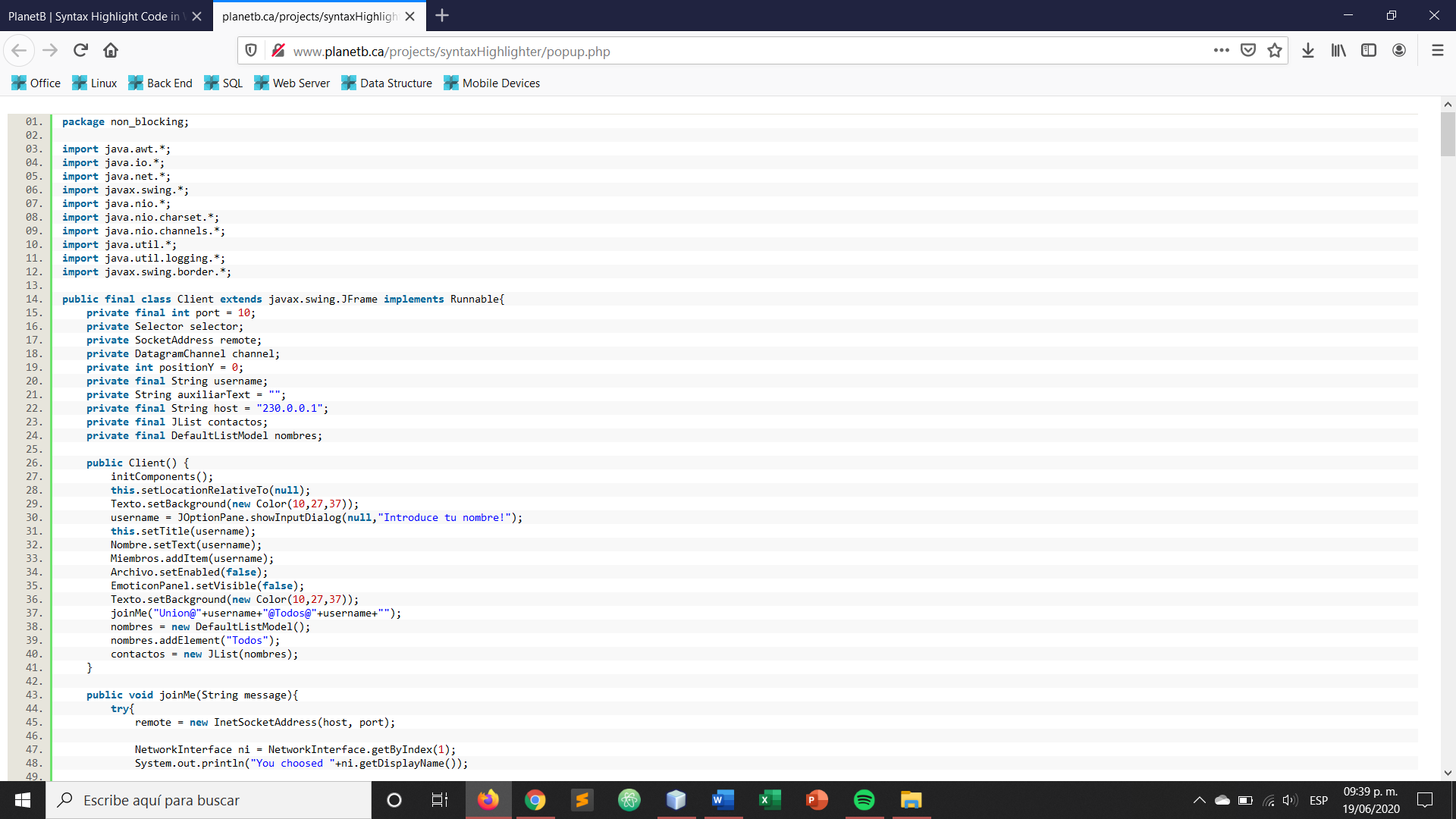
