Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il management

Ingegneria del Software per Sistemi Distribuiti: Architetture, Tecnologie e Best Practice

Relatore:
Dott. Davide Rossi

Presentata da:
Dominik Duda

Sessione III
Anno Accademico 2024/2025

A tutte le persone care,
per il loro supporto infinito

Indice

Capitolo 1

Introduzione

I **sistemi distribuiti** rappresentano una delle aree più avanzate e fondamentali nell'informatica moderna, costituendo la base per la realizzazione di applicazioni scalabili, resilienti e performanti in una vasta gamma di domini. Questi sistemi sono utilizzati per gestire enormi volumi di dati, per assicurare alta disponibilità e per fornire servizi in tempo reale a livello globale. La crescente necessità di operare in ambienti altamente dinamici e con risorse distribuite geograficamente ha reso fondamentale l'approfondimento di tecniche e tecnologie avanzate per la progettazione, la gestione e la manutenzione di sistemi distribuiti.

La seguente ricerca si propone di esplorare l'ambito dei sistemi distribuiti analizzando le **best practices** e le tecnologie emergenti, con l'obiettivo di fornire un quadro generale e dettagliato delle principali sfide e soluzioni. Lo studio analizza in particolare le questioni relative alla **comunicazione tra nodi**, alla **sincronizzazione dei dati**, alla **consistenza**, alla **tolleranza ai guasti** e alla **sicurezza**.

1.1 Obiettivi e Motivazione della Ricerca

L'obiettivo principale di questa ricerca è quello di fornire un'analisi approfondita delle caratteristiche, dei modelli architetturali e delle tecnologie che permettono di realizzare sistemi distribuiti efficaci. In particolare, la ricerca si concentra su:

- **Analizzare le sfide comuni nei sistemi distribuiti**, come la gestione della consistenza dei dati, la tolleranza ai guasti e la scalabilità.
- **Esplorare le tecnologie e gli strumenti moderni** per la gestione di sistemi distribuiti, come Docker, Kubernetes, Apache Kafka, e database distribuiti.
- **Valutare le pratiche di progettazione** più efficaci per costruire sistemi resilienti e ad alta disponibilità.

- **Indagare l'impatto della comunicazione e della sincronizzazione** tra nodi su performance e affidabilità dei sistemi distribuiti.

La motivazione principale dietro questa ricerca è l'importanza sempre crescente dei sistemi distribuiti nel nostro mondo digitale. È fondamentale trovare soluzioni che rispondano alle sfide emergenti, come la gestione di grandi volumi di dati e delle risorse in tempo reale, oltre a garantire operazioni sicure e senza intoppi. Inoltre, il continuo progresso delle tecnologie, insieme alla crescente complessità dei sistemi distribuiti, ci spinge a sviluppare approcci più avanzati e automatizzati per gestirli al meglio.

1.2 Metodologia e Approccio

La metodologia adottata per questa ricerca si basa su un'analisi della letteratura, che comprende articoli scientifici, report di settore, documentazione tecnica e white papers riguardanti i sistemi distribuiti e le relative tecnologie.

1.3 Struttura della Tesi

La tesi è strutturata come segue:

- **Capitolo 1 – Introduzione**: spiegazione degli obiettivi della tesi e la sua struttura.
- **Capitolo 2 - Fondamenti dei Sistemi Distribuiti**: Introduzione ai concetti di base dei sistemi distribuiti.
- **Capitolo 3 - Sincronizzazione dei Dati nei Sistemi Distribuiti**: Analisi delle tecnologie moderne per la gestione dei dati e la loro sincronizzazione con i vari nodi presenti in un sistema distribuito.
- **Capitolo 4 - Modelli Architetturali per Sistemi Distribuiti**: Analisi dei principali modelli architetturali di un sistema distribuito, inclusi i modelli Client-Server, Peer-to-Peer, Microservizi e Event-Driven.
- **Capitolo 5 - Tecnologie per Sistemi Distribuiti**: Analisi delle tecnologie moderne utilizzate per il deployment e la gestione di sistemi distribuiti, tra cui Docker, Kubernetes, Apache Kafka, e database distribuiti.
- **Capitolo 6 - Comunicazione nei Sistemi Distribuiti**: Discussione sui protocolli di comunicazione dei sistemi distribuiti, analizzando middleware,REST, gRPC e WebSockets.

- **Capitolo 7 - Best Practice nello Sviluppo di Sistemi Distribuiti**: Identificazione delle migliori pratiche nello sviluppo di sistemi distribuiti resilienti, sicuri e performanti, inclusi aspetti relativi a monitoraggio, resilienza e gestione dei guasti.
- **Capitolo 8 - Conclusioni e Sviluppi Futuri**: Sintesi dei risultati ottenuti e identificazione delle direzioni future per la ricerca e lo sviluppo nel campo dei sistemi distribuiti.

Questa struttura permette di affrontare in modo organico e completo le diverse problematiche e soluzioni legate ai sistemi distribuiti, offrendo al lettore una panoramica chiara e approfondita dell'argomento.

Capitolo 2

Fondamenti dei Sistemi Distribuiti

2.1 Definizione e caratteristiche principali dei sistemi distribuiti

Un sistema distribuito[2] è un'architettura computazionale composta da più entità autonome, tipicamente chiamate nodi di elaborazione distinti, che collaborano tramite una rete di comunicazione per eseguire operazioni distribuite con un obiettivo condiviso.

Questi sistemi implementano paradigmi di **concorrenza, parallelismo e comunicazione asincrona o sincrona**, utilizzando protocolli di coerenza e sincronizzazione per garantire la consistenza dei dati e la coordinazione tra nodi. I nodi possono essere unità fisiche indipendenti oppure essere istanze isolate eseguite su un'instrastruttura condivisa.

Le principali caratteristiche di un sistema distribuito includono:

- **Trasparenza**: Un sistema distribuito deve presentarsi come un'unica entità agli utenti, nascondendo la complessità dei nodi e delle comunicazioni tra di essi. Ciò include la trasparenza di accesso ovvero gli utenti non devono sapere dove risiedono le risorse e la trasparenza di replica dove gli utenti non devono preoccuparsi della replica dei dati.

- **Scalabilità:** Un sistema distribuito deve essere in grado di adattarsi facilmente per gestire un aumento del numero di nodi o della quantità di dati. La scalabilità riguarda vari aspetti tra cui la **scalabilità orizzontale** (aggiunta di nuovi nodi) e la **scalabilità verticale** (potenziamento dei nodi esistenti).

- **Affidabilità e tolleranza ai guasti**: La capacità del sistema di continuare a funzionare correttamente anche in caso di guasti di uno o più nodi. I sistemi distribuiti devono essere progettati in modo da garantire una completa **tolleranza ai guasti** e una **alta disponibilità**.

- **Concorrenza**: Essendo che più nodi lavorano contemporaneamente, un sistema distribuito deve essere in grado di gestire operazioni concorrenti e coordinare l'accesso alle risorse per evitare conflitti.
- **Gestione della latenza**: Ridurre al minimo i ritardi nella comunicazione tra nodi è fondamentale per ottenere prestazioni elevate.

L'implementazione di un sistema distribuito può basarsi su modelli architetturali eterogenei, tra cui **peer-to-peer, client-server, architetture a microservizi e computing su cloud**, con il fine di ottimizzare risorse computazionali, ridurre i colli di bottiglia e garantire disponibilità elevata anche in presenza di guasti parziali del sistema. Questi argomenti verranno trattati nei prossimi capitoli.

2.2 Differenza tra Sistemi Centralizzati e Sistemi Distribuiti

Quando si parla di architettura dei sistemi informatici, decidere se optare per un sistema centralizzato o uno distribuito è davvero una scelta cruciale. Questa decisione influisce non solo sulla progettazione, ma anche sulle prestazioni e sulla gestione complessiva dell'applicazione. Entrambi i sistemi mirano a soddisfare le esigenze degli utenti e delle applicazioni, ma gestiscono le risorse computazionali, la comunicazione tra i nodi e la manipolazione dei dati in modo molto diverso. Le scelte progettuali che caratterizzano un sistema centralizzato o distribuito rispondono infatti a requisiti e contesti specifici.

2.2.1 Sistemi Centralizzati

Un sistema centralizzato[17] è un'architettura in cui tutte le risorse computazionali, i dati e le funzionalità sono concentrati in un unico nodo o server centrale. In tale configurazione, i client o le applicazioni interagiscono direttamente con il server centrale per l'esecuzione delle operazioni e per l'accesso alle risorse condivise. Il server centrale ha il compito di gestire tutte le richieste dei client, di elaborare i dati e di mantenere la coerenza tra tutte le risorse.

Vantaggi:

Uno dei principali vantaggi[17] di un sistema centralizzato è la **semplicità nella progettazione e nella gestione**. Poiché tutte le risorse sono concentrate in un singolo punto, non è necessario gestire la complessità di una rete distribuita, con i relativi costi e sfide. La configurazione hardware e software risulta quindi meno complessa e, in generale, più facile da monitorare e mantenere. La **coerenza dei dati** è un altro aspetto che favorisce il modello centralizzato: poiché tutti i dati sono archiviati in un solo luogo,

la sincronizzazione tra le varie risorse non è necessaria, garantendo un elevato livello di integrità e consistenza dei dati. Inoltre, il controllo **centralizzato** delle risorse rende più semplice l'implementazione di politiche di sicurezza uniformi, senza la necessità di monitorare ogni singolo nodo. In termini di **prestazioni**, un sistema centralizzato offre un buon livello di stabilità, poiché tutte le risorse sono gestite e ottimizzate a livello di singolo punto.

Svantaggi:

Come possiamo notare, un sistema centralizzato presenta anche dei limiti importanti[17]. Il più evidente è il **single point of failure** (SPOF): se il server centrale smette di funzionare, l'intero sistema diventa inutilizzabile, con gravi ripercussioni sulla disponibilità e sull'affidabilità del servizio, questo implica un blocco totale o parziale del sistema. La **scalabilità** è un'altra sfida, poiché il nodo centrale ha risorse limitate: quando il numero di utenti o la quantità di dati cresce, il server centrale potrebbe non essere in grado di gestire il carico in maniera efficiente, provocando rallentamenti o malfunzionamenti. Un altro punto è la **latenza** perchè tutte le richieste devono passare attraverso il server centrale questo può provocare colli di bottiglia, specialmente se la rete è congestionata o se il server si trova geograficamente lontano dai client. Inoltre, la **resilienza** di un sistema centralizzato è più vulnerabile perchè un attacco esterno o un guasto hardware al nodo centrale può compromettere l'intero sistema.

2.2.2 Sistemi Distribuiti

In un sistema distribuito, le risorse e i dati sono distribuiti su più nodi, che possono essere geograficamente separati. Ogni nodo gestisce una parte del carico di lavoro e i nodi comunicano tra loro attraverso una rete. Questa configurazione permette una maggiore flessibilità rispetto ai sistemi centralizzati, poiché i nodi possono trovarsi in diverse ubicazioni, offrendo vantaggi in termini di **scalabilità** e **affidabilità**.

Vantaggi:

Una delle caratteristiche principali dei sistemi distribuiti è la loro **affidabilità e tolleranza ai guasti**. Grazie alla distribuzione dei dati e delle risorse, la perdita o il malfunzionamento di un singolo nodo non compromette il funzionamento dell'intero sistema. La capacità di replicare i dati su più nodi consente al sistema di continuare a operare, anche se uno o più nodi falliscono. Inoltre, i sistemi distribuiti sono **altamente scalabili**: sia orizzontalmente (aggiungendo nuovi nodi) che verticalmente (potenziando i nodi esistenti), la capacità di gestire carichi di lavoro crescenti è garantita. Questo li rende ideali per applicazioni che devono gestire grandi volumi di dati o numerosi utenti. I sistemi distribuiti permettono inoltre di eseguire operazioni in **parallelo**, migliorando significativamente le prestazioni, soprattutto in applicazioni che richiedono alte capacità di calcolo. Un altro vantaggio importante è la **flessibilità**: le architetture distribuite,

come quella a microservizi, consentono di progettare sistemi modulabili, in cui i singoli componenti possono essere aggiornati o scalati indipendentemente, senza interferire sull'intero sistema. Infine, grazie alla loro natura **geograficamente distribuita**, questi sistemi possono ridurre la latenza e migliorare l'accesso ai dati da diverse regioni del mondo.

