Valutazione sperimentale di Redis come soluzione di data store per un contesto aziendale

Lorenzo Dantonio
Lorenzo Raia

Sommario

Il presente lavoro valuta Redis come possibile soluzione di data store per un'azienda che intende adottarlo in produzione. Dopo una rassegna teorica delle caratteristiche architetturali di Redis, si è condotta una sperimentazione empirica volta a verificare se il comportamento osservato è coerente con quanto previsto dalla teoria, con particolare attenzione alla scalabilità orizzontale e alla tolleranza ai guasti.

Indice

1 Introduzione

Si assume il ruolo di un team tecnico incaricato di valutare l'adozione di Redis in un'azienda. L'obiettivo è quello di rispondere alla seguente domanda: *la teoria studiata regge nella pratica?* A tal fine il lavoro è organizzato in quattro fasi:

1. Verifica delle proprietà teoriche su un nodo singolo
2. Misurazione di una baseline di prestazioni
3. Sperimentazione della scalabilità su due assi indipendenti numero di nodi a parità di dati, e volume dati a parità di nodi sia in lettura che in scrittura
4. Verifica del comportamento in presenza di guasti

2 Studio teorico

2.1 Modello dati e single-threaded event loop

Redis si distingue come un data store in-memory avanzato, spesso definito un vero e proprio "data structures server". Oltre al classico e semplice modello chiave-valore, supporta nativamente strutture dati complesse come stringhe, hash, liste, set, sorted set, bitmap, HyperLogLog e stream. Dal punto di vista architetturale, il motore di elaborazione dei comandi è intrinsecamente single-threaded e si basa su un event loop ottimizzato che sfrutta l'I/O multiplexing. Questa scelta progettuale fondamentale garantisce che l'esecuzione di ogni singolo comando sia strettamente atomica: le operazioni sui dati avvengono in sequenza, eliminando alla radice le race condition e rimuovendo del tutto la necessità di implementare complessi meccanismi di lock applicativi lato client. Sebbene Redis abbia introdotto il multi-threading per delegare la gestione dell'I/O di rete e alleggerire il carico sui socket, l'esecuzione effettiva e la mutazione del dataset rimangono confinate a un singolo thread dedicato, preservando così la sua bassa latenza senza compromettere la coerenza dei dati.

2.2 Persistenza: RDB e AOF

Per ovviare alla volatilità intrinseca della memoria RAM, Redis offre due meccanismi di persistenza su disco complementari e configurabili. Il primo è l'RDB (Redis Database), che genera snapshot point-in-time dell'intero dataset a intervalli regolari. Essendo file binari estremamente compatti, i dump RDB sono ideali per i backup storici e per un ripristino veloce in caso di disastro, sebbene comportino il rischio di perdere i dati scritti nell'intervallo successivo all'ultimo snapshot. Il secondo meccanismo è l'AOF (Append-Only File), che opera come un transaction log registrando ogni singola operazione di scrittura ricevuta dal server. La durabilità dell'AOF è modulabile tramite la direttiva `appendfsync`, che permette di forzare la scrittura su disco per ogni comando (`always`), una volta al secondo (`everysec`, il compromesso più diffuso), oppure di delegarla al sistema operativo (`no`). Negli ambienti di produzione, la best practice prevede l'abilitazione congiunta di entrambi i meccanismi, bilanciando la garanzia di non perdere dati (tramite AOF) con un riavvio rapido ed efficiente del motore (tramite RDB).

2.3 Architettura a cluster e hash slot

Redis Cluster rappresenta la soluzione nativa per la scalabilità orizzontale e il partizionamento dei dati (sharding). A differenza dei sistemi basati su hashing consistente tradizionale, l'architettura di Redis divide logicamente lo spazio delle chiavi in esattamente 16384 hash slot

distribuiti tra i vari nodi master del cluster. L'algoritmo di routing determina l'allocazione di una determinata chiave calcolandone l'impronta tramite l'algoritmo CRC16 e applicando un'operazione di modulo (`CRC16(key) mod 16384`). Gli sviluppatori mantengono il controllo sul routing forzando l'assegnazione di chiavi correlate allo stesso slot tramite gli "hash tag", requisito indispensabile per eseguire transazioni multi-chiave valide nel cluster. In questa topologia distribuita, ogni nodo master è tipicamente affiancato da una o più repliche (slave), che non partecipano attivamente allo sharding primario ma agiscono da copie di sicurezza pronte a intervenire per garantire l'alta affidabilità.

