Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

AGGREGAZIONE DI DATI CON GESTIONE A EVENTI SU NODE JS

Tesi di Laurea in Tecnologie Web

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Stefano Gombi

Sessione 1
Anno Accademico 2012/2013

Indice generale

1 Introduzione....3
2 Il contesto attuale....15
  2.1 Twitter e lo tsunami....17
  2.2 Gli eventi....18
    2.2.1 La nascita del concetto di event loop....18
    2.2.2 Eventi e thread....18
    2.2.3 Apache2 vs NGINX....19
    2.2.4 Il problema C10K....21
    2.2.5 Utilizzo della memoria....22
  2.3 Node JS....23
    2.3.1 Node JS e Javascript....23
    2.3.2 Event loop....24
    2.3.3 Programmazione asincrona....25
    2.3.4 Vantaggi per lo sviluppatore....27
3 Il sistema di aggregazione Mangrove....29
  3.1 Centralizzazione dei dati....30
    3.1.1 Registrazione e login....30
    3.1.2 Registrazione delle fonti dati....31
    3.1.3 Raccolta e centralizzazione dei dati....31
  3.2 Accesso ai dati centralizzati....33
    3.2.1 Nodi proxy e nodi aggregatori....33
    3.2.2 Strutturazione dei nodi....34
    3.2.3 Ricerca applicata a dati centralizzati....35
  3.3 Aggregazione di timetable....36
    3.3.1 Inserimento di dati relativi a singoli negozi....36
    3.3.2 Inserimento di dati relativo a catene di negozi....37
    3.3.3 Ricerca di dati relativi a negozi....37
4 Descrizione tecnica del sistema....39
  4.1 Struttura decentralizzata....39
  4.2 Timetable e UTM....40
  4.3 Nodi aggregatori....42
    4.3.1 Gestione utenti....43
    4.3.2 Gestione subscription....43
    4.3.3 Scansione subscription....43
    4.3.4 Rilevamento richieste proxy....44

4.3.5 Gestione del database....44
4.4 Nodi proxy....44
4.4.1 Collegamento con i nodi inferiori....45
4.4.2 Richieste da interfaccia web....45
4.4.3 Inoltrare richieste ai nodi inferiori....45
4.4.4 Raccogliere le risposte....45
4.4.5 Diminuzione del timeout....46
4.5 Database....47
4.6 Interfacce grafiche....49
5 Test e valutazioni....51
5.1 Definizione dei test....51
5.1.1 Test strutturali....51
5.1.2 Stress test....52
5.2 Realizzazione dei test....52
5.2.1 Disposizione del software....52
5.2.2 Stress e prestazioni....54
5.2.3 Definizione test preliminari....54
5.2.4 Versione errata del software....55
5.3 Risultati....55
5.3.1 Test preliminari....55
5.3.2 Risultati completi....57
5.4 Valutazioni sulla qualità del codice....59
6 Conclusioni....61
6.1 Problemi non risolti....62
6.2 Possibili miglioramenti futuri....64
6.3 Event loop e futuro....65
7 Bibliografia....67
8 Ringraziamenti....70

1 Introduzione

L'obbiettivo di questa tesi è dimostrare che è attualmente possibile realizzare un sistema di aggregazione dati in grado di gestire un numero elevato di connessioni contemporanee con relativamente poco sforzo per lo sviluppatore grazie all'utilizzo di Node JS [1], piattaforma che utilizza internamente una gestione a eventi a livello di linguaggio di programmazione.

Per aggregazione si intende la raccolta di dati pubblicati da diverse fonti e la loro centralizzazione all'interno di singoli nodi, dai quale sarà possibile accedere alle informazioni memorizzate attraverso parametri di ricerca. Nonostante questa definizione sia molto generica, sono coerenti con questa interpretazione di aggregazione le azioni che vengono svolte quotidianamente dai motori di ricerca web. Essenzialmente il loro compito è quello di memorizzare alcuni dati sulle pagine disponibili e rendere navigabili questi dati attraverso una interfaccia grafica e quindi si tratta di aggregazione come intesa sopra. Anche i social network si occupano di svolgere azioni simili, con la differenza che non sono i software a raccogliere le informazioni ma gli utenti che le inseriscono direttamente (non è previsto il *crawling*[*]).

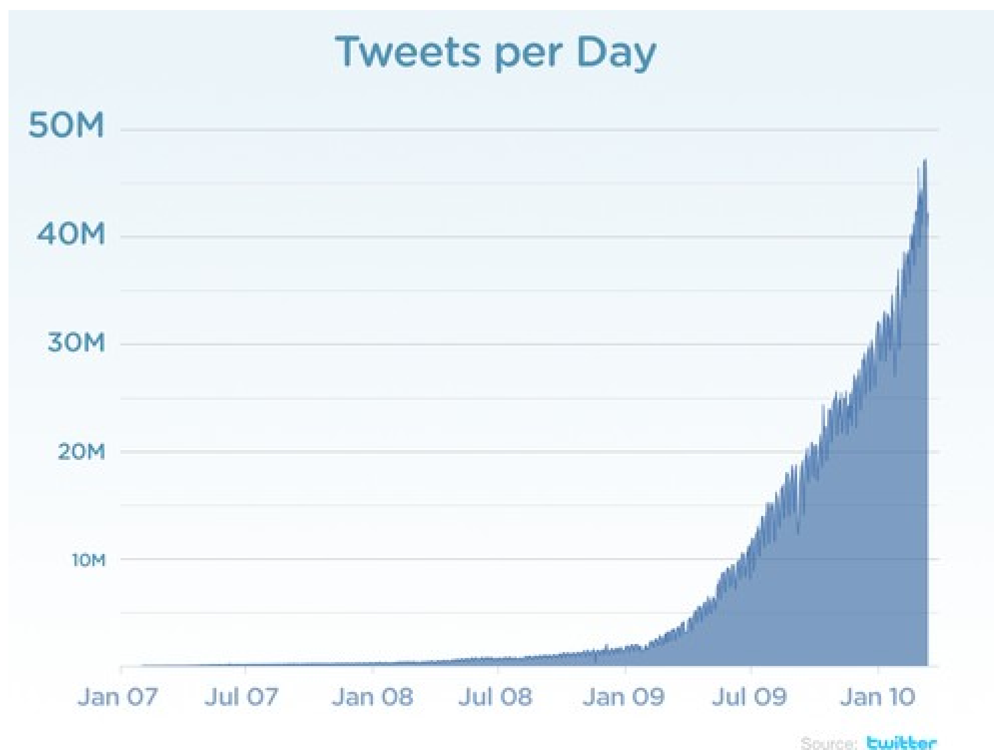
Per dimostrare questa tesi ho implementato un sistema di aggregazione dati che compie le azioni appena descritte utilizzando tecniche recentissime per la gestione di connessioni di rete. L'intento è quello di dimostrare il guadagno apportato da queste tecniche in termini di prestazioni e semplicità d'utilizzo in contesti che prevedono migliaia di connessioni al secondo.

I settori che utilizzano questo tipo di aggregazione dati sono oggi molto attivi e importanti per l'informatica. In particolare, la ricerca di continui incrementi in prestazioni e scalabilità del software che svolge queste azioni è molto attuale a causa dell'espansione vertiginosa di internet. Il discorso specifico che affronta questa tesi è la capacità degli approcci più innovativi per gestire le risorse macchina di affrontare il nuovo contesto in cui si ritrova internet, ovvero con un numero di utenti

* Navigazione e raccolta dati accessibili tramite internet.

sensibilmente più alto di qualche anno fa e destinato a crescere a ritmi elevati.

Per esempio Youtube [2] ha superato nel 2010 i due miliardi di visite al giorno [3] e nel gennaio 2013 ha superato i quattro miliardi [4]. Nel 2012, Facebook [5] ha raggiunto il miliardo di post e 2,7 miliardi di "mi piace"/commenti al giorno [6]. Per la prima volta nel Giugno 2013 è stato raggiunto il numero massimo di tweet al giorno [7] da parte della polizia di Calgary, Toronto, e il grafico sottostante mostra il numero di tweet al giorno dal 2007 al 2010.



A causa di questa crescita il numero di connessioni alle macchine che offrono servizi web è aumentato in generale e, in determinati casi, il numero è cresciuto così tanto da mostrare alcuni limiti nella gestione tradizionale delle connessioni.

Centrale per questo discorso è il concetto di thread, ovvero suddivisioni di un processo software in diverse linee di esecuzione. Il pattern oggi più diffuso per gestire le connessioni all'interno di programmi server è quello di avviare un nuovo thread per ogni client collegato, facendo sì che lo stesso codice venga eseguito contemporaneamente fra più istanze senza avviare un nuovo processo ogni volta. I thread condividono alcune aree di memoria del processo padre ed è minima l'area specifica richiesta da ognuno; utilizzando questa tecnica per gestire i client sono stati scritti molti software e fino a pochi anni fa non si sentiva la necessità di superare questo approccio. Il problema è emerso utilizzando questo metodo con quantità di connessioni contemporanee elevate (nell'ordine di migliaia o decine di migliaia al

secondo), facendo emergere i suoi limiti. In particolare in questi casi l'operazione di *context switch* necessaria per mettere in pausa un thread e avviarne un altro, nonostante sia molto economica, comincia a risultare troppo dispendiosa. I thread in ascolto di richieste attraverso la rete devono essere spesso messi in attesa per via della lentezza delle comunicazioni e questa operazione, se ripetuta molte volte, fa sentire il suo peso: per ogni nuovo thread deve essere creato lo spazio in memoria contenente lo stack e ad ogni context switch devono essere sostituiti i valori in memoria e tutto questo, ripetuto migliaia di volte al secondo, crea rallentamenti nel software.

Ovviamente il tempo necessario a effettuare un context switch varia da processore a processore, però può essere utile riportare qualche dato esemplificativo. Una ricerca [8] sui costi di questa operazione effettuata su un Intel Pentium Xeon da 2.0 Ghz ha riportato un tempo diretto di cambio thread di 3.8 μs, ma un tempo indiretto (ovvero relativo alla cache e dipendente dalle variabili del software in esecuzione sul thread) che può variare dai pochi alle centinaia di μs. Questo overhead, ripetuto per migliaia o decine di migliaia di volte al secondo, può generare un ritardo sensibile agli occhi dell'utente.

Sono usciti negli ultimi anni alcuni articoli relativi ai possibili overhead dovuti ai context switch, come [9] su cui è possibile approfondire l'argomento. Maggiori dettagli su come questo approccio generi rallentamenti sono forniti all'interno del secondo capitolo di questa dissertazione.

Si può osservare un esempio di questo fenomeno nelle giornate direttamente successive allo tsunami che ha colpito il Giappone nel 2011. Il social network fu inondato da messaggi nella settimana successiva al tragico evento e il sistema informatico su cui si sosteneva mostrò una serie di problemi, portando i ritardi nelle comunicazioni a un livello molto elevato. A causa di questi ritardi venne messo in esecuzione Blender, un software a cui gli sviluppatori per fortuna stavano già da lavorando. Blender doveva utilizzare le ultime novità in fatto di gestione connessioni e grazie ad esso furono in grado di rendere Twitter tre volte più veloce di prima [10], garantendo un tempo di risposta eccellente nonostante la quantità elevata di messaggi.

La radice di questo miglioramento sta nella gestione interna a Blender delle *connessioni basata su eventi*. È necessario riuscire a comprendere il concetto di evento e di event loop, descritto nei dettagli all'interno del secondo capitolo, per comprendere quindi come gli sviluppatori di Twitter abbiano risolto il problema delle

molte connessioni.

I programmi che utilizzano al loro interno una gestione a eventi sono detti asincroni e differiscono rispetto al classico modo di gestire le risorse della macchina nell'utilizzo che fanno dei thread. Come rifportato nelle FAQ di NGINX [11], un web server asincrono, *"Un server asincrono, dall'altra parte, è event-driven e gestisce le richieste in un singolo (o almeno, pochi) thread"* [12], destinati all'esecuzione praticamente continua di codice, mentre le operazioni di I/O sono legate a thread non bloccanti che non eseguono codice. I thread destinati all'esecuzione non rimangono in pausa in attesa di operazioni di I/O ma continuano a eseguire codice per tutte le connessioni presenti, mentre una parte del software (*event loop*) si occupa di aggiungere alla coda di esecuzione il codice corrispondente ogni volta che un thread non bloccante termina le operazioni di comunicazione. In questo modo i context switch vengono ridotti al minimo così da cercare di ridurre notevolmente il tempo di risposta da parte dei software.

Questo approccio al problema sta venendo adottato in sempre più servizi e dal suo utilizzo si possono ricavare dati molto interessati. Abbiamo un importante esempio di confronto tra due software analoghi: Apache2, storico web server che fa corrispondere a ogni client un thread e NGINX, web server più giovane ma con approccio a eventi. Secondo alcuni benchmark le performance di NGINX risultano superiori rispetto quelle di Apache2 [13] [14] [15] e il fatto che NGINX stia pian piano incrementando la propria diffusione (anche se solo al 12% del mercato contro il 57% di Apache2 [16]) è possibile proprio grazie alle novità tecniche che apporta.

Il problema di far gestire a un web server 10.000 richieste contemporanee è stato formalizzato nel problema C10K di Dan Kegel e il secondo capitolo lo tratta nel dettaglio, mostrando come sia possibile astrarre questa problematica per qualsiasi ambito in cui venga prevista una comunicazione con molti client attraverso una rete. In particolare, viene riportata una ricerca [17] che mette a confronto l'approccio sincrono e asincrono per la messaggistica istantanea e che riporta una superiorità netta dell'approccio a eventi.

L'approccio a eventi risulta molto vantaggioso anche in termini di memoria RAM utilizzata e il capitolo fa una breve comparazione sempre fra Apache2 e NGINX. Il motivo di questo vantaggio sta nel fatto che siccome i thread in esecuzione sono un numero costante la memoria utilizzata non aumenta col numero di connessioni. Nell'approccio precedente per ogni client veniva lanciato un thread e quella minuscola parte di memoria che ogni thread ha privatamente si sarebbe potuta trasformare in

un grosso numero di MB nel caso di molte connessioni.

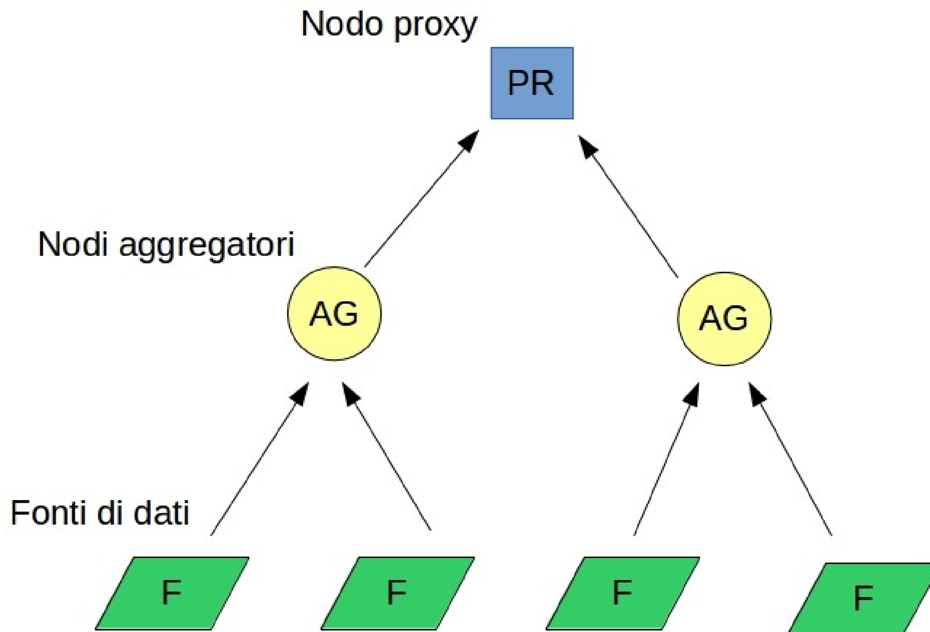
Uno degli elementi più innovativi fra tutti i software basati su eventi è Node JS, una piattaforma molto recente che porta il concetto di event loop direttamente a livello di linguaggio di programmazione. Node JS è costituito dal motore Javascript [18] del browser Chrome eseguito senza il resto del browser, così da permettere l'esecuzione di codice Javascript anche lato server. Pubblicato da Ryan Dahl nel 2009, utilizza il Javascript proprio in virtù di quelli che normalmente vengono considerati i suoi difetti, ovvero le carenze nel gestire elementi del sistema operativo.

Node JS prevede la gestione in modo automaticamente asincrono delle operazioni di I/O più lunghe (come le comunicazioni di rete e le letture dal disco fisso), permettendo allo sviluppatore di usare un event loop senza aggiungere framework o altro. Node JS utilizza un solo thread per eseguire il codice; questo porta anche un altro vantaggio fondamentale: la totale assenza di codice parallelo [19]. Come afferma lo stesso Ryan Dahl, *"In Node tutto è parallelo, tranne il codice"* [20]. Anche se è opinabile il fatto che questo sia un vantaggio, è indiscusso che permetta di scrivere codice più semplice e aiuti molto il programmatore nella stesura del programma poiché non c'è alcun bisogno di tenere in conto eventuali problemi derivanti dall'esecuzione di codice concorrente. Tuttavia mettendo assieme tutti questi elementi si può rendere più semplice agli sviluppatori non esperti di thread implementare un software performante e in grado di gestire al meglio le operazioni di I/O poiché si evita di gestire aspetti del software potenzialmente molto complicati e specifici.

Node JS è anche una lama a doppio taglio e va usato con cautela: poiché esiste un solo thread che esegue codice, calcoli troppo complessi potrebbero bloccare tutto il software per lunghi periodi di tempo. Per questo motivo è consigliabile eseguire software semplici e senza lunghe operazioni interne.

Nel caso della mia tesi ho implementato, usando Node JS, un software generico di aggregazione dati, intesa come definita all'inizio di questa dissertazione, così da dimostrarne la semplicità e le prestazioni elevate basato su internet. Per creare un sistema di aggregazione coerente con le caratteristiche di Node JS ho scelto di basarmi su un'architettura decentralizzata la quale utilizza una struttura ad albero: una simile architettura consente di creare tanti piccoli nodi comunicanti fra loro, ideali per eseguire su Node JS. Questo approccio consente inoltre di distribuire geograficamente più nodi e permette di estendere il sistema in modo relativamente indolore. Per esempio avere la possibilità di distribuire nodi sul territorio risulterebbe molto comodo nel caso si dovessero aggregare dati relativi a luoghi fisici (negozi,

aziende).



I nodi dell'albero che rappresenta il sistema di aggregazione potranno essere di tre tipi:

- **Fonti di dati**: saranno le foglie dell'albero e non sono dipendenti dalla realizzazione del sistema di aggregazione poiché cambieranno a seconda del tipo di dato da aggregare. Questi nodi sono i detentori dei dati non ancora aggregati. Gli unici nodi padri rispetto questi nodi potranno essere i nodi aggregatori.
- **Nodi aggregatori**: saranno i nodi dell'albero direttamente superiori alle foglie e avranno il compito di comunicare con le fonti di dati. Questi nodi sono i detentori dei dati aggregati e accettano richieste dai nodi superiori dell'albero. Gli unici nodi padri rispetto questi nodi potranno essere i proxy.
- **Nodi proxy**: tutti i nodi che non siano foglie o padri di foglie sono nodi proxy. Il compito di questi nodi è quello di accettare richieste tramite la loro interfaccia grafica o un nodo padre e di inoltrarla ai nodi sottostanti. Gli unici nodi padri rispetto questi nodi potranno essere altri proxy.