Svantaggi:

Nonostante i vantaggi, i sistemi distribuiti presentano anche delle difficoltà. La gestione complessa è una delle sfide principali, visto che è necessario coordinare diversi nodi e assicurarsi che funzionino bene insieme. Un'altro punto è la sincronizzazione e la **coerenza dei dati** tra i nodi perchè i dati devono essere replicati e mantenuti consistenti su più punti, questo processo può portare ad una complessità maggiore e potenziali incoerenze. La gestione della **comunicazione** tra i nodi è un altro aspetto critico perchè ritardi di rete, congestione o errori di comunicazione possono influire negativamente sulle prestazioni e sulla stabilità del sistema. Inoltre, la **sicurezza** è più difficile da implementare in un sistema distribuito, ogni nodo rappresenta un potenziale punto di vulnerabilità perchè può essere attaccato dall'esterno. Ogni nodo deve essere protetto, e la gestione della sicurezza deve essere distribuita e costantemente monitorata. Infine, i **costi operativi e infrastrutturali** sono generalmente più elevati rispetto ai sistemi centralizzati perchè è necessaria una gestione completa dei nodi, della rete e delle risorse.

Conclusioni

La scelta tra un sistema centralizzato e uno distribuito dipende da numerosi fattori, tra cui le esigenze specifiche di scalabilità, la necessità di tolleranza ai guasti, la gestione dei dati e i vincoli di budget. I sistemi centralizzati offrono un'architettura più semplice e facile da gestire principalmente per applicazioni di piccole e medie dimensioni che non richiedono grandi capacità di scalabilità. Tuttavia, quando le applicazioni crescono in complessità e volume, un sistema distribuito può offrire maggiore flessibilità, affidabilità e scalabilità, sebbene a costo di una gestione più complessa. Con l'espansione delle tecnologie cloud e dei microservizi, i sistemi distribuiti stanno diventando sempre più la scelta principale per applicazioni ad alte prestazioni e ad alta disponibilità.

2.3 Classificazione dei sistemi distribuiti

È possibile classificare i tipi di sistemi distribuiti[24] in base alle loro soluzioni tecnologiche e agli obiettivi che hanno. Possiamo distinguere i sistemi distribuiti in:

- Distributed computing systems;
- Distributed information systems;

- Distributed pervasive systems.

Adesso parliamo un po' più in dettaglio delle singole classificazioni.

2.3.1 Distributed computing systems

L'espressione distributed computing[24] si riferisce a un'architettura digitale in cui più nodi collaborano per eseguire operazioni di calcolo. Anche se possono trovarsi lontani tra loro, questi computer operano in modo indipendente ma strettamente coordinato, suddividendo i compiti per lavorare in sinergia.

In questo ambito vi sono due grandi categorie:

- Cluster computing: Si tratta di un insieme di computer, generalmente simili tra loro e situati nella stessa area, collegati attraverso una rete dati ad alta velocità (LAN). Un esempio di applicazione del cluster computing è il calcolo delle previsioni meteorologiche. Lo European Centre for Medium Range Weather Forecasts (ECMWF), situato a Shinfield Park, Reading (Regno Unito), era composto da centinaia di migliaia di sistemi simili, interconnessi tramite una rete locale estremamente veloce, con capacità nell'ordine dei Terabit.
- Grid computing: A differenza del cluster computing, qui le risorse sono eterogenee e possono appartenere a istituzioni diverse. Essendo distribuite geograficamente, queste risorse vengono collegate tramite reti di ampia scala, come Internet. Un esempio noto di grid computing è il progetto SETI, dedicato alla ricerca di segnali di vita extraterrestre.

La programmazione dei sistemi di distributed computing avviene tramite librerie specifiche, che semplificano l'interazione con l'infrastruttura fisica sottostante. La scelta tra un'architettura cluster e grid dipende dalla quantità di dati che devono essere scambiati tra i nodi. Se il volume di dati è elevato, è più adatto il cluster computing. Al contrario, se i dati sono più sparsi, si può optare per il grid computing.

2.3.2 Distributed information systems

I sistemi di informazione distribuita[24] hanno l'obiettivo di gestire dati che sono sparsi su più sistemi. In altre parole, i dati sono distribuiti su diversi nodi, ma devono essere gestiti in modo che appaiano come se fossero parte di un unico sistema.

Un esempio classico di questo tipo di architettura è il database distribuito.

Si possono avere implementazioni:

- Distributed transaction processing (TPS: transaction processing system);
- Enterprise application integration (EAI)

Nel Distributed Transaction Processing, nei sistemi transazionali si applicano le proprietà ACID (Atomicità, Coerenza, Isolamento, Durabilità) verso un database.

I sistemi transazionali distribuiti presentano delle sfide intrinseche maggiori rispetto a quelli centralizzati. Ad esempio, si può avere una transazione che coinvolge più basi dati, ciascuna localizzata su sistemi diversi. Se una transazione viene completata solo parzialmente in una base dati, è necessario annullare le modifiche effettuate in tutte le basi dati precedenti, anche se quelle operazioni sono state concluse correttamente.

Per implementare sistemi transazionali distribuiti, si utilizza solitamente un Transaction Processing Monitor (TPM). Questo sistema centrale raccoglie le richieste e le distribuisce ai vari sistemi, monitorando attentamente lo stato di ciascuna operazione. In questo modo, mentre l'informazione è distribuita, la logica di controllo rimane centralizzata.

Un'altra alternativa consiste nell'integrare le applicazioni anziché i database, dando vita al concetto di Enterprise Application Integration. In questo caso, diversi server sono dotati di interfacce di comunicazione peer-to-peer. Le informazioni desiderate vengono recuperate accedendo ai vari server che le contengono e poi combinate per creare il risultato finale. Questo approccio risulta essere più complesso rispetto al TPM.

2.3.3 Distributed pervasive systems

I sistemi distribuiti pervasivi[3][24] fanno parte del nostro ambiente quotidiano. I dispositivi che li compongono devono essere in grado di scoprire i servizi disponibili e adattarsi al contesto circostante, utilizzando anche informazioni provenienti dall'esterno.

Esempi di sistemi distribuiti pervasivi sono:

- Sistemi domestici;
- Sistemi elettronici per assistenza sanitaria;
- Reti di sensori wireless.

I *sistemi domestici* (noti anche come *smart home systems*) sono reti di dispositivi connessi in modo intelligente all'interno di una casa. Questi dispositivi sono in grado di comunicare tra loro per ottimizzare le funzioni quotidiane e migliorare il comfort, la sicurezza e l'efficienza energetica. Esempi tipici includono:

- **Termostati intelligenti** che regolano la temperatura della casa in base alle preferenze dell'utente e alle condizioni ambientali;
- **Luci intelligenti** che si accendono o si spengono automaticamente in base alla presenza di persone o al momento della giornata;
- **Sistemi di sicurezza** che monitorano la casa tramite telecamere e sensori di movimento e inviano avvisi in caso di intrusione;
- **Elettrodomestici intelligenti** che si attivano in modo autonomo per ottimizzare i consumi energetici o la gestione delle risorse (ad esempio, frigoriferi che monitorano scadenze alimentari o lavatrici che avviano cicli basati sull'orario).

Questi sistemi sono considerati pervasivi perché operano in background, adattandosi all'ambiente in tempo reale senza richiedere un'azione diretta da parte degli utenti.

I *sistemi elettronici per l'assistenza sanitaria* (Electronic Health Care Systems)[3] fanno parte di un approccio in cui i dispositivi medici e i sensori sono integrati per monitorare costantemente lo stato di salute di un paziente e trasmettere i dati raccolti a sistemi centralizzati per l'elaborazione e il controllo. Alcuni esempi includono:

- **Sensori indossabili**, come braccialetti o smartwatch, che monitorano parametri vitali come la frequenza cardiaca, la pressione sanguigna, l'ECG (elettrocardiogramma) o il livello di ossigeno nel sangue;
- **Body-Area Networks (BAN)**, che collegano vari sensori indossabili (come un sensore di movimento, sensori di temperatura, ecc.) a un hub centrale che raccoglie e analizza i dati;
- **Telemedicina**, dove i dati vengono trasmessi a medici o ospedali per monitorare i pazienti a distanza, consentendo diagnosi rapide e interventi tempestivi senza necessità di spostamenti fisici.

A differenza dei sistemi domestici, i sistemi sanitari devono gestire una grande quantità di dati sensibili e necessitano di una gestione e elaborazione dei dati *in-network* (cioè direttamente nel luogo dove i dati sono generati), per garantire un'elaborazione rapida e sicura senza eccessivi ritardi.

Le *reti di sensori wireless* (Wireless Sensor Networks - WSN) sono composte da piccoli sensori autonomi distribuiti in un'area geografica che rilevano dati fisici o ambientali (come temperatura, umidità, movimento, inquinamento, ecc.). I sensori raccolgono i dati in modo cooperativo e li trasmettono a un sistema centrale per l'elaborazione. Queste reti sono molto utilizzate in contesti come:

- **Monitoraggio ambientale**, dove i sensori misurano parametri come la qualità dell'aria, la temperatura, l'umidità, ecc.;
- **Sicurezza e sorveglianza** in ambienti pubblici o industriali, dove sensori di movimento e telecamere monitorano le aree per identificare situazioni anomale;
- **Agricoltura intelligente**, dove i sensori monitorano le condizioni del terreno e dell'ambiente per ottimizzare l'irrigazione e l'uso di fertilizzanti.

Le WSN hanno risorse limitate, come potenza di calcolo e memoria, quindi devono essere progettate per lavorare in modo efficiente e con un consumo energetico ridotto. I dati raccolti possono essere inviati a un operatore centrale, ma per risparmiare energia, di solito si adottano modelli di comunicazione come il pattern `pub/sub` (publish/subscribe).

Capitolo 3

Sincronizzazione dei Dati nei Sistemi Distribuiti

Parlare di sistemi distribuiti significa confrontarsi con una delle sfide più importanti: la sincronizzazione dei dati. Questi sistemi sono caratterizzati dal fatto che i dati sono condivisi tra diversi nodi e devono essere mantenuti coerenti e aggiornati in tempo reale. A volte, però, possono sorgere problemi come latenze o guasti. Qui entra in gioco la sincronizzazione, che si occupa di fare in modo che tutti i nodi abbiano una visione chiara e allineata dei dati. Questo è cruciale per evitare errori nei processi aziendali e garantire il corretto funzionamento dei servizi.

3.1 Clock nei Sistemi Distribuiti

3.1.1 Clock Fisici e la Sincronizzazione dei Clock

La sincronizzazione dei clock fisici tra i nodi di un sistema distribuito è essenziale per gestire l'ordinamento temporale degli eventi e delle operazioni. In un ambiente distribuito, i nodi non hanno un clock comune e possono affrontare ritardi diversi nelle comunicazioni. Questo significa che le informazioni temporali devono essere sincronizzate per evitare discrepanze. In un sistema distribuito, non è possibile avere un clock fisico unico che funzioni per tutti i processi.

Per sincronizzare i **clock fisici**, vengono utilizzati protocolli come:

- **NTP (Network Time Protocol)**[19]: è uno dei protocolli più comuni per la sincronizzazione dei clock, che permette ai nodi di un sistema distribuito di allineare i loro clock base a un clock globale. NTP è adatto per applicazioni che non richiedono una sincronizzazione ad alta precisione, con una **precisione** di ordine **millisecondo**.

- **PTP (Precision Time Protocol)**[20]: un protocollo più preciso rispetto a NTP, utilizzato per sincronizzare i clock con una **precisione sub-microsecondo**. Comunemente impiegato in applicazioni industriali o scientifiche che richiedono una sincronizzazione estremamente precisa.

Tuttavia, la sincronizzazione dei clock fisici in un ambiente distribuito è spesso complicata da **latenze di rete**, **errore di sincronizzazione** e **instabilità** della rete stessa. Per questo motivo, la sincronizzazione **logica** è spesso preferita in alcuni ambiti.

Analizzeremo ora in dettaglio dei due protocolli menzionati precedentemente: il NTP e PTP.

NTP (Network Time Protocol) [19, 18] è un protocollo descritto nel documento RFC 958 relativo alla **sincronizzazione dei clock** nei sistemi informatici. Si basa sul protocollo senza connessione UDP (porta 123) e appartiene alla famiglia dei protocolli Internet. Per il processo di sincronizzazione NTP utilizza il tempo coordinato universale (UTC) adottato dai singoli client e server secondo un sistema gerarchico.

L'obiettivo principale del protocollo NTP è quello di assicurare che tutti i dispositivi all'interno di una rete condividano lo stesso orario esatto. Per raggiungere questo scopo, NTP utilizza un sistema gerarchico suddiviso in diversi strati. I dispositivi ai livelli superiori (strato 1) si sincronizzano direttamente con fonti temporali di altissima precisione, come gli orologi atomici e i satelliti GPS, e quindi distribuiscono queste informazioni ai livelli inferiori (strati 2, 3 e così via).

