



**Politecnico
di Torino**

Politecnico di Torino

Corso di Laurea

A.a. 2025/2026

Sessione di laurea Mese Anno

Design ed Implementazione di una Libreria ad Alte Prestazioni per Data Pipeline in Go

Relatori:

Risso Fulvio

Candidati:

Omar Ferro

Sommario

Il presente lavoro di tesi si concentra sul design e l'implementazione di **Goccia**, una libreria ad alte prestazioni per la costruzione di *data pipeline* nel linguaggio Go. Go è un linguaggio di programmazione compilato sviluppato da Google nel 2009, caratterizzato da un modello di concorrenza basato su *goroutine* e *channel* che segue il principio “do not communicate by sharing memory; share memory by communicating”. Le goroutine sono unità leggere di esecuzione gestite dal runtime che consentono di scrivere codice concorrente efficiente, mentre i channel forniscono primitive di comunicazione sicure tra goroutine. Il linguaggio include inoltre supporto nativo per profiling, osservabilità e pacchetti essenziali per reti e operazioni atomiche, costituendo fondamenta robuste per la progettazione di infrastrutture di *data processing* ad alte prestazioni.

Nel contesto dei sistemi di elaborazione dati, una pipeline è una sequenza di stadi attraverso cui i messaggi fluiscono in modo ordinato, trasformandosi progressivamente da una rappresentazione grezza a una forma più strutturata e significativa dal punto di vista del dominio applicativo. Ogni stage ha responsabilità ben delimitata e definita: riceve input da stage precedenti o da sorgenti esterne, applica una trasformazione specifica, e produce un output che diventa l'ingresso dello stage successivo. Questa decomposizione architettonica promuove modularità, separazione delle responsabilità, riusabilità e facilita la comprensione e il testing di ciascun componente isolatamente. Dal punto di vista della concorrenza, il modello pipeline consente un notevole beneficio di throughput: ogni stage può operare in parallelo sugli elementi che riceve. Concretamente, mentre lo stage 1 elabora il messaggio N, lo stage 2 può già processare il messaggio N-1, e lo stage 3 il messaggio N-2. Questo parallelismo stadio-a-stadio incrementa significativamente il throughput complessivo del sistema rispetto a un'elaborazione sequenziale: invece di attendere che ogni messaggio attraversi tutta la catena prima di processarne uno nuovo, il sistema mantiene un “pipelining” attivo dove più messaggi sono elaborati contemporaneamente in stadi diversi. Un aspetto centrale riguarda i meccanismi di collegamento tra stage, poiché devono garantire sia la sicurezza rispetto alla concorrenza che la capacità di assorbire variazioni locali di carico. In particolare, l'impiego di buffer intermedi consente di mitigare la *backpressure*, il fenomeno per

cui uno stage più lento a valle rallenterebbe la produzione di uno stage più veloce a monte se non vi fossero buffer di accumulo. Un buffer consente temporaneamente di disaccoppiare la velocità di produzione da quella di consumo, riducendo l'impatto di variabilità di carico sulla latenza.

L'architettura di **Goccia** si fonda su due componenti essenziali: *Stage* e *Connector*. Ogni stage implementa tre metodi per gestirne il *lifecycle*: **Init** per l'inizializzazione delle risorse, **Run** per l'elaborazione dei dati, e **Close** per il rilascio delle risorse. Gli stage vengono eseguiti in almeno una goroutine dedicata, permettendo l'esecuzione parallela degli stadi della pipeline. Gli stage sono categorizzati in tre gruppi principali: *Ingress Stage*, che rappresentano i punti di ingresso dei dati nella pipeline; *Processor Stage*, che implementano trasformazioni modulari sui messaggi; ed *Egress Stage*, che rappresentano i punti di uscita della pipeline, persistendo o trasmettendo messaggi verso destinazioni esterne. Un aspetto distintivo di **Goccia** riguarda l'implementazione dei *Connector*, ovvero i buffer frapposti tra gli stage. La libreria utilizza *ring buffer lock-free*, in luogo dei *channel* nativi di Go. Questa scelta progettuale mira a massimizzare il throughput in scenari ad alto carico, implementando tre varianti: SPSC (*Single Producer Single Consumer*), utilizzato per la comunicazione tra stage consecutivi; SPMC (*Single Producer Multiple Consumer*), impiegato per operazioni di *fan-out*; e MPSC (*Multiple Producer Single Consumer*), per operazioni di *fan-in*.

L'implementazione dei ring buffer si basa su operazioni atomiche *Compare-and-Swap* (CAS), che consentono di modificare una locazione di memoria in modo atomico a livello CPU, eliminando condizioni di *data race* senza ricorrere a lock espliciti. Un elemento cruciale è il padding delle cache line tramite `cpu.CacheLinePad`. Nelle architetture x86-64, le cache line hanno dimensione di 64 byte: quando la CPU accede a un dato in memoria, l'intero blocco di 64 byte contenente quel dato viene caricato nella cache locale del core. In contesti concorrenti, questo può generare il fenomeno del *false sharing*, che si verifica quando due thread accedono a variabili distinte che però risiedono nella stessa cache line: il protocollo di coerenza della cache costringe l'intera cache line a essere invalidata e ricaricata ogni volta che uno dei due thread la modifica, anche se le variabili non condividono logicamente dati. Separando `head` e `tail` tramite padding, si garantisce che i due indici risiedano su cache line diverse, eliminando la contesa e migliorando il throughput di un ordine di grandezza in scenari di alta contesa multicore. Le operazioni di enqueue e dequeue adottano una strategia ibrida: inizialmente tentano l'operazione con spinning limitato; quindi, si bloccano su una *condition variable* se il buffer rimane pieno o vuoto, combinando l'efficienza del *busy-waiting* per brevi contese con il blocco efficiente per periodi prolungati.

Gli *Ingress Stage* rappresentano i punti di ingresso dei dati nella pipeline. **Goccia** fornisce diverse implementazioni: *Ticker Stage*, che genera messaggi periodici sfruttando `time.Ticker`, utile per testing e benchmarking; *UDP Stage*,

che riceve datagrammi UDP dalla rete, implementando una strategia di buffering con size configurabile e utilizzando un `sync.Pool` per il riuso della memoria, riducendo la pressione sul *garbage collector*; *TCP Stage*, che gestisce connessioni TCP persistenti, supportando due modalità di framing (*Delimited* e *Length-Prefixed*) e implementando un'architettura multi-goroutine, con una goroutine principale che accetta connessioni e una goroutine dedicata per ciascun client connesso; *Kafka Stage*, che consuma messaggi da topic Kafka, integrandosi con architetture *event-driven*; *eBPF Stage*, che si collega alla map ring buffer dei programmi eBPF, permettendo l'acquisizione di eventi di basso livello come syscall e pacchetti di rete; e *File Stage*, che legge dati da file su disco, supportando diverse modalità di lettura.

I *Processor Stage* rappresentano unità di elaborazione intermedia, implementando trasformazioni modulari sui messaggi. Esempi significativi includono *Cannelloni Decoder/Encoder*, specializzati per la serializzazione e deserializzazione di messaggi CAN nel formato Cannelloni, uno standard di incapsulamento per il trasporto di messaggi CAN su reti IP; *CAN Processor*, che decodifica messaggi CAN grezzi in strutture tipizzate; *Filter Stage*, che filtra messaggi in base a predicati definiti dall'utente; *Tee Stage*, che duplica il flusso di messaggi verso molteplici connettori di output, abilitando il pattern *fan-out*; *Reorder Buffer* (ROB), che riordina messaggi fuori sequenza in base ad un sequence number e ne stima il timestamp tramite il metodo di Holt per il *Double Exponential Smoothing*; e *Custom Stage*, che consente agli utenti di implementare logica di elaborazione arbitraria tramite l'implementazione di un'interfaccia personalizzata `CustomHandler`, fornendo un'astrazione generica che delega completamente la logica di processing all'utente.

Il metodo di Holt, introdotto nel 1957, estende il *simple exponential smoothing* per permettere la previsione e l'analisi di serie temporali che presentano un andamento di tendenza lineare. Mentre il *simple exponential smoothing* si limita a modellare una componente di livello costante e risulta inadatto a serie con trend manifesto, il metodo di Holt introduce una componente aggiuntiva dedicata alla stima della tendenza (slope), rendendo il modello maggiormente adatto all'elaborazione di dati con comportamento non stazionario. Nel contesto di **Goccia**, il metodo di Holt è impiegato per stimare e correggere il timestamp di messaggi ricevuti, in particolare quando i messaggi arrivano fuori ordine o affetti da jitter. Il metodo fornisce un meccanismo efficiente per separare la componente di livello (il timestamp smussato, depurato dal jitter) dalla componente di trend (il rate di crescita sistematico dei receive time). Poiché i timestamp di ricezione seguono un trend lineare crescente, il metodo di Holt è particolarmente adatto a catturare questa dinamica e a fornire stime robuste nonostante le fluttuazioni dovute al jitter di rete.

Gli *Egress Stage* rappresentano i punti di uscita della pipeline, persistendo o trasmettendo messaggi verso destinazioni esterne. **Goccia** fornisce diverse implementazioni: *UDP/TCP Egress*, che inviano messaggi verso endpoint remoti

via protocolli UDP o TCP; *Kafka Egress*, che pubblica messaggi su topic Kafka, con supporto per tracciamento distribuito tramite propagazione del “contesto” OpenTelemetry negli header dei messaggi di Kafka; *File Egress*, che persiste messaggi su disco in modalità *append-only*, con buffering configurabile e flushing basato su soglie di riempimento o deadline temporali; *QuestDB Egress*, che inserisce dati in QuestDB, un database *time-series* colonnare progettato specificamente per la gestione di flussi di dati indicizzati temporalmente, gestendo automaticamente pooling di connessioni e flushing; e *Sink Egress*, che consuma e distrugge messaggi senza persistenza, utilizzato per testing e benchmarking.

Goccia implementa un *Worker Pool* per la gestione dinamica della concorrenza, composto da code di *fan-out* (distribuzione task ai worker), code di *fan-in* (raccolta risultati), uno *Scaler* per l'autoscaling, mentre i *Worker* per l'elaborazione effettiva. Lo *Scaler* implementa un algoritmo adattivo che regola il numero di worker attivi in funzione della profondità della coda, con parametri configurabili. La riduzione dei worker (scale-down) adotta un meccanismo di *exponential backoff* per prevenire oscillazioni eccessive. Il *Worker Pool* è utilizzato sia dai *Processor Stage* che dagli *Egress Stage* quando configurati in “running mode” *Worker Pool*, garantendo scalabilità dinamica e throughput elevato tramite meccanismi di *fan-out*, *fan-in* e autoscaling adattivo.

Il quinto capitolo della tesi descrive l'implementazione di un sistema di telemetria automobilistica per SquadraCorse, che rappresenta il principale caso d'uso che ha motivato lo sviluppo di **Goccia**. Il sistema è organizzato in layer distinti: *Ingestion Layer*, che acquisisce dati CAN dal bus del veicolo via UDP, utilizzando il protocollo Cannelloni; *Processing Layer*, che utilizza effettivamente la libreria **Goccia**, decodifica i messaggi CAN, applica correzioni di timestamp e riordina i messaggi; *Storage Layer*, che persiste i dati telemetrici in QuestDB per analisi real-time; *Observability Layer*, che integra OpenTelemetry per tracciamento distribuito, raccogliendo metriche e log strutturati; e *Visualization Layer*, che presenta i dati tramite dashboard Grafana, permettendo il monitoraggio in tempo reale di pressioni, tensioni, temperature e stati delle macchine a stati finiti del veicolo.

Il capitolo 6 delinea le direzioni di evoluzione futura della libreria, includendo il supporto a nuovi protocolli di trasporto (QUIC per comunicazioni a bassa latenza, MQTT per IoT, FastHTTP per servizi REST ad alte prestazioni), l'implementazione di pattern avanzati (*Request-Reply* con *Futures*, *Content-Based Routing*), l'estendibilità con *Custom Ingress/Egress* e supporto a InfluxDB, e il miglioramento della test coverage.

Goccia rappresenta una soluzione completa per la costruzione di *data pipeline* ad alte prestazioni in Go, combinando i vantaggi del modello di concorrenza di Go con tecniche avanzate di sincronizzazione *lock-free*. L'architettura modulare e l'impiego di ring buffer ottimizzati consentono di raggiungere throughput elevati in scenari multicore, mentre l'integrazione nativa con OpenTelemetry garantisce

osservabilità completa. Il caso d'uso di SquadraCorse dimostra l'applicabilità della libreria in contesti real-time, dove l'elaborazione di dati telemetrici richiede latenze ridotte e affidabilità. Gli sviluppi futuri mirano a estendere le capacità della libreria con nuovi protocolli e pattern architetturali, consolidando **Goccia** come framework di riferimento per *data pipeline* in Go.

Ringraziamenti

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tristique senectus et netus et malesuada fames ac turpis. Pretium nibh ipsum consequat nisl vel pretium lectus quam. Urna molestie at elementum eu facilisis sed odio morbi quis. Sed egestas egestas fringilla phasellus faucibus scelerisque eleifend. At in tellus integer feugiat. Mauris rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar. Commodore sed egestas egestas fringilla. Nunc lobortis mattis aliquam faucibus purus in massa. Facilisis magna etiam tempor orci eu lobortis elementum nibh. Elementum curabitur vitae nunc sed velit dignissim. Neque volutpat ac tincidunt vitae. Massa id neque aliquam vestibulum morbi blandit cursus risus at. Porta non pulvinar neque laoreet suspendisse interdum consectetur. Turpis in eu mi bibendum. Ut tristique et egestas quis ipsum suspendisse. Integer quis auctor elit sed vulputate mi sit amet. Viverra nam libero justo laoreet sit amet cursus.

Indice

Elenco delle figure	XII
1 Introduzione	1
1.1 Motivazioni	1
1.2 Contesto applicativo	2
1.3 Obiettivi	2
1.4 Contributi principali	3
1.5 Struttura del documento	3
2 Fondamenti teorici e tecnologie	5
2.1 Linguaggio Go	6
2.2 Modello pipeline	7
2.3 Lock-free data structures	8
2.4 Ring buffer	9
2.5 Controller Area Network (CAN bus)	11
2.6 Apache Kafka	12
2.7 eBPF	13
2.8 QuestDB	14
2.9 OpenTelemetry	15
2.10 Il Metodo di Holt per il <i>Double Exponential Smoothing</i>	16
2.10.1 Fondamenti Teorici e Motivazione	16
2.10.2 Formulazione Matematica	16
2.10.3 Interpretazione Componenziale	17
2.10.4 Condizioni Iniziali	17
2.10.5 Correzione dei <i>Timestamp</i> di Ricezione	18
2.10.6 Applicazioni Pratiche	19
3 Architettura e implementazione della libreria Goccia	20
3.1 Pipeline	20
3.2 Stage	21
3.3 Connector	22

3.4	Ingress Stage	25
3.4.1	Ticker	25
3.4.2	UDP	26
3.4.3	TCP	28
3.4.4	Kafka	33
3.4.5	eBPF	34
3.4.6	File	37
3.5	Worker Pool	40
3.6	Processor Stage	43
3.6.1	Cannelloni	43
3.6.2	CAN	45
3.6.3	CSV	46
3.6.4	Custom	47
3.6.5	Filter	49
3.6.6	Tee	49
3.6.7	Reorder Buffer	50
3.7	Egress Stage	54
3.7.1	UDP Egress	55
3.7.2	TCP Egress	55
3.7.3	Kafka Egress Stage	56
3.7.4	File Egress Stage	57
3.7.5	QuestDB Egress Stage	58
3.7.6	Sink Egress Stage	59
4	Benchmarks della libreria Goccia	60
4.1	Micro-benchmarking: Analisi dei Connettori Interni	61
4.1.1	Metodologia e setup sperimentale	61
4.1.2	Analisi in regime stazionario	62
4.1.3	Scalabilità in scenari di contesa	63
4.1.4	Discussione architetturale	64
5	Implementazione del sistema di telemetria automobilistica di SquadraCorse	66
5.1	Ingestion Layer	68
5.2	Processing Layer	70
5.3	Storage Layer	72
5.4	Observability Layer	75
5.5	Visualization Layer	77

6	Sviluppi futuri	80
6.1	Evoluzione dei protocolli di trasporto	80
6.1.1	Supporto al protocollo QUIC	80
6.1.2	Integrazione del protocollo MQTT	81
6.1.3	Servizi REST ad alte prestazioni con FastHTTP	81
6.2	Pattern avanzati di Routing e Flusso	81
6.2.1	Implementazione del pattern Request-Reply (Futures)	81
6.2.2	Content-Based Routing	82
6.3	Estendibilità e Storage	82
6.3.1	InfluxDB Egress Stage	82
6.3.2	Custom Ingress ed Egress	82
6.4	Qualità e Affidabilità	83
6.4.1	Miglioramento della Test Coverage	83
7	Conclusioni	84
	Bibliografia	85

Elenco delle figure

3.1	Struttura generale di una pipeline in Goccia	21
3.2	Worker Pool nella libreria Goccia	42
3.3	Rappresentazione di un frame del protocollo Cannelloni UDP	44
5.1	Stage di Goccia usati in sc-telemetry	71
5.2	Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (pressioni)	78
5.3	Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (tensioni/temperature)	78
5.4	Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (FSM)	79

Capitolo 1

Introduzione

Negli ultimi anni l'adozione di architetture *event-driven* e di sistemi di *stream processing* ha reso centrale il problema di costruire pipeline di dati in grado di coniugare semplicità d'uso, modularità e prestazioni elevate su architetture multicore. In questo contesto, il linguaggio Go offre primitive di concorrenza ad alto livello basate su *goroutine* e *channel*, ma tali astrazioni, pur risultando estremamente espressive, non sempre consentono di sfruttare appieno l'hardware nel caso di applicazioni con requisiti stringenti di throughput e latenza. La libreria **Goccia** nasce con l'obiettivo di colmare questo divario, fornendo una soluzione per la definizione di pipeline concorrenti in Go che integri un modello a stadi chiaro con connettori interni ottimizzati, basati su *ring buffer lock-free* e tecniche di riduzione della contesa in cache, e viene utilizzata all'interno di un sistema reale di telemetria automobilistica sviluppato per SquadraCorse Polito.

1.1 Motivazioni

Molti sistemi moderni di telemetria, osservabilità e *data ingestion* richiedono la gestione continua di flussi di messaggi eterogenei, provenienti da reti veicolari, sensori o microservizi, con vincoli di affidabilità e di latenza *end-to-end* che rendono critico il layer di trasporto e di buffering dei dati. L'impiego di primitive generiche per la comunicazione concorrente, come i soli *channel* di Go, può introdurre overhead di sincronizzazione e fenomeni di contesa che, in scenari di traffico elevato o in presenza di più core, rendono il layer di trasporto un potenziale collo di bottiglia rispetto alla logica applicativa. Diventa pertanto necessario disporre di una libreria infrastrutturale che esponga un'API di alto livello per la modellazione di pipeline, ma che internamente adotti strutture dati specializzate, come *ring buffer lock-free*, per ridurre al minimo le penalità introdotte dalla sincronizzazione, cercando allo stesso tempo di distribuire il lavoro su tutti i core disponibili.

1.2 Contesto applicativo

Il caso d'uso scelto per valutare **Goccia** è il sistema di telemetria in tempo reale del prototipo di Formula SAE di SquadraCorse Polito, organizzato secondo un'architettura a più livelli che separa nettamente *ingestion*, *processing*, *storage*, osservabilità e visualizzazione. I messaggi provenienti dalle due linee CAN della vettura vengono incapsulati secondo il protocollo Cannelloni, trasportati su datagrammi UDP attraverso una VPN verso un server centrale e quindi elaborati da una pipeline costruita con **Goccia**, che decodifica i frame CAN e li trasforma in segnali ad alto livello prima della persistenza in un database *time-series* ad alte prestazioni come QuestDB. A corredo, il sistema integra un *Observability Layer* basato su OpenTelemetry Collector, Prometheus e Grafana Tempo, e un *Visualization Layer* centrato su Grafana e Caddy, che consentono di monitorare simultaneamente sia lo stato della vettura sia il comportamento interno della pipeline, offrendo un banco di prova realistico per le capacità della libreria.

1.3 Obiettivi

L'obiettivo generale del lavoro è progettare e implementare una libreria per *data pipeline* in Go che renda esplicito il modello a stadi, fornendo un orchestratore centrale e un'interfaccia **Stage** unificata, con metodi che ne determinano il *lifecycle* (**Init**, **Run**, **Close**), ben definiti e facilmente estendibili. Sul piano architetturale, la libreria deve permettere di comporre pipeline costituite da stage di tipo *Ingress*, *Processor* ed *Egress*, collegati tramite un'astrazione, il *Connector*, che incapsula i dettagli delle strutture dati concorrenti, garantendo *thread-safety*, gestione della *backpressure* e terminazione ordinata tramite **context**. Un ulteriore obiettivo è dimostrare che, grazie all'impiego di *ring buffer lock-free* e a scelte implementative orientate alla località dei dati, **Goccia** è in grado di superare soluzioni basate unicamente sui *channel* standard, riducendo la latenza e aumentando il throughput.

Più in dettaglio, il lavoro si propone di: definire un'API pubblica che consenta di configurare pipeline e stage tramite i tipi *generics* di Go, mantenendo il controllo statico sui tipi dei messaggi lungo la catena di elaborazione, e riducendo la necessità di conversioni dinamiche a runtime. In secondo luogo, mira a progettare e implementare diverse famiglie di *Connector* e *ring buffer* interni (SPSC per la comunicazione uno-a-uno tra stage consecutivi, SPMC per il *fan-out* nel *worker pool*, MPSC per il *fan-in*), integrando strategie ibride di *backpressure* che combinano fasi di *spinning* controllato con il blocco su *condition variable* in presenza di contesa prolungata. Infine, si pone l'obiettivo di applicare la libreria a un caso reale, la telemetria di SquadraCorse, e offrire altre integrazioni con tecnologie come Kafka,

eBPF, QuestDB e OpenTelemetry, al fine di verificarne non solo le prestazioni ma anche la capacità di inserirsi in un ecosistema *cloud-native* complesso.

1.4 Contributi principali

Il primo contributo è la definizione di un modello di pipeline coerente e riusabile in Go, centrato su una struttura “orchestratrice” e su stage con responsabilità ben delimitate, classificati in *Ingress*, *Processor* ed *Egress* per riflettere le fasi tipiche dei sistemi di *stream processing*. Il secondo contributo è l’introduzione di connettori ad alte prestazioni basati su *ring buffer lock-free*, inclusi adattamenti per *fan-out* e *fan-in* tramite varianti SPMC e MPSC, e con scelte implementative orientate alla scalabilità su architetture multicore (operazioni atomiche, padding per evitare *false sharing*). Il terzo contributo è la disponibilità di un’API documentata e riutilizzabile come libreria, pubblicata su GitHub [1] e consultabile anche tramite la documentazione di `pkg.go.dev` [2]. Il quarto contributo è la valutazione sperimentale dei connettori interni: i risultati riportati mostrano miglioramenti di latenza e speedup che, a seconda di architettura e pattern di contesa, arrivano tipicamente nell’ordine di circa $1.4\times-3.0\times$ rispetto a baseline basate su *channel* in scenari rappresentativi.

1.5 Struttura del documento

Il Capitolo 2 introduce i fondamenti teorici e le tecnologie utilizzate: vengono presentati il linguaggio Go e il suo modello di concorrenza, il paradigma pipeline, le strutture dati *lock-free* e i *ring buffer*, oltre alle tecnologie specifiche del caso d’uso, quali CAN bus, Apache Kafka, eBPF, QuestDB e OpenTelemetry, insieme al metodo di Holt per la correzione del jitter temporale. Il Capitolo 3 descrive in dettaglio l’architettura e l’implementazione della libreria **Goccia**, illustrando la `struct Pipeline`, l’interfaccia `Stage`, le categorie di stage (*Ingress*, *Processor*, *Egress*), il *worker pool* e i connettori interni basati su *ring buffer*, con particolare attenzione alle scelte progettuali orientate alle prestazioni e alla gestibilità del ciclo di vita.

Il Capitolo 4 è dedicato ai benchmark della libreria, con la definizione della metodologia sperimentale, l’analisi del comportamento in regime stazionario e sotto contesa, e una discussione architetturale dei risultati ottenuti, mentre il Capitolo 5 presenta l’implementazione completa del sistema di telemetria automobilistica di SquadraCorse, organizzato in *ingestion*, *processing*, *storage*, *observability* e *visualization layer*. Il Capitolo 6 illustra possibili sviluppi futuri della libreria, tra cui il supporto a protocolli di trasporto moderni come QUIC e MQTT, pattern

avanzati di routing e nuove destinazioni di storage, e il Capitolo 7 conclude il lavoro, sintetizzandone i risultati e delineando le principali direzioni di evoluzione.

Capitolo 2

Fondamenti teorici e tecnologie

2.1 Linguaggio Go

Go è un linguaggio di programmazione compilato, tipizzato staticamente e con gestione automatica della memoria, sviluppato da Google nel 2009 con l'obiettivo di semplificare lo sviluppo di software di sistema concorrente e distribuito su larga scala. La sintassi intenzionalmente essenziale—inspirata a C ma con numerose semplificazioni—favorisce leggibilità e manutenibilità del codice in ambienti industriali [3]. Caratteristiche distintive includono l'assenza di ereditarietà classica, rimpiazzata da composizione e da un modello ad interfacce strutturale più flessibile.

Uno degli elementi più distintivi di Go è il suo modello di concorrenza, basato su *goroutine* e *channel*, che incapsula il principio fondamentale “do not communicate by sharing memory; share memory by communicating” [4]. Le goroutine sono unità leggere di esecuzione gestite direttamente dal runtime, non mappate uno-a-uno sui thread del sistema operativo: il runtime mantiene internamente uno scheduler che multiplessa migliaia (o milioni) di goroutine su un numero limitato di thread, consentendo di scrivere codice concorrente senza la complessità tradizionale della gestione manuale di thread. I channel sono primitive di comunicazione e sincronizzazione tipizzate che consentono il passaggio di messaggi in modo sicuro tra goroutine, eliminando la necessità di mutex e condizioni di race quando utilizzati correttamente.

Il linguaggio Go fornisce inoltre supporto nativo per strumenti di profiling e osservabilità (runtime statistics, memory profiling, CPU profiling) e include nella libreria standard pacchetti essenziali per reti (**net**), gestione del tempo (**time**), operazioni atomiche (**sync/atomic**) e primitivi di sincronizzazione (**sync**), costituendo fondamenta robuste per la progettazione di infrastrutture di data processing ad alte prestazioni. La compilazione Go produce un singolo eseguibile binario, senza dipendenze runtime esterne, facilitando il deployment in ambienti containerizzati e cloud-native. Infine, il supporto nativo a test e benchmarking tramite il package **testing** rende naturale la pratica del test-driven development.

