

---

# WEB SERVER

---

CON ADATTAMENTO DINAMICO DI CONTENUTI STATICI



A.A. 2019/2020

**Professore:**

Francesco Lo Presti

**Studenti:**

Emanuele Alfano  
Filippo Badalamenti

## Sommario

<b>Scelte di sviluppo.....</b>	<b>3</b>
Tool di progettazione .....	3
Tool di sviluppo .....	3
<b>Struttura Sistema .....</b>	<b>4</b>
Generale .....	4
Interazione Tra i Componenti .....	5
Vantaggi Architettureali .....	5
Dad process .....	6
SimpleWeb .....	6
Connection .....	6
HttpHeader.....	6
NCS (Network Control System) .....	7
Accept.....	7
Queue.....	8
NCS workflow .....	9
CES (Core Elaboration System).....	10
Worker .....	10
HttpMgt.....	10
HttpMessage .....	11
Cache Management .....	11
<b>Esempi di esecuzione.....</b>	<b>14</b>
Flusso interno.....	14
Interazioni Esterne .....	14
<b>Sviluppo e testing .....</b>	<b>15</b>
Strumenti di sviluppo .....	15
Linguaggi usati.....	15
Analisi dei Test .....	15
<b>Manuali d'installazione ed uso .....</b>	<b>16</b>

## Scelte di sviluppo

Il nostro gruppo di lavoro ha optato per la creazione di un Web Server in **C++ 17** (con le funzioni relative alla parte di rete scritte in C), che potesse sfruttare due caratteristiche utili alla scalabilità del progetto:

- Impiego della funzione *poll()* (al posto di *select()*), per consentire operazioni non bloccanti sulla socket d'ascolto e la gestione asincrona dell'I/O.
- L'utilizzo di classi, per astrarre le funzionalità dei moduli e permettere un maggior incapsulamento delle funzionalità.
- La possibilità di creare un *Log* delle attività del server su file, facilmente attivabile a tempo di compilazione e con impatto ridotto sulle performance del sistema, utile in fase di debug e manutenzione del sistema.

## Tool di progettazione

Nella modellazione del sistema, è risultato cruciale l'utilizzo di **UML** (Unified Modeling Language), in modo da ridurre allo stretto necessario l'interdipendenza del codice, la sovrapposizione di funzionalità - possibile dovendo lavorare da remoto in regioni diverse d'Italia -, e la gestione della sincronia. Pertanto, nel mostrare le caratteristiche del sistema, verranno utilizzati solamente diagrammi UML all'interno della relazione, lasciando l'implementazione completa delle funzioni nel luogo ad essa opportuno, nel codice sorgente.

## Tool di sviluppo

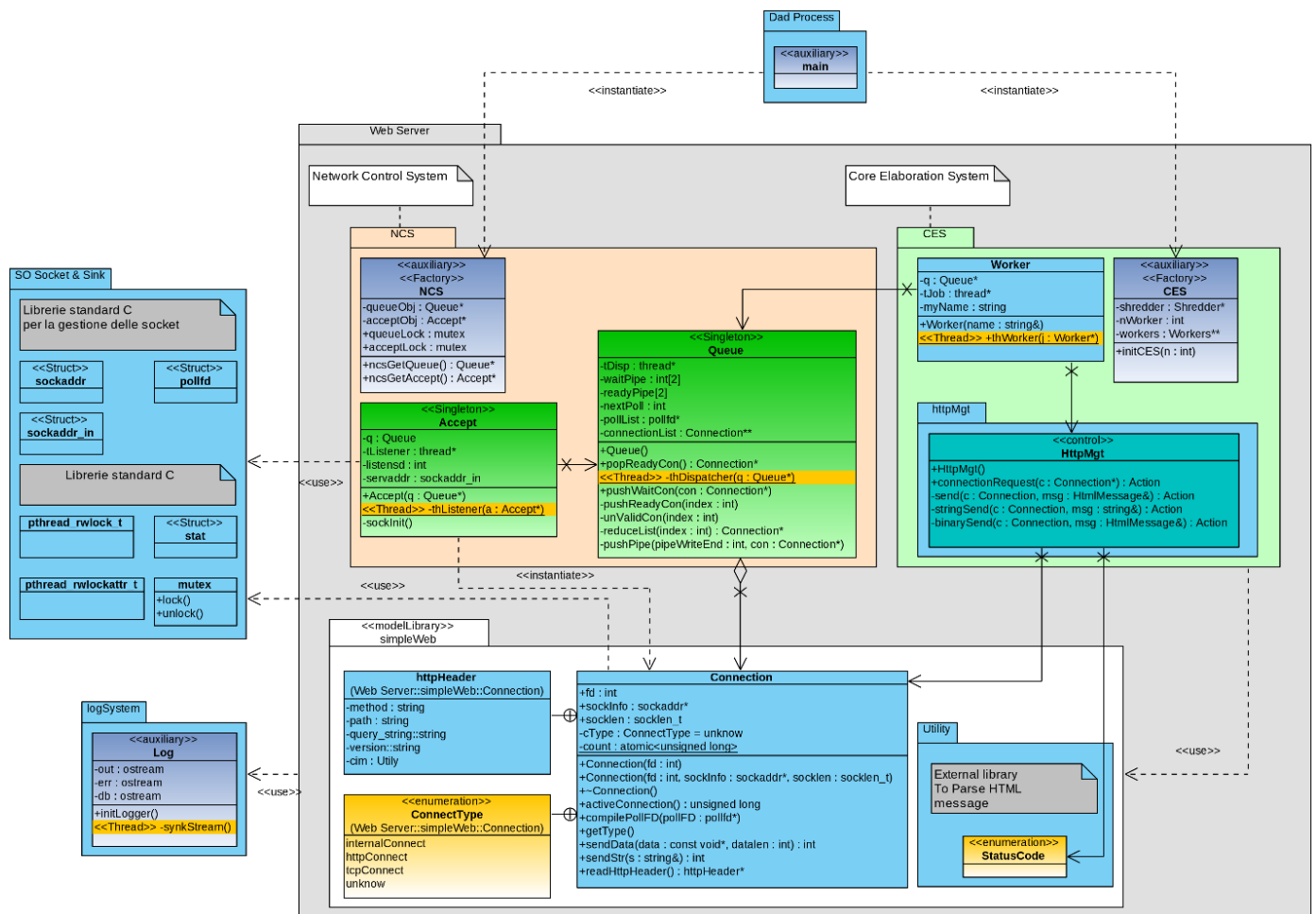
Tra i tool di sviluppo ed analisi possiamo annoverare:

- **CMake**, per la creazione dei Makefile che, chiamati tramite *make*, forniscono gli eseguibili del progetto.
- Tool di debug dinamico quali **GDB** e **Address Sanitizer** per lo sviluppo del codice.
- L'uso di **GoHttpBench** per l'esecuzione dei test di benchmark sul server.
- Script in **Python** e **Matlab** per la raccolta dei dati di benchmark e la loro visualizzazione su grafici.

Il progetto si trova al seguente indirizzo: <https://github.com/Alfystar/IIW2020>. Si consiglia la lettura dei Readme e le varie guide presenti direttamente su GitHub, poiché scritte in linguaggio *markdown* e quindi visualizzate in modo corretto nel visualizzatore integrato del sito.