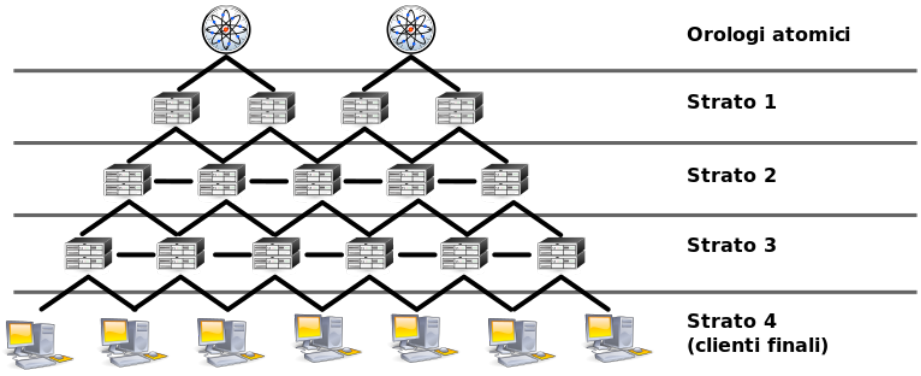

Figura 3.1: Struttura gerarchica divisa a strati del NTP[7]

La **struttura gerarchica** del NTP[7] è organizzata in **strati**, che definiscono la distanza di un dispositivo dalla fonte temporale primaria:

- **Strato 0:** clock di riferimento, come orologi atomici o ricevitori GPS, che forniscono l'ora precisa.

- **Strato 1:** Server collegati direttamente agli strati 0 e che distribuiscono l'ora esatta ad altri dispositivi.
- **Strato 2 e seguenti:** Dispositivi che ottengono l'ora dai server ai livelli superiori e la ritrasmettono a quelli inferiori.

Il protocollo è capace di gestire fino a 15 livelli di strati. Tuttavia, va detto che con ogni livello in più, il ritardo tende a crescere e la precisione diminuisce. NTP funziona seguendo un modello client-server, utilizzando uno scambio di timestamp per calcolare la differenza tra gli orologi dei dispositivi e per regolarli quando necessario. Questo processo si articola in quattro fasi principali:

- Il client NTP invia una richiesta di sincronizzazione al server NTP includendo un timestamp di origine.
- Il server registra l'ora esatta in cui riceve la richiesta e aggiunge un timestamp di ricezione.
- Il server risponde al client con il proprio timestamp di trasmissione.
- Il cliente registra l'orario di arrivo del pacco e utilizza questi dati per calcolare il ritardo e adeguarsi di conseguenza.

Questo scambio di informazioni consente all'NTP di mantenere a **precisione fino a 10 millisecondi** su reti Internet pubbliche e persino **microsecondi** nelle reti locali.

Il **Precision Time Protocol (PTP)**[22] è uno standard sviluppato dall'IEEE (Institute of Electrical and Electronics Engineers) con lo scopo di garantire una sincronizzazione precisa degli orologi dei dispositivi all'interno di una rete di telecomunicazioni.

Questo protocollo è essenziale per quelle applicazioni che richiedono un'altissima precisione temporale, come l'automazione industriale, i sistemi di misurazione e controllo, le reti finanziarie e le infrastrutture di telecomunicazione.

Il PTP funziona attraverso un **meccanismo di sincronizzazione a feedback chiuso**. In questa configurazione, un dispositivo secondario, noto come slave, regola continuamente il proprio orologio sulla base dei riferimenti temporali forniti da un dispositivo principale, chiamato master. Così facendo, lo slave riceve segnali di temporizzazione dal master e li utilizza per ottimizzare il proprio orologio, stabilendo un ciclo di regolazione costante.

La sincronizzazione avviene mediante uno scambio continuo di messaggi tra il Grandmaster Clock (GM) e i dispositivi subordinati (Slave Clocks). Il Grandmaster Clock invia timestamp di precisione elevatissima agli slave, che li usano per allineare i loro orologi locali al riferimento principale. Questo processo di aggiustamento continua fino a quando gli orologi subordinati non raggiungono un perfetto allineamento con il Grandmaster

Clock, garantendo così una sincronizzazione temporale estremamente accurata all'interno della rete.

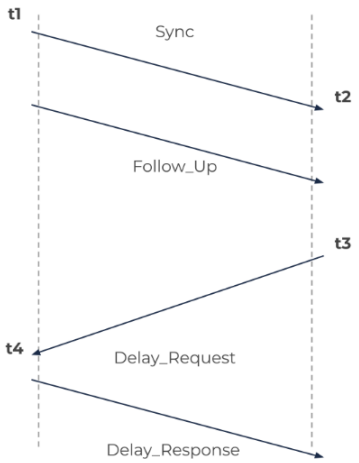

Figura 3.2: Struttura dei messaggi nel PTP

Il PTP ha bisogno di alcune fasi per procedere alla sincronizzazione[23]:

- Announce: Il Grandmaster Clock annuncia il suo stato e la sua qualità.
- Sync: Gli clock schiavi iniziano a sincronizzare i propri clock in base ai timestamp inviati dal Grandmaster Clock.
- Follow-Up: Il Grandmaster Clock invia ulteriori informazioni di timing.
- Delay Request: Gli clock schiavi chiedono al Grandmaster Clock il ritardo che si verifica durante la trasmissione dei pacchetti.
- Delay Response: Il Grandmaster Clock risponde con il ritardo.

Ogni sequenza PTP coinvolge una serie di quattro messaggi tra ptp master e ptp slave:

- Sync message dal **ptp master** al **ptp slave**
- Followup sync message dal **ptp master** al **ptp slave**
- Delay request message dal **ptp slave** al **ptp master**

- Delay response message dal **ptp master** al **ptp slave**

Questa sequenza produce quattro **timestamp** diversi:

- T1 quando il **master ptp** invia un Sync message
- T2 quando lo **slave ptp** riceve il Sync message
- T3 quando lo **slave ptp** invia un messaggio Delay request
- T4 quando il **master ptp** riceve il messaggio Delay request

Il PTP masterclock invia i suoi quattro timestamp allo slave PTP durante la fase di delay response. In questo modo, lo slave è in grado di calcolare la latenza di rete tra il master e lo slave in entrambe le direzioni. Grazie a hardware dedicato, i dispositivi PTP slave possono evitare di introdurre latenze extra legate al sistema operativo..

Il PTP ha moltissimi vantaggi, e la sua precisione è senza dubbio il suo punto di forza, arrivando a livelli incredibili, fino al nanosecondo, a seconda di come viene implementato e della qualità della rete. Ma non è tutto: questo protocollo è noto anche per la sua scalabilità e l'efficienza, il che lo rende perfetto per una serie di situazioni diverse. Può essere utilizzato in reti di qualsiasi dimensione, dalle piccole reti locali alle WAN più ampie. Inoltre, aiuta a ridurre il numero di dispositivi hardware necessari per la sincronizzazione, che spesso possono essere costosi, migliorando così l'efficienza complessiva della rete.

3.1.2 Comparazione tra NTP e PTP

Il **Network Time Protocol (NTP)** e il **Precision Time Protocol (PTP)** sono entrambi progettati per la sincronizzazione degli clock sui dispositivi di rete, ma presentano alcune differenze fondamentali nelle loro caratteristiche e modalità di funzionamento.

Di seguito una tabella riassuntiva con tutti i vari elementi citati in precedenza messi a confronto[23]:

Caratteristica	NTP (Network Time Protocol)	PTP (Precision Time Protocol)	Conclusioni
Precisione	Millisecondi / microsecondi	Microsecondi / nanosecondi	Il PTP è generalmente più preciso rispetto all'NTP. La precisione effettiva dipende anche da altri fattori.
Architettura	Feedback aperto: nessun controllo attivo sul flusso di sincronizzazione	Feedback chiuso: il Grandmaster Clock controlla ogni singolo slave	Il PTP garantisce maggiore precisione e sicurezza grazie alla sua architettura.
Applicazioni	Adatto per applicazioni senza esigenze di sincronizzazione estrema	Indispensabile per applicazioni che richiedono massima precisione	La scelta dipende dai requisiti specifici dell'applicazione.
Complessità	Semplice da configurare e integrare nei sistemi esistenti	Più complesso da configurare e implementare	Maggiore precisione implica una configurazione più complessa.

3.1.3 Clock Logici e Ordinamento degli Eventi

Lo strumento utilizzato per affrontare il problema di sincronizzazione dei clock fisici nei sistemi distribuiti è il clock logico, basato sull'ordinamento temporale feedback. Lamport e Vector sono i più diffuse:

- **Clock di Lamport**: è un sistema di numerazione degli eventi, che permette di determinare un **ordine causale**. Ogni nodo mantiene un **contatore logico** che viene incrementato ogni volta che un evento accade per mantenere l'ordinamento tra gli eventi. I clock di Lamport sono efficaci nel dare una certezza sull'ordinamento degli eventi, ma non sono in grado di risolvere conflitti simultanei di operazioni concorrenti.
- **Vector Clock**: è un'estensione dei clock di Lamport, i **vector clock** sono più complessi in cambio offrono una risoluzione migliore dei conflitti simultanei. Ogni nodo memorizza un vettore di contatori, uno per ogni nodo nel sistema. Il vettore di contatori è più informativo dello stato di tracciamento degli eventi causali perché consente di rilevare e risolvere correttamente i conflitti di scrittura.

Lamport ha introdotto il concetto di ordinamento parziale per stabilire un criterio temporale nei sistemi distribuiti. Alla base di questa idea c'è la relazione *happens-before*,

indicata come $a \rightarrow b$ (a happens-before b), che significa che l'evento a avviene prima dell'evento b.

Questa relazione può essere determinata in due casi principali:

- Se due eventi, a e b, avvengono nello stesso processo e a si verifica prima di b, allora vale $a \rightarrow b$.
- Se un processo invia un messaggio (evento a) e un altro lo riceve (evento b), allora anche in questo caso si ha $a \rightarrow b$.

Inoltre, la relazione *happens-before* è transitiva, il che significa che se $a \rightarrow b$ e $b \rightarrow c$, allora necessariamente $a \rightarrow c$.

D'altra parte, se due eventi a e b appartengono a processi distinti che non comunicano tra loro, non è possibile stabilire alcuna relazione di *happens-before* tra di essi.

Nei sistemi distribuiti, i processi operano indipendentemente l'uno dall'altro, e non esiste un clock globale che permetta di stabilire un ordine assoluto degli eventi. Per ovviare a questo problema, Lamport ha introdotto i logical clock, un meccanismo che assegna un valore temporale a ogni evento, in modo da poterli ordinare coerentemente.

L'idea è che ogni processo mantiene un proprio clock logico C che segue questa regola fondamentale: se $a \rightarrow b$, allora il timestamp assegnato a a deve essere inferiore a quello di b, ovvero $C(a) < C(b)$.

Un principio essenziale dei logical clock è che il tempo non può mai diminuire: il valore del clock cresce sempre e non viene mai ridotto. Questo garantisce che l'ordine degli eventi venga sempre rispettato.

Il meccanismo dei logical clock viene gestito tra il livello applicativo e il livello di rete che trasmette i messaggi. Il funzionamento avviene in tre semplici passi:

1. **Aggiornamento del clock locale:** prima di generare un evento, il processo incrementa il proprio logical clock:

$$C_i = C_i + 1 \tag{3.1}$$

2. **Invio di un messaggio:** quando un processo invia un messaggio, gli assegna un timestamp pari al valore del proprio logical clock:

$$ts(m) = C_i \tag{3.2}$$

3. **Ricezione di un messaggio:** quando un processo riceve un messaggio, confronta il proprio clock con il timestamp ricevuto e adotta il valore più alto, poi lo incrementa di uno:
$$C_j = \max(C_j, ts(m)) + 1 \tag{3.3}$$

Una volta aggiornato il clock, il messaggio viene passato all'applicazione.

Questo sistema permette di stabilire un ordine coerente tra gli eventi in un sistema distribuito. Anche se non fornisce un'ordinazione assoluta di tutti gli eventi (dato che alcuni possono essere considerati concurrent, ossia senza un ordine definito), i logical clock garantiscono che, se un evento ha causato un altro, l'ordine venga rispettato.

Grazie a questa tecnica, è possibile mantenere la consistenza temporale nei sistemi distribuiti, assicurando che i processi possano interpretare gli eventi in un ordine coerente senza bisogno di un clock globale.

Questi approcci risolvono problemi legati all'ordinamento degli eventi e alla gestione delle **concorrenze**, ma richiedono una gestione continua e un'interazione frequente tra i nodi per garantire che i vettori siano aggiornati correttamente.

3.2 Modelli di Consistenza nei Sistemi Distribuiti

3.2.1 Consistenza Forte

Un modello di consistenza forte garantisce che tutte le letture dei dati riflettano l'ultima scrittura. In altre parole, ogni volta che un nodo scrive dati, è garantito che tutti i dati scritti successivamente siano visibili a tutti i nodi. Questo modello è molto utile per le applicazioni in cui è necessario avere una visione coerente dei dati in tempo reale, come nei sistemi bancari o finanziari.