2.2 Modello pipeline

Nel contesto dei sistemi di elaborazione dati, una pipeline è una sequenza di stadi (stage) attraverso cui i messaggi fluiscono in modo ordinato, trasformandosi progressivamente da una rappresentazione grezza (raw data) a una forma più strutturata e significativa dal punto di vista del dominio applicativo. Ogni stage ha responsabilità ben delimitata e definita: riceve input da stage precedenti o da sorgenti esterne, applica una trasformazione specifica, e produce un output che diventa l'ingresso dello stage successivo. Questa decomposizione architettonica promuove modularità, separazione delle responsabilità, riusabilità e facilita la comprensione e il testing di ciascun componente isolatamente [5].

Dal punto di vista della concorrenza, il modello pipeline consente un notevole beneficio di throughput: ogni stage può operare in parallelo sugli elementi che riceve. Concretamente, mentre lo stage 1 elabora il messaggio N, lo stage 2 può già processare il messaggio N-1, e lo stage 3 il messaggio N-2. Questo parallelismo stadio-a-stadio incrementa significativamente il throughput complessivo del sistema rispetto a un'elaborazione sequenziale: invece di attendere che ogni messaggio attraversi tutta la catena prima di processarne uno nuovo, il sistema mantiene un "pipelining" attivo dove più messaggi sono elaborati contemporaneamente in stadi diversi.

Un aspetto centrale riguarda i meccanismi di collegamento tra stage, poiché devono garantire sia la sicurezza rispetto alla concorrenza (thread-safety) sia la capacità di assorbire variazioni locali di carico. In particolare, l'impiego di buffer intermedi (code) consente di mitigare la *backpressure*, il fenomeno per cui uno stage più lento a valle "rallenterebbe" la produzione di uno stage più veloce a monte se non vi fossero buffer di accumulo. Un buffer consente temporaneamente di disaccoppiare la velocità di produzione da quella di consumo, riducendo l'impatto di variabilità di carico sulla latenza end-to-end [6]. Per massimizzare le prestazioni in scenari ad alto throughput, i connettori tra stage devono essere implementati con strutture dati lock-free, come ring buffer specializzati, che sostituiscono i meccanismi tradizionali di sincronizzazione.

2.3 Lock-free data structures

Le strutture dati *lock-free* sono progettate per consentire l'accesso concorrente da parte di molteplici thread senza ricorrere a primitive di sincronizzazione bloccanti come mutex o semafori, affidandosi invece a operazioni atomiche e a protocolli non bloccanti basati su hardware [7]. In ambienti multicore, questo approccio riduce drasticamente la contesa sulle risorse, limita il context switching (scambio di contesto) tra thread, e migliora la scalabilità del sistema al crescere sia del numero di core che del carico di lavoro.

La fondazione delle strutture lock-free è l'operazione *Compare-and-Swap* (CAS). Una operazione CAS è un'istruzione atomica a livello hardware che, in un'unica fase indivisibile, legge una locazione di memoria, confronta il suo valore con un valore atteso e, solo se coincidono, lo sostituisce con un nuovo valore, ritornando il risultato del confronto [8]. Formalmente: `CAS(addr, atteso, nuovo)` legge il valore in `addr`, se è uguale ad `atteso` lo sostituisce con `nuovo` e ritorna vero, altrimenti non modifica nulla e ritorna falso. Poiché l'intera operazione è atomica a livello CPU, nessun altro thread può leggere o modificare `addr` durante l'esecuzione di CAS, eliminando condizioni di race intrinseche.

Le operazioni atomiche rappresentano una alternativa ai lock bloccanti: invece di acquisire un lock e mantenere un diritto esclusivo su una risorsa per la durata di una sezione critica, un thread modifica la risorsa tramite CAS e, se rileva che il valore è cambiato da quando l'ha letto (indicando interferenza da parte di un altro thread), ritenta l'operazione. Questo schema funziona bene quando la contesa è ridotta e i conflitti sono rari, offrendo latenza inferiore rispetto ai lock bloccanti in tali scenari [9]. La proprietà lock-free garantisce inoltre che, indipendentemente dallo scheduling dei thread, almeno un thread farà sempre progressi verso il completamento della propria operazione, anche se altri thread vengono sospesi o rallentati.

2.4 Ring buffer

Il *ring buffer* (o *circular buffer*) è una coda circolare che utilizza un array di dimensione fissa e due indici (**head** e **tail**) per tracciare le posizioni di scrittura e lettura, aggiornati in modo ciclico tramite aritmetica modulo la capacità del buffer [10]. Quando **head** raggiunge la fine dell'array, si avvolge al principio, creando una struttura ciclica che riutilizza la memoria della stessa allocazione senza necessità di reallocazione o spostamento di elementi. Questa proprietà è fondamentale per le applicazioni di streaming: invece di accodare elementi a una fine e deaccodarli dall'altra (con relativa necessità di shift), il ring buffer semplicemente avanza i due indici in modo circolare, mantenendo complessità $O(1)$ per enqueue e dequeue [11].

Per ottimizzare le prestazioni su architetture multicore, gli indici **head** e **tail** sono rappresentati mediante tipi atomici, garantendo che le operazioni di lettura e incremento siano atomiche a livello CPU e visibili a tutti i core senza necessità di lock espliciti. Inoltre, la capacità del buffer è tipicamente arrotondata alla potenza di due più vicina, consentendo di sostituire l'operazione di modulo (costosa, richiedendo una divisione) con una maschera bit-a-bit ($\text{index} \& \text{capMask}$), dove $\text{capMask} = \text{capacity} - 1$. Ad esempio, se la capacità è 256 (2^8), capMask è 255 (0xFF in binario), e calcolare $(\text{head} + 1) \& 255$ è equivalente a $(\text{head} + 1) \% 256$ ma richiede solo un'operazione bitwise anziché una divisione.

Un aspetto cruciale dell'implementazione di ring buffer concorrenti è il sistematico impiego del *padding di cache line*. Nelle architetture moderne x86-64, le cache line hanno tipicamente una dimensione di 64 byte. Quando la CPU accede a un dato in memoria, l'intero blocco di 64 byte contenente quel dato viene caricato nella cache locale del core. In contesti concorrenti, questo può generare un fenomeno noto come *false sharing*: quando due thread accedono a variabili distinte che però risiedono nella stessa cache line, il protocollo di coerenza della cache costringe l'intera cache line a essere invalidata e ricaricata ogni volta che uno dei due thread la modifica, anche se le variabili non condividono logicamente dati [12].

In un ring buffer concorrente, dove producer (chi scrive) e consumer (chi legge) operano rispettivamente su **head** e **tail**, il false sharing comporterebbe un overhead catastrofico: ogni scrittura su **head** da parte del producer invaliderebbe la cache line contenente **tail** nei core dei consumer, e viceversa, causando centinaia di cicli di latenza aggiuntivi per ogni accesso. Separando **head** e **tail** tramite padding (ovvero inserendo campi dummy di 64 byte tra loro), si garantisce che i due indici risiedano su cache line diverse, eliminando completamente la contesa. Studi empirici hanno dimostrato che questo padding può migliorare il throughput di un ordine di grandezza in scenari di alta contesa multicore.

I ring buffer specializzati utilizzati sono:

- **SPSC (Single Producer Single Consumer)**: Un singolo thread scrive, un

singolo legge. Non è necessaria sincronizzazione ulteriore oltre alle barriere di memoria fornite dalle operazioni atomiche. È la variante più semplice e performante.

- **SPMC (Single Producer Multiple Consumer)**: Un singolo thread scrive, molteplici leggono. Richiede coordinamento affinché ogni dato sia consumato una sola volta, tipicamente tramite flag atomici e operazioni compare-and-swap.
- **MPSC (Multiple Producer Single Consumer)**: Molteplici thread scrivono, un singolo legge. Simmetricamente a SPMC, richiede coordinamento sulla scrittura tramite compare-and-swap per determinare quale producer ha il diritto di scrivere nella prossima posizione.

Le operazioni di enqueue (scrittura) e dequeue (lettura) sui ring buffer adottano tipicamente una *strategia ibrida di backpressure* [13]: inizialmente tentano l'operazione tramite un numero limitato di tentativi (spinning), cedendo brevemente la goroutine tra un tentativo e l'altro per evitare busy-waiting puro. Se dopo la fase di spinning il buffer rimane pieno (per enqueue) o vuoto (per dequeue), il thread si blocca su una condition variable fino a quando non viene segnalato spazio disponibile o il buffer viene chiuso. Questo approccio ibrido combina l'efficienza del busy-waiting per brevi periodi di contesa con il blocco efficiente per periodi prolungati, riducendo il consumo di CPU in scenari di alta pressione.

2.5 Controller Area Network (CAN bus)

Il Controller Area Network (CAN) è uno standard di comunicazione seriale progettato originariamente da Bosch negli anni '80 per collegare in modo efficiente le centraline elettroniche (ECU) all'interno dei veicoli, riducendo la complessità del cablaggio grazie a un'architettura di tipo bus condiviso. Il protocollo è stato successivamente standardizzato dall'ISO nella famiglia di norme ISO 11898, che ne descrivono il livello data link e il livello fisico secondo il modello OSI [14]. A differenza di altre reti, il CAN adotta un paradigma multi-master e broadcast: ogni nodo può trasmettere sul bus e tutti i nodi ricevono ogni frame, decidendo localmente se elaborarlo o ignorarlo in base all'identificatore del messaggio (CAN ID). Questa modalità si adatta in modo naturale a scenari real-time in cui molteplici centraline devono condividere il loro stato.

Sul piano fisico, il CAN bus utilizza una coppia di fili intrecciati (CAN_H e CAN_L) e una codifica differenziale per aumentare la robustezza al rumore elettromagnetico. Lo stato logico dominante (bit 0) è rappresentato da una differenza di tensione di circa 2,5 V tra le due linee, mentre lo stato recessivo (bit 1) corrisponde a una condizione in cui entrambe le linee sono riportate a un livello intermedio tramite resistenze passive. L'utilizzo di segnali differenziali, in combinazione con le resistenze di terminazione da 120 Ω alle estremità del bus, consente di raggiungere distanze dell'ordine di decine o centinaia di metri con velocità fino a 1 Mbit/s (CAN 2.0) o fino a 5 Mbit/s nel caso di CAN FD (Flexible Data-rate).

2.6 Apache Kafka

Apache Kafka è una piattaforma di streaming distribuito nata originariamente in LinkedIn e successivamente resa open source, progettata per fungere da log distribuito ad alto throughput e da sistema di messaggistica publish/subscribe scalabile e fault-tolerant [15]. Concettualmente, Kafka organizza i dati in topic, a loro volta suddivisi in partizioni: ciascuna partizione è implementata come un log append-only ordinato, memorizzato su disco, in cui i record sono identificati da un offset progressivo. Questa astrazione di log distribuito consente di gestire in modo naturale flussi di eventi temporali, come stream di dati, metriche o log applicativi.

I produttori (producer) scrivono messaggi in append nelle partizioni dei topic, mentre i consumatori (consumer) leggono sequenzialmente i messaggi mantenendo localmente il proprio offset di lettura. La separazione tra il log persistente sul broker e lo stato di consumo lato client permette di avere molteplici consumer group che rileggono gli stessi dati in momenti diversi. Kafka garantisce, per ogni partizione, l'ordinamento totale dei messaggi, mentre la scalabilità orizzontale è ottenuta distribuendo le partizioni su un cluster di broker con meccanismi di replica per la tolleranza ai guasti.

Dal punto di vista dello storage, ogni partizione è fisicamente rappresentata da una directory che contiene una sequenza di segmenti di log (file .log) e relativi indici (offset index, time index). Quando un segmento raggiunge una certa dimensione o anzianità, viene “ruotato” e ne viene creato uno nuovo. Questa organizzazione facilita sia le politiche di retention (delete o compact) sia l'accesso sequenziale ad alta efficienza, poiché le letture dei consumer si concentrano spesso sui segmenti più recenti.

Per quanto riguarda le garanzie di consegna, Kafka supporta diverse semantiche (at-most-once, at-least-once, exactly-once) a seconda della configurazione dei producer, dei consumer e delle applicazioni di stream processing a valle. L'integrazione con sistemi come Kafka Streams, Flink o Spark Streaming consente di costruire pipeline di elaborazione stateful strettamente integrate con il log sottostante. In contesti di stream processing, Kafka viene spesso utilizzato come buffer tra sorgenti rumoouse/ad alto volume (sensori, microservizi, gateway di bordo) e sistemi di persistenza o analisi batch, permettendo l'implementazione di architetture scalabili e resilienti.

2.7 eBPF

L'eBPF (extended Berkeley Packet Filter) è una tecnologia che permette l'esecuzione di programmi in un contesto privilegiato all'interno del kernel Linux, in modo sicuro e controllato, senza dover modificare il sorgente del kernel o caricare moduli kernel tradizionali [16]. Originariamente derivato dal Berkeley Packet Filter, pensato per il filtraggio efficiente dei pacchetti di rete, l'eBPF si è evoluto in una piattaforma generica per estendere dinamicamente le funzionalità del kernel mediante bytecode verificato e, successivamente, compilato JIT in codice nativo.

Dal punto di vista architetturale, l'eBPF introduce una pipeline che coinvolge un compilatore (tipicamente LLVM o GCC) capace di generare bytecode eBPF, un loader in user space (come libbpf o librerie di più alto livello) e un runtime in kernel space che include un interprete, un JIT compiler e un verifier. Il verifier svolge un ruolo cruciale in termini di sicurezza: analizza staticamente il bytecode per garantire l'assenza di loop non terminanti, accessi a memoria fuori limite o altre operazioni potenzialmente pericolose, rifiutando i programmi che non soddisfano i vincoli di sicurezza. Solo i programmi che superano la verifica vengono caricati ed eventualmente compilati JIT per ottenere prestazioni comparabili al codice nativo.

I programmi eBPF possono essere “agganciati” (hooked) a numerosi punti di attacco nel kernel: interfacce di rete (XDP, TC), system call, kprobe e uprobe, tracepoint e altri. Ogni programma viene eseguito in risposta a un evento (ad esempio la ricezione di un pacchetto, l'invocazione di una syscall, o un evento di performance), e può aggiornare strutture dati persistenti chiamate map (array, hash map, ring buffer, ecc.) utilizzate per comunicare con lo spazio utente. In ambito osservabilità, questo consente di implementare strumenti avanzati di tracing, profiling e monitoraggio senza richiedere riavvii del sistema o modifiche intrusive alla configurazione del kernel.

La libreria **Goccia** sfrutta eBPF tramite la libreria Go cilium/ebpf, che fornisce un wrapper Go-nativo per il caricamento e la gestione di programmi eBPF compilati. Lo eBPF Ingress stage si collega a una map di tipo ring buffer nel kernel, attraverso la quale i programmi eBPF inviano eventi verso lo spazio utente. Grazie all'uso dei generics in Go, lo stage eBPF di **Goccia** può deserializzare i record del ring buffer direttamente in struct tipizzate, permettendo all'utente di definire liberamente il layout dei dati prodotti dal programma eBPF. Questo modello abilita scenari in cui dati di basso livello (syscall, pacchetti di rete, eventi di performance) vengono campionati nel kernel e trasferiti in modo efficiente a una pipeline di elaborazione ad alte prestazioni.

2.8 QuestDB

QuestDB è un database time-series open source progettato specificamente per la gestione di flussi di dati indicizzati temporalmente, con particolare attenzione a throughput di ingestion elevati e query analitiche a bassa latenza [17]. A differenza dei database relazionali tradizionali, QuestDB adotta uno storage column-oriented e un modello dati nativamente temporale: il timestamp è un tipo primitivo e può fungere sia da chiave di partizionamento sia da dimensione centrale per le operazioni di query. Questo design si presta in modo naturale all'analisi di serie storiche, come dati di telemetria, metriche di sistema, segnali finanziari o misure IoT.

L'architettura interna di QuestDB si basa su una pipeline a tre livelli: un livello di ingest caldo tramite Write-Ahead Log (WAL), uno storage binario nativo columnar e, opzionalmente, uno strato di persistenza in formato Parquet su storage locale o remoto. I dati vengono inizialmente scritti in log transazionali, quindi riordinati e deduplicati in base al timestamp e infine materializzati nel formato colonnare partizionato nel tempo. Questo approccio consente di combinare durabilità, capacità di ingestion ad alta velocità e interoperabilità con ecosistemi esterni (ad esempio data lake basati su formato Parquet).

QuestDB implementa un'interfaccia SQL estesa con primitive specifiche per le serie temporali, come ASOF JOIN, SAMPLE BY, LATEST ON e funzioni di downsampling e aggregazione temporale. Queste estensioni semplificano la formulazione di query tipiche del dominio time-series (calcolo di medie mobili, aggregazioni su finestre temporali, ricostruzione dello stato più recente di un sistema) mantenendo una sintassi familiare a chi è abituato al paradigma relazionale.

Dal punto di vista dell'integrazione, QuestDB supporta più protocolli di ingestion: il wire protocol PostgreSQL (PGwire), l'InfluxDB line protocol e un'interfaccia TCP ottimizzata per l'ingest di metriche e telemetria. Questo consente di utilizzare QuestDB come sostituto drop-in per altri database time-series o come backend in architetture esistenti. In aggiunta, l'interfaccia HTTP/REST e la web console integrata forniscono strumenti di interrogazione e visualizzazione immediati, utili durante la fase di sviluppo e debugging.

2.9 OpenTelemetry

OpenTelemetry è un framework e toolkit di osservabilità open source, nato sotto la Cloud Native Computing Foundation (CNCF), che fornisce un insieme unificato di API, SDK e strumenti per l'instrumentation, la generazione, la raccolta e l'esportazione di dati di telemetria (tracce, metriche e log) da sistemi distribuiti eterogenei [18]. L'obiettivo principale di OpenTelemetry è standardizzare il modo in cui le applicazioni producono telemetria, riducendo il lock-in verso specifici vendor e permettendo di sostituire o affiancare backend diversi (ad esempio Prometheus, Jaeger, o soluzioni commerciali) senza modificare il codice applicativo. In questo senso, OpenTelemetry non è un sistema di storage o visualizzazione dei dati, bensì uno strato di instrumentation e trasporto vendor-agnostic, concepito per diventare parte integrante dell'infrastruttura software.

La specifica OpenTelemetry definisce un modello dati comune per i tre segnali fondamentali dell'osservabilità (traces, metrics, logs), le semantiche di propagazione del contesto distribuito e il protocollo di trasporto OTLP (OpenTelemetry Protocol). A partire da tali basi comuni, vengono fornite API e SDK per molteplici linguaggi, che consentono di instrumentare manualmente il codice (creazione esplicita di span, metriche, log strutturati) oppure di sfruttare l'instrumentation automatica di librerie e framework diffusi. Un aspetto centrale del modello è la correlazione tra segnali: le log entry e le misure metriche possono essere arricchite con TraceId e SpanId, consentendo di passare in modo diretto da un log di errore alla traccia corrispondente, o di correlare un picco di latenza osservato su una metrica con il path di esecuzione che lo ha generato.

Elemento chiave dell'ecosistema è l'OpenTelemetry Collector, un container eseguibile che implementa una pipeline di elaborazione generica per la telemetria: riceve dati da molteplici sorgenti, li trasforma e li esporta verso uno o più backend [19]. L'architettura del Collector è esplicitamente pipeline-based e suddivisa in componenti: i receiver ingeriscono i dati (ad esempio via OTLP/gRPC, OTLP/HTTP, Prometheus, Jaeger), i processor applicano trasformazioni (batching, filtering, arricchimento di attributi, tail sampling), mentre gli exporter inviano i dati verso sistemi di persistenza o analisi. Questa configurazione è descritta tramite file YAML che definiscono una o più pipeline per tipo di segnale (traces, metrics, logs), in modo concettualmente analogo al modello di pipeline implementato dalla libreria **Goccia**: ricezione, processing e consegna come fasi separate e componibili.

2.10 Il Metodo di Holt per il *Double Exponential Smoothing*

2.10.1 Fondamenti Teorici e Motivazione

Il metodo di Holt, introdotto da Holt nel 1957 [20], estende il *simple exponential smoothing* (SES) per permettere la previsione e l'analisi di serie temporali che presentano un andamento di tendenza lineare. Mentre il SES si limita a modellare una componente di livello costante e risulta inadatto a serie con *trend* manifesto, il metodo di Holt introduce una componente aggiuntiva dedicata alla stima della tendenza (*slope*), rendendo il modello maggiormente adatto all'elaborazione di dati con comportamento non stazionario.

Nel contesto specifico dell'applicazione proposta—ovvero la stima e la correzione di *timestamp* di messaggi ricevuti da *socket* UDP—il metodo di Holt fornisce un meccanismo efficiente per separare la componente di livello (il *timestamp* “smussato”, depurato dal *jitter*) dalla componente di *trend* (il *rate* di crescita sistematico dei *receive time*). Poiché i *timestamp* di ricezione seguono un *trend* lineare crescente, il metodo di Holt è particolarmente adatto a catturare questa dinamica e a fornire stime robuste nonostante le fluttuazioni dovute al *jitter* di rete [21].

2.10.2 Formulazione Matematica

Il metodo di Holt per il *double exponential smoothing* si basa su un'equazione di previsione e due equazioni di *smoothing*, una per il livello e una per la tendenza [20, 22, 23, 24]:

Equazione di Previsione:

$$\hat{y}_{t+h|t} = s_t + h \cdot b_t \quad (2.1)$$

Equazione di Livello (*Level Smoothing*):

$$s_t = \alpha y_t + (1 - \alpha)(s_{t-1} + b_{t-1}) \quad (2.2)$$

Equazione di Tendenza (*Trend Smoothing*):

$$b_t = \beta^*(s_t - s_{t-1}) + (1 - \beta^*)b_{t-1} \quad (2.3)$$

dove:

- y_t è l'osservazione (o misurazione) effettiva al tempo t
- s_t denota la stima del livello della serie al tempo t , rappresentante il “valore smussato” dell'osservazione corrente

- b_t denota la stima della tendenza (*slope*) della serie temporale al tempo t , ossia il *rate* di variazione tra periodi consecutivi
- α è il **parametro di *smoothing* per il livello**, con $0 \leq \alpha \leq 1$
- β^* è il **parametro di *smoothing* per la tendenza**, con $0 \leq \beta^* \leq 1$
- h è l'orizzonte di previsione (numero di passi temporali avanti)
- $\hat{y}_{t+h|t}$ è la previsione al tempo t per h periodi in avanti

2.10.3 Interpretazione Componentiale

L'equazione di livello mostra che s_t rappresenta una media ponderata tra l'osservazione corrente y_t e la “previsione a un passo” della serie al tempo $t - 1$, data dalla somma della stima precedente del livello e della stima precedente della tendenza: $s_{t-1} + b_{t-1}$ [22, 23]. Il parametro α controlla il grado di reattività del livello rispetto alle nuove osservazioni: un valore di α prossimo a 1 attribuisce peso maggiore ai dati recenti, mentre un valore prossimo a 0 privilegia le stime precedenti.

L'equazione di tendenza stima b_t come media ponderata tra due fonti di informazione: la differenza tra i due ultimi livelli stimati ($s_t - s_{t-1}$), che fornisce una stima della tendenza nel periodo corrente, e la stima precedente della tendenza (b_{t-1}) [22]. Il parametro β^* regola quanto rapidamente la tendenza stimata si adatta a variazioni nel *rate* di cambiamento: valori elevati di β^* consentono alla pendenza di mutare più rapidamente tra periodi consecutivi, mentre valori bassi generano stime di tendenza più stabili.

Il metodo è denominato “*double exponential smoothing*” poiché impiega due parametri di *smoothing* (α e β^*), in contrasto con il SES che ne utilizza uno solo (α) [22, 23].

2.10.4 Condizioni Iniziali

L'applicazione del metodo di Holt richiede l'inizializzazione di s_0 (il livello iniziale) e b_0 (la tendenza iniziale) [25, 22]. Quando $t = 1$, le equazioni di *smoothing* non possono essere applicate direttamente poiché mancano i valori precedenti. Le pratiche comuni per inizializzare questi valori includono:

1. **Inizializzazione semplice:** porre $s_0 = y_1$ (il primo valore osservato)
2. **Stima della tendenza iniziale:** calcolare b_0 come la differenza media tra coppie di osservazioni iniziali, ad esempio $b_0 = \frac{y_2 - y_1}{\Delta t}$, oppure utilizzare una regressione lineare su un sottoinsieme iniziale dei dati

Alternativamente, s_0 e b_0 possono essere stimati simultaneamente insieme ai parametri α e β^* minimizzando la somma dei quadrati degli errori (SSE) sull'intera serie [22, 23].

2.10.5 Correzione dei *Timestamp* di Ricezione

Nel contesto del presente lavoro, il metodo di Holt è impiegato per stimare e correggere il *timestamp* di messaggi ricevuti, in particolare quando i messaggi arrivano fuori ordine o affetti da *jitter*. In questo scenario applicativo:

- y_t (**osservazione**): il *receive time* effettivo di un pacchetto al tempo di ricezione t , affetto da *jitter* causato da variazioni nella latenza di trasmissione
- s_t (**livello smussato**): la stima del *timestamp* “corretto”, depurato dalle fluttuazioni di *jitter* ad alta frequenza
- b_t (**tendenza**): la stima del *rate* di crescita sistematico dei *receive time*, catturando il *trend* lineare intrinseco della sequenza di messaggi

La componente di livello s_t funge da stimatore robusto del *timestamp* di ricezione corretto, mentre la componente di tendenza b_t cattura il *rate* medio di arrivo dei pacchetti sulla connessione, permettendo di discriminare efficacemente tra variazioni casuali dovute al *jitter* e la dinamica sistematica della ricezione.

Questa decomposizione è particolarmente vantaggiosa rispetto a metodi alternativi (come il *Kalman filter* esteso) in quanto:

- **Semplicità computazionale**: il metodo di Holt richiede solo operazioni aritmetiche elementari, riducendo la complessità e l'overhead, quindi diminuendo la latenza [21]
- **Robustezza rispetto al *jitter***: la separazione esplicita della componente di tendenza consente di discriminare efficacemente tra *jitter* (variazioni ad alta frequenza) e *trend* lineare (variazioni lente e sistematiche) [21]
- **Efficienza numerica**: non richiede inversioni di matrici, a differenza dei filtri di Kalman [21]

Ricerche comparative hanno dimostrato che il metodo di Holt per la compensazione del *jitter* opera circa 100 volte più velocemente rispetto al *Kalman filter* esteso (EKF), pur mantenendo o addirittura migliorando le prestazioni di riduzione del *jitter* (circa 18-20% di miglioramento relativo) [21].

2.10.6 Applicazioni Pratiche

Nel contesto pratico di sistemi di comunicazione in tempo reale, il metodo di Holt trova applicazione diretta nel filtraggio e nella correzione dei *timestamp* di ricezione (*receive time*).

Filtraggio del *Jitter* di Ricezione: Il *jitter* nei *timestamp* di ricezione è causato da variazioni nella latenza di trasmissione dovute a congestione di rete, *routing* variabile, e variabilità della velocità di elaborazione nei nodi intermedi. L'equazione di livello del metodo di Holt filtra efficacemente queste fluttuazioni ad alta frequenza, estraendo il valore “vero” del *timestamp* depurato dal *jitter*.