2.4 Replica e failover

La replica in Redis è progettata per essere asincrona e non bloccante: i nodi master trasmettono in streaming un flusso continuo di comandi alle rispettive repliche. Questo approccio massimizza le prestazioni, sebbene introduca un fisiologico, seppur minimo, ritardo di allineamento. La resilienza del cluster è gestita da un sistema di monitoraggio peer-to-peer basato su protocollo gossip, in cui i nodi comunicano ininterrottamente su un bus dedicato. Se la maggioranza dei nodi master non riesce a raggiungere un nodo per un tempo superiore alla soglia definita nel parametro `cluster-node-timeout`, il nodo irraggiungibile viene dichiarato in stato di "FAIL". A questo punto si innesca automaticamente una procedura di elezione distribuita basata sul consenso: la replica con i dati più aggiornati viene promossa al ruolo di master, ereditando la gestione degli hash slot del nodo guasto. Durante questa transizione, che dura tipicamente pochi secondi, le richieste destinate agli slot in riassegnazione restituiscono errori (`CLUSTERDOWN`). Di conseguenza, i client devono implementare logiche resilienti di retry e backoff esponenziale per aggiornare la propria mappa della topologia del cluster e ripetere le operazioni una volta conclusa l'elezione.

3 Metodologia sperimentale

3.1 Ambiente di test e limite metodologico

Tutti i test sono stati eseguiti in locale tramite container Docker orchestrati con Docker Compose. Questo è un limite metodologico importante perchè la scalabilità orizzontale in un contesto aziendale reale distribuisce i nodi su macchine fisiche separate, ciascuna con risorse (CPU, rete, disco) dedicate; qui, invece, tutti i nodi condividono le stesse risorse fisiche. Ci si aspetta quindi che, oltre un certo numero di nodi, l'aumento di contesa superi il beneficio teorico dello sharding un effetto che, come si vedrà, è stato osservato (Sezione 6.1).

3.2 Strumenti

- **redis-py**: usato in Fase 1 per la verifica funzionale delle proprietà teoriche (TTL, atomicità, persistenza) e in Fase 4 per generare carico continuo durante il test di guasto.
- **memtier_benchmark**: usato in Fase 2 e Fase 3 per i benchmark di throughput e latenza, eseguito come container Docker separato collegato alla stessa rete dei nodi Redis.

3.3 Parametri comuni ai benchmark

Salvo dove diversamente indicato, ogni benchmark usa 10 client $\times$ 4 thread (40 connessioni parallele), payload di 100 byte o 1 KB, e riporta percentili di latenza p50/p95/p99.

3.4 Nota sulle due metodologie di budget delle richieste

Asse 1 e Asse 2 (Sezione 6) usano due convenzioni diverse per il numero di richieste, entrambe corrette ma *non direttamente confrontabili tra loro* sul singolo punto in comune:

- **Asse 1** (nodi variabili): richieste fisse a 100.000 per client, identico per ogni punto della serie stessa convenzione usata nella baseline di Fase 2.
- **Asse 2** (volume variabile): budget di scrittura fisso, dimensionato sul caso più grande (10M chiavi), cosicché ogni punto della serie (1M/5M/10M) abbia la stessa durata; solo il volume di dati risultante varia. Questo isola correttamente la variabile in esame (il volume), a scapito della confrontabilità diretta con l'Asse 1.

4 Fase 1 — Verifica delle proprietà teoriche su nodo singolo

Su un'istanza Redis singola sono stati eseguiti test funzionali mirati a verificare empiricamente le proprietà discusse nello studio teorico:

- **Scadenza (TTL)**: una chiave con `EXPIRE` a 3 secondi risulta correttamente assente dopo 4 secondi.
- **Atomicità**: 10 thread concorrenti eseguono ciascuno 1000 `INCR` sulla stessa chiave; il valore finale osservato è esattamente 10.000, confermando che il modello single-threaded rende l'operazione atomica senza lock applicativi.
- **Strutture dati**: hash, liste, insiemi e sorted set si comportano secondo le semantiche attese.
- **Persistenza**: un valore scritto e sottoposto a `BGSAVE` sopravvive al riavvio del container.

Tutti i test sono stati superati: la teoria regge nella pratica per queste proprietà di base.

5 Fase 2 — Baseline su nodo singolo

La Tabella 1 riporta i risultati del benchmark su nodo singolo (100.000 richieste/client, 40 connessioni).