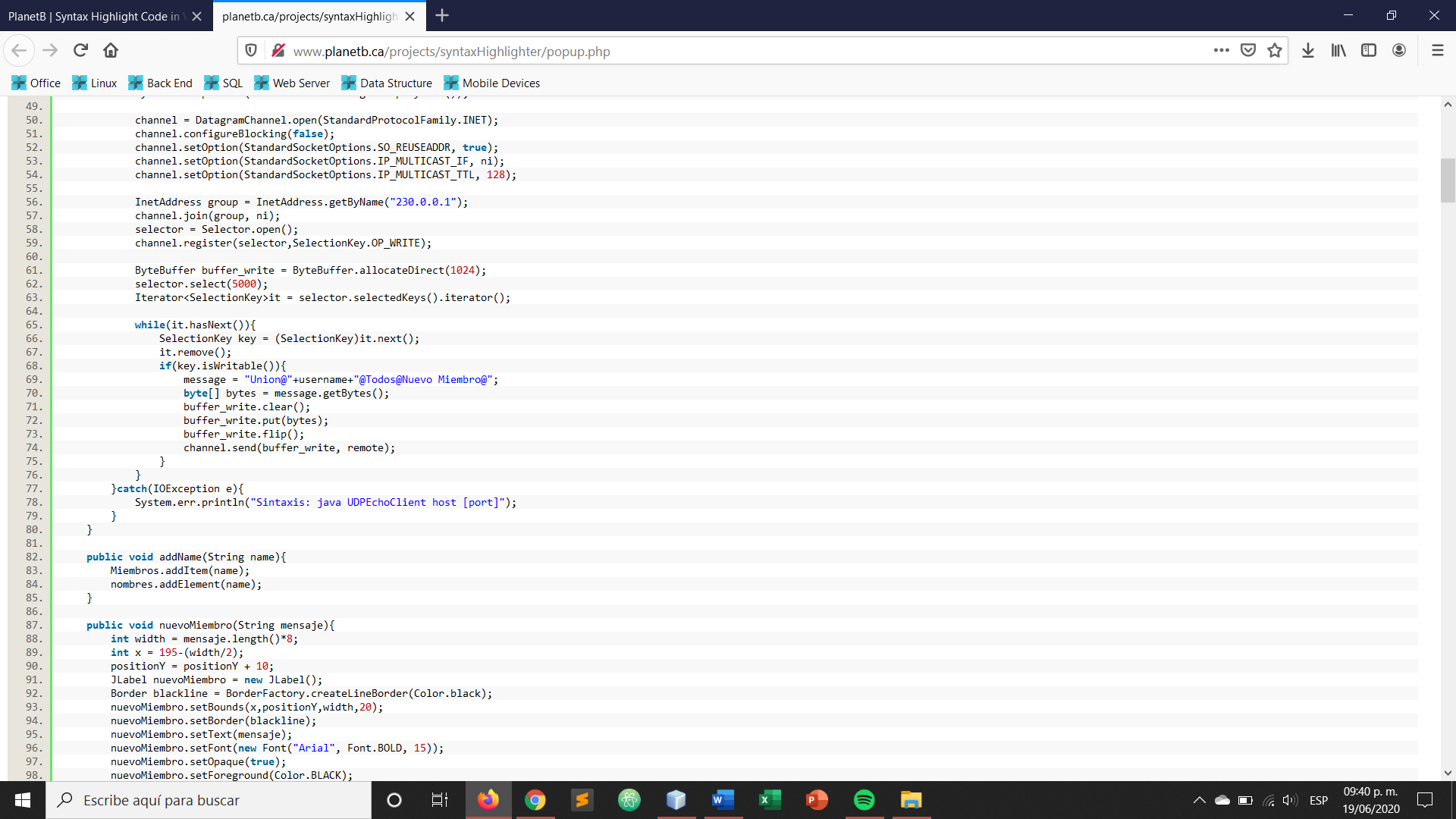




