

Report IncApache

Bianchi Luca <4805748@studenti.unige.it>

30 dicembre 2020

Indice

1	Introduzione	1
2	Debugging	1
3	Aggiuntivo: analisi prestazioni	3
3.1	Test su Rasberry Pi 3	3
3.2	Test su macchina locale	7
3.2.1	Test aggiuntivo	10
4	Breve conclusione	13

1 Introduzione

Tutti i sorgenti per sono stati compilati con GCC su architettura x86_64, 6/12 4.5GHz, sistema operativo Ubuntu 20.04, e su un secondo dispositivo con architettura armv7l (32 bit), 4/4 1.2GHz per eseguire test con un numero di core a disposizione ridotto. I grafici visualizzati sono creati con Apache JMeter (usato per effettuare test di carico), ed ogni punto sul grafico equivale ad una media dei tempi di risposta alle richieste effettuate in 10ms.

Per appesantire il carico sul dispositivo principale, le richieste che sarebbero state effettuate nell'arco di 10 secondi, hanno visto il proprio tempo di creazione ridotto ad un decimo.

2 Debugging

Usando Apache JMeter, ho provato a mettere sotto stress il server quanto più possibile per controllare se, con un alto carico di richieste, i thread avrebbero cominciato ad andare in conflitto e sovrascriversi aree di memoria in comune. Fortunatamente, impostando i lock adeguati ciò non è successo. Un problema possibile, ma che non sono sicuro se considerare un bug, considerando che si tratta di un fattore derivato dall'impostazione del server stesso, è che a più utenti possa essere assegnato lo stesso UID. Nel caso specifico del sito IncApache questo

non è un grande problema, ma, nel caso volessimo usare questo server per tener traccia della sessione di un utente con login ad un sito, ciò comporterebbe un grave problema di sicurezza, permettendo ad un utente di accedere all'account di un'altra persona. Ad esempio, simulando 500 richieste derivanti da utenti diversi, e replicando la richiesta mantenendo il cookie dato all'utente altre 256 volte, il tracking di tutti i cookie si aggirava intorno alle 500 volte, evidenziando il fatto che lo stesso UID possa essere condiviso da più utenti (circa 2 richieste per cookie all'interno di ogni iterazione). Il reset del tracker funziona quando viene impostato il cookie ad un nuovo utente, ma questo non invalida il cookie per l'utente precedente, creando, di fatto, 2 utenti che aggiornano il conto dello stesso tracker.

Per controllare la correttezza delle risposte date, mi sono affidato alle assertion di JMeter, andando dunque a controllare se il codice di risposta ricevuto fosse 200 e che la dimensione del documento ricevuto fosse corretta nel caso di richiesta a /, 404 in caso di richiesta ad una pagina inesistente e 501 in caso di richiesta con metodo non implementato (ad esempio GETttttt), non rilevando problemi anche sotto carichi pesanti. Non mi sono impegnato in test da miliardi di richieste per controllare se l'overflow di UserTracker e di CurUID avrebbero causato problemi, di conseguenza ho preferito prevenire inserendo un controllo che li eviti. Un problema riscontrato, e del quale non ho rilevato l'origine, è stato l'aumento drastico dei tempi di risposta verso il termine dei test, richiedendo in media circa 30 secondi per rispondere alle prime 9500 richieste, e tempi oltre il minuto per rispondere alle ultime 500. Riusciamo invece ad evitare problemi legati a drop della connessione dovuti a tempi di risposta eccessivamente lunghi da parte del server, che si notano invece nell'implementazione 1.0.

