UNIVERSITÀ DEGLI STUDI DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Informatica

Dipartimento di Informatica — Scienza e Ingegneria — DISI

TESI DI LAUREA in

Reti di Calcolatori T

Valutazione di strumenti per l'osservabilità di Apache Kafka con OpenTelemetry

Tesi di Laurea di:
Alessio Giorgetti

Relatore:
Chiar.mo Prof. Ing. Antonio Corradi

Correlatori:

Dr. Andrea Sabbioni
Dott. Luca Serfilippi

Anno Accademico 2023 – 2024

Indice

Introduzione

La necessità di avere risorse computazionali pronte all'uso, accessibili velocemente, non richiedenti uno sforzo gestionale e soprattutto a costi decisamente bassi è stata al centro di grandi problematiche che nel corso degli anni hanno trovato soluzione nel Cloud Computing. Una volta adottato un modello organizzativo di questo tipo, è stato possibile non solo ridurre significativamente i costi di utilizzo e gestione, ma anche deallocare le risorse in modo efficiente, consentendo la loro condivisione e offrendo agli utenti l'impressione di avere a disposizione risorse illimitate. La nascita del cloud computing ha infatti cambiato molti paradigmi che ormai negli anni erano stati assodati, è nato un nuovo modo di programmare, non più orientato a monoliti ma parti dello stesso sistema sono suddivise in microservizi autonomi, le risorse non devono più per forza essere gestite e mantenute dall'azienda che li possiede ma possono essere affidate a strutture esterne e specializzate causando un abbattimento dei costi rilevante. La tendenza a deallocare servizi e separare i componenti di un'applicazione ha creato la necessità di trasmettere enormi quantità di dati, sia quelli raccolti e archiviati dalle applicazioni, sia le comunicazioni che consentono l'interoperabilità tra questi servizi.

Nascono dunque nuovi componenti architetturali interamente pensati per garantire una gestione corretta ed estremamente efficiente di questi flussi di dati. Una delle architetture più diffuse per gestire stream dati e applicazioni ad eventi è Apache Kafka. Questa infatti permette di montare un'infrastruttura caratterizzata da produttori e consumatori che messi in comunicazione da diversi broker possono immagazzinare, leggere e scrivere enormi quantità di record in tempi molto ridotti.

L'efficienza permessa da questi sistemi però si scontra con la facilità di comprendere le numerose interazioni che i sistemi distribuiti compiono con i vari attori dell'infrastruttura, diventa dunque difficile gestire ed individuare problemi, malfunzionamenti o eventuali rallentamenti del sistema. È emerso rapidamente che i tradizionali sistemi di monitoraggio, alla base delle tecniche di programmazione più diffuse, non erano adatti a questo nuovo ambiente così dinamico. Questo ha portato all'evoluzione del concetto di osservabilità: la capacità di dedurre e comprendere gli stati interni di un sistema software partendo dalla conoscenza dei suoi output esterni. Una tecnica che si è subito rivelata fondamentale per il

monitoraggio nel mondo del cloud e per aiutare gli sviluppatori a comprendere le numerose interazioni tra i sistemi.

L'integrazione di strumenti adibiti all'osservabilità come OpenTelemetry con infrastrutture distribuite quali Apache Kafka ha rappresentato una combinazione estremamente efficiente per comprendere il comportamento di un sistema così variabile, il cui funzionamento dipende da un'infrastruttura altamente scalabile. L'infrastruttura Kafka, grazie alla sua alta personalizzazione e scalabilità, si dimostra una soluzione ideale non solo per gestire grandi flussi di dati, ma anche per scenari meno dispendiosi in termini di risorse, rendendola versatile per qualsiasi esigenza lavorativa. La capacità di comprendere lo stato interno di un sistema software in costante evoluzione consente di raccogliere tutte le informazioni necessarie per monitorare, ottimizzare e individuare le configurazioni più appropriate in tempo reale, in modo che il sistema possa rispondere alle sfide del carico in modo affidabile, efficiente e resiliente.

L'obiettivo di questo progetto sarà esattamente questo: utilizzare OpenTelemetry per testare diverse configurazioni di architetture Kafka, profondamente differenti ma basate sullo stesso modello, al fine di comprendere come queste variazioni influenzino lo scambio di messaggi all'interno del sistema. Nel mondo dei sistemi distribuiti infatti la velocità con cui i messaggi viaggiano nel sistema non è l'unica cosa da ricercare in un progetto, caratteristiche fondamentali sono anche la capacità dell'applicazione di scalare in base al carico a cui è sottoposta o di tollerare malfunzionamenti senza intaccare il funzionamento generale del sistema, soprattutto in contesti di sistemi real-time o critici. È dunque evidente che la conoscenza di tali aspetti risulta significativa per una corretta scelta dei componenti architetturali e delle specifiche tecnologie che li realizzano.

L'organizzazione di questo elaborato prevede di ampliare e approfondire le idee e le tecnologie appena introdotte, fornendo inizialmente una visione d'insieme del mondo del cloud computing e delle applicazioni distribuite. Successivamente, si entrerà nel dettaglio delle specifiche che caratterizzano le comunicazioni in questo tipo di ambiente. Questi passaggi saranno necessari per introdurre le principali tecnologie alla base del progetto e spiegare come verranno fatte interoperare. Infine, il progetto verrà illustrato, insieme ai relativi obiettivi, e su di esso verranno condotti test, i cui risultati saranno analizzati e discussi.

1 Cloud Computing

Nella prima parte verrà introdotto il concetto di cloud computing e le sue caratteristiche principali. Nello specifico esporremo non solo la sua definizione e principali caratteristiche ma seguiranno i principali vantaggi, i vari modelli di cloud deployment e i principali modelli di servizio. Ci soffermeremo su ognuno di essi descrivendone caratteristiche, vantaggi e svantaggi.

Il cloud computing consiste nella fornitura di servizi di computing, quali software, database, server e reti, tramite connessione Internet. Ciò significa che gli utenti finali sono in grado di accedere a software e dati ovunque si trovino. Questo permette di eliminare la necessità di memorizzare dati e applicazioni sui dispositivi locali.
Il NIST (National Institute of Standards Technology) fornisce la seguente definizione:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resource that can be rapidly provisioned and released with minimal management effort or service provider interaction.[1]

Il cloud computing dunque si basa sul presupposto che le principali attività di elaborazione avvengono su una macchina in remoto, attraverso la decentralizzazione delle risorse hardware verso enti esterni all'azienda che si prendono carico della gestione delle macchine offrendo un insieme di servizi elastici pay-per-use. Questo permette all'utente di usufruire di servizi complessi e dispendiosi, sia dal punto di vista delle risorse informatiche che dalle energie utilizzate, senza la necessità di possedere hardware avanzati e personale specializzato. Un'organizzazione di questo genere permette di offrire al clienti servizi altamente scalabili e adattabili in caso di variazione della domanda di risorse, una buona tolleranza ai guasti, qualità di servizio, sicurezza, alta disponibilità e bassi costi.
In un sistema basato sul cloud computing possiamo riconoscere cinque principali attori che interagiscono tra di loro: [2]

- **Cloud Provider:** si occupa di mettere a disposizione i servizi ai clienti (storage, applicazioni, capacità di calcolo).
- **Cloud Consumer:** è la persona o organizzazione che usufruisce dei servizi forniti dal Provider.
- **Cloud Broker:** è l'intermediario tra il provider e il consumer. Principalmente si occupa della guida nella scelta dei servizi offerti.
- **Cloud Auditor:** è colui che si occupa di valutare i servizi offerti dal cloud provider in termini di sicurezza, impatto sulla privacy, prestazioni e altro.
- **Cloud Carrier:** è l'intermediario che fornisce connettività e trasporto ai servizi cloud tra cloud consumer e cloud provider.

Affinché un'applicazione cloud possa essere definita come tale, è necessario che i servizi e i componenti che la realizzano dimostrino determinate caratteristiche comportamentali. Difatti non è sufficiente che l'applicazione sia basata sul web per definirla come servizio cloud. Secondo il NIST le caratteristiche chiave del cloud computing sono [3]:

On-demand Self-Service. Un cliente può usufruire di un servizio in modo completamente automatico, senza la necessità di un intervento manuale da parte di un operatore. Sia il processamento della richiesta che la sua soddisfazione devono essere completamente automatizzati.

Broad Network Access. L'hardware fisico deve essere distribuito su scala globale e facilmente accessibile attraverso meccanismi standard, permettendo a dispositivi eterogenei l'accesso ai dati in qualsiasi luogo a patto di avere una connessione ad Internet.

Resource Pooling. Le risorse computazionali del Provider vengono usate in modelli condivisi per servire più clienti contemporaneamente. Diverse risorse sia fisiche che virtuali vengono così assegnate e riassegnate in base alla richiesta dei clienti. L'utente dunque è davanti ad un'astrazione che gli permette di non sapere e quindi non preoccuparsi di dove sono allocate le risorse. Questo causa un abbattimento dei costi ed una maggiore flessibilità per i fornitori dei servizi cloud.

Rapid Elasticity. L'ambiente cloud è in grado di scalare in base al traffico a cui è sottoposto, infatti le risorse anche se sono disponibili non vengono utilizzate finché non sono necessarie. Grazie a controlli automatici del carico i sistemi cloud possono infatti adattare le risorse agli utenti in base al loro bisogno e in caso fare fronte agilmente a richieste di traffico molto elevate garantendo la soddisfazione di tutte le richieste dei clienti.

Measured Service. Ogni servizio cloud deve essere in grado di fornire una misura delle risorse utilizzate. Tale utilizzo può essere quantificato in termini di banda, tempo o dati. Questo aspetto è alla base del modello "pay-per use", caratteristica chiave del cloud computing in cui i clienti pagano in base alla quantità effettiva di risorse utilizzate.

Per fornire una definizione completa sul cloud computing il NIST definisce altre due macro-categorie: i **modelli di servizio** ed i **modelli di deployment** (*Figura 1.1*). Nel seguito verranno definiti i tre modelli di servizio e i quattro modelli di deployment principali. [4]

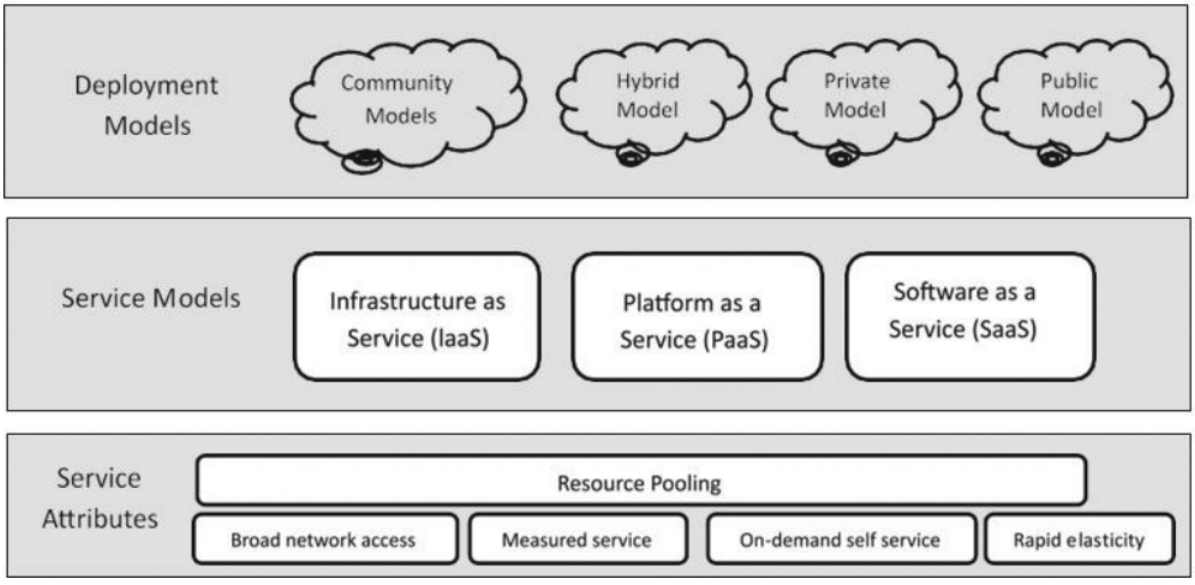

Figura 1.1: Modello di ambiente cloud secondo il NIST

Fonte: T.B. Rehman, 'Cloud Computing Bases', 2018

1.2 Modelli di cloud deployment

I modelli di cloud deployment rappresentano le diverse modalità attraverso le quali vengono messi a disposizione di utenti e organizzazioni i servizi e le risorse cloud.
Il NIST definisce **quattro modelli principali di deployment** che fungono da fondamenta per la pianificazione e l'implementazione delle infrastrutture cloud e si differenziano per costi, necessità organizzative, attività realizzabili diverse e obiettivi diversi. [5]

1.2.1 Private Cloud

I *Private Cloud* rappresentano un modello in cui tutte le infrastrutture necessarie per la realizzazione dell'ambiente cloud sono dedicate ed accessibili solo all'azienda che li possiede. Questi tipi di infrastrutture sono solitamente alloggiate in un datacenter presso l'organizzazione che li possiede, è tuttavia possibile che vengano forniti da un provider esterno e indipendente.
Un'organizzazione simile non consente una divisione delle responsabilità netta in quanto l'azienda sarà totalmente responsabile per acquisto, gestione, manutenzione e sicurezza dell'intera infrastruttura.
I principali aspetti positivi sono non solo una maggiore sicurezza e controllo dei propri dati ma anche una maggiore libertà di personalizzazione sia hardware sia software in base alle necessità. Si aggiunge anche la possibilità di avere completo controllo in quello che riguarda manutenzione e monitoraggio dei sistemi.
Nonostante i benefici appena descritti, sull'azienda peserebbe anche una maggiore responsabilità a fronte di tutti i problemi legati alla gestione autonoma quali l'acquisto, aggiornamento e manutenzione di tutto l'hardware di cui è composta l'infrastruttura, considerando anche il fatto che un ambiente simile deve poter sopportare un aumento del carico improvviso e spesso anche solo temporaneo.
Ad aggiungersi ai costi, già elevati, appena esposti ci sarebbero quelli relativi agli specialisti e sistemisti che avrebbero il compito di monitorare e gestire tali infrastrutture.

1.2.2 Community Cloud

I *Community Cloud* sono una versione di cloud parzialmente pubblici, essi infatti sono condivisi tra i membri di un gruppo selezionato di organizzazioni. Vari possono essere i motivi che spingono un'azienda ad aderire a tale modello, i principali sono sicuramente dati dall'affinità di obiettivi che caratterizzano certe aziende nonché la necessità di partizionare i costi e gli eventuali problemi che caratterizzano un *private cloud*.

Anche in questo caso il provider del servizio può essere sia esterno che un membro dell'organizzazione o addirittura diviso tra diversi membri della comunità.

La posizione intermedia di questo modello tra il pubblico e il privato gli consente di avere benefici quali l'abbattimento dei costi, la condivisione di risorse e la ripartizione delle responsabilità.

Il Problema principale è quello che la gestione dell'infrastruttura non è centralizzata ma suddivisa tra i membri della comunità, cosa che può causare problemi se non vengono presi accordi precisi.

1.2.3 Public Cloud

Quello del *Public Cloud* del modello più diffuso dove i servizi cloud vengono distribuiti da un provider il quale si occupa della loro completa gestione ed amministrazione in termini di memoria, gestione dei dati e degli applicativi.

I principali benefici offerti da ambienti cloud pubblici sono identificabili in termini di disponibilità, scalabilità, accessibilità e abbattimento dei costi. La disponibilità in particolare può essere garantita secondo il variare delle richieste del cliente, e analogamente, la scalabilità garantisce ad ogni fruitore una gestione del carico personalizzabile. I provider devono dunque garantire massima accessibilità ai propri servizi garantendone l'utilizzo ad un bacino eterogeneo di clienti e dispositivi.

Infine il fatto che i clienti paghino secondo un meccanismo "pay-per use" garantisce un abbattimento considerevole dei costi per i consumatori.

Tuttavia seguono diversi svantaggi relativi al fatto che la gestione è affidata ad un ente terzo, quali i problemi della gestione dei dati e della loro sicurezza, nonché dal fatto che ogni fruitore dipenda dai vari e ricorrenti aggiornamenti e downtime forzati del provider.

1.2.4 Hybrid Cloud

Il modello *Hybrid Cloud* è la combinazione di almeno due modelli di cloud. Tali modelli non sono mescolati insieme, ogni ambiente è a se stante e separato.
Questa è una soluzione caratterizzata da grande flessibilità e maggiore possibilità di adattamento alle varie dinamiche aziendali tuttavia porta con sé una maggiore complessità nella gestione di ambienti cloud differenti.
I benefici principali sono principalmente quelli delle architetture che lo realizzano oltre a come abbiamo appena esposto, flessibilità e adattabilità. Infatti è possibile gestire risorse differenti in maniera più varia, come ad esempio selezionare quali risorse esportare a cloud pubblici e quali a cloud privati.
Principali problemi causati da un'infrastruttura simile sono quelli di avere una inevitabilmente maggiore complessità operativa scaturita dalla necessità di far interoperare ambienti diversi, ognuno con le proprie regole e procedure anche molto differenti tra loro. Inoltre particolare attenzione va prestata nell'interazione dei diversi ambienti quando si trattano dati condivisi.

1.3 Modelli di servizio

Introduciamo ora i diversi servizi che un cloud provider può offrire. In seguito andrò a descrivere i **tre principali modelli di servizio cloud** che il NIST identifica: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) e Software as a Service (SaaS). [6]
Questi servizi possono essere offerti in modo differente consentendo o meno all'utente la libertà di decidere cosa fare a diversi livelli dell'infrastruttura applicativa.
Ogni modello di servizio cloud implica diversi livelli di autonomia, controllo e gestione da parte degli utenti (*Figura 1.2*).

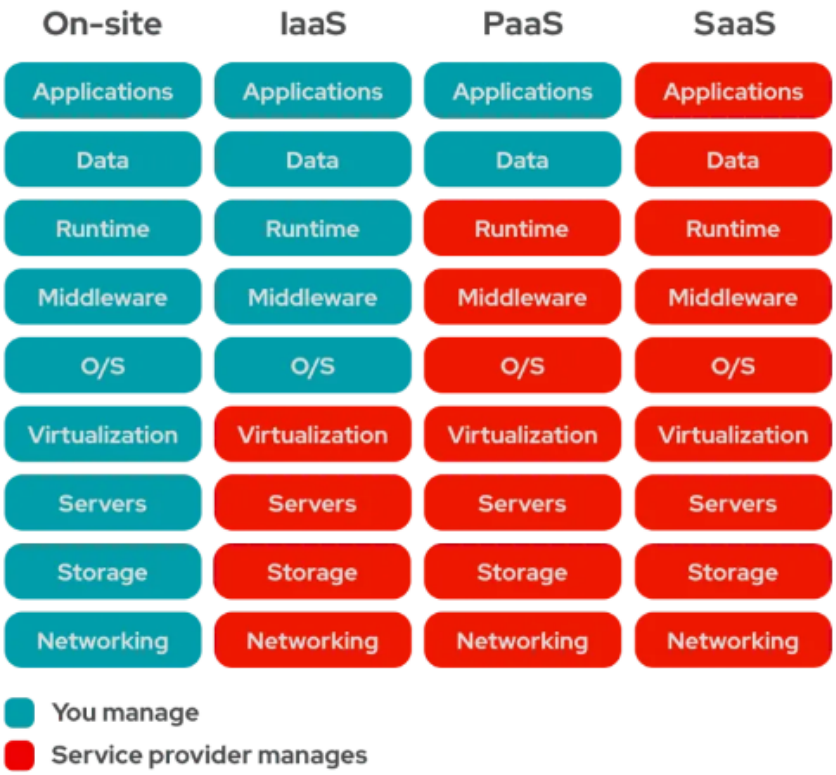

Figura 1.2: Confronto tra Iaas, Saas, Paas

Fonte: https://www.redhat.com/it/topics/cloud-computing/iaas-vs-paas-vs-saas

1.3.1 Infrastructure as a Service (IaaS)

Iaas è un modello di servizio cloud che fornisce all'utente solo le infrastrutture di base, quali server, macchine virtuali e sistemi per il networking.

Un sistema simile consente di rimpiazzare datacenter privati pur mantenendo una certa flessibilità e minori costi di gestione. In questo sistema le responsabilità lasciate al provider sono limitate alla gestione fisica dei dispositivi, lasciando all'utente la possibilità di dedicarsi alla installazione, configurazione e gestione del software e relativa sicurezza.

Vi sono due principali implementazioni di un servizio IaaS. La prima vede il service provider fornire al cliente esclusivamente le risorse hardware a lui necessarie, permettendogli un completo accesso alle macchine fisiche. In questo caso viene a meno la possibilità di distribuzione e scaling del sistema. Il modello più comune invece vede il fornitore mettere a disposizione una macchina virtuale completamente fruibile al cliente. Questo porta ad una distribuzione del sistema su diverse macchine ed a una possibile condivisione di risorse hardware tra più clienti.

1.3.2 Platform as a Service (PaaS)

Il *PaaS* rappresenta un modello di servizio cloud dove vengono forniti al cliente solamente un sistema operativo ed una piattaforma di sviluppo. Questo permette al cliente di dedicarsi alla realizzazione di applicazioni senza preoccuparsi dell'infrastruttura di supporto. Un consumer di un servizio PaaS è responsabile del ciclo di vita completo di un'applicazione Web, in modo più rapido ed economico rispetto ad una soluzione analoga on-premises, utilizzando le risorse rese disponibile dal fornitore e rendono quindi PaaS un servizio scalabile, facile e conveniente.

In un tale sistema le responsabilità sono maggiormente distribuite tra service provider e fruitore del servizio, in particolare il fornitore è generalmente responsabile per tutti i livelli infrastrutturali inferiori al sistema operativo offerto, mentre al cliente resta la gestione dei dati, l'installazione e la manutenzione dell'applicazione.

Prerogativa del cliente nel servizio PaaS è la personalizzazione dell'applicazione del quale è l'unico responsabile. Anche l'analisi delle attività degli utenti che influiscono poi sulla scelta di scaling è compito del fruitore.

Infine i dati seppur memorizzati dal service provider sono pienamente accessibili all'utente in modo diretto.

1.3.3 Software as a Service (SaaS)

Si tratta di un modello di distribuzione software in cui agli utenti viene fornita l'intera applicazione gestita da un provider, alla quale si accede tramite una rete da vari dispositivi client come browser web, senza la necessità di doverla installare localmente sui propri dispositivi. Spesso, una singola istanza dell'applicazione gestisce più clienti, ma i dati di ciascun cliente sono mantenuti separatamente per garantire la privacy, è inoltre importante fare in modo che ogni utente abbia la percezione di essere l'unico utilizzatore del software in questione.

Il provider risulta dunque responsabile per l'aggiornamento e la disponibilità delle applicazioni, per la corretta gestione dei dati e per il monitoraggio di tutto il sistema.

Gli aspetti principali riguardano la personalizzazione del servizio da parte del cliente che tuttavia spesso si limitano a piccole modifiche di interfaccia.

I lati positivi sono la possibilità di adattarsi più facilmente ad un aumento di richieste senza compromettere la qualità del servizio, l'efficienza, e la riduzione dei costi.
Un lato negativo oltre alla poca personalizzazione possibile è che un utente dipende dal calendario degli aggiornamenti del sistema nonostante possano non preoccuparti direttamente.

2 Data Streaming e Architettura a eventi

Nel seguente capitolo introdurremo il concetto di Data Streaming e di architetture distribuite basate su eventi, ne vedremo le caratteristiche principali e perché sono fondamentali per gestire la grande mole di informazioni che vengono scambiate al giorno d'oggi. Infine esporremo i benefici che l'utilizzo di una simile tecnologia può portare ad un ambiente distribuito. Questi concetti saranno fondamentali per capire la piattaforma di streaming Apache Kafka descritta successivamente.

La quantità di dati scambiati e trasmessi è costantemente aumentata nel corso del tempo. L'elaborazione e l'analisi rapida di enormi quantità di dati utilizzando algoritmi di apprendimento automatico sono cruciali nell'era dei big data, soprattutto quando i dati sono sotto forma di flussi. Senza dubbio, i big data sono diventati un'importante fonte di conoscenza e intuizione nel processo decisionale. [7]
Poiché molte attività commerciali aspirano a un vantaggio competitivo, l'analisi in tempo reale sui dati in streaming si è recentemente diffusa. La capacità di organizzare in modo efficiente enormi quantità di dati per prendere decisioni aziendali può fornire un vantaggio non marginale all'azienda [8]. La gestione di questo tipo di dati pone però notevoli ostacoli. [9]

Interessati ad una più immediata ed istantanea elaborazione di flussi di dati però non ci sono solamente le grandi aziende, infatti al giorno d'oggi un utente che utilizza un'applicazione o usufruisce di un servizio si aspetta un'esperienza digitale in tempo reale, più i tempi di elaborazione di una richiesta e la fornitura della risposta sono bassi più un cliente è soddisfatto del servizio. In passato, quando la quantità di informazioni e di elaborazioni era minore, le applicazioni che fornivano risposte in tempo reale erano basate su database e sistemi di elaborazione dei messaggi, tuttavia, quei sistemi non sono più in grado di gestire le elevate quantità di dati generati oggigiorno.
In un modello di data streaming, i dati vengono registrati in un log invece di essere archiviati in un database. I consumatori dunque possono leggere qualsiasi parte del flusso e accedervi in qualunque momento.

Il data streaming è un flusso continuo di informazioni in tempo reale e costituisce la base del modello di software per l'architettura guidata dagli eventi (la vedremo in seguito). Un flusso di dati è quindi una sequenza numerabile e potenzialmente infinita di elementi che viene utilizzato per rappresentare campioni di dati resi disponibili nel tempo. Esempi sono sensori del traffico, sensori medici, transazioni effettuate dai clienti, visite ad un sito web, log delle attività, processi di produzione, email.[10]
Tutte queste informazioni possono essere usate nelle applicazioni moderne dove vengono elaborate, memorizzate e infine analizzate.

