

Dok: A Distributed, Real-Time Collaborative Document Editor

Sistemi Distribuiti (A.A. 2025/2026)

Alessandro Stefani

Lorenzo Tosi

Visione del Sistema & Trasparenza

Perché la Distribuzione?

Concorrenza di Scala: Superamento dei limiti prestazionali e dei colli di bottiglia di CPU/IO del singolo processo Node.js.

Tolleranza ai Guasti: Resilienza automatica a crash dei singoli container backend senza interruzioni di servizio.

Stato Volatile & Persistente: Separazione netta tra lo stato condiviso in RAM e lo stable storage su MongoDB.

Obiettivi di Trasparenza

Accesso: Interfacce uniformi (HTTP REST e Socket.IO) indipendenti dall'architettura.

Localizzazione e Replicazione: Client connessi a un unico Virtual IP; backend multipli invisibili al browser.

Guasti: Proxy che maschera i crash e gestisce la riconnessione automatica con riallineamento dello stato.

Coerenza e Amalgamazione: Nodi autonomi che cooperano per presentarsi come un sistema unitario.

Architettura Software

Layered & Event-Based

Struttura & Topologia dei Nodi

1. Presentation Tier (Vue 3 SPA)

Interfaccia utente reattiva, connessa via REST e canali Socket.IO.

2. Application Tier (Node.js)

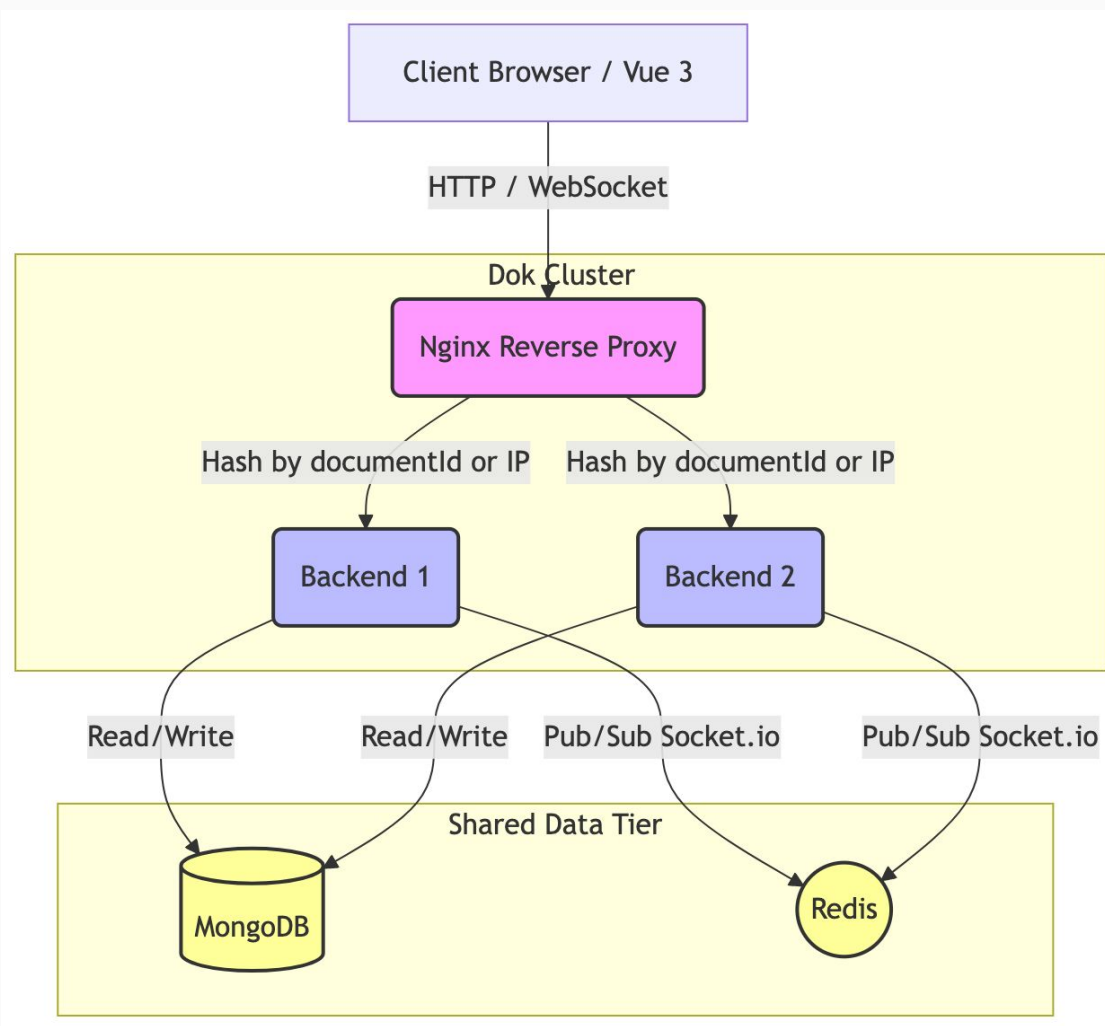
Due repliche (backend1, backend2) orizzontalmente scalate.

3. Shared State Tier (MongoDB + Redis)

Persistenza su MongoDB; Event Bus & coordinamento su Redis.

Proxy Layer

Cluster Nginx Active-Passive con Virtual IP gestito da Keepalived.