Tabella 1: Baseline nodo singolo

Scenario	Ops/sec	Latenza media (ms)	p95 (ms)	p99 (ms)
Write 100B	269.550	0,148	0,191	0,231
Read 100B	304.680	0,130	0,167	0,191
Write 1KB	216.615	0,184	0,247	0,311
Read 1KB	288.708	0,138	0,175	0,215

La lettura è sistematicamente più veloce della scrittura, coerentemente con la teoria: la scrittura comporta anche l'append sincrono al log AOF, assente nel percorso di lettura. L'aumento del payload da 100B a 1KB riduce il throughput e aumenta la latenza in entrambe le direzioni, ma l'effetto è più marcato in scrittura, dove il costo di persistenza scala con la dimensione del dato scritto.

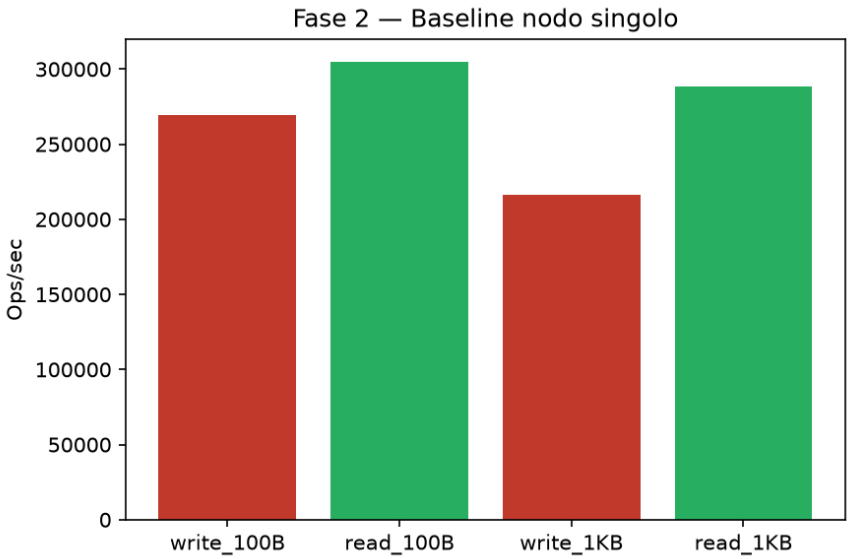

Figura 1: Throughput baseline nodo singolo per scenario e payload.

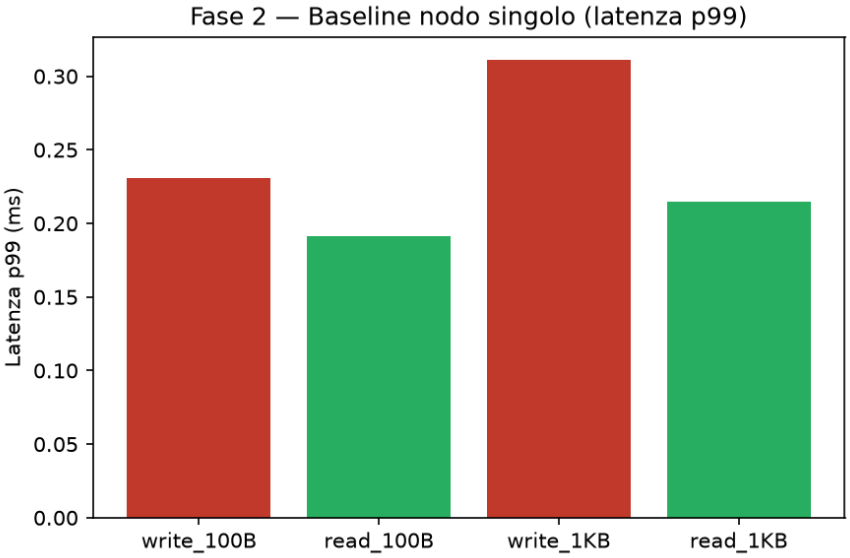

Figura 2: Latenza (p99) baseline nodo singolo per scenario e payload.

