ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Triennale

Esecuzione Efficiente di Web App Rust su Piattaforma Web Assembly

Tesi in Tecnologie Web

Relatore
prof. Paolo Bellavista

Laureando
Davide Crociati

Ottobre 2023

Sommario

Piacere, sommario.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

WASM: verso il Web come piattaforma per la programmazione distribuita

1.1 Contestualizzazione

1.1.1 Trend di evoluzione del web

Inizialmente le applicazioni web erano costituite da semplici **pagine statiche** contenenti testo e immagini. Con l'evoluzione dell'ecosistema web, negli ultimi anni si è assistito a una trasformazione radicale delle applicazioni, guidata da una serie di tendenze e innovazioni tecnologiche che hanno portato a un'esperienza utente sempre più coinvolgente e interattiva.

L'introduzione di JavaScript e di librerie e framework correlati, hanno progressivamente arricchito l'interazione con le applicazioni web, introducendo livelli crescenti di interattività e trasformando le semplici pagine statiche in ambienti dinamici e convolgenti. Un cambiamento significativo in tal senso, è avvenuto con l'avvento delle **Single Page Application (SPA)** e di **AJAX**. Questo approccio ha rivoluzionato il tradizionale modello di navigazione e di sviluppo, consentendo alle applicazioni di essere caricate una sola volta e offrendo interazioni rapide e fluide grazie al caricamento dinamico di contenuti. Questo nuovo paradigma ha introdotto un'esperienza utente simile

a quella delle applicazioni native, con transizioni scorrevoli tra le sezioni dell'applicazione e senza interruzioni visibili.

Parallelamente, la complessità delle funzionalità offerte è cresciuta in modo esponenziale, arrivando a ciò che viene riconosciuto come Web 3.0 o **Web Semantico**. Ormai è la norma spaziare da applicazioni che usano algoritmi di intelligenza artificiale, a simulatori, alla modifica istantanea di documenti, immagini e video.

Questo enorme sviluppo di funzionalità è stato trainato dalla crescente potenza di elaborazione dei dispositivi e dalla capacità delle tecnologie web di sfruttare appieno queste risorse, facendo diventare normale interfacciarsi con siti web in grado di gestire complesse operazioni in tempi rapidi.

Tuttavia questo progresso è stato accompagnato da un aumento smisurato del **numero di richieste** effettuate in rete e dall'utilizzo intensivo di risorse computazionali, sia lato cliente, che lato servitore. È diventato cruciale bilanciare l'aggiunta di funzionalità sofisticate con la necessità di mantenere tempi di risposta rapidi e un'esperienza fluida per gli utenti. Inoltre, l'uso intensivo delle risorse ha sollevato questioni di scalabilità e di utilizzo efficiente delle risorse server.

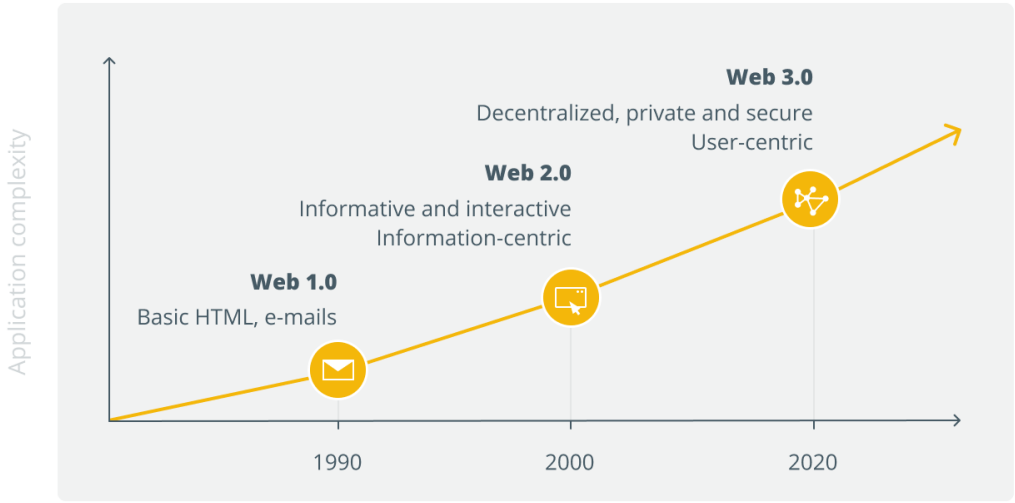

Figura 1.1: L'evoluzione del Web.

1.1.2 Importanza dell'ottimizzazione

Ad oggi, l'ottimizzazione delle prestazioni è quindi diventata un aspetto cruciale nello sviluppo di applicazioni web. Gli utenti si aspettano interazioni con bassa latenza, caricamenti rapidi e risposte immediate. Questa esigenza mette in risalto l'importanza di bilanciare l'aggiunta di nuove funzionalità, con l'offerta di una *User Experience* ottimale.
I tempi di caricamento prolungati possono portare a un alto tasso di abbandono delle pagine, riducendo l'opportunità di coinvolgere nuovi utenti. Inoltre, con l'aumentare dell'utilizzo di **dispositivi mobili** e di conseguenza, di **connessioni instabili**, l'ottimizzazione diventa ancor più critica per assicurare un'esperienza coerente su diverse piattaforme e condizioni di rete. Tutto ciò non riguarda solo il lato client, ma coinvolge anche il lato server. Un carico eccessivo sui server può influire negativamente sulla scalabilità, causando ritardi nelle risposte e possibili interruzioni del servizio. L'ottimizzazione deve quindi coinvolgere tutti gli aspetti dell'architettura delle applicazioni web.
Nell'implementare ottimizzazioni, sono nati varie soluzioni interessanti. Ad esempio, per gestire task che svolgono molte operazioni di Input/Output si è distinto il runtime environment **Node.js**[1], mentre per quanto riguardo l'esecuzione di attività che sfruttano molto la CPU è emerso **WebAssembly(Wasm)**[2]. Quest'ultimo, nato inizialmente per consentire l'esecuzione di codice ad alta efficienza all'interno del browser, è diventato portabile anche su piattaforme server side grazie allo sviluppo dell'interfaccia di sistema **WebAssembly System Interface (WASI)**[3].

1.2 Motivazioni e Obiettivi

La **crescente complessità** delle applicazioni web e l'esigenza di offrire agli utenti esperienze interattive sempre più coinvolgenti hanno portato l'ambito dello sviluppo web a una svolta significativa. Le aspettative degli utenti si sono evolute verso applicazioni che offrano prestazioni reattive, interattività immediata e funzionalità avanzate.
È proprio questo insieme di aspettative a essere alla base delle motivazioni che hanno guidato la scelta del tema di questa tesi di laurea.
In particolare, la presente ricerca, si propone di confrontare in modo dettagliato, due differenti approcci di sviluppo per un'applicazione con funzionalità **fortemente CPU-intensive**.

1.2.1 Tipologia di applicazione

Per tale confronto, si è optato per una web-app che implementi alcune tecniche di **elaborazione di immagini**. In particolare, l'utente avrà la possibilità di eseguire l'upload di immagini su un server e indicare una serie di modifiche da apportare (ad esempio "ridimensionamento del 50%"). Il server manipolerà i file in base ai parametri ricevuti e infine restituirà al cliente le immagini modificate.
Non verrà esplorato in modo approfondito il campo dell'elaborazione digitale di immagini, ma ci si limiterà all'implementazione di funzionalità usate spesso da utenti comuni, come ad esempio, ridimensionamento, rotazione, aumento/diminuzione di luminosità/contrasto e altre che verranno specificate nel capitolo 3.
Tale tipologia di applicazione, si sposa bene per lo scopo finale della tesi: valutare come due approcci (e due linguaggi) piuttosto differenti, ma sempre più diffusi al giorno d'oggi, risolvano il problema di un'applicazione web che svolga operazioni dall'alto costo computazionale.

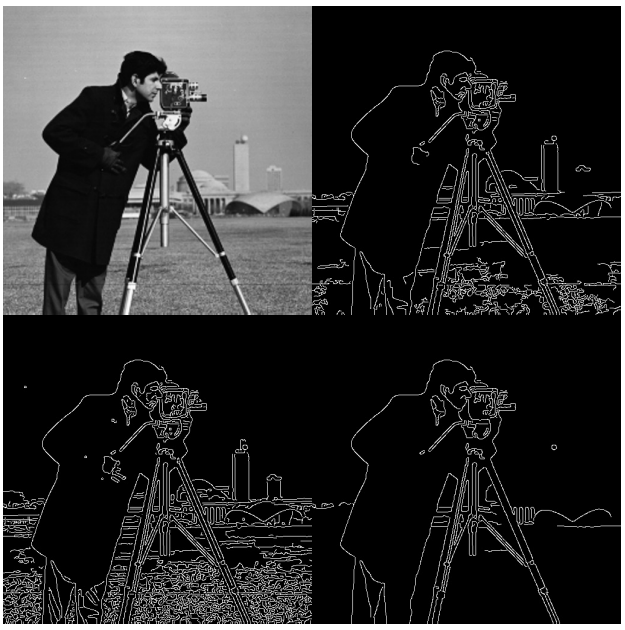

Figura 1.2: Il campo dell'elaborazione digitale di immagini è molto ampio e sarebbe possibile integrare anche operazioni avanzate (edge detection, pattern recognition etc.), che richiederebbero risorse computazionali molto elevate. Per lo scopo di questa tesi ci si limiterà a elaborazioni più semplici, ma in ogni caso rilevanti e sufficienti a mettere alla prova la CPU.

1.2.2 Metodologie confrontate

Le due modalità di sviluppo in esame riguarderanno l'utilizzo delle tecnologie JavaScript e WebAssembly lato server e quindi rispettivamente del runtime environment Node.js e del linguaggio di programmazione **Rust** in combinazione con WebAssembly System Interface[4].
La scelta di Node.js deriva dal suo utilizzo sempre maggiore grazie all'utilizzo del linguaggio JavaScript, dall'approccio asincrono nella gestione delle richieste e dalla sua ottima scalabilità per applicazioni fortemente File-System-Intensive. Per quanto riguarda la seconda tecnologia si è invece optato per Rust, in quanto consente sia la scrittura di **moduli** successivamente **compilabili in WebAssembly**, sia il loro utilizzo efficiente all'interno del codice, mantenendo in questo modo un'ottima coerenza e risultando, teoricamente, una buona scelta per lo sviluppo di applicazioni computazionalmente complesse.

1.2.3 Analisi Comparativa delle Tecnologie

Un elemento iniziale di questa ricerca sarà un'analisi dettagliata delle tecnologie prese in esame. Inizialmente, nel capitolo 2, verrà svolta un'analisi comparativa delle due tecnologie, presentando i vantaggi e gli svantaggi teorici che caratterizzano entrambe le opzioni.
Sarà infatti fondamentale comprendere come ciascuna affronti la complessità legata ad operazioni I/O-intensive e CPU-intensive. Questo ci permetterà di valutare nel modo più opportuno i risultati che emergeranno succesivamente durante i test e i benchmark dell'applicazione sviluppata.

Figura 1.3: Rust e WebAssembly

Si procederà poi ad illustrare in modo approfondito il funzionamento delle API sfruttate. Verranno analizzate le peculiarità del linguaggio Rust che lo rendono adatto ad applicazioni ad alta intensità computazionale. Allo stesso modo ci si soffermerà sull'efficacia di WebAssembly nell'esecuzione di codice di basso livello con prestazioni paragonabili a quelle dei linguaggi nativi.
Nel contempo si analizzerà anche la metodologia basata su Node.js, concentrandosi sull'efficienza e la flessibilità che questo ambiente di esecuzione JavaScript può offrire in ambito web.

Figura 1.4: Node.js

Si proseguirà affrontando un'approfondimento sulle prestazioni di ciascuna tecnologia, evidenziando scenari in cui una risulti più vantaggiosa. Questo approfondimento sarà alla base delle successive valutazioni sulle prestazioni delle applicazioni sviluppate con i due metodi presi in esame.

1.2.4 Valutazione dell'Impatto di Wasm

WebAssembly, in particolare, è emerso come **un'innovazione** cruciale nel mondo dello sviluppo web. Consentendo l'esecuzione di codice a basso livello con prestazioni paragonabili a quelle dei linguaggi nativi, Wasm offre la possibilità di ottenere prestazioni elevate all'interno di un ambiente browser-based.
Parallelamente, WebAssembly System Interface (WASI) gioca un ruolo chiave nell'estendere il potenziale di WebAssembly. WASI fornisce un'interfaccia standardizzata per l'**accesso a risorse di sistema**, consentendo alle applicazioni di interagire con l'ambiente circostante in modo controllato e sicuro. Tale capacità è particolarmente rilevante nell'ambito delle applicazioni web CPU-intensive, in quanto consente di accedere, in maniera efficiente, alle risorse necessarie per eseguire complesse operazioni di calcolo e manipolazione dei dati.