L'elaborazione del flusso si riferisce all'analisi dei dati al volo, per produrre nuovi risultati non appena diventano disponibili nuovi input, così da consentire a queste applicazioni di reagire in modo rapido, se necessario. Il tempo è un concetto centrale nell'elaborazione degli stream in quasi tutti i modelli, ogni elemento dello stream è associato a uno o più timestamp da un dato dominio temporale che potrebbero indicare, ad esempio, quando l'elemento è stato generato, la validità del suo contenuto o quando è diventato disponibile per l'elaborazione.[11]
Un flusso di dati è definito dalle seguenti caratteristiche specifiche[12]:

Significativo dal punto di vista cronologico. I singoli elementi di un flusso di dati contengono marche temporali. Il flusso di dati stesso può essere sensibile al tempo e diminuire la propria importanza dopo uno specifico intervallo di tempo. Ad esempio, un'applicazione consiglia un ristorante in base alla posizione attuale dell'utente. È necessario agire sui dati di geolocalizzazione dell'utente in tempo reale, altrimenti i dati perdono significato.

Flusso continuo. Un flusso di dati non ha inizio né fine. Raccoglie dati in modo costante e continuo per tutto il tempo necessario. Ad esempio, i log delle attività del server si accumulano finché il server funziona.

Univoco. La trasmissione ripetuta di un flusso di dati è difficile a causa della sensibilità temporale. Pertanto, un'elaborazione accurata dei dati in tempo reale è

fondamentale. Purtroppo, le disposizioni per la ritrasmissione sono limitate nella maggior parte delle origini di dati in streaming.

Non omogeneo. Alcune fonti possono trasmettere dati in streaming in più formati strutturati come JSON, valori separati da virgole (CSV) con tipi di dati che includono stringhe, numeri, date e tipi binari. I sistemi di elaborazione dei flussi devono essere in grado di gestire tali variazioni di dati.

Imperfetto. Errori temporanei all'origine possono causare elementi danneggiati o mancanti nei dati trasmessi in streaming. Può essere difficile garantire la coerenza dei dati a causa della natura continua del flusso. I sistemi di elaborazione e analisi dei flussi includono in genere una logica di convalida dei dati per ridurre o minimizzare tali errori.

2.1 Scenari di utilizzo

Come già esposto in precedenza i flussi di dati sono fondamentali e trovano applicazione diffusa in tutti quegli scenari in cui vengono continuamente generati dati nuovi e dinamici. Si applica alla maggior parte dei settori e dei casi d'uso relativi ai Big Data.
Di seguito descriverò i principali scenari in cui l'analisi di data stream caratterizza il sistema [12]:

Analisi dei dati. Le applicazioni elaborano flussi di dati per produrre report ed eseguire azioni in risposta, ad esempio generare allarmi quando una determinata misurazione supera una soglia predefinita. Le applicazioni di elaborazione dei flussi più sofisticate estraggono approfondimenti applicando algoritmi di machine learning ai dati delle attività aziendali e dei clienti.

Applicazioni IoT. I dispositivi dell'Internet delle cose (IoT) sono un altro caso di utilizzo dei flussi di dati. I sensori nei veicoli, le apparecchiature industriali e i macchinari agricoli possono inviare dati a un'applicazione in streaming. L'applicazione monitora le prestazioni, rileva anticipatamente i guasti e ordina la parte di ricambio, evitando tempi di inattività.

Analisi finanziaria. Un istituto finanziario utilizza il flusso di dati per monitorare le variazioni del mercato azionario in tempo reale, elaborando il valore a rischio e riequilibrando automaticamente i portafogli finanziari in base alle fluttuazioni dei prezzi delle azioni. Un altro caso d'uso finanziario è il rilevamento delle frodi sulle transazioni con carta di credito, utilizzando l'inferenza in tempo reale rispetto a inviare in streaming i dati delle transazioni.

Suggerimenti in tempo reale. Le applicazioni immobiliari rilevano i dati di geolocalizzazione dai dispositivi mobili dei consumatori e forniscono suggerimenti in tempo reale sulle proprietà da visitare. Analogamente, le applicazioni pubblicitarie, alimentari, di vendita al dettaglio e di consumo possono integrare suggerimenti in tempo reale per dare più valore ai clienti.

Garanzie di servizio. È possibile implementare l'elaborazione dei flussi di dati per monitorare e mantenere i livelli di servizio delle applicazioni e delle apparecchiature. Una società che opera nel settore fotovoltaico, per non incorrere in penali, deve mantenere stabile la velocità di trasmissione effettiva per tutti i clienti. Implementa un'applicazione di streaming dei dati che monitora tutti i pannelli sul campo e pianifica l'assistenza in tempo reale. In questo modo, può ridurre al minimo i periodi di velocità di trasmissione effettiva bassa di ciascun pannello e le relative penali.

Media e videogiochi. Gli editori di media eseguono lo streaming di miliardi di record a richiesta dei propri prodotti online, aggregano e arricchiscono i dati con informazioni demografiche sugli utenti e ottimizzano il posizionamento dei contenuti. Ciò aiuta gli editori a offrire un'esperienza migliore e più pertinente al pubblico. Analogamente, le aziende di giochi online utilizzano l'elaborazione dei flussi di eventi per analizzare le interazioni tra i giocatori e offrire esperienze dinamiche per coinvolgere i giocatori.

Controllo del rischio. Le piattaforme di live streaming e social catturano i dati sul comportamento degli utenti in tempo reale per il controllo del rischio sulle attività

finanziarie degli utenti, come ricariche, rimborsi e premi. Le piattaforme visualizzano dashboard in tempo reale per adattare in modo flessibile le strategie di rischio.

2.2 Attori che compongono lo Streaming

I componenti principali che realizzano un Data Stream sono due: [12]
Il primo è detto **Produttore di flussi,** esso è un componente software facente parte dell'applicazione o del sistema IoT che si occupa di raccogliere dati. Il suo principale compito è quello di trasmettere verso lo stream un record composto da: nome dello stream, valore dei dati e numero di sequenza. Il processore bufferizza o raggruppa temporaneamente i record di dati in base al nome del flusso. Utilizza il numero di sequenza per tracciare la posizione univoca di ciascun record ed elaborare i dati in ordine cronologico. Talvolta ha anche il compito di inviare dati differenti a diversi tipi di stream e sarà suo compito selezionare i record da mandare ad una destinazione piuttosto che ad un'altra.

Il secondo componente fondamentale è il **Consumatore di flussi** in modo inverso al produttore egli ha il compito di ricevere e quindi analizzare i flussi di dati bufferizzati nel processore. Ciascun consumatore dispone di capacità di analisi quali correlazioni, aggregazioni, filtraggio, campionamento o machine learning. Ogni flusso può avere più consumatori e ogni consumatore può elaborare numerosi flussi. I consumatori possono anche inviare i dati modificati al processore per creare nuovi flussi per altri consumatori.

Questi due attori devono però essere sorretti da un'architettura adibita allo streaming.
Necessari sono i livelli di archiviazione ed elaborazione, il livello di storage deve supportare l'ordinamento dei record e una consistenza forte per consentire letture e scritture rapide, poco costose e riproducibili relative a flussi di dati di grandi dimensioni. Il livello di elaborazione ha invece il compito di elaborare i dati provenienti dal livello di storage, eseguire calcoli e quindi inviare notifiche per eliminare i dati non più necessari.

2.3 Criticità

Come abbiamo detto la necessità di soluzioni di questo genere e il grande bacino di utilizzo di questa tecnologia rende gli streaming di dati un'infrastruttura necessaria nell'uso quotidiano di applicativi e servizi su internet, tuttavia non possiamo ignorare il grande livello di complessità che un servizio del genere porta con sé insieme ai relativi problemi.

I flussi di dati devono trasmettere informazioni sequenziali in tempo reale. Le applicazioni di data streaming dipendono dai flussi, che devono garantire alti livelli di disponibilità e coerenza anche nei periodi di attività intensiva. Riuscire a fornire e/o consumare flussi di dati che rispettino sempre tali parametri può diventare complesso.
La quantità di dati non elaborati in un flusso può aumentare esponenzialmente in breve tempo. Pensiamo, ad esempio, agli enormi volumi di dati nuovi che vengono introdotti durante una svendita immediata di titoli azionari, ai post sui social media in occasione di un evento sportivo o ancora all'attività del log quando si verifica un errore di sistema. I flussi di dati devono essere scalabili per definizione. Anche nei periodi di attività intensiva, i flussi devono privilegiare il sequenziamento appropriato dei dati, la coerenza dei dati e la disponibilità. I flussi devono essere inoltre progettati per garantire durabilità in caso di guasto parziale di un sistema.
In un ambiente di cloud ibrido distribuito, i cluster di data streaming hanno delle esigenze speciali. In linea generale i broker di data streaming sono stateful e devono essere preservati in caso di riavvio. La scalabilità richiede un'orchestrazione attenta per garantire che i servizi di messaggistica si comportino come previsto ed evitare le perdite di record. Tutti questi problemi devono essere affrontati su larga scala.[13]

2.4 Panoramica dell'architettura a eventi

L'architettura a eventi è un modello di architettura software per la progettazione di applicazioni. Un sistema guidato dagli eventi è pensato per acquisire, comunicare ed elaborare gli eventi che si verificano tra i servizi disaccoppiati. Ciò significa che i sistemi possono rimanere asincroni ma continuare a condividere le informazioni e svolgere le attività.

Le applicazioni guidate dagli eventi possono essere create in qualsiasi linguaggio di programmazione perché l'essere "event driven" indica un approccio alla programmazione e non un linguaggio.
Un'architettura guidata dagli eventi viene definita a basso livello di accoppiamento perché chi crea un evento (noto come producer eventi) non conosce né il destinatario dell'evento (noto come consumer eventi) né le conseguenze che derivano dal verificarsi dell'evento. [14]

Un evento è descritto come una registrazione di qualsiasi avvenimento o cambiamento nello stato hardware o software di un sistema. La sorgente di un evento può risalire da input esterni o interni. Gli eventi possono essere generati da un utente, per esempio con un clic del mouse o un tasto della tastiera, da una sorgente esterna (come un sensore), oppure dal sistema, per esempio con il caricamento di un programma. [15]

Essendo il Data Streaming la base per l'architettura ad eventi il funzionamento di quest'ultima è anch'esso costituito dalla dinamica produttore (publisher) e consumatore (titolare di una sottoscrizione). Il producer, una volta rilevato un evento lo riproduce sotto forma di messaggio ma, a causa del disaccoppiamento, non conosce il consumer o il risultato dell'evento. Il producer trasmette il messaggio al consumer tramite canali appositi, dove una piattaforma elabora l'evento in modo asincrono.
Una volta che l'evento è mandato dal producer i consumer sottoscritti ai canali appositi vengono notificati dell'evento, a quel punto possono elaborare il messaggio e in caso subirne le conseguenze (*Figura 2.1*).

Vi sono diverse piattaforme distribuite che permettono di realizzare comportamenti in tempo reale come la pubblicazione, l'archiviazione e l'elaborazione dei flussi di eventi, nonché la sottoscrizione agli stessi. Questi ambienti sono in grado di garantire anche una elevata scalabilità e produttività, fondamentali in questo caso, e una riduzione al minimo di interazioni punto a punto, permettendo di ridurre la latenza a millisecondi.

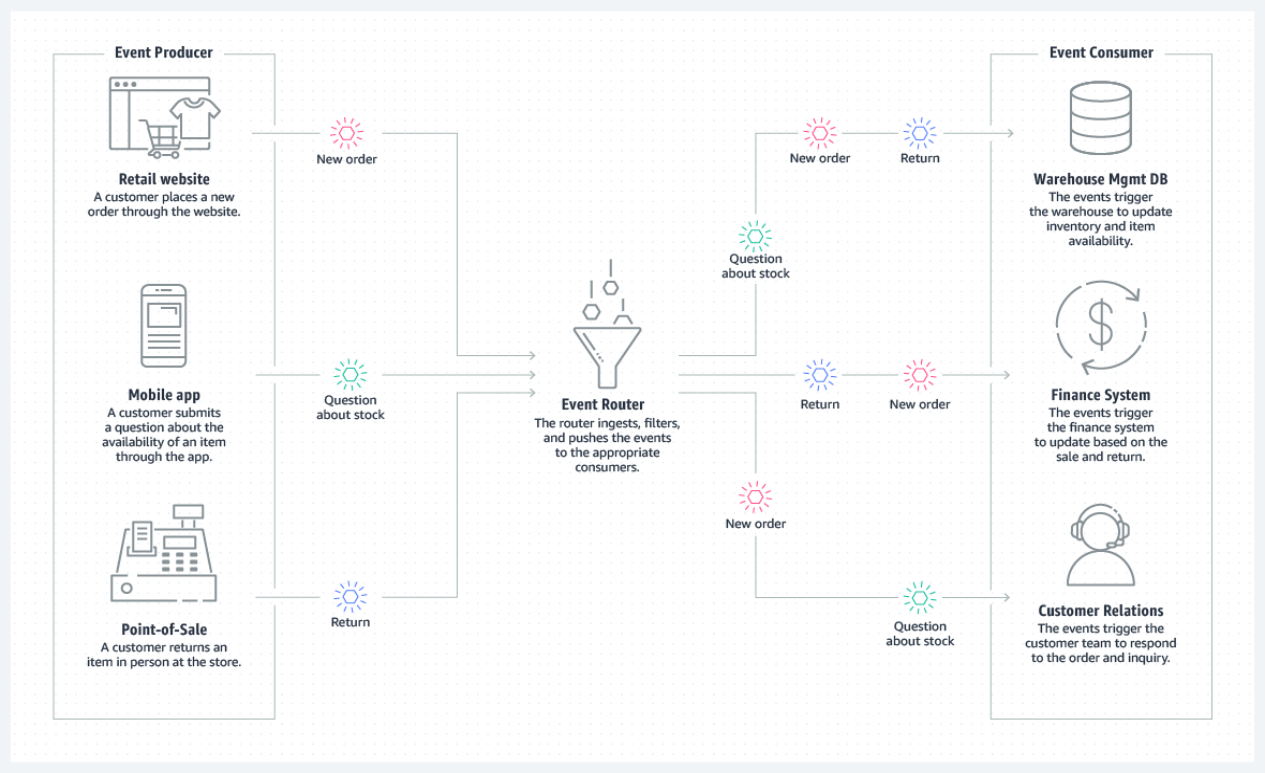

Figura 2.1: esempio di architettura basata su eventi per un sito e-commerce

Fonte: https://aws.amazon.com/it/event-driven-architecture/

2.4.1 Modelli di architettura a eventi

Ci sono due tipi principali di architettura: una basata su pub/sub e l'altra su un flusso di eventi. [15]

- **Modello pub/sub:** Si tratta di un'infrastruttura di messaggistica basata sulle sottoscrizioni a un flusso di eventi. Con questo modello, un evento che si verifica o che viene pubblicato viene inviato ai titolari della sottoscrizione che devono essere informati.
- **Modello flusso di eventi:** In questo caso, gli eventi vengono scritti in un registro. I consumer eventi non effettuano una sottoscrizione al flusso di eventi, ma possono leggere qualsiasi parte del flusso e accedervi in qualunque momento.

Un'architettura guidata dagli eventi può aiutare le organizzazioni a realizzare un sistema flessibile che consenta loro di trasformarsi e prendere decisioni in tempo reale, ottimizzando così i flussi di lavoro. Ciò consente di prendere decisioni di business, sia in modo manuale che automatizzato, utilizzando tutti i dati disponibili sullo stato attuale dei sistemi.

Gli eventi vengono acquisiti mentre si verificano dalle sorgenti, come per esempio dai dispositivi Internet of Things (IoT), dalle applicazioni e dalle reti. In questo modo i producer eventi e i consumer eventi possono condividere il proprio stato e le informazioni sulla risposta in tempo reale.

Le organizzazioni possono aggiungere un'architettura guidata dagli eventi ai propri sistemi e applicazioni per migliorarne la reattività e la scalabilità, infine accedere ai dati e al contesto necessari per ottimizzare il processo decisionale.

L'architettura guida dagli eventi offre il vantaggio del disaccoppiamento, con cui i produttori e i consumatori di dati o servizi non sono costretti a comunicare in modo diretto. Ciò conferisce al sistema maggiore flessibilità e scalabilità. Di conseguenza, l'integrazione di nuovi componenti risulta più semplice, la tolleranza agli errori aumenta e l'efficienza complessiva del sistema viene ottimizzata.

3 Osservabilità

Il termine "osservabilità" deriva dalla teoria del controllo, un'area dell'ingegneria che si occupa di automatizzare il controllo di un sistema dinamico. Tra gli esempi vi sono la regolazione del flusso dell'acqua attraverso un tubo o il controllo della velocità di un'automobile su pendenze e discese, sulla base del feedback del sistema.
Nell'IT e nel cloud computing, l'osservabilità implica l'utilizzo di strumenti e pratiche software. Questi strumenti servono per l'aggregazione, la correlazione e l'analisi di un flusso costante di dati sulle prestazioni da un'applicazione distribuita e dall'hardware e la rete su cui viene eseguita. Questo processo aiuta a monitorare, risolvere i problemi ed eseguire in modo efficace il debug di applicazioni e reti.[16]

3.1 Definizione

IBM definisce l'osservabilità in questo modo:

L'osservabilità è la misura in cui è possibile comprendere lo stato interno o la condizione di un sistema complesso basato solo sulla conoscenza dei suoi output esterni. Quanto più un sistema è osservabile, tanto più rapidamente e accuratamente sarà possibile passare da un problema di prestazioni identificato alla sua causa principale, senza ulteriori test o codifiche. [17]

Per output si intende per esempio logs, metriche e tracciature. In altre parole, indica in che misura è possibile monitorare l'infrastruttura, le applicazioni e le loro interazioni.
Con il cloud e le architetture basate sui microservizi che si stanno velocemente diffondendo le applicazioni stanno raggiungendo livelli di complessità molto elevati, dovuti soprattutto alla grande quantità di interazioni che avvengono durante un servizio. Questo porta ad una più elevata probabilità del verificarsi di malfunzionamenti, guasti o disservizi, trovare a quel punto la fonte dell'errore diventa molto complesso.
Le metodologie di indagine prevalenti negli ultimi venti anni come il monitoraggio delle prestazioni delle applicazioni (APM) non sono più sufficienti per supportare la complessità dei sistemi odierni. Le organizzazioni stanno dunque adottando pratiche di rilevamento più

moderne ed adattabili, che possano dunque permettere una comprensione migliore delle proprietà e delle prestazioni di un'applicazione, nonché rilevare i problemi causati da nuovi errori fino ad ora mai incontrati.

Prima di procedere però è opportuno specificare le differenze che caratterizzano l'osservabilità dal monitoraggio. Spesso infatti viene erroneamente descritta l'osservabilità come un "rebranding" delle antiche tecnologie di monitoraggio. In realtà, l'osservabilità è un'evoluzione naturale dei vecchi metodi di raccolta dei dati che risponde meglio alla natura sempre più rapida, distribuita e dinamica delle implementazioni di applicazioni cloud-native. L'osservabilità non sostituisce il monitoraggio, ma consente di migliorare il monitoraggio.
La distinzione è di fatto molto sottile ma importante, infatti il monitoraggio è un processo di raccolta, analisi e utilizzo delle informazioni per controllare i progressi verso il raggiungimento di un obiettivo. L'osservabilità invece si occupa di farci comprendere in maniera più chiara il comportamento intrinseco delle architetture distribuite che abbiamo sviluppato. Insomma mentre il monitoraggio segnala semplicemente un errore l'osservabilità raccoglie informazioni su cosa non ha funzionato e perché. [18]

3.2 Capire la tracciatura distribuita

La traccia distribuita consente di osservare le richieste mentre si propagano attraverso sistemi complessi e distribuiti. La traccia distribuita migliora la visibilità dell'applicazione e dell'integrità del sistema e consente di eseguire il debug di comportamenti difficili da riprodurre localmente. Poiché l'osservabilità è una tecnologia relativamente nuova, non ci sono ancora standard informatici ben definiti per assicurarla in un sistema. Tuttavia, sono stati individuati tre elementi essenziali, noti come i tre pilastri dell'osservabilità, che sono cruciali per ottenere una chiara comprensione dello stato interno di un sistema. [19]

3.2.1 Logs

Il log è una registrazione sequenziale e cronologica delle operazioni effettuate da un sistema informatico. Il log può derivare da varie fonti, ad esempio: un server, un'applicazione, un

client, un software e vengono memorizzati in specifici file non modificabili. Sono stati pensati per poter tracciare le operazioni e in caso di malfunzionamenti o attacchi informatici per esempio, poter capire chi o cosa è stato a creare il problema.
Il log è caratterizzato da un timestamp, ovvero una marca temporale, che indica la data e l'ora precisa dell'evento, un codice di stato, per categorizzare l'evento, e un payload ovvero il corpo del messaggio che contiene informazioni relative all'evento.
I dati presenti in quest'ultimo sono molto utili durante la fase di monitoraggio, è possibile anche aggiungere informazioni relative al contesto per avere una visione più completa dello stato dell'applicazione in un dato momento.
Ci sono diverse tipologie di log, le principali sono le seguenti [20]:

- **Log di accesso:** Un log di accesso registra l'elenco di tutte le richieste di singoli file che le persone o le applicazioni richiedono a un sistema. Include informazioni sull'autenticazione dell'utente, su chi ha richiesto un determinato file di sistema, quando lo ha richiesto e altre informazioni associate.
- **Log di sistema:** Un log di sistema registra gli eventi del sistema operativo, come le modifiche al sistema, i messaggi di avvio, gli errori, gli avvertimenti e gli arresti imprevisti.
- **Log del server:** Si tratta di un file di log che un server crea e mantiene automaticamente. Contiene un elenco di attività svolte dal server, come il numero di richieste di pagine, gli indirizzi IP dei client, i tipi di richieste e così via.
- **Log delle modifiche:** Un log delle modifiche è un file che contiene un registro cronologico delle modifiche apportate al software. Ad esempio, può registrare le modifiche tra le diverse versioni di un'applicazione o le modifiche di configurazione di un sistema.

Sebbene i log siano facili da generare hanno un grande svantaggio, l'elevata complessità di elaborazione. Soprattutto quando riguardano archiviazione e trasferimento dei file di log il sistema può richiedere molte risorse e strumenti specializzati per la gestione efficace dei log.
Nel caso di una infrastruttura a microservizi sarebbe inoltre importante aggregare tutti i log dei singoli container in un unico posto per avere una descrizione completa dell'applicazione.

Pratica in genere molto dispendiosa quando si ha a che fare con applicazioni complesse. [21]

3.2.2 Metriche

Sono statistiche sulle prestazioni, solitamente archiviate e rappresentate come serie temporali. Ad esempio si potrebbero usare per monitorare la latenza media dell'applicazione nel tempo. Le metriche sono dunque molto utili per ottenere una comprensione del comportamento di un sistema sia nel presente che nelle previsioni future. Caratterizzazione fondamentale è che le metriche scalano molto bene, poiché la memorizzazione di una metrica con un valore numerico come l'esempio di prima genera un file che occupa lo stesso spazio sia che siano state elaborate poche richieste al secondo che molte. Questo le rende ideali per creare dashboard che riflettono l'andamento storico del sistema, inoltre con il passare del tempo, è possibile aggregare i dati conservati in frequenze giornaliere o settimanali.
La struttura più diffusa di una metrica è composta da un nome univoco come identificativo e una serie di coppie chiave-valore. I dati conservati nella serie temporale sono definiti campioni e si compongono di due parti: un valore in formato float64 e un timestamp di precisione millisecondi.

Le principali differenze con i logs stanno nella differenza di carico di archiviazione e trasferimento, le metriche infatti comportano un carico costante mentre i logs possono causare un aumento del carico improvviso e significativo con l'aumento del traffico degli utenti o delle attività di sistema. Le metriche raccolte inoltre possono essere più facilmente analizzate e trasformate in rappresentazioni matematiche, statistiche e probabilistiche, particolarmente funzionali per valutare lo stato di un sistema.
Come i log però non sono sufficienti per capire completamente quanto tempo impiega e quale percorso segue una richiesta all'interno di un sistema distribuito. Questa mancanza viene colmata dall'uso del tracciamento, uno strumento che facilita la comprensione delle applicazioni basate su microservizi. [21]

3.2.3 Tracciature

Una traccia distribuita, più comunemente nota come traccia, registra i percorsi intrapresi dalle richieste (effettuate da un'applicazione o da un utente finale) mentre si propagano attraverso architetture multiservizio, come microservizi e applicazioni serverless. L'architettura moderna spesso comprende molteplici componenti indipendenti più piccoli. Questi componenti comunicano costantemente e scambiano dati tramite API per eseguire operazioni complesse. Con il tracciamento distribuito, gli sviluppatori possono tracciare, o seguire visivamente, un percorso di richiesta tra diversi microservizi. Tale visibilità aiuta a risolvere errori o correggere bug e problemi di prestazioni.

Durante l'elaborazione di una richiesta, un'applicazione potrebbe eseguire diverse azioni. Queste azioni sono rappresentate come intervalli (span) nel tracciamento distribuito. Ad esempio, uno span potrebbe essere una chiamata API, l'autenticazione dell'utente o l'abilitazione dell'accesso allo spazio di archiviazione. Se una singola richiesta comporta diverse azioni, l'intervallo iniziale (o principale) può diramarsi in diversi intervalli secondari. Questi livelli nidificati di intervalli padre e figlio formano una rappresentazione logica continua dei passaggi intrapresi per soddisfare la richiesta di servizio.