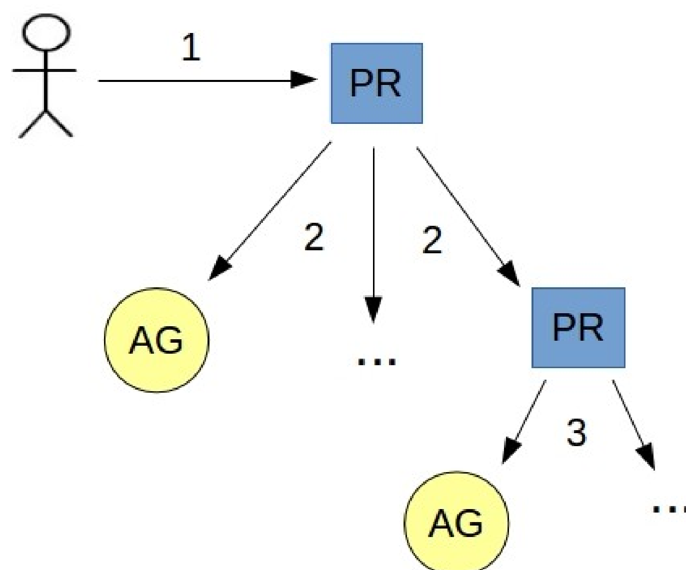
Procedendo a un'analisi più a basso livello dei due tipi di nodi da me implementati risultano evidenti i vari tipi di comunicazioni che essi dovranno intrattenere per dar vita alla rete di aggregazione. I nodi aggregatori svolgono molti compiti, spiegati nel dettaglio all'interno del quarto capitolo, come ad esempio occuparsi di gestire la registrazione di utenti tramite attivazione per email, così da poter legare qualsiasi

azione eseguita sull'aggregatore a un responsabile. Un aggregatore dovrà inoltre occuparsi di gestire il database locale permettendo la ricerca e la modifica di dati, in modo da salvarli e permetterne la consultazione. Un'altra funzionalità presente è quella di gestire le richieste da parte del proxy a cui si è collegato: siccome le richieste per determinati dati saranno inviabili solo da un proxy, gli aggregatori dovranno solo stare in ascolto per eventuali richieste e rispondere a seconda dei parametri specificati nel corpo della richiesta. L'aggregatore dovrà anche occuparsi di attingere alla fonte dei dati inserite ogni qual volta un dato superi la propria data di scadenza, così da avere dati sempre aggiornati.

Per poter accedere a tutte queste funzionalità tramite interfaccia grafica l'aggregatore offre una API pubblica tramite un web server integrato. Le varie *route* disponibili sono poche ma sufficienti a permettere il login di utenti, la loro registrazione, l'inserimento di sottoscrizioni e la richiesta di scansioni di queste ultime.

I nodi proxy sono invece molto più semplici rispetto agli aggregatori a causa dei loro minor compiti. Ad ogni proxy potranno collegarsi sia aggregatori che altri proxy in modo da dar vita a alberi con più livelli di profondità; durante la dissertazione farò perciò riferimento a dei generici "nodi inferiori" intendendo proxy e/o aggregatori con lo stesso proxy padre. Un proxy è in grado di accettare connessioni da nodi inferiori: poiché in questo caso l'albero si costruisce dalle foglie verso la radice saranno i nodi inferiori ad avviare per primi la connessione con i nodi padri. I proxy in questo caso dovranno dare inizio a un'autenticazione tramite sfida per verificare che al nodo inferiore sia permesso il collegamento.

Anche i proxy offrono una web API pubblica per permettere a interfacce grafiche di specificare ricerche da eseguire sui dati aggregati. Una volta specificata una richiesta con i dovuti parametri essa verrà inoltrata a tutti i nodi inferiori collegati al proxy così da farla arrivare fino ai veri detentori dei dati, i nodi aggregatori. In questo passaggio sarà necessario compiere il minor numero possibile di modifiche alla richiesta così da rendere praticamente "agnostici" i proxy verso il tipo di dato aggregato. Infine il proxy gestisce le varie risposte che seguiranno alle richieste, raccogliendole e inviandole una volta ricevute da tutti i nodi inferiori. Nel caso alcune risposte arrivassero dopo un determinato intervallo di tempo configurabile, verranno inviati i dati parziali e ignorate le risposte successive.



Tutti i dettagli sulle funzionalità specifiche dei due moduli sono contenute all'interno del quarto capitolo.

Per dimostrare il funzionamento del software da me implementato su un caso reale mi sono appoggiato a una precedente tesi scritta da Vincenzo Ferrari [21] relativa alla definizione di file XML, le **timetable**. Le timetable hanno lo scopo di descrivere dati e orari relativi a negozi (o, più in generale, luoghi pubblici) e all'interno vi possono essere descritti più luoghi fisici. Per dimostrare effettivamente il funzionamento dell'aggregatore ho deciso di utilizzare questo lavoro collegandolo al mio progetto, perciò le timetable sono da considerare il dato fondamentale di tutto il sistema di aggregazione. Le timetable possono essere prodotte da UTM, un software realizzato da Vincenzo Ferrari e a cui ho apportato solo piccole modifiche per potervi interagire con il sistema di aggregazione. Un esempio di timetable è riportato all'inizio del quarto capitolo.

Le sottoscrizioni invece sono lo strumento che utilizzo per permettere ai moduli aggregatori di registrare una timetable; sono, in altre parole, l'anello che lega le fonti di dati ai nodi aggregatori.

La scelta del database per l'implementazione del progetto è ricaduta su MongoDB [22], il quale funziona internamente su oggetti JSON i quali sono effettivamente oggetti Javascript e quindi consente la loro elaborazione senza traformazioni. La scelta specifica di MongoDB è stata poi fatta in funzione della sua capacità di esprimere query dinamiche simili a quelle dell'SQL, l'ideale per ottenere in tempo reale risposte alle diverse richieste esprimibili tramite interfaccia web. Le query map-reduce, lo strumento più famoso dei sistemi NoSQL [23], sono troppo lente per

restituire risultati in tempo reale ma torneranno comode in futuro per generare statistiche sui dati memorizzati. Sono però emersi anche alcuni problemi derivanti da questa scelta, come la necessaria divisione delle timetable e la necessità di far eseguire a Node JS delle funzioni per raffinare ulteriormente i risultati di una ricerca su Mongo.

Per entrambi i software ho implementato due interfacce grafiche web molto basilari per poter testare il loro funzionamento. Le interfacce sono state scritte grazie al framework Javascript EXTJS [24] e sono molto basilari; in particolare, l'interfaccia del proxy non deve considerarsi completa e utilizzabile in progetti reali, quanto piuttosto uno strumento per verificare l'effettivo funzionamento del sistema di aggregazione.

Una volta implementato i due moduli per l'aggregazione ho pensato a una serie di test per poterne valutare la bontà, suddividendo i possibili aspetti testabili in due categorie: la verifica della comunicazione fra i nodi e le prestazioni del sistema complessivo, ovvero sulla struttura.

Su quest'ultima categoria di test è possibile valutare il funzionamento di cinque elementi fondamentali che determinano il buon funzionamento del software, ovvero:

1. la capacità di proxy e aggregatori di comunicare anche se su macchine differenti;
2. la capacità di un aggregatore di gestire più fonti dati;
3. la capacità di un proxy di gestire più aggregatori;
4. la capacità di un proxy di accettare nodi padre;
5. la capacità di un proxy di comunicare con proxy e aggregatori contemporaneamente.

Per gli stress test invece una indagine accurata necessiterebbe di confrontare software analoghi scritti in modo sincrono. Questa tesi non prevede la realizzazione di aggregatori scritti in altri linguaggi, quindi gli aspetti testati sono la capacità del sistema di resistere a carichi di lavoro onerosi e le sue prestazioni a seconda dei diversi carichi di lavoro.

Per i test di struttura ho disposto una rete di software ospitati sulle macchine dei laboratori di informatica dell'università di Bologna, la quale comprende in sé tutti gli elementi elencati durante la definizione dei test strutturali. La disposizione viene mostrata nel quinto capitolo di questa dissertazione, assieme ad altri dati relativi a questi test e ad alcuni criteri per stressare il software e valutare i suoi tempi di

risposta in contesti critici. Per gli stress test ho definito alcune prove con centinaia e migliaia di richieste contemporanee per osservare il comportamento del software.

Tutti i test hanno dimostrato un funzionamento adeguato del software, dimostrando di riuscire a rispondere anche a migliaia di richieste contemporanee con tempi accettabili (mediamente 1 secondo e 150 millesimi per 1700 richieste). Anzi, il tempo medio necessario per rispondere mediamente a una singola richiesta secondo i test si abbassa con il loro aumentare, passando dai 1.9 ms necessari per 50 richieste fino ai 0.66 ms necessari per 1750 richieste. Tutti i grafici riassuntivi sono inseriti all'interno del capitolo “Test e valutazioni”.

Per scopi dimostrativi ho testato anche una vecchia versione del codice che utilizzava comandi sincroni per scrivere su file, la quale è risultata circa cinquanta volte più lenta del codice corretto. Questo esempio serve a dimostrare la forza dell'approccio asincrono ma, allo stesso tempo, i possibili regressi delle prestazioni in caso di semplici errori con la gestione delle operazioni sincrone. Tutti i risultati precisi sono riportati verso la fine del quinto capitolo.

Per quanto riguarda la qualità del software, un indice della sua bontà sta nei relativamente brevi tempi di transizione da un software generale a un software orientato alle timetable, durata circa un mesetto. Il software non è completamente “neutro” dal dato che deve aggregare poiché si sarebbe verificato un trade-off tra la semplicità del software e la sua neutralità dal dato originale; nonostante questo, in caso di cambiamenti il codice da modificare è molto poco, dimostrando quindi una buona riusabilità del codice.

La semplicità del software da me realizzato e le sue provate capacità di gestire migliaia di connessioni al secondo mi portano a concludere che riesca a dimostrare la tesi iniziale, ovvero che grazie a Node JS sia possibile oggi scrivere software scalabile senza eccessivo sforzo per lo sviluppatore.

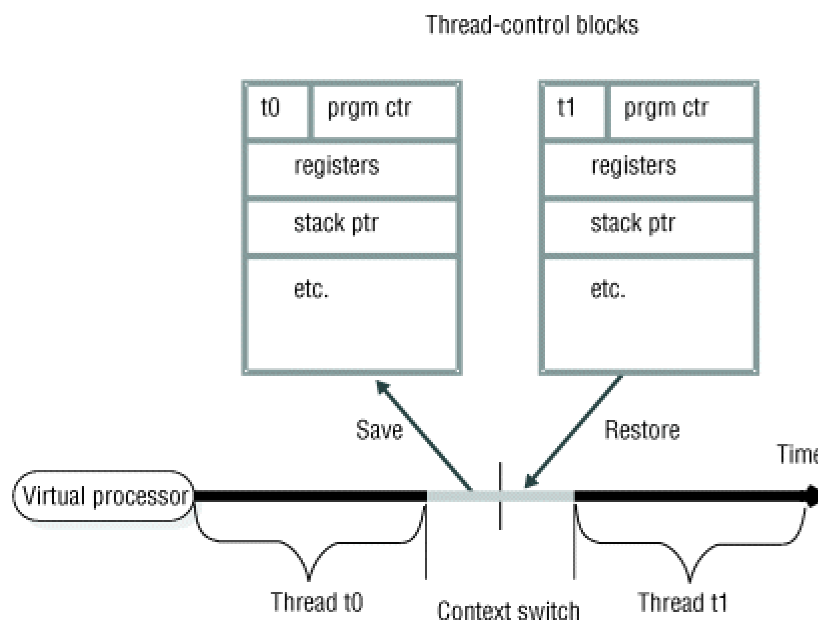
In questo lavoro è possibile identificare alcuni punti deboli di Node JS (mancanza di pattern, linguaggio in via di definizione) ma si tratta quasi sempre di mancanze passeggere, dovute alla breve vita di questa piattaforma; alcuni di questi limiti hanno già trovato una soluzione alla versione attuale (0.10), altri sono in via di risoluzione ed è mia opinione che entro breve tempo sarà possibile scrivere codice esente da questi limiti. Ci sono ovviamente anche problemi in questo progetto non imputabili a Node JS, ma che spesso derivano dalla mia scelta di utilizzare tecnologie stabili o al particolare caso delle timetable. In particolare, sono presenti alcuni punti di “attrito” fra le caratteristiche delle timetable e MongoDB che dovranno essere risolti, in futuro,

con un cambio di formato o di database. Ciò nonostante, la mia opinione è che il software dimostri la tesi prefissata, rendendo possibile a un non esperto di thread la creazione di un programma in grado di gestire un numero molto elevato di connessioni contemporanee.

La tesi conclude offrendo alcuni spunti per futuri miglioramenti del software, come un sistema di caching su ogni proxy o l'utilizzo dell'HTML chunked.

2 Il contesto attuale

A causa della crescita del numero di connessioni e di utenti internet degli ultimi anni [25] alcuni software che hanno il web come canale di accesso principale stanno affrontando un nuovo problema, ovvero quello di riuscire a sostenere migliaia o decine di migliaia di connessioni contemporanee. I motori di ricerca, i social network e in generale le applicazioni web di successo devono poter gestire un numero di accessi ogni giorno più alto e recentemente gli approcci tradizionali per la gestione delle connessioni stanno mostrando alcuni limiti nell'affrontare questa sfida. In particolare, questo problema si è manifestato nella gestione dei thread.



Senza scendere nel dettaglio poiché il concetto fa parte del patrimonio dell'informatica da molto tempo, un thread è una suddivisione di un processo in più linee di esecuzione, condividendo alcune aree di memoria relative al software e permettendo l'esecuzione in modo parallelo delle diverse esecuzioni. L'utilizzo dei thread è stato fondamentale per lo sviluppo del web poiché nell'architettura client-server il pattern più utilizzato per il web è quello di lanciare un thread del programma server per ogni client collegato, così da minimizzare lo spreco di risorse e

avere in esecuzione più copie dello stesso software.

Durante le comunicazioni di rete necessarie per il procedimento dell'esecuzione, operazioni molto lunghe se comprate con il ritmo di lavoro di una CPU, un thread viene messo in attesa da parte del sistema operativo e un altro viene messo in esecuzione. In un software con molte comunicazioni di rete questa operazione è frequente e in questo possiamo trovare la radice del problema.

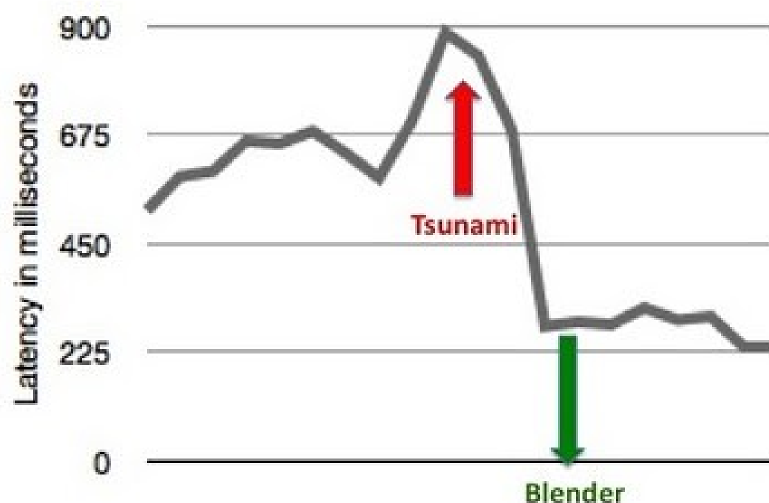
L'avvicendamento dei thread in esecuzione richiede una commutazione di contesto, più famosa nella sua traduzione inglese context switch, ovvero il salvataggio dei dati del thread messo in pausa e il caricamento dei dati del nuovo in esecuzione. Questa operazione è velocissima e praticamente istantanea, quindi per lungo tempo questo metodo di gestire più connessioni di rete è stato adeguato e non si è presentata la necessità di cambiarlo. Con la crescita del numero di dispositivi connessi alla rete sono cominciati a vedersi i primi limiti di questo approccio: questa operazione, se pur molto economica in termini di risorse, non è gratis. Nel caso di un numero elevato di connessioni le risorse sprecate per fare cambio di contesto diventano considerevoli, rallentando il funzionamento del software nel suo complesso e occupando molta memoria.

È difficile fornire dati precisi riguardo al costo di un context switch in quanto questo valore varia molto a seconda della CPU utilizzata e dalle variabili del software in esecuzione. Una ricerca [8] che cerca di far luce sui fattori in gioco durante questa operazione stima che su un Intel Pentium Xeon a 2GHz il costo di un context switch sia 3.8µs. Questo costo è insignificante e potrebbe passare inosservato se non fosse che si tratta solo dei costi "diretti", ovvero dell'effettivo cambio di riferimenti all'interno della CPU. Ci sono dei costi definibili "indiretti" legati ai sistemi di cache della CPU che variano a seconda della memoria utilizzata dai thread e che possono portare il costo complessivo del singolo context switch sull'ordine delle centinaia di microsecondi. Questo costo, pagato migliaia o decine di migliaia di volte a causa delle molte connessioni, può diventare considerevole. Alcuni sviluppatori esperti nel settore lo ritengono eccessivo per contesti in cui è necessario gestire tutte queste connessioni nel minor tempo possibile.

Inoltre, associare un thread a ogni connessione è rischioso e poco economico anche dal punto di vista della memoria se si tiene conto del numero di connessioni che alcuni di questi software si trovano oggi a dover affrontare. Un esempio calzante di questo problema si può trovare nelle ultime vicende riguardanti Twitter.

2.1 Twitter e lo tsunami

Twitter è uno dei social network più famosi al mondo. Durante lo tsunami che colpì il Giappone nel 2011 si registrò un picco di *"tweet"* sul sito, tanto da mettere in crisi il sistema e da rendere i tempi di risposta dei server molto elevati. Il sistema di gestione dei front-end di Twitter (il sistema intero è molto complicato e la sua comprensione non è necessaria ai fini di questa tesi) era costituito da alcune istanze di Ruby on Rails [26] ottimizzate per ottenere massime prestazioni, ma evidentemente di fronte ad un così elevato di messaggi e visualizzazioni il sistema tradizionale di avvicendamento thread offerto dal web server non era più sufficiente. Per fortuna i



programmatori interni all'azienda stavano già lavorando da tempo su Blender, un nuovo software in grado di superare questi limiti. Blender venne mandato in produzione ad una settimana dallo tsunami e rese Twitter tre volte più veloce di quanto non fosse prima [10].

Nonostante in Blender fossero inclusi tanti miglioramenti, la radice di questo aumento di prestazioni è da trovarsi nell'uso di Netty [27], un framework per la gestione delle connessioni di rete per software scritti in Java.

Citando direttamente il blog degli ingegneri di Twitter:

> *"Sappiamo da molto tempo che il modello sincrono di processare le richieste usa le nostre CPU in modo inefficiente. [..] Blender risolve questo problema [..] creando un sistema di aggregazione completamente asincrono. No thread waits on network I/O to complete."*

I concetti di evento e di codice asincrono stanno alla base della nuova frontiera di programmi destinati a poter sopportare migliaia di connessioni contemporaneamente offrendo sempre il miglior servizio possibile, come in questo caso abbiamo visto con

Twitter. Questi concetti sono anche alla base di questa tesi, la quale cerca di dimostrare praticamente come sia possibile oggi riuscire a costruire un programma che utilizzi fino ai livelli più profondi un approccio basato su eventi. Andiamo dunque ad approfondire questi termini e a capire come essi possono essere utili agli sviluppatori in questo contesto.

2.2 Gli eventi

2.2.1 La nascita del concetto di event loop

Tutti i problemi descritti precedentemente hanno portato gli sviluppatori a cercare una soluzione al di fuori dei normali modelli di programmazione usati negli ultimi anni. Così è nato il concetto di event loop [28]: un programma che usa un event loop ha al suo interno un meccanismo di gestione di determinati eventi asincroni rilevati dal programma, i quali scatenano determinate conseguenze quando si verificano in un legame azione-reazione. Questo metodo innovativo di scrivere programmi che hanno a che fare con comunicazioni di rete in elevata quantità risponde a due precise necessità che si sentivano nel mondo della programmazione: la prima, e più immediata, è quella di aiutare la strutturazione di programmi che hanno a che fare con molti eventi asincroni.

I linguaggi di programmazione tradizionali non sono stati pensati per un contesto così "asincrono" come quello del web, delle applicazioni online, delle comunicazioni massive di rete e così via. Sono stati modellati secondo un'ottica anteriore, ovvero più orientata a del software che esegue operazioni complesse ma con (relativamente) poche comunicazioni di rete.

Utilizzare un sistema a eventi aiuta lo sviluppatore a strutturare il proprio codice per questo tipo di problematica, astraendo tutta una serie di aspetti tecnici relativi al rilevamento di un evento e alle conseguenze che da esso sono determinate. Questo è comunque un vantaggio minore se confrontato con l'altro grande aiuto portato da programmi basati su eventi, ovvero la migliore gestione delle risorse.

2.2.2 Eventi e thread

Spesso i programmi basati su eventi vengono contrapposti ai programmi che utilizzano thread ma è necessario fare un po' di chiarezza per non confondersi. Gli eventi non sono metodi diversi di gestire il processore rispetto ai thread e non è possibile comparare queste due gestioni, anzi i programmi basati su eventi fanno un utilizzo intensivo di thread. Quello che però i programmi basati su eventi rendono

disponibile è una astrazione sull'utilizzo diretto dei thread così da fornire al programmatore un utilizzo delle risorse già ottimizzato. In altre parole, allo sviluppatore che utilizza motori/piattaforme basate su eventi viene nascosto il complesso sistema di thread dietro a questi software e ci si può concentrare maggiormente sui problemi specifici del proprio software, delegando il problema dell'I/O all'event loop.