Stima del *Trend* Lineare di Arrivo: In una sequenza di messaggi ricevuti in modo continuo, i *receive time* seguono naturalmente un *trend* lineare crescente: ogni messaggio arriva approssimativamente a intervalli regolari. Il metodo di Holt cattura questa dinamica nella componente di tendenza b_t , che rappresenta il *rate* medio di arrivo dei pacchetti. Questa stima permette di stabilire se un pacchetto in arrivo tardivo è coerente con il *trend* osservato o rappresenta una anomalia.

Ricezione *Out-of-Order*: Utile per i protocolli, quali UDP, che non garantiscono l'arrivo ordinato dei pacchetti. Quando, per esempio, un'applicazione riceve pacchetti fuori sequenza, il metodo di Holt consente di stimare il *timestamp* corretto di ricezione per ciascun pacchetto basandosi sia sulla cronologia di arrivo precedente che sul *trend* lineare osservato, facilitando il riordinamento logico dei messaggi.

La semplicità computazionale del metodo di Holt lo rende particolarmente adatto a questi scenari, dove ogni pacchetto ricevuto deve essere elaborato con latenza minima per non introdurre ritardi significativi nelle applicazioni tempo-reale [21].

Capitolo 3

Architettura e implementazione della libreria Goccia

3.1 Pipeline

Una *pipeline*, in informatica, è una sequenza di stadi (*stage*) attraverso cui i dati fluiscono in modo ordinato. Ciascuno *stage* riceve un input, lo elabora e produce un output che diventa l'input dello *stage* successivo. Ogni *stage* ha una responsabilità ben delimitata, favorendo così modularità, riuso e facilità di ragionamento sull'intero sistema.

In contesti concorrenti, ogni *stage* può operare in parallelo sugli elementi che riceve. Ciò significa che, mentre lo *stage* 1 elabora un nuovo messaggio, lo *stage* 2 può già processare quello precedente, incrementando così il *throughput* complessivo della pipeline.

All'interno della libreria **Goccia**, il concetto di *pipeline* è incarnato dalla struct **Pipeline**, che funge da orchestratore degli *stage* e rappresenta il punto di ingresso per l'utilizzo della libreria stessa. Questa struct definisce i tre metodi principali che ogni *stage* deve implementare: **Init**, **Run** e **Close**, corrispondenti alle diverse fasi del ciclo di vita della pipeline (si veda la sezione successiva per ulteriori dettagli). Inoltre, il metodo **AddStage** consente di aggiungere nuovi *stage* fintanto che la pipeline non è in esecuzione.

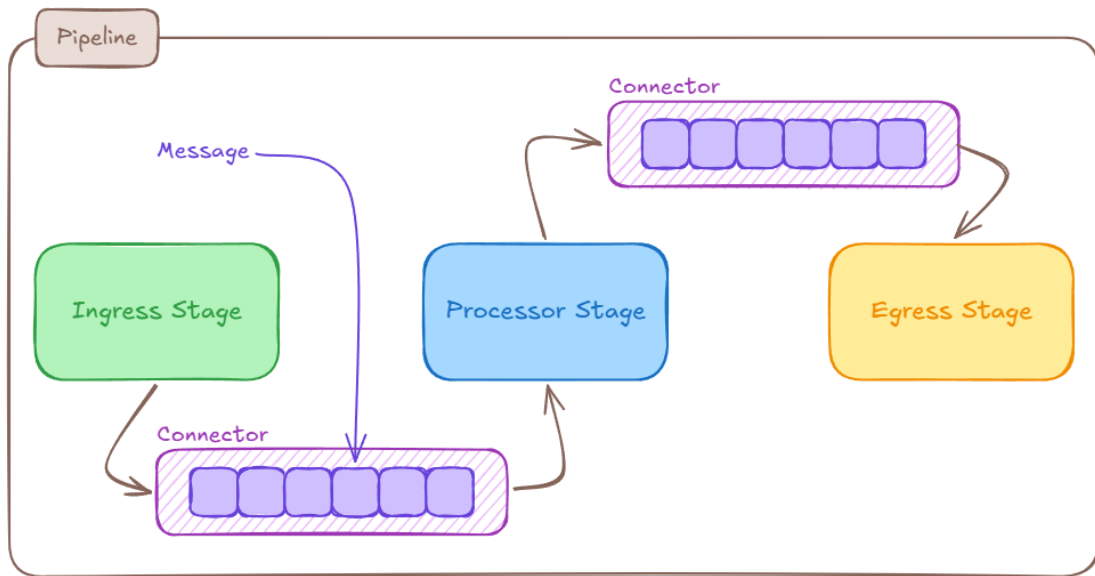


Figura 3.1: Struttura generale di una pipeline in Goccia

3.2 Stage

Uno *stage* di una pipeline rappresenta un'unità modulare di elaborazione che riceve un input da *stage* precedenti, oppure da una sorgente esterna, lo elabora e produce un output per gli *stage* successivi. In tal modo, gli *stage* formano una catena di trasformazione dei dati scalabile e facilmente componibile. Dal punto di vista concettuale, uno *stage* costituisce un limite logico e sequenziale all'interno della *pipeline*, con un flusso dati che attraversa le fasi di input, elaborazione e output, in maniera simile a quanto avviene nei processi CI/CD (build, test, deploy).

Nella libreria **Goccia**, gli *stage* devono aderire all'interfaccia **Stage**, la quale definisce i metodi necessari per rendere una struct di Go compatibile con la pipeline principale. Tali metodi coincidono con le tre fasi del ciclo di vita: **Init**, per l'inizializzazione delle risorse; **Run**, per l'elaborazione dei dati in input; e **Close**, per il rilascio delle risorse e la chiusura dello *stage*.

La libreria categorizza gli *stage* in tre gruppi principali, **Ingress**, **Processor** ed **Egress**, tutti aderenti all'interfaccia generica **Stage**. Un aspetto rilevante riguarda la modalità di esecuzione: il metodo **Run** viene invocato all'interno di una goroutine dedicata per ciascuno *stage*. Di conseguenza, una pipeline composta da cinque *stage* disporrà di almeno cinque goroutine attive, oltre a quella principale di esecuzione nel main.

3.3 Connector

Affinché una pipeline possa effettivamente operare, una volta creati gli stage, è necessario metterli in comunicazione tramite un componente capace di trasferire i dati da uno stage al successivo. Tale componente deve essere *thread safe*, poiché ogni stage viene eseguito su una goroutine distinta, e deve disporre di un meccanismo di *buffer* per mitigare gli effetti della *backpressure*, fenomeno che si manifesta quando uno stage a valle elabora i dati più lentamente del precedente, causando un rallentamento a catena.

Il linguaggio Go fornisce, a tal fine, i *channel* (anche bufferizzati) come tipo primitivo per la comunicazione tra goroutine. Tuttavia, la libreria **Goccia** adotta un approccio differente, basato su *ring buffer lock-free*, con l'obiettivo di massimizzare il throughput. L'implementazione dei *ring buffer* si basa su tre varianti distinte, ciascuna ottimizzata per specifici pattern di accesso concorrente: Single Producer Single Consumer (SPSC), Single Producer Multiple Consumer (SPMC) e Multiple Producer Single Consumer (MPSC). La scelta della variante dipende dal contesto di utilizzo: la versione SPSC viene impiegata per la comunicazione tra stage consecutivi nella pipeline, poiché la relazione è di tipo 1:1. La versione SPMC è utilizzata nel contesto del *worker pool* per operazioni di *fan-out*, dove un singolo produttore distribuisce i *task* a molteplici *consumer*. Simmetricamente, la versione MPSC è impiegata per operazioni di *fan-in*, dove molteplici produttori convergono i risultati verso un singolo *consumer*.

Per mantenere la libreria estensibile a futuri sviluppi, viene definita l'interfaccia **Connector**, dotata di tre metodi principali:

- **Write**: consente di scrivere un dato nel connettore, ritornando **ErrClosed** qualora il connettore sia stato chiuso.
- **Read**: legge un dato disponibile dal connettore, accettando un **context.Context** come parametro per la gestione della cancellazione. Questo design consente di implementare strategie intelligenti di rilascio delle risorse: quando il contesto viene cancellato (ad esempio, a seguito della ricezione di un segnale **SIGINT** o **SIGTERM**), il metodo ritorna immediatamente con **ctx.Err()**, permettendo così una terminazione ordinata della pipeline.
- **Close**: chiude il connettore, impedendo ulteriori operazioni di scrittura e lettura.

L'implementazione del *ring buffer* nella libreria **Goccia** sfrutta operazioni atomiche per massimizzare le prestazioni in contesti multicore. La struct **RingBuffer** incapsula tre implementazioni interne distinte (**spsc**, **spmc**, **mpsc**), selezionate in fase di creazione tramite apposito parametro.

Tutte le varianti di *ring buffer* condividono una struttura comune, `commonBuffer`, che incapsula i campi fondamentali per la gestione del *buffer* circolare. Il campo `head` rappresenta l'indice della prossima posizione di scrittura nel *buffer*, mentre `tail` indica l'indice della prossima posizione di lettura. Entrambi i campi sono di tipo `atomic.Uint64` [26], garantendo che le operazioni di incremento e lettura siano atomiche e visibili a tutti i *core* della CPU senza necessità di *lock* espliciti. Il campo `capacity` memorizza la dimensione totale del *buffer*, sempre arrotondata alla potenza di 2 superiore più vicina per consentire l'uso della maschera bit-a-bit `capMask` (pari a `capacity - 1`) nel calcolo dell'indice effettivo tramite l'operazione `index & capMask`, evitando così la costosa operazione di modulo.

```
1 type commonBuffer struct {
2     head atomic.Uint64
3
4     _ cpu.CacheLinePad
5
6     tail atomic.Uint64
7
8     _ cpu.CacheLinePad
9
10    capacity uint64
11    capMask  uint64
12
13    _ cpu.CacheLinePad
14 }
```

Listing 3.1: Definizione campi comuni per l'implementazione di ring buffer (internal/rb/common.go, `commonBuffer` struct)

Un aspetto cruciale dell'implementazione è l'uso sistematico del *padding* delle *cache line* tramite il tipo `cpu.CacheLinePad` [27]. Nelle architetture moderne x86-64, le *cache line* hanno una dimensione tipica di 64 byte. Quando la CPU accede a un dato in memoria, l'intero blocco di 64 byte contenente quel dato viene caricato nella *cache*. In contesti concorrenti, questo può generare un fenomeno critico per le prestazioni noto come *false sharing*.

Il *false sharing* si verifica quando due *thread* accedono a variabili distinte che però risiedono nella stessa *cache line*. Anche se le variabili sono logicamente indipendenti, ogni volta che un *thread* modifica la propria variabile, l'intera *cache line* viene invalidata nelle *cache* degli altri *core*, forzando un *refresh* costoso. Nei *ring buffer* concorrenti, dove *producer* e *consumer* operano rispettivamente su `head` e `tail`, questo fenomeno comporterebbe un *overhead* significativo: ogni scrittura su `head` da parte del *producer* invaliderebbe la *cache line* contenente `tail` nei *core* dei *consumer*, e viceversa. Separando `head` e `tail` tramite *padding*, si elimina la

contesa tra *producer* e *consumer* sulle *cache line*, migliorando drasticamente le prestazioni in scenari ad alto throughput [28].

Ciascuna variante del *buffer* implementa internamente le operazioni *push* e *pop* con semantiche differenti. La versione SPSC utilizza un *buffer* circolare con accesso diretto agli indici, senza necessità di sincronizzazione tra *producer* e *consumer* oltre alle barriere di memoria fornite dalle operazioni atomiche. La versione SPMC/MPSC introduce *slot* con *flag* atomici `dataReady` per coordinare l'accesso concorrente di molteplici *consumer*, garantendo che ogni dato sia consumato una sola volta tramite l'operazione di *compare-and-swap* sulla *tail/head*.

```
1 type slot[T any] struct {
2     dataReady atomic.Bool
3     data      T
4 }
```

Listing 3.2: Definizione *slot* per ring buffer SPMC/MPSC (internal/rb/common.go, *slot struct*)

Il metodo `Write` del *ring buffer* implementa una strategia di *backpressure* progressiva. Inizialmente, tenta di inserire il dato tramite un numero limitato di tentativi, cedendo la goroutine tramite `runtime.Gosched` [29] tra un tentativo e l'altro. Qualora il *buffer* rimanga pieno dopo la fase di *spinning*, il *thread* si blocca su una *condition variable* (`notFull`) fino a quando non viene segnalato spazio disponibile o il *buffer* viene chiuso. Questo approccio ibrido combina l'efficienza del *busy-waiting* per brevi periodi di contesa con il blocco efficiente per periodi prolungati, riducendo il consumo di CPU in scenari di alta pressione.

Il metodo `Read` implementa una logica analoga, applicando inizialmente una fase di *spin* per tentare di estrarre un dato, seguita da un blocco su una *condition variable* (`notEmpty`) qualora il *buffer* rimanga vuoto. La differenza sostanziale rispetto alla versione precedente risiede nella gestione della cancellazione tramite `context.Context`. Il metodo `wait` interno coordina l'attesa sulla *condition variable* con il contesto fornito dall'utente.

```
1 func (rb *RingBuffer[T]) wait(ctx context.Context, cond
   *sync.Cond) error {
2     done := make(chan struct{})
3
4     go func() {
5         defer close(done)
6         cond.Wait()
7     }()
8
9     select {
10    case <-done:
11        return nil
```

```
12
13     case <-ctx.Done():
14         // Wake up the waiting goroutine
15         cond.Broadcast()
16         <-done
17         return ctx.Err()
18     }
19 }
```

Listing 3.3: Implementazione metodo wait usato dai ring buffer (internal/rb/ring_buffer.go, metodo wait)

Quando il contesto viene cancellato, la goroutine in attesa viene risvegliata tramite `Broadcast()` e il metodo ritorna l'errore di cancellazione. Questo meccanismo garantisce che le operazioni di lettura e scrittura possano essere interrotte in modo controllato, prevenendo *deadlock* o attese indefinite durante lo *shutdown* della pipeline.

3.4 Ingress Stage

Uno *Ingress stage* rappresenta il punto di ingresso dei dati all'interno della *pipeline* di elaborazione. Costituisce la prima componente che riceve gli input provenienti dall'esterno del sistema e li introduce nel flusso di elaborazione, alimentando così l'intera catena di *stage* successivi. In sostanza, uno *Ingress stage* agisce come una sorgente di dati, traducendo eventi, messaggi o pacchetti provenienti da differenti protocolli o interfacce in un formato gestibile dalla *pipeline* di **Goccia**.

Nel contesto della libreria, gli *Ingress stage* sono progettati per essere altamente performanti, concorrenti e facilmente integrabili con i meccanismi di comunicazione già definiti dal framework. Ogni *Ingress stage* è quindi responsabile di stabilire una connessione con la sorgente, acquisire i dati in ingresso, trasformarli nel formato previsto e inoltrarli verso lo *stage* successivo.

La libreria **Goccia** fornisce diversi tipi di *Ingress stage*, ciascuno progettato per scenari operativi o fonti di dati differenti.

3.4.1 Ticker

Lo *Ticker stage* rappresenta l'*Ingress stage* più semplice messo a disposizione dalla libreria **Goccia** ed è concepito per generare in modo autonomo un flusso regolare di messaggi, senza dipendere da sorgenti esterne quali socket di rete, file o code di messaggistica. Dal punto di vista concettuale, si tratta di una sorgente periodica di eventi, particolarmente utile per casi d'uso quali il testing della *pipeline*, il benchmarking di prestazioni, l'iniezione di traffico sintetico o la generazione di

"heartbeat" [30], ovvero segnali utili a stabilire lo stato di "salute" di un servizio (per esempio, chiamate HTTP verso un web-server per stabilire l'operabilità del servizio), ad intervalli costanti. Ogni "tick" corrisponde alla produzione di un messaggio che attraversa l'intera *pipeline*, permettendo di osservare il comportamento dei vari *stage* a valle in presenza di un carico controllato.

La logica di generazione dei messaggi sfrutta il `time.Ticker` della libreria standard del Go, il quale mette a disposizione un channel notificato ogni volta che l'intervallo T viene raggiunto, dove T è definito dalla configurazione dello *stage*.

```
1 for {
2     // ...
3     select {
4         case <-ctx.Done():
5             return
6         case <-ts.ticker.C:
7             // generate the ticker message
8     }
9 }
```

Listing 3.4: Ciclo principale del Ticker stage (ingress/ticker.go, metodo run)

Nel frammento di codice riportato, è possibile osservare come il costrutto `select` permetta di ascoltare su molteplici channel in modalità multiplex. In questo caso specifico, il *runtime* rimane in attesa di essere notificato dal ticker oppure della ricezione di un segnale di cancellazione del contesto (`ctx.Done()`). Quest'ultimo meccanismo è fondamentale per implementare strategie intelligenti di rilascio delle risorse, ad esempio quando il processo riceve un segnale `SIGINT` o `SIGTERM`, consentendo così l'implementazione di uno *graceful shutdown* coerente nelle applicazioni. Per questa ragione, nelle librerie Go moderne è prassi richiedere un `context.Context` come primo argomento di funzioni che impiegano socket o richiedono tempistiche significative per la terminazione.

Il tipo di messaggio prodotto dallo *Ticker stage* è descritto dalla struct `TickerMessage` e contiene un unico campo specifico, `TickNumber`, utilizzato per numerare progressivamente i tick. Al momento della generazione del messaggio vengono popolati i campi generici di metadato (tempo di ricezione, timestamp) e viene inizializzata la *root span* per la tracciatura degli attraversamenti degli *stage* successivi, conformemente allo standard OpenTelemetry [31].

3.4.2 UDP

Lo *UDP stage* rappresenta un *Ingress stage* progettato per ricevere datagrammi *UDP* provenienti dalla rete. A differenza dello *Ticker stage*, che genera autonomamente messaggi a intervalli regolari, lo *UDP stage* rimane in ascolto su un socket di rete e resta in attesa di nuovi datagrammi in arrivo. Questa tipologia di *Ingress stage*

risulta particolarmente utile in contesti dove è necessario elaborare flussi di dati provenienti da sorgenti esterne, quali sensori IoT, applicazioni remote, strumenti di monitoraggio di rete o qualsiasi sistema che comunichi tramite il protocollo *UDP*. La natura *connectionless* del protocollo *UDP* consente di ricevere messaggi da molteplici mittenti senza la necessità di stabilire connessioni esplicite, rendendolo ideale per scenari ad alto *throughput* dove la perdita occasionale di datagrammi è accettabile in cambio della bassa latenza.

La libreria **Goccia** mette a disposizione tre parametri di configurazione per lo *UDP stage*: indirizzo IP, porta e dimensione del buffer. I primi due servono a impostare la sorgente da cui ricevere i pacchetti e dispongono di valori di default, ovvero "0.0.0.0" per l'indirizzo (che indica l'ascolto su tutte le interfacce di rete) e 20.000 per la porta. La dimensione del buffer utilizzato per leggere il payload dei datagrammi *UDP* è impostata di default a 1474 byte, poiché la dimensione massima per un payload Ethernet standard è di 1500 byte, da cui si sottraggono 28 byte relativi all'header *UDP*.

Internamente, lo *stage* utilizza un puntatore a una connessione *UDP* fornita dalla libreria standard del Go [32]. Tale connessione viene inizializzata nel metodo `Init` del ciclo di vita dello *stage*. Una volta che l'inizializzazione ha esito positivo, è possibile procedere con l'esecuzione dello *stage* tramite il metodo `Run`. Questo metodo implementa un ciclo `for` in cui viene richiamato il metodo `Read` (operazione bloccante) della connessione *UDP*, il quale scrive i byte ricevuti in un buffer pre-allocato alla dimensione configurata. Parallelamente al ciclo principale, viene avviata una goroutine ausiliaria il cui compito è chiudere la connessione *UDP* nel momento in cui viene ricevuta una notifica di cancellazione tramite `context.Context`. Una volta che la connessione è chiusa, il metodo `Read` ritorna un errore di tipo `net.ErrClosed` [33], permettendo così al ciclo di uscire in modo ordinato.

```
1 go func() {
2     <-ctx.Done()
3     us.conn.Close()
4 }()
5
6 buf := make([]byte, us.bufferSize)
7
8 for {
9     // Read the UDP payload
10    n, err := us.conn.Read(buf)
11    if err != nil {
12        // Check if the connection is closed
13        if errors.Is(err, net.ErrClosed) {
14            // Check if caused by context cancellation
15            select {
```



```
16     case <-ctx.Done():
17         return
18     default:
19     }
20 }
21
22 // ...
23 }
24
25 // Handle the buffer and send the message ...
26 }
```

Listing 3.5: Ciclo principale dello UDP stage (ingress/udp.go, metodo run)

Il tipo di messaggio prodotto dallo *UDP stage* è descritto dalla struct `UDPMessage`, contenente due campi specifici: `Payload` e `PayloadSize`. Il messaggio implementa l'interfaccia `message.Serializable` definita nel package `message`, consentendo il collegamento di differenti tipi di *Processor stage* a valle che accettano messaggi recanti buffer di byte. Al fine di soddisfare l'interfaccia, è sufficiente implementare il metodo `GetBytes`, il quale restituisce una slice di byte corrispondente al campo `Payload` del messaggio.

```
1 func (um *UDPMessage) GetBytes() []byte {
2     return um.Payload
3 }
```

Listing 3.6: Metodo `GetBytes` di `UDPMessage` (ingress/udp.go)

Poiché il payload è caratterizzato da dimensioni significative, al fine di ridurre il carico sul *garbage collector* attraverso il riuso della memoria e l'eliminazione di continue allocazioni e deallocazioni di migliaia di oggetti, si è optato per l'utilizzo di un *object pool* (`sync.Pool`) [34]. Gli oggetti `UDPMessage` vengono quindi pre-allocati nel pool e riutilizzati per ogni nuovo datagramma ricevuto, riducendo così la pressione sul sistema di gestione della memoria durante l'elaborazione ad alto throughput.

Lo *stage* espone due metriche destinate al monitoraggio del numero di messaggi e byte ricevuti, implementate tramite l'uso di contatori asincroni [35]. Tali metriche possono essere impiegate per monitorare il *throughput* in ingresso alla *pipeline*, fornendo visibilità sulla velocità di ricezione dei dati dalla sorgente di rete.

3.4.3 TCP

Lo *TCP stage* rappresenta un *Ingress stage* progettato per ricevere flussi dati da connessioni *TCP*. A differenza dello *UDP stage*, che gestisce singoli datagrammi

provenienti da molteplici mittenti senza stato, il *TCP stage* stabilisce connessioni persistenti con i client e mantiene lo stato della comunicazione per ciascuna connessione. Questa caratteristica lo rende particolarmente adatto per scenari in cui l'affidabilità del trasporto è critica, quali l'acquisizione di log da sistemi remoti, la ricezione di comandi da applicazioni client o l'integrazione con protocolli applicativi che richiedono una comunicazione orientata allo *stream*. Diversamente dallo *Ticker stage* e dallo *UDP stage*, il *TCP stage* introduce una complessità significativamente maggiore, poiché deve gestire molteplici connessioni concorrenti, ciascuna potenzialmente caratterizzata da uno stato diverso, e deve risolvere il problema fondamentale di come delimitare i messaggi all'interno di un flusso di byte continuo.

La libreria **Goccia** mette a disposizione una configurazione flessibile per il *TCP stage*, incapsulata nella struct `TCPConfig`. I parametri fondamentali sono analoghi a quelli dello *UDP stage*: un indirizzo IP e una porta definiscono il punto di ascolto su cui il server *TCP* rimane in attesa di connessioni in ingresso, mentre un buffer di lettura viene utilizzato per acquisire i dati dalla connessione. Tuttavia, il *TCP stage* introduce ulteriori parametri specifici per la gestione avanzata del flusso.

Il parametro `ReadTimeout` specifica il tempo massimo di inattività consentito su una connessione prima che essa venga forzatamente chiusa. Questo meccanismo protegge il server da client che si collegano e rimangono silenti indefinitamente, occupando risorse preziose del sistema. Il parametro `MaxMessageSize` (default 4 MB) definisce la dimensione massima consentita per un messaggio; qualora il buffer accumulato superi questo valore, la connessione viene chiusa al fine di prevenire attacchi di *denial of service* basati su messaggi abnormemente grandi.

Un aspetto cruciale della configurazione è il *framing mode*, definito dal campo `FramingMode`, il quale specifica come i messaggi vengono delimitati all'interno del flusso *TCP*. Sono supportate due modalità distinte:

- ***Delimited*** (default): I messaggi sono separati da un delimitatore, tipicamente una sequenza di byte come `"\r\n"`, impostabile dall'utente nel campo `Delimiter`. In questa modalità, il *parser* ricerca la sequenza delimitatrice all'interno del buffer accumulato e la utilizza come marcatore di fine messaggio. Esempi di protocolli di livello superiore che impiegano questa strategia sono HTTP/1.x, SMTP e FTP.
- ***Length-Prefixed***: I messaggi sono preceduti da un header contenente la lunghezza del messaggio. Questa modalità introduce parametri aggiuntivi per estrarre la lunghezza del payload a partire da un header:
 - `HeaderLen` (default 16 byte): la dimensione totale dell'header.
 - `MessageLengthFieldOffset` (default 0): l'offset all'interno dell'header dove inizia il campo di lunghezza.

- `MessageLengthFieldLen`: la dimensione del campo di lunghezza (1, 2, 4 o 8 byte).
- `MessageLengthFieldEndianness`: l'ordine dei byte (*little-endian* o *big-endian*) del campo di lunghezza.

Infine, il parametro `OutputQueueSize` specifica la dimensione della coda interna (*fan-in*) che media tra le goroutine dedicate alle connessioni e il *connector* verso lo *stage* successivo.

Il *TCP stage* implementa un'architettura basata su molteplici goroutine sincronizzate. Il ciclo principale si occupa di accettare nuove connessioni tramite il metodo `Accept` del *listener TCP* [36] e, per ciascuna nuova connessione in ingresso, avvia una nuova goroutine dedicata alla gestione della relativa comunicazione. In questo modo, si ottiene un pattern di *fan-out*, in cui una singola goroutine principale genera *N* goroutine worker, una per ogni connessione cliente.

