UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

TESI DI LAUREA IN INFORMATICA

Analisi della scalabilità del linguaggio AbU per la programmazione di sistemi distribuiti basato su regole ECA

CANDIDATO

Alvise Bruniera

RELATORE

Prof. Marino Miculan

CORRELATORE

Arch. Michele Pasqua

Anno accademico 2021-2022

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

Sommario

GoAbU è un'implementazione del calcolo AbU (Attribute-based memory Updates) per programmare sistemi distribuiti di dispositivi IoT. In questa tesi ci poniamo l'obbiettivo di studiarne il comportamento all'aumentare del numero di nodi, sotto diversi tipi di carico, utilizzando il simulatore AbUsim. Quindi stabilire un metodo ragionevole e realizzare degli strumenti, per analizzare il comportamento di GoAbU. A questo scopo abbiamo realizzato alcuni programmi e script. In particolare `aeg` per generare parametricamente configurazioni per il simulatore, in modo da creare più facilmente configurazioni complesse con numero variabile di nodi, anche centinaia. Ed una serie di script per misurare il traffico di rete di ogni nodo (in termini di quantità di byte e pacchetti) ed il traffico riportato a livello di messaggi da uno dei nodi (quantità di byte e di messaggi), e per estrarre grafici e metriche utili da questi dati.
Abbiamo analizzato il traffico per 10, 50 e 100 nodi, sotto tre tipi di carico: nessun carico (idle), carico alto ma realistico (medio), carico molto alto (alto). In tutti i casi, con solo 10 nodi il carico è accettabile. Il traffico è risultato molto alto sia sotto carico medio che alto, in particolare, sotto carico alto, diventava problematico già da 50 dispositivi. Tuttavia, confrontando i risultati con il traffico riportato a livello di libreria da GoAbU, risulta che la compressione svolta dal middleware riesce a rendere il traffico molto minore di quanto potrebbe essere.

Indice

1

Introduzione

In questa tesi studieremo il comportamento di GoAbU, un'implementazione in Golang del calcolo AbU. Si tratta di un paradigma (e un linguaggio) per la programmazione di sistemi distribuiti. In particolare per grandi quantità di dispositivi IoT, visto che è pensato per essere leggero, completamente disaccoppiato nello spazio, ed adatto alla sincronizzazione ed alla diffusione di informazioni. Non avendo a disposizione centinaia di dispositivi, e non avendo nemmeno un sistema per distribuire la configurazione su questi dispositivi, abbiamo utilizzato il simulatore AbUsim. Un software per avviare GoAbU in dei container Docker, seguendo una configurazione, ed interagirci.
Ovviamente per studiarne il comportamento bisogna prima decidere un metodo sistematico e possibilmente automatizzato per analizzare GoAbU in modo ragionevole, quindi raccogliendo dati rilevanti per produrre metriche utili. Ovviamente in modo che sia riapplicabile ad una futura implementazione di AbU.

Per automatizzare l'analisi abbiamo prodotto alcuni script che si occupano di avviare dei test e raccogliere i dati dal sistema, da cui ricavare le metriche. Più precisamente, siamo interessati a metriche riguardanti il traffico di rete, come il numero di pacchetti o di byte scambiati tra i nodi. Per scoprire come un diverso numero di nodi od una diversa configurazione, impattano sulla rete a cui devono essere connessi. Alcune di queste sono ricavate dal sistema operativo, e altre sono ricavate direttamente dai log del simulatore.

Avevamo bisogno di un modo per creare automaticamente delle configurazioni che testassero le parti del sistema di cui abbiamo bisogno, ed avere una struttura di test su cui basarci. Abbiamo scritto il programma generatore di configurazioni `aeg`. Che costruisce una configurazione parametrica seguendo sempre una stessa struttura. In questo modo per provare una configurazione completamente diversa, che stressi un comportamento diverso, è sufficiente variare pochi parametri del generatore.

Abbiamo scelto di analizzare il comportamento di GoAbU nella gestione di tre tipi di carico diverso, uno nullo, uno medio ed uno molto alto. E verificato come il sistema scalava all'aumentare dei nodi, provandolo su 10, 50 e 100 dispositivi.

Sinossi La tesi sarà organizzata in 5 sezioni.

In Sezione 2 forniremo una descrizione del calcolo AbU e le sue caratteristiche. Quindi il suo paradigma ed i suoi punti di forza. Oltre ad una breve analisi delle sue limitazioni.

Segue, in Sezione 3 una descrizione del setting sperimentale, in cui descriveremo AbUsim, il sistema hardware utilizzato per i test, ed infine alcune problematiche incontrate di questa impostazione.

Nella Sezione 4 piegheremo nel dettaglio la struttura delle configurazioni di test prodotte automaticamente. Nonché la scelta delle configurazioni utilizzate nel benchmark e le caratteristiche di queste configurazioni.

In Sezione 5 descriveremo il metodo di raccolta dei dati dagli esperimenti. Ed anche le metriche prodotte con questi dati, analizzando più nel dettaglio quelle che abbiamo scoperto più rilevanti.

Per finire, in Sezione 6 se analizzeremo i risultati. Fornendo alcune nuove osservazioni riguardo l'implementazione GoAbU, ottenute grazie a questi test.

Come appendice A forniremo tutte le metriche prodotte.

2

Il calcolo AbU

AbU [3] (Attribute-based memory Updates) è un paradigma di calcolo distribuito basato sull'utilizzo di regole ECA (event-condition-action) per sincronizzare e scambiare informazioni tra grandi quantità di nodi.
Le regole non sono molto diverse da delle normali ECA: ogni regola è legata ad una o più variabili del nodo, e quando una di queste variabili viene modificata (da una regola o da un dispositivo di input), se una condizione specificata è verificata, viene eseguita l'azione. La caratteristica che distingue AbU è la possibilità di specificare regole "esterne", che agiscano sulle variabili degli altri nodi del sistema. Una regola esterna può specificare condizioni sia su variabili locali che esterne, mentre nelle azioni possono specificare solo variabili esterne. Un'azione avrà effetti su un nodo solo se possiede tutte le variabili esterne menzionate nella regola. Utilizzando queste regole, la memoria dei nodi viene aggiornata in base ad i suoi attributi (le variabili che possiede il nodo) da cui il nome.
In questo modo, è possibile programmare un sistema distribuito completamente disaccoppiato nello spazio, poiché l'effetto di un'azione sugli altri nodi dipende dai loro attributi e non richiede di conoscere la posizione delle variabili. Una regola interna AbU è della forma:

```
rule <name> on <variable> for <condition> do <action>
```

Ad esempio `rule turn_off on temperature for temperature > target do heating = false`, viene controllata al modificare della variabile `temperature`, e se la temperatura supera un `target`, spegne il riscaldamento impostando `heating = false`.
Una regola esterna, invece, è della forma:

```
rule <name> on <variable> for all <condition> do <action>
```

(notare `for all` al posto di `for`) e può accedere a variabili remote (non per l'attivazione). Ad esempio `rule set_target on target for all this.target < limit && ext.is_heater do ext.target = this.target`, viene attivata quando l'utente sposta la temperatura `target`, e se questo non supera un limite massimo, viene impostato quel target su tutti i nodi con la variabile `is_heater = true`.

Prima di proseguire forniamo un breve elenco dei principali punti di forza del calcolo AbU. Il paradigma del AbU permette di programmare facilmente sistemi di dimensioni variabili, anche grandi, diffondendo informazioni tra i nodi senza bisogno di conoscenze sul loro funzionamento interno o la topologia della rete. L'approccio ad attributi fornisce un modello di comunicazione basato interamente sul

ruolo del destinatario (quindi degli attributi che possiede), quindi più vicino alla semantica del sistema piuttosto che alla sua architettura. Ed infine, il design a regole permette di avere una knowledge base più semplice da leggere e da scrivere per il programmatore, quindi fornisce gli strumenti per produrre in breve tempo sistemi complessi.

Limitazioni del calcolo AbU Durante lo sviluppo dei test abbiamo incontrato due limitazioni del linguaggio, una delle quali facilmente risolvibile una volta definita la semantica di una funzionalità aggiuntiva, l'altra invece potrebbe eventualmente essere mitigata ma non risolta del tutto.
Il linguaggio non ammette che le regole esterne eseguano azioni sulle variabili locali. Questo è in parte perché se le regole impostassero una variabile locale al valore di una esterna, il valore finale della variabile locale non solo dipenderebbe dagli agenti esterni, ma anche dall'ordine in cui eseguono le azioni. Tuttavia, questo problema non sussiste se nel right-hand-side dell'assegnamento compaiono solo costanti e variabili locali. Tuttavia, permettere questa funzionalità (che aiuterebbe nella scrittura dei programmi) richiede di rivedere la semantica delle azioni esterne, per decidere in che momento va eseguita l'azione.
L'altro problema invece riguarda la consistenza del sistema. Per il CAP theorem [1], sappiamo che un sistema distribuito non può essere contemporaneamente "consistente", "resistente alla partizione" e "disponibile"; AbU è sia resistente che disponibile, quindi non si può ottenere consistenza, ed anche la consistenza eventuale sarebbe fuori dagli obbiettivi del paradigma. Tuttavia è particolarmente suscettibile al problema dei lost update. Nello specifico, se ad esempio un nodo `foo` possiede la variabile intera `x = 0` (e nessun altro nodo la possiede) n nodi causano (con successo) l'esecuzione di un'azione esterna del tipo `ext.x = ext.x + 1;`, il valore di `x` al termine dell'esecuzione delle azioni, sarà contenuto in $[0, n]$, ma l'effettivo valore dipenderà interamente dal momento in cui le n azioni vengono inserite nella coda degli eventi di `foo`. Come annunciato in precedenza non si può rendere AbU completamente consistente, quindi già a prescindere non è mai garantito che tutti gli update abbiano effetto. Tuttavia nella situazione attuale, anche quando la rete funziona correttamente, non è possibile implementare un contatore "affidabile" che venga incrementato da più nodi.
Entrambi questi problemi non rientrano nell'obbiettivo della tesi, ma sono venuti alla luce perché per la prima volta il calcolo AbU è stato adottato in un contesto applicativo.

3

Setting sperimentale

GoAbU è un'implementazione di questo modello di calcolo scritta in linguaggio Go. È costruita utilizzando la libreria memberlist come middleware per la comunicazione in gruppo e la scoperta dei nodi, Grule come rule engine, ed il framework gobot per renderla utilizzabile in ambito IoT.
Il focus principale della tesi sarà sia definire un metodo di test ragionevole e riproducibile per testare un'implementazione del calcolo AbU (e gli strumenti per eseguire test), che un'analisi di GoAbU stesso.

Protocollo di GoAbU in breve Ogni volta che viene attivata ua regola esterna, GoAbU inizia una "transazione". Una transazione è composta da una sequenza di messaggi ad alto livello inviati *in unicast* tramite memberlist. Questi messaggi conterranno il mittente, il tipo di messaggio (ad esempio `"do_commit"`, od `"interested"`), e dei dati (ad esempio, l'albero sintattico di un'azione da eseguire). Il messaggio è serializzato in `json` e successivamente in `base64`.
Una transazione si svolge in più fasi. Prima, il nodo che ha attivato la regola (l'iniziatore) invia un messaggio a tutti i nodi che conosce, per scoprire se sono interessati. Poi con un nuovo scambio di messaggi invierà il comando da eseguire sulle variabili a tutti i nodi interessati, che decideranno se abortire o eseguire il comando, ed informeranno l'iniziatore della decisione. Se un nodo abortisce viene comunicato anche agli altri, altrimenti viene comunicato di eseguirlo. Se un nodo non risponde entro un certo timeout, l'iniziatore ripete il messaggio.

3.1 Il simulatore AbUsim

I test sono stati eseguiti utilizzando il "simulatore" AbUsim. Quest'ultimo si basa sull'utilizzo di Docker ed è progettato sulla falsa riga di Docker Compose, ovvero preso in input un file `yaml` che descrive una serie di container, li avvia e mette in comunicazione. I file di configurazione di AbUsim permettono di specificare un elenco di dispositivi che eseguiranno GoAbU. Ogni dispositivo ha un suo elenco di variabili (che possono essere inizializzate arbitrariamente) ed un elenco di regole in linguaggio AbU. Per comodità è possibile specificare un prototipo in ogni dispositivo. I prototipi specificano anch'essi un elenco di regole e di variabili che i dispositivi erediteranno. In questo modo si possono raggruppare regole in comune tra tanti dispositivi. Si possono configurare anche il tick time (il tempo che il dispositivo aspetterà tra le esecuzioni delle regole) e l'immagine docker da usare per i nodi ed il coordinatore.

