ArchMiner
Note di progetto

Giuseppe Muntoni

December 18, 2019

Contents

1 Premessa

Le seguenti note potrebbero contenere errori o imprecisioni. Servono a mantenere traccia del lavoro svolto e possono essere utilizzate come base di discussione per le riunioni con i relatori.

Il lavoro di progettazione svolto fino ad ora ha condotto ad una soluzione che è compatibile solo con Kubernetes. L'idea è però quella di rendere la logica di riconoscimento dei pattern di comunicazione indipendente dalla piattaforma di deployment utilizzata. Mentre sarà dipendente da essa il modulo che si occupa di recuperare i containers dell'applicazione e monitorarli con TCPdump.

2 App distribuite con Kubernetes

2.1 Cluster di Kubernetes

Inizialmente ho creato un cluster di Kubernetes che fosse utile per effettuare dei test. Ho installato sulla mia macchina fisica tre macchine virtuali con Ubuntu 18.04, 2 core e 3 gb di ram ciascuna. Uno è il K8S master node e gli altri due sono worker node.

2.2 Generazione dei nodi

L'idea iniziale era quella di effettuare il parsing dei file yaml di deployment. Questo approccio presenta dei problemi: ogni object di Kubernetes viene definito attraverso un file yaml con uno specifico schema che varia, anche se di poco, a seconda del campo "ApiVersion" presente all'inizio del file. Questo rende necessario lo sviluppo di un parser che sia compatibile con tutte le versioni dell'API attuali e con eventuali versioni successive, rendendo di fatto il tool difficilmente mantenibile. Inoltre, i pod vengono deployati utilizzando vari tipi di controller quali ReplicaSet, ReplicationController, StatefulSet, DaemonSet, Deployment, Jobs, CronJobs. Tutti questi hanno una schema yaml differente rendendo così inutilmente complesso il parser.

Una soluzione alternativa sarebbe quella di utilizzare direttamente l'API Rest di Kubernetes. Bastano delle GET per ottenere un file JSON con la lista di tutti i pods e un file JSON con la lista di tutti i services. Con questo metodo è possibile mantenere aggiornate le informazioni relative ai pods. Questo perchè le richieste effettuate tramite l'API di Kubernetes restituiscono, oltre che lo stato desiderato, anche lo stato attuale. Quindi, se ad esempio un pod fallisce e ne viene creato uno nuovo che ha un indirizzo IP differente, posso notarlo.

2.3 Generazione degli archi

L'idea iniziale era quella di utilizzare Istio, un tool in grado, tra le altre cose, di generare dei log in cui sono presenti tutte le informazioni relative alle comunicazioni tra i Pods. Istio essenzialmente è il control-plane di un service mesh. Un service mesh è un layer infrastrutturale che si occupa di fornire una

serie di funzionalità quali service discovery, load balancing, tracing, monitoring, end-to-end authentication ed altro. È composto di norma da un data plane e da un control plane. Il data plane contiene le istanze dei servizi, ognuna delle quali è collegata ad un sidecar proxy. Il control plane si occupa di configurare e controllare i sidecar proxy affinchè siano disponibili le funzionalità elencate in precedenza. Istio utilizza di default Envoy come sidecar proxy.

Fonte: What is a service mesh - NGINX

Dato che Istio fornisce tantissime funzionalità e il nostro scopo è solo quello di individuare le comunicazioni tra i servizi, ho cercato una soluzione che fosse meno invasiva. Inizialmente pensavo di installare un sidecar proxy che facesse solo il monitoring delle comunicazioni e null'altro. Il problema di quest'approccio sta nel fatto che la configurazione di un sidecar proxy è estremamente complessa.