Architettura Software

Layered & Event-Based

Entità	Implementazione	Ruolo
Event Bus	Container Redis (<code>redis:alpine</code>) via protocollo RESP	Connettore di disaccoppiamento spaziale/referenziale
Publisher	<code>pubClient</code> (<code>RedisClientType</code>) sulle repliche di backend	Emittitore di eventi di dominio
Subscriber	<code>subClient</code> (<code>RedisClientType</code>) sulle repliche di backend	Ricevitore asincrono e dispatcher verso i socket locali
Event Stream	Canali Redis Pub/Sub e WebSocket su Socket.IO	Canale di propagazione dati ed eventi a bassa latenza

Eliminazione spof Nginx

Problema: In una classica architettura con load balancer, il singolo reverse proxy Nginx costituisce un single point of failure.

Soluzione: Cluster active-passive con due nodi (nginx-master e nginx-backup) orchestrati con Keepalived.

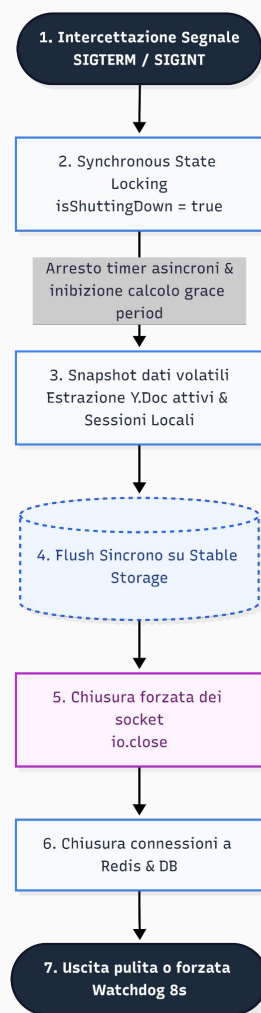
Protocollo VRRP in Unicast: Utilizzo del *Virtual Router Redundancy Protocol* configurato esplicitamente in unicast per aggirare i limiti di routing dei pacchetti multicast nelle reti containerizzate di Docker.

Virtual IP: Gestione fluttuante dell'indirizzo 172.20.0.100. Al fallimento dei controlli di health-check sul master, il backup acquisisce il VIP.

Locale vs Produzione: In locale viene impiegato un semplice container HAProxy come entripoint; in produzione i record DNS punterebbero direttamente al VIP fluttuante.

Graceful Shutdown

1. Intercettazione dei segnali del sistema operativo (SIGTERM o SIGINT) e blocco sincrono dello stato tramite flag `isShuttingDown`.
2. Flush sincrono su MongoDB dello stato volatile del backend (documenti Yjs e sessioni).
3. Chiusura forzata dei socket (`io.close()`) per innescare immediatamente la riconnessione dei client verso il nodo superstite.
4. Watchdog timer di sicurezza (8 secondi) per garantire l'uscita del processo.



Consistent Routing

- Connessione dedicata: Client apre Socket.IO con query param **?documentId=<id>**.
- Nginx Consistent Hashing: Direttiva hash \$route_key consistent.
- Vantaggi Architettureali:
 - Session Stickiness: Tutti i client di uno stesso documento instradati al medesimo backend.
 - Località del dato: Una singola istanza RAM di Y.Doc attiva per documento.
 - Zero Lock Distribuiti: Nessun overhead di sincronizzazione ad ogni singola digitazione.
 - Stabilità su scale-up/down: Solo una minima parte delle chiavi viene rimappata se un nodo varia.

Collaborazione Real-Time: Yjs, CRDT e Modello di Consistenza

- Join & Sync Iniziale:
 - Evento join-document -> join della room Socket.IO.
 - Se assente in RAM: caricamento snapshot da MongoDB e creazione Y.Doc.
 - Invio sync-document: il client integra le operazioni mancanti (unione CRDT)
- 2. Modifica del Testo (CRDT):
 - Edit Tiptap → generazione update binario Yjs → emissione crdt-update.
 - Server applica su Y.Doc locale e fa broadcast nella room.
- 3. Cursori & Awareness (Effimero):
 - Coordinate cursore, selezione, colore utente.
 - Broadcast immediato, zero persistenza su DB (best-effort, recovery immediato).

Perché Yjs e i CRDT

- **Strong Eventual Consistency (SEC):**
 - Se tutti i nodi ricevono lo stesso set di messaggi, convergono deterministicamente allo stesso identico stato senza bisogno di un ordinatore centrale.
- **Confronto Architetture:**
 - OT (Operational Transformation): Richiede un server centrale ordinatore (sequencer) e log complessi di trasformazione.
 - CRDT (Yjs): Calcolo peer-to-peer / distribuito; fusione matematica deterministica.

Consenso

- Dok non implementa un algoritmo di consenso
 - non richiede di eleggere un leader applicativo per approvare o ordinare le modifiche al testo.
- Perché il testo non richiede consenso:
 - Yjs evita il problema rendendo gli update fondibili. Non è necessario che i nodi concordino prima un ordine globale.

Ciclo di vita dei documenti

- **Dual-Tier State:**
 - In-Memory (RAM): Y.Doc attivo è la single source of truth real-time.
 - Cold Storage (MongoDB): Checkpoint binario (yjsState) + visualizzazione (tiptapJson).
- **Checkpointing Dinamico:**
 - Timer a 3 secondi dalla prima modifica (batching di modifiche concorrenti).
- **Gestione dei Fallimenti:**
 - Graceful Shutdown: Flush su DB di tutti i Y.Doc, chiusura socket e connessioni, watchdog a 8s.
 - Crash Improvviso / Kill: Rischio perdita delle modifiche successive all'ultimo checkpoint; recupero delegato alla replica locale del client alla riconnessione.

Fine

Alessandro Stefani

Lorenzo Tosi