Protocolos de Comunicación

Trabajo Especial

Revisión 2

Alumnos:
- Galindo, José Ignacio Santiago.
- Homovc, Federico.
- Pereyra, Cristian Adrián.

Índice

1. Protocolos desarrollados:

1.1 HTTP 1.1

Se implementó un proxy HTTP 1.1 con distintas funcionalidades y configuraciones extra pedidas por la cátedra y/o implementadas por nosotros mismos. [1]

En particular, se implementaron servicios de monitoreo y configuración accesibles directamente desde telnet o netcat mediante el uso del protocolo *HPCP* desarrollado por el equipo.

El servicio de monitoreo permite ver información sobre la transferencia total del proxy, número de bloqueos y transformaciones efectuadas, y cantidad de conexiones establecidas en el momento. Por otro lado el servicio de configuración permite habilitar o deshabilitar distintos filtros y bloqueos tanto globalmente como para un *User-Agent,* Sistema Operativo, IP, o Subred en particular. Ambos servicios tienen un acceso autenticado, cuyas credenciales se manejan en un archivo de texto almacenado dentro del ejecutable de java "jar".

1.2 HPCP (HTTP Proxy Configuration Protocol)

Se decidió diseñar un protocolo de tipo Request-Response para poder configurar el servidor y acceder a sus parámetros. Los usuarios deben acceder al servidor mediante un nombre de usuario y contraseña y a partir de entonces pueden acceder a la configuración actual y a determinados indicadores para monitoreo del proxy.

1.2.1 Descripción del Protocolo HPCP

El protocolo HPCP fue pensado para proveer una forma realmente simple de configurar los distintos parámetros del proxy, así como proveer una forma unívoca de acceso por usuario a los distintos servicios.

Todos los mensajes del protocolo se hacen en una sola línea, y el delimitador "\n" termina cualquier comando del cliente o mensaje del servidor.

1.2.1.1 Autenticación

Para poder acceder al servicio de configuración y datos el usuario debe primero autenticarse. El cliente debe conectarse al servidor mediante el puerto de configuración y enviar un mensaje “HELO\n” para iniciar la interacción con el servidor. El servidor DEBE enviar un mensaje de reconocimiento que empiece con el código 250.

Una vez hecho esto, el servidor le solicitará al usuario sus credenciales, en el caso de ser inválidas, el servidor DEBE contestar con un mensaje inicializado con el número 404, cuyo código indica un login inválido. En el caso de ser válidas, se devolverá el código 210.

Para terminar una sesión, el usuario debe insertar el comando "LOGOUT\n". Seguido de esto, el servidor responderá con código 201 para confirmar el logout.

El servidor PUEDE ofrecer mensajes extra luego de los códigos, de mensaje, para permitir a un usuario humano entrar mediante netcat o telnet.

Ejemplo: (Rojo: servidor, Azul: cliente)

HELO
250; Hello user, I am glad to meet you
Enter user name: cris
Enter password: cr15
210;Login OK
LOGOUT
201; Logout OK
Enter user name: juancho
404; Login Incorrect

1.2.1.3 Comandos de configuración

Cualquier usuario una vez que es autorizado por el servidor puede ingresar los siguientes comandos: “GET”, “SET” y “GETALL” terminando la sentencia al presionar la tecla ‘enter’. Todos los comandos son case-sensitive.

“GET” recibe como parámetro una propiedad y debe devolver su valor actual en la forma “valor_actual\n”. En caso de ser una lista de valores los mismos se separan con espacios. En caso de no tener parámetros o que los mismos no se correspondan con las propiedades se debe indicar con codigo de error 402.

“SET” recibe como parámetros una propiedad y un valor o lista de valores separados por una coma (“,”) y sin espacios y cambia el valor de dicha propiedad al valor o valores dado como segundo parametro. En caso de ser una lista de valores, los mismos se agregan al final de esta, y si no hay ningún valor la lista queda vacía. Una vez finalizada la acción se debe mostrar el mensaje “nombre_propiedad set to valor_actual\n”; en caso de que sea una lista el valor se muestra entre corchetes y con los items separados por comas (“,”) y un espacio. En caso de que no haya parámetros o propiedades o los mismos sean inválidos se debe indicar con un mensaje de error “Invalid Parameter\n”

“GETALL” no recibe ningún parámetro y muestra una lista de todas las propiedades de la configuración de proxy en la forma “nombre_parametro=valor_actual” con las propiedades separadas por comas (“,”) y terminando en “\n”. En caso de que el valor actual sea una lista se muestra la misma entre corchetes y con los valores separados por comas (“,”).

1.2.1.3 Comandos de configuración por criterio

Los comandos de configuración anteriores se pueden combinar con distintos criterios. Utilizando la forma {TOKEN} {VALUE} {COMANDO} <parametros>. Se puede filtrar a determinado cliente o grupo de clientes.

TOKEN puede tener como valor los distintos filtros posibles:

- **OS:** Para restringir un sistema operativo en particular, VALUEs posibles: **ANDROID, MAC_OS_X, WINDOWS, IOS, LINUX.**
- **BROWSER:** Para restringir un navegador en particular, VALUEs posibles: **FIREFOX, OPERA, CHROME, ANDROID, SAFARI.**
- **IP:** Para restringir un cliente IP en particular, toma como VALUE cualquier IP.
- **SUBNET:** Para restringir un cliente IP en particular, toma como VALUE cualquier Subnet con formato cidr.

Los comandos y valores son los mismos definidos en 1.2.1.2.

Ejemplo: (Rojo: servidor, Azul: cliente)

HELO
250; Hello user, I am glad to meet you
Enter user name: cris
Enter password: cr15
210;Login OK
GETALL
{ LISTADO DE PARAMETROS }
BROWSER CHROME GETALL
{ LISTADO DE PARAMETROS PARA CHROME }
BROWSER CHROME SET blockedURLs
200; blockedURLS set to []
BROWSER CHROME GET blockedURLs

GET blockedURLs
[www.thepiratebay.se]
LOGOUT
201; Logout OK

1.2.1.4 Propiedades

Las propiedades de configuración son esencialmente 11: blockAll, l33t, rotateImages, maxResSize, blockedIPs, blockedURLs, blockedMediaTypes, chainProxy, chainProxyPort, chainProxyHost y maxResEnabled. Las mismas son case-insensitive al momento de ingresarlas en cualquier comando de la configuración. Estas propiedades determinan:

- blockAll (booleano): bloquea todo acceso a través del proxy.
- l33t (booleano): en caso de que el formato del contenido sea text/plain, se pasa el mismo a formato l33t.
- rotateImages (booleano): rota la imagenes 180 grados.
- maxResSize (entero): limita el tamaño máximo de un recurso al cual se puede acceder.
- blockedIPs (lista): una lista con las IPs que se encuentran bloqueadas y no pueden acceder a ningún recurso a través de este proxy. Cada IP se escribe de la forma “ip/mascara”. Si algun valor no es valido, no se agrega ese ni ninguno de los siguientes.
- blockedURLs (lista): una lista con las URLs que se encuentran bloqueadas y no pueden ser accedidas por ningún usuario a través de este proxy.También acepta expresiones regulares.
- blockedMediaTypes (lista): una lista con todos los tipos de datos que se encuentran bloqueados y no pueden ser descargados a través de este proxy.
- chainProxy (booleano): indica si este proxy se debe conectar a través de otro proxy o no.
- chainProxyPort (entero): indica el puerto del proxy remoto por el cual se encadenarán las conexiones.
- chainProxyHost (cadena): indica el host del proxy remoto por el cual se encadenarán las conexiones.
- maxResEnabled (booleano): indica si se encuentra activado el límite al tamaño de algún recurso.

Para los valores booleanos el comando para setearlos es “SET nombre_parametro true|false”; cualquier otro valor no nulo setea la propiedad como “false”. En el caso de los

valores de listas es “SET nombre_parametro valor1,valor2,valor3,...,valor_n”; en caso de no poner ningún valor en la lista esta no se modifica. La propiedad maxResSize se setea mediante el comando “SET maxResSize valor” donde valor debe ser un número entero de bytes.

2. Problemas encontrados durante el diseño y la implementación

2.1 Content Coding:

Tras empezar a crear nuestro proxy, encontramos distintos problemas con los encodings a la hora de manipular las cadenas o imágenes. Por ejemplo, si la información se envía en gzip o deflate, hay que poder interpretar ese contenido para poder modificarlo posteriormente. Decidimos cambiar el accept-encoding a identity para evitar estos problemas.

2.2 Chunked Transfer Coding:

Inicialmente tuvimos distintos problemas con el transfer coding chunked, ya que como manipulabamos mal los encodings, no siempre funcionaba bien. Una vez solucionado el problema de los charsets, se reescribió la clase encargada de interpretar el chunked transfer coding y se solucionó el problema.

