

Planteamiento del problema

Desarrollar un programa en Java, cual implementará un token (un número entero de 16 bits) que se enviará de un nodo a otro nodo mediante sockets seguros, en una topología lógica de anillo.

El anillo consta de seis nodos:

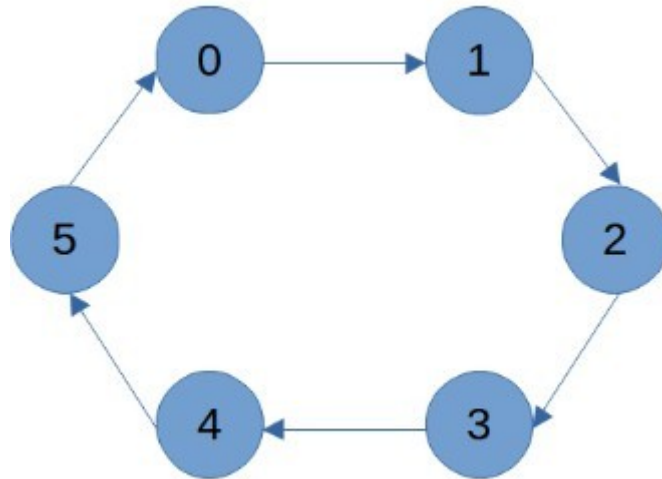


Imagen 1. Topología lógica de anillo de 6 nodos.

Desarrollo de la tarea

Para empezar primero explicaremos las variables globales que usaremos en el programa y su función dentro del programa:

- `DataStream salida` Es el canal de salida para que los clientes puedan enviar el valor actual del token al servidor con el que les corresponde establecer conexión de acuerdo al orden que llevan por ejemplo en el caso del cliente del nodo 1 le tocaría conectarse con el servidor del nodo 2.
- `DataStream entrada` Es el canal de entrada para que los servidores puedan recibir el token por parte del cliente, que de acuerdo a la topología de anillo, el cliente es un número de nodo anterior al del servidor, es decir, el cliente del nodo N se conecta al servidor N+1.
- `short token` Es el token que se compartirá entre los nodos hasta que este llegue al valor de 500.
- `boolean inicio` Esta variable la usaremos al inicio para saber que se acaba de ejecutar nuestro nodo, esto con la finalidad de que se trate del nodo cero, este proceda a enviar el token al nodo siguiente a este, que en este caso sería el nodo 1.
- `int nodo` Esta variable la usamos para guardar el número de nodo que recibimos como argumento al inicio de ejecución del programa.