Dalla documentazione di Kubernetes si evince che tutti i containers in esecuzione in un pod condividono la stessa rete. Allora, la soluzione più semplice è quella di eseguire in tutti i pods un ulteriore container Docker che esegua il tool TCPdump per fare il tracing dei pacchetti. Dal momento che per ogni pod, attraverso l'API di K8S, posso conoscere l'indirizzo IP e anche la porta sulla quale è in ascolto il container all'interno di esso, posso individuare le comunicazioni container-to-container dell'applicazione.

L'approccio appena descritto è stato testato con l'applicazione demo Sock-Shop sul cluster da me creato ed ha funzionato correttamente.

AGGIUNGERE ESEMPIO

3 App distribuite con Docker swarm

Da discutere.

4 Individuazione patterns di comunicazione e database

4.1 Message Router

Si distinguono varie tipologie di message router. Vi sono i service discovery che possono essere progettati seguendo il client-side discovery pattern o il server-side (platform-based) discovery pattern; inoltre vi sono le API Gateway. La logica di riconoscimento è differente per ognuna di esse. Analizziamole singolarmente.

4.1.1 Client-side discovery

Il client (un generico microservizio) si collega ad un service registry per ottenere gli indirizzi IP delle istanze del microservizio col quale vuole comunicare. In seguito effettua load balancing con una certa politica tipo Random o Round-robin. Questo corrisponde alla proprietà booleana "dynamicDiscovery" delle relazioni "InteractsWith" nel microtosca. L'idea è quella di associare ad ogni nodo un nome e ad ogni istanza di uno stesso nodo un id (ad esempio l'indirizzo

IP). Basta verificare che siano presenti ID distinti per i pacchetti in uscita verso un determinato nodo “x”.

FONTE: Microservices Pattern, Richardson

4.1.2 Server-side discovery

Il client comunica con un servizio esterno che si occupa di: ottenere le istanze disponibili attraverso il service registry e di instradare la richiesta su una di esse. Vi possono essere service discovery che operano a livello di rete e che operano a livello applicazione. Quelli che operano a livello di rete effettuano l'instradamento e il load balancing tra le istanze basandosi solamente sugli indirizzi IP e le porte. In questo caso il payload del pacchetto TCP non viene modificato. È possibile quindi, determinare se un nodo è un message router in questo modo:

1. Calcolare la SHA-256 del payload dei pacchetti TCP in ingresso e uscita.
2. Verificare se, data l'hash di un pacchetto in ingresso, ne esiste una uguale in uscita.
3. Dato che il monitoring con TCPDump parte in qualunque momento mentre l'appicazione è in distribuzione, è necessario notare che se un nodo è un message router potrei individuare un hash in uscita che non è presente in ingresso e un hash in ingresso che non è presente in uscita. Quindi è necessario che vi sia una corrispondenza tra hash in input ed in output non completa ma sopra una certa percentuale da stabilire.

Inoltre, vi possono essere service discovery che operano a livello di applicazione. Ad esempio, se viene utilizzato il protocollo HTTP, allora il routing può essere fatto basandosi sul path o sul metodo. In questo caso è necessario verificare se vengono aggiunti header quali X-Forwarded-For nel pacchetto HTTP. Da alcuni load balancer, tipo AWS ELB, vengono utilizzati anche degli header personalizzati. Si potrebbe mantenere un DB con gli header utilizzati dai load balancer più comuni.

Qui è possibile leggere l'header che viene settato da AWS ELB.

Se l'applicazione è distribuita con Kubernetes, allora il server-side discovery tra le istanze di un pod può essere effettuato dagli oggetti di tipo "Service", che possono essere ottenuti con una semplice GET al Kubernetes master.

Questo nel microtosca corrisponderà ad un nodo di tipo "MessageRouter".

FONTI: Microservices Pattern, Richardson, Documentazione Kubernetes Services

4.1.3 API Gateway

Una API Gateway è un servizio che è un entry-point nell'applicazione dal mondo esterno. Questa è responsabile di effettuare il routing delle richieste, API Composition, Protocol Translation, autenticazione, caching, logging. Inoltre, fornisce una API specifica per ogni tipologia di client.