Il sistema di tracciamento distribuito assegna un ID univoco a ogni richiesta per tracciarla. Ogni span eredita lo stesso trace ID dalla richiesta originale a cui appartiene. Gli span sono inoltre contrassegnati da un ID di span univoco che aiuta il sistema di tracciamento a consolidare i metadati, i log e i parametri che raccoglie.

Man mano che ogni richiesta passa attraverso microservizi diversi, aggiunge parametri che forniscono agli sviluppatori informazioni approfondite e precise sul comportamento del software. Si possono raccogliere dati come il tasso di errore, il timestamp, il tempo di risposta e altri metadati con gli intervalli. Dopo che la traccia ha completato un intero ciclo, lo strumento di tracciamento distribuito consolida tutti i dati raccolti. Questi record vengono di solito registrati in modo asincrono su disco prima di essere inviati separatamente a un collettore centrale (*Figura 3.1*), che è in grado di ricostruire il flusso di esecuzione basandosi su diversi record emessi da diverse parti del sistema. In questo modo sarà possibile comprendere e ricostruire l'intero ciclo di vita della richiesta, il quale risulta fondamentale per il debug dei record che attraversano diversi servizi. [21]

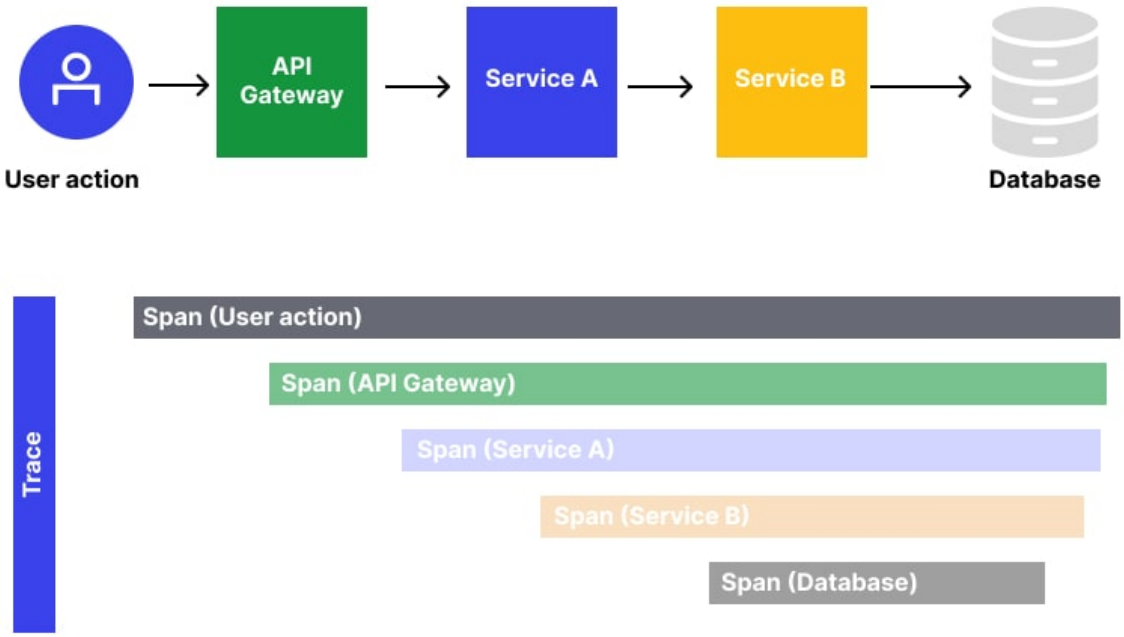

Figura 3.1: Struttura della traccia

Fonte: https://www.wallarm.com/what/what-is-distributed-tracing-full-guide

Il tracciamento distribuito ha semplificato gli sforzi degli sviluppatori nella diagnosi, nel debug e nella risoluzione dei problemi software. Nonostante ciò, rimangono le seguenti sfide a cui i team di software devono prestare attenzione quando scelgono gli strumenti di tracciamento. Per esempio alcuni strumenti di tracciamento richiedono ai team di software di strumentare manualmente i propri codici per generare le tracce necessarie. Quando gli sviluppatori modificano i codici per tracciare le richieste, vi sono rischi di errori di codifica che influiscono sulle release di produzione. Inoltre, la mancanza di automazione complica il tracciamento, con conseguenti ritardi e forse una raccolta di dati imprecisa. [22]

4 Apache Kafka

Verrà ora introdotta e spiegata nel dettaglio la piattaforma distribuita open-source Apache Kafka. Verranno esposte la storia della piattaforma, le caratteristiche principali i concetti che la realizzano, descrivendone le qualità vedremo perché è tra le architetture più usate per il data streaming.

4.1 Definizione e storia di Kafka

LinkedIn sviluppa Kafka nel 2011 come broker di messaggi con un'elevata velocità di trasmissione per uso proprio, quindi lo ha reso open source e lo ha donato alla fondazione Apache Software Foundation. Oggi, Kafka si è evoluto nella piattaforma di streaming più utilizzata, in grado di acquisire ed elaborare milioni di record al giorno senza alcun ritardo percettibile nelle prestazioni, al crescere del volume dei dati. Le organizzazioni Fortune 500, come ad esempio Target, Microsoft, AirBnB e Netflix, si affidano a Kafka per offrire ai propri clienti esperienze basate sui dati e in tempo reale.
La descrizione sul sito ufficiale di apache kafka è la seguente:

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. [23]

In altre parole Kafka è una piattaforma di streaming open source distribuita che consente lo sviluppo di applicazioni in tempo reale basate sugli eventi.
Come descritto nei capitoli precedenti oggi ci sono miliardi di fonti che creano un flusso continuo di dati, questi record devono essere trattati da piattaforme adatte a consentire agli sviluppatori di creare applicazioni che utilizzano ed elaborano continuamente questi flussi a velocità estremamente elevate, con un alto livello di fedeltà e accuratezza, in base al corretto ordine in cui si verificano, Kafka è una di queste piattaforme (*Figura 4.1*).
Kafka viene spesso descritto come un “distributed commit log” o più recentemente come una “distributing streaming platform”. Un filesystem o un database di log è progettato per

fornire una registrazione duratura di tutte le transazioni in modo che possano essere riprodotte per mostrare uno stato consistente del sistema.
Allo stesso modo i dati all'interno di Kafka vengono archiviati in modo duraturo, in ordine, e possono essere letti in modo deterministico. Inoltre, i dati possono essere distribuiti all'interno del sistema per fornire protezioni aggiuntive contro i guasti, nonché per avere l'opportunità di ridimensionare le prestazioni. [24]

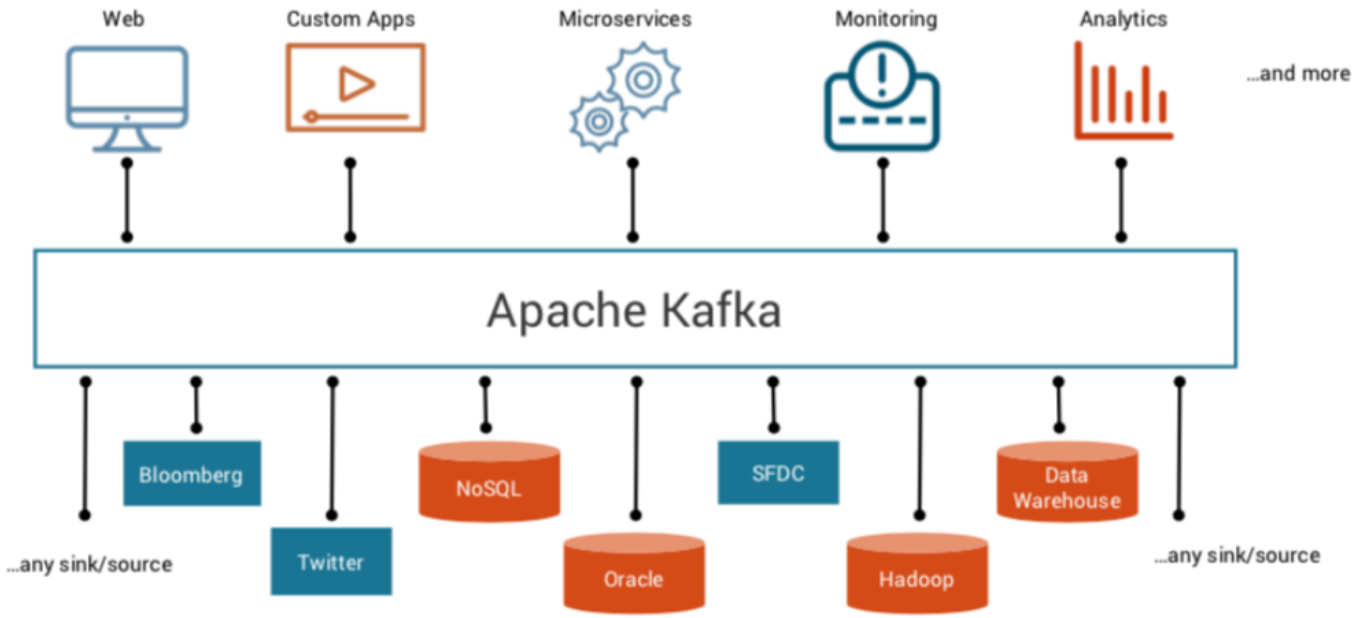

Figura 4.1: Architettura Apache Kafka, sorgenti e destinazioni
Fonte: https://www.linkedin.com/pulse/introduction-streaming-apache-kafka-filipe-balseiro

4.2 Funzionamento

Kafka è definito da tre funzionalità principali [25]:

- Consente alle applicazioni di pubblicare o sottoscrivere flussi di dati o eventi.
- Archivia i record accuratamente (ad esempio, nell'ordine in cui si sono verificati) in modo tollerante agli errori e durevole.
- Elabora i record in tempo reale (mentre si verificano).

Gli sviluppatori possono avvalersi delle precedenti funzionalità mediante quattro api:

API Producer. consente ad un'applicazione di pubblicare un flusso in un *argomento* Kafka. Un argomento è un log denominato che memorizza i record nell'ordine in cui si sono verificati l'uno rispetto all'altro. Dopo essere stato scritto in un argomento, un record non può essere modificato o eliminato; rimane nell'argomento per un periodo di tempo preconfigurato, ad esempio per due giorni, o fino all'esaurimento dello spazio di storage.

API Consumer. consente a un'applicazione di sottoscrivere uno o più argomenti e di inserire ed elaborare il flusso memorizzato nell'argomento. Può funzionare con i record nell'argomento in tempo reale o può inserire ed elaborare i record antecedenti.

API Streams. è progettata sulla base delle API Consumer e Producer e aggiunge funzionalità complesse di elaborazione che consentono a un'applicazione di eseguire un'elaborazione continua e di front-to-back del flusso, nello specifico, utilizzare i record di uno o più argomenti per analizzarli, aggregarli o trasformarli come richiesto e pubblicare i flussi che ne risultano negli stessi argomenti o in argomenti differenti. Mentre le API Producer e Consumer possono essere utilizzate per l'elaborazione semplice dei flussi, è l'API Streams che consente lo sviluppo delle più avanzate applicazioni di streaming di dati ed eventi.

API Connector. consente agli sviluppatori di creare connettori che rappresentano producer e consumer riutilizzabili in grado di semplificare e automatizzare l'integrazione di una fonte di dati in un cluster Kafka.

4.2.1 Messaggi e Gruppi

Il singolo dato in Kafka è detto *messaggio*, è semplicemente un array di byte quindi non necessita di una formattazione particolare. Ogni messaggio può anche possedere dei metadati, chiamati *key,* anch'essi formati da un array di byte. Le keys sono utili quando ci si vuole assicurare che i messaggi con gli stessi metadati vengano registrati nelle stesse partizioni.[26]

Per una questione di efficienza i messaggi sono scritti dentro Kafka in gruppi, chiamati *batches*. Un batch è una collezione di messaggi accomunati dall'essere registrati tutti nella stessa partizione della stessa *topic* (la definizione verrà fornita più avanti). Infatti mandare singolarmente ogni messaggio causerebbe uno spreco eccessivo di risorse e per risolverlo si è optato per questa soluzione. Il tutto deve essere ovviamente bilanciato per non avere raggruppamenti di messaggi troppo grandi che aumenterebbero il tempo di propagazione. Per questo motivo si ricorre ad una compressione delle batch per provvedere ad un trasferimento e una memorizzazione più efficiente. [27]

4.2.2 Topics e Partizioni

I messaggi in Kafka vengono categorizzati all'interno di *topics.* Per chiarire le idee si può immaginare la topic come se fosse una tabella in un database oppure una cartella in un file system. Le topic sono ulteriormente suddivise in *partizioni.*

I messaggi possono essere scritti nella topic soltanto in "append" e letti in ordine dal primo all'ultimo. È necessario notare che ogni topic normalmente è suddivisa in diverse partizioni e questo non permette di garantire l'ordine temporale dei messaggi nell'intera topic ma soltanto nelle singole partizioni. Nella *Figura 4.1* è rappresentata una topic frazionata in quattro partizioni, ogni nuovo messaggio scritto viene registrato alla fine di ognuna di esse. [28] Le partizioni sono fondamentali perché è il modo che Kafka usa per provvedere a ridondanza e scalabilità. Infatti ognuna di esse può essere mantenuta su server separati, dunque ogni singola topic può essere espansa su differenti server per garantire prestazioni molto al di là di quelle di un singolo server. [27]

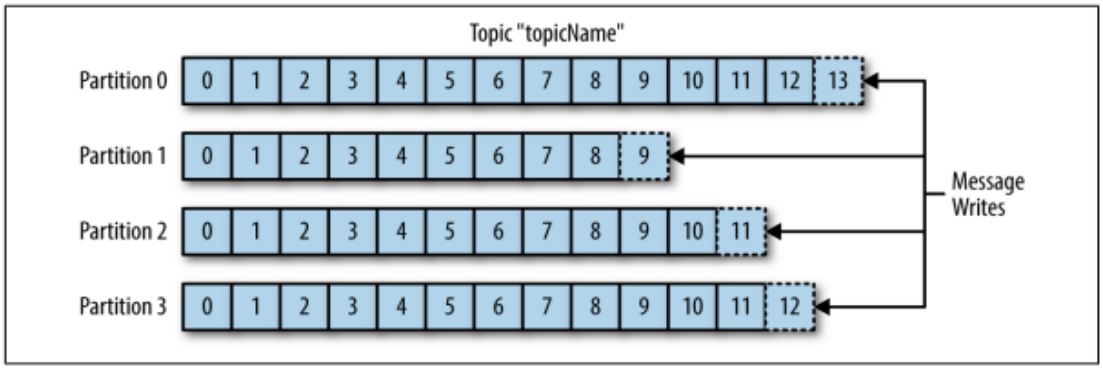

Figura 4.1: Rappresentazione di una topic con partizioni multiple.

Fonte: Neha Narkhede, Gwen Shapira, Todd Palino, 'Kafka the definitive guide', 2017

4.2.3 Producers e Consumers

I “Kafka clients” sono gli utenti del sistema, quelli base sono di due tipi: producers e consumers. Utenti più avanzati possono essere creati unendo quelli base per sviluppare funzionalità di alto livello. [29]
I Producers creano nuovi messaggi per una topic, di default al producer non interessa in quale partizione specifica un messaggio è scritto dunque cercherà di bilanciarli su tutte le partizioni. In alcuni casi invece il producer può usare la key del messaggio per scegliere arbitrariamente di inviarlo a una partizione a sua scelta.

Il Consumer invece si iscrive e legge i messaggi di una o più topics nell’ordine in cui sono prodotti. Quest’ultimo tiene anche traccia del numero di offset dell’ultimo messaggio consumato, univoco e creato quando viene prodotto, per evitare di rileggere l’intera partizione. I consumatori cooperano come parte di un “consumer group” e insieme lavorano per leggere dalla topic. Questo permette di assicurarsi che ogni partizione venga letta da un solo consumatore. Come mostrato nella *Figura 4.2* ci sono tre consumatori, facenti parte dello stesso gruppo, che leggono da una topic. In questo esempio due dei consumatori rappresentati leggono da una singola partizione mentre il terzo da due contemporaneamente. In questa maniera i consumer possono scalare orizzontalmente per prelevare dati da una topic con un gran numero di messaggi. Inoltre se un singolo fallisce i membri rimanenti possono ribilanciare la lettura delle partizioni per compensare la perdita. [27]

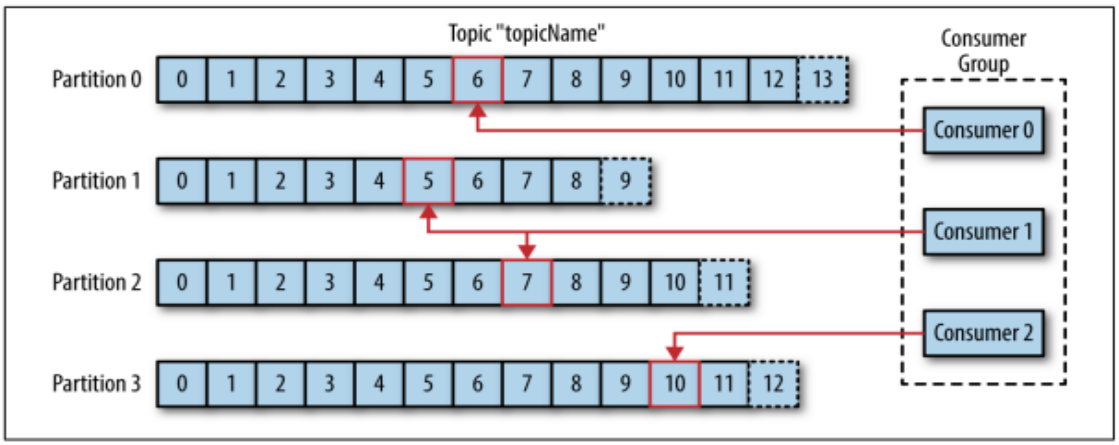

Figura 4.2: Un gruppo di consumatori leggono da una topic.
Fonte: Neha Narkhede, Gwen Shapira, Todd Palino, 'Kafka the definitive guide', 2017

4.2.4 Brokers e Clusters

Un singolo server di Kafka è detto *broker*. Il broker riceve i messaggi dai produttori, gli assegna un offset e li scrive in memoria. Si occupa anche di servire i consumatori, essi gli inviano le richieste per le partizioni e lui risponde col messaggio che è stato salvato su disco. Un singolo broker può arrivare facilmente a gestire migliaia di partizioni e milioni di messaggi al secondo.

Caratteristica fondamentale dei broker è che sono stati pensati e sviluppati per operare come parte di un insieme detto *cluster*. Tra i broker facenti parte di questo cluster verrà eletto un broker che funzionerà da controller. Quest'ultimo è incaricato di gestire le operazioni amministrative, assegnare le partizioni ai broker e monitorarne il corretto funzionamento.

Una partizione è posseduta solo da un broker nel cluster ed esso prende il nome di *leader* della partizione, tutti i produttori e consumatori che operano su tale partizione devono connettersi a quest'ultimo. Una singola partizione può anche essere replicata su più broker per garantire ridondanza e quindi maggiore sicurezza in caso di guasto del leader (*Figura 4.3*). [30]

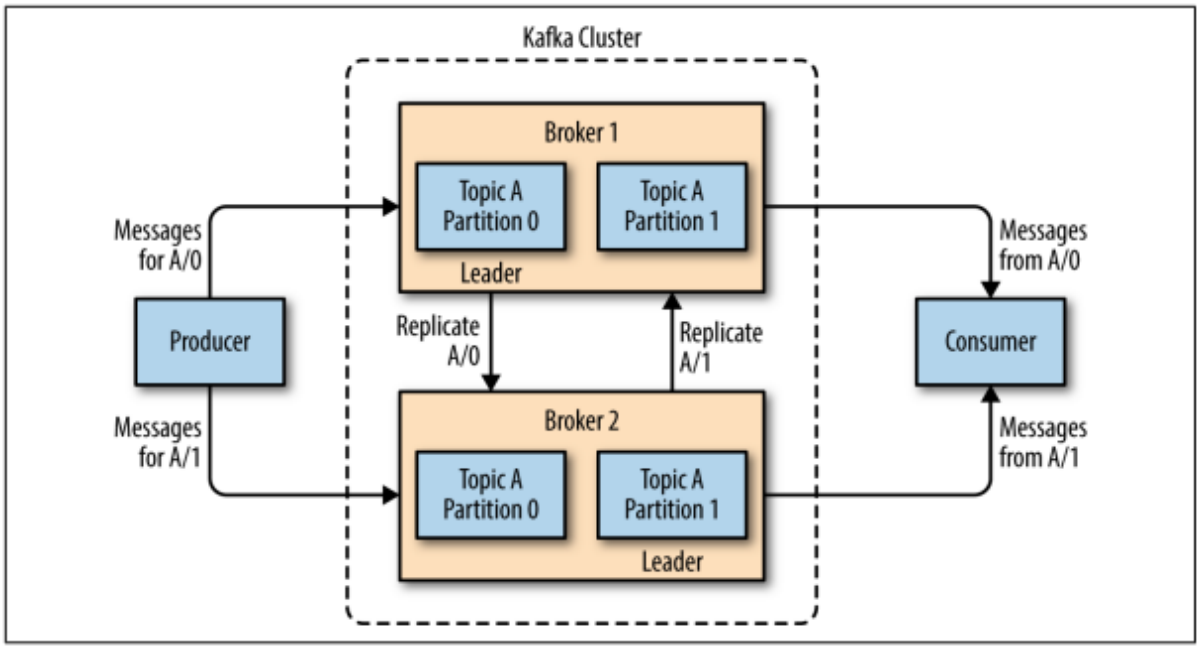

Figura 4.3: Fattore di replica in un cluster

Fonte: Neha Narkhede, Gwen Shapira, Todd Palino, 'Kafka the definitive guide', 2017

Funzionalità chiave di Apache Kafka è la memorizzazione dei messaggi per un certo periodo di tempo. La configurazione a default dei broker per le topic può essere: mantenere i messaggi per un periodo di tempo (es. 7 giorni) o fino a che la topic non raggiunge una certa dimensione (es. 1GB). Una volta raggiunti questi limiti i messaggi vengono eliminati. [27]
Ad ogni topic possono essere assegnate le configurazioni più adatte ad essa così che i messaggi siano memorizzati per tutto il tempo in cui possano essere utili. Per esempio, delle informazioni di tracking devono essere memorizzate per determinati giorni mentre delle metriche relative ad applicazioni possono essere salvate anche solo per qualche ora.

La scelta di appoggiarsi ad una infrastruttura quale Apache Kafka comporta numerosi vantaggi ai quali diventa difficile rinunciare se l'intenzione è quella dell'analisi di una quantità vasta di dati che devono essere prodotti e consumati in continuazione.
Il primo vantaggio è sicuramente quello della possibilità di avere produttori multipli, questi possono scrivere su topic differenti ma anche sulla stessa, questo rende il sistema ideale per aggregare dati da sistemi differenti e renderli consistenti.
Identica cosa si ha con i consumatori, Kafka è progettato per avere tanti consumatori che leggono ogni stream di messaggi senza interferire tra di loro. Come spiegato in precedenza inoltre più consumatori possono aggregarsi e condividere una topic distribuendo così il carico del lavoro.
Altro fattore fondamentale, già ampiamente descritto, è la memorizzazione dei dati. In questo modo i consumatori non devono necessariamente essere sempre attivi quando si hanno dei dati da leggere. I messaggi salvati su disco possono essere conservati in base a configurazioni personalizzate per ogni singola topic. Questo garantisce che anche in caso di malfunzionamento oppure di manutenzione sui consumer i dati mandati dai producer non vengano persi. [31]

La natura dell'ambiente in cui Kafka si trova ad operare rende necessario l'abilità di poter scalare in modo semplice e veloce a fronte di picchi improvvisi di traffico dati. Un'azienda che ha appena iniziato il proprio business vorrà partire con un numero limitato di broker, insieme all'azienda poi crescerà anche il carico di dati a cui è sottoposta e grazie a Kafka sarà immediato poter aumentare i broker attivi per poter gestire traffici maggiori.

L'ampliamento dei broker è fattibile senza problemi anche se il servizio è online, quindi la disponibilità non ne risente.

4.3 Conclusioni

Tutte queste funzionalità si uniscono per rendere Apache Kafka un sistema di publish/subscribe di stream di messaggi caratterizzato da prestazioni eccellenti con grandi carichi di lavoro. Tutti gli attori quali produttori, consumatori e broker possono essere scalati facilmente per gestire grandi quantità di messaggi. Tutto questo garantendo una latenza di scrittura/lettura sotto al secondo.

5 OpenTelemety

Nel mondo del cloud computing, l'osservabilità svolge un ruolo chiave nell'assicurare che le operazioni si svolgano in modo efficiente e sicuro, le organizzazioni di solito la implementano utilizzando una varietà di metodi di strumentazione tra cui l'impiego di strumenti open source come ad esempio OpenTelemetry, permettendo di correlare e analizzare i dati.
OpenTelemetry nasce ufficialmente a seguito della fusione di due progetti precedenti, OpenTracing e OpenCensus. Entrambi erano stati creati per risolvere il medesimo problema: la mancanza di standard su come strumentalizzare il codice e inviare le telemetrie ad un backend per una più immediata analisi.
Nessuno dei due progetti però riusciva a risolvere i vari problemi così si è optato per unirli e trarre i vantaggi di ognuno per offrire un singolo standard solido.