6 Fase 3 — Sperimentazione della scalabilità

6.1 Asse 1 — Numero di nodi variabile, dati fissi (1M chiavi)

Tabella 2: Asse 1: scalabilità al variare del numero di master (1M chiavi)

Master	Write 100B (ops/sec)	Read 100B (ops/sec)	Write 1KB (ops/sec)	Read 1KB (ops/sec)
3	385.378	493.210	207.133	493.248
5	373.286	565.088	205.249	505.176
7	318.373	375.891	151.906	312.850

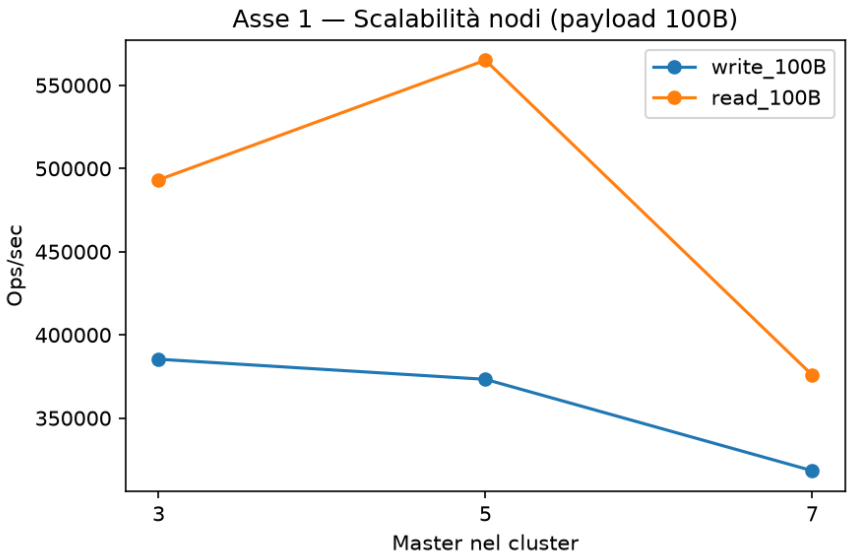

Figura 3: Asse 1: throughput (ops/sec) al variare del numero di master, payload 100B.

Osservazioni. La teoria prevede che l'aggiunta di nodi aumenti la capacità aggregata del cluster (più shard in parallelo). I dati mostrano l'opposto per la scrittura, che cala monotonicamente all'aumentare dei master (−17% tra 3 e 7 master sul payload 100B, −27% su 1KB); la lettura mostra invece una curva a campana, con un picco a 5 master seguito da un calo marcato a 7. In entrambi i casi, il calo più netto si osserva passando da 5 a 7 master (14 container totali contando le repliche). L'interpretazione più plausibile, coerente con il limite dichiarato in Sezione 3.1, è che su un'unica macchina fisica l'aggiunta di nodi introduca solo contesa aggiuntiva senza un corrispondente aumento di risorse di calcolo reali: oltre un certo numero di nodi, il collo di bottiglia è la macchina host, non l'architettura di Redis.

6.2 Asse 2 — Volume dati variabile, nodi fissi (3 master)

Osservazioni. La teoria prevede che l'accesso a una struttura hash sia $O(1)$ rispetto al numero di elementi memorizzati, indipendentemente dal volume totale del dataset. I dati di lettura confermano questa previsione: il throughput resta sostanzialmente costante al crescere del volume (397K → 382K → 395K ops/sec), con oscillazioni compatibili con il rumore di misura piuttosto che con un trend reale. La scrittura mostra invece un calo graduale ma

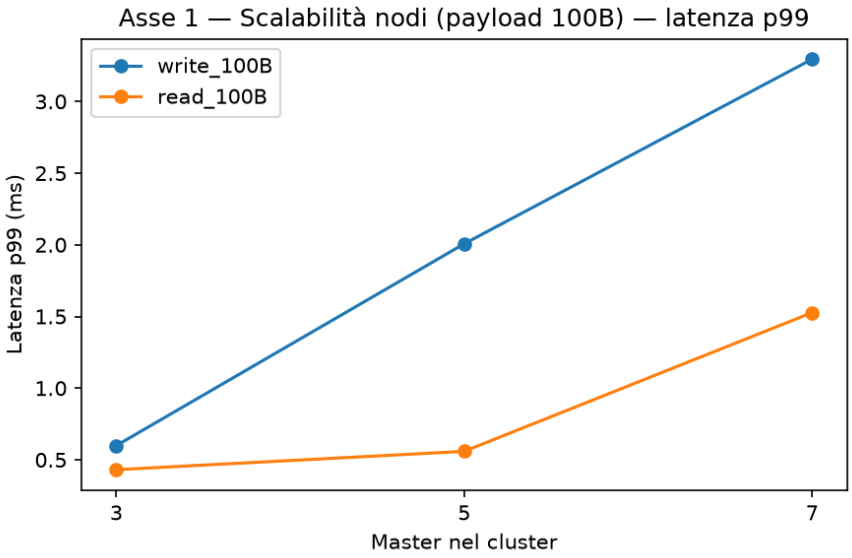

Figura 4: Asse 1: latenza (p99) al variare del numero di master, payload 100B.