3 Aggiuntivo: analisi prestazioni

3.1 Test su Rasberry Pi 3

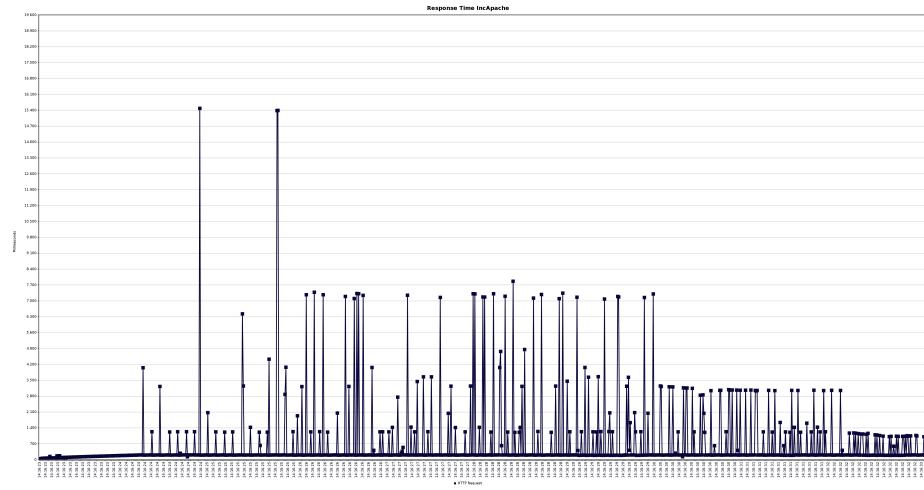


Figura 1: Response Time Graph RPi3, 1000 richieste su 10 secondi, HTTP/1.0

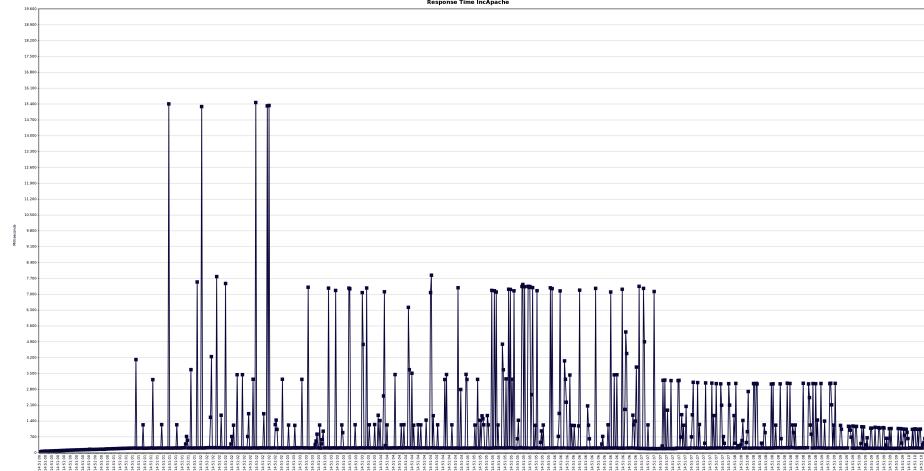


Figura 2: Response Time Graph RPi3, 1000 richieste su 10 secondi, HTTP/1.1

Sopra, le prestazioni con pipeline e senza. Per ora tutte le richieste in entrambi i casi sono andate a buon fine e non si notano grandi differenze prestazionali nei tempi di risposta, ma sarebbe strano aspettarsi il contrario da un hardware limitato.