2.3 Proxy ITBA - Content-Length:

En el proxy del bar del ITBA, encontramos un problema inesperado: Páginas que envían transfer-coding chunked, son juntadas y enviadas sin content length, utilizando HTTP 1.0, donde se cierra la conexión directamente si no se utiliza el header "Connection: keep-alive". Tuvimos que cambiar la implementación para soportar esto y poder cerrar la conexión.

2.4 Manejo de conexiones persistentes del lado del cliente remoto:

Inicialmente esta fue una implementación *naive*, por cada request se establecía un socket con el cual se hacía un request y se devolvía la respuesta. Pero esto trajo a la larga muchos problemas, no se encontraba sincronizado, y había que ordenar y bufferear las respuestas para tener muy poca ganancia. Además, el rfc 2616 prohíbe el uso de más de dos sockets por conexión. Por lo que pasamos a utilizar un sólo socket por conexión, y mantuvimos un pool con las conexiones que quedaban abiertas para aprovechar la performance extra que nos dejaban.

2.5 Comunicación entre threads:

Si bien utilizamos NIO, decidimos delegar a distintos threads distintas tareas, por lo cual planteamos un modelo de eventos los cuales contenían metadata de la información, con la cual armaban toda la estructura del request o del response. A la hora de hacer tests de performance, encontramos distintas problemáticas relacionadas con el sincronismo en determinados lugares tanto del cliente como del servidor. Una vez solucionados estos problemas (deadlocks, puntualmente) mediante el uso de la herramienta *JVMMonitor* la performance aumentó considerablemente.

2.6 Manejo de errores:

El manejo de errores se tornó complicado en determinadas situaciones, la desconexión de un cliente en un thread afectaba al servidor en otro, y hubo que crear un evento del tipo error para pasar esos mensajes y coordinar el cierre de los recursos asignados a cada uno de los lados del proxy, el cliente y el servidor.

2.7 Diseño del monitoreo:

Inicialmente contábamos la cantidad de sockets abiertos en el cliente y el servidor a medida que llegaban los requests, pero eso no nos permitía tener una noción de cuando el proxy no estaba funcionando, por lo que decidimos cambiar eso utilizando un timer que le pregunta a la clase encargada de manejar las conexiones de más alto nivel, cuantas conexiones tiene, y como el servidor y cliente siempre tienen la misma cantidad de conexiones, la cantidad de conexiones total es el doble a esa cantidad más los sockets abiertos en espera.

2.8 Manejo de multiples workers:

El diseño inicial planteaba que sólo utilizaríamos tres threads, uno para el servidor, otro para el cliente y un tercero para el manejo lógico del proxy, pero ese tercer thread se llevaba mucha carga, y en un sistema de multiproceso no se aprovechaban los recursos. Por lo tanto cambiamos la implementación utilizando un *composite pattern*, este pattern se utilizó también para los filtros y los readers.

2.9 Manejo de charsets:

Inicialmente desconocíamos en detalle cómo se manipulan los charsets en Java, y tuvimos problemas al trabajar con los byte arrays que nos llegaban desde el servidor, quisimos parsearlos directamente como string, y terminamos modificando el contenido del arreglo de bytes. Lo solucionamos encodeando y desencodeando correctamente donde necesitábamos los requests, y evitando tocar la información propiamente dicha.

3. Limitaciones de la aplicación

3.1 Content coding:

Al evitar utilizar gzip, deflate y los otros distintos tipos de content-codings, limitamos mucho al proxy en cuanto al tamaño de responses que va a tener, con lo cual se afecta directamente a la performance. Decidimos dejarlo en este estado ya que no es un *MUST* para la implementación requerida y no tuvimos tiempo para aplicarlo.

3.2 Transfer chunked:

Si bien el proxy decodea transfer chunked, no vuelve a encodear, con lo cual todos los responses utilizan content-length, esto afecta notoriamente el renderizado de las páginas.

3.3 Configuración embebida al jar:

Si bien se puede acceder al jar dezippeandolo, es incómodo, y es una limitación actual de la aplicación, al compilarlo y embeber todo en el .jar, las configuraciones quedan embebidas dentro y sólo se pueden editar directamente desde telnet. Una solución a futuro es la de agregar un path para los archivos de configuración mediante parámetros extra. Lo mismo se aplica al nivel de verbosity de los logs.

3.4 Assets a modificar con tamaño excesivo:

Si bien se pide la modificación de assets del tipo image/* y text/plain, ya que llevamos siempre a memoria toda la información antes de modificarla, es posible que haya algún out of memory error.

3.5 POST con Multipart Data y Transfer Encoding Chunked:

No conseguimos llegar a dar soporte a este tipo de requests, por lo que sumado al punto anterior, puede que hayan problemas a la hora de subir assets demasiado grandes.

4. Posibles extensiones

Creemos que las tres limitaciones anteriormente mencionadas pueden ser claramente implementadas y contarían como extensión.

También consideramos otras extensiones posibles a las siguientes:

- Soporte para pipelining.
- Más transformaciones a las imágenes.
- Transformaciones a los html.
- Mejores filtros por user agent.

5. Conclusiones

El desarrollo de un proxy http, al igual que un cliente o un servidor, no es una tarea sencilla, este tipo de software es uno que normalmente pasa desapercibido para los usuarios finales pero es el tipo de software que más *pulido* necesita estar, y requiere ser altamente tolerante a fallas.

Creemos que en términos generales nuestro diseño se adaptó a lo que inicialmente planteamos en la preentrega, lo cual es algo muy positivo. Sin embargo tuvimos que acoplar mucho algunas clases para poder adaptarlas a los requerimientos sobre el final. Aun así, algunas funcionalidades no pudieron ser implementadas por diversos motivos y quedaron pendientes para futuras versiones.

Consideramos que es muy interesante el RFC 2616 y el protocolo en HTTP en general, es un protocolo que considera muchas de las problemáticas posibles a la hora de transmitir hipertexto por la web. Sin embargo creemos que es necesario resaltar que en el mundo real, el RFC 2616 no es el único que se aplica, algunos headers como Proxy-Connect son ampliamente utilizados en la web hoy en día y no son parte del estándar, por eso decidimos ignorarlos y centrarnos en los requerimientos del RFC 2616.

6. Ejemplos de testeo

6.1 Tests Unitarios:

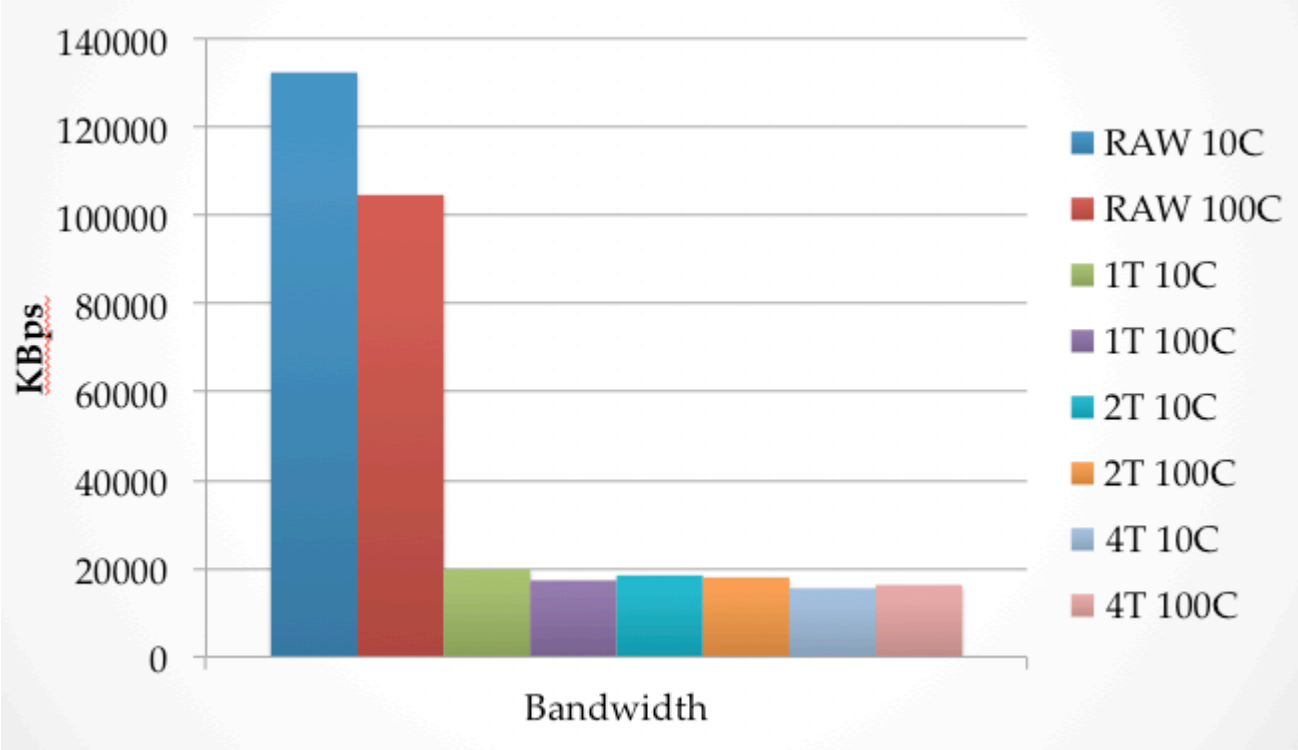
Utilizamos distintos tests unitarios para garantizar el funcionamiento correcto de las clases básicas. Para testear utilizamos JUnit. Los mismos se encuentran en la carpeta src/test y se pueden ejecutar corriendo el comando *mvn test*