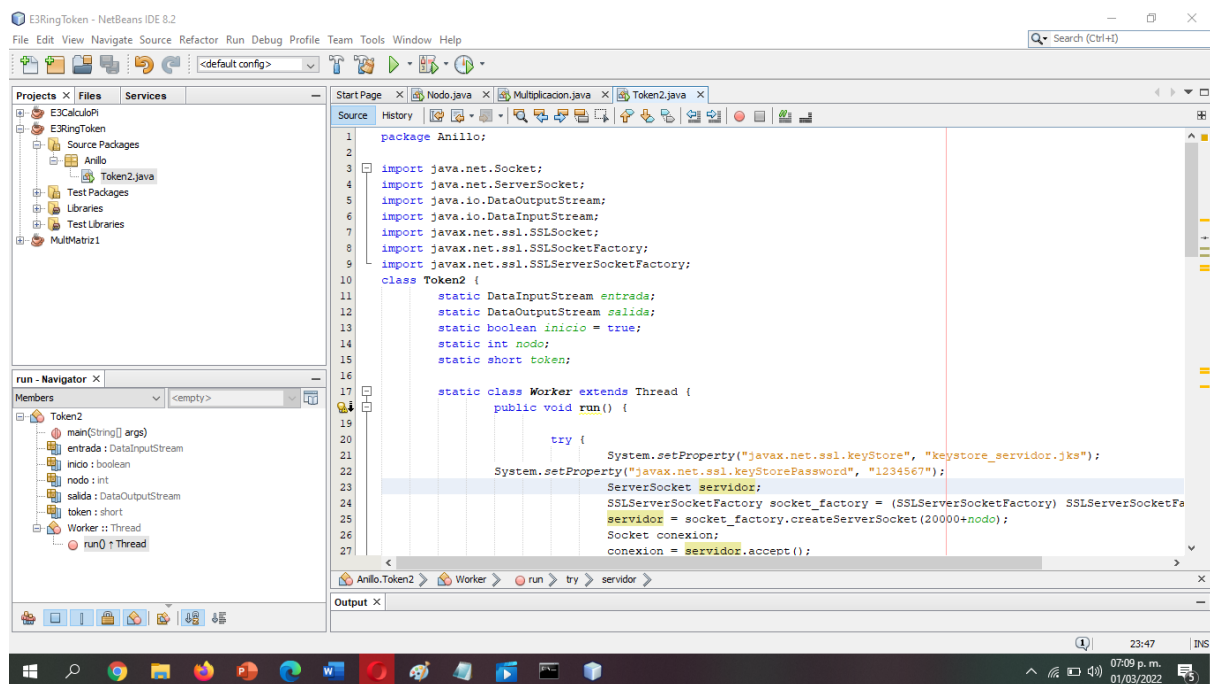
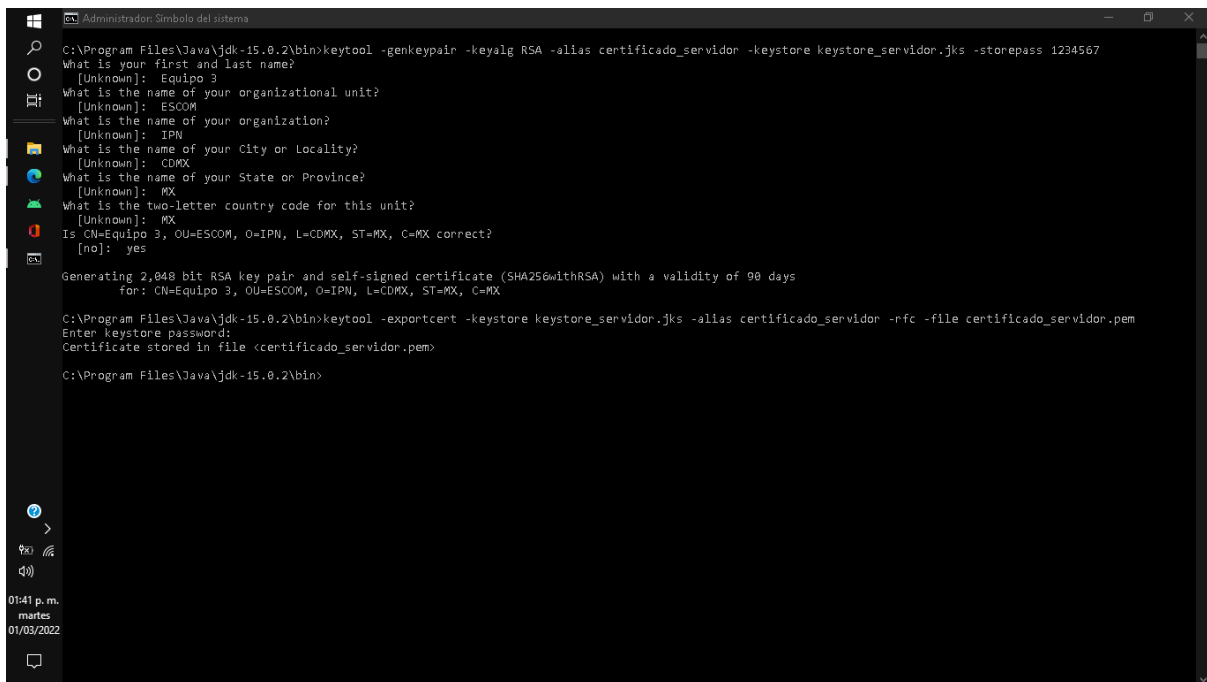


Imagen 2. Captura de las variables globales a utilizar en la práctica.

Generación de las keystore del cliente y del servidor

Antes de continuar procedemos a crear las llaves para nuestro cliente (`keystore_cliente.jks`) y servidor (`keystore_servidor.jks`), recordando que la llave del cliente debe tener la misma contraseña.