OpenTelemetry è un framework per l'osservabilità, un kit di strumenti progettati per creare e gestire dati sulla telemetria come le già citate tracce, metriche e logs. OpenTelemetry non si occupa di backend, infatti è agnostico verso gli strumenti scelti per organizzare e visualizzare i dati estratti dalle proprie applicazioni (es. Jaeger, Prometheus, …). Esso invece si concentra sulla generazione, raccolta, gestione e invio delle telemetrie, inoltre la strumentalizzazione è resa molto semplice perchè non è legata ad un linguaggio, ad una infrastruttura o a un ambiente di runtime preciso.
OpenTelemetry è dunque una collezione standard di strumenti, API e SDK (*Figura 5.1*) col solo scopo di permettere agli sviluppatori la comprensione delle performance e del comportamento delle loro applicazioni cloud, questo significa che possono spendere meno tempo per sviluppare meccanismi di raccolta dati e usare più tempo per creare nuove funzionalità. [32]

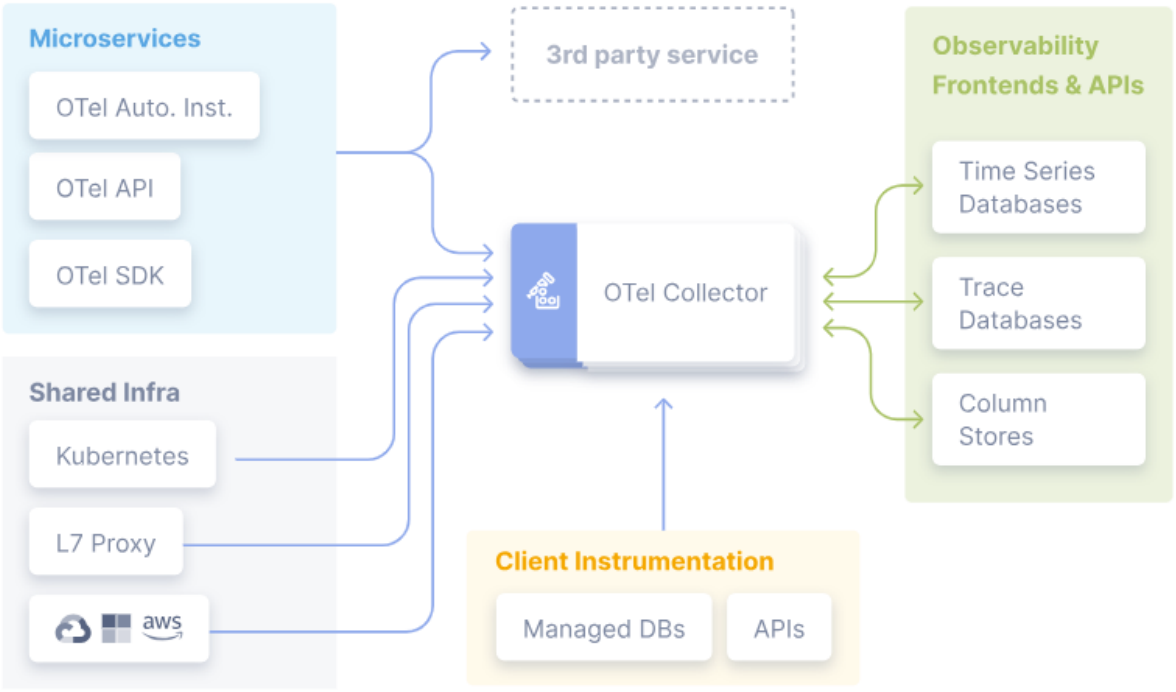

Figura 5.1: Architettura di riferimento di OpenTelemetry

Fonte: https://opentelemetry.io/docs/

5.1 Funzionamento

OpenTelemetry è un protocollo specializzato per collezionare dati telemetrici ed esportarli verso un sistema esterno. Il ciclo di vita di questi dati ha molteplici step dalla loro creazione al loro invio e in seguito descriverò questi passaggi [33]:

- Strumentalizzare il proprio codice con delle API, specificare ai componenti del sistema quale metriche deve raccogliere e come farlo.
- Raggruppare i dati usando gli SDK e trasportarli per il processamento e l'asporto.
- Scomporre i dati, campionarli, filtrarli per ridurre errori e arricchirli usando con i vari contesti di esecuzione.
- Convertire ed esportare i dati.
- Vengono infine eseguiti ulteriori filtraggi in gruppi di messaggi basati sul tempo. Successivamente vengono inviati ad un backend predeterminato.

L'*Ingestion* è un processo critico per raggruppare i dati che per noi sono fondamentali. Ci sono due modalità principali [33]:

- **Local Ingestion:** Avviene una volta che i dati sono memorizzati nella cache locale. Più comune nei sistemi in esecuzione in locale o in ambienti ibridi, dove i dati di serie temporale vengono poi trasmessi al cloud. I database nel cloud sono ottimi per la memorizzazione di grandi quantità di dati per una successiva analisi, questi dati inoltre possono avere delle restrizioni legate alla privacy oppure un'importanza legata al business.
- **Span Ingestion:** Possiamo anche raccogliere i dati delle tracce con un formato span. Quest'ultime sono spesso indicizzate e consistono sia in span di root sia span figlie. Sono dati molto importanti perché contengono, tra le tante cose, metadati e informazioni legate agli eventi.

5.2 Componenti principali

Come vediamo nella *Figura 5.2* successiva OpenTelemetry è composta da diversi componenti: [34]

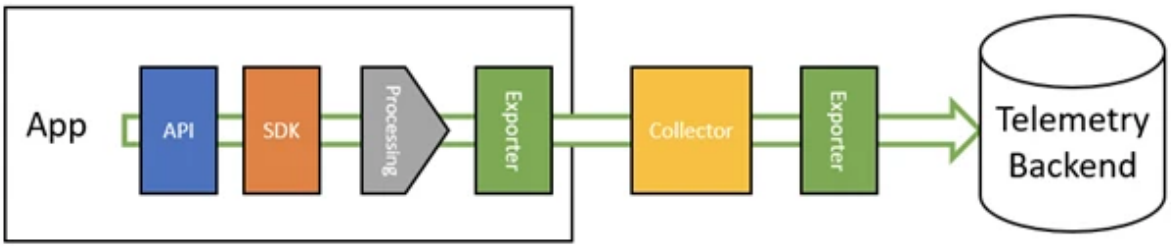

Figura 5.2: Componenti di OpenTelemetry

Fonte: https://www.dynatrace.com/news/blog/what-is-opentelemetry-2/

API. Componenti chiave e specifici di un linguaggio specifico (come Java, Python, .Net, …). Le API definiscono il come si generano i dati telemetrici.

SDK. Anch'essi specifici di un linguaggio. Sono il ponte che collega le API a l'exporter, è anche possibile aggiungere configurazioni ulteriori come un filtraggio delle richieste e campionamenti delle transazioni.

Collector. Esso riceve, processa ed esporta i dati telemetrici, nonostante non sia obbligatoriamente richiesto, garantisce un grande flessibilità all'architettura di OpenTelemetry per quanto riguarda la ricezione e l'invio al backend delle telemetrie.
Il collector ha due tipi di modelli di deployment [35]:

- Come agente che risiede sullo stesso host dell'applicazione, come processo separato.
- Come un collettore remoto completamente separato, che può risiedere in un contenitore o macchina virtuale, ricevendo dati telemetrici da ciascun agente ed esportandoli nei sistemi backend.

Exporter. Consente di configurare su quale sistema di backend si vuole che i dati vengano inviati. Serve principalmente a disaccoppiare l'instrumentazione dalle configurazioni di backend, questo semplifica cambiare backend senza reinstrumentare diversamente il codice. L'interfaccia dell'exporter, implementata dagli SDK OpenTelemetry, utilizza un modello plug-in che traduce i dati telemetrici nel formato richiesto per un particolare sistema backend prima di trasmetterli. Questo modello supporta anche la composizione e il concatenamento degli exporter, facilitando la condivisione delle funzionalità tra diversi protocolli.

5.3 Compatibilità ed Estensibilità

Compatibilità ed estensibilità sono due caratteristiche chiave dell'infrastruttura di OpenTelemetry, il primo garantisce una grande varietà di integrazione fra ecosistemi differenti, inoltre OpenTelemetry è usata da un gran numero di organizzazioni che la implementano direttamente provvedendo così un supporto commerciale ed un contributo al progetto.
L'estensibilità è un altro concetto chiave nell'infrastruttura, ciò garantisce una vasta modalità di estensione disponibile ad OpenTelemetry, come per esempio: aggiungere un ricevitore al collector di OpenTelemetry così da supportare dati telemetrici da una fonte personalizzata, caricare instrumentazioni personalizzate nell'SDK, distribuire un SDK o un Collector su misura ad un caso d'uso specifico.

Nonostante queste modifiche nella maggior parte delle volte non siano necessarie OpenTelemetry è stato progettato per renderle disponibili ed immediate.[36]

5.4 Vantaggi di OpenTelemetry

Con l'avvento del cloud computing, delle architetture di microservizi e di requisiti aziendali sempre più complessi, la necessità di osservabilità del software e delle infrastrutture è più grande che mai.
OpenTelemetry soddisfa l'esigenza di osservabilità seguendo due principi chiave:

1. I dati generati sono unicamente di proprietà dell'utilizzatore.
2. L'utilizzatore deve solamente imparare un singolo set di API e convenzioni.

La combinazione di entrambi i principi garantisce ai team e alle organizzazioni la flessibilità di cui hanno bisogno nel moderno mondo informatico di oggi.

5.5 Apache Kafka con OpenTelemetry

Come abbiamo già spiegato in precedenza al giorno d'oggi Apache Kafka viene utilizzato come sistema nervoso in un ambiente distribuito da un vasto numero di organizzazioni, kafka permette a servizi differenti di comunicare attraverso lo scambio di messaggi, eventi o come piattaforma di streaming.
Data la natura distribuita di Kafka però diventa difficile tracciare come i messaggi si muovono nell'infrastruttura. È qui che entra in gioco OpenTelemetry, provvedendo diverse librerie di instrumentazione per aggiungere una tracciatura ai messaggi scambiati dai Kafka clients.[37]

5.5.1 Tracciatura dei Kafka Clients

Prendiamo come esempio una semplice applicazione dove due Kafka clients producono e consumano messaggi. Siamo interessati ad aggiungere la tracciatura solamente per le parti relative a Kafka e non alla logica di business dell'applicazione, ovvero vogliamo solo le informazioni relative a lettura e scrittura su kafka.
Ci sono due modi diversi per raggiungere questo obiettivo [38].

Il primo è quello di usare un agente esterno, parallelo all'applicazione, che garantisca la raccolta di tutte le informazioni sul tracciamento.
Il secondo è quello di abilitare il tracciamento direttamente sui client di Kafka dell'applicazione.

Il primo metodo è quello più tradizionale ed è definito come un approccio automatico. Infatti l'applicazione non viene minimamente modificata, l'agente avviato insieme all'applicazione si occupa di intercettare i messaggi mandati e ricevuti e ne aggiunge le informazioni di tracciamento.
Il secondo invece è la modalità più recente ed è un approccio più manuale. L'applicazione infatti deve essere instrumentata direttamente, ovvero devono essere aggiunte al progetto specifiche dipendenze e fare dei cambiamenti nel codice.

5.5.1.1 Tracciamento tramite agente

Si tratta dell'approccio più semplice e automatico, non è necessario nessun cambiamento al codice dell'applicazione e non serve nessuna dipendenza specifica alle librerie di OpenTelemetry.
Per farlo è necessario usare un "OpenTelemetry Agent" che viene avviato insieme all'applicazione così da poter iniettare la logica di tracciamento ai messaggi intercettati, da o per un cluster Kafka. [39]

5.5.1.2 Instrumentazione dei Client Kafka

Per prima cosa è necessario che all'applicazione vengano aggiunte le dipendenze ai moduli che provvedono alla tracciatura dei clienti kafka. Successivamente si importa la dipendenza dall'exporter che si intende utilizzare.
Il passo successivo è quello di creare e impostare un'istanza di OpenTelemetry nel codice dell'applicazione che gestirà l'intera instrumentazione. Questa, una volta creata, deve essere registrata globalmente così da poter essere resa disponibile alle librerie dei client.
Abbiamo due modi per fare ciò: il primo consiste nell'usare un'estensione dell'SDK per una configurazione automatica basata sull'ambiente dell'applicazione. L'alternativa è usare

delle classi di costruttori dell'SDK per creare e quindi configurare l'istanza di OpenTelemetry in modo manuale e personalizzato.

Se si decide di procedere con la configurazione manuale allora è necessario importare delle dipendenze che permettono di usare le estensioni dell'SDK relative all'auto configurazione. Quando queste librerie vengono usate per prima cosa controllano se esiste un'istanza di OpenTelemetry già creata e registrata, in caso negativo inizializza l'istanza automaticamente.
Questa configurazione però non è del tutto automatica, si basa infatti su alcune variabili di ambiente che possono essere modificate per personalizzare l'istanza.
Le principali sono:

- "OTEL_SERVICE_NAME": permette di specificare il nome logico del servizio, torna utile quando si utilizzano delle interfacce utente per mostrare i dati raccolti (ad esempio Jaeger UI).
- "OTEL_TRCES_EXPORTER": specifica la lista degli exporter utilizzati. Se per esempio si vuole utilizzare Jaeger come backend è necessario avere anche la corrispondente dipendenza nell'applicazione.

Alternativa a questa configurazione automatica è per l'appunto quella manuale. In questo modo l'istanza di OpenTelemetry è costruita completamente da zero attraverso i costruttori dell'SDK ed è completamente configurabile.
Per poter utilizzare i costruttori in questione è necessario importare le relative dipendenze.
Il codice seguente (*Figura 5.3*) mostra come viene creata l'istanza e in che modo gli attributi fondamentali, come il nome del servizio e l'exporter scelto, vengono impostati.

```
Resource resource = Resource.getDefault()
.merge(Resource.create(Attributes.of(ResourceAttributes.SERVICE_NAME, "my-kafka-service")));

SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
.addSpanProcessor(BatchSpanProcessor.builder(JaegerGrpcSpanExporter.builder().build()).build())
        .setSampler(Sampler.alwaysOn())
        .setResource(resource)
        .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
.setTracerProvider(sdkTracerProvider)
.setPropagators(ContextPropagators.create(W3CTraceContextPropagator.getInstance()))
.buildAndRegisterGlobal();
```

Figura 5.3: Creazione istanza di OpenTelemetry

Fonte: https://opentelemetry.io/blog/2022/instrument-kafka-clients/

Una volta creata e configurata l'istanza di OpenTelemetry è necessario avere un modo per campionare i dati scambiati all'interno dell'infrastruttura dai client. Se ne occupano le API dei clienti Kafka, esse provvedono a di intercettare i messaggi, prima che essi siano inviati ai broker o appena sono ricevuti da parte degli stessi. Questo approccio è molto usato quando vi è la necessità di applicare una logica o un contenuto al messaggio prima che esso venga inviato, allo stesso tempo è utile per gestire dei messaggi consumati poco prima che vengano passati ai livelli più alti dell'applicazione. Si tratta di un modo molto utile nel tracciamento quando si vuole creare o chiudere una span su messaggi che devono essere mandati e ricevuti.

La libreria di instrumentazione per kafka prevede due tipi di *interceptor* che sono configurati per aggiungere automaticamente le informazioni di tracciamento. Questi interceptor devono essere impostati tra le proprietà dei client Kafka, uno per i produttori ed uno per i consumatori.

Un altro modo, in alternativa agli interceptor, è quello di usare i *wrapper*, ovvero ricoprire l'istanza di un Producer o un Consumer in modo che i messaggi prodotti o consumati vengano intercettati. [38]

5.6.2 Instrumentazione in azione

Completati i precedenti passaggi per configurare l'istanza di OpenTelemetry, scelto l'exporter desiderato e infine instrumentato i produttori e i consumatori con gli interceptor o i wrapper allora possiamo procedere a testare se l'osservabilità dell'architettura Kafka funziona.

Per prima cosa quindi facciamo partire il Kafka cluster avviando lo zookeeper e i vari brokers. Successivamente avviamo i produttori e consumatori della relativa infrastruttura e lasciamo che si scambino dei messaggi. Le tracce contenenti le relative span di questi messaggi scambiati verranno intercettati, campionati, collezionati e infine inviati al servizio di backend, per esempio Jaeger. Una volta aperto Jaeger potremo vedere le tracce dei messaggi prodotti e consumati come in *Figura 5.3*.

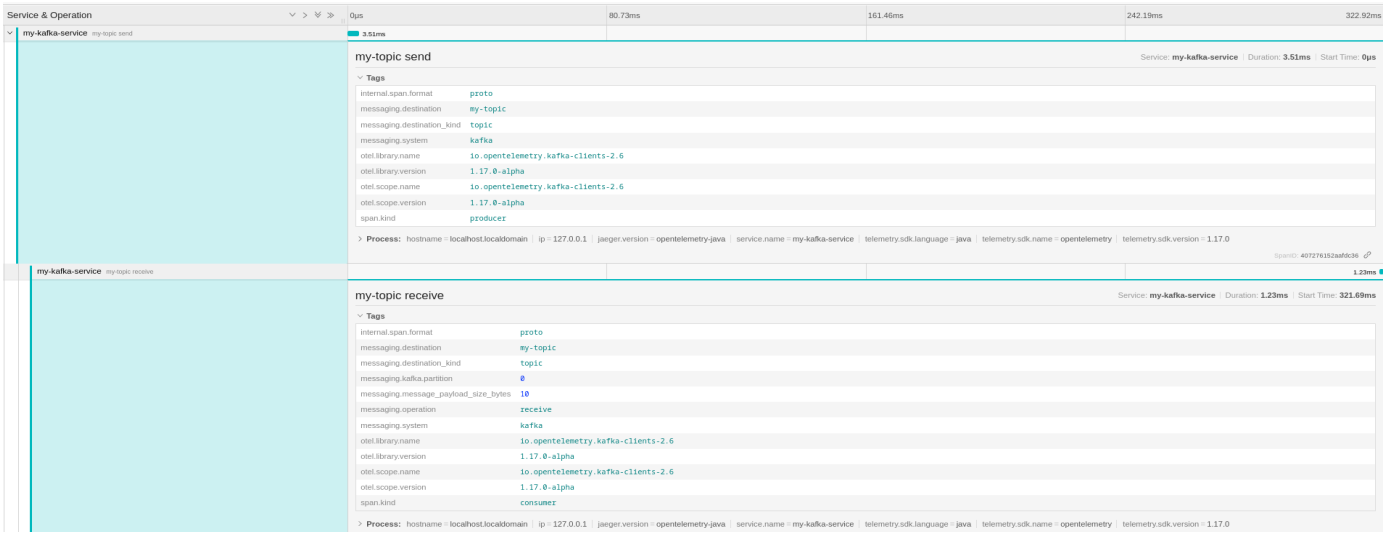

Figura 5.3

Fonte:https://opentelemetry.io/blog/2022/instrument-kafka-clients/#setting-up-the-opentelemetry-instance

In questo caso il messaggio mandato e ricevuto fanno parte della stessa traccia e quello ricevuto è identificato come "child_of" di quello mandato. Possiamo anche notare come queste tracce siano arricchite di dettagli riguardanti la telemetria e i relativi tags che la caratterizzano. [38]

6 Sviluppo di una piattaforma per l'osservabilità di Apache Kafka

Dopo aver introdotto i concetti e le tecnologie principali che vedremo utilizzati nel progetto andremo ad analizzare gli obiettivi di quest'ultimo.
Abbiamo visto ciò che rende Apache Kafka un servizio così veloce ed ottimo per scalare la quantità di traffico gestita senza intaccare la velocità con cui i dati vengono smistati, scritti e letti. Infatti gestire le varie topic di un cluster spartendole fra broker per la gestione del carico, dividendole in partizioni per garantire operazioni in parallelo e replicandole in caso di malfunzionamenti garantisce una latenza delle operazioni sotto al secondo.
Anche se Apache Kafka fornisce alcune ottimizzazioni predefinite [40], non definisce rigorosamente il modo in cui ciascuna topic deve essere distribuita in modo efficiente nelle partizioni. È ancora un problema di ricerca aperto quello di mettere a punto una modalità ben formulata per migliorare le prestazioni di un cluster Kafka.

6.1 Obiettivo

L'obiettivo di questo progetto è proprio quello di porre l'attenzione sul numero di broker attivi in un cluster e il numero di partizioni associate a ciascuna topic, verranno provate più configurazioni dell'infrastruttura e, se possibile, raggiungerne una più conveniente delle altre in termini di efficienza.
Vederemo innanzitutto un insieme di topic come esempio base di infrastruttura e descrivendo in che modo i messaggi vengono prodotti, letti e scambiati fra esse.
Successivamente descriveremo nel dettaglio i producer e consumer che sono stati creati appositamente per i test, vedremo in che modo è stata integrata la telemetria di OpenTelemetry, saranno descritti i due tipi di Operator posti a gestire l'infrastruttura e infine parleremo di tutte le possibili configurazioni, con broker e partizioni come dati variabili, che verranno testate.

6.2 Infrastruttura proposta

Per la realizzazione dell'obiettivo è stato necessario creare un insieme di topic per simulare diversi possibili percorsi che i dati scambiati tra produttori e consumatori avrebbero dovuto attraversare (*Figura 6.1*).

L'architettura è stata formata quindi da **cinque topic** differenti. Su quest'ultime interagiscono **due produttori** e **cinque consumatori**, tre di questi svolgeranno anche il ruolo di produttori propagando i messaggi letti attraverso topic predefinite.

Come vediamo in *Figura 6.1* ci sono due percorsi possibili ai messaggi, come prima cosa sia il produttore **P1** che **P2** scrivono 1000 messaggi a testa sulla prima topic, lì verranno divisi: il primo gruppo viene letto dal primo consumatore e successivamente verrà scritto sulla topic 2. Il secondo gruppo subirà la stessa sorte ma invece di essere mandato sulla seconda topic sarà scritto sulla topic 3 dove verrà consumato e scritto sulla topic 4. Una volta letto dalla topic 4 il gruppo intero sarà inviato nuovamente alla topic 3 e contemporaneamente sulla topic 5 dove i messaggi termineranno il loro percorso una volta letti dall'ultimo consumatore.

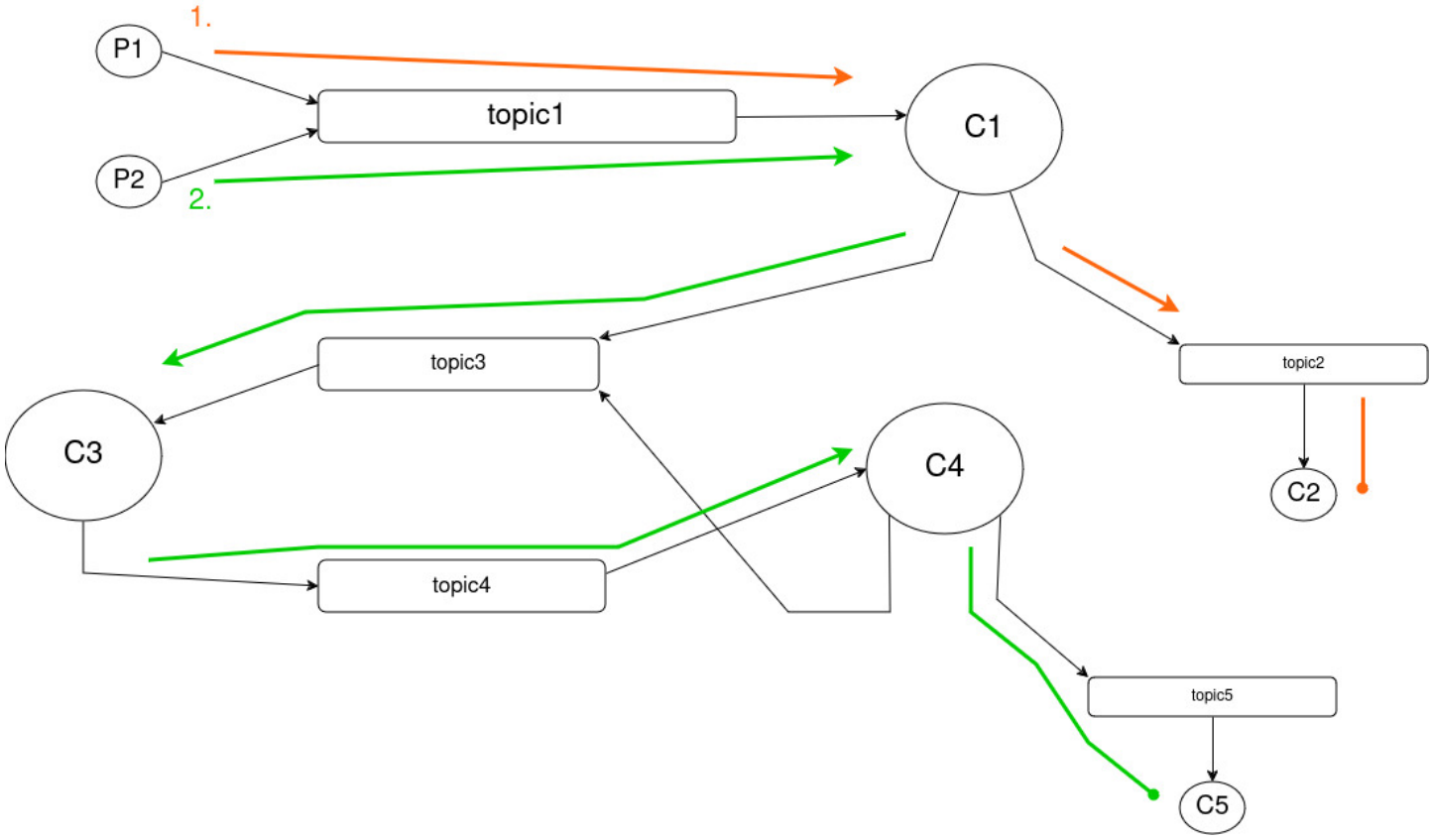

Figura 6.1

Come prima cosa viene costruita l'infrastruttura Kafka: viene avviato lo zookeeper e i broker. Lo step successivo è quello di avviare dei **Topic Operator** scritti apposta per creare le topic, sceglierne il nome, decidere da quante partizioni esse siano composte, impostare il fattore di replica desiderato e impostare le configurazioni per la topic. Questi Operator in seguito hanno anche la possibilità di aumentare il numero di partizioni per una topic, modificare il fattore di replica e anche cambiare le impostazioni della topic. Una volta che le topic sono state create correttamente sia i consumatori che i produttori principali verranno avviati contemporaneamente in modo da simulare un'esecuzione normale del servizio a runtime.