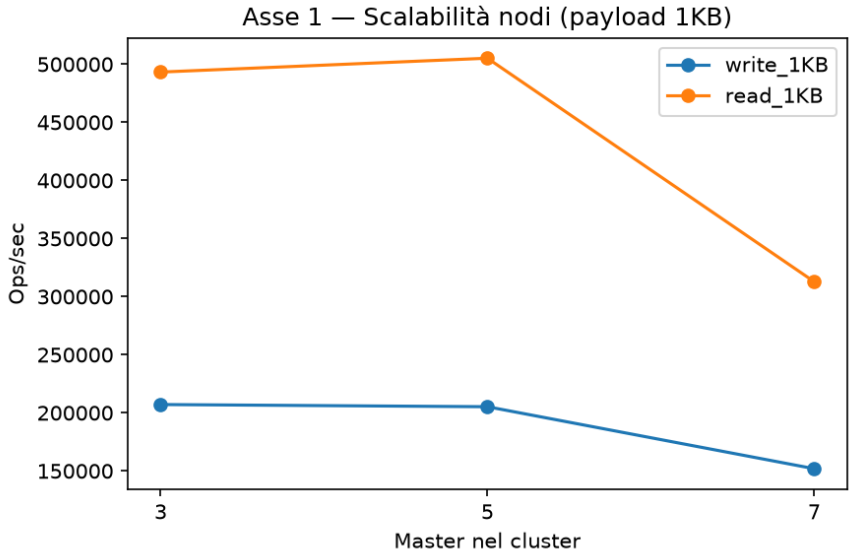

Figura 5: Asse 1: throughput (ops/sec) al variare del numero di master, payload 1KB.

Tabella 3: Asse 2: scalabilità al variare del volume dati (3 master fissi)

Chiavi	Write 100B (ops/sec)	Read 100B (ops/sec)
1M	267.220	397.052
5M	244.495	381.581
10M	231.522	394.745

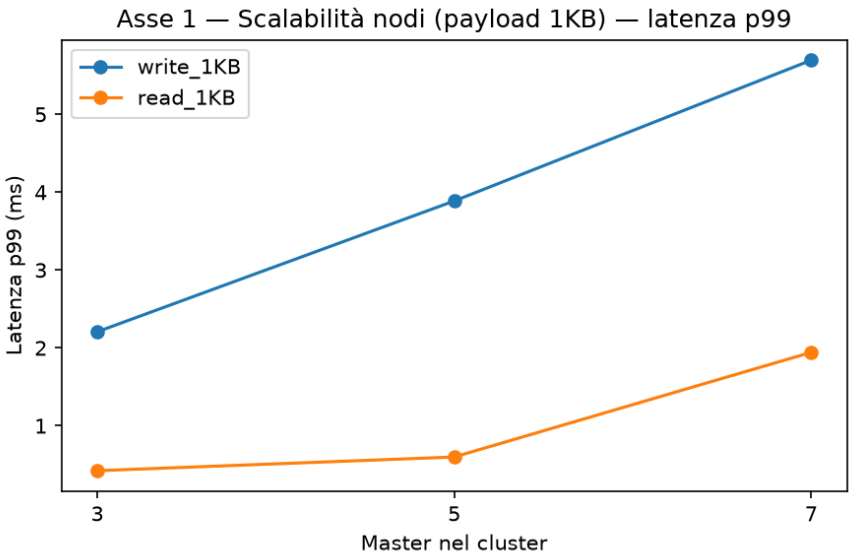

Figura 6: Asse 1: latenza (p99) al variare del numero di master, payload 1KB.