Un modo possibile per riconoscere una API Gateway è quella di verificare se tra gli header della risposta HTTP è presente l'header X-Forwarded-For. Inoltre è possibile che, a seconda dell'API Gateway utilizzata, venga inserito un header personalizzato. Si potrebbe mantenere un DB con gli header utilizzati dalle API Gateway più comuni.

Inoltre, se il deployment dell'applicazione è effettuato con Kubernetes, per verificare se viene utilizzata una API Gateway basta verificare se è presente un oggetto di tipo "Ingress".

Qui è possibile trovare la documentazione ufficiale degli Ingress di Kubernetes.

4.2 Message Broker

Questa tabella mostra quali sono le varia tipologie di IPC in una applicazione a microservizi.

	One-to-One	One-to-Many
Synchronous	Request/response	–
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Figure 1: IPC

Per le comunicazioni di tipo asincrono abbiamo architetture brokerless o architetture con message broker.

La maggior parte dei message broker sviluppati da terzi utilizzano protocolli di messagistica standard. I protocolli individuati sono AMQP, MQTT e STOMP. Un message broker può essere quindi riconosciuto attraverso l'analisi dei messaggi nel payload del pacchetto TCP.

Analizziamo singolarmente il metodo di riconoscimento di un message broker a seconda del protocollo di messagistica utilizzato.

4.2.1 Comunicazione con AMQP 0.9.1

AMQP è un protocollo multi-canale. In una singola connessione TCP possono essere infatti creati più canali. I canali sono indipendenti l'uno dall'altro e consentono quindi di effettuare più operazioni contemporaneamente su una stessa connessione.

La seguente figura mostra l'architettura di AMQP:

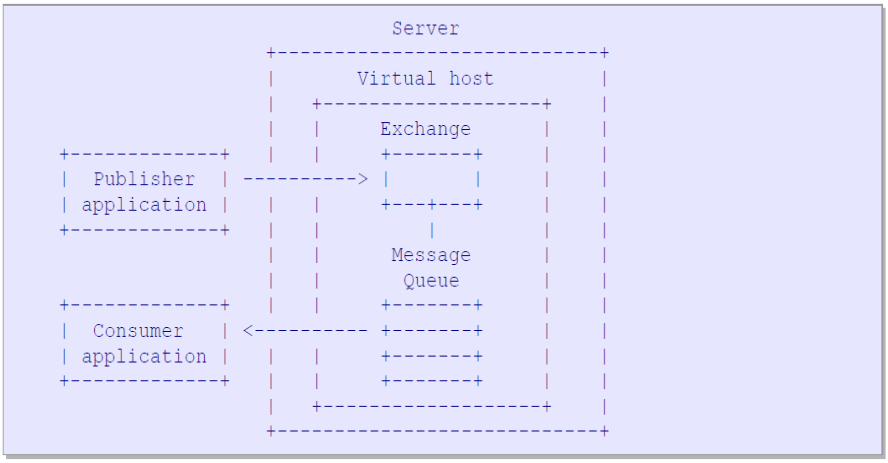

Figure 2: AMQP Architecture

In estrema sintesi, l'exchange si occupa di accettare i messaggi dai publisher e di inoltrarli, tramite determinate politiche di routing, ad una o più code di messaggi dove verranno consumati dai consumatori.

Osserviamo ora, qual è il formato dei frames trasportati:

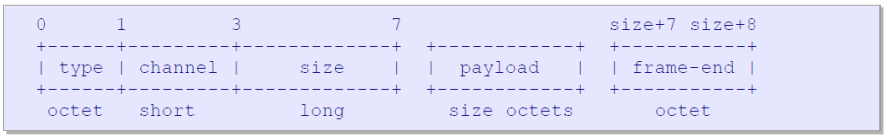

Figure 3: AMQP wire level frames

L'header del frame è composto dai primi 7 bytes.