6.3 Produttori e Consumatori

Come appena descritto, i client del cluster quali produttori e consumatori sono avviati in parallelo. Appena avviati entrambi settano le proprietà necessarie per interagire con l'ambiente kafka come ad esempio l'indirizzo del broker (*Figura 6.2*) di riferimento e la porta dalla quale fornisce il servizio, grazie al quale possono interfacciarsi al sistema.

```
properties.put("bootstrap.servers", "192.168.178.119:9092");
```

Figura 6.2: Impostazione indirizzo broker

Successivamente il produttore inizia subito a trasmettere i messaggi sulla topic (*Figura 6.3*)

```
ProducerRecord<String, String> record = new ProducerRecord<>(topic, message);
producer.send(record);
```

Figura 6.3: Scrittura messaggi sulla topic

Il broker di riferimento riceve i dati e li smista sulle partizioni di cui è leader altrimenti li passa al broker incaricato. Una volta completata la scrittura il produttore si chiude.

Anche il consumatore si interfaccia con il broker assegnato, in più viene settato il gruppo di consumatori al quale fa parte, successivamente si iscrive ad una topic e attende i messaggi (*Figura 6.4*).

```
ConsumerRecords<String, String> records = consumer.poll(1000);
```

Figura 6.4: Lettura messaggi dalla topic

Quando i messaggi vengono letti dal consumatore, nel caso sia uno degli intermediari fra topic, si occupa anche di propagare i messaggi alla topic successiva assegnata e successivamente termina.
Nel nostro caso della *Figura 6.1* il produttore **C1** possiede anche la logica per discriminare i messaggi appartenenti ai due gruppi differenti e poi li invia alle corrispondenti topic.

6.4 Integrazione di OpenTelemetry

Dopo aver completato la scrittura dei produttori e dei consumatori, si è proceduto con l'instrumentazione degli stessi utilizzando OpenTelemetry. Il primo passo ha previsto l'importazione nel progetto di tutte le dipendenze necessarie. Successivamente, è stata creata e configurata un'istanza di OpenTelemetry. È stata scelta una configurazione manuale tramite i builder SDK, che ha consentito di specificare non solo il nome del servizio, ma anche l'indirizzo del backend a cui inviare le telemetrie, in questo caso a Jaeger. Infine, l'istanza è stata registrata globalmente.
In seguito, si è deciso di aggiungere degli interceptor ai client di Kafka per intercettare i messaggi scambiati e arricchirli con informazioni utili al tracciamento. Infine, nel codice è stato implementato l'avvio delle span, ogni produttore ne avvia una per ogni messaggio inviato e per ognuna si occupa di propagare il contesto di tracciamento così che i consumatori possano riconoscerle ed aggiungerci a loro volta informazioni. Queste ultime appartengono alle tracce ottenute dall'istanza di OpenTelemetry globale precedentemente descritta.

Quando le span vengono fermate OpenTelemetry le invia automaticamente a Jaeger, da cui è possibile accedervi tramite un'interfaccia utente o scaricare i dati registrati in formato JSON per un'analisi autonoma.

6.5 Topic Operator

Come già accennato in precedenza le topic sono inizializzate ed in seguito gestite e monitorate dagli operator. Essi sono delle entità che eseguono parallelamente al ciclo di vita della topic e ne gestiscono gli aspetti principali. Questi infatti possono non solo avviare la topic con il nome desiderato, il numero di partizioni volute e le impostazioni scelte ma anche monitorarne lo stato in tempo reale e fare variazioni a runtime. Gli operator una volta avviati, come anche i client del cluster, si collegano al broker di riferimento specificandone indirizzo e porta, in seguito creano la topic con tutte le informazioni desiderate. Le possibili variazioni che possono fare durante la normale esecuzione del cluster Kafka sono:

- Visualizzare e se desiderato aumentare il numero di partizioni di cui la topic è composta.
- Modificare le configurazioni attuali della topic. Queste configurazioni riguardano la *cleanup policy*, ovvero il tempo massimo durante il quale i messaggi possono rimanere nel topic, la dimensione massima occupata dai messaggi e il modo in cui gestirli una volta raggiunto uno dei due limiti. I messaggi possono essere eliminati oppure compattati.
- Ottenere informazioni sui client della topic: tutte le informazioni riguardanti consumatori come l'id, il gruppo in cui fanno parte, l'host, la partizione a cui sono assegnati e così via. Stessa cosa per quanto riguarda i produttori infatti di essi verrà mostrato l'id, il numero e la data dell'ultimo messaggio scritto ecc…
- Mostrare il numero di record in una topic, sia totali sia per ogni partizione.
- Mostrare il numero di messaggi inviati alla topic in un minuto

L'ultimo comportamento che garantisce l'operator è quello di inviare all'**Operatore Centrale** il nome della topic seguita dalle impostazioni salienti quali: il numero di partizioni, il numero di clienti sulla topic, il fattore di replicazione e infine il numero di broker attualmente attivi sul cluster.

6.6 Operatore Centrale

L'operatore Centrale è quello che si occupa della supervisione dello stato dell'intero sistema. Questo svolge due importanti compiti per il sistema: il primo si tratta del ruolo di orchestratore del cluster, comunicando continuamente con i topic operator esso calcola il numero di partizioni e di broker ideali per il sistema in base al numero di utenti collegati e fattori di replicazione e li comunica a questi ultimi. In secondo luogo si occupa di richiedere al server di Jaeger i dati delle telemetrie raccolti da OpenTelemetry, elaborarli, salvarli e mostrarli sotto forma di grafici.

6.6.1 Orchestratore del cluster

Una volta avviato in modo ciclico riceve dai Topic Operator tutti i dati riguardanti le topic create nel cluster, le rispettive partizioni, i clienti collegati, il fattore di replicazione e i broker attivi. Questi dati vengono successivamente sottoposti ad un algoritmo (*Figura 6.5*) in grado di calcolare il numero ideale di partizioni e broker di cui rispettivamente topic e sistema dovrebbero essere composti, il tutto per massimizzare l'efficienza dell'architettura in questione.

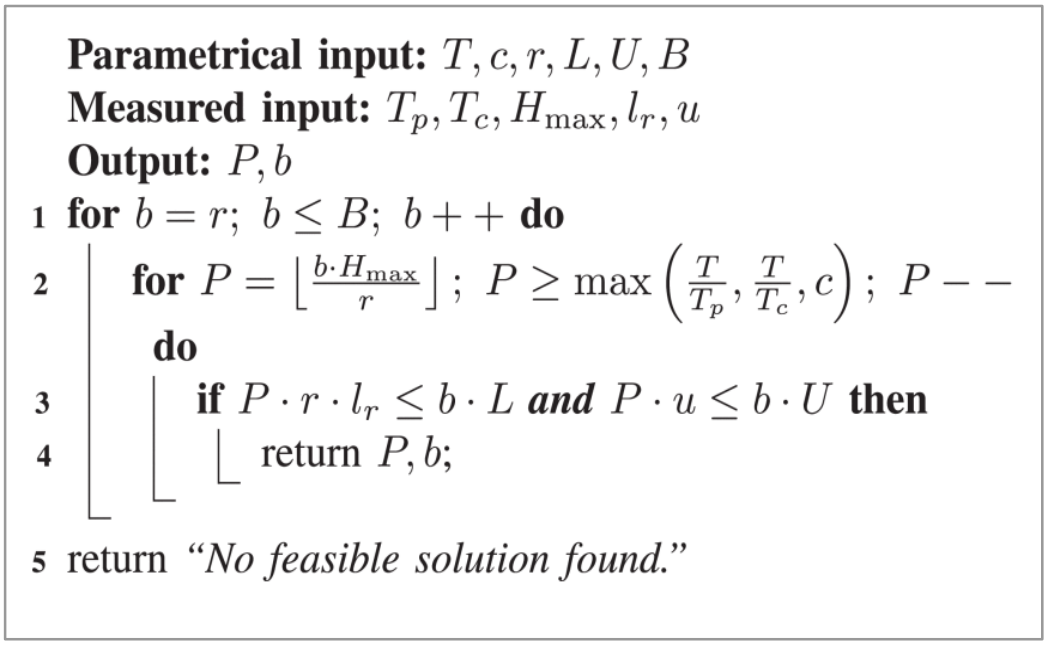

Figura 6.5: Algoritmo per l'ottimizzazione cluster kafka

Fonte: Theofanis P. Raptis, Andrea Passarella, 'On Efficiently Partitioning a Topic in Apache Kafka', 2022

L'algoritmo non solo serve a determinare un numero efficiente di partizioni e broker partendo da input specifici, ma è anche progettato per ridurre al minimo indispensabile questi elementi. Durante la ricerca condotta da Theofanis P. Raptis e Andrea Passarella [41], sono stati individuati diversi tipi di algoritmi di questo genere, ma quello scelto si è rivelato ideale, data la scarsità di risorse nel sistema esaminato. Gli input fissi, che sono stati presi in considerazione come nei test descritti nel documento citato, includono:

- **H_{max}:** numero massimo di file che possono essere aperti su un broker.
- **l_r:** latenza di replicazione
- **L:** soglia massima per la latenza di replicazione
- **u:** tempo di indisponibilità
- **U:** soglia per il tempo di indisponibilità
- **T:** throughput
- **T_p, T_c:** throughput scrittura/lettura

Per quanto riguarda invece gli input variabili sono stati settati in base alle impostazioni delle topic che vengono inviate dai vari Topic Operator, essi sono:

- **c:** clienti su una specifica topic
- **r:** fattore di replicazione

Infine il parametro **B** che indica il numero massimo di broker disponibili è stato settato a 5. Una volta processati i dati ricevuti l'algoritmo, come detto in precedenza, ritornerà il numero ideale di partizioni e broker per una determinata topic. A questo punto l'operatore centrale invierà ai singoli Topic Operator il numero di partizioni che devono essere aggiunte e provvederà ad avviare i broker extra necessari al cluster.

6.6.2 Elaboratore di telemetrie

In seguito l'operator invia una richiesta al server di Jaeger e riceve in formato json tutti i dati relativi alle tracciature raccolti da OpenTelemetry, li riordina, li analizza e attraverso l'uso di grafici permette di mostrare in modo semplice le tempistiche che hanno caratterizzato i due gruppi di messaggi scambiati nel cluster.

In particolare verranno mostrati grafici riguardanti: il tempo di latenza medio dei messaggi appartenenti ai due tipi di percorsi disponibili, il tempo di latenza di tutti i messaggi colorati in modo differente in base al percorso intrapreso e infine la media mobile del tempo di latenza di entrambi i gruppi di messaggi. L'operatore, grazie ai dati scaricati da Jaeger sarà anche in grado di calcolare il throughput complessivo del sistema ovvero i messaggi al secondo che il cluster è in grado di gestire. Si è deciso di non riportare anche i tempi di processamento (lettura/scrittura) dei messaggi in quanto essi dipendono dalla logica di business e dalla complessità dei produttori/consumatori.
Dopo aver analizzato i dati di tracciatura, sarà possibile prendere decisioni riguardanti la configurazione del cluster, in particolare in merito alle partizioni e al numero di broker.

6.7 Configurazioni oggetto di test

Al fine di ottenere risultati che potessero essere i più generali possibile, si è scelto di testare più volte varie configurazioni che fossero abbastanza differenti tra loro sia riguardo al numero di broker presenti nel cluster che di partizioni da cui le topic fossero formate.
I primi test oggetto dello studio verranno effettuati con un singolo broker variando solamente il numero di partizioni nelle topic. In seguito sarà aumentato il numero di broker a tre e anche in questo caso verranno testate diverse configurazioni con diversi numeri di partizioni. Infine il numero di broker verrà aumentato a cinque. Come detto in precedenza verrà posta particolare attenzione ai dati quali la latenza dei messaggi e il throughput del sistema per trarre tutte le informazioni possibili allo scopo di giungere ai pro e contro delle varie architetture. I test sono stati svolti nella seguente modalità: per ogni configurazione sono state effettuate quattro esecuzioni ed è stato preso come risultato finale la media dei valori di queste ultime.

7 Risultati dei test

Nel presente ed ultimo capitolo verranno illustrati e discussi i risultati dei test svolti sulle architetture descritte in precedenza. In particolare saranno mostrati i tempi di latenza dei messaggi e di throughput del sistema caratterizzato da un diverso numero di broker e di partizioni. Generalmente vedremo che spesso nonostante si osservi un miglioramento nella capacità del sistema di elaborare più messaggi contemporaneamente, ciò non sempre si traduce in una riduzione del tempo necessario per l'elaborazione di ciascun messaggio. In alcuni casi, questo tempo potrebbe addirittura aumentare.

7.1 Infrastruttura con un Broker

La prima architettura di cui si mostreranno i risultati sarà quella con un solo Broker centrale interrogato da tutti gli attori del sistema già descritti.
La presenza di un solo broker semplificherà sicuramente le interazioni con l'infrastruttura, poiché le comunicazioni saranno limitate esclusivamente con i clienti. Di conseguenza, anche gli scambi di messaggi saranno ristretti a questi ultimi. Sicuramente però ne verrà afflitta non solo la scalabilità del sistema, dato che non sarà possibile bilanciare il carico con altri broker, ma anche in caso di malfunzionamento l’intero sistema sarà a rischio.
Andremo ora a paragonare il funzionamento del sistema con diverso numero di partizioni.

7.1.1 Partizione singola

Avviamo dunque i test con un singolo broker e una singola partizione per topic

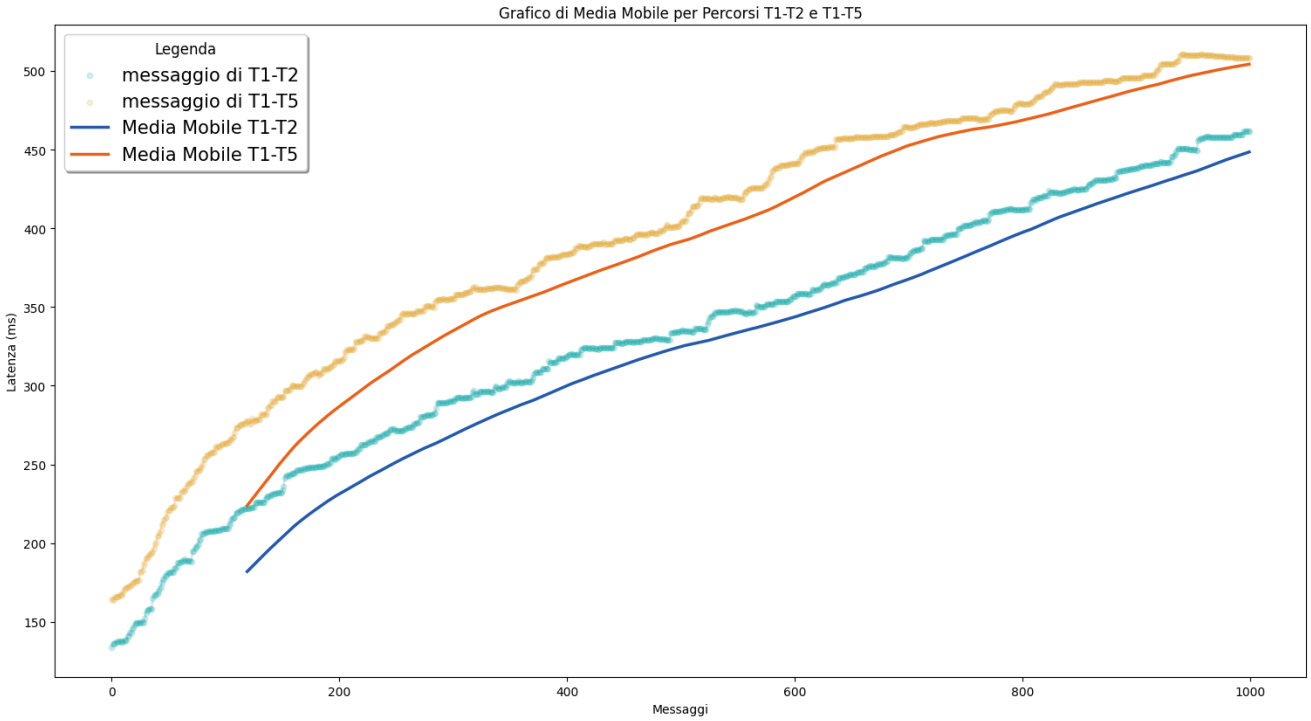

Figura 7.1: Grafico di media mobile per i messaggi del cluster, singola partizione e singolo broker

Come possiamo vedere chiaramente dalla *Figura 7.1* notiamo che l'utilizzo del singolo broker permette in partenza di mantenere un latenza durante tutto il percorso del cluster molto bassa, questo grazie alla minima complessità delle comunicazioni messe in atto nell'infrastruttura ed a una quasi nulla logica di gestione delle partizioni. Con l'aumento dei messaggi però, non essendoci la possibilità di distribuzione del carico, notiamo che il tempo che i messaggi impiegano per attraversare i propri percorsi aumenta in modo direttamente proporzionale al numero di messaggi scambiati.

Non solo la mancanza di distribuzione del carico influisce negativamente su questa architettura ma anche la totale assenza di replicazione delle partizioni delle topic rende il sistema estremamente vulnerabile a guasti e malfunzionamenti, che rischia così non solo di interrompere il servizio finché il guasto al broker principale non è riparato ma rischia anche di perdere tutti i record precedentemente scritti sulle topic.

7.1.2 Cinque partizioni

Per i successivi test il numero di partizioni di cui le topic sono composte è stato aumentato a cinque in modo da consentire di parallelizzare la produzione e il consumo di messaggi. Ogni partizione può essere consumata indipendentemente da un consumer, permettendo un'elaborazione più veloce se i consumatori sono ben bilanciati.

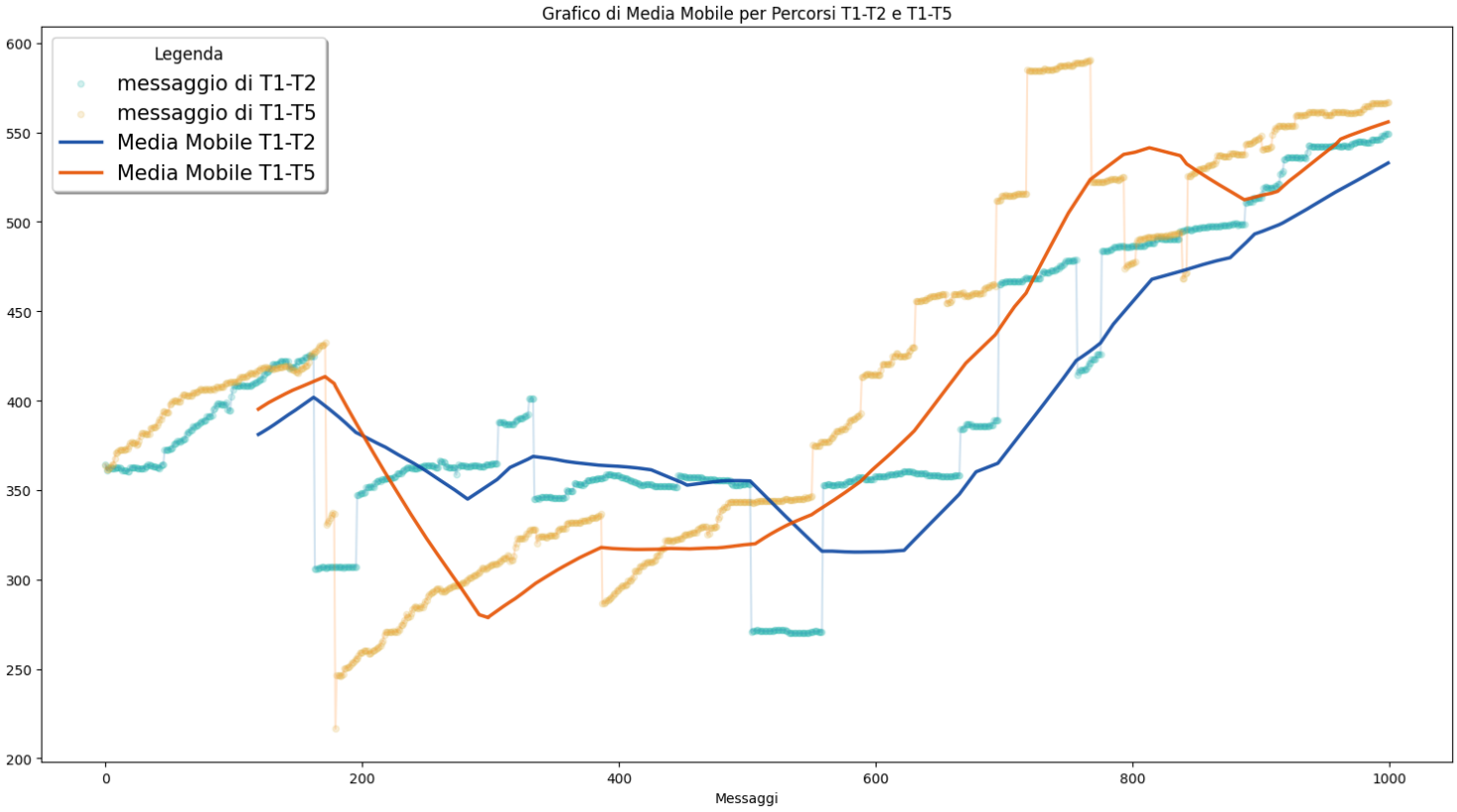

Figura 7.2: Grafico di media mobile per i messaggi del cluster, cinque partizioni e singolo broker

Si prestano alla nostra analisi dei risultati già molto differenti dai precedenti. In questo caso (*Figura 7.2*) si può subito notare come la parallelizzazione possa causare in certi messaggi un crollo nel tempo di latenza, questo perché aumentando la capacità di scrivere in parallelo sulle topic del sistema non solo aumenta la velocità dei produttori di scrivere più messaggi possibile in un minore tempo ma anche i consumatori sono in grado di consumare più record, diminuendo così i tempi di attesa. Notiamo comunque che la presenza di un singolo broker, nonostante maggiore throughput dato dalle partizioni, influisce ancora negativamente sulle prestazioni del sistema con un alto volume di messaggi, infatti, quando si raggiunge la soglia

dei 2000 messaggi totali circolanti nell'infrastruttura, si osserva come anche in questo caso i tempi si apprestano a salire regolarmente.

7.1.3 Dieci partizioni

Quest'ultimo tipo di test è stato svolto con l'intenzione di verificare in quale proporzione aumentare il numero di partizioni causa un aumento della complessità della gestione dei dati. Se un messaggio deve attraversare troppe partizioni per completare il suo percorso potrebbe infatti introdurre overhead.

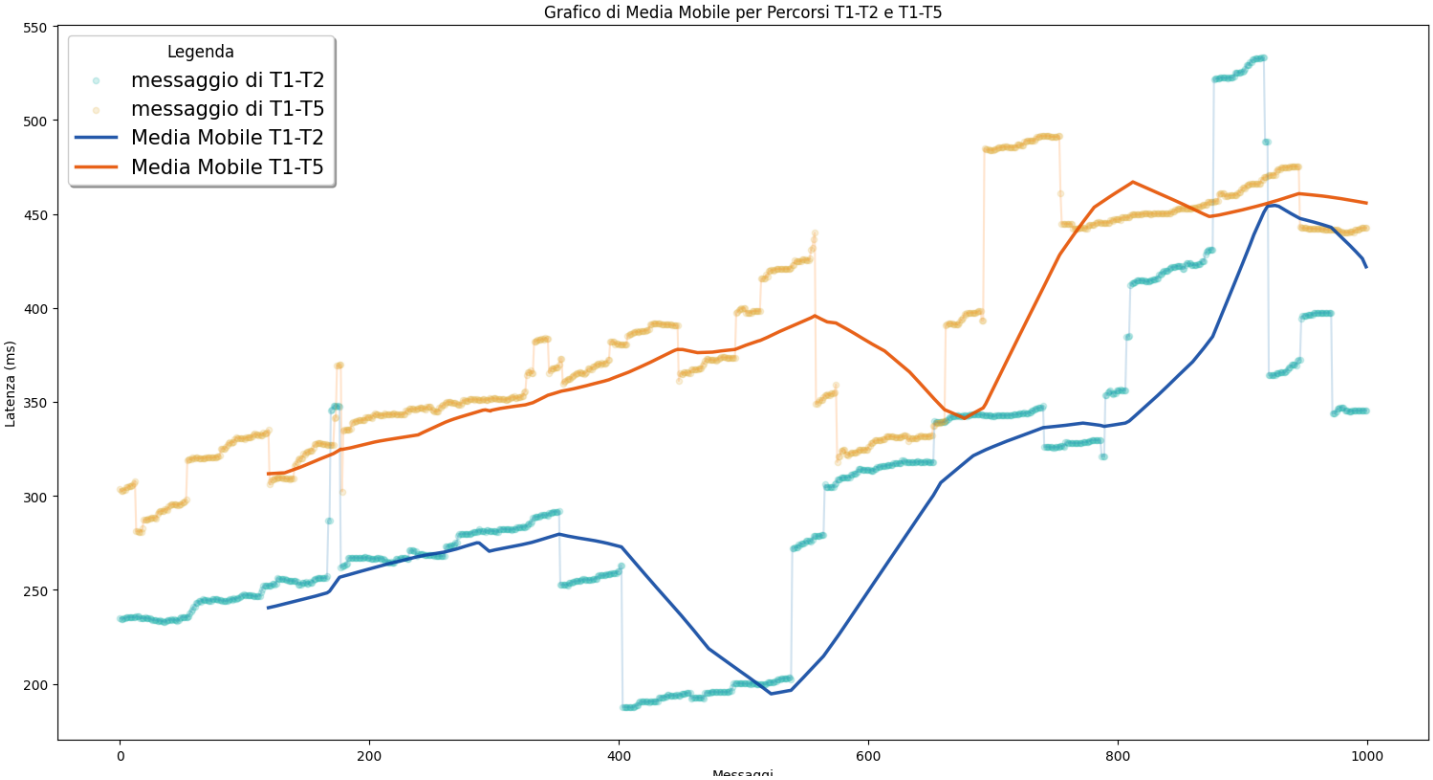

Figura 7.3: Grafico di media mobile per i messaggi del cluster, dieci partizioni e singolo broker

In primo luogo ciò che il grafico (*Figura 7.3*) mostra è il crollo a cui alcuni tempi di latenza sono soggetti. Come mostrato nel test precedente (*Figura 7.2*) infatti la parallelizzazione delle topic, ovvero la possibilità di leggere e scrivere molti più dati insieme, influisce positivamente sulla velocità con cui i messaggi attraversano l'infrastruttura. In questo caso però la considerazione più importante è quella che scaturisce dall'osservazione nel momento più critico dello scambio, ovvero vicino alla soglia dei 2000 messaggi. Nel test attuale infatti

la presenza di un numero di partizioni doppio rispetto al precedente riesce in qualche modo ad attenuare le prestazioni negative che ha la presenza di un singolo broker. È possibile osservare infatti che i tempi, nonostante siano più alti, non raggiungono livelli troppo elevati, sembrano anzi stabilizzarsi.