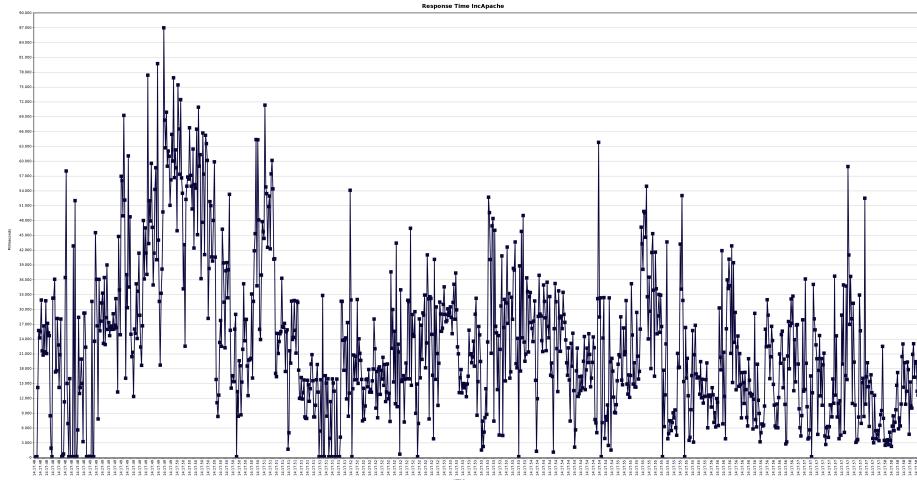


Figura 3: Response Time Graph RPi3, 5000 richieste su 10 secondi, HTTP/1.0

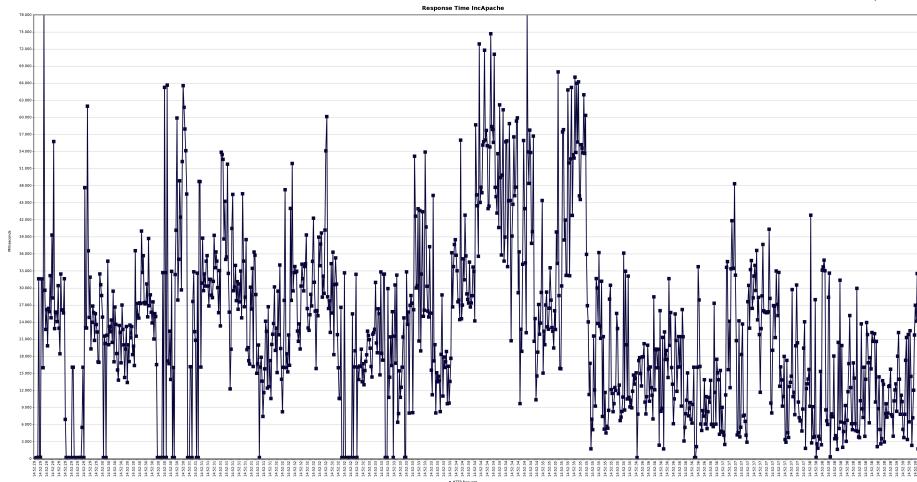


Figura 4: Response Time Graph RPi3, 5000 richieste su 10 secondi, HTTP/1.1

Le prestazioni da questi grafici sembrano ancora invariate, il tempo di risposta medio sulle richieste che hanno avuto successo sembrerebbe essere sempre molto simile tra le due implementazioni.