<disegno singola goroutine che genera N goroutine per N connessioni>

Questa architettura è resa possibile dal *runtime* di Go, il quale è in grado di gestire un numero elevato di goroutine e di schedularle efficientemente su un numero finito di thread del sistema operativo.

```
1 for {
2     // ...
3
4     conn, err := ts.listener.Accept()
5     // Check and handle the error ...
6
7     // Spawn a goroutine to handle the connection
8     go ts.handleConn(ctx, conn)
9 }
```

Listing 3.7: Ciclo principale del TCP stage (`ingress/tcp.go`, metodo `run`)

Per quanto riguarda la gestione della singola connessione, il flusso è concettualmente simile a quello dello *UDP stage*, con l'eccezione rilevante della divisione del *stream* in messaggi. Tale divisione richiede l'utilizzo di un buffer ausiliario di accumulazione, oltre al buffer di lettura. Il buffer di accumulazione è fondamentale nei casi in cui un messaggio risulti frammentato su più letture: ad esempio, se il buffer di lettura è di 4 KB e il messaggio trasmesso sulla connessione è di 8 KB, saranno necessarie due letture i cui contenuti dovranno essere accumulati sequenzialmente.

```
1 // Preallocate the accumulator
2 accBaseCap := min(4*ts.bufferSize, ts.maxMsgSize)
3 acc := make([]byte, 0, accBaseCap)
4
5 // ...
6
```

```
7  for {
8      // ...
9
10     // Set the read deadline
11     conn.SetReadDeadline(time.Now().Add(ts.readTimeout))
12
13     // Read the TCP stream
14     n, err := conn.Read(buf)
15     // Check and handle the error ...
16
17     // Append the new bytes to the accumulator
18     acc = append(acc, buf[:n]...)
19
20     // Prevent accumulator from growing too large
21     if len(acc) > ts.maxMsgSize {
22         ts.tel.LogWarn("message too large, closing connection")
23         return
24     }
25
26     for {
27         // Divide the accumulator into messages ...
28     }
29
30     // ...
31 }
```

Listing 3.8: Buffer per la gestione connessione del TCP stage (ingress/tcp.go, metodo handleConn)

Dopo l'accumulazione dei byte trasmessi, entra in gioco la logica di divisione dei messaggi secondo la modalità configurata nello *stage* (*delimited* oppure *length-prefixed*). In modalità delimitata, la ricerca della sequenza delimitatrice è effettuata tramite `bytes.Index`; in modalità *length-prefixed*, il *parser* estrae la lunghezza dal campo di header designato, utilizzando funzioni che gestiscono le diverse combinazioni di lunghezza e *endianess* supportate.

```
1  // ...
2
3  for {
4      accLen := len(acc)
5
6      // If the accumulator is smaller than the minimum length,
7      // continue reading the TCP stream
8      if accLen < minAccLen {
9          continue loop
10     }
```

```
11
12 // Get the length of the message.
13 msgLen := 0
14 totLen := 0
15 switch ts.framingMode {
16 case TCPFramingModeDelimited:
17     // Search for the delimiter
18     msgLen = bytes.Index(acc, ts.delimiter)
19     totLen = msgLen + ts.delimiterLen
20
21 case TCPFramingModeLengthPrefixed:
22     msgLen = ts.parseHeader(acc[:ts.headerLen])
23     totLen = msgLen + ts.headerLen
24 }
25
26 if msgLen == -1 || accLen < totLen {
27     // If the message length is not found or the
28     // accumulator is too small,
29     // break the loop and continue reading the TCP stream
30     break
31 }
32
33 // Extract the message
34 msg := acc[:totLen]
35
36 // Handle the message and send the result to the output
37 // connector ...
38
39 // Remove the message from the accumulator
40 acc = acc[totLen:]
41
42 // Check if the accumulator should be reset ...
43 }
44
45 // ...
```

Listing 3.9: Divisione messaggi del TCP stage (ingress/tcp.go, metodo handleConn)

Il tipo di messaggio prodotto dal *TCP stage* è descritto dalla struct `TCPMessage`, la quale implementa l'interfaccia di serializzazione, in modo analogo al messaggio dello *UDP stage*. A differenza di `UDPMessage`, `TCPMessage` non utilizza un *object pool*, poiché i messaggi *TCP* possono avere dimensioni molto variabili e l'impiego di un pool con buffer pre-allocato potrebbe risultare inefficiente o insufficiente. Il messaggio porta con sé il payload in byte (`Message`), la dimensione del payload

(`MessageSize`) e l'indirizzo remoto della connessione (`RemoteAddr`), fornendo così sia il contenuto informativo sia il contesto di provenienza.

Dal punto di vista dell'osservabilità, il *TCP stage* è per molti aspetti simile allo *UDP stage*. Anche in questo caso sono presenti metriche per il numero totale di messaggi e di byte ricevuti, che consentono di monitorare il *throughput* complessivo della *pipeline*. In aggiunta, viene introdotta una metrica specifica, `open_connections`, che traccia il numero di connessioni *TCP* attualmente aperte. Tale metrica offre visibilità sul grado di utilizzo delle risorse del server e permette di individuare con facilità situazioni anomale, come un numero insolitamente elevato di connessioni persistenti o client che non rilasciano correttamente le proprie sessioni.

3.4.4 Kafka

Il *Kafka stage* rappresenta un *Ingress stage* progettato per consumare messaggi da topic *Kafka*, integrandosi direttamente con architetture *event-driven* e sistemi di *streaming* dati. A differenza dei precedenti *Ingress stage* (*Ticker*, *UDP*, *TCP*), il *Kafka stage* non implementa un protocollo di trasporto diretto, ma si affida a un broker centralizzato (*Kafka*) per la gestione persistente e distribuita dei messaggi. Questa caratteristica lo rende ideale in scenari in cui i dati provengono da molteplici produttori asincroni, in cui è necessario garantire una semantica di consegna robusta (*at-least-once*, *exactly-once*), oppure in cui si desiderano costruire pipeline di elaborazione fortemente *decoupled*. La libreria **Goccia** sfrutta la libreria open-source *segmentio/kafka-go*, un'implementazione Go-nativa del protocollo *Kafka* che evita dipendenze da librerie esterne come *librdkafka* [37, 38].

La configurazione del *Kafka stage* è estremamente flessibile e riflette la complessità intrinseca del sistema *Kafka*. La struct `KafkaConfig` mima la struttura di configurazione `ReaderConfig` fornita dalla dipendenza *kafka-go*, così da consentire all'utente una personalizzazione completa della connessione verso i broker *Kafka* [39]. In questo modo è possibile controllare, tra gli altri, parametri relativi ai broker, al *consumer group*, alle politiche di bilanciamento delle partizioni, ai timeout di lettura e alle strategie di backoff. Di default, lo *stage* consuma i messaggi utilizzando un *group id*, in modo da sfruttare la gestione automatica degli *offset* offerta dal broker quando i dati vengono letti dai topic.

La logica di lettura dello *stage* è volutamente semplice: il cuore dell'implementazione è un unico ciclo `for` che invoca ripetutamente il metodo `ReadMessage` del reader *kafka-go* [40]. Ogni chiamata restituisce un nuovo messaggio *Kafka*, che viene poi trasformato in una struct `KafkaMessage` e inserito nel *connector* di output dello *stage*. In caso di errore, se questo è dovuto alla cancellazione del `context`, il ciclo termina in modo ordinato; negli altri casi l'errore viene registrato tramite il sottosistema di telemetria, e il ciclo procede al tentativo successivo, garantendo una maggiore robustezza in presenza di problemi transitori di rete o di broker.

Dal punto di vista dell'osservabilità, oltre alle metriche comuni a tutti gli *Ingress stage* (numero di messaggi e byte ricevuti), il *Kafka stage* introduce la possibilità di estrarre il contesto di *tracing* direttamente dagli *header* del messaggio *Kafka*. A tale scopo viene utilizzata una struttura dedicata che implementa l'interfaccia *TextMapPropagator* definita dallo standard *OpenTelemetry* [41]. Questo meccanismo consente di propagare i *trace* tra servizi differenti, permettendo la costruzione di un sistema di *tracing* distribuito in cui ogni messaggio *Kafka* può trasportare il contesto di esecuzione end-to-end lungo l'intera pipeline.

<disegno screenshot trace distribuito>

3.4.5 eBPF

Lo *eBPF stage* rappresenta un *Ingress stage* unico nel panorama della libreria **Goccia**, in quanto non legge dati da una sorgente di rete tradizionale o da un broker esterno, ma comunica direttamente con il kernel Linux tramite programmi *eBPF* (*extended Berkeley Packet Filter*) che sfruttano una map di tipo *ringbuf*. L'*eBPF* consente l'esecuzione sicura di codice *sandbox* direttamente nel kernel, con *overhead* minimo, abilitando scenari di osservabilità, monitoraggio e *security* altamente efficienti. Lo *eBPF stage* è quindi ideale per la raccolta di eventi dal sistema operativo — quali *syscall*, pacchetti di rete, eventi di file system o segnali di performance — senza richiedere una copia dei dati verso lo spazio utente fino al momento dell'effettiva elaborazione. La libreria **Goccia** impiega la libreria *open-source cilium/ebpf* [42], un *wrapper* Go-nativo che semplifica il caricamento e la gestione di programmi *eBPF* compilati, eliminando la necessità di dipendenze esterne come *libbpf* scritta in C.

L'aspetto più distintivo dello *eBPF stage* è la sua astrazione generica rispetto ai tipi di dati. A differenza degli altri *Ingress stage*, che operano su tipi di messaggio predefiniti, lo *eBPF stage* utilizza i *generics* del Go per permettere agli utenti di definire completamente la struttura dei dati che il programma *eBPF* invia nel *ringbuffer*. Questo approccio consente massima flessibilità: ogni programma *eBPF* produce dati in un formato specifico, e tramite i *generics*, lo *stage* può deserializzare automaticamente i dati grezzi nel tipo Go appropriato.

La configurazione, incapsulata in `EBPFConfig[0, OPtr]`, si compone di tre componenti funzionali critici:

- **LoadFn**: Una funzione che carica la specifica *eBPF* compilata (generata da *bpf2go*, uno strumento che converte codice *eBPF* scritto in C nelle *bindings* corrispondenti in Go). Questa funzione ritorna una `ebpf.CollectionSpec`, ossia la rappresentazione in memoria del programma *eBPF* compilato.
- **LinkFn**: Una funzione che “attacca” il programma *eBPF* a un punto di *hook* nel kernel. A seconda del tipo di programma *eBPF*, l'*hook* può essere una

syscall, un'interfaccia di rete (*XDP*), un punto di traccia kernel (*kprobe/uprobe*), oppure altri. Questa funzione ritorna un `link.Link` [43], che rappresenta la connessione attiva tra il programma *eBPF* e il kernel.

- **RingBufferGetter**: Una funzione che estrae la map *eBPF* di tipo *ring buffer* dagli oggetti caricati. Il *ring buffer* rappresenta il meccanismo di comunicazione tra il codice *eBPF* in *kernel space* e l'applicazione in *user space*: il programma *eBPF* scrive dati nel *ring buffer* e lo *stage* legge continuamente da esso.

Un parametro opzionale è `UseUnsafe`, il quale controlla la strategia di deserializzazione dei dati. Se impostato a `true`, i dati grezzi dal *ring buffer* vengono castati direttamente a struct Go tramite `unsafe.Pointer`, operazione molto veloce ma che richiede una corrispondenza bit-per-bit tra il layout della struct C del programma *eBPF* e la struct Go. Se impostato a `false` (default), viene utilizzato `binary.Read` con decodifica *LittleEndian*, approccio più lento ma robusto e portabile. Opzionalmente, `CollectionOptions` consente di passare opzioni di caricamento avanzate, quali limiti di memoria, selezioni di programmi specifici, o configurazioni di verifica personalizzate.

L'inizializzazione dello *eBPF stage* è complessa e coinvolge molteplici fasi critiche. Nel metodo `Init`, la prima operazione è rimuovere i limiti di memoria bloccata tramite `rlimit.RemoveMemlock` [44]: i programmi *eBPF* richiedono che determinati buffer (come il *ring buffer*) siano *pinned* in memoria fisica, e il kernel di default limita quanto spazio un processo utente possa bloccare (tipicamente 64 KB). Questa operazione preliminare è essenziale per consentire l'allocazione della memoria necessaria ai programmi *eBPF* senza violare i vincoli di sistema.

Dopo aver rimosso questo limite, la specifica *eBPF* viene caricata tramite `LoadFn` e i programmi compilati vengono istanziati nel kernel tramite `spec.LoadAndAssign`. Successivamente, il programma viene attaccato al punto di *hook* tramite `LinkFn`, attivando così l'esecuzione del codice *eBPF* nel kernel. Infine, la map *ring buffer* viene estratta tramite `RingBufferGetter` e passata al metodo `init` della sorgente.

```
1 func (es *EBPFStage[T, O, OPtr]) Init(ctx context.Context)
   error {
2     // Remove resource limits for locked memory
3     if err := rlimit.RemoveMemlock(); err != nil {
4         es.tel.LogError("failed to remove memlock limits", err)
5         return err
6     }
7
8     // Load the compiled eBPF ELF file
9     spec, err := es.cfg.LoadFn()
10    if err != nil {
11        es.tel.LogError("failed to load eBPF spec", err)
```



```
12     return err
13 }
14
15 // Load the eBPF objects
16 var dummyObjs 0
17 objs := OPtr(&dummyObjs)
18 if err := spec.LoadAndAssign(objs,
19     es.cfg.CollectionOptions); err != nil {
20     es.tel.LogError("failed to load eBPF objects", err)
21     return err
22 }
23 es.objs = objs
24
25 // Get the link
26 link, err := es.cfg.LinkFn(objs)
27 if err != nil {
28     es.tel.LogError("failed to attach eBPF program", err)
29     return err
30 }
31 es.link = link
32
33 // Get the ring buffer map
34 ringBufferMap := es.cfg.RingBufferGetter(objs)
35
36 // ...
37 }
```

Listing 3.10: Inizializzazione del eBPF stage (ingress/ebpf.go, metodo Init)

Durante la fase di `Run`, il ciclo principale esegue la lettura dal *ring buffer* tramite il suo metodo `Read`, che rimane bloccante in attesa di nuovi record. Per ogni record disponibile, viene effettuata la deserializzazione del contenuto nel tipo Go appropriato e il messaggio risultante è inoltrato al *connector* di uscita.

Il tipo di messaggio `EBPFMessage[T]` è completamente generico e contiene un singolo campo `Data` di tipo `T` generico. Questo design semplificato riflette il fatto che i dati *eBPF* sono tipicamente *self-contained*: un evento di *syscall* o un pacchetto di rete contiene tutte le informazioni rilevanti all'interno della struct, senza metadati aggiuntivi da preservare come accade negli altri *Ingress stage*.

Il metodo `Close` dello *stage* è critico, in quanto deve rilasciare le risorse kernel in modo ordinato e sincronizzato. Innanzitutto, il *ring buffer reader* viene chiuso tramite `es.rb.Close()`. Successivamente, gli oggetti *eBPF* caricati vengono chiusi, scaricando i programmi dal kernel e liberando le mappe associate. Infine, il *link* viene chiuso tramite `es.link.Close()`, distaccando il programma *eBPF* dal punto di *hook* kernel. Questa sequenza garantisce che, al termine dello *stage*, tutte le

risorse kernel siano state rilasciate e lo stato del kernel sia coerente, prevenendo *resource leak* o comportamenti anomali.

3.4.6 File

Lo *File stage* rappresenta un *Ingress stage* progettato per leggere dati da file all'interno di una o più directory monitorate, offrendo un meccanismo flessibile per l'elaborazione di flussi di dati persistenti. A differenza dei precedenti *Ingress stage*, che consumano dati da sorgenti esterne in tempo reale, lo *File stage* consente di leggere sia file statici preesistenti sia file che vengono creati o modificati dinamicamente durante l'esecuzione della *pipeline*. Questo approccio lo rende ideale per scenari quali l'elaborazione offline di log, il monitoraggio di directory in cui sistemi esterni depositano file di dati, o l'implementazione di *pipeline* di elaborazione *batch* altamente reattive. La libreria **Goccia** sfrutta la libreria *open-source fsnotify* [45] per il monitoraggio dei cambiamenti nel file system, consentendo di rilevare automaticamente la creazione, modifica o eliminazione di file.

La configurazione dello *File stage* offre un controllo granulare su come i file vengono letti e processati. Il parametro `WatchedDirs` specifica l'elenco delle directory da monitorare tramite *fsnotify*. Ogni file presente o creato in tali directory sarà automaticamente sottoposto a lettura.

I parametri di lettura controllano aspetti tecnici dell'acquisizione dei dati. Il parametro fondamentale è `ChunkSize` (default 4 KB), il quale definisce la dimensione della finestra di lettura dal file tramite il *reader* bufferizzato offerto dalla libreria standard di Go, `bufio.Reader` [46]. Tuttavia, lo *File stage* introduce un meccanismo ibrido di delimitazione dei chunk tramite i parametri `ChunkDelim` (default `'\n'`) e `MaxChunkSize` (default 32 KB). Per impostazione predefinita, il lettore non si limita a restituire semplici chunk di dimensione fissa, ma continua a leggere oltre il `ChunkSize` finché non incontra il delimitatore (tipicamente newline), oppure raggiunge la dimensione massima. Questo consente di gestire automaticamente linee di lunghezza variabile senza frammentarle, comportamento particolarmente utile per file di log dove ogni riga rappresenta un record logico indivisibile. Tale comportamento può essere disabilitato, tornando a una lettura sempre di dimensione fissa.

Il parametro `ForceReRead` (default `false`) controlla il comportamento in caso di riapertura di un file. Se `false`, il lettore memorizza l'*offset* dell'ultima lettura e riprende da lì; se `true`, la lettura riinizia dall'inizio del file, opzione utile per scenari di *replay* o elaborazione idempotente. Il parametro `CloseDebounce` (default 1 secondo) specifica il tempo di attesa dopo il raggiungimento di EOF prima di chiudere effettivamente il file. Questo meccanismo è fondamentale in scenari in cui il file viene modificato frequentemente: invece di chiudere e riaprire il

file ripetutamente, il lettore attende che siano trascorsi T secondi di inattività, riducendo così l'*overhead* di aperture e chiusure successive.

Lo *File stage* implementa un'architettura distribuita basata su molteplici *reader*, uno per ogni file in lettura. Ogni *reader* è eseguito su una goroutine dedicata e comunica tramite una coda condivisa (*fan-in*) con il *bridge*, una goroutine ausiliaria responsabile di inoltrare i messaggi al *connector* di uscita. Questo design consente di leggere parallelamente da molteplici file, mentre la sincronizzazione tramite *fan-in* garantisce un ordinamento coerente dei messaggi verso il resto della *pipeline*.

<disegno stage, reader, fan-in>

Lo *stage* mantiene una mappa di *reader* attivi (**readers**) e monitora continuamente la directory tramite `fsnotify.Watcher` [47]. Quando *fsnotify* segnala un evento (creazione, modifica, rimozione di file), viene determinata l'azione appropriata: se il file è stato creato o modificato, il lettore è avviato (oppure creato e avviato se non esiste già); se il file è stato eliminato o rinominato, il lettore è chiuso e rimosso dalla mappa. La lettura iniziale dei file preesistenti avviene in `readExistingFiles`, una routine che scansiona le directory configurate e avvia i *reader* per tutti i file già presenti, poiché *fsnotify* non genera eventi per file che esistevano prima dell'inizio del monitoraggio.

```
1 // ...
2
3 // Before starting the watcher, read all the existing files
4 fs.readExistingFiles(ctx)
5
6 for {
7     select {
8     case <-ctx.Done():
9         return
10
11     case event, ok := <-fs.watcher.Events:
12         if !ok {
13             return
14         }
15
16         // Handle the fsnotify event
17         fs.handleEvent(ctx, event)
18
19     case err, ok := <-fs.watcher.Errors:
20         if !ok {
21             return
22         }
23
24         fs.tel.LogError("watcher error", err)
25     }
```

26 }

Listing 3.11: Ciclo principale del File stage (ingress/file.go, metodo run)

```
1 func (fs *fileSource) handleEvent(ctx context.Context,
   event fsnotify.Event) {
2     path := event.Name
3
4     // Handle file deletion/renaming
5     if event.Op&fsnotify.Remove == fsnotify.Remove ||
6        event.Op&fsnotify.Rename == fsnotify.Rename {
7
8         if fs.hasReader(path) {
9             fs.removeReader(path)
10        }
11
12        return
13    }
14
15    // Handle file creation
16    if event.Op&fsnotify.Create == fsnotify.Create {
17        if fs.hasReader(path) {
18            fs.startReader(ctx, path)
19        } else {
20            fs.addAndStartReader(ctx, path)
21        }
22
23        return
24    }
25
26    // Handle file modification
27    if event.Op&fsnotify.Write == fsnotify.Write {
28        if fs.hasReader(path) {
29            fs.startReader(ctx, path)
30        } else {
31            fs.addAndStartReader(ctx, path)
32        }
33
34        return
35    }
36 }
```

Listing 3.12: Gestione eventi fsnotify del File stage (ingress/file.go, metodo handleEvent)

Ogni *reader* mantiene una macchina a stati con quattro stati principali: **Idle** (non ancora avviato), **Started** (in lettura attiva), **Paused** (in attesa dopo EOF), e **Closed** (chiuso e liberato). Il ciclo di lettura del *reader* legge chunk sequenziali dal file. Quando la ricerca del delimitatore è abilitata, il lettore applica la seguente logica: dopo ogni lettura, verifica se l'ultimo byte del chunk è il delimitatore; se non lo è, continua a leggere byte aggiuntivi finché non trova il delimitatore o raggiunge **MaxChunkSize**. Questo approccio ibrido combina l'efficienza della lettura bufferizzata con la correttezza logica delle linee complete.

Quando EOF è raggiunto, il lettore entra nello stato **Paused** e attende per **CloseDebounce**. Se durante questo periodo il file viene modificato e *fsnotify* invia un evento di scrittura, il lettore è riavviato; altrimenti, dopo il *timeout*, il file è chiuso e il *reader* terminato. Questo meccanismo di *debounce* riduce drasticamente la frequenza di aperture e chiusure, specialmente in scenari in cui file di log vengono scritti incrementalmente.

Anche il messaggio dello *File stage* implementa l'interfaccia di serializzazione. La struct **FileMessage** contiene metadati specifici per i file: **Path** (il percorso del file), **Chunk** (il buffer contenente i dati letti), **ChunkSize** (la dimensione effettiva del chunk), **Offset** (l'*offset* nel file prima che il chunk fosse letto), e **DelimiterFound** (un booleano che indica se il chunk termina con il delimitatore configurato). Tali metadati consentono ai *Processor stage* a valle di accedere sia ai dati sia alle informazioni di contesto, abilitando scenari sofisticati quali la rielaborazione selettiva di range di file o il debug basato sulla posizione.

Lo *File stage* espone tre metriche: **readers** (numero totale di *reader* gestiti), **active_readers** (numero di *reader* in lettura attiva), e **read_bytes** (numero totale di byte letti). Inoltre, ogni chunk è accompagnato da uno span *OpenTelemetry* che registra la dimensione del chunk, la presenza di dati letti oltre il **ChunkSize**, e il percorso del file, fornendo visibilità completa sulla performance di lettura.

3.5 Worker Pool

Il Worker Pool rappresenta un pattern architetturale fondamentale per la gestione efficiente del parallelismo controllato all'interno della libreria **Goccia**. Diversamente dall'approccio diretto di generare una goroutine per ogni task in arrivo, il Worker Pool mantiene un numero limitato di goroutine riutilizzabili che elaborano task provenienti da una coda condivisa. Questa strategia consente di evitare la saturazione delle risorse di sistema causata dalla creazione eccessiva di unità di lavoro concorrenti, riducendo al contempo l'overhead del context switching e della gestione della memoria.

Il Worker Pool della libreria **Goccia** costituisce il nucleo esecutivo condiviso sia dai Processor Stage che dagli Egress Stage, garantendo scalabilità dinamica e

throughput elevato tramite meccanismi di fan-out, fan-in e autoscaling adattivo.

L'architettura del Worker Pool si articola in quattro componenti principali: il Fan-Out, il Fan-In, lo Scaler e i Worker stessi. Il Fan-Out rappresenta il meccanismo di distribuzione dei task dai produttori verso i worker. Concretamente, esso incapsula un ring buffer lock-free di tipo Single Producer Multiple Consumer (SPMC). Questo buffer consente a un singolo stage a monte di inserire messaggi al suo interno, i quali vengono poi estratti in modalità concorrente da molteplici worker.

Il Fan-In realizza la funzione duale di aggregazione: raccoglie i risultati prodotti da molteplici worker e li ricompone sequenzialmente in un unico flusso di output. Esso utilizza un ring buffer lock-free di tipo Multiple Producer Single Consumer (MPSC). In questo caso, i worker scrivono i messaggi elaborati, mentre una singola goroutine a valle legge dal buffer, garantendo così l'ordinamento e la serializzazione dei risultati prima del trasferimento allo stage successivo.

Lo Scaler rappresenta il componente responsabile dell'adattamento dinamico del numero di worker in esecuzione, in funzione del carico corrente. Esso monitora periodicamente la profondità della coda di ingresso (fan-out) e confronta tale valore con soglie configurabili per determinare se incrementare o ridurre il numero di worker attivi.

Infine, i Worker sono le unità di elaborazione effettive che estraggono task dalla coda di fan-out, invocano la logica specifica di uno stage, e depositano i risultati nella coda di fan-in.

Lo Scaler implementa un algoritmo di autoscaling adattivo che regola dinamicamente il numero di worker attivi in funzione della profondità della coda di ingresso. La logica di scaling è incapsulata nella struct `Scaler`, definita nel package `internal/pool/scaler`. Il parametro critico è `QueueDepthPerWorker` (default 64), che rappresenta il numero target di task in coda per ciascun worker. Se il rapporto tra il numero di task pendenti e il numero di worker correnti supera questa soglia, lo scaler decide di incrementare il numero di worker.

La decisione di scaling è presa periodicamente, con un intervallo configurabile tramite `AutoScaleInterval` (default 3 secondi). Durante ogni ciclo di valutazione, lo scaler calcola il numero ottimale di worker tramite la formula. Il numero di worker è vincolato dai limiti `MinWorkers` (default 1) e `MaxWorkers` (default `runtime.NumCPU`).

Per quanto riguarda la riduzione del numero di worker (scale-down), lo scaler implementa un meccanismo di exponential backoff. Quando viene rilevata la necessità di ridurre il numero di worker, il parametro `ScaleDownFactor` (default 0.1) determina la frazione di worker da rimuovere, mentre il parametro `ScaleDownBackoff` (default 1.5) controlla il fattore di incremento esponenziale del tempo di attesa tra riduzioni consecutive. Questo meccanismo previene oscillazioni eccessive del numero di worker in presenza di carichi fluttuanti, garantendo stabilità operativa.

Lo Scaler notifica il Worker Pool tramite due canali, uno per segnalare la necessità di creare e far partire un nuovo Worker, e l'altro per terminarlo.