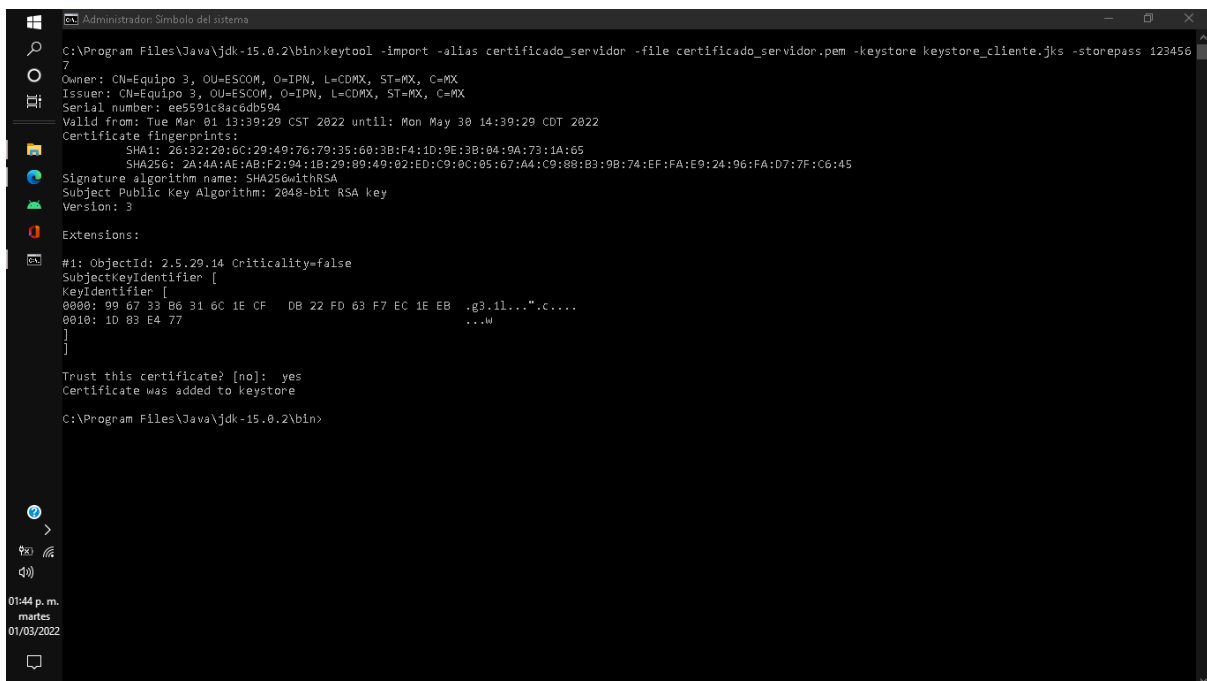
Servidor:



```
Administrador: Símbolo del sistema
C:\Program Files\Java\jdk-15.0.2\bin>keytool -genkeypair -keyalg RSA -alias certificado_servidor -keystore keystore_servidor.jks -storepass 1234567
What is your first and last name?
[Unknown]: Equipo 3
What is the name of your organizational unit?
[Unknown]: ESCOM
What is the name of your organization?
[Unknown]: IPN
What is the name of your City or Locality?
[Unknown]: CDMX
What is the name of your State or Province?
[Unknown]: MX
What is the two-letter country code for this unit?
[Unknown]: MX
Is CN=Equipo 3, OU=ESCOM, O=IPN, L=CDMX, ST=MX, C=MX correct?
[no]: yes
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 90 days
for: CN=Equipo 3, OU=ESCOM, O=IPN, L=CDMX, ST=MX, C=MX
C:\Program Files\Java\jdk-15.0.2\bin>keytool -exportcert -keystore keystore_servidor.jks -alias certificado_servidor -rfc -file certificado_servidor.pem
Enter keystore password:
Certificate stored in file <certificado_servidor.pem>
C:\Program Files\Java\jdk-15.0.2\bin>
```

Imagen 3. Generación de la keystore del servidor.

Cliente:



```
Administrador: Símbolo del sistema
C:\Program Files\Java\jdk-15.0.2\bin>keytool -import -alias certificado_servidor -file certificado_servidor.pem -keystore keystore_cliente.jks -storepass 1234567
Owner: CN=Equipo 3, OU=ESCOM, O=IPN, L=CDMX, ST=MX, C=MX
Issuer: CN=Equipo 3, OU=ESCOM, O=IPN, L=CDMX, ST=MX, C=MX
Serial number: ee5591c8ac6db594
Valid from: Tue Mar 01 13:39:29 CST 2022 until: Mon May 30 14:39:29 CDT 2022
Certificate fingerprints:
  SHA1: 26:32:20:6C:29:49:76:79:35:60:3B:F4:1D:9E:3B:04:9A:73:1A:65
  SHA256: 2A:4A:AE:AB:F2:94:1B:29:89:49:02:ED:C9:0C:05:67:A4:C9:88:B3:9B:74:EF:FA:E9:24:96:FA:D7:7F:C6:45
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3
Extensions:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 99 67 33 B6 31 6C 1E CF DB 22 FD 63 F7 EC 1E EB .g3.11...".c....
0010: 1D B3 E4 77 ...M
]
]
Trust this certificate? [no]: yes
Certificate was added to keystore
C:\Program Files\Java\jdk-15.0.2\bin>
```

Imagen 4. Generación de la keystore del cliente.

