Università degli Studi di Pisa

Facoltà di scienze matematiche, fisiche e naturali

Corso di studi in Informatica

Monitoraggio di sistemi connessi in rete usando dati di telemetria

Candidato: Fulvio Schiano

Relatore: Luca Deri

Anno accademico 2017-2018

Introduzione

Questa tesi tratta le limitazioni del paradigma pull utilizzato da molti protocolli di gestione di sistemi di rete quali SNMP, proponendo un approccio basato sul modello pull. Tale modello usa dati di telemetria che inviati in modo costante, permettono di ottenere vantaggi rispetto al modello pull.

Successivamente sono analizzati alcune tool per l'invio di messaggi utilizzati per implementare il paradigma pool: viene discuso che cosa sono, e cosa fa di un sistema di messaggistica un buon sistema. elencando i vari problemi e le possibili soluzioni.

Sono analizzati nel dettaglio quattro sistemi di messaggistica open source maggiormente utilizzati nel mercato, nonchè confrontati tra di loro. in base alle caratteristiche importanti per la consegna di dati di telemetria

Il resto della tesi tratta il progetto vero e proprio, che consiste nella creazione di un tool per il monitoraggio remoto di sistemi tramite telemetria. Viene spiegato in dettaglio il disegno e l'implementazione e come le sue prestazioni cambiano in base all'utilizzo dee 4 i sistemi di messaggistica.

Infine viene validato il lavoro analizzando la sua performance e tempi di risposta, confrontando i dati ottenuti e valutando i vantaggi e svantaggi delle varie soluzioni.

1. Sommario

1. Introduzione

La raccolta dei dati per l'analisi e la risoluzione dei problemi è sempre stata un aspetto importante nel monitoraggio dello stato di una rete. [1] Meccanismi come SNMP, CLI e Syslog per raccogliere dati da una rete hanno limitazioni che riguardano l'automazione e la scalabilità. Una limitazione è l'uso del modello pull, in cui la richiesta iniziale di dati dagli elementi di rete proviene dal client.Il modello di pull non è scalabile quando c'è più di una Network Management Station (NMS) nella rete. Con questo modello, i server forniscono i dati solo quando i client li richiedono. Per avviare tali richieste, è necessario un intervento manuale continuo che rende il modello pull inefficiente. Gli indicatori di stato della rete, le statistiche di rete e le informazioni sull'infrastruttura sono esposti al livello dell'applicazione, dove vengono utilizzati per migliorare le prestazioni operative e ridurre i tempi di risoluzione dei problemi. Un modello push utilizza questa funzionalità per inviare continuamente dati alla rete e notificare il client . La telemetria abilita il modello push, che fornisce accesso quasi in tempo reale ai dati di monitoraggio.

1.1. La Telemetria

La telemetria è nata dalla domanda "Come possiamo ottenere più dati dal router il più velocemente possibile in un modo che ne renda facile l'utilizzo?" Come primo passo, dovevamo eliminare le inefficienze associate a un meccanismo di polling come SNMP. Gli operatori di rete eseguono periodicamente il polling perché desiderano i dati a intervalli regolari. Quindi, perché non inviarli solo i dati che vogliono quando lo desiderano e saltare il sovraccarico dei sondaggi? Così è nata

l'idea di "streaming". Invece di estrarre i dati dalla rete, siediti e lascia che la rete ti spinga.[2]

La telemetria fornisce un meccanismo per selezionare i dati di interesse dai router e trasmetterli in formato strutturato alle stazioni di gestione remota per il monitoraggio. Questo meccanismo consente la sintonizzazione automatica della rete in base a dati in tempo reale, che è cruciale per il suo funzionamento senza interruzioni. Una granularità più fine e una maggiore frequenza dei dati disponibili tramite telemetria consentono un migliore monitoraggio delle prestazioni e, pertanto, una migliore risoluzione dei problemi. Aiuta un utilizzo più efficiente della larghezza di banda, utilizzo dei collegamenti, valutazione e controllo dei rischi, monitoraggio remoto e scalabilità. La telemetria integrata, quindi, consente la rapida estrazione e analisi di enormi set di dati per migliorare il processo decisionale [3]

	PUSH	**PULL**
S C O P E R T A AGENTI	L'agente invia automaticamente le metriche non appena si avvia, assicurando che vengano immediatamente rilevate e monitorate continuamente. La velocità di scoperta è indipendente dal numero degli agenti	Richiede al collezionista di spazzare periodicamente lo spazio degli indirizzi per trovare nuovi agenti. La velocità di individuazione dipende dall'intervallo di scoperta e dalla dimensione dello spazio di indirizzamento.

SCALABILITÀ	Attività di polling distribuita completamente tra gli agenti, con conseguente scalabilità lineare. Il collettore centrale leggero ascolta gli aggiornamenti e memorizza le misurazioni. Lavoro minimo per gli agenti di inviare periodicamente serie fissa di misurazioni. Gli agenti sono stateless e esportano i dati non appena vengono generati.	Il carico di lavoro sul poller centrale aumenta con il numero di dispositivi interrogati. Lavoro aggiuntivo sul poller per generare richieste e mantenere lo stato della sessione al fine di far corrispondere richieste e risposte. Lavoro aggiuntivo per gli agenti per analizzare ed elaborare le richieste. Gli agenti spesso hanno bisogno di mantenere lo stato in modo che le metriche possano essere recuperate in un secondo momento
SICUREZZA	Gli agenti push sono intrinsecamente sicuri contro gli attacchi remoti poiché non ascoltano le connessioni di rete.	Il protocollo di polling può potenzialmente aprire il sistema agli attacchi di accesso remoto e denial of service.

COMPLESSITÀ	Configurazione minima richiesta per gli agenti: intervallo di polling e indirizzo del collector. I firewall devono essere configurati per la comunicazione unidirezionale delle misure dagli agenti al collettore.	Il poller deve essere configurato con l'elenco dei dispositivi da interrogare, le credenziali di sicurezza per accedere ai dispositivi e il set di misurazioni da recuperare. I firewall devono essere configurati per consentire la comunicazione bidirezionale tra poller e agenti.
LATENZA	Il basso overhead e la natura distribuita del modello push consentono di inviare le misurazioni più frequentemente, consentendo al sistema di gestione di reagire rapidamente ai cambiamenti. Inoltre, molti protocolli push, come sFlow, sono implementati su UDP, fornendo un trasporto di misure non bloccante a bassa latenza.	La mancanza di scalabilità nel polling in genere significa che le misurazioni vengono recuperate meno spesso, con conseguente visualizzazione ritardata delle prestazioni che rende il sistema di gestione meno sensibile alle modifiche. La comunicazione a due vie coinvolta nel polling aumenta la latenza in quanto le connessioni vengono stabilite e autenticate prima che le misurazioni possano essere recuperate.

FLESSIBILITÀ	Relativamente inflessibile: serie di misure predeterminate e fisse vengono periodicamente esportate	Flessibile: il poller può richiedere qualsiasi metrica in qualsiasi momento

Il modello push è particolarmente interessante per ambienti cloud su larga scala in cui i servizi e gli host vengono costantemente aggiunti, rimossi, avviati e interrotti. Mantenere elenchi di dispositivi per il polling delle statistiche in questi ambienti è impegnativo e la scoperta, la scalabilità, la sicurezza, la bassa latenza e la semplicità del modello push ne fanno un chiaro vincitore.[15]

CASI D'USO: [4]

- Supponiamo di avere molti client che richiedono informazioni riguardo un sistema. Tramite il modello pull ogni client invia continue richieste al sistema per avere le informazioni e questo crea un grande sovraccarico. Nel modello push invece tutti i client eseguono la subscription sui dati che gli interessano e il sistema invierà i dati a tutti i client senza dover gestire un gran numero di richieste asincrone da tutti i client.

- Event-driven: supponiamo che vogliamo sapere quando un interfaccia di rete da up diventa down. Noi vogliamo avere questa informazione il prima possibile e nel frattempo se l'interfaccia non sta andando down non vogliamo sapere continuamente che è up,up,up tramite continue richieste.

Vogliamo eliminare tutto questo traffico non necessario, quindi avere aggiornamenti quando lo stato cambia e non aggiornamenti su stato che non cambia.

I dati di telemetria possono essere trasmessi in streaming utilizzando questi metodi:

- Telemetria basata su modelli: fornisce un meccanismo per lo streaming di dati da un dispositivo con funzionalità MDT a una destinazione.
  I dati da trasmettere sono gestiti tramite abbonamento.

- La telemetria basata su cadenza (CDT): trasmette continuamente i dati (statistiche operative e transizioni di stato) a cadenza configurata. I dati trasmessi in streaming aiutano gli utenti a identificare gli schemi nella rete.

- Telemetria basata su criteri: trasmette i dati di telemetria a una destinazione utilizzando un file di criteri. Un file di criteri definisce i dati da trasmettere e la frequenza alla quale i dati devono essere trasmessi in streaming

1.2. I sistemi di messaggistica

Le connessioni aumentano sempre di piu e connettere i computer è così difficile che software e servizi per fare ciò sono affari da milioni di dollari. Quindi viviamo in un mondo in cui la connettività è avanti di anni rispetto alla nostra capacità di usarla. Solo le più grandi e ricche aziende possono permettersi di creare applicazioni connesse. C'è un cloud, ma è proprietario. I nostri dati e le nostre conoscenze stanno scomparendo dai nostri personal computer in nuvole a cui non possiamo accedere e con cui non possiamo competere. Il punto è che

mentre Internet offre il potenziale del codice connesso in massa, la realtà è che questo è fuori dalla portata della maggior parte di noi, e problemi di così ampia portata (salute, istruzione, economia, trasporti ecc.) rimangono irrisolti perché non c'è modo di collegare il codice, e quindi non c'è modo di collegare i cervelli che potrebbero lavorare insieme per risolvere questi problemi. [10]

Ci sono stati molti tentativi per risolvere la sfida del codice connesso. Ci sono migliaia di specifiche IETF, ognuna delle quali risolve parte del puzzle. Per gli sviluppatori di applicazioni, HTTP è forse l'unica soluzione per essere abbastanza semplice da funzionare, ma probabilmente peggiora il problema incoraggiando gli sviluppatori e gli architetti a pensare in termini di grandi server e piccoli client stupidi.

Quindi oggi le persone stanno ancora connettendo le applicazioni usando UDP e TCP non elaborati, protocolli proprietari, HTTP e Websockets. Rimane doloroso, lento, difficile da scalare ed essenzialmente centralizzato. Le architetture P2P distribuite sono principalmente per il gioco, non per il lavoro. Quante applicazioni usano Skype o Bittorrent per scambiare dati? Il che ci riporta alla scienza della programmazione. Per sistemare il mondo, dovevamo fare due cose. Uno, per risolvere il problema generale di "come connettere qualsiasi codice a qualsiasi codice, ovunque". Due, per avvolgerlo nei blocchi più semplici che le persone potrebbero capire e usare facilmente.