La lógica de programación es igual que a la práctica de chat multicast, sin embargo, la única diferencia es la manera en la que se transmiten los datos. Realmente, la parte importante es abrir un Datagram Channel en una dirección multicast, configurarla para que sea no bloqueante y cambiarlo su método (lectura o escritura) dependiendo de la acción que vamos a realizar.

Ahora vamos a explicar la parte del cliente.

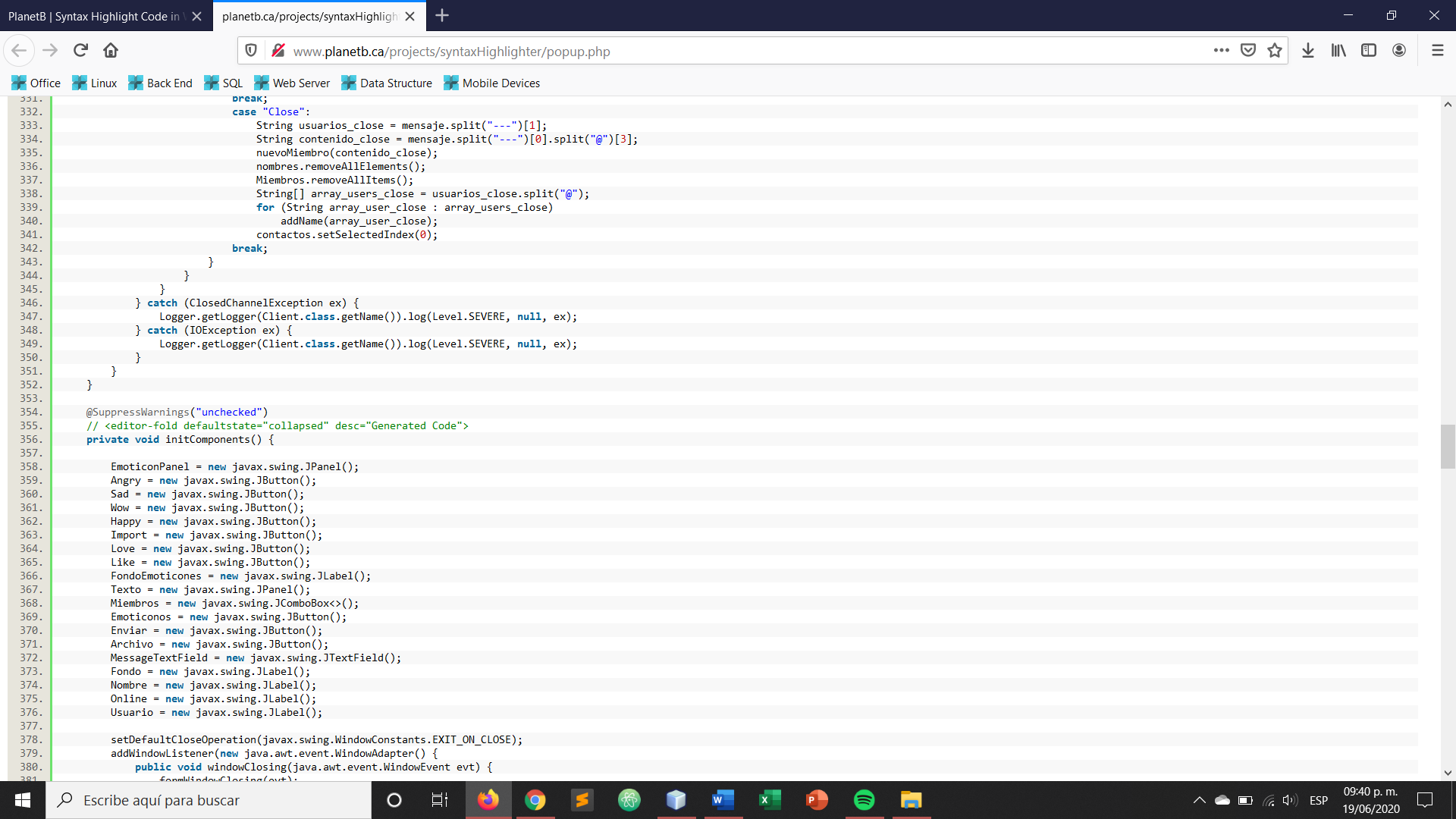
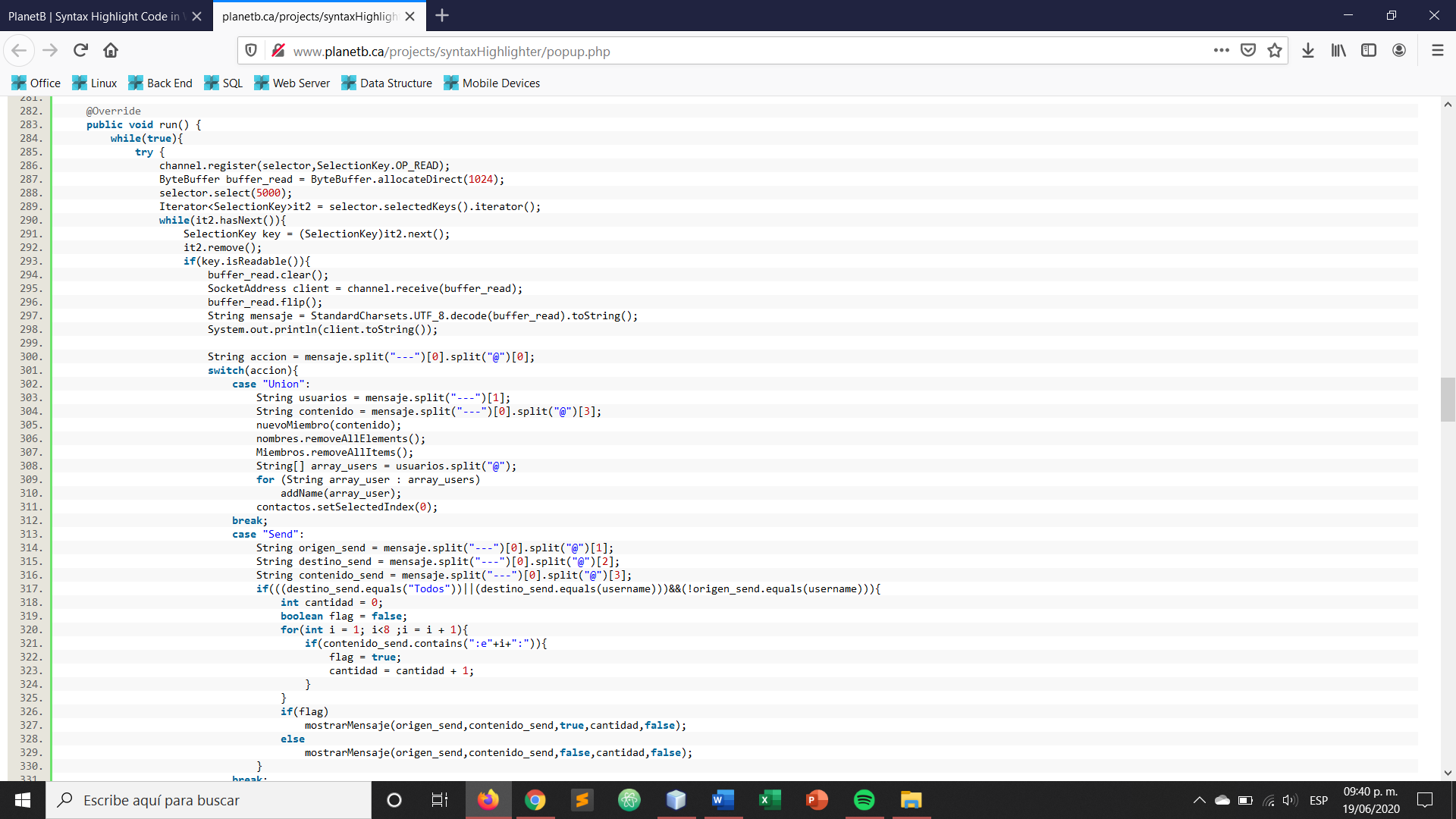
La parte de la interfaz es la misma de la práctica pasada, se reutilizó el código y al igual que el servidor, lo que cambio, fue la manera de transmisión de los datos.