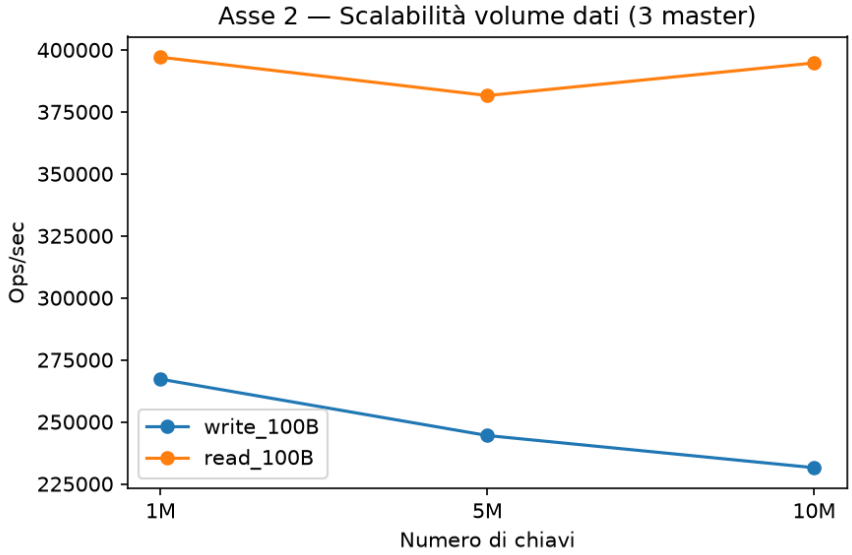

Figura 7: Asse 2: throughput (ops/sec) al variare del volume dati, per scrittura e lettura (payload 100B).

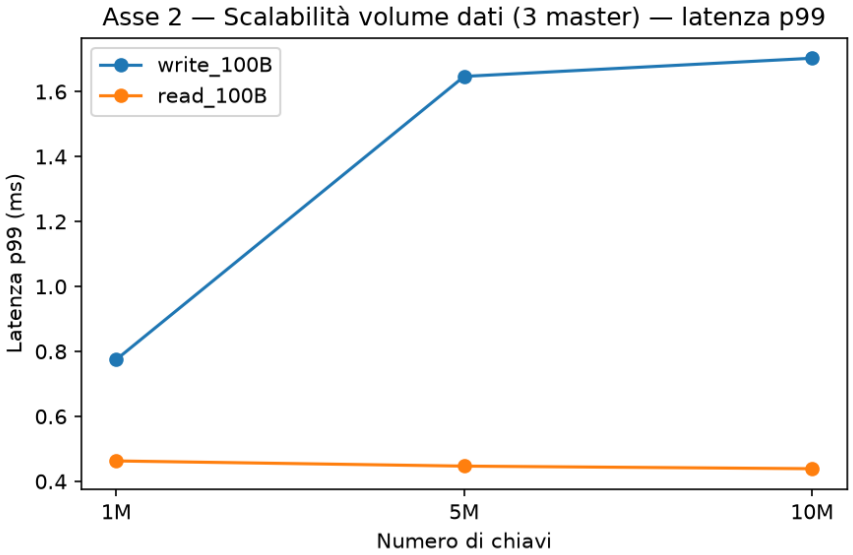

Figura 8: Asse 2: latenza (p99) al variare del volume dati, per scrittura e lettura (payload 100B).

consistente (−13% tra 1M e 10M chiavi), verosimilmente legato alla crescita del file AOF e all'aumento del working set in memoria, entrambi fattori non catturati dalla sola complessità algoritmica $O(1)$ dell'operazione di set.

6.3 Confronto tra i due assi

I due assi rispondono a domande diverse e, come discusso in Sezione 3.4, non sono confrontabili punto per punto sullo scenario condiviso: l'Asse 1 isola l'effetto del numero di nodi, l'Asse 2 l'effetto del volume dati. Nel complesso, però, emerge un messaggio coerente: la scalabilità verticale del dato è pressoché gratuita in lettura; la scalabilità orizzontale richiede infrastruttura fisica realmente distribuita per essere vantaggiosa.

7 Fase 4 — Tolleranza ai guasti

È stato generato un carico continuo (circa 20 richieste/secondo) verso il cluster, interrotto a metà test da un `docker kill` del container master corrente, seguito dal riavvio del nodo e dal suo re-inserimento nel cluster come replica.