Chiunque abbia avuto a che fare in questi ultimi anni con la programmazione di software destinato alla gestione di più comunicazioni contemporanee ha per forza avuto a che fare con il classico pattern che consiste nel creare un thread all'avvio di una nuova comunicazione e la sua esecuzione in armonia con le altre richieste. Oggi può risultare necessario avere programmi che gestiscano migliaia di richieste al secondo e, in questo caso, solo programmatori veramente esperti dell'utilizzo dei thread dovrebbero lavorare su queste problematiche, non certo programmatori esperti di web o database.

È nato così il bisogno di astrarre le i meccanismi relativi ai thread e di offrire una base ottimizzata per la programmazione orientata alla comunicazione massiva, il quale ha dato origine a prodotti come Twisted [29] per Python, EventMachine [30] per Ruby, Reactor [31] per Java e in generale tutta una serie di librerie/framework che utilizzano il Reactor pattern [32] volte a fornire una gestione di eventi a linguaggi che ne sono sprovvisti.

Node JS è una piattaforma che addirittura basa tutta la sua ragion d'essere su questo modo di gestire le comunicazioni; ne parlerò più approfonditamente più avanti in questo capitolo. Prima di continuare vorrei rimarcare la novità introdotta nel mondo della programmazione dalla nascita di questa tecnica parlando del caso di Apache2 e NGINX.

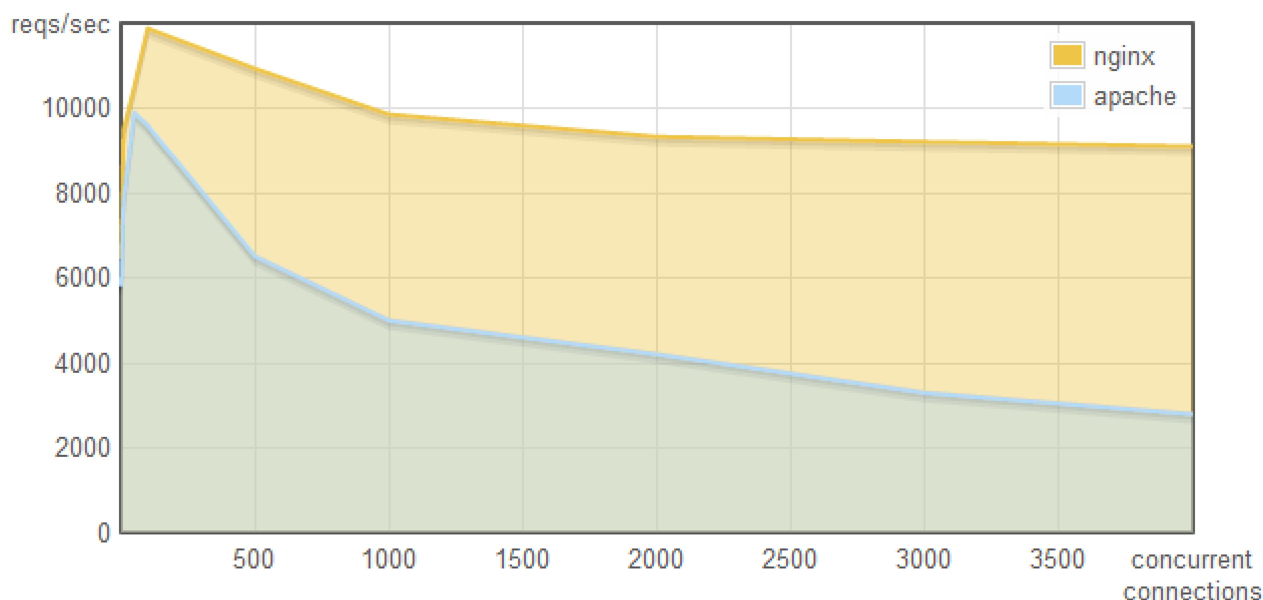
2.2.3 Apache2 vs NGINX

Chiunque abbia lavorato nel mondo della programmazione web conosce quale importante ruolo abbia giocato il web server Apache2 [33] in questi anni. Le sue caratteristiche lo hanno reso molto celebre all'interno del settore, tanto da essere incluso all'interno della famosa sigla LAMP (Linux, Apache2, MySQL e PHP) che indica il set di tecnologie dominanti per la realizzazione di siti web negli ultimi anni. La completezza di funzionalità e le performance offerte da Apache2 lo rendono un ottimo web server e sicuramente il suo successo è meritato, ma negli ultimi tempi si erano fatte più insistenti le critiche riguardo al metodo con cui esso gestiva i processi

e sui suoi limiti in contesti che prevedevano un numero molto elevato di connessioni. Apache2 utilizzava l'approccio classico di gestione richieste, ovvero faceva corrispondere a ogni client un thread (anche se la sua gestione interna è molto più complicata.

Sulla base di questa necessità è nato NGINX [11], un web server che utilizza internamente un event loop e che sta rapidamente prendendo il posto di Apache2 nei contesti in cui si presentano i limiti di quest'ultimo. Citando direttamente le FAQ di NGINX [12]:

> *Nginx e Lighttpd sono probabilmente i due server asincroni più conosciuti e Apache è indubbiamente è il miglior server process-based. [...] il vantaggio principale dell'approccio asincrono è la* ***scalabilità****. In un server process-based, ogni connessione simultanea richiede un thread che implica un overhead significativo. Un server asincrono, dall'altra parte, è event-driven e gestisce le richieste in un singolo (o almeno, pochi) thread.*
>
> *[...]*
>
> *Mentre un server process-based può avere di media delle prestazioni alla pari con un server asincrono sotto carichi leggeri, con un carico più pesante di solito consuma più RAM con un significativo degrado delle performance.*



È significativo notare come NGINX si stia lentamente diffondendo proprio sulla base della tecnica dell'event loop. Apache2 è stato ideato per un contesto che non è più quello attuale, con un numero di connessioni di rete inferiore e per questo risulta problematico in alcuni scenari. NGINX si presenta come un'evoluzione che risolve esattamente questo problema. Partendo da questo esempio, è facile generalizzare

questo concetto per capire quanto possa essere utile in un qualsiasi software fortemente orientato alla comunicazione di rete.

Questo è il perno fondamentale attorno a cui gira la mia tesi, ovvero l'utilità della gestione a eventi per programmi con molte connessioni.

2.2.4 Il problema C10K

Il problema di gestire un certo numero di connessioni contemporanee al secondo è stato esplicitato nel campo dei web server da Dan Kegel con il problema C10K [34], ovvero il problema da parte di un web server di gestire 10.000 richieste al secondo. Questo problema è strettamente collegato con la gestione delle risorse da parte del sistema operativo ed è attorno a esso che sono nati web server come NGINX e altri. Ad esempio, Lighttpd [35] è un web server meno diffuso di NGINX ma servì come prototipo per risolvere il problema del C10K grazie a un sistema a eventi.

Il problema del C10K è esattamente quello che la gestione a eventi mira a risolvere, cioè riuscire a gestire un numero molto alto di connessioni contemporaneamente. Il problema può essere facilmente astratto e trasportato ad altri campi con le stesse problematiche, come per esempio l'aggregazione di dati e la messaggistica istantanea.

Una ricerca [36] che tratta questi elementi riporta un confronto fra software con tecnologia non bloccante (asincrona) e bloccante (sincrona) per una rete di comunicazione XMPP [37]. In particolare, confronta software realizzato con Java, Scala [38] e Node JS (di cui parlò più approfonditamente in seguito) il quale usa un approccio basato sugli eventi. È importante sottolineare che, se confrontati, il codice in esecuzione su Node JS risulta nettamente inferiore come velocità di elaborazione rispetto agli altri due linguaggi. Ciò nonostante, i risultati sono molto interessanti e l'approccio a eventi risulta nettamente superiore in termini di messaggi consegnati e di precisione. Riporto solo alcuni dati dalla ricerca, di cui consiglio comunque una consultazione per approfondire il discorso.

Su una simulazione di 10.000 utenti i quali inviano un messaggio a testa (quindi lo stesso contesto del C10K), questa tabella mostra i messaggi consegnati al secondo su varie configurazioni di test:

Software	Minimo	Medio	Massimo
Java	109	270	322
Scala	214	249	267
Node JS	1360	1485	1608

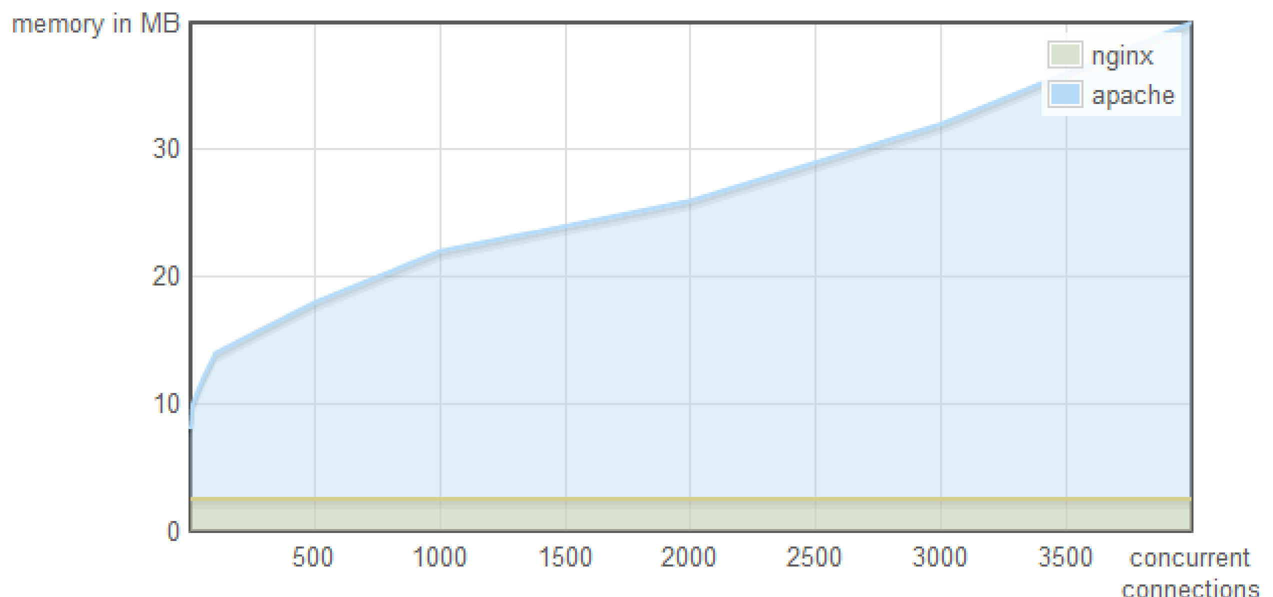
Un altro che prevede l'ingresso e l'uscita quasi immediata di 5.000 utenti invece ha prodotto questo risultato su uscite/entrate al secondo:

Software	Minimo	Medio	Massimo
Java	14	76	141
Scala	29	82	108
Node JS	518	621	745

I dati sono interessanti e mostrano, almeno nel contesto di questi test, una netta superiorità dell'approccio a eventi non bloccante.

2.2.5 Utilizzo della memoria

Un altro vantaggio importante dei sistemi basati su eventi rispetto all'approccio classico dei thread è l'utilizzo della memoria RAM. L'uso dei thread a sua volta è un miglioramento, sotto questo punto di vista, rispetto al precedente approccio orientato ai processi poiché i thread condividono più memoria rispetto a due processi dello stesso software. Comunque, esattamente come avviene per la CPU nel momento del context switch, l'uso dei thread non è gratuito e necessariamente "spreca" una certa porzione di memoria RAM.



Questo approccio considerato in un programma che associa un thread a ogni connessione può risultare pericoloso in un contesto con migliaia di connessioni. I software basati su eventi invece utilizzano un numero costante di thread quindi sotto

questo punto di vista l'utilizzo di memoria risulta quasi costante e prescinde dal numero di connessioni attive. Questo grafico si basa sempre sul confronto fra Apache2 e NGINIX rende bene l'idea di quali vantaggi si possano trarre da questo approccio.

La discussione sui vantaggi e i pericoli derivanti dalla gestione a eventi delle comunicazioni di rete in realtà non è conclusa e alcune ricerche [39] sostengono che con una adeguata impostazione dei thread sia possibile raggiungere le stesse prestazioni utilizzando un sistema più classico. Nonostante personalmente ritengo che ricerche di questo tipo spesso sottovalutino aspetti come la semplicità d'utilizzo o il consumo di memoria di software basati su eventi, oggettivamente i software orientati alla gestione contemporanea di molte connessioni stanno lentamente muovendo verso un approccio ad eventi. Questo modo di risolvere il problema sta diventando dominante anche se il fenomeno risulta poco evidente poiché è oggettivamente ancora un problema non molto diffuso. Questa tesi vuole essere da parte mia un contributo a questa discussione per dimostrare non solo l'efficacia di questo approccio, ma anche la sua semplicità di utilizzo grazie a software innovativi come Node JS.

2.3 Node JS

2.3.1 Node JS e Javascript

Avendo in mente il concetto di event loop spiegato nel capitolo precedente è possibile comprendere le caratteristiche innovative introdotte da Node JS nel mondo della programmazione lato server. Node JS è una piattaforma costruita su V8 [40], il motore Javascript utilizzato nei browser chrome e chromium. Pubblicata da Ryan Dahl nel 2009, si tratta di un'"estrazione" del motore del browser chrome leggermente modificato per poterlo utilizzare come interprete, così da eseguire del codice Javascript che utilizzi una gestione a eventi delle operazioni di I/O.

Se l'importanza del Javascript come linguaggio lato client è molto nota risulta invece più difficile capire che ruolo può giocare questo linguaggio in un contesto per lui nuovo e in cui sono presenti alternative molto più adottate e longeve. Cercando di evitare "guerre di religione" si può affermare che il Javascript nasce come un linguaggio per animare pagine web e che quindi presenta caratteristiche adeguate a questo compito. Mancano perciò tutta una serie di funzionalità presenti di norma in altri linguaggi a causa dell'ambiente specifico in cui può essere eseguito il linguaggio: nessun supporto ai thread, nessuna comunicazione con il sistema operativo, assenza di una strutturazione in classi solida, assenza di un sistema per la gestione di

moduli/librerie e tanto altro.

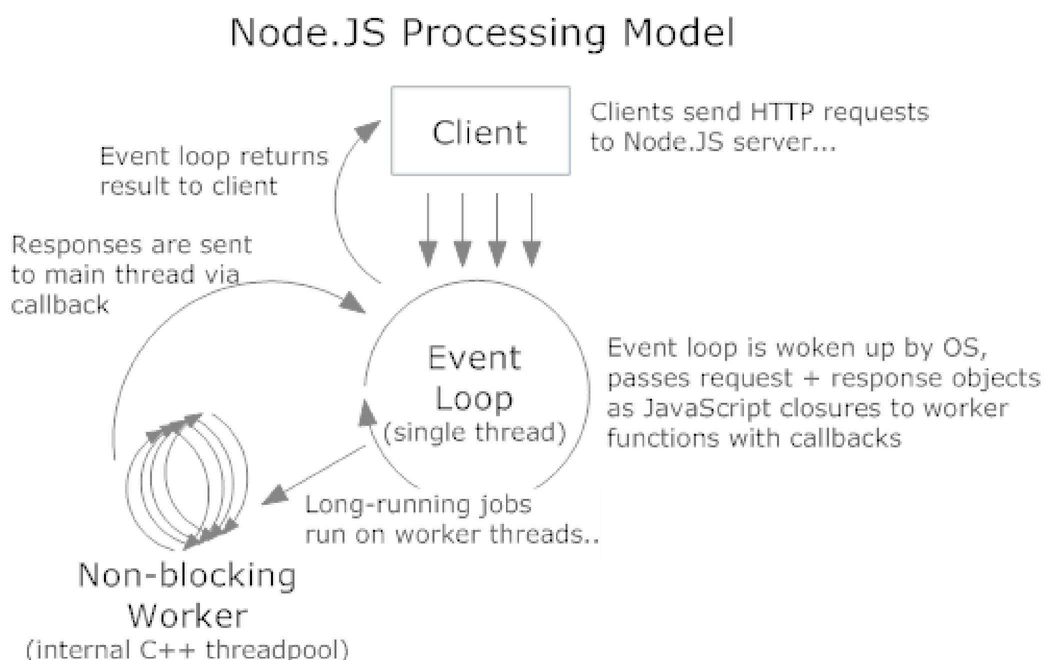
Queste caratteristiche hanno però reso il Javascript particolarmente appetibile per essere utilizzato in un sistema completamente basato su eventi, ed è per questo motivo che nasce Node JS. Di fronte alle esigenze di migliorare le performance dei software di rete Ryan Dahl ha creato una piattaforma in che esegue le operazioni di I/O particolarmente lente (comunicazioni di rete o accesso al disco) in modo asincrono, rendendo la programmazione su Node JS diversa da qualsiasi esperienza con altri linguaggi. Il Javascript è risultato ideale allo scopo esattamente per i limiti che presenta: non ha una gestione dei thread nativa quindi è relativamente più semplice programmare con Javascript in un ambiente in cui lo sviluppatore non deve avere accesso a questo livello del sistema operativo.

2.3.2 Event loop

Il funzionamento interno di V8 è abbastanza complesso e spiegarlo nei minimi dettagli esula dallo scopo di questo scritto. Mi limiterò a dare un'infarinatura generale per poter permettere al lettore di comprendere come Node JS implementa il concetto di event loop a livello base permettendo di scrivere programmi completamente asincroni.

Nel momento in cui una operazione di I/O considerata lenta (di solito lo è se riguarda la rete o il disco fisso) viene eseguita da un programma in Node JS, V8 si occupa di trasferire la chiamata su un thread non bloccante fra quelli che ha a disposizione nella sua *thread-pool* base. In questo modo, il thread principale con il codice può continuare la sua esecuzione senza context switch. Nel momento in cui una operazione collegata ai thread non bloccanti è terminata il *kernel* segnala che questo thread può tornare in coda di esecuzione. A questo punto però V8 si occuperà di intercettare il messaggio, mettere nella propria coda di esecuzione la funzione di callback specificata con l'operazione di I/O terminata e di rimettere il thread non bloccante a disposizione per altre operazioni di I/O. Così facendo, virtualmente il thread che esegue codice non si ferma mai, avvicendando le funzioni di callback delle varie operazioni terminate.

Questa immagine può aiutare nella comprensione del funzionamento di http, uno degli esempi più classici del funzionamento di Node JS:



Bisogna però sempre tenere a mente, quando si progetta di scrivere un programma con Node JS, che questa architettura ha un effetto collaterale molto pesante: le operazioni che occupano per lungo tempo il thread in cui viene eseguito il codice (operazioni di calcolo onerose) bloccano l'interno software. Per questo motivo Node JS è assolutamente sconsigliato in caso di operazioni di calcolo complesse e nella fase di progettazione di programmi che usano questa piattaforma è sempre necessario utilizzare questa caratteristica come criterio fondamentale per la scrittura del codice.

2.3.3 Programmazione asincrona

La prima caratteristica che rende particolare la programmazione con Node JS è il fatto che utilizza un paradigma asincrono. Per spiegare questo concetto penso sia più efficace un esempio diretto. Prendiamo per esempio il Python (ma si potrebbe prendere qualsiasi linguaggio sincrono) ed esaminiamo una porzione di codice in cui viene fatta una query e stampato il risultato a schermo:

```
a = db.getOne("query per prendere un elemento dal database")
print a
print "fine del programma"
```

Quando viene eseguito questo codice l'interprete si occupa di far eseguire al database la query e aspetta una risposta. Probabilmente il sistema operativo, accorgendosi della necessità di aspettare una risposta da parte di questo programma passerà il controllo a un altro thread/processo. Quando il database avrà risposto il SO si occuperà di

rilanciare al momento opportuno il thread il quale stamperà il risultato della query e infine il messaggio “fine del programma”. Proviamo a guardare lo stesso codice scritto con Node JS in modo asincrono:

```
db.getOne("query per prendere un elemento dal database", function(result) {
        console.log(result);
});
console.log("fine del programma");
```

Come si può vedere l'azione in cui viene utilizzato il risultato dal database non è quella sequenzialmente successiva alla query, ma viene anzi passata come parametro della query stessa. Questo stile di programmazione permette all'event loop di Node JS di capire che codice eseguire nel momento in cui l'operazione di I/O con il database termina. Così facendo, il codice verrà messo in coda di esecuzione solamente quando il risultato sarà arrivato, mentre nel frattempo il programma potrà continuare a eseguire le operazioni successive. Rispetto all'esempio precedente, questo frammento di codice stamperà prima “fine del programma” istantaneamente e, solo successivamente, il risultato della query.