Il parsing delle regole di ogni nodo viene fatto direttamente dal simulatore, che dopo aver costruito l'elenco di regole e variabili di ogni nodo, codificherà gli alberi di sintassi in `json` e successivamente in `base64` per passarli come argomenti al programma in esecuzione su ogni nodo, proprio come sono codificati i messaggi. È possibile collegare altri nodi esterni alla rete del simulatore che eseguano GoAbU, ma non è necessario per questa tesi.
Viene avviato anche un coordinatore, che gestisce la comunicazione con l'esterno sia per inviare comandi che per ricevere log. Questo è strettamente parte del simulatore e non un elemento che ci si aspetta di trovare in una reale applicazione di AbU, che invece è pensato per essere decentralizzato. Va notato, allo scopo del progetto, che la rete virtuale su cui viaggiano i messaggi di GoAbU è separata da quella su cui viaggia la comunicazione di gestione del simulatore ed i log.
Opzionalmente si può avviare un container che offre un'interfaccia web per controllare i nodi, quindi permette di verificarne lo stato, inviare comandi, ed impostare il livello dei log. Tutto ciò può essere fatto anche programmaticamente tramite una libreria python.

3.2 Macchina dei test

Tutti i test sono stati svolti sulla stessa macchina fornita dall'Università degli studi di Udine con le seguenti specifiche:

- Precessore: Intel Xeon Gold 6238R (56 core, 2.195 GHz)
- RAM: 64 GiB
- OS: Debian 11
- Kernel: Linux 5.10.0-18-amd64

A causa di un problema secondario (affrontato nella sezione successiva) è stato necessario aumentare la dimensione massima della tabella ARP. Per sicurezza è stata impostata al suo valore massimo di 2^{31}, ma sarebbe stata sufficiente anche una dimensione molto minore.

3.3 Problemi secondari

Durante lo sviluppo del generatore di test abbiamo incontrato alcune limitazioni od errori del simulatore. Questi errori sono indipendenti da GoAbU, e riguardano puramente il funzionamento del simulatore, quindi verranno trattati solo brevemente.

Limite alla dimensione della configurazione Come già spiegato nella sezione precedente, la configurazione di un nodo (quindi l'elenco delle variabili e delle regole) viene codificato in un'unica lunga stringa, e questa viene passata al nodo come argomento del comando shell. Poiché la dimensione del vettore `argv` è limitata (la dimensione effettiva dipende dal sistema operativo, ma solitamente si parla di 128 KiB) anche la quantità e la lunghezza delle regole. 128 KiB potrebbero sembrare molti, ma anche se fossero passate in plaintext significherebbe che (supponendo le regole siano lunghe tendenzialmente 128 caratteri) si possono avere solo 1024 regole, quando l'obbiettivo sarebbe quello di ammettere diverse

migliaia di regole. In realtà la codifica attualmente utilizzata per la configurazione, è meno efficiente e limita ulteriormente la quantità di regole, in modo dipendente dalla complessità sintattica di ogni regola. Una soluzione proposta e testata in un fork del simulatore prevede di non passare la configurazione come argomento, ma attraverso il buffer di input, che invece non è limitato. Questa soluzione funziona e può essere implementata semplicemente, ma non è l'unica possibilità. Alla fine, per i test non è stato necessario nessun cambiamento, perché la quantità di regole era sufficientemente bassa.

Loop nel logger Una delle difficoltà incontrate non dipende dal simulatore, ma ha permesso di scoprire un problema del coordinatore (che tra le altre cose si occupa di raccogliere i log dai nodi). La dimensione massima per le tabelle ARP impostata di default su molti sistemi operativi non è sufficiente a permettere a più di ~ 40 nodi di comunicare, per poter testare più dispositivi bisogna impostare un limite più alto dal sistema operativo. Tuttavia, per come funziona il simulatore, quando un nodo non riesce a collegarsi agli altri si chiude dopo un certo tempo. Ed in questo modo abbiamo scoperto che se uno nodo smette di rispondere alla richiesta di nuovi log, un thread del coordinatore rimane bloccato in un loop, utilizzando al 100 % una delle cpu. Si tratta di un problema marginale poiché si verifica solo quando uno dei nodi termina inaspettatamente, cosa che non dovrebbe succedere se non per problemi della setting come questo limite alle tabelle ARP.

Ricompilazione delle regole L'ultimo problema, invece, riguarda una funzionalità mancante della libreria GoAbU ma necessaria per l'applicazione nel mondo reale di AbU. Un rule engine è efficiente a trovare le regole da eseguire perché utilizza una struttura dati complessa, ma per costruirla deve prima eseguire una computazione complessa. E Grule non fa eccezione, costruire la struttura dati dal set di regole è costoso, quindi avviando il simulatore su una grande quantità di regole (non necessario per questa tesi) potrebbe passare molto tempo prima che i nodi siano completamente avviati. Ovviamente per il simulatore non si può fare molto, ma per un dispositivo reale che viene programmato una volta sola con un migliaio di regole, non è accettabile aspettare diversi minuti ad ogni riavvio perché sta ricostruendo la struttura. Soprattutto considerato che un dispositivo del genere potrebbe essere molto più lento di un computer che esegue AbUsim, e quindi potrebbe diventare troppo lento ancora prima. Per risolvere questo problema bisognerebbe aggiungere una funzionalità per memorizzare la struttura precompilata in modo che successivamente il dispositivo debba solo leggerla per tornare operativo dopo un riavvio. Per il simulatore questo non vale poiché lo scopo di AbUsim è la prototipazione, ed è previsto che ad ogni avvio la configurazione cambi, rendendo meno utile conservare la struttura.

4

Generazione dei test

4.1 Il generatore aeg

Uno degli strumenti principali utilizzati per l'analisi è il generatore di test `aeg` (AbUsim Example Generator). Volevamo un modo per poter generare configurazioni che stressassero una caratteristica a piacere di GoAbU. L'utilizzo di questo strumento ha permesso sia di provare agilmente grandi configurazioni, ad esempio quelle da migliaia di regole per dispositivo che hanno permesso di scoprire i lunghi tempi di compilazione (Sottosezione 3.3); che di apportare rapidamente correzioni alle configurazioni selezionate come test cases, e di poter generare facilmente configurazioni uguali che variassero solo nel numero di nodi.

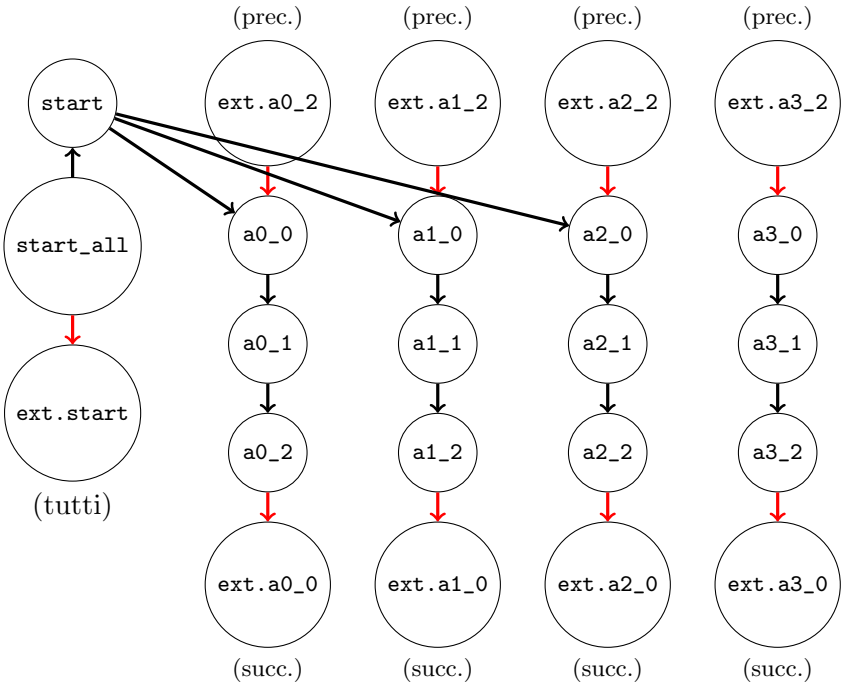

Figura 4.1: catene di variabili interne (regole esterne in rosso)

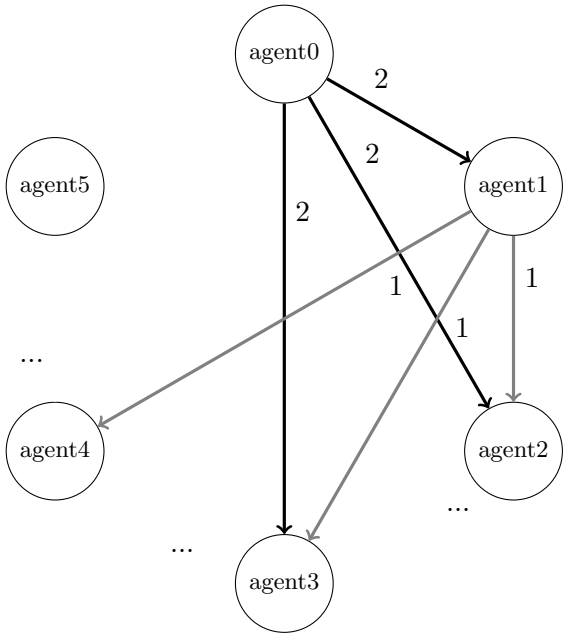

Figura 4.2: Attivazione dei nodi successivi

Struttura delle configurazioni Le configurazioni generate con `aeg` seguono tutte una stessa struttura, che permette di avviare una computazione che coinvolgerà tutti i nodi e terminerà sempre, i dettagli della configurazione sono controllati da 6 parametri (interi maggiori di 1), indicati con a, b, c, d, e, ed f.

Tutti gli a nodi di una configurazione sono simili tra loro: anno le stesse variabili ed eseguono regole uguali, che differiscono al massimo per alcuni valori hardcoded. Ogni nodo ha due variabili booleane `start` e `start_all` inizializzate a `false` che sono usate per avviare una computazione, più $b \times c$ (altro su b e c più avanti) variabili intere con nome `ai_j` con $i \in [0, b)$ e $j \in [0, c)$ (e.g.: `a0_0`), tutte inizializzate a 0. Queste variabili rappresentano b "catene", ciascuna lunga c variabili, come rappresentato in Figure 4.1. Ogni nodo ha anche una variabile intera `id`, ciascuna inizializzata ad un valore diverso e mai modificata, sono utilizzate solo per indirizzare il singolo nodo.

Ogni dispositivo ha 3 "regole di avvio" (rappresentate alla sinistra in Figure 4.1), una si attiva quando `start_all` viene impostato a `true`, e non fa altro che impostare la variabile `start` a `true` su tutti i nodi. Una resetta `start_all`. Ed infine quando `start` viene impostato a `true`, una "variabile iniziale" (la variabile `ai_0`) delle prime $d \leq b$ catene viene impostata a f, e la variabile `start` viene resettata. Notare che `start_all` non attiva *tutte le catene* (infatti non attiva nessuna catena, ma setta `start`), invece attiva le catene di *tutti i nodi*.

Alle prime $c - 1$ variabili di ogni catena (quindi da 0 $c - 2$) corrisponde una regola (rappresentate in nero in Figure 4.1) che si attiva quando la sua variabile viene impostata ad un valore maggiore di 0, e contemporaneamente copierà il valore della variabile in quella successiva nella catena (e.g.: da `a1_2` ad `a1_3`) e resetterà la sua variabile. Quando la c-esima variabile dell'i-esima catena viene impostata ad un valore $k > 0$, non c'è una corrispondente regola "normale". Invece ce ne sono una locale che resetta il valore della variabile a 0, ed una esterna (rappresentate in rosso in fondo a Figure 4.1) che setterà a $k - 1$ il valore della variabile iniziale dell'i-esima dei successivi e dispositivi (il dispositivo `id` attiva quelli da $((\texttt{id} + 1) \mod a)$ ad $((\texttt{id} + e) \mod a)$). In questo modo la parte di computazione sul dispositivo

corrente è completata, e continua sui dispositivi successivi, come visibile in Figure 4.2. Dopo f livelli, il valore che viene passato sulle catene è arrivato ad 1, e ai prossimi dispositivi viene passato il valore 0, che le regole del $f + 1$-esimo livello di dispositivi ignoreranno.