6.2 Tests de Carga:

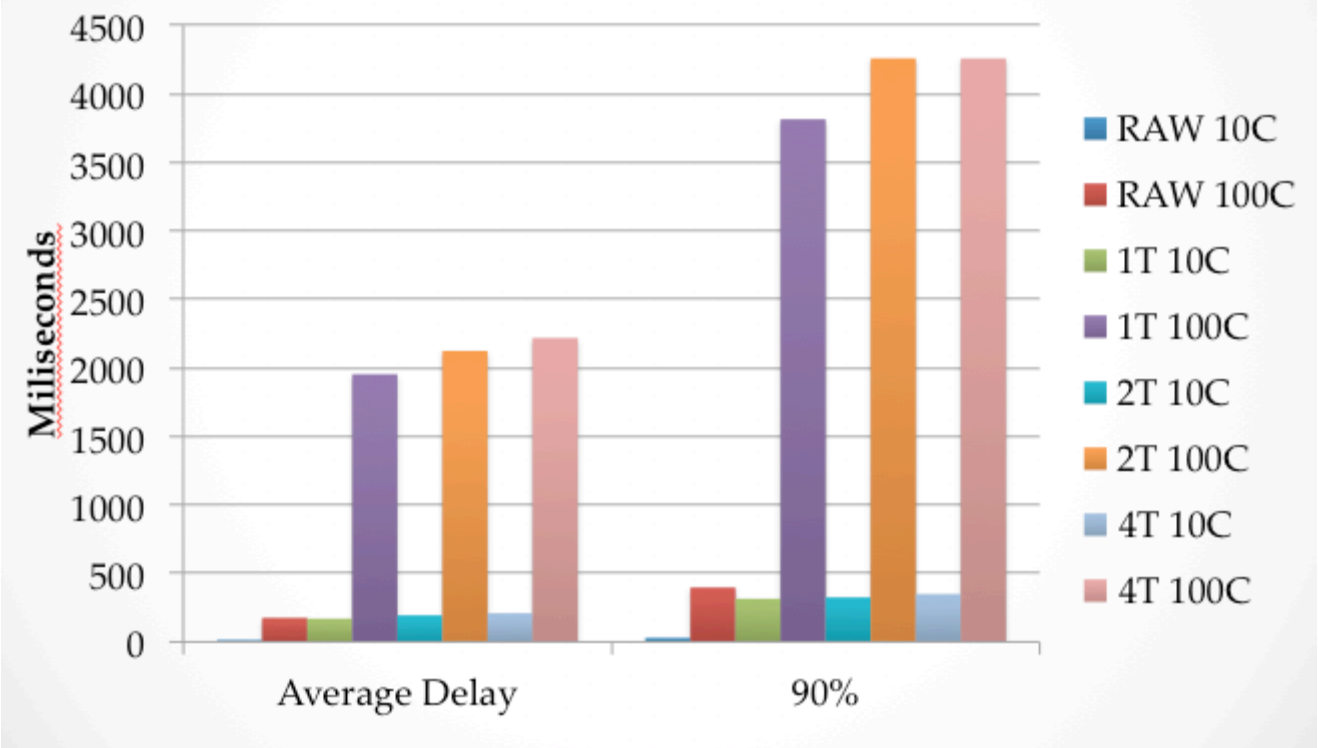
Para realizar tests de carga y ver cómo varía la performance en la aplicación con JMeter hicimos unos tests de carga, y vimos cómo varían el throughput, los requests por segundo, y la latencia con la cantidad de worker threads, y con la cantidad de conexiones concurrentes.

A continuación se adjuntan las diversas gráficas que muestran los resultados de estos tests. En los mismos se conectó contra un servidor local nginx.

La siguiente gráfica muestra cómo decae varía con distinta cantidad de threads la performance del proxy. (T = Threads, C = Conexiones concurrentes, RAW = Sin proxy)



En esta gráfica a su vez se pueden ver los delays que surgen a la hora de utilizar el proxy con múltiples conexiones concurrentes. El crecimiento de los delays es entendible, si bien se hubiese esperado que fuese solo multiplicado por 10, se ve que es por un factor un poco más grande, por lo que se puede esperar que haya algún cuello de botella extra.


4500
4000
3500
3000
2500
2000
1500
1000
500
0
Miliseconds
Average Delay
90%
RAW 10C
RAW 100C
1T 10C
1T 100C
2T 10C
2T 100C
4T 10C
4T 100C

7. Guía de instalación

Se adjunta en el README.md del repositorio git.

8. Instrucciones para la configuración

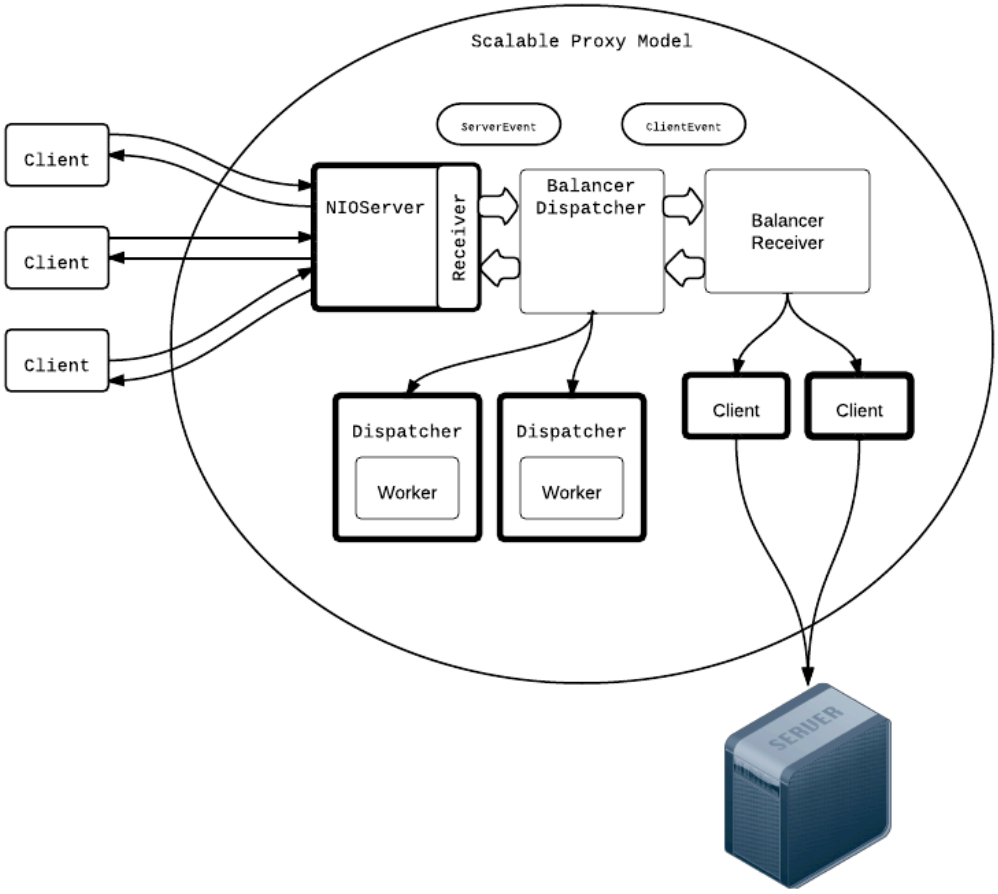
Fueron detalladas en la descripción de HPCP (Punto 1.2.1.3). Las configuraciones propias de cada instancia del proxy se pueden ver con el comando --help a la hora de ejecutar el .jar

9. Ejemplos de Configuración

Se encuentran en la carpeta src/main/resources/configuration.properties

10. Diseño del Proxy

10.1. Diagrama de Componentes



El proxy está conformado principalmente por tres componentes autónomos que funcionan en conjunto. Por un lado un NIOServer, que se ocupa de recibir la petición del cliente que está accediendo al proxy; por otro lado, un NIOClient que se ocupa de enviar la petición al server externo, y un Dispatcher, que posee un Worker dentro de él, que se ocupa de conectar al NIOServer con del NIOClient, procesando la petición.