# Struttura Sistema

## Generale



Il Server Web “BadAlpha” (crasi e fusione dei nostri cognomi) è formato da una serie di moduli, ognuno dei quali soddisfa una specifica necessità del sistema; essi sono a loro volta composti dalle classi necessarie al loro funzionamento. In particolare, abbiamo:

- **SimpleWeb**, container di classi, utili a livello di connettività sulla rete e comuni a più parti del sistema.
- **NCS (Network Control System)**, deputato all’accettazione delle connessioni TCP ed al tracciamento di quelle già generate; inoltre, si preoccupa di consegnare ai singoli Thread di elaborazione (da qui in avanti definiti “Worker”) le connessioni che hanno ricevuto nuove richieste.
- **CES (Core Elaboration System)**, deputato alla gestione delle richieste dei client, quali:
  - Creazione messaggi HTTP opportuni.
  - Recupero e invio delle pagine HTML salvate nel file system
  - Elaborazione dell’immagine (attraverso un opportuno sottosistema)

In aggiunta, come forma di resilienza, il server non è direttamente il processo principale, bensì è ottenuto tramite **fork()**, in modo da permettere, in caso eventuali segnali non gestiti nel programma, la sua ripartenza automatizzata, in base al codice di errore di ritorno alla terminazione del figlio (**Dad Process**).

Infine, è possibile attivare o disattivare, tramite delle macro di compilazione, varie funzionalità del programma, così da generare diverse versioni del server, o disattivare i moduli di debug in fase di Release del codice. Per i dettagli, vedere la [guida alla compilazione](#).

## Interazione Tra i Componenti

Prima di entrare nel dettaglio dei singoli sottosistemi, risulta utile descrivere come essi cooperino per permettere di offrire le funzionalità base di un Web Server. Nel nostro progetto abbiamo cercato di coniugare i punti di forza presenti nelle varie architetture possibili, scegliendo per:

1. Struttura del codice: server multi-threaded con pre-threading statico.
2. Gestione delle richieste: schema Leader-Follower.
3. Assegnazione delle richieste: ad Eventi.

Addentrando in una descrizione più discorsiva, il server:

- È multi-Threaded con Pre-threading statico, poiché il processo esposto in rete è unico e contiene al proprio interno più Thread, dei quali tre fissi (**Accept**, **Queue**, **Logger**) ed **N** parametrici (**Worker**).  
Il Pre-threading statico è calibrato sulle caratteristiche della macchina sottostante, trovando un compromesso tra grado di parallelismo e sovra utilizzo di risorse.
- Segue lo schema Leader-Follower, dato che l'**Accept** è l'unica entità che entra in contatto con le connessioni dei client ed esse vengono inviate alla **Queue**, la quale assegna dinamicamente ai **Worker** una richiesta da gestire.
- Assegna le richieste tramite Eventi, in quanto la **Queue** è in attesa su delle *pending connections*, e cede il controllo ai **Worker** solo su quelle che hanno del lavoro da essere svolto. I **Worker**, terminata la richiesta, se non chiudono loro stessi la comunicazione, riaffidano la gestione della connessione alla **Queue**, e chiedono a quest'ultima una nuova richiesta da evadere.

## Vantaggi Architettureali

Grazie a questo mix di caratteristiche, il server risulta **scalabile**, in quanto riassegna dinamicamente le risorse su un numero di worker fisso ed ottimale anziché istanziare all'occorrenza nuovi thread per la gestione delle richieste. Infatti, in un confronto d'esempio con 3000 connessioni parallele da gestire:

Server Multi-Thread	Server BadAlpha
L'unico Thread presente è l'Accept, e il sistema dovrà quindi: <ol style="list-style-type: none"><li>1. Restare in ascolto sulla Socket.</li><li>2. Per ogni connessione, generare un thread che possa evaderla completamente (e in quanto thread, dovrà essere schedulato dal sistema operativo)</li><li>3. Portare avanti le 3000 connessioni, con il SO che dovrà garantire la <i>fairness</i> tra tutti i thread in stato <b>ready</b>.</li><li>4. De-allocare 3000 Thread e 3000 Connessioni</li></ol>	Poiché sono già state allocate staticamente tutte le strutture necessarie del sistema: <ol style="list-style-type: none"><li>1. L'Accept è in ascolto sulla Socket</li><li>2. Vengono generate 3000 istanze di Connection (la risorsa che mappa le connessioni del server)</li><li>3. Le richieste vengono assegnate e quindi evase dai Worker, ed al termine di ognuna di esse viene riassegnata alla Queue (se la connessione è ancora aperta), passando quindi a gestire un'altra richiesta pendente.</li><li>4. Vengono distrutte le 3000 Connection.</li></ol>

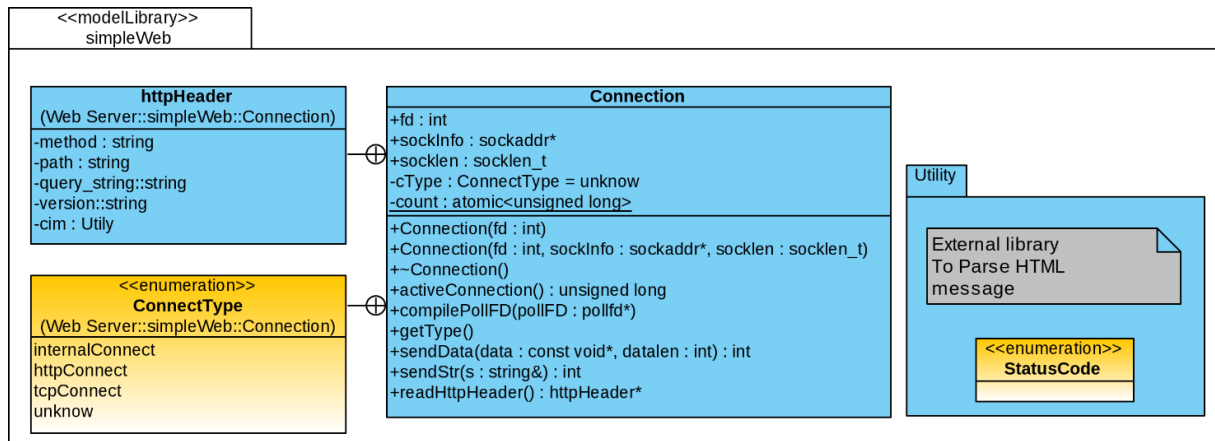
Confrontando le due scelte, risulta evidente come la soluzione da noi scelta riduca fortemente l'overhead causato dalla creazione di strutture nel kernel, lasciandone solo in *Userspace* durante l'allocazione e de-allocazione dinamica delle connessioni tra server e client.

## Dad process

Il server è stato progettato tenendo in considerazione che la *fork()*, pur avendo una maggiore intuitività nell'utilizzo, ha di contro un overhead di due ordini di grandezza superiore alla creazione di un thread. Pertanto, l'unico suo impiego nel nostro progetto si ha per consentire la ripartenza del server nel caso di errori di quest'ultimo, analizzando i codici di uscita per risalire alla causa.

Inoltre, il *main()* del processo padre si occupa di effettuare, prima della *fork()*, il setup del sistema, come il cambio della directory corrente di lavoro e l'impostazione di alcune variabili necessarie al programma e alla raccolta di risultati per i test.

## SimpleWeb



Questo package è pensato per contenere le funzionalità utili e comuni alla parte di rete del sistema.

È diviso in due blocchi principali:

- Il **sub-package Utility**, una libreria open-source per il *parsing* dell'Header di richiesta HTTP.
- La **classe Connection**, una **Entity** del sistema che ha il compito di interagire con le Socket del sistema operativo per lo scambio dei messaggi.

Entrambi accorpano al proprio interno tutte le logiche a “basso livello” del server, permettendo di interfacciarsi tramite chiamate a metodi più astratti ed intuitivi.