7.1.4 Considerazioni

Ora che sono stati effettuati tutti i test per la configurazione ad un broker verranno mostrate comparazioni in termini di latenza media dei messaggi e di throughput, successivamente discuteremo dei risultati.

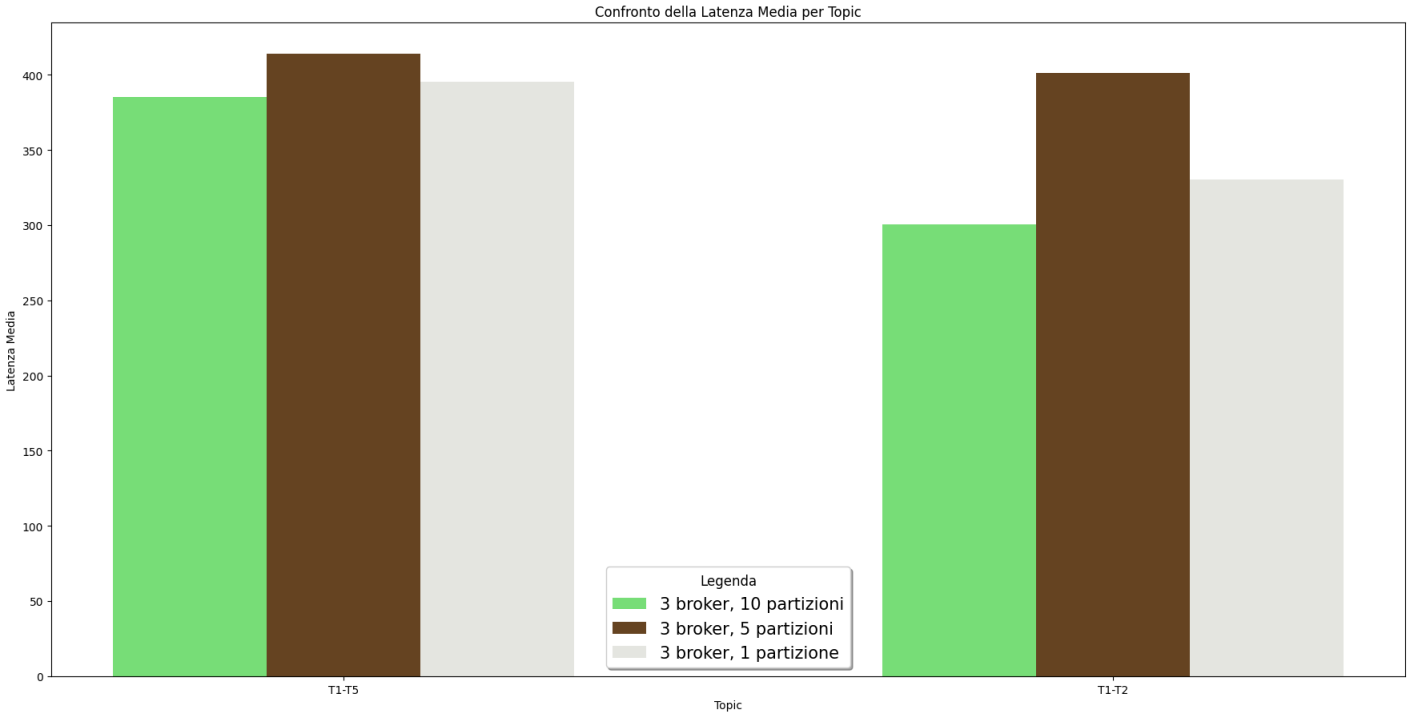

Figura 7.4: Comparazioni tempo di latenza medio per percorso dei differenti test

La *Figura 7.4* illustra i tempi di latenza medi per messaggio relativi ai due tipi di percorsi nella nostra architettura. Possiamo notare come sia la configurazione con una sola partizione (*Figura 7.1*) sia quella con dieci (*Figura 7.3*) abbiano latenza inferiore a quella con cinque (*Figura 7.2*). Questo può essere collegato al fatto che avere diverse partizioni causa un overhead dovuto ad una maggiore complessità algoritmica nella gestione dei dati. È evidente però che un numero maggiore di partizioni in questo caso favorisce un miglior

bilanciamento del carico. Nel caso del terzo test, l'aumento dell'overhead viene compensato dalla maggiore velocità complessiva del sistema, cosa che non avviene con sole cinque partizioni.

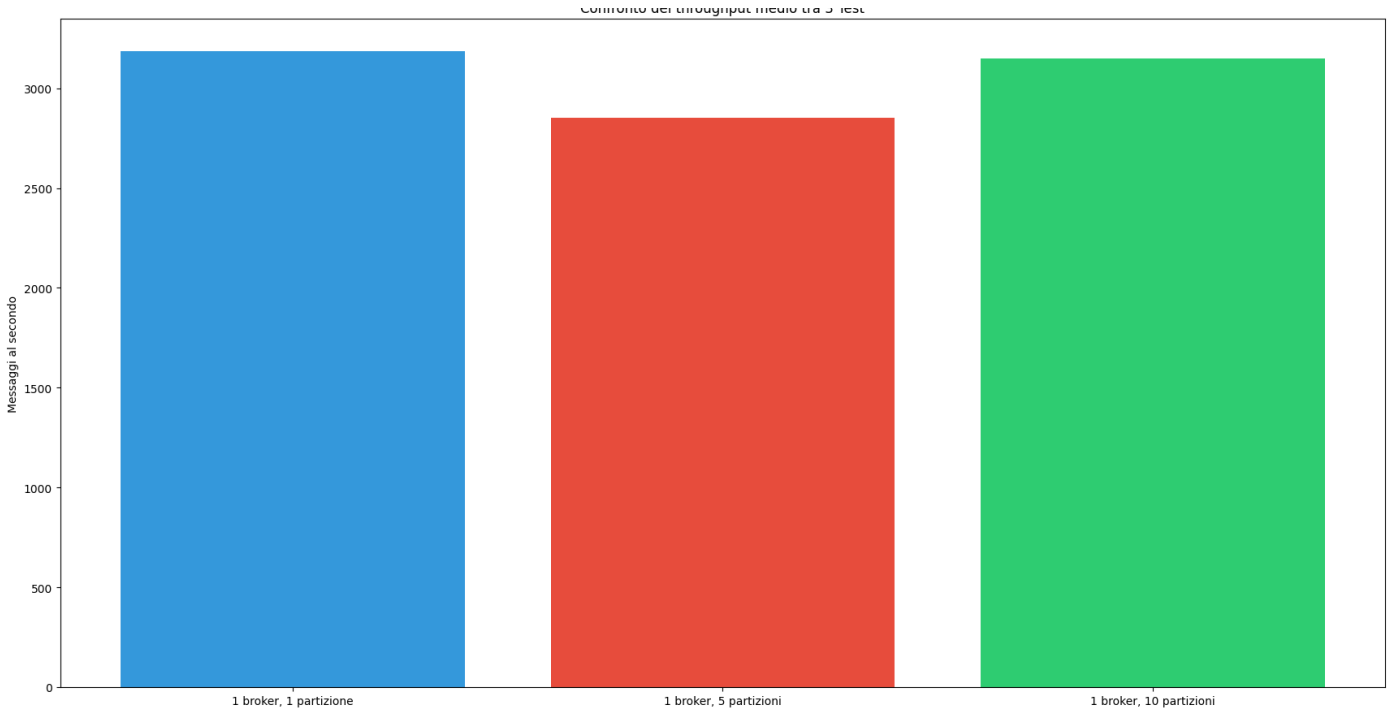

Figura 7.5: Comparazione tempo di throughput dei vari test

Ciò che testimonia la *Figura 7.5* riguardante il throughput è il fatto che per tutte le infrastrutture testate il numero di messaggi scambiati da tutto il sistema si attesta intorno alla soglia dei 3000 messaggi al secondo. Solo la configurazione con cinque partizioni presenta valori leggermente inferiori, probabilmente a causa dello stesso motivo che ha causato un aumento della latenza. La complessità introdotta da un numero maggiore, ma ancora insufficiente, di partizioni ha infatti avuto effetti negativi sul sistema.

7.2 Infrastruttura con tre Broker

Passiamo ora all'infrastruttura caratterizzata da tre broker, faremo i medesimi test del sistema precedente. Gli attori ora sono collegati ai vari broker in modo da bilanciare il carico delle comunicazioni. L'introduzione di due broker aggiuntivi potrebbe rendere il sistema leggermente più lento a causa della maggiore complessità delle interazioni con l'architettura e dell'aumento degli scambi di messaggi tra i broker per consentire la cooperazione.

Tuttavia, ciò comporta un aumento della sicurezza in caso di malfunzionamenti, poiché i topic possono essere replicati anche sui broker che non sono leader. Inoltre, il carico potrà essere meglio distribuito all'interno del sistema, riducendo il rischio di sovraccarichi e colli di bottiglia nella rete.

7.2.1 Partizione singola

Nonostante una maggiore cooperazione da parte dei broker la limitazione di avere le topic composte solamente da una partizione e quindi nessun tipo di parallelismo influenzerà negativamente la capacità del sistema nel distribuire il carico di lavoro.

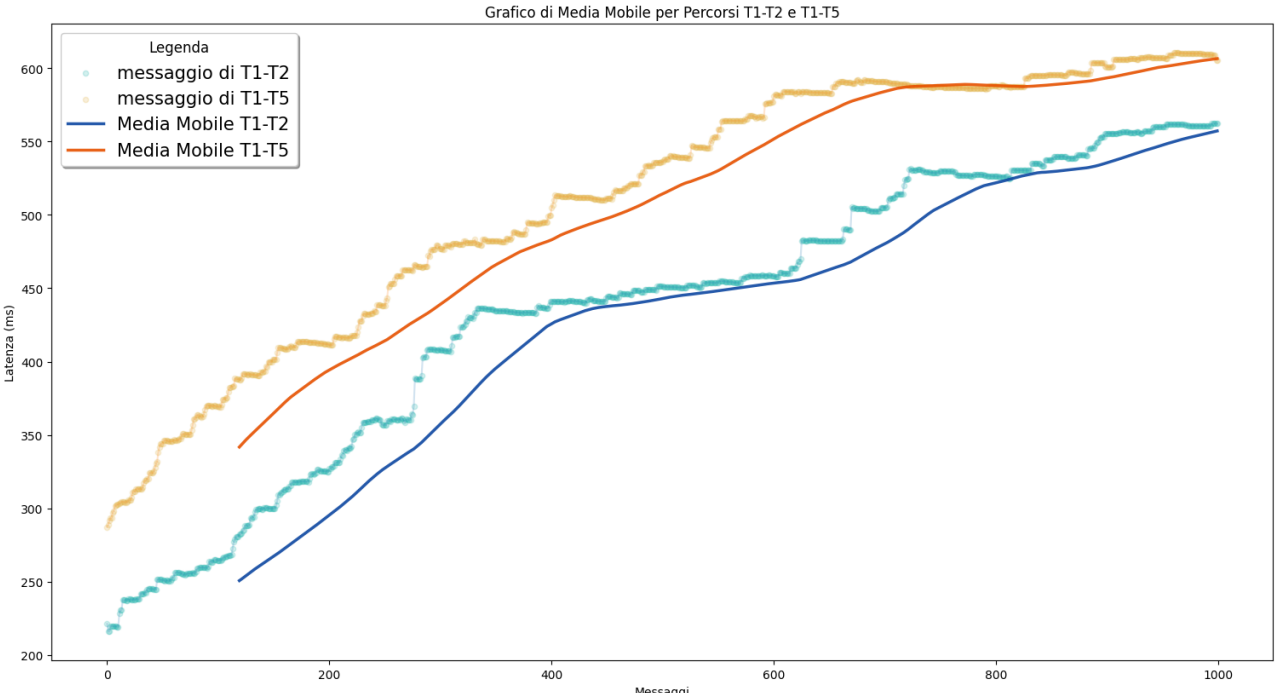

Figura 7.6: Grafico di media mobile per i messaggi del cluster, singola partizione e tre broker

Notiamo come i risultati della *Figura 7.6* siano molto simili a quelli con un singolo broker (*Figura 7.1*), i messaggi infatti partono molto velocemente e finché il sistema rimane leggero i tempi di latenza sono molto bassi. Quando però il carico diventa consistente allora i tempi si alzano in modo proporzionale. Se entriamo nel dettaglio possiamo notare però che a differenza dei risultati con il singolo broker qui i tempi sono leggermente aumentati di circa 100 millisecondi, causato probabilmente dalla necessità di avere più interazioni fra i

broker che riguardano le responsabilità sulle topic e sulle repliche. Questo contribuisce sicuramente a migliorare la sicurezza contro malfunzionamenti e sovraccarichi, ma inevitabilmente aumenta il tempo di trasmissione dei record.

7.2.2 Cinque partizioni

Aumentando le partizioni a cinque nell'infrastruttura con tre boker, testando più volte il sistema, si presentano i seguenti risultati.

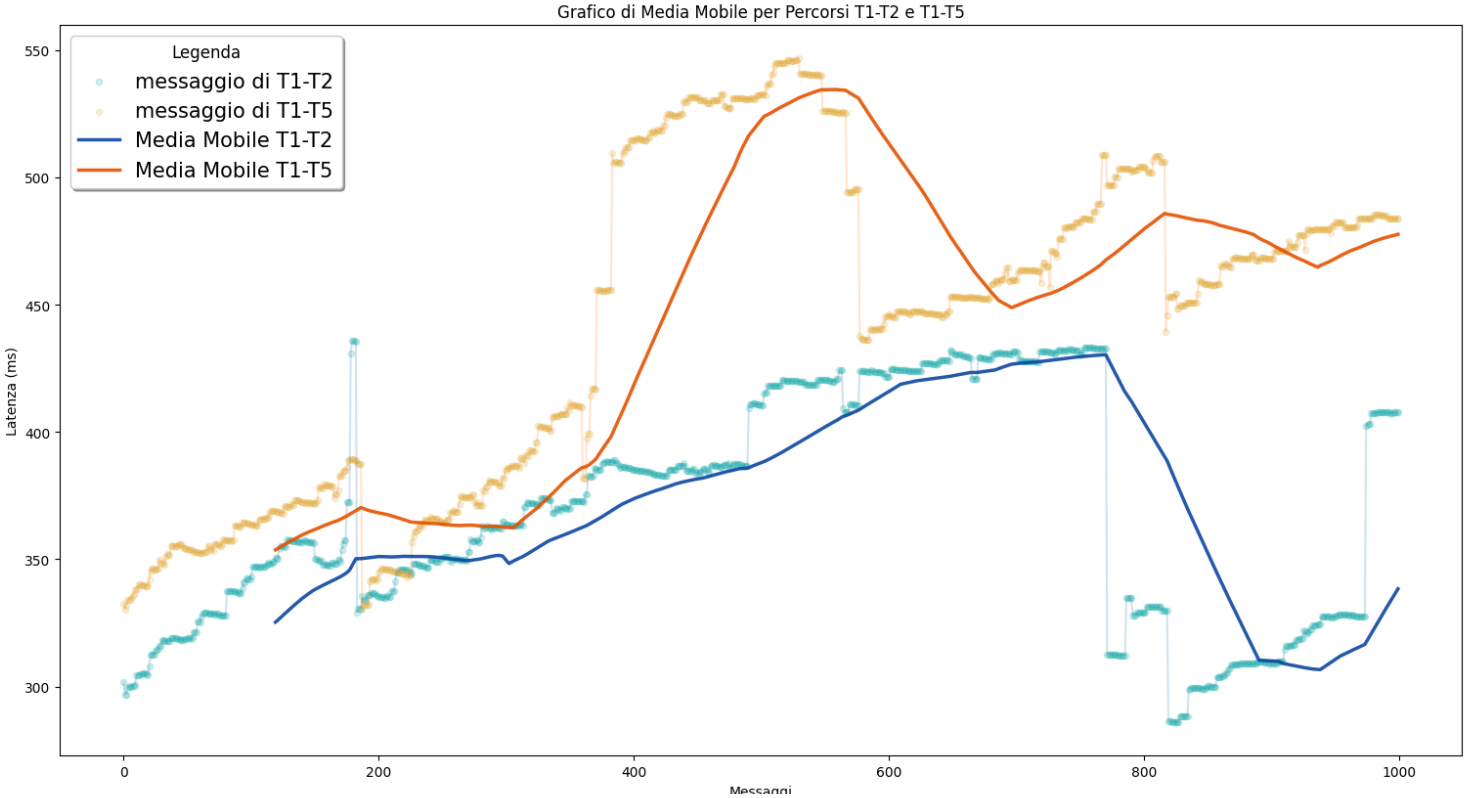

Figura 7.7: Grafico di media mobile per i messaggi del cluster, cinque partizioni e tre broker

Mettendo a paragone i risultati della *Figura 7.7* con gli stessi ottenuti da un singolo broker (*Figura 7.2*) è facile notare i benefici di questa infrastruttura. Come in precedenza anche in questo caso la suddivisione delle topic in più partizioni aiuta ad abbattere i tempi di latenza della maggior parte dei messaggi scambiati. Si può facilmente vedere come la parallelizzazione delle topic permette a tutti gli attori del sistema di gestire un carico considerevole di messaggi molto più facilmente, complice del miglioramento delle

prestazioni è anche la presenza di più broker che collaborando garantiscono una gestione del carico più bilanciata unita a tutti i benefici già ampiamente descritti in precedenza. Confrontando questi risultati con quelli ottenuti con un singolo broker, si nota chiaramente che, alla soglia dei 2000 messaggi, dove in precedenza si era registrato un leggero calo delle prestazioni, l'aumento del tempo di latenza è ora significativamente inferiore.

7.2.3 Dieci partizioni

Completiamo il set di test, anche su questa architettura, aumentando a dieci le partizioni che costituiscono la topic.

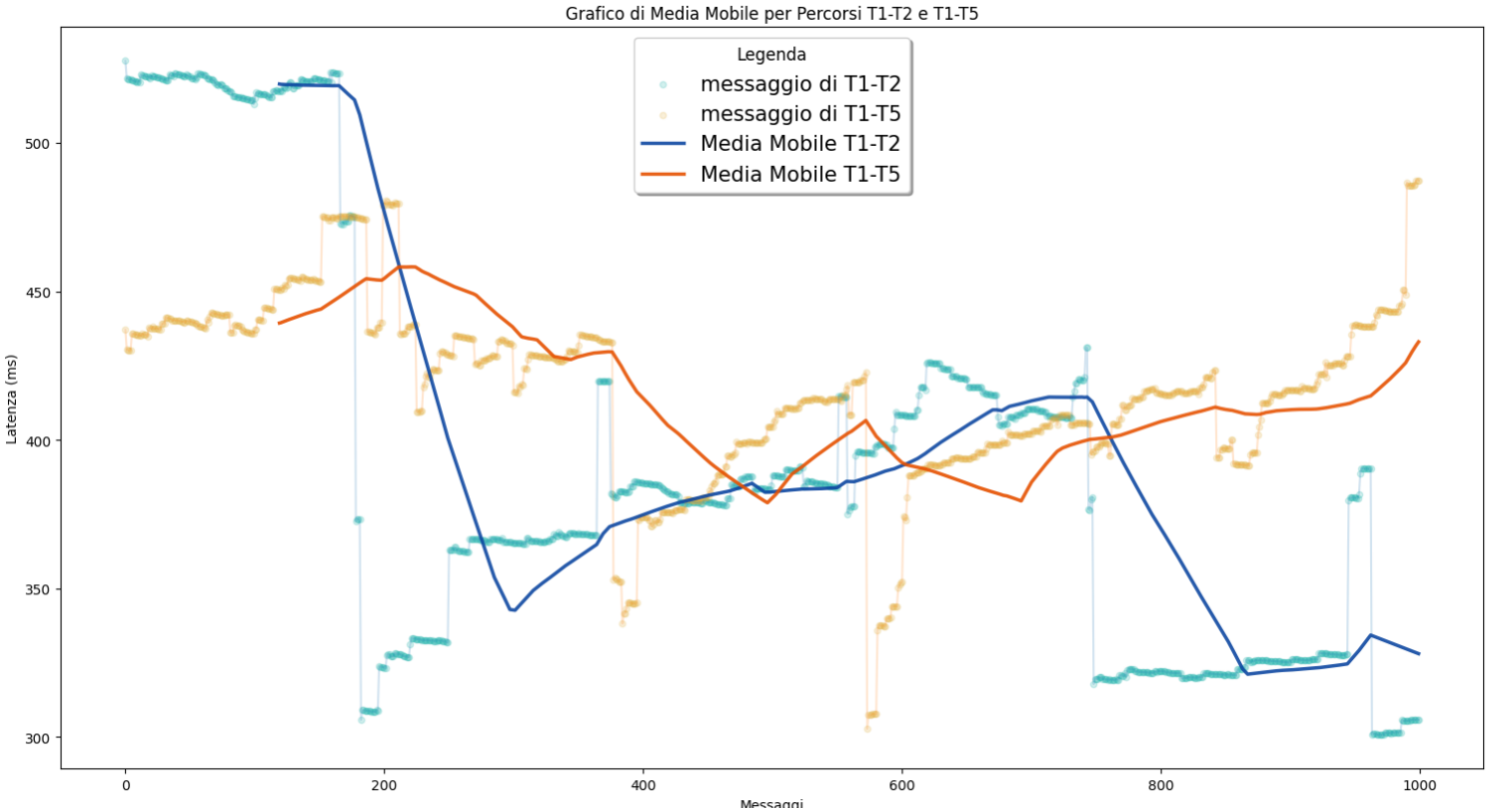

Figura 7.8: Grafico di media mobile per i messaggi del cluster, dieci partizioni e tre broker

Questa volta i risultati della *Figura 7.8* evidenziano un comportamento decisamente diverso del sistema. L'overhead introdotto dalla maggiore complessità algoritmica, dovuta alla cooperazione di tre broker che si suddividono le responsabilità di numerose partizioni di topic e delle relative repliche, comporta un avvio mediamente più lento rispetto alle configurazioni precedentemente testate. Nella fase iniziale, con un carico di messaggi ancora leggero, si osserva un aumento nel tempo di latenza. Tuttavia, man mano che

l'esecuzione prosegue e il carico di lavoro aumenta, il sistema non solo mantiene le prestazioni stabili, ma in alcuni casi riesce addirittura a ridurre sostanzialmente i tempi di latenza sui percorsi più brevi, come dimostrano i messaggi del percorso T1-T2.
Ne si deduce che un'infrastruttura di questo tipo è particolarmente adatta per gestire carichi elevati, mentre non è consigliabile per carichi più leggeri.

7.2.4 Considerazioni

Come già fatto in precedenza saranno ora mostrati i tempi di latenza medi che hanno caratterizzato i test svolti e dopo aver mostrato la capacità di throughput degli stessi discuteremo quanto appreso.