Vantaggi:

- **Coerenza immediata**: tutte le richieste di lettura forniscono i dati più recenti.
- **Mantenimento semplificato**: non è necessario gestire conflitti o versioni multiple dei dati.

Svantaggi:

- **Scalabilità limitata**: per garantire una forte coerenza, tutti i nodi devono comunicare e condividere informazioni continuamente, il che può causare qualche ritardo durante le richieste.
- **Latenza di scrittura**: possono esserci operazioni di scrittura significativamente più lente perchè ogni nodo deve riconoscere la scrittura.

3.2.2 Consistenza Eventuale

La **consistenza eventuale** [26] è un modello in cui i dati sono sincronizzati tra i nodi, ma non è garantito che tutte le letture restituiscano l'ultimo valore scritto immediatamente. Questo modello è usato in sistemi che necessitano di **alta disponibilità** e **bassa latenza**, come nel caso di **social media** o **motori di ricerca**.

Vantaggi:

- **Alta disponibilità e performance**: i nodi possono continuare a operare anche se alcuni di essi sono temporaneamente non disponibili.
- **Scalabilità**: è più facile aggiungere nuovi nodi senza compromettere le prestazioni.

Svantaggi:

- **Incoerenza temporanea**: i dati potrebbero non essere immediatamente visibili su tutti i nodi.
- **Gestione dei conflitti complessa**: quando i nodi sono sincronizzati, possono sorgere conflitti.

3.2.3 Consistenza Causale

La **consistenza causale** cerca di mantenere un ordine logico degli eventi utilizzando le sue relazioni tra gli eventi. Un esempio di questo modello è la sincronizzazione di applicazioni che richiedono la gestione di **versioni conflittuali** di dati, come nei **sistemi di versionamento** dei documenti come github.

Vantaggi:

- **Maggiore coerenza rispetto alla consistenza eventuale**: gli eventi causali sono ordinati, evitando conflitti tra operazioni indipendenti.
- **Supporta la concorrenza**: è adatto a sistemi che operano su **operazioni concorrenti**.

Svantaggi:

- **Maggiore complessità**: richiede una gestione avanzata degli clock logici, come i **vector clock**.
- **Potenziale per incoerenze temporanee**: anche con la consistenza causale, possono esistere discrepanze tra i dati di diversi nodi.

3.3 Tecniche di Sincronizzazione dei Dati

3.3.1 Algoritmi di Sincronizzazione

Gli algoritmi di sincronizzazione dei dati sono progettati per garantire che i dati vengano aggiornati e replicati correttamente tra i nodi del sistema. Alcuni degli algoritmi più utilizzati includono:

- **Quorum-based Replication**: un algoritmo in cui un'operazione di scrittura è considerata valida solo se viene confermata da una certa percentuale di nodi. Allo stesso modo, una lettura è valida solo se è effettuata su un quorum di nodi.
- **Two-Phase Commit (2PC)**: un protocollo di coordinamento che garantisce che tutte le operazioni di scrittura siano completate con successo su tutti i nodi interessati.
- **Paxos e Raft**[1, 25]: algoritmi di consenso che permettono ai nodi di raggiungere un accordo su un singolo valore, garantendo che il sistema resti consistente anche in caso di guasti.

3.3.2 Gestione dei Conflitti

Nei sistemi distribuiti che utilizzano la consistenza eventuale, la gestione dei conflitti è un aspetto cruciale. Ci sono diversi approcci per risolvere i conflitti, tra cui:

- **Last-Write-Wins (LWW)**: in caso di conflitto, si considera valida solo l'ultima scrittura effettuata.
- **Operazioni di Merge**: i conflitti vengono risolti unendo i dati, come nel caso di unire due versioni di un documento.

3.3.3 Vantaggi e Svantaggi della Sincronizzazione dei Dati

Vantaggi

- **Migliore coerenza dei dati:** Una corretta sincronizzazione assicura che tutti i nodi abbiano una visione coerente dei dati, prevenendo errori di lettura e scrittura.
- **Affidabilità:** La sincronizzazione assicura che ci sia un'esecuzione affidabile delle operazioni, il che diminuisce le probabilità di incoerenze.
- **Maggiore disponibilità e scalabilità**: I sistemi distribuiti ben sincronizzati sono più scalabili e in grado di gestire un volume maggiore di richieste simultanee mantenendo comunque la disponibilità dei dati.

Svantaggi

- **Sovraccarico e complessità:** La sincronizzazione richiede l'uso di algoritmi complessi, che possono comportare un sovraccarico nei sistemi.
- **Scalabilità limitata (per coerenza forte**): I sistemi con coerenza forte hanno solitamente problemi di scalabilità.
- **Conflitti di scrittura:** Nei modelli di coerenza eventuale, i conflitti tra nodi possono essere piuttosto difficili da risolvere.

È fondamentale prestare grande attenzione alla sincronizzazione dei dati, perché è essenziale per il buon funzionamento di un sistema distribuito. In un contesto dove i dati sono replicati su più nodi, garantire coerenza e affidabilità delle informazioni diventa davvero critico. Ci sono diversi modelli di coerenza, come la consistenza forte e quella eventuale, ognuno con pregi e difetti. I requisiti specifici di un'applicazione determinano quale modello sia più appropriato.

Utilizzare gli orologi vettoriali o gli orologi di Lamport permette di segnare le relazioni causali tra le operazioni, riducendo i conflitti e assicurando che gli eventi siano sequenziali in modo corretto. Questo aiuta a gestire l'ordine temporale degli eventi, creando una rappresentazione causale degli eventi senza conflitti grazie all'uso di orologi fisici e logici.

Tecnologie come gli algoritmi Paxos o Raft e altri che si basano su quorum forniscono meccanismi per mantenere i dati allineati su tutti i nodi attivi anche in caso di guasti e ritardi nella comunicazione, assicurando così che i sistemi altamente geo-distribuiti e scalabili rimangano operativi. Anche se questi meccanismi migliorano le prestazioni nella risoluzione dei conflitti, la latenza in coda continua a essere una sfida aperta nel contesto dei modelli di consistenza eventuale.

In conclusione, quando si tratta di sincronizzazione dei dati, c'è sempre un compromesso che i progettisti devono affrontare, il che può variare a seconda del dominio applicativo in termini di coerenza, disponibilità e scalabilità, come indicato dal teorema CAP. In un sistema distribuito, questi approcci dovrebbero cercare di bilanciare le prestazioni ottimizzando l'incoerenza al minimo possibile.

In sintesi, affrontare la mancanza di coerenza nei dati con algoritmi moderni è una questione complessa da risolvere. D'altra parte, progettare sistemi distribuiti presenta diverse contraddizioni tecniche, ma grazie a queste si riesce a controllare l'essenza della coerenza, rendendo le applicazioni scalabili, resilienti e affidabili.

Capitolo 4

Modelli Architetturali per Sistemi Distribuiti

Quando si parla di **sistemi distribuiti**, una delle prime cose da considerare è **come** questi sistemi vengono organizzati. In altre parole, come i diversi componenti comunicano tra loro, come vengono gestiti i dati e in che modo il sistema è strutturato per garantire prestazioni, affidabilità e scalabilità.

Le architetture dei sistemi distribuiti[9] possono variare molto a seconda dell'obiettivo che devono raggiungere. Alcune sono pensate per **ottimizzare la velocità e ridurre la latenza**, altre per **gestire grandi quantità di dati**, mentre altre ancora puntano a **garantire affidabilità e tolleranza ai guasti**.

Di seguito vedremo i principali **modelli architetturali** utilizzati nei sistemi distribuiti, cercando di capire **quando e perché** sceglierne uno piuttosto che un altro.

4.1 Modello Client-Server

Il modello client-server è una delle architetture più comuni e verificate nei sistemi distribuiti. In questo paradigma, un'entità centrale, il server, è responsabile della gestione dei dati e dei servizi, mentre i client interagiscono con esso mandando le richieste e ricevendo risposte. Il server funge da nodo principale per la comunicazione e il coordinamento delle operazioni.

Una delle principali caratteristiche di questo modello è la separazione dei ruoli tra client e server: i client non interagiscono tra loro, ma dipendono dal server per accedere alle risorse e ai servizi. Questa organizzazione consente un maggiore controllo sulla sicurezza e sull'accesso ai dati, rendendo il sistema più prevedibile e gestibile. Il problema principale e che l'affidabilità complessiva dipende principalmente dal server centrale, che rappresenta un punto critico: eventuali guasti o sovraccarichi possono compromettere

l'intero sistema.

L'implementazione di un'architettura client-server è relativamente semplice, il che la rende ideale per molte applicazioni web, database centralizzati e servizi cloud. Tuttavia, la scalabilità può diventare problematica perchè con l'aumento del numero di client, il server può diventare un collo di bottiglia, influenzando le prestazioni e aumentando la latenza delle risposte. Per affrontare questo problema, di solito si ricorre a strategie come la replica dei server o il bilanciamento del carico.

4.2 Architettura Peer-to-Peer

L'architettura peer-to-peer (P2P) si distingue dal modello client-server per la totale decentralizzazione della gestione delle risorse. In questo paradigma, ogni nodo della rete può agire sia come client che come server, partecipando attivamente alla condivisione e distribuzione delle informazioni. Cosi da eliminare l'entità centrale che coordina le operazioni, e permettere che i nodi comunichino direttamente tra loro per scambiarsi dati e servizi.

Questa architettura offre una scalabilità potenzialmente illimitata perchè all'aumentare del numero di nodi, aumenta anche la capacità complessiva del sistema. Inoltre, la tolleranza ai guasti è migliorata rispetto al modello client-server perchè l'assenza di un unico punto di controllo riduce il rischio di interruzioni totali del servizio. Tuttavia, la gestione della sicurezza e della coerenza dei dati diventa più complessa,i nodi devono coordinarsi autonomamente per garantire l'affidabilità delle informazioni scambiate.

L'architettura P2P è particolarmente efficace in applicazioni che richiedono una distribuzione efficiente delle risorse, come reti di file sharing (ad esempio BitTorrent), sistemi di comunicazione decentralizzati e criptovalute come Bitcoin. Però la mancanza di un controllo centralizzato può rendere molto difficile il monitoraggio e la regolazione degli accessi, aumentando il rischio di utilizzi impropri della rete.

4.3 Architettura a Microservizi

L'architettura a microservizi utilizza un approccio modulare, suddividendo un'applicazione in componenti indipendenti, chiamati microservizi, ciascuno dedicato a una specifica funzionalità. Ogni microservizio opera autonomamente e può essere sviluppato, distribuito e scalato in modo indipendente dagli altri.

Questa architettura offre un'elevata flessibilità e scalabilità perchè ogni servizio può essere implementato utilizzando tecnologie differenti, a seconda delle esigenze specifiche.

La comunicazione tra microservizi avviene attraverso API REST, gRPC o messaggi asincroni, garantendo un'alta interoperabilità tra componenti. Grazie alla separazione dei servizi, la resilienza del sistema è notevolmente migliorata quindi se un microservizio subisce un malfunzionamento, gli altri possono continuare a operare senza interruzioni.

Tuttavia, questa architettura introduce una maggiore complessità nella gestione e nel coordinamento dei servizi. La necessità di orchestrare molteplici componenti richiede strumenti avanzati per il monitoraggio, il logging e la gestione degli errori. Inoltre, la comunicazione tra i microservizi può generare overhead, con possibili impatti sulle prestazioni complessive. Per questo motivo, le architetture a microservizi vengono spesso adottate in combinazione con sistemi di orchestrazione come Kubernetes, che automatizzano il deployment e la gestione dei servizi distribuiti. Parleremo nei capitoli successivi di Docker, Kubernetes, API REST, gRPC.

4.4 Architettura Event-Driven

L'architettura event-driven[9] si basa sulla comunicazione asincrona tra componenti attraverso eventi e messaggi. In questo modello, i sistemi non si limitano a scambiarsi richieste e risposte dirette, ma reagiscono a eventi generati in modo distribuito. Gli eventi possono essere propagati tramite message broker come Apache Kafka, RabbitMQ o AWS SNS/SQS, che facilitano la gestione della comunicazione tra componenti indipendenti.

Uno dei principali vantaggi di questa architettura è la sua scalabilità. Grazie alla gestione distribuita degli eventi, il sistema può adattarsi dinamicamente all'aumento del carico senza impattare sulle prestazioni. l'integrazione di nuovi servizi può avvenire in modo più flessibile, senza influenzare le componenti esistenti. La resilienza è anch'essa migliorata, ogni servizio opera in modo autonomo e non è vincolato a risposte sincrone da altri componenti.