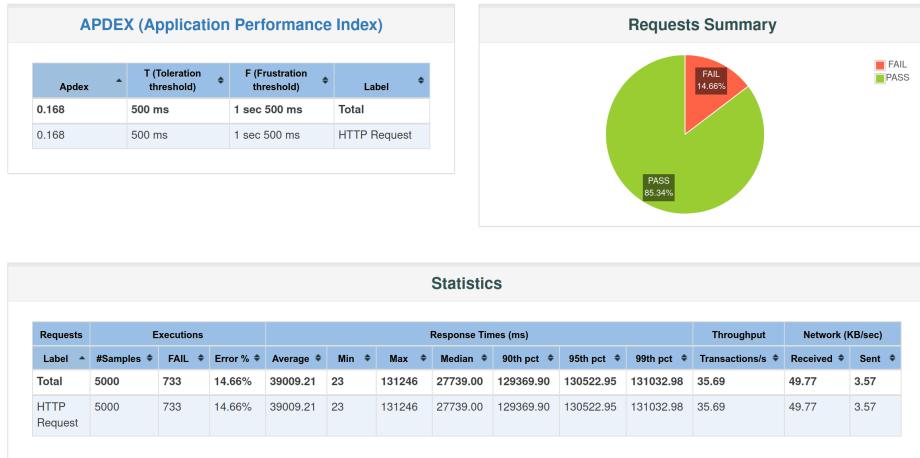


Figura 5: Pass% RPi3, 5000 richieste su 10 secondi, HTTP/1.0

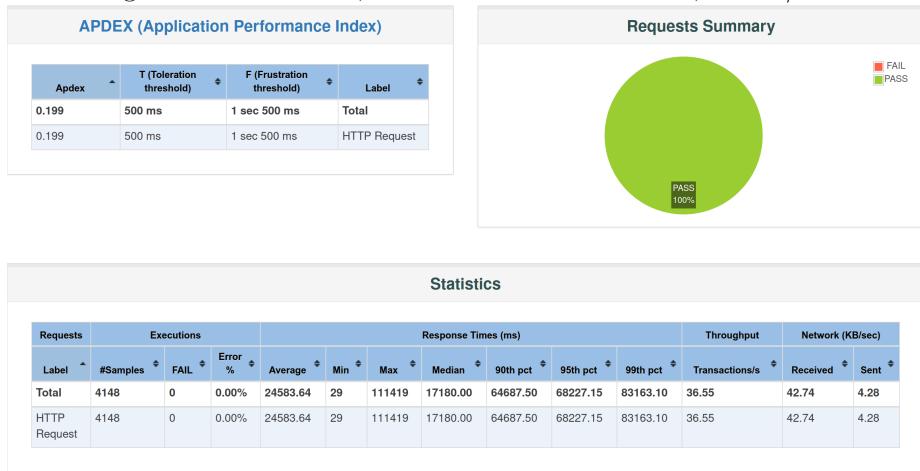


Figura 6: Pass% RPi3, 5000 richieste su 10 secondi, HTTP/1.1

Ma, andando ad osservare i dati con un occhio diverso, osserviamo come il 15% circa delle richieste all'implementazione senza pipelining sia fallita per timeout o reset della connessione, mentre nel caso dell'implementazione multi threaded tutte le richieste hanno ricevuto risposta correttamente.