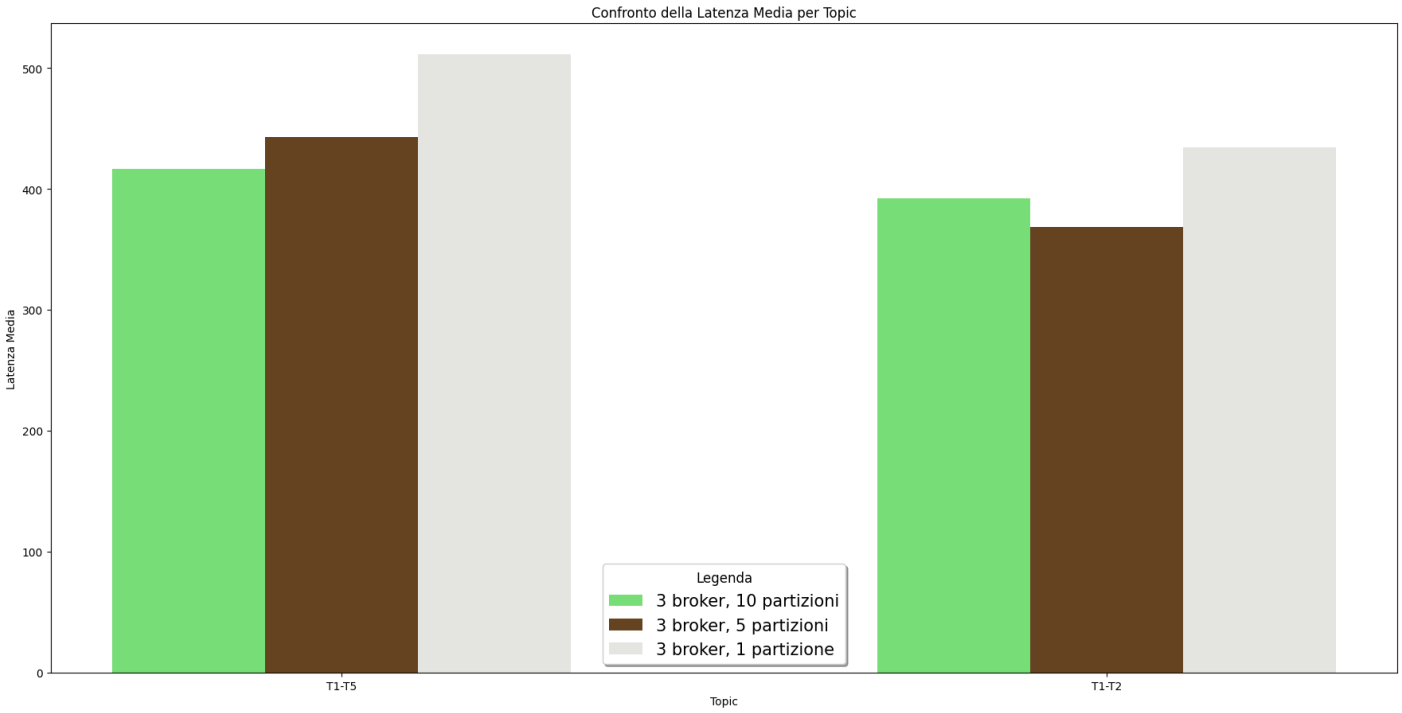

Figura 7.9: Comparazioni tempo di latenza medio per percorso dei differenti test

Il grafico (*Figura 7.9*) rivela innanzitutto che i tempi di latenza medi sono generalmente maggiori di quelli ad un solo broker (*Figura 7.4*), questo però non ci sorprende visto quanto detto in precedenza. Altro fattore molto importante che è facile notare è che per i percorsi con tante iterazioni, come T1-T5 in questo tipo di architettura, attraversare delle topic suddivise in molte partizioni aiuta a mantenere i tempi di latenza più contenuti. Questi tempi rimangono uguali anche per i viaggi più corti, come T1-T2, ma in questo caso vi sono delle

prestazioni migliori se il numero di partizioni è inferiore. In generale risulta che avere le topic partizionate comporta un considerevole vantaggio durante l'esecuzione del sistema. Vediamo ora le comparazioni del throughput riguardante i tre test:

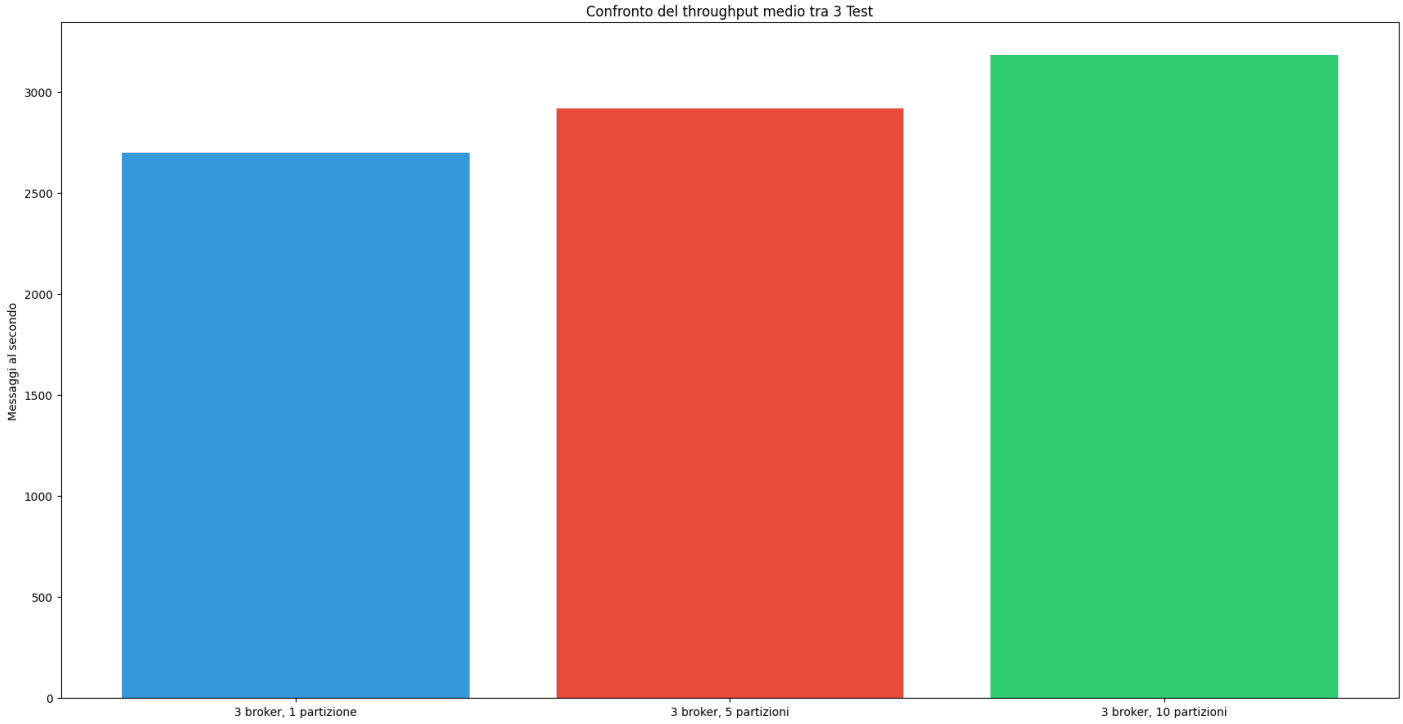

Figura 7.10: Comparazione tempo di throughput dei vari test

Vediamo come generalmente il numero di messaggi che il sistema riesce a elaborare in un secondo si assesta nella soglia dai 2500 ai 3000, soltanto nell'ultimo caso il sistema permette che il carico dei messaggi sia gestito più velocemente. Nell'infrastruttura attuale i risultati sono molto chiari: più una topic è suddivisa in partizioni maggiore sarà il carico in parallelo che il sistema potrà affrontare e maggiore sarà anche il numero di messaggi al secondo che possono essere elaborati, abbassando così la latenza e permettendo di compensare i vari rallentamenti causati da un'elaborazione più complessa.

7.3 Infrastruttura con cinque Broker

Aumentando ancora il numero di broker nell'infrastruttura si cercherà di mostrare fino a che punto il sistema può gestire, far comunicare e cooperare questi elementi senza causare un grave aumento sui tempi degli scambi di messaggi. Esamineremo fino a che punto è

possibile bilanciare una maggiore sicurezza attraverso la replicazione e un miglior controllo del carico, a fronte di un peggioramento contenuto nei tempi di latenza.

7.3.1 Partizione singola

Primo set di test riguarda l'infrastruttura con le topic suddivise in una sola partizione, vediamo i risultati dopo diverse prove:

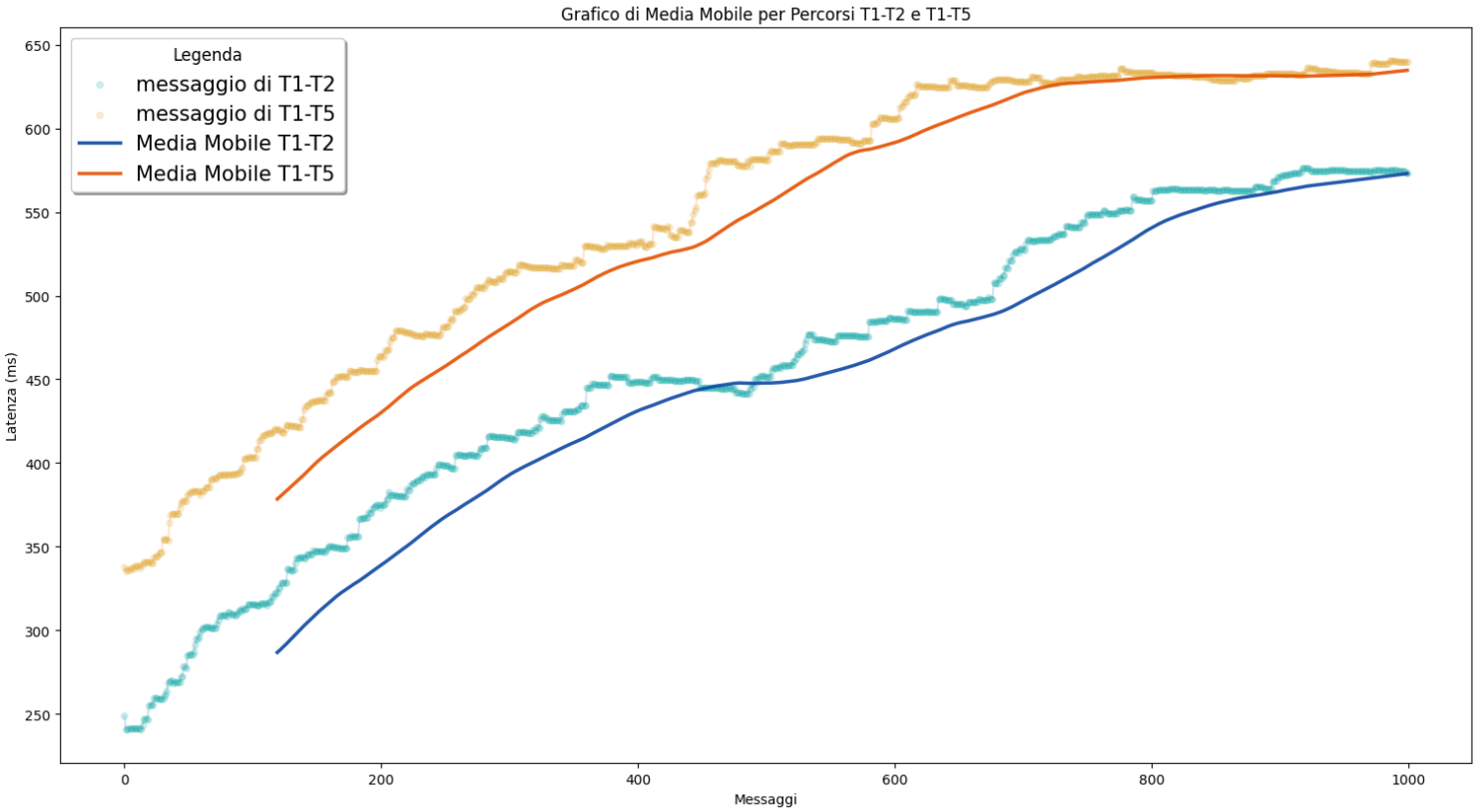

Figura 7.11: Grafico di media mobile per i messaggi del cluster, singola partizione e cinque broker

Il risultato della *Figura 7.11* è molto simile alle precedenti architetture con topic ad una partizione (*Figura 7.1, Figura 7.6*), notiamo infatti che il comportamento è praticamente identico. Vediamo inoltre che, come nel caso con tre broker, i tempi medi di latenza sono ulteriormente aumentati. Questo ritardo è sicuramente dovuto alla necessità di una maggiore interazione tra i broker.

7.3.2 Cinque partizioni

Aumentando a cinque le partizioni di cui sono composte le topic vengono ottenuti i seguenti risultati:

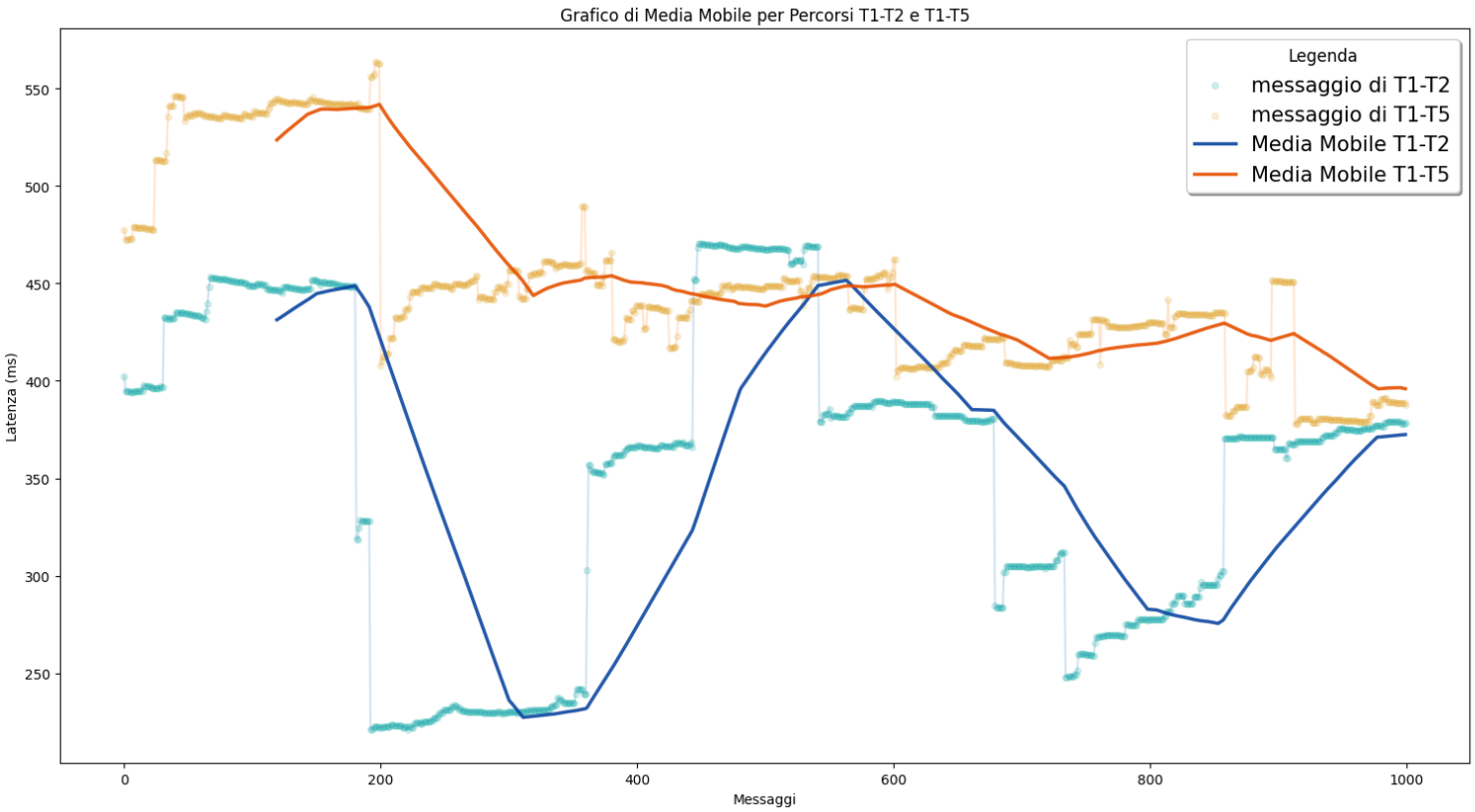

Figura 7.12: Grafico di media mobile per i messaggi del cluster, cinque partizioni e cinque broker

Partiamo con l'analizzare i risultati del percorso più lungo T1-T5. Nei test è opportuno evidenziare come i primi messaggi appartenenti a questo gruppo abbiano tempi di latenza piuttosto elevati, superata la soglia dei 200 messaggi però il sistema si adatta e, grazie alla collaborazione tra broker e alla scrittura/lettura in parallelo resa possibile dalle partizioni, i tempi di latenza subiscono un taglio considerevole per poi assestarsi per tutta la durata della comunicazione. È possibile notare come, a differenza dei test precedenti con meno broker ma stesso numero di partizioni (*Figura 7.2, Figura 7.7*), con la configurazione attuale i tempi alla fine dell'esecuzione quindi con carico massimo sono inferiori a quelli ottenuti precedentemente.

Il secondo percorso invece, sebbene termini l'esecuzione sullo stesso tempo del primo, è caratterizzato da gruppi di messaggi con tempi significativamente diversi nonostante comunque mediamente inferiori. È probabile che la causa sia una scrittura su partizioni diverse e che alcune partizioni siano lette molto prima di altre dai consumatori.

7.3.3 Dieci partizioni

Anche quest'ultima infrastruttura è stata testata suddividendo le topic in dieci partizioni, vediamo i risultati:

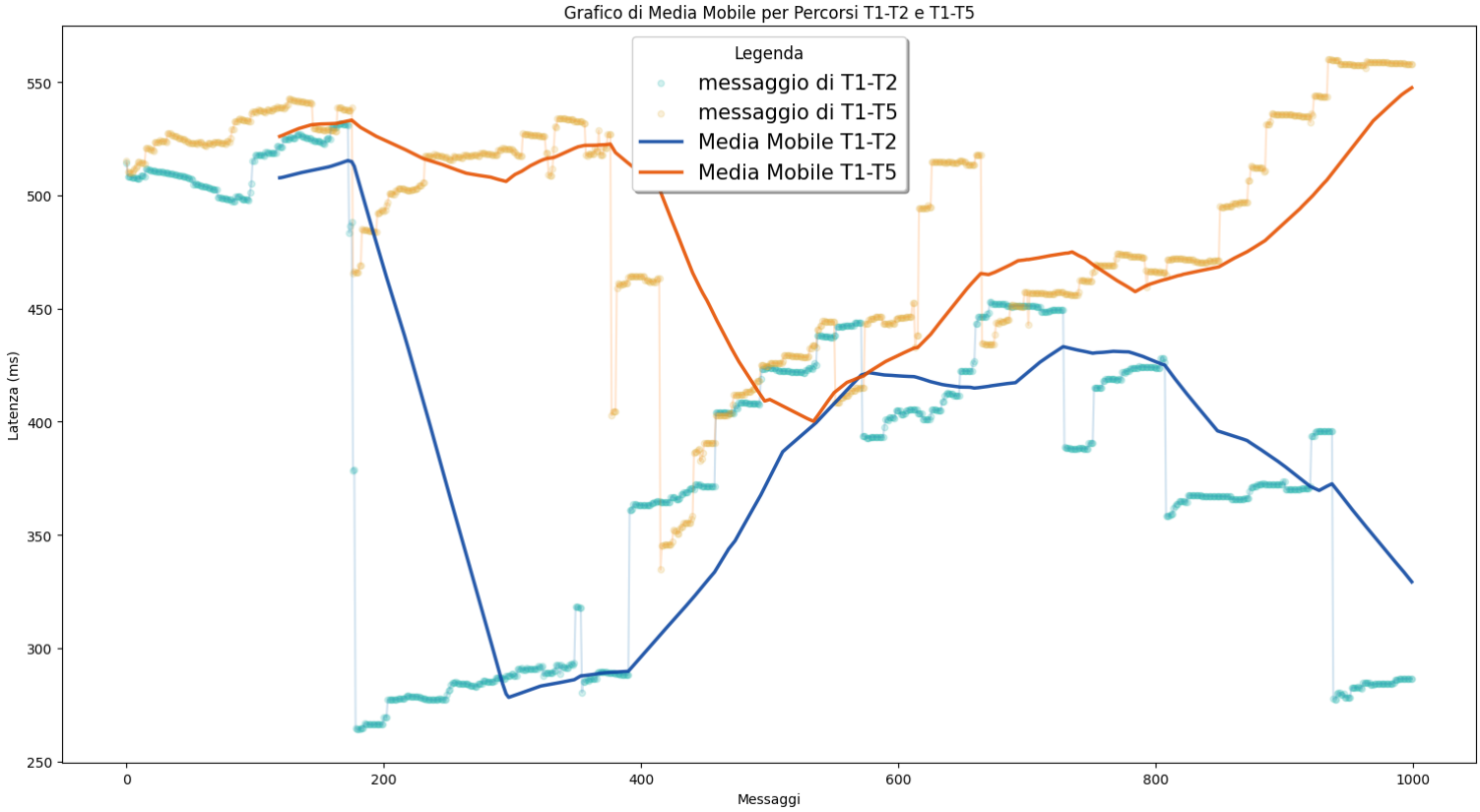

Figura 7.13: Grafico di media mobile per i messaggi del cluster, dieci partizioni e cinque broker

I dati ottenuti (*Figura 7.13*) da quest'ultimo tipo di configurazione sono profondamente diversi da quelli ricavati dalle architetture e dalle configurazioni precedenti. Si nota infatti una difficoltà generale del sistema nel mantenere dei tempi di latenza costanti fra i messaggi, vediamo che dopo una partenza lenta da parte del percorso T1-T5 si verifica un taglio nei tempi ma che ben presto tornano a salire superando addirittura quelli iniziali. Anche il

percorso T1-T2 ha delle prestazioni peggiori rispetto alle precedenti, nonostante anch'esso subisca un taglio alla latenza dopo una partenza lenta.
Questi risultati evidenziano la maggiore difficoltà del sistema nel gestire le interazioni tra numerosi broker che devono cooperare per gestire le responsabilità sui topic, le molte partizioni e le relative repliche. In questo caso, la complessità computazionale legata alla gestione dell'infrastruttura incide negativamente sulle prestazioni del sistema. Questo risultato conferma i dubbi relativi al rischio di saturare l'architettura con le comunicazioni necessarie affinché i broker possano interoperare efficacemente.

7.3.4 Considerazioni

Mostriamo e analizziamo infine i grafici riguardanti la latenza media e il throughput delle tre configurazioni testate:

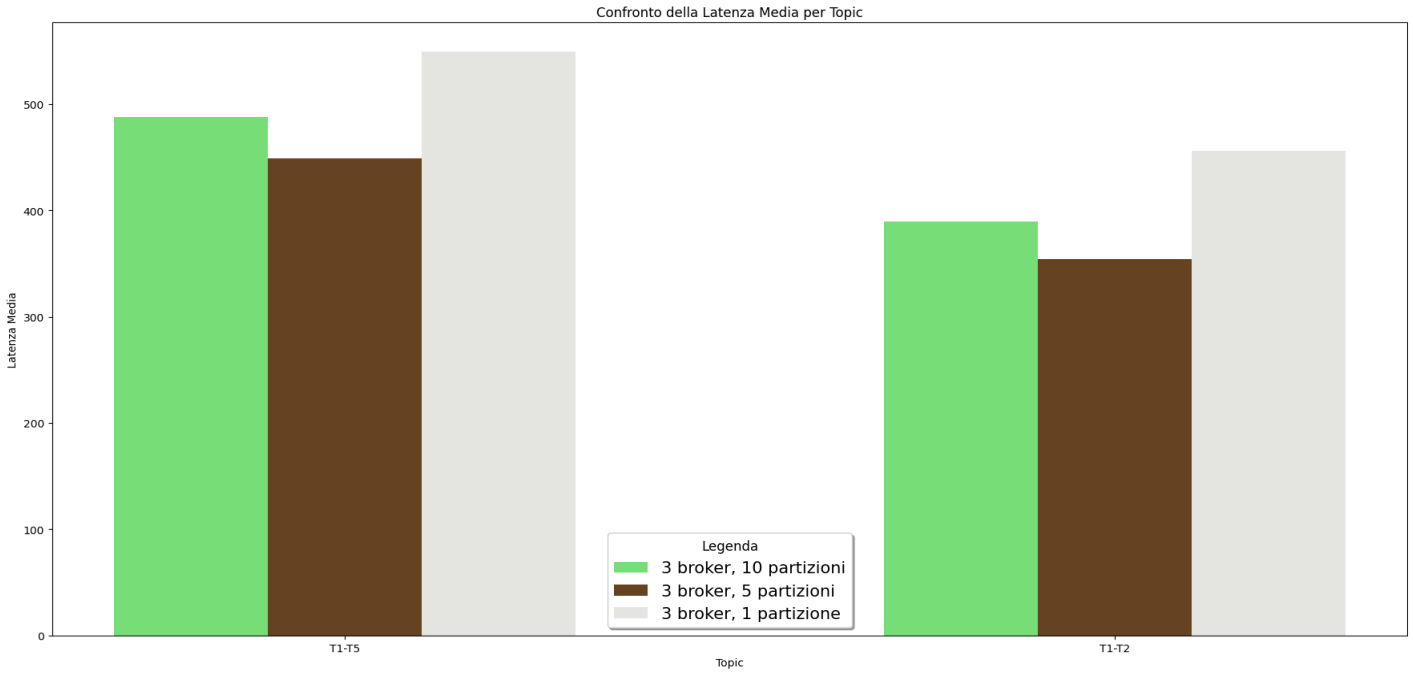

Figura 7.14: Comparazioni tempo di latenza medio per percorso dei differenti test

I dati risultanti (*Figura 7.14*) confermano quanto visto dai grafici di media mobile appena descritti: la configurazione migliore in questo tipo di architettura è quella con un numero non troppo grande di partizioni che riescano a gestire bene il carico di messaggi senza sovraccaricare il sistema. Vediamo appunto che i tempi ottenuti durante il sovraccarico sono

quelli più alti tra tutte le configurazioni con dieci partizioni precedentemente testate (*Figura 7.3, Figura 7.8*), anche se sono comunque migliori di quelle ad una partizione.
Vediamo ora la capacità di throughput per le tre configurazioni:

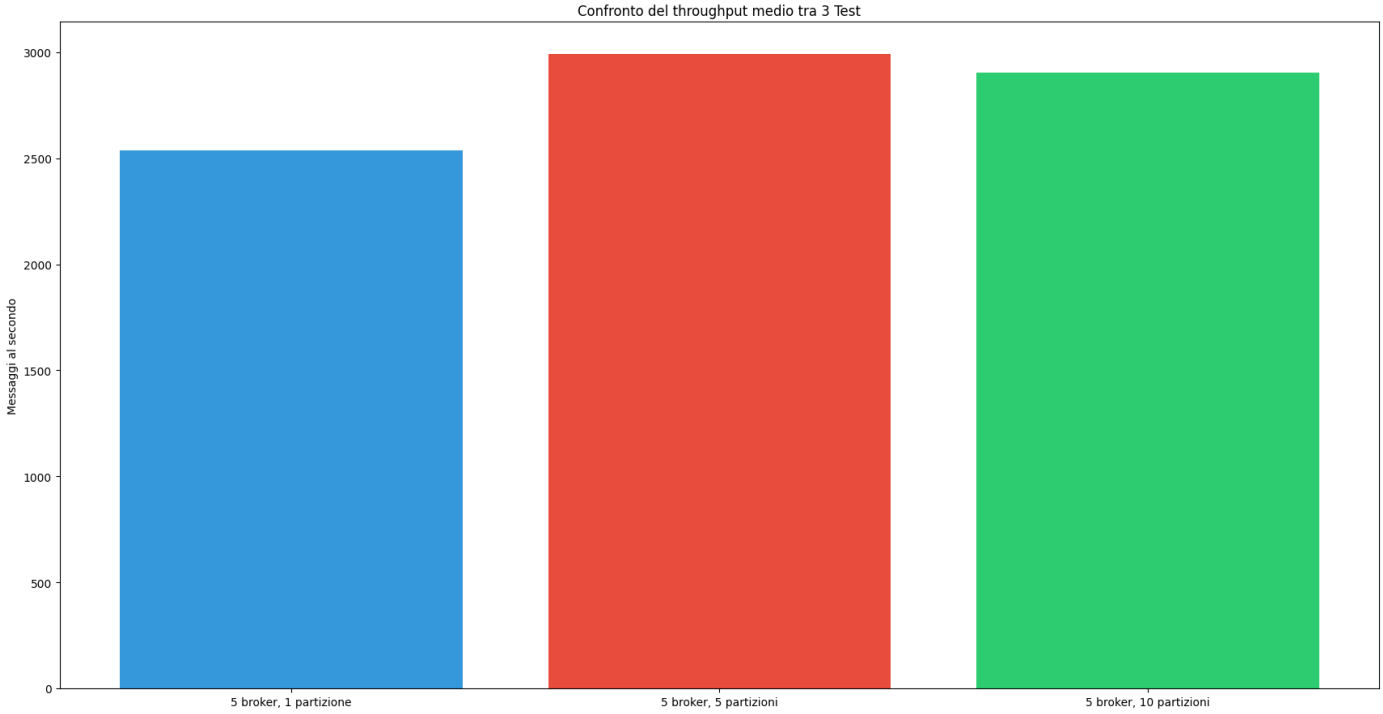

Figura 7.15: Comparazione tempo di throughput dei vari test

Anche il numero di messaggi al secondo segue la tendenza del tempo di latenza, Infatti, un sistema più bilanciato come quello con cinque partizioni è capace di gestire un numero maggiore di messaggi al secondo, accelerando di conseguenza anche la trasmissione. Nonostante però la latenza con dieci partizioni (*Figura 7.13*) sia notevolmente aumentata i messaggi al secondo che questa configurazione riesce ad elaborare è rimasta piuttosto stabile in tutti e tra i tipi di architettura.