D'altra parte per garantire la consistenza dei dati in un'architettura event-driven è più complesso rispetto ad altri modelli. La natura asincrona della comunicazione rende difficile tracciare il flusso esatto degli eventi, complicando il monitoraggio del sistema. Inoltre, l'introduzione di un message broker aggiunge un ulteriore livello di gestione, con possibili latenze nella propagazione degli eventi.

Questa architettura è ampiamente adottata in applicazioni IoT, sistemi di monitoraggio in tempo reale ed e-commerce, dove la rapidità di reazione agli eventi è un fattore critico per il successo del sistema.

Ogni modello architetturale presenta vantaggi e svantaggi che ne determinano l'applicabilità a seconda delle esigenze del sistema distribuito. Il modello client-server, pur essendo semplice da implementare, può soffrire di limitazioni in termini di scalabilità e tolleranza ai guasti. L'architettura peer-to-peer, invece, offre maggiore resilienza ma introduce

complessità nella gestione della sicurezza e della coerenza dei dati. I microservizi rappresentano una soluzione flessibile e scalabile, ma richiedono un'attenta orchestrazione per gestire la comunicazione tra componenti. Infine, l'architettura event-driven si distingue per la sua elevata scalabilità e reattività, pur presentando sfide legate alla consistenza e al monitoraggio degli eventi.

Scegliere l'architettura giusta dipende dal contesto in cui ci si trova e dagli obiettivi che vogliamo raggiungere. Spesso, si uniscono diversi modelli per sfruttare al meglio i loro vantaggi, creando così sistemi distribuiti sempre più resilienti ed efficienti.

Capitolo 5

Tecnologie per Sistemi Distribuiti

Negli ultimi anni, il mondo dell'informatica ha assistito a un cambiamento radicale nel modo in cui le applicazioni vengono sviluppate e gestite. Se un tempo tutto girava su un unico server fisico o, al massimo, su più macchine collegate tra loro, oggi i sistemi distribuiti sono la norma. Questo ha portato alla nascita di nuove tecnologie che rendono più semplice ed efficiente la gestione di applicazioni su larga scala.

In un sistema distribuito, diversi componenti dell'applicazione possono essere eseguiti su server diversi, potenzialmente sparsi in tutto il mondo. Questo approccio offre numerosi vantaggi, tra cui maggiore affidabilità, scalabilità e flessibilità, ma introduce anche nuove sfide: come possiamo garantire che tutto funzioni in modo coordinato? Come gestiamo la comunicazione tra i diversi componenti? Come evitiamo che un singolo errore blocchi l'intero sistema?

Per rispondere a queste domande, negli anni sono stati sviluppati strumenti potenti come Docker, Kubernetes, Apache Kafka e database distribuiti come Cassandra e MongoDB. In questo capitolo esploreremo nel dettaglio queste tecnologie, cercando di capire **come funzionano e perché sono diventate così importanti**.

5.1 Docker e la containerizzazione: perché è così rivoluzionario?

Docker[12] è un popolare software libero che utilizza la virtualizzazione a livello di sistema operativo per eseguire applicazioni in ambienti isolati chiamati container.

Immaginiamo di sviluppare un'applicazione che deve girare su diversi server. Ogni server potrebbe avere un sistema operativo leggermente diverso, librerie installate in versioni differenti e configurazioni uniche. Il rischio è che il software funzioni perfettamente su una macchina, ma dia problemi su un'altra quindi Docker risolve questo problema grazie

alla **containerizzazione**.

Un **container**[6] è un ambiente isolato in cui gira un'applicazione con tutte le sue dipendenze. È come se ogni applicazione avesse il proprio piccolo sistema operativo, indipendente dal resto. La cosa bella è che questi container sono **leggeri e veloci**, perché condividono il kernel del sistema operativo invece di creare un'intera macchina virtuale da zero. Docker può impacchettare un'applicazione e le sue dipendenze in un container virtuale che può essere eseguito su qualsiasi sistema Linux, Windows o macOS.

Docker può essere usato sia in ambienti on-premise che su cloud pubblici o privati, offrendo grande flessibilità nella gestione delle applicazioni.

Su Linux, Docker sfrutta i container nativi del sistema, utilizzando le funzionalità di isolamento del kernel per separare le applicazioni dal resto del sistema.

A differenza di una macchina virtuale, un container Docker non ha un proprio sistema operativo completo. Invece, condivide il kernel con il sistema host, sfruttando meccanismi di isolamento per gestire in modo sicuro CPU, memoria, rete e altre risorse. Questo permette di eseguire le applicazioni in ambienti isolati senza il peso di un'intera VM. Per farlo, Docker utilizza la libreria **libcontainer** (introdotta con la versione 0.9) o altri strumenti come **libvirt, LXC** o **systemd-nspawn**.

Grazie ai container, ogni applicazione può funzionare in uno spazio completamente separato, con il proprio file system, rete e identificativo. Anche se condividono lo stesso kernel, è possibile assegnare a ciascun container un limite di utilizzo per CPU, memoria e altre risorse, garantendo efficienza e stabilità.

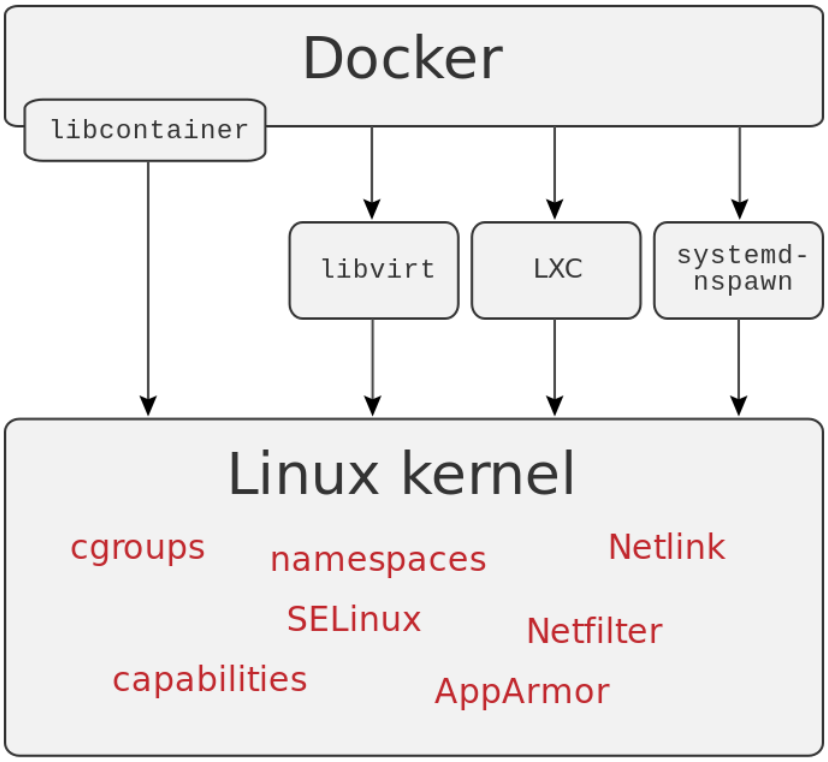

Figura 5.1: Docker utilizza diverse interfacce per accedere alle funzionalità di virtualizzazione del kernel Linux[6]

Docker porta alcuni vantaggi non indifferenti nel mondo dei sistemi distribuiti:

- **Portabilità totale**: se un'applicazione funziona su un container Docker, funzionerà ovunque, che sia il nostro PC, un server in azienda o un data center nel cloud.
- **Efficienza**: rispetto alle macchine virtuali, i container occupano meno risorse e si avviano in pochi millisecondi.
- **Scalabilità**: possiamo avviare più container in base alle necessità, aumentando o riducendo la capacità del sistema in tempo reale.

Docker ha cambiato completamente il modo in cui gli sviluppatori creano e distribuiscono software, ma quando il numero di container cresce, diventa difficile gestirli manualmente. Qui entra in gioco **Kubernetes**.

5.2 Kubernetes

Kubernetes[16], spesso abbreviato in K8s, è una piattaforma open-source che ci aiuta a orchestrare e gestire i container. Iniziato come progetto da Google, oggi è mantenuto dalla Cloud Native Computing Foundation. Kubernetes supporta diversi runtime per i container, tra cui Docker.

Se pensiamo a Docker come a una scatola che contiene un'applicazione pronta all'uso, Kubernetes è il sistema che ci permette di gestire centinaia o addirittura migliaia di queste scatole senza farci prendere dal panico.

Immaginiamo di avere un'app che deve servire milioni di utenti. Per farlo, dobbiamo eseguire numerosi container su diversi server, distribuendo il carico di lavoro in modo efficiente e facendo in modo che, se qualcosa va storto, il sistema possa auto-ripararsi. Kubernetes si occupa di tutto questo in modo automatico.

5.2.1 Architettura di Kubernetes

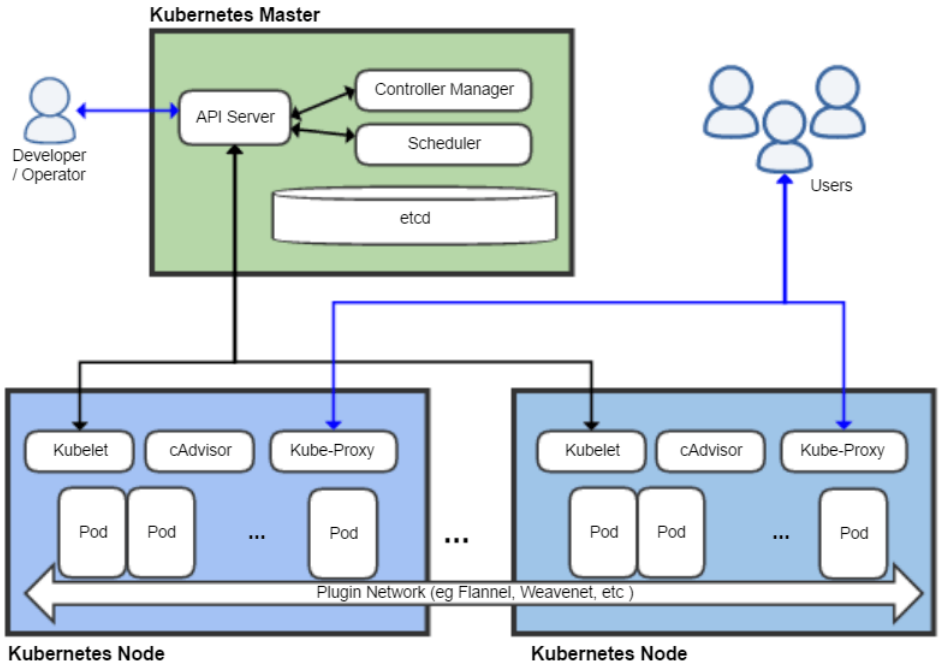

Figura 5.2: Architettura di un cluster Kubernetes

Un cluster Kubernetes[16] è composto da più nodi, suddivisi in **nodi master** (o control plane) e **nodi worker**. Ogni nodo può essere un server fisico o virtuale e tutti collaborano per eseguire applicazioni containerizzate.

- Il **Control Plane** gestisce l'intero cluster, monitorando lo stato del sistema e distribuendo i carichi di lavoro.
- Il **Data Plane**, invece, comprende i componenti che eseguono concretamente i container e gestiscono le risorse di calcolo.

L'intero sistema funziona basandosi sul **desired state**, ovvero lo stato desiderato dell'infrastruttura definito dagli utenti. Kubernetes si occupa di mantenere questo stato, intervenendo quando necessario per correggere eventuali anomalie.

Componenti principali

Control Plane Node (Master)

Il **master** è il cuore del cluster e si occupa esclusivamente della gestione e dell'orchestrazione, senza eseguire direttamente i container. Per garantire alta disponibilità, il master può essere replicato su più server.

- **kube-apiserver** → Espone le API di Kubernetes e funge da punto di comunicazione per gli amministratori e i nodi del cluster.
- **kube-controller-manager** → Controlla lo stato del cluster e interviene per mantenerlo allineato al desired state.
- **kube-scheduler** → Decide su quale nodo eseguire i nuovi carichi di lavoro in base alle risorse disponibili.
- **etcd** → Database distribuito che memorizza lo stato del cluster. Essendo un elemento critico, è spesso replicato per garantire affidabilità.

Worker Node

I **nodi worker** sono responsabili dell'esecuzione delle applicazioni containerizzate. Ogni nodo deve avere un **container runtime** (come Docker) per poter avviare i container.

- **kubelet** → Controlla l'esecuzione dei container su ogni nodo, mantenendo il sistema allineato al desired state.
- **kube-proxy** → Gestisce il traffico di rete tra i nodi e facilita la comunicazione tra i servizi.

- **Container runtime** → Componente che esegue i container, come Docker o containerd.

Risorse di Kubernetes

In Kubernetes, tutto viene gestito attraverso il concetto di **risorse**, che descrivono lo stato desiderato del sistema. Una volta definita una risorsa, Kubernetes si occupa di applicare le modifiche necessarie per raggiungere lo stato desiderato.