Tanto el servidor (NIOServer) como el cliente (NIOClient), están conectados con el Dispatcher por medio de EventReceivers que, como su nombre lo indica, reciben eventos que están conformados por un arreglo de bytes, con el texto de la petición, y por el socket que se usó para atender a la misma. Para diferenciar desde donde viene cada evento y para que el worker la pueda procesar de forma acorde, existen dos tipos de eventos, uno que viene desde el servidor interno (ServerEvent), y otro que viene desde el cliente interno (ClienteEvent).

Cada uno de estos tres componentes corre en un thread separado, y tanto para la parte del Dispatcher como del NIOClient, se puede escalar y usar varios Dispatchers (con su correspondientes workers) y/o varios NIOClients de manera de poder distribuir la carga entre ellos.

También, gracias a esta separación que existe entre los diferentes componentes, es posible utilizar un servidor especial como mecanismo de configuración y monitoreo.

10.2. Diagramas de clases y explicación del código;

10.2.1. Sequencia optima de manejo de un request/response

Normalmente, para todos los requests válidos, el proxy se comportaría de la siguiente manera:

1. Llegaría información al NIOServer, y el Handler del mismo leería a información, enviandola a un eventdispatcher, quien se encargaría de enviarla a un worker determinado.
    a. La información se envía en forma de ServerDataEvent, el cual es un evento conteniendo toda la metadata relacionada a la conexión con el cliente del proxy (socketchannel e información enviada).
2. Dentro de ese worker, se intenta parsear el header dentro de una clase encargada de eso (HTTPRequestEventHandler).
    a. Si hay algún error, se tira una excepción y se devuelve el código de error adecuado (No se puede resolver el dns, hay algún filtro que decide bloquear el request, etc).
        i. Para los filtros se utiliza la clase HTTPBaseFilter la cual resuelve según el request los criterios y decide si se bloquea o no.
        ii. HTTPBaseFilter es un composite de HTTPFilters que se construye estáticamente.
    b. Si el request es del tipo POST se lee el cuerpo del mensaje.
3. Se envia el header al cliente NIO mediante el event dispatcher con un evento especifico llamado ClientDatataEvent, el mismo contiene el puerto y host de conexión, más un "attachment" que es normalmente un objeto que representa el evento de conexión al proxy, su hashcode y equals se basan en el socketchannel del cliente (usuario final).
4. De esta forma, se abre un socket a los servidores remotos por cliente que haga un request, en el caso de que el host abierto cambie, ese socket también cambia.
5. Una vez enviado el request, comienza a llegar el response, y se envía mediante el dispatcher al mismo worker donde se recibió la información. Y este worker (HTTPProxyWorker) se encarga de manejar el evento.
6. Dentro de este worker se parsean los headers y luego se lee la información.
    a. Para detectar el fin de un request, aplicar modificaciones y decodear el transfer encoding chunked, se utiliza la clase HTTPBaseReader, que es un composite de HTTPReaders, los cuales son capaces de transformar, modificar los headers y bufferear el request sólo en caso de que sea necesario.
    b. Es necesario aclarar que todas las operaciones mencionadas buscan reducir el buffering al extremo, si es posible leer la información y enviarla directamente, así se hace, tanto en el request como en el response.
7. Es posible que el punto 1 a 6 se repitan varias veces hasta que haya una orden de cerrar la conexión con Connection:close, donde se le adjunta al evento la propiedad CanClose en true, y se le dice al NIOServer que cierre la conexión.

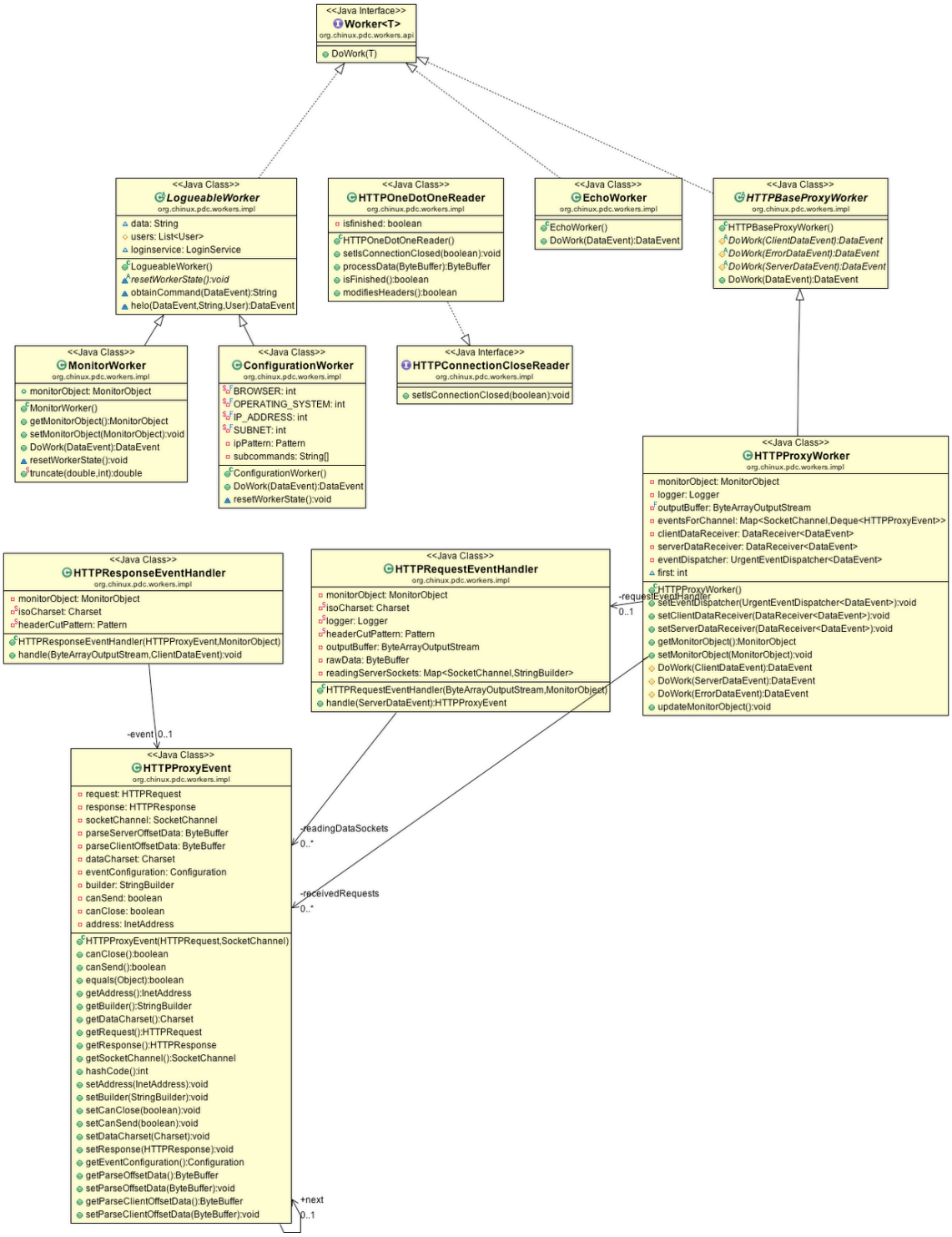
10.2.2. Manejo de errores

Si bien hay muchos tipos de errores para ser handleados que talvez no se enumeren en este listado, enumeramos los más relevantes.

