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.

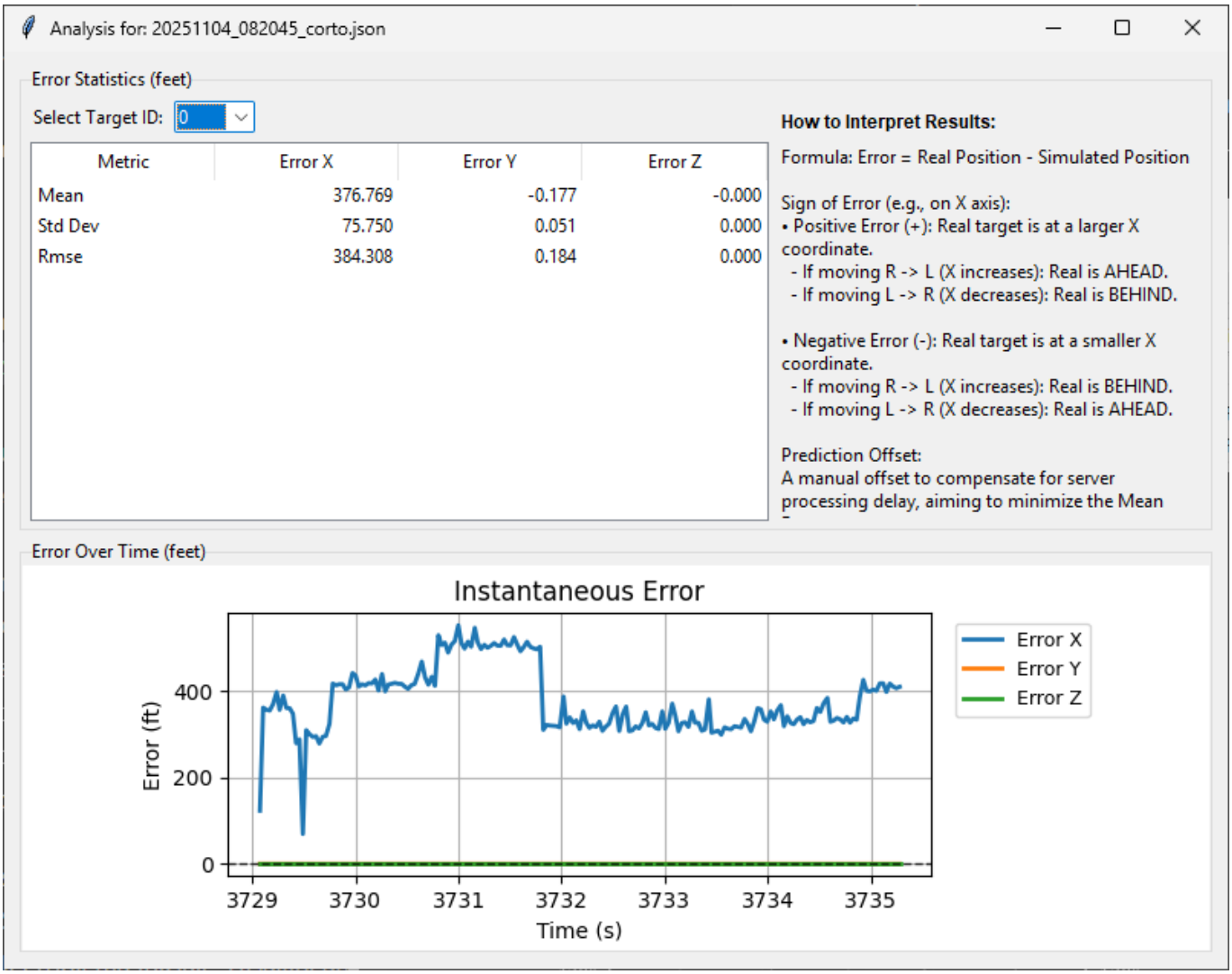
Il risultato finale è una riduzione dell'errore sistematico di oltre il 90%, portando il disallineamento medio a valori prossimi allo zero e fornendo una base di analisi della performance del sistema estremamente più accurata e affidabile.

---

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.