Questo stile di programmazione era già presente in Javascript e si sposa perfettamente con le necessità di un sistema basato su eventi. Per quanto riguarda gli effetti che questa scelta comporta per lo sviluppatore software, bisogna ammettere che lavorare con il paradigma asincrono dopo molte esperienze sincrone risulta difficile e non immediato. Allo stesso tempo, il codice prodotto può essere molto meno leggibile rispetto a del codice sincrono (l'esempio precedente lo dimostra) a causa dei molti livelli di indentazione e definizione di funzioni. Ciò nonostante, questo modo di concepire la programmazione è il più corretto per un ambito di utilizzo in cui più software comunicano attraverso internet e aiuta il programmatore a focalizzarsi sui ritardi dovuti alla rete e alla distanza; inoltre, esso apporta anche notevoli vantaggi.

2.3.4 Vantaggi per lo sviluppatore

Il primo vantaggio è che diventa non necessario lavorare con i thread. I thread sono stati una tappa fondamentale dello sviluppo informatico ma, come spiegato precedentemente, oggi dovrebbero essere usati solo da esperti, ovvero da programmatori a basso livello in grado di comprenderne a pieno il potenziale. Far lavorare con dei thread programmatori di linguaggi ad alto livello spesso può rivelarsi più dannoso che altro. In Node JS esiste un solo thread che esegue il codice, e tutte

le operazioni di gestione dell'I/O sono invisibili allo sviluppatore [19]. In altre parole, questa piattaforma astrae il concetto di thread all'utente finale offrendo come base già un'ottima gestione, nascondendo tutta una serie di dettagli al programmatore e rendendo la scrittura di software più semplice dal punto di vista concettuale e conoscitivo. L'altro grande vantaggio (che discende dal precedente) sta nell'impossibilità di eseguire codice parallelo.

In Node JS, "tutto è parallelo tranne il codice". Questa frase di Ryan Dahl, ormai onnipresente negli articoli che spiegano il funzionamento di questa piattaforma, può sembrare criptica ma in realtà rappresenta un concetto molto semplice: poichè esiste un solo thread che esegue codice, all'interno di un singolo programma che esegue su Node JS non è possibile avere due frammenti di codice che eseguono parallelamente. Questo particolare semplifica notevolmente il processo di scrittura del software, esonerando il programmatore dal dover tener conto di problemi di sincronizzazione.

È quindi impossibile per un singolo programma scritto in Node JS eseguire su più core di una CPU contemporaneamente, almeno utilizzando i costrutti base (ci sono già librerie che consentono queste operazioni). Bisogna però tenere a mente che di solito i programmi che necessitano di elaborazione parallela multi-core hanno il compito di eseguire calcoli onerosi; Node JS non è stato pensato per questo compito, ed è consigliabile scegliere in questi casi linguaggi più a basso livello e più performanti rispetto al Javascript.

La mia tesi è che utilizzando questa piattaforma innovativa sia possibile scrivere programmi capaci di sopportare molte connessioni con relativamente poco sforzo per lo sviluppatore.

3 Il sistema di aggregazione Mangrove

Per dimostrare la tesi sostenuta da questa dissertazione ho ideato e realizzato un sistema di aggregazione dati, disponibili tramite internet, eseguibile su Node JS quindi basato su una gestione a eventi. Il nome di questo sistema di software è Mangrove, tradotto in italiano mangrovia, il cui tipo di crescita delle radici ricorda la disposizione dei nodi nel sistema.

Per aggregazione di dati si intende la raccolta automatica di dati direttamente dalle fonti che li pubblicano, ovvero tramite crawling. Una volta raccolti, questi dati dovranno essere centralizzati in singoli punti e accessibili attraverso una interfaccia grafica che permetta di specificare i parametri di ricerca.

Questo tipo di aggregazione è affine con le operazioni svolte dai motori di ricerca e, con qualche modifica, a quelle svolte dai social network, entrambe categorie di software che spesso si trovano a dover gestire migliaia di connessioni al secondo. Con questo software vorrei dimostrare quanto possa essere semplice realizzare una piattaforma di questo tipo capace di gestire molte connessioni grazie a Node JS.

L'operazione di centralizzazione dei dati e quella di ricerca sono asincrone e logicamente scollegate, perciò concettualmente è possibile trattarle in modo separato.

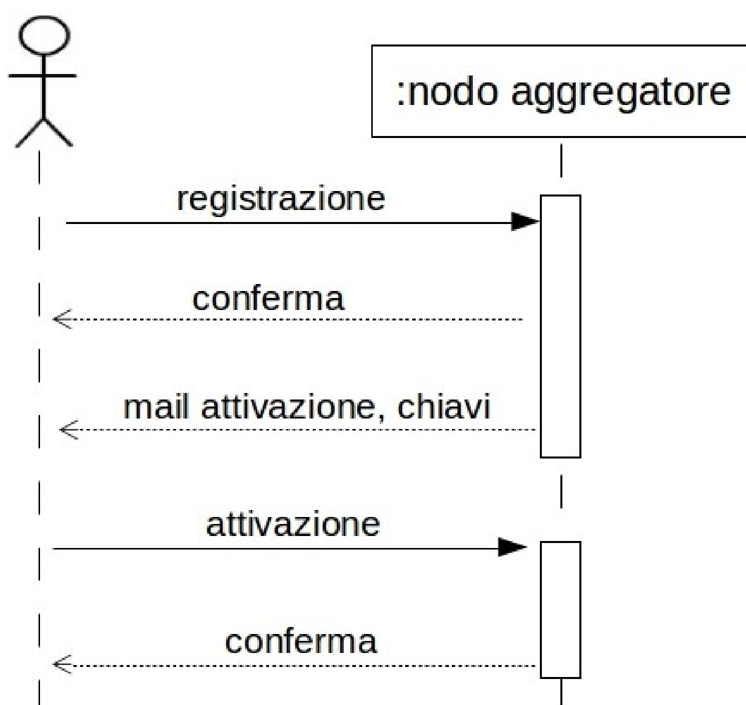
Durante le seguenti spiegazioni utilizzerò il termine "nodo aggregante" per descrivere un singolo elemento del sistema di aggregazione, il quale viene inteso in questo caso come una singola istanza del software finale, in grado di compiere tutte le operazioni elementari svolte dal sistema nel suo complesso. Con il procedere della dissertazione diverranno più chiare le caratteristiche di questo nodo e le differenze fra i possibili nodi.

3.1 Centralizzazione dei dati

3.1.1 Registrazione e login

L'operazione di centralizzazione necessita di associare ogni raccolta di dati a un determinato utente per evitare problemi di sicurezza come inserimento di dati falsi o modifica dei file esistenti non lecita. In questo contesto per utente si intende un generico detentore dei dati, per esempio il proprietario di un negozio o di un'azienda.

Per consentire questa associazione ogni nodo aggregante dovrà permettere la registrazione e l'autenticazione utente. Particolare enfasi va posta sulla fase di registrazione poiché è da questo evento che vengono generati dei dati fondamentali per permettere la centralizzazione dei dati.

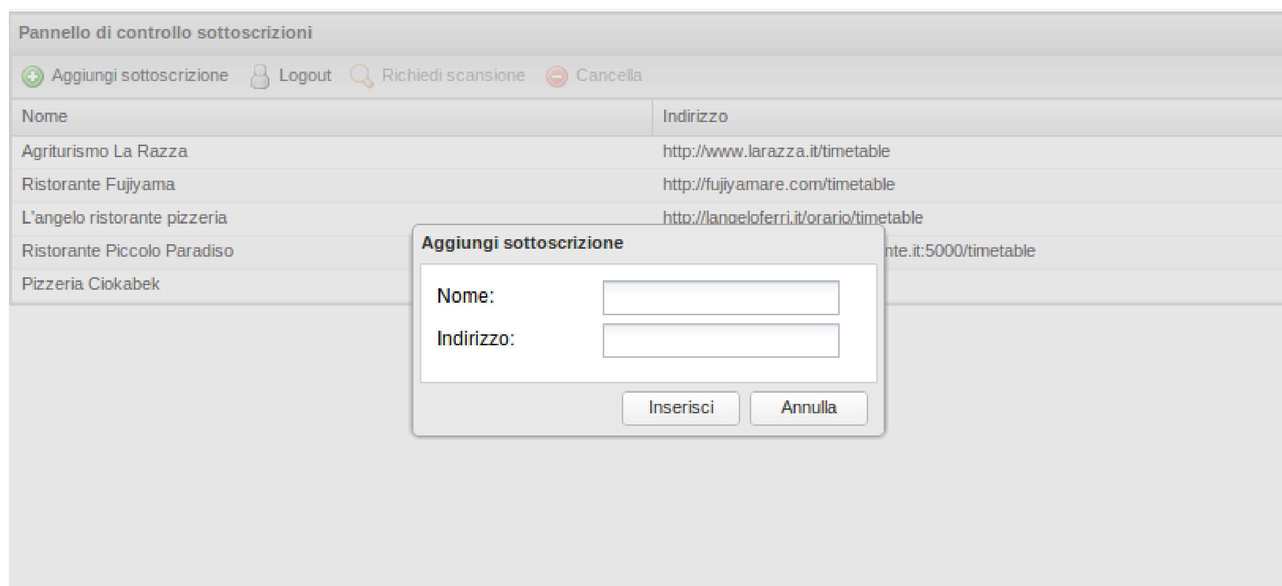


Siccome l'indirizzo email dell'utente è un dato richiesto dalla procedura di registrazione, una volta terminata la procedura il nodo sarà in grado di inviare a questo indirizzo un link di attivazione account, senza il quale non sarà possibile autenticarsi. In questa stessa mail dovranno essere presenti due chiavi generate automaticamente dal nodo, il cui utilizzo verrà spiegato successivamente.

Una volta terminata la registrazione e visitato il link di attivazione, l'utente potrà autenticarsi tramite interfaccia grafica e accedere alle funzionalità del nodo di aggregazione.

3.1.2 Registrazione delle fonti dati

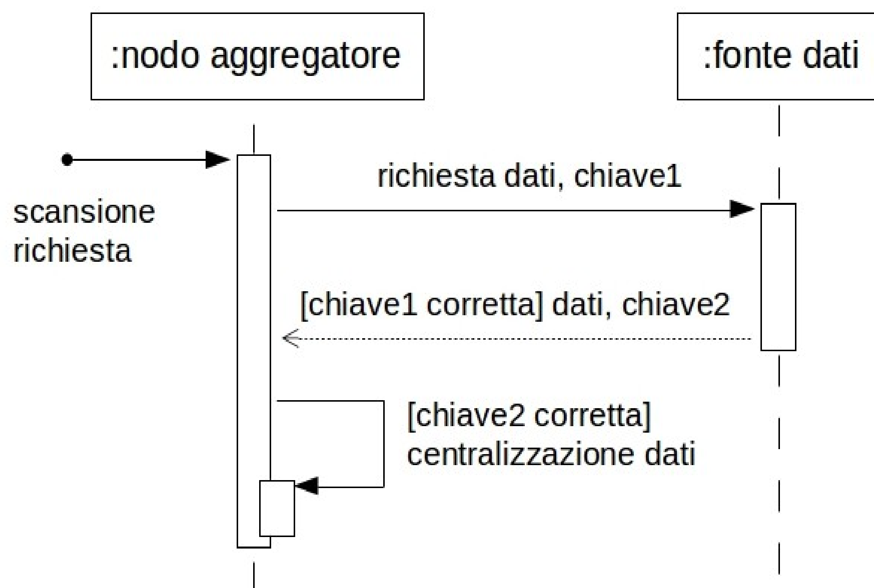
Una volta autenticato con successo all'interno del nodo, all'utente che possiede i dati (o chi delegato per esso) viene presentata una interfaccia di registrazione fonti. Il nodo di aggregazione per poter andare a recuperare i dati direttamente da chi li pubblica in modo automatico ha la necessità di conoscere l'indirizzo web a cui poter poterli richiedere. In questo caso si dice che l'utente inserisce una sottoscrizione, ovvero una coppia nome-indirizzo che renda possibile al nodo recuperare i dati. Ovviamente durante le elaborazioni interne le sottoscrizioni si arricchiscono di altri dati, ma all'utente vengono richiesti solo poche informazioni così da rendere l'interazione col software più semplice.



Le sottoscrizioni inserite all'interno del nodo restano visibili all'utente, il quale può decidere di cancellarle nel caso non siano più desiderate.

3.1.3 Raccolta e centralizzazione dei dati

Una volta forniti al nodo di aggregazione l'indirizzo per poter accedere alle fonti di dati si avvia subito l'operazione di raccolta dati. Il nodo contatta la fonte dati inviando una delle due chiavi ottenute dall'utente in fase di registrazione, più precisamente la chiave "aggregatore". In questo modo il nodo dimostra di essere realmente se stesso evitando possibili camuffamenti da parte di software maligni. La fonte di dati dovrà rispondere, nel caso il codice sia corretto, con i dati corredati dall'altra chiave fornita all'utente. In questo modo anche la fonte di dati può dimostrare di essere realmente se stessa. I dati allegati con la risposta vengono perciò memorizzati assieme a quelli di eventuali fonti diverse, così da centralizzarli.



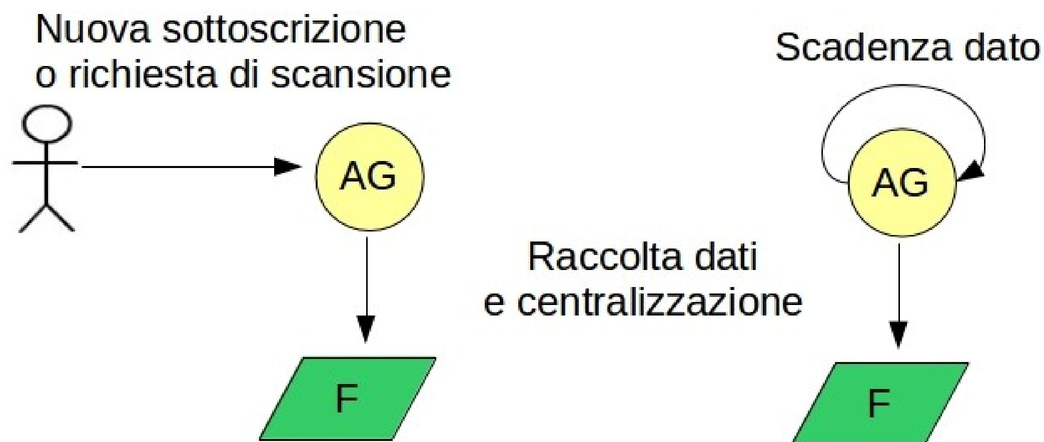
Questa operazione è il cuore di tutto il processo di aggregazione ed è necessario specificare alcuni dettagli in merito. Innanzitutto, poiché le chiavi non vengano intercettate durante la richiesta dei è necessario che il canale tramite cui dialogano la fonte dei dati e il nodo sia criptato; nel caso del mio progetto, poiché è ideato per funzionare attraverso internet, questo significa che è necessario specificare indirizzo che utilizzi il protocollo https.

Un altro dettaglio importante è che per poter eseguire correttamente la raccolta la fonte deve rispondere in un determinato modo, come descritto sopra. Questo però significa che il software che fornisce i dati non potrà essere completamente agnostico dai metodi utilizzati dai nodi di aggregazione. In particolare, dovrà riuscire ad accettare e fornire le chiavi prodotte in fase di registrazione utente. Per questo motivo le fonti di dati fanno concettualmente parte dell'albero che rappresenta il sistema di aggregazione.

Infine, descritta questa procedura sorge il problema di come propagare una eventuale modifica dei dati dalla fonte ai dati centralizzati. Il nodo prevede due modalità per poter acquisire nuovamente i dati dalla fonte: la prima è su richiesta dell'utente che ha inserito la sottoscrizione corrispondente alla fonte. Attraverso l'interfaccia grafica è possibile richiedere una scansione tramite un pulsante apposito, il quale farà corrispondere una nuova scansione della fonte da parte del nodo.

L'altra modalità è invece la scansione automatica. Nel caso il dato aggregato includa la data in cui si potrà considerare scaduto il nodo dovrà curarsi di controllare questa data e impostare una nuova scansione della fonte nei momenti direttamente

successivi. Il controllo sulla scadenza dei dati verrà effettuato a intervalli prestabiliti configurabili in fase di avvio del nodo. La fase di recupero dei dati e la registrazione della loro fonte, in tal caso, sono eventi asincroni fra di loro.



3.2 Accesso ai dati centralizzati

Per come descritto fino a questo punto, un sistema di aggregazione consiste in uno o più nodi aggreganti i quali a loro volta hanno la possibilità di raccogliere dati da una o più fonti. Un sistema del genere è sicuramente incompleto e manca della seconda azione elementare relativa all'aggregazione di dati, ovvero la possibilità di accedere ai dati facendo una ricerca con parametri personalizzati. La successiva parte di dissertazione spiega come questa azione venga implementata all'interno del software da me realizzato.

3.2.1 Nodi proxy e nodi aggregatori

I "nodi aggregatori" come definiti nella sezione precedente sono in grado di raccogliere i dati ma sono sprovvisti di strumenti per comunicare fra loro e per permettere a utenti di accedere a questi dati.

In questo caso il concetto di "utente" è differente rispetto a quello usato per nodo aggregante: mentre prima si parlava di personaggi in possesso di dati con l'intento di registrare la propria fonte al sistema di aggregazione, qui si intendono invece utenti senza legami con le fonti di dati il cui scopo è quello di accedere ai dati aggregati riuscendo a modificare alcuni semplici parametri di ricerca.

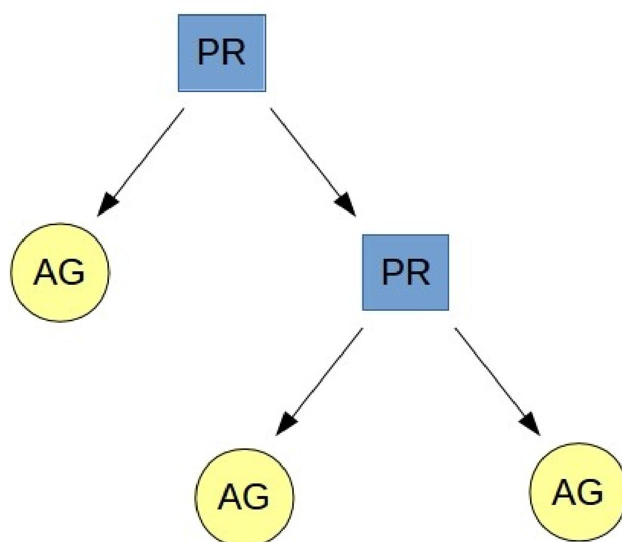
Anche se fosse prevista un'interfaccia grafica di ricerca per ogni nodo aggregatore, se un utente volesse unire dati che sono raccolti da due nodi diversi si troverebbe

impossibilitato a compiere questa semplice operazione. È qui che si rivela lo scopo del secondo tipo di nodo da me ideato per questo sistema software, ovvero il nodo di tipo proxy. Il compito dei nodi proxy è quello di collegare fra di loro diversi nodi aggregatori e offrire all'utente una interfaccia grafica per la ricerca dei dati. Quando un'utente commissionerà una ricerca a un proxy allora esso si occuperà di inoltrare la ricerca ai nodi aggregatori con cui è collegato e di restituire il risultato totale.

Prima di capire però come funziona l'operazione di ricerca all'interno di una rete di nodi composta da nodi aggreganti, nodi proxy e fonti di dati è necessari definire in modo preciso quali relazioni si possono instaurare fra questi tipi di nodi.

3.2.2 Strutturazione dei nodi

Più che di collegamenti fra nodi si può parlare di vere e proprie gerarchie. Si può pensare ai nodi del sistema di aggregazione come ai nodi di un albero: in questo caso le foglie sono interpretate dalle fonti di dati in quanto fanno parte del sistema di aggregazione e sono gli ultimi elementi della catena, ovvero chi realmente possiede i dati originali. Questi dati devono essere raccolti dai nodi aggregatori quindi i nodi direttamente superiori alle foglie potranno essere solo di tipo aggregatore. Ogni aggregatore invece potrà avere come padre solo nodi proxy così che questi ultimi si occupano di inoltrare le richieste a tutti gli aggregatori che fanno loro riferimento.



