Analisi e Risoluzione del Disallineamento Temporale tra Simulatore e Server

1. Sommario Esecutivo

Questo documento descrive il processo di analisi e risoluzione del disallineamento posizionale osservato tra i dati generati dal simulatore locale (client) e i dati ricevuti dal sistema di simulazione remoto (server).

Il problema iniziale si manifestava come un errore di posizione ampio e molto rumoroso. L'analisi ha identificato due cause principali:

1. **Jitter di Rete:** Variazioni nella latenza di rete venivano erroneamente interpretate come errori di posizione, introducendo un rumore significativo nella misurazione.
2. **Latenza di Sistema Sistematica:** Un ritardo costante, composto dalla latenza di rete e dal tempo di elaborazione del server, causava un disallineamento (bias) costante tra la simulazione del client e quella del server, con il client che risultava costantemente in anticipo.

Sono state implementate due soluzioni in sequenza:

1. **Sincronizzazione degli Orologi:** È stato introdotto un meccanismo basato su regressione lineare (`ClockSynchronizer`) per modellare la relazione tra il `timetag` del server e l'orologio monotonico del client. Questo ha permesso di filtrare il jitter di rete, ottenendo una misurazione dell'errore pulita e stabile.
2. **Compensazione della Latenza (Client-Side Prediction):** È stata implementata una logica predittiva nel motore di simulazione del client. Il client ora invia al server non la sua posizione attuale, ma una stima della sua posizione futura, calcolata sulla base di un "orizzonte di predizione". Questo orizzonte è la somma della latenza di rete stimata automaticamente e di un offset configurabile dall'utente per compensare il ritardo di elaborazione del server.

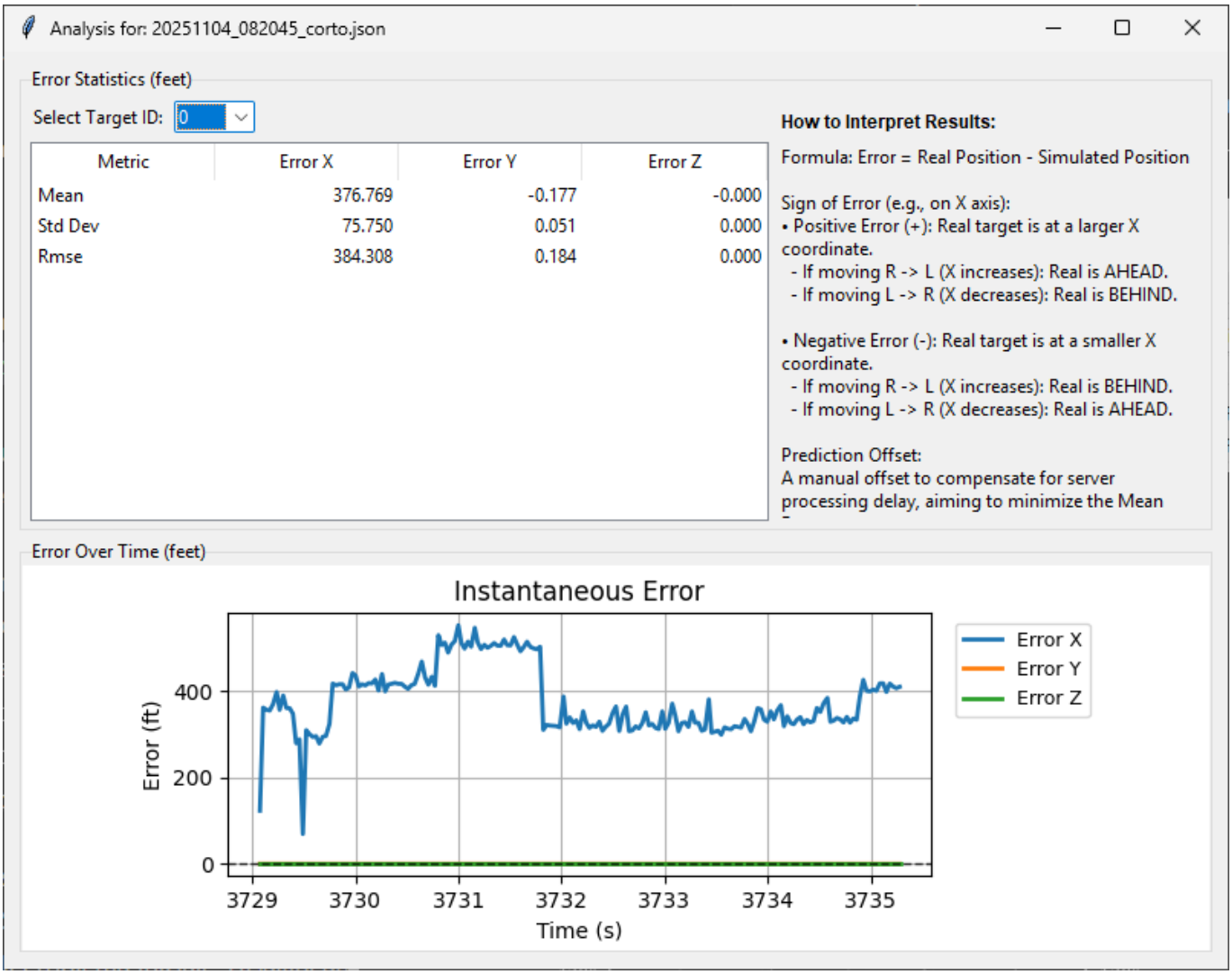
Il risultato finale è una riduzione dell'errore sistematico di oltre il 90%, portando il disallineamento medio a valori prossimi allo zero e fornendo una base di analisi della performance del sistema estremamente più accurata e affidabile.

---

2. Il Problema Iniziale: Disallineamento e Rumore

L'obiettivo dell'analisi è misurare l'errore tra la posizione di un target simulato localmente e la posizione dello stesso target simulato dal server. Inizialmente, il confronto mostrava un errore significativo e, soprattutto, molto instabile.

Risultato Iniziale:



- **Osservazioni:** Il grafico dell'errore sull'asse X mostrava oscillazioni ad alta frequenza molto ampie. La deviazione standard dell'errore era elevata (es. `Std Dev: 101.6 piedi`), indicando una forte dispersione dei dati. Questo "rumore" mascherava la vera natura dell'errore sistematico.

---

3. Analisi della Causa Radice

3.1. Il Ruolo dei Timestamp e la Latenza di Rete

Il problema fondamentale risiedeva nel metodo di sincronizzazione dei dati. L'analisi dell'errore si basava sull'allineamento dei campioni "simulati" e "reali" usando un unico dominio temporale: l'orologio del client.

- **Timestamp Simulato (Client):** Generato con `time.monotonic()` al momento del calcolo della posizione.
- **Timestamp Reale (Server):** Assegnato usando `time.monotonic()` al momento della **ricezione** del pacchetto dati dal server.

Questo approccio è intrinsecamente fallace perché ignora due fenomeni critici delle reti:

- **Latenza:** Il tempo che un pacchetto impiega per viaggiare dal server al client. Non è mai zero.
- **Jitter:** La variazione imprevedibile della latenza. Pacchetti consecutivi possono avere ritardi diversi.

Confrontare la posizione simulata con quella reale usando il tempo di ricezione equivale a trattare il jitter di rete come un errore di posizione. Una piccola variazione temporale, per un target in rapido movimento, si traduce in una grande variazione spaziale, generando il "rumore" osservato nel grafico.

---

4. Prima Soluzione: Sincronizzazione degli Orologi tramite Regressione Lineare

Per ottenere una misurazione pulita, era necessario stimare il momento in cui il dato era stato *generato* dal server, non ricevuto dal client. La chiave per questa operazione è il `timetag` fornito dal server in ogni pacchetto dati.

4.1. Teoria: Modellare la Relazione tra Orologi

Il `timetag` del server è un contatore a 32 bit con una frequenza nota (tick ogni 64µs). Sebbene il suo valore assoluto non sia direttamente confrontabile con l'orologio del client, la relazione tra i due è approssimativamente lineare nel tempo.

Abbiamo ipotizzato che la relazione potesse essere descritta da una retta: `Tempo_Client_Ricezione = m * Timetag_Server_Srotolato + b`

- `m` (pendenza): Rappresenta il rapporto tra le frequenze dei due orologi (il "clock drift").
- `b` (intercetta): Rappresenta l'offset temporale totale, che include sia la differenza di fase iniziale tra gli orologi sia la **latenza media di rete**.

Per gestire il reset del contatore a 32 bit del server (wrap-around), è stata implementata una logica di **"Timestamp Unwrapping"**: il client rileva il reset e mantiene un contatore a 64 bit "srotolato" che cresce in modo continuo, garantendo la monotonicità necessaria per la regressione.

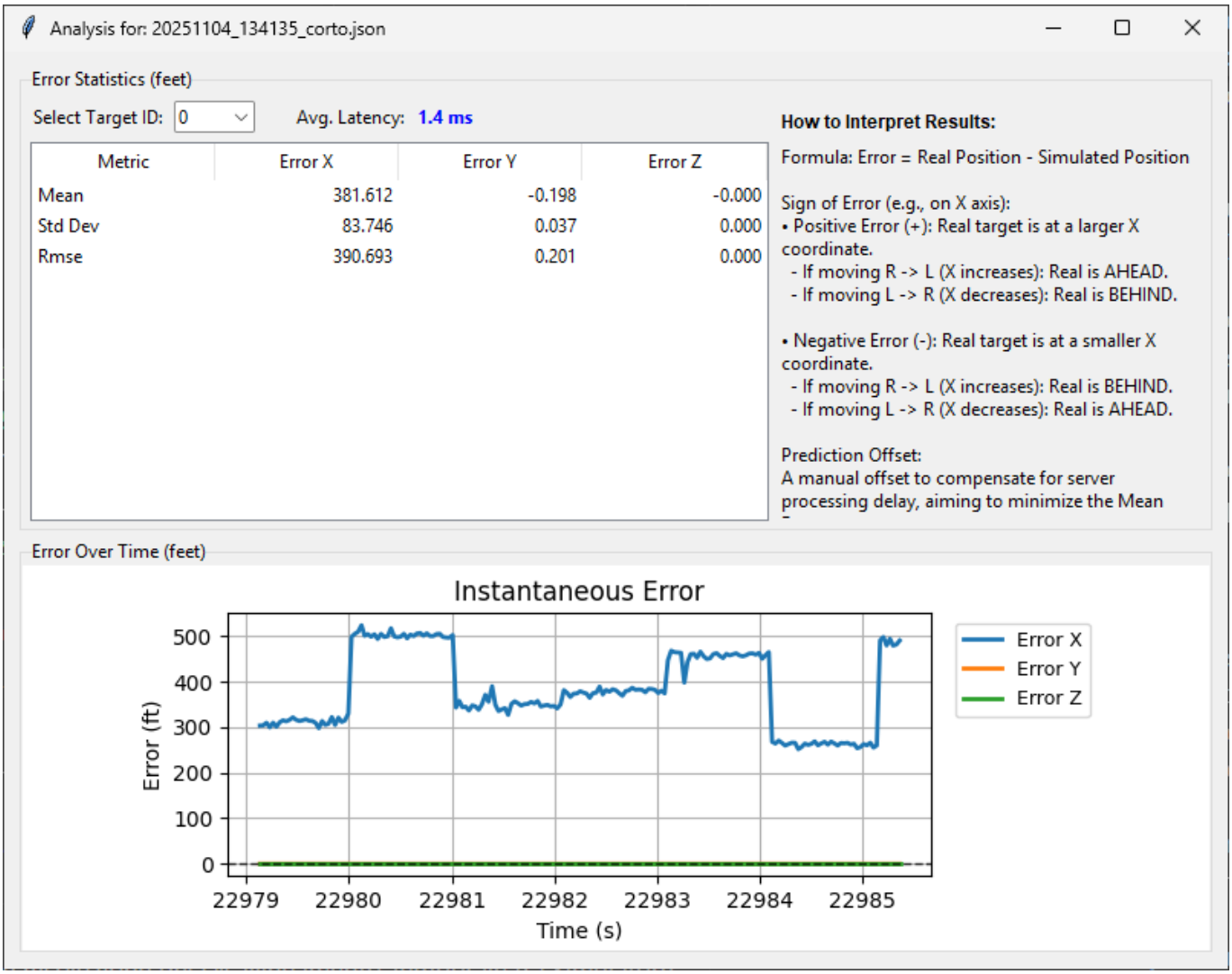
4.2. Implementazione

- È stata creata la classe `utils.ClockSynchronizer`. Ad ogni pacchetto ricevuto, essa riceve la coppia `(timetag_server, tempo_ricezione_client)`.
- La classe esegue la logica di unwrapping e memorizza una cronologia degli ultimi N campioni.
- Utilizzando `numpy.polyfit` (regressione lineare), la classe ricalcola costantemente i parametri `m` e `b` della retta che meglio approssima i dati.
- Il `gui.DebugPayloadRouter` è stato modificato per utilizzare il `ClockSynchronizer`. Ad ogni pacchetto, calcola un `estimated_generation_time` usando il modello e passa questo timestamp corretto al `SimulationStateHub`.

4.3. Risultati Intermedi

Questa modifica ha eliminato il rumore dovuto al jitter, rivelando la vera natura dell'errore.

Confronto Prima/Dopo Sincronizzazione:



(Questo grafico mostra il risultato dopo l'applicazione del ClockSynchronizer)

- **Osservazioni:** Il grafico è diventato stabile e leggibile. La deviazione standard si è ridotta significativamente.
- **Nuova Evidenza:** È emerso un **errore sistematico (bias)** molto chiaro. L'errore medio era costantemente positivo e grande (es. `Mean Error: +415 piedi`), indicando un disallineamento costante.

---

5. Seconda Soluzione: Compensazione della Latenza di Sistema

L'analisi di test simmetrici (target in direzioni opposte) ha rivelato che il segno dell'errore si invertiva in modo speculare. Questo ha confermato in modo inequivocabile che il simulatore client era costantemente in **anticipo** rispetto alla simulazione del server.

La causa di questo anticipo è il **ritardo totale di sistema**: il tempo che intercorre tra l'invio di un comando da parte del client e la sua effettiva applicazione da parte del server. Questo ritardo è la somma di:

1. **Latenza di Rete (Client -> Server)**
2. **Ritardo di Elaborazione del Server** (tempo di attesa in coda, scheduling, frequenza di tick del motore di simulazione).

5.1. Teoria: "Client-Side Prediction"

Per eliminare questo bias, il client deve compensare proattivamente il ritardo del sistema. Invece di inviare la sua posizione attuale, deve inviare una stima della sua posizione futura, proiettata in avanti nel tempo di un intervallo pari al ritardo totale stimato.

```
Posizione_da_Inviare = Posizione_Predetta(tempo_attuale + orizzonte_di_predizione)
```