Al igual que en el servidor, se creo un Datagram Channel, con la excepción de que está vez es global, como la comunicación es asíncrona, se puede utilizar en distintas partes del programa mientras se realizan otras actividades.

Para enlazar el canal del cliente con el del servidor, se realiza lo siguiente:

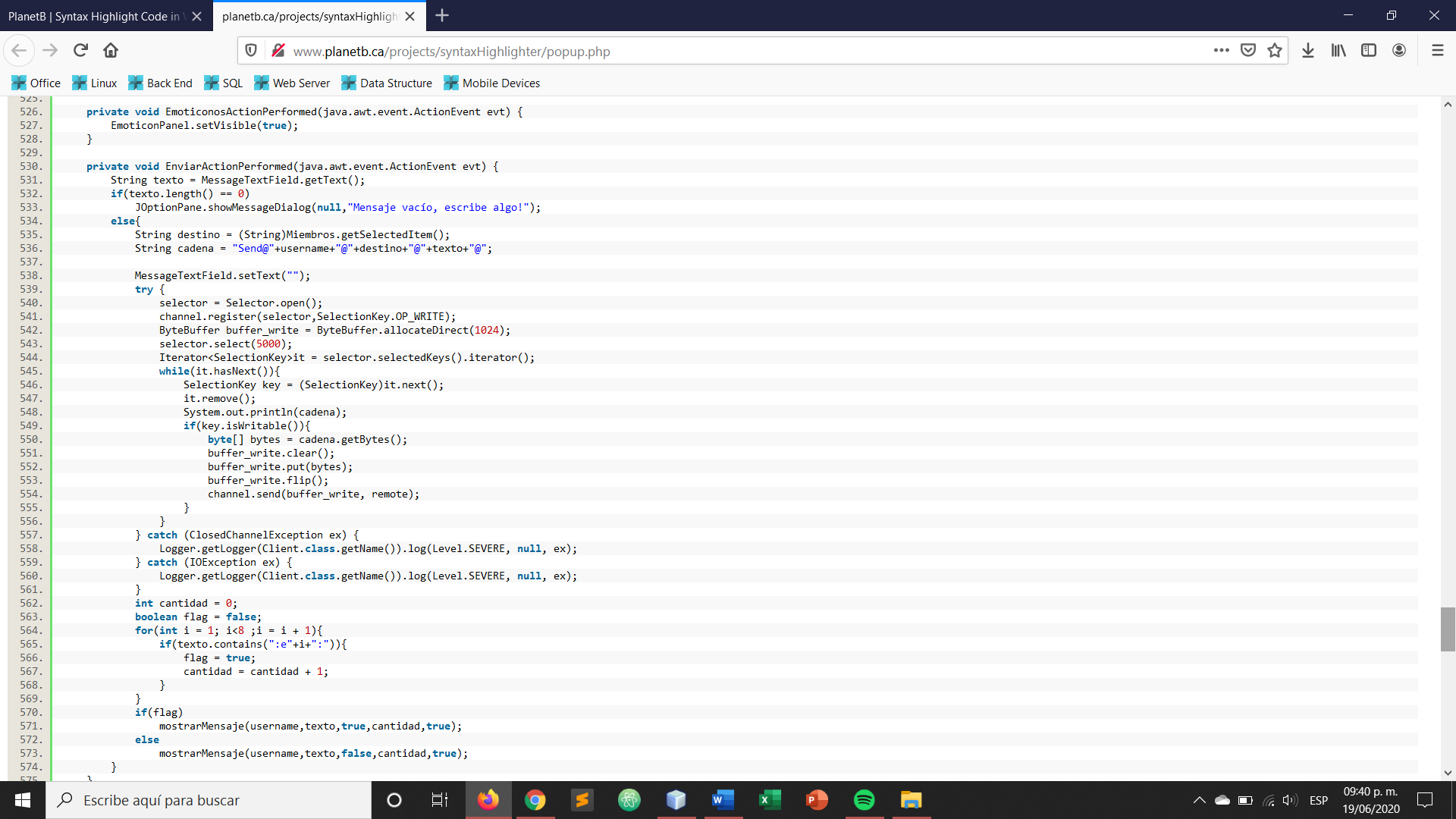
* Asignar la interfaz de red al canal.
* Asociar el canal al grupo creado por el servidor en la dirección “230.0.0.1”.