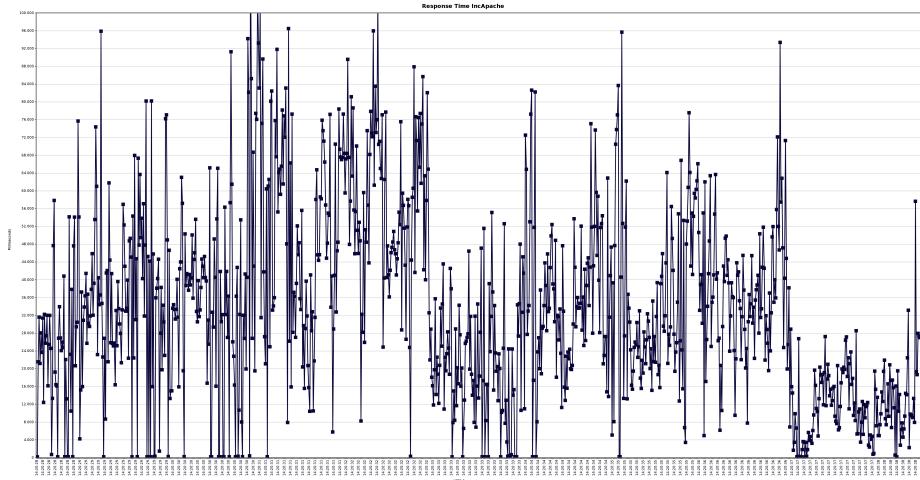


Figura 7: Response Time Graph RPi3, 10000 richieste su 10 secondi, HTTP/1.0

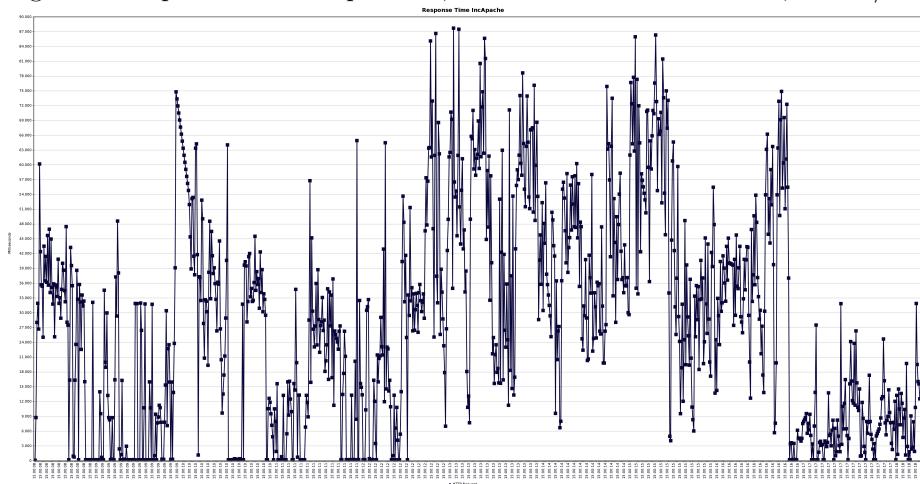


Figura 8: Response Time Graph RPi3, 10000 richieste su 10 secondi, HTTP/1.1

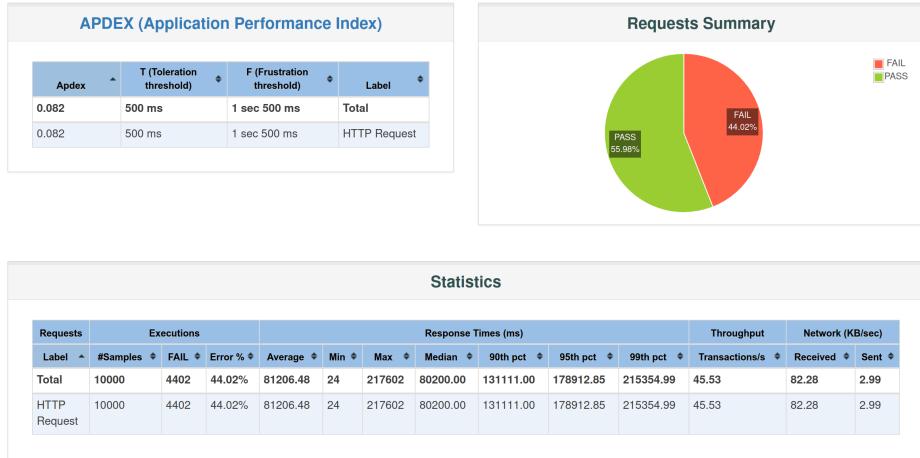


Figura 9: Pass% RPi3, 10000 richieste su 10 secondi, HTTP/1.0

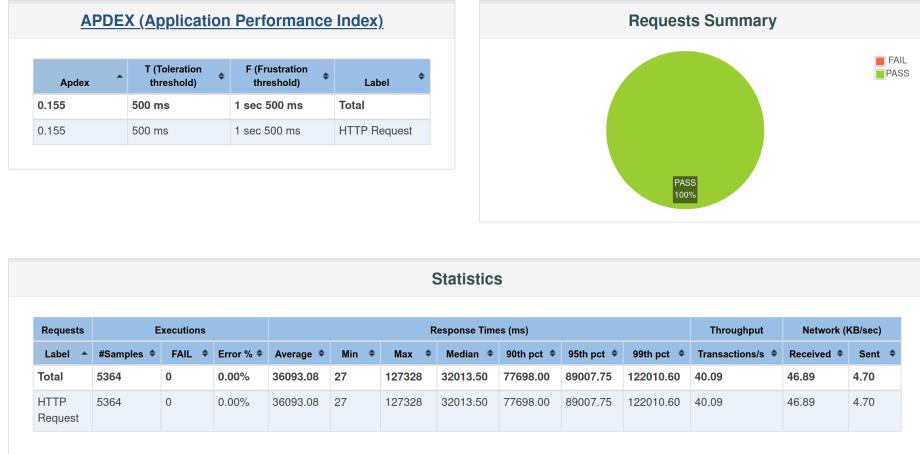


Figura 10: Pass% RPi3, 10000 richieste su 10 secondi, HTTP/1.1

Aumentando lo stress sul server, le differenze tra le due implementazioni iniziano a farsi ancora più marcate, ma è curioso notare come, sebbene il tempo di risposta medio sia più basso, anche il throughput (sia misurato come transazioni al secondo, sia misurato come KB/s), sia nettamente minore nell'implementazione HTTP/1.1

3.2 Test su macchina locale

Ho effettuato gli stessi test indicati in precedenza, riducendo il ramp-up time da 10s ad 1s, che hanno però prodotto risultati discordanti, in quanto la versione 1.0 sembrava funzionare addirittura meglio della 1.1 in alcune situazioni. Lascio

di seguito solo i report sui risultati, più completi rispetto ai grafici di response time, che in questo caso sono di poco interesse