Al giorno d'oggi molte applicazioni sono costituite da componenti che si estendono su un tipo di rete, una LAN o Internet. Così tanti sviluppatori di applicazioni finiscono per fare una sorta di messaggistica. Alcuni sviluppatori usano i prodotti di accodamento dei messaggi, ma la maggior parte delle volte lo fanno da soli, usando TCP o UDP. Questi protocolli non sono difficili da usare, ma c'è una grande differenza tra l'invio di pochi byte da A a B e l'invio di messaggi in qualsiasi modo affidabile.

Diamo un'occhiata ai problemi tipici che affrontiamo quando iniziamo a connettere i pezzi usando il TCP non elaborato. Qualsiasi livello di messaggistica riutilizzabile dovrebbe risolvere tutti o la maggior parte di questi:

- Come gestiamo l'I / O? La nostra applicazione blocca o gestiamo l'I / O in background? Questa è una decisione progettuale chiave. Il blocco degli I / O crea architetture che non si adattano bene. Ma l'I / O in background può essere molto difficile da eseguire correttamente.

- Come gestiamo i componenti dinamici, ovvero i pezzi che vanno via temporaneamente? Suddividiamo formalmente i componenti in "client" e "server" e imponiamo che i server non possano scomparire? Che cosa succederebbe se volessimo connettere i server ai server? Cerchiamo di riconnetterci ogni pochi secondi?

- Come rappresentiamo un messaggio? Come possiamo inquadrare i dati in modo che sia facile da scrivere e leggere, al sicuro da buffer overflow, efficiente per piccoli messaggi, ma adeguato per dati piu grandi?

- Come gestiamo i messaggi che non possiamo consegnare immediatamente? In particolare, se stiamo aspettando che un componente ritorni online? Scartiamo i messaggi, li inseriamo in un database o in una coda di memoria?

- Dove immagazziniamo le code dei messaggi? Cosa succede se il componente che legge da una coda è molto lento e causa l'accumulo delle code? Qual è la nostra strategia allora?

- Come gestiamo i messaggi persi? Aspettiamo nuovi dati, richiediamo un rinvio o creiamo una sorta di livello di affidabilità che garantisce che i messaggi non possano essere persi? Cosa succede se il livello stesso si blocca?

- Cosa succede se abbiamo bisogno di utilizzare un altro trasporto di rete. Multicast invece di TCP unicast? O IPv6? Abbiamo bisogno di riscrivere le applicazioni, oppure il trasporto è astratto in qualche strato?

- Come possiamo indirizzare i messaggi? Possiamo inviare lo stesso messaggio a più peer? Possiamo inviare le risposte a un richiedente originale?

- Come scriviamo un'API per un altro linguaggio? Re-implementiamo un protocollo a livello di filo o riconfezioniamo una libreria? Se il primo, come possiamo garantire stack efficienti e stabili? Se quest'ultimo, come possiamo garantire l'interoperabilità?

- Come rappresentiamo i dati in modo che possano essere letti tra diverse architetture? Applichiamo una particolare codifica per i tipi di dati? Quanto è lontano il lavoro del sistema di messaggistica piuttosto che uno strato superiore?

- Come gestiamo gli errori di rete? Dobbiamo aspettare e riprovare, ignorarli in silenzio o abortire?

1.3. Obiettivo del lavoro

In questo progetto creeremo uno strumento che permetterà di analizzare dati su sistemi connessi in rete tramite remoto, utilizzando dati di telemetria per la comunicazione.

Analizzeremo dati riguardanti cpu, ram, disco,numero di processi, traffico di rete in ingresso e in uscita.

I sistemi quali vogliamo analizzare collezioneranno e invieranno i dati tramite telemetria ad un pc che sarà in ascolto di questi dati e sul quale verranno creati grafici relativi.

Infine valuteremo le performance e analizzeremo i vantaggi di utilizzare la telemetria e confronteremo i vari sistemi di messaggistica tra loro vedendo quale è il piu efficiente e che si presta meglio a questo compito.

2. Stato dell’arte

In questa sezione andrò ad elencare e descrivere i 4 sistemi di messaggistica che ho utilizzato per il progetto.

- MQTT:
- NATS:
- 0MQ:
- Apache Kafka

2.1. MQTT

MQTT (Message Queue Telemetry Transport), protocollo di messaggistica leggero di tipo publish subscribe posizionato in cima a TCP/IP.

Al posto del modello client/server di HTTP, il protocollo MQTT adotta un meccanismo di pubblicazione e sottoscrizione per scambiare messaggi tramite un appostivo "message broker". Invece di inviare messaggi a un determinato set di destinatari, i mittenti pubblicano i messaggi su un certo argomento (detto topic) sul message broker, Ogni destinatario si iscrive agli argomenti che lo interessano e, ogni volta che un nuovo messaggio viene pubblicato su quel determinato argomento, il message broker lo distribuisce a tutti i destinatari. In questo modo è molto semplice configurare una messaggistica uno-a-molti.

MQTT è un protocollo di messaggistica estremamente leggero progettato per dispositivi limitati e reti a bassa larghezza di banda, alta latenza o sostanzialmente inaffidabili. I principi su cui si basa sono quelli di abbassare al minimo le esigenze in termini di ampiezza di banda e risorse mantenendo nel contempo una certa affidabilità e grado di certezza di invio e ricezione dei dati. Protocollo orientato all IoT ed a quelle applicazioni mobili per le quali va tenuto in maggior conto il consumo di banda in rete e di energia dei dispositivi [5]

2.2. NATS

NATS è un sistema di messaggistica open source e nativo del cloud.I principi fondamentali alla base di NATS sono le prestazioni, la scalabilità e la facilità d'uso. Sulla base di questi principi, NATS è progettato attorno alle seguenti caratteristiche principali:

- Altamente performante (veloce)
- Sempre attivo e disponibile (segnale di linea)
- Estremamente leggero (ingombro ridotto)
- Supporto per molteplici qualità di servizio (inclusa la consegna "almeno per una volta" con NATS Streaming)
- Supporto per vari modelli di messaggistica e casi d'uso (flessibile)

NATS è un sistema di messaggistica semplice ma potente progettato per supportare nativamente le moderne architetture cloud. Poiché la complessità non scala, NATS è progettato per essere facile da usare e implementare, offrendo al contempo molteplici qualità di servizio. [9]

- Fanout dei messaggi ad alto throughput : un piccolo numero di produttori di dati (editori) devono inviare frequentemente dati a un gruppo molto più ampio di consumatori (abbonati), molti dei quali condividono interessi comuni in specifici gruppi di dati o categorie (soggetti)

- Indirizzamento, individuazione: invio di dati a istanze, dispositivi o utenti specifici dell'applicazione o scoperta di tutte le istanze / dispositivi / utenti dell'applicazione connessi alla propria infrastruttura.

- Comando e controllo (piano di controllo) : invio di comandi a applicazioni / dispositivi in esecuzione e ricezione dello stato da applicazioni / dispositivi, ad es. SCADA, telemetria satellitare, IOT.

- Bilanciamento del carico: le applicazioni producono un volume elevato di elementi di lavoro o richieste e si desidera utilizzare un pool scalabile dinamicamente delle istanze dell'applicazione di lavoro per assicurarsi di soddisfare gli SLA o altri obiettivi di prestazioni.

- Scalabilità a N: desideri che la tua infrastruttura di comunicazione sfrutti al massimo i meccanismi di concorrenza / scheduling altamente efficienti di Go per migliorare la scalabilità orizzontale e verticale indipendentemente dall'ambiente.

- Trasparenza della posizione: le applicazioni devono essere ridimensionate a un numero molto elevato di istanze distribuite geograficamente e non è possibile permettersi la fragilità dell'accoppiamento stretto delle applicazioni con informazioni dettagliate e specifiche sulla configurazione

degli endpoint su dove si trovano altre applicazioni e su quale tipo di applicazioni dati che stanno producendo o consumando.

- Tolleranza agli errori: l'applicazione deve essere altamente resiliente alla rete o ad altre interruzioni che potrebbero essere al di fuori del controllo dell'utente e occorre la comunicazione dei dati dell'applicazione sottostante per ripristinare senza problemi da interruzioni di connettività

2.3. 0MQ

ZeroMQ (noto anche come ØMQ, 0MQ o zmq) si presenta come una libreria di rete incorporabile ma si comporta come un framework di concorrenza. Fornisce socket che trasportano messaggi atomici su vari trasporti come in-process, inter-process, TCP e multicast. È possibile collegare i socket N-to-N con pattern come fan-out, pub-sub, request-receive.

Originariamente lo zero in ZeroMQ era inteso come "zero broker" e (il più vicino possibile) "zero latenza" (il più possibile). Da allora, ha raggiunto diversi obiettivi: zero amministrazione, zero costi, zero sprechi. Più in generale, "zero" si riferisce alla cultura del minimalismo che permea il progetto. Aggiungiamo potenza rimuovendo la complessità piuttosto che esponendo nuove funzionalità.

Perché 0MQ:

La maggior parte dei progetti di messaggistica utilizza il "broker", che fa indirizzamento, instradamento e accodamento. Ciò si traduce in un protocollo client / server o un set di API in aggiunta a un protocollo non documentato che consente alle applicazioni di parlare con questo broker. I broker sono un'ottima soluzione per ridurre la complessità delle reti di grandi dimensioni. Ma un broker diventa rapidamente un collo di bottiglia e un nuovo rischio da gestire. Se il software lo supporta, possiamo aggiungere un secondo, terzo e quarto broker, le persone lo fanno. Crea più pezzi mobili, più complessità e più cose che possono non funzionare. E una configurazione incentrata sul broker ha bisogno del proprio team operativo. Hai letteralmente bisogno di guardare i broker giorno e notte e batterli con un bastone quando iniziano a comportarsi male. Hai bisogno di scatole e hai bisogno di scatole di sicurezza e hai bisogno di persone per gestire quelle scatole. Vale solo la pena di fare grandi applicazioni con molti pezzi mobili, costruiti da diversi team di persone per diversi anni.

Quindi gli sviluppatori di applicazioni medio-piccole sono intrappolati. O evitano la programmazione di rete e rendono le applicazioni monolitiche non scalabili. Oppure passano alla programmazione di rete e creano applicazioni fragili e complesse difficili da mantenere. Oppure scommettono su un prodotto di messaggistica e finiscono con applicazioni scalabili che dipendono da una tecnologia costosa e facilmente interrotta. Non c'è stata una scelta davvero buona, forse perché la messaggistica è rimasta bloccata nel secolo scorso e suscita forti emozioni: negative per gli utenti, gioia gioiosa per chi vende supporto e licenze.