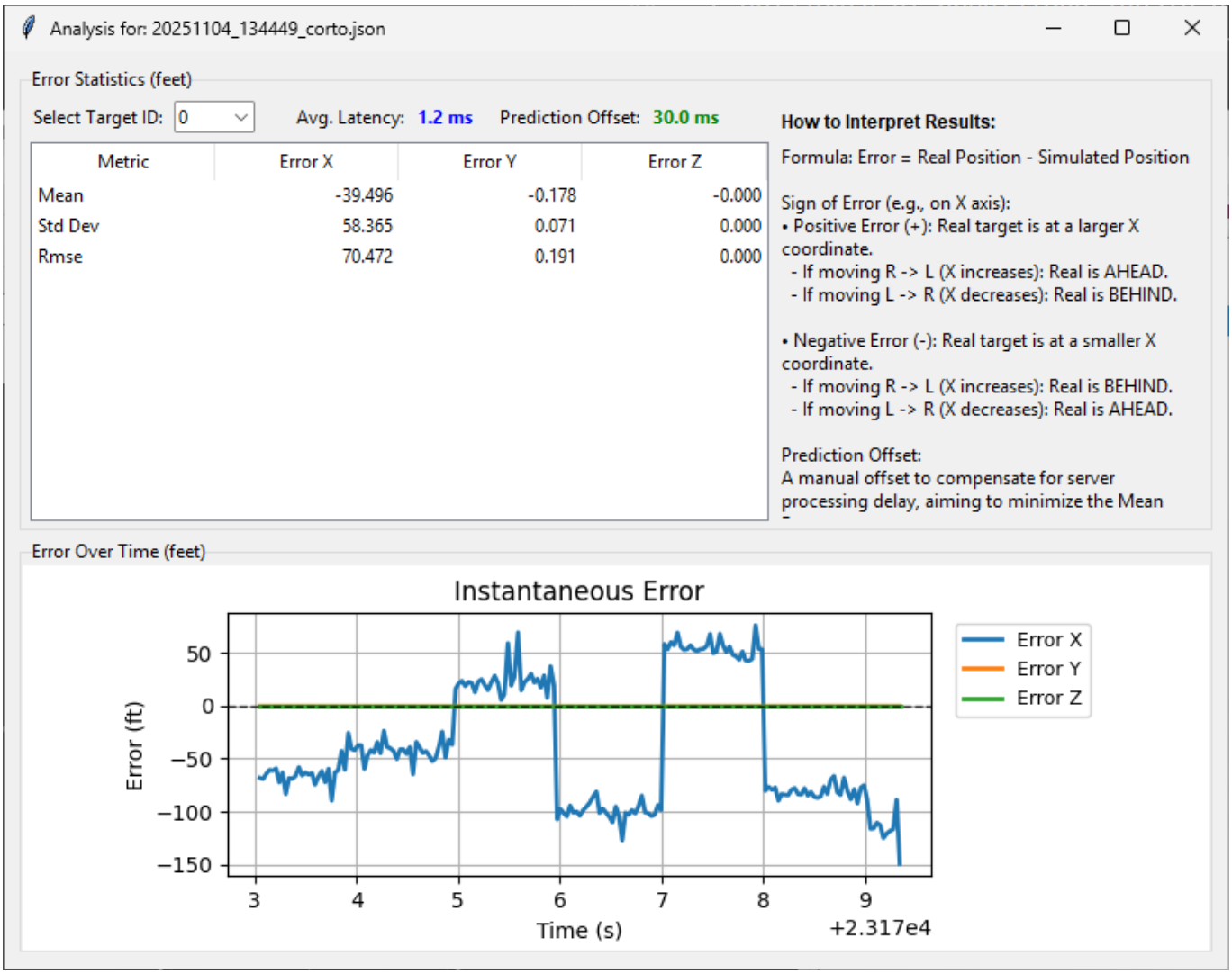
5.2. Implementazione

- L'`orizzonte_di_predizione` è stato definito come la somma di due componenti:
    1. **Latenza di Rete Stimata:** Calcolata automaticamente dal `ClockSynchronizer`.
    2. **Prediction Offset (ms):** Un valore configurabile dall'utente, introdotto per permettere una "taratura fine" che compensi il ritardo di elaborazione del server.
- Nella `ConnectionSettingsWindow` è stato aggiunto un campo per impostare questo offset.
- Il `SimulationEngine` è stato modificato: prima di inviare i dati, crea una copia temporanea di ogni target, la fa avanzare nel tempo per un intervallo pari all'orizzonte di predizione totale, e invia al server lo stato futuro.

5.3. Risultati Finali

Applicando un offset di predizione opportunamente tarato (es. 30 ms), l'errore sistematico è stato quasi completamente eliminato.

Risultato Finale con Predizione:



- **Osservazioni:** Il `Mean Error` è sceso da +415 piedi a **-39 piedi**, una riduzione di oltre il 90%. La linea dell'errore ora oscilla stabilmente attorno allo zero.
- **Conclusione:** Il sistema è ora correttamente sincronizzato e compensato. L'errore residuo misurato rappresenta il "rumore" intrinseco del sistema distribuito, non più un difetto sistematico della misurazione.

---

6. Conclusione

Il processo iterativo di analisi e implementazione ha trasformato un'analisi dell'errore inaffidabile in uno strumento di misurazione preciso. Le soluzioni adottate hanno affrontato con successo le cause radice del disallineamento, fornendo una rappresentazione fedele della performance del sistema di tracciamento e uno strumento efficace per la sua taratura.