Il primo indica il tipo del messaggio trasportato nel payload che può essere:

- Type = 1, "METHOD": method frame.
- Type = 2, "HEADER": content header frame.
- Type = 3, "BODY": content body frame.
- Type = 4, "HEARTBEAT": heartbeat frame.

Seguono due bytes per il numero del canale, che sarà 0 per tutti i frames che sono globali alla connessione e 1-65535 per i frames che si riferiscono a canali specifici.

Il campo size di 4 bytes indica la dimensione del payload escluso il frame-end di un byte.

La figura seguente indica la struttura del METHOD frame:

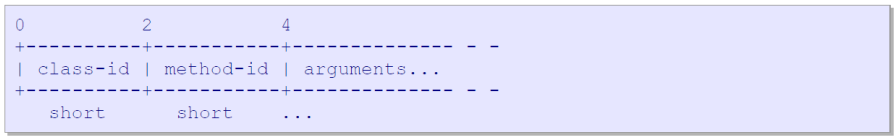

Figure 4: AMQP METHOD frame

Ogni metodo appartiene ad una specifica classe e può essere inviato dal client al server, dal server al client o entrambe. Gli arguments non sono nient'altro che i parametri del metodo. Inoltre un metodo potrebbe trasportare dei contenuti: se così fosse, allora il METHOD frame sarebbe seguito esattamente da un CONTENT HEADER frame e da zero o più CONTENT BODY frames.

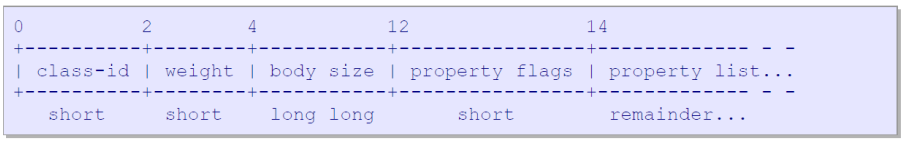

Figure 5: AMQP CONTENT HEADER frame

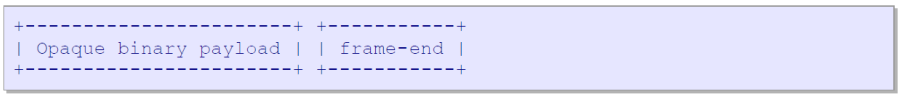

Figure 6: AMQP CONTENT BODY frame

La specifica AMQP definisce le classi standard e la specifica dei metodi di ciascuna di esse. Dei metodi viene specificato anche se vengono chiamati dal client, dal server o se possono essere chiamati da entrambi.

TCPdump fornisce il dissector per AMQP 0.9.1. Basta quindi analizzare il class-id e il method-id dei frame in ingresso/uscita da un nodo per comprendere se di tratta di un client o di un server. Il server corrisponde esattamente ad un message broker.

Qui è possibile scaricare la documentazione ufficiale del protocollo.

4.2.2 Comunicazione con AMQP 1.0.0

AMQP 1.0.0 è una versione di AMQP completamente differente rispetto alla 0.9.1. Una rete AMQP consiste di nodi interconnessi via link. I nodi sono entità con nome. Esempi di nodi sono produttori, consumatori e code. I link sono route unidirezionali tra due nodi, di cui uno è il nodo sorgente, l'altro è il nodo target. Questi vengono utilizzati per il trasferimento di messaggi.

Ad ogni nodo è associato un container. Esempi di container possono essere i clients ed i brokers. Un container può contenere più nodi. Ad esempio un generico client può essere sia un produttore che un consumatore.

Nel caso di AMQP 1.0.0 non esiste una distinzione tra client e server, bensì esistono generici peers.

Analizziamo come avviene la comunicazione tra i peers. Innanzitutto vi è un connessione TCP tra due containers. Una connessione è divisa in un certo numero di canali unidirezionali, ai quali è associato un numero.