I proxy però a loro volta possono avere un padre di tipo proxy: questo consente di creare reti strutturate con alberi a più livelli. Prendiamo per esempio nodi aggreganti che raccolgano i dati relativi ai negozi di varie provincie italiane: permettendo a più proxy di concatenarsi fra di loro potremmo dividere i negozi a livello di singola

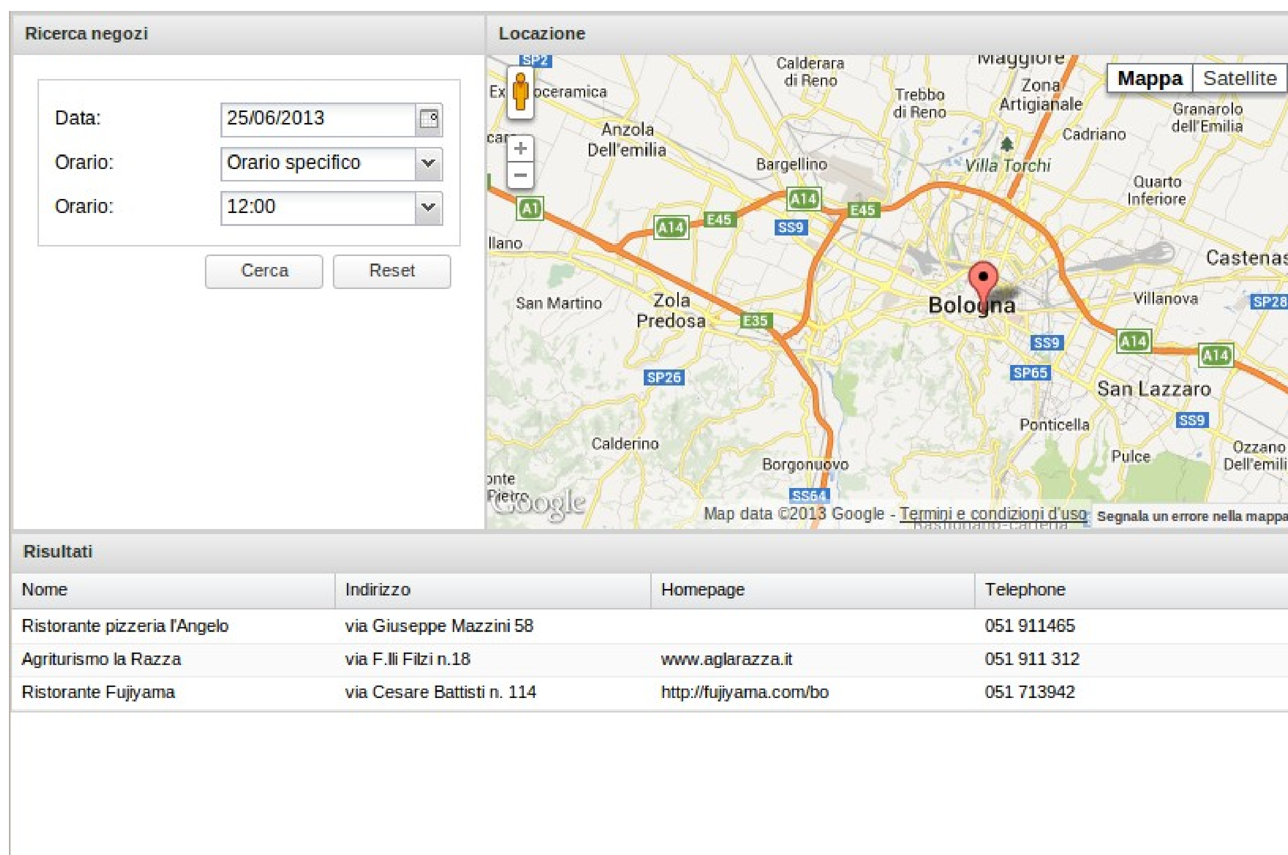
provincia, poi di regione accorpando tutti i nodi proxy con su altri superiori, e infine raccogliere tutti in un unico nodo proxy nazionale.

Permettere ai proxy di accettare richieste da altri proxy consente anche, in caso di bisogno, di risolvere il problema di avere troppe connessioni su un singolo nodo. Nel caso in cui un nodo proxy si trovasse ad avere un numero troppo elevato di nodi aggregatori inferiori e un numero molto elevato di richieste, anche con la gestione a eventi prima o poi si creerebbero dei rallentamenti di sistema; in questo caso sarebbe sufficiente eseguire un nuovo proxy e collegare una parte dei nodi a questo nuovo proxy. Così facendo, il numero di connessioni che il proxy originale dovrebbe gestire si abbassa considerevolmente pagando un piccolo prezzo, ovvero il tempo di comunicazione fra il proxy originario e quello nuovo.

Abbiamo quindi tutti che tutti i nodi che non sono foglie o non sono padri di foglie devono essere di tipo proxy.

Viste le strutture realizzabili con i nodi del sistema di aggregazione possiamo andare a osservare nel dettaglio l'operazione di ricerca da me ideata.

3.2.3 Ricerca applicata a dati centralizzati



I nodi proxy rendono disponibile agli utenti una semplice interfaccia di ricerca in cui

specificare alcuni parametri relativi ai dati che vengono aggregati. Una volta impostati i parametri e confermata la ricerca, il proxy si occupa di comunicare a tutti i nodi inferiori, a prescindere dal tipo, i parametri della ricerca. In questo passaggio il proxy dovrebbe elaborare al minimo la natura dei parametri specificati dall'utente tramite l'interfaccia grafica così da essere più "agnostico" nei confronti del tipo di dato aggregato. I nodi inferiori se di tipo aggregatore eseguiranno la ricerca sui dati memorizzati e restituiranno una risposta, mentre se di tipo proxy faranno la stessa operazione del nodo padre, ovvero inoltreranno ai nodi inferiori la richiesta.

3.3 Aggregazione di timetable

Per provare il software su un caso d'uso reale ho scelto di appoggiarmi su un precedente lavoro di tesi scritto da Vincenzo Ferrari, "Modelli per l'aggregazione di dati federati: il caso degli orari di apertura degli esercizi commerciali e enti pubblici" [21]. All'interno di questa tesi vengono definite le timetable, ovvero file XML scritti secondo specifici parametri e in grado di descrivere luoghi fisici e relativi orari di apertura. È possibile per esempio utilizzare le timetable per specificare dati relativi a negozi fisici, come indirizzo, homepage, orari di apertura e altro.

È facile immaginare l'utilità di poter cercare negozi tramite internet specificando dettagli geografici o riguardanti l'orario di apertura. È altrettanto facile immaginare come un servizio del genere, se avviato e ben pubblicizzato, possa arrivare ad avere molte visite giornaliere e quindi un numero elevato di connessioni da dover gestire. Anche Google sta pubblicizzando un servizio analogo [41], il che suggerisce quanto un servizio del genere possa diventare importante oggi.

In questo contesto concreto si mostrano i vantaggi apportati da Node JS, ovvero la possibilità di scrivere programmi con una buona gestione delle connessioni senza eccessivo sforzo da parte del programmatore non esperto di thread e sistemi operativi. Un programma che riuscisse nell'aggregare timetable correttamente e gestisse bene un elevato numero di connessioni permetterebbe di dimostrare la tesi di questa dissertazione.

Andiamo a vedere alcuni esempi pratici di utilizzo del mio sistema di aggregazione applicato al contesto delle timetable.

3.3.1 Inserimento di dati relativi a singoli negozi

Il sistema di registrazione utente ideato risulta comodo in un contesto in cui molti negozi necessitano di esprimere i dati relativi alla loro attività ma di non poter

vedere e modificare dati altri. Il gestore, o chi delegato da esso, potrà registrarsi a un determinato aggregatore a seconda di come sia stata organizzata la distribuzione dei nodi. Ad esempio, come in un esempio precedente, potrebbe essere una buona idea distribuire un nodo aggregatore per ogni provincia così da riunire poi tutte le provincie all'interno di nodi proxy regionali e questi ultimi a loro volta sotto un nodo proxy nazionale.

Prendendo come ipotetica base questa distribuzione di nodi, gestori o commessi potrebbero registrarsi al loro nodo di riferimento e registrare la propria fonte di timetable all'aggregatore. Dove sia fisicamente in esecuzione la fonte di dati non è rilevante; verosimilmente potrebbe essere su un server privato dell'azienda o sulla stessa macchina che esegue il nodo aggregatore.

Una volta registrato l'indirizzo della timetable, il nodo aggregatore si occuperebbe di recuperare i dati, elaborarli per rendere più veloce la ricerca nel caso ce ne fosse bisogno e inserirli in una base di dati. Già pochi secondi dopo sarebbe inclusa nelle ricerca anche la nuova timetable appena inserita.

Le timetable, provviste di una data di scadenza, verrebbero raccolte nuovamente poco dopo la suddetta data.

3.3.2 Inserimento di dati relativo a catene di negozi

Una timetable può essere utilizzata per descrivere in un solo file più luoghi fisici e un loro utilizzo verosimile potrebbe prevedere la specifica di tutti i negozi relativi a una stessa ditta/catena in una sola timetable. Presa come base la distribuzione di nodi descritta nella sottosezione precedente, nel caso in cui i negozi fossero in diverse provincie e/o regioni non si riuscirebbe già più a capire a quale nodo aggregatore dovrebbe far riferimento una persona incaricata di registrare la fonte della timetable.

Grazie alla struttura ad albero e alla connessione dal basso verso l'alto dei nodi, risolvere questo problema risulta molto semplice: è sufficiente aggiungere a livello regionale e/o nazionale un nodo aggregatore per quelle timetable con dati misti e consentirvi l'iscrizione di queste catene di negozi. La rete esistente prima di questa aggiunta non dovrebbe subire alcun cambiamento e i nodi non verrebbero mai fermati.

3.3.3 Ricerca di dati relativi a negozi

Una volta istanziati i nodi e aggregati i dati sarà possibile pubblicare l'interfaccia

grafica di ricerca del nodo proxy nazionale e permettere agli utenti di cercare i negozi. Posizione geografica e orari di apertura sono i parametri più classici di ricerca e l'interfaccia dovrebbe permettere la loro specifica. Nel caso anche le API web venissero pubblicate sarebbe possibile costruire altre interfacce o software che automaticamente si occupino di raccogliere ed elaborare i dati.

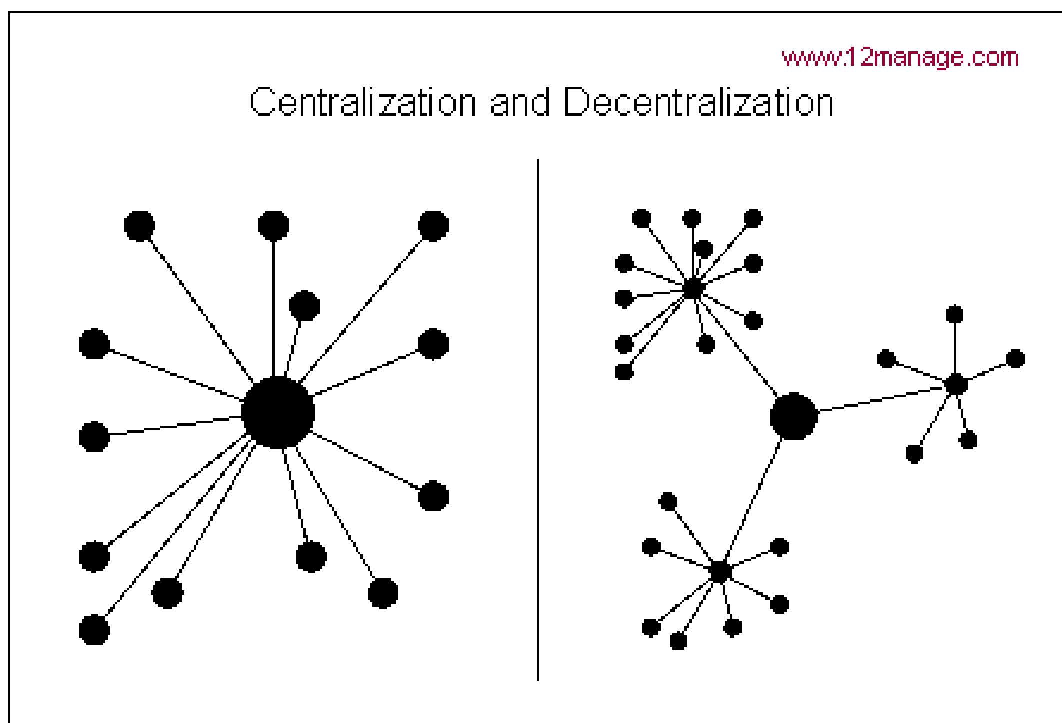
La mia implementazione del software segue queste descrizioni e il capitolo successivo introduce più concretamente come vengono implementate le funzionalità appena descritte.

4 Descrizione tecnica del sistema

4.1 Struttura decentralizzata

Una volta descritti i compiti del sistema di aggregazione Mangrove e dei nodi che lo compongono è possibile scendere a un livello più basso di descrizione, trattando elementi tecnici e singole funzioni svolte dai nodi.

A livello di struttura viene utilizzata un'architettura decentralizzata, adeguata a rappresentare l'albero del sistema di aggregazione.



Nonostante le critiche [42] che vengono mosse a questo modello esso presenta due importanti vantaggi se visto nel contesto specifico di questo progetto: il primo è che si presta bene a tenere il passo con la crescita della rete (intesa sia nel senso di quantità di dati che di dispositivi). La possibilità di espandere la struttura dal basso in modo arbitrario renderà possibile, nel momento in cui sarà necessario, aggiungere

nodi in modo indolore rispetto alla struttura già esistente. Un altro vantaggio è che l'idea di "piccoli nodi" che accettano migliaia di comunicazioni ben si sposa con le caratteristiche di Node JS, il quale permette di scrivere software che gestisce in modo molto efficace le comunicazioni di rete ma decisamente peggio calcoli complessi. Si tratta quindi di costruire un albero di nodi che comunichino fra di loro.

La descrizione segue la struttura dell'albero di aggregazione partendo dai nodi foglie, le fonti di dati, passando per i nodi aggregatori e terminando con i nodi proxy.

4.2 Timetable e UTM

Il lavoro specifico riguardante il formato delle timetable e il software per la loro creazione è reperibile presso la biblioteca dell'università di Bologna nella tesi di Vincenzo Ferrari; qui introduco solo superficialmente i concetti base per permettere la comprensione del mio lavoro e la cooperazione dei due sistemi.

Una timetable è essenzialmente un file scritto con XML contenente dati relativi a uno o più negozi. Si compone di tre parti in cui la prima contiene informazioni relative al file (data di creazione, autore, nome della timetable). In questo campo è molto importante il campo che descrive la data di scadenza del file, oltre alla quale il nostro sistema di aggregazione dovrà provvedere a recuperare una nuova versione del file.

La seconda parte che compone le timetable è la sezione "venues", letteralmente sedi, in cui possono essere contenute una o più descrizioni di luoghi fisici corrispondenti a negozi. Non ci sono restrizioni su quale negozi è possibile descrivere in questo campo anche se il buon senso ci indica che sia utilizzabile per descrivere in un unico luogo tutti i negozi relativi a una catena o a un singolo paese. Questo è il punto più importante della timetable per quanto riguarda questo progetto, in quanto contiene le informazioni su cui si andranno poi a fare delle ricerche, come posizione del negozio, date di apertura, orario di apertura e così via.

Infine l'ultimo blocco di cui è composta una timetable è la sezione "locale", contenente elementi per poter contestualizzare le date di una timetable in una determinata lingua. Per esempio, in questa sezione sono contenuti i nomi dei giorni della settimana e dei mesi in italiano, così come le festività riconosciute in Italia.

```
<?xml version='1.0' encoding='utf-8'?>
<timetable xmlns="http://www.fabiovitali.it/tt"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.fabiovitali.it/tt  Timetable-prova.xsd">
```

```
	<metadata>
		<this href="" />
		<permanentUri href="" />
		<title value="Timetable UTM" />
		<author href="www.duma.blackmesalabs.it" showAs="Stefano Gombi" />
		<created value="14/05/2013" />
		<validFrom value="15/05/2013" />
		<validTo value="14/05/2014" />
		<authoringTool value="" />
		<authoritative value="" />
		<previousDocument href="data/Timetable.2013-03-21.xml" />
		<source href="" />
	</metadata>
	<venues>
		<venue id="txhb1295X0" showAs="Agriturismo la Razza">
			<information>
				<name value="Agriturismo la Razza" />
				<address value="via F.lli Filzi n.18" />
				<telephone value="051 911 312" />
				<coords value="44.49493291809099,11.342101015869162" />
				<homepage href="www.aglarazza.it" />
			</information>
			<hours>
				<hour id="txhb1295X0WeeklysUo5vGfvlB"
pattern="dom,lun,mar,mer,gio,ven,sab:0830-1230,1330-1730." type="Weekly" />
				<hour id="txhb1295X0FestivityAURR290S0B"
pattern="#festivi:0830-1230,1330-1730." type="Festivity" />
			</hours>
		</venue>
	</venues>
	<locale>
		<weekDays value="dom lun mar mer gio ven sab" />
		<months value="gen feb mar apr mag giu lug ago set ott nov dic" />
		<dateFormat value="d/M" />
		<weekStartsWith showAs="Lunedì" value="1" />
		<dicts>
			<dict href="http://www.fabiovitali.it/placeTypes" id="placeTypes" />
			<dict href="http://www.fabiovitali.it/shopTypes" id="shopTypes" />
			<dict id="festivi">
				<items>
					<item id="01 Jan" showAs="Capodanno" value="01/gen" />
					<item id="06 Jan" showAs="Epifania" value="06/gen" />
```

```
                    <item id="31 Mar" showAs="Pasqua" value="31/mar" />

                </items>
            </dict>
        </dicts>
    </locale>
</timetable>
```

La creazione di file XML che rispettino il formato delle timetable è possibile grazie a un software costruito da Vincenzo Ferrari, UTM. UTM è un software scritto in Python e EXTJS che permette a un utente di modificare e creare contenuti rispettando le specifiche tramite una comoda interfaccia web. Questo software rappresenta, per il progetto realizzato in questa tesi, l'origine dei dati che vengono raccolti nel sistema di aggregazione. Ho dovuto apportare solo alcune modifiche secondarie per permettere la comunicazione fra i due software. UTM è quindi da considerarsi a tutti gli effetti la fonte di dati di questa particolare implementazione.

Le "subscription", termine di cui farò uso più avanti nel testo, non sono altro che la registrazione di un determinato UTM nel sistema di aggregazione, ovvero le sottoscrizioni descritte nel terzo capitolo. Concettualmente si tratta di una coppia nome-indirizzo tramite cui il sistema può risalire alla fonte del dato e reperirlo a intervalli regolari.

4.3 Nodi aggregatori

I nodi aggregatori sono tutti i nodi padri delle fonti di dati e sono il centro dell'azione di centralizzazione dati; per questo motivo il compito che contraddistingue questi nodi dagli altri è il dialogo diretto con le fonti di dati e con il database, necessario per la memorizzazione in un unico punto dei diversi dati raccolti.

Seguendo la descrizione del nodo fatta nel capitolo precedente possiamo distinguere le azioni fondamentali implementate all'interno del software che lo rappresenta:

- Gestire la registrazione di utenti autorizzati ad inserire subscription
- Gestire l'inserimento/cancellazione di subscription
- Scansionare periodicamente le subscription ottenendone timetable
- Accettare richieste da parte di proxy con determinati parametri
- Comunicare con un database per memorizzare/ricercare timetable e subscription

Andiamo a vedere più in dettaglio le singole funzionalità per avere un quadro più chiaro di questo componente.

4.3.1 Gestione utenti

Poiché non dovrà essere possibile inserire subscription a chiunque senza poter effettivamente determinare una responsabilità su tale inserimento, è necessario prevedere la registrazione di utenti e la loro attivazione per evitare utenti falsi (possibilmente generati da altre macchine). Dovrà anche essere prevista la generazione di due chiavi di autenticazione: una per identificare l'aggregatore agli occhi del software che fornisce la timetable, e una per l'operazione inversa. Dovrà essere possibile usufruire di questi servizi tramite un'interfaccia web.

Per non complicare inutilmente la situazione, le chiavi sono fornite per utente e non per subscription, ovvero se sono presenti diversi software tutti gestiti dallo stesso utente, essi dovranno includere lo stesso paio di chiavi; invece ogni utente ha la propria chiave per identificare l'aggregatore, così da diminuire le possibilità di "camuffarsi" da aggregatore da parte di terzi malintenzionati. Il software con cui si creano le timetable dovrà essere in grado di gestire l'autenticazione tramite queste due chiavi.

4.3.2 Gestione subscription

Una volta registrato e attivato, l'utente dovrà poter accedere ad una interfaccia grafica tramite cui gestire le proprie subscription. Dovrà essere presente un sistema di autenticazione per permettere solo al proprietario la modifica delle voci già inserite.

Le azioni possibili sulle subscription sono poche ed elementari: creazione, eliminazione, richiesta di scansione.