En esta parte del código, se encuentra el verdadero funcionamiento del programa, ya que es el que va a administrar las comunicaciones que se reciben del servidor y gestionar la acción necesaria para cada cliente.

Primero, iniciamos un ciclo while infinito para estar siempre escuchando la respuesta del servidor, después nuestro canal será registrado como de lectura y leemos el mensaje que corresponde a una cadena de texto y dentro de su contenido, se muestra la acción a realizar:

1. **Unión:** nuevo usuario en el grupo.
2. **Envío:** un usuario envió un mensaje.
3. **Cierre:** un usuario abandono el grupo.



Esa función es importante porque es la que se encarga de leer el mensaje que el usuario ingresa y enviárselo al servidor, además de que actualiza la parte gráfica y muestra el mensaje en pantalla.



Por último, está función se ejecuta cuando se cierra la ventana, significando que el usuario abandona el grupo. Envía un mensaje al servidor notificando el abandono para actualizar los contactos y muestra el mensaje en pantalla.

FUNCIONAMIENTO DE LA APLICACIÓN

A continuación, mostraremos un breve ejemplo de cómo funciona nuestra aplicación.

Esta es la pantalla principal, en el encabezado se muestra el nombre de usuario que previamente ingresaste. En la esquina superior derecha se observa un Combo Box donde se enlistan los nombres de todos los usuarios, se puede seleccionar un usuario individual o todos.

La parte central es en donde se irán mostrando los mensajes y notificaciones para el usuario.

En la parte inferior, se muestra un botón con un ícono de emoticono, al ser presionado se mostrará una lista de emoticones disponibles para enviar. Después aparece un Text Field para poder escribir el mensaje, seguido de un botón con un ícono de Clip, este se utiliza para poder mandar archivos. Por último, el botón de enviar.

Para demostrar el funcionamiento, es necesario ingresar al menos 3 usuarios. Como se muestra en la imagen 1, a cada uno de los usuarios se les notifica si alguien se unió al grupo a partir de que ellos se unieron.