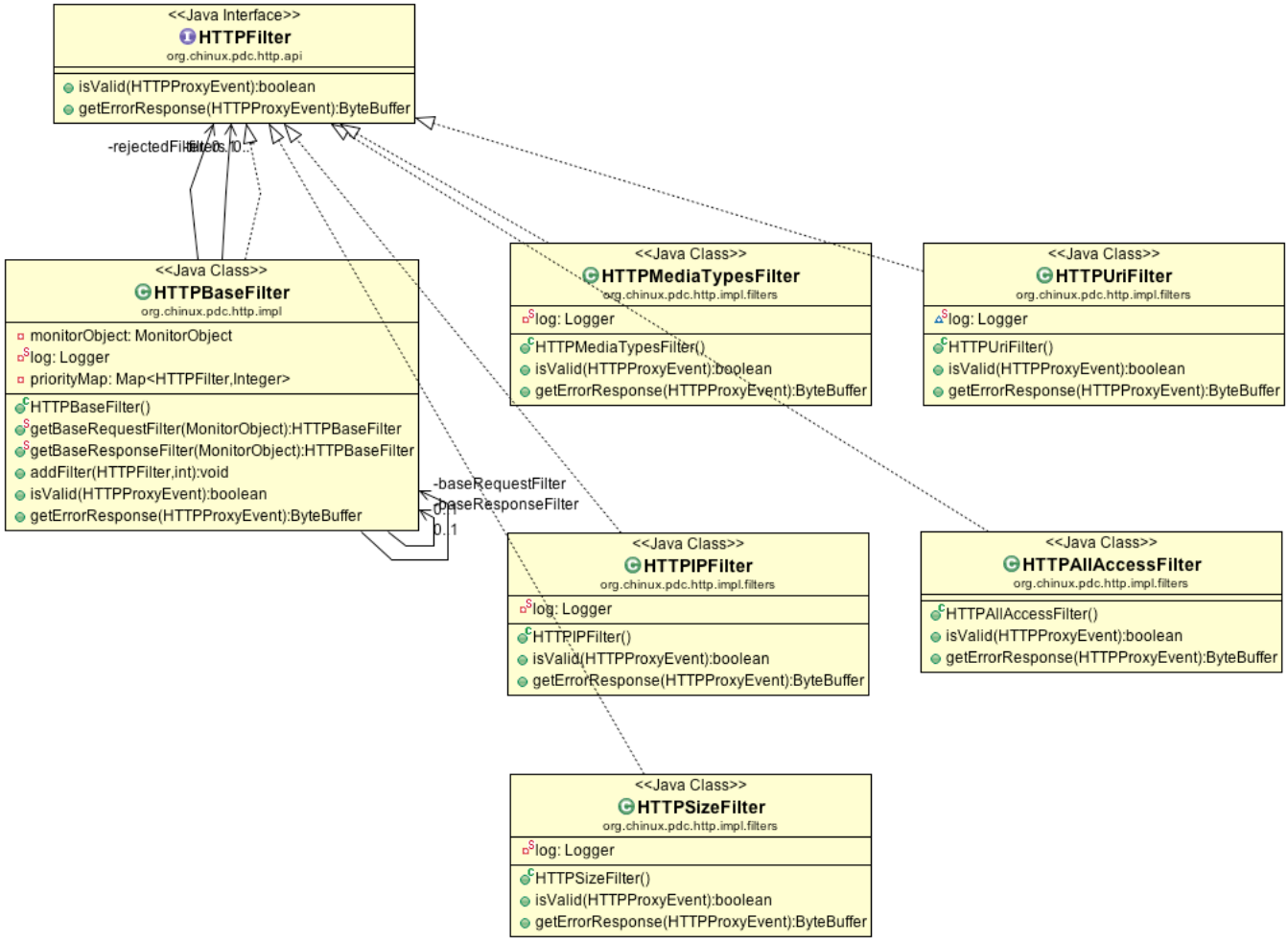
1. En el caso de que el cliente del proxy se desconecte inesperadamente (por ejemplo, se cancele una descarga). No se envía un evento, sino que al llegar el evento de lectura del http request, se pregunta si el cliente remoto está activo o no, si no está activo entonces se descarta todo el mensaje y sus consecutivos, y se le pide al cliente NIO que guarde esas conexiones abiertas en el pool de conexiones.
2. En el caso de que alguna conexión haya expirado a la hora de intentar reutilizarla en el pool de conexiones, se crea un nuevo socketchannel y se asigna la información pendiente del socket channel anterior a ese socket channel.
3. En caso de no tener un *route to host* al intentar conectarse a un servidor. Se tira ErrorDataEvent acorde a la cola del eventdispatcher, y se devuelve un error 502 Bad Gateway al cliente.
4. Se efectura lo mismo (Error 502), pero con una excepción a la hora de parsear los requests, si no se puede resolver el DNS del header Host.
5. También se tira un ErrorDataEvent en el caso de que el cliente remoto se haya desconectado de forma abrupta, haciendo lo mismo con el cliente.

10.2.3 Diagramas de clases

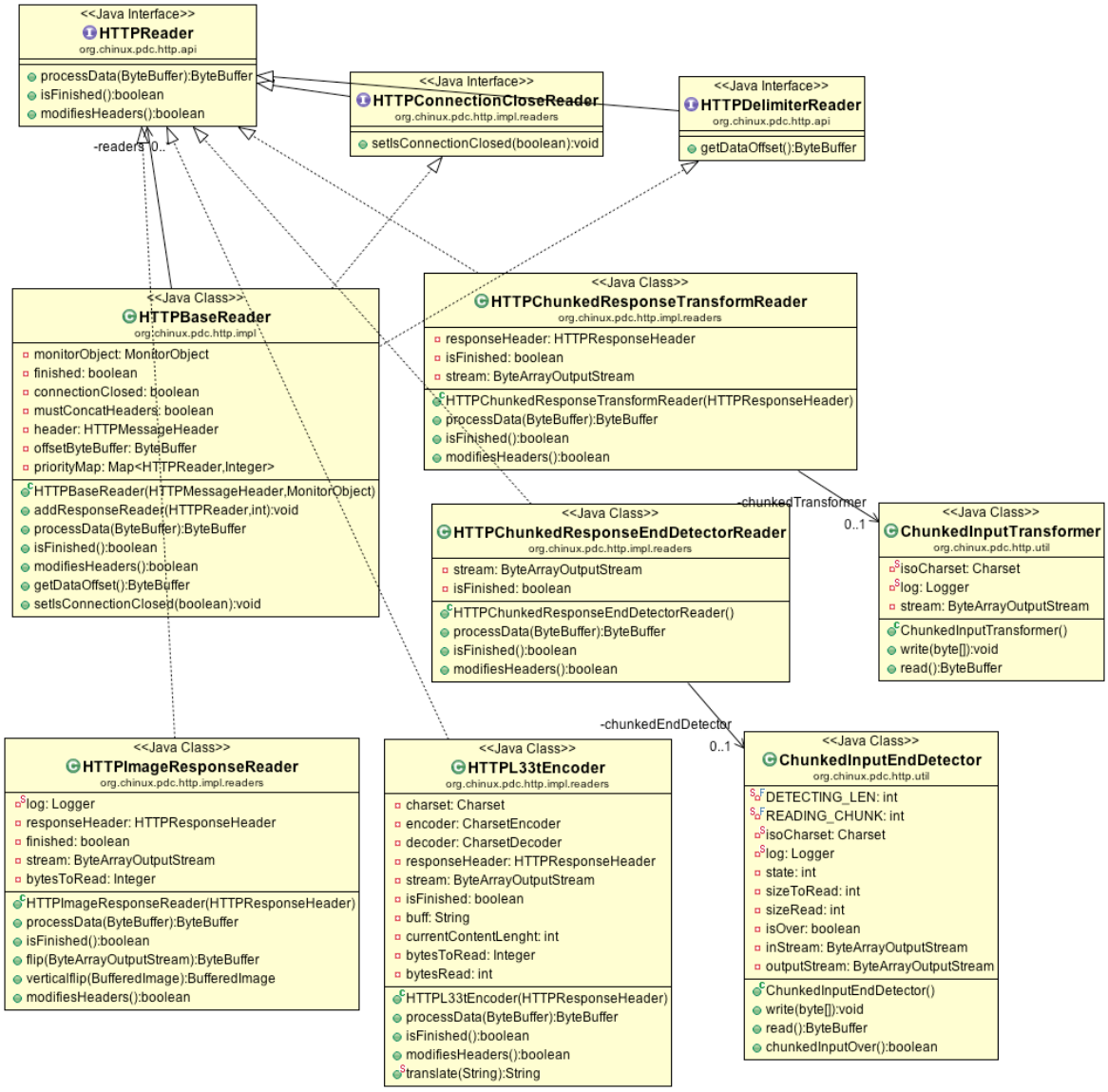
Workers



Filters


<<Java Interface>>
HTTPFilter
org.chinux.pdc.http.api
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
-rejectedFilters
0..1
<<Java Class>>
HTTPBaseFilter
org.chinux.pdc.http.impl
monitorObject: MonitorObject
log: Logger
priorityMap: Map<HTTPFilter,Integer>
HTTPBaseFilter()
getBaseRequestFilter(MonitorObject):HTTPBaseFilter
getBaseResponseFilter(MonitorObject):HTTPBaseFilter
addFilter(HTTPFilter,int):void
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
-baseRequestFilter
-baseResponseFilter
0..1
0..1
<<Java Class>>
HTTPMediaTypesFilter
org.chinux.pdc.http.impl.filters
log: Logger
HTTPMediaTypesFilter()
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
<<Java Class>>
HTTPUriFilter
org.chinux.pdc.http.impl.filters
log: Logger
HTTPUriFilter()
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
<<Java Class>>
HTTPIPFilter
org.chinux.pdc.http.impl.filters
log: Logger
HTTPIPFilter()
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
<<Java Class>>
HTTPAllAccessFilter
org.chinux.pdc.http.impl.filters
HTTPAllAccessFilter()
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer
<<Java Class>>
HTTPSizeFilter
org.chinux.pdc.http.impl.filters
log: Logger
HTTPSizeFilter()
isValid(HTTPProxyEvent):boolean
getErrorResponse(HTTPProxyEvent):ByteBuffer