Tabella 4: Fase 4: risultati del test di guasto (due esecuzioni)

Esecuzione	Richieste totali	Fallite	Stallo massimo osservato
Run precedente	733	1 (0,1%)	∼6 s
Run corrente	1696	0 (0,0%)	∼2,3 s

Osservazioni. In entrambe le esecuzioni il cluster ha eletto un nuovo master e ripristinato il servizio in pochi secondi, senza intervento manuale. Il client `redis-py` esegue retry interni verso il nuovo master, il che spiega perché la stragrande maggioranza delle richieste durante la finestra di failover non risulti fallita agli occhi dell'applicazione, a fronte comunque di un picco di latenza ben visibile. La differenza tra le due esecuzioni (una richiesta fallita e stallo più

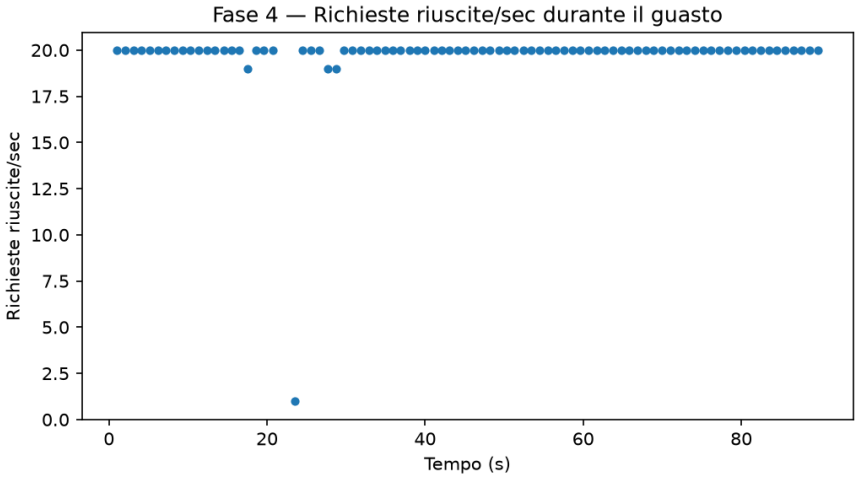

Figura 9: Fase 4: richieste riuscite al secondo durante il test di guasto. Il calo visibile intorno a metà test corrisponde alla finestra di failover del master ucciso.

lungo nella prima, zero fallimenti e stallo più breve nella seconda) è coerente con la variabilità ambientale già osservata negli Assi 1 e 2: su una macchina fisica condivisa, la durata esatta dell'elezione del nuovo master non è perfettamente deterministica.

8 Discussione: dove la teoria regge, dove no

- **Regge pienamente**: modello dati e atomicità (Fase 1); asimmetria lettura/scrittura dovuta alla persistenza (Fase 2); indipendenza del throughput di lettura dal volume dati, $O(1)$ (Asse 2); resilienza applicativa al guasto di un master grazie a replica e retry client-side (Fase 4).
- **Regge parzialmente**: il degrado di throughput in scrittura al crescere del volume (Asse 2) non è previsto dalla sola complessità algoritmica, ma è spiegabile con fattori di sistema (AOF, memoria).
- **Non regge, ma per un motivo identificato**: la scalabilità orizzontale al crescere dei nodi (Asse 1) peggiora anziché migliorare le prestazioni. Questo non contraddice la teoria dello sharding di per sé, ma evidenzia il limite del testbed: tutti i nodi condividono la stessa CPU e la stessa rete virtuale. In un'azienda reale, con nodi su macchine fisiche distinte, ci si attende che il beneficio teorico dello sharding si manifesti.

Il punto precedente trova una conferma diretta nella sezione seguente.

9 Validazione su una seconda piattaforma (Windows)

L'intera pipeline sperimentale (Fasi 2-4) è stata rieseguita, senza alcuna modifica al codice, su una seconda macchina fisica con Windows. Lo scopo non è quello di verificare quanto i risultati discussi finora dipendano dal singolo host usato.

9.1 Prestazioni assolute: un divario atteso

La macchina Windows è circa 5 volte più lenta in termini assoluti. Questo di per sé non è sorprendente (hardware diverso, e soprattutto backend di virtualizzazione Docker diverso) e da

Tabella 5: Fase 2 (baseline): confronto tra le due macchine

Scenario	macOS (ops/sec)	Windows (ops/sec)	Rapporto
Write 100B	269.550	52.109	5,2×
Read 100B	304.680	55.322	5,5×
Write 1KB	216.615	47.751	4,5×
Read 1KB	288.708	55.010	5,3×

solo non dice nulla su Redis: conferma però, con un secondo dato indipendente, che il numero assoluto di ops/sec misurato in questo lavoro è un artefatto della macchina di test, non una proprietà di Redis.

9.2 Un risultato più interessante: l'andamento cambia, non solo il valore assoluto

Se il divario di prestazioni assolute era prevedibile, non lo è la Tabella 6: l'*andamento* al crescere del numero di master, per tre scenari su quattro, cambia segno tra le due macchine.

