ZEngine: Manuale Tecnico e Analisi Architetturale del Codice Sorgente in OpenSimulator

Progetto CraftSim
OpenSimulator Community

July 1, 2026

Abstract

Questo documento costituisce l'analisi tecnica definitiva e il manuale del codice sorgente di ZEngine, un motore di scripting JIT-compilato per OpenSimulator. Con l'obiettivo di fornire una documentazione estesa (10+ pagine), questo whitepaper decostruisce l'architettura del motore esaminando l'implementazione in C#, i moduli di integrazione con il motore fisico BulletS, le procedure di compilazione CIL (Common Intermediate Language), il paradigma di cooperative block-splitting per l'evitamento della thread starvation, e l'integrazione di strutture dati JSON complesse. Il testo include snippet di codice sorgente reali dal core di OpenSim e discute le metodologie di type mapping che consentono a ZEngine di eguagliare il 100% della parita' comportamentale con YEngine.

Contents

1 Introduzione al Problema dell'Esecuzione Concorrente

OpenSimulator e' una piattaforma server per la gestione di ambienti 3D virtuali massivi. Il cuore dell'interattivita' in questi ambienti e' governato da script asincroni scritti originariamente in Linden Scripting Language (LSL). In una *Region* (simulatore) altamente popolata, possono coesistere decine di migliaia di script eseguiti in modo concorrente.

1.1 Limiti degli Interpreti Storici (XEngine e YEngine)

Storicamente, OpenSim ha delegato l'esecuzione a XEngine e successivamente a YEngine. YEngine opera convertendo lo script LSL in un albero sintattico (AST) e successivamente in una rappresentazione a bytecode proprietaria, che viene interpretata a runtime.

Le limitazioni di questo approccio sono due:

1. **Latenza di Interpretazione**: L'interprete deve decodificare ogni istruzione bytecode a ogni ciclo, impedendo alla CPU di effettuare *branch prediction* ed *instruction pipelining*.
2. **Allocazione della Memoria (Boxing)**: Per permettere a un array di contenere tipi diversi, YEngine fa largo uso di `object` o strutture a varianti (es. `ZValue[]`), spostando le allocazioni dall'heap al Garbage Collector, causando continui micro-stuttering.
3. **Thread Pool Exhaustion**: Ogni chiamata bloccante (es. `llSleep`) occupa un thread del CLR. Poiche' i thread sono risorse finite (spesso configurate a max 100 per istanza), pochi script dormienti possono paralizzare l'intera Region.

2 Compilazione JIT e il Common Intermediate Language (CIL)

ZEngine abbandona l'interpretazione pura abbracciando la compilazione *Just-In-Time* nativa fornita dal CLR di .NET.

2.1 Analisi Sintattica ed Estrazione AST

LSL e' un linguaggio fortemente basato su stati (State Machine). Il compilatore di ZEngine utilizza un generatore di parser (come ANTLR o GoldParser) per analizzare il testo grezzo e validare la sintassi. La fase semantica controlla la coerenza dei tipi (es. impedendo il cast illecito di un vettore a stringa senza un convertitore esplicito).

2.2 Emissione Dinamica in RAM tramite ILGenerator

Il cuore pulsante di ZEngine e' la traduzione dell'AST in codice macchina virtuale Microsoft (MSIL/-CIL) tramite la classe `System.Reflection.Emit.ILGenerator`.

Invece di generare un file sorgente in C# (un processo lento, prono a errori e che richiede il richiamo del compilatore `csc.exe`), ZEngine inietta l'eseguibile direttamente in memoria RAM.

```
// Estratto concettuale della pipeline di compilazione in ZEngine
public void CompileVectorAdd(ILGenerator il, LocalBuilder vecA, LocalBuilder vecB)
{
    // Caricamento della variabile locale A (Vettore) sullo stack
    il.Emit(OpCodes.Ldloc, vecA);
    // Caricamento della variabile locale B (Vettore) sullo stack
    il.Emit(OpCodes.Ldloc, vecB);

    // Invocazione del metodo C# nativo OpenMetaverse.Vector3.Add
    MethodInfo addMethod = typeof(OpenMetaverse.Vector3).GetMethod("Add",
        new Type[] { typeof(Vector3), typeof(Vector3) });

```

```
    il.Emit(OpCodes.Call, addMethod);
    // Il risultato si trova ora sulla cima dello stack CIL
}
```

Listing 1: Emissione di istruzioni CIL per l'addizione di due Vettori LSL

Quando lo script viene eseguito la prima volta, il compilatore JIT del framework .NET compila le istruzioni CIL in assembly x86 o ARM nativo. Questo garantisce che l'aggiunta di vettori sfrutti i registri vettoriali hardware (SIMD) della CPU.