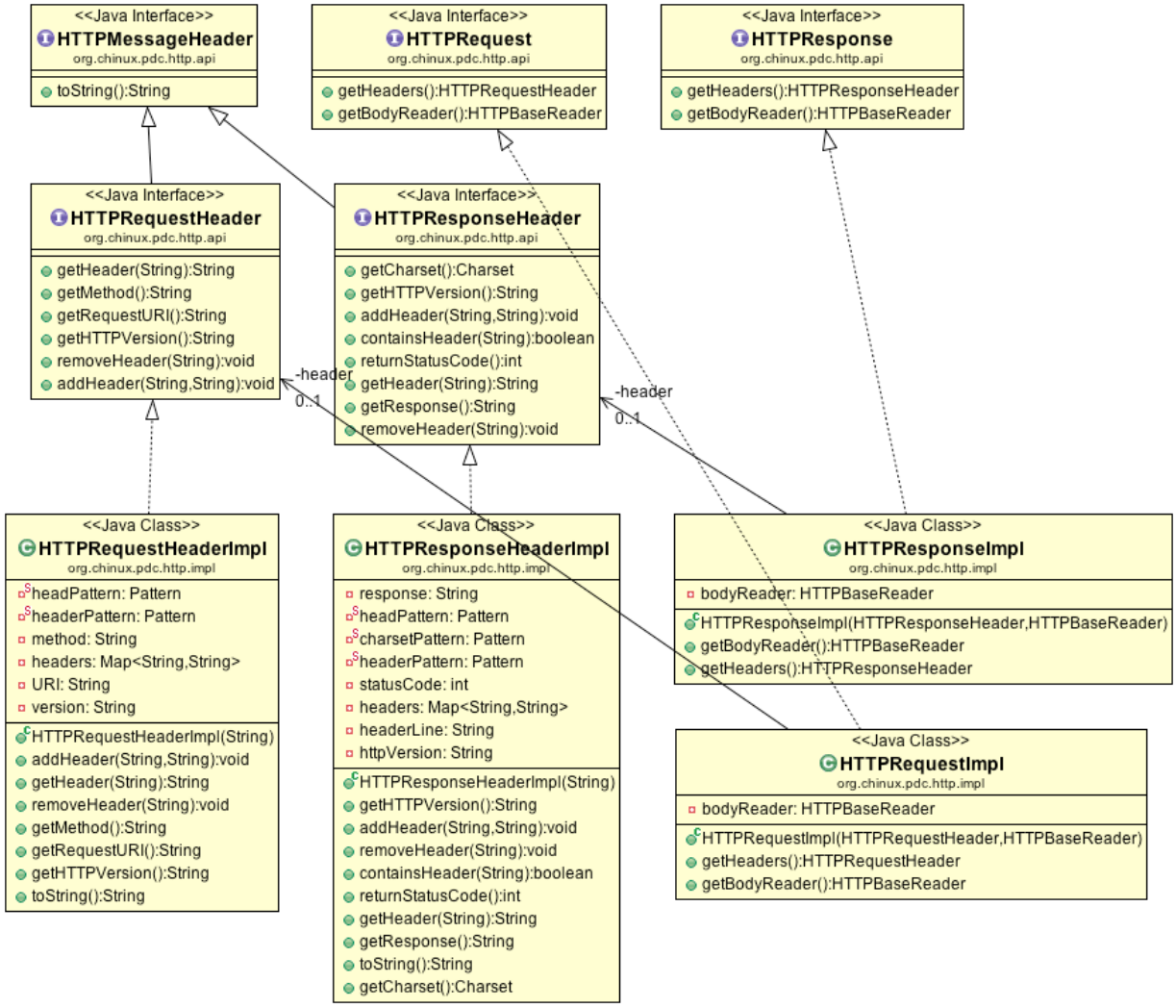
Readers

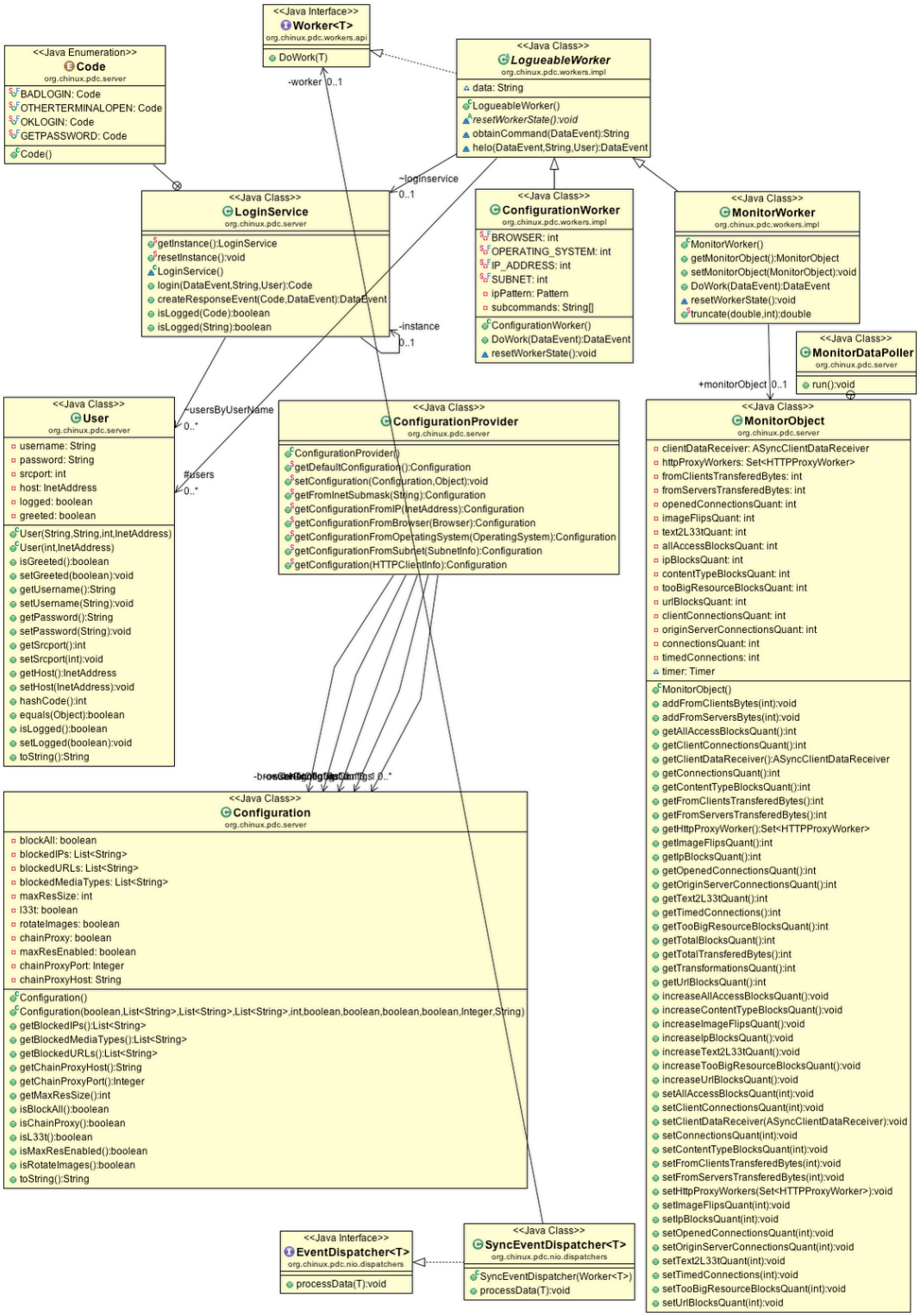

<<Java Interface>>
HTTPReader
org.chinux.pdc.http.api
processData(ByteBuffer):ByteBuffer
isFinished():boolean
modifiesHeaders():boolean
-readers 0..
<<Java Interface>>
HTTPConnectionCloseReader
org.chinux.pdc.http.impl.readers
setIsConnectionClosed(boolean):void
<<Java Interface>>
HTTPDelimiterReader
org.chinux.pdc.http.api
getDataOffset():ByteBuffer
<<Java Class>>
HTTPBaseReader
org.chinux.pdc.http.impl
monitorObject: MonitorObject
finished: boolean
connectionClosed: boolean
mustConcatHeaders: boolean
header: HTTPMessageHeader
offsetByteBuffer: ByteBuffer
priorityMap: Map<HTTPReader,Integer>
HTTPBaseReader(HTTPMessageHeader,MonitorObject)
addResponseReader(HTTPReader,int):void
processData(ByteBuffer):ByteBuffer
isFinished():boolean
modifiesHeaders():boolean
getDataOffset():ByteBuffer
setIsConnectionClosed(boolean):void
<<Java Class>>
HTTPChunkedResponseTransformReader
org.chinux.pdc.http.impl.readers
responseHeader: HTTPResponseHeader
isFinished: boolean
stream: ByteArrayOutputStream
HTTPChunkedResponseTransformReader(HTTPResponseHeader)
processData(ByteBuffer):ByteBuffer
isFinished():boolean
modifiesHeaders():boolean
-chunkedTransformer
0..1
<<Java Class>>
HTTPChunkedResponseEndDetectorReader
org.chinux.pdc.http.impl.readers
stream: ByteArrayOutputStream
isFinished: boolean
HTTPChunkedResponseEndDetectorReader()
processData(ByteBuffer):ByteBuffer
isFinished():boolean
modifiesHeaders():boolean
<<Java Class>>
ChunkedInputTransformer
org.chinux.pdc.http.util
isoCharset: Charset
log: Logger
stream: ByteArrayOutputStream
ChunkedInputTransformer()
write(byte[]):void
read():ByteBuffer
-chunkedEndDetector
0..1
<<Java Class>>
HTTPImageResponseReader
org.chinux.pdc.http.impl.readers
log: Logger
responseHeader: HTTPResponseHeader
finished: boolean
stream: ByteArrayOutputStream
bytesToRead: Integer
HTTPImageResponseReader(HTTPResponseHeader)
processData(ByteBuffer):ByteBuffer
isFinished():boolean
flip(ByteArrayOutputStream):ByteBuffer
verticalflip(BufferedImage):BufferedImage
modifiesHeaders():boolean
<<Java Class>>
HTTPL33tEncoder
org.chinux.pdc.http.impl.readers
charset: Charset
encoder: CharsetEncoder