### Connection

La classe, oltre a rappresentare una connessione aperta, è l'unica a poter usare direttamente la Socket sottostante, mentre espone metodi per le altre classi che vi volessero interagire quali:

- *SendData*
- *SendStr*
- *readHttpHeader*

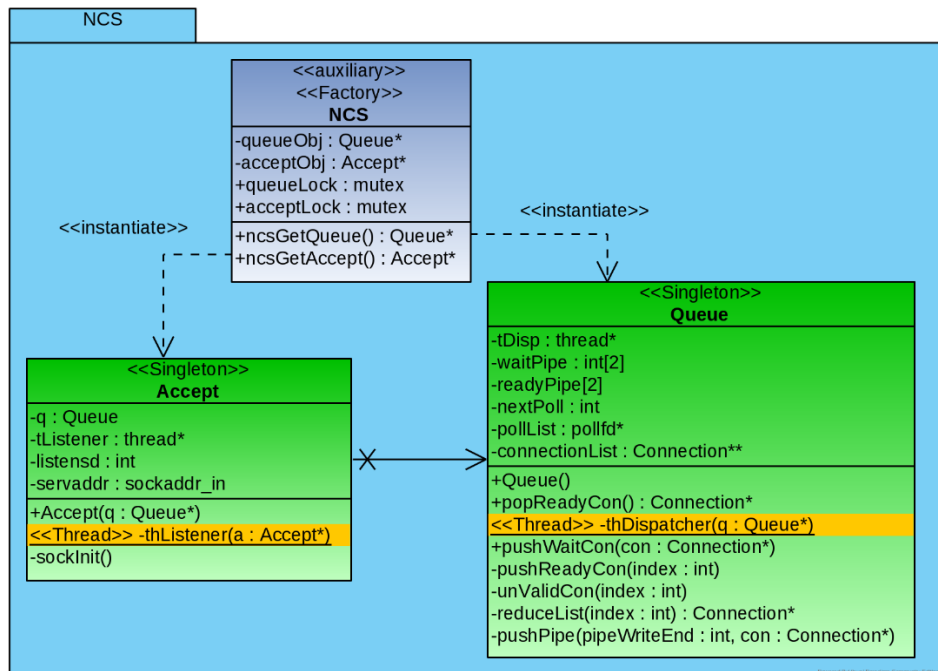
Il distruttore di `Connection` si occupa anche di chiudere la connessione TCP che contiene, e libera tutte le risorse in precedenza allocate per svolgere la richiesta del client.

L'unica classe che ha l'autorizzazione a creare nuove istanze di **Connection** è l'**Accept**, presente all'interno del sottosistema **NCS** (Network Core System). Inoltre, per semplificare il lavoro della **Queue** con la *poll()*, la `Connection` può rappresentare anche un semplice file descriptor ([vedi \*ppoll\(\)\* nella Queue per approfondire](#)).

### HttpHeader

Classe accessoria utilizzata dal sottosistema **CES** (Core Elaboration System), i cui campi vengono compilati in base alla richiesta del client usando metodi presenti nella libreria open-source contenuta in `Utility`.

# NCS (Network Control System)



Il **Network Control System** è il sottosistema delegato alla gestione e mantenimento delle connessioni all'interno del programma. Esso segue il pattern *Factory Method*, dove la classe **NCS** si occupa, oltre a contenere le opportune variabili di sincronia, di istanziare e conservare i riferimenti delle due componenti principali:

- **Accept**
- **Queue**

Entrambe le classi, inoltre, possiedono al loro interno un thread che permette loro di svolgere in concorrenza gli incarichi ad esse assegnati.

## Accept

La classe **Accept** è la prima con la quale una richiesta esterna viene in contatto col sistema. Ad essa è delegato il compito di:

1. Avviare la Socket di comunicazione che espone il server.
2. Restare in ascolto della Socket per nuove connessioni.
3. Istanziare una Connection per tenere traccia della connessione.
4. Impostare correttamente i parametri della connessione.

Al ricevimento di una connessione valida, quest'ultima viene trasmessa alla **Queue** per farla gestire successivamente dai **Worker**.

## Socket Option

Nell'Accept la Socket è stata configurata utilizzando i seguenti parametri:

### A livello di processo

- **RLIMIT\_NOFILE** := il limite di FD del processo.

Impostato pari a **due** volte il numero di connessioni massime ammissibili, in quanto oltre alle socket, è necessario interagire tramite FD con il filesystem.

### Socket di Ascolto

- **SO\_REUSEADDR** := permette il riutilizzo della porta utile in caso di crash del server.

### Socket delle Connessioni:

- **SO\_KEEPALIVE** := Keep alive nel server.
- **TCP\_KEEPINTVL** := tempo di attesa dall'attivazione del KeepAlive, impostato a 1 secondo.
- **TCP\_KEEPCNT** := Numero Probe del KeepAlive, 10.
- **TCP\_NODELAY** := On/Off [Nagle Algorithm](#), Attivato per aumentare l'efficienza complessiva.

## Queue

La classe **Queue** ha il compito di osservare un vettore di Connection valide ma senza trasmissioni in atto, e in caso una di esse avesse un nuovo messaggio da leggere, modificarne lo stato in **READY** e assegnarla ad un worker non appena disponibile. Prima di proseguire, è opportuno definire i possibili stati di una connessione:

1. **WAIT** state: Connessione VALIDA ma senza alcun pacchetto in arrivo
2. **READY** state: Connessione VALIDA e con dei dati leggibili dalla socket

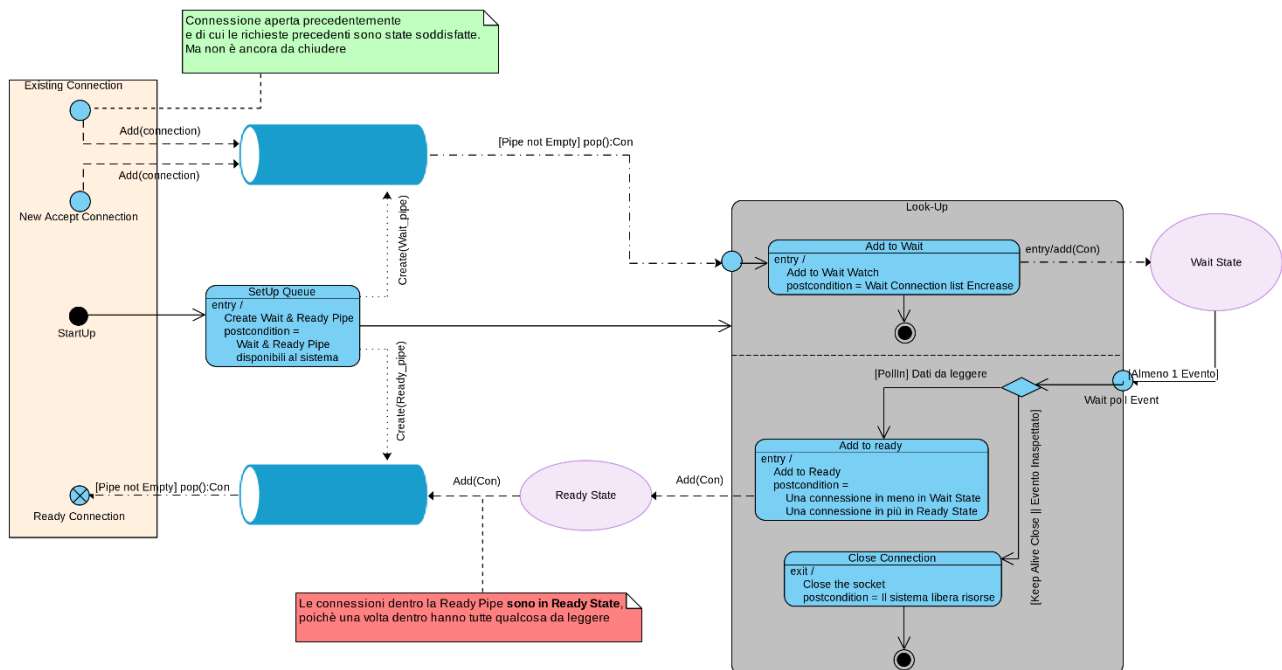
Tutte le connessioni in stato **WAIT** sono poste in un'opportuna lista da fornire alla **ppoll()**, e quest'ultima, presente nella classe Queue, metterà il thread della classe stessa in attesa, mentre l'esecuzione verrà ripresa non appena una di queste connessioni sarà disponibile, o verrà chiusa forzatamente dal client. Si è optato per questa system call per le seguenti ragioni:

1. La **poll()** è una system-call più moderna della **select()**, che permette di osservare una lista di FD specificata dal programmatore, e scegliere (essendo una funzione bloccante) il tipo di evento per il quale si debba riprendere l'esecuzione.
2. La variante **ppoll()** permette l'attivazione di una maschera di segnali del sistema operativo da ignorare, consentendo quindi di gestire l'errore tramite **errno**.