Figura 11: Summary, 1000 richieste su 1 secondo, HTTP/1.0



Figura 12: Summary, 1000 richieste su 1 secondo, HTTP/1.1

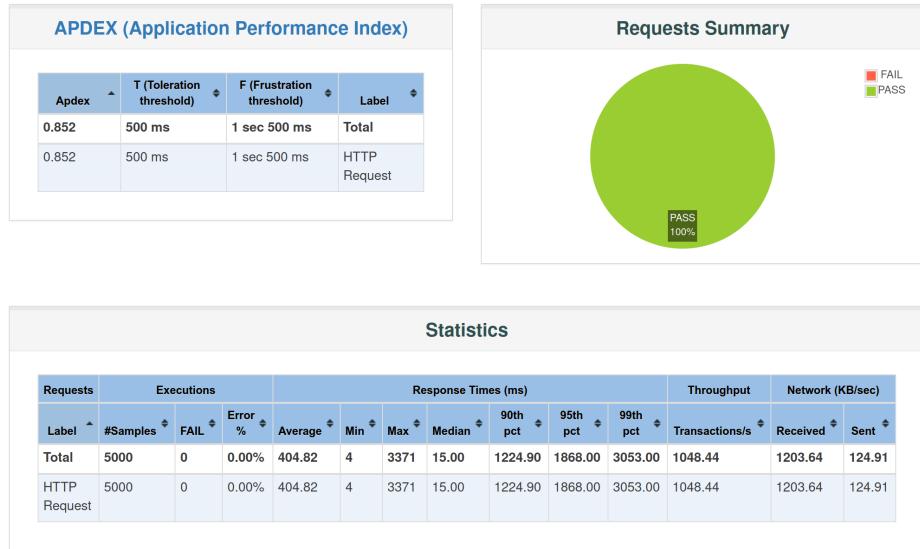


Figura 13: Summary, 5000 richieste su 1 secondo, HTTP/1.0

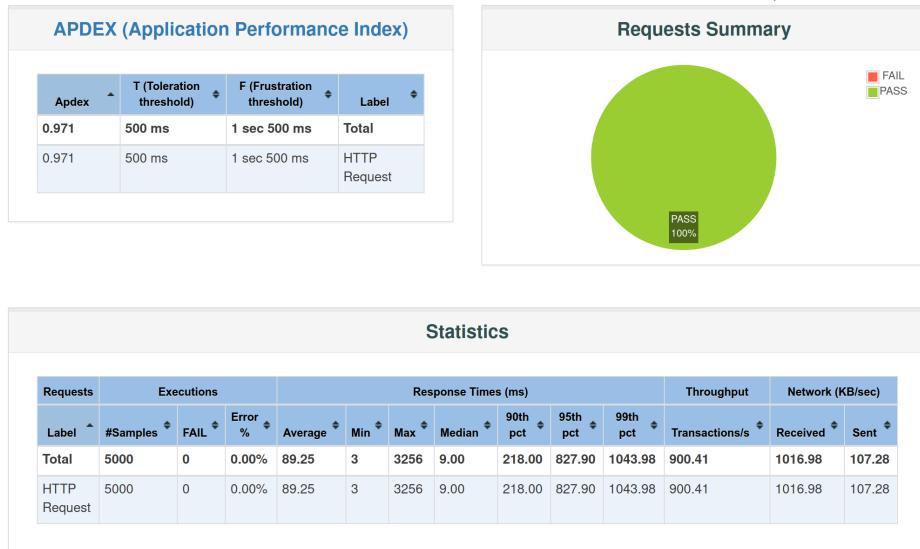


Figura 14: Summary, 5000 richieste su 1 secondo, HTTP/1.1

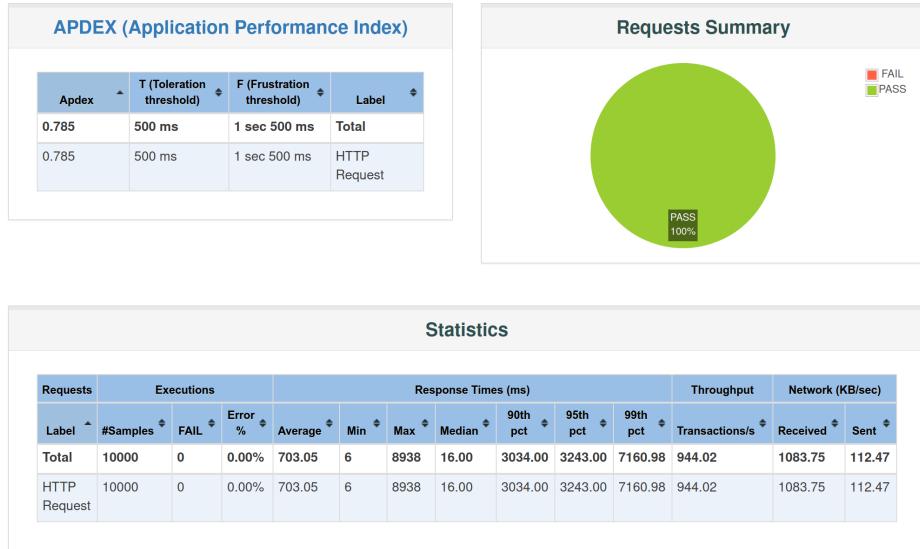


Figura 15: Summary, 10000 richieste su 1 secondo, HTTP/1.0

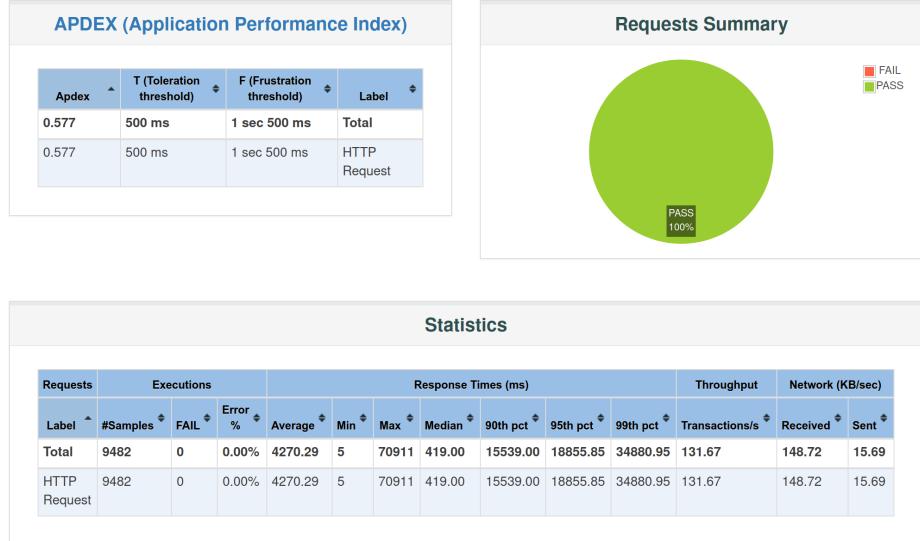


Figura 16: Summary, 10000 richieste su 1 secondo, HTTP/1.1

3.2.1 Test aggiuntivo

Avendo notato quest'anomalia nei dati, ripetuta anche in testi successivi, ho deciso di forzare leggermente la mano in fase, decidendo dunque di eseguire un nuovo ciclo di test che rispondesse a 1.000.000 di richieste nel minor tempo possibile. In teoria le richieste si sarebbero dovute generare in contemporanea,

ma essendo pressochè impossibile, sono state create nell'arco di tutta la durata del test non appena si liberavano abbastanza risorse per creare nuovi thread. Il massimo di thread che ho visto accumulare in attesa di risposta alla versione 1.0 è stato di circa 25.000, mentre per la versione 1.1 di circa 14.000, ed i dati ottenuti ci mostrano effettivamente quello che ci aspettavamo di vedere, ovvero un incremento prestazionale molto evidente nel caso di pipelining attivo. I dati sono stati campionati usando un range di 10 secondi per campione.