Tabella 6: Asse 1: variazione % tra 3 e 7 master, macOS vs Windows

Scenario	Variazione macOS	Variazione Windows
Write 100B	$-17,4\%$	$+8,6\%$
Read 100B	$-23,8\%$	$-11,0\%$
Write 1KB	$-26,7\%$	$+39,4\%$
Read 1KB	$-36,6\%$	$-5,2\%$

Su macOS, aggiungere nodi peggiora sistematicamente le prestazioni (Sezione 6.1); su Windows, la scrittura migliora invece all'aumentare dei master (fino a +39% sul payload 1KB), mentre la lettura peggiora ma in misura più contenuta. Le due macchine raccontano quindi due storie di scalabilità diverse, a tratti opposte, per lo *stesso* codice e la *stessa* architettura software. Questo è, di fatto, l'evidenza più forte raccolta in questo lavoro a sostegno della tesi già discussa in Sezione 7: le curve di scalabilità misurate qui non sono una proprietà di Redis, ma il prodotto dell'interazione tra il carico e le specificità di virtualizzazione della singola macchina host. Una stessa architettura, distribuita su hardware diverso, può scalare in direzioni opposte motivo in più per non trarre conclusioni di capacity planning da un proof-of-concept su singolo host, qualunque esso sia.

Anche la conferma della teoria $O(1)$ in lettura sull'Asse 2 (Sezione 6.2), solida su macOS (throughput piatto 1M→10M), non si ripete allo stesso modo su Windows, dove la lettura cala del 29% passando da 1M a 10M chiavi.

9.3 Conclusione della validazione incrociata

La replica su una seconda piattaforma non ha prodotto "second draft" dei medesimi risultati, ma un secondo esperimento indipendente che rafforza la lezione metodologica centrale di questo lavoro: su un singolo host, indipendentemente da quale, il comportamento di scalabilità osservato è dominato dalle caratteristiche di quell'host più che dall'architettura distribuita in sé. Per un'azienda che debba decidere un'infrastruttura di produzione, la conclusione operativa è la stessa già espressa in Sezione 7, ma ora con doppia evidenza empirica a supporto: la scalabilità di Redis va validata sull'infrastruttura target, non stimata da un ambiente di sviluppo locale, quale che sia il sistema operativo usato.

10 Conclusioni

La sperimentazione condotta conferma la validità di Redis come soluzione di data store per un ambiente di produzione aziendale, dimostrando che l'impianto teorico si traduce in garanzie funzionali solide, pur con vincoli architetturali ben precisi legati al deployment. Sulla base dei dati raccolti, emergono le seguenti evidenze operative per l'adozione in azienda:

- **Affidabilità e tolleranza ai guasti confermate**: Le proprietà core e i meccanismi di alta affidabilità operano come da specifiche. Il failover automatico garantisce un ripristino del servizio nell'ordine di pochi secondi; accoppiato a un'opportuna logica di retry lato client, rende i guasti infrastrutturali quasi del tutto trasparenti all'applicazione.
- **Impatto della persistenza**: Il differenziale di throughput tra lettura e scrittura è fisiologico. L'overhead imposto dall'append sincrono all'AOF e l'aumento della latenza sui payload più pesanti impongono all'azienda di calibrare attentamente le policy di `fsync` in base al reale compromesso richiesto tra durabilità del dato e massime prestazioni.
- **Tenuta del volume dati**: La stabilità del throughput di lettura al decuplicare delle chiavi (da 1M a 10M) valida empiricamente l'efficienza algoritmica di Redis. La scalabilità verticale del singolo nodo è garantita finché i dati risiedono fisicamente nella memoria RAM.
- **Il limite del testbed locale**: I risultati contrastanti ottenuti sull'aggiunta di master tra macOS e Windows dimostrano inequivocabilmente che i colli di bottiglia osservati sono artefatti del livello di virtualizzazione dell'host condiviso, non limiti dell'architettura a cluster di Redis.

L'adozione di Redis in produzione è pertanto fortemente raccomandata per le sue eccellenti caratteristiche di base. Tuttavia, il processo di *capacity planning* aziendale non può basarsi su benchmark eseguiti in ambienti di sviluppo locali. Il passo successivo obbligato per l'azienda sarà misurare la reale scalabilità orizzontale eseguendo test di carico direttamente sull'infrastruttura distribuita di destinazione, assegnando risorse computazionali e di rete dedicate a ciascun nodo del cluster.