- **Pod** → È l'unità base di esecuzione in Kubernetes e può contenere uno o più container che condividono risorse di rete e storage. I pod possono essere scalati automaticamente in base al carico di lavoro.
- **Service** → Definisce come i pod vengono esposti alla rete, sia internamente al cluster che verso l'esterno. I pod vengono associati ai servizi tramite **labels**, etichette chiave-valore che permettono di organizzarli in modo dinamico.

Grazie al suo approccio dichiarativo e alla flessibilità offerta, Kubernetes è oggi lo standard per la gestione di infrastrutture cloud-native, rendendo più semplice il deployment e la scalabilità delle applicazioni.

5.3 Apache Kafka: il motore della comunicazione asincrona

Nei sistemi distribuiti, la gestione efficiente della comunicazione tra componenti è fondamentale per garantire affidabilità, scalabilità e prestazioni elevate. Apache Kafka[13, 15] è una piattaforma open-source progettata per gestire flussi di dati in tempo reale, fornendo un'infrastruttura di **messaggistica distribuita** ad alte prestazioni.

Kafka adotta un modello pub-sub (publish-subscribe), dove i dati vengono creati e utilizzati da entità chiamate producer e consumer. Questi dati vengono organizzati in topic, ognuno dei quali si divide in più partizioni. Questo sistema aiuta a distribuire il carico di lavoro su diversi nodi, migliorando così la velocità di elaborazione e l'affidabilità globale.

- **Producer**: i servizi che inviano dati ai topic di Kafka.
- **Broker**: i nodi del cluster Kafka che memorizzano e distribuiscono i dati.
- **Consumer**: i servizi che leggono i dati dai topic per elaborarli.

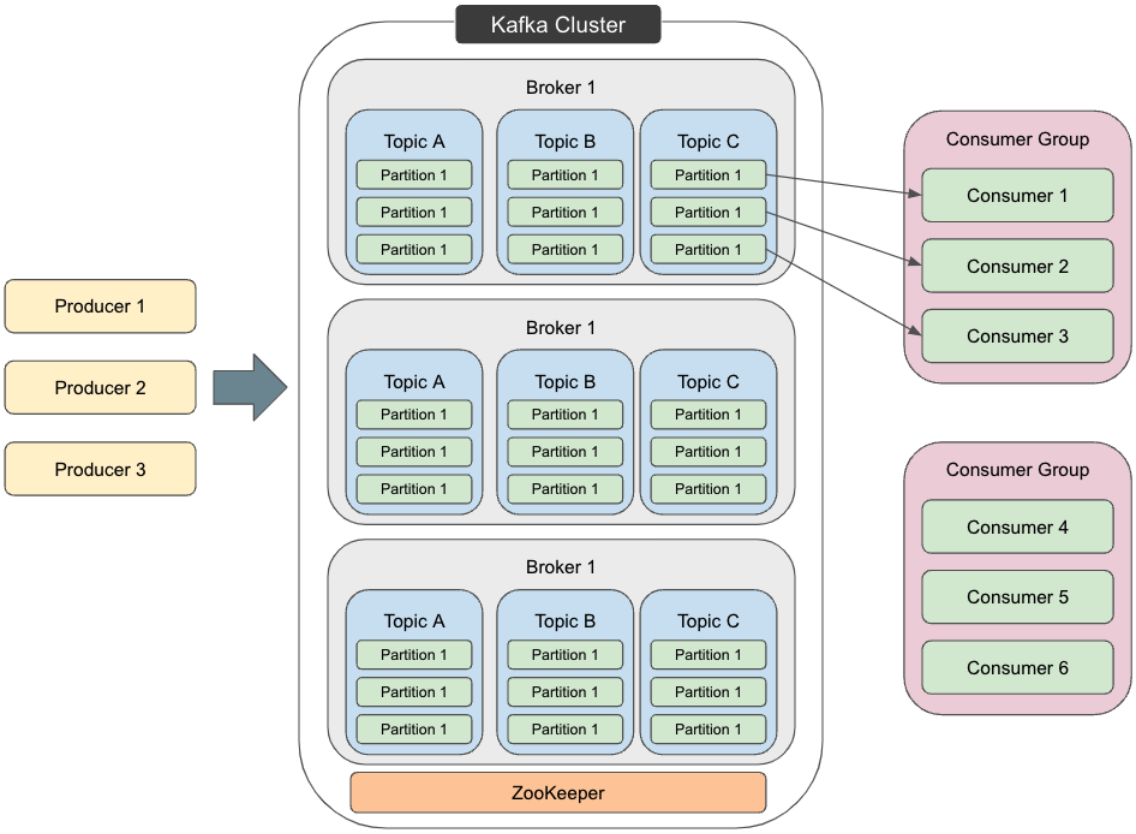

Figura 5.3: Architettura dei messaggi Kafka Apache [11]

A differenza dei sistemi di messaggistica tradizionali, Kafka non elimina i messaggi dopo la loro consegna, ma li **memorizza in modo persistente** per un periodo configurabile. Questa caratteristica consente ai consumer di leggere e rileggere i dati quando necessario, garantendo **tolleranza ai guasti** e affidabilità nelle applicazioni distribuite.

Uno dei principali vantaggi di Kafka nei sistemi distribuiti è la sua capacità di **disaccoppiare** i componenti di un'architettura. In una comunicazione tradizionale basata su richieste sincrone (ad esempio, HTTP REST), i servizi devono essere attivi e rispondere immediatamente alle richieste. Con Kafka i dati vengono prodotti e consumati in modo asincrono cosi da consentire ai servizi di operare in modo indipendente. Questo approccio migliora la **resilienza** del sistema, evitando colli di bottiglia e garantendo che i componenti possano scalare autonomamente.

Un altro aspetto distintivo è la sua scalabilità orizzontale: Kafka può essere eseguito su cluster di più nodi, con le partizioni distribuite tra i broker. Questo permette di gestire volumi di dati sempre crescenti semplicemente aggiungendo nuovi nodi al cluster, senza compromettere le prestazioni.

Kafka è ampiamente utilizzato in diversi scenari reali all'interno dei sistemi distribuiti, tra cui:

- **Microservizi**: facilita la comunicazione tra microservizi senza che questi debbano interagire direttamente tra loro, migliorando modularità e indipendenza.
- **Monitoraggio e Logging**: raccoglie e distribuisce log in tempo reale, consentendo l'analisi e il rilevamento di anomalie.
- **Internet of Things (IoT)**: gestisce flussi di dati provenienti da dispositivi connessi, elaborandoli in modo scalabile.
- **Analisi dei dati e Machine Learning**: supporta pipeline di dati per l'addestramento di modelli su dataset aggiornati in tempo reale.

Apache Kafka è un elemento fondamentale nei sistemi distribuiti moderni. Offre un'infrastruttura solida per gestire i flussi di dati in tempo reale. Con la sua architettura scalabile, la persistenza dei dati e la capacità di separare i vari componenti di un sistema, Kafka è diventato un vero punto di riferimento per l'orchestrazione di eventi e per la comunicazione tra i servizi distribuiti. Inoltre, la sua integrazione con tecnologie come Kubernetes e Apache Spark lo rende una soluzione molto potente per gestire dati su larga scala, sia in ambienti cloud che on-premise.

5.4 Database distribuiti: come gestire i dati in un mondo decentralizzato?

Quando parliamo di sistemi distribuiti, anche i database devono essere distribuiti. Non possiamo più affidarci a un singolo server, perché diventerebbe un collo di bottiglia e un punto di guasto critico.

Ecco alcune delle soluzioni più utilizzate:

- **Apache Cassandra**: ideale quando abbiamo bisogno di alta disponibilità e scalabilità. Funziona senza un server centrale, garantendo che i dati siano replicati su più nodi. È perfetto per applicazioni che devono resistere ai guasti hardware.
- **MongoDB**: utilizza un modello a documenti JSON, offrendo flessibilità nella gestione dei dati. È molto usato nelle applicazioni web moderne e nei big data.
- **Redis**: un database in-memory velocissimo, ideale per caching, sessioni utente e dati che devono essere accessibili quasi istantaneamente.

Grazie a questi database, le applicazioni distribuite possono **gestire milioni di utenti senza problemi di performance**.

Docker, Kubernetes, Kafka e i database distribuiti sono i pezzi fondamentali di un moderno sistema distribuito. Ognuno di questi strumenti risolve problemi specifici:

- **Docker** semplifica la creazione di applicazioni portabili.
- **Kubernetes** gestisce e coordina l'esecuzione dei container su larga scala.
- **Kafka** permette ai servizi di comunicare in modo asincrono e scalabile.
- **I database distribuiti** garantiscono archiviazione efficiente dei dati.

Queste tecnologie, lavorando insieme, rendono possibile la creazione di sistemi distribuiti affidabili, scalabili e performanti, in grado di gestire milioni di utenti e dati in tempo reale.

Capitolo 6

Comunicazione nei Sistemi Distribuiti

La comunicazione è un elemento fondamentale nei **sistemi distribuiti**, in quanto consente l'interazione tra nodi dislocati in diverse locazioni fisiche. I sistemi distribuiti sono caratterizzati dalla presenza di più entità autonome, che collaborano per raggiungere obiettivi comuni come detto in precedenza. Tali entità si scambiano informazioni tramite protocolli di comunicazione che garantiscono la consistenza dei dati, la resilienza alle problematiche di rete e la scalabilità necessaria per gestire grandi volumi di traffico.

Di seguito, esamineremo i principali protocolli e tecnologie che facilitano la comunicazione tra i componenti di un sistema distribuito.

6.1 Middleware

Il **middleware** è un livello software che si posiziona tra le applicazioni e la rete, facilitando la comunicazione tra componenti distribuiti. Il suo obiettivo principale è **nascondere i dettagli tecnici della comunicazione**, consentendo ai servizi di interagire senza doversi preoccupare di aspetti come il trasporto dei dati, la gestione degli errori o la sicurezza.

Senza middleware, ogni servizio dovrebbe implementare manualmente tutte queste funzionalità, aumentando la complessità del sistema. Con il middleware, invece, la comunicazione diventa più efficiente e standardizzata.

6.1.1 Tipologie di Middleware

Esistono diverse categorie di middleware, ognuna adatta a specifiche esigenze:

- **Middleware basato su RPC (Remote Procedure Call)**: Permette a un servizio di eseguire una funzione su un altro nodo come se fosse locale.

  - **Esempi**: gRPC, Apache Thrift, Java RMI.

- **Message-Oriented Middleware (MOM)**: Si basa su code di messaggi per permettere la comunicazione asincrona tra servizi, garantendo che i messaggi vengano consegnati anche se il destinatario non è immediatamente disponibile.
  - **Esempi**: RabbitMQ, Apache Kafka, ActiveMQ.

- **Middleware per architetture Event-Driven**: Gestisce la comunicazione basata su eventi, in cui i servizi reagiscono a determinati trigger senza connessioni dirette.
  - **Esempi**: Apache Kafka, AWS EventBridge.

- **Service Mesh**: Un middleware specifico per microservizi che gestisce il traffico tra i servizi, offrendo funzionalità avanzate come il load balancer, la sicurezza e il monitoraggio.
  - **Esempi**: Istio, Linkerd.

6.1.2 Funzionalità del Middleware

Il middleware è un elemento fondamentale nell'architettura dei sistemi distribuiti, non limitandosi semplicemente a gestire la comunicazione tra i vari componenti. Offre una serie di funzionalità avanzate che aiutano a superare le sfide di scalabilità, affidabilità e sicurezza, tutte caratteristiche indispensabili per il buon funzionamento di sistemi complessi e distribuiti su larga scala. Qui di seguito, vediamo alcune delle principali funzioni che il middleware può offrire per migliorare le performance di un sistema distribuito.

Gestione della comunicazione

Una delle funzioni più importanti del middleware è quella di rendere più facile la comunicazione tra i vari servizi e componenti di un sistema. I sistemi distribuiti sono formati da diversi nodi che possono usare tecnologie di comunicazione differenti. Tuttavia, il middleware fa in modo che questa complessità non venga vista dagli sviluppatori e dagli utenti finali. In parole semplici, offre un'interfaccia chiara e standardizzata che consente ai servizi di dialogare tra loro senza preoccuparsi dei dettagli implementativi dei protocolli di comunicazione, come TCP/IP, HTTP e altro ancora. Questo rende lo sviluppo di applicazioni più semplice e modulare, riducendo la dipendenza dalle specifiche tecniche della rete. In questo modo, la gestione delle connessioni diventa più facile e si migliora anche la portabilità e la manutenzione del sistema.