4.3.3 Scansione subscription

Questa funzionalità è interna all'aggregatore e non presenterà alcuna interfaccia utente. Il software dovrà occuparsi di recuperare le timetable ottenibili all'indirizzo indicato da una subscription, utilizzando per l'autenticazione le due chiavi fornite in fase di registrazione utente. In tre casi dovrà avvenire una scansione:

- subito dopo l'inserimento nel database di una nuova subscription, per popolare il database con il nuovo dato;
- a richiesta dell'utente, per poter rispecchiare nel database eventuali modifiche fatte sulla timetable locale;

- allo scadere di una timetable. Ogni timetable ha una data di scadenza e il software dovrà essere in grado di riconoscere se una timetable è scaduta e, di conseguenza, recuperarne una versione valida. Il controllo della scadenza è effettuato periodicamente; di base viene effettuato ogni giorno, ma dovrà essere possibile specificare un valore differente.

4.3.4 Rilevamento richieste proxy

L'aggregatore dovrà essere in grado di collegarsi a un (e solo un) proxy tramite TCP e rimanere in attesa di richieste da parte sua. I primi scambi di messaggi fra l'aggregatore e il proxy dovranno servire per autenticazione tramite sfida, mentre le comunicazioni successive simboleggeranno richieste da parte del proxy, a cui l'aggregatore dovrà rispondere con i dati ricevuti dal database.

4.3.5 Gestione del database

Rimanendo fedeli alla filosofia di Node JS di lasciare più lavoro possibile ad altri software specializzati, l'aggregatore dovrà occuparsi di interpretare le richieste pervenute dal proxy per trasformarle in query del database a disposizione per svolgere il minor lavoro possibile. Dovrà anche essere possibile inserire i dati ricevuti dalla scansione delle subscription, prevedendo una eventuale fase di trasformazione nel caso i dati "grezzi" necessitassero di essere lavorati per motivi di velocità.

A questo punto è possibile immaginarsi l'aggregatore come un programma con il compito di dialogare con quattro agenti esterni: il client web, il database, il proxy e i fornitori di dati. Queste comunicazioni però saranno presenti in quantità decisamente differenti: mentre l'interrogazione con un fornitore avverrà abbastanza raramente (idealmente una volta all'anno per fornitore) e la comunicazione con il client web avverrà solo per nuove iscrizioni di fornitori ed eventuali modifiche, una richiesta da parte di un proxy potrà avvenire molto spesso. Allo stesso modo, ogni richiesta da parte del proxy richiederà un'interrogazione del database, quindi anche in questo caso stiamo parlando di un'ordine di grandezza molto superiore rispetto ai primi due casi discussi. È possibile quindi definire un aggregatore quasi totalmente in funzione delle sue connessioni.

4.4 Nodi proxy

I proxy saranno invece, nel nostro albero che rappresenta il sistema di aggregazione, tutti i nodi che non sono foglie o padri di foglie. Dovrà essere possibile concatenarne più assieme così da creare alberi con due o più livelli di profondità e il loro unico

compito dovrà essere quello di accettare richieste e inoltrarle ai propri client. La tecnologia utilizzata per collegare fra di loro più nodi proxy e proxy con nodi aggregatori è TCP.

Più in particolare, le funzionalità che il software offre sono:

- Accettare il collegamento di nodi inferiori (client, sia aggregatori che proxy)
- Ricevere richieste da una interfaccia web
- Inoltrare richieste ai nodi inferiori
- Unire le singole risposte e restituirle come risultato della richiesta originale

Vediamo più in dettaglio le varie funzionalità.

4.4.1 Collegamento con i nodi inferiori

I diversi nodi dell'albero si uniscono fra di loro partendo dal basso, ovvero è sempre un nodo figlio che si connette con il nodo padre. Nel caso del proxy, esso può avere come figlio un aggregatore o un altro proxy ma sarà sempre compito del figlio avviare la connessione TCP all'indirizzo del padre. Un proxy dovrà quindi rimanere in attesa di richieste di connessione e nel momento in cui avvengono dovrà avviare la procedura per autenticare il client tramite sfida.

4.4.2 Richieste da interfaccia web

Il proxy dovrà esporre delle web API accessibili tramite http(s) per permettere ad una ipotetica interfaccia di esprimere richieste e visualizzare i risultati. In altre parole, il proxy includerà anche un web server in grado di dialogare con diverse interfacce web, così da non legare il software ad una particolare interfaccia.

4.4.3 Inoltrare richieste ai nodi inferiori

Una volta ricevuta una richiesta da parte di una interfaccia il proxy dovrà occuparsi di inoltrare i parametri di questa ricerca ai propri client, siano essi nodi aggregatori o altri nodi proxy. Ad ogni inoltro sarà necessario associare un identificatore così da poter riconoscere le risposte ad esso appartenenti.

4.4.4 Raccogliere le risposte

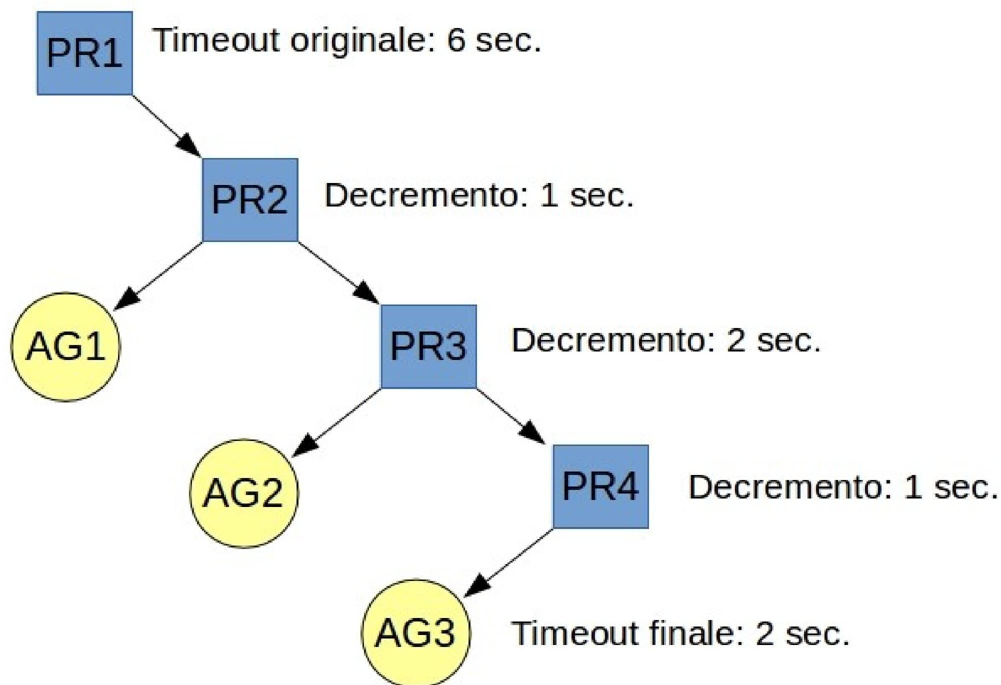
Una volta inoltrata una richiesta il nodo proxy si dovrà mettere in ascolto delle risposte causate dall'inoltro. Nel caso in cui non tutte le risposte arrivassero entro un intervallo di tempo prestabilito (specificabile in fase di configurazione) il proxy dovrà

inviare le risposte parziali al suo nodo genitore e rifiutare le risposte che arriveranno dopo questo lasso di tempo.

Per rendere il proxy il più agnostico possibile rispetto ai vari contesti di utilizzo (interfaccia web, tipo di dato elaborato, database utilizzato...) esso dovrà elaborare e cambiare il meno possibile i parametri specificati in una richiesta e le risposte ottenute dai nodi inferiori; idealmente, non dovrebbe esserci nessuna modifica e il software si dovrebbe occupare solo di comunicare i dati in un formato prestabilito. Riprendendo il metodo usato per l'aggregatore e a maggior ragione vista la quasi completa mancanza di elaborazione interna, possiamo esprimere un proxy in funzione delle comunicazioni che ha con gli altri agenti di questo sistema:

Anche in questo caso dobbiamo tenere conto delle differenze quantitative fra le varie comunicazioni: è vero che le richieste da parte dell'interfaccia web potranno essere molte e contemporanee, ma per ognuna di esse dovranno corrispondere tante comunicazioni quanti sono i client collegati al proxy, quindi si tratta di molte più comunicazioni. Penso che sia in questi casi in cui si può trovare l'utilità del sistema a eventi di Node JS, in cui riuscire a gestire un gran numero di comunicazioni contemporaneamente risulta l'elemento più determinante per le performance del sistema.

4.4.5 Diminuzione del timeout



Uno dei parametri che è possibile specificare nella configurazione di un nodo proxy è

la diminuzione del timeout prevista nel caso in cui la richiesta provenga da un nodo padre. Per spiegare meglio il concetto è utile fare una dimostrazione pratica di cosa avviene quando una richiesta viene passata da nodo a nodo. Siccome il nodo originario a cui viene fatta la richiesta, ovvero la radice, avrà impostato un suo timeout nella configurazione, è necessario che anche i proxy sottostanti rispettino questo timeout. A sua volta però, deve essere calcolato il tempo necessario per le comunicazioni fra la radice e gli altri nodi, ed è esattamente a questo che serve il parametro timeout_decrease: quando un proxy riceve una richiesta dal nodo padre inoltrerà a sua volta le richieste non con il suo timeout specifico ma con la differenza fra il timeout del padre e il contenuto di timeout_decrease. Possiamo dire che questo parametro simboleggia una generica stima del massimo tempo di comunicazione fra i due nodi proxy accettabile.

In futuro si potrebbe prevedere un algoritmo che riconosca automaticamente e a seconda del caso il valore ottimale da impostare.

Nell'immagine precedente, un esempio di una richiesta inoltrata fra più proxy che mostra il comportamento del timeout.

4.5 Database

Per implementare il sistema di aggregazione ho già accennato all'utilizzo di MongoDB. L'implementazione non è strettamente collegata a questa scelta ed è possibile sostituire Mongo con altri database a patto di implementare i metodi offerti dalla classe MongodbWrapper. Vorrei però soffermarmi brevemente sulle caratteristiche di questo prodotto che mi hanno portato a sceglierlo e su altre che purtroppo si sono rivelate problematiche per la realizzazione del progetto, presentando alcune incompatibilità con il caso delle timetable.

MongoDB è un database di tipo NoSQL orientato ai documenti. Probabilmente, fra tutti i database NoSQL oggi in circolazione, Mongo è quello che ha avuto maggior successo e che dispone della comunità più solida [43]. Mongo è stato spesso utilizzato per progetti scritti con Node JS e questa lunga sinergia ha dato vita a ottime librerie, framework e conoscenze da parte della comunità per risolvere problemi legati all'utilizzo contemporaneo delle due tecnologie. In particolare, per questo progetto sono due le qualità che mi hanno spinto ad adottarlo come base di dati: il supporto al JSON e le query dinamiche.

Mongo (come d'altra parte quasi tutti i database NoSQL) utilizza internamente il formato BSON [44] che non è altro che del JSON; a loro volta, gli oggetti JSON non

sono altro che oggetti Javascript. Disporre di un database che accetta e restituisce dati che sono interpretati nativamente da Node JS consente di risparmiare molti calcoli all'aggregatore. In effetti, il formato delle timetable è XML quindi sarebbe stato logico utilizzare direttamente questo formato per memorizzare e cercare le i dati in un database basato su questo linguaggio. Questa scelta avrebbe però comportato una conversione da XML ad oggetti Javascript per ogni richiesta (in un'istanza della classe Filter), ovvero dei calcoli per un'operazione che deve essere eseguita molto spesso.

Tale conseguenza non sarebbe stata coerente con lo stile dei programmi in Node JS di ridurre al minimo del operazioni riguardanti la CPU. Un database in grado di fornire dati già elaborabili senza conversioni è quindi da preferire, anche se necessita una conversione da XML a JSON nella fase di raccolta. Per questo scopo ho creato un file XSL per tradurre le timetable e una patch al software UTM (il creatore di timetable) in modo da eseguire far eseguire ad esso la conversione.

Un vantaggio che MongoDB aveva rispetto a tutti gli altri database basati su JSON era la possibilità di esprimere query dinamiche molto simili a quelle dell'SQL. Le query più utilizzate su sistemi NoSQL sono di tipo map-reduce, ovvero un metodo di ricerca molto avanzato tramite funzioni Javascript passate come parametro di input della query. Il problema delle operazioni di map-reduce è che sono lente se comparate con i tempi generalmente accettabili di attesa nelle ricerche su internet. Le query esprimibili tramite il metodo "find" messo a disposizione dal wrapper di Mongo sono invece molto più veloci e utilizzabili per un sistema di ricerca in real-time come quello ideato per questo progetto.

Le query map-reduce si potranno rivelare più utili in seguito per realizzare calcoli sui dati raccolti e produrre statistiche e indici di utilizzo. Citando un articolo [45] che studia l'utilità di questa tecnica per elaborare grosse quantità di dati:

> *"MapReduce è un modello di programmazione per gestire grosse quantità di dati (più di un terabyte) diffuso da Google nel 2004. Il puntointeressante di questo modello è che le due primitive Map e Reduce sono facilmente parallelizzabili e possono lavorare su grandi insiemi di dati. Si adatta perfettamente alle applicazioni distribuite su larga scala e l'elaborazione di grandi quantità di dati."*

In futuro sarà possibile quindi compiere operazioni complesse sui dati nonostante siano distribuiti su nodi diversi.

La scelta di utilizzare MongoDB ha anche portato però alcuni piccoli inconvenienti

che è necessario conoscere per comprendere i limiti di questa tesi e poterli in futuro superare. Il problema fondamentale e che sta alla base di tutti gli altri è che le query esprimibili col metodo "find" di Mongo non riescono a compiere tutte le operazioni di ricerca desiderate sui dati definiti all'interno delle timetable. Se, per esempio, si volesse compiere una ricerca sull'orario di apertura dei negozi come è possibile specificare nelle timetable, una query di Mongo non sarebbe in grado di riuscire a esprimere tale vincolo. Per questo motivo la classe Filter prevede di restituire sia una query da dare in input al metodo "find" sia una funzione da applicare ai risultati di questa query, per compiere un filtro a maglia più stretta. La soluzione adottata sembra efficace e non ha dato particolari problemi, ma va concettualmente a contraddire il principio di "far fare il meno possibile" ai programmi che eseguono su Node JS. L'ideale sarebbe stato avere un database che accettasse delle query in grado di esprimere al completo i vincoli imponibili alle timetable, ma questo non è il caso di Mongo.

Un altro difetto è che per poter permettere la ricerca sui singoli negozi (venue) descritti in una timetable ho dovuto suddividere gli elementi contenuti in un singolo file XML in più oggetti JSON. Questo comporta qualche problema in fase di ricerca, che comunque viene risolto sempre dalla seconda passata di ricerca in cui si utilizza una funzione Javascript.

4.6 Interfacce grafiche

Entrambi i software sono corredati di interfacce grafiche per poter permettere il loro utilizzo. Le interfacce non devono essere considerate in grado di presentare tutte le potenzialità di questo sistema di aggregazione; allo stesso modo, non sono adattabili all'aggregazione di altri dati. Esse sono state pensate solo per dare un esempio di come ci si può interfacciare con i web server dei rispettivi software e per permettere di testare il software a esseri umani, così da valutarne concretamente il funzionamento.

Le interfacce sono implementate con EXTJS 4.2, un framework Javascript molto diffuso e che mi ha permesso di scrivere il codice in modo strutturato (usando il pattern MVC) e veloce. Le interfacce vengono avviate automaticamente con il rispettivo software sulla porta specificata in fase di configurazione e per accedervi è sufficiente andare con un browser recente sull'indirizzo composto dall'url e dalla porta del nodo aggregatore. Il codice di EXTJS non è però incluso nel software allegato a questa dissertazione poiché il suo peso è eccessivo; per poter utilizzare le interfacce

sarà perciò necessario scaricare EXT 4.2 gratuitamente da internet e inserirlo dentro la cartella “client” con nome “extjs”.

Nel caso si volesse sostituire l'interfaccia grafica è sufficiente cancellare il contenuto della cartella “client” nel rispettivo software e inserivi la propria versione.

Pannello di controllo sottoscrizioni

Aggiungi sottoscrizione | Logout | Richiedi scansione | Cancella

Nome	Indirizzo
Agriturismo La Razza	http://www.larazza.it/timetable
Ristorante Fujiyama	http://fujiyamare.com/timetable
L'angelo ristorante pizzeria	http://langeloferri.it/orario/timetable
Ristorante Piccolo Paradiso	https://piccoloparadisoristorante.it:5000/timetable
Pizzeria Ciokabek	http://ciokabek.it/timetable

5 Test e valutazioni

5.1 Definizione dei test

Una volta realizzato il sistema di aggregazione ci sono diversi elementi che è necessario testare per essere certi del suo funzionamento complessivo. Possiamo dividere i test in due rami: quelli sul funzionamento della struttura e gli stress test.

5.1.1 Test strutturali

Per strutturali si intende la verifica della corretta comunicazione fra i diversi nodi relativi al sistema di aggregazione senza badare ad aspetti come le prestazioni o il consumo di risorse.

Un primo elemento che dovrà essere sottoposto a test è la capacità sia dei proxy che degli aggregatori di comunicare con software in esecuzione su altre macchine. Questo dettaglio potrà sembrare banale, ma molti software di default permettono le comunicazioni solo con programmi in locale, quindi è necessario testare che i software siano stati configurati per accettare e avviare connessioni con l'esterno.

Un secondo elemento da testare è la capacità di un aggregatore di poter gestire più fonti di dati, ovvero collegamenti con diverse istanza del software UTM. Questo significa gestire diverse date di scadenza timetable, riuscire a non confondere i dati durante la raccolta presso diverse fonti e il loro inserimento nel database, gestire l'eventuale indisponibilità di un UTM registrato e riuscire a restituire, su richiesta del proxy, dati relativi a UTM diversi.

Un terzo elemento da testare è la capacità di un proxy di gestire più aggregatori contemporaneamente, il che comporta inoltrare richieste a più nodi inferiori, riuscire a gestire più autenticazioni separatamente, riuscire a unire in modo corretto più risposte e inviare dati parziali nel caso non arrivino risposte da tutti gli aggregatori.

Un quarto elemento da testare è la capacità di un proxy di accettare richieste da un nodo proxy superiore, il che comporta accettare richieste TCP, gestire la diminuzione

del timeout e non confondere le proprie richieste con quelle del nodo superiore.

L'ultimo elemento che questo test si propone di indagare è la capacità di un proxy di gestire sia aggregatori che proxy contemporaneamente come nodi inferiori, trattando i due tipi di nodi come entità dello stesso tipo.

5.1.2 Stress test

Questo aspetto è sicuramente il più difficile da testare poiché non è possibile comparare il sistema con altre versioni identiche a esso ma sincrone. L'ideare sarebbe quello di avere altri programmi, scritti magari in PHP, Python o Ruby e poter misurare la differenza rispetto ai risultati dell'approccio che utilizza Node JS. Questa tesi non comprende altri software di aggregazione scritti con linguaggi diversi; per un confronto di questo tipo è necessario rivolgersi ai numerosi test presenti su internet.

Quello che però ci è possibile misurare è la velocità del software qui implementato, la velocità di un sistema composto da più nodi e la reazione di tale sistema in condizioni di "stress" per osservarne il comportamento. Il test più efficace in questo caso è inviare un numero molto alto di richieste di ricerca ai nodi proxy, possibilmente il proxy radice, così da massimizzare il numero di comunicazioni che si verificheranno all'interno del sistema.

