Laboratorio #03

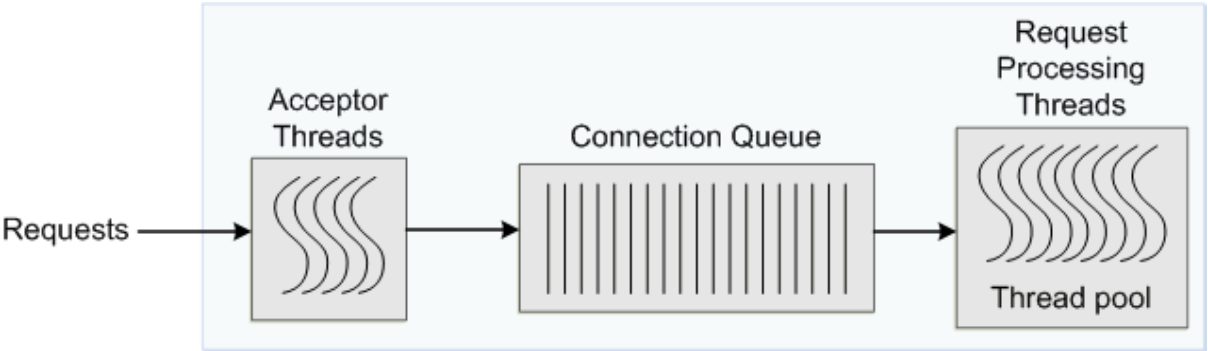
DDoS

Ciencias de la Computación VIII

Instrucciones:

Este laboratorio tiene como objetivo de conocer más sobre Request/Response Header de los métodos GET, POST y HEAD e implementar un thread pool previamente hecho en el Laboratorio #02. Lectura de **RFC's 791, 2460, 1180, 792.**

La implementación de un **threadPool** se utiliza para limitar la cantidad de conexiones a su servidor y evitar la saturación. Un diagrama de flujo de aceptación de Request:



Para indicar cual es el límite o definir cuantos Thread pueden ejecutarse vamos a pasarlo como primer parámetro en el MakeFile.

```
make threads=5
```

Se adjunta la estructura base del Laboratorio en el cual hay la ejecución del ThreadPool, la aceptación de Requests y el uso de Logger, recuerde que no debe utilizar ninguna librería que solucione el problema, con las librerías proporcionadas en los ejemplos debe ser suficiente. **EXTRAS 25pts** aparte del Thread Pool puede implementar otro método de seguridad limitando las conexiones por IP de cliente conectado, por ejemplo si vienen de la misma IP pública se repartirá la limitación de threads atendiendo, si es diferente la IP tiene disponibilidad de otro límite.

Actualmente su servidor debe de ser capaz de procesar y servir un objeto solicitado por medio de un HTTP **request**, debe de crear las implementaciones de los métodos GET, POST y HEAD.

Si el objeto existe, debe construir el HTTP **response** con **status-code 200 OK**, el **contenido del objeto solicitado** y enviarlo como respuesta al cliente.
Si el objeto no existe, debe construir el HTTP **response** correspondiente al **error**, enviar un archivo html con el status-page con la razón 404 Not Found, 500 Internal Server Error o 501 Not Implemented y enviarlo como respuesta al cliente.
Recuerde que si no se indica objeto por defecto es **index.html.**
Hasta el momento el método GET y POST su comportamiento es similar y solo soportamos **content-type: text/html.**

Para este laboratorio vamos a soportar varios tipo MIME para colocarlo en el **content-type** según el archivo solicitado (**TIP:** cargue **Cocoon** en el Browser para ver los content-type de los archivos, la misma que se envía en www) esto para el **método GET** el objetivo de esto es que tengan múltiples Http Request y que en el Browser tengan un Sitio Web que sea funcional y navegable, para el **método POST** vamos a crear un tipo de archivo con **extensión cc8** que al ser solicitado va a retornar como **content-type text/html** y para estos se va a procesar el **body Http Request del POST**, esto ser realizará en su servidor.

Procesamiento de Body del método POST

Cuando el sitio web de **Cocoon** que está dentro de la carpeta de ./www/ sea funcional desde su Web Server en la **sección de Contact** http://localhost:1850/contact.html va a encontrar dos formulario de dos tipos de **enctype** el primero es **application/x-www-form-urlencoded** que tiene el siguiente formato en el body en su HTTP Request

Ejemplo:

Request Header POST : application/x-www-form-urlencoded

```
POST /Test.cc8 HTTP/1.1
Host: www.galileo.edu
Connection: keep-alive
Content-Length: 68
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0

class=CC8&year=2020&email=yorch%40galileo.edu&text=Texto+Prueba+POST
```

Como se mencionaba anteriormente los archivos para el método POST tendrán la extensión **cc8** esto es solo por mera nomenclatura de nuestro servidor de CCVIII, así como lo es un servidor APACHE que trabaja con archivos PHP.