Clase Worker

La clase worker es una subclase de la clase Hilo la cual utilizamos para inicializar nuestros servidores para lo que es necesario antes especificarle al programa mediante el método `setProperty` que utilizaremos un certificado el cual se encuentra en la llave del servidor (`keystore_servidor.jks`) y también indicando la contraseña con la cual se encriptó. Después creamos una instancia de la clase `ServerSocket` llamada `servidor` y una instancia de la clase `SSLServerSocketFactory` (`socket_factory`) para iniciar la creación de nuestro socket seguro usando como ip (`localhost`) y como puerto al numero 20000 al cual se le sumará el número de nodo ya que al estar todos los servidores ejecutándose en el mismo equipo, cada uno tendrá que usar un puerto distinto para distinguirlos y saber a qué servidor se quiere conectar el cliente. El servidor lo usaremos para capturar el socket que nos regresa el método `createServerSocket` de la clase `SSLServerSocketFactory` una vez hecho esto procedemos a llamar al método `accept` para poner a esperar al servidor hasta que logre establecer conexión con un cliente y posteriormente creamos el de entrada para poder recibir el token del cliente con el que establezca conexión.

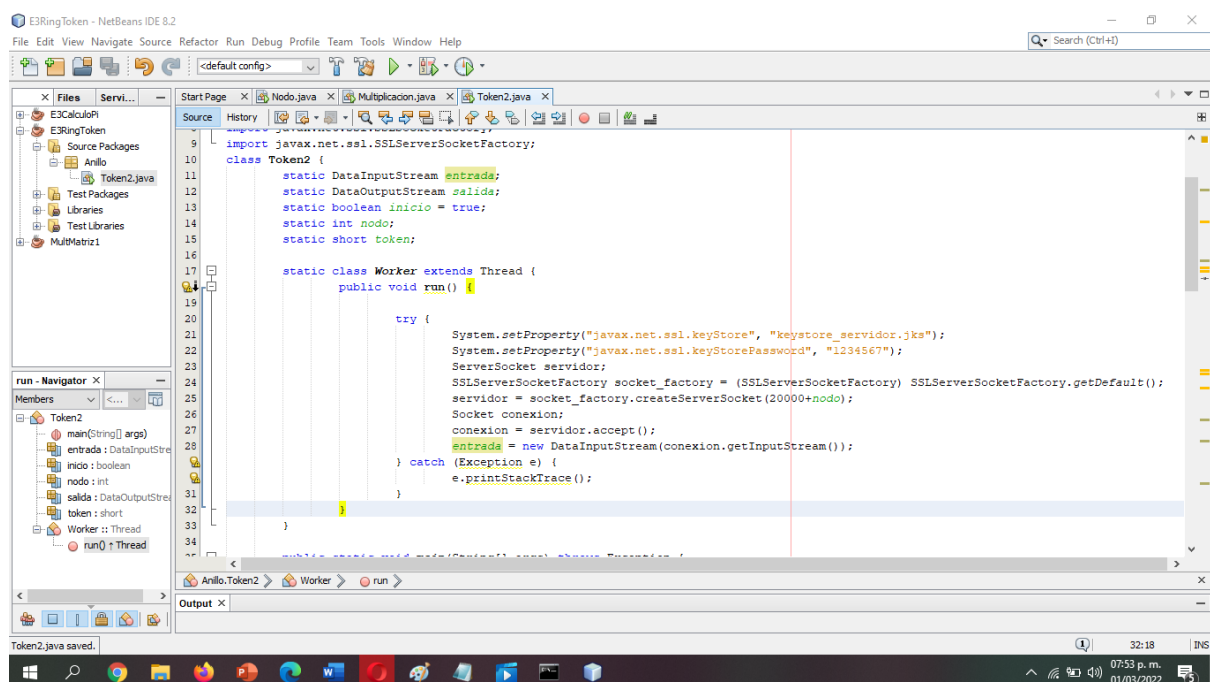


Imagen 5. Clase Worker para crear los servidores seguros usando una keystore y una contraseña, además de que el servidor es multi thread.

Metodo main

Empezando con el main principal, primero que hacemos es verificar que se haya pasado el nodo como argumento ya que en caso contrario se terminará la ejecución del programa. También es necesario recordar que debemos establecer las propiedades dentro de lo que es nuestro programa para que se pueda hacer uso de la llave que creamos para nuestro cliente