Ciò di cui abbiamo bisogno è qualcosa che faccia il lavoro di messaggistica, ma lo fa in un modo così semplice ed economico che può funzionare in qualsiasi applicazione, con costi quasi pari a zero. Dovrebbe essere una libreria che si collega, senza altre dipendenze. Dovrebbe funzionare su qualsiasi sistema operativo e funzionare con qualsiasi linguaggio di programmazione. E questo è ZeroMQ: una libreria efficiente e incorporabile che risolve la maggior parte dei

problemi che un'applicazione deve diventare piacevolmente elastica su una rete, senza molti costi.

- Gestisce l'I / O in modo asincrono, nei thread in background. Questi comunicano con i thread dell'applicazione utilizzando strutture di dati prive di lock, quindi le applicazioni ZeroMQ simultanee non necessitano di lock, semafori o altri stati di attesa.

- componenti possono entrare e uscire in modo dinamico e ZeroMQ si ricollegherà automaticamente. Ciò significa che è possibile avviare i componenti in qualsiasi ordine. È possibile creare "architetture orientate ai servizi" (SOA) in cui i servizi possono unirsi e uscire dalla rete in qualsiasi momento.

- Accoda i messaggi automaticamente quando necessario. Lo fa in modo intelligente, spingendo i messaggi il più vicino possibile al ricevitore prima di metterli in coda.

- Ha modi di trattare con code troppo pieno (chiamato "high water mark"). Quando una coda è piena, ZeroMQ blocca automaticamente i mittenti o elimina i messaggi, a seconda del tipo di messaggio che si sta facendo (il cosiddetto "modello").

- Consente alle applicazioni di comunicare tra loro su trasporti arbitrari: TCP, multicast, in-process, inter-process. Non è necessario modificare il codice per utilizzare un trasporto diverso.

- Gestisce i lettori lenti / bloccati in modo sicuro, utilizzando strategie diverse che dipendono dal modello di messaggistica.

- Ti consente di instradare i messaggi utilizzando una varietà di pattern come request-reply e pub-sub. Questi schemi sono il modo in cui crei la topologia, la struttura della tua rete

- Consente di creare proxy per mettere in coda, inoltrare o acquisire messaggi con una singola chiamata. I proxy possono ridurre la complessità di interconnessione di una rete.

- Fornisce messaggi interi esattamente come sono stati inviati, usando una semplice cornice sul filo. Se scrivi un messaggio di 10k, riceverai un messaggio di 10k.

- Non impone alcun formato sui messaggi.

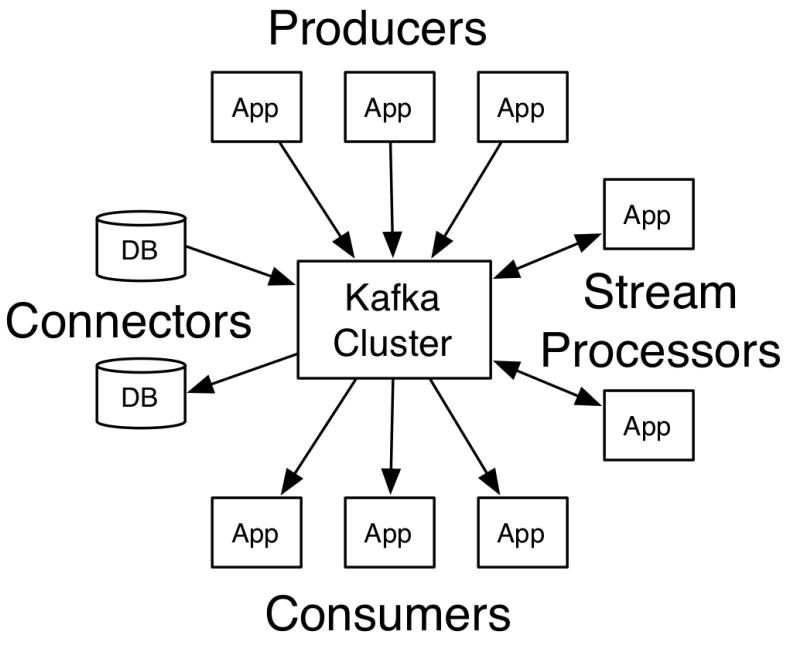
- Gestisce gli errori di rete in modo intelligente, riprovando automaticamente nei casi in cui ha senso.

In realtà ZeroMQ fa più di questo. Ha un effetto sovversivo su come si sviluppano le applicazioni che supportano la rete. Superficialmente, è un'API ispirata al socket su cui si eseguono zmq_recv () e zmq_send (). Ma l'elaborazione dei messaggi diventa rapidamente il ciclo centrale e la tua applicazione si scompone presto in una serie di attività di elaborazione dei messaggi. È elegante e naturale. E scala: ciascuna di queste attività viene mappata a un nodo e i nodi comunicano tra loro attraverso trasporti arbitrari. Due nodi in un processo (il nodo è un thread), due nodi su un box (il nodo è un processo) o due nodi su una rete (il nodo è una scatola) - è lo stesso, senza modifiche al codice. [11]

2.4. Apache Kafka

Apache Kafka è una piattaforma di streaming distribuita. Una piattaforma di streaming ha tre funzionalità chiave:

- Pubblica e sottoscrivi stream di record, simili a una coda di messaggi.
- Memorizza flussi di record in modo duraturo e tollerante ai guasti.
- Elabora flussi di record man mano che si verificano.



Kafka ha quattro componenti principali:

- Producer: consente a un'applicazione di pubblicare un flusso di record su uno o più argomenti di Kafka.

- Consumer: consente a un'applicazione di sottoscrivere uno o più argomenti e elaborare il flusso di record prodotti a loro.

- Streams consente a un'applicazione di agire come un processore di flusso, consumando un flusso di input da uno o più argomenti e producendo un flusso di output su uno o più argomenti di output, trasformando efficacemente i flussi di input in flussi di output.

- Connector consente di creare ed eseguire produttori o consumatori riutilizzabili che collegano gli argomenti di Kafka alle applicazioni o ai sistemi di dati esistenti.

In che modo la nozione di stream di Kafka si confronta con un sistema di messaggistica aziendale tradizionale? La messaggistica ha tradizionalmente due modelli: accodamento e pubblicazione-sottoscrizione. In una coda, un gruppo di consumatori può leggere da un server e ogni record va a uno di essi; in publish-subscribe il record viene trasmesso a tutti i consumatori. Ciascuno di questi due modelli ha una forza e una debolezza. La forza della messa in coda è che consente di suddividere l'elaborazione dei dati su più istanze consumer, consentendo di ridimensionare l'elaborazione. Sfortunatamente, le code non sono multi-abbonati: una volta che un processo legge un dato questo non è più disponibile. Publish-subscribe consente di trasmettere dati a più processi, ma non ha modo di ridimensionare l'elaborazione poiché ogni messaggio viene inviato a tutti gli abbonati.

Il concetto di gruppo di consumatori in Kafka generalizza questi due concetti. Come con una coda, il gruppo di consumatori consente di suddividere l'elaborazione su una raccolta di processi (i membri del gruppo di consumatori). Come con publish-subscribe, Kafka ti consente di trasmettere messaggi a più

gruppi di consumatori.Il vantaggio del modello di Kafka è che ogni argomento ha entrambe queste proprietà - può scalare l'elaborazione ed è anche multi-abbonato - non è necessario scegliere l'uno o l'altro.Kafka ha anche garanzie di ordine più forti rispetto a un sistema di messaggistica tradizionale.Una coda tradizionale conserva i record in ordine sul server e se più utenti consumano dalla coda, il server distribuisce i record nell'ordine in cui sono memorizzati. Tuttavia, sebbene il server distribuisca i record in ordine, i record vengono consegnati in modo asincrono ai consumatori, in modo che possano arrivare fuori servizio su consumatori diversi. Ciò significa in effetti che l'ordine dei record viene perso in presenza di un consumo parallelo. I sistemi di messaggistica spesso aggirano questo problema avendo una nozione di "consumatore esclusivo" che consente a un solo processo di consumare da una coda, ma ovviamente ciò significa che non c'è parallelismo nell'elaborazione. Kafka lo fa meglio. Avendo una nozione di parallelismo - la partizione - all'interno degli argomenti, Kafka è in grado di fornire sia le garanzie di ordinazione che il bilanciamento del carico su un pool di processi di consumo. Ciò si ottiene assegnando le partizioni nell'argomento ai consumatori nel gruppo di consumatori in modo che ogni partizione venga consumata da un solo consumatore nel gruppo. In questo modo ci assicuriamo che il consumatore sia l'unico lettore di quella partizione e consuma i dati in ordine. Poiché ci sono molte partizioni, questo bilancia ancora il carico su molte istanze di consumo. Si noti tuttavia che non ci possono essere più istanze di consumo in un gruppo di consumatori rispetto alle partizioni.

Qualsiasi coda di messaggi che consente di pubblicare messaggi disaccoppiati dal loro consumo agisce effettivamente come un sistema di archiviazione per i messaggi in volo. Ciò che è diverso di Kafka è che è un ottimo sistema di archiviazione. I dati scritti su Kafka vengono scritti su disco e replicati per tolleranza d'errore. Kafka consente ai produttori di attendere il riconoscimento in modo che una scrittura non sia considerata completa finché non viene completamente replicata e viene garantita la persistenza anche se il server non riesce a scrivere. Le strutture del disco usate da Kafka si adattano bene - Kafka

eseguirà lo stesso se si dispone di 50 KB o 50 TB di dati persistenti sul server. Come conseguenza di prendere sul serio lo storage e consentire ai clienti di controllare la loro posizione di lettura, è possibile pensare a Kafka come a una sorta di file system distribuito per scopi speciali dedicato all'archiviazione, alla replica e alla propagazione di registri di commit a bassa latenza ad alte prestazioni. [13]

3. Progetto

3.1. Introduzione

In questo capitolo vedremo come creare dei tool di analisi di sistemi tramite remoto utilizzando i 4 protocolli sopra elencati.

Il progetto consiste nel creare un client che viene eseguito sui sistemi che si vogliono gestire e che analizza dati riguardo, carico della cpu, quantità di ram disponibile, numero dei processi, quantità di disco e dati riguardo il traffico di rete quali: traffico tcp/udp inbound/outbound e altro traffico.

Il client analizza e raccoglie i dati sopraelencati e invia questi dati a un server tramite uno dei protocolli di telemetria.

Dall'altra parte creeremo un programma che viene eseguito sulla macchina con la quale vogliamo vedere i dati di tutti i client tramite remoto. Da qui potremo eseguire la subscribe ai topic di nostro interesse e vedere tramite remoto come si comportano le macchine sul quale è stato installato il client.