Tolleranza ai guasti

Uno degli aspetti fondamentali che distingue un sistema distribuito è la necessità di gestire correttamente i guasti. I sistemi distribuiti sono suscettibili a vari tipi di

guasti, che possono verificarsi a livello di rete, hardware o software. Il middleware, in questo contesto, gioca un ruolo determinante nel garantire la **continuità operativa** e nel migliorare la **resilienza** del sistema. Le funzionalità integrate, come il **retry automatico**, permettono di ripetere automaticamente le operazioni che non sono riuscite a causa di errori temporanei. Questo riduce al minimo l'interruzione dei servizi e migliora l'affidabilità complessiva. Il **failover** è un altro meccanismo critico, in quanto consente il passaggio automatico a un nodo di backup o a una risorsa alternativa quando un componente del sistema fallisce. Inoltre, il middleware offre strumenti avanzati di gestione degli errori che monitorano costantemente il sistema per rilevare anomalie e guasti, garantendo che il sistema possa adattarsi rapidamente e riprendersi senza impattare l'esperienza dell'utente finale.

Sicurezza

La sicurezza rappresenta una delle sfide più complesse nei sistemi distribuiti, dove i dati attraversano diversi nodi, talvolta in ambienti poco sicuri. Il middleware offre diversi strumenti e protocolli per tutelare la comunicazione e i dati sensibili. La crittografia dei dati è uno strumento fondamentale che assicura che le informazioni condivise tra i servizi siano protette da accessi non autorizzati. Può essere applicata sia ai dati in transito che a quelli archiviati, garantendo così la riservatezza delle informazioni delicate. Inoltre, il middleware gestisce anche l'autenticazione e l'autorizzazione, assicurando che solo gli utenti e i servizi legittimi possano accedere a risorse e informazioni protette. In pratica, l'autenticazione consente di verificare l'identità di un'entità (come un utente o un servizio) tramite metodi come password, token o certificati, mentre l'autorizzazione determina quali privilegi e risorse può utilizzare quella specifica entità. Implementare questi meccanismi riduce notevolmente il rischio di attacchi esterni, come quelli man-in-the-middle o accessi non autorizzati.

Monitoraggio e logging

Un'altra funzionalità chiave del middleware è il monitoraggio continuo del sistema e la registrazione delle attività attraverso il **logging**. Nei sistemi distribuiti, la tracciabilità è fondamentale per identificare e risolvere problemi, ottimizzare le performance e mantenere una panoramica complessiva sul funzionamento dell'intera architettura. Il middleware raccoglie informazioni dettagliate riguardo a tutte le comunicazioni tra i vari servizi, registrando ogni operazione effettuata, errori verificatisi e risposte ricevute. Questi log possono essere utilizzati per il **debugging**, ossia la diagnostica e la risoluzione di eventuali problematiche che emergono nel sistema, consentendo agli sviluppatori di capire la causa principale dei malfunzionamenti. Inoltre, i dati di log sono cruciali per l'**ottimizzazione** del sistema, in quanto permettono di identificare colli di bottiglia o operazioni inefficienti che potrebbero essere migliorate. I log possono anche servire per analizzare l'andamento delle performance nel tempo e per effettuare operazioni di **monitoraggio proattivo**, dove l'analisi dei dati raccolti consente di intervenire prima

che un problema critico si verifichi, riducendo al minimo i tempi di inattività del sistema.

Scalabilità

Infine, il middleware contribuisce alla **scalabilità** del sistema distribuito, ossia alla capacità del sistema di gestire un aumento del carico di lavoro o dell'utenza senza compromettere le performance. Il middleware permette di gestire la distribuzione dei carichi di lavoro su più server, bilanciando dinamicamente le richieste tra i vari nodi per ottimizzare l'uso delle risorse e garantire che il sistema possa crescere in modo fluido senza dover rivedere completamente l'architettura. In presenza di un alto volume di richieste, il middleware può integrare strategie come il **load balancing**, che distribuisce equamente il traffico tra diversi server, e la **scalabilità orizzontale**, che aggiunge più nodi al sistema senza interrompere il servizio.

6.2 Protocolli REST e gRPC

La comunicazione tra i componenti di un sistema distribuito può avvenire utilizzando **protocolli di rete** che definiscono le modalità con cui i dati vengono scambiati tra client e server. Tra i protocolli più diffusi per questo tipo di comunicazione ci sono **REST** e **gRPC**.

- **REST (Representational State Transfer)**[8] è un'architettura basata su HTTP che consente la gestione delle risorse tramite operazioni standard, come GET, POST, PUT, DELETE, e PATCH. Le risorse sono rappresentate come URL, e le operazioni su queste risorse sono realizzate tramite l'invio di richieste HTTP. Il principale vantaggio di REST è la sua semplicità e la sua adesione al principio **stateless**, per cui ogni richiesta è indipendente dalle precedenti e deve contenere tutte le informazioni necessarie per essere elaborata. La statelessness semplifica la gestione delle sessioni e la scalabilità delle applicazioni, ma limita le performance in scenari complessi, dove la connessione e la comunicazione tra client e server necessitano di un mantenimento di stato o di una connessione persistente.

Inoltre, REST è ampiamente supportato in molte piattaforme, rendendolo una scelta comune per l'implementazione di **API web**. Tuttavia, REST può essere meno efficiente rispetto ad altre soluzioni, come gRPC, soprattutto in ambienti ad alta latenza e con frequenti scambi di messaggi tra client e server.

- **gRPC (gRPC Remote Procedure Call)**[21], sviluppato da Google, è un framework di comunicazione basato su **HTTP/2** e utilizza il formato di serializzazione **Protocol Buffers** (protobuf). gRPC permette di invocare procedure remote tra client e server come se fossero chiamate locali, ma sfruttando la rete. Una delle principali caratteristiche che differenzia gRPC da REST è che esso supporta connessioni

persistenti e bidirezionali, grazie all'uso di **streaming**. Questo approccio riduce significativamente il sovraccarico nelle comunicazioni a bassa latenza, permettendo una comunicazione più fluida e rapida tra i vari componenti del sistema.

Inoltre, gRPC offre supporto nativo per la **compressione dei dati**, la **gestione della sicurezza** tramite SSL/TLS e l'autenticazione tramite certificati, rendendolo ideale per l'utilizzo in sistemi ad alta performance, come i **microservizi**, dove è richiesta una bassa latenza, un'alta efficienza e la gestione di grandi volumi di traffico.

6.3 WebSockets e Comunicazione in Tempo Reale

Un approccio fondamentale per la comunicazione in tempo reale nei sistemi distribuiti è l'utilizzo dei WebSockets. A differenza dei tradizionali protocolli HTTP, che funzionano su un modello di richieste e risposte indipendenti, i WebSockets stabiliscono una connessione bidirezionale e persistente tra client e server. Questo significa che la connessione rimane aperta, permettendo a entrambe le parti di scambiarsi messaggi in tempo reale, senza bisogno di aprire una nuova connessione per ogni messaggio.

Usare WebSockets è particolarmente utile per applicazioni che richiedono basse latenze e aggiornamenti costanti, come le app di chat, i giochi online e i servizi di monitoraggio in tempo reale. Grazie alla sua connessione persistente, riduce il sovraccarico legato all'apertura e chiusura delle connessioni HTTP, migliorando le prestazioni in scenari con alto traffico.

Inoltre, i WebSockets si integrano bene con il protocollo HTTP/2, ulteriormente ottimizzando le performance rispetto alla versione precedente. La loro capacità di mantenere una connessione aperta e di supportare la comunicazione bidirezionale li rende uno degli strumenti più efficaci per la comunicazione in tempo reale in un sistema distribuito.

La comunicazione nei sistemi distribuiti è vitale per assicurare che le applicazioni funzionino correttamente. Qui, il middleware gioca un ruolo chiave nell'astrarre la complessità della comunicazione, mentre protocolli come REST e gRPC, i WebSockets e message brokers come Kafka e RabbitMQ offrono strumenti specifici per varie situazioni.

Capitolo 7

Best Practice nello Sviluppo di Sistemi Distribuiti

La progettazione di un sistema distribuito robusto[14] richiede un approccio sistematico che consideri le sfide intrinseche legate alla distribuzione dei dati e delle operazioni tra diversi nodi. Per garantire che il sistema sia scalabile, efficiente, sicuro e resiliente, è necessario adottare delle best practice durante tutte le fasi di sviluppo. In questa sezione, esploreremo alcune delle principali best practice per la progettazione e la gestione di sistemi distribuiti.

7.1 Gestione della Consistenza e della Disponibilità (CAP Theorem

Una delle sfide principali nello sviluppo di un sistema distribuito è la gestione della **consistenza**, **disponibilità** e **tolleranza alla partizione** (CAP). Il **Teorema CAP**[4], formulato da Eric Brewer, afferma che in un sistema distribuito, è possibile garantire solo due delle seguenti proprietà simultaneamente[5]:

- **Consistenza (C):** Ogni lettura del sistema restituisce il valore più recente scritto, assicurando che tutti i nodi abbiano la stessa visione dei dati.
- **Disponibilità (A):** Ogni richiesta (sia di lettura che di scrittura) riceve una risposta, anche se alcuni nodi non sono disponibili.
- **Tolleranza alla partizione (P):** Il sistema continua a funzionare correttamente anche se ci sono delle partizioni di rete che impediscono la comunicazione tra alcuni nodi.

Adottare una strategia che soddisfi i compromessi imposti dal teorema CAP è essenziale nella progettazione di un sistema distribuito. Ad esempio:

- **CP (Consistenza e Tolleranza alla Partizione)**: Alcuni database distribuiti, come **HBase**, adottano questo approccio, dando priorità alla consistenza dei dati anche a costo di una minore disponibilità in caso di fallimento di una partizione.
- **AP (Disponibilità e Tolleranza alla Partizione)**: **Cassandra** è un esempio di sistema che preferisce la disponibilità e la tolleranza alla partizione, rinunciando parzialmente alla consistenza immediata per garantire che il sistema resti operativo.
- **CA (Consistenza e Disponibilità)**: Questo scenario è raro nei sistemi distribuiti reali, poiché la tolleranza alla partizione è generalmente indispensabile.

È fondamentale scegliere l'approccio che meglio si adatta ai requisiti del sistema, in particolare per quanto riguarda il bilanciamento tra **coerenza e latenza**.

7.2 Resilienza e Tolleranza ai Guasti

La **resilienza** è un principio cardine nello sviluppo di sistemi distribuiti. Poiché i nodi di un sistema distribuito sono vulnerabili a guasti (sia hardware che software), il sistema deve essere progettato per tollerare tali guasti senza compromettere l'affidabilità complessiva.

Le principali strategie per migliorare la resilienza includono:

- **Replicazione dei dati:** La replica dei dati tra nodi riduce il rischio di perdita di dati in caso di guasti. La replica può avvenire in modo sincrono o asincrono, a seconda delle esigenze di consistenza del sistema.
- **Failover automatico:** Implementare meccanismi di failover per consentire che un nodo guasto venga sostituito automaticamente da un nodo di backup.
- **Circuit Breakers:** Un pattern di progettazione utilizzato per evitare che il sistema tenti continuamente operazioni su un servizio che è fallito, evitando così l'accumulo di errori e il degrado delle prestazioni.
- **Partizionamento e Sharding:** Utilizzare il partizionamento dei dati per distribuire il carico tra più nodi. Questo aiuta a mantenere l'operatività del sistema anche se alcune partizioni diventano non disponibili.

La **tolleranza ai guasti** deve essere testata attraverso simulazioni di guasti nei nodi per garantire che il sistema possa continuare a operare in modo affidabile anche sotto stress.

7.3 Sicurezza nei Sistemi Distribuiti

La **sicurezza** è una preoccupazione primaria nella progettazione di sistemi distribuiti, poiché i dati e le risorse sono spesso distribuiti su più nodi e reti, aumentando la superficie di attacco. Alcune best practice per garantire la sicurezza includono[10]:

- **Crittografia:** Utilizzare crittografia sia per i dati a riposo che per i dati in transito per proteggere la privacy e l'integrità dei dati. La crittografia SSL/TLS deve essere implementata nelle comunicazioni tra i nodi.
- **Autenticazione e autorizzazione:** Implementare meccanismi di autenticazione robusti per garantire che solo gli utenti e i servizi autorizzati possano accedere ai dati. Le soluzioni di autenticazione centralizzata, come **OAuth** o **JWT (JSON Web Tokens)**, possono semplificare la gestione delle credenziali.
- **Auditing e logging:** Registrare ogni accesso e modifica ai dati per creare un audit trail. Questo è cruciale per rilevare attività sospette e per garantire la trasparenza nelle operazioni del sistema.

Poiché i sistemi distribuiti sono spesso esposti a minacce provenienti da diverse reti, è importante proteggere i nodi con tecniche di **firewalling** e **segregazione delle reti**.