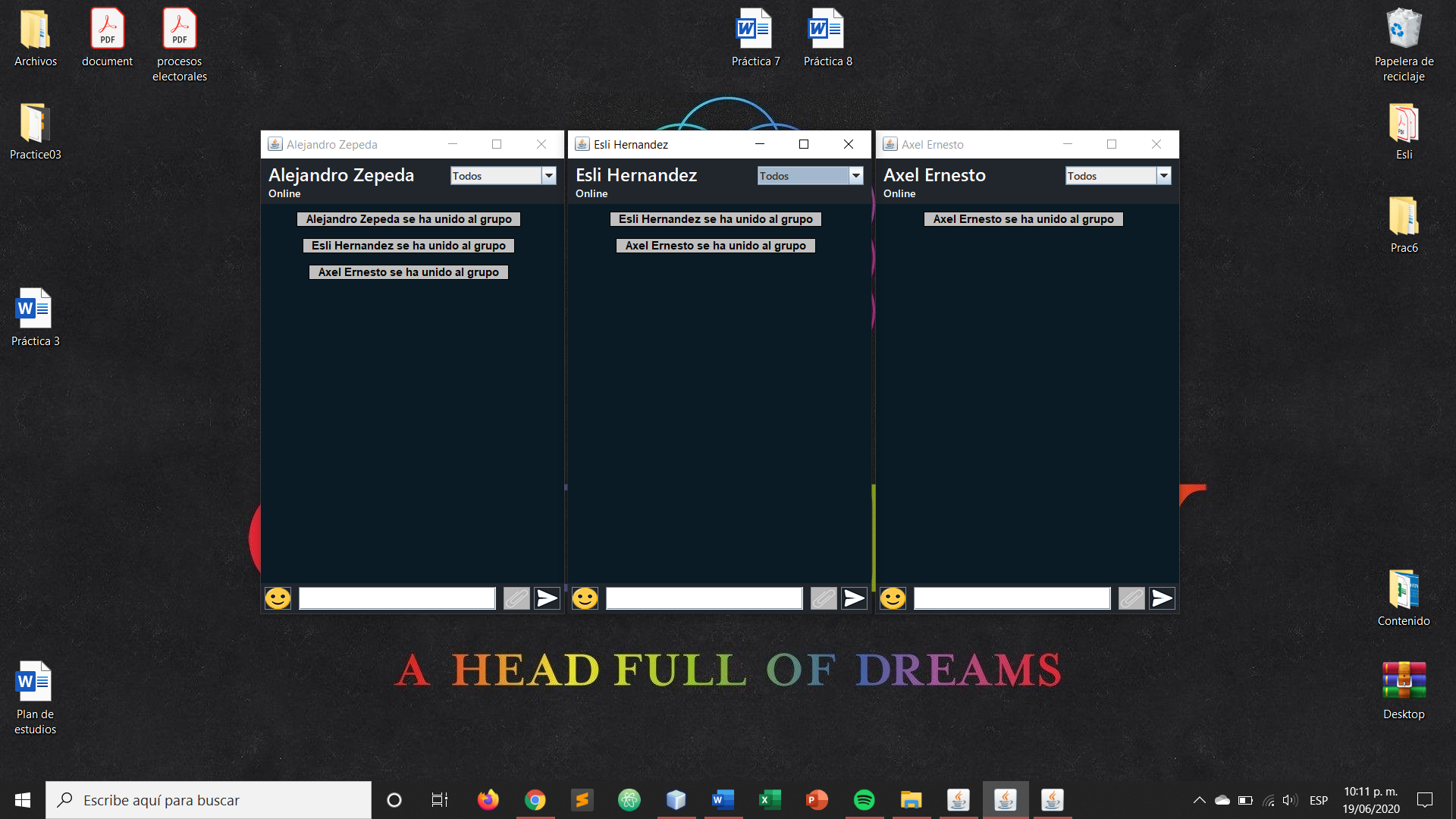


Imagen 1. Creación del grupo.

Como se observa en la imagen 2, aquí se puede apreciar el envío de un usuario a todos los destinos posibles. De igual manera, en el usuario central se enlistan las reacciones posibles que puedes agregar en un mensaje. Por último, en el usuario final se muestra el Combo Box con los usuarios disponibles para mandar mensaje.



Imagen 2. Envío de mensajes con reacciones.

En la imagen 3, se muestra como el usuario inicial se comunica con el usuario central; existe un intercambio de mensajes, sin que el usuario final se entere de esta conversación. De igual manera, vemos como el usuario final se comunica con todos los usuarios.



Imagen 3. Envío de mensajes personalizados.

Por último, en la imagen 4, se muestra como el usuario final ha abandonado la conversación. El programa notifica al resto de los usuarios con un mensaje en la aplicación y de igual manera, en el Combo Box de los destinos posibles se actualiza para evitar confusión a la hora de enviar mensajes.

Además, en el usuario inicial, se muestra como el número de mensajes supero el tamaño inicial del panel, por lo que se redimensiona automáticamente y agrega un Scroll para poder visualizar los nuevos mensajes y no perder los anteriores.



Imagen 4. Abandono de usuario.

CONCLUSIÓN

El uso de sockets no bloqueantes facilita la comunicación entre el cliente y servidor, ya que permite que otras actividades se ejecuten y no bloquear el programa esperando la respuesta de algún usuario.

Dentro de la parte de interfaz gráfica, se reutilizó el código de la práctica de sockets bloqueantes, el verdadero problema fue la implementación de los sockets, ya que utilizar JAVA NIO fue algo complicado, me basé en el código del profesor pero si fue un verdadero reto, ya que como la comunicación es asíncrona, tuve que encontrar la forma de que la lectura de datos se realizará en cierto tiempo, de lo contrario se ejecutaba al mismo tiempo que otras actividades sin datos que leer y era un error seguro.