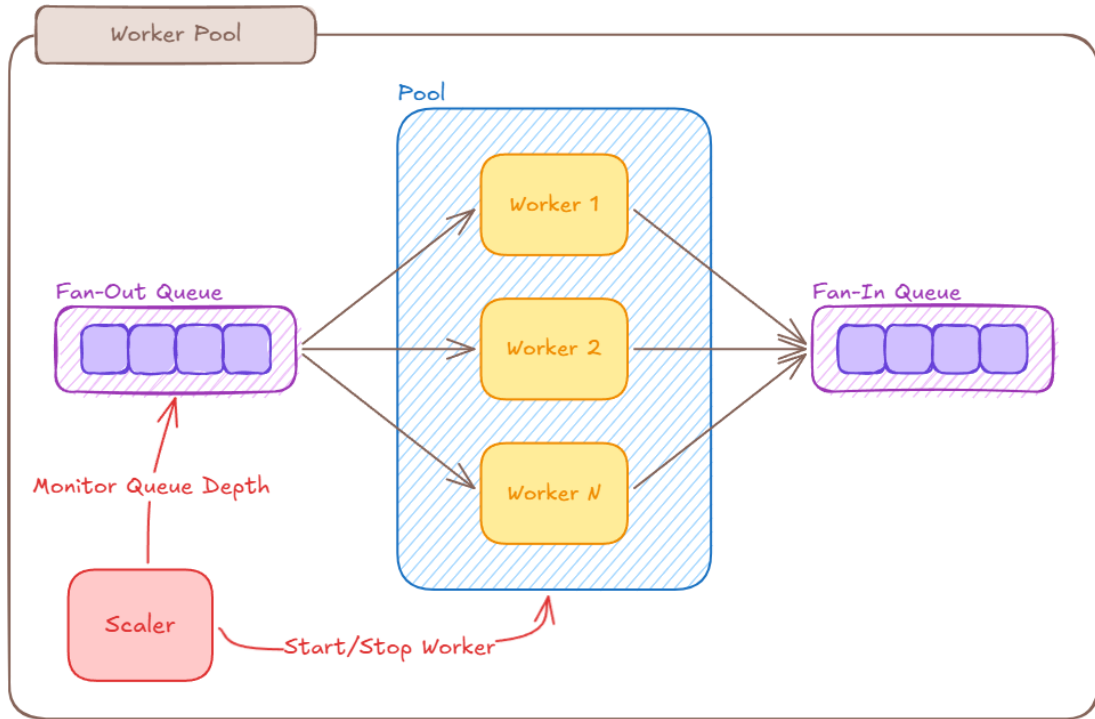


Figura 3.2: Worker Pool nella libreria Goccia

Il Worker Pool espone molteplici metriche per monitorare lo stato operativo e identificare eventuali colli di bottiglia. Le metriche principali includono il numero di task pendenti nella coda di fan-out e il numero di worker attivi.

Il Worker Pool è utilizzato sia dai Processor Stage che dagli Egress Stage quando configurati in modalità Worker Pool (`runningMode=StageRunningModePool`). Nel caso dei Processor Stage, il Worker Pool gestisce l'elaborazione concorrente di messaggi provenienti da uno stage a monte, applicando trasformazioni definite dall'utente e inoltrando i risultati allo stage successivo tramite il connector di output. Per gli Egress Stage, il Worker Pool coordina l'invio parallelo di messaggi verso destinazioni esterne quali database, broker di messaggistica, o socket di rete.

La decisione di utilizzare il Worker Pool o un singolo worker (Single Worker Mode) è determinata dalla configurazione dello stage. La modalità pool è particolarmente vantaggiosa in scenari ad alto throughput, dove l'elaborazione di ciascun messaggio è computazionalmente costosa e il parallelismo può sfruttare efficacemente le risorse multi-core disponibili.

3.6 Processor Stage

Un *Processor stage* rappresenta un'unità di elaborazione intermedia all'interno della *pipeline* di **Goccia**. Diversamente dagli *Ingress stage*, che fungono da punto di ingresso, i *Processor stage* ricevono messaggi da uno *stage* precedente tramite un *connector* di input, li trasformano secondo una logica specifica, e inoltrano il risultato verso lo *stage* successivo tramite un *connector* di output. In altri termini, ogni *Processor stage* implementa una funzione di trasformazione pura: $f(\text{messaggio_in}) \rightarrow \text{messaggio_out}$, consentendo di costruire *pipeline* complesse attraverso la composizione di trasformazioni modulari. Nel contesto della libreria **Goccia**, i *Processor stage* sono progettati per essere altamente flessibili e riusabili, mantenendo una chiara separazione tra l'infrastruttura di orchestrazione e la logica di elaborazione specifica.

3.6.1 Cannelloni

Il *Cannelloni stage* rappresenta un *Processor stage* specializzato per la serializzazione e deserializzazione di messaggi nel formato *Cannelloni* [48], uno standard di incapsulamento per messaggi *CAN* (*Controller Area Network*) progettato per il trasporto su reti IP. La libreria **Goccia** fornisce due varianti dello *stage*: il *Cannelloni Decoder stage*, che trasforma dati *Cannelloni* grezzi in messaggi *CAN* strutturati, e il *Cannelloni Encoder stage*, che compie l'operazione inversa. Sebbene logicamente rappresentino un unico tipo di elaborazione (bidirezionale), sono implementati come due *stage* separati per conformarsi al modello unidirezionale di trasformazione della *pipeline*.

Il protocollo *CAN* è ampiamente utilizzato in ambito automotive e nei sistemi embedded per la comunicazione *real-time* a bassa latenza. Tuttavia, il *CAN* tradizionale è limitato a reti locali (bus seriale). Il formato *Cannelloni* risolve questo vincolo consentendo l'incapsulamento di molteplici messaggi *CAN* in un singolo pacchetto *UDP*, abilitando il trasporto su reti IP. Questo approccio è particolarmente rilevante in scenari dove i dati *CAN* provengono da *gateway* remoti e devono essere elaborati centralmente.

Il formato *Cannelloni* è organizzato gerarchicamente. L'header del frame è composto da 5 byte: il campo **version** (1 byte) specifica la versione del protocollo; **opCode** (1 byte) contiene il codice operativo; **sequenceNumber** (1 byte) è il numero sequenziale del frame, utilizzato per il riordino e la rilevazione di perdite; **messageCount** (2 byte, *big-endian*) indica il numero di messaggi *CAN* contenuti nel frame. Segue quindi la sequenza dei messaggi *CAN*, ciascuno con lunghezza variabile (minimo 5 byte): **canID** (4 byte, *big-endian*) è l'identificatore del messaggio; **dataLen** (1 byte) specifica la lunghezza dei dati (0-8 per *CAN* 2.0, 0-64 per *CAN* FD), con il bit più significativo (0x80) che indica se il messaggio è *CAN*

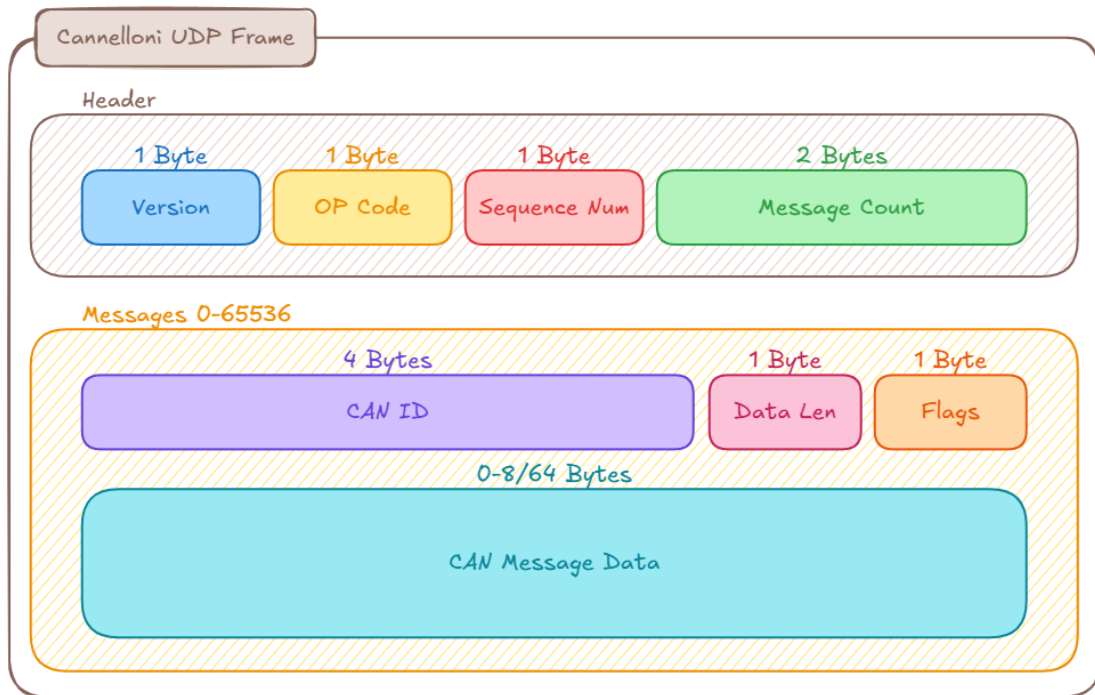


Figura 3.3: Rappresentazione di un frame del protocollo Cannelloni UDP

FD; `canFDFlags` (1 byte, opzionale) contiene flag specifici di *CAN FD*, presenti solamente se il bit di *CAN FD* è impostato; `data` (variabile) è il payload del messaggio.

Il *Cannelloni Decoder stage* riceve dati *Cannelloni* grezzi (tipicamente da uno *UDP stage*) e li decodifica in messaggi *CAN* strutturati, eseguendo il parsing sequenziale del buffer contenente i dati grezzi. Questo buffer è ottenuto richiamando il metodo `GetBytes` definito dall'interfaccia di serializzazione utilizzata dagli *Ingress stage* quali *UDP*, *TCP*, *File* e *Kafka*. Il *decoder* verifica innanzitutto che il buffer contenga almeno 5 byte per l'header, quindi estrae i campi utilizzando accesso diretto ai byte e `binary.BigEndian` per i campi multi-byte. Per ogni messaggio *CAN* nel frame, procede sequenzialmente estraendo l'identificatore, la lunghezza dei dati, e verificando se il messaggio è di tipo *CAN FD*, per concludere copiando il payload nel messaggio deserializzato.

Il *Cannelloni Encoder stage* compie l'operazione inversa: riceve messaggi di tipo `CannelloniMessage` (tipicamente generati da uno *stage* precedente o costruiti manualmente) e li serializza in *buffer* di byte. Questo *buffer* viene successivamente inserito nel campo apposito del tipo `CannelloniEncodedMessage`.

Il messaggio `CannelloniMessage` implementa l'interfaccia `message.ReOrderable`, la quale espone il metodo `GetSequenceNumber`. Questo consente allo *stage ROB*

(*ReOrder Buffer*) posizionato a valle del decoder di riordinare i frame in arrivo in caso di perdita di pacchetti o arrivo fuori ordine dalla rete. Il numero sequenziale è mantenuto durante la codifica, garantendo che il riordino sia semanticamente corretto.

Sia il decoder che l'encoder generano uno span *OpenTelemetry* per ogni frame elaborato, registrando il numero di messaggi *CAN* contenuti nel frame tramite l'attributo `message_count`. Questo fornisce visibilità granulare sui volumi di dati elaborati e sulla struttura interna dei frame, facilitando il monitoraggio della salute della *pipeline* e la diagnostica di anomalie.

3.6.2 CAN

Il *CAN stage* rappresenta un *Processor stage* specializzato per la decodifica di messaggi *CAN* (*Controller Area Network*) da un formato grezzo a una rappresentazione strutturata di segnali decodificati. A differenza dello *Cannelloni stage*, che gestisce il livello di trasporto e incapsulamento su reti IP, il *CAN stage* opera al livello semantico, estraendo i singoli segnali contenuti nei messaggi *CAN* secondo una definizione di schema.

Un messaggio *CAN* grezzo è semplicemente un identificatore (*CAN ID*) associato a una serie di byte (payload). Tuttavia, i byte non sono auto-descrittivi: la loro interpretazione dipende completamente dallo schema di decodifica definito dai progettisti del sistema. Ad esempio, un payload di 8 byte potrebbe contenere pressione, temperatura, stato di allarme e altre grandezze fisiche, ognuna occupando un intervallo di bit specifico all'interno del buffer. Questi campi contenuti nel payload prendono il nome di *segnali*. Lo *CAN stage* traduce questa rappresentazione binaria in segnali nominati con valori tipizzati (booleani, interi, numeri in virgola mobile, enumerazioni), rendendo i dati direttamente intelligibili alle applicazioni a valle della *pipeline*.

La libreria **Goccia** integra la libreria *acmelib* [49], un *framework* Go per la definizione e manipolazione di schemi *CAN*. La configurazione dello *CAN stage* accetta un elenco di oggetti `acmelib.Message`, ognuno dei quali rappresenta il modello di un messaggio *CAN* specifico (identificato da un *CAN ID*) e contiene la definizione del layout dei segnali.

All'inizializzazione dello *stage*, viene costruita una mappa che associa a ogni *CAN ID* la funzione di decodifica corrispondente. Durante l'elaborazione, quando arriva un messaggio grezzo, il *decoder* recupera la funzione dalla mappa e l'invoca sul payload, ottenendo una lista di segnali decodificati con nome, valore grezzo e valore tipizzato.

Il messaggio ritornato dallo *stage* incorpora la lista dei segnali decodificati. Per ogni segnale viene utilizzato un approccio *tagging* tramite il campo `Type` per indicarne il tipo (booleano, intero, numero in virgola mobile, enumerazione), e

il valore effettivo è memorizzato nel corrispondente campo tipizzato (`ValueFlag`, `ValueInt`, `ValueFloat`, `ValueEnum`). Questo approccio consente una gestione *type-safe* senza ricorrere a interfacce generiche o *reflection*, a scapito di un maggiore utilizzo di memoria.

3.6.3 CSV

Lo *CSV stage* rappresenta un *Processor stage* specializzato per la serializzazione e deserializzazione di dati in formato *CSV* (*Comma-Separated Values*). Come lo *Cannelloni stage*, la libreria **Goccia** fornisce due varianti: il *CSV Decoder stage*, che trasforma dati *CSV* grezzi in messaggi strutturati, e il *CSV Encoder stage*, che compie l'operazione inversa. Questo consente l'integrazione di sorgenti dati *CSV* (file, stream di rete) all'interno della *pipeline* e l'esportazione di risultati elaborati in formato *CSV*.

La configurazione dello *CSV stage* è basata su uno schema dichiarativo composto dalle definizioni delle colonne. Per ogni colonna viene specificato il nome, il tipo di dato (stringa, intero, numero in virgola mobile, booleano, *timestamp*) e, per *timestamp*, il layout di *parsing* (ad esempio RFC3339, ISO 8601, o layout personalizzato). Questo approccio consente di gestire file *CSV* con tipi di dati eterogenei senza perdere informazioni di tipo durante la deserializzazione.

Il *CSV Decoder stage* riceve dati *CSV* grezzi (tipicamente da uno *File stage* o *UDP stage*) e li decodifica in messaggi strutturati. L'algoritmo di decodifica scansiona sequenzialmente il buffer di dati grezzi, carattere per carattere, accumulando i dati di una colonna in un buffer di stringhe. Quando incontra un byte che fa parte di un carattere multi-byte UTF-8 (bit più significativo impostato), utilizza `utf8.DecodeRune` [50] per estrarre correttamente il carattere *Unicode*, garantendo la compatibilità con dataset internazionali. I delimitatori di colonna (virgola) e di riga (*newline*) segnalano il completamento di un valore e l'inizio di un nuovo. Il decoder gestisce correttamente righe incomplete, *newline* assenti all'ultimo valore, e *carriage return* (`\r`) (comune nei file *CSV* esportati da sistemi Windows). Per ogni colonna completata, il decoder estrae il tipo dalla definizione dello schema e invoca il metodo di decodifica specializzato sfruttando il pacchetto `strconv` della libreria standard [51]. Se la conversione del tipo fallisce (ad esempio, una stringa non convertibile a intero), nel messaggio generato dallo *stage* viene impostato a `false` un flag che segnala l'insuccesso della decodifica. Questo approccio di *soft-validation* è utile in scenari dove dati sporchi sono comuni.

Il *CSV Encoder stage* compie l'operazione inversa: riceve messaggi `CSVMessage` (liste di righe) e li serializza in buffer *CSV* grezzi. L'encoder, per ogni riga e colonna, scrive il valore tipizzato nello stream testuale, utilizzando le funzioni `strconv` per convertire interi e float a stringhe. Nel caso in cui una colonna non dovesse contenere un dato valido, l'encoder provvede a scrivere un valore di *default*,

garantendo che il chunk di file *CSV* risultante sia sempre ben formattato, anche con dati incompleti. L'encoder utilizza un `strings.Builder` con capacità pre-allocata per ridurre le allocazioni dinamiche, massimizzando l'efficienza della serializzazione.

Un aspetto notevole è l'uso di *object pooling* per i messaggi `CSVMessage`, in maniera analoga ai messaggi risultanti dallo *UDP Ingress stage*. Viene utilizzato un `sync.Pool` per riciclare le istanze dei messaggi, riducendo la pressione sul *garbage collector*. Quando un messaggio è distrutto, viene ripulito e restituito al pool per essere riutilizzato nella prossima decodifica. Questo è particolarmente importante in scenari con alto *throughput* dove decine di migliaia di messaggi *CSV* vengono processati al secondo. Similarmente agli altri *Processor stage*, il *CSV Decoder stage* è generico rispetto al tipo di input, consentendo di accettare dati *CSV* da qualunque *stage* che ritorni un messaggio implementante l'interfaccia di serializzazione. Il flusso tipico potrebbe essere: *File stage* → *CSV Decoder stage* → elaborazione specifica del dominio, oppure il percorso inverso per l'esportazione.

3.6.4 Custom

Lo *Custom stage* è un *Processor stage* straordinariamente flessibile che consente agli utenti di implementare logica di elaborazione arbitraria tramite l'implementazione di un'interfaccia personalizzata. A differenza degli *stage* predefiniti (*Filter*, *CAN*, *CSV*), che implementano trasformazioni specifiche del dominio, il *Custom stage* fornisce un'astrazione generica che delega completamente la logica di *processing* all'utente, consentendo di elaborare qualunque tipo di messaggio e produrre qualunque tipo di output.

Il cuore dello *Custom stage* è l'interfaccia `CustomHandler`, che richiede di implementare quattro metodi:

- **Init:** Invocato una sola volta al momento dell'inizializzazione dello *stage*, prima che qualunque messaggio sia processato. Utile per inizializzare risorse, connessioni, cache, o stato interno specifico dell'applicazione.
- **Handle:** Invocato per ogni messaggio in ingresso. Riceve il messaggio in input in sola lettura e un puntatore a quello di output pre-allocato, che l'handler deve popolare con il risultato della trasformazione. Questo design evita allocazioni per ogni messaggio e consente all'handler di controllare completamente l'output.
- **Close:** Invocato una sola volta al momento della chiusura dello *stage*, utile per rilasciare risorse acquisite in **Init**.
- **SetTelemetry:** Utilizzato dalla *pipeline* per fornire all'handler accesso al sottosistema di telemetria, permettendo all'utente di aggiungere log, metriche e *span* di tracciamento personalizzati.

Lo *stage* è implementato con tre parametri di tipo generici, consentendo composizioni arbitrarie: ad esempio, uno *stage* potrebbe accettare messaggi `CannelloniMessage` e produrre messaggi `CANMessage`, oppure ricevere `CSVMessage` e produrre un tipo completamente personalizzato.

Quando lo *stage* è eseguito in modalità *worker pool*, ogni *worker* riceve una copia della istanza `CustomHandler` fornita. Questo significa che lo stato interno dell'handler non è condiviso tra *worker* (a meno che non sia esplicitamente sincronizzato tramite meccanismi concorrenti quali `sync.Mutex` o canali). Questo design garantisce *thread-safety* per costruzione, ma obbliga l'utente a gestire manualmente lo stato condiviso se necessario.

La libreria fornisce `CustomHandlerBase`, una classe base che implementa già `Init`, `Close` e `SetTelemetry` con comportamenti di *default* (*no-op* per i primi due, semplice assegnamento per il terzo). Gli utenti possono ereditare da questa classe e implementare solamente il metodo `Handle`, semplificando il *boilerplate*.

```
1 type MyHandler struct{
2     processor.CustomHandlerBase
3 }
4
5 func (h *MyHandler) Handle(ctx context.Context, msgIn
6     msgInType, msgOut msgOutType) error {
7     // My custom logic ...
8
9     return nil
10 }
```

Listing 3.13: Esempio implementazione interfaccia `CustomHandler`

La configurazione dello *Custom stage* include un campo `Name` che identifica univocamente lo *stage* nella telemetria e nella tracciatura, consentendo di distinguere tra molteplici *stage* personalizzati nella stessa *pipeline*. Lo *stage* supporta inoltre sia *Single Worker Mode* che *Worker Pool Mode*, fornendo all'utente il controllo sul livello di parallelismo.

In sintesi, il *Custom stage* è lo strumento ideale per scenari in cui la logica di elaborazione è specifica dell'applicazione, dinamica, o semplicemente troppo niche per giustificare l'implementazione di uno *stage* predefinito. Esempi includono: *machine learning inference*, trasformazioni semantiche complesse, aggregazioni statistiche, deduplicazione, arricchimento dati da fonti esterne, o qualunque operazione personalizzata di dominio.

3.6.5 Filter

Lo *Filter stage* è un *Processor stage* generico che applica un predicato di filtraggio ai messaggi in transito nella *pipeline*, decidendo per ciascuno se debba proseguire verso gli *stage* successivi oppure essere scartato. Opera quindi come un filtro basato su una funzione booleana definita dall'utente, che incapsula la logica di accettazione/rifiuto a livello applicativo (ad esempio: tenere solo messaggi con un certo campo valorizzato, scartare quelli di errore, limitare il flusso a un sottoinsieme di segnali rilevanti).

Dal punto di vista dell'esecuzione, il *Filter stage* invoca la funzione di filtro sul messaggio in ingresso: se il predicato restituisce **false**, il messaggio viene marcato come *dropped* tramite il metodo dedicato nel messaggio e non sarà processato dagli *stage* successivi; se restituisce **true**, il messaggio viene semplicemente propagato in uscita senza modifiche. In questo modo lo *stage* è completamente trasparente rispetto al contenuto e si limita ad agire come “valvola” di selezione.

Grazie all'uso di *generics* sul tipo `T`, il *Filter stage* può essere inserito in qualunque punto della *pipeline*, indipendentemente dal tipo di messaggio in transito. Tipici esempi includono l'applicazione di filtri su messaggi *CAN* decodificati, su righe *CSV* strutturate, o su eventi provenienti da *Ingress stage* differenti, senza necessità di adattatori intermedi.

3.6.6 Tee

Lo *Tee stage* è un *Processor stage utility* specializzato nel duplicare i messaggi verso molteplici output *connector*, consentendo il *branching* della *pipeline*. Il suo nome richiama l'omonimo comando *Unix tee*, che copia lo standard input sia verso lo standard output che verso un file. Analogamente, questo *stage* prende un messaggio in ingresso e lo distribuisce parallelamente verso *N* output *connector*, permettendo a molteplici sottopipeline di elaborare indipendentemente lo stesso dato.

Un aspetto cruciale del *Tee stage* è che non esegue una copia profonda dei dati. Al contrario, la libreria **Goccia** implementa un sistema di *reference counting* sui messaggi: quando il metodo `Clone` del messaggio è invocato, viene creato un nuovo envelope che riferisce lo stesso payload sottostante, incrementando un contatore di riferimenti. Solo quando il contatore scende a zero (ossia quando tutti i cloni sono distrutti), il payload effettivo è deallocato. Questo design consente al *Tee stage* di distribuire messaggi a costo praticamente costante, indipendentemente da quanto grandi siano i dati sottostanti. Di fatto vengono copiati solamente i metadati del messaggio, operazione resa possibile dal fatto che gli *stage* a valle utilizzano il messaggio in input come un'istanza *read-only*.

Il *Tee stage* legge un messaggio dal *connector* di input e lo clona per ogni output *connector* configurato. Non vi è alcuna elaborazione o filtraggio: ogni clone è identico all'originale in termini di payload e metadati. Se uno degli output

connector non riesce a ricevere il clone (ad esempio, è pieno o chiuso), solo quel ramo è interessato; gli altri continuano a ricevere i loro cloni. L'errore è loggato ma non interrompe la distribuzione verso gli altri output.

A differenza dei *Processor stage* visti in precedenza, il *Tee stage* non può utilizzare un *worker pool*. È invece implementato come *stage standalone* che gestisce direttamente il ciclo di lettura e distribuzione. La configurazione è minimale: l'utente specifica semplicemente il *connector* di input e un elenco di output *connector* tramite un parametro *variadic*. Al momento dell'inizializzazione, lo *stage* valida che almeno un output *connector* sia stato fornito.

Il *Tee stage* è indispensabile in scenari dove i dati devono essere elaborati in parallelo da molteplici sottosistemi: ad esempio, un singolo flusso di messaggi *CAN* potrebbe essere distribuito simultaneamente a uno *stage* di storage, a uno di analisi *real-time* e a uno di *machine learning inference*. Il *Tee stage* garantisce che ogni sottopipeline riceva una copia coerente del messaggio originale, senza contaminazione tra rami e senza duplicazione effettiva dei dati.

3.6.7 Reorder Buffer

Lo *Reorder Buffer* (ROB) *stage* è specializzato nel riordinamento di messaggi che arrivano fuori ordine, garantendo che vengano elaborati e inoltrati in ordine sequenziale basato su un numero di sequenza contenuto nei messaggi stessi. Questo *stage* può essere eseguito solamente in *Single Worker Mode*, poiché il riordinamento richiede uno stato coerente che non può essere facilmente sincronizzato tra *worker* paralleli.

Il cuore del ROB è un'architettura *dual-buffer*: il **primary buffer** e l'**auxiliary buffer**. Il **primary buffer** è più piccolo ed è ottimizzato per il caso comune dove i messaggi arrivano in ordine o con ritardi brevi. L'**auxiliary buffer** è più grande e accoglie messaggi che arrivano molto fuori ordine. I due buffer operano in tandem: quando l'**auxiliary buffer** raggiunge un certo livello di pienezza (configurabile tramite `FlushThreshold`, *default* 30%), il **primary buffer** è interamente liberato nel connettore di output e il contenuto dell'**auxiliary buffer** è trasferito nel **primary buffer**.

<disegno schema doppio ring buffer>

Quando un messaggio arriva allo *stage*, viene passato al ROB tramite il metodo `Enqueue`. Il ROB classifica il risultato dell'operazione di `enqueue` in una di quattro categorie:

- *In-Order*: il numero di sequenza corrisponde al prossimo numero atteso. Il messaggio è immediatamente inoltrato al connettore di output senza buffering. Questo rappresenta il caso ideale (*fast path*).

- *Primary*: il numero di sequenza è fuori ordine ma rientra nella finestra del `primary` buffer. Il messaggio è inserito in posizione corretta all'interno del buffer. In questo caso, il messaggio non rientra nel caso ideale, ma l'overhead rimane comunque contenuto.
- *Auxiliary*: il numero di sequenza è oltre la finestra del `primary` buffer ma rientra nella finestra dell'`auxiliary` buffer. Il messaggio è accodato nell'`auxiliary` buffer, ricadendo nella peggior casistica in termini di velocità.
- *Error*: il numero di sequenza è invalido (duplicato, troppo grande, fuori della finestra complessiva). Il messaggio è scartato e un errore è registrato nelle metriche opportune.

```
1 func (rob *ROB[T]) Enqueue(item T) (EnqueueStatus, error) {
2     seqNum := item.GetSequenceNumber()
3
4     // Check the sequence number validity
5     if !rob.primaryBuf.IsValidSize(seqNum) {
6         return EnqueueStatusErr, ErrSeqNumTooBig
7     }
8
9     // Initialize the ROB if Enqueue is called for the first
10    time ...
11
12    // Try to enqueue the item in the primary buffer
13    status, err := rob.enqueuePrimary(item)
14    if err == nil {
15        return status, nil
16    }
17
18    if errors.Is(err, ErrSeqNumDuplicated) {
19        return EnqueueStatusErr, err
20    }
21
22    // Enqueue the item in the auxiliary buffer
23    err = rob.enqueueAuxiliary(item)
24    return EnqueueStatusAuxiliary, err
25 }
```

Listing 3.14: Logica di Enqueue del ROB stage (internal/rob/rob.go, metodo Enqueue)

Entrambi i buffer mantengono una finestra di numeri di sequenza validi. La finestra del `primary` buffer copre il primo range (fino a `PrimaryBufferSize`), mentre la finestra dell'`auxiliary` buffer copre da `PrimaryBufferSize` alla dimensione

del buffer ausiliario, `AuxiliaryBufferSize`. Quando uno dei buffer esaurisce i numeri di sequenza disponibili (ad esempio, tutti i messaggi nella finestra sono stati processati), la finestra si sposta in avanti incrementando il numero di sequenza atteso successivo. Questo è implementato tramite il metodo `shiftLeft` che compatta i dati nel buffer e azzerà gli ultimi slot.