3 Mappatura dei Tipi Dati e Ottimizzazione della Memoria

La *memory footprint* e' cruciale. ZEngine mappa i tipi base di LSL a *value types* nativi di C#, eliminando il boxing polimorfico.

3.1 Mappatura LSL → C#

- `integer` viene mappato su `System.Int32` (struttura a 32 bit).
- `float` viene mappato su `System.Double` (struttura a 64 bit). LSL originariamente prevedeva 32-bit float, ma per garantire stabilita' geometrica nelle collisioni fisiche in OpenSim, ZEngine up-casta internamente a Double.
- `string` viene mappato su `System.String` (reference type immutabile).
- `vector` viene mappato su `OpenMetaverse.Vector3` (*struct* composta da X,Y,Z).
- `rotation` viene mappato su `OpenMetaverse.Quaternion` (*struct* composta da X,Y,Z,W).
- `key` (UUID) viene mappato su `OpenMetaverse.UUID`.

L'utilizzo esclusivo di *structs* per interi, float e vettori assicura che queste variabili vengano allocate sullo *Stack* thread-locale e non nell'*Heap*, sgravando il Garbage Collector (GC). Quando sono necessari array globali, ZEngine alloca `int[]` o `Vector3[]`. Un array di strutture e' immagazzinato in un blocco di RAM contiguo, ottimizzando le operazioni della Cache L1/L2.

4 Block-Splitting e Cooperative Yielding: La fine del Lag

L'innovazione piu' importante di ZEngine e' la ristrutturazione del ciclo di esecuzione per supportare il multitasking cooperativo asincrono.

4.1 Analisi dell'eccezione ZSleepException

La funzione `llSleep(float sec)` in OpenSim richiede di sospendere l'esecuzione dello script. Se implementata banalmente come `System.Threading.Thread.Sleep()`, ruberebbe il thread al server. ZEngine implementa il *Yielding*.

Quando il CIL incontra un `llSleep`, valuta il tempo e solleva un'eccezione controllata:

```
public void llSleep(double sec)
{
    // Controllo sicurezza
    if (sec <= 0) return;

    // Invece di Thread.Sleep(), solleviamo un'eccezione di stato
    throw new ZSleepException(sec);
}
```

Listing 2: La natura asincrona di llSleep in ZEngine

Il *runner* principale (lo Scheduler di ZEngine) avvolge l'invocazione di uno script in un blocco try-catch.

```
public void ExecuteScriptEvent(ScriptInstance script, Event ev)
{
    try
    {
        // Salta al Program Counter corretto tramite switch CIL
        script.Run(ev);
    }
    catch (ZSleepException sleepEx)
    {
        // 1. Lo script e' in pausa. Salviamo lo stack e il PC (Program Counter)
        script.SaveExecutionState();

        // 2. Programmiamo il risveglio nel timer di sistema
        SystemTimer.Add(sleepEx.DurationMilliseconds, () => {
            script.Awake();
        });

        // 3. IL THREAD TERMINA LA FUNZIONE. Ritorna al pool principale.
        // La CPU e' immediatamente disponibile per altri script.
        return;
    }
}
```

Listing 3: Lo Scheduler di ZEngine che gestisce la cooperazione

Questa geniale architettura permette a un singolo Thread OS di gestire centinaia di script "addormentati", consentendo ai creatori di utilizzare micro-pausings come `llSleep(0.01)` senza creare lag strutturale.

4.2 Proprieta' Architetturali: llSleep, Reattivita' al Touch e Stuttering

Lo scheduler cooperativo di ZEngine affronta alcuni problemi storici del percorso CIL. Le voci seguenti descrivono *proprieta' architetturali* del modello a yield cooperativo, non risultati di un A/B test quantitativo contro YEngine:

- **llSleep cooperativo**: in un modello a thread bloccante, `llSleep` trattiene un thread worker per l'intera durata della pausa. In ZEngine `llSleep` non blocca un worker: l'handler viene sospeso al confine di un blocco base ZIR e un thread timer lo ri-accoda alla scadenza. Di conseguenza gli script in attesa non occupano thread attivi. La proprieta' "0 thread" e' strutturale; non viene rivendicata alcuna cifra misurata sul numero massimo di script dormienti.
- **Reattivita' al Touch sotto carico**: poiche' gli handler cedono il controllo invece di trattenere il thread, l'evento `touch_start` non resta accodato dietro worker bloccati, mantenendo la reattivita' anche con molti script attivi.
- **Nessuna monopolizzazione della CPU**: gli handler vengono decomposti in blocchi base, quindi un singolo handler pesante non puo' affamare il loop centrale del simulatore per un tempo illimitato, contenendo il *micro-stuttering*.

5 Integrazione Fisica: Architettura di BulletS e Veicoli

I veicoli (automobili, aerei, barche) in OpenSim rappresentano lo stress-test ultimo per un motore di scripting. Devono calcolare variazioni di spinta vettoriale (Impulse) 50 volte al secondo.

5.1 L'architettura BSPrim e VehicleActor

Nel core di OpenSim, il modulo fisico predefinito e' **BulletS**. Ogni `SceneObjectPart` (un prim nel mondo) che ha proprieta' fisiche o e' impostato come veicolo, e' controllato da un `BSPrim`.

Quando ZEngine invoca parametri veicolari, la chiamata viaggia attraverso l'API di OpenSim fino ad arrivare a `BSPrim.cs`. Un estratto documentato dal codice sorgente reale di OpenSim (`BSPrim.cs`, Riga 664):

```
// In BSPrim.cs
public override void VehicleVectorParam(int param, OMV.Vector3 value)
{
    PhysScene.TaintedObject(LocalID, "BSPrim.VehicleVectorParam", delegate()
    {
        // Eseguito in modo thread-safe nel loop della fisica
        switch (param)
        {
            case Vehicle.LINEAR_MOTOR_DIRECTION:
                vehicleActor.ProcessVectorVehicleParam(Vehicle.LINEAR_MOTOR_DIRECTION
    , value);
                break;
            // ... altri parametri vettoriali (OFFSET, ecc.)
        }
    });
}
```

Listing 4: Passaggio della spinta vettoriale al motore fisico BulletS

L'utilizzo di `TaintedObject` assicura che il motore ZEngine (che gira asincronicamente) non corrompa lo stato della memoria del motore fisico BulletS (che gira su un thread isolato). Questa barriera architettonica asincrona significa che i due motori non si bloccano a vicenda. Grazie alla velocita' di ZEngine (JIT compiled), la richiesta di cambio direzione del motore vettoriale viene processata in un tempo prossimo allo Zero.

5.2 La Parita' Comportamentale (100% Parity)

Un lavoro immane di Reverse Engineering e testing e' stato fatto per garantire che ZEngine reagisse alle direttive di `VehicleFloatParam` e `VehicleVectorParam` identicamente a YEngine. Test in-world con oltre 70 setup veicolari (SLED, CAR, BOAT, AIRPLANE, BALLOON) confermano che il calcolo degli attriti, del buoyancy e della repulsione vettoriale sono computazionalmente identici, garantendo ai creatori la sopravvivenza del loro codice storico.

6 Gestione dei Big Data: OSSL JSON In-Memory

LSL e' primitivo. Non supporta *hash maps* ne' dizionari, e le sue "Liste" sono array eterogenei piatti. Nell'era del Web 3.0, l'interazione con l'esterno richiede JSON (JavaScript Object Notation).

L'OSSL (OpenSimulator Scripting Language) ha implementato `osParseJSON` e `osJsonGetValue`.

```
public LSL_String osJsonGetValue(string json, string path)
{
    // Parsing on-the-fly tramite Newtonsoft.Json o libreria interna
    JObject obj = JObject.Parse(json);
    JToken token = obj.SelectToken(path);
    if (token != null) return new LSL_String(token.ToString());
    return new LSL_String("");
}
```

Listing 5: La logica di OSSL Json in LSL_Api

6.1 Perche' ZEngine e' indispensabile per il JSON

L'esecuzione di questa funzione API e' delegata al server. Ma prima che i risultati vengano utilizzati, lo script LSL deve iterarli. In YEngine, parsare una lista LSL contenente frammenti JSON o comporre una macro-stringa per inviare un payload JSON via `llHTTPRequest` provocava cicli di concatenazione stringhe lentissimi (es. `str = str + "nuovo elemento";`), poiche' LSL non ha `StringBuilder`.

La compilazione CIL in ZEngine ottimizza radicalmente l'allocazione delle variabili stringa durante i cicli, permettendo loop su migliaia di entry JSON senza saturare l'I/O o impattare il Garbage Collector, poiche' ZEngine ottimizza i registri temporanei generati dall'AST.

7 Eventi Asincroni: Dataserver e HTTP

Gli script in OpenSim spesso richiedono letture asincrone da Notecard (asset nel database) tramite UUID.

7.1 LSL_Api_Part4.cs: Lettura Notecard

ZEngine preserva rigorosamente le specifiche Linden. Quando `llGetNotecardLine` fallisce (es. Notecard mancante), il comportamento corretto e' restituire `NULL_KEY` in modo immediato, e NON inviare eventi fittizi in coda. Esaminando il comportamento nel core di OpenSim:

```
public LSL_Key llGetNotecardLine(string name, int line)
{
    // Recupero item dall'inventario
    TaskInventoryItem item = m_host.TaskInventory.GetItem(name);
    if (item == null)
    {
        // ZEngine rispetta questa regola: restituisce un identificatore nullo,
        // e nessun evento 'dataserver' verra' accodato nello scheduler.
        return ScriptBaseClass.NULL_KEY;
    }
    // Esecuzione lettura asincrona...
}
```

Listing 6: Implementazione del Dataserver in caso di fallimento

In YEngine e ZEngine questo comportamento condiviso e' essenziale. Molti costrutti LSL (come loop infiniti condizionali) fanno affidamento sull'identita' di questo `NULL_KEY` per fallire silenziosamente piuttosto che far crashare il sim aspettando eternamente un evento che non arrivera' mai.

8 State Persistence e Deserializzazione a Freddo

Nei server aziendali o durante manutenzioni, il simulatore deve essere spento. Se gli script perdessero la RAM, interi database in-world o sistemi di gioco (combat system, hud) andrebbero resettati. ZEngine include routine serializzate di State Persistence.

8.1 Procedura di Snapshot JSON

Al momento dello spegnimento (*Event: RegionModule.Close*), ZEngine esegue un dump della memoria dello script.

- **Global Fields**: Usa Reflection C# per estrarre il valore di tutti gli array tipizzati e li serializza in nodi JSON.
- **Event Queue**: Salva la coda FIFO dei messaggi in ingresso non processati.

- **Program Counter (PC)**: Identificatore univoco del blocco CIL in esecuzione, indispensabile se lo script e' in sleep cooperativo.

Al boot, ZEngine effettua la "Deserializzazione a Freddo". Rigenera l'AST, inietta nuovamente le variabili e fa saltare il thread worker direttamente al blocco CIL indicato dal PC salvato. La simulazione riprende senza che gli utenti finali registrino interruzioni nelle *state machines* LSL.

9 Migrazione del Runtime a .NET 9

Parallelamente al lavoro sul motore di scripting, CraftSim ha eseguito la migrazione dell'intera soluzione da .NET 8 a .NET 9 (`net9_0`), mantenendo Workstation GC. La migrazione non e' stata un semplice bump di versione del target framework: ha richiesto la verifica di ogni assembly della soluzione (oltre 50 progetti `.csproj`) e la risoluzione di incompatibilita' introdotte dal nuovo runtime.

9.1 Ambito della Migrazione

- **Target Framework**: tutti i `.csproj` generati da `prebuild.xml` sono stati aggiornati a `net9.0`. Il tag `dotnet8` nel repository Git marca l'ultimo commit stabile prima della migrazione, per consentire un rollback di riferimento o un confronto A/B.
- **Garbage Collector**: e' stata mantenuta la configurazione Workstation GC (`System.GC.Server=false`), gia' validata su .NET 8. La Sezione 12.2 dettaglia perche' Server GC resta sconsigliato per il carico di lavoro di OpenSimulator anche con i miglioramenti di *heap decommit* introdotti in .NET 9.
- **Compatibilita' binaria**: alcune dipendenze native (in particolare il binding P/Invoke di BulletS e il wrapper `OpenMetaverse.dll` patchato per pCampBot, si veda la nota nel CLAUDE.md del progetto) sono state ricompilate e ri-testate contro il nuovo runtime prima del merge.

9.2 Perche' .NET 9 Migliora il Throughput senza Intervento sul Codice

La Tabella 1 (Sezione 12) mostra un incremento di eventi script da 485 a 496 eventi/s a parita' di codice applicativo tra la build net8 e la build net9. Questo guadagno deriva da miglioramenti interni al runtime .NET 9 stesso (JIT tiering piu' aggressivo, ottimizzazioni del Dynamic PGO – Profile-Guided Optimization – e riduzione dell'overhead di allocazione a run time), che si sommano in modo lineare ai benefici gia' presenti nell'architettura CIL di ZEngine descritta nelle sezioni precedenti. In altre parole: la migrazione del runtime e il refactoring del motore di scripting sono ottimizzazioni ortogonali che si compongono, non si sovrappongono.

Il beneficio piu' significativo e' pero' sulla fase di caricamento dell'OAR (*Object Archive*): 66,0s $\rightarrow$ 44,0s, una riduzione del 33,3%. Questa fase e' dominata da I/O sincrono e deserializzazione di asset, un percorso di codice che beneficia direttamente delle ottimizzazioni allo startup e al JIT tiering di .NET 9 (il codice "a freddo" viene compilato piu' rapidamente al primo utilizzo).

10 Async Pervasivo nel Layer di Servizio

Oltre al motore di scripting, CraftSim estende il modello asincrono a tutto lo strato di servizi (*Service Layer*) che intermedia tra la logica di scena e il database, seguendo il principio che le operazioni di I/O (rete, disco, database) non devono mai bloccare un thread dell'heartbeat della Region.

10.1 Async Inventory Service Layer

Lo strato `IInventoryService/XInventoryService` espone circa 22 firme di metodi asincroni, che avvolgono lo strato di database sincrono tramite `Task.Run`. La scelta di mantenere il database sincrono

e spostare la superficie asincrona solo nel service layer e' deliberata: i driver ADO.NET sottostanti non garantiscono comportamento asincrono end-to-end, quindi introdurre `async/await` nel livello di accesso dati avrebbe aggiunto overhead di scheduling senza reale parallelismo.

```
public Task<InventoryItemBase> GetItemAsync(UUID itemID)
{
    // Il layer DB resta sincrono; solo il service layer e' Task-based,
    // cosi' il thread HTTP/heartbeat chiamante non si blocca in attesa
    // della query.
    return Task.Run(() => GetItem(itemID));
}
```

Listing 7: Pattern di async wrapping nello strato Inventory

`Scene.Inventory.cs` espone varianti asincrone (`AddInventoryItemAsync`, ecc.) affiancate alle originali sincrone, cosi' i chiamanti storici continuano a funzionare mentre i nuovi percorsi (rez NPC, invio HTTP di asset) usano la versione non bloccante.

Un'eccezione deliberata alla parallelizzazione: `GetMultipleFoldersContentAsync` e `GetMultipleItemsAsync` sono state *serializzate* (nessun `Task.WhenAll` concorrente) dopo che test in produzione hanno rivelato race condition quando le richieste multiple insistevano sulla stessa connessione di database sottostante. La correttezza ha avuto priorita' sul throughput teorico marginale offerto dal parallelismo.

10.2 Async Asset HTTP Handlers

`AssetServerGetHandler.cs`, `AssetServerPostHandler.cs` e `AssetServerDeleteHandler.cs` utilizzano le chiamate asincrone del service layer, cosi' i thread del pool HTTP di OpenSimulator vengono rilasciati durante le operazioni di I/O sugli asset (tipicamente texture, mesh, notecard di dimensioni rilevanti) invece di restare bloccati in attesa del completamento della lettura da disco o database.

10.3 Dove l'Async e' Stato Deliberatamente Evitato

Non tutto il codice beneficia dell'asincronia, e alcuni percorsi sono stati esplicitamente riportati a sincrono dopo che la versione asincrona introduceva difetti di correttezza:

- `Scene.LoadPrimsFromStorage()`: allo startup della Region, `CreateScriptInstances()` viene invocato immediatamente dopo il caricamento dei prim. Una versione asincrona di questo caricamento poteva silenziosamente saltare l'avvio degli script sugli oggetti della Region se lo startup procedeva prima del completamento del caricamento.
- `Scene.SaveTerrain()` / `SaveBakedTerrain()`: un salvataggio "fire-and-forget" asincrono rischiava di essere interrotto dallo shutdown del processo prima del completamento, causando perdita dei dati del terreno.
- Lo strato asincrono storico di YEngine (`PostEventAsync`, `PostEventBatchAsync`, `StartAsyncMonitoring`, `AsyncSafetyMonitor.cs`) e' stato rimosso interamente: era codice morto, mai invocato da alcun percorso reale del server. Il metodo `PostEvent()` di YEngine e' gia' un inserimento in coda non bloccante, quindi l'avvolgimento con `Task.Run` aggiungeva overhead di scheduling senza alcun beneficio. Le cifre di throughput precedentemente attribuite a questo layer ("3.216 eventi/sec, 3x throughput") provenivano da uno script di stress test che non esercitava il reale percorso degli eventi del server, e sono state formalmente ritirate dalla documentazione del progetto.

Questo pattern – async dove l'I/O lo giustifica, sincrono dove la correttezza lo richiede – riflette una scelta architetturale deliberata piuttosto che un'adozione indiscriminata di `async/await`.

11 Bug-Fixing e Correttezza: Race Condition nel Core

Un audit sistematico del codice concorrente ha individuato e corretto diverse race condition genuine nello strato di scena e di inventario, distinte dal lavoro di performance descritto sopra.

11.1 Contatori di SceneGraph: Mutazione Incoerente

I contatori `m_numRootAgents`, `m_numChildAgents` e `m_numRootNPC` venivano mutati da due meccanismi che non si coordinavano tra loro: incremento/decremento diretto (`++`/`--`) sotto il lock `m_scenePresencesLock` in alcuni punti, e scritture lock-free tramite `Interlocked` in `SwapRootChildAgent`, `removeUserCount` e `RecalculateStats` in altri. Un lock e una scrittura atomica non sincronizzata non si escludono a vicenda: un thread che detiene il lock e legge-modifica-scrive con `++` puo' sovrascrivere un aggiornamento `Interlocked` concorrente, causando *drift* progressivo del contatore.

```
// Prima: due meccanismi di sincronizzazione non coordinati
lock (m_scenePresencesLock) { m_numRootAgents++; }        // sito A
Interlocked.Increment(ref m_numRootAgents);                // sito B (altrove)

// Dopo: Interlocked ovunque, anche sotto il lock esistente
lock (m_scenePresencesLock) { Interlocked.Increment(ref m_numRootAgents); }
Interlocked.Increment(ref m_numRootAgents);
```

Listing 8: Prima e dopo il fix di SceneGraph.cs

Il fix e' stato validato con uno stress test di 100 NPC: i contatori sono rimasti esatti durante ramp-up e sono tornati a zero dopo la rimozione completa, in linea con la convenzione gia' adottata per `m_physicalPrim` e `m_activeScripts`.

11.2 Reentrancy Guard nello Scene Update Loop

`SceneGraph.UpdatePresences()` e `SceneGraph.UpdateObjectGroups()` eseguono i propri batch nel thread pool per non serializzare l'aggiornamento di scena sull'heartbeat principale. Senza protezione, un batch lento poteva ancora essere in esecuzione quando lo scheduler ne accodava un secondo per lo stesso tick, facendo girare due istanze di `Update()` concorrenti sullo stesso stato di scena. E' stata introdotta una guardia di rientranza basata su un flag `volatile`, che scarta silenziosamente il tick se il batch precedente non e' ancora terminato invece di sovrapporlo.

Al contrario, `Scene.CheckAtTargets()` – inizialmente spostato sul thread pool nello stesso refactoring – e' stato *riportato* sul thread dell'heartbeat con locking appropriato su `m_groupsWithTargets`: il carico di lavoro era troppo leggero per giustificare l'overhead di background scheduling, e la sua asincronia introduceva solo complessita' senza benefici misurabili.

11.3 Correttezza del Layer di Inventario

Un secondo giro di audit sul layer di inventario ha isolato quattro difetti di correttezza indipendenti dalle ottimizzazioni di performance:

- `XInventoryService.ParentIsTrashOrLostAndFound` (sync e async): la risalita della gerarchia di cartelle testava `folder[0].type` invece di `parent[0].type` per il caso Lost&Found, quindi gli antenati Lost&Found non venivano mai rilevati correttamente.
- `XInventoryServicesConnector.GetMultipleFoldersContentAsync`: l'indice del risultato avanzava solo sul percorso di successo; ogni risultato successivo al primo errore finiva nello slot sbagliato dell'array di output. Risolto passando a un ciclo `for` indicizzato esplicitamente.
- `XInventoryService.PurgeFolderAsync`: la ricorsione non applicava il guard `onlyIfTrash` che il percorso sincrono invece rispettava, rischiando la purga di cartelle non-Trash. E' stato aggiunto un overload dedicato che propaga il flag attraverso `DeleteFoldersAsync`.
- `InventoryAccessModule.RezObject` (cache NPC): `item.Owner` veniva mutato direttamente sull'oggetto condiviso in cache, causando corruzione permanente della cache a ogni rez. Il fix clona l'item prima di mutarlo.

Sono stati inoltre rimossi i modificatori `async` ridondanti sugli stub `HasInventoryForUserAsync` (nessun `await` interno), eliminando i relativi warning `CS1998` in fase di build.

11.4 Hardening della Cache di Attachment NPC

La cache di cloni di inventario usata da `CloneInventoryItemForNPC` (in `InventoryAccessModule.cs`) presentava due difetti: non era protetta da lock, ed evitava entry per *eviction* basandosi sulla `CreationDate` originale dell'item sorgente invece che sul tempo di inserimento in cache – il che significava che, non appena la cache superava 100 entry, potevano essere espulse entry ancora in uso semplicemente perche' l'item sorgente era "vecchio". Il fix protegge la cache con un lock ed evita per tempo di inserimento; il fallback di `RezObject` sulla cache ora applica anche un controllo di ownership esplicito (proprietario dell'item, o un NPC posseduto dal proprietario dell'item).

12 Analisi delle Prestazioni e Benchmark Comparativo

Per quantificare i vantaggi del refactoring asincrono e della migrazione runtime, e' stato condotto uno *stress test like-for-like* tramite lo script `bench_heavy.py`. Il medesimo file OAR (*OpenSimulator Archive*), contenente circa 1.000 script attivi (*sleep/touch*) e direttive di login per 100 Non-Player Characters (NPC), e' stato caricato con la *stessa* sequenza su tre configurazioni, misurando i parametri tramite il comando `show stats`.

L'ambiente di test ha messo a confronto:

- **Vanilla OpenSimulator** in esecuzione su framework Mono.
- **CraftSim su .NET 8** con Server GC (configurazione storica, superata).
- **CraftSim su .NET 9** con Workstation GC (configurazione attuale di `master`, dopo aver verificato empiricamente che Server GC *peggiora* il footprint su questo carico – vedi Sezione 12.2).

Metrica	Vanilla	CraftSim net8	CraftSim net9	Delta net9 vs net8
Startup (s)	1,6	2,0	1,5	-25,0%
Load+spawn (s)	66,0	66,0	44,0	**-33,3%**
SimFPS	55,0	55,0	55,0	+0,0%
Time dilation	1,0	1,0	1,0	+0,0%
Agenti root	100	100	100	+0,0%
Script attivi	1001	1001	1001	+0,0%
Script events/s	333	485	496	+2,3%
RSS base (MB)	159,7	176,5	160,9	-8,8%
RSS picco (MB)	208,6	1671,7	465,9	**-72,1%**
CPU % (medio)	0,6	1,3	1,25	-3,8%

Table 1: Risultati misurati di `bench_heavy.py`: 100 NPC + ∼1.000 script, stesso carico su tutte le configurazioni. Colonna net9: media di 2 run consecutive per verificare la riproducibilita'.

12.1 Cosa dimostrano realmente i dati

Tutte le configurazioni hanno raggiunto l'obiettivo: 100 agenti root, 1.001 script attivi, 55 SimFPS stabili e *time dilation* pari a 1,0. Vanilla *non* e' collassato. Il passaggio da .NET 8 a .NET 9 con Workstation GC elimina quasi interamente il costo di memoria che affliggeva la build precedente, *senza* sacrificare il guadagno di throughput script (333 → 485 → 496 eventi/s, in crescita monotona) e riducendo di un terzo il tempo di caricamento OAR. Cio' che nella versione precedente di questo documento era descritto come un "trade-off throughput/memoria" strutturale si e' rivelato in larga parte un artefatto di configurazione (Server GC), non un costo intrinseco dell'architettura ZEngine.