La **poll()** richiede sia un puntatore ad un array di struct, che la lunghezza dello stesso; per evitare che la chiusura delle connessioni possa lasciare degli elementi non validi all'interno dell'array e ridurre il tempo di elaborazione della system call (che cresce in maniera lineare alla dimensione del vettore), si è deciso di adottare la seguente politica sull'aggiunta e rimozione delle struct:

- Ogni nuovo elemento viene aggiunto in coda all'array, il puntatore all'ultimo elemento viene spostato di conseguenza e viene effettuato +1 al numero di FD.
- Alla chiusura di una connessione, si chiude il relativo file descriptor, l'elemento viene eliminato e il suo posto è preso dall'ultimo elemento in coda nell'array; alla fine, viene fatto -1 al numero dei file descriptor.

Il funzionamento della classe e le azioni che essa può compiere sono sintetizzati nel seguente "pseudo" State Diagram:



Come è possibile osservare, le connessioni che passano allo stato **READY** vengono poste in una **coda FIFO**, in modo che i Worker si possano porre sull'estremo di uscita ed ottenere la prima Connection disponibile in tale stato.

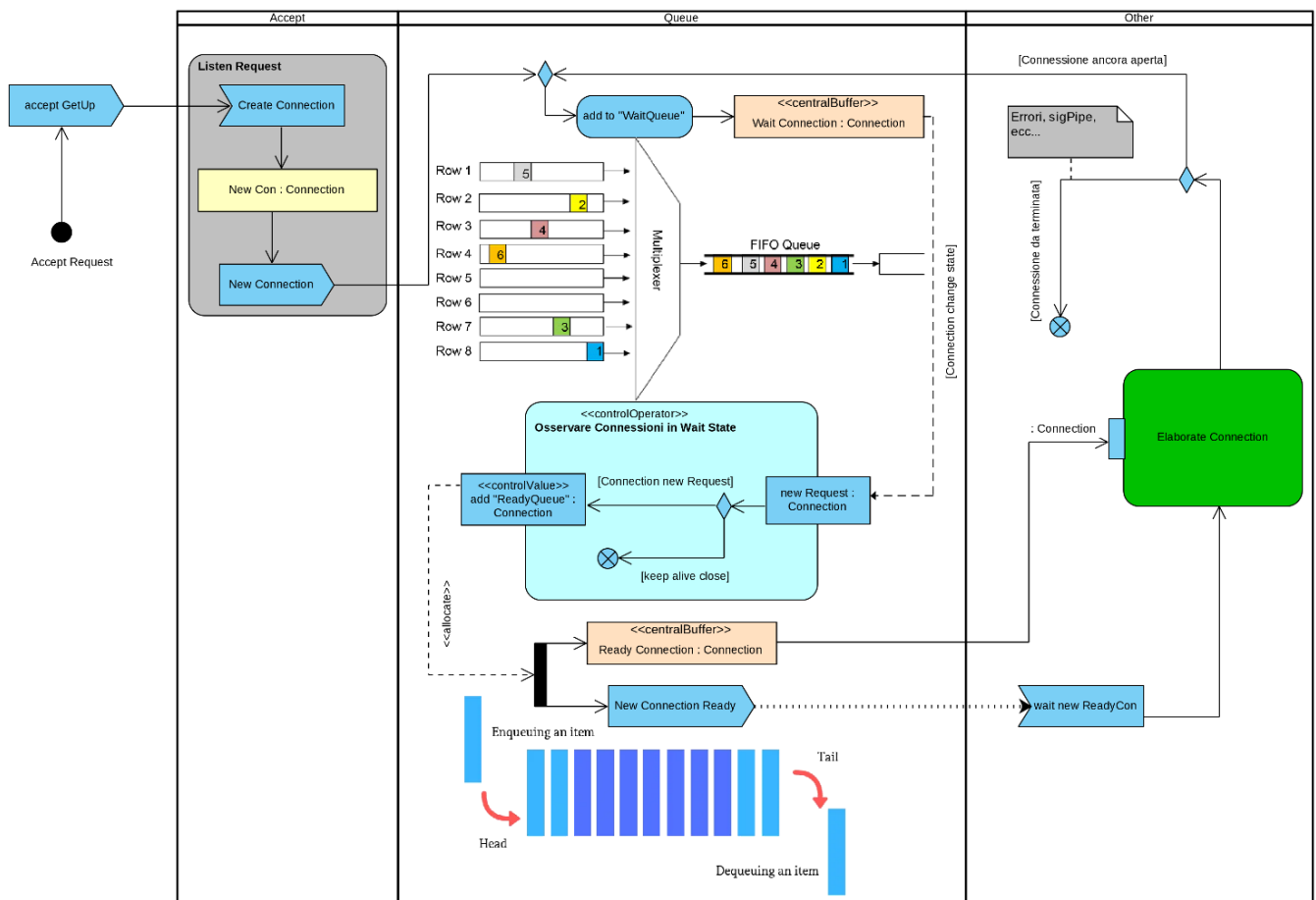


Le code sono state implementate usando le *pipe* del sistema operativo (mostrate nel diagramma come due tubi), che semplificano la gestione della concorrenza in quanto *thread-safe* purché ogni *write* effettuata sopra di esse non superi le dimensioni del buffer interno al sistema operativo (e spostando solo puntatori di Connection, siamo ampiamente sotto tale limite).

Segue che l'utilizzo del processore è rilevante soltanto se presente dell'effettivo lavoro da svolgere, mentre nel rimanente tempo il sistema rimane totalmente in idle (0.0% su *htop* per la CPU).