Sarà quindi fondamentale valutare e cercare di misurare l'impatto di WASI, sia per quanto riguarda le prestazioni sia per quanto riguarda lo sviluppo e l'integrazione dello stesso all'interno di un'applicazione web.

1.2.5 Obiettivi

Si intende esplorare le opportunità offerte dalle tecnologie enunciate sopra, nell'ottica di un'ottimizzazione delle prestazioni. In questo contesto l'obiettivo centrale è quello di comprendere i **benefici specifici** legati a ciascun approccio, individuando le circostanze in cui uno dei due possa risultare vantaggioso in termini di **efficienza computazionale** e reattività. Inoltre si intende sottolineare il fatto che le decisioni intraprese sin dalla fase di progettazione, possono avere un impatto significativo sulle prestazioni e sull'esperienza utente di un'applicazione.
Si vuole mettere in evidenza l'importanza di una valutazione attenta e accurata dell'architettura e delle esigenze dell'applicazione stessa. Aspetto da affrontare attraverso un'analisi dettagliata che considera le funzionalità dell'applicazione e valuta se esse siano orientate al calcolo intensivo o ad un'ampia manipolazione del File System.
Per raggiungere in maniera **efficace e quantificabile** gli obiettivi, verrà fatta un'analisi approfondita delle performance e delle prestazioni delle due applicazioni sviluppate. Questo consentirà di valutare in modo empirico e misurabile l'impatto di ciascuna tecnologia nel contesto di applicazioni CPU-intensive.
Sarà inoltre trattato anche il tema della **scalabilità**. Esso diventa di primaria importanza quando i volumi di traffico e i carichi di lavoro aumentano. Valutare come diverse tecnologie gestiscano richieste concorrenti è fondamentale per determinare quale sia più adatta alle esigenze di un progetto.
Infine si considererà anche la **facilità di sviluppo e l'espandibilità**. Questi temi avanzano spesso di pari passo ed è importante, in un ambiente web in rapida evoluzione, che un progetto si adatti facilmente, sia all'aggiunta di nuovi requisiti e funzionalità, sia all'integrazione con altri sistemi e servizi. Tale flessibilità può infatti avere un impatto significativo sulla vita e sulla sostenibilità di un progetto.

1.3 Struttura della tesi

La tesi seguirà una struttura articolata in modo da affrontare in maniera approfondita gli aspetti chiave delle tecnologie e delle analisi proposte.
Nel Capitolo 2, sarà presentata una panoramica esaustiva delle tecnologie coinvolte in questa ricerca. Questo capitolo fornirà un'analisi dettagliata di WebAssembly/WASI in combinazione con Rust e di Node.js, esaminandone le caratteristiche, i vantaggi e le **API** utili allo sviluppo dell'applicazione. Verranno esplorati i contesti in cui ciascuna tecnologia si dimostra più adeguata, fornendo le basi per una comprensione approfondita delle successive misurazioni e valutazioni.
Nel Capitolo 3, il focus sarà sul prototipo dell'applicazione sviluppata. Saranno illustrate le scelte progettuali, l'architettura complessiva e le funzionalità implementate. Successivamente, verranno presentati i dettagli delle misurazioni condotte, insieme ai criteri di valutazione delle prestazioni delle tecnologie in esame. Saranno discussi i risultati ottenuti e i confronti tra le diverse implementazioni, evidenziando gli aspetti in cui ognuno degli approcci studiati, si distingue in termini di ottimizzazione delle prestazioni.

Capitolo 2

Tecnologie utilizzate

Come già introdotto brevemente nel capitolo 1, il focus di questa tesi è stato impostato sulla comparazione di due approcci radicalmente distinti, ma entrambi sempre più rilevanti nelle applicazioni web moderne. Si inizierà esaminando l'uso di WebAssembly e dell'interfaccia di sistema WASI, nonché la loro integrazione all'interno di un'applicazione Rust, per poi concludere con un'analisi di Node.js.

2.1 WebAssembly

WebAssembly (**Wasm**) è uno standard che definisce un formato **binario** (.wasm) e un relativo formato **testuale** (.wat) per la scrittura di codice eseguibile nelle pagine web. Attualmente, il suo sviluppo è gestito in modo open-source da un *Community Group* del W3C, il quale coinvolge sviluppatori dei browser più utilizzati.
Esso è nato principalmente come **integrazione a JavaScript**, con l'obiettivo di consentire l'esecuzione di codice ad una velocità paragonabile a quella del codice nativo. Ciò è possibile grazie alle dimensioni ridotte dei file binari generati e alla loro efficienza.
Inoltre grazie all'esecuzione all'interno di una sandbox garantisce un ottima sicurezza, anche sotto il punto di vista della memoria.
Grazie al formato testuale (.wat), è possbile eseguire debug, test, ottimizzazioni, scrivere a mano programmi di basso livello, ma anche visualizzare i sorgenti dei moduli wasm, quando questi sono utilizzati una pagina web. [5]
Oggi WebAssembly sta vivendo una crescita costante, sia nel numero di linguaggi che

possono avere come *compilation target* Wasm (Rust, C, Go, Kotiln, etc), sia nel numero di siti che sfruttano tale tecnologia. Questo successo è dovuto anche al fatto che praticamente tutti i browser attualmente in uso offrono supporto per WebAssembly.

Figura 2.1: WebAssembly all'interno del browser

2.1.1 Storia e Origini

WebAssembly nasce nel 2015, ispirandosi fortemente ad altre due tecnologie preesistenti: Google Native Client e asm.js.

I predecessori

Native Client (NaCl) è stato un progetto open source rilasciato nel 2011 da Google. L'obiettivo di NaCl era consentire l'esecuzione di codice nativo all'interno del browser, confinato in una **sandbox** con privilegi limitati. In particolare si stava cercando di supportare software che richiedevano grande sforzo computazionale, come simulazioni, elaborazioni audio-video e giochi.
Il progetto si rivelò un successo sotto il punto di vista prestazionale. Vennero infatti rilasciati diversi software e giochi che presentavano prestazioni simili alla rispettiva versione Desktop. Tuttavia NaCl presentava diverse limitazioni, in quanto il codice ottenuto era eseguibile solo nel browser Chrome ed era impossibile l'interazione con JavaScript o con altre API web.
Un tentativo di evoluzione fu presentato nel 2013 dal team di Mozilla. Si trattava di **asm.js**, un sottoinsieme di JavaScript che consentiva l'invocazione di funzioni scritte in linguaggi come C, C++ o Rust, in diversi browser e direttamente da JavaScript. A discapito di una maggior portabilità ci fu una significativa diminuzione delle prestazioni,

dovuta alla lentezza dell'interprete JavaScript, che era stato caricato di un notevole overhead.
Queste due soluzioni dimostrarono la possbilità di eseguire codice in una sandbox, o con ottime prestazioni, ma solo all'interno di Chrome (NaCl), oppure in diversi browser ma con prestazioni decisamente inferiori (asm.js). Si voleva quindi trovare un modo per unificare gli enormi vantaggi offerti da ognuno dei due approcci.[6]

La nascita di Wasm

La nascita di WebAssembly è avvenuta nel giugno 2015, quando Brendan Eich (il creatore di JavaScript) insieme ad altri sviluppatori di Mozilla ha annunciato l'inizio dello sviluppo. [7] Wasm venne presentato come "un nuovo standard open source che definiva un formato e un modello di esecuzione portabile, efficiente in termini di dimensioni e tempo di caricamento, specificamente progettato come target di compilazione per il web". WebAssembly prometteva prestazioni fino a 20 volte superiori rispetto ad asm.js, grazie alla maggiore velocità di decodifica dei file binari, rispetto al parsing da parte dell'interprete JavaScript per i file di asm.js.
Nel 2017 è stato lanciato il *minimum viable product* (MVP), che conteneva pressochè le stesse funzionalità presenti in asm.js e venne dichiarata conclusa la fase di preview. Capendo le potenzialità di ciò che stava venendo sviluppato hanno dato il loro contributo al progetto, aziende del calibro di Google, Microsoft, Apple, Unity.

2.1.2 Casi d'uso

Inizialmente ci si è concentrati sull'utilizzo di WebAssembly all'interno del browser e venivano presentati svariati casi d'uso:

- Elaborazione di immagini e video;
- Giochi;
- Applicazioni peer-to-peer;
- Applicazioni CAD;
- Realtà virtuale e realtà aumentata con latenza minima;
- Riconoscimento di immagini;

- Simulazione/emulazioni di sistemi operativi (QEMU, DOSBox);
- Applicazioni per desktop ridimensionamento;
- Web server locali;

Si era pensato fin da subito anche alla portabilità ed erano stati presentati diversi possibili utilizzi anche per ambienti non-web:

- Esecuzione server-side di codice non attendibile;
- Servizi per la distribuzione di giochi;
- Generiche applicazioni server side;
- Calcolo simmetrico, distribuito su più nodi;

2.1.3 Concetti chiave

WebAssembly codifica un linguaggio di programmazione di basso livello, simile ad Assembly. Tale linguaggio è strutturato attorno ai seguenti concetti:[8]

Valori

In WebAssembly sono presenti:

- Un tipo *byte* per la rappresentazione di byte non interpretati;
- Quattro tipi di valori numerici: interi e numeri a virgola mobile, ognuno da 32 o 64 bit (*i32, i64, f32, f64*);
- Un tipo vector a 128 bit (*i128*) contenente anch'esso valori numerici (ad esempio 2 f64, oppure da 4 i32 etc.);
- Un tipo riferimento per puntatori a differenti entità;

Quest'ultimo è definito "opaco", in quanto non ne è visibile né la dimensione, né la rappresentazione in bit. Al contrario i primi due tipi si dicono "trasparenti".

Istruzioni

Il modello computazionale di WebAssembly è basato su uno **stack**. Il codice è costituito da una sequenza di istruzioni eseguite in ordine. Le istruzioni sfruttano una struttura dati detta *operand stack* e possono essere di tipo semplice, o di controllo.
Le operazioni semplici svolgono manipolazioni basilari sui dati, prelevando parametri dallo stack (*pop*) e inserendo il risultato nello stesso (*push*). Le operazioni di controllo, si occupano invece di alterare il flusso di esecuzione grazie a costrutti condizionali, blocchi e cicli.

Traps

Alcune istruzioni possono fallire e generare degli errori (traps) che non è possibile gestire all'interno di WebAssembly. Tali errori vengono infatti lanciati nell'ambiente di esecuzione dell'host dove, al contrario, possono essere catturati e gestiti opportunamente.

Funzioni

Il codice è diviso in funzioni. Ognuna di queste ha una certa sequenza di valori sia come parametri di input, che come tipo di ritorno. In una funzione viene eseguita una serie di istruzioni, possono inoltre essere invocate altre funzioni (anche ricorsivamente) e create variabili locali.

Tabelle

Una tabella è un array di valori "opachi" di un particolare tipo. Tale struttura consente al programma di ottenere i valori indirettamente attraverso un indice dinamico. Tramite le tabelle è possibile, per esempio invocare funzioni indirettamente, emulando in questo modo i puntatori a funzione.

Memoria lineare

La memoria lineare è un array continuo di byte. Ha una dimensione iniziale che può crescere dinamicamente al bisogno. Un programma può leggere e scrivere valore in memoria (operazioni di *load/store*) in un qualsiasi indirizzo al suo interno (anche in maniera non allineata).

Moduli

Un modulo è l'unità di deployment per un programma WebAssembly. Esso conterrà le definizioni di funzioni, tabelle, memoria etc. In un modulo è anche possibile esportare o importare definizioni, inizializzare tabelle o memoria lineare e anche definire una funzione *start* che verrà eseguita automaticamente.

Embedder

Solitamente un modulo WebAssembly sarà integrato in un host che ne definirà l'inizializzazione, la risoluzione delle funzioni importate e le modalità di accesso di quelle esportate. L'Embedder è l'entità che implementa la connessione tra l'ambiente host e il modulo Wasm. Ci si aspetta che l'embedder interagisca con la semantica di WebAssembly in un modo ben definito nelle specifiche del formato Wasm.