y la otra propiedad la usamos para notificarle al programa la contraseña del cliente. Posteriormente de conseguir el número de nodo procedemos a crear una instancia de la clase worker, la cual como vimos se trata de un servidor multithread y procedemos a mandar a llamar al método start para inicializar nuestro servidor. Después mantenemos ciclado a nuestro hilo principal (el main), en donde trataremos de que el cliente del nodo que estemos ejecutando establezca conexión con el servidor que le corresponde e igualmente a como se hizo con el servidor se le especifica al programa que se hará uso de una llave y una contraseña para tener una conexión segura, en caso de no lograr establecer conexión se espera medio segundo antes de volver a intentar conectarse al servidor que le corresponde al cliente del nodo que se esté ejecutando y una vez se haya logrado establecer la conexión creamos nuestro canal de salida para que el cliente pueda mandar el token a su servidor. Al llamar al método join de nuestra clase worker, estamos permitiendo que los nodos pueden ejecutarse en cualquier orden, pero recordemos que para haber llegado hasta ese punto del código los clientes debieron lograr conexión con el servidor del nodo siguiente, por ejemplo el cliente 1 no puede conectarse hasta que el servidor 2 este en línea, por lo que hasta que no hayamos inicializado cada uno de nuestros nodos nos mantendremos en esta barrera esperando hasta que esto pase una vez todos estén en línea. Ciclamos al programa en un bucle infinito del cual solo se podrá salir una vez que el token haya alcanzado el valor de 500. Estando ya dentro del ciclo verificamos si el nodo que está en ejecución es el nodo 0, ya que recordemos que el nodo 0 es el que inicia enviando el token, por lo que en caso de ser este tenemos que checar además si inicio es verdadero, ya que recordemos que el valor inicial de esta variable es true y por lo tanto le asignamos a token el valor de 1. Esto lo hacemos así que el nodo 0 tiene que enviar el token una vez que inicie, mientras que el resto está en espera a leer el valor del token enviado por el cliente y la razón de usar esa condición con la variable inicio debe a que el programa estar ciclado si nosotros no ponemos esta condición cada vez que el nodo 0 regrese al inicio del bucle volvería a mandar al token con el valor de 1 y el programa nunca acabaría, por lo que esta condición nos permite que el nodo solo envíe el valor de 1 la primera vez y después de esto tenga el mismo comportamiento que el resto de los nodos, donde se ponen en espera a leer el valor que les envíe el cliente con ayuda del método read Short y posteriormente incrementamos en 1 al token. Antes de proseguir verificamos si el nodo en ejecución es el 0 y si el valor del token es 500, en caso afirmativo procedemos a terminar el programa y en caso contrario procedemos a mandar el token al servidor que le corresponde al cliente de nuestro nodo. Esto pasa debido a que solo el nodo 0 puede terminar su ejecución y en caso de que el token no haya alcanzado el valor de 500 o superior el nodo 0 se comportará igual que el resto de los nodos, recibiendo el valor del token para después incrementarlo en 1 y enviarlo al servidor que le corresponde antes de volver a empezar otra vez el ciclo.

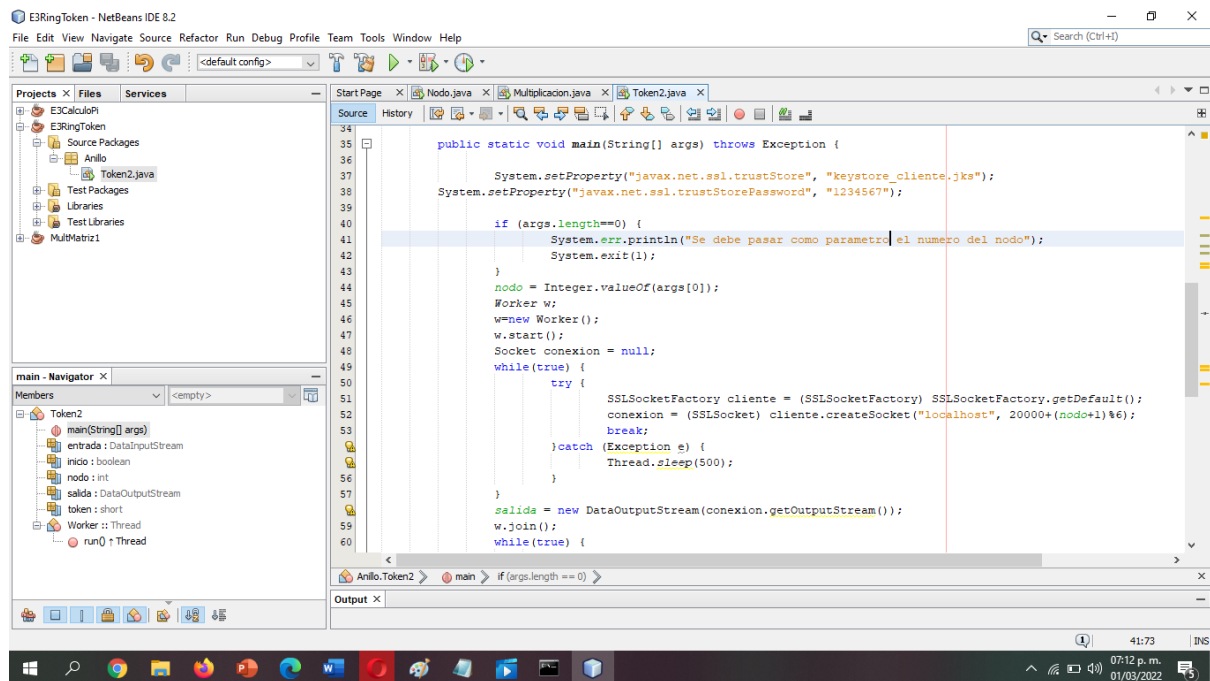


Imagen 6. Primera parte del método Main.