Una sessione AMQP è composta da due canali unidirezionali che insieme formano una conversazione bidirezionale e sequenziale tra due containers. È possibile avere più sessioni contemporaneamente su un'unica connessione, consentendo di avere flussi di comunicazione indipendenti l'uno dall'altro.

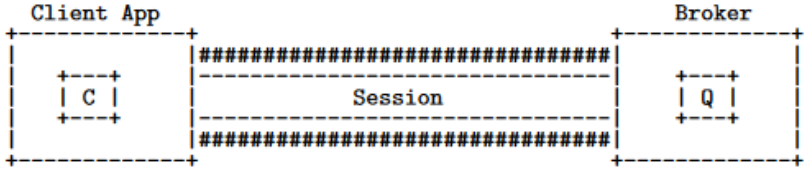

Figure 7: AMQP Session

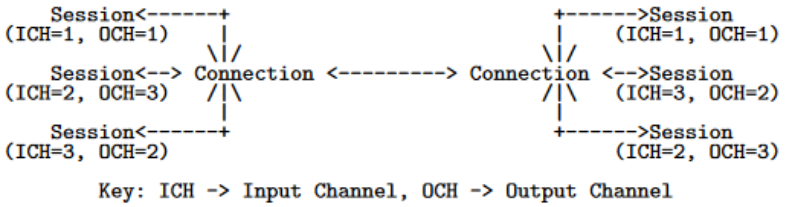

Figure 8: Sessioni multiple

Per connettere due nodi in due containers vengono utilizzati i links. Ad ogni link è associata una sessione. Ad una sessione possono essere associati molti links. Le terminazioni possono essere sorgenti o targets. Un nodo sorgente tiene traccia dei messaggi in uscita, mentre un nodo target tiene traccia dei messaggi in ingresso.

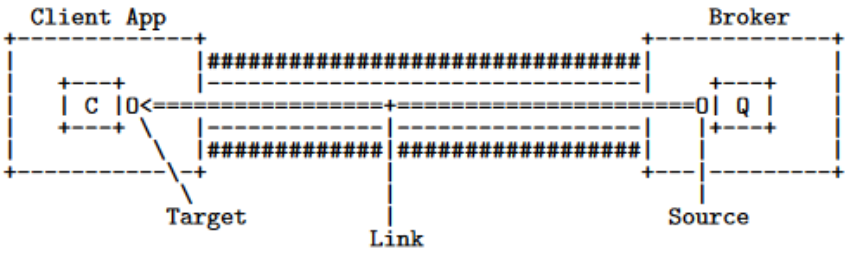

Figure 9: AMQP Link

Vediamo ora qual è la struttura dei frames trasportati sulla rete.

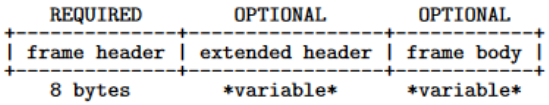

Figure 10: Frame

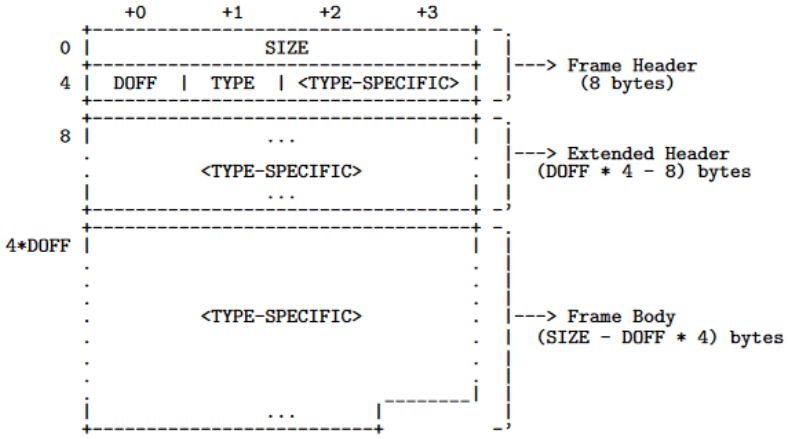

Figure 11: General frame

- SIZE: contiene la dimensione del frame.
- DOFF: contiene l'offset dei dati espresso in parole da 4 bytes.
- TYPE: contiene il tipo del frame, ad esempio 0x00 indica un frame AMQP.