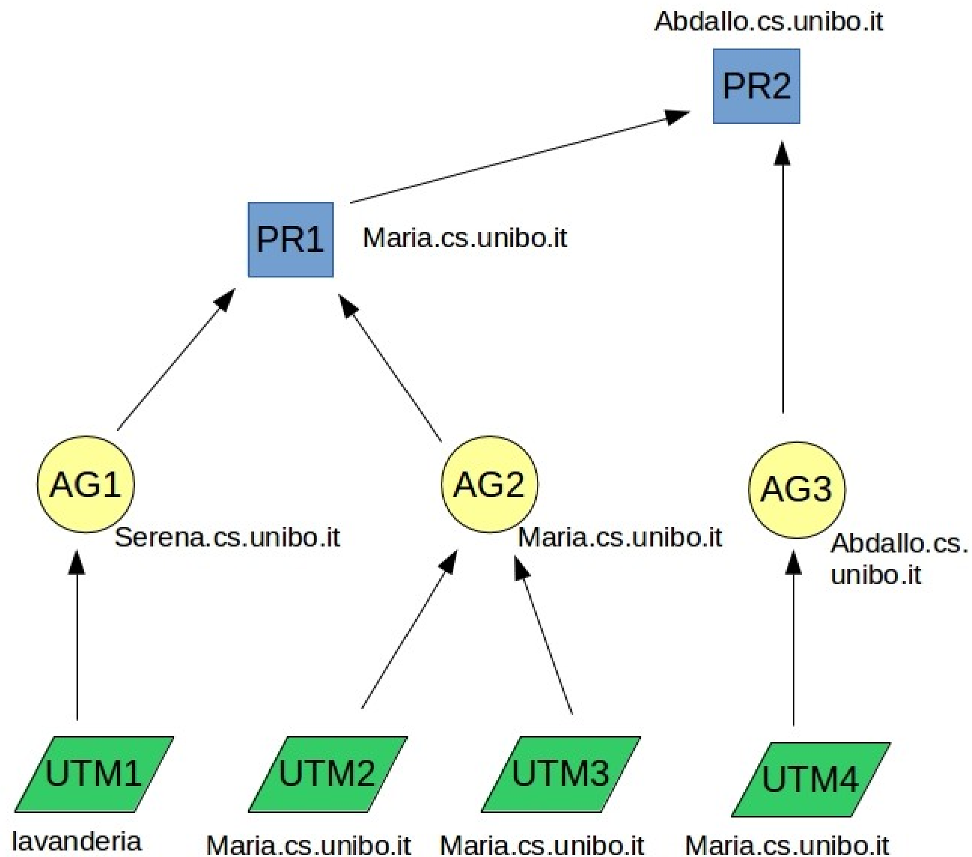
In particolare, una richiesta al nodo proxy radice scatenerà comunicazioni fra tutti i proxy sottostanti, che a loro volta dovranno comunicarlo a i loro nodi sottostanti fino ai nodi aggregatori, i quali interrogheranno il database. Infine, la risposta dovrà seguire il sentiero inverso fino alla radice. Mandare migliaia di richieste consecutivamente ci permetterà di controllare che il software resista sotto carichi di lavoro elevati, che non abbia colli di bottiglia e che tempi di risposta medi siano accettabili.

5.2 Realizzazione dei test

5.2.1 Disposizione del software

Per eseguire i vari software del sistema di aggregazione ho utilizzato i computer messi a disposizione dal laboratorio di scienze dell'informazione dell'università di Bologna Il grafico riportato a inizio capitolo mostra le istanze e i collegamenti realizzati fra di esse, dove per AG si intendono i nodi aggregatori, per PR i proxy e con UTM i software di creazione timetable. Solo UTM1 non è in esecuzione su una macchina di laboratorio ma su un mio server esterno privato situato a casa mia

(lavanderia).



In questa disposizione dei software è possibile ritrovare tutti i requisiti dei test strutturale prima esposti:

- il collegamento fra UTM1 e AG1 verifica la capacità di dialogare dell'aggregatore con software su macchine diverse;
- il collegamento fra AG1 e PR1 verifica la capacità del proxy di dialogare con aggregatori su macchine diverse;
- il collegamento fra UTM2, UTM3 e AG2 verifica la capacità dell'aggregatore AG2 di gestire più fonti di dati contemporaneamente;
- il collegamento fra AG1, AG2 e PR1 verifica la capacità del proxy PR1 di gestire più aggregatori contemporaneamente;
- il collegamento fra PR1 e PR2 verifica la capacità del proxy PR1 di accettare un proxy superiore;
- il collegamento fra PR1, PR2 e AG3 verifica la capacità del proxy PR2 di gestire sia aggregatori che proxy contemporaneamente.

Un altro elemento presente in questo scenario di test è la verifica della gestione da parte di AG1 di un mancato collegamento con UTM1: poiché a volte il mio server privato viene spento o rimane senza connessione, AG1 dovrà essere in grado di gestire la mancata comunicazione e reagire di conseguenza.

5.2.2 Stress e prestazioni

Gestire una stima realistica delle prestazioni di un sistema accessibile tramite internet spesso risulta difficile a causa dei ritardi dovuti alla rete, i quali non sono assolutamente prevedibili per una serie di fattori in gioco. Per eliminare una buona parte dell'incognita dovuta alla rete i test saranno eseguiti sempre su un computer del laboratorio di informatica dell'università di Bologna, così da prevedere una comunicazione di rete il più diretta possibile.

Anche le caratteristiche delle macchine sono fattori determinanti per la velocità di esecuzione di ogni software, perciò risulta necessario accompagnare ogni test alle specifiche delle singole macchine su cui vengono eseguiti. I computer del laboratorio di informatica sono posseggono dei processori Intel Core2 Duo E7500 @ 2.93GHz, 2GB di memoria RAM e come memoria di massa dischi SATA. Essi sono condivisi fra tutti gli studenti del corso di informatica quindi non è possibile sapere se contemporaneamente ci siano in esecuzione programmi da parte di altri utenti. Per evitare questa interferenza ho eseguito i test durante la notte, controllando che la macchina fosse libera da altri processi durante i test così da ridurre i rischi al minimo.

Per stressare la macchina ho realizzato dei semplici script in Node JS il cui compito è quello di avviare molte richieste ai nodi proxy e misurare il numero di millisecondi impiegati a soddisfarle tutte. I test stampano i risultati sullo schermo ma anche su un file con formato CSV così da poter riutilizzare più facilmente i risultati. Ogni serie di test è ripetuta più volte, così da poter avere un risultato medio più realistico.

5.2.3 Definizione test preliminari

Ho ideato una serie di test preliminari non molto significativi dal punto di vista dello stress imposto al software ma che mi permetteranno di trarre tutta una serie di conclusioni dai loro risultati:

1. inviare 100 richieste contemporanee a PR2 per 5 volte così da stressare tutto l'albero del sistema ;
2. inviare 100 richieste contemporanee a PR1 per 5 volte così da notare se ci

sono differenze sostanziali con PR2 ;

3. inviare 100 richieste contemporanee a PR1 e a PR2 per 5 volte, così da vedere come si comporta PR1 avendo sia richieste private che richieste provenienti dal nodo superiore ;
4. inviare 50 richieste contemporanee per 100 volte a PR2, così da notare eventuali cambiamenti nel tempo di risposta per richiesta rispetto al primo test ;
5. inviare 1000 richieste contemporanee a PR2 per 5 volte, così valutare le differenze col risultato del primo test moltiplicato per 10.

Questi test sono solo una fase preliminare per controllare se si possono trarre informazioni fra le loro differenze.

Successivamente ho implementato altri test che incrementeranno costantemente il numero di richieste fatte ai nodi proxy per vedere l'andamento del software a seconda del numero di comunicazioni.

5.2.4 Versione errata del software

Durante i test mi sono imbattuto in un mio errore nella gestione del logging la quale utilizzava una istruzione sincrona per scrivere su file. Questo errore va contro alle regole di utilizzo di Node JS e rendeva il programma molto più lento nel gestire le comunicazioni. Ciò nonostante, dopo aver corretto il software ho scelto di inserire ugualmente i risultati del programma errato rispetto ai cinque test preliminari perché ritengo che forniscano informazioni interessanti e che permettano di capire quanto i punti di debolezza di Node JS, se non evitati accuratamente, possano rallentare il software.

5.3 Risultati

5.3.1 Test preliminari

I risultati dei cinque test preliminari sono riportati nella tabella sottostante; nella colonna del tempo il risultato è ottenuto facendo la media del tempo totale di risposta per fra tutte le serie effettuate all'interno di uno stesso test.

Nodo testato	Num. richieste	Serie	Tempo programma corretto (ms)	Tempo programma errato (ms)
PR2	100	5	153.4	8315.4
PR1	100	5	127.6	7008.8
PR1 e PR2	100	5	193.4	16126
PR2	50	100	84.86	4308.26
PR2	1000	5	733.4	87738

Possiamo fare diverse considerazioni sui risultati ottenuti. Partiamo dal programma che utilizza in modo errato le istruzioni sincrone:

- tra il test 1 e 2 notiamo come la minor complessità della rete produca tempi di attesa sensibilmente inferiori; possiamo quindi dire che la differenza fra i due risultati è causata dal maggiore livello di profondità della rete di PR2.
- Notiamo anche che nel test numero 3 il tempo di riposta medio risulta un po' più alto rispetto a quello che ci si sarebbe idealmente aspettato, ovvero i tempi del test 1 sommati a quelli del test 2. Questo però è spiegabile con le molte connessioni di tipo diverso che deve gestire PR1, che in questo test deve comunicare su ogni canale molto spesso.
- Un'ultima osservazione che si può trarre è che anche all'aumentare del numero di richieste contemporanee, ovvero dalle 100 del test 1 alle 1000 del test 5, il tempo di risposta medio rimane circa lo stesso. Questo è un buon segnale poiché indica che, entro un certo limite, l'aumento consistente del numero di connessioni non rallenta l'esecuzione, i cui tempi rimangono costanti e sono determinati quasi solo dal codice da me implementato.

I risultati in termini assoluti sono però deludenti (87 secondi per rispondere a mille richieste). I risultati del programma corretto sono invece molto soddisfacenti: un utilizzo corretto di Node JS ci permette di osservare appieno la sua potenza e la sua utilità in questo contesto. Ci sono anche delle altre novità positive che derivano da questo test:

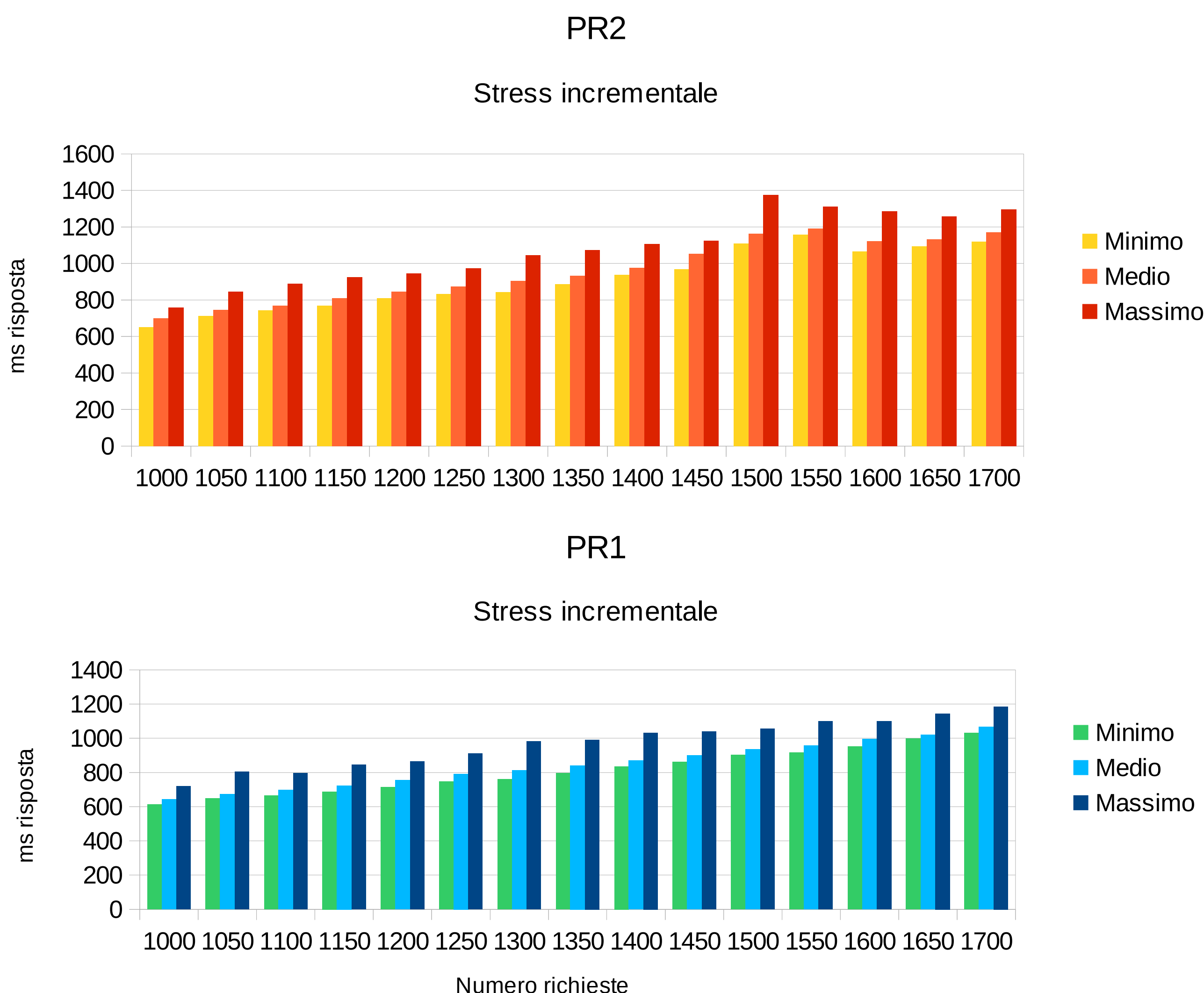
- il test 3, che nella serie precedente era risultato più lento della somma di test 1 e test 2, in questo caso è molto inferiore alla somma di 1 e 2, risparmiando quasi un millisecondo su un totale di 3 della somma.
- Il tempo medio di risposta su richiesta del test 5 risulta molto veloce, più del doppio confronto al test 4 con le sue 50 richieste. Questo è spiegabile con le

politiche di gestione interne del SO che avranno lasciato più tempo in esecuzione il thread di Node JS, ma è impressionante osservare come la rete, sotto carico elevato, arrivi a rispondere addirittura in meno della metà del tempo per richiesta.

Questo esperimento è stato fatto poiché dimostra come all'interno di codice Node JS sia importante utilizzare codice asincrono in operazioni ripetute molte volte. Una mancanza come quella presente nella prima versione del Logger può portare a rallentamenti anche più gravi poiché va ad annullare il vantaggio fondamentale che deriva dall'utilizzo di Node JS, ovvero la gestione a eventi delle operazioni di I/O.

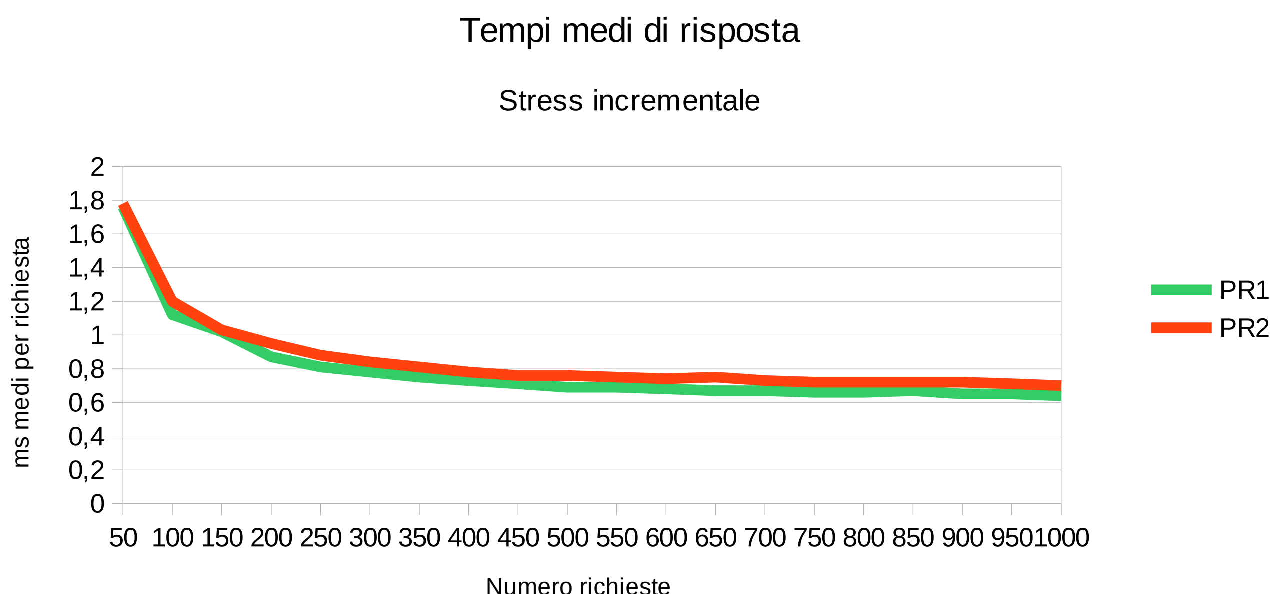
5.3.2 Risultati completi

Il grafico sotto mostra una serie di test effettuati su PR2 con nessuna timetable inserita nel sistema di aggregazione. Per ogni numero di richieste sono state calcolate dieci serie di test. Il grafico sottostante invece mostra lo stesso test fatto su PR1.



Com'è possibile vedere dai test il nodo PR1 impiega sensibilmente meno tempo a rispondere alle richieste grazie alla minor profondità dell'albero di aggregazione. È evidente anche che il tempo di risposta medio è molto più vicino al tempo minimo che al massimo.

Entrambi i grafici mostrano come all'aumentare del numero di connessioni aumenti proporzionalmente il tempo necessario per tutte le richieste. I risultati sono buoni, ma c'è un elemento in particolare che vorrei sottolineare: il tempo di risposta medio necessario per soddisfare ogni richiesta non cresce linearmente con il numero di richieste. Il grafico del tempo di risposta medio per ogni richiesta riesce a mostrare meglio questo fenomeno:

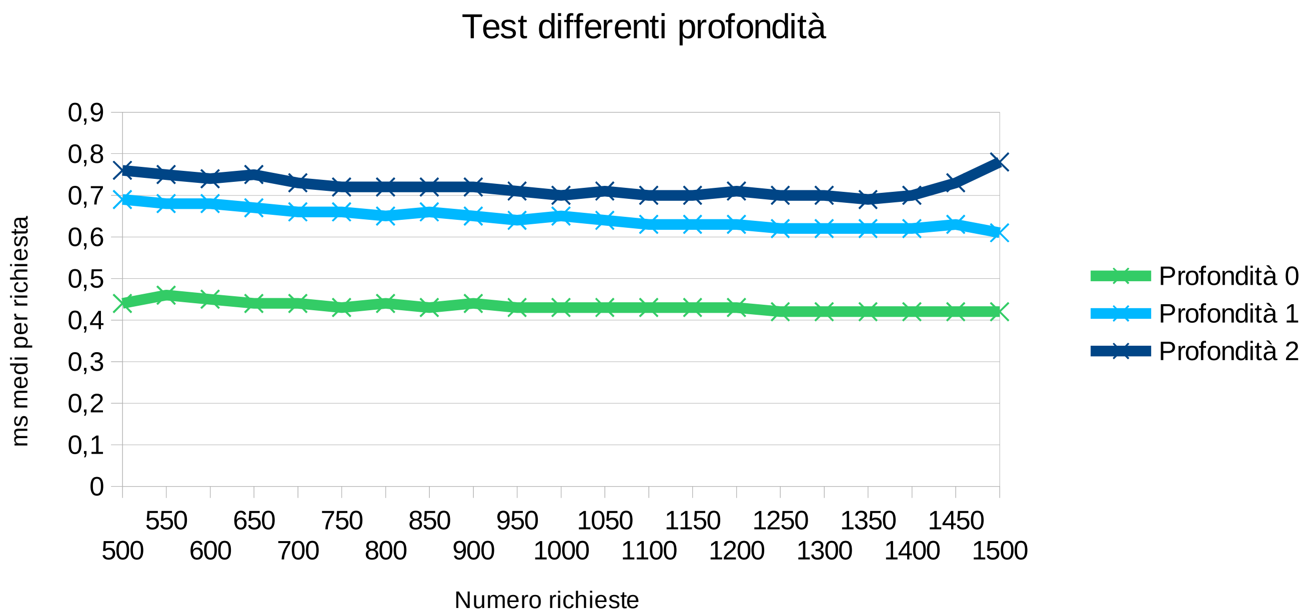


Questo si verifica grazie al mantenimento in cache delle variabili utilizzate da Node JS poiché il loro accesso è richiesto più frequentemente, ma anche grazie al fatto che il SO lascia in esecuzione il thread del codice di Node JS; questo, avendo il controllo del processore per più tempo, riesce a rispondere a più richieste grazie alla sua gestione eventi che mantiene tutto il codice in un solo thread e sfrutta al massimo il tempo concesso dallo scheduler. É lo stesso fenomeno che si è verificato fra i tempi dei test preliminari 1 e 5.

Ovviamente c'è un limite e col l'aumento del numero di richieste si sarebbe raggiunto un valore minimo di tempo medio per risposta sotto cui non è possibile scendere. Per non occupare completamente le macchine condivise del laboratorio sono stato sotto alle 2000 richieste contemporanee, non trovando però questo minimo. Questo, secondo la mia opinione, è un segnale della buona gestione del software poiché con l'aumentare delle connessioni il tempo di risposta medio per richiesta non aumenta

ma anzi diminuisce.