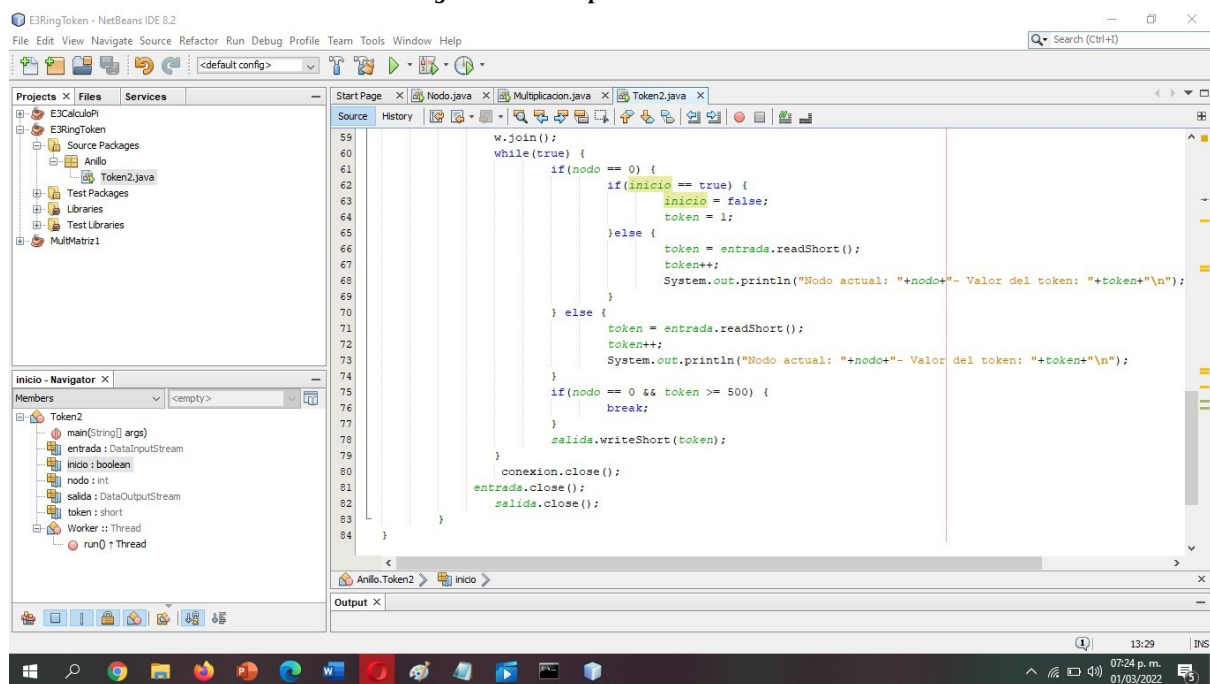


Imagen 7. Segunda parte del método Main.