Per gestire efficientemente lo spazio nel buffer e identificare slot vuoti vs occupati, il ROB utilizza una *bitmap*: un array di byte dove ogni bit rappresenta uno slot nel buffer. Operazioni come `set` e `isSet` consentono di tracciare quali posizioni contengono messaggi e quali sono vuote con complessità $O(1)$. Il metodo `getConsecutive` scansiona la bitmap a partire dall'indice 0 e conta quanti bit consecutivi sono impostati, fornendo il numero di messaggi pronti per il dequeue sequenziale.

Un aspetto importante del ROB è il *timeSmoother*, che implementa un *Double Exponential Smoother* (noto anche come *Holt's Linear Exponential Smoothing*, descritto nella sezione 2.10) per regolare e levigare i timestamp associati ai messaggi. A differenza di tecniche di smoothing tradizionali che utilizzano un singolo parametro di smorzamento, il *Double Exponential Smoother* mantiene due componenti di stato: il *level* (stima del valore attuale) e il *trend* (stima della velocità di cambiamento).

Lo smoothing può essere abilitato o disabilitato tramite un apposito parametro di configurazione. Quando abilitato, il smoother applica due fattori di smorzamento: `EstimatorAlpha` (fattore di smoothing dei dati, *default* 0.8) e `EstimatorBeta` (fattore di smoothing del trend, *default* 0.5). Il *level* è aggiornato combinando il timestamp osservato con una previsione basata sul *level* e il *trend* precedenti, scalati dalla distanza sequenziale tra il messaggio corrente e quello precedente. Il *trend* è aggiornato combinando il cambiamento nel *level* con il *trend* precedente, permettendo al smoother di adattarsi a variazioni dinamiche nella frequenza di arrivo dei messaggi. Un vincolo di monotonia garantisce che il timestamp aggiustato non retrocede mai nel tempo, preservando la coerenza temporale del flusso di output indipendentemente dall'ordine di arrivo dei messaggi.

```
1 func (ts *timeSmoother[T]) adjust(item T) {
2     // Extract the current receive time
3     recvTime := item.GetReceiveTime()
4     currValue := float64(recvTime.UnixNano())
5
6     // Extract the current sequence number
7     // and the distance to the previous one
8     seqNum := item.GetSequenceNumber()
9     ts.prevSeqNum = seqNum
10    distance := getSeqNumDistance(seqNum, ts.prevSeqNum,
11                                   ts.maxSeqNum)
```

```
12 // Force the distance to be at least 1
13 if distance == 0 {
14     distance = 1
15 }
16
17 // Estimate the timestamp value and convert to a timestamp
18 estimatedValue := ts.estimator.estimate(currValue,
19     distance)
20 currTimestamp := time.Unix(0, int64(estimatedValue))
21
22 // Enforce monotonicity
23 if currTimestamp.Before(ts.prevTimestamp) {
24     currTimestamp = ts.prevTimestamp
25 } else {
26     ts.prevTimestamp = currTimestamp
27 }
28 item.SetTimestamp(currTimestamp)
29 }
```

Listing 3.15: Calcolo del timestamp corretto nel ROB stage (internal/rob/time_smoother.go, metodo adjust)

```
1 func (dee *doubleExponentialEstimator) estimate(value
2     float64, n uint64) float64 {
3     // ...
4     // Get the forecasted value based on the previous level
5     // and trend
6     prevForecasted := dee.prevLevel + dee.prevTrend*float64(n)
7     // Calculate the current level and trend to be used
8     // by the next item
9     currLevel := dee.alpha*value +
10         (1-dee.alpha)*(prevForecasted)
11     currTrend := dee.beta*(currLevel-dee.prevLevel) +
12         (1-dee.beta)*dee.prevTrend
13
14     dee.prevLevel = currLevel
15     dee.prevTrend = currTrend
16
17     dee.estimateCount++
18     return prevForecasted
19 }
```

Listing 3.16: Implementazione Double Exponential Smoothing (internal/rob/time_smoother.go, metodo estimate)

Il ROB implementa un meccanismo di reset basato su timeout. Durante l'esecuzione, lo *stage* legge i messaggi in ingresso con un timeout pari a `ResetTimeout` (*default* 100 ms) e se nessun messaggio arriva entro questo periodo, il buffer viene resettato e tutti i messaggi presenti al suo interno vengono inoltrati allo *stage* successivo. Questo protegge da situazioni di deadlock dove un messaggio critico per il riordinamento non arriva mai, causando un accumulo indefinito di messaggi nel buffer. Il reset è inoltre eseguito quando il connettore di input si chiude o quando il contesto di esecuzione è cancellato (`context.Done`).

Lo *stage* traccia metriche granulari per ogni categoria di enqueue, come il numero di messaggi ricevuti in ordine, quelli fuori ordine salvati nel `primary/auxiliary buffer`, infine il numero di errori e i reset.

Il ROB *stage* è essenziale in scenari dove il flusso di messaggi in ingresso subisce riordinamento dovuto a latenze variabili o buffering di rete. Esempi includono: raccolta di pacchetti trasmessi con protocolli lossy come *UDP*, i quali potrebbero essere ricevuti non in ordine, o telemetria *real-time* da sistemi remoti dove la variabilità di latenza è significativa.

In sintesi, il ROB *stage* è uno strumento sofisticato che trasforma un flusso potenzialmente disordinato in un flusso ordinato e temporalmente coerente, combinando tecniche di buffering adattivo, bitmap compatte per efficienza spaziale, *Double Exponential Smoothing* statistico, e timeout intelligenti per garantire output di alta qualità anche in condizioni avverse di rete, latenza, e sincronizzazione temporale.

3.7 Egress Stage

Un Egress stage rappresenta il punto terminale della pipeline di **Goccia**, responsabile dell'esportazione dei messaggi elaborati verso destinazioni esterne al sistema. Diversamente dai Processor stage, che trasformano messaggi e li inoltrano a stage successivi, gli Egress stage ricevono messaggi da uno stage precedente tramite un connector di input e li materializzano verso sink esterni quali database, broker di messaggistica, file system o socket di rete. In altri termini, ogni Egress stage implementa una funzione di output: $f(\text{messaggio_in}) \rightarrow \text{void}$, terminando così il flusso di elaborazione e rendendo i dati disponibili per sistemi downstream o per la persistenza.

3.7.1 UDP Egress

Lo *UDP stage* è un *Egress stage* specializzato nell'invio di messaggi verso un endpoint remoto tramite il protocollo *UDP*. Contrariamente allo *UDP Ingress stage*, che riceve dati grezzi da una socket *UDP*, l'*UDP Egress stage* conclude la *pipeline* trasmettendo messaggi elaborati a una destinazione esterna, fungendo da punto di uscita per il flusso di dati.

La configurazione dello *stage* è minimale: richiede semplicemente l'indirizzo IP di destinazione (con *default* 127.0.0.1) e la porta di destinazione (*default* 20000).

L'*UDP Egress stage* può ricevere messaggi serializzati da qualunque *Processor stage* che produca output serializzabile: dati *CSV*, messaggi *CAN* decodificati, output di elaborazioni personalizzate, eccetera. Non vi è accoppiamento di tipo; l'unico vincolo è che il messaggio implementi l'interfaccia **Serializable**.

Il cuore dello *stage* è il suo *worker*, il quale si occupa di estrarre il payload serializzato dal messaggio tramite `GetBytes()` e di trasmetterlo sul socket *UDP* mediante `conn.Write` [52].

Se l'operazione di scrittura fallisce (ad esempio, per perdita di connessione, *timeout* di rete, o satura del *buffer* del kernel), il *worker* registra l'errore ma non interrompe l'elaborazione: il messaggio è semplicemente marcato come fallito nella metrica `delivering_errors`. Questo riflette la natura *best-effort* di *UDP*: non vi è garanzia di consegna, ritrasmissione automatica, o conferma di ricezione. È responsabilità dello *stage* o dell'applicazione a monte gestire affidabilità se richiesta (ad esempio, tramite *ACK* a livello applicativo o ricezione esplicita).

Come altri *Egress/Processor stage*, l'*UDP stage* supporta sia *Single Worker Mode* che *Worker Pool Mode*. In modalità *pool*, il *framework* gestisce automaticamente lo *scaling* dei *worker* in base al carico, usando la stessa connessione *UDP* condivisa tra tutti i *worker*. Questo consente di sfruttare il parallelismo *multicore* per aumentare il *throughput* di trasmissione, purché il kernel e l'hardware di rete lo supportino.

In sintesi, l'*UDP Egress stage* incarna il paradigma di esportazione minimalista e ad alte prestazioni: prende un flusso di messaggi elaborati e li spedisce direttamente a una destinazione esterna usando il protocollo *UDP*, senza *buffering* persistente, rielaborazione, o garanzie di consegna. È ideale per scenari dove la velocità è prioritaria rispetto all'affidabilità, come telemetria *real-time*, *streaming* di dati, o distribuzione di messaggi a sistemi esterni a bassa latenza.

3.7.2 TCP Egress

Lo *TCP Egress stage* è un *Egress stage* specializzato nell'invio di messaggi verso un endpoint remoto tramite una connessione *TCP* persistente. A differenza dello *UDP Egress stage*, che invia datagrammi isolati senza stato, il *TCP Egress stage* mantiene una connessione bidirezionale affidabile con il destinatario, garantendo consegna ordinata e completa di tutti i dati.

La configurazione dello *TCP Egress stage* è simile a quella dello *UDP Egress stage*, ma con un parametro aggiuntivo: `WriteTimeout`, che specifica il tempo massimo di attesa per una singola operazione di scrittura (*default* 10 secondi). Questo timeout protegge da situazioni di deadlock dove la connessione è ancora aperta ma il destinatario non legge dati (ad esempio, a causa di crash o sovraccarico).

Contrariamente a *UDP (best-effort)*, *TCP* garantisce che ogni byte trasmesso arriverà al destinatario nell'ordine esatto, oppure che un errore sia riportato. Se la connessione si interrompe o il destinatario non legge i dati entro il timeout configurato, il *worker* registra un errore e la metrica `delivering_errors` è incrementata.

Il *worker* dello *stage* estrae il payload serializzato dal messaggio in input e lo trasmette tramite `conn.Write` [53] come sequenza continua di byte, senza delimitatori impliciti: se il messaggio necessita di delimitazione (ad esempio, *newline*), deve essere già incluso nel payload serializzato dallo *stage* precedente.

Il *TCP Egress stage* supporta solamente la *Single Worker Mode*, in quanto, sebbene il metodo `conn.Write` possa essere chiamato da goroutine differenti in maniera sicura, il contenuto dei dati trasmessi potrebbe intervallarsi. Per esempio, se una goroutine scrivesse “hello” e un'altra “word”, il dato trasmesso potrebbe essere una sequenza del tipo “heworldllo”.

Lo *TCP Egress stage* è ideale per scenari dove l'affidabilità è critica: trasmissione di comandi verso sistemi embedded, esportazione di dati strutturati verso data warehouse, o comunicazione con sistemi remoti che richiedono ricezione garantita.

In sintesi, il *TCP Egress stage* rappresenta l'alternativa affidabile allo *UDP Egress stage*: conclude la *pipeline* trasmettendo messaggi elaborati tramite una connessione *TCP* persistente, garantendo consegna ordinata e completa, con timeout configurabile per proteggere da blocchi indefiniti.

3.7.3 Kafka Egress Stage

Lo *Kafka Egress stage* è uno *Egress stage* specializzato nell'invio di messaggi verso un broker *Apache Kafka*. Lo stage conclude la pipeline inviando messaggi a topic *Kafka* con semantiche di consegna affidabili e configurabili, fungendo da punto di uscita per il flusso di dati verso un sistema *event-driven* distribuito.

La configurazione dello *Kafka Egress stage* riprende i campi definiti dalla libreria *kafka-go* nella sua struttura `kafka.Writer` [54], in maniera speculare a come viene fatto nell'omonimo stage di *Ingress*. Il parametro fondamentale resta quello del campo `Brokers`, che definisce gli endpoint a cui connettersi. Contrariamente allo stage di *Ingress*, i topic sono definiti all'interno del messaggio, permettendo di scegliere a *runtime* dove indirizzare il messaggio.

La logica dello stage è minimale, in quanto viene semplicemente preso il messaggio in ingresso, viene creata la struttura usata per contenere le informazioni riguardanti gli header, la chiave, il `value` (payload) e il topic in cui il messaggio deve essere

inoltrato. Infine, viene richiamata la funzione `WriteMessages` [55] che si occuperà di passare il dato al broker *Kafka*.

Come altri *Egress stage*, il *Kafka Egress stage* supporta sia *Single Worker Mode* che *Worker Pool Mode*, poiché il writer è *thread-safe* e gestisce internamente la serializzazione e il batching dei messaggi.

Un aspetto distintivo del *Kafka Egress stage* è l'integrazione nativa del tracciamento distribuito. Ogni messaggio inviato include nel suo header *Kafka* il contesto di trace (trace ID, span ID, baggage) estratto dal messaggio in transito, consentendo la propagazione del contesto di esecuzione end-to-end attraverso il cluster *Kafka* verso sistemi consumer a valle, abilitando così la correlazione completa dei flussi di dati in architetture microservizi.

3.7.4 File Egress Stage

Lo *File Egress stage* rappresenta uno stage di esportazione progettato per persistere messaggi elaborati dalla pipeline in un file su disco, funzionando come punto di uscita persistente per flussi di dati che richiedono archiviazione locale o condivisione tramite *filesystem*. Esso fornisce un meccanismo semplice e deterministico per scrivere sequenzialmente messaggi su *filesystem* locale (append only), ideale per logging, tracing, analisi offline, e backup di dati elaborati.

La libreria **Goccia** fornisce una configurazione minimalista per il *File Egress stage*, che mira a controllare i comportamenti di buffering e flushing dei dati verso il disco. Il parametro principale è `Path`, ovvero il percorso del file di destinazione, e la dimensione del buffer in cui accumulare i byte prima della scrittura su disco (default 4096 byte). Inoltre, vi sono altri due parametri per regolare il flushing, utili per forzare una scrittura del file a determinate condizioni. Quest'ultime riguardano la percentuale di riempimento del buffer e una deadline temporale (default: 1 secondo). La configurazione del buffering consente di controllare il trade-off tra latenza e throughput: buffer di piccole dimensioni e threshold basse producono flush frequenti, incrementando il numero di operazioni I/O su disco; buffer più grandi e threshold elevate riducono il numero di operazioni I/O ma introducono latenza maggiore nella persistenza.

Lo stage adotta un'architettura *Single Worker Mode*, in contrasto con altri stage che supportano *worker pool*. Questa scelta è dovuta alla natura sequenziale e seriale della scrittura su file: accedere contemporaneamente allo stesso file descriptor tramite molteplici goroutine richiederebbe sincronizzazione aggiuntiva e potrebbe compromettere l'ordine di scrittura dei messaggi. Mantenere un singolo worker garantisce che i messaggi siano scritti nell'ordine esatto di ricezione dalla pipeline, preservando la causalità del flusso dati. Nel caso in cui si voglia scrivere più di un file, è richiesto l'uso di N stage di questo tipo, quanti sono i file.

Il messaggio in ingresso allo stage è generico, ma deve implementare l'interfaccia `message.Serializable`, come molti altri stage della libreria. Per ciascun messaggio ricevuto, viene estratto il payload del messaggio, che poi è scritto nel `bufio.Writer` [56]. Questa operazione non scrive immediatamente su disco, ma accumula i byte nel buffer in memoria. Il numero di byte scritti è registrato per tracciamento. Successivamente, viene valutata la soglia di flush: se il numero di byte accumulati supera la soglia configurata, il metodo di flush viene invocato per trasmettere immediatamente i dati dal buffer al kernel. Parallelamente al meccanismo di flush basato su soglia, una goroutine ausiliaria rimane in ascolto su un ticker, e ogni volta che l'intervallo di deadline configurata scade, invoca il flush.

Lo stage *File Egress* espone tre metriche per il monitoraggio: la prima conta il numero dei byte scritti, la seconda il numero di errori durante l'operazione di scrittura nel buffer, mentre l'ultima conta gli errori di flushing.

Il *File Egress stage* è ideale per scenari in cui la persistenza locale è sufficiente e la semplicità è prioritaria rispetto alla distribuzione. Un esempio pratico è il logging.

In sintesi, il *File Egress stage* incarna il paradigma di persistenza locale e sequenziale: prende un flusso di messaggi elaborati dalla pipeline e li scrive ordinatamente in un file su disco, offrendo controllo granulare sul buffering tramite soglie dinamiche e flushing periodico, preservazione rigorosa dell'ordine causale, e sincronizzazione robusta verso il *filesystem*. È il componente ideale per conclusioni di pipeline che richiedono archiviazione durabile, facilità di analisi offline, e semplicità operativa.

3.7.5 QuestDB Egress Stage

Lo *QuestDB Egress stage* rappresenta uno stage di esportazione progettato per persistere messaggi elaborati verso *QuestDB* [57], un database time-series colonnare ad alte prestazioni.

La configurazione dello stage presenta un unico campo specifico, ovvero l'indirizzo di riferimento del database. La semplicità della configurazione rispecchia l'approccio di *QuestDB*: la libreria client *go-questdb-client* [58] gestisce internamente aspetti quali il pooling delle connessioni, il flushing automatico dei messaggi, e la ricongiunzione in caso di fallimento.

Lo stage riceve messaggi del tipo `QuestDBMessage`, che incapsula una collezione di righe (rows) da inserire. Ciascuna riga è associata a una tabella specifica e contiene simboli (symbol columns) e colonne (value columns). I simboli rappresentano colonne di tipo categorico in *QuestDB*, ottimizzate per l'indicizzazione e la deduplicazione: ogni simbolo deve essere inserito prima di qualunque altra colonna della riga. Le colonne rappresentano dati di tipo vario: booleano, intero, float, stringa, timestamp, e long integer (*big.Int*).

Lo stage, per ciascun messaggio ricevuto, elabora tutte le righe contenute tramite un iteratore. Per ogni riga, seleziona la giusta tabella, poi inserisce tutti i simboli prima di inserire i valori delle colonne. Una volta caricati tutti i valori delle colonne, viene impostato il timestamp della riga.

Lo stage supporta *Worker Pool Mode* (oltre a *Single Worker Mode*). In modalità pool, molteplici worker condividono lo stesso `LineSenderPool` [59], dal quale ciascun worker estrae un sender indipendente. Poiché il pool garantisce *thread-safety*, il parallelismo multicore è sfruttato per incrementare il throughput di inserimento verso *QuestDB*. Il pool gestisce automaticamente il flushing dei buffer quando la soglia di autoflush è raggiunta, coordinando gli inserimenti provenienti da molteplici worker in modo trasparente. Se il pool rileva fallimenti di connessione, applica retry automatici con timeout configurato, garantendo resilienza a livello di trasporto.

Lo stage espone una singola metrica di monitoraggio che tiene traccia del numero di righe inserite.

Il *QuestDB Egress stage* è ideale per scenari dove la persistenza strutturata in un database time-series è prioritaria. Un esempio classico di tale utilizzo è l'implementazione di un sistema di telemetria *real-time*, per inviare metriche, tracce e log strutturati verso *QuestDB* per analisi e visualizzazione su dashboard. Questo è anche il principale caso d'utilizzo che ha portato alla creazione dello stesso progetto **Goccia**.

In sintesi, il *QuestDB Egress stage* incarna il paradigma di persistenza strutturata e ad alte prestazioni: prende un flusso di messaggi elaborati e li inserisce in una banca dati time-series, offrendo tipizzazione di dati, pooling di connessioni, flushing automatico, e supporto a molteplici tabelle e simboli categorici. È il componente ideale per conclusioni di pipeline che richiedono persistenza analitica, query *real-time*, e integrazione con ecosistemi di *business intelligence* e monitoraggio.

3.7.6 Sink Egress Stage

Lo *Sink Egress stage* rappresenta il caso limite di uno stage di esportazione: non persiste, non trasmette, non scrive alcun dato verso destinazioni esterne. Invece, consuma semplicemente tutti i messaggi in ingresso, li distrugge, e prosegue. È concepito esclusivamente per scopi di testing e benchmarking, permettendo di valutare le prestazioni della pipeline senza l'overhead di I/O verso *filesystem*, rete, o database.

Capitolo 4

Benchmarks della libreria Goccia

4.1 Micro-benchmarking: Analisi dei Connettori Interni

L'architettura della libreria **Goccia** fonda la propria efficienza sulla minimizzazione della latenza nel trasferimento dei messaggi tra gli stadi della pipeline. Poiché il tempo di elaborazione della logica di business all'interno di uno stage ($T_{process}$) è indipendente dall'infrastruttura di trasporto, l'ottimizzazione delle prestazioni complessive dipende strettamente dalla riduzione del tempo di attraversamento del connettore ($T_{transfer}$). In questa sezione viene presentata un'analisi quantitativa volta a dimostrare il vantaggio prestazionale dei *ring buffer* lock-free implementati rispetto alle primitive native del linguaggio Go (channel), a parità di semantica di comunicazione.

4.1.1 Metodologia e setup sperimentale

Il benchmark è stato progettato per isolare il costo computazionale delle operazioni di scrittura (**Write**) e lettura (**Read**) nel connettore, escludendo qualsiasi logica di elaborazione del payload per evitare rumore nelle misurazioni. Il confronto avviene tra due implementazioni:

- **Baseline:** Un wrapper attorno a un `chan int` bufferizzato standard del linguaggio Go, che utilizza mutex interni gestiti dal runtime per la sincronizzazione delle goroutine.
- **RingBuffer:** L'implementazione della libreria **Goccia** ottimizzata nelle varianti **SPSC** (*Single Producer Single Consumer*), **SPMC** (*Single Producer Multiple Consumer*) e **MPSC** (*Multiple Producer Single Consumer*), caratterizzata dall'uso di operazioni atomiche e padding delle cache-line per evitare il *false sharing*.

Qui sotto viene riportata l'implementazione della Baseline:

```
1 type baselineRingBuffer struct {
2     ch chan int
3 }
4
5 func (b *baselineRingBuffer) Write(val int) error {
6     b.ch <- val
7     return nil
8 }
9
10 func (b *baselineRingBuffer) Read(ctx context.Context)
    (int, error) {
```

```
11     select {
12     case <-ctx.Done():
13         return 0, ctx.Err()
14     case val := <-b.ch:
15         return val, nil
16     }
17 }
```

Listing 4.1: Implementazione della Baseline basata su canali

I test sono stati eseguiti su tre ambienti hardware distinti per valutare la consistenza dei risultati su diverse microarchitetture e set di istruzioni:

- **Server CI:** AMD EPYC 7763 64-Core (x86_64).
- **Workstation:** Intel Core i7-6700 @ 3.40GHz (x86_64).
- **Laptop:** Apple M1 (ARM64).

La valutazione si articola in due scenari di carico:

- **Steady State:** Misura il throughput in condizioni di regime, dove le operazioni di scrittura e lettura avvengono in un ciclo continuo bilanciato senza interruzioni forzate.
- **Contention:** Valuta il comportamento sotto stress concorrente, variando il numero di produttori o consumatori (da 1 a 16 goroutine) per sollecitare i meccanismi di sincronizzazione e lo scheduler del sistema operativo.

Il codice sorgente completo è disponibile nel repository del progetto sotto il package `internal/rb` [60]. Per replicare i benchmark e verificare i risultati presentati, è possibile utilizzare il toolchain standard del Go lanciando il seguente comando dalla directory di root del progetto:

```
1 go test -bench=. -benchmem ./internal/rb
```

Listing 4.2: Comando per l'esecuzione dei benchmark

4.1.2 Analisi in regime stazionario

I risultati ottenuti nello scenario *Steady State* dimostrano una netta superiorità dei ring buffer rispetto ai canali Go in tutte le configurazioni hardware testate. La tabella sottostante riassume i tempi di latenza media per operazione (ns/op) e il relativo *speedup* ottenuto.

L'analisi dei dati evidenzia che l'implementazione custom riduce drasticamente l'overhead di sincronizzazione. Il guadagno è particolarmente marcato su architettura server (AMD EPYC), dove lo speedup raggiunge il fattore **4.75x**. Ciò è

Architettura	Variante	Baseline (ns/op)	RingBuffer (ns/op)	Speedup
AMD EPYC	SPSC	82.47	17.38	4.75x
	SPMC	82.47	20.78	3.97x
	MPSC	82.47	20.92	3.94x
Intel i7	SPSC	105.60	38.61	2.74x
	SPMC	105.60	49.63	2.13x
	MPSC	105.60	49.26	2.14x
Apple M1	SPSC	56.30	35.05	1.61x
	SPMC	56.30	37.39	1.51x
	MPSC	56.30	38.27	1.47x

Tabella 4.1: Confronto latenza e speedup in regime stazionario

attribuibile alla gestione ottimizzata della coerenza della cache nei ring buffer: in sistemi *multi-core*, il costo dei lock dei canali standard aumenta significativamente, mentre le operazioni atomiche *lock-free* scalano con maggiore efficienza. Su architettura ARM64 (Apple M1), pur mantenendo un vantaggio netto ($\sim 1.5x$), il divario si riduce, suggerendo un'implementazione dei canali nativi particolarmente ottimizzata per il modello di memoria rilassato di ARM o un costo relativo delle primitive di sincronizzazione inferiore.

4.1.3 Scalabilità in scenari di contesa

Mentre i test a regime stazionario dimostrano l'efficienza di base del *data-path*, è nello scenario di *Contention* che si verifica la robustezza dell'architettura in condizioni critiche. In un sistema reale come la contesa si manifesta in due pattern principali: **Fan-Out** (SPMC), dove un singolo stadio distribuisce pacchetti a un pool di worker, e **Fan-In** (MPSC), dove molteplici sorgenti convergono verso un unico buffer.

Per valutare il comportamento limite, sono stati confrontati i ring buffer contro i canali Go configurando il massimo livello di parallelismo testato (fino a 16 goroutine concorrenti). La tabella seguente riporta i risultati per le configurazioni più significative.