Parametri di configurazione Quindi il comportamento della configurazione è deciso dai (già elencati nella descrizione) parametri:

- a (alt.: devices-number) Il numero di nodi nella configurazione
- b (alt.: chains-number) Il numero di catene in ogni nodo (non tutte saranno attivate automaticamente)
- c (alt.: chain-length) La lunghezza di ogni catena
- d (alt.: chain-width) Il numero di catene attivate automaticamente con `start=true`
- e (alt.: devices-width) Il numero di dispositivi attivati al termine di una catena
- f (alt.: devices-length) Il numero di livelli attraversati prima di arrivare a 0

Oltre a questi 6 parametri di configurazione, se ne possono impostare anche alcuni riguardanti il setting dell'esperimento, come l'immagine Docker da usare per i nodi ed il coordinatore, oppure il tick. Per questa tesi sono rilevanti solo i 6 parametri di configurazione, in quanto il setting sperimentale (tempo di tick, immagini Docker, etc.) è rimasto lo stesso in ogni test.

4.2 Terminazione forte

Per semplicità affronteremo la terminazione in due parti. La prima parte riguarda solo la parte "interna" della computazione, quella che si occupa di trasportare i valori nelle catene di variabili, ignorando i valori che potrebbero essere aggiunti da una regola esterna avviata da un altro nodo. E la seconda riguardante solo i dispositivi che vengono attivati dalle regole esterne, assumendo che la parte interna termini correttamente, attivando i nodi successivi. Sarebbe possibile fornire una dimostrazione unica che tratti direttamente entrambe le parti della computazione, ma è molto più difficile.

Stabilizzazione interna La terminazione sarà dimostrata prima in maniera meno formale, considerando un sistema in cui la regola da eseguire viene scelta dando la precedenza quelle che scrivono le variabili più vicine alla fine di una catena, e dando la precedenza alle catene con indice più alto. Questo permette di fissare l'ordine di esecuzione delle regole in uno su cui è più comodo ragionare. E infine sarà accennato perché un ordine così stringente non è necessario. Inoltre assumeremo, come annunciato ad inizio sezione, che nessun nodo esterno imposti il valore di alcuna variabile, tuttavia in seguito spiegheremo perché non sia un problema nel paragrafo sulla convergenza totale.

Sia $(a_{0,0}, ..., a_{0,c-1}, a_{1,0}, ..., a_{1,c-1}, ..., a_{b-1,c-1}) \in \mathbb{N}^{bc}$ il vettore che rappresenta lo stato interno di un generico nodo della configurazione (ogni variabile $a_{i,j}$ rappresenta la variabile `ai_j`). È facile verificare che ogni regola normale viene attivata da uno stato del tipo $(..., a_{i,j}, a_{i,j+1}, ...)$ e causa uno stato del tipo $(..., 0, a_{i,j}, ...)$ (notare che l'indice della catena è sempre i) e causerà l'inserimento di una nuova regola

nella coda degli eventi. Ricordiamo che con l'ordinamento stabilito ad inizio paragrafo, la regola attivata dallo stato $(..., a_{i,j}, a_{i,j+1}, ...)$ verrà eseguita solo quando $a_{i,j+1} = 0$, altrimenti ci sarebbe una regola con precedenza più alta. Una regola finale, invece è causata da uno stato del tipo $(..., a_{i,j}, a_{i+1,0}, ...)$ e ne causa uno del tipo $(..., 0, a_{i+1,0}, ...)$ senza causare nessun inserimento.

Sapendo che: la computazione termina se e solo se il vettore che rappresenta lo stato interno è $(0, ..., 0)$, altrimenti ci sarebbero ancora regole in coda, attivate quando sono state impostate le variabili diverse da 0; e che ogni regola causa uno stato di ordine lessicografico minore rispetto a quello che lo ha attivato; possiamo concludere che la computazione deve terminare per forza altrimenti potrebbe scendere di ordine lessicografico all'infinito. Ed al termine di questa computazione, saranno state eseguite le regole esterne per attivare i nodi successivi, perché le uniche regole che non aggiungono nuove regole sono attivate dallo stesso stato che causa l'attivazione delle regole esterne.

AbU, per sua natura, non fa garanzie sull'ordine di esecuzione delle regole causate dallo stesso stato, quindi per completare la dimostrazione formale dobbiamo mostrare che funzionerebbe anche senza questo ordinamento. Possiamo vedere che non importa perché: esiste una corrispondenza diretta tra le variabili e le regole, e non ci sono "interferenze" tra le regole causate dallo stesso stato (non cercano di leggere la stessa variabile o di scrivere la stessa), ed è sufficiente entrare una volta in uno stato ed uscirne subito per attivare una regola. Quindi l'ordine non importa, e quindi la computazione interna termina sempre.

Stabilizzazione esterna Come annunciato all'inizio della sezione, affronteremo la stabilizzazione esterna considerando corretta la parte interna della computazione. Ovvero assumendo che nodo che viene "attivato" da un valore k, eventualmente attiverà e nodi con un valore $k - 1$. Questo segue dalla dimostrazione precedente.

Consideriamo l'attivazione di un nodo come il passaggio di un token, ogni token ha un'"intensità" k, ovvero il numero di livelli che può attraversare. Quando un nodo elabora un token, questo viene spezzato in e token di intensità $k - 1$, e quando un token ha intensità 0, il nodo che lo riceve lo scarta. Nelle nostre computazioni, l'intensità del token corrisponde al valore k con cui il nodo viene attivato, ovvero il valore che la regola finale esterna inserisce all'inizio della catena di variabili dei nodi successivi. Quando viene impostata `start = true` su di un nodo, questo genera b token di intensità f. Quando viene impostata `start_all = true` su un qualsiasi nodo, tutti gli a nodi generano b token ciascuno di intensità f.

Sia $(t_f, t_{f-1}, ..., t_1) \in \mathbb{N}^f$ il vettore dei token, dove ogni variabile t_k rappresenta la quantità di token di intensità k in circolazione sulla rete. Notiamo che i token di intensità 0 non vengono contati, perché sono scartati subito. Possiamo osservare che ogni volta che un token di intensità k viene consumato, il vettore passa da $(..., t_k, t_{k-1}, ...)$ a $(..., t_k - 1, t_{k-1} + e, ...)$. Quindi ad ogni consumo, il nuovo vettore ha ordine lessicografico più basso di quello iniziale.

Sapendo che la computazione esterna si ferma se e solo se non ci sono più token in circolazione, quindi il vettore ha valore $(0, ..., 0)$, e che ad ogni avanzamento della computazione l'ordine lessicografico scende, sappiamo che la computazione deve terminare per forza.

Ammettendo che più token vengano consumati contemporaneamente, il risultato non cambia, perché si passerebbe comunque ad uno vettore più in basso nell'ordine lessicografico.

Stabilizzazione totale Per mettere insieme i due risultati manca solo dimostrare che ammettere attivazioni esterne non impedisce la stabilizzazione interna. Riuscendo a dimostrare questo, avremmo che internamente la computazione termina in ogni caso, ed esternamente termina se quella interna termina.

Dimostrare questa non interferenza è facile utilizzando lo stesso ordine rigido che abbiamo usato per rendere più breve la dimostrazione della stabilizzazione interna. Con quell'ordine rigido, le regole esterne che attiverebbero il nodo catena non verrebbero mai selezionate per prime per l'esecuzione, perché scrivono sulle variabili più lontane dalla fine della catena. In questo modo, il nodo comincerebbe ad eseguire le regole di attivazione solo dopo aver terminato la computazione interna già in corso. Quindi l'attivazione non interferirebbe con la stabilizzazione.

Ovviamente AbU non prevede questo ordinamento, ma come spiegato nel paragrafo sulla stabilizzazione interna, imporre questo ordinamento non influisce sul comportamento della nostra computazione.

4.3 Scelta delle configurazioni

Per l'analisi di GoAbU sono stati scelti tre casi di test di base, e poi ogni caso è stato testato con 10, 50 e 100 nodi, ciascuno dei tre vuole mettere in evidenza diversi aspetti di AbU.

In tutti i casi, l'unico dato che ci importava raccogliere erano le informazioni sul traffico di rete. Nella prossima sezione 5.1 spiegheremo perché questa scelta.

Idle Il primo caso non prevede nessuna regola, il sistema è composto solo da nodi "vuoti" ed i test in questo caso (a contrario degli altri due) sono stati condotti senza avviare mai la computazione. L'obbiettivo è quello di osservare il comportamento dell'implementazione quando non succede nulla, verificare la presenza di overhead o di un "rumore di fondo" da poter rimuovere dai test successivi.

In particolare ci interessava il traffico causato dal middleware memberlist per mantenere la comunicazione di gruppo attiva anche senza nessuna transazione.

Questo test serve per scoprire se un sistema con tanti nodi sia impraticabile a prescindere dal tipo di lavoro che svolgono. Se mantenere i nodi connessi causa già troppo traffico per poter utilizzare GoAbU in una rete vera, bisognerebbe cambiare completamente l'implementazione.

- $a = 10|50|100$
- $b = 1$
- $c = 1$
- $d = 1$
- $e = 1$
- $f = 1$

Medio Progettando questo test l'obbiettivo era quello di ricreare un test che anche se esigente, fosse in qualche modo realistico, un test in cui ogni nodo ha una certa quantità di regole, ed alcune (non tutte) di queste si attivano. Ed una frazione di queste causano degli effetti su un solo altro nodo (non troppi).

L'obbiettivo era non essere troppo pesante sulla rete, ma bilanciare il carico tra rete e nodo interno. In un certo tempo, ogni nodo avrà molte più regole locali di cui occuparsi, e dovrebbe utilizzare di meno la rete.

Questo test, come anche il successivo, è stato progettato in modo che richieda 500 s per terminare la computazione. Ed effettivamente con pochi dispositivi è così, ma in realtà l'overhead dell'esecuzione del rule engine, e soprattutto quello della rete, fanno sì che in realtà richieda più tempo. Il tempo reale e dipendente dal setting sperimentale, quindi dalla macchina che esegue il simulatore, e non direttamente da GoAbU. Tuttavia, il fatto che con 100 dispositivi la macchina sia in difficoltà, significa che potrebbero esserlo anche i dispositivi di rete.

In ogni caso, abbiamo osservato che il tempo di esecuzione è molto consistente tra le ripetizioni della computazione. Quindi dopo aver provato una volta ad eseguire una computazione con un certo numero di nodi, possiamo assumere che la prossima impieghi un tempo molto simile.

- $a = 10|50|100$
- $b = 10$
- $c = 5$
- $d = 5$
- $e = 1$
- $f = 20$

Nodi	Regole	
	Interne	Esterne
10	5000	1000
50	25000	5000
100	50000	10000

Alto L'ultimo test è stato progettato con l'obbiettivo esplicito di mettere in difficoltà la rete quanto più possibile in un tempo limitato. Come si vede dalla configurazione, sono presenti molte meno regole interne rispetto a quelle esterne. Invece ogni dispositivo attiva direttamente altri 3 dispositivi e si prosegue in 5 livelli di profondità. In questo modo il numero di messaggi inviati nella rete cresce in modo quasi esponenzialmente nel tempo.

Questo test è stato costruito in cercando una combinazione di e ed f tali che con pochi dispositivi la computazione termini in 500 s. Però il carico è molto alto e quando si arriva a 100 dispositivi, la computazione viene completata in 2000 s. Come detto in precedenza, questo tempo non è significativo poiché dipende principalmente dalla macchina su cui viene eseguito il simulatore. Però una differenza del genere suggerisce che anche il traffico di rete sarà particolarmente alto.

- $a = 10|50|100$
- $b = 1$
- $c = 1$
- $d = 1$
- $e = 3$
- $f = 5$

Nodi	Regole	
	Interne	Esterne
10	1210	3630
50	6050	18150
100	12100	36300

5

Raccolta dati

5.1 Quali dati raccogliere

Prima di affrontare qualsiasi tipo di test bisogna stabilire quali sono i dati che importano e come raccoglierli in modo efficace. In un sistema distribuito sono tanti gli aspetti che si potrebbero prendere in considerazione. Traffico di rete, latenza dell'esecuzione delle regole, throughput dei singoli nodi, throughput dell'intero sistema. Tutte queste opzioni sono state considerate prima di scegliere che test eseguire.