Compilacion y ejecucion del programa

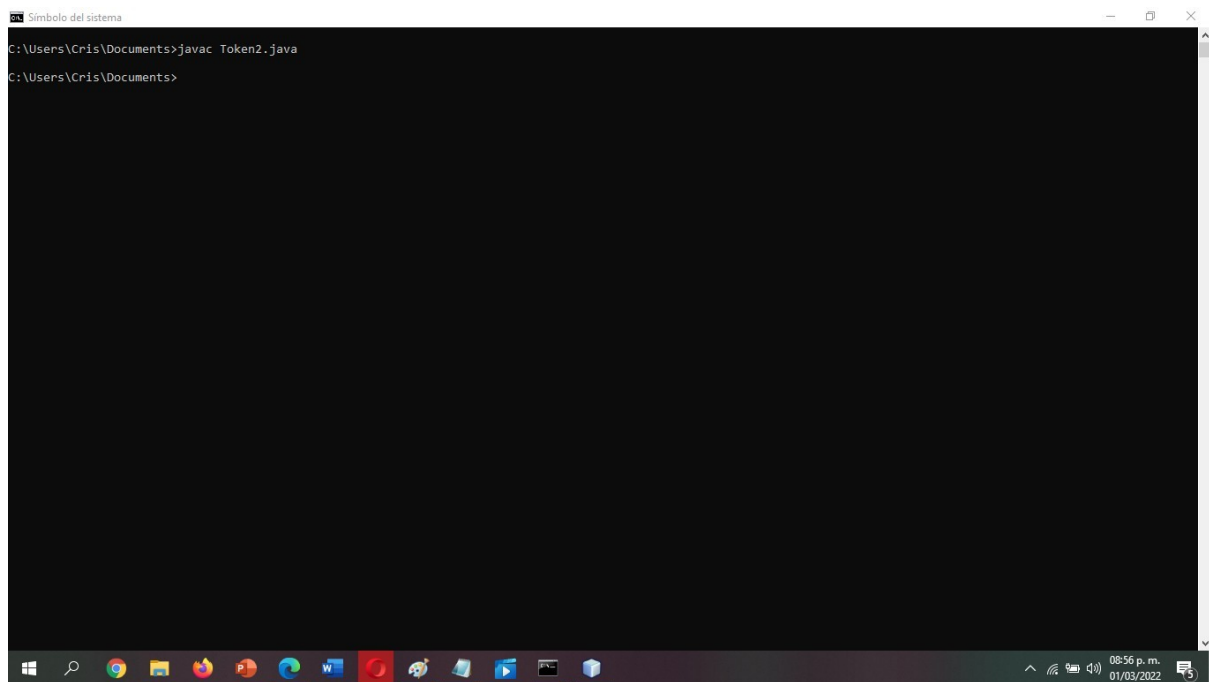


Imagen 8. Compilacion del programa

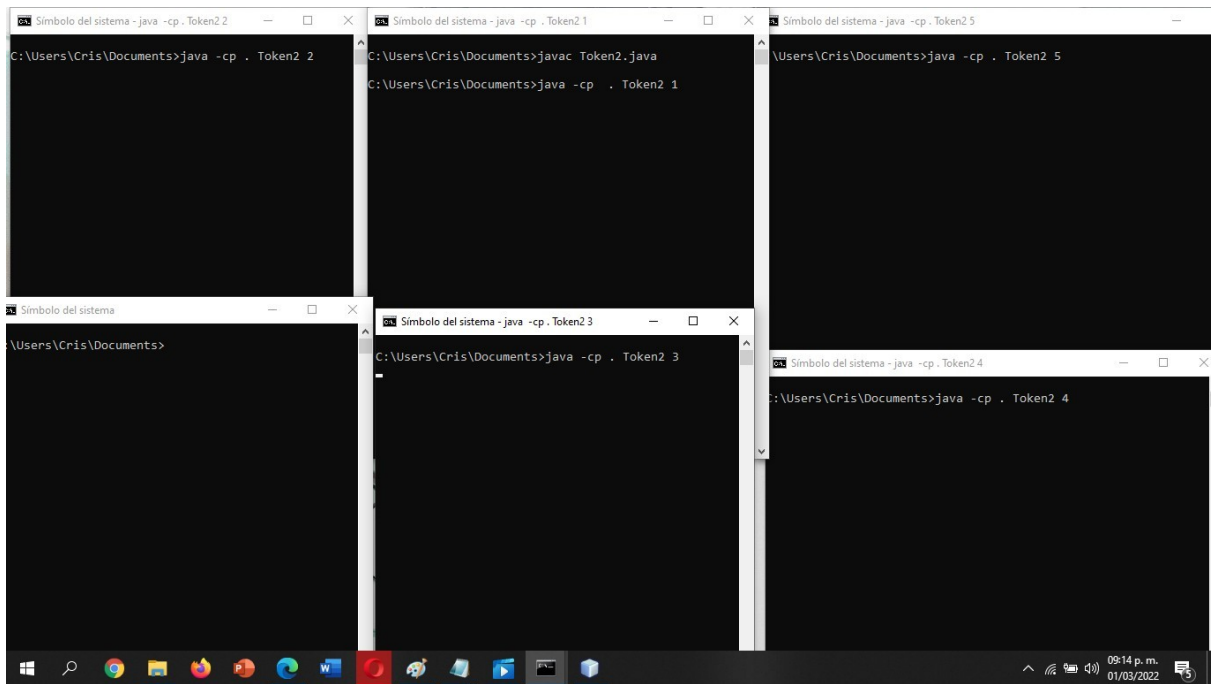


Imagen 9. 4 los 5 nodos (del 1 al 5) esperando que el nodo 0 inicie el conteo.