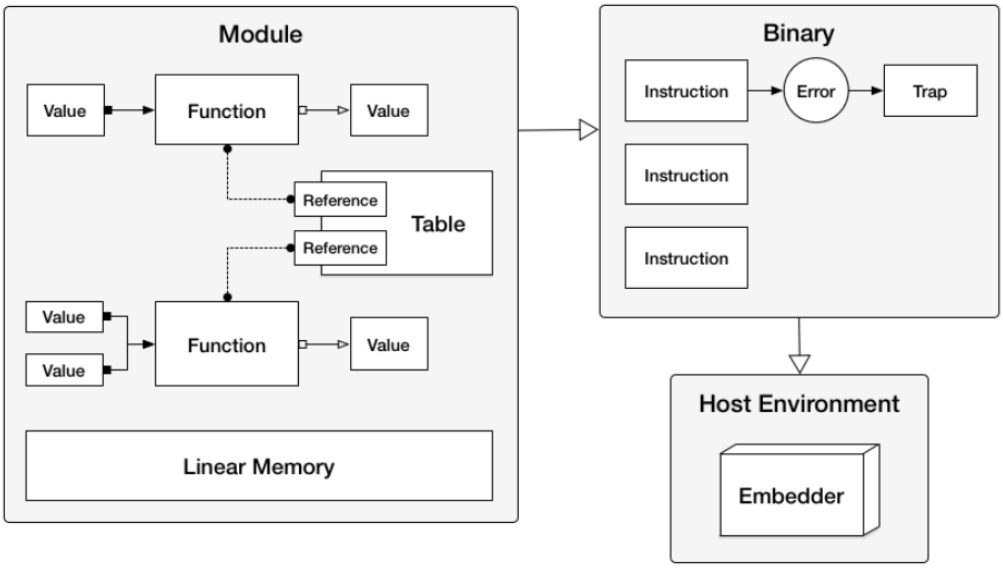

Figura 2.2: Architettura di WebAssembly

2.1.4 Fasi semantiche

Nella semantica di WebAssembly è possibile individuare tre fasi principali: decodifica, validazione ed esecuzione.[8]

Decodifica

I moduli WebAssembly sono distribuiti in formato binario (.wasm) e per questo è necessario decodificarli in modo da ottenere una rappresentazione interna del modulo, con cui il web browser o il runtime environment potrà lavorare.

Validazione

Dopo aver decodificato i moduli binari, per poterli istanziare, è necessario controllare che questi siano validi. La validità è verificata grazie ad un sistema di tipi basato sulla sintassi astratta di un modulo e sul suo contenuto. In particolare per ogni componente della sintassi è presente una regola che specifica le condizioni da rispettare perchè il modulo risulti valido. Ad esempio viene controllato l'ordine delle istruzioni nel corpo di una funzione assicurandosi che lo stack sia utilizzato nel modo corretto.

(value types) $t ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$
(packed types) $tp ::= \mathsf{i8} \mid \mathsf{i16} \mid \mathsf{i32}$
(function types) $tf ::= t^* \to t^*$
(global types) $tg ::= \mathsf{mut}^? \; t$

$unop_{iN} ::= \mathbf{clz} \mid \mathbf{ctz} \mid \mathbf{popcnt}$
$unop_{fN} ::= \mathbf{neg} \mid \mathbf{abs} \mid \mathbf{ceil} \mid \mathbf{floor} \mid \mathbf{trunc} \mid \mathbf{nearest} \mid \mathbf{sqrt}$
$binop_{iN} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div}\_sx \mid \mathbf{rem}\_sx \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \mid \mathbf{shl} \mid \mathbf{shr}\_sx \mid \mathbf{rotl} \mid \mathbf{rotr}$
$binop_{fN} ::= \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \mathbf{min} \mid \mathbf{max} \mid \mathbf{copysign}$
$testop_{iN} ::= \mathbf{eqz}$
$relop_{iN} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt}\_sx \mid \mathbf{gt}\_sx \mid \mathbf{le}\_sx \mid \mathbf{ge}\_sx$
$relop_{fN} ::= \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{lt} \mid \mathbf{gt} \mid \mathbf{le} \mid \mathbf{ge}$
$cvtop ::= \mathbf{convert} \mid \mathbf{reinterpret}$
$sx ::= \mathbf{s} \mid \mathbf{u}$

(instructions) $e ::= \mathbf{unreachable} \mid \mathbf{nop} \mid \mathbf{drop} \mid \mathbf{select} \mid$
$\mathbf{block}\ tf\ e^*\ \mathbf{end} \mid \mathbf{loop}\ tf\ e^*\ \mathbf{end} \mid \mathbf{if}\ tf\ e^*\ \mathbf{else}\ e^*\ \mathbf{end} \mid$
$\mathbf{br}\ i \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ i^+ \mid \mathbf{return} \mid \mathbf{call}\ i \mid \mathbf{call\_indirect}\ tf \mid$
$\mathbf{get\_local}\ i \mid \mathbf{set\_local}\ i \mid \mathbf{tee\_local}\ i \mid \mathbf{get\_global}\ i \mid$
$\mathbf{set\_global}\ i \mid t.\mathbf{load}\ (tp\_sx)^?\ a\ o \mid t.\mathbf{store}\ tp^?\ a\ o \mid$
$\mathbf{current\_memory} \mid \mathbf{grow\_memory} \mid t.\mathbf{const}\ c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*\ \mathbf{func}\ tf\ \mathbf{local}\ t^*\ e^* \mid ex^*\ \mathbf{func}\ tf\ im$
(globals) $glob ::= ex^*\ \mathbf{global}\ tg\ e^* \mid ex^*\ \mathbf{global}\ tg\ im$
(tables) $tab ::= ex^*\ \mathbf{table}\ n\ i^* \mid ex^*\ \mathbf{table}\ n\ im$
(memories) $mem ::= ex^*\ \mathbf{memory}\ n \mid ex^*\ \mathbf{memory}\ n\ im$
(imports) $im ::= \mathbf{import}$ "$name$" "$name$"
(exports) $ex ::= \mathbf{export}$ "$name$"
(modules) $m ::= \mathbf{module}\ f^*\ glob^*\ tab^?\ mem^?$

Figura 2.3: Una parte della sintassi astratta di WebAssembly

Se nelle specifiche esaminiamo un'operazione binaria tra tipi numerici (che quindi avrà due valori in ingresso ed un valore in uscita; es: i32.add, f64.xor etc.), notiamo che essa è del tipo $t.binop$ e dato un contesto C essa è valida solamente con tipo $[t\ t] \to [t]$.

Tale regola si può anche rappresentare con la seguente notazione formale:

$$\frac{}{C \vdash t.binop : [t\ t] \rightarrow [t]}$$

Esaminando invece le regole di validazione di un'istruzione per l'aumento di memoria lineare *memory.grow* (tale istruzione si aspetta l'offset che indica di quanto la memoria è da espandere e ritorna la dimensione della memoria precedente all'espansione) notiamo che:

- La memoria $C.mems[0]$ deve essere definita nel contesto;
- Allora l'istruzione risulta valida con tipo $[i32] \rightarrow [i32]$;

In notazione formale:

$$\frac{C.mems[0] = memtype}{C \vdash memory.grow : [i32] \rightarrow [i32]}$$

Esecuzione

Terminata la validazione di ogni istruzione il modulo può finalmente essere istanziato. L'istanza di un modulo ne è la rappresentazione dinamica. Esso comprende lo stack, sul quale operano le istruzioni WebAssembly e uno ***store*** astratto, contenente lo stato globale (è la rappresentazione a runtime di tutte le istanze di funzioni, tabelle, memorie etc. che sono state allocate dall'istanziazione). Terminata l'istanziazione, diventa effettivamente possibile l'esecuzione di istruzioni WebAssembly.
In particolare, se nel modulo era stata definita una funzione *_start*, essa sarà eseguita subito dopo la creazione dell'istanza, altrimenti sarà possibile invocare funzioni esportate, chiamandole direttamente dall'istanza stessa.
Per ogni istruzione, è presente una regola che specifica l'effetto della sua esecuzione sullo stato del programma. Rimanendo coerenti con gli esempi sulla validazione, segue il comportamento dell'istruzione *t.binop*:

- In seguito alla validazione, possiamo assumere che due valori di tipo t si trovino in cima allo stack;
- Viene eseguita l'operazione di **pop** del valore $t.const\ c_1$ dallo stack;
- Viene eseguita l'operazione di **pop** del valore $t.const\ c_2$ dallo stack;
- Se $binop_t(c_1, c_2)$ è definita allora:

  - Sia c il possibile risultato di $binop_t(c_1, c_2)$;
  - Viene eseguita l'operazione di **push** del valore $t.const\ c$ nello stack.

- Altrimenti:
  - Trap.

In notazione formale:

$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \quad \hookrightarrow \quad (t.\mathsf{const}\ c) \qquad (\text{if } c \in binop_t(c_1, c_2))$$
$$(t.\mathsf{const}\ c_1)\ (t.\mathsf{const}\ c_2)\ t.binop \quad \hookrightarrow \quad \mathsf{trap} \qquad (\text{if } binop_t(c_1, c_2) = \{\})$$

Non viene riportato l'esempio anche sull'istruzione *memory.grow* essendo una funzionalità decisamente più complessa (11 diversi passaggi, con svariati sottocasi) che comporterebbe obbligatoriamente, l'introduzione di ulteriori dettagli che esulano dallo scopo di questa tesi. Si noti che sia l'istanziazione, che l'invocazione di funzioni, sono operazioni che avvengono all'interno dell'ambiente di esecuzione dell'host.

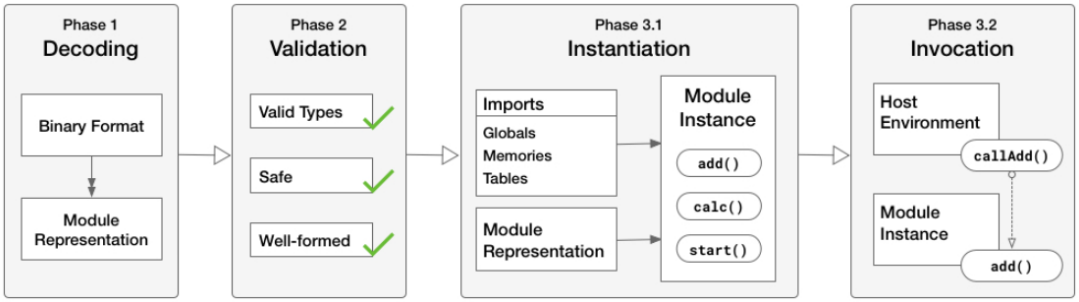

Figura 2.4: La semantica di WebAssembly

2.1.5 Correttezza logica

Grazie alla sua semantica è possibile dire che il sistema di tipi di WebAssembly è logicamente corretto (*sound*)[9]. Esso infatti garantisce sia ***type safety***, che ***memory safety***.
Per quanto riguarda la type safety, tutti i tipi controllati durante la validazione saranno rispettati anche a runtime:

- Ogni variabile (locale o globale) conterrà valori del tipo corretto.
- Ogni istruzione verrà applicata solo ad operandi del tipo che ci si aspetta e restituirà valori del tipo corretto.
- Ogni funzione restituirà valori del tipo previsto, a meno di errori (trap).

Per quanto riguarda la *memory safety*, è garantito che non verrà acceduta nesun'area di memoria, diversa da quelle esplicitamente specificate dal programma. Ogni volta che per esempio, verrà fatta un'operazione di load/store in un certo indirizzo di memoria, si controllerà che quest'ultimo sia nel range di indirizzi possibili per l'istanza attuale di WebAssembly.
Inoltre sono garantite altre proprietà:

- L'**assenza di comportamenti indefiniti**: le regole di esecuzione sono mutualmente consistenti e coprono qualsiasi caso che possa capitare in un programma validato.
- L'**incapsulamento** per funzioni e moduli: nessuna variabile locale può essere acceduta al di fuori della funzione in cui è dichiarata e nessun componente di un modulo può essere acceduto fuori dallo stesso, a meno che il componente in questione non sia esportato o importato.

Infine è importante notare che le regole di validazione si occupano solamente dei componenti statici di un programma WebAssembly. Per dimostrare correttezza in maniera precisa, sono state estese le *typing rules* anche ai componenti dinamici, come lo **store**, le **configurations** (coppie Store - Thread di esecuzione) e le istruzioni amministrative. Dopo aver definito nelle specifiche una configurazione valida (una configurazione che rispetti le typing rules) è stato possibile enunciare due teoremi e derivare un corollario.
In sostanza tali teoremi affermano che: ogni thread in una *configuration* valida, o esegue

per sempre, o termina con un errore (trap), o termina con un risultato del tipo atteso. Di conseguenza, dato uno *store* valido, nessun calcolo derivato dall'istanzazione o dall'invocazione di un modulo valido, potrà presentare comportamenti diversi da quelli definiti nelle specifiche. [10]

2.1.6 Creazione e utilizzo di moduli WebAssembly

Sebbene sia possibile scrivere a mano il codice dei moduli WebAssembly, è molto più comune scrivere il codice in un linguaggio di alto livello e poi compilare quest'ultimo in un modulo Wasm. Nonostante durante lo sviluppo iniziale del linguaggio ci si fosse concentrati solo sull'utilizzo di C/C++, ad oggi sono più di 10 i linguaggi che supportano WebAssembly come *compilation target*. Tra questi spiccano Rust, C#, Python, Kotlin, Swift, Go.
Per esempio, utilizzando Rust sarebbe sufficiente scrivere un normale programma e poi lanciare il seguente comando nella directory del progetto:

```
wasm-pack build
```

Tale comando compilerà il programma in un modulo WebAssembly, generando un file **.wasm** utilizzabile dal codice JavaScript della nostra pagina web.
Bisogna però sottolineare che sin dagli albori di WebAssembly, non era mai stata fatta nessuna assunzione sull'ambiente host e che soprattutto il suo utilizzo non sarebbe stato limitato alle sole pagine web.
In particolare, per lo scopo di questa tesi, sarà necessario che i moduli WebAssembly vengano istanziati ed eseguiti server side, ma soprattuto che tali moduli abbiano accesso al **file system** e a funzioni di **I/O**. Tali funzioni in WebAssembly non sono disponibili, a causa dell'esecuzione all'interno di una sandbox e delle caratteristiche di sicurezza. Per questi e per molti altri motivi è stato sviluppato WASI (WebAssembly System Interface).