Infine vedremo come collegare il programma con una interfaccia per poter visualizzare i dati raccolti

Per fare tutto questo utilizzeremo:

- Java: come linguaggio di programmazione, Java si presta bene a questo compito in quanto presenta una buona documentazione e molte librerie adatte a quello che dovremo fare senza dover perdere troppo tempo ad implementare funzionalità per l'analisi delle performance della macchina,

e che permettono la creazione di programmi cross-platform. Utilizzeremo la versione 10.0 del jdk.

http://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html

- Eclipse: Eclipse è un ambiente di sviluppo integrato multi-linguaggio e multipiattaforma. Eclipse è un ambiente di sviluppo nato per scrivere progammi in Java ma risulta molto versatile in quanto permette, tamite l'installazione di plugin aggiuntivi, di ottenere un vero e proprio IDE multilinguaggio (C/C++, Python, PHP, Javascript, Android e via dicendo). https://www.eclipse.org/

3.2. Raccolta dei dati

La prima fase è quella di creazione del client che viene installato sulla macchina della quale si vogliono analizzare i dati sopraelencati.

Per fare questo utilizzeremo:

- com.sun.management.OperatingSystemMXBean: Interfaccia di gestione specifica della piattaforma per il sistema operativo su cui è in esecuzione la Java virtual machine. Disponibile dalla versione 9.0 del jdk. Utilizzeremo questa interfaccia per ricavare i dati rigurdo cpu,ram,disco e numero dei processi dalla macchina.

- java.lang.ProcessHandle: Disponibile dalla versione 9.0 del jdk. Identifica e fornisce il controllo dei processi nativi. Ogni singolo processo può essere monitorato, elencare i suoi figli, ottenere informazioni sul processo o distruggerlo.

- java.io.File: Una rappresentazione astratta di nomi di file e directory. Le interfacce utente e i sistemi operativi utilizzano stringhe di nomi di percorso dipendenti dal sistema per denominare file e directory. Questa classe presenta una visione astratta indipendente dal sistema dei percorsi gerarchici.

- pcap4j: Libreria Java per la cattura del traffico di rete. Permette di catturare pacchetti in entrata e in uscita dalla scheda di rete e di poter classificare i pacchetti catturati. Fornisce una astrazione per libpcap/ Winpcap indipendente dal sistema operativo sul quale si esegue il programma.

  https://www.pcap4j.org/

Per quanto riguarda la parte di analisi dei dati il codice è indipendente dal protocollo di telemetria che utilizzeremo. Vedremo nei capitoli successivi invece come implementare la comunicazione specifica per ogni protocollo.

RACCOLTA DEI DATI SUL SISTEMA:

- ManagementFactory.getOperatingSystemMXBean():

  La classe ManagementFactory è una classe factory per ottenere bean[1] gestiti per la piattaforma Java. Questa classe è costituita da metodi statici ciascuno dei quali restituisce una o più piattaforme MXBeans che rappresentano l'interfaccia di gestione di un componente della macchina virtuale Java.

- OperatingSystemMXBean.getSystemCpuLoad(): Restituisce il "recente utilizzo della CPU" per l'intero sistema. Questo valore è un double nell'intervallo [0.0,1.0]. Un valore di 0.0 significa che tutte le CPU erano inattive durante il periodo di tempo recente osservato, mentre un valore di 1.0 significa che tutte le CPU stavano funzionando attivamente il 100% del tempo durante il periodo recente osservato. A seconda delle attività in corso nel sistema, sono possibili tutti i valori compresi tra 0.0 e 1.0. Se l'utilizzo recente della cpu del sistema non è disponibile, il metodo restituisce un valore negativo.

[1] Un bean è una classe scritta in un linguaggio ad oggetti che presenta un costruttore vuoto ed un insieme di campi leggibili e scrivibili solo attraverso metodi get e set o delle proprietà. I bean sono utilizzati per rappresentare in modo logico dei dati che sono memorizzati su disco fisso, o per costruire dei tipi di dati adatti ad essere inviati come flissi di byte.

- OperatingSystemMXBean.getSystemFreePhysicalMemorySize(): Restituisce la quantità di memoria fisica libera in byte.

- File.getFreeSpace(): Restituisce il numero di byte non allocati nella partizione nominata da questo nome di percorso astratto, nel nostro caso "/" per avere i byte liberi nel disco.

- ProcessHandle.allProcesses(): Restituisce tutti i processi in esecuzione sulla macchina.

RACCOLTA DEI DATI SUL TRAFFICO DI RETE:

- PcapNetworkInterface Pcaps.getDevByAddress(*ip*):

  Definiamo l'interfaccia di rete della quale vogliamo catturare il traffico

- PcapHandle PcapNetworkInterface.openLive

  (snapLen, PromiscuousMode.PROMISCUOUS,readTimeout):

  Ascolto dalla interfaccia di rete in modalità promiscua[2], specificando il numero di byte catturati per ogni pacchetto e il timeout di lettura (I sistemi operativi passano i pacchetti a Pcap4j dopo che il buffer si riempie o scade il timeout di lettura).

[2] La modalità promiscua è una modadlità di controllo in cui l'interfaccia di rete cablata o wireless fa passare all'unità centrale tutto il traffico che può osservare in rete azichè solo i messaggi con destinatario l'indirizzo dell'interfaccia.

- PcapHandle.setFilter(*filter*, BpfCompileMode.OPTIMIZE):

  Inserisco un filtro per la cattura dei pacchetti. In entrata “dst net myip” in uscita “src net myip”

- PcapHandle.loop(-1, PcapListener): cattura dei pacchetti. Per ogni pacchetto viene richiamata la callback. -1 significa loop infinito.

- PacketListener: Semplicemente una classe che contiene un metodo gotPacket() da implementare che rappresenta l’azione da eseguire con ogni pacchetto ricevuto da PcapHandle.

3.3. Comunicazione MQTT

MQTT: come gia descritto nella introduzione questo protocollo utilizza un broker. Il producer crea dei topic relativi a cosa vuole inviare ed invia i dati al broker. Il consumer esegue la subscribe sui topic che gli interessano e il broker è quello che si occupa di ricevere i messaggi e di inviarli ai consumer che hanno eseguito la subscribe.

Per implementare la comunicazione via MQTT utilizzeremo: [7] [8]

- Mosquitto: Eclipse Mosquitto è un broker di messaggi open source (con licenza EPL / EDL) che implementa le versioni 3.1 e 3.1.1 del protocollo MQTT. Mosquitto è leggero ed è adatto per l'uso su tutti i dispositivi da

computer a scheda singola a basso consumo a server completi. https://mosquitto.org/

- Eclipse Paho: libreria Java che implementa tutti i metodi per la comunicazione MQTT.

  https://www.eclipse.org/paho/

PRODUCER:

MqttClient client = new MqttClient("tcp://*brokerIp*:1883",
MqttClient.generateClientId);

client.connect();

MqttMessage message = **new** MqttMessage();

message.setPayload(dati da inviare);

client.publish(*topicName, message*);

CONSUMER:

MqttClient client = new MqttClient("tcp://*brokerIp*:1883",
MqttClient.generateClientId);

client.setCallback(**new** SimpleMqttCallBack());

client.connect();

client.subscribe(*topicName*);

DESCRIZIONE: [5]

- MqttClient: Client leggero per parlare con un server MQTT utilizzando metodi che bloccano fino al completamento di un'operazione. Questa classe implementa l'interfaccia IMqttClient in cui tutte le azioni vengono bloccate fino al completamento (o al timeout). Questa implementazione è compatibile con tutte le runtime Java SE dalla 1.4.2 in poi. Un'applicazione può connettersi a un server MQTT utilizzando: un semplice socket TCP, un socket SSL / TLS sicuro. Sono disponibili diverse opzioni per la persistenza (su file o in memoria) , vedere la documentazione per dettagli.

  https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttClient.html

  È necessario specificare un ID client client IDd con meno 65535 caratteri. Deve essere unico su tutti i client che si connettono allo stesso server. Il clientId viene utilizzato dal server per archiviare i dati relativi al client, quindi è importante che il clientId rimanga lo stesso quando ci si connette a un server se sono richiesti abbonamenti durevoli o messaggistica affidabile. Viene fornito un metodo di convenienza per generare un ID client casuale che dovrebbe soddisfare questo criterio - generateClientId (). Poiché l'identificatore del client viene utilizzato dal server per identificare un client quando si riconnette, il client deve utilizzare lo stesso identificatore tra le connessioni se sono richieste sottoscrizioni durature o recapito affidabile di messaggi.

- MqttCient.connect(): ci si collega a un server MQTT utilizzando le opzioni predefinite. Per dettagli : https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttConnectOptions.html

- MqttMessage: Un messaggio MQTT, contiene il payload dell'applicazione e le opzioni che specificano il modo in cui il messaggio deve essere consegnato. Il messaggio include un payload (il corpo del messaggio) rappresentato come un byte[]

- MqttClient.publish(): Pubblica un messaggio su un topic sul server.

  Consegna un messaggio al server alla qualità di servizio richiesta e restituisce il controllo una volta che il messaggio è stato consegnato. Nel caso in cui la connessione fallisca o il client si fermi, tutti i messaggi che sono in via di consegna verranno consegnati una volta ristabilita la connessione al server a condizione che: La connessione viene ristabilita con lo stesso clientID

- MqttClient.setCallback(): Imposta il listener di callback da utilizzare per eventi che si verificano in modo asincrono.Ci sono un certo numero di eventi che verranno ascoltati dall'ascoltatore. Questi includono: È arrivato un nuovo messaggio ed è pronto per essere elaborato, la connessione al server è stata persa, la consegna di un messaggio al server è stata completata.

- MqttClient.subscribe(): sottoscrizione ad uno o più topic.

- I topic: ln MQTT, la parola topic fa riferimento a una stringa UTF-8 utilizzata dal broker per filtrare i messaggi per ciascun client connesso. L'argomento è costituito da uno o più livelli di argomento. Ogni livello di argomento è separato da una barra (separatore di argomenti) es:

  livello1/livello2/……../livelloN

  Rispetto ad una coda di messaggi, gli argomenti di MQTT sono molto leggeri. Il cliente non ha bisogno di creare l'argomento desiderato prima di pubblicarlo o di sottoscriverlo. Il broker accetta ogni argomento valido senza alcuna inizializzazione. Quando un cliente si iscrive a un argomento, può sottoscrivere l'argomento esatto di un messaggio pubblicato o può utilizzare le wildcard per iscriversi a più argomenti contemporaneamente. Un wildcard può essere utilizzato solo per iscriversi agli argomenti, non per pubblicare un messaggio. Esistono due diversi tipi di wildcard: a livello singolo e multilivello.

  Livello singolo (wildcard +): Qualsiasi argomento corrisponde a un argomento con wildcard a livello singolo se contiene una stringa arbitraria anziché la wildcard

  Multilivello (wildacard #): Qualsiasi argomento corrisponde a un argomento con wildcard a multilivello se contiene una stringa arbitraria anziché la wildcard per tutti i livelli sottostanti.

3.4. Comunicazione NATS

Come MQTT anche NATS utilizza un broker per gestire le comunicazioni tra producer e consumer di dati.

Per implementare la comunicazione via NATS utilizzeremo:

- gnatsd (Go Nats Daemon): broker di NATS, molto semplice da configurare e da usare in quanto fornisce un ricco set di comandi e parametri per configurare tutti gli aspetti del server. È possibile eseguire il server senza alcun comando sulla porta predefinita o utilizzare la riga di comando o un file di configurazione per configurare il server. Offre varie funzionalità:
    - Clustering: Uno degli obiettivi di design di NATS è che sia sempre attivo e disponibile come un segnale di linea. Per implementare questo obiettivo di progettazione, NATS fornisce meccanismi per i server di clustering per ottenere resilienza e alta disponibilità.
    - Auto-pruning: NATS fornisce meccanismi integrati per gestire i consumatori lenti e gli ascoltatori pigri. Se un utente non è abbastanza veloce o non risponde, il server NATS lo interrompe.
    - Monitoring and troubleshooting: Il server NATS espone una porta di monitoraggio e diversi endpoint con payload JSON che è possibile utilizzare per monitorare il sistema e creare applicazioni di monitoraggio personalizzate.

    - Server Statistics: L'utility nats-top consente di monitorare l'attività del server NATS in tempo reale e visualizzare le statistiche di iscrizione.

- Jnats: Libreria java per NATS

  https://github.com/nats-io/java-nats

PRODUCER:

Connection nc = Nats.connect("nats://*brokerIp*:4222");

nc.publish(*subject, message*);

CONSUMER:

Connection nc = Nats.connect("nats://*brokerIp*:4222")

Dispatcher d = nc.createDispatcher((msg) -> { *azione all'arrivo del messaggio* });

d.subscribe(*subject*);

DESCRIZIONE: [9]

- Connection Nats.connect(): La classe Connection è il cuore del client Java NATS. Fondamentalmente una connessione rappresenta una singola connessione di rete al server gnatsd. Ogni connessione creata comporterà la creazione di un singolo socket e diversi thread: Un thread di lettura per prendere i dati dal socket, un thread di scrittura per mettere i dati sul

socket, un thread del timer per alcuni timer di manutenzione, un thread di invio per gestire il traffico di richiesta / risposta. Le connessioni, per impostazione predefinita, sono configurate per provare a riconnettersi al server se si verifica un errore di rete . È possibile creare una sottoscrizione che consentirà di leggere i messaggi in modo sincrono utilizzando il metodo nextMessage oppure è possibile creare un Dispatcher. Il Dispatcher creerà una discussione per ascoltare i messaggi su uno o più abbonamenti. Il Dispatcher raggruppa una serie di abbonamenti in un singolo thread listener che chiama il codice dell'applicazione per ciascun messaggio..

- Connection.publish(): pubblica il messaggio sul broker con il topic specificato.

- Dispatcher.subscribe(): sottoscrizione ad un argomento. Per ogni messaggio ricevuto viene eseguito il metodo del dispatcher.

- I topic: il protocollo NATS è un semplice protocollo di pubblicazione / sottoscrizione basato su testo. I client si connettono e comunicano con gnatsd (il server NATS) attraverso un normale socket TCP / IP usando una piccola serie di operazioni di protocollo che vengono terminate da newline. A differenza dei sistemi di messaggistica tradizionali che utilizzano un formato di messaggio binario che richiede un'API da utilizzare, il protocollo NATS basato su testo semplifica l'implementazione dei client in un'ampia varietà di linguaggi di programmazione e scripting. I nomi dei topic, distinguono tra maiuscole e minuscole e devono essere stringhe alfanumeriche non vuote senza spazi bianchi incorporati e

facoltativamente delimitati da token utilizzando il carattere punto (.) che caratterizza il livello ,es:

liv1.liv2……livN

Wildcard: NATS supporta l'uso di wildcard nella subscribe per argomenti. Il carattere asterisco (*) corrisponde a qualsiasi token a qualsiasi livello del soggetto. Il simbolo maggiore di (>), noto anche come wildcard completo, corrisponde a uno o più token alla coda di un soggetto e deve essere l'ultimo token.

3.5. Comunicazione 0MQ

A differenza dei protocolli sopraelencati, 0MQ non utilizza un broker per lo scambio di messaggi tra producer e consumer ma utilizza direttamente le socket.

Per implementare la comunicazione via 0MQ utilizzeremo:

- Jeromq: Implementazione pura di Java di libzmq

  https://github.com/zeromq/jeromq

PRODUCER:

ZMQ.Context context = ZMQ.*context*(1);

ZMQ.Socket publisher = context.socket(ZMQ.***PUB***);

publisher.connect("tcp://*consumerIp*:"+ port);

publisher.send((*topicName* +"\n" + *messaggio*));

CONSUMER:

ZMQ.Context context = ZMQ.*context*(1);

ZMQ.Socket subscriber = context.socket(ZMQ.***SUB***);

subscriber.bind("tcp://*localhost:*"+port);

subscriber.subscribe("");

subscriber.recvStr();

DESCRIZIONE: [11]

- ZMQ.Context: Un contesto di ØMQ è thread-safe e può essere condiviso tra tutti i thread di applicazione necessari, senza alcun blocco aggiuntivo richiesto da parte del chiamante. Le socket individuali ØMQ non sono thread-safe, tranne nel caso in cui vengano emesse barriere di memoria complete durante la migrazione di un socket da un thread a un altro. In pratica ciò significa che le applicazioni possono creare un socket in un thread con zmq_socket () e quindi passarlo a un thread appena creato come parte dell'inizializzazione del thread. Contesti multipli possono coesistere all'interno di una singola applicazione. Pertanto, un'applicazione

può utilizzare direttamente ØMQ e allo stesso tempo utilizzare qualsiasi numero di librerie o componenti aggiuntivi che fanno uso di ØMQ.

- ZMQ.Socket: Socket ØMQ nel contesto specificato. Restituisce un handle del socket appena creato. L'argomento type specifica il tipo di socket, che determina la semantica della comunicazione sul socket (Publish per il producer, subscribe per il consumer) .Il socket appena creato inizialmente non è associato e non è associato ad alcun endpoint. Per stabilire un flusso di messaggi, un socket deve prima essere connesso ad almeno un endpoint con zmq_connect, oppure deve essere creato almeno un endpoint per accettare le connessioni in entrata con zmq_bind. Un socket di tipo ZMQ_PUB viene utilizzato da un publisher per distribuire i dati. I messaggi inviati vengono distribuiti in modalità fan out a tutti i peer connessi. La funzione zmq_recv (3) non è implementata per questo tipo di socket. Quando un socket ZMQ_PUB entra in uno stato eccezionale a causa del raggiungimento del limite massimo di abbonati per un utente, allora tutti i messaggi che verrebbero inviati all'abbonato in questione verranno invece rilasciati fino allo scadere dello stato eccezionale. La funzione zmq_send () non deve mai bloccare per questo tipo di socket. Un socket di tipo ZMQ_SUB viene utilizzato da un sottoscrittore per sottoscrivere dati distribuiti da un editore. Inizialmente un socket ZMQ_SUB non è iscritto a nessun messaggio, utilizzare l'opzione ZMQ_SUBSCRIBE di zmq_setsockopt (3) per specificare a quali messaggi sottoscrivere. La funzione zmq_send () non è implementata per questo tipo di socket. In generale, i socket convenzionali presentano un'interfaccia sincrona a flussi di byte affidabili orientati alla connessione o datagrammi non affidabili senza connessione. In confronto, i socket ØMQ presentano un'astrazione di una coda di messaggi asincroni, con la semantica di accodamento esatta a seconda del tipo di socket in uso. Dove

i socket convenzionali trasferiscono flussi di byte o datagrammi discreti, i socket ØMQ trasferiscono messaggi discreti. I socket ØMQ essendo asincroni significa che i tempi della configurazione fisica di connessione e interruzione, ricollegamento e consegna effettiva sono trasparenti per l'utente e organizzati da ØMQ stesso. Inoltre, i messaggi possono essere accodati nel caso in cui un peer non sia disponibile a riceverli. I socket convenzionali consentono solo stringhe one-to-one (due peer), molti-a-uno (molti client, un server) o, in alcuni casi, relazioni uno-a-molti (multicast). I socket ØMQ possono essere collegati a più endpoint usando zmq_connect (), accettando allo stesso tempo le connessioni in ingresso da più endpoint legati al socket usando zmq_bind (), consentendo così relazioni many-to-many.

- ZMQ.Socket.bind(): Crea un endpoint per accettare le connessioni e associarlo al socket a cui fa riferimento l'argomento socket.

- ZMQ.Socket.connect(): Connette il socket all'endpoint specificato

- ZMQ.Socket.send(): invia il messaggio all'endpoint al quale il socket è connesso

- ZMQ.Socket.subscribe(): Stabilisce un nuovo filtro messaggi su un socket 'ZMQ_SUB'. I socket 'ZMQ_SUB' appena creati devono filtrare tutti i messaggi in arrivo, quindi dovresti chiamare questa opzione per stabilire un filtro dei messaggi iniziale. Un filtro vuoto di lunghezza zero significa essere sottoscritto a tutti i messaggi in arrivo. Un filtro non vuoto significa

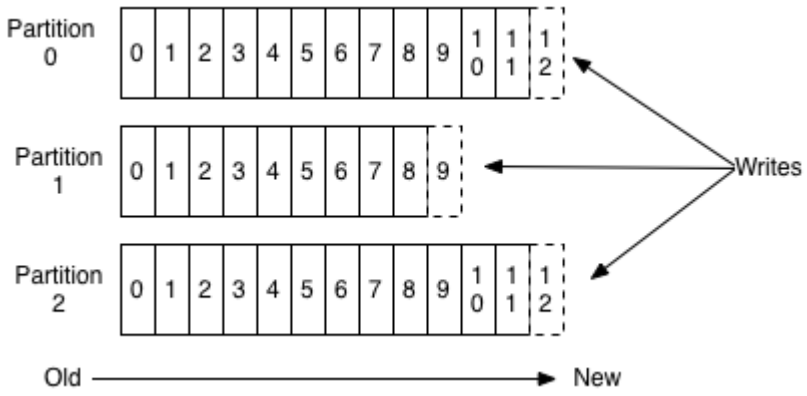
essere sottoscritto a tutti i messaggi che iniziano con il prefisso specificato. Filtri multipli possono essere collegati a una singola socket "ZMQ_SUB", nel qual caso un messaggio deve essere accettato se corrisponde a almeno un filtro.

- ZMQ.Socket.recvStr(): riceve un messaggio sottoforma di stringa.

3.6. Comunicazione Kafka

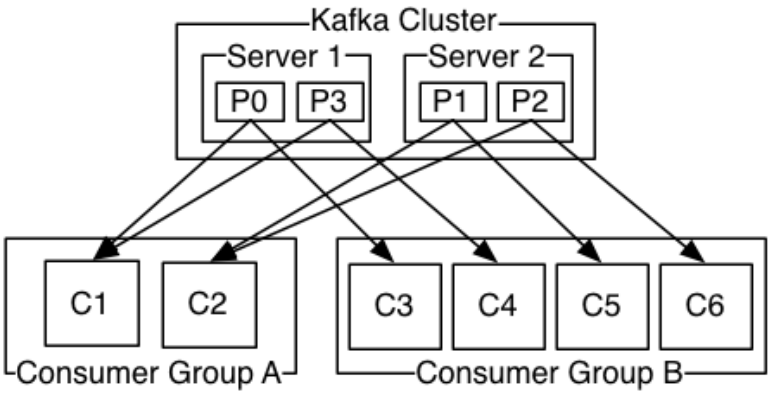
Prima di spiegare la comunicazione in Kafka dobbiamo capire come funziona la gestione dei topic in quanto è leggermente più complicata rispetto agli altri protocolli.

Un topic è un nome di categoria a cui i record sono pubblicati. Gli argomenti in Kafka sono sempre multi-abbonato; cioè, un argomento può avere zero, uno o più consumatori che si abbonano ai dati scritti su di esso. Per ogni argomento, il cluster Kafka gestisce un log partizionato simile a questo:



Ogni partizione è una sequenza ordinata e immutabile di record che viene continuamente aggiunta a un registro di commit strutturato. Ai record nelle partizioni viene assegnato un numero ID sequenziale chiamato offset che identifica in modo univoco ogni record all'interno della partizione. Il cluster di Kafka resiste durevolmente a tutti i record pubblicati, indipendentemente dal fatto che siano stati consumati o meno, utilizzando un periodo di conservazione configurabile. Per ogni consumatore viene mantenuta posizione (offset) di tale consumatore nel log. Questo offset è controllato dal consumatore: normalmente un consumatore farà avanzare il suo offset linearmente mentre legge i record, ma, in effetti, dal momento che la posizione è controllata dal consumatore, può consumare i record in qualsiasi ordine desiderato. Ad esempio, un consumatore può ripristinare un offset precedente per rielaborare i dati dal passato o saltare al

record più recente e iniziare a consumare da "now". Questa combinazione di caratteristiche significa che i consumatori di Kafka sono molto economici: possono andare e venire senza un grande impatto sul cluster o su altri consumatori. I consumatori si etichettano con un nome di gruppo di consumatori e ogni record pubblicato su un argomento viene consegnato a un'istanza di consumatore all'interno di ciascun gruppo di consumatori iscritto. Le istanze consumer possono essere in processi separati o su macchine separate. Se tutte le istanze consumer hanno lo stesso gruppo di consumatori, i record verranno effettivamente bilanciati sul carico rispetto alle istanze consumer. Se tutte le istanze dei consumatori hanno gruppi di consumatori diversi, ogni record verrà trasmesso a tutti i processi del consumatore.



Per implementare la comunicazione via Kafka utilizzeremo:

- Kafka server: il broker di kafka, si occupa di gestire le comunicazioni tra producer e consumer di dati. Utilizza ZooKeeper per il mantenimento dei dati.

- Apache ZooKeeper: ZooKeeper è un servizio centralizzato per il mantenimento delle informazioni di configurazione, la denominazione, la

sincronizzazione distribuita e la fornitura di servizi di gruppo. Tutti questi tipi di servizi sono utilizzati in una forma o nell'altra da applicazioni distribuite

ZooKeeper consente ai processi distribuiti di coordinarsi tra loro attraverso uno spazio dei nomi gerarchico condiviso dei registri dati (che chiamiamo questi registri znodes), proprio come un file system. A differenza dei normali file system, ZooKeeper offre ai suoi client un elevato throughput, bassa latenza, accesso altamente disponibile e strettamente ordinato agli znodes. Gli aspetti prestazionali di ZooKeeper consentono di utilizzarlo in sistemi distribuiti di grandi dimensioni. Gli aspetti di affidabilità impediscono che diventi l'unico punto di errore nei grandi sistemi. Il suo rigoroso ordinamento consente di implementare sofisticate primitive di sincronizzazione sul client. o spazio dei nomi fornito da ZooKeeper è molto simile a quello di un file system standard. Un nome è una sequenza di elementi di percorso separati da una barra ("/"). Ogni znode nello spazio dei nomi di ZooKeeper è identificato da un percorso. E ogni znode ha un genitore il cui percorso è un prefisso dello znode con un elemento in meno; l'eccezione a questa regola è root ("/") che non ha genitore. Inoltre, esattamente come i file system standard, un znode non può essere cancellato se ha figli. Le principali differenze tra ZooKeeper e i file system standard sono che ogni znode può avere dei dati associati (ogni file può anche essere una directory e viceversa) e gli znodi sono limitati alla quantità di dati che possono avere. Il servizio stesso viene replicato su un insieme di macchine che comprendono il servizio. Queste macchine mantengono un'immagine in memoria dell'albero dei dati insieme a registri delle transazioni e istantanee in un archivio permanente. Poiché i dati vengono mantenuti in memoria, ZooKeeper è in grado di ottenere un throughput molto alto e numeri a bassa latenza. Lo svantaggio di un database in memoria è che la dimensione del database che ZooKeeper può gestire è limitata dalla memoria.I server che compongono il servizio

ZooKeeper devono conoscersi reciprocamente. I client si connettono solo a un singolo server ZooKeeper. Il client mantiene una connessione TCP attraverso la quale invia richieste, ottiene risposte.

- org.apache.kafka: libreria Java per kafka.

PRODUCER:

```
Properties props = new Properties();
props.put("bootstrap.servers", "brokerIp:9092");
KafkaProducer<String,String> producer = new KafkaProducer<>(props);
ProducerRecord<String,String> message = new ProducerRecord<>(topicName,dati);
producer.send(message);
```

CONSUMER:

```
Properties props = new Properties();
props.put("bootstrap.servers", "brokerIp:9092");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Pattern.compile(".*"));
ConsumerRecords<String, String> records = consumer.poll(100);
```

DESCRIZIONE:

- KafkaProducer<K,V>: Un client Kafka che pubblica record nel cluster Kafka (Ip specificato nelle Properties). Il produttore è thread-safe e la condivisione di una singola istanza di produttore su thread sarà generalmente più veloce di avere più istanze. Il produttore è costituito da un pool di spazio buffer che contiene i record che non sono ancora stati trasmessi al server e un thread I / O in background responsabile della trasformazione di questi record in richieste e della loro trasmissione al cluster.Il produttore mantiene buffer di record non inviati per ogni partizione. Questi buffer hanno una dimensione specificata dalla configurazione batch.size. Rendere questo più grande può comportare un maggiore numero di batch, ma richiede più memoria (dato che generalmente avremo uno di questi buffer per ogni partizione attiva). Di default è disponibile un buffer da inviare immediatamente anche se nel buffer c'è spazio aggiuntivo non utilizzato. Tuttavia, se si desidera ridurre il numero di richieste, è possibile impostare linger.ms su un valore superiore a 0. Questo imporrà al produttore di attendere fino a quel numero di millisecondi prima di inviare una richiesta nella speranza che arriveranno altri record per riempire il stesso lotto. e i record vengono inviati più velocemente di quanto possano essere trasmessi al server, questo spazio del buffer sarà esaurito. Quando lo spazio del buffer è esaurito, le chiamate di invio aggiuntive verranno bloccate. Per gli usi in cui si desidera evitare qualsiasi blocco è possibile impostare block.on.buffer.full = false che causerà la chiamata di invio a generare un'eccezione.

- ProducerRecord<K,V>: Una coppia chiave / valore da inviare a Kafka. Questo consiste di un nome di argomento al quale viene inviato il record, un numero di partizione facoltativo e una chiave e un valore facoltativi. Se viene specificato un numero di partizione valido, tale partizione verrà

utilizzata durante l'invio del record. Se non viene specificata alcuna partizione ma è presente una chiave, verrà scelta una partizione usando un hash della chiave. Se nessuna chiave o partizione è presente, verrà assegnata una partizione in modalità round robin.

- KafkaProducer.send(): Invia in modo asincrono un record a un argomento

- KafkaConsumer<K,V>: Un client che consuma record da un cluster Kafka. Kafka utilizza il concetto di gruppi di consumatori per consentire a un gruppo di processi di dividere il lavoro di consumo e elaborazione dei record. Questi processi possono essere eseguiti sulla stessa macchina o possono essere distribuiti su molte macchine per fornire scalabilità e tolleranza agli errori per l'elaborazione. Tutte le istanze consumer che condividono lo stesso gruppo.id faranno parte dello stesso gruppo di consumatori. Ogni utente di un gruppo può impostare dinamicamente l'elenco degli argomenti a cui vuole iscriversi tramite una delle API di iscrizione. Kafka consegnerà ogni messaggio negli argomenti sottoscritti a un processo in ciascun gruppo di consumatori. Ciò viene ottenuto bilanciando le partizioni tra tutti i membri nel gruppo di consumatori in modo che ogni partizione sia assegnata a un solo consumatore nel gruppo. Quindi, se c'è un argomento con quattro partizioni e un gruppo di consumatori con due processi, ogni processo consumerebbe da due partizioni. L'appartenenza a un gruppo di consumatori viene mantenuta dinamicamente: se un processo fallisce, le partizioni assegnate ad esso verranno riassegnate ad altri utenti dello stesso gruppo. Allo stesso modo, se un nuovo consumatore si unisce al gruppo, le partizioni verranno spostate dai consumatori esistenti a quello nuovo. Questo è noto come riequilibrare il gruppo ed è discusso in maggior dettaglio di seguito. Il

ribilanciamento di gruppo viene anche utilizzato quando nuove partizioni vengono aggiunte a uno degli argomenti sottoscritti o quando viene creato un nuovo argomento che corrisponde a un'espressione regolare sottoscritta. Il gruppo rileva automaticamente le nuove partizioni aggiornando periodicamente i metadati e li assegna ai membri del gruppo. Concettualmente si può pensare a un gruppo di consumatori come a un singolo abbonato logico che si compone di più processi. Essendo un sistema a più abbonati, Kafka sostiene naturalmente di avere un numero qualsiasi di gruppi di consumatori per un determinato argomento senza duplicare i dati (i consumatori aggiuntivi sono in realtà piuttosto economici).Questa è una leggera generalizzazione della funzionalità che è comune nei sistemi di messaggistica. Per ottenere una semantica simile a una coda in un sistema di messaggistica tradizionale, tutti i processi farebbero parte di un singolo gruppo di consumatori e quindi il recapito dei record sarebbe bilanciato sul gruppo come con una coda. A differenza di un sistema di messaggistica tradizionale, tuttavia, è possibile avere più di questi gruppi. Per ottenere una semantica simile a pub-sub in un sistema di messaggistica tradizionale, ogni processo avrebbe il proprio gruppo di consumatori, quindi ogni processo si iscriverebbe a tutti i record pubblicati sull'argomento.

- KafkaConsumer.subscribe(): subscribe a tutti gli argomenti corrispondenti al modello specificato per ottenere partizioni assegnate dinamicamente. La corrispondenza del modello verrà eseguita periodicamente rispetto agli argomenti esistenti al momento del controllo.

- KafkaConsumer.poll(): consuma il prossimo record (in base all'offset) nel topic a cui ha eseguito la subscribe. Per leggere più messaggi questo metodo è spesso utilizzato dentro un ciclo.

3.7. Rappresentazione grafica dei dati

Per visualizzare i dati raccolti dai sistemi sottoforma di grafici ho utilizzato: [15]

- InfluxDB: un archivio dati ad alte prestazioni scritto appositamente per i dati delle serie temporali. Permette l'acquisizione, la compressione e l'interrogazione in tempo reale ad alto throughput degli stessi dati. Fornisce InfluxQL come linguaggio di query simile a SQL per interagire con i tuoi dati. È stato creato con cura per essere familiare a chi proviene da altri ambienti SQL o SQL, fornendo allo stesso tempo funzionalità specifiche per l'archiviazione e l'analisi dei dati delle serie temporali. InfluxQL supporta anche espressioni regolari, espressioni aritmetiche e funzioni specifiche di serie temporali per accelerare l'elaborazione dei dati.

Può gestire milioni di punti dati al secondo. Lavorare con così tanti dati per un lungo periodo di tempo può creare problemi di archiviazione. InfluxDB compatterà automaticamente i dati per ridurre al minimo lo spazio di archiviazione. Inoltre, è possibile ridimensionare facilmente i dati; mantenendo i dati grezzi di alta precisione solo per un periodo limitato e memorizzando la precisione più bassa, i dati riepilogati per molto più tempo o per sempre.

- Chronograf: Interfaccia grafica, ti permette di vedere rapidamente i dati che hai archiviato in InfluxDB in modo da poter creare query e avvisi robusti. È semplice da utilizzare e include modelli e librerie per consentire di creare rapidamente dashboard con visualizzazioni in tempo reale dei dati. Offre una soluzione di dashboard completa per la visualizzazione dei dati. Sono disponibili oltre 20 dashboard pre-inscatolati per consentirti di iniziare molto rapidamente. È possibile clonare facilmente uno di questi dashboard pre-inscatolato per creare dashboard personalizzati o crearli da zero, in entrambi i casi, è possibile creare il dashboard perfetto per soddisfare le esigenze di visualizzazione.

- org.influxdb: libreria java per influxdb

UTILIZZO DELLA LIBRERIA:

InfluxDB influxDB = InfluxDBFactory.connect("*http://localhost:8086*");

influxDB.createDatabase(“*nome database*”);

Point point = Point.*measurement*(“*nome misurazione*”).addField("*nome campo*",valore).time(“*current time*").build();

influxDB.setDatabase("*nome database*");

influxDB.write(point);

DESCRIZIONE:

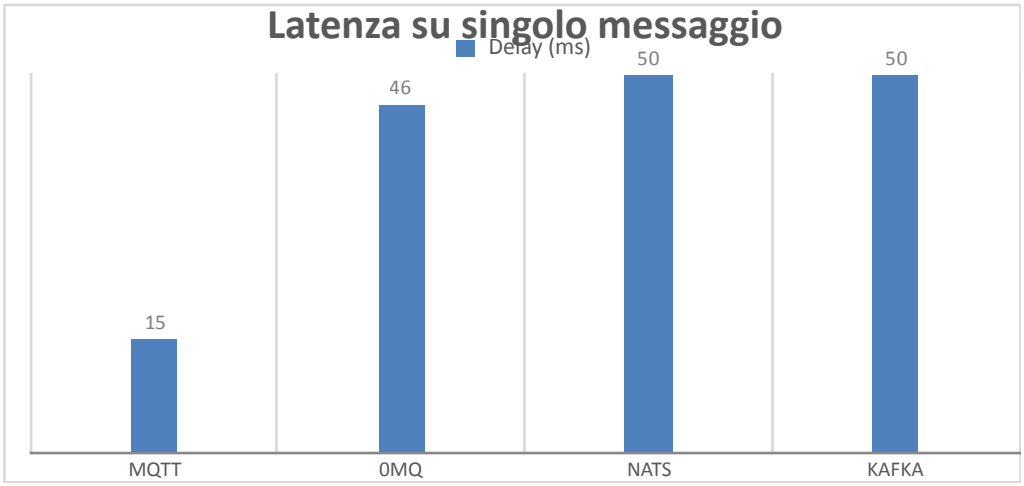
- InfluxDBFactory.connect(): Si connette ad una sessione aperta di InfluxDB e restituisce un handler per interagire col database
- InfluxDB.createDatabase(): crea un nuovo database con il nome specificato come argomento
- Point.build(): crea una entry del database con nome, valori e tempo di misurazione specificati.
- InfluxDB.setDatabase(): seleziona il database da utilizzare
- InfluxDB.write(): scrive il punto passato come argomento nel database selezionato tramite setDatabase.

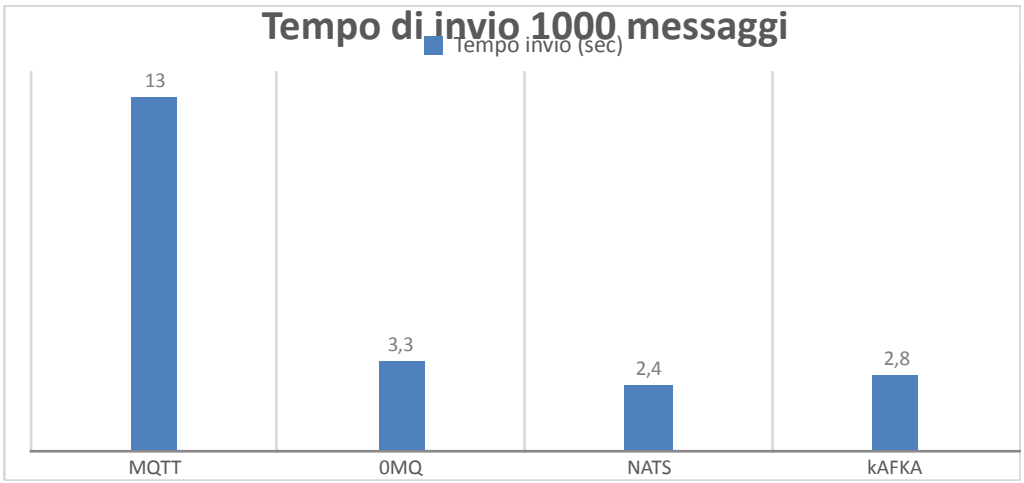
4. Valutazione performance e confronti

In questo capitolo confronteremo i 4 sistemi di messaggistica utilizzati.

Confronteremo le performance di ognuno tramite appositi programmi di test per vedere come si comportano quando stressati.

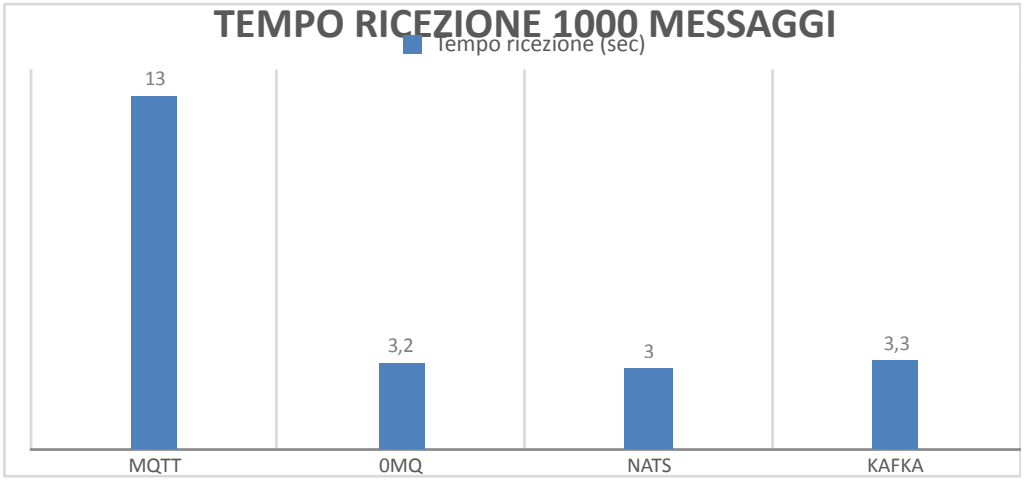
4.1. Performance

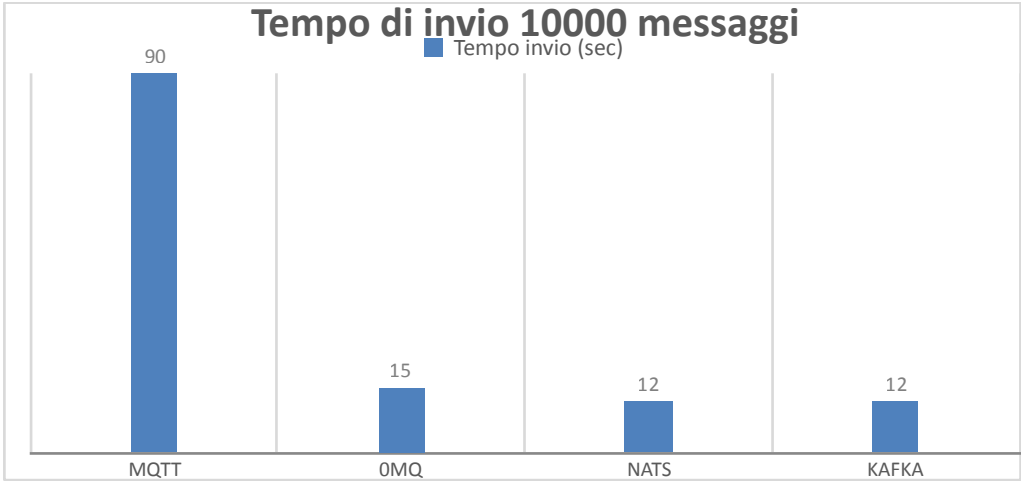




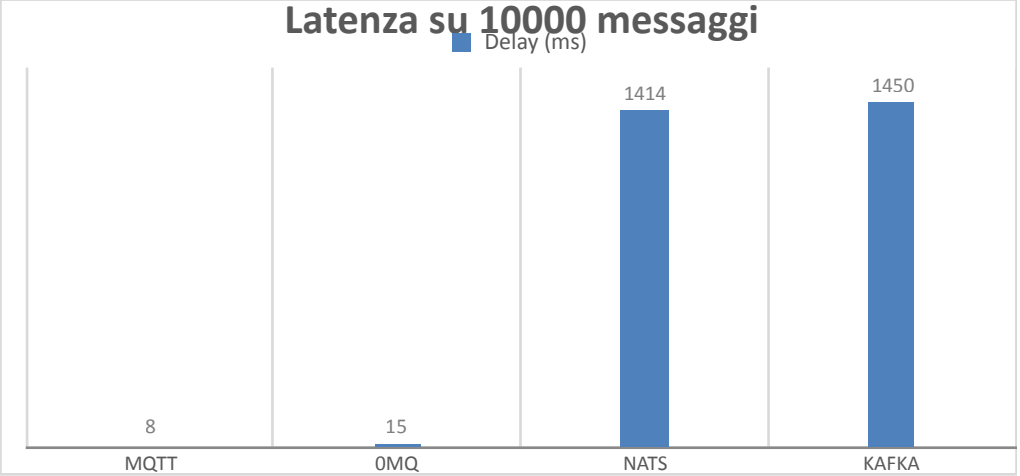

Persi su 1000 messaggi
Messaggi persi
0
0
0
0
MQTT
0MQ
NATS
KAFKA


Latenza su 1000 messaggi
Delay (ms)
10
20
370
640
MQTT
0MQ
NATS
KAFKA


TEMPO RICEZIONE 1000 MESSAGGI
Tempo ricezione (sec)
13
3,2
3
3,3
MQTT
0MQ
NATS
KAFKA