L'analisi dei dati di contesa evidenzia tre dinamiche fondamentali:

1. **Resilienza al "Thundering Herd":** Nello scenario **SPMC** (Fan-Out) su architettura Intel i7, si osserva lo speedup più elevato in assoluto (**3.00x**). I canali Go, basati su mutex, soffrono il risveglio simultaneo di molti consumatori per un singolo dato. Il ring buffer SPMC, utilizzando operazioni atomiche **CAS** (*Compare-And-Swap*), risolve la contesa a livello hardware senza richiedere un intervento oneroso dello scheduler del sistema operativo.

Architettura	Pattern	Config.	Baseline (ns)	RB (ns)	Speedup
AMD EPYC	SPSC	P1-C1	118.20	44.41	2.66x
	SPMC	P1-C16	149.40	63.66	2.35x
	MPSC	P16-C1	146.40	62.91	2.33x
Intel i7	SPSC	P1-C1	193.60	71.47	2.71x
	SPMC	P1-C16	275.50	91.70	3.00x
	MPSC	P16-C1	274.40	106.50	2.58x
Apple M1	SPSC	P1-C1	99.07	62.72	1.58x
	SPMC	P1-C16	92.41	64.89	1.42x
	MPSC	P16-C1	207.50	76.62	2.71x

Tabella 4.2: Confronto latenza e speedup in scenario di contesa

2. **Superiorità nel Fan-In su ARM64:** Un risultato notevole emerge dall'architettura **Apple M1** nello scenario **MPSC**, dove lo speedup tocca il **2.71x**. In questo contesto, il meccanismo di locking dei canali Go introduce un overhead non lineare, mentre l'approccio ottimistico dei ring buffer scala efficacemente sfruttando l'architettura *load/store* dei processori Apple Silicon.
3. **Stabilità della Latenza:** Confrontando la Baseline a bassa contesa (P1-C1) con quella ad alta contesa (P1-C16), si nota come i canali Go tendano a degradare le prestazioni. Sebbene anche i ring buffer subiscano un incremento di latenza, il valore assoluto rimane in un ordine di grandezza (< 100 ns) tale da garantire un throughput sostenuto superiore ai 10 milioni di messaggi al secondo, ampiamente sufficiente per i requisiti automotive.

4.1.4 Discussione architetturale

La superiorità prestazionale dei ring buffer di **Goccia** rispetto ai canali Go valida le scelte progettuali descritte nel Capitolo 4. Due fattori tecnici principali giustificano questi risultati:

- **Assenza di Lock (Lock-freedom):** L'utilizzo di operazioni atomiche per l'aggiornamento degli indici di testa e coda elimina la necessità di sospendere le goroutine tramite mutex, riducendo drasticamente il costo del *context switch*.
- **Mitigazione del False Sharing:** L'implementazione rigorosa del padding delle cache-line (`cpu.CacheLinePad`) tra gli indici di lettura e scrittura impedisce l'invalidazione non necessaria delle linee di cache L1/L2 quando produttore e consumatore operano su core fisici diversi. I canali standard del Go, dovendo rimanere primitive generiche, non possono applicare ottimizzazioni di memoria così aggressive e specifiche.

In conclusione, l'adozione di strutture dati specializzate permette al layer di trasporto della libreria di non divenire il collo di bottiglia del sistema, liberando cicli CPU per la logica di analisi dei dati.

Capitolo 5

Implementazione del sistema di telemetria automobilistica di SquadraCorse

Il sistema di telemetria in tempo reale per il prototipo di Formula SAE di SquadraCorse Polito è stato progettato secondo un'architettura a più livelli che separa chiaramente le responsabilità tra acquisizione, elaborazione, persistenza, osservabilità e visualizzazione dei dati.

L'autovettura, collegata a Internet tramite un gateway 5G, espone due linee CAN (Controller Area Network); i messaggi trasmessi sui due bus vengono incapsulati secondo il protocollo Cannelloni, poi in datagrammi UDP, e inviati tramite una VPN Tailscale verso un server centrale su cui è eseguito un insieme di servizi orchestrati tramite Docker.

Dal punto di vista logico, l'architettura è suddivisa in cinque layer principali:

Livello di acquisizione dati (*Ingestion Layer*) Responsabile della ricezione sicura dei datagrammi UDP provenienti dall'auto e del loro instradamento verso i servizi interni.

Livello di elaborazione (*Processing Layer*) Implementato tramite la libreria **Goccia**, si occupa di decodificare i frame Cannelloni, interpretare i messaggi CAN e trasformarli in segnali ad alto livello.

Livello di persistenza (*Storage Layer*) Gestisce la memorizzazione dei dati di telemetria in un database time-series ad alte prestazioni.

Livello di osservabilità (*Observability Layer*) Raccoglie, struttura e correla metriche e tracce relativi al comportamento dell'intero sistema di telemetria, fornendo una base unificata per il monitoraggio tecnico.

Livello di visualizzazione (*Visualization Layer*) Espone in modo sicuro agli utenti (tramite dashboard e interfacce web) sia i dati di telemetria persistiti sia i segnali di osservabilità, fungendo da punto di accesso unificato per l'analisi in tempo reale e post.

<disegno architettura generale>

5.1 Ingestion Layer

Il *Livello di acquisizione dati* comprende tutti i componenti responsabili di ricevere i messaggi provenienti dall'auto, tramite la rete 5G e la VPN Tailscale, e di recapitarli ai container di telemetria basati su **Goccia**.

Connettività Tailscale

Il container `tailscale` stabilisce una rete privata virtuale basata su WireGuard fra il gateway 5G a bordo vettura e il server di telemetria. Una volta stabilita la VPN, i datagrammi UDP generati dal gateway vengono incapsulati nel tunnel Tailscale e consegnati all'istanza `tailscale` in esecuzione sul server, la quale li rende disponibili agli altri servizi Docker.

Il reale mittente dei pacchetti è una PCB custom sviluppata internamente al team [61] che si occupa dell'effettiva lettura dei due bus CAN ed effettua l'incapsulamento secondo il protocollo Cannelloni. Questa scheda è programmata, lato firmware, per inviare i datagrammi UDP verso un unico indirizzo IP, ma su due porte diverse, una per ciascuna linea CAN.

Per poter trasmettere i dati fino al server in sicurezza, il gateway (basato su OpenWRT) configura innanzitutto un alias sull'interfaccia LAN (`br-lan`), in modo da *simulare un host* con indirizzo IP corrispondente alla destinazione attesa dalla PCB. L'alias viene creato come interfaccia `lan_alias` di tipo statico, associata a `br-lan` e configurata con l'indirizzo, ad esempio, `192.168.10.254/32`.

Successivamente, il gateway abilita l'inoltro IPv4 e definisce delle regole di *DNAT* nella tabella di `PREROUTING` che riscrive la destinazione di tutto il traffico proveniente da `br-lan` e indirizzato a `192.168.10.254` verso l'indirizzo IP del container Tailscale del server.

In questo modo, la PCB continua a inviare i pacchetti verso un indirizzo IP locale fisso (l'alias sulla LAN), mentre il gateway, in maniera trasparente, li inoltra attraverso la VPN Tailscale verso il server di telemetria. Il firmware della scheda resta così indipendente dai dettagli di configurazione della VPN e dell'infrastruttura remota.

Instradamento dei datagram

Il servizio `udp-proxy` è il punto di demarcazione tra la VPN e il resto dello stack di telemetria. Esso condivide il medesimo namespace di rete del container Tailscale (tramite `network_mode: service:tailscale`) e riceve quindi direttamente i datagrammi UDP provenienti dalla vettura.

La sua funzione è quella di *instradare* i pacchetti UDP verso i corretti consumer a valle, in base alla porta di destinazione. Per ciascuna porta di ascolto configurata

viene creata un'istanza del proxy, che apre una socket UDP in ascolto e apre una connessione verso l'endpoint che si occuperà del processamento.

Il cuore del componente è implementato utilizzando il server UDP offerto dalla libreria standard di Go.

5.2 Processing Layer

Il *Livello di elaborazione* è implementato dal servizio denominato **sc-telemetry** [62], eseguito in due istanze distinte per i due bus CAN. Ciascuna istanza realizza, tramite la libreria **Goccia**, una pipeline di elaborazione a sei stadi che parte dai datagrammi UDP incapsulati in Cannelloni e termina con la persistenza dei segnali CAN in QuestDB.

L'immagine **scomarferro/sc-telemetry** [63] incapsula in un unico binario Go l'implementazione necessaria per gestire il flusso di dati proveniente dal veicolo. In fase di avvio, il servizio carica la configurazione leggendo un file YAML (di default `/app/config/config.yaml`) e applica eventuali override tramite variabili d'ambiente. Il percorso del file di configurazione può essere modificato impostando la variabile `CONFIG_PATH`, mentre ogni campo del file YAML può essere sovrascritto da un corrispondente *environment variable*. Questa strategia consente di mantenere una configurazione di base versionata nel repository e, al contempo, di adattare rapidamente i parametri di runtime al contesto specifico senza ricompilare il binario.

La pipeline implementata da **sc-telemetry** è composta da sei stadi principali:

- **UDP Ingress**: riceve i datagrammi UDP provenienti dalla scheda SScanner tramite il proxy UDP e li inserisce nella pipeline.
- **Cannelloni Decoder Processor**: prende in ingresso i payload dei pacchetti UDP e decodifica il payload secondo la specifica del protocollo Cannelloni.
- **ROB Processor (Re-Order Buffer)**: ordina i messaggi dello stage precedente in base al numero di sequenza, compensando il fatto che i datagrammi UDP possono arrivare fuori ordine o con *jitter* significativo. Oltre al riordino, lo stadio corregge i timestamp associati ai messaggi applicando tecniche di smoothing temporale descritte nei capitoli precedenti, in modo da stimare con maggiore precisione l'istante effettivo di generazione del messaggio a bordo vettura.
- **CAN Processor**: prende in ingresso i messaggi Cannelloni riordinati, estrae i frame CAN grezzi e li decodifica secondo le specifiche contenute nel file DBC [64]. Questo tipo di file rappresenta lo standard del settore automobilistico per definire la struttura dei messaggi trasmessi nelle linee CAN. Il file DBC viene letto, in fase di inizializzazione, dalla libreria **acmelib** [49].
- **CAN Message Handler (Custom Processor)**: è l'unico stadio della pipeline implementato specificamente per questa applicazione, e ha il compito di trasformare i messaggi CAN decodificati in messaggi pronti per essere inseriti in QuestDB.
- **QuestDB Egress**: si occupa di inserire i messaggi nel database time-series.

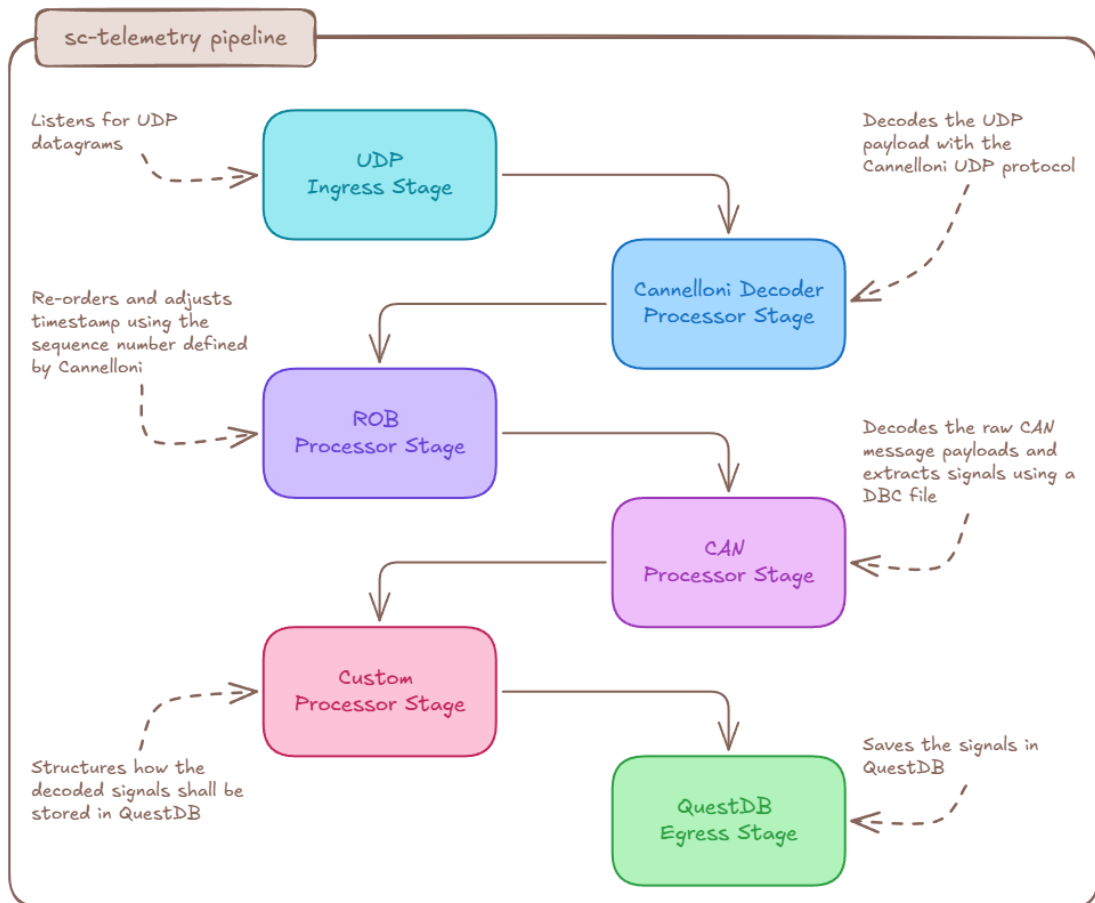


Figura 5.1: Stage di Goccia usati in sc-telemetry

5.3 Storage Layer

Il *Livello di persistenza* ha il compito di memorizzare in modo durevole e strutturato i dati di telemetria prodotti dalle pipeline, rendendoli disponibili sia per la visualizzazione quasi real-time sia per analisi differite.

QuestDB

QuestDB è stato scelto come backend di persistenza poiché è un database time-series open source progettato specificamente per carichi ad alto throughput su dati indicizzati temporalmente. A differenza dei database relazionali tradizionali, adotta uno storage column-oriented e un modello dati nativamente temporale: il timestamp è un tipo primitivo e può fungere da chiave di partizionamento, consentendo l'organizzazione fisica delle tabelle per intervalli temporali (ad esempio per giorno). Questo approccio, combinato con tecniche di compressione e partizionamento, permette di gestire l'ingestion di milioni di record al secondo e query analitiche su grandi volumi di dati mantenendo latenze contenute.

L'interfaccia di accesso si basa su SQL esteso con primitive specifiche per le serie temporali, come **SAMPLE BY** per il downsampling, **LATEST ON** per il recupero dello stato più recente, che risultano particolarmente adatti alle analisi di dati telemetrici. Inoltre, il supporto ai protocolli standard come PGwire (compatibile PostgreSQL) e alle API HTTP/REST facilita l'integrazione con strumenti esterni e con la piattaforma Grafana. Nel contesto di questa architettura, QuestDB agisce come single source of truth per i dati di telemetria, mentre i dati di osservabilità relativi al funzionamento del sistema (metriche e trace) sono affidati ai backend specializzati Prometheus e Tempo.

Data Model

Nel modello dati adottato per QuestDB, i segnali CAN decodificati vengono suddivisi in più tabelle in base al *tipo di valore* associato al segnale. In particolare, esistono quattro tabelle principali: **flag_signals** per i segnali booleani, **int_signals** per i segnali interi, **float_signals** per i segnali in virgola mobile ed **enum_signals** per i segnali enumerativi. Questa scelta consente di mantenere omogenee le colonne di ciascuna tabella e di semplificare sia l'ingestione sia le query analitiche successive.

Tutte le tabelle condividono un insieme di colonne comuni: un campo simbolico **name**, che identifica il segnale (ad esempio **motor_rpm** o **coolant_temp**), il campo intero **can_id**, che memorizza l'ID del frame CAN da cui il segnale è stato estratto, e il campo intero **raw_value**, che conserva il valore grezzo così come presente nel payload del frame. A queste colonne comuni si affiancano poi colonne specifiche per ciascun tipo: nella tabella **flag_signals** è presente la colonna booleana **flag_value**; in **int_signals** la colonna intera **integer_value**; in **float_signals**

la colonna a virgola mobile `float_value`. Per i segnali enumerativi, memorizzati in `enum_signals`, oltre a `can_id` e `raw_value` viene utilizzato anche un simbolo `enum_value`, che rappresenta la label dell'enumerazione corrispondente al valore numerico sottostante.

```
1 CREATE TABLE float_signals (  
2     timestamp TIMESTAMP,  
3     name SYMBOL,  
4     can_id LONG,  
5     raw_value LONG,  
6     /*  
7     integer_value LONG for int_signals table  
8     flag_value BOOLEAN for flag_signals table  
9     enum_value SYMBOL for enum_signals table  
10    */  
11     float_value DOUBLE  
12 ) TIMESTAMP(timestamp) PARTITION BY DAY;
```

Listing 5.1: Definizione tabella dei segnali in QuestDB

In questo modo, l'organizzazione dei dati in QuestDB risulta sia normalizzata rispetto al tipo del segnale, sia sufficientemente ricca da consentire analisi flessibili: le colonne `name` ed eventuali simboli aggiuntivi (`enum_value`) permettono di filtrare e aggregare per segnali logici di alto livello, mentre `can_id` e `raw_value` preservano il legame con la rappresentazione originaria sul bus CAN.

Throughput, partizionamento e retention

Dal punto di vista prestazionale, QuestDB è dimensionato per gestire senza difficoltà il volume di dati generato dai due bus CAN del prototipo SquadraCorse. La documentazione ufficiale e benchmark indipendenti riportano capacità di ingest nell'ordine dei milioni di record al secondo su hardware moderno, valori ben superiori al carico previsto in questa applicazione. [17] [65] [66]

La strategia di partizionamento temporale adottata (tipicamente per giorno) consente di implementare in modo semplice politiche di retention basate su orizzonti temporali, eliminando intere partizioni in un'unica operazione, e al tempo stesso riduce la quantità di dati che devono essere scansionati dalle query, che nella pratica si concentrano spesso su singole sessioni di test o giornate di prove. Ne risulta una migliore località dei dati su disco, con effetti positivi sulle latenze di interrogazione.

Infine, la presenza di QuestDB come layer esclusivamente *di storage* per i dati di telemetria – separato dallo storage di metriche (Prometheus) e trace (Tempo) – rende l'architettura più modulare: è possibile, ad esempio, estendere in futuro il sistema con ulteriori database (per analisi offline o machine learning) senza

impattare sul percorso critico di acquisizione ed elaborazione dei segnali provenienti dalla vettura.

5.4 Observability Layer

Il *Livello di osservabilità* raccoglie e struttura i segnali relativi al comportamento del sistema (metriche e trace), indipendentemente da come verranno successivamente visualizzati. In questa architettura è realizzato da tre componenti principali: OpenTelemetry Collector [19], Prometheus [67], Tempo [68] e Grafana.

OpenTelemetry Collector

Questo è il componente general-purpose per la ricezione, il processamento e l'esportazione di segnali di osservabilità provenienti da più sorgenti.

Nel sistema di telemetria di SquadraCorse, il Collector riceve tramite protocollo OTLP/gRPC i trace e le metriche prodotti dalle istanze di **sc-telemetry**, strumentate tramite **Goccia**; su questi dati applica quindi una catena di *processor* (ad esempio un **batch** per raggruppare gli eventi, un **memory_limiter** per controllare il consumo di memoria ed eventualmente processor di *sampling* per ridurre il volume dei trace) e, infine, esporta le metriche verso Prometheus ed i trace verso Grafana Tempo.

Prometheus

Prometheus viene utilizzato come database di serie temporali per le metriche del sistema di telemetria. Raccoglie principalmente le metriche esposte dall'OpenTelemetry Collector, che fornisce una vista aggregata sul comportamento dei vari servizi, con la possibilità di integrare in modo incrementale ulteriori sorgenti (ad esempio metriche esposte direttamente da altri componenti).

Adotta un modello dati dimensionale, in cui ogni serie temporale è identificata da un nome di metrica e da un insieme di label; ciò consente di distinguere, ad esempio, le metriche per servizio, istanza o stage della pipeline e di formulare query che filtrano o aggregano i dati (throughput per stage, latenza p99, numero di errori, ecc.) con grande flessibilità.

Grafana Tempo

Per il *distributed tracing* viene impiegato **Grafana Tempo**, un backend progettato per memorizzare e interrogare trace di applicazioni distribuite a partire da segnali OpenTelemetry, Jaeger o Zipkin.

Nel sistema proposto, Tempo riceve i trace esportati dall'OpenTelemetry Collector. Ogni messaggio di telemetria elaborato da **sc-telemetry** genera una root span associata alla ricezione del datagramma UDP, mentre ciascuno degli stage della pipeline **Goccia** (UDP ingress, decoder Cannelloni, ROB, decoder CAN, handler, egress QuestDB) contribuisce con uno span figlio. Ciò permette di ricostruire il

percorso completo di elaborazione di un messaggio end-to-end, misurare la latenza introdotta da ciascun stage e individuare colli di bottiglia o anomalie.

Grafana Tempo è particolarmente adatto a questo scenario perché evita l'uso di database di ricerca generici (come Elasticsearch) e utilizza storage ottimizzati per trace compressi, riducendo i costi operativi e mantenendo al contempo capacità di interrogazione avanzate tramite TraceQL. Inoltre, esso è nativamente integrato in Grafana, semplificando la configurazione di quest'ultimo.

5.5 Visualization Layer

Il *Livello di visualizzazione* ha il compito di rendere disponibili, in modo sicuro e fruibile, i dati di telemetria e i segnali di osservabilità agli utenti umani (ingegneri di pista, sviluppatori, data analyst). In questa architettura è realizzato principalmente da **Grafana** [69], come interfaccia di consultazione unificata, e da **Caddy** [70], come reverse proxy e terminazione TLS verso l'esterno.

Grafana

Grafana costituisce il frontend di riferimento per la visualizzazione dei dati prodotti dagli altri layer del sistema. Tramite opportuni data source, interroga QuestDB per accedere ai segnali CAN decodificati e memorizzati nel Livello di persistenza, Prometheus per le metriche di sistema (throughput, latenze, errori, stato dei servizi) e Grafana Tempo per l'esplorazione dei trace distribuiti. Le dashboard realizzate a supporto dell'attività di SquadraCorse combinano questi sorgenti informativi per offrire, in un'unica interfaccia, una vista completa sia dello stato della vettura sia del comportamento del sistema di telemetria.

La definizione delle dashboard non avviene manualmente tramite l'interfaccia grafica, ma segue un approccio *infrastructure as code*: le configurazioni sono descritte in Jsonnet [71] utilizzando la libreria Grafonnet [72], che fornisce primitive ad alto livello per la costruzione di pannelli, row e datasource. I file Jsonnet vengono quindi compilati in manifest JSON standard di Grafana e inclusi nel processo di provisioning, così da poter versionare non solo il codice applicativo, ma anche la struttura e il contenuto dei dashboard. Questo approccio semplifica la manutenzione e l'evoluzione delle viste, consente di riutilizzare componenti comuni tra dashboard diversi e garantisce la riproducibilità dell'ambiente di visualizzazione su installazioni differenti.

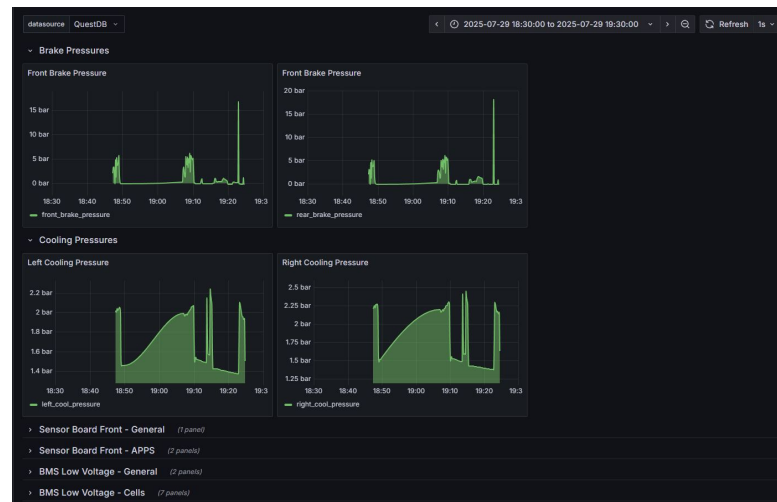


Figura 5.2: Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (pressioni)

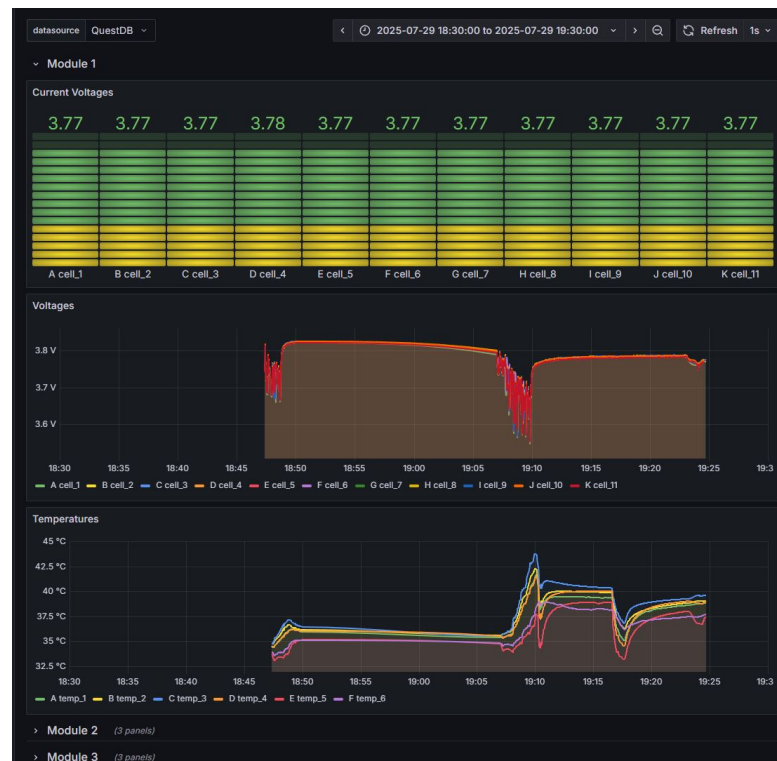


Figura 5.3: Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (tensioni/temperature)

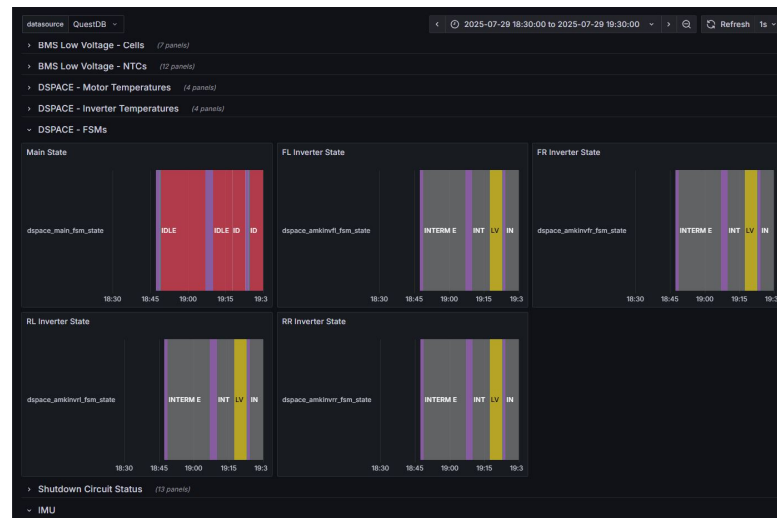


Figura 5.4: Dashboard Grafana tratte dal sistema di telemetria di SquadraCorse (FSM)

Caddy

Il container Caddy funge da reverse proxy e da punto di terminazione TLS per i servizi web esposti dal sistema, in primo luogo Grafana. In ascolto sulla porta 443/TCP, Caddy si occupa di gestire automaticamente i certificati HTTPS (ad esempio tramite Let's Encrypt), riducendo in modo significativo la complessità operativa legata alla sicurezza del trasporto, e di instradare le richieste in ingresso verso l'istanza di Grafana esposta sulla rete interna.