2.2 WebAssembly System Interface

WebAssembly System Interface è una famiglia di APIs progettate dai membri della Bytecode Alliance durante lo sviluppo del progetto **Wasmtime**.[3]
WASI si propone di standardizzare l'accesso a varie funzionalità tipicamente associate ai sistemi operativi, da parte di programmi scritti in WebAssembly. In particolare, lo sviluppo è partito da una serie di funzioni basiche simili a quelle contenute in POSIX. D'ora in avanti queste funzioni verranno spesso menzionate come **syscall**, in quanto esse hanno uno scopo simile o analogo a quello delle system call in programmi eseguibili nativi. Bisogna però tenere a mente che tali "syscall" sono semplicemente una serie di funzioni che permettono di eseguire operazioni di I/O. Ne sono un esempio:

- Accesso a file e File System;
- Funzioni per le socket di Berkeley;
- Funzionalità relative al tempo e ai numeri random;

WASI è progettato in modo da essere indipendente dai browser e da non dipendere da JavaScript o da API web. Inoltre utilizza un sistema di sicurezza *capability based* che consente di estendere le caratteristiche di sandboxing presenti in WebAssembly, rendendo possibile l'I/O.
Attualmente WASI ha attraversato due fasi di preview (preview0, preview1), le quali dovrebbero essere seguite da due ulteriori preview e infine dalla prima release ufficiale (WASI 1.0).

2.2.1 Gli obiettivi

WASI è nato con una serie di obiettivi di alto livello, che si ispirano e integrano gli obiettivi di WebAssembly.[11] Esso si propone di:

- Creare un insieme di API specificamente progettate per programmi WebAssembly. Esse devono essere portabili (utilizzabili su diversi sistemi), modulari (composte da più componenti riutilizzabili), indipendenti dal runtime sottostante e devono mantenere la caratteristiche di sicurezza garantite da WebAssembly (sandbox), integrandole con la presenza di **Capabilities**;

- Sviluppare e implementare funzionalità in maniera incrementale. Inizialmente si ha un Minimum Viable Product (MVP) che include funzionalità essenziali, che verrà gradualmente espanso aggiungendo nuove feature sulla base di feedback ed esperienze pratiche;
- Arricchire la progettazione delle API con documentazione e test completi e quando possibile, fornire anche implementazioni di riferimento condivisibili tra diversi runtime Wasm;
- Lavorare al fianco degli sviluppatori di strumenti e librerie Wasm, così da garantire ai loro utenti anche il supporto per WASI;
- Consentire alle proprie API di evolvere nel tempo. Ciò significa anche consentire l'evoluzione di moduli standardizzati e l'introduzione di nuovi moduli. Per garantire compatibilità possono essere create implementazioni dei moduli standardizzati, utilizzando librerie basate su nuove API, consentendo in questo modo una transizione fluida mentre la piattaforma evolve;

2.2.2 I principi di design

WebAssembly System Interface è tutt'ora in fase di sviluppo secondo una serie di ***design principle*** che verranno di seguito illustrati:[12]

Capability

WASI segue un concetto di sicurezza basata su capability che era già stata presentata qualche anno prima da CloudABI's.
File, directory, socket e altre risorse sono identificati da **indici interi** (paragonabili ai file descriptor utilizzati nei sistemi UNIX), di tabelle esterne che contengono le capabilities per un certo modulo WASI.
Le API di WASI potranno quindi accedere ed utilizzare una risorsa, solo se a questa è associata una determinata capability. Ad esempio, per l'apertura di un file WASI mette a disposizione una system call simile alla **openat**. Essa si aspetta che il processo chiamante abbia il file descriptor della directory contenente tale file, il quale rappresenterà la capability per l'apertura dei file all'interno di quella specifica directory.
In realtà è anche possibile un approccio più simile alla classica system call "open". Infatti,

grazie alla pre-apertura di una directory al lancio del programma essa verrà inclusa nella tabella delle capabilities, la quale verrà automaticamente controllata a runtime ogni volta che il programma invocherà la "open".
Tutto ciò è simile a ciò che avviene normalmente in POSIX, con la differenza che POSIX consente ad un processo di richiedere qualsiasi file descriptor nell'intero file system e poi l'accesso sarà concesso a seconda delle policy di sistema, mentre in WASI è necessario che un programma abbia le capability giuste per accedere alle risorse di una certa directory. In questo modo diventa possbile l'esecuzione di codice non attendibile, dando permessi per specifiche directory a runtime, senza la necessità di impostare permessi nel file system.[13]

Interposizione

L'interposizione è la capacità di un'istanza di WebAssembly di implementare un'interfaccia WASI che poi potrà essere utilizzata in maniera trasparente da un'altra istanza (consumer). Ciò può essere sfruttato per adattare o attenuare le funzionalità di un API WASI, senza cambiare il codice che la utilizza. Tale concetto viene anche spesso chiamato "virtualizzazione".

Compatibilità

La compatibilità con applicazioni e host diversi è un obiettivo primario di WASI. Certe volte, però, è possibile che essa sia in conflitto con una progettazione chiara delle API, oppure con la sicurezza, le performance o la portabilità. Perciò si è cercato di appesantire il meno possibile le API di WASI con aspetti riguardanti la compatibilità e di garantire questo tema grazie all'utilizzo di librerie. Ne è un esempio WASI-libc, libreria costruita on top alle system call di WASI che fornisce una grande varietà di API compatibili con posix (standard I/O, file I/O, gestione della memoria, stringhe etc.). In questo modo applicazioni che non hanno bisogno di particolari funzionalità di compatibilità, non verranno appesantite da complessità non necessaria.

Portabilità

La possibilità di eseguire codice in maniera consistente ed efficace in runtime environment o su piattaforme differenti è molto importante nel contesto di WASI. Bisogna però sottolineare che il significato di portabilità potrebbe variare molto da un API all'altra.

Come già introdotto precedentemente WASI nasce come una tecnologia modulare e non necessariamente ogni sua API dovrà essere implementata da ogni WebAssembly Engine. La decisione è stata quella di non escludere un API solo perchè alcune tipologie di host non sarebbero in grado di implementarla totalmente.
In ogni caso verranno preferite API che possano essere eseguite in più ambienti possibili, ma la valutazione su cosa sia sufficientemente portabile e cosa no, verrà effettuata caso per caso.

Modularità

WASI è progettato per offrire svariate interfacce ed è pressochè impossibile che tutte siano appropriate per tutti gli *host environment*. Per questo motivo WASI utilizza un approccio a **componenti** che consente di descrivere quali API siano adatte per un certo ambiente e quali no. In quaesto modo ci si assicura che un programma avrà accesso solamente alle interfacce che sono rilevanti per un dato environment.

2.2.3 Wasmtime

Per lo scopo di questa tesi è necessaria l'esecuzione di un modulo WebAssembly, che sfrutti anche API di WASI, all'interno di un **server** sviluppato nel linguaggio di programmazione Rust. Al momento le soluzioni possbili per un embedding di questo tipo, sono diverse (Wasmtime, Wasmer etc.), ma non così numerose. Tra queste si è optato per Wasmtime, un progetto sviluppato della Bytecode Alliance (gli stessi sviluppatori di WASI).
Wasmtime è un **runtime environment standalone**, ottimizzato per WebAssembly e WASI. Permette di eseguire codice WebAssembly all'esterno del browser in applicazioni di qualsiasi grandezza e scritte in molteplici linguaggi differenti (Rust, C, Python, Go, Ruby, Bash etc.).
Proprio come in WebAssembly, un obiettivo di Wasmtime è quello di eseguire codice non attendibile in sicurezza all'interno di una sandbox. Inoltre Wasmtime implementa le API di WASI per l'accesso al file system, in modo che un programma possa accedere solo ai file e alle directory per i quali ha le capability necessarie. [14]

2.2.4 Rust

Come è già stato menzionato, per l'esecuzione di moduli WebAssembly lato server, si è scelto il linguaggio di programmazione Rust. Rust è un linguaggio compilato, multi-paradigma, sviluppato nel 2010 da Mozilla in collaborazione con la comunità open-source.[15] Ha guadagnato sempre più popolarità grazie alle prestazioni offerte, alla sicurezza e alla possibilità di avere controlli di basso livello, evitando però errori di programmazione comuni. Le caratteristiche più importanti di Rust sono:

- **Memory safety**: il concetto di ownership (proprietà) ci assicura che un programma non conterrà bug relativi alla memoria come **null pointer, dangling pointer,** o data races;
- **Concorrenza**: Rust fornisce supporto built-in per programmi concorrenti;
- **Astrazione a costo zero**: è possibile scrivere codice espressivo e leggibile di alto livello con prestazioni equivalenti allo stesso codice, ma di più basso livello;
- **Ecosistema**: le librerie e gli strumenti (**crates**), sono facilmente integrabili all'interno dei progetti, grazie anche al package manager cargo che risolve automaticamente le dipendenze;
- **Multipiattaforma**: Rust supporta svariate architetture e sistemi operativi, rendendolo un ottima scelta per una vasta gamma di applicazioni, dai sistemi embedded, ai web server;
- **Forte tipizzazione**: La tipizzazione contribuisce alla diminuzione di crash a runtime, catturando molti errori già a tempo di compilazione;
- **WebAssembly**: Rust consente la compilazione di codice in moduli WebAssembly, consentendo l'esecuzione di codice nativo all'interno del browser o di altri environment WebAssembly;

Grazie a queste peculiarità Rust si presta ad una grande varietà di casi d'uso, come la programmazione di sistema, lo sviluppo di web application prestazionali grazie per esempio, ai framework Rocket o Actix, lo sviluppo di giochi ma anche di applicazioni per sistemi con poche risorse a disposizione (IoT e sistemi embedded).

2.2.5 Esempi

System Call WASI

Per capire meglio che cos'è realmente una system call WASI seguono una serie di esempi riguardanti funzioni che saranno poi invocate anche dal modulo scritto per il prototipo di questa tesi.[16] Si parte esaminando la funzione **path_open** che consente l'apertura di un file o directory.
path_open(fd: fd, dirflags: lookupflags, path: string, oflags: oflags, fs_rights_base: rights, fs_rights_inheriting: rights, fdflags: fdflags) -> Result<fd, errno>
Parametri:

- fd: **fd**;
- dirflags: **lookupflags** Flag che indica il metodo di risoluzione del path (se il path è un link simbolico esso verrà espanso);
- path: **string**;
- oflags: **oflags** Flag booleani per indicare il metodo di apertura (creat, directory, excl, trunc);
- fs_rights_base: **rights** Permessi iniziali del file descriptor appena creato. Sono permessi che si applicano esclusivamente ad operazioni fatte usando il file descriptor stesso. Può capitare che l'implementazione restituisca un file descriptor con permessi più restrittivi di quelli specificati, se tali permessi non sono applicabili al tipo di file aperto;
- fs_rights_inheriting: **rights** Permessi che si applicano a file descriptor derivati da quello iniziale;
- fdflags: **fdflags** Flag per il file descriptor restituito (append, dsync, noblock, rsync, sync);

Risultato:

- ok: **fd**
- err: **errno** Codice errore restituito da una funzione (noent, perm, notsock etc.)

Per capire meglio il funzionamento delle capability, si citano alcuni permessi (**rights**) applicabili a un file descriptor:

- fd_read: **bool** Permesso di invocare le funzioni **fd_read** e **sock_recv**
- fd_write: **bool** Permesso di invocare **fd_write** e **sock_send**
- path_create_directory: **bool** Permesso di invocare **path_create_directory**
- sock_accept: **bool** Permesso di invocare **sock_accept**

Si procede ad esaminare la funzione **fd_read**, simile alla **readv** di POSIX. **fd_read(fd: fd, iovs: iovec_array) -> Result<size, errno> Parametri**:

- fd: **fd** File descriptor del file che è stao aperto;
- iovs: **iovec_array)** Lista di *scatter/gather vector* in cui contenere i dati letti;

Risultato:

- ok: **size** Numero di byte letti.
- err: **errno** Codice errore restituito da una funzione (noent, perm, notsock etc.)

Esempio completo

Concludiamo la trattativa su WASI con un esempio completo partendo da un semplice codice Rust che sfrutti il filesystem.
Esso sarà poi compilato in WebAssembly ed infine eseguito da linea di comando grazie al runtime environment Wasmtime. Il codice che segue tenta di:

- Creare un file con nome uguale al valore della variabile "outfile";
- Scrivere in tale file il valore della variabile "greeting";
- Aprire il file appena creato;
- Leggere il contenuto del file e stamparlo su standard output;

```
1        use std::fs;
2        use std::io::{Read, Write};
3
```

```rust
fn main() {
    let to_write = "Esempio WASI!";
    let file_name = "esempio.txt";
    let mut output_file = fs::File::create(file_name)
    .expect(&format!("Error creating {}", file_name));

    output_file.write_all(to_write.as_bytes())
    .expect(&format!("Error writing: {}", file_name));

    let mut input_file = fs::File::open(file_name)
    .expect(&format!("Error opening {}", file_name));

    let mut input = String::new();
    input_file.read_to_string(&mut input)
    .expect(&format!("Error reading: {} ", file_name));

    println!("Read in from file: {}", input);
}
```

Listato 2.1: Esempio in Rust

A questo punto è necessario compilare il file con *compilation target* WebAssembly. Per fare ciò eseguiamo il comando:

```
cargo build --release --target wasm32-wasi
        Compiling esempio_wasi v0.1.0 (.\esempio_wasi)
         Finished release [optimized] target(s) in 1.09s
```

In questo modo verrà creato il file binario esempio_wasi.wasm all'interno della directory **target/wasm32-wasi/release/**.

Se convertiamo tale file nell'equivalente testuale esempio_wasi.wat possiamo notare l'utilizzo delle API di WebAssembly System Interface.

```
...
(import "wasi_snapshot_preview1" "fd_filestat_get" (func (;0;) (type 2)))
(import "wasi_snapshot_preview1" "fd_seek" (func (;1;) (type 6)))
```

```
(import "wasi_snapshot_preview1" "path_open" (func (;2;) (type 7)))
(import "wasi_snapshot_preview1" "fd_read" (func (;3;) (type 8)))
(import "wasi_snapshot_preview1" "fd_write" (func (;4;) (type 8)))
(import "wasi_snapshot_preview1" "environ_get" (func (;5;) (type 2)))
(import "wasi_snapshot_preview1" "fd_close" (func (;6;) (type 9)))
(import "wasi_snapshot_preview1" "fd_prestat_get" (func (;7;) (type 2)))
(import "wasi_snapshot_preview1" "proc_exit" (func (;8;) (type 0)))
...
```

Listato 2.2: Le API importate di WASI

A questo punto possiamo eseguire il file binario grazie a wasmtime. Si noti però che l'esecuzione viene interrotta non appena si tenta di eseguire un'operazione sul filesystem.

```
wasmtime target/wasm32-wasi/release/esempio_wasi.wasm
    thread 'main' panicked at 'Error creating esempio.txt: Custom
       { kind: Uncategorized, error: "failed to find a pre-opened
       file descriptor through which \"esempio.txt\" could be
       opened" }', src\main.rs:8:6
    Error: failed to run main module
       `target/wasm32-wasi/release/esempio_wasi.wasm`

    Caused by:
        0: failed to invoke command default
        1: error while executing at wasm backtrace:
  ...
```

Ciò accade perchè l'host non ha conferito al programma le capability necessarie all'esecuzione di operazioni (come ad esempio la create), su un determinato file descriptor (quello di esempio.txt).

Per far sì che il programma possa operare sul file in questione, si deve aggiungere l'opzione **"- -dir=."** al comando wasmtime. In questo modo, wasmtime aprirà la directory corrente prima dell'esecuzione del modulo Wasm e la renderà disponibile al programma sotto forma di capability, per eseguire operazioni sui file contenuti in essa.

```
wasmtime --dir=. target/wasm32-wasi/release/esempio_wasi.wasm
Read in from file: Esempio WASI!
```

2.3 Node.js

Node.js è un **runtime environment** che consente l'esecuzione di codice JavaScript server-side e fornisce una piattaforma versatile ed efficiente per lo sviluppo di applicazioni web concorrenti e scalabili.

2.3.1 Funzionalità cardine

Node.js sta acquisendo sempre più popolarità negli anni grazie alle sue peculiarità che lo rendono un'ottima scelta per una grande varietà di applicazioni. In particolare Node.js si è distinto per:

- **Architettura Event-Driven**: viene usato un modello di I/O asincrono, non bloccante che gestisce in maniera efficiente richieste concorrenti per operazioni di I/O;
- **V8 JavaScript Engine**: Node.js è costruito su questo engine, conosciuto per la sua velocità ed efficienza, paragonabile a quella del codice nativo;
- **NPM**: il core di Node è stato progettato per essere piccolo e snello; i moduli che fanno parte del core si focalizzano su protocolli e formati di uso comune. Per ogni altra cosa, si usa npm NPM un ecosistema di librerie open-source in costante crescita. Gli sviluppatori in questo modo, possono sfruttare migliaia di moduli già pronti, per la scrittura di codice;
- **Thread singolo non bloccante**: in caso di I/O bloccante, un server "tradizionale" usa più thread per limitare l'attesa, ma comunque, ogni thread passa la maggior parte del tempo in attesa di I/O. Andare verso altissimi numeri di thread introduce overhead di context switching e aumenta significativamente l'utilizzo di memoria. Al contrario, in Node.js, è presente un solo thread che fa continuamente fetching di eventi da una coda. In questo modo diventa possbile gestire efficientemente molte richieste concorrenti, senza l'overhead di context-switching dei tradizionali server multi-threaded;
- **Multipiattaforma**: Node.js può eseguire sia su Windows, che su macOS, che su molte distribuzioni di Linux, garantendo così flessibilità e portabilità;

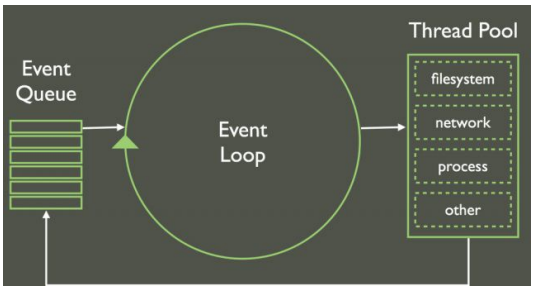

Figura 2.5: Architettura event-driven di Node.js

2.3.2 Casi d'uso

Applicazioni Web

Node.js risulta spesso una buona scelta per applicazioni che debbano gestire un alto numero di richieste concorrenti, che implicano operazioni di I/O, sia da filesystem, che da un database.

Microservizi

Node.js viene utilizzato anche per lo sviluppo di architetture a microservizi, consentendo la scrittura di piccoli servizi specializzati, ben integrati tra loro. Grazie all'architettura event-based, al thread singolo e alle performance elevate node.js si è ben presto rivelato un'ottima alternativa anche in quest'ambito. Si sottolinea inoltre, che l'approccio a microservizi può migliorare in maniera selettiva la scalabilità e semplificare ampiamente il deployment di applicazioni complesse.

Real Time Application

Node.js eccelle anche in applicazioni come giochi online, strumenti collaborativi, condivisione di documenti, chat online e molti altri. Tale tipologia di applicazione deve gestire numerose richieste utente che eseguiranno molte operazioni pesanti di I/O. Se si utilizzasse un thread per ogni richiesta, si arriverebbe molto in fretta ad un sovraccarico del server.

2.4 Confronto tra tecnologie

Al giorno d'oggi può rivelarsi cruciale la scelta di un determinato approccio di sviluppo per la buona riuscita di un progetto. Si procede quindi con un'analisi comparativa tra i due approcci confrontati durante questa tesi, che aiuterà a capire quando uno potrebbe essere preferibile rispetto all'altro.

2.4.1 Prestazioni

Uno degli aspetti chiave di WebAssembly è l'abilità di ottenere prestazioni paragonabili all'esecuzione di **codice nativo**. Può inoltre eseguire codice in parallelo con thread multipli. Rust inoltre consente di controllare in maniera precisa la **memoria e le risorse di sistema**, permettendo in questo modo la scrittura di codice altamente ottimizzato che minimizza l'uso di risorse e massimizza la velocità di esecuzione. La combinazione di Rust e WebAssembly, risulta quindi un'ottima scelta quando le prestazioni sono l'elemento chiave di un progetto.
D'altro canto Node.js, grazie al suo approccio event-driven non bloccante e all'utilizzo del JavaScript engine V8, consente di gestire un **numero elevato di connessioni** in maniera efficiente e di eseguire operazioni di I/O in modo asincrono. Bisogna però fare attenzione all'esecuzione di task CPU-intensive che potrebbero rimanere bloccati per molto tempo nella coda degli eventi.
Sia Node.js che Rust in combinazione con WebAssembly, promettono quindi ottime performance, ma sotto punti di visti totalmente differenti. Nella scelta tra i due è quindi fondamentale valutare la **natura di un applicazione**, cercando di capire se essa sia più orientata all'utilizzo della CPU o del filesystem.

2.4.2 Sicurezza

Sia WebAssembly che Rust garantiscono in modi diversi un elevato livello di sicurezza. In particolare si parla di **memory safety** sia per l'uno che per l'altro (2.1.5), ma anche di type safety e di esecuzione confinata in una sandbox per WASM. Sostanzialmente costituiscono un ambiente sicuro e robusto per lo sviluppo e l'esecuzione di software.
Node.js invece può presentare qualche problema derivante dalla tipizzazione debole di JavaScript. Può accadere spesso di incontrare errori a runtime per conversioni

inaspettate, o per accesso a variabili *undefined*. Viene richiesta quindi una buona diligenza nella scrittura del codice per far sì che gli errori siano gestiti correttamente evitando crash dell'applicazione.
Risulta evidente che Node.js non presenta *by design* lo stesso livello di sicurezza offerto da Rust con WebAssembly.

2.4.3 Facilità di Sviluppo

I vantaggi di Rust in termini di sicurezza e prestazioni, come precedentemente illustrato, sono indubbiamente attraenti. Tuttavia, per tradurre tali vantaggi in risultati concreti, potrebbe essere necessario un notevole sforzo iniziale da parte degli sviluppatori. La forte tipizzazione di Rust, combinata con la rigorosità del suo compilatore, potrebbe rappresentare una sfida considerevole per coloro che si avvicinano al linguaggio per la prima volta. Questi due fattori contribuiscono a rendere ancora più ripida una curva di apprendimento già impegnativa, derivata dalle caratteristiche uniche di Rust.
D'altra parte, un primo approccio a Node.js può risultare notevolmente diverso. L'utilizzo di JavaScript, un linguaggio ampiamente utilizzato, insieme al package manager NPM, può accelerare significativamente il processo di sviluppo.
La scelta tra questi due approcci può influenzare notevolmente le tempistiche di sviluppo complessive. Tuttavia, è importante sottolineare che la decisione dovrebbe essere guidata dai requisiti specifici del progetto e non dalla semplice volontà di accelerare il processo. Inoltre, va considerato che, nonostante le sfide iniziali, l'adozione di Rust potrebbe portare a un codice più robusto e affidabile nel lungo termine.

2.4.4 Scalabilità ed Espandibilità

Rust offre diverse possibilità per far fronte ad esigenze di scalabilità. Supporta la programmazione asincrona e grazie alle caratteristiche di esecuzione parallela di WebAssembly è possibile sfruttare appieno anche processori multi-core. Rust permette lo sviluppo di microservizi e la loro integrazione con Wasm/WASI.
Node.js riesce a scalare orizzontalmente grazie a tecniche di *clustering* e *load balancing*. Ciò però è efficace soprattuto per task I/O bound. A causa dell'utilizzo di un singolo thread in Node.js, la scalabilità per task CPU-intensive è notevolmente limitata. Si potrebbere scalare in verticale un singolo server, aumentando le risorse disponibili, ma è

importante notare che sono presenti limiti pratici a quest'approccio. Infine anche in Node.js è possbile implementare un'architettura a microservizi.

2.5 Conclusioni preliminari

La comprazione appena effettuata, ha ulteriormente enfatizzato l'importanza di condurre un'attenta analisi sui requisiti dell'applicazione che si sta progettando. In particolare, l'approccio basato su Rust combinato con WebAssembly si distingue per le sue ottime prestazioni, l'alto livello di sicurezza offerto e la sua capacità di risolvere efficacemente problemi legati alla scalabilità.
D'altra parte, Node.js si contraddistingue per la sua facilità di sviluppo e la notevole scalabilità nelle operazioni fortemente orientate all'I/O.
Nel capitolo successivo si cercherà di valutare se tali differenze abbiano un impatto significativo all'interno di un'applicazione reale, cercando di misurare l'influenza di ciascuna tecnologia in un contesto in cui le operazioni sono prevalentemente CPU-intensive.

Capitolo 3

Prototipo per esecuzione efficiente di codice Wasm con tecniche di Image Processing

Nel capitolo precedente sono state esaminate le differenze sostanziali tra un'approccio basato su Rust in combinazione con WebAssembly e uno basato su Node.js. In questo capitolo si presenterà il prototipo sviluppato con l'obiettivo di comprendere l'impatto di tali differenze in un'applicazione pratica.

3.1 Descrizione dell'applicazione

Il prototipo sviluppato è un'applicazione dedicata all'elaborazione digitale di immagini, concepita per simulare un contesto realistico in cui le operazioni richiedono una considerevole quantità di elaborazioni da parte della CPU.
Dato il limitato tempo disponibile, non è stato possibile esplorare in dettaglio il vasto campo del *digital image processing*. Pertanto, sono state impiegate librerie preesistenti in entrambi i linguaggi, evitando di immergersi eccessivamente nella programmazione di basso livello.
L'architettura dell'applicazione segue un modello client-server in entrambe le implementazioni. Il cliente ha il compito di fornire i file da elaborare insieme alle relative

specifiche sulle modifiche da apportare. Il server eseguirà le modifiche richieste e restituirà al cliente il percorso della nuova immagine, pronta per il download.
Nel processo di selezione delle possibili modifiche da apportare, è stato essenziale individuare due librerie nei rispettivi linguaggi utilizzati. Successivamente, per garantire uniformità nelle opzioni di modifica disponibili, sono state estratte le seguenti funzionalità comuni:

- ridimensionamento;
- rotazione di 90°;
- ribaltamento in orizzantale;
- conversione in bianco e nero;
- regolazione del contrasto;
- modifica della luminosità;

Tali operazioni sono state selezionate poiché rappresentano funzionalità frequentemente utilizzate anche da utenti comuni, oltre a caratterizzarsi per la loro eterogeneità. Alcune di queste coinvolgono esclusivamente la manipolazione dei pixel, come ad esempio la rotazione e il ribaltamento, mentre altre, come la conversione in scala di grigi o la modifica del contrasto/luminosità, comportano modifiche dirette ai pixel stessi.

Figura 3.1: Esempio di immagine a cui sono state apportate tutte le elaborazioni specificate.

3.2 Implementazione in Rust e Wasm/WASI

Per quanto riguarda l'implementazione si è deciso di partire dal prototipo sviluppato in Rust poiché rappresentava l'aspetto più innovativo e richiedeva un considerevole impegno in termini di tempo.
Si è resa inoltre necessaria la ricerca di un framework che consentisse la creazione di un web server per la gestione delle richieste utente.
La scelta è ricaduta su actix-web, un web framework potente ed estremamente veloce per Rust.
Il client è costituito da una semplice pagina HTML contenente un form per il caricamento delle immagini e due riquadri che mostrano l'immagine pre e post modifiche. Tale pagina eseguirà una richiesta AJAX al server, inviando il file da elaborare e le relative specifiche sulle modifiche.
Terminata l'elaborazione il server restituirà il percorso della nuova immagine pronta per essere scaricata.

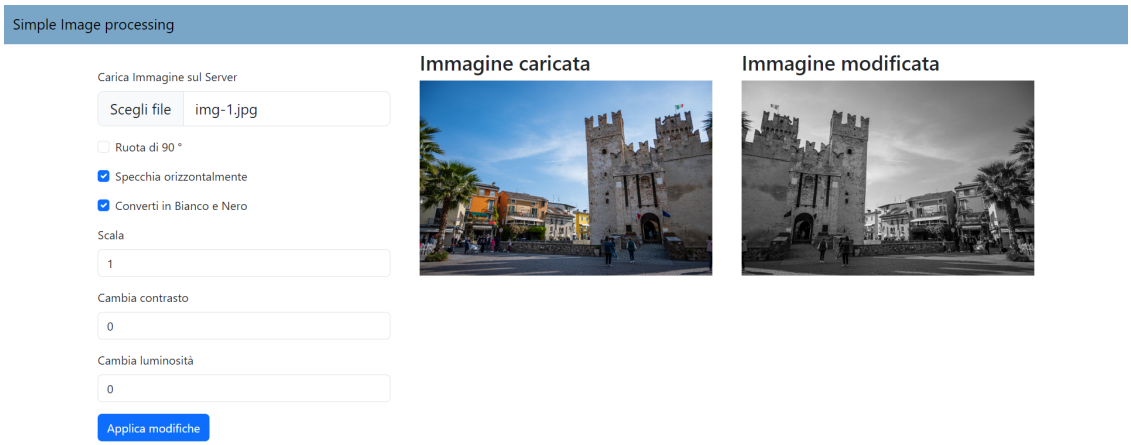

Figura 3.2: View nel browser del client

3.2.1 Actix-Web

Lo studio del funzionamento del framework Actix-Web ha costituito un punto focale durante la fase iniziale dell'implementazione.

Tale framework si è dimostrato estremamente flessibile ed adatto per lo sviluppo del prototipo in questione. Grazie all'impiego di **extractors**, di **handlers** e di altre funzionalità integrate, la gestione di richieste HTTP è risultata intuitiva e diretta.

Al fine di garantire un codice più ordinato e manutenibile è stata presa la decisione di strutturare l'applicazione in diversi file. Tra questi, il file primario è **main.rs**, che svolge un ruolo centrale nell'architettura complessiva.

Configurazione server

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(move || {
        App::new()
            .route("/", web::get().to(server::handlers::index))
            .route("/upload", web::post().to(server::handlers::upload))
```

```
7               .service(fs::Files::new("/script", "./src/static/"))
8               .service(fs::Files::new("/img", "./img/"))
9               .app_data(state.clone())
10      })
11      .bind("127.0.0.1:8080")?
12      .run()
13      .await
14  }
```

Listato 3.1: Porzione del file main.rs

Nel frammento di codice fornito, viene presentata la creazione di un HttpServer e un'istanza di App tramite il framework Actix-Web. In Actix-Web, ogni server è costruito attorno a un'istanza di App, che consente di configurare le "regole di routing" per risorse di vario tipo, registrare servizi HTTP e gestire stato di livello applicazione.
Nel caso specifico, vengono configurate due regole di routing:

- La prima regola gestisce le richieste dirette alla **home** del sito: in questo caso verrà semplicemente restituita la pagina index.html presente nella directory "static".
- La seconda regola gestisce richieste all'endpoint **upload**, per le quali è necessario ricevere un'immagine e le relative modifiche da apportare all'interno del modulo WebAssembly. Successivamente, verrà restituito il percorso della nuova immagine elaborata.

Si sottolinea che entrambe le regole di routing mappano le richieste a funzioni presenti all'interno del file **handlers.rs** nel modulo server.
Successivamente sono stati registrati due servizi HTTP per garantire l'accesso a risorse statiche: script Javascript e immagini elaborate dall'applicazione.
Infine, dopo aver configurato il metodo di dispatching delle richieste ed aver messo a disposizione le risorse necessarie ai client, si è passati alla scrittura del codice nel file **handlers.rs**. Tale file, come suggerito dal nome, contiene gli handler delle richieste HTTP, con la conseguente esecuzione di codice WebAssembly per richieste di modifica di immagini.

```rust
#[derive(MultipartForm)]
pub struct ImageUpload {
    image: TempFile,
    scala: Text<f32>,
    contrasto: Text<f32>,
    luminosita: Text<i32>,
    ruota: Text<bool>,
    specchia: Text<bool>,
    bw: Text<bool>
}
#[derive(Serialize, Deserialize)]
pub struct Editings{
    scala: f32,
    ...
}
pub async fn index() -> HttpResponse {
    HttpResponse::Ok().content_type("text/html")
        .body(include_str!("../static/index.html"))
}

pub async fn upload(form: MultipartForm<ImageUpload>) -> HttpResponse {
    let time = SystemTime::now().duration_since(SystemTime::UNIX_EPOCH);
    let filepath = format!("img/uploaded/{:?}_{}", time,
        form.0.file_name.as_str());
    match form.0.image.file.persist(filepath) {
        Ok(_) => {
            let editings = Editings{
                scala : form.0.scala.0,
                ...
            };
            edit(editings)
        },
```

```
33          Err(e) => {
34              HttpResponse::InternalServerError().finish()
35          },
36      }
37  }
```

Listato 3.2: Operazioni principali presenti nel file handlers.rs

Nel codice quì presentato, si possono facilmente notare i due endpoint del prototipo: la funzione **index()** per richieste alla pagina home e la funzione **upload()** per richieste di elaborazione di immagini.
Per quanto riguarda la prima tipologia si risponde ai client semplicemente fornendo il file HTML statico "index.html".
Al contrario la gestione delle richieste di upload è notevolmente più articolata e per questo motivo coinvolge una funzione di supporto denominata **edit()**. Sarà questa funzione che si occuperà dell'istanziazione del modulo WebAssembly/WASI e della sua esecuzione.
Si sottolinea anche il parametro ricevuto dalla funzione "upload": un **MultipartForm<ImageUpload>**. Il framework Actix-Web, con il supporto del crate actix-multipart semplifica notevolmente la gestione di richieste provenienti da form html, anche in presenza di campi di input di tipo "file".
Ogni campo definito nella struttura "ImageUpload" corrisponde a un parametro proveniente dalla richiesta HTML e viene automaticamente popolato prima dell'invocazione dell'handler, consentendo un accesso immediato ai valori inviati al server.
In sintesi la funzione **upload()** rende persistente il file temporaneo ricevuto ed inoltre crea un ulteriore *struct* di supporto (**Editings**) che verrà sfruttata dalla funzione **edit()** per velocizzare il successivo accesso alle elaborazioni da effettuare.
Va sottolineato che "edit()" viene invocata solo se il file è stato reso persistente senza errori. Infatti, tale funzione si occupa dell'istanziazione del modulo WebAssembly e non ha senso tentare di eseguire modifiche su un file che non è stato salvato correttamente.

3.2.2 Integrazione modulo Wasm/WASI

Come precedentemente indicato, per l'esecuzione di codice WebAssembly, si è deciso di utilizzare il runtime environment **Wasmtime**. La sua integrazione all'interno di un'applicazione Rust è resa possibile primariamente grazie ai crate **wasmtime**, **wasmtime-wasi** e **wasi-common**.

Scambio di dati

Prima ancora di iniziare l'implementazione è stato fondamentale trovare un modo per **scambiare dati** tra il modulo WebAssembly e l'host.
Per soddisfare i requesti imposti in questa tesi, è necessario che il modulo WebAssembly riceva tutte le elaborazioni da effettuare su un'immagine e il nome del file da modificare, ma come introdotto in sezione 2.1.3, una funzione WebAssembly può ricevere solamente valori di **tipo numerico**.
Analizzando le API WebAssembly e WASI sono emerse diverse soluzioni a questo problema. Tra queste l'utilizzo della memoria lineare, la scrittura su file, la modifica di variabili d'ambiente o la comunicazione attraverso standard input. Tuttavia, alcuni di questi approcci potrebbero risultare complessi da implementare e considerando che per l'applicazione sviluppata sarebbe stato sufficiente scambiare una stringa per ottenere tutte le informazioni necessarie, si è scelto di adottare la comunicazione tramite **standard input**.
Nello specifico, per consentire lo scambio di dati tra host e guest, verrà implementato un protocollo composto dalla seguente sequenza di operazioni:

- Serializzazione in formato JSON della struct "Editing" contenente tutte le elaborazioni e il nome del file da modificare;
- Inserimento sullo standard input del modulo Wasm, della stringa ottenuta dalla serializzazione;
- Lettura della stringa da standard input all'interno del modulo, sfruttando le API messe a disposizione da WebAssembly System Interface;
- Deserializzazione in una Struct Editing equivalente a quella di partenza;

Si noti che, se necessario, questo protocollo potrebbe essere applicato anche per ottenere strutture dati in output dal modulo WebAssembly.

Condizioni necessarie per l'esecuzione

Prima di poter eseguire un modulo tramite Wasmtime, sono necessarie diverse operazioni preliminari.
Dopo aver serializzato la struttura dati e predisposto una **ReadPipe** (crate wasi-common) per mettere a disposizione la stringa serializzata su standard input, la prima operazione necessaria è la creazione dell'**Engine** Wasmtime.
Esso rappresenta un contesto globale per la compilazione e l'esecuzione di moduli Wasm, che nel nostro specifico caso adotterà la configurazione di default.
Si procederà poi con l'istanziazione di un **wasmtime::Linker**. Il linker faciliterà l'istanziazione del modulo Wasm, risolvendo le diverse import (tra cui quelle per le syscall WASI).
Non bisogna poi dimenticare, che a causa dell'architettura di WASI, sarà possibile accedere ed utilizzare i file presenti in una certa directory, solamente se al programma sono state fornite le **capabilities** necessarie.
Per ottenere le capabilities per operare sulle immagini caricate dagli utenti, è necessario aprire la cartella "img" prima dell'istanziazione del modulo WASI.