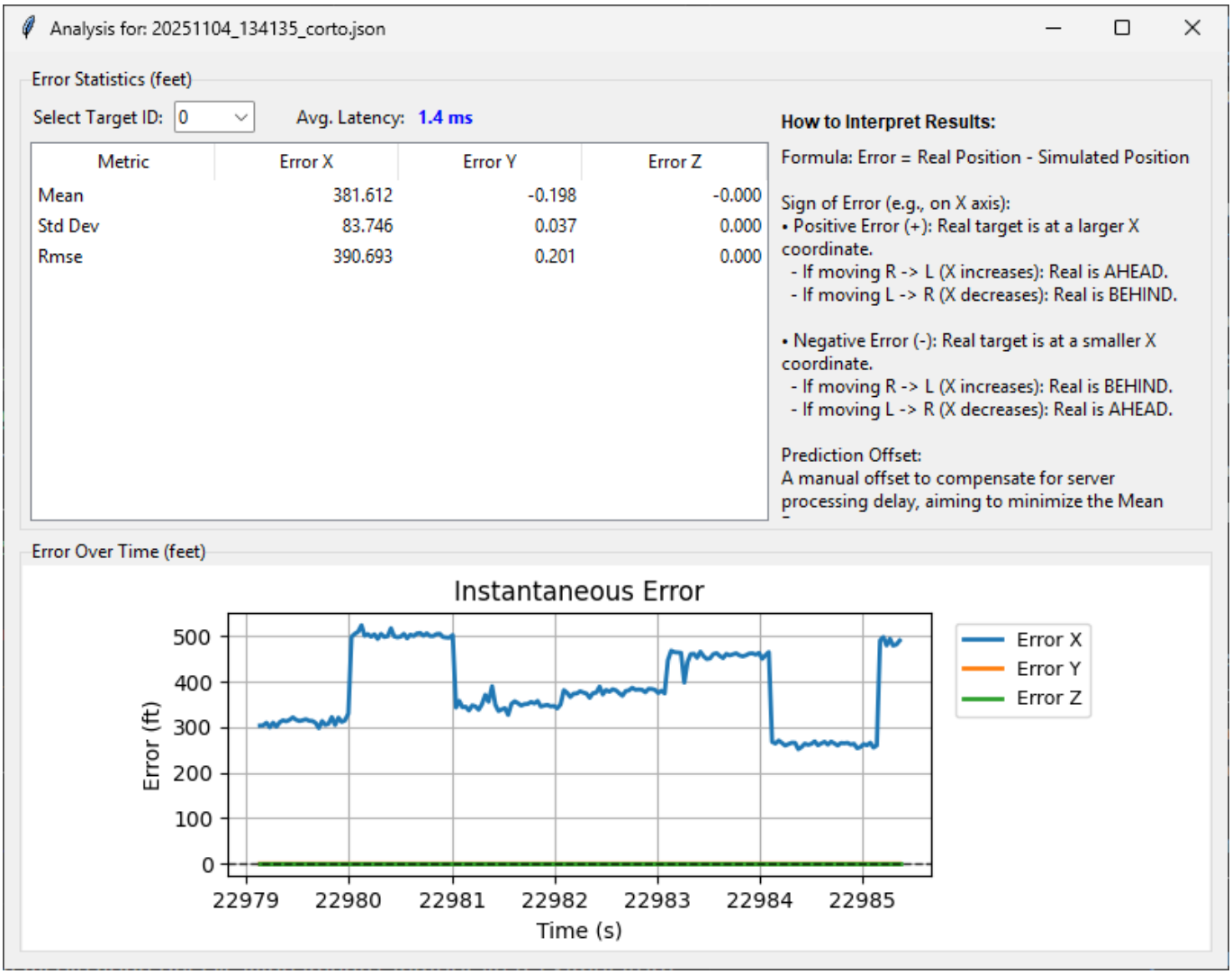
4.2. Implementazione

- 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)
```

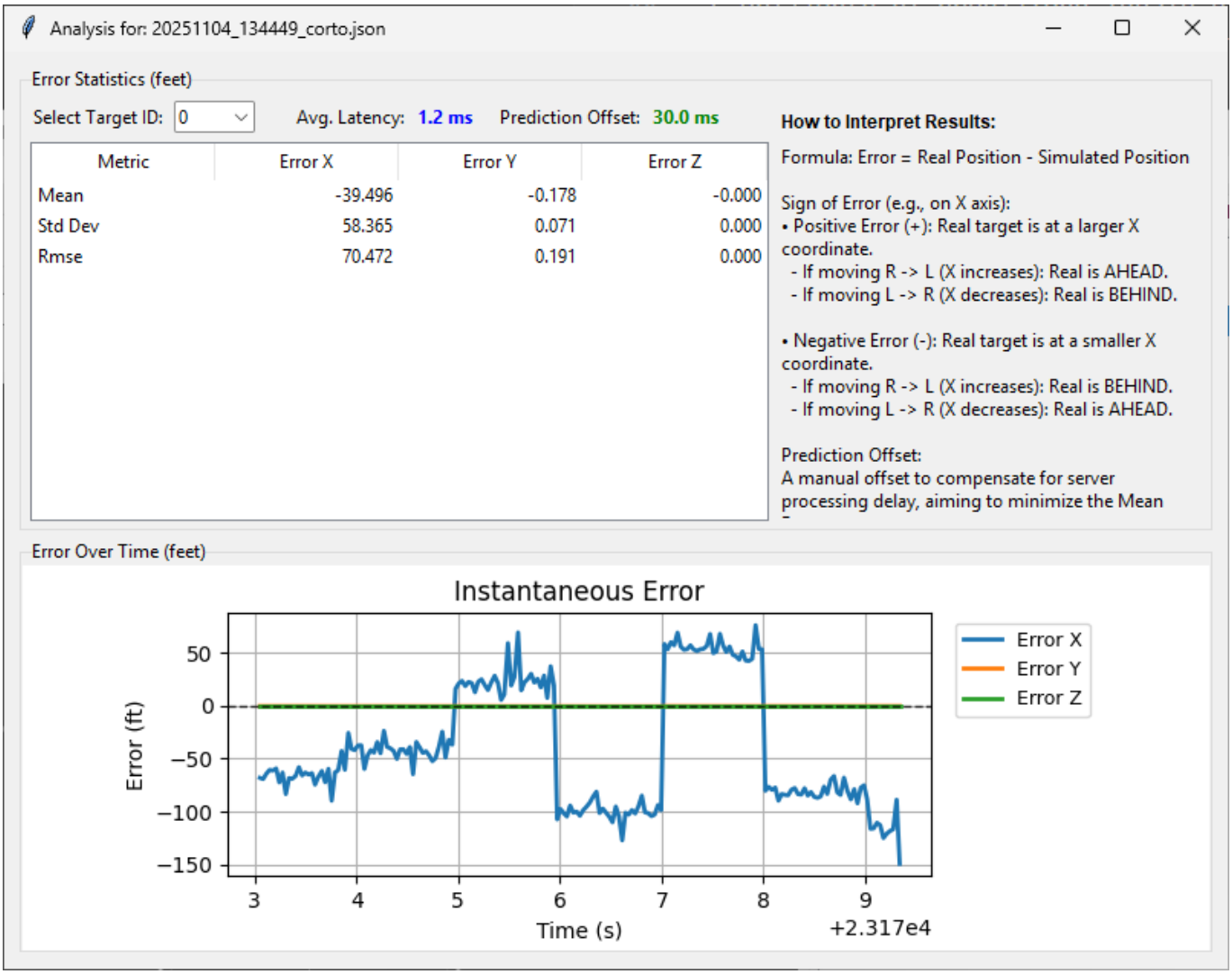
5.2. Implementazione