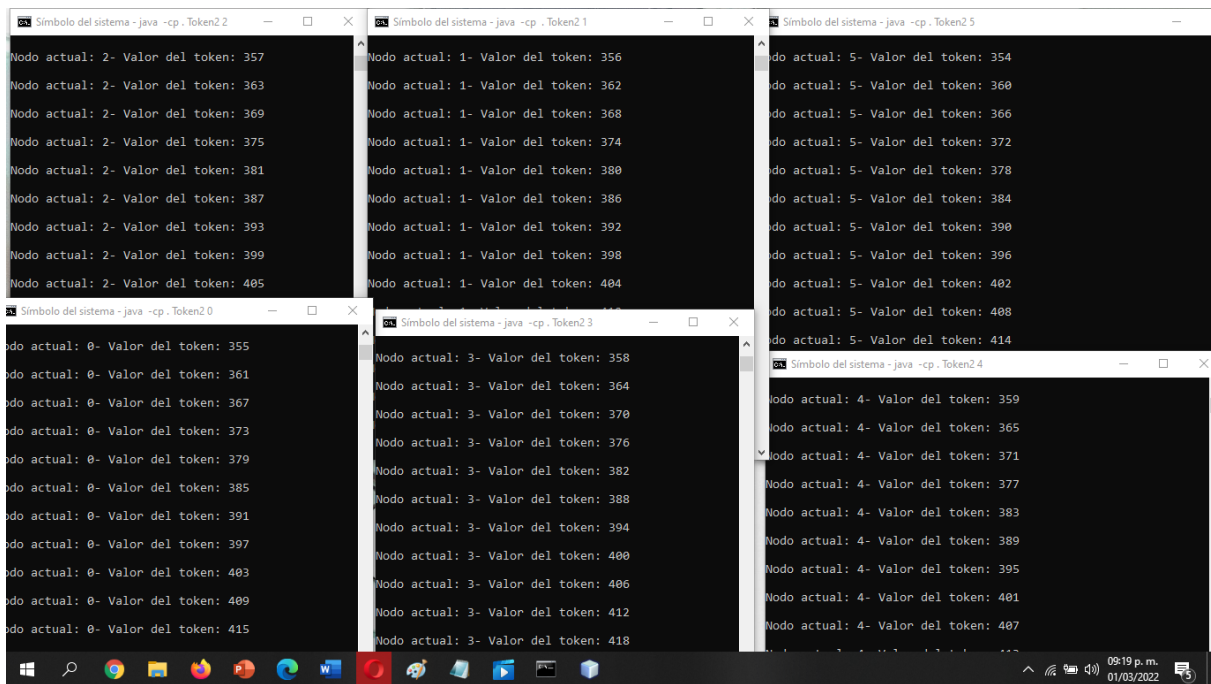


Imagen 10. los nodos comienzan el conteo y se envían mediante sockets seguros el token con una gran velocidad.

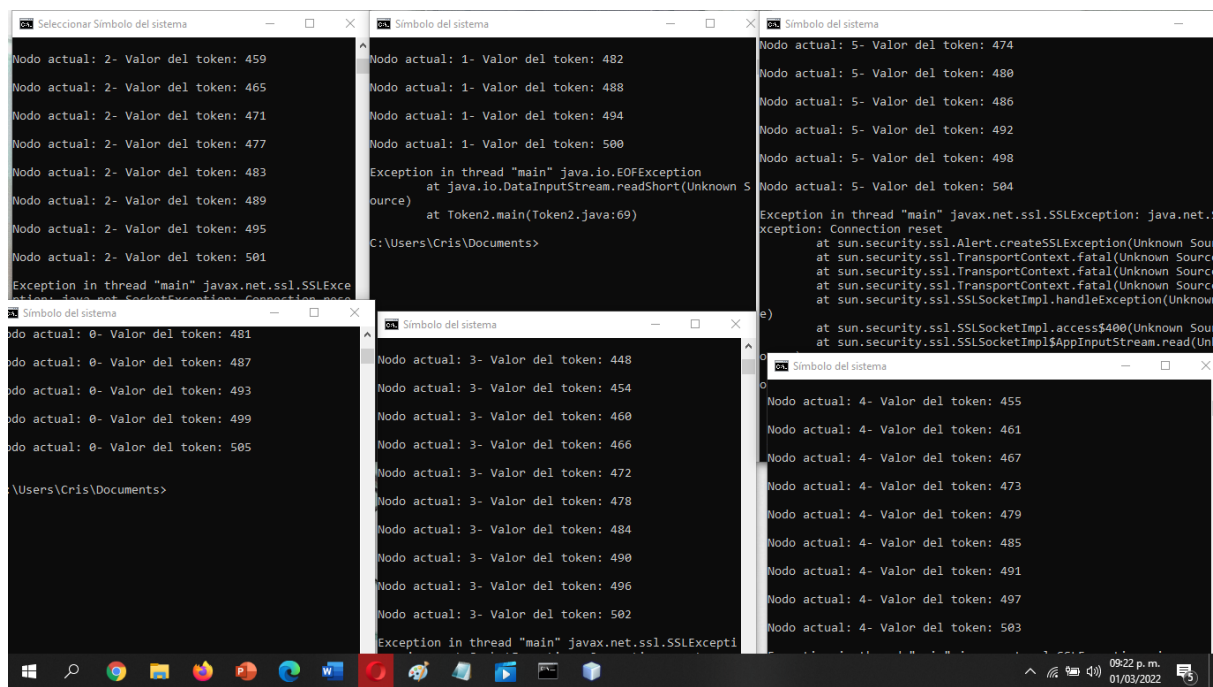


Imagen 11. El nodo 0 se desconecta en el momento que recibe el token con valor de 505 y termina con su ejecución.