Cual es la idea detrás de estos archivos, en su interior van a tener una estructura HTML y su **Http Response** va a ser **content-type text/html** dentro existen **TAGs** de la forma **{keyValue}** donde estas hacen referencia a los **<input name="keyValue">** del formulario que se envían en el Body del Request POST.

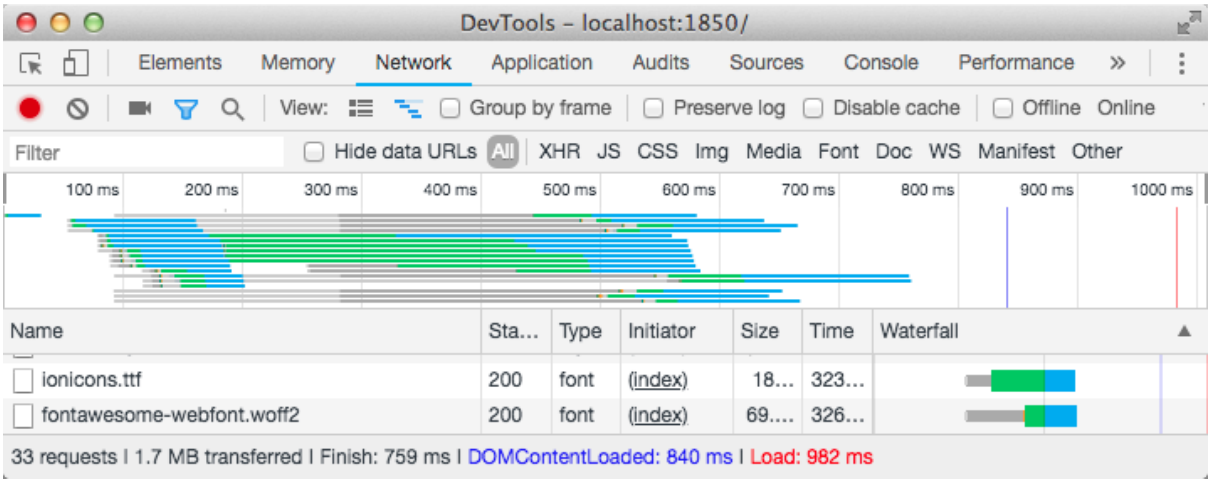
Ejemplo de un archivo cc8 (Test.cc8)

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Test POST</title>
</head>
<body>
<h1>La clase es {class}</h1>
<h4>El correo es {email}</h4>
<p>{text}</p>
</body>
</html>
```

EXTRAS 15pts El **enctype** multipart/form-data y desde PostMan con un POST con un body request de application/json

Prueba con Sitio Web:

Genere muchos requests con la plantilla **Cocoon** en ./www/



Nota: La idea del ThreadPool es mantener múltiples threads esperando hasta ser atendidos esto por la ejecución concurrente limitada por el programa de supervisión que creará.

Actividad en clase:

Ejecutar el Laboratorio validar el comportamiento de los tipos de instancias que existe en el ThreadPool ExecutorService:

- newCachedThreadPool
- newFixedThreadPool
- newScheduledThreadPool
- newSingleThreadExecutor
- newWorkStealingPool

Realice un documento contestando las siguientes preguntas:

1. ¿Es posible implementar todos?
2. ¿Cuáles son las diferencias?
3. ¿Se requiere alguna modificación en el código?
4. ¿Cuál es el comportamiento al ejecutar los Threads?
5. Comente por qué la mejor opción es utilizar newFixedThreadPool o Podría ser otro?

Referencia:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html

Entrega:

Que debe de llevar la entrega en el ZIP:

- Debe mantener la estructura dada de inicio de Laboratorio y de ser necesario agregar más clases
  - README.txt (Extras)
  - Makefile
  - DDoS.sh
  - killProces.sh
  - src
    - FileFormatter.java
    - ConsoleFormatter.java
    - Server.java
    - ClientHandler.java
    - www/ (página de Coccon)

- El laboratorio debe ser entregado por medio del GES, en un ZIP.
- El laboratorio debe estar hecho en Java (v.20)
- En el archivo **README.txt** se utilizará para los **EXTRAS** donde debe de describir que implementa y cómo resolvió el problema, **de lo contrario no se le tomará en cuenta.**
- El laboratorio debe ejecutarse

```
make
o
make threads=15
```

- De lo contrario puede tener una calificación de cero, si no sigue instrucciones o se detecta plagio.
- El sitio web al ser cargado no debe presentar ningún ERROR, los únicos permitidos son los siguientes: