Chameleon Table

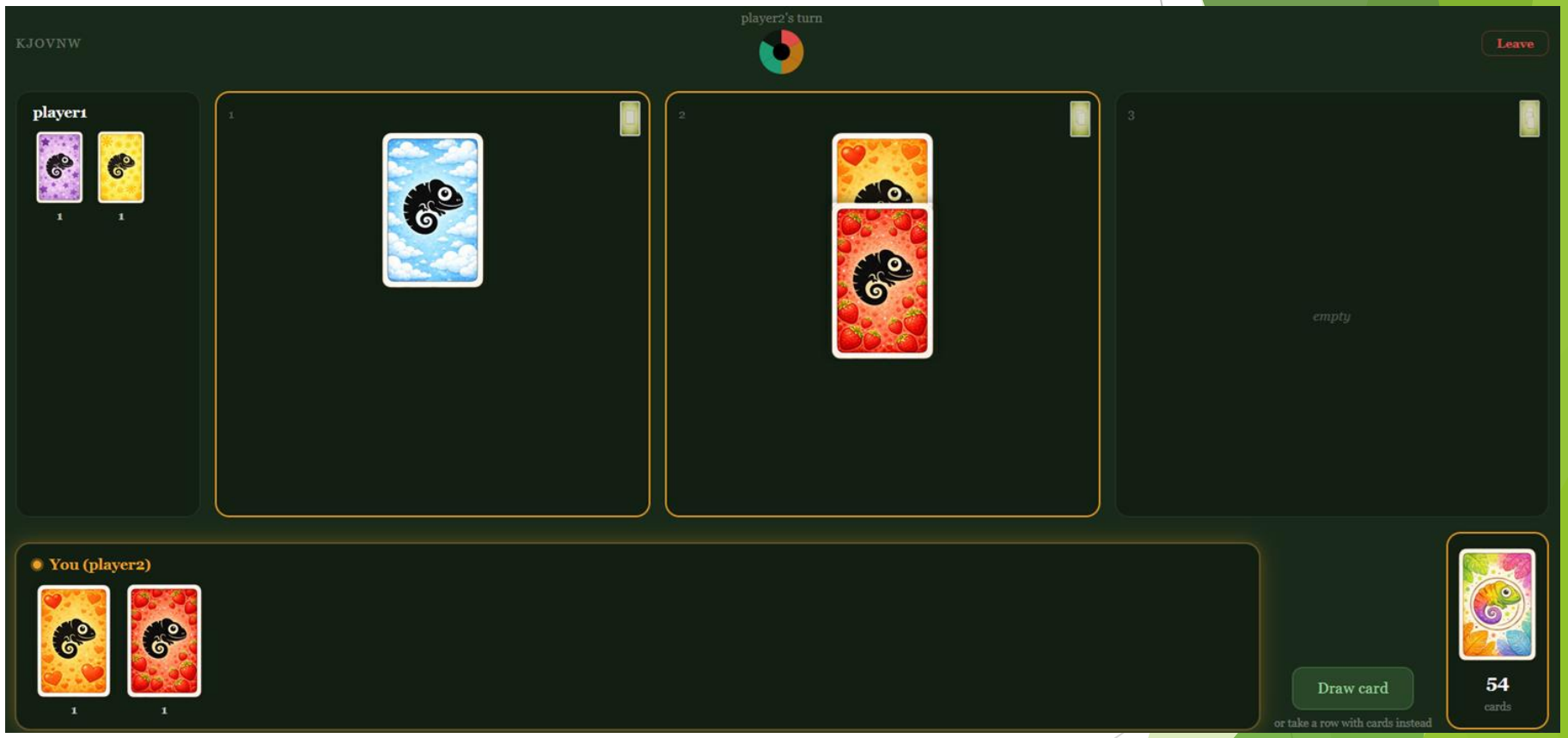
A Distributed Multiplayer Card Game

Andrea La Rosa

Sistemi Distribuiti — A.A. 2025/2026

Cos'è Chameleon Table

- Gioco di carte ispirato a Coloretto
- Da 2 a 5 giocatori + fino a 4 osservatori
- Browser desktop, nessuna installazione
- Azioni rapide: un minuto per agire



Perché serve la distribuzione

- **Resource sharing**

stato di partita condiviso da dispositivi e luoghi diversi — la ragione primaria

- **Fault tolerance**

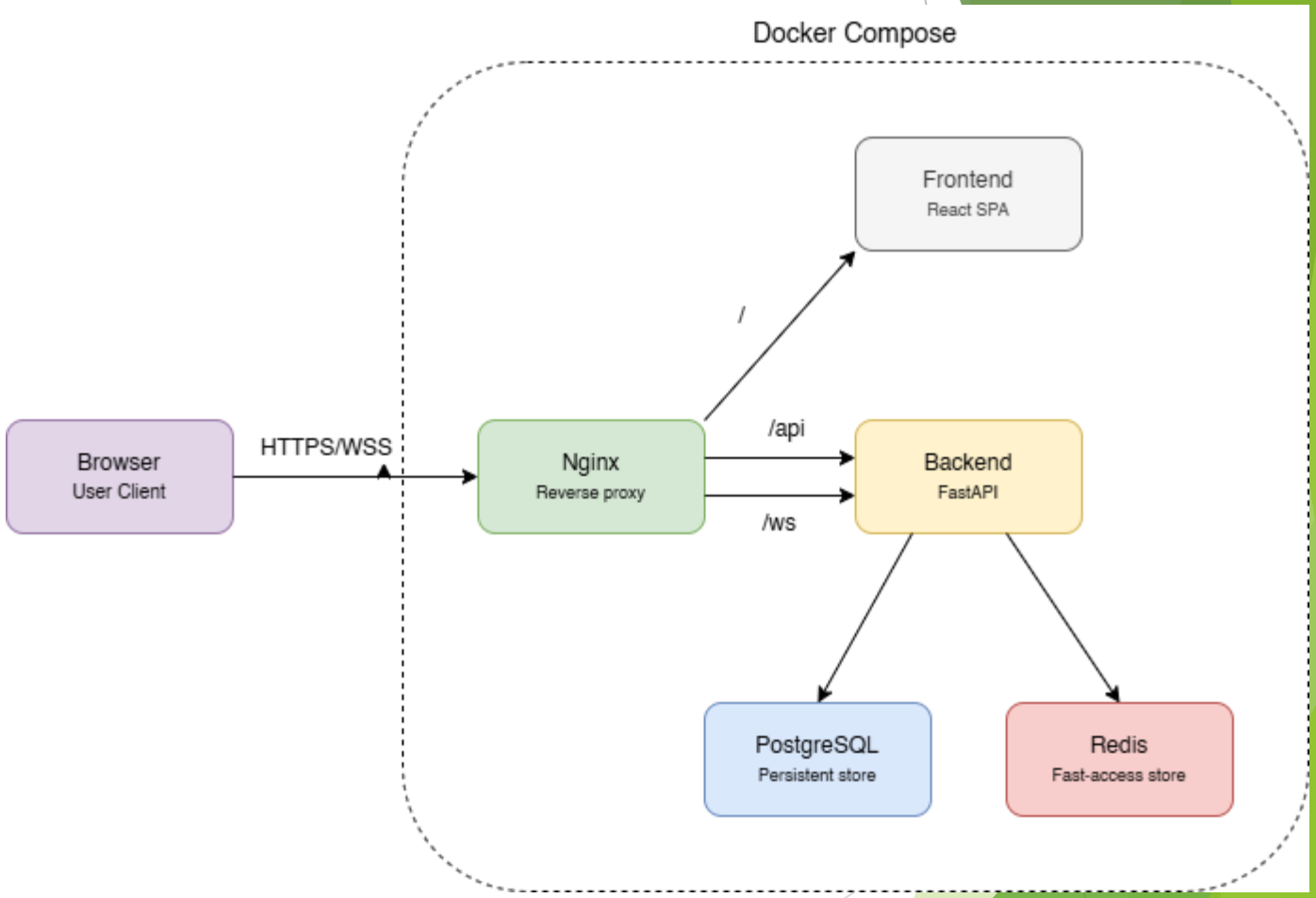
le partite attive sopravvivono al riavvio del server

- **Liveness, con bound esplicito**

una partita può congelarsi fino a tre minuti, mai indefinitamente

Architettura: server autoritativo

- Nginx — unico punto d'ingresso
- Backend FastAPI — unica autorità sullo stato
- Redis — stato caldo della partita
- PostgreSQL — persistenza e recovery
- Client — display, non repliche



Perché l'autorità sta tutta sul server

- **Information hiding**

Mazzo e ordine delle carte non lasciano mai il server

- **Autorità sul turno**

il server è l'unico a stabilire chi può agire, e quando

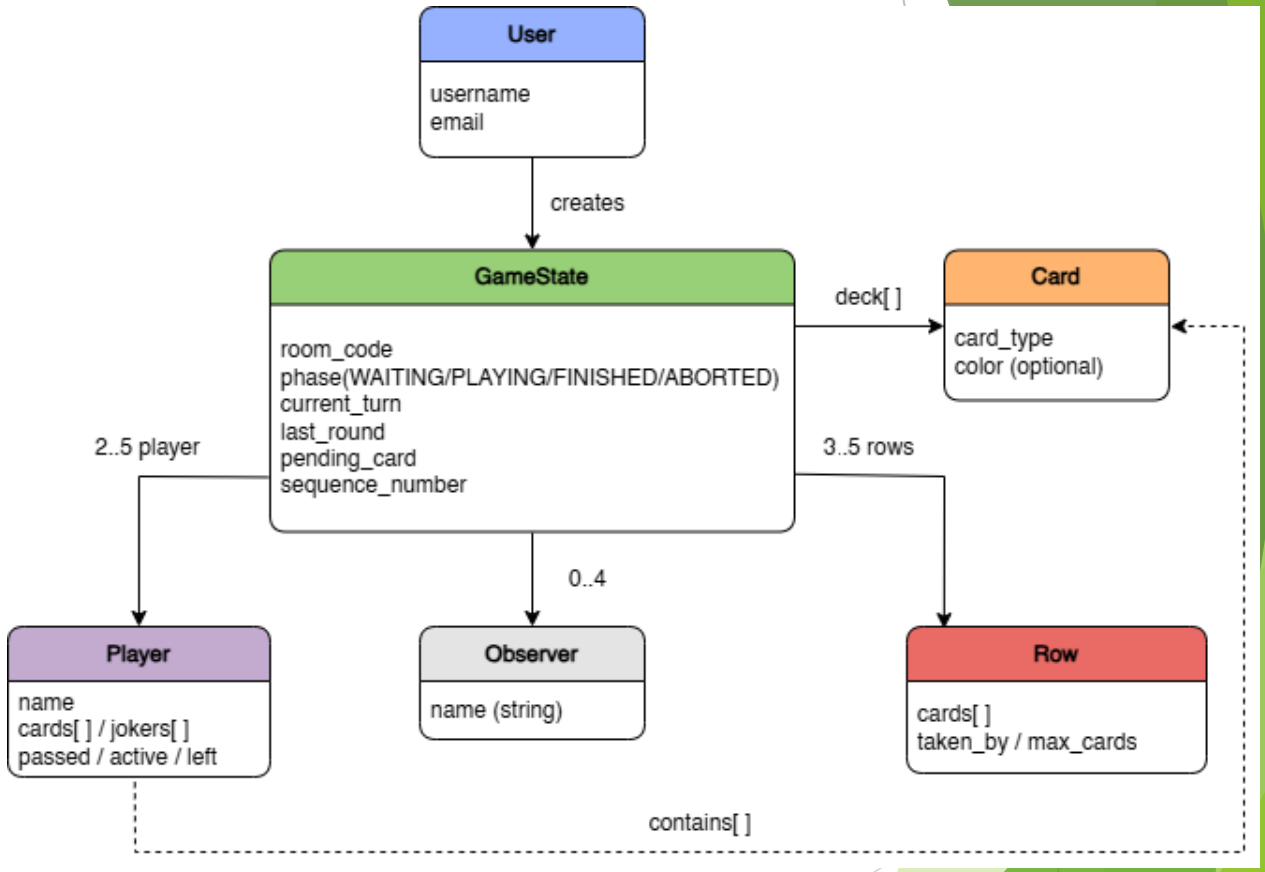
- **Recovery centralizzato**

basta uno snapshot per partita; in P2P servirebbe coordinazione tra i pari

Il prezzo: single point of failure.
Mitigato dalla crash recovery, non eliminato.

Dove vive lo stato

- Stato di partita → Redis, con snapshot su PostgreSQL ad ogni azione
- Dati utente → solo PostgreSQL, mai in Redis
- Client → copia per la sola visualizzazione



Unica fonte di verità.

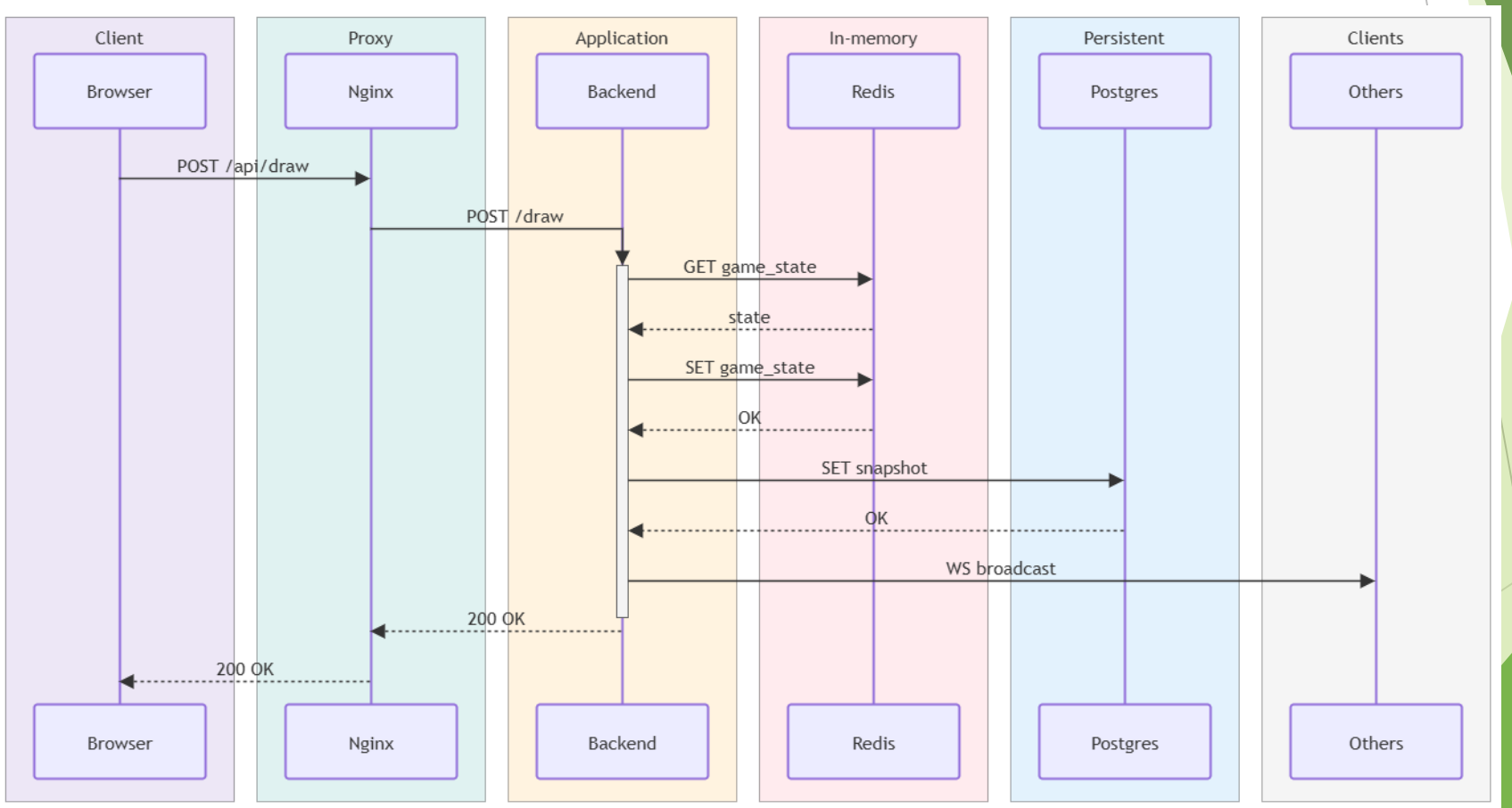
Due canali, due pattern di interazione

- **REST — request-response**

azioni: pesca, piazza, prendi fila

- **WebSocket — push**

lo stato verso tutti i client della room



Il client non chiede lo stato: lo riceve — salvo risincronizzazione

Aggiornamento in tempo reale
link al video

Il sequence number

Non serve a riordinare i messaggi — dentro una connessione TCP lo fa già il trasporto.

Serve a farmi accorgere che sono rimasto indietro:

- Race tra la risposta sincrona di un'azione REST e il broadcast che la stessa azione genera
- Riconnessione: una connessione nuova non ha ordine rispetto a quella chiusa
- Risincronizzazione: heartbeat e recovery da snapshot stale

Concorrenza: il lock, e il suo costo

- Un asyncio.Lock per room serializza le azioni sulla stessa partita
- Room diverse non si bloccano tra loro: parallelismo preservato

Il costo, dichiarato: il broadcast è atteso client per client dentro il lock.
Un client lento ritarda tutti gli altri — head-of-line blocking.

Accettabile con 5 giocatori e 4 osservatori. Da rivedere prima di crescere.

Fault tolerance: guasto → scelta → prezzo

Guasto	Scelta	Prezzo
Cade un client	Grace period, turno congelato; poi esclusione, se serve abort	Prediligo la consistenza, pago in continuità
Cade PostgreSQL	Degrado controllato: si prosegue su Redis	Prediligo la continuità, pago in durabilità
Cade Redis	Rollback recovery dal checkpoint su Postgres	Non è un trade-off: è un guasto. Costo = MTTR
Guasti in sequenza	Il ripristino fa regredire il sequence number	Unica violazione dichiarata di monotonic reads

Demo — crash recovery

Crash recovery
Link al video

Tre guasti di Redis, tre meccanismi

Il backend riparte

Eager, globale — una passata al lifespan ripara tutto

Redis perde la chiave

Lazy, per-room — cache-aside alla prima richiesta

Chiave presente ma stale

Watcher: PING periodico, reload sul fronte di salita

Coprono guasti mutuamente esclusivi: ne scatta sempre esattamente uno.

Progettarli tutti e tre riconosce che i componenti falliscono in modo indipendente.

Demo — riavvio silenzioso di Redis ?

Riavvio silenzioso di Redis Link al video

Cosa questo sistema non è

CAP	non ho repliche da partizionare — non è per il single point of failure
Channel state	stato in un posto solo, nessun messaggio in transito tra processi
Consenso	autorità unica, nessuna vista divergente da riconciliare
Logging	salvo stati, non eventi: sono checkpoint-based

Tre su quattro esclusi dalla stessa proprietà: l'unicità dell'autorità.

In positivo — PACELC, ramo ELC: nessuna partizione tra Redis e PostgreSQL, eppure privilegio deliberatamente la latenza sulla consistenza.

Validazione

87 test automatici

- 49 unit — logica di gioco isolata
- 25 integration — stack HTTP completo
- 13 fault tolerance e timing

Cosa i test non hanno preso

- Mixed content: ws:// hardcoded, invisibile in locale
- Zombie connection su rete mobile reale
- Sequence number mancante su join e abort

Trovati testando in produzione e su rete reale, non dalla suite. La copertura delle transizioni di stato meno comuni resta il limite.

Limiti e sviluppi

- Istanza singola: nessuno scaling orizzontale, single point of failure
- Frontend minimale, non ottimizzato per mobile

Redis pub/sub risolverebbe esattamente una cosa: propagare il broadcast tra istanze.

- quale istanza tiene il lock per una data room
- come resta raggiungibile una WebSocket accettata da un'altra istanza
- se le azioni sono idempotenti, e quindi sicure da ritentare

Pianificato, non risolto.

Grace Period on player
Link al video