AbU non è pensato per svolgere calcoli, ma per sincronizzare dispositivi. Quindi in un'applicazione reale non c'è un vero risultato in output, un qualcosa di cui misurare il throughput. Al massimo si possono misurare la quantità di regole eseguite, ma quelle sono legate al tick che è impostato manualmente.

La latenza delle regole anzitutto dipende più dal numero totale di regole, e dalle prestazioni del dispositivo che esegue GoAbU, che da GoAbU stesso, quindi anche questa metrica non è molto indicativa delle prestazioni. In ogni caso, per migliorarla bisognerebbe passare ad un rule engine più "aggressivo", o trovare un modo di utilizzare meno regole. In ogni caso, il tempo speso per attendere il passare del tick è molto più lungo della latenza di una regola, a meno che la knowledge base non sia enorme. Quindi anche questo dato è stato considerato non necessario, almeno per ora.

Alla fine, abbiamo deciso di raccogliere solo i dati relativi al traffico di rete. Più precisamente abbiamo raccolto sia la dimensione totale dei messaggi inviati e ricevuti (come numero di byte totali), che la quantità di pacchetti (frame di livello 2) inviati e ricevuti. Entrambi questi dati sono reperibili direttamente dal sistema operativo. Inoltre, abbiamo deciso di raccogliere anche il numero di messaggi inviati a livello di libreria, e la dimensione di questi messaggi. Per poterle confrontare con i dati del sistema operativo e scoprire quando overhead viene aggiunto dal middleware. Queste ultime informazioni sono state raccolte solo per gli ultimi due casi, quello medio e quello alto, perché in idle la libreria non produce azioni.

5.2 Ispezione dei nodi

Le informazioni sul traffico di rete sono fornite direttamente dal sistema operativo, che offre degli pseudo-file per processo dove sono listate diverse metriche (in particolare byte e pacchetti, inviati e ricevuti) per ogni interfaccia di rete, utilizzate fino a quel momento dal processo.

Come già visto in precedenza 3.1 AbUsim utilizza due reti virtuali (quindi ogni nodo ha due interfacce di rete), una utilizzata per la comunicazione di GoAbU, ed una per il resto delle funzioni del simulatore, come l'invio dei log. Quindi è facile ottenere solo i dati riguardanti GoAbU senza che siano influenzati dal simulatore. È sufficiente identificare prima l'interfaccia di rete utilizzata per GoAbU, sempre usando gli pseudo-file, e poi raccogliere solo le metriche di quell'interfaccia.

Per separare i dati riguardanti una computazione da quelli precedente e da quelli della discovery, raccogliamo i dati prima e dopo la computazione, ed teniamo la differenza.

I dati sui messaggi, invece, sono raccolte tramite i log del simulatore, che sono stati modificati appositamente per includere questi dati. Purtroppo, per quanto riguarda i messaggi, abbiamo raccolto i log di un solo nodo poiché le dimensioni erano troppo grandi. Raccogliendo i dati di un solo nodo sono stati prodotti 230 MiB di file di log, quindi raccoglierli per tutti i nodi sarebbe stato dispendioso. In futuro, si potrebbe trovare un modo per fornire questi dati in modo più snello, per poter analizzare più nodi.

5.3 Metriche

Elenchiamo tutte le metriche ottenute, con una breve descrizione

- `received bytes`: Indica il numero totale di byte ricevuti da un agente in un test
- `received packets`: Indica il numero totale di pacchetti ricevuti da un agente in un test
- `sent bytes`: Indica il numero totale di byte inviati da un agente in un test
- `sent packets`: Indica il numero totale di pacchetti inviati da un agente in un test
- `received bytes per seconds`: Come sopra, ma diviso per la durata del test
- `received packets per seconds`: Come sopra, ma diviso per la durata del test
- `sent bytes per seconds`: Come sopra, ma diviso per la durata del test
- `sent packets per seconds`: Come sopra, ma diviso per la durata del test
- `bytes ratio`: Rapporto tra byte ricevuti ed inviati
- `packets ratio`: Rapporto tra pacchetti ricevuti ed inviati
- `bytes per packet`: Dimensione in byte dei pacchetti
- `total bytes`: Byte inviati in totale nella rete
- `total packets`: Pacchetti inviati in totale nella rete

- `total bytes per seconds`: Come sopra, ma diviso per la durata del test
- `total packets per seconds`: Come sopra, ma diviso per la durata del test
- `logs_expected bytes`: Dimensione totale dei messaggi riportata nei log
- `logs_actual bytes`: Byte effettivamente inviati dal nodo loggato
- `logs_expected packets`: Numero di messaggi riportati nei log
- `logs_actual packets`: Pacchetti effettivamente inviati dal nodo loggato
- `logs_bytes ratio`: Rapporto tra bytes effettivamente inviati e quelli riportati nel log
- `logs_packets ratio`: Rapporto tra pacchetti inviati e messaggi riportati nel log

Le metriche riguardanti il tempo sono state raccolte, ma non sono rilevanti, perché il tempo dipende principalmente dal tick, e nel nostro caso anche dalla macchina che esegue il simulatore. Quindi non verranno analizzati.

Le metriche riguardanti il traffico ricevuto, sono interessanti, ma non particolarmente utili. Ogni dato ricevuto è un nodo che era stato inviato da un altro nodo, quindi i dati inviati in totale e quelli ricevuti dovrebbero essere simili. Dal rapporto possiamo ottenere il packet loss della rete, però questo non fornisce nessuna informazione sul comportamento di GoAbU, ma sul comportamento delle reti Docker.

I dati sui pacchetti di livello 2 sono interessanti in quanto sono indicativi del lavoro che dovranno gestire i dispositivi di rete a cui AbU sarà connesso.

Confrontare i dati del sistema con quelli dei log permette di osservare cosa il middleware è in grado di fare per ridurre il traffico. In particolare, i due rapporti `logs_bytes ratio` e `logs_packets ratio` facilitano questo confronto. Più precisamente, il primo mette in evidenza la capacità del middleware di comprimere ed ottimizzare i dati. Il secondo invece mette in evidenza quanti pacchetti servono per un messaggio.

Oltre a questo, le metriche totali, quindi `total bytes` e `total packets`, forniscono una misura complessiva del traffico, quindi la quantità di byte e pacchetti, che attraversa la rete da parte di tutti i nodi. Questo è uno dei dati più importanti, perché potrebbe essere il fattore più limitante in una rete reale. Se il data-rate fosse una limitazione, sarebbe sufficiente allungare il tick dei nodi che generano più traffico, ma la quantità totale di dati resta la stessa.

Bisogna notare che i pacchetti sono di livello 2, quindi oltre a quelli causati da GoAbU sono inclusi anche altri protocolli ancillari IP e le richieste ARP, perché il sistema non li separa. Per quanto sarebbe interessante avere un dato più preciso, questi pacchetti “non GoAbU” non saranno molti se paragonati a quelli di GoAbU. Inoltre, si tratta di un dato più indicativo dello stress che viene messo sulla rete, in particolare sugli switch ed access point.

6

Risultati

6.1 Grafici

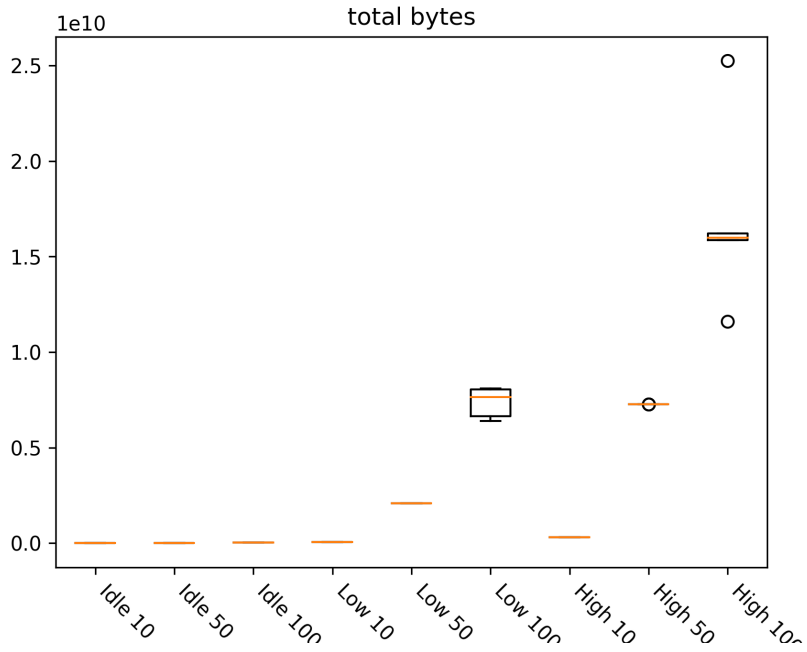

Figura 6.1: Byte inviati in totale nella rete

`total bytes` In questo grafico possiamo osservare il traffico totale che attraversa la rete, in byte.

Notiamo anzitutto che in idle il traffico è molto basso, come ci si aspetta. Significa quindi che il middleware non ha bisogno di grandi scambi di dati per mantenere la comunicazione di gruppo ed il gossiping. All'aumentare dei nodi passa da $\sim 2\,\text{MB}$ a $\sim 9\,\text{MB}$, a $\sim 20\,\text{MB}$, di traffico, l'aumento sembra approssimativamente lineare rispetto al numero di nodi. I test in idle sono gli unici dove ha senso guardare le metriche al secondo, e si tratta di pochi kB/s (grafico in appendice, Figure A.11).

I risultati per i test sotto carico medio sono molto meno promettenti. Già con 10 nodi, i valori sono ben oltre il carico in idle per 100 nodi, con $\sim 62\,\text{MB}$ di dati.
Più preoccupante è il salto tra i 50 ed i 100 nodi, in cui il traffico passa da $\sim 208\,\text{MB}$ a $\sim 7.4\,\text{GB}$.

L'aumento non sembra affatto lineare, ma non siamo sorpresi, perché implementando il broadcast come messaggi unicast ad ogni nodo, ci *aspettiamo* una message complexity di $\Theta(a^2)$.

I risultati per i test sotto carico alto sono pessimi. Ci aspettavamo di vedere un traffico molto alto, però un traffico di $\sim 302\,\text{MB}$ con soli 10 nodi, rende AbU inadatto all'utilizzo in una rete realistica per problemi così pesanti sulla rete. Ovviamente si tratta di un benchmark con un carico estremo e poco realistico già per cominciare, ma AbU vuole essere utilizzabile anche in sistemi molto grandi.

Inaspettatamente, all'aumentare dei nodi passa a $\sim 7.3\,\text{GB}$ e poi $\sim 16\,\text{GB}$, che *sembra* molto più vicina ad una crescita lineare rispetto a quella del carico medio. Per indagare ulteriormente su questo risultato inaspettato. Considerando anche il traffico per regola esterna otteniamo:

Test	kB/rule
`low 10`	~ 62
`low 50`	~ 104
`low 100`	~ 736
`high 10`	~ 83
`high 50`	~ 401
`high 100`	~ 467

Notiamo che, per 10 e soprattutto 50 nodi, il traffico per regola causato dal carico alto è molto maggiore di quello sotto carico medio. Con 100 nodi, invece, il traffico per regola sotto carico alto è molto più vicino al caso con 50 nodi e molto più basso ($\sim 36.7\,\%$ in meno) del traffico per regola sotto carico medio. Sembra che ad un certo punto aumentare le regole cominci a ridurre i costi per la singola regola.

Nonostante questo risultato promettente, c'è un outlier del test con 100 nodi che ha prodotto $\sim 25\,\text{GB}$ di traffico, significa che in determinate condizioni potrebbe causare molto più traffico. Il traffico per regola in quel caso è di $689\,\%$, che è comunque più basso del test sotto carico medio. Ad ogni modo, anche se la singola regola è meno costosa, nel complesso i dati passati nella rete sono aumentati di molto.