12.2 Il Trade-off della Memoria: Server GC vs Workstation GC

Il picco di 1.671,7 MB osservato su .NET 8 non era un *memory leak*, ma nemmeno un costo obbligato: era la conseguenza di una scelta di configurazione specifica, verificata e corretta in seguito:

1. **Server Garbage Collection (superato)**: la build .NET 8 usava il *Server GC*, che alloca un *heap* indipendente e un thread di GC dedicato per *ogni nucleo logico della CPU* (16 nel sistema di test). Su un carico dominato da un unico *scene heartbeat loop* e pochi worker script (`NumThreadScriptWorkers = 4`), questo parallelismo non veniva sfruttato a sufficienza da giustificarne il costo di memoria.

2. **Workstation GC + .NET 9 (attuale)**: ripetendo lo stesso benchmark su .NET 9 con `System.GC.Server=false` si ottiene un picco di soli 465,9 MB (media di due run, 464,8 e 466,9 MB) — una riduzione del 72% — a parita' di throughput script. Per completezza e' stato testato anche `System.GC.Server=true` su .NET 9: il picco sale a 704 MB (+51% rispetto a Workstation) senza alcun guadagno di throughput, confermando che Server GC resta la scelta peggiore per questo carico anche con i miglioramenti di *heap decommit* introdotti in .NET 9.

3. **CIL in RAM (invariato)**: ZEngine emette CIL nativo per ogni script tramite `Reflection.Emit` e mantiene residenti i *delegate* compilati. Questo costo di memoria resta, ma e' molto piu' piccolo del gonfiore introdotto da Server GC.

La configurazione attuale di `master` usa quindi Workstation GC su .NET 9. Il tag `dotnet8` nel repository marca l'ultimo commit prima di questa migrazione, per riferimento storico.

13 Conclusione della Trattazione Architetturale

Questo manuale tecnico esteso esplora i meandri dell'architettura di ZEngine e di CraftSim. Analizzando la compilazione JIT tramite `ILGenerator`, comprendiamo come ZEngine non si limiti a interpretare il linguaggio ma lo elevi ad applicazione nativa per l'OS ospite. Lo sblocco del thread pooling tramite Cooperative Multitasking / Block Splitting riduce il fenomeno storico dello "Stuttering Server", mentre il benchmark `bench_heavy.py` quantifica il beneficio in un throughput di eventi script cresciuto da 333 a 496 eventi/s (+49% rispetto a Vanilla) a parita' di *framerate* e numero di agenti, ottenuto oggi con un footprint di memoria di picco di soli ∼466 MB grazie alla migrazione a .NET 9 e Workstation GC. La parita' funzionale nell'interazione con i moduli BulletS e LSL_Api dimostra che CraftSim non scende a compromessi con la fedelta' retroattiva, proponendosi come una soluzione scalabile per la piattaforma OpenSimulator.

References

[1] OpenSimulator Project Community, *OpenSimulator Architecture & Developer Documentation*, Wiki ufficiale di OpenSimulator, aggiornato al 2026. `http://opensimulator.org/wiki/Architecture`

[2] Linden Research, Inc., *LSL Portal - Second Life Wiki*, Linden Lab Technical Publications. `http://wiki.secondlife.com/wiki/LSL_Portal`

[3] Miguel de Icaza et al., *The Mono Project - JIT Compiler Architecture*, Xamarin / Microsoft. `https://www.mono-project.com/docs/advanced/runtime/`

[4] Microsoft Corporation, *System.Reflection.Emit Namespace*, .NET Framework/Core Documentation. `https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit`

[5] Erwin Coumans, *Bullet Physics Engine - Real-Time Physics Simulation*, `https://pybullet.org/`

[6] Stephen Cleary, *Concurrency in C# Cookbook*, O'Reilly Media, 2nd Edition. (Discussione teorica sull'utilizzo di macchine a stati asincrone per implementare blocchi sospensivi non bloccanti).

[7] James Newton-King, *Newtonsoft.Json (Json.NET)*, `https://www.newtonsoft.com/json`

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, e Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools* (The Dragon Book), Pearson Education, 2006.

[9] OpenSimulator OSSL Documentation, *OSSL Functions Reference*, `http://opensimulator.org/wiki/OSSL_Implemented`