```
pub fn edit(editing : Editings) -> HttpResponse {
    let serialized_input = serde_json::to_string(&editing);
    let stdin = ReadPipe::from(serialized_input);

    let engine = Engine::default();

    let mut linker: Linker<WasiCtx> = Linker::new(&engine);
    wasmtime_wasi::add_to_linker(&mut linker, |s| s);
    let  image_directory = Dir::open_ambient_dir("img",
    ambient_authority());
    ...
}
```

Listato 3.3: File handlers.rs: operazioni preliminari

A questo punto è possibile creare e configurare un contesto WASI tramite la struct **wasmtime_wasi::WasiCtxBuilder**.

Tramite tale oggetto si specifica che lo standard input sarà prelevato dall'oggetto contenente la serializzazione, lo standard output e lo standard error saranno ereditati dalla macchina host ed infine si fornisce una directory precedentemente aperta, che sarà disponibile al percorso "img".
Sfruttando il contesto WASI è ora possibile la creazione dello **Store** Wasmtime, l'oggetto designato per l'effettiva istanziazione del modulo WebAssembly e che successivamente conterrà tutte le funzioni, la memoria, le tabelle e lo stato interno del programma.

```
1    let builder = WasiCtxBuilder::new()
2    .stdin(Box::new(stdin.clone()))
3    .inherit_stdout()
4    .inherit_stderr()
5    .preopened_dir(image_directory,"img");
6    let wasi = builder.build();
7
8    let mut store = Store::new(&engine, wasi);
```

Listato 3.4: File handlers.rs: creazione di contesto WASI e Store

Ora risulta possibile la creazione del **Module** WebAssembly e il suo collegamento con il Linker per poi ottenere, tramite quest'ultimo, un'**istanza** relativa al Module e allo Store specificati nel metodo *linker.instantiate()*.

```
1    let module = Module::from_file(&engine,
     "src/server/image_proc_module.wasm");
2    linker.module(&mut store, "", &module)
3    let instance = linker.instantiate(&mut store, &module);
```

Listato 3.5: File handlers.rs: istanziazione modulo Wasm

Esecuzione del modulo WebAssembly

Avendo configurato Store e contesto WASI, il programma possiede già tutti gli argomenti necessari per il corretto funzionamento e l'unico passo rimanente consiste nell'effettiva esecuzione della funzione del modulo WebAssembly.
Per fare ciò è necessario ottenere un'istanza di **wasmtime::Func** tramite l'operazione

get_typed_func() sull'istanza WebAssembly. A questo punto l'invocazione della funzione richiesta è finalmente possibile grazie al metodo *Func::call()*.
Ad esecuzione terminata verrà eliminato lo store dalla memoria e restituito il percorso dell'immagine modificata al cliente.

```
1    let instance_main = instance.get_typed_func::<(), ()>(&mut store,
     "_start");
2    instance_main.call(&mut store, ());
3    drop(store);
4    HttpResponse::Ok()
5    .content_type("text/plain")
6    .body(e.modified_file_path)
```

Listato 3.6: File handlers.rs: invocazione funzione _start presente nel modulo Wasm

Si noti che in questi esempi di codice non è presente alcuna gestione degli errori. Tuttavia nel prototipo sviluppato, l'utente finale otterrà il nuovo percorso del file, solo nel caso in cui ciascuna delle operazioni illustrate sarà andata a buon fine.

3.2.3 Modulo WebAssembly/WASI

Il modulo WebAssembly risulta a questo punto piuttosto semplice.
Esso si occupa infatti della lettura dei parametri da **standard input** e della loro deserializzazione in una struct Editings.
Successivamente vengono utilizzati i metodi forniti dal crate **image** di Rust per aprire l'immagine ricevuta, modificarla secondo le specifiche dell'utente e salvarla nel percorso specificato.[17]

```rust
#[derive(Serialize, Deserialize)]
pub struct Editings{
    scala: f32,
    ...
}
fn main() {
    let mut serialized_params = String::new();
    std::io::stdin().read_to_string(&mut serialized_params);
    let editings : Editings = serde_json::from_str(&serialized_params);
    let mut img = image::open(editings.filepath)
    if editings.scala != 0.0 {
        let new_width = (img.width() as f32) * editings.scala;
        let new_heigth = (img.height() as f32) * editings.scala;
        img = img.resize(new_width as u32, new_heigth as u32,
    image::imageops::FilterType::Nearest);
    }
    if editings.ruota {
        img = img.rotate90();
    }
    ...
    img.save(editings.modified_filepath);
}
```

Listato 3.7: Codice Rust che successivamente verrà compilato in WebAssembly

Terminata l'esecuzione del modulo, il controllo ritornerà all'host che si occuperà dell'invio di una risposta adeguata al client:

- Se durante l'esecuzione del modulo Wasm tutte le operazioni sono terminate correttamente verrà restituito il percorso della nuova immagine generata;
- Altrimenti verrà restituito un messaggio di errore;

Terminata la scrittura del codice, è necessaria la compilazione per ottenere un modulo WebAssembly utilizzabile da wasmtime.
Ciò si può fare agovelmente grazie al package manager **cargo**, ed in particolare grazie al seguente comando:

```
cargo build --release --target wasm32-wasi
        Compiling image_proc_module v0.1.0 (.\image_proc_module)
            Finished release [optimized] target(s) in 1.21s
```

In questo caso risulta pressochè obbligatoria la presenza del flag **- - release**, in quanto diverse funzioni del crate image, sono estremamente lente se utilizzate in debug mode.
Infine si sottolinea che avendo svolto tutte le operazioni necessarie nel main, nel momento in cui l'host invocherà il metodo *get_typed_func()*, sarà necessario specificare la funzione denominata **_start**.

3.3 Implementazione in Node.js

Dopo aver terminato l'implementazione del prototipo mediante l'utilizzo di Rust e WebAssembly, si è proceduto con l'implementazione di un'applicazione dotata delle medesime funzionalità, questa volta utilizzando il runtime environment **Node.js**.
La seconda implementazione è risultata notevolmente più immediata. In particolare è stato possibile riutilizzare il client sviluppato precedentemente, senza necessità di un'analisi approfondita per l'integrazione di un modulo WebAssembly. Come verrà sottolineato in seguito, ciò ha permesso di ottenere un **codice** decisamente più **pulito** e di lunghezza proporzionata alla semplicità dell'applicazione sviluppata.
La ricerca si è dunque focalizzata sulla selezione di un moduli adeguati per la creazione di un'applicazione web in grado di supportare l'upload e l'elaborazione di immagini. La scelta è ricaduta sul web framework **express.js**, in combinazione con il middleware **multer** e la libreria **Jimp**.

3.3.1 Express.js

Express.js è un **web framework** che semplifica lo sviluppo di applicazioni web robuste e scalabili. Esso è stato riconosciuto come lo **standard de facto** in quest'ambito.
Il design di Express è relativamente minimale, tuttavia, grazie all'impiego di **plugin** e **middleware** come Multer, è in grado di fornire una vasta gamma di funzionalità.
Il framework inoltre rende più semplice il routing, gestendo richieste e risposte HTTP in modo semplice e diretto.
Per quanto concerne la nostra applicazione, dopo aver importato i moduli richiesti, è essenziale effettuare una configurazione breve ma precisa, per la gestione delle richieste necessarie e per la configurazione dell'upload dei file.

```
1    const express = require('express');
2    const app = express()
3    const port = 3000
4
5    app.use(express.static('./static'))
6    app.use(express.static('./img/modified'))
7
8    app.post('/upload', upload.single('image'), (req, res) => {
```

```
9       try{
10      ...
11      }
12      } catch (error){
13        console.log(error);
14        res.status(500).send('Error processing the file');
15      }
16    })
17
18    app.listen(port, () => {
19      console.log(`Server listening on port ${port}`)
20    })
```

Listato 3.8: Configurazione Express.js

Nello specifico viene creata un'applicazione Express tramite l'omonimo metodo *express()*. Successivamente tramite i metodi *express.use(express.static(...))* vengono rese accessibili ai client le directory "static"(contenente il file HTML della home e gli script correlati) e "img/modified" (necessaria per il download delle immagini elaborate dal server).
Il server viene poi messo in ascolto sulla porta 3000 e configurato per gestire richieste di tipo POST dirette all'endpoint "/upload".
A questo punto, per la corretta gestione del caricamento di immagini, risulta necessario configurare l'applicazione affinché sia in grado di ricevere richieste con Content-Type "multipart/form-data".

3.3.2 Multer

Come introdotto inizialmente, per la gestione di richieste provenienti da un form contenente campi di input di tipo file, è stato adottato il middleware **multer**.

```
1 const multer  = require('multer');
2 const storage = multer.diskStorage({
3     destination: 'img/uploaded/',
4     filename: (req, file, cb) => {
5     cb(null, file.originalname);
6     }
7 });
```

```
8  const upload = multer({ storage });
9  app.post('/upload', upload.single('image'), (req, res) => {
10     ...
11 })
```

Listato 3.9: Configurazione upload immagine

Una volta importato questo modulo, è necessario impostare, tramite il metodo *multer.diskStorage()*, la destinazione (per coerenza viene mantenuta la directory img/uploaded) e il nome del file appena caricato (viene utilizzato lo stesso nome del file fornito dall'utente).
In seguito attraverso l'utilizzo del metodo *multer()* otteniamo un'istanza, tramite la quale è possibile specificare, nei parametri della callback function per richieste all'endpoint upload, che un singolo file dovrà essere reso persistente, come precedentemente specificato nella variabile storage.

3.3.3 Jimp

Una volta terminata la configurazione del server tramite express.js, è possibile procedere con la modifica del file ricevuto.

```
1  app.post('/upload', upload.single('image'), (req, res) => {
2    try{
3      const uploadededFilePath =
           'img/uploaded/' + req.file.originalname
4      const newFileName =  Date.now() + req.file.originalname
5      const modifiedFilePath = 'img/modified/' + newFileName;
6
7      Jimp.read(uploadededFilePath, (err, img) => {
8        if (err) throw err
9        else{
10         img.scale(req.body.scala)
11            .rotate(req.body.ruota)
12            .mirror(req.body.specchia,false)
13            .contrast(req.body.contrasto/100)
14            .brightness(req.body.luminosita/100, function(){
15               if(req.body.bw) img.grayscale()
```