Questa separazione tra la responsabilità di visualizzazione, demandata a Grafana, e quella di pubblicazione sicura verso l'esterno, affidata a Caddy, permette di mantenere tutti i componenti interni — inclusi Prometheus, Tempo e QuestDB — confinati in reti Docker private, esponendo all'esterno un unico endpoint HTTPS gestito e controllato. In tal modo, il Livello di visualizzazione rimane facilmente accessibile ai membri del team senza introdurre superfici di attacco aggiuntive o accoppiamenti stretti con la topologia interna del sistema di telemetria.

Capitolo 6

Sviluppi futuri

L'architettura della libreria **Goccia**, descritta nei capitoli precedenti, ha dimostrato di soddisfare i requisiti di performance e scalabilità richiesti dal sistema di telemetria di Squadra Corse. Tuttavia, l'evoluzione continua dei protocolli di rete e la necessità di supportare scenari di utilizzo sempre più complessi suggeriscono diverse direzioni per l'evoluzione del progetto. Questo capitolo illustra le principali estensioni pianificate, organizzate per area tematica: miglioramenti ai protocolli di ingresso e uscita, pattern avanzati di routing e gestione del flusso, estendibilità dell'API pubblica e consolidamento della qualità del software.

6.1 Evoluzione dei protocolli di trasporto

6.1.1 Supporto al protocollo QUIC

Attualmente, la libreria supporta protocolli di trasporto tradizionali come TCP e UDP. Un'evoluzione naturale per il layer di *Ingress* è l'adozione di QUIC (Quick UDP Internet Connections), un protocollo di trasporto moderno basato su UDP che offre latenza ridotta e migliore gestione della congestione rispetto al TCP.

L'implementazione di un *QUIC Ingress Stage*, basato sulla libreria `quic-go` [73], permetterebbe di gestire stream affidabili e multiplexati su una singola connessione UDP, eliminando il problema dell'*head-of-line blocking* tipico del TCP. Questo scenario è ideale, per esempio, per un sistema di telemetria che opera in condizioni di rete instabili (es. 4G/5G in movimento), dove QUIC garantisce un recupero più rapido dai pacchetti persi. Lo stage opererebbe accettando stream in ingresso e parallelizzando la lettura dei dati in maniera analoga a quanto avviene già per l'ingress stage TCP.

6.1.2 Integrazione del protocollo MQTT

Parallelamente all'adozione di QUIC, si ritiene fondamentale estendere la connettività della libreria verso il mondo IoT implementando un *MQTT Ingress Stage*. Il protocollo MQTT (*Message Queuing Telemetry Transport*), grazie alla sua leggerezza e al modello *publish-subscribe*, rappresenta lo standard *de facto* per la comunicazione machine-to-machine in ambienti a banda limitata, rendendolo un complemento ideale per scenari di telemetria meno critici in termini di real-time ma che richiedono elevata scalabilità.

L'implementazione proposta, basata sulla libreria `eclipse/paho.mqtt.golang` [74], prevederebbe la creazione di un client capace di sottoscrivere a topic multipli, supportando anche l'uso di *wildcards* (es. `telemetry/+/sensors/#`) per aggregare flussi dati eterogenei in un unico punto di ingresso. Un aspetto cruciale sarà la mappatura configurabile tra i livelli di QoS (*Quality of Service*) del protocollo e le garanzie di delivery della pipeline: lo stage dovrà permettere all'utente di bilanciare latenza e affidabilità, scegliendo tra una semantica *fire-and-forget* (QoS 0) per dati ad alta frequenza o confermata (QoS 1/2) per eventi critici, convertendo in modo trasparente i payload binari nel formato di messaggio interno di **Goccia**.

6.1.3 Servizi REST ad alte prestazioni con FastHTTP

Per estendere l'applicabilità di **Goccia** oltre il puro stream processing, si propone l'introduzione di un *Ingress Stage* basato su `valyala/fasthttp` [75], un'alternativa ad alte prestazioni alla libreria standard `net/http` di Go, capace di gestire carichi superiori a 100k richieste al secondo grazie a una gestione aggressiva della memoria (zero memory allocations in hot paths).

Questo sviluppo permetterebbe di esporre endpoint REST direttamente dalla pipeline, trasformando **Goccia** in un motore per microservizi *event-driven*.

6.2 Pattern avanzati di Routing e Flusso

6.2.1 Implementazione del pattern Request-Reply (Futures)

L'introduzione di un ingresso HTTP pone una sfida architetturale: la pipeline è unidirezionale (*fire-and-forget*), mentre HTTP è intrinsecamente sincrono (richiesta-risposta). Per risolvere questa dicotomia, si prevede l'implementazione di un sistema intelligente di *Futures*.

L'architettura proposta prevede due componenti accoppiati:

HTTP Ingress Stage: Alla ricezione di una richiesta, genera un *Correlation ID* univoco e crea un canale di ritorno (la *future*). Il contesto della richiesta

viene sospeso in attesa su questo canale, oppure su un'altra primitiva di sincronizzazione, mentre il payload viene inviato nella pipeline.

HTTP Reply Egress Stage: Uno stadio terminale speciale che, ricevendo il messaggio elaborato, utilizza il *Correlation ID* per recuperare il canale sospeso e inviare la risposta all'Ingress, sbloccando così l'handler HTTP.

Questo meccanismo permetterebbe di eseguire elaborazioni complesse (es. validazione, arricchimento dati, query al database) in modo totalmente asincrono e parallelo, restituendo il risultato al client HTTP senza bloccare i thread di gestione delle connessioni.

6.2.2 Content-Based Routing

Attualmente, la pipeline è lineare o ramificata staticamente (tramite lo stage **Tee**). Per introdurre logiche decisionali dinamiche, si propone lo sviluppo di un *Router Processor*, ispirato al pattern EIP *Content-Based Router* [76].

A differenza dei processori standard che possiedono un singolo connettore di output, questo stage gestirebbe una mappa di connettori di destinazione. Una funzione di routing, definita dall'utente, analizzerebbe il contenuto del messaggio (es. tipo di sensore, livello di priorità, flag di errore) per determinare dinamicamente verso quale ramo della pipeline inoltrare il dato. Ciò abiliterebbe scenari complessi come la separazione, per esempio, del traffico di "Allarme" su un canale prioritario rispetto ad un traffico ordinario.

6.3 Estendibilità e Storage

6.3.1 InfluxDB Egress Stage

Per potenziare le capacità di storicizzazione delle serie temporali, si pianifica l'aggiunta di un *InfluxDB Egress Stage*, utilizzando il client ufficiale `influxdb-client-go` [77]. Questo stage renderebbe la libreria idonea ad essere utilizzata in svariati scenari e contesti, come dal semplice monitoraggio dei sistemi informatici, all'immagazzinamento di dati telemetrici dei satelliti.

6.3.2 Custom Ingress ed Egress

Attualmente, la libreria permette la definizione di logica custom solo per gli stage intermedi (*Processor*). Si intende generalizzare questo concetto introducendo le interfacce `CustomIngress` e `CustomEgress`.

Custom Ingress: Permetterà agli utenti di implementare sorgenti dati proprietarie (es. driver per hardware specifico, protocolli industriali legacy) fornendo semplicemente un canale o una funzione di callback per l'iniezione dei messaggi nella pipeline.

Custom Egress: Consentirà di definire destinazioni arbitrarie (es. API di terze parti, sistemi di messaggistica non supportati nativamente) implementando una semplice interfaccia

Questa modifica renderà **Goccia** una libreria agnostica rispetto ai protocolli, permettendo agli utilizzatori di estenderne le capacità senza modificarne il nucleo.

6.4 Qualità e Affidabilità

6.4.1 Miglioramento della Test Coverage

La robustezza di una libreria infrastrutturale critica come **Goccia** richiede una copertura di test rigorosa. Gli sviluppi futuri includeranno:

Integration Tests: Utilizzo di container Docker effimeri (tramite librerie come **Testcontainers**) per validare gli stage di Ingress/Egress contro istanze reali di Kafka, InfluxDB e QuestDB, superando i limiti dei mock attuali.

Fuzz Testing: Applicazione di tecniche di *fuzzing* sui parser dei protocolli (Cannelloni, CSV, JSON) per identificare edge case e potenziali panic causati da input malformati, garantendo la stabilità del sistema anche in presenza di dati corrotti.

Capitolo 7

Conclusioni

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Porttitor eget dolor morbi non arcu risus quis varius. Libero id faucibus nisl tincidunt. Neque laoreet suspendisse interdum consectetur libero id faucibus nisl tincidunt. Scelerisque in dictum non consectetur a erat. Leo a diam sollicitudin tempor id eu. Sodales ut eu sem integer vitae justo eget magna fermentum. A cras semper auctor neque vitae. Cursus euismod quis viverra nibh cras pulvinar. Mi tempus imperdiet nulla malesuada pellentesque elit eget gravida cum. Dictum at tempor commodo ullamcorper a lacus vestibulum sed. Ultricies mi eget mauris pharetra. In pellentesque massa placerat dui ultricies lacus. Fringilla phasellus faucibus scelerisque eleifend donec pretium vulputate.

Aliquam sem fringilla ut morbi tincidunt augue interdum. Risus at ultrices mi tempus imperdiet nulla malesuada pellentesque. Semper quis lectus nulla at volutpat. Nullam non nisi est sit amet facilisis. Eget velit aliquet sagittis id consectetur purus ut faucibus pulvinar. Ultricies tristique nulla aliquet enim tortor at auctor urna nunc. Pharetra diam sit amet nisl suscipit adipiscing bibendum est ultricies. Amet facilisis magna etiam tempor. Nunc non blandit massa enim nec dui nunc mattis. At ultrices mi tempus imperdiet nulla malesuada pellentesque. Cursus metus aliquam eleifend mi in nulla posuere. In eu mi bibendum neque egestas congue quisque. Augue eget arcu dictum varius dui at consectetur.

Bibliografia

- [1] Omar Ferro. *Goccia: High Performance Golang Pipeline Library*. 2025. URL: <https://github.com/Ferro02000/goccia> (visitato il giorno 17/02/2026) (cit. a p. 3).
- [2] Omar Ferro. *goccia package - github.com/Ferro02000/goccia*. 2025. URL: <https://pkg.go.dev/github.com/Ferro02000/goccia> (visitato il giorno 17/02/2026) (cit. a p. 3).
- [3] The Go Authors. *The Go Programming Language*. 2025. URL: <https://golang.org> (visitato il giorno 31/01/2026) (cit. a p. 6).
- [4] Rob Pike. *Concurrency is not Parallelism*. 2012. URL: <https://go.dev/blog/waza-talk> (visitato il giorno 31/01/2026) (cit. a p. 6).
- [5] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. 2018 (cit. a p. 7).
- [6] Gidi Grinstein e Stuart Feldman. *Reactive Streams: Processing Data-Intensive Workloads*. 2014. URL: <https://www.reactivemanifesto.org/> (visitato il giorno 31/01/2026) (cit. a p. 7).
- [7] Herb Sutter. *Lock-Free Programming Survey*. 2023. URL: <https://www.1024cores.net/> (visitato il giorno 31/01/2026) (cit. a p. 8).
- [8] Maurice P. Herlihy. *Wait-Free Synchronization*. 1991 (cit. a p. 8).
- [9] Maurice P. Herlihy e Nir Shavit. *The Art of Multiprocessor Programming*. 2020 (cit. a p. 8).
- [10] Donald E. Knuth. *The Art of Computer Programming: Volume 1 - Fundamental Algorithms*. 1997 (cit. a p. 9).
- [11] Martin Thompson et al. *Disruptor: High-performance inter-thread messaging*. 2024. URL: <https://github.com/LMAX-Exchange/disruptor> (visitato il giorno 31/01/2026) (cit. a p. 9).

- [12] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill e David A. Wood. «DBMSs On A Modern Processor: Where Does Time Go?» In: *Proceedings of the 25th International Conference on Very Large Data Bases* (1999), pp. 266–277 (cit. a p. 9).
- [13] Thomas E. Anderson. *The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors*. 1990 (cit. a p. 10).
- [14] International Organization for Standardization. *ISO 11898-1:2015 – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. 2015. URL: <https://www.iso.org/standard/63648.html> (visitato il giorno 15/02/2026) (cit. a p. 11).
- [15] Apache Software Foundation. *Apache Kafka Documentation*. 2023. URL: <https://kafka.apache.org/documentation/> (visitato il giorno 15/02/2026) (cit. a p. 12).
- [16] eBPF Foundation. *What is eBPF?* 2025. URL: <https://ebpf.io/what-is-ebpf/> (visitato il giorno 15/02/2026) (cit. a p. 13).
- [17] QuestDB. *Architecture Overview - QuestDB*. 2025. URL: <https://questdb.com/docs/architecture/questdb-architecture/> (visitato il giorno 15/02/2026) (cit. alle pp. 14, 73).
- [18] OpenTelemetry Authors. *What is OpenTelemetry?* 2025. URL: <https://opentelemetry.io/docs/what-is-opentelemetry/> (visitato il giorno 15/02/2026) (cit. a p. 15).
- [19] OpenTelemetry Authors. *OpenTelemetry Collector Documentation*. 2026. URL: <https://opentelemetry.io/docs/collector/> (visitato il giorno 07/02/2026) (cit. alle pp. 15, 75).
- [20] Charles C. Holt. *Forecasting Seasonals and Trends by Exponentially Weighted Moving Averages*. 1957. (Visitato il giorno 17/01/2026) (cit. a p. 16).
- [21] M. G. Chung et al. *Efficient Jitter Compensation Using Double Exponential Smoothing*. 2013. (Visitato il giorno 17/01/2026) (cit. alle pp. 16, 18, 19).
- [22] Rob J. Hyndman e George Athanasopoulos. *Forecasting: Principles and Practice*. 2014. URL: <https://otexts.com/fpp2/> (visitato il giorno 17/01/2026) (cit. alle pp. 16–18).
- [23] Rob J. Hyndman e George Athanasopoulos. *Forecasting: Principles and Practice*. 2022. URL: <https://otexts.com/fpp3/holt.html> (visitato il giorno 17/01/2026) (cit. alle pp. 16–18).
- [24] John Galt Solutions. *Holt's Double Exponential Smoothing*. 2024. URL: <https://johngalt.com/forecasting-methods/holts-double-exponential-smoothing> (visitato il giorno 17/01/2026) (cit. a p. 16).

- [25] Holt C. C. *Holt's Double Exponential Smoothing Theory*. Narotama University. 2020. URL: https://m.narotama.ac.id/ngupload/P-20200430-155040Holt's%20Double%20Exponential_smoothing_theory.pdf (visitato il giorno 17/01/2026) (cit. a p. 17).
- [26] The Go Authors. *Package sync/atomic - Uint64 Type*. 2025. URL: <https://pkg.go.dev/sync/atomic#Uint64> (visitato il giorno 27/01/2026) (cit. a p. 23).
- [27] The Go Authors. *Package cpu - CacheLinePad Type*. 2025. URL: <https://pkg.go.dev/golang.org/x/sys@v0.39.0/cpu#CacheLinePad> (visitato il giorno 27/01/2026) (cit. a p. 23).
- [28] Trisha Thompson. *Dissecting the Disruptor: Why it's so fast (Part Two) - Magic Cache Line Padding*. 2011. URL: https://trishagee.com/2011/07/22/dissecting_the_disruptor_why_its_so_fast_part_two__magic_cache_line_padding/ (visitato il giorno 27/01/2026) (cit. a p. 24).
- [29] The Go Authors. *Package runtime - Gosched Function*. 2025. URL: <https://pkg.go.dev/runtime#Gosched> (visitato il giorno 27/01/2026) (cit. a p. 24).
- [30] Wikipedia. *Heartbeat (computing)*. 2026. URL: [https://en.wikipedia.org/wiki/Heartbeat_\(computing\)](https://en.wikipedia.org/wiki/Heartbeat_(computing)) (visitato il giorno 15/02/2026) (cit. a p. 26).
- [31] Cloud Native Computing Foundation. *OpenTelemetry Trace API Specification*. 2025. URL: <https://opentelemetry.io/docs/specs/otel/trace/api/> (visitato il giorno 29/12/2025) (cit. a p. 26).
- [32] The Go Authors. *Package net - UDPConn Type*. 2025. URL: <https://pkg.go.dev/net#UDPConn> (visitato il giorno 29/12/2025) (cit. a p. 27).
- [33] The Go Authors. *Package net - ErrClosed Variable*. 2025. URL: <https://pkg.go.dev/net#ErrClosed> (visitato il giorno 29/12/2025) (cit. a p. 27).
- [34] The Go Authors. *Package sync - Pool Type*. 2025. URL: <https://pkg.go.dev/sync#Pool> (visitato il giorno 29/12/2025) (cit. a p. 28).
- [35] Cloud Native Computing Foundation. *OpenTelemetry Metrics API Specification - Asynchronous Counter*. 2025. URL: <https://opentelemetry.io/docs/specs/otel/metrics/api/#asynchronous-counter> (visitato il giorno 29/12/2025) (cit. a p. 28).
- [36] The Go Authors. *Package net - TCPListener.Accept Method*. 2025. URL: <https://pkg.go.dev/net#TCPListener.Accept> (visitato il giorno 30/12/2025) (cit. a p. 30).
- [37] Segmentio. *kafka-go: Golang client library for Apache Kafka*. 2026. URL: <https://github.com/segmentio/kafka-go> (visitato il giorno 02/01/2026) (cit. a p. 33).

- [38] Confluent Inc. *librdkafka: The Apache Kafka C/C++ client library*. 2026. URL: <https://github.com/confluentinc/librdkafka> (visitato il giorno 02/01/2026) (cit. a p. 33).
- [39] Segmentio. *Package kafka - ReaderConfig Type*. 2026. URL: <https://pkg.go.dev/github.com/segmentio/kafka-go@v0.4.49#ReaderConfig> (visitato il giorno 02/01/2026) (cit. a p. 33).
- [40] Segmentio. *Package kafka - Reader.ReadMessage Method*. 2026. URL: <https://pkg.go.dev/github.com/segmentio/kafka-go@v0.4.49#Reader.ReadMessage> (visitato il giorno 02/01/2026) (cit. a p. 33).
- [41] Cloud Native Computing Foundation. *OpenTelemetry TextMap Propagator Specification*. 2026. URL: <https://opentelemetry.io/docs/specs/otel/context/api-propagators/#textmap-propagator> (visitato il giorno 02/01/2026) (cit. a p. 34).
- [42] Cilium Project. *cilium/ebpf: eBPF library for Go*. 2026. URL: <https://github.com/cilium/ebpf> (visitato il giorno 02/01/2026) (cit. a p. 34).
- [43] Cilium Project. *Package link - Link Interface*. 2026. URL: <https://pkg.go.dev/github.com/cilium/ebpf@v0.20.0/link#Link> (visitato il giorno 02/01/2026) (cit. a p. 35).
- [44] Cilium Project. *Package rlimit - RemoveMemlock Function*. 2026. URL: <https://pkg.go.dev/github.com/cilium/ebpf@v0.20.0/rlimit#RemoveMemlock> (visitato il giorno 02/01/2026) (cit. a p. 35).
- [45] fsnotify Contributors. *fsnotify: File system notifications for Go*. 2026. URL: <https://github.com/fsnotify/fsnotify> (visitato il giorno 02/01/2026) (cit. a p. 37).
- [46] The Go Authors. *Package bufio - Reader Type*. 2026. URL: <https://pkg.go.dev/bufio#Reader> (visitato il giorno 02/01/2026) (cit. a p. 37).
- [47] fsnotify Contributors. *Package fsnotify - Watcher type*. 2026. URL: <https://pkg.go.dev/github.com/fsnotify/fsnotify@v1.9.0#Watcher> (visitato il giorno 02/01/2026) (cit. a p. 38).
- [48] Martin Günther. *Cannelloni: CAN over Ethernet gateway*. 2026. URL: <https://github.com/mguentner/cannelloni> (visitato il giorno 03/01/2026) (cit. a p. 43).
- [49] Squadra Corse Polito. *acmelib: CAN message definition and decoding library for Go*. 2026. URL: <https://github.com/squadracorsepolito/acmelib> (visitato il giorno 03/01/2026) (cit. alle pp. 45, 70).
- [50] The Go Authors. *Package utf8 - DecodeRune Function*. 2026. URL: <https://pkg.go.dev/unicode/utf8#DecodeRune> (visitato il giorno 04/01/2026) (cit. a p. 46).

- [51] The Go Authors. *Package strconv - String Conversion*. 2026. URL: <https://pkg.go.dev/strconv> (visitato il giorno 04/01/2026) (cit. a p. 46).
- [52] The Go Authors. *Package net - UDPConn.Write Method*. 2026. URL: <https://pkg.go.dev/net#UDPConn.Write> (visitato il giorno 07/01/2026) (cit. a p. 55).
- [53] The Go Authors. *Package net - TCPConn.Write Method*. 2026. URL: <https://pkg.go.dev/net#TCPConn.Write> (visitato il giorno 09/01/2026) (cit. a p. 56).
- [54] Segmentio. *Package kafka - Writer Type*. 2026. URL: <https://pkg.go.dev/github.com/segmentio/kafka-go@v0.4.49#Writer> (visitato il giorno 22/01/2026) (cit. a p. 56).
- [55] Segmentio. *Package kafka - Writer.WriteMessages Method*. 2026. URL: <https://pkg.go.dev/github.com/segmentio/kafka-go@v0.4.49#Writer.WriteMessages> (visitato il giorno 22/01/2026) (cit. a p. 57).
- [56] Go Authors. *Package bufio - Writer Type*. 2026. URL: <https://pkg.go.dev/bufio#Writer> (visitato il giorno 24/01/2026) (cit. a p. 58).
- [57] QuestDB. *QuestDB: Time Series Database*. 2026. URL: <https://questdb.com/> (visitato il giorno 24/01/2026) (cit. a p. 58).
- [58] QuestDB. *go-questdb-client: QuestDB Go Client Library*. 2026. URL: <https://github.com/questdb/go-questdb-client> (visitato il giorno 24/01/2026) (cit. a p. 58).
- [59] QuestDB. *Package questdb - LineSenderPool Type*. 2026. URL: <https://pkg.go.dev/github.com/questdb/go-questdb-client/v3@v3.2.0#LineSenderPool> (visitato il giorno 24/01/2026) (cit. a p. 59).
- [60] FerroO2000. *Goccia - Internal Ring Buffer Tests*. 2026. URL: https://github.com/Ferro02000/goccia/blob/master/internal/rb/ring_buffer_test.go (visitato il giorno 08/02/2026) (cit. a p. 62).
- [61] Squadra Corse Polito. *SCanner: Telemetry acquisition board firmware and tools*. 2026. URL: <https://github.com/squadracorsepolito/SCanner> (visitato il giorno 07/02/2026) (cit. a p. 68).
- [62] Squadra Corse Polito. *sc-telemetry: Telemetry processing server for CAN bus data*. 2026. URL: <https://github.com/squadracorsepolito/sc-telemetry> (visitato il giorno 07/02/2026) (cit. a p. 70).
- [63] Omar Ferro. *scomarferro/sc-telemetry Docker Image*. 2026. URL: <https://hub.docker.com/r/scomarferro/sc-telemetry> (visitato il giorno 07/02/2026) (cit. a p. 70).

- [64] FileFormat.com. *DBC File Format Specification*. 2026. URL: <https://docs.fileformat.com/it/database/dbc/> (visitato il giorno 07/02/2026) (cit. a p. 70).
- [65] QuestDB. *QuestDB Fast Ingestion Benchmark*. 2026. URL: <https://questdb.io/blog/questdb-fast-ingestion-benchmark> (visitato il giorno 07/02/2026) (cit. a p. 73).
- [66] GridDB. *Time Series Database Benchmark Comparison: GridDB vs QuestDB vs TimescaleDB*. 2026. URL: <https://griddb.net/en/blog/time-series-database-benchmark-comparison-griddb-questdb-timescaledb> (visitato il giorno 07/02/2026) (cit. a p. 73).
- [67] The Prometheus Authors. *Prometheus: From metrics to insight*. 2026. URL: <https://prometheus.io/> (visitato il giorno 07/02/2026) (cit. a p. 75).
- [68] Grafana Labs. *Grafana Tempo: High volume, high cardinality distributed tracing*. 2026. URL: <https://grafana.com/oss/tempo/> (visitato il giorno 07/02/2026) (cit. a p. 75).
- [69] Grafana Labs. *Grafana: The open and composable observability platform*. 2026. URL: <https://grafana.com/> (visitato il giorno 07/02/2026) (cit. a p. 77).
- [70] Caddy Authors. *Caddy Documentation*. 2026. URL: <https://caddyserver.com/docs/> (visitato il giorno 07/02/2026) (cit. a p. 77).
- [71] Google. *Jsonnet: The data templating language*. 2026. URL: <https://jsonnet.org/> (visitato il giorno 07/02/2026) (cit. a p. 77).
- [72] Grafana Labs. *grafonnet: Jsonnet library for Grafana dashboards*. 2026. URL: <https://github.com/grafana/grafonnet> (visitato il giorno 07/02/2026) (cit. a p. 77).
- [73] Marten Seemann. *quic-go: A QUIC implementation in pure Go*. 2026. URL: <https://github.com/quic-go/quic-go> (visitato il giorno 10/02/2026) (cit. a p. 80).
- [74] Eclipse Foundation. *eclipse/paho.mqtt.golang: Eclipse Paho MQTT Go client*. 2026. URL: <https://github.com/eclipse-paho/paho.mqtt.golang> (visitato il giorno 10/02/2026) (cit. a p. 81).
- [75] Aliaksandr Valyala. *fasthttp: Fast HTTP package for Go*. 2026. URL: <https://github.com/valyala/fasthttp> (visitato il giorno 10/02/2026) (cit. a p. 81).
- [76] Gregor Hohpe e Bobby Woolf. *Content-Based Router - Enterprise Integration Patterns*. 2026. URL: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html> (visitato il giorno 10/02/2026) (cit. a p. 82).

- [77] InfluxData. *influxdb-client-go: InfluxDB 2 Client Go*. 2026. URL: <https://github.com/influxdata/influxdb-client-go> (visitato il giorno 10/02/2026) (cit. a p. 82).