decoder: CharsetDecoder
responseHeader: HTTPResponseHeader
stream: ByteArrayOutputStream
isFinished: boolean
buff: String
currentContentLenght: int
bytesToRead: Integer
bytesRead: int
HTTPL33tEncoder(HTTPResponseHeader)
processData(ByteBuffer):ByteBuffer
isFinished():boolean
modifiesHeaders():boolean
translate(String):String
<<Java Class>>
ChunkedInputEndDetector
org.chinux.pdc.http.util
DETECTING_LEN: int
READING_CHUNK: int
isoCharset: Charset
log: Logger
state: int
sizeToRead: int
sizeRead: int
isOver: boolean
inStream: ByteArrayOutputStream
outputStream: ByteArrayOutputStream
ChunkedInputEndDetector()
write(byte[]):void
read():ByteBuffer
chunkedInputOver():boolean

Data Events

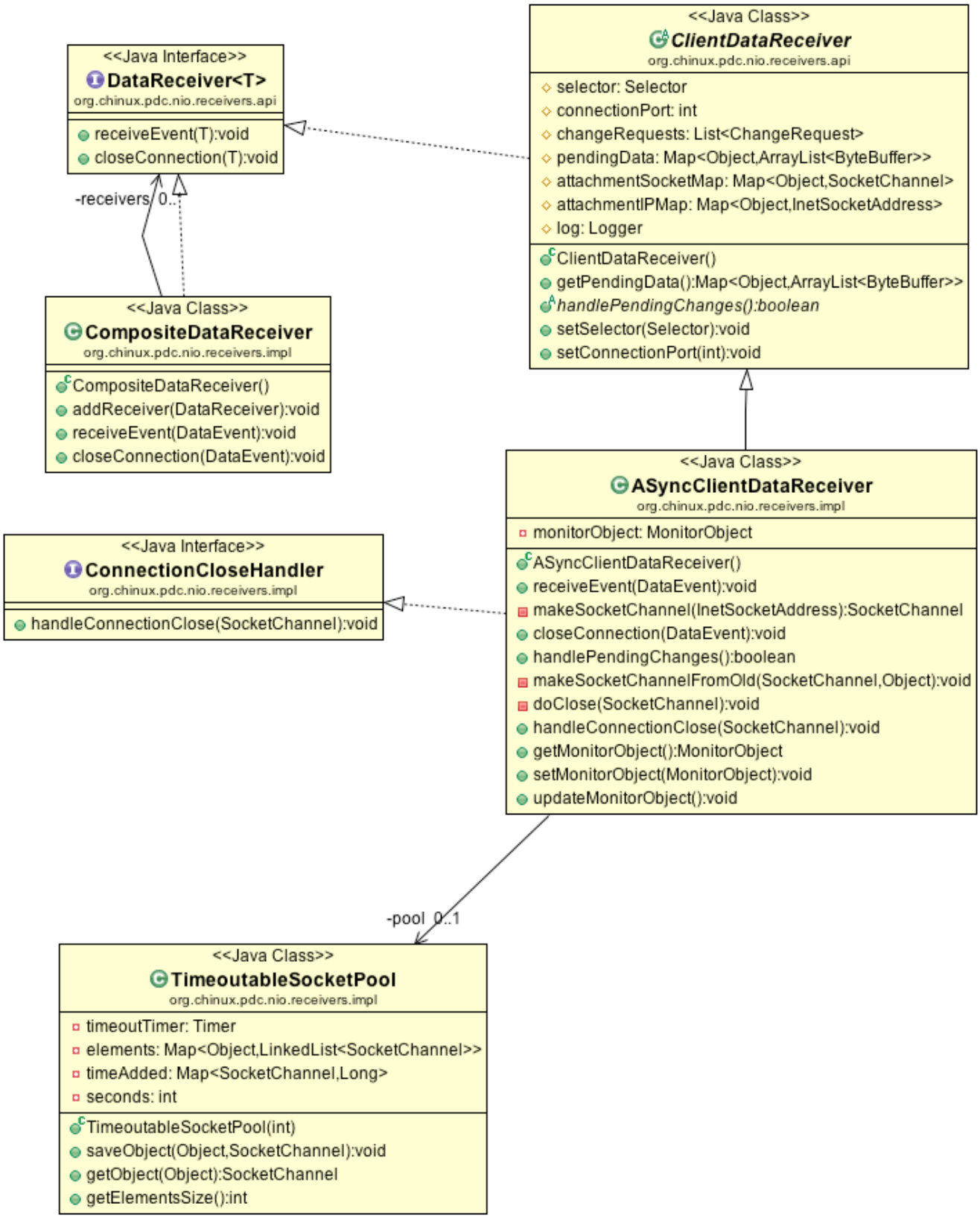

<<Java Class>>
DataEvent
org.chinux.pdc.nio.events.api
receiver: DataReceiver<DataEvent>
data: ByteBuffer
canSend: boolean
canClose: boolean
toString():String
DataEvent(ByteBuffer,DataReceiver<DataEvent>)
DataEvent(ByteBuffer,DataReceiver<DataEvent>,boolean,boolean)
setCanClose(boolean):void
setCanSend(boolean):void
getData():ByteBuffer
setData(ByteBuffer):void
canSend():boolean
canClose():boolean
getReceiver():DataReceiver<DataEvent>
clone():Object
<<Java Class>>
ErrorDataEvent
org.chinux.pdc.nio.events.impl
PROXY_CLIENT_DISCONNECT: int
REMOTE_CLIENT_DISCONNECT: int
errorType: int
attachment: Object
owner: Object
ErrorDataEvent(int,Object,Object)
getAttachment():Object
getErrorType():int
getReceivers():CompositeDataReceiver
getOwner():Object
setOwner(Object):void
<<Java Class>>
ServerDataEvent
org.chinux.pdc.nio.events.impl
channel: SocketChannel
ServerDataEvent(SocketChannel,ByteBuffer)
ServerDataEvent(SocketChannel,ByteBuffer,DataReceiver<DataEvent>)
getChannel():SocketChannel
toString():String
<<Java Class>>
ClientDataEvent
org.chinux.pdc.nio.events.impl
address: InetAddress
attachment: Object
owner: Object
port: Integer
ClientDataEvent(ByteBuffer,Object,Object)
ClientDataEvent(ByteBuffer,Object)
ClientDataEvent(ByteBuffer,InetAddress,Object)
ClientDataEvent(ByteBuffer,DataReceiver<DataEvent>,InetAddress,Object,Object)
getAddress():InetAddress
getAttachment():Object
toString():String
getOwner():Object
setOwner(Object):void
getPort():Integer
setPort(Integer):void