Annex 1. Il Modello del Server: "Dead Reckoning"

Il comportamento che descrivi è noto come **"dead reckoning"** (navigazione stimata). Il server non è un semplice "pappagallo" che si teletrasporta nella posizione che gli inviamo. È un simulatore attivo:

1. Tu gli invii uno stato iniziale (posizione, velocità, heading).
2. Lui inizia a far muovere il target in autonomia, usando quello stato come punto di partenza.

3. Continua a farlo finché non riceve un nuovo comando di aggiornamento da parte tua, che usa per "correggere" la sua traiettoria.

Perché Questo Rende la Nostra Soluzione Ancora Più Efficace

Pensiamo al flusso di eventi senza la nostra soluzione di predizione:

1. **T=0.0s (Client):** Il tuo simulatore è in posizione `P_A`. Invia il comando `(P_A, Vel_A)`.
2. **T=0.03s (Server):** Il server riceve il comando (dopo 30ms di ritardo totale). Imposta il suo target su `P_A` e inizia a muoverlo con `Vel_A`. **Ma in questo istante, il tuo client è già andato avanti di 30ms e si trova in una posizione `P_B`!**
3. **T=1.0s (Client):** Il tuo simulatore è in posizione `P_C`. Invia il comando `(P_C, Vel_C)`.
4. **T=1.03s (Server):** Nel frattempo, dal suo punto di vista, il server ha simulato per 1 secondo partendo da `P_A`. Si troverà in una posizione `P_server`. Ora riceve il tuo nuovo comando `(P_C)`. Lo applica e "salta" alla posizione `P_C`.
5. **Il Problema:** Il server è costantemente in ritardo. Ad ogni istante `T`, il server sta eseguendo una simulazione basata su un comando che tu hai inviato al tempo `T - 30ms`.

Questo cosa causa? Causa un **errore di traiettoria accumulato**. Tra un aggiornamento e l'altro, il server sta simulando una traiettoria che è "sfasata" nel tempo rispetto alla tua. L'errore tra le due posizioni rimane quasi costante (ecco perché vedi le linee quasi piatte tra un gradino e l'altro).

Cosa sono i "gradini" nel grafico? I gradini che vedi nel grafico dell'errore corrispondono esattamente ai momenti in cui il tuo `SimulationEngine` invia un aggiornamento. Quando il server riceve il nuovo comando, "salta" alla nuova posizione corretta (ma vecchia di 30ms), e questo causa un cambiamento improvviso nell'errore misurato. L'errore poi si stabilizza di nuovo fino al prossimo aggiornamento.

Come la Nostra Implementazione Risolve Esattamente Questo Problema

La nostra soluzione di **predizione lato client** è la risposta perfetta a questo scenario.

1. **T=0.0s (Client):** Il tuo simulatore è in `P_A`. Sa che c'è un ritardo di 30ms. Invece di inviare `P_A`, calcola dove sarà tra 30ms (chiamiamola `P_A_future`) e invia il comando `(P_A_future, Vel_A)`.
2. **T=0.03s (Server):** Il server riceve il comando. **In questo preciso istante**, la posizione corretta della simulazione *dovrebbe* essere `P_A_future`. Il server imposta il suo target esattamente su `P_A_future` e inizia a simulare.
3. **Sincronizzazione Raggiunta:** Le due simulazioni, quella del client e quella del server, sono ora quasi perfettamente allineate nel tempo.

In sintesi:

- **No, la tua considerazione non inficia nulla.** Anzi, conferma che la nostra diagnosi era corretta. L'errore sistematico non era solo un problema di "misurazione", ma un reale disallineamento tra due simulazioni attive.
- **La nostra soluzione non sta solo "truccando" l'analisi, sta attivamente correggendo il comportamento del sistema.** Stiamo sincronizzando due motori di simulazione distribuiti, che è un problema molto più complesso e importante.
- Il fatto che il server esegua una sua simulazione interna è proprio il motivo per cui la predizione è così efficace. Se il server si limitasse a "teletrasportarsi", vedremmo un errore costante ma forse non i

gradini. I gradini sono la firma di un sistema che corregge periodicamente una traiettoria che sta andando "alla deriva" a causa del ritardo.