DOCUMENTO DI PROGETTO: LS-TRIS

SCOPO E SINTESI

Questo documento descrive le scelte progettuali e implementative per un sistema Client-Server containerizzato pensato per essere riproducibile, semplice da distribuire e scalabile.

L'architettura è composta da tre macro-componenti, rendendola di fatti un'architettura a tre livelli:

- Un backend in C che espone servizi su socket TCP (Lato server);
- Un bridge in Express che traduce tra WebSocket e socket TCP (Lato server);
- Un frontend web in Angular 20 (Lato client).

L'intero stack è orchestrato con Docker e Docker Compose. Inoltre, ci si è avvalsi di NGINX per servire il frontend tramite container. Sono stati predisposti Bash Scripts e CMD Scripts per avviare e fermare rapidamente i container.

Lo sviluppo è stato effettuato in un ambiente DevContainer integrato con Visual Studio Code. Inoltre, per permettere un miglior sviluppo in team, ci siamo avvalsi dello strumento di versioning git. L'intero progetto è in una repository remota hostata da GitHub.

OBIETTIVI DI PROGETTO

L'obiettivo primario è costruire un software robusto e scalabile che:

- Permetta una comunicazione end-to-end tra browser e backend C;
- Sia semplice da avviare e sviluppare in team;
- Metta in evidenza buone pratiche di programmazione e gestione della concorrenza.

Queste direttive hanno guidato ogni scelta tecnologica e architetturale.

ARCHITETTURA AD ALTO LIVELLO

Il sistema è suddiviso in tre componenti principali che vivono su due Docker Compose networks differenti:

FRONTEND: Applicazione Angular 20 scritta in TypeScript; comunica con messaggi JSON via WebSocket con il bridge; styling realizzato con Sass; build servita da NGINX in produzione. È nella docker network di "frontend".

BRIDGE: Applicazione Express scritta in JavaScript; espone un server WebSocket verso il browser e mantiene connessioni TCP tramite socket verso il backend C; si occupa di traduzione dei messaggi e gestione della sessione. È nella docker network di "backend".

BACKEND: Server in C, multithreaded, che accetta connessioni socket TCP, elabora messaggi JSON usando json-c e persiste dati in SQLite. È nella docker network di "backend".

Questa separazione consente di isolare responsabilità, migliorare la manutenibilità e permettere un'esperienza utente soddisfacente, difatti permettendogli di accedere all'applicativo da qualsiasi dispositivo abbia una connessione a internet.

La scelta di inserire bridge e backend sulla stessa docker network deriva dal fatto che in questo modo è possibile sfruttare il Docker DNS per riferirsi all'altro container nella rete, evitando così di dover impostare manualmente gli indirizzi IP per metterli in comunicazione.

FLUSSO DI DATI

Il flusso è intenzionalmente lineare e basato su JSON:

- L'utente si interfaccia al frontend Angular tramite browser,
- Angular comunica con il Bridge via WebSocket inviando messaggi JSON,
- Il Bridge intercetta questi messaggi dalla WebSocket e li inoltra al backend C su socket TCP, sempre in formato JSON,
- Il backend elabora la richiesta, interroga il database SQLite e risponde con JSON che risale il percorso inverso. Mantenere JSON coerente su tutta la catena permette logging omogeneo e facilità di debug.

SCELTE TECNOLOGICHE

Backend

Linguaggio C

Il linguaggio C è un linguaggio di programmazione a uso generale di natura imperativa e procedurale. È un linguaggio di programmazione di alto livello, potente e veloce, ideale per sistemi che richiedono controllo diretto su memoria e risorse.

Un controllo esplicito delle risorse e delle performance ha permesso di ottenere un basso overhead, un'esecuzione efficiente e maggiore controllo sulla gestione della concorrenza e dell'I/O di rete.

Threading con pthreads

Gestire client multipli tramite thread permette al backend di servire più connessioni simultanee con bassa latenza, dato che istanziare un nuovo thread è molto più veloce rispetto al creare un sottoprocesso.

In questo modo si ha un modello semplice da comprendere e ottimizzare, con la possibilità di evolvere in un thread pool per carichi maggiori.

JSON con json-c

JSON è un formato leggero e leggibile per rappresentare dati strutturati, ampiamente usato per lo scambio di informazioni tra client e server. La scelta di usare la libreria json-c è motivata dal fatto che è una libreria matura, leggera, con un'interfaccia intuitiva e ben documentata.

Grazie a una libreria già ampiamente utilizzata e testata, è stato possibile serializzare e deserializzare JSON nel contesto C in modo semplice ed efficiente.

Database embedded SQLite

SQLite è un sistema di gestione dati integrato direttamente nell'applicazione, che non richiede un server esterno e salva i dati in un file locale.

La scelta di usare un database embedded, come SQLite, è motivata dal fatto che in questo modo è stato possibile integrare un database potente ed efficiente per applicazioni embed e locali, usufruendo di un'estrema semplicità di configurazione. Inoltre, SQLite rispetta le proprietà ACID (atomicità, consistenza, isolamento, persistenza), per cui non causa problematiche di race conditions.

Bridge

Express

Node è una piattaforma JavaScript lato server, tramite la quale viene utilizzato Express, un framework minimalista per costruire API e gestire richieste HTTP/WebSocket in modo rapido ed efficiente.