## NCS workflow

Il seguente Activity Diagram mostra lo svolgimento delle attività di **NCS** (il ruolo di **CES** in questo caso è schematizzato da *Other*, poiché NCS è agnostico sul come vengano usate le connessioni):



È possibile notare come il sottosistema NCS, dall'esterno, sia accessibile solo da:

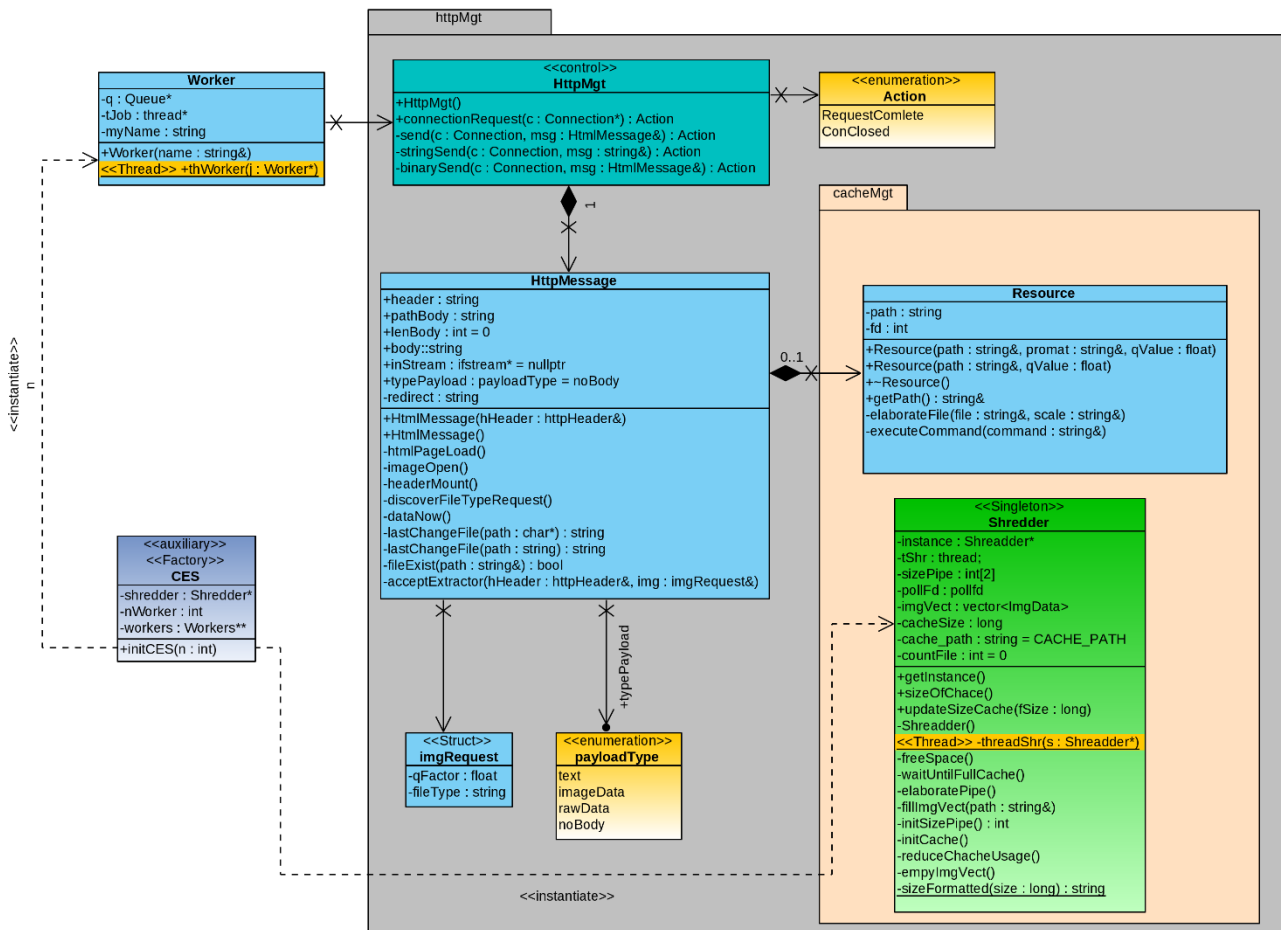
1. **Accept**, che ascoltando la Socket avvia una nuova connessione;
2. **Queue**, per restituire una connessione ancora valida ma in WAIT.

Di contro, l'unica uscita del sottosistema sono le Connection **READY** ritornate dalla **Queue**.

Risulta evidente come il cuore del sistema sia a tutti gli effetti la **Queue**, la quale gestisce connessioni in **WAIT** e le rilascia quando quest'ultime passano allo stato **READY**. I punti di ingresso ed uscita alla Queue sono delle code FIFO (come visibile nella parte bassa dell'Activity Diagram), in modo da bufferizzare le richieste.

Per un ulteriore approfondimento sull'implementazione dell'Accept e della Queue, è possibile trovare il codice sorgente nella directory ["2\\_src/ncs"](#).

# CES (Core Elaboration System)



Questo sottosistema racchiude le unità funzionali che permettono la gestione delle richieste dei client.

La sua struttura segue il pattern UML della *Factory Method*, per cui, una volta chiamata la classe dal main() del programma, tramite la funzione *initCES()* vengono istanziate le sottoclassi necessarie al sistema, ovvero i **Worker** e lo **Shredder**. Essendo l'unico punto di inizializzazione di queste istanze, è garantito il controllo sul livello di *concurrency* delle elaborazioni e sull'accesso al *caching* su disco.

## Worker

I Worker si presentano come classi con al proprio interno un *Thread* ("named" per facilitare la tracciabilità degli stessi durante l'esecuzione) che ha il compito di elaborare le richieste dei client. La presa in carico avviene non appena una delle connessioni **READY** bufferizzate dalla Queue è disponibile; queste *pending connections*, infatti, devono ancora svolgere la propria attività, ed una volta completata la richiesta, si potranno verificare i seguenti casi:

- La Connection è ancora **valida** e potrebbe essere usata per ricevere nuove HttpRequest da parte del client. In tal caso, il Worker cede la connessione alla Queue che la possederà finché rimarrà in stato di **WAIT**.
- La Connection è **terminata** per un segnale d'errore o era di tipo *close*. Segue pertanto la chiusura della Socket corrispondente e vengono liberate le risorse di memoria usate per tenere traccia della connessione (responsabilità di Connection nel distruttore).

## HttpMgt

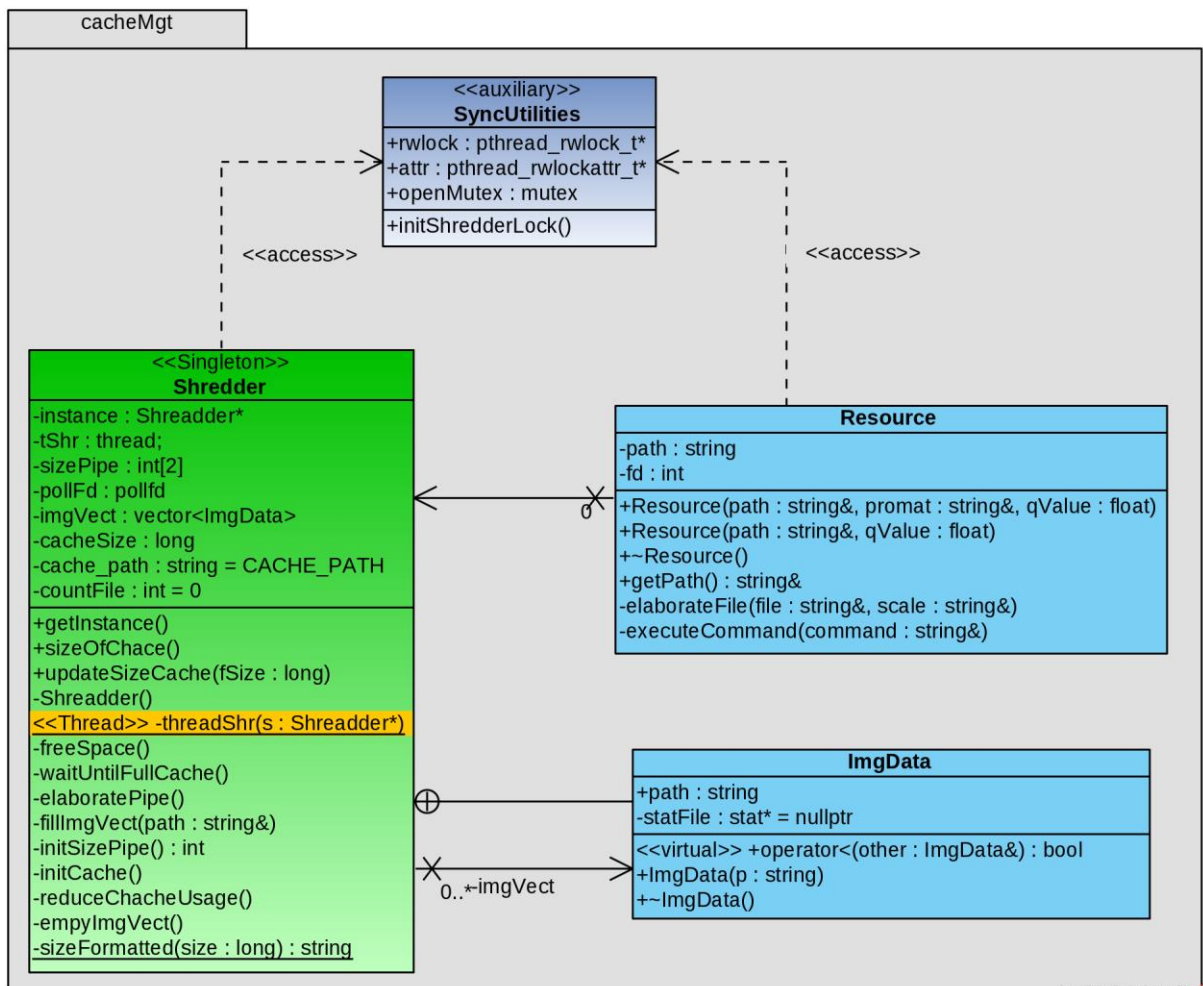
Tale classe ha la responsabilità di gestire la Connection ad essa passata, e al termine della stessa ritornare al chiamante lo stato della connessione (ancora valida, quindi da conservare, o con errore, quindi da terminare). Inoltre, possiede la responsabilità dell'invio della risposta HTTP, mentre la compilazione opportuna di quest'ultima avviene tramite i metodi messi a disposizione dalla classe *HttpMessage*.