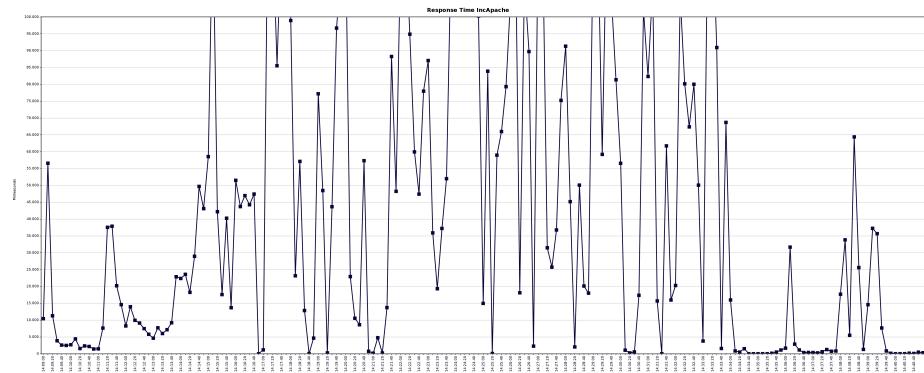


Figura 17: Response Time Graph, 1M richieste, HTTP/1.0

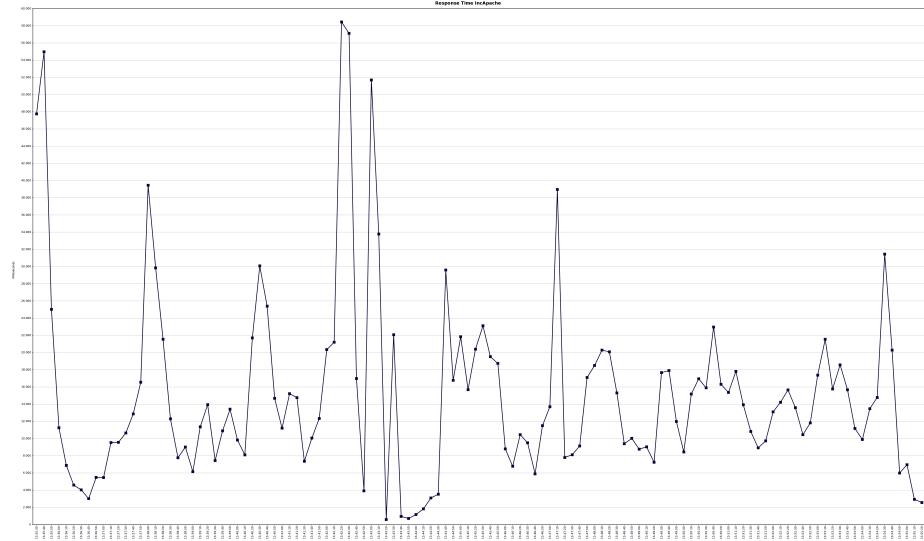


Figura 18: Response Time Graph RPi3, 1M richieste, HTTP/1.1

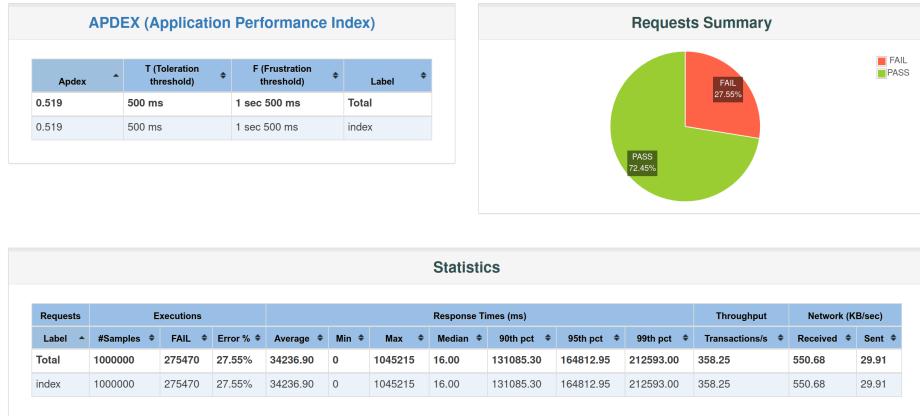


Figura 19: Pass%, 1M richieste, HTTP/1.0



Figura 20: Pass%, 1M richieste, HTTP/1.1

Possiamo notare come, all'aumentare in modo spropositato delle richieste, aumenti l'efficacia del pipelining, che riesce a ridurre il numero di errori per caduta della connessione, a ridurre il tempo medio di risposta ed a aumentare il throughput.

Tengo a precisare come questi dati non siano paragonabili con quelli precedentemente raccolti, in quanto generati in modo diverso (i dati precedenti sono stati generati effettuando richieste da un terzo dispositivo verso il server, questi sono stati raccolti generando richieste dallo stesso dispositivo del server), inoltre mi sono permesso, solo per questo test, di aumentare il numero massimo di connessioni a 12.

4 Breve conclusione

Possiamo quindi concludere, sottolineando come la differenza tra le implementazioni sia molto più evidente al crescere del carico di richieste verso il server