Tempo di invio 10000 messaggi
Tempo invio (sec)
90
15
12
12
MQTT
0MQ
NATS
KAFKA

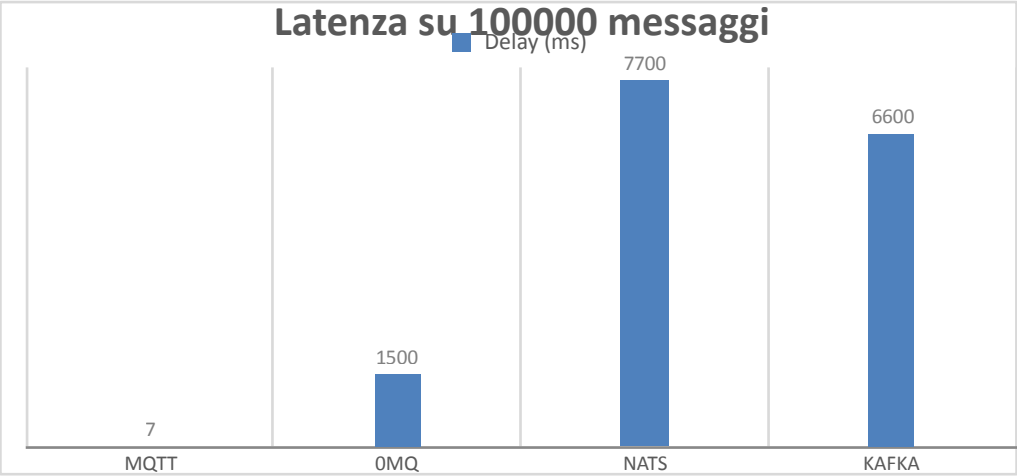

Persi su 10000 messaggi
Messaggi persi
0
0
0
0
MQTT
0MQ
NATS
KAFKA

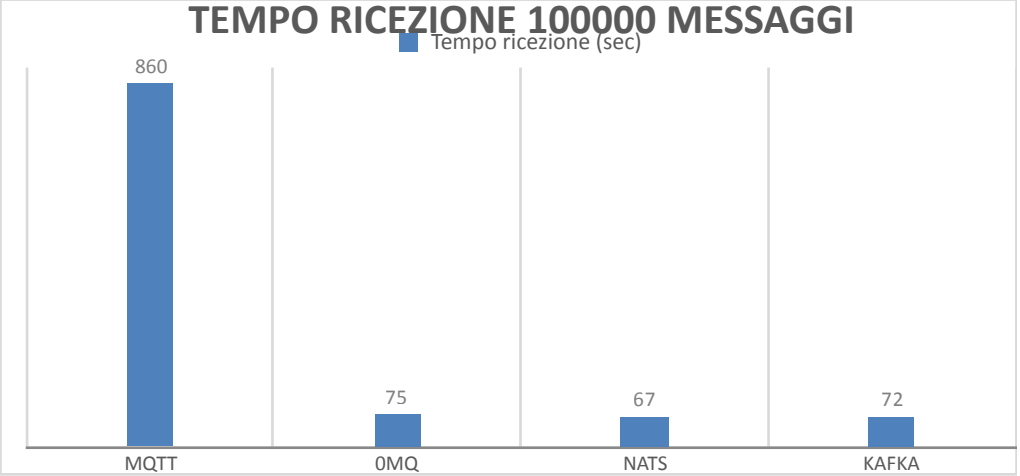

Latenza su 10000 messaggi
Delay (ms)
8
15
1414
1450
MQTT
0MQ
NATS
KAFKA


TEMPO RICEZIONE 10000 MESSAGGI
Tempo ricezione (sec)
90
15
14
14
MQTT
0MQ
NATS
KAFKA


Tempo di invio 100000 messaggi
Tempo invio (sec)
860
72
51
61
MQTT
0MQ
NATS
KAFKA


Persi su 100000 messaggi
Messaggi persi
0
0
0
0
MQTT
0MQ
NATS
KAFKA





4.2. Considerazioni

Dai test possiamo vedere che:

- MQTT è il protocollo più inefficiente quando si tratta di inviare un gran numero i messaggi ad alta velocità, infatti i suoi tempi di invio e ricezione sono decisamente più alti rispetto agli altri sistemi. In compenso però mantiene una sincronia tra produttore e consumatore quindi un basso ritardo tra invio e ricezione. Un messaggio viene quindi ricevuto dal consumatore non appena viene prodotto il che lo rende un sistema buono quando si necessita l'invio di pochi messaggi ma che devono essere ricevuti tempestivamente e non buono se si necessita l'invio di molti messaggi a gran velocità. Rimane comunque un sistema affidabile in quanto non ho rilevato perdite di messaggi.

- 0MQ dai test si è rivelato il miglior sistema di messaggistica sotto tutti i punti di vista. Veloce nell invio di grandi quantità di dati e nel frattempo mantiene un ritardo basso tra invio e ricezione. L'assenza del broker lo rende anche un sistema più leggero degli altri. Risulta affidabile in quanto non ci sono perdite di messaggi e i messaggi non ricevuti vengono mantenuti in un buffer dal produttore.

- NATS e KAFKA risultano essere entrambi sistemi veloci ed affidabili ma non mantengono sincronia tra produttore e consumatore infatti risulta un alto ritardo tra invio e ricezione. Questo può essere un problema se si vogliono ricevere messaggi tempestivamente. In compenso l'utilizzo di un broker permette delle funzionalità esclusive come il mntenimento dello storico dei messaggi inviati e a possibilità di un consumatore di connettersi in ritardo rispetto al produttore e ricevere i messaggi inviati in precedenza.

Nel contesto di questo progetto, il candidato migliore per trasportare misure di telemetria e' OMQ in alternativa NATS/KAFKA se l'utilizzo di un broker e' necessario. In appendice e' possibile trovare i riferimenti al codice sorgente del gestore della telemetria per host basati su sistema operativo Windows e Linux. A causa dei limiti hardware del testbed, non e' stato possibile validare come cambia l'implementazione quando si hanno tanti produttori e un solo ricevitore (nella nostra simulazione abbiamo un produttore e un consumatore) anche se da quanto siamo riusciti a provare, non sono state evidenziate differenze in performance

5. Conclusioni e lavoro futuro

Abbiamo visto che la telemetria, quindi un passaggio ad un modello push porta molti benefici in termini di efficienza sicurezza e scalabilità. Tutti i sistemi di messaggistica utilizzati per il progetto si sono rivelati validi per le esigenze del progetto stesso.

In futuro continuerò il progetto utilizzando solamente 0MQ in quanto si è rivelato il migliore tra tutti i sistemi provati.

Prevedo di implementare un sistema di sicurezza utilizzando CurveZMQ. [12]

CurveZMQ ha lo scopo di prevenire intercettazioni, dati fraudolenti, dati alterati, attacchi di replay, attacchi di amplificazione, attacchi man-in-the-middle, attacchi di furto di chiavi, attacchi di identità e certi attacchi denial-of-service. CurveZMQ utilizza la curva ellittica Curve25519, progettata da Daniel J. Bernstein per ottenere buone prestazioni con dimensioni di chiave brevi (256 bit).[19]

BIBLIOGRAFIA

1. Cisco telemetry - Shelly Cadora - April 13, 2016 - Why You Should Care About Model-Driven Telemetry https://blogs.cisco.com/sp/why-you-should-care-about-model-driven-telemetry

2. Krishna Chaitanya Kotha - December 20, 2017 - It's time to move away from SNMP and CLI and use Model-Driven Telemetry https://blogs.cisco.com/developer/its-time-to-move-away-from-snmp-and-cli-and-use-model-driven-telemetry

3. Shelly Cadora – Ten lessons for telemetry - Cisco Systems https://www.nanog.org/sites/default/files/Cadora_Ten_Lessons.pdf

4. Don Jacob, Technical Marketing Specialist - June 12, 2018 - Network Basics by Packet Design: What is Streaming Telemetry? https://www.packetdesign.com/blog/network-basics-by-packet-design-what-is-streaming-telemetry/

5. Mqtt - http://mqtt.org/ - 08/2018

6. Andy Stanford-Clark and Hong Linh Truong - MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2 November 14, 2013

7. Mosquitto - MQTT broker - https://mosquitto.org/ - 08/2018

8. Eclipse Paho – Libreria Java per MQTT - https://www.eclipse.org/paho - 08/2018

9. Nats: https://nats.io/documentation/ - 08/2018

10. Pieter Hintjens, CEO of iMatix - 0MQ the guide - http://zguide.zeromq.org/page:all - 08/2018

11. Pieter Hintjens, CEO of iMatix - CurveZMQ - Security for 0MQ - http://curvezmq.org/page:read-the-docs - 08/2018

12. Apache Kafka - https://kafka.apache.org/ - 08/2018

13. Apace Zookeeper – Server di mantenimento informazioni di Kafka- http://zookeeper.apache.org/ - 08/2018

14. InfluxData technologies - https://www.influxdata.com/ - 08/2018

15. Justin Ryburn - SNMP must die - The tyranny of OIDs - https://www.nanog.org/sites/default/files/1_Ryburn_SNMP_Must_Die.pdf

16. Clean Code - A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)

17. Carlos Campana - December 2015 - Introduction to Streaming Telemetry - http://ix.br/pttforum/9/slides/ixbr9-telemetry.pdf

18. Curve 25519 - https://en.wikipedia.org/wiki/Curve25519 - 08/2018

APPENDICE – CODICE SORGENTE

Il software sviluppato in questo progetto e' liberamente disponibile all'indirizzo

http://github.com/flack90/Tesi