## HttpMessage

Questa classe ha la responsabilità di compilare l'Header del messaggio HTTP di risposta e di accodare ad esso il body opportuno leggendolo da file system; in caso di errore, viene creata una pagina HTML dinamica per comunicare al client lo specifico errore avvenuto.

Nella versione statica del Web Server, le risorse richieste dal client vengono direttamente accedute su file system, mentre la variante con ridimensionamento delle immagini interPELLa il Cache Management, il quale fornirà l'opportuno *path* al quale accedere per ottenere l'immagine nel formato desiderato.

## Cache Management



Il sottosistema Cache Management si occupa della creazione e gestione di immagini temporanee al fine di ridurre i tempi medi di elaborazione della risorsa richiesta in uno specifico formato o dimensione.

Il programma utilizzato per l'elaborazione dei file è **ImageMagick**<sup>1</sup>, che esiste in ambiente UNIX come eseguibile stand-alone e quindi riduce la dipendenza del server dai pacchetti presenti nel sistema. Nel caso di conversione di immagini in formato *webp*<sup>2</sup>, è necessaria prima la decodifica in PNG tramite il pacchetto *dwebp* (menzionato nei manuali), e successivamente l'utilizzo di ImageMagick, poiché nella sua release pubblica il programma non include tale estensione. (sarebbe necessaria la ricompilazione del codice sorgente, ma risulterebbe un lavoro più tedioso dell'installazione di un pacchetto extra).

Il Cache Management è composto da:

- *SyncUtilites()*, classe accessoria che contiene tutte le variabili necessarie alla sincronia.

<sup>1</sup> <https://imagemagick.org/index.php>

<sup>2</sup> <https://it.wikipedia.org/wiki/WebP>

- *Resource()*, **Entity** che fornisce un'interfaccia al Worker per interagire con l'opportuno file elaborato sul file system tramite il suo path.
- *Shredder()*, classe **Singleton** che si occupa della riduzione del numero delle immagini salvate, in modo da non consumare più spazio sul disco di quanto consentito.

È infine definita una classe interna allo **Shredder**, *ImgData*, sfruttata proprio da esso per incapsulare meglio la logica necessaria e fornire un'interfaccia d'utilizzo intuitiva.

### *SyncUtilities*

La classe funge da appoggio per le variabili di sincronia utilizzate dal Cache Management; a seconda della funzionalità richiesta, vi troviamo:

- **rwLock**, istanza di tipo *pthread\_rwlock\_t*. Questo tipo di lock garantisce una politica *fair* che impedisce la *starvation* di thread reader e writer della stessa priorità; in particolare, nuovi lettori che provano ad entrare in una regione critica del codice verranno bloccati se nella stessa vi è già presente uno scrittore di pari o superiore priorità.

Con la definizione del parametro del lock `THREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP`, inoltre, si impedisce la *starvation* del writer in generale, purché l'applicazione non abbia dei lock in lettura ricorsivi (non presenti nel nostro caso).

Viene utilizzata dalla classe *Resource* (che avrà il ruolo di "reader") e dalla classe *Shredder* (che invece sarà il "writer").

- **openMutex**, di tipo *std::mutex* (C++), utilizzato in fase di creazione del file dalla classe *Resource*; per garantire che la generazione della copia elaborata del file sia atomica e gestire l'eventualità che essa sia già presente all'interno del filesystem, è presente una sezione critica che impedisce di fatto la *open()* concorrente di più file (dato che per la logica del codice è necessario verificare il risultato della chiamata a sistema ed eventualmente effettuare un'altra in sola lettura).

Ad eccezione di questo unico punto del codice, e grazie al *rwLock* definito sopra, ogni Worker può operare su una diversa *Resource* in parallelo, di fatto garantendo lo sfruttamento massimo delle unità di elaborazione presenti.

### *Resource*

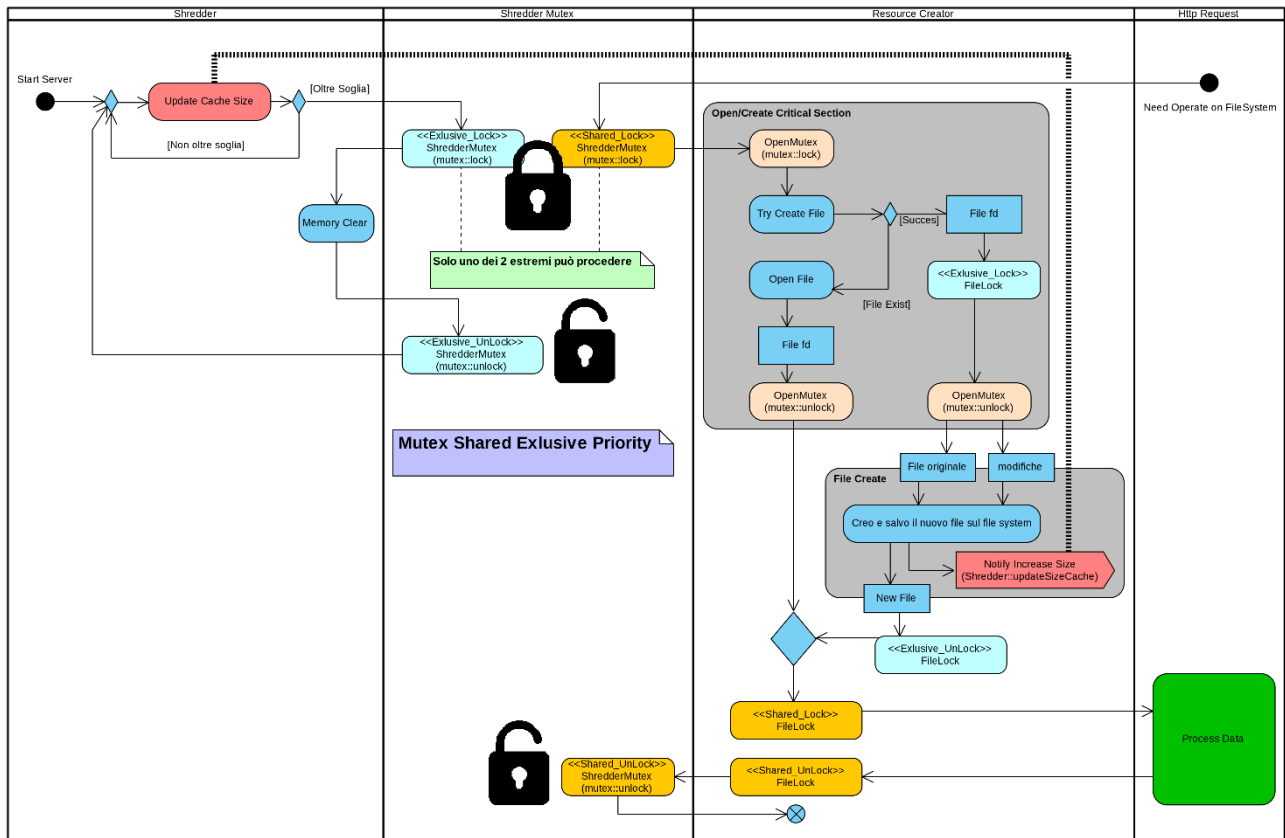
Questa classe *Entity* permette l'opportuna manipolazione di immagini modificate usando *ImageMagick* e/o *dwebp*; al suo interno non sono previsti altri metodi pubblici se non quello che faccia ottenere il path alla risorsa richiesta, incapsulando la sua creazione nel costruttore. Al termine della negoziazione del file, la *Resource* viene deallocata per ridurre l'uso della RAM nel tempo.