Requests and responses modeling




<<Java Interface>>
Worker<T>
org.chinux.pdc.workers.api
DoWork(T)
<<Java Enumeration>>
Code
org.chinux.pdc.server
BADLOGIN: Code
OTHERTERMINALOPEN: Code
OKLOGIN: Code
GETPASSWORD: Code
Code()
<<Java Class>>
LogueableWorker
org.chinux.pdc.workers.impl
data: String
LogueableWorker()
resetWorkerState():void
obtainCommand(DataEvent):String
helo(DataEvent,String,User):DataEvent
-worker 0..1
~loginservice
0..1
<<Java Class>>
LoginService
org.chinux.pdc.server
getInstance():LoginService
resetInstance():void
LoginService()
login(DataEvent,String,User):Code
createResponseEvent(Code,DataEvent):DataEvent
isLogged(Code):boolean
isLogged(String):boolean
-instance
0..1
<<Java Class>>
ConfigurationWorker
org.chinux.pdc.workers.impl
BROWSER: int
OPERATING_SYSTEM: int
IP_ADDRESS: int
SUBNET: int
ipPattern: Pattern
subcommands: String[]
ConfigurationWorker()
DoWork(DataEvent):DataEvent
resetWorkerState():void
<<Java Class>>
MonitorWorker
org.chinux.pdc.workers.impl
MonitorWorker()
getMonitorObject():MonitorObject
setMonitorObject(MonitorObject):void
DoWork(DataEvent):DataEvent
resetWorkerState():void
truncate(double,int):double
<<Java Class>>
MonitorDataPoller
org.chinux.pdc.server
run():void
+monitorObject 0..1
-usersByUserName
0..*
#users
0..*
<<Java Class>>
User
org.chinux.pdc.server
username: String
password: String
srcport: int
host: InetAddress
logged: boolean
greeted: boolean
User(String,String,int,InetAddress)
User(int,InetAddress)
isGreeted():boolean
setGreeted(boolean):void
getUsername():String
setUsername(String):void
getPassword():String
setPassword(String):void
getSrcport():int
setSrcport(int):void
getHost():InetAddress
setHost(InetAddress):void
hashCode():int
equals(Object):boolean
isLogged():boolean
setLogged(boolean):void
toString():String
<<Java Class>>
ConfigurationProvider
org.chinux.pdc.server
ConfigurationProvider()
getDefaultConfiguration():Configuration
setConfiguration(Configuration,Object):void
getFromInetSubmask(String):Configuration
getConfigurationFromIP(InetAddress):Configuration
getConfigurationFromBrowser(Browser):Configuration
getConfigurationFromOperatingSystem(OperatingSystem):Configuration
getConfigurationFromSubnet(SubnetInfo):Configuration
getConfiguration(HTTPClientInfo):Configuration
<<Java Class>>
MonitorObject
org.chinux.pdc.server
clientDataReceiver: ASyncClientDataReceiver
httpProxyWorkers: Set<HTTPProxyWorker>
fromClientsTransferedBytes: int
fromServersTransferedBytes: int
openedConnectionsQuant: int
imageFlipsQuant: int
text2L33tQuant: int
allAccessBlocksQuant: int
ipBlocksQuant: int
contentTypeBlocksQuant: int
tooBigResourceBlocksQuant: int
urlBlocksQuant: int
clientConnectionsQuant: int
originServerConnectionsQuant: int
connectionsQuant: int
timedConnections: int
timer: Timer
MonitorObject()
addFromClientsBytes(int):void
addFromServersBytes(int):void
getAllAccessBlocksQuant():int
getClientConnectionsQuant():int
getClientDataReceiver():ASyncClientDataReceiver
getConnectionsQuant():int
getContentTypeBlocksQuant():int
getFromClientsTransferedBytes():int
getFromServersTransferedBytes():int
getHttpProxyWorker():Set<HTTPProxyWorker>
getImageFlipsQuant():int
getIpBlocksQuant():int
getOpenedConnectionsQuant():int
getOriginServerConnectionsQuant():int
getText2L33tQuant():int
getTimedConnections():int
getTooBigResourceBlocksQuant():int
getTotalBlocksQuant():int
getTotalTransferedBytes():int
getTransformationsQuant():int
getUrlBlocksQuant():int
increaseAllAccessBlocksQuant():void
increaseContentTypeBlocksQuant():void
increaseImageFlipsQuant():void
increaseIpBlocksQuant():void
increaseText2L33tQuant():void
increaseTooBigResourceBlocksQuant():void
increaseUrlBlocksQuant():void
setAllAccessBlocksQuant(int):void
setClientConnectionsQuant(int):void
setClientDataReceiver(ASyncClientDataReceiver):void
setConnectionsQuant(int):void
setContentTypeBlocksQuant(int):void
setFromClientsTransferedBytes(int):void
setFromServersTransferedBytes(int):void
setHttpProxyWorkers(Set<HTTPProxyWorker>):void
setImageFlipsQuant(int):void
setIpBlocksQuant(int):void
setOpenedConnectionsQuant(int):void
setOriginServerConnectionsQuant(int):void
setText2L33tQuant(int):void
setTimedConnections(int):void
setTooBigResourceBlocksQuant(int):void
setUrlBlocksQuant(int):void
<<Java Class>>
Configuration
org.chinux.pdc.server
blockAll: boolean
blockedIPs: List<String>
blockedURLs: List<String>
blockedMediaTypes: List<String>
maxResSize: int
l33t: boolean
rotateImages: boolean
chainProxy: boolean
maxResEnabled: boolean
chainProxyPort: Integer
chainProxyHost: String
Configuration()
Configuration(boolean,List<String>,List<String>,List<String>,int,boolean,boolean,boolean,boolean,Integer,String)
getBlockedIPs():List<String>
getBlockedMediaTypes():List<String>
getBlockedURLs():List<String>
getChainProxyHost():String
getChainProxyPort():Integer
getMaxResSize():int
isBlockAll():boolean
isChainProxy():boolean
isL33t():boolean
isMaxResEnabled():boolean
isRotateImages():boolean
toString():String
<<Java Interface>>
EventDispatcher<T>
org.chinux.pdc.nio.dispatchers
processData(T):void
<<Java Class>>
SyncEventDispatcher<T>
org.chinux.pdc.nio.dispatchers
SyncEventDispatcher(Worker<T>)
processData(T):void

Receivers


<<Java Interface>>
DataReceiver<T>
org.chinux.pdc.nio.receivers.api
receiveEvent(T):void
closeConnection(T):void
-receivers 0..*
<<Java Class>>
ClientDataReceiver
org.chinux.pdc.nio.receivers.api
selector: Selector
connectionPort: int
changeRequests: List<ChangeRequest>
pendingData: Map<Object,ArrayList<ByteBuffer>>
attachmentSocketMap: Map<Object,SocketChannel>
attachmentIPMap: Map<Object,InetSocketAddress>
log: Logger
ClientDataReceiver()
getPendingData():Map<Object,ArrayList<ByteBuffer>>
handlePendingChanges():boolean
setSelector(Selector):void
setConnectionPort(int):void
<<Java Class>>
CompositeDataReceiver
org.chinux.pdc.nio.receivers.impl
CompositeDataReceiver()
addReceiver(DataReceiver):void
receiveEvent(DataEvent):void
closeConnection(DataEvent):void
<<Java Class>>
ASyncClientDataReceiver
org.chinux.pdc.nio.receivers.impl
monitorObject: MonitorObject
ASyncClientDataReceiver()
receiveEvent(DataEvent):void
makeSocketChannel(InetSocketAddress):SocketChannel
closeConnection(DataEvent):void
handlePendingChanges():boolean
makeSocketChannelFromOld(SocketChannel,Object):void
doClose(SocketChannel):void
handleConnectionClose(SocketChannel):void
getMonitorObject():MonitorObject
setMonitorObject(MonitorObject):void
updateMonitorObject():void
<<Java Interface>>
ConnectionCloseHandler
org.chinux.pdc.nio.receivers.impl
handleConnectionClose(SocketChannel):void
-pool 0..1
<<Java Class>>
TimeoutableSocketPool
org.chinux.pdc.nio.receivers.impl
timeoutTimer: Timer
elements: Map<Object,LinkedList<SocketChannel>>
timeAdded: Map<SocketChannel,Long>
seconds: int
TimeoutableSocketPool(int)
saveObject(Object,SocketChannel):void
getObject(Object):SocketChannel
getElementsSize():int

Dispatchers