7.4 Test con messaggi incrementati

Per concludere la serie di test condotti sul sistema, ho scelto due delle configurazioni con i migliori risultati e ho iniettato nel sistema prima 10 messaggi al secondo, poi 50, 100 e infine 200 messaggi, fino a raggiungere un totale di 2800 messaggi. Per prima cosa però sono stati mandati in input 800 messaggi, poi scartati, in modo da mandare a regime il sistema per ottenere dei risultati dei test realistici.

Questo approccio ha permesso di simulare un incremento graduale del traffico nell'infrastruttura, al fine di osservare come un sistema ben bilanciato gestisce l'aumento progressivo del carico.

I risultati ottenuti dalla configurazione a **tre broker e cinque partizioni** sono riportati nella *Figura 7.16*

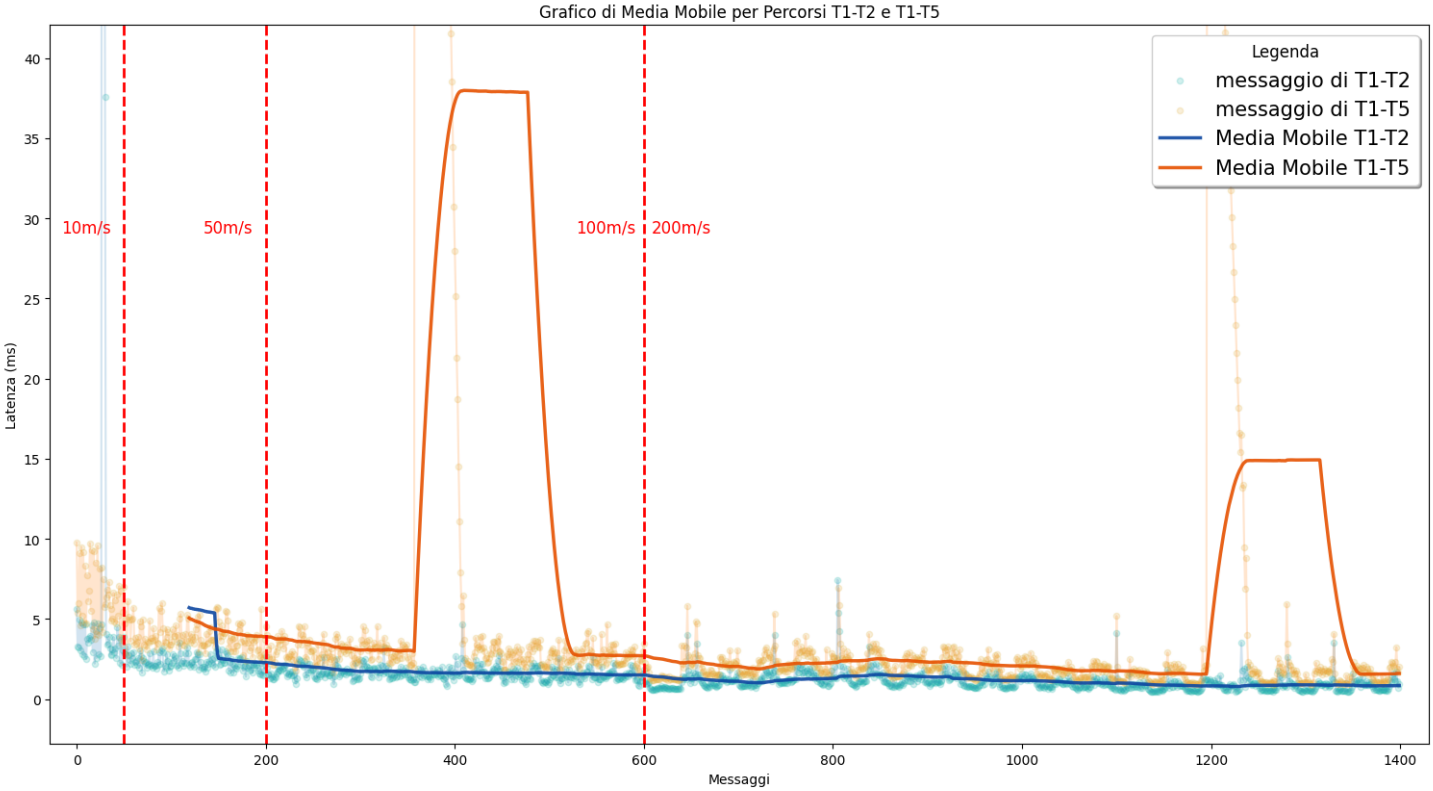

Figura 7.16: Grafico di media mobile per infrastruttura tre broker e cinque partizioni con traffico crescente

Notiamo subito che il momento più critico del sistema lo si osserva appena avviato, i primi messaggi scambiati infatti hanno un tempo di latenza leggermente maggiore di quelli che attraversano il sistema pienamente a regime. Sebbene la media della latenza, anche aumentando il carico, si assesti sotto i 5 millisecondi siamo testimoni di alcuni picchi che, anche se per un breve periodo, aumentano il tempo di coda dei messaggi.

Questi picchi però sono rari e non influiscono considerevolmente sull'efficienza dell'architettura che si impegna a garantire una latenza addirittura sui 2,5 secondi quando pienamente a regime.

Vediamo ora i risultati riguardanti la configurazione con **cinque broker e cinque partizioni**:

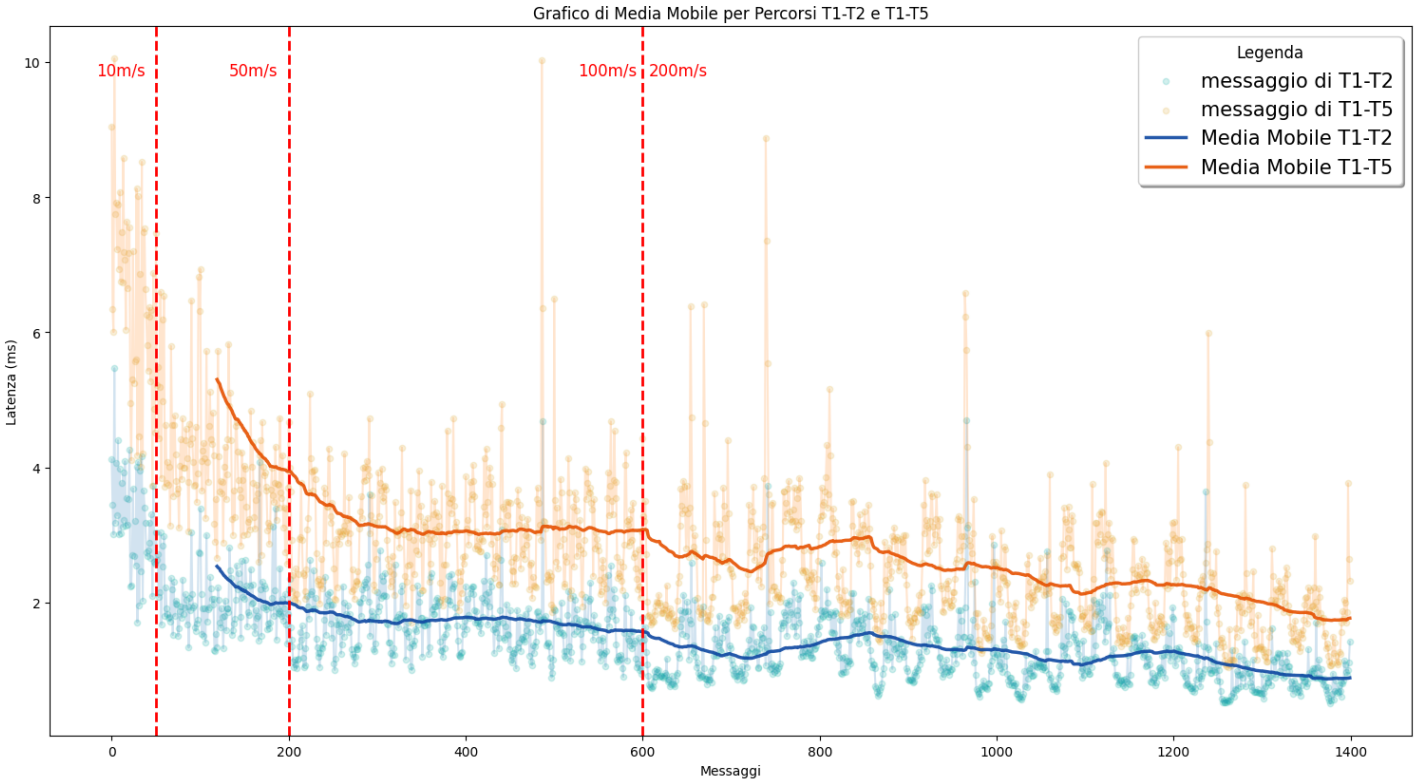

Figura 7.17: Grafico di media mobile per infrastruttura cinque broker e cinque partizioni con traffico crescente

I risultati riportati nella *Figura 7.17* sono molto simili a quelli della configurazione precedente sia per quanto riguarda la partenza che il comportamento a regime. La differenze più evidenti riguarda l'assenza di code, infatti, a differenza dell'architettura precedente in questa, nonostante alcuni messaggi abbiano picchi singoli con una latenza maggiore alla media, non vi sono code di messaggi come se ne notano dal grafico precedente. Questo è sicuramente dovuto alla maggiore parallelizzazione del lavoro garantito da cinque broker invece di tre. Per quanto riguarda il comportamento a regime, entrambi i sistemi si comportano in modo identico abbassando la latenza sotto i 2,5 millisecondi.

Ciò che possiamo ricavare da questi test è che il sistema, quando ben bilanciato, consente di gestire carichi di messaggi anche crescenti nel tempo che non intaccano la latenza con la quale gli stessi viaggiano all'interno dell'infrastruttura. Superato il momento critico

dell'avvio, ovvero quando è ancora in fase di assestamento, il sistema è in grado di gestire cambiamenti anche considerevoli nell'input di messaggi al secondo senza causare alcun ritardo percettibile nella comunicazione e anzi come vediamo dai test ridurre il tempo di coda dei messaggi immessi nel sistema.

7.5 Test su creazione topic e partizioni

Sono stati misurati per completezza anche il tempo di creazione di topic e partizioni, sono state testate tre volte le configurazioni caratterizzate rispettivamente da uno, tre e cinque broker e dei risultati ne è stata fatta la media. Ogni topic inizialmente è stata creata con cinque partizioni e successivamente ne è stato aumentato il numero di altre cinque, questo per misurare sia il tempo di creazione della topic sia quello di aggiunta di più partizioni.
i risultati ottenuti sono esposti nella seguente tabella:

	Creazione topic	1° topic modificata	2° topic modificata	3° topic modificata	4° topic modificata	5° topic modificata
Un broker	323,1 ms	48 ms	69,7 ms	94,3 ms	112,3 ms	131,3 ms
Tre broker	276,9 ms	32 ms	53,3 ms	71 ms	87,3 ms	103,7 ms
Cinque broker	264,8 ms	42,3 ms	69,7 ms	99,7 ms	128 ms	143,3 ms

I risultati evidenziano chiaramente come la distribuzione della responsabilità di **creazione di topic** su più broker consenta di **parallelizzare e dunque velocizzare il processo**, riducendo così il tempo necessario per renderli disponibili al cluster. Diversa considerazione va fatta per le partizioni, nei test d'esempio infatti sono state **aggiunte 5 ulteriori partizioni per topic** in modo da verificare se la parallelizzazione influenzi positivamente anche queste misure. Quello che ci appare dai risultati dei test è che la **distribuzione** del carico di lavoro tra i broker influisce **positivamente** sulle misure solamente **se l'ammontare di comunicazioni che garantiscono l'interoperabilità dei broker non causa un overhead** tale da rallentare il processo di aggiunta di partizioni. Notiamo molto chiaramente come i tempi di modifica delle topic tendano ad aumentare molto di più nella configurazione con cinque broker, il ritardo è probabilmente causato dalle numerose operazioni che il sistema

effettua per coordinare tutti i broker a spartirsi le nuove partizioni. L'architettura ad un broker è caratterizzata da tempi leggermente inferiori, il tempo risparmiato a coordinare i broker in questo caso è utilizzato dal singolo broker carico di tutte le modifiche al cluster da effettuare. Opzione migliore è invece quella meglio bilanciata, i broker sono in numero giusto per spartirsi le responsabilità sulla creazione di nuove partizioni senza causare ritardi significativi per la cooperazione all'interno del cluster.

7.6 Considerazioni finali sui risultati sperimentali

Lo scopo di questo progetto è quello di mostrare in modo chiaro come un sistema così personalizzabile come Apache Kafka possa essere influenzato dal cambiamento di due caratteristiche fondamentali: il **numero di Broker** e il **numero di partizioni** che compongono le Topic. A tal fine, sono stati condotti numerosi test con un ampio spettro di variazioni di queste caratteristiche, al fine di individuare la configurazione più adatta in base alle specifiche esigenze.

Le caratteristiche su cui si è posta l'attenzione sono: la velocità con cui i messaggi attraversano il sistema, la capacità del sistema di elaborare molti messaggi in un secondo, la possibilità di bilanciare il carico su più dispositivi e la capacità di far fronte a downtime dovuti a malfunzionamenti, guasti o attacchi informatici.

Attraverso i test portati sulle varie configurazioni è stato possibile riconoscere i vari pro e contro che ognuna di esse può garantire durante l'esecuzione. Ad esempio se un'organizzazione che usa il sistema ha bisogno di mandare e ricevere pochi messaggi nel minor tempo possibile e i guasti sono estremamente rari e tollerabili, allora una configurazione con il minor numero di broker e il minor numero di partizioni possibili sarà quello più ideale. Se invece è imperativo avere lo scambio di una gran quantità di messaggi, dove i downtime sono probabili e i dati preziosi, allora sarà naturale optare per un maggior numero di broker e partizioni, soprattutto per le topic più affollate.

Nel caso in cui si ha la necessità di avere un via di mezzo tra tutte queste caratteristiche allora per trovare il bilanciamento corretto occorrerà più attenzione. Nel caso d'esempio un ottimo bilanciamento è quello di avere 3-5 broker e 3-7 partizioni, questo garantisce un ottimo throughput, un buon bilanciamento del carico, una maggiore sicurezza nel caso di

problemi, grazie anche alla replicazione delle partizioni, e dei tempi di latenza non troppo influenzati dalla maggiore complessità di gestione del sistema.

Per concludere quindi nell'utilizzare uno strumento simile è opportuno avere chiare le caratteristiche che si ritengono più importanti nel proprio sistema, un sistema più robusto e adatto alla gestione di molti messaggi infatti causerà un aumento della latenza di questi ultimi mentre un sistema più veloce agirà a discapito di sicurezza e gestione del carico.

Diversi studi hanno analizzato un'infrastruttura Kafka, testandola con vari algoritmi per determinare il numero ideale di broker e partizioni [41]. Questi studi considerano una vasta gamma di parametri, come la latenza di replicazione, il tempo di indisponibilità e il carico sul sistema operativo. Pertanto, se si decide di seguire un determinato algoritmo o di condurre test diretti sull'infrastruttura per trovare la configurazione più adatta, è fondamentale distinguere tra le caratteristiche essenziali e quelle meno critiche. Questo permette di ottenere un bilanciamento dell'infrastruttura personalizzato in base alle proprie esigenze.

Conclusioni e sviluppi futuri

Nel corso di questa tesi sono state esposte tutte le tecnologie e architetture che caratterizzano il mondo del cloud computing e delle applicazioni distribuite. Nei primi capitoli in particolare è stato ampiamente trattato il cloud computing con le sue caratteristiche principali e le più diffuse implementazioni. Successivamente sono stati introdotti i concetti di data streaming, come sono caratterizzati e il perché sono fondamentali, questo ha permesso di spiegare le architetture basate sugli eventi.

Tutto ciò ha gettato le basi per descrivere l'infrastruttura di Apache Kafka, che ha avuto un ruolo centrale nell'elaborato. Su questa piattaforma sono state create tutte le architetture successivamente testate e misurate. Di quest'ultima è stato esposto nel dettaglio il funzionamento seguito dalla spiegazione di tutti gli attori principali che compongono il cluster come: produttori, consumatori, broker e sulle strutture con le quali interagiscono come topic e partizioni. Successivamente è stato introdotto il concetto di osservabilità per un sistema distribuito, cos'è, a cosa serve e perché è fondamentale nel mondo del cloud. Si è poi parlato di uno strumento per realizzare l'osservabilità in ambiente distribuito quale OpenTelemetry, delle caratteristiche che lo descrivono e in che modo può essere integrato in un cluster Kafka.

Infine è stato spiegato nel dettaglio il progetto, gli obiettivi che si intendevano perseguire, la descrizione dell'infrastruttura Kafka usata come esempio e oggetto di test, in quale modo era stato integrata l'osservabilità con OpenTelemetry e quali attori principali componevano il cluster.

L'elaborato si è concluso con un'analisi dettagliata dei risultati dei test condotti sull'infrastruttura con un singolo broker, inizialmente suddividendo i topic in una, poi cinque e infine dieci partizioni. Successivamente, l'esperimento è stato ripetuto con tre e poi cinque broker, esaminando le prestazioni e il comportamento del sistema in ciascuna configurazione. È emerso che il comportamento varia significativamente con ogni configurazione, ciascuna delle quali presenta vantaggi e svantaggi specifici. Riuscire a trovare un equilibrio ottimale tra queste configurazioni non è un compito semplice.

Complice di questa difficoltà è anche la mancanza di uno standard specifico o un pattern da seguire. Apache Kafka nonostante fornisca alcune ottimizzazioni predefinite, non definisce

rigorosamente il modo in cui ciascuna topic deve essere distribuita in modo efficiente in partizioni. La messa a punto ottimale per migliorare le prestazioni di un cluster Apache Kafka rimane ancora un problema di ricerca aperto.

In futuro, questo progetto si propone di testare diverse configurazioni del sistema Kafka in un ambiente distribuito molto più ampio dove vi è alla base un più complicato intreccio di comunicazioni con molti più attori interessati allo scambio di messaggi. Saranno presenti dunque un ricco insieme di topic e partizioni con i quali produttori e consumatori interagiranno. Un'ambiente distribuito così complesso e simile ad applicazioni reali sarà ideale per testare se i risultati ottenuti sull'elaborato rimangono affidabili su larga scala.

Bibliografia e sitografia

[1] Peter Mell, Tim Grace, ‘The NIST Definition of Cloud Computing’, 2011

[2] Michael D. Hogan, Fang Liu, Annie W. Sokol, Tong Jin, ‘NIST Cloud Computing Standards Roadmap’, 2011

[3] T.B. Rehman, ‘Cloud Computing Bases’, 2018

[4] Alexa Huth, James Cebula, ‘The Basics of Cloud Computing’, 2011

[5] Derrick Rountree, Ileana Castrillo, ‘The Basics of Cloud Computing’, 2014

[6] “https://www.redhat.com/it/topics/cloud-computing/iaas-vs-paas-vs-saas”

[7]“https://www.spotfire.com/glossary/what-is-data-streaming#:~:text=Real%2Dtime%20stream%20processing%20techniques,and%20make%20time%2Dcritical%20decisions.”

[8]“https://www.informatica.com/content/dam/informatica-com/it/collateral/data-sheet/informatica-intelligent-streaming_data-sheet_3236it.pdf”

[9] “https://ensembleai.io/learn/streaming/lessons/challenges-with-stream-processing”

[10] Matthew N. O. Sadiku, Damilola S. Adesina and Sarhan M. Musa, ‘Big Data Streaming: An Introduction’, 2019

[11] Alessandro Margara, Tilmann Rabl, ‘Definition of Data Streams’, 2018

[12] “https://aws.amazon.com/it/what-is/streaming-data/”

[13] “https://www.redhat.com/it/topics/integration/what-is-streaming-data”

[14] “https://aws.amazon.com/it/event-driven-architecture/”

[15] “https://www.redhat.com/it/topics/integration/what-is-event-driven-architecture”

[16] Cindy Sridharan, ‘Distributed Systems Observability’, 2018

[17] “https://www.ibm.com/it-it/topics/observability”

[18] Tommaso Moroni, ‘Observability vs monitoring: combinare i due approcci per evitare downtime e disservizi’, 2022

[19] Charity Majors, Liz Fong-Jones, George Miranda, ‘Observability Engineering’, 2022

[20] “https://aws.amazon.com/it/what-is/log-files/”

[21] SolarWinds, ‘From Zero to Hero With Application Observability’

[22] “https://www.wallarm.com/what/what-is-distributed-tracing-full-guide”

[23] “https://kafka.apache.org/”

[24] Ankush Sharma, René Gómez, ‘Apache Kafka’, 2015

[25] “https://www.ibm.com/it-it/topics/apache-kafka”

[26]“https://www.ibm.com/docs/en/integration-bus/10.0?topic=applications-processing-kafka-messages”

[27] Neha Narkhede, Gwen Shapira, Todd Palino, 'Kafka the definitive guide', 2017

[28]“https://dattell.com/data-architecture-blog/what-is-a-kafka-topic/#:~:text=Kafka%20topics%20are%20the%20categories,consumers%20read%20data%20from%20topics.”

[29] ”https://medium.com/@cobch7/kafka-producer-and-consumer-f1f6390994fc”

[30] ”https://learn.conduktor.io/kafka/kafka-brokers/”

[31]“https://www.confluent.io/what-is-apache-kafka/?utm_medium=sem&utm_source=google&utm_campaign=ch.sem_br.nonbrand_tp.prs_tgt.kafka_mt.xct_rgn.emea_lng.eng_dv.all_con.kafka-general&utm_term=apache%20kafka&creative=&device=c&placement=&gad_source=1&gclid=Cj0KCQjwxsm3BhDrARIsAMtVz6Nntesbpgwv0_vAz6QjSFJBeGgB2RtbXsjbbHEupYYFBBeFRuZld0MaAnXoEALw_wcB”

[32] Roman Hemens, ‘Automatic Instrumentation With OpenTelemetry for Software Visualization’, 2024

[33] “https://www.dynatrace.com/news/blog/what-is-opentelemetry-2/”

[34] “https://opentelemetry.io/docs/concepts/components/”

[35] “https://www.servicenow.com/it/products/observability/what-is-opentelemetry.html#what-is-a-collector-in-opentelemetry”

[36] “https://opentelemetry.io/docs/zero-code/java/agent/extensions/”

[37] “https://www.instaclustr.com/blog/tracing-apache-kafka-with-opentelemetry/”

[38] “https://opentelemetry.io/blog/2022/instrument-kafka-clients/”

[39] “https://www.nexsoft.it/opentelemetry-tracing-microservizi/”

[40] “https://kafka.apache.org/documentation/”

[41] Theofanis P. Raptis, Andrea Passarella, ‘On Efficiently Partitioning a Topic in Apache Kafka’, 2022