Nel costruttore di default (ve ne è un altro che sfrutta la *Delegation* per evitare duplicazione di codice) viene innanzitutto fatta una verifica sulle caratteristiche del file richiesto e, se sono uguali a quelle dell'originale su disco, si salva nell'attributo *path* della *Resource* proprio quest'ultimo, dato che il campo verrà acceduto all'infuori della classe solo in lettura. Successivamente viene effettuato un *readLock* sul **rwlock**, e se avviene, impedirà la partenza dello *Shredder* per tutto il tempo di esistenza della risorsa; in caso contrario, non si interagirà con il file system per tutta la durata della pulizia della cache.

Segue la sezione critica delimitata dall'**openMutex**, che coinvolge due *open()* in sequenza, con la prima che prova a fare una "create exclusive", e la seconda - di sola lettura - che può essere chiamata solo se la prima è fallita, segnalando come il file fosse già presente. Nel caso non lo fosse, prima del rilascio dell'*openMutex*, è effettuato un **flock(EXclusive)** che permette di rendere indipendente l'elaborazione del file dalla sezione critica e di aumentare sensibilmente il numero di richieste di Risorse al secondo (specie sfruttando file già elaborati in precedenza, visto che per essi il programma segue un percorso del codice differente).

Una volta elaborato, il *flock* diventa **Shared**, consentendo così ai vari costruttori in parallelo che dovevano leggere il file di terminare la creazione della *Resource*, e la dimensione definitiva dell'immagine è passata tramite una pipe allo *Shredder*, che quindi ottiene solo l'informazione necessaria al suo funzionamento.

Il flusso di esecuzione e le sue possibili ramificazioni sono descritti nel seguente **Activity Diagram**, riguardante la creazione della risorsa ed il suo ruolo in relazione alla classe Shredder, che verrà trattata immediatamente dopo.



## Shredder

Classe Singleton inizializzata in `initCes`; si presenta come un thread che, in base alla dimensione attuale della cache, prova un *writelock* sui file già esistenti e ne riduce opportunamente il numero. Lo Shredder include al proprio interno un vettore di oggetti di tipo **ImgData**, che differiscono dalle *Resource* in quanto la loro esistenza è legata all'ordinare in ordine decrescente di ultimo accesso i file nella cache, tramite la sovrascrittura dell'operatore "<" (minore) limitatamente all'oggetto stesso.

Il `thread_Shredder` rimane in attesa della dimensione di un nuovo file elaborato grazie ad una **ppoll()** sull'estremo di lettura della pipe, e se la dimensione totale della cache supera quella definita in una macro, allora cerca di ottenere il *rwlock* in scrittura per poterla liberare.

Non appena la creazione di *Resource* è quindi fermata, il thread effettua la lettura di tutto il file system a partire dal percorso `CACHE_PATH`, comprese le sottocartelle relative alle singole risorse conservate fino a quel momento; leggendo tramite `fstat()` il dato di ultimo accesso, si crea un vettore di *ImgData* che viene successivamente ordinato tramite la funzione `std::sort` (con complessità computazionale pari a  $O(n \cdot \log(n))$ ).

Poiché la dimensione della cache è salvata in memoria, il vettore viene svuotato fino al raggiungimento di metà del valore massimo ammissibile per lo spazio occupato, o se rimane un solo elemento nella cache; questo perché il singolo file potrebbe essere più grande della dimensione massima consentita (con cache molto piccole), ma proprio per questa ragione, eccessivamente lento da dover essere ricreato ogni volta.

Una volta terminata la propria attività, lo Shredder rilascia il *rwLock*, consentendo così l'ottenimento di nuove *Resource*.