Come già discusso, la differenza dei risultati mostra il peso della maggior profondità dell'albero di aggregazione in termini di tempo. Per avere altre informazioni su quanto il software sia veloce dipendentemente dalla profondità dell'albero ho testato il nodo PR2 scollegandolo dagli altri, ovvero creando un albero di profondità 1. Il grafico seguente mostra il confronto fra i vari livelli di profondità, riportando i risultati dei test precedenti assieme a quelli dell'albero a profondità 1:



5.4 *Valutazioni sulla qualità del codice*

Node JS è una piattaforma molto nuova e in continuo mutamento. La novità di portare la gestione a eventi ad un livello così basso ha aperto nuove problematiche che gli sviluppatori stanno risolvendo con il tempo e l'esperienza. Per esempio la gestione degli errori che avvengono in modo asincrono è diversa dal più classico approccio del tipo “try-catch”. Ho cercato, nella mia implementazione, di utilizzare solo elementi definiti stabili dalle specifiche ufficiali di Node JS, le quali però cambiano molto velocemente e con l'uscita delle nuove versioni sarà possibile utilizzare tecniche e costrutti più avanzati e performanti. Mancano anche pattern chiari e affermati all'interno della comunità, fatto che costringe ogni sviluppatore a risolvere individualmente problemi che spesso in altri linguaggi sono stati già trattati a dovere. In aggiunta a questo bisogna contare il fatto che la mia personale conoscenza di Node JS è limitata dai miei 7 mesi di effettivo utilizzo di questa

piattaforma, quindi alcuni errori dovuti all'inesperienza saranno presenti all'interno del codice.

Ciò nonostante, il codice mi sembra abbastanza pulito e ben strutturato. L'aggregatore, ovvero il nodo più complesso creato per questa tesi, è composto da sei classi in totale. La classe MongoDB ha circa 320 righe di codice, la classe Webservice 230; tutte le altre hanno 100 righe di codice o meno. Questo indica che il codice è conciso e che è stata rispettata la caratteristica di Node JS che vuole il software non troppo complesso.

Anche la riusabilità del codice sembra abbastanza buona; una prova di questo fatto è che fino a un mese prima della consegna di questa tesi il software di aggregazione era completamente generico e non orientato al problema delle timetable. Solo successivamente sono stati modificati gli elementi che hanno adattato il sistema alle caratteristiche delle timetable. In particolare, è stato necessario apportare modifiche alla classe MongoDB per definire le varie entità in gioco, alla classe Filter per definire quali criteri di ricerca fossero accettabili e creare una interfaccia per i nodi proxy. Anche se con qualche problema (dovuto principalmente a MongoDB), la conversione è stata relativamente veloce e semplice.

6 Conclusioni

Il software realizzato per la dimostrazione riesce a svolgere l'aggregazione di dati con una buona capacità nel gestire molte richieste contemporaneamente. Grazie a Node JS ho ottenuto un simile risultato senza dover gestire nessun aspetto riguardante i thread, quindi evitando tutta una serie di difficoltà inerenti alla gestione delle risorse.

Grazie alle caratteristiche di Node JS mi è stato possibile evitare di specificare tutta una serie di dettagli che avrebbero richiesto molte conoscenze per riuscire a ottenere risultati simili a quelli ottenuti dalla mia implementazione di Mangrove. Penso che sia questo il cuore del discorso e la dimostrazione della mia tesi: il software gestisce migliaia di connessioni al secondo in modo efficiente e senza richiedere una profonda conoscenza di thread e dettagli generalmente considerati a “basso livello”.

Certo questa esperienza non è stata esente da ostacoli: l'utilizzo di Node JS ha comportato anche alcune difficoltà che hanno influito sulla semplicità di realizzazione del progetto e la qualità finale del codice. Durante la fase di scrittura del codice i due ostacoli più grandi che ho incontrato sono la mancanza di pattern ben definiti all'interno della comunità e la stesura di codice asincrono.

La mancanza di pattern e di risoluzioni di problemi utilizzate e testate dalla comunità è un problema che si può imputare solo alla breve vita di Node JS. La situazione sta già cambiando e in modo molto rapido: il numero di pacchetti scaricabili da NPM[*] aumenta ogni giorno, la comunità di Node JS cresce ad un ritmo molto elevato e si moltiplicano gli articoli riguardo alla risoluzione di problemi comuni con questa piattaforma. Fra qualche anno molto probabilmente Node JS sarà una piattaforma molto più diffusa e a quel punto il problema dovrebbe ridursi.

Per quanto riguarda la programmazione asincrona, anche in questo caso la colpa non è del tutto imputabile a Node: se è causa delle sue caratteristiche che molti programmatori si sentono smarriti di fronte allo stile asincrono, è anche vero che sta nell'abilità del programmatore riuscire a comprendere il paradigma di programmazione e utilizzarlo al meglio. Sicuramente il progetto presenterà alcuni

* Node Package Manager, tool per la gestione di software scaricabili ed eseguibili su Node JS.

errori relativi alla mia breve esperienza nel campo ma, anche in questo caso, probabilmente dopo una esperienza più prolungata sarei in grado di scrivere un software ancora più performante. Ciò non toglie che questo problema sia imputabile principalmente a me e che quindi non sia completamente riconducibile a Node.

Programmare in Node JS mi ha anche semplificato la vita sotto molti aspetti. Mi ritengo una di quelle persone che Ryan Dahl definisce "non esperti di thread" e creare un software capace di gestire migliaia di connessioni al secondo con buone prestazioni sarebbe stato probabilmente impossibile per me in così breve tempo.

Grazie all'astrazione sui thread che Node JS offre allo sviluppatore sono invece riuscito in pochi mesi a creare un sistema con una gestione ottimizzata delle risorse e con potenzialmente molti meno errori. La sincronizzazione, la gestione di risorse condivise, i parametri legati al sistema operativo e tanto altro mi sono stati nascosti dal sistema di Node e penso che questo, al contrario di una mancanza di libertà, sia stato per me un gran aiuto nello sviluppare il software.

In generale, l'opinione che mi sono fatto è che Node JS debba ancora maturare per poter essere completamente utilizzabile in produzione. Questo sicuramente è a causa della mancanza di pattern chiari, ma è anche dovuto alla struttura interna relativamente stabile che ha Node. Alcuni elementi della struttura base, come i domini, sono ancora in fase sperimentale a detta degli stessi autori e ci vorranno ancora delle versioni e dei test perché questi costrutti sia completamente utilizzabile. Nonostante la piattaforma sia in parte immatura, penso che utilizzarla per costruire questo software sia stata la scelta giusta.

Il giusto equilibrio fra buone prestazioni e semplicità di utilizzo che Node JS già oggi può offrire si rivelano estremamente utili per software come l'aggregatore qui realizzato. Non mi pento quindi di questa scelta ed anzi penso che i fatti dimostrino come questa sia stato un buon progetto, in grado di dimostrare la tesi prefissata.

6.1 Problemi non risolti

Nell'implementazione del software sono rimasti irrisolti alcuni problemi che ritengo opportuno elencare e motivare non solo per evitare incomprensioni ai futuri utilizzatori del software ma, soprattutto, per dare uno spunto di partenza per eventuali futuri sviluppi della tesi.

Sicuramente il problema più grande riguarda l'utilizzo di MongoDB. Nonostante non si possa definire davvero un "problema", bisogna ammettere che le query dinamiche di Mongo mal si sposano con le caratteristiche delle timetable. Questo fa sì che sia

necessario eseguire del codice Javascript ad ogni richiesta pervenuta da un aggregatore. Probabilmente, se avessi adattato il programma a un diverso caso, questo limite non sarebbe emerso. Ma è colpa di Mongo o del formato timetable? La verità sta in mezzo, ovvero sta nell'interazione fra i due che risulta difficoltosa. Non penso che il problema sia risolvibile con altri database NoSQL e nemmeno con SQL. Forse risulterebbe più semplice utilizzare tecnologie basate su XML, ma ciò comporterebbe lavoro extra per nodi aggregatori andando contro alle caratteristiche strutturali di un programma che esegue su Node JS.

Penso che sarà l'utilizzo del software a determinare se i limiti che ho riscontrato con Mongo siano realmente problematici o solo una piccola incoerenza; nel frattempo stanno uscendo nuovi modelli di database NoSQL ibridi. Forse, assieme a una piccola revisione del formato timetable, il problema potrà essere risolto grazie a questi nuovi software. Ribadisco ancora che questo è un problema riscontrabile solo a seconda della natura dei dati che si vuole memorizzare e probabilmente nella stragrande maggioranza dei casi non si incontreranno problemi.

Un altro limite di questo lavoro sta nel non aver usato i *domains* all'interno del codice. I domains sono degli oggetti tramite il quale è possibile intercettare allo stesso modo errori sincroni e asincroni, rendendo molto più semplice e coerente la gestione degli errori nel codice asincrono. Quando ho iniziato lo sviluppo del sistema di aggregazione la versione stabile di Node era la 0.8 e i domains erano considerati instabili; ho preferito quindi non utilizzare questa novità e affidarmi al pattern al momento dominante che consisteva nel passare alla funzione di callback come primo parametro un eventuale errore. In futuro sarà necessario utilizzare i domains in quanto sono lo strumento adeguato per gestire gli errori in un contesto asincrono; i metodi attuali sono solo "trucchetti" temporanei. Già adesso, con la versione 0.10 di Node JS, i domains sono considerati più stabili.

Un ultimo elemento riguarda la gestione degli eventi. Node JS permette di emettere degli eventi a cui possono essere specificati alcune funzioni di callback. Si tratta degli stessi eventi che si utilizzano per permettere l'utilizzo dell'event loop, solo che specificati dal programma e non dal SO. Specificare un evento a livello di software è una tecnica che permette di rendere le classi del progetto più indipendenti fra di loro e garantisce una migliore riusabilità del codice. Nella mia implementazione utilizzo metodi più classici per far comunicare i diversi oggetti fra di loro non avendo compreso appieno la potenza degli eventi. Un miglioramento in questo senso renderebbe il codice molto più semplice e riutilizzabile.

6.2 Possibili miglioramenti futuri

Nonostante la buona riuscita del progetto, durante lo sviluppo sono emerse numerose idee che esulavano dal progetto originale che potrebbero portare miglioramenti al funzionamento del sistema nel suo complesso. Elenco brevemente le più immediate e realizzabili di queste idee, così da aggiungere eventuali spunti per il proseguo di questo lavoro.

Un primo miglioramento possibile è utilizzare una tecnica di streaming per restituire i dati ai proxy. Le ultime versioni del protocollo http supportano tecniche di streaming dei dati, permettendo di non fornire un'unica grossa risposta a una richiesta http ma un costante afflusso di dati che dovrà essere gestito dal client. In un'ottica di lavoro con diversi aggregatori, attualmente in Mangrove la risposta a una richiesta è vincolata ai tempi di risposta dell'aggregatore più lento (o, nel caso peggiore, al timeout configurato). Una trasmissione che invii i risultati man mano che arrivano garantirebbe invece di poter inviare subito i primi dati ricevuti e successivamente le risposte dai diversi nodi del sistema più lenti.

Questa tecnica è ancora poco usata e molti framework per interfacce web (come EXTJS per esempio) non supportano completamente questa funzionalità; sarà una sfida per il futuro sviluppatore riuscire a utilizzare l'HTML chunked.

Un altro miglioramento possibile sarebbe aggiungere un sistema di cache ai proxy. Ovviamente questo complicherebbe la struttura interna dei nodi, i quali sono stati progettati per essere il più piccoli e veloci possibile, però questa complicazione può essere giustificata dai tempi di risposta minori nel caso alcuni test li confermassero. È necessario tenere a mente che una struttura decentralizzata può possedere nodi sparsi per tutto il globo, quindi eventualmente grossi ritardi dovuti alla distanza e all'intasamento della rete. Avere una copia locale dei dati più richiesti o più lenti da trasferire potrà garantire prestazioni molto migliori al prezzo di una complicazione del software di proxy. Si tratta comunque di un software molto semplice, quindi probabilmente c'è ancora spazio per complicazioni di questo tipo senza andare a contraddire le richieste architetturali di Node JS.

Questi due miglioramenti, se utilizzati assieme, potrebbero portare un miglioramento in termini di prestazioni molto evidente e penso che sia questa la strada da seguire se si vuol rendere questo progetto ancora più completo e veloce.

6.3 Event loop e futuro

Secondo le previsioni è probabile che l'aumento della quantità di dati presente su internet e i dispositivi in grado di visualizzarli continueranno a crescere velocemente nel futuro prossimo. La diffusione dei social network e dei servizi online sembra incontrastata; l'accesso a una connessione internet sempre più veloce permette di spostare molti servizi online, i quali richiedono un numero elevato di comunicazioni con i dispositivi dell'utente; nuovi piani tariffari delle compagnie telefoniche rendono disponibile una connessione a internet mobile a prezzi sempre più economici e a velocità sempre maggiori; si stanno concretizzando i numerosi progetti di creare reti wireless su larga scala.

Penso che la necessità di adeguare il software a questo nuovo contesto di sviluppo si sentirà con sempre maggior forza durante i prossimi anni e gli elementi trattati all'interno di questa dissertazione saranno discussi a livello più ampio. Anche le critiche all'approccio basato su eventi si faranno sentire con maggior forza e questo non può che essere un bene se viste nell'ottica di correggere eventuali errori nella gestione risorse. Personalmente, trovo che la gestione basata su eventi sia un cambiamento qualitativo rispetto a una risoluzione del problema che risulta limitata perché nata in un contesto differente e ideata per risolvere altri problemi. Certo, non tutti i server o i software dovranno accettare 10.000 richieste al secondo in un immediato futuro, ma il numero di connessioni sta velocemente crescendo e sempre più sviluppatori si devono confrontare con questo problema.

Questa tesi vuole essere il mio modesto tentativo di dimostrare quanto oggi sia facile, tramite Node JS, scrivere programmi in grado di scalare molto bene un carico di lavoro composto da molte connessioni contemporanee e penso sia riuscita nel suo intento. Node JS, pur nella sua immaturità, ha dato prova di essere una piattaforma stabile e semplice da utilizzare. Credo che il suo futuro sia quello di continuare per ancora qualche tempo a conservare i ritmi di crescita molto elevati che ha avuto in questi due anni. Bisogna ammettere che una buona parte del successo di Node non è dovuta alla sua gestione dell'I/O ma al Javascript, linguaggio in generale molto utilizzato ma che era rilegato solo all'ambito client web. Il successo del sistema di gestione I/O di Node gli ha fatto raggiungere una fama tale da essere conosciuto anche da sviluppatori che volevano semplicemente poter programmare in Javascript anche lato server, inconsapevoli della gestione a eventi.

Il futuro chiarirà le idee. Sempre più articoli su internet riguardano questo argomento e cercano di fare chiarezza riguardo alle caratteristiche di Node. Col tempo e con

l'esperienza i principi di funzionamento di questa piattaforma verranno compresi meglio dalla maggior parte dei programmatori e Node troverà il successo per ciò che realmente fa bene, ovvero permettere a sviluppatori inesperti di utilizzare le risorse della macchina al meglio rendendo lo sviluppo più semplice.

7 Bibliografia

[1] Node JS, http://nodejs.org/

[2] Youtube, http://www.youtube.com/

[3] Ben Parr, *YouTube Surpasses Two Billion Video Views Daily* (17/05/2010), http://mashable.com/2010/05/16/youtube-2-billion-views/

[4] Alexei Oreskovic, *YouTube hits 4 billion daily video views* (23/01/2013), http://www.reuters.com/article/2012/01/23/us-google-youtube-idUSTRE80M0TS20120123

[5] Facebook, https://www.facebook.com/

[6] Cara Pring, *100 more social media statistics for 2012* (13/02/2012), http://thesocialskinny.com/100-more-social-media-statistics-for-2012/

[7] Nicole Bogart, *Calgary flooding puts police in Twitter jail, crashes city website* (21/06/2013), http://globalnews.ca/news/661138/calgary-flooding-puts-police-in-twitter-jail-crashes-city-website/

[8] Chuanpeng Li, Chen Ding, Kai Shen, *Quantifying The Cost of Context Switch* (2007)

[9] Benoit Sigoure, *How long does it take to make a context switch?* (14/11/2010), http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

[10] Krishna Gade, *Twitter search is now 3x faster* (06/04/2011), https://blog.twitter.com/2011/twitter-search-now-3x-faster

[11] NGINX web server, http://nginx.org/

[12] NGINX FAQ, http://wiki.nginx.org/Faq

[13] Joe Williams, *Apache vs Nginx : Web Server Performance Deathmatch* (28/02/2008),

http://joeandmotorboat.com/2008/02/28/apache-vs-nginx-web-server-performance-deathmatch/

[14] Nicolas Bonvin, *Serving static files: a comparison between Apache, Nginx, Varnish and G-WAN* (14/03/2011), http://nbonvin.wordpress.com/2011/03/14/apache-vs-nginx-vs-varnish-vs-gwan/

[15] Esen Sagynov, *What is Nginx?* (2012), http://www.cubrid.org/blog/textyle/222115

[16] *April 2012 Web Server Survey* (04/04/2012), http://news.netcraft.com/archives/2012/04/04/april-2012-web-server-survey.html

[17] Leigh Griffin, Kieran Ryan, Eamonn de Leastar, Dmitri Botvich, *Scaling Instant Messaging Communication Services:A Comparison of Blocking and Non-Blocking techniques* (2011), http://repository.wit.ie/1636/

[18] Javascript, http://it.wikipedia.org/wiki/JavaScript

[19] Mikito Takada, *Understanding the node.js event loop* (01/02/2011), http://blog.mixu.net/2011/02/01/understanding-the-node-js-event_loop/

[20] Ryan Dahl, *Introduction to Node.js*, http://www.youtube.com/watch?v=M-sc73Y-zQA

[21] Vincenzo Ferrari, *Modelli per l'aggregazione di dati federati: il caso degli orari di apertura degli esercizi commerciali e enti pubblici* (2011)

[22] MongoDB, http://www.mongodb.org/

[23] NoSQL Databases, http://nosql-database.org/

[24] EXT JS Framework, http://www.sencha.com/products/extjs

[25] Internet usage statistics - The Internet big picture, http://www.internetworldstats.com/stats.htm

[26] Ruby on Rails Framework, http://rubyonrails.org/

[27] Netty Framework, http://netty.io/

[28] Event loop, http://en.wikipedia.org/wiki/Event_loop

[29] Twisted Engine, http://twistedmatrix.com/trac/

[30] EventMachine Framework, http://rubyeventmachine.com/

[31] Reactor Framework, https://github.com/reactor/reactor

[32] Reactor pattern, http://en.wikipedia.org/wiki/Reactor_pattern

[33] Apache2, http://httpd.apache.org/

[34] Dan Kegel, *The C10K problem* (2011), http://www.kegel.com/c10k.html

[35] Lighttpd web server, http://www.lighttpd.net/

[36] Leigh Griffin, Kieran Ryan, Eamonn de Leastar, Dmitri Botvich, *Scaling Instant Messaging Communication Services:A Comparison of Blocking and Non-Blocking techniques* (2011), http://repository.wit.ie/1636/

[37] Protocollo XMPP, http://xmpp.org/

[38] The Scala Programming Language, http://www.scala-lang.org/

[39] Rob von Behren, Jeremy Condit, Eric Brewer, *Why Events Are A Bad Idea (for high-concurrency servers)* (2003), http://static.usenix.org/events/hotos03/tech/full_papers/home/staff/alex/export/vonbehren/vonbehren_html/

[40] V8 Javascript engwinge, https://code.google.com/p/v8/

[41] Google Places, http://www.google.com/business/placesforbusiness/

[42] Arvind Narayanan, Solon Barocas, Vincent Toubiana, Helen Nissenbaum, Dan Boneh, *A Critical Look at Decentralized Personal Data Architectures* (22/02/2012)

[43] MongoDB – The Leading NoSQL Database, http://www.10gen.com/leading-nosql-database

[44] BSON, http://bsonspec.org/

[45] Laurent Bonnet, Anne Laurent, Michel Sala,Ḃenedicte Laurent, Nicolas Sicard , *REDUCE, YOU SAY: What NoSQL can do forData Aggregation and BI in Large Repositories* (2011)

8 Ringraziamenti

Desidero ringraziare mia madre Emilia e mio fratello Alex per avermi sostenuto in questo periodo e aver sopportato in silenzio la mia abitudine di camminare in casa per ore la notte.

Desidero ringraziare mia zia Gabriella per avermi aiutato nell'arduo compito di scrivere in italiano corretto.

Desidero ringraziare Michele Lambertini per avermi aperto le porte della facoltà ogni volta che ne avevo bisogno (ho il badge ancora smagnetizzato). Senza il suo supporto sicuramente non mi sarei laureato.

Desidero ringraziare anche il professor Fabio Vitali per avermi dato fiducia in molti aspetti della tesi, così da farmi lavorare su ciò che veramente mi piace.

Infine desidero ringraziare Alberto Vezzani per avermi dato la forza, con le sue lezioni di yoga, di completare questo lavoro.