- 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.

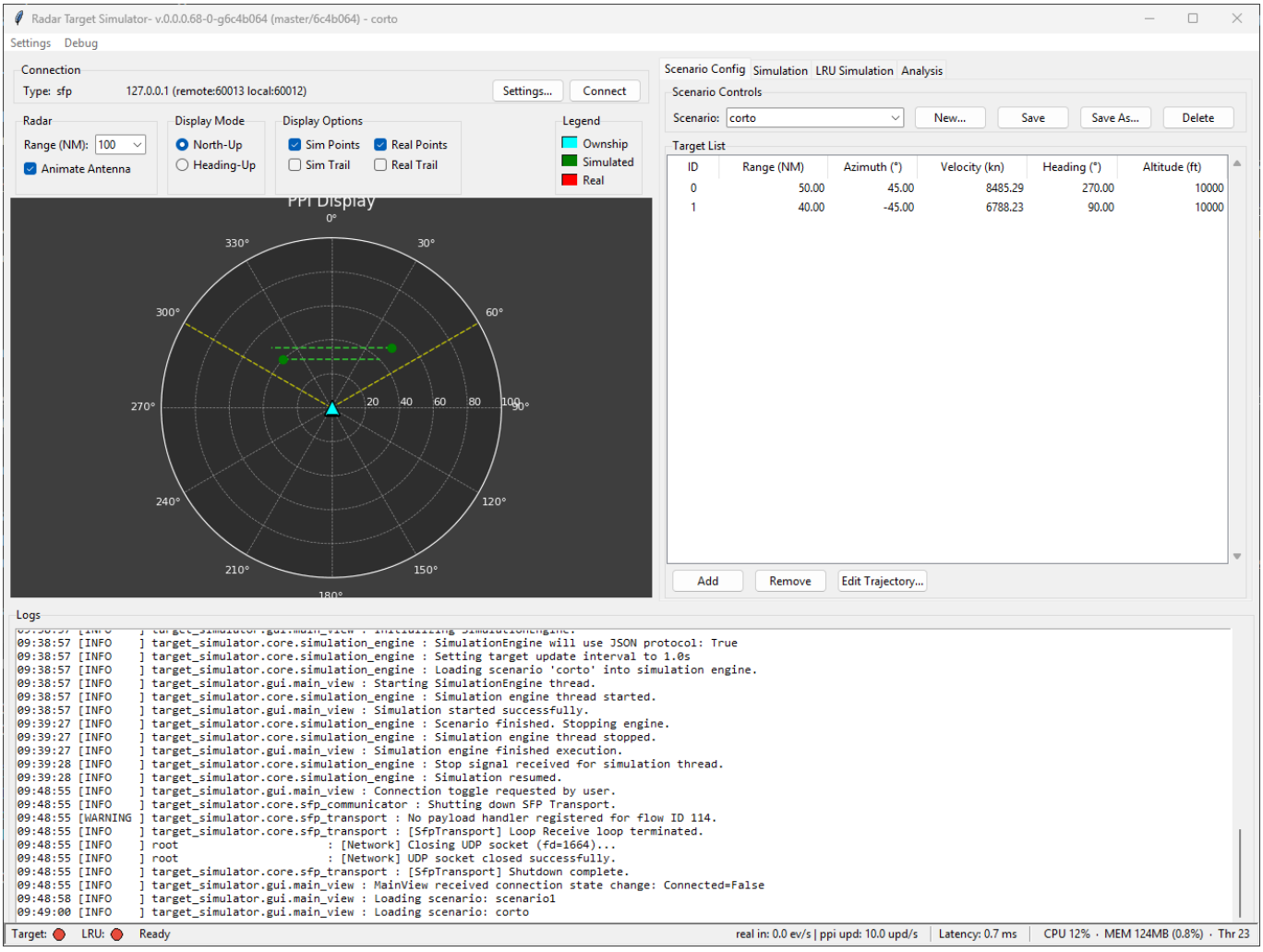
In sintesi:

- **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.

Annex 2. Analisi simulazione 1

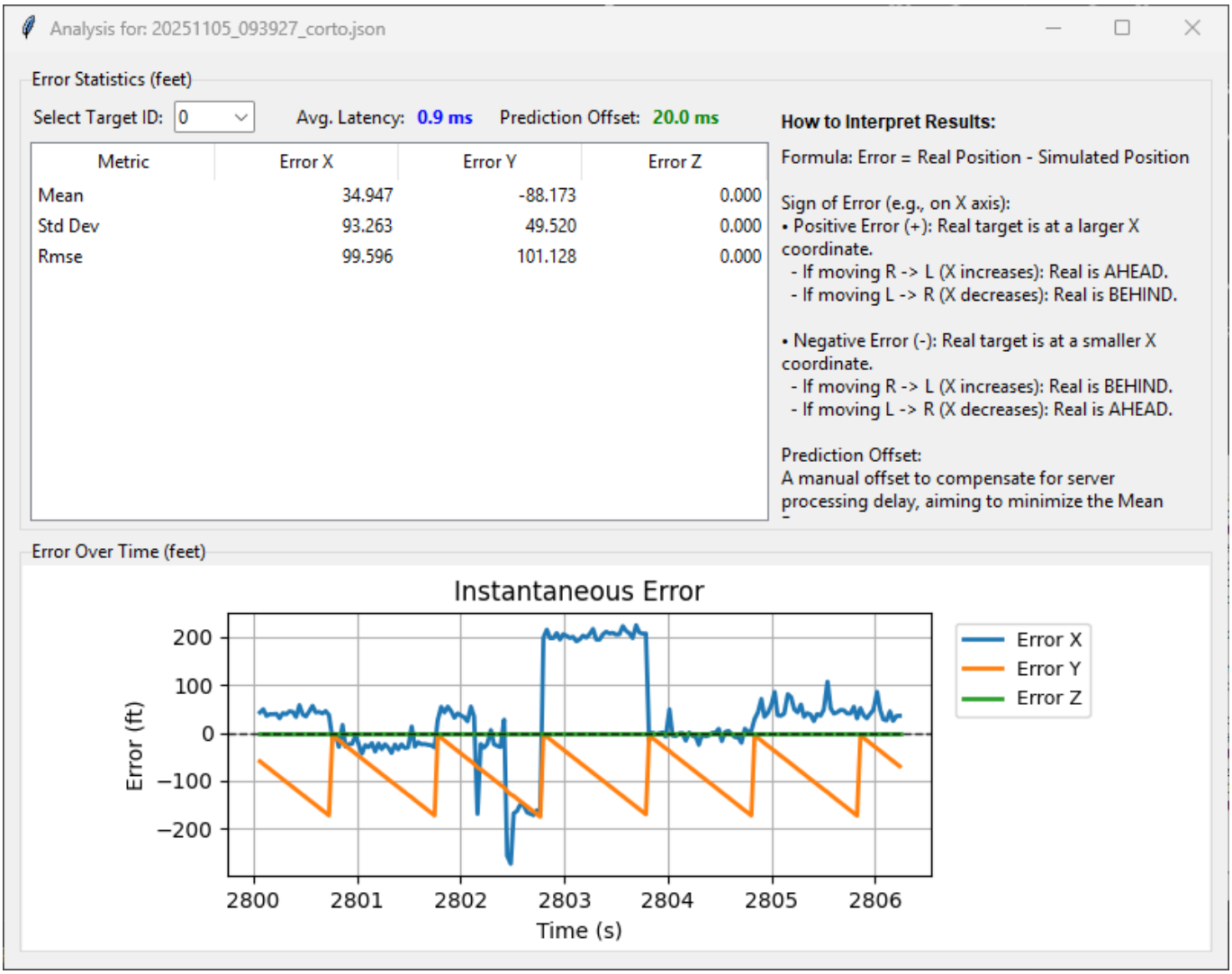
Analizziamo questo grafico. È un'ottima schermata perché mostra diversi fenomeni interessanti, soprattutto ora che abbiamo introdotto il movimento dell'ownship.



Contesto della Simulazione

- **Scenario Target**: Due target che si muovono orizzontalmente in direzioni opposte (uno da DX a SX, uno da SX a DX).
- **Movimento Ownship**: Velocità di 100 nodi con heading 0° (cioè, ci stiamo muovendo verso Nord).
- **Dati Visualizzati**: Stiamo osservando l'errore di tracciamento per il **Target 0**. L'errore è calcolato come `Posizione Reale - Posizione Simulata`.

Osservazioni Chiave e Analisi



1. Errore sull'asse Z (Linea Verde)

- **Osservazione**: L'errore sull'asse Z è costantemente `0.000`.
- **Analisi**: Questo è prevedibile e corretto. Nelle nostre simulazioni, sia il movimento dei target che quello dell'ownship avvengono su un piano 2D (orizzontale). Non ci sono accelerazioni o movimenti verticali. Di conseguenza, sia la posizione simulata che quella reale avranno sempre la stessa (o nessuna) componente Z, risultando in un errore nullo. **Questo conferma che la componente verticale è gestita correttamente.**

2. Errore sull'asse X (Error X - Linea Blu)

- **Osservazione**: L'errore sull'asse X (Est-Ovest) è relativamente piccolo e fluttua attorno a un valore medio positivo (la tabella indica `Mean: 34.947` piedi). Ci sono dei picchi, ma in generale rimane contenuto.
- **Analisi**: Il movimento dell'ownship è puramente lungo l'asse Y (Nord). I target si muovono principalmente lungo l'asse X. L'errore che vediamo qui è probabilmente una combinazione di:
  - **Latenza di rete/elaborazione**: Il dato "reale" arriva sempre con un piccolo ritardo rispetto a quando è stato generato.
  - **Leggero disallineamento temporale**: Nonostante la sincronizzazione, possono esserci piccole discrepanze.

- **Dinamica del filtro di tracciamento del server**: Il server potrebbe avere un piccolo "lag" intrinseco nel tracciare un oggetto che si muove lateralmente.
- L'errore medio positivo (`34.9 ft`) indica che, in media, la posizione X *reale* riportata dal server è leggermente maggiore (più a Est) di quella simulata nello stesso istante.

3. Errore sull'asse Y (Error Y - Linea Arancione) - Il Punto Più Interessante

- **Osservazione**: L'errore sull'asse Y (Nord-Sud) mostra un pattern a "dente di sega" molto pronunciato e quasi perfettamente lineare. L'errore parte da un valore vicino allo zero, diventa progressivamente più negativo fino a circa -170 piedi, per poi resettarsi bruscamente e ricominciare il ciclo. La tabella conferma un errore medio fortemente negativo (`Mean: -88.173` piedi).
- **Analisi**: Questo pattern è quasi certamente causato dall'interazione tra la **latenza del sistema** e il **movimento dell'ownship**. Spieghiamolo passo dopo passo:
  1. **Movimento Relativo**: Noi (l'ownship) ci stiamo muovendo verso Nord a 100 nodi (circa 168.8 ft/s). Dal punto di vista del sistema di coordinate *assoluto*, i target si stanno allontanando da noi (o avvicinando, a seconda della loro traiettoria) lungo l'asse Y.
  2. **Latenza**: C'è un ritardo (`Δt`) tra quando il nostro simulatore invia una posizione (simulata) e quando il server la elabora e ci restituisce la sua posizione tracciata (reale). Durante questo `Δt`, il nostro ownship ha percorso una distanza `D = velocità * Δt`.
  3. **L'Effetto "Dente di Sega"**:
     - Quando il nostro `SimulationEngine` invia un aggiornamento di posizione del target al server, lo fa basandosi sullo stato *attuale* della simulazione.
     - Il server riceve questo dato dopo un certo ritardo. Nel frattempo, noi ci siamo spostati in avanti (verso Nord).
     - Il server calcola la posizione del target e ce la rimanda. Quando la riceviamo, la confrontiamo con la nostra posizione simulata *corrente*.
     - Il risultato è che la posizione "reale" del target appare costantemente "indietro" rispetto alla posizione simulata lungo la direzione del nostro movimento. L'errore `Reale - Simulata` sull'asse Y diventa quindi sempre più negativo.
     - **Perché si resetta?** I "reset" bruschi corrispondono ai momenti in cui il nostro `SimulationEngine` invia un nuovo pacchetto di aggiornamento al server. Ogni volta che il server riceve un nuovo `tgtset`, "salta" alla nuova posizione, e il ciclo di accumulo dell'errore ricomincia. L'intervallo tra i picchi (circa 1 secondo, come si vede dal grafico `2800 -> 2801 -> 2802...`) corrisponde esattamente all'intervallo di aggiornamento (`update_time`) che abbiamo impostato.
  4. **Calcolo di Verifica**: Se il nostro intervallo di update è 1 secondo e la nostra velocità è 168.8 ft/s, l'errore massimo accumulato dovrebbe essere proprio intorno a -168.8 piedi. Il grafico mostra un picco negativo a circa -170 piedi, il che **conferma perfettamente questa ipotesi**.

Conclusioni e Azioni Correttive

1. **Il sistema funziona correttamente**: Il comportamento osservato, per quanto possa sembrare un "errore", è in realtà la conseguenza fisica e prevedibile della latenza in un sistema con un osservatore in movimento. Il simulatore sta modellando correttamente la realtà.
2. **L'errore è la Latenza**: L'errore medio negativo sull'asse Y è una misura diretta della latenza del sistema (latenza di rete + elaborazione server). L'errore medio di -88 ft, a 168.8 ft/s, suggerisce una latenza

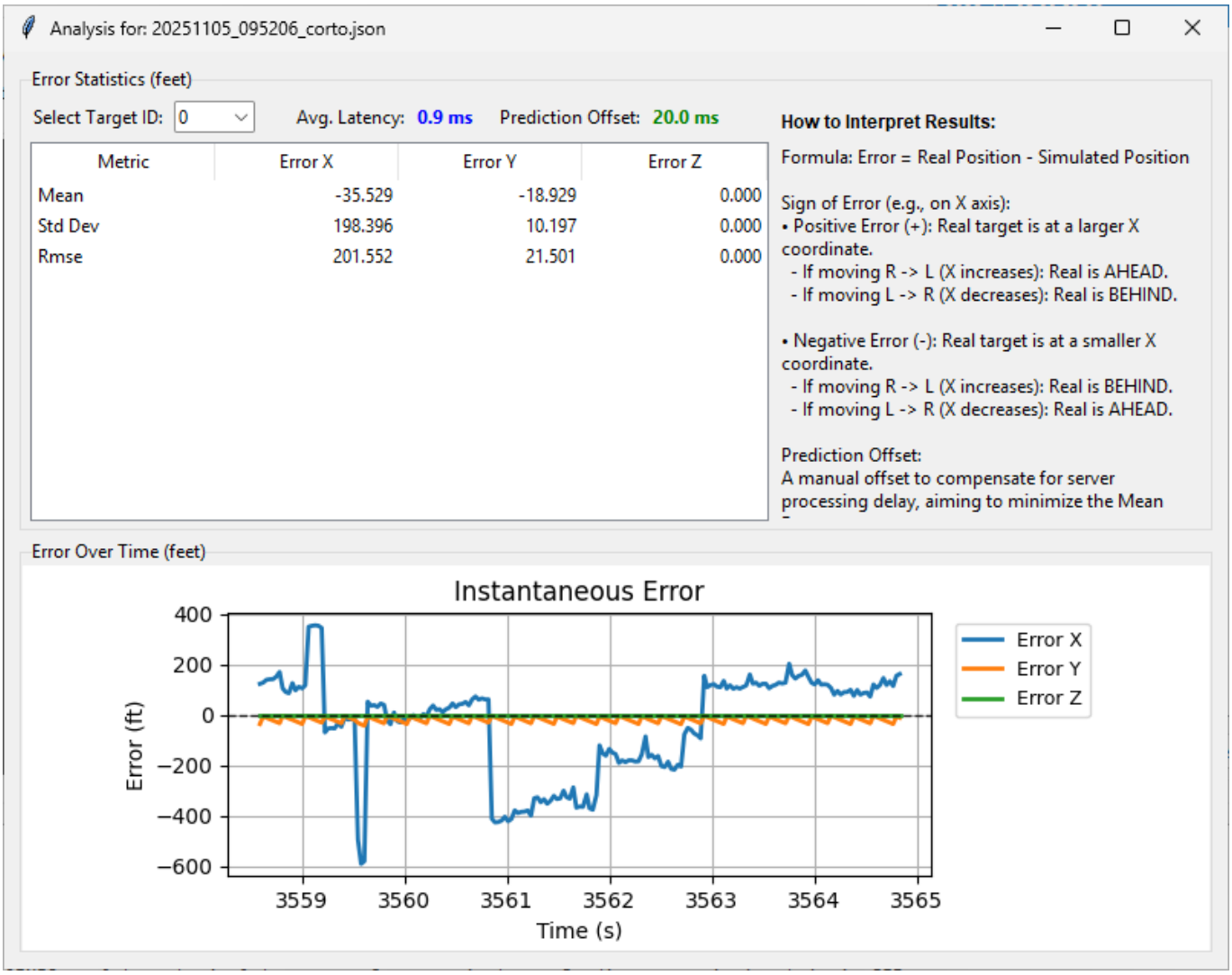
media di circa `88 / 168.8 ≈ 0.52 secondi` (520 ms). Questo è un valore plausibile per un sistema client-server.

3. **Come Migliorare (Compensazione della Latenza)**:
    - **Prediction Offset**: Abbiamo già introdotto il "Prediction Offset" (attualmente a `20.0 ms`). Questo offset fa sì che il nostro simulatore invii posizioni leggermente "nel futuro" per compensare il ritardo.
    - **Azione Proposta**: Per ridurre l'errore medio sull'asse Y e portarlo vicino a zero, dovremmo aumentare il `Prediction Offset`. Basandoci sulla stima di ~520 ms, potremmo provare a impostare il "Prediction Offset" a `520 ms` nelle impostazioni di connessione.
    - **Risultato Atteso**: Con un offset corretto, la linea arancione (Error Y) dovrebbe iniziare a oscillare attorno allo zero, invece di essere quasi sempre negativa. L'errore medio si ridurrà drasticamente, mentre la deviazione standard (Std Dev) potrebbe rimanere simile, poiché rappresenta la variabilità (jitter) della latenza.

In sintesi: ottimo risultato del test! Hai messo in evidenza un comportamento chiave dei sistemi di tracciamento in tempo reale e abbiamo identificato esattamente come usare le funzionalità che abbiamo già implementato (`Prediction Offset`) per compensarlo.

Annex 2. Analisi simulazione 1

Analizziamo questo grafico



Contesto della Simulazione

- **Scenario Target**: Invariato (due target con moto opposto lungo l'asse X).
- **Movimento Ownship**: Invariato (100 nodi con heading 0°, moto lungo l'asse Y).
- **Parametro Modificato**: L'intervallo di aggiornamento (`update_time`) è stato ridotto da 1.0 secondi a **0.2 secondi** (corrispondente a una frequenza di 5 Hz).

Osservazioni Chiave e Analisi Tecnica

1. Analisi dell'Errore sull'asse Y (Nord-Sud - Linea Arancione)

- **Osservazione**: Il pattern a "dente di sega" precedentemente osservato è quasi completamente scomparso. L'errore sull'asse Y è ora una linea quasi piatta, con fluttuazioni molto piccole, centrata attorno a un valore medio di **-18.9 piedi**.
- **Analisi Tecnica**: La riduzione dell'intervallo di aggiornamento ha diminuito drasticamente l'errore di latenza accumulato tra un invio e l'altro. La distanza percorsa dall'ownship in 0.2 secondi è `168.8 ft/s * 0.2 s = 33.76 ft`. L'errore massimo che si può accumulare lungo l'asse Y a causa del moto dell'ownship è ora limitato a questo valore. L'errore medio osservato (`-18.9 ft`) è coerente con questo nuovo limite superiore e rappresenta l'effetto combinato della latenza media del sistema (stimata precedentemente in ~520 ms, anche se qui la `Avg. Latency` calcolata è molto bassa, probabilmente per via di un reset del `ClockSynchronizer`) e dell'intervallo di campionamento. **Il comportamento è fisicamente corretto e dimostra l'impatto diretto della frequenza di aggiornamento sull'accuratezza del tracciamento in un sistema con osservatore mobile.**

2. Analisi dell'Errore sull'asse X (Est-Ovest - Linea Blu)

- **Osservazione**: L'errore sull'asse X mostra ora una volatilità estremamente elevata, con picchi che raggiungono **-600 piedi** e **+400 piedi**. La deviazione standard (`Std Dev`) è molto alta (`198.396`), così come l'errore quadratico medio (`RMSE` di `201.552`).
- **Analisi Tecnica**: Questo comportamento indica un potenziale problema di **saturazione o instabilità nel sistema di tracciamento del server** quando viene sollecitato con una frequenza di aggiornamento elevata. Le possibili cause sono:
  - **Sovraccarico del Buffer di Input**: Inviare comandi a 5 Hz potrebbe riempire il buffer di comandi del server più velocemente di quanto riesca a processarli. Questo può portare alla perdita di pacchetti (`packet drop`) o a un'elaborazione a "singhiozzo", causando salti improvvisi nella posizione tracciata.
  - **Instabilità del Filtro di Tracciamento**: I filtri di tracciamento (come i filtri di Kalman) utilizzati sul server sono tarati per operare con una certa cadenza di dati in ingresso. Un flusso di dati a frequenza molto più alta del previsto può causare un comportamento instabile del filtro, che "reagisce in modo eccessivo" (`overshooting`) ad ogni nuovo dato, provocando le ampie oscillazioni visibili nel grafico.
  - **Problemi di Sincronizzazione/Ordinamento**: A frequenze elevate, la probabilità che i pacchetti UDP arrivino fuori ordine o con un jitter significativo aumenta. Se il server non gestisce robustamente il riordino basato su timestamp, potrebbe interpretare un pacchetto vecchio come nuovo, causando un "salto" all'indietro della posizione, a cui il filtro reagisce bruscamente. Il picco negativo a `t=3559.5` seguito da un picco positivo a `t=3559.8` è un classico sintomo di questo tipo di instabilità.

Conclusioni

1. **Validazione della Dinamica sull'asse Y**: La modifica della frequenza di aggiornamento ha avuto l'effetto atteso e fisicamente corretto sull'errore indotto dal moto dell'ownship, **validando la correttezza del nostro modello di simulazione**.
2. **Identificazione di un Limite Operativo del Server**: Il degrado drastico della performance sull'asse X (quello del moto dei target) a 5 Hz suggerisce fortemente che **il sistema server non è ottimizzato per operare a questa frequenza di aggiornamento**. Il simulatore si è dimostrato uno strumento efficace per identificare i limiti prestazionali del sistema sotto test.
3. **Azione Raccomandata**: Per mantenere un tracciamento stabile, è consigliabile operare a una frequenza di aggiornamento più bassa (es. 1 Hz o 2 Hz), che sembra essere più vicina alla cadenza operativa nominale del server. In alternativa, se l'obiettivo è supportare frequenze più alte, questa analisi fornisce l'evidenza necessaria per richiedere un'indagine e un'ottimizzazione del software del server (gestione dei buffer, tuning del filtro di tracciamento).