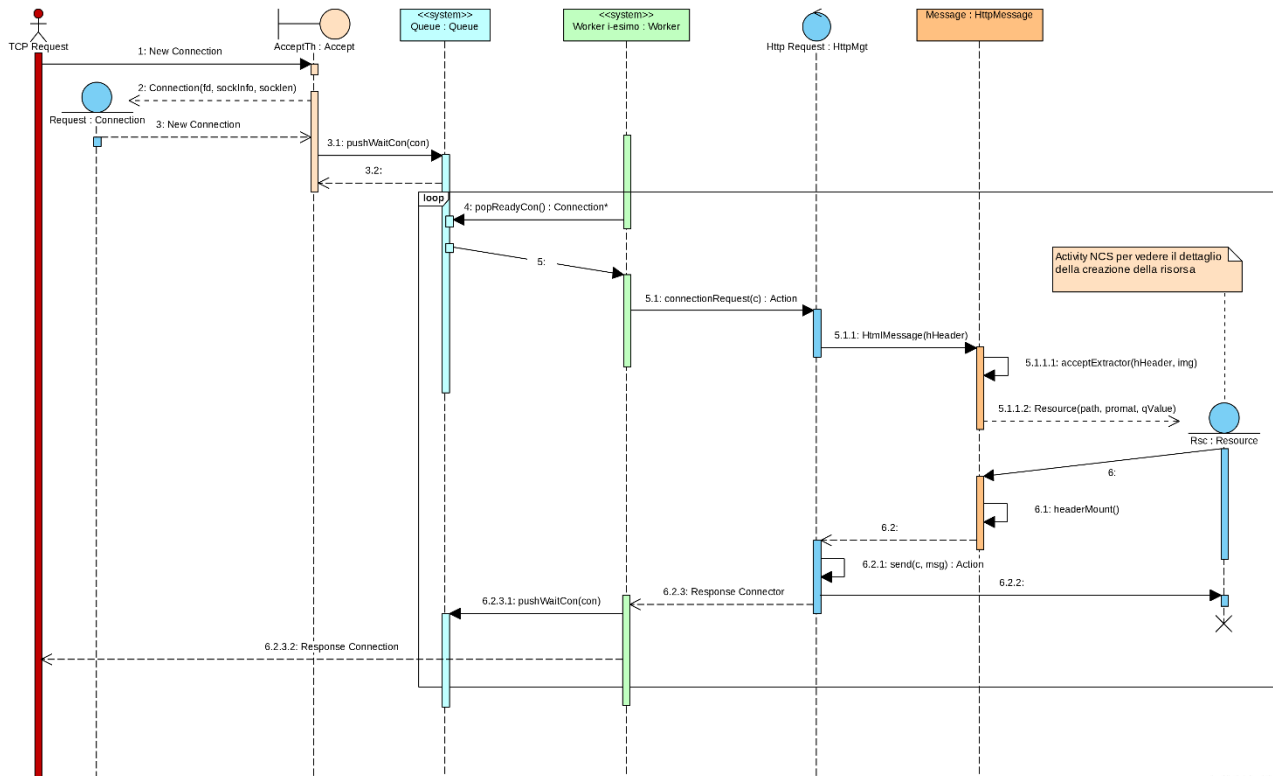
# Esempi di esecuzione

## Flusso interno

Data la natura modulare del codice, risulta comodo seguire il flusso di esecuzione utilizzando un Sequence Diagram, che mostra le interazioni che avvengono tra le classi a partire da un attore esterno che interagisce col sistema.

La sezione **loop** interessa prettamente classi di **Control** ed **Entity**, mentre la **Boundary** del sistema è l'Accept in quanto mediatrice delle singole richieste TCP.

Infine, questo tipo di diagramma permette di mostrare efficacemente soltanto delle interazioni mono-thread; a causa di questa limitazione, possiamo far sovrapporre in questo caso la sezione di esecuzione parallela del sistema con il **loop** sopra citato.



## Interazioni Esterne

Sono state effettuate catture del traffico scambiato tra un client e il server su Wireshark, sulle quali è possibile osservare come il nostro server e quello di prova su Apache 2 interagiscano all'esterno in maniera perfettamente sovrapponibile.

{Immagini di wireShark}

# Sviluppo e testing

## Strumenti di sviluppo

- Lo sviluppo del progetto è avvenuto sulla distro **KDE Neon 20.04** (Debian-based).
- L'IDE scelto è stato **CLion** di JetBrains (licenza studenti), preferito grazie alla sua forte integrazione con:
  - GDB
  - Cmake
  - Address-sanitizer
  - Etc.

Inoltre, sempre della stessa compagnia, è stato impiegato **PyCharm** per la scrittura dello script di raccolta dati, e **Matlab** per la visualizzazione degli stessi.

- Per la creazione dei diagrammi è stato usato il programma **Visual Paradigm Community Edition**<sup>3</sup>, grazie al quale si è potuto progettare in UML la struttura del Server, riducendo il tempo di scrittura del codice alla mera codifica delle classi in questo modo già definite. Tra le funzionalità del programma, inoltre, vi è la possibilità di esportare i diagrammi come PNG, con watermark non invasivi (usati poi nella relazione per la descrizione della struttura).
- **Wireshark** è stato adoperato in fase di debug per analizzare le interazioni esterne del server e scovare eventuali anomalie attraverso il traffico scambiato in rete.

## Linguaggi usati

Nel progetto sono stati utilizzati più linguaggi di programmazione, per rendere il più semplice e rapido possibile il raggiungimento degli obiettivi preposti:

### *Internamente al progetto*

- È stata scelta la revisione più recente del linguaggio **C++ (versione 17)**, in modo da poter sfruttare le librerie per l'interazione con il filesystem, che sono state (finalmente) integrate all'interno delle librerie Standard.
- Alcune funzioni avanzate richieste e non presenti nativamente nel linguaggio (gestione di stringhe e altre variabili di sistema) sono incluse nelle librerie esterne "**Boost**"<sup>4</sup>, che estendono le potenzialità di C++. Necessitano di essere installate dal gestore di pacchetti (vedere [manuale di compilazione](#)).
- **Cmake** è stato utilizzato per creare il **Makefile** richiesto per la compilazione del sistema. Per sfruttare al meglio le sue potenzialità, ogni sottosistema rappresenta una libreria *non-Shared*, e questa caratteristica è codificata nei **CMakeLists** di ognuno di essi. Il **CMakeLists** generale include i file prima definiti e si occupa dell'attivazione o meno di determinate direttive di compilazione, che controllano ad esempio la verbosità del debug o il luogo di output dello *stdout* (vedi [guida alla compilazione](#)).

Infine, come da richiesta del progetto, è stato configurato e messo in opera su risorsa statica il server Apache 2.4.46, che gestisce la stessa struttura di risorse del nostro server. La sua configurazione verrà in seguito approfondita nella sezione [Manuali](#).

### *All'esterno del progetto*

Per meglio mostrare le caratteristiche del server al variare di parametri fondamentali quali numero di Worker o connessioni parallele, si è reso necessario l'utilizzo di altri linguaggi e programmi per la raccolta e il processamento dei dati:

- Per eseguire i test sul server è stato utilizzato il tool **GoHttpBench**<sup>5</sup>. È stato preferito ad *httpPerf* in quanto permette di impostare un numero fissato di connessioni, timeout, ed altre caratteristiche utili per portare al

---

<sup>3</sup> <https://www.visual-paradigm.com/editions/community/>

<sup>4</sup> <https://www.boost.org/>

<sup>5</sup> <https://github.com/parkghost/gohttpbench>

limite un server HTTP, fornendo allo stesso tempo un output ricco di informazioni facilmente raccogliabili in maniera automatizzata.

- Per eseguire il tool di benchmark in tutte le configurazioni di interesse si è creato uno script **Python** che, sfruttando il tool appena descritto, avvia i server (nostro e successivamente Apache 2) variando il numero di worker e le connessioni parallele, ed eseguendo i test su tutte le risorse presenti. Infine, raccoglie tutti i dati in dei file *.dat* pronti per essere analizzati.
- Per l'analisi di questi dati e la creazione dei grafici si è adoperato uno script **Matlab**, che prendendo in ingresso un file contenente i path di tutti i file *.dat*, li analizza singolarmente salvando in formato *.png* i plot dei risultati ottenuti.

Grazie a questi script, è stato possibile raccogliere e selezionare una grande mole di dati in maniera totalmente automatizzata. Il loro codice sorgente è contenuto nella cartella "*4\_Bench*", nel caso si volessero adoperare per replicare i test svolti, si rimanda alla guida per l'esecuzione dei test.

## Analisi dei Test

{Grafici e commento}



## Manuali d'installazione ed uso

Tutto il materiale utilizzato in progetto è disponibile in un repository [GitHub](#).

Il [README](#) nella home del progetto contiene un rapido riepilogo delle peculiarità e punti di forza del codice prodotto, oltre ai link dei manuali scritti per l'utilizzo del sistema. Sono presenti, inoltre, gli errori di installazione incontrati e le relative soluzioni, in modo da fornire una guida per le più comuni anomalie di configurazione sulle distribuzioni derivate da Debian.

Le guide sono scritte in linguaggio *markdown* (.md), in modo da evidenziare i comandi da eseguire sul terminale permettendone una copia rapida. Se ne consiglia pertanto la visione tramite un apposito visualizzatore, o direttamente sul sito web di [GitHub](#).

Per poter eseguire lo script di Matlab è richiesta la versione *r2020b* sulla propria macchina, mentre per quelli Python l'environment necessario è già incluso nel repository di GitHub.

{STUDIARSI IL MODO PER GRAFICARE LE NUOVE COLONNE}

{vedere immagine su skype per colonne; inoltre, se sul .dat facciamo find(-1) e ci sono, scarto il file, non lo elaboro e scrivo su un file tutti i path che hanno questo problema}