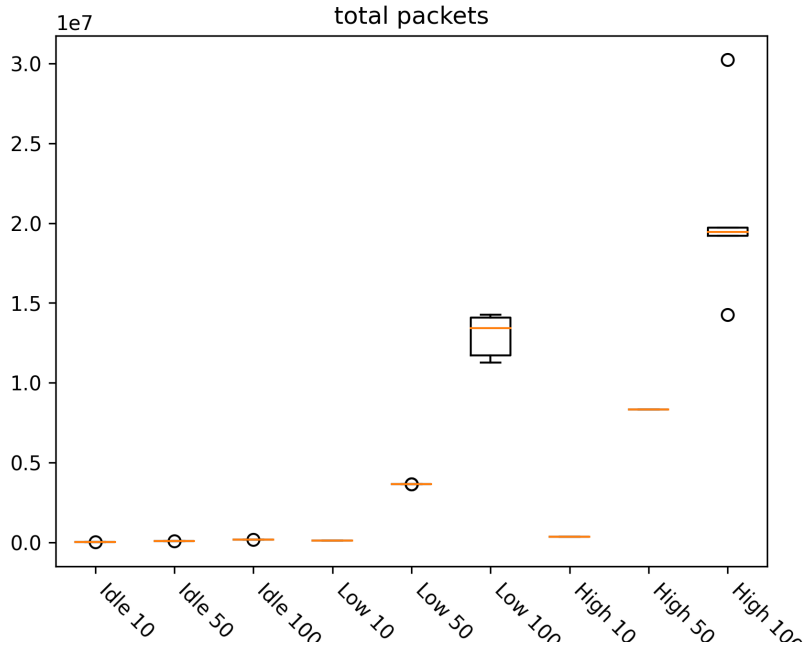

Figura 6.2: Pacchetti inviati in totale nella rete

`total packets` Questo grafico, per i test sotto carico medio ed alto, non fornisce particolari informazioni oltre a quelle messe in evidenza dal grafico `total bytes` Figure 6.1.

Dai pacchetti risulta chiaro che la message complexity (stavolta come numero di pacchetti e non dimensione) non sia lineare. Infatti, il numero di pacchetti nei test con 100 nodi è anche 100 volte più grande di quello nei test con 10 nodi, mentre con 50 nodi è 30 volte più grande.

Per i test in idle, a contrario di quanto visto nel grafico precedente, anche senza carico, il numero di pacchetti cresce in modo non lineare, ma comunque meno di come cresce sotto carico. In seguito vedremo altri dati su questo argomento quando parleremo del grafico `bytes per packets` Figure 6.5.

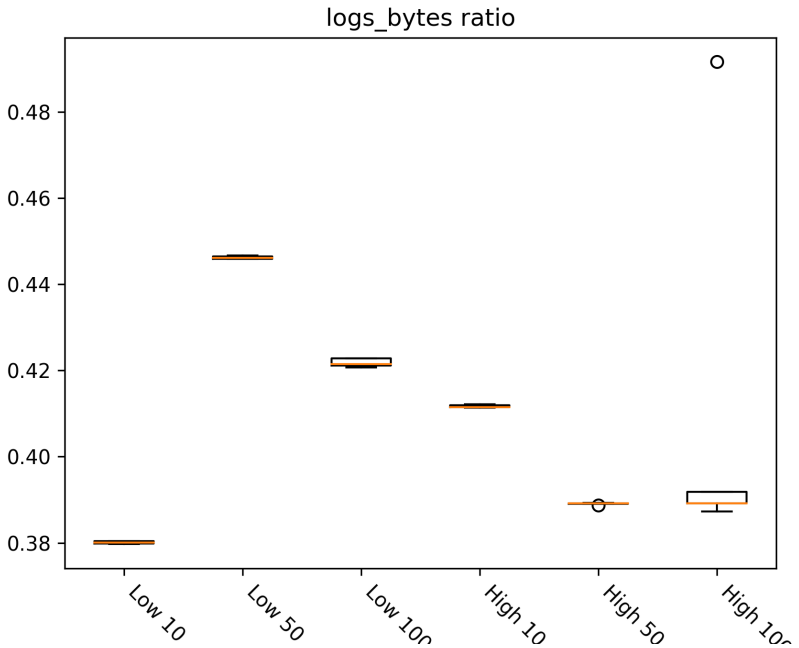

Figura 6.3: Rapporto tra bytes effettivamente inviati e quelli riportati nel log

`logs_bytes ratio` In questo grafico possiamo osservare se l'utilizzo del middleware memberlist causa overhead o comprime i dati in modo osservabile. È calcolato confrontando la quantità byte dati inviati riportata dal sistema, con la somma della dimensione di tutti i messaggi eseguite a dalla libreria. Quindi un valore maggiore di 1 indica un complessivo overhead, mentre un valore minore di 1 indica una compressione migliore.

Possiamo osservare dai rapporti con valori da 0.38 a 0.46, che la compressione riesce a rendere i dati da inviare meno della metà dei dati di partenza. Il rapporto di compressione è tra il 2.2 ed il 2.6, che è molto alto.

Come già visto nella Sezione 3, il corpo dei è una serializzazione `json` che contiene l'albero sintattico della regola codificata in `base64`. La stringa risultante, passata al middleware per l'invio è molto lunga, ma avendo poche informazioni (relativamente alla lunghezza) ci aspettiamo che abbia poca entropia, e che per questo possa essere compressa così tanto, così facilmente.

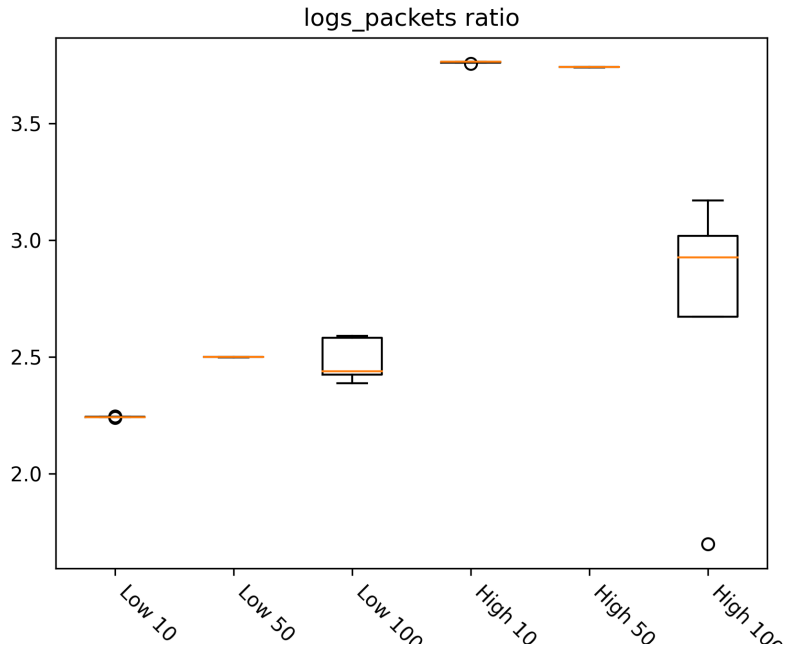

Figura 6.4: Rapporto tra pacchetti effettivamente inviati e messaggi riportati nel log

`logs_packets ratio` Questa metrica mette in evidenza quanti pacchetti sono *inviati* (non scambiati in totale), approssimativamente, per ogni messaggio inviato. Se il rapporto fosse minore di 1 probabilmente saremmo in presenza di un'errore nella raccolta dei dati, a meno che il middleware non faccia ottimizzazione del recapito utilizzando una rete overlay, ma nel nostro caso non è così.

Istintivamente verrebbe da pensare che il numero di pacchetti sia influenzato dai nodi, che causano un aumento dei pacchetti per mantenere il gruppo. Però dal grafico osserviamo che il numero di nodi influenza solo marginalmente il numero di pacchetti inviati, anzi nel caso con carico alto, la quantità di pacchetti diminuisca all'aumentare dei nodi.

Possiamo vedere che il numero di pacchetti per messaggio è influenzato principalmente dal tipo di carico. Comunque, anche sotto carico alto, sono meno di 4 pacchetti per messaggio.

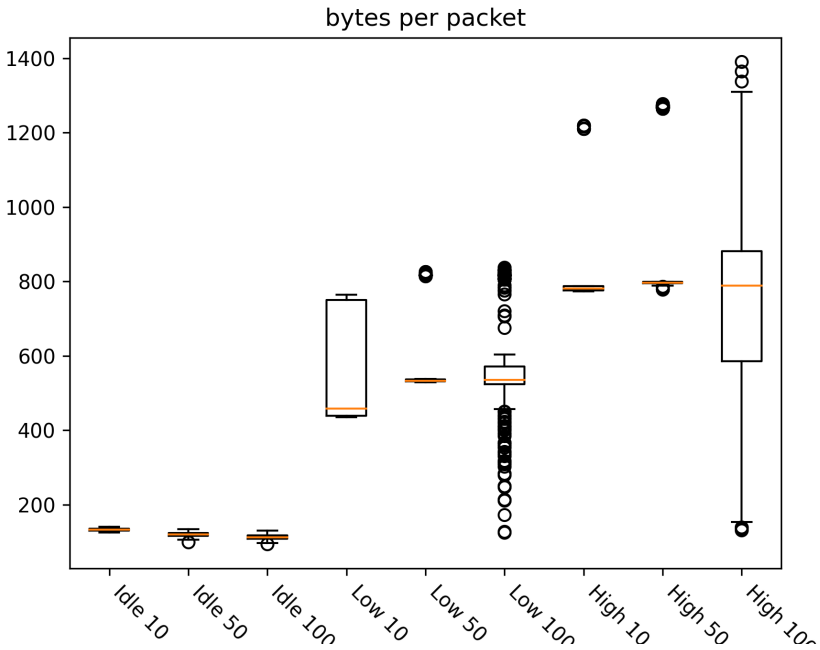

Figura 6.5: Dimensione in byte dei pacchetti

`bytes per packet` In questo grafico è rappresentata la dimensione media dei pacchetti, ottenuta dividendo i byte per il numero di pacchetti inviati da ogni nodo per ogni computazione.

Analizzando il grafico `total packets` 6.2 abbiamo notato che il numero di pacchetti inviati in idle sembrava crescere più che linearmente rispetto al numero di nodi; quando invece la quantità di dati in `total bytes` Figure 6.1 sembrava quasi lineare. Osservando questo grafico, possiamo vedere che la dimensione dei pacchetti diminuisce all'aumentare dei nodi, come ci si aspetta dai due dati precedenti.

Notiamo che come il numero di pacchetti per transizione, anche la dimensione dei pacchetti è influenzata principalmente dal tipo di carico. Notiamo anche che nessuno dei carichi fa raggiungere consistentemente la `MTU` (che è di 1500 B).

Nonostante fosse così anche nelle altre metriche, in questa risulta molto evidente che all'aumentare dei nodi, i risultati diventano sempre più imprecisi. Alcuni nodi ricevono più o meno traffico. Anche se i messaggi sono in broadcast e dovrebbero essere distribuite tra tutti i nodi. Probabilmente si tratta di un'effetto del gossiping probabilistico in una rete molto grande.

6.2 Conclusioni e proposte di miglioramento

Dopo aver analizzato questi risultati, possiamo concludere che GoAbU non riesce a scalare come sperato. La libreria riesce a mantenere un costo per regola non troppo alto quando si arriva a grandi quantità di regole esterne.Ma il traffico complessivo rimane troppo alto (vedi Figure 6.1 e Figure 6.2).

Il middleware riesce ridurre ulteriormente il traffico grazie ad un qualche meccanismo di compressione (vedi Figure 6.3), questo permette a GoAbU di scalare leggermente di più. Comunque, una codifica più efficiente dei messaggi permetterebbe di ridurre l'affidamento che facciamo sulla compressione. Ma per permettere veramente a GoAbU di scalare su reti più grandi, la soluzione non è ridurre la dimensione dei messaggi.

Possiamo imputare il problema al modello di broadcast implementato, in cui ogni nodo contatta ogni altro nodo in unicast, seguendo il modello "reliable/atomic broadcast" [2]. Potremmo migliorare le prestazioni utilizzando anche a più basso livello la comunicazione broadcast, invece che simularlo in una rete unicast. Però questo richiede di modificare la semantica del linguaggio per quanto riguarda la decisione della consegna delle regole. Inoltre, in questo modo, non solo tutti i segmenti della rete (inclusi quelli che non usano GoAbU) riceverebbero i pacchetti di GoAbU, ma bisognerebbe gestire in modo speciale i nodi che non si trovano nella stessa LAN.

Sfruttando le assunzioni già esistenti sulla rete, in particolare completezza della rete e fair loss, si potrebbe scegliere di limitare ad numero di nodi k a cui viene inoltrato il messaggio di broadcast e fare maggiore affidamento sul gossiping. Ovviamente è necessario scegliere opportunamente questo k in modo che il broadcast resti affidabile, e magari si può rendere configurabile per adattarsi alle necessità di sistemi specifici.
In questo modo si eviterebbe di modificare troppo il modello, pur riducendo anche di molto la message complexity del sistema.

Si può ottenere un risultato migliore utilizzando algoritmi di broadcast efficiente, basati magari sull'utilizzo di una rete overlay non completa e sull'inoltro dei messaggi. Un metodo del genere può ridurre di molto il traffico, ma ad un costo, sia di latenza, che soprattutto computazionale. E ricordiamo che per l'utilizzo in campo IoT è necessario trovare un algoritmo che possa essere utilizzato anche da dispositivi molto piccoli.

Ad ogni modo, il paradigma del calcolo AbU ha delle grandi potenzialità, grazie alla sua semplicità di programmazione. Ed è particolarmente adatto alla coordinazione di grandi sistemi, fornendo un modo semplice di diffondere informazioni. Anche se GoAbU non è ancora adatto a scalare, è solo la prima implementazione, e ci sono ancora molte opzioni per migliorare.

A

Altri grafici

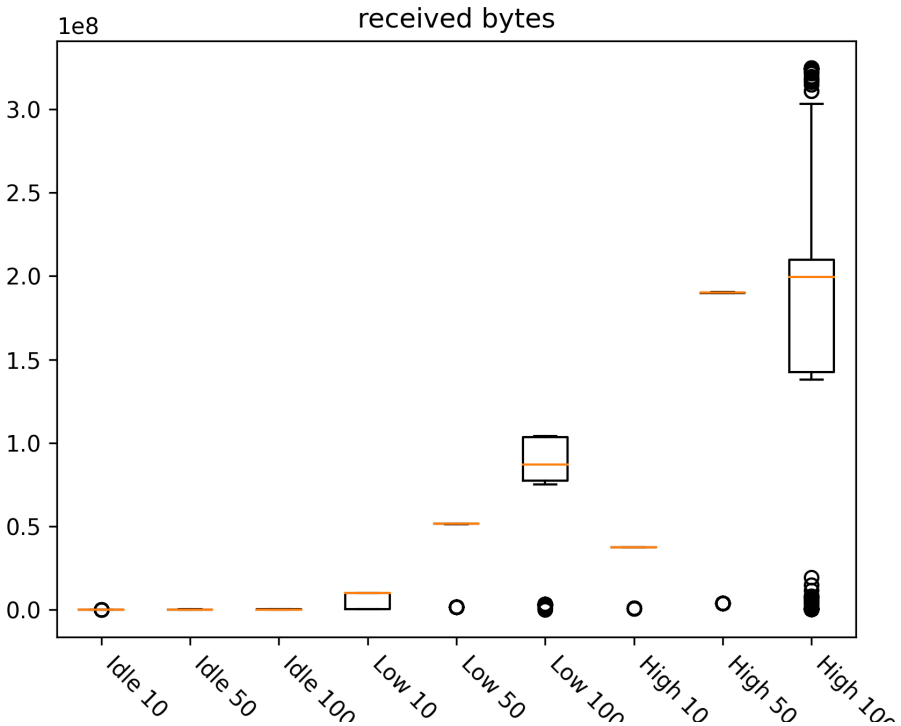

Figura A.1: Indica il numero totale di byte ricevuti da un agente in un test

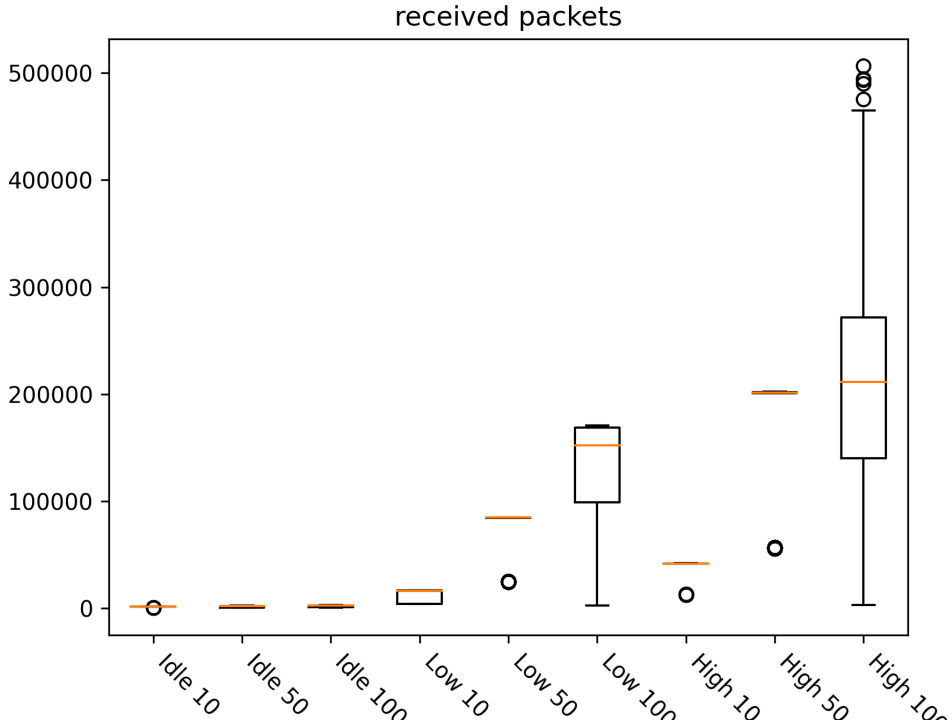

Figura A.2: Indica il numero totale di pacchetti ricevuti da un agente in un test