7.4 Monitoraggio e Logging

Il **monitoraggio** e il **logging** sono essenziali per garantire che i sistemi distribuiti possano essere controllati in tempo reale e che eventuali problemi possano essere rilevati tempestivamente. Le best practice includono:

- **Metriche di sistema:** Monitorare il **CPU**, la **memoria**, il **traffico di rete** e la **latency** dei nodi, così da poter rilevare colli di bottiglia o malfunzionamenti.
- **Logs centralizzati:** Utilizzare strumenti come **ELK Stack (Elasticsearch, Logstash, Kibana)** o **Prometheus** per centralizzare i log e le metriche da più nodi. Ciò consente di visualizzare i dati in tempo reale e di effettuare analisi più approfondite.
- **Alerting:** Configurare alert per avvisare gli amministratori di sistema su eventi critici, come guasti del nodo, malfunzionamenti delle applicazioni o violazioni di sicurezza.

Il monitoraggio proattivo è cruciale per prevenire i guasti e ottimizzare le prestazioni del sistema.

7.5 Testing di Sistemi Distribuiti

Il **testing** di un sistema distribuito è particolarmente complesso, poiché è necessario simulare scenari che coinvolgono molteplici nodi, comunicazioni di rete e guasti. Alcune best practice includono:

- **Test di integrazione:** Verificare che i componenti del sistema interagiscano correttamente in scenari distribuiti. Strumenti come **Docker Compose** o **Kubernetes** possono essere utilizzati per simulare ambienti di test reali.
- **Test di carico e performance:** Simulare carichi di traffico per verificare la scalabilità del sistema e il suo comportamento sotto stress. Utilizzare strumenti come **JMeter** o **Gatling** per eseguire test di performance.
- **Test di resilienza:** Simulare guasti nei nodi per verificare la resilienza e la tolleranza ai guasti del sistema, utilizzando strumenti come **Chaos Monkey**, che interrompe casualmente le risorse per testare la capacità del sistema di mantenere il funzionamento.

Capitolo 8

Conclusioni e Sviluppi Futuri

8.1 Sintesi dei Risultati Ottenuti

Nel corso di questa ricerca, sono stati analizzati i **fondamenti dei sistemi distribuiti**, inclusi i modelli architetturali, le tecnologie emergenti, e le sfide comuni nell'ambito della **comunicazione**, **coerenza**, **disponibilità** e **tolleranza ai guasti**. È emerso chiaramente che un sistema distribuito efficiente richiede un bilanciamento tra diverse proprietà che, spesso, devono essere adattate alle specifiche esigenze del caso d'uso.

I principali punti emersi includono:

- **Modelli Architetturali:** Ogni modello (Client-Server, Peer-to-Peer, Microservizi, Event-Driven) ha vantaggi specifici in base ai requisiti di scalabilità, flessibilità e gestione della comunicazione. La scelta dell'architettura dipende strettamente dal contesto applicativo e dalle risorse disponibili.

- **Tecnologie per la Containerizzazione e Orchestrazione:** L'adozione di tecnologie come **Docker**, **Kubernetes**, e **Apache Kafka** ha reso più facili la gestione e la scalabilità dei sistemi distribuiti, permettendo di affrontare le sfide della gestione di grandi volumi di dati e della comunicazione tra microservizi.

- **Sincronizzazione e Consistenza dei Dati:** L'integrazione di modelli di consistenza forti o deboli (ad esempio, **Eventual Consistency** vs **Strong Consistency**) è stata discussa, evidenziando come il contesto delle applicazioni (es. sistemi di pagamento vs. social media) influisca sulla scelta tra coerenza e disponibilità.

- **Gestione dei Guasti:** Le tecniche di resilienza, come la replicazione dei dati, il failover automatico e l'uso di circuit breakers, sono risultate cruciali per garantire che i sistemi distribuiti siano affidabili anche in presenza di guasti.

Le tecnologie moderne hanno migliorato la capacità di progettare sistemi distribuiti più robusti, ma restano ancora sfide complesse, specialmente nei casi in cui la scalabilità e l'affidabilità devono essere garantite a livello globale.

8.2 Direzioni Future per la Ricerca

Nonostante i notevoli progressi nel campo dei sistemi distribuiti, ci sono ancora molte aree che meritano di essere esplorate e sviluppate. Le direzioni future per la ricerca potrebbero includere:

1. **Miglioramento della Gestione della Consistenza:** Ci sono già vari modelli di consistenza come la 'Eventual Consistency' e la 'Causal Consistency', ma una delle sfide che ci aspetta è quella di creare modelli che trovino un equilibrio migliore tra le esigenze di coerenza e quelle di disponibilità, soprattutto in situazioni molto dinamiche e con carichi di lavoro variabili. Potremmo lavorare su nuovi algoritmi in grado di garantire una consistenza ottimizzata in tempo reale, migliorando così l'efficienza delle risorse.

2. **Tecnologie Avanzate di Orchestrazione:** Anche se Kubernetes è attualmente il leader nel settore dell'orchestrazione dei container, ci sono molte opportunità di ricerca su come automatizzare l'orchestrazione in ambienti complessi, come quelli multi-cloud o ibridi. Gli sforzi di ricerca potrebbero concentrarsi su soluzioni più intelligenti in grado di adattarsi dinamicamente alle fluttuazioni del carico, ottimizzando l'allocazione delle risorse in base a condizioni in tempo reale.

3. **Sistemi Autonomi e Self-Healing:** I sistemi autonomi, in grado di rilevare automaticamente errori, riprendersi dai guasti e ottimizzare le performance senza l'intervento umano, rappresentano un settore in rapida evoluzione. L'integrazione di tecniche di intelligenza artificiale e machine learning per migliorare la capacità di ripristino automatico e l'ottimizzazione continua delle operazioni è un'area di ricerca molto promettente

4. **Sistemi Distribuiti a Basso Consumo Energetico:** Con la crescita dei sistemi distribuiti, aumenta anche la necessità di renderli più sostenibili. I futuri sforzi di ricerca potrebbero focalizzarsi sull'ottimizzazione energetica dei data center e sull'uso di tecnologie ecocompatibili, come l'edge computing, per distribuire i carichi di lavoro in modo più efficiente e ridurre l'impronta di carbonio.

5. **Sicurezza Avanzata per Sistemi Distribuiti:** I sistemi distribuiti sono spesso soggetti a minacce provenienti da più fronti. Oltre alla protezione contro gli attacchi DDoS o le vulnerabilità nei protocolli di comunicazione, il miglioramento delle tecniche di **sicurezza post-quantistica** e la gestione delle **identità distribuite**

saranno cruciali per la protezione dei dati sensibili nelle nuove architetture, in un futuro dove la crittografia potrebbe dover fronteggiare sfide di sicurezza derivanti dai progressi nel calcolo quantistico.

6. **Edge Computing e Sistemi Distribuiti Decentralizzati:** Con la crescente importanza dell'**Internet of Things (IoT)**, l'edge computing diventa un argomento caldo. I sistemi distribuiti potrebbero evolversi verso una **decentralizzazione** ancora maggiore, dove i nodi di calcolo si trovano più vicino agli utenti finali (dispositivi edge), riducendo latenza e migliorando l'efficienza delle operazioni.

In conclusione, la ricerca sui sistemi distribuiti continua a evolversi e offre molteplici ambiti di studio per i futuri sviluppi. Anche se i progressi sono stati tanti, è chiaro che emergono continuamente nuove sfide tecnologiche, organizzative e di sicurezza, aprendo la strada a ricerche innovative e soluzioni all'avanguardia. Questo studio ha creato una base solida per comprendere le tecnologie e metodologie attuali, ma il futuro dei sistemi distribuiti si costruirà su questi concetti, mirando a una maggiore automazione, resilienza e sostenibilità.

Bibliografia

[1] Luciano De Falco Alfano. *Paxos*. URL: https://luciano.defalcoalfano.it/ds-appunti/14_paxos.html.

[2] Luciano De Falco Alfano. *Sistemi Distribuiti*. URL: https://luciano.defalcoalfano.it/ds-appunti/01_definizione.html#.

[3] Luciano De Falco Alfano. *Sistemi Distribuiti - Categorie dei sistemi distribuiti*. URL: https://luciano.defalcoalfano.it/ds-appunti/02_categorie.html.

[4] Eric Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[5] Eric Brewer. " CAP twelve years later: How the "rules" have changed ". In: *Computer* 45.02 (feb. 2012), pp. 23–29. ISSN: 1558-0814. DOI: 10.1109/MC.2012.37. URL: https://doi.ieeecomputersociety.org/10.1109/MC.2012.37.

[6] Wikipedia contributors. *Docker*. 2019. URL: https://it.wikipedia.org/wiki/Docker#.

[7] Wikipedia contributors. *NTP*. 2024. URL: https://it.wikipedia.org/wiki/Network_Time_Protocol.

[8] Bruno Costa et al. "Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture?" In: *2014 IEEE/IFIP Conference on Software Architecture*. 2014, pp. 105–114. DOI: 10.1109/WICSA.2014.29.

[9] G. Coulouris et al. *Distributed Systems: Concepts and Design*. Pearson Education, 2011. ISBN: 9780133001372. URL: https://books.google.it/books?id=3ZouAAAAQBAJ.

[10] George Coulouris et al. *Distributed Systems*. Pearson Deutschland, 2011, p. 1064. ISBN: 9780273760597. URL: https://elibrary.pearson.de/book/99.150005/9781447930174.

[11] DevKuma. *Apache Kafka - Architettura*. 2024. URL: https://www.devkuma.com/docs/apache-kafka/intro/.

[12] Docker. *Documentazione ufficiale*. URL: https://docker.com.

[13] Red Hat. *What is Apache Kafka?* 2024. URL: https://www.redhat.com/it/topics/integration/what-is-apache-kafka.

[14] "IEEE Recommended Practice for Fault Diagnosis and Protection in Smart Distribution System". In: *IEEE Std 2748-2023* (2024), pp. 1–76. DOI: 10.1109/IEEESTD.2024.10542673.

[15] Apache Kafka. *Documentazione ufficiale*. URL: https://kafka.apache.org.

[16] Kubernetes. *Documentazione ufficiale*. URL: https://kubernetes.io.

[17] Vito Lavecchia. *Differenza tra sistemi centralizzati e sistemi distribuiti*. URL: https://vitolavecchia.altervista.org/differenza-tra-sistemi-centralizzati-e-sistemi-distribuiti/.

[18] mundobytes. *NTP*. 2024. URL: https://mundobytes.com/it/protocollo-ntp/.

[19] *Network Time Protocol (NTP)*. RFC 958. Set. 1985. DOI: 10.17487/RFC0958. URL: https://www.rfc-editor.org/info/rfc958.

[20] Leader Phabrix. *Understanding the Fundamentals of PTP and SMPTE ST 2110*. URL: https://leaderphabrix.com/it/understanding-the-fundamentals-of-ptp-and-smpte-st-2110/.

[21] Pradeep K. Sinha. "Remote Procedure Calls". In: *Distributed Operating Systems: Concepts and Design*. 1997, pp. 167–230. DOI: 10.1109/9780470544419.ch4.

[22] Sincron Sistemi. *PTP e NTP*. 2024. URL: https://www.sincron-sistemi.it/news/precision-time-protocol-ptp.

[23] Sincron Sistemi. *Sincronizzazione di rete*. 2025. URL: https://www.sincron-sistemi.it/news/sincronizzazione-di-rete.

[24] Maarten van Steen e Andrew S. Tanenbaum. "A brief introduction to distributed systems". In: *Computing* 98.10 (2016), pp. 967–1009. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7. URL: https://doi.org/10.1007/s00607-016-0508-7.

[25] Robbert Van Renesse e Deniz Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (feb. 2015). ISSN: 0360-0300. DOI: 10.1145/2673577. URL: https://doi.org/10.1145/2673577.

[26] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (2009). Available at https://cacm.acm.org/practice/eventually-consistent/, pp. 40–44. DOI: 10.1145/1435417.1435432.

Ringraziamenti

Per primo, vorrei ringraziare il Dott. Davide Rossi per la sua disponibilità e per avermi indirizzato durante lo sviluppo della tesi.

Ringrazio i miei amici e la mia famiglia per avermi sostenuto lungo questo percorso dandomi la forza di perseguire i miei sogni.

Un ringraziamento ancora più profondo lo dedico alla mia ragazza, Alice. Sei stata il mio porto sicuro nei momenti più difficili, quando la stanchezza e il dubbio prendevano il sopravvento. Grazie di cuore per esserci sempre stata. Ti amo.

Infine, voglio ringraziare mia madre e mio padre. Ogni giorno mi impegno per rendervi fieri, consapevole di tutti i sacrifici che avete affrontato per permettermi di studiare all'università. Vi ringrazio immensamente.

A tutti voi, grazie. Senza il vostro sostegno, questo traguardo non sarebbe stato possibile.