I bytes 6 e 7 di un frame AMQP contengono il numero del canale. Nel body del frame troviamo la performative, ovvero l'operazione che si intende eseguire, seguito da un payload opaco.

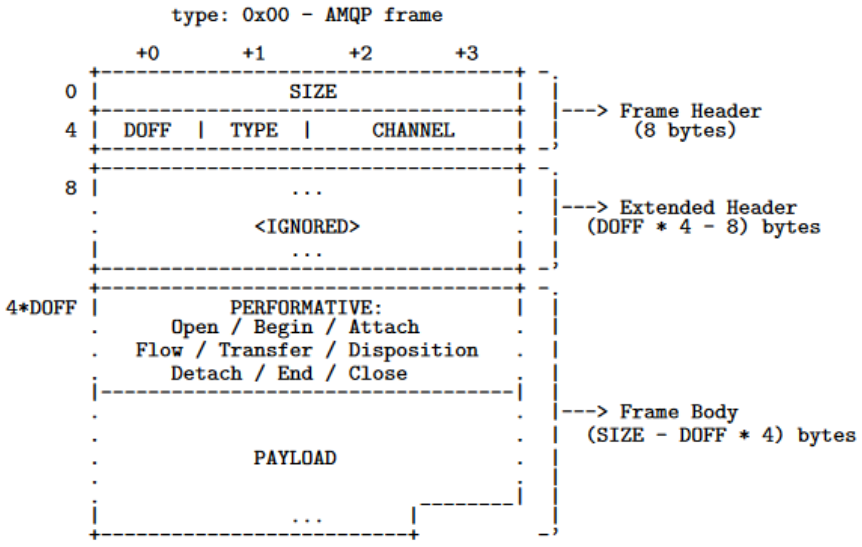

Figure 12: General frame

Open è il primo frame che deve essere inviato all'apertura di una connessione e contiene varie infromazioni tra cui la massima dimensione dei frames che verranno trasportati su quella connessione. **Close** chiude la connessione.

Begin viene utilizzato per iniziare una sessione su un canale mentre **End** per chiuderla.

Attach viene utilizzato per creare un link su una determinata sessione. Ad ogni link viene associato, oltre che un nome, un handle, un numero che verrà utilizzato nei transfer frames. **Detach** viene utilizzato per chiudere un link.

Transfer serve per inviare messaggi attraverso i links. È obbligatorio specificare l'handle del link sul quale vuole essere trasferito il messaggio.

Disposition può essere utilizzato per informare sullo stato della spedizione del messaggio.

Il payload contiene a tutti gli effetti i messaggi che vengono scambiati tra i nodi.

Un "annotated message" è composto da un bare message che è il messaggio effettivamente inviato più sezioni per le annotazioni.

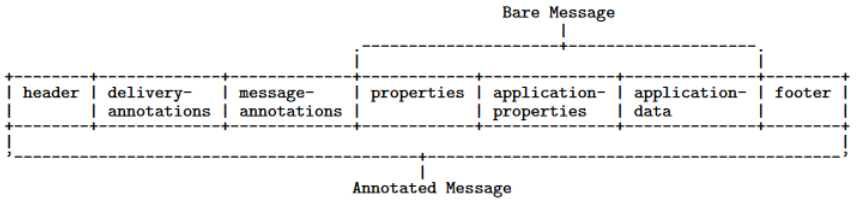

Figure 13: AMQP message

Il messaggio consiste di:

- Zero o un header.
- Zero o una delivery-annotations.
- Zero o una message-annotations.
- Zero o una properties.
- Zero o una application-properties.
- Il body consiste di una delle tre possibilità: una o più data sections, una o più AMQP sequence actions o una singola amqp-value section.
- Zero o un footer.