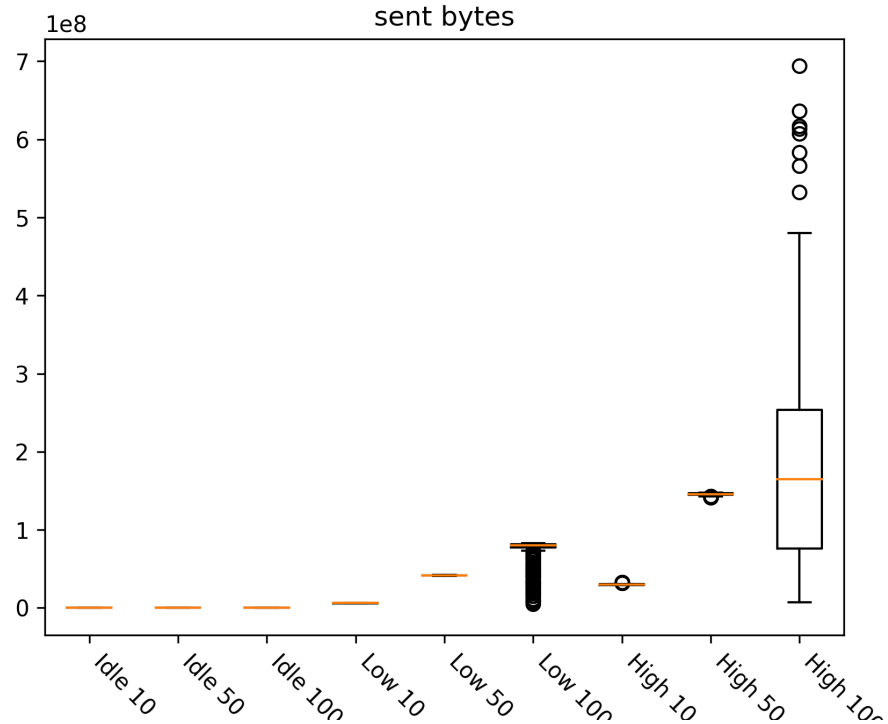

Figura A.3: Indica il numero totale di byte inviati da un agente in un test

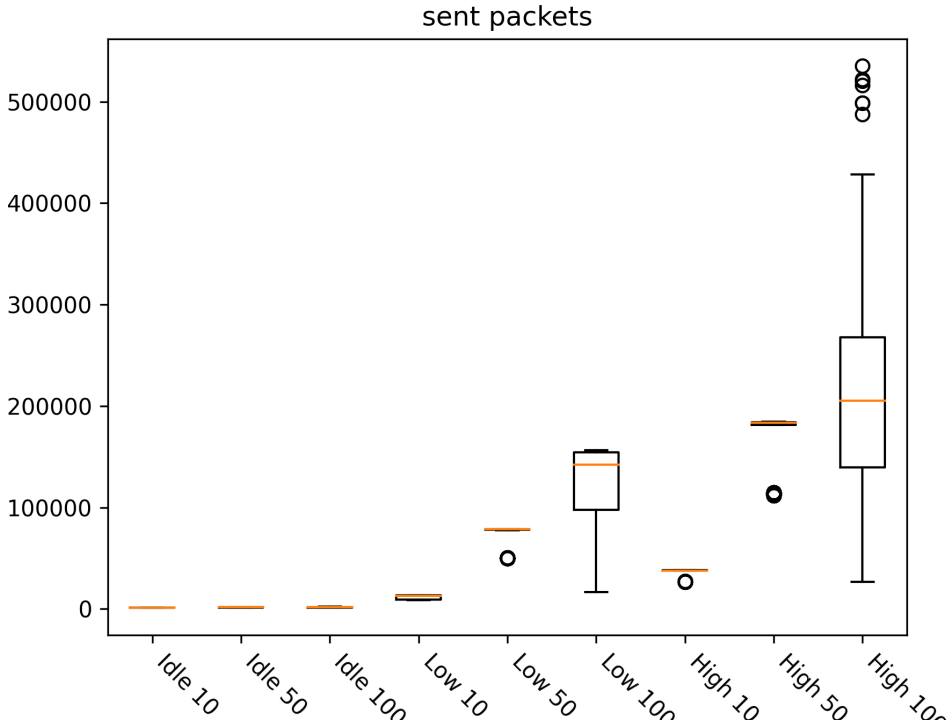

Figura A.4: Indica il numero totale di pacchetti inviati da un agente in un test

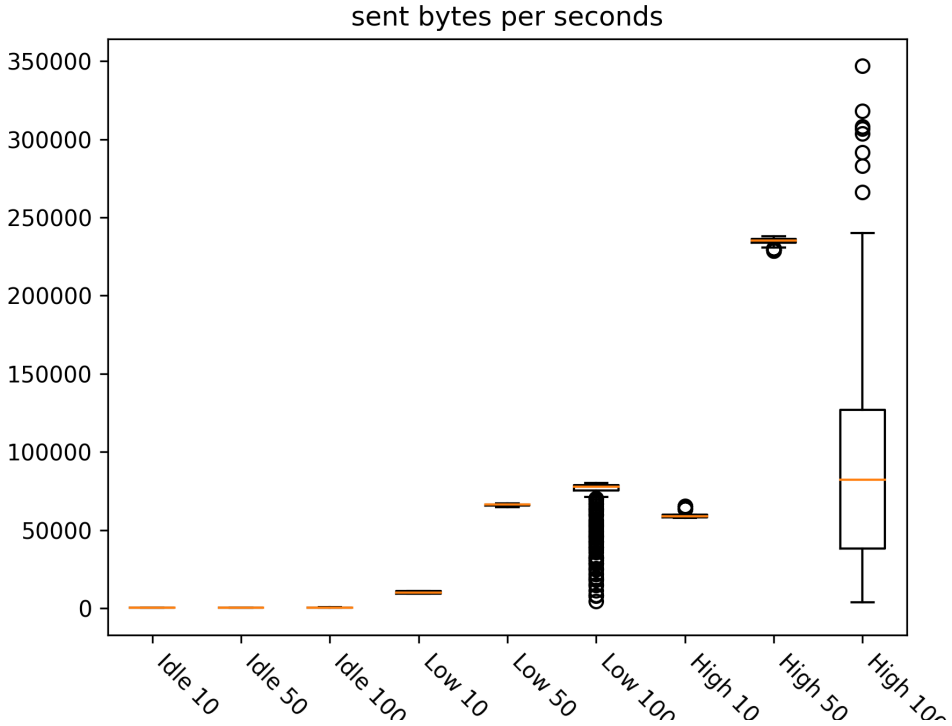

Figura A.5: Come sopra, ma diviso per la durata del test

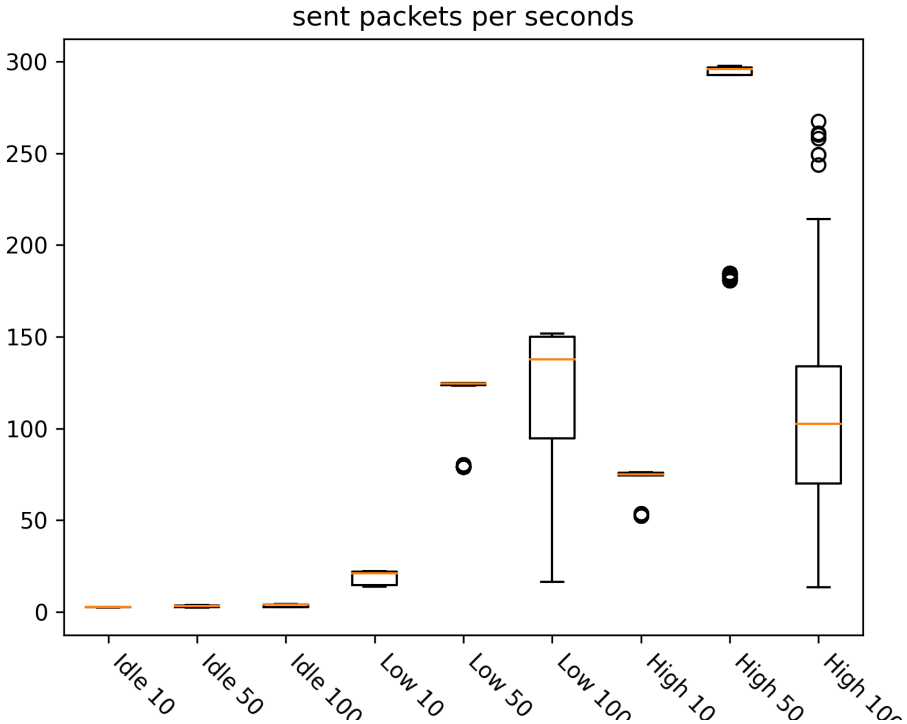

Figura A.6: Come sopra, ma diviso per la durata del test

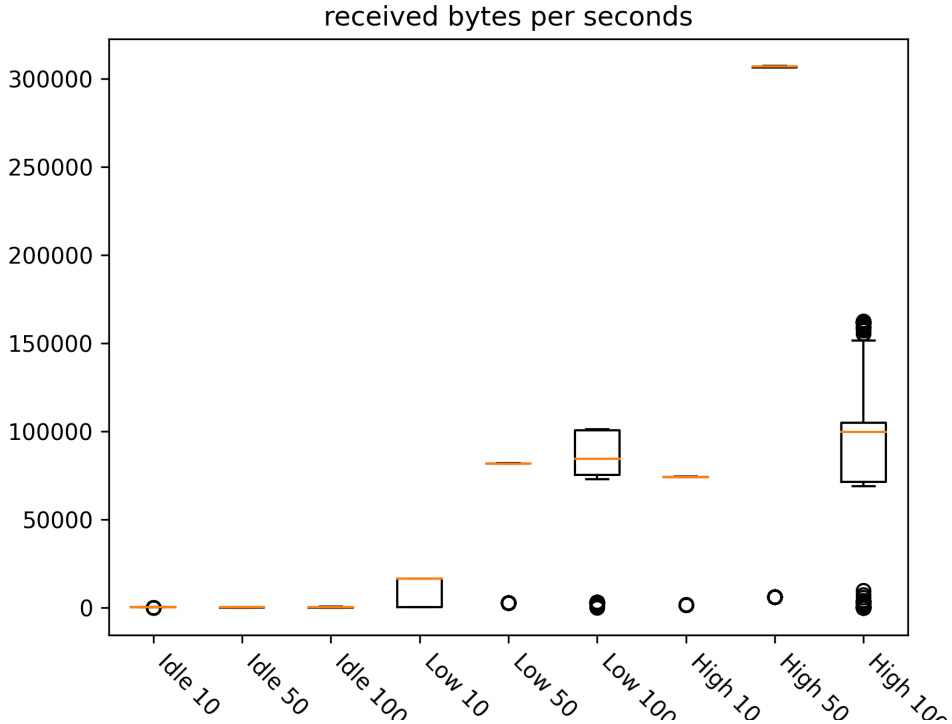

Figura A.7: Come sopra, ma diviso per la durata del test

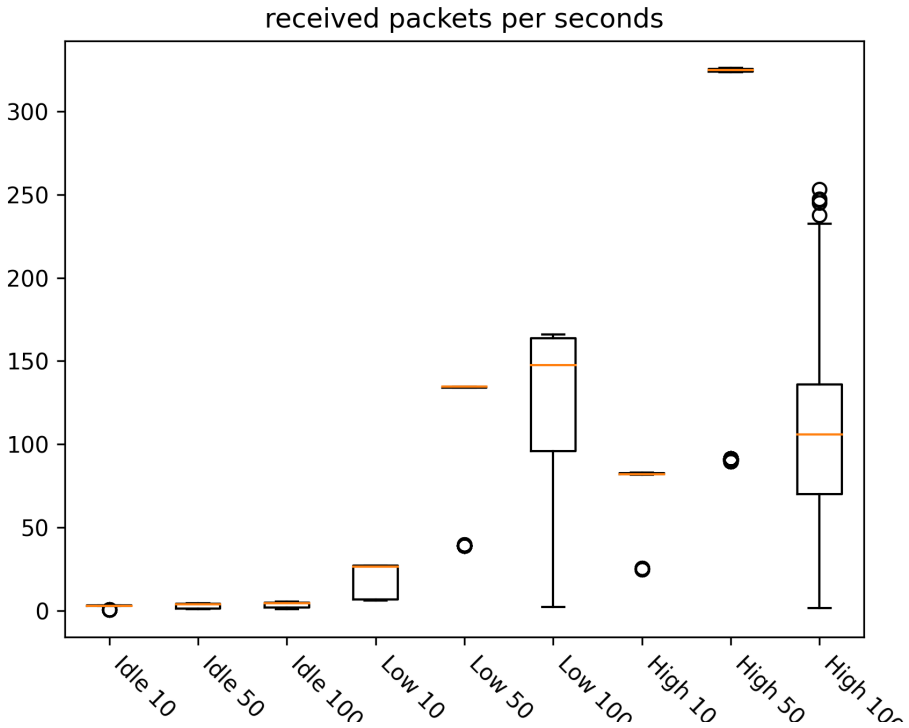

Figura A.8: Come sopra, ma diviso per la durata del test

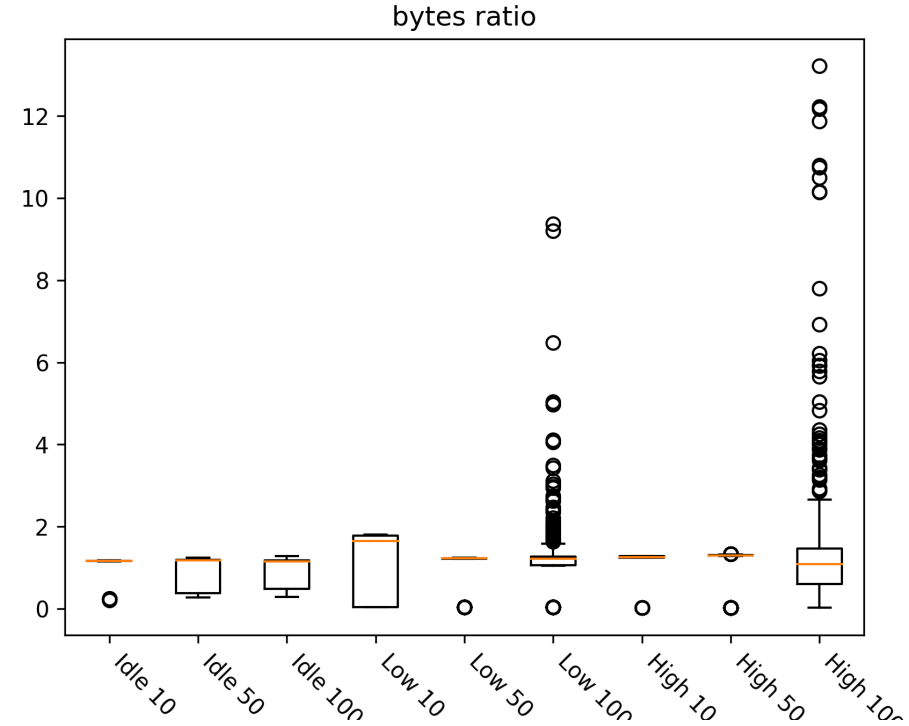

Figura A.9: Rapporto tra byte ricevuti ed inviati

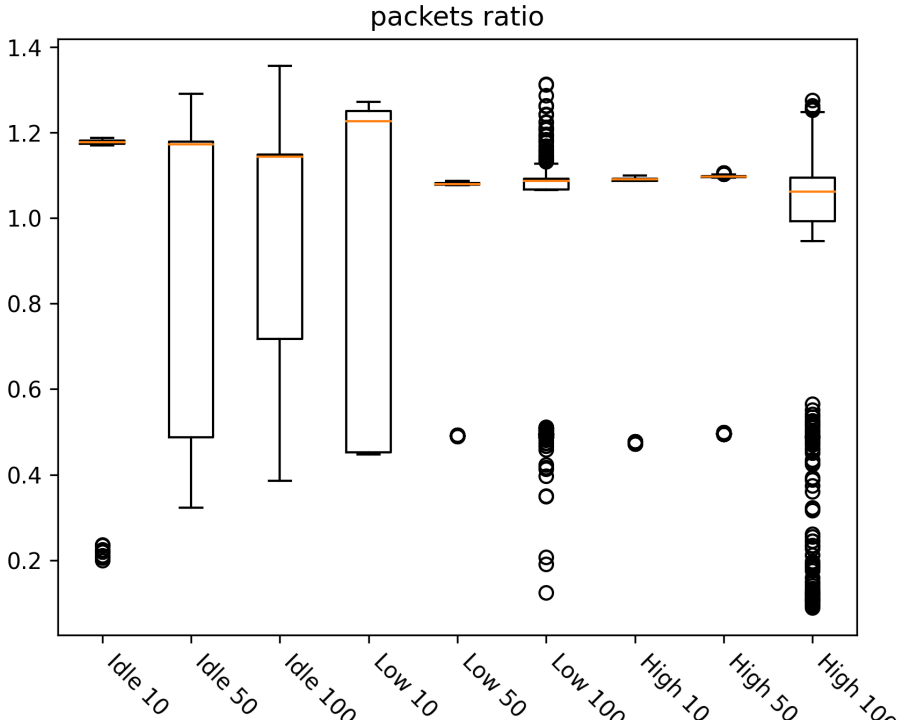

Figura A.10: Rapporto tra pacchetti ricevuti ed inviati

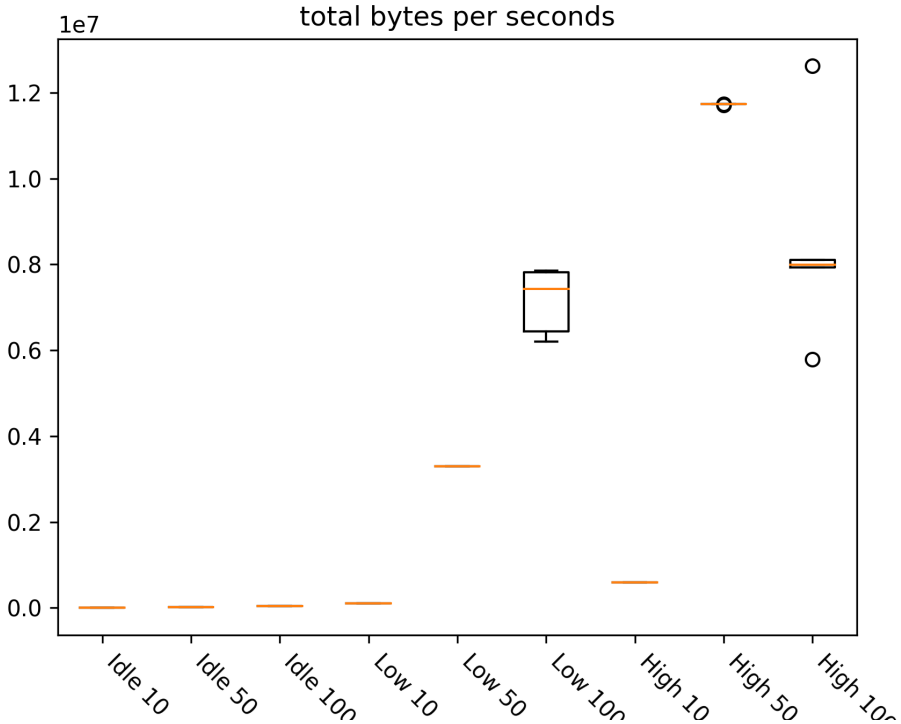

Figura A.11: Come sopra, ma diviso per la durata del test

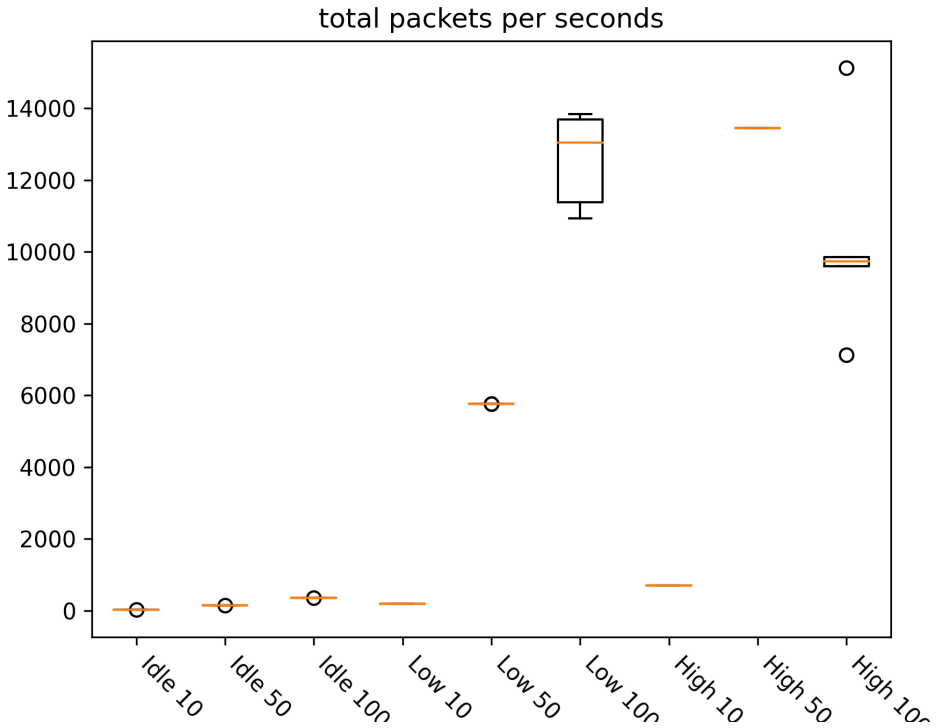