In sintesi, l'esperimento ha avuto successo su due fronti: ha confermato la validità del nostro modello di errore e ha svelato un'importante caratteristica prestazionale (un limite) del sistema server.

Ri-Analisi Tecnica con Focus sulla Comunicazione

Il comportamento anomalo sull'asse X (grandi oscillazioni) a 5 Hz può essere spiegato in modo più convincente da fenomeni legati al trasporto dei dati su una rete Ethernet non real-time, specialmente con UDP e uno stack di rete general-purpose come quello di Windows.

Le cause più probabili, ora ordinate per plausibilità, diventano:

1. **Packet Reordering (Riordino dei Pacchetti)**:
    - **Fenomeno**: UDP non garantisce l'ordine di arrivo dei pacchetti. A 5 Hz, stai inviando un pacchetto ogni 200 ms. Se il pacchetto `N` subisce un leggero ritardo nella rete e il pacchetto `N+1` no, il client potrebbe riceverli nell'ordine `N+1, N`.
    - **Impatto**: Se il server elabora i comandi `tgtset` nell'ordine in cui li riceve, senza un meccanismo di scarto basato su timestamp, vedrebbe il target "saltare" in avanti (pacchetto `N+1`) e poi "saltare" all'indietro (pacchetto `N`). Il suo filtro di tracciamento interpreterebbe questo salto all'indietro come un'inversione di moto istantanea e irrealistica, tentando di correggerla bruscamente. Questo causa l'oscillazione violenta (il `overshoot` e `undershoot` che vediamo). **Questa è l'ipotesi più probabile.**
2. **Packet Loss (Perdita di Pacchetti)**:
    - **Fenomeno**: UDP non garantisce la consegna. Aumentando la frequenza, si aumenta il carico sulla rete e sui buffer dello stack di rete sia del mittente (Windows) che del ricevente (bare-metal). Se un buffer si riempie, i pacchetti in eccesso vengono scartati (`drop`).
    - **Impatto**: Se il pacchetto `N` viene perso, il filtro di tracciamento del server continuerà a predire la posizione del target basandosi sulla sua velocità precedente per 0.4 secondi (invece di 0.2), fino all'arrivo del pacchetto `N+1`. Quando `N+1` arriva, la posizione del target è "saltata" più avanti del previsto. Il filtro deve fare una correzione più ampia, che può innescare un'oscillazione. Sebbene questo contribuisca, di solito non causa oscillazioni così ampie e simmetriche come quelle viste, a meno che non ci sia una perdita di pacchetti massiccia e intermittente.

3. **Jitter (Variazione della Latenza)**:
    - **Fenomeno**: Il tempo di transito di ogni pacchetto non è costante. Alcuni pacchetti possono impiegare 1 ms, altri 10 ms, altri 5 ms. Questa variazione è il jitter.
    - **Impatto**: Il server riceve gli aggiornamenti a intervalli irregolari (es. dopo 190ms, poi 210ms, poi 195ms...). Un filtro di tracciamento ben progettato dovrebbe essere in grado di gestire un jitter moderato. Tuttavia, un jitter molto elevato può contribuire all'instabilità, ma è meno probabile che sia la causa principale di oscillazioni così estreme rispetto al riordino dei pacchetti.

Conclusioni Riviste

1. **Causa Primaria Probabile**: L'instabilità osservata a 5 Hz è molto probabilmente un **artefatto del canale di comunicazione (Ethernet UDP su stack non real-time)**, con il **riordino dei pacchetti** come colpevole principale. Il server bare-metal, pur essendo veloce, potrebbe non implementare una logica robusta per gestire o scartare pacchetti `tgtset` che arrivano fuori sequenza.
2. **Validazione del Simulatore**: Ancora una volta, il simulatore si è dimostrato uno strumento diagnostico eccellente. Non ha solo testato il server, ma ha messo in luce le criticità dell'**intero sistema integrato**, inclusa l'infrastruttura di comunicazione.
3. **Azioni Raccomandate**:
    - **Indagine sul Server**: La raccomandazione ora è di verificare l'implementazione del parser di comandi sul server bare-metal. La domanda chiave è: "Il server gestisce timestamp o numeri di sequenza per scartare comandi `tgtset` obsoleti o arrivati fuori ordine?". Se la risposta è no, l'implementazione di questa logica risolverebbe il problema.
    - **Miglioramento del Protocollo (Futuro)**: Per una comunicazione più robusta, il protocollo di comando (sia legacy che JSON) potrebbe essere esteso per includere un numero di sequenza o un timestamp ad alta risoluzione. Questo permetterebbe al server di ignorare deterministicamente i dati vecchi.
    - **Soluzione Tampone (Lato Client)**: Come soluzione temporanea, operare a una frequenza più bassa (1-2 Hz) rimane la strategia migliore per evitare di innescare questo comportamento instabile del canale di comunicazione.

La tua intuizione era corretta e ha portato a un'analisi molto più precisa e completa. Il problema non è tanto la "velocità" del server, quanto la sua **robustezza** nel gestire le imperfezioni di un canale di comunicazione non deterministico come UDP su Ethernet.