Poiché i browser non supportano socket TCP raw, si è reso necessario un adattatore WebSocket a socket TCP e viceversa. Node è ideale per I/O asincrono e WebSocket, permettendo uno sviluppo rapido e con ampio ecosistema per utilità (logger, healthcheck, autenticazione).

Frontend

Angular

Angular è un framework frontend opinionated, basato su TypeScript e pensato per creare applicazioni web scalabili e modulari con una solida architettura.

Scegliere Angular ha permesso di costruire un frontend strutturato e manutenibile, che grazie a TypeScript (un superset di JavaScript con tipizzazione) migliora la qualità del codice e facilita la gestione di UI complesse. Inoltre, un framework opinionated mette a disposizione avanzati tooling a supporto.

Sass

Sass è un'estensione del CSS che introduce funzionalità avanzate come variabili, mixin e nesting.

Sass è la scelta ideale per semplificare la scrittura e la manutenzione degli stili su progetti medio-grandi.

NGINX

NGINX è un server web ad alte prestazioni usato per servire contenuti statici, fare da reverse proxy e bilanciare il carico tra più servizi.

NGINX si è rivelato il giusto equilibrio in quanto a efficienza nel servire asset statici e semplicità di configurazione in container. È uno strumento ampiamente usato in produzione, favorendo la stabilità del software e con la possibilità di integrare funzionalità avanzate come un sistema di caching.

Docker e Docker Compose

Docker e Docker Compose sono strumenti per creare ambienti isolati e riproducibili tramite container. Compose permette di definire e orchestrare più servizi tramite un unico file.

In questo modo, è possibile riprodurre lo stesso ambiente in modo isolato e semplice. Qualsiasi membro del team può avviare lo stack con pochi comandi.

DevContainer

DevContainer è un ambiente di sviluppo preconfigurato e containerizzato, integrato con Visual Studio Code, che garantisce coerenza tra i membri del team e semplifica la configurazione iniziale del progetto in modalità sviluppo.

Con DevContainer, qualsiasi membro può lavorare al progetto con gli stessi strumenti e usufruire delle stesse toolchain.

Git

Git è un sistema di versionamento distribuito che consente di tracciare ogni modifica al codice sorgente, facilitando il lavoro in team e il recupero di versioni precedenti.

Integrare git nel progetto permette di gestire rami di sviluppo paralleli e mantenere uno storico completo e affidabile delle modifiche.

GitHub

GitHub è una piattaforma online che ospita repository git, aggiungendo funzionalità collaborative come pull request, issue tracking e integrazione continua.

In questo modo, viene semplificata la collaborazione tra sviluppatori, consente revisione del codice, automatizza test e deployment e offre visibilità pubblica o privata al progetto.

DOCKERIZZAZIONE E FLUSSI DI BUILD

Per i Dockerfile di backend e frontend è stato adottato un approccio multi-stage: la compilazione viene effettuata tramite container di build temporanei, dopodiché gli artefatti prodotti vengono copiati in container di runtime più snelli e minimali. Questo rende le immagini e i containers di runtime estremamente leggeri e riduce la superficie d'attacco.

In Docker Compose sono stati definiti volumi per la persistenza del file SQLite, che favorisce hot-reload in ambiente di sviluppo. Inoltre, sono state definite reti interne per una migliore gestione e isolamento dei containers. I containers che condividono la stessa rete sono richiamabili tramite Docker DNS.

STRUMENTI DI SVILUPPO E OPERATIVI

È stato adottato DevContainer per usufruire di un ambiente di sviluppo comune a tutti i membri del team: il progetto viene aperto con le estensioni di Visual Studio Code, i linter e le toolchain già installate. Inoltre, definire scripts di utilità per avviare/fermare rapidamente lo stack permette un onboarding immediato, sia in sviluppo che in produzione. Sono anche stati predisposti README in Markdown per istruzioni utente e comandi utili per i diversi progetti.

Queste scelte rendono l'esperienza di sviluppo uniforme e replicabile su macchine diverse.

BENEFICI ARCHITETTURALI

Usufruire di un bridge tra frontend e backend permette anche ad applicazioni web based di interagire con il backend in C, che funge da unica fonte di verità. In questo modo, è possibile espandere il progetto con altre tipologie di client (app mobile, app desktop, ecc...) pur mantenendo la stessa logica di backend.

L'uso di Docker Compose velocizza lo sviluppo: costruendo immagini docker con tutto l'ambiente e le dipendenze preconfigurate, si evita di avere a che fare con la gestione di versioni preinstallate delle dipendenze necessarie e si minimizza il tempo dedicato all'allineamento degli ambienti di sviluppo tra i diversi membri del team.

LIMITAZIONI E RISCHI

SQLite è ottimo per demo e carichi leggeri, ma per un uso a scala elevata è consigliabile migrare a un DBMS dedicato, come Postgres.

La gestione manuale dei thread e della sincronizzazione in C richiede attenzione ai dettagli per evitare memory leak o race condition. È, inoltre, consigliabile adottare un tipo di autenticazione più affidabile così da evitare attacchi informatici volti a rubare l'identità del giocatore.