Figura A.12: Come sopra, ma diviso per la durata del test

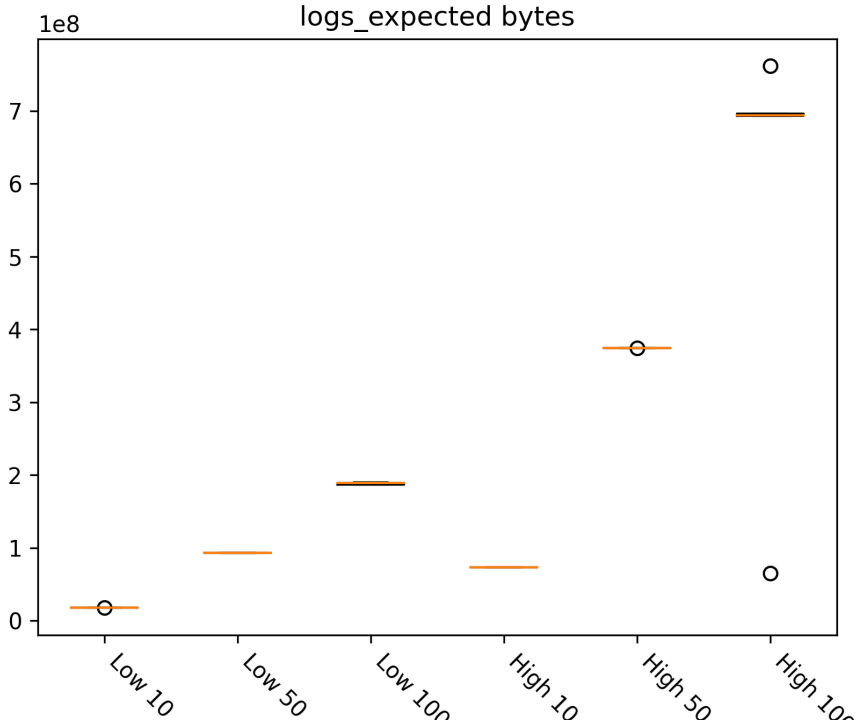

Figura A.13: Dimensione totale dei messaggi riportati nei log

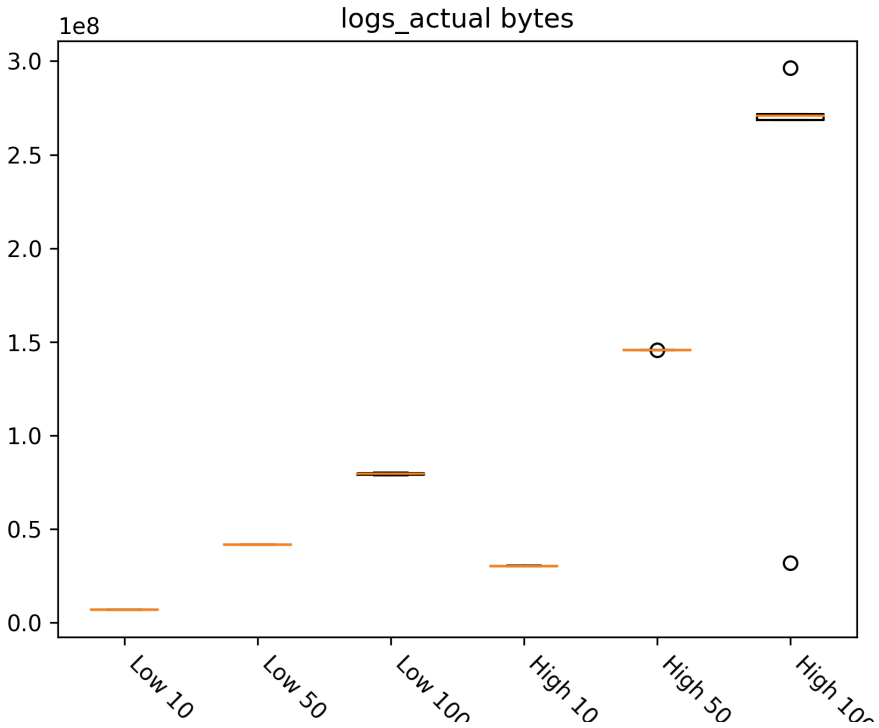

Figura A.14: Byte effettivamente inviati dal nodo loggato

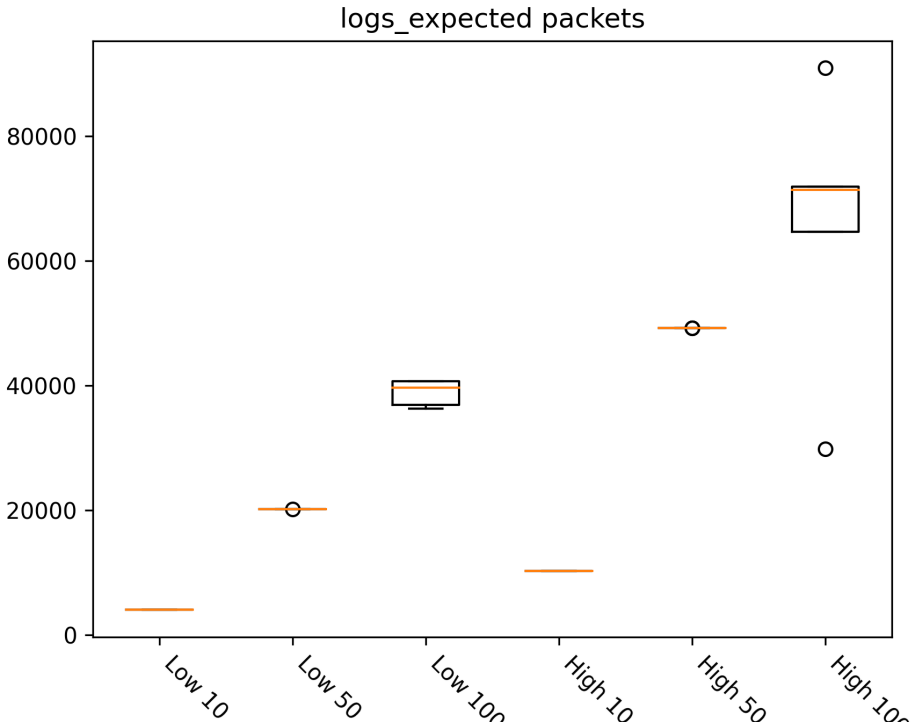

Figura A.15: Numero di messaggi riportati nei log

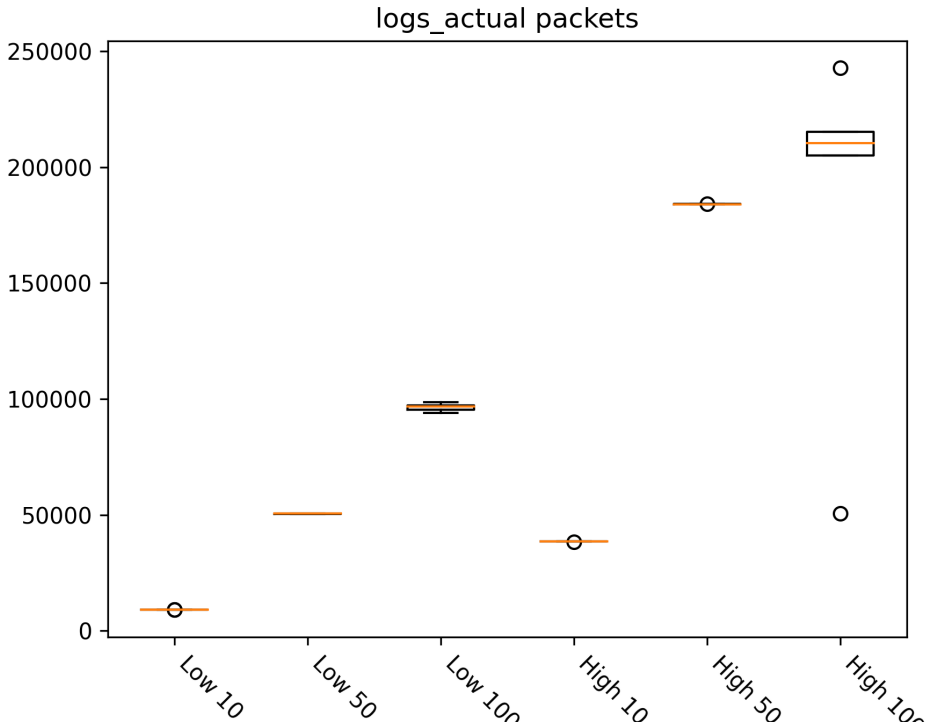

Figura A.16: Pacchetti effettivamente inviati dal nodo loggato

Bibliografia

[1] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pp. 343477–343502. Portland, OR, 2000.

[2] Thomas A Joseph e Kenneth P Birman. Reliable broadcast protocols. Relazione tecnica, 1989.

[3] Marino Miculan e Michele Pasqua. A calculus for attribute-based memory updates In *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Nur-Sultan, Kazakhstan, Sep. 6-10, 2021, Proceedings*. A cura di Antonio Cerone e Peter Ölveczky, Lecture Notes in Computer Science. Springer, 2021.