Un aspetto fondamentale è il fatto che il Bare Message è immutabile nella rete di nodi che attraversa il frame per raggiungere la destinazione. Ovvero i nodi intermedi non possono modificare in alcun modo il Bare Message. Le altre sezioni possono invece subire delle modifiche dai nodi intermedi.

Esiste un dissector di AMQP 1.0.0 in TCPDump, che ci consente di non dovere scrivere un parser.

Potremmo considerare, allora, i nodi intermedi come message brokers e dato un generico nodo verificare se si tratta o meno di un nodo intermedio in questo modo:

1. Considerare i frame AMQP in arrivo e in uscita dal nodo.
2. Calcolare la SHA-256 del Bare Message dei frame che contengono "Transfer" performative.
3. Verificare, data l'hash di un Bare Message di un frame in ingresso al nodo, se ne esiste una uguale di un Bare Message di un frame in uscita dal nodo.
4. Dato che il monitoring con TCPDump parte in qualunque momento mentre l'appicazione è in distribuzione, è necessario notare che se un nodo è intermedio potrei individuare un Bare Message in uscita che non è presente in ingresso e un Bare Message in ingresso che non è presente in uscita. Quindi è necessario che vi sia una corrispondenza tra hash in input ed in output non completa ma sopra una certa percentuale da stabilire.

TO-DO: Scrivere algoritmo che considera la frammentazione dei messaggi.
Qui è possibile trovare la documentazione ufficiale di questo protocollo.

4.2.3 Comunicazione con MQTT

In questo protocollo è presente la distinzione tra client e server.
Vediamo come è strutturato un pacchetto MQTT:

Fixed Header, present in all MQTT Control Packets
Variable Header, present in some MQTT Control Packets
Payload, present in some MQTT Control Packets

Figure 14: MQTT Control Packet

Il "Fixed Header" è così strutturato:

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Figure 15: Fixed Header

A seconda del "MQTT Control Packet Type" è possibile capire se il pacchetto va dal client al server o dal server al client.
Qui è possibile trovare la documentazione ufficiale di questo protocollo.

4.2.4 Comunicazione con STOMP

Con STOMP il meccanismo è equivalente a quello di MQTT.
I frames hanno questa struttura:

```
COMMAND
header1:value1
header2:value2

Body^@
```

Figure 16: STOMP Frame

A seconda del COMMAND è possibile capire se un pacchetto va dal client al server o viceversa.
Qui è possibile trovare la documentazione ufficiale di questo protocollo.

4.2.5 Comunicazione con protocollo proprietario

Da discutere. Si potrebbe creare un meccanismo affinchè possa essere definito il tipo dei messaggi? Vedi IDL.

4.3 Timeout - WORK IN PROGRESS

Il timeout è chiaramente utilizzato per comunicazioni di tipo sincrono. Analizziamo il meccanismo di rilevamento a seconda del protocollo di comunicazione utilizzato.

4.3.1 gRPC su HTTP/2

L'idea è quella di verificare se nella richiesta HTTP sia presente l'header "gRPC Timeout". Se tale header viene omesso, allora il server può assumere un timeout infinito. Ma è comunque probabile che, nonostante venga omesso, in realtà ci sia comunque una deadline lato client. Può accadere infatti che un metodo, pur avendo successo lato server, lato client fallisca con errore DEADLINE EXCEEDED perchè la deadline è scaduta prima di ottenere la risposta.

4.3.2 HTTP - TO-DO

4.4 Circuit Breaker - WORK IN PROGRESS

4.5 Database - TO-DO

4.6 Comunicazione cifrata: TLS o SSL - TO-DO

5 Low-level design - TO-DO