```
16                img.write(modifiedFilePath, function(){
17                  res.status(200).send(newFileName);
18                });
19            });
20        }
21      });
22
23    } catch (error){
24      res.status(500).send('Error processing the file');
25    }
26  }
```

Listato 3.10: Elaborazione immagine grazie ai metodi della libreria Jimp

Il file è aperto mediante l'impego del metodo **Jimp.read()**, il quale fornisce un'istanza utilizzabile per moodificare l'immagine secondo specifiche. Quest'ultime possono essere acquisite tramite l'oggetto req.body, passato alla callback function *app.post('/upload',...)*. Dopo aver apportato le opportune modifiche all'immagine, essa viene salvata nella cartella "img/modified" con un nuovo nome al fine di garantire univocità tra i vari file.
Infine, se tutte le operazioni precedenti vengono eseguite correttamente, verrà restituito all'utente il percorso dell'immagine modificata. In caso contrario, verrà restituito un messaggio di errore.

3.4 Metodologia di test

Per valutare le performance di ciascuna implementazione presentata, si è deciso di eseguire lo stesso **set di test** su entrambe.
In particolare è stata scelta un'elaborazione specifica tra quelle disponibili e durante la sua esecuzione sono stati misurati diversi parametri.
Per scegliere la tipologia di operazione si è empiricamente tentato di capire quale fosse, tra le elaborazioni implementate, quella più computazionalmente complessa. Eseguendo vari test su una stessa immagine e verificando la latenza è stato scelto il **resize** di un'immagine ed in particolare il resize del 10% (x0.9), al fine di manipolare un elevato numero di pixel e di ottenere un'immagine di dimensioni simili a quella iniziale.
In seguito sono state scelte le immagini di test e i parametri da misurare. Per quanto riguarda le immagini, si è scelto di utilizzare tre file con dimensione crescente:

- 1114 x 742 (0.82 Megapixel, 925 KiloByte);
- 2869 x 1912 (5.48 Megapixel, 4.63 MegaByte);
- 5578 x 3712 (20.7 Megapixel, 17.4 MegaByte);

Come parametri da misurare si è deciso di focalizzarsi su:

- Latenza per ogni richiesta;
- Utilizzo di CPU;
- Consumo di memoria;

3.5 Setup sperimentale

Per eseguire i test presentati è stato utilizzato un laptop con processore Intel Core i5-10210 (1.6 - 2.10 GHz), 8 GigaByte di Ram e sistema operativo Windows 11. Per quanto riguarda i software utilizzati, sono state scelte le ultime versioni al momento disponibili:

- Rust 1.71
- Wasmtime 11.0
- Node 18.17.1 (LTS)

Il seguente setup, pur non essendo propriamente tipico di un server, ha comunque permesso di osservare differenze notevoli tra i due apprrocci esaminati.

3.6 Valutazione delle prestazioni

Per ottenere un campione di risultati affidabile sono state eseguite 10 misurazioni per ogni immagine. In seguito sono stato calcolati il **valore medio** del campione, ottenuto come media aritmetica e la **deviazione standard**:

$$E[X] = \frac{1}{n}\sum_{i=0}^{n} x_i$$

$$\sigma_X = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - E[x])^2}$$

Successivamente verranno presentati i grafici per ciascuno dei parametri scelti, utilizzando rispettivamente Rust in combinazione con WebAssembly e Node.js. Ogni grafico conterrà i risultati per ognuna delle tre immagini utilizzati, consentendo così anche una facile analisi visiva.

3.6.1 Latenza

Il primo parametro misurato è stato quello della latenza.
Nello specifico si è analizzato il tempo intercorso tra l'invio di una richiesta di elaborazione e la successiva risposta contenente il percorso della nuova immagine elaborata.
Per quanto riguarda Rust si sono ottenuti i seguenti risultati sperimentali:

$$E[X]_{small} = 850ms, \qquad \sigma_X = 154.94ms$$
$$E[X]_{medium} = 1487ms, \qquad \sigma_X = 144.99ms$$
$$E[X]_{large} = 4212ms, \qquad \sigma_X = 169.95ms$$

Relativemente a Node.js invece:

$$E[X]_{small} = 704ms, \qquad \sigma_X = 79.19ms$$
$$E[X]_{medium} = 3081ms, \qquad \sigma_X = 240.58ms$$
$$E[X]_{large} = 11335ms, \qquad \sigma_X = 282.73ms$$

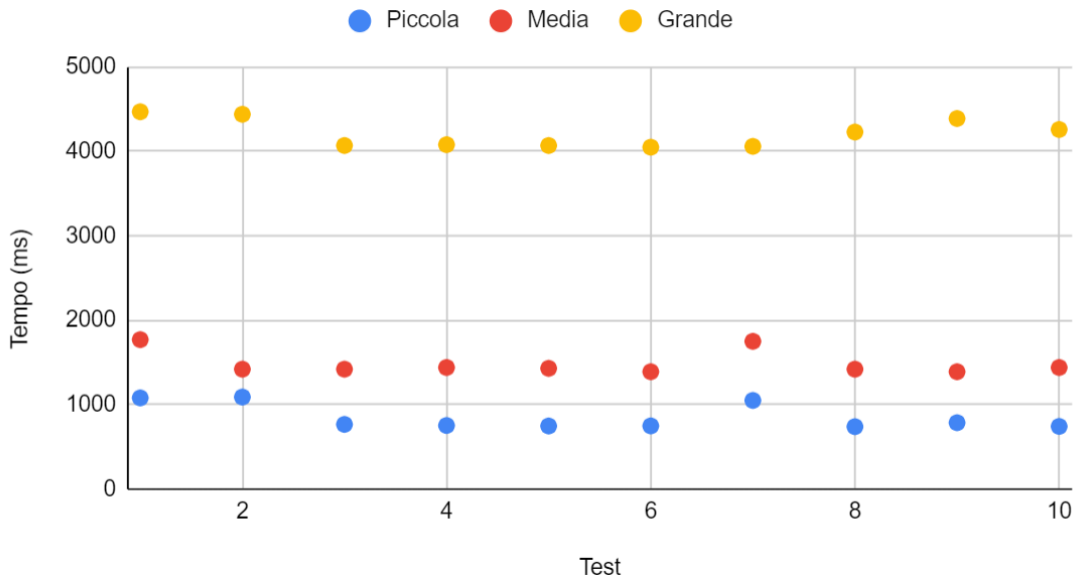

Figura 3.3: Latenza misurata utilizzando Rust in combinazione con WebAssembly.

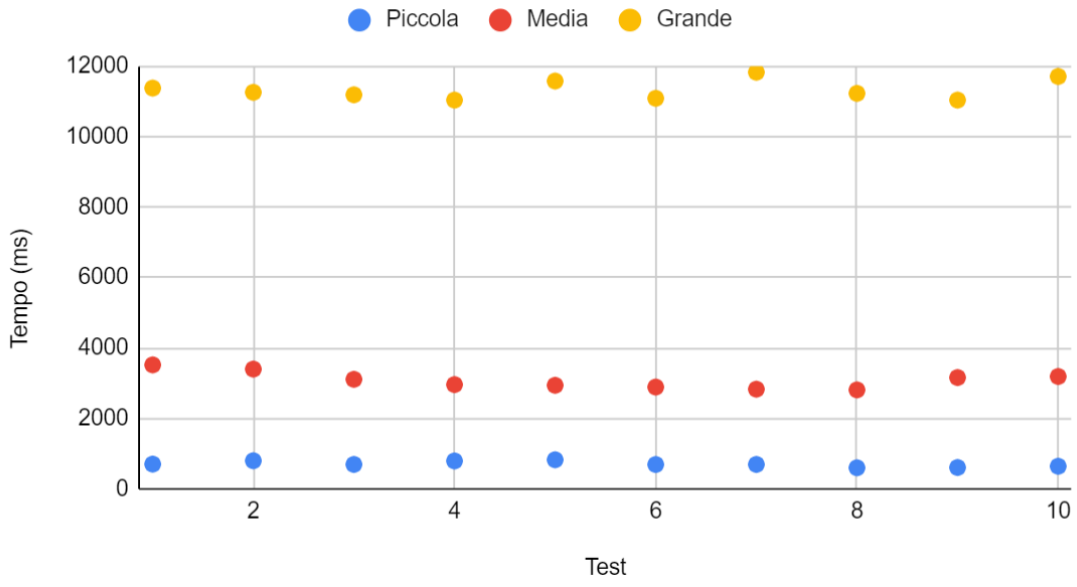

Figura 3.4: Latenza misurata utilizzando Node.js

Come si può notare il tempo di latenza risulta comparabile solamente durante l'elaborazione dell'immagine con dimensione minore. Durante l'elaborazione delle rimanenti due immagini, WebAssembly ha ottenuto performance notevolmente migliori, richiedendo circa un terzo del tempo necessario a Node.js per l'elaborazione

dell'immagine più grande.
È plausibile ipotizzare che, durante la modifica di immagini di piccole dimensioni, WebAssembly non riesca a mettere in evidenza i suoi vantaggi, soprattutto a causa dell'overhead introdotto dalla creazione di un'istanza del modulo e dalle operazioni correlate.
Tuttavia, quando il numero di operazioni a carico della CPU aumenta, emergono sia la tanto citata "velocità di esecuzione pari a quella del codice nativo" di WebAssembly che il bottleneck dovuto all'approccio event-driven di Node.js

3.6.2 Utilizzo CPU

In seguito è stato misurato l'utilizzo della CPU per ciascuno dei due approcci durante l'esecuzione di una richiesta, utilizzando le medesime immagini e impostazioni precedentemente indicate.
Anche in questo secondo test si è partiti dall'approccio basato Rust e sono stati ottenuti i seguenti risultati sperimentali:

$$E[X]_{small} = 16.09\%, \qquad \sigma_X = 3.32\%$$
$$E[X]_{medium} = 55.35\%, \qquad \sigma_X = 3.64\%$$
$$E[X]_{large} = 82.44\%, \qquad \sigma_X = 1.93\%$$

Relativemente a Node.js, invece:

$$E[X]_{small} = 20.87\%, \qquad \sigma_X = 6.89\%$$
$$E[X]_{medium} = 16.33\%, \qquad \sigma_X = 3.81\%$$
$$E[X]_{large} = 18.96\%, \qquad \sigma_X = 2.67\%$$

Come si può notare, in questo secondo test sono stati ottenuti risultati sostanzialmente differenti dal preceddente.
Nell'approccio basato su WebAssembly si può notare un utilizzo di CPU direttamente proporzionale alla dimensione dell'immagine elaborata, mentre utilizzando Node.js l'utilizzo rimane pressochè costante per tutte e tre le immagini esaminate.
Questi risultati mettono in luce come Rust, combinato con WebAssembly, riesca a sfruttare appieno le capacità computazionali della macchina quando le circostanze lo richiedono.

Al contrario, Node.js dimostra limitazioni nell'impiego ottimale della CPU, traducendosi in tempi di latenza maggiori per richieste ad alta intensità computazionale.

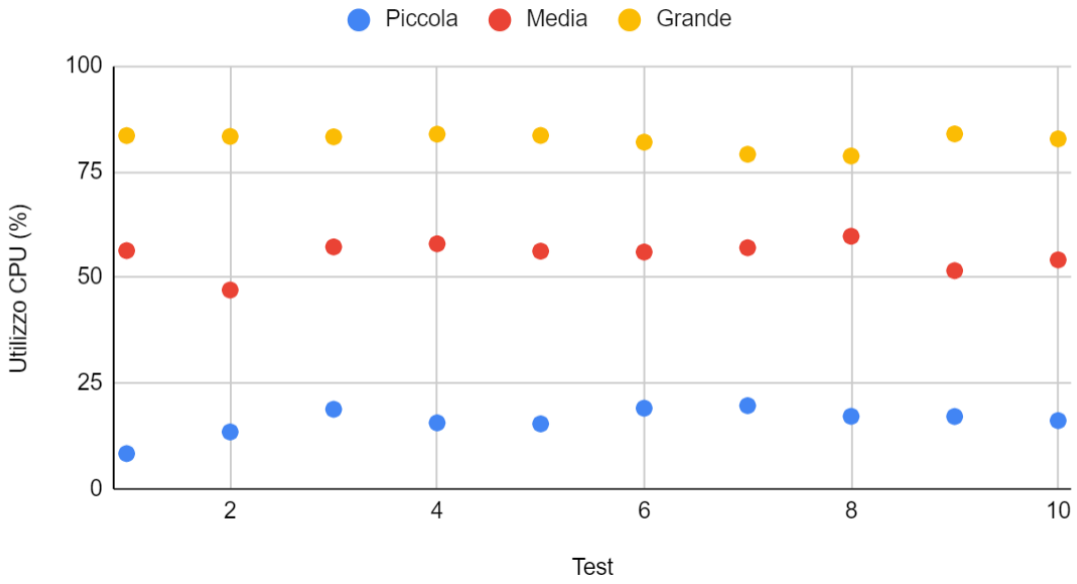

Figura 3.5: Utilizzo CPU misurato utilizzando Rust in combinazione con WebAssembly.

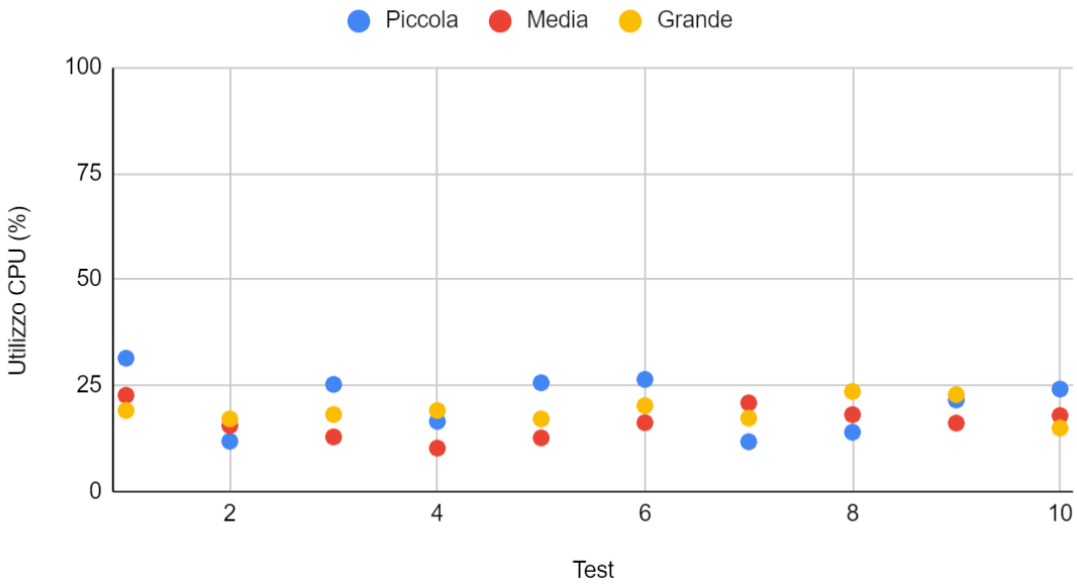

Figura 3.6: Utilizzo CPU misurato utilizzando Node.js

3.6.3 Consumo memoria

Come menzionato in precedenza, l'ultimo test eseguito riguarda il consumo di memoria durante l'esecuzione di una richiesta di elaborazione.
Per quanto riguarda Rust sono stati ottenuti i seguenti risultati relativi al consumo di memoria:

$$E[X]_{small} = 50.24\text{MB}, \qquad \sigma_X = 4.91\text{MB}$$
$$E[X]_{medium} = 48.79.\text{MB}, \qquad \sigma_X = 1.02\text{MB}$$
$$E[X]_{large} = 47.51.\text{MB}, \qquad \sigma_X = 0.90\text{MB}$$

Per quanto riguarda Node.js, invece:

$$E[X]_{small} = 80.51\text{MB}, \qquad \sigma_X = 2.73\text{MB}$$
$$E[X]_{medium} = 79.80\text{MB}, \qquad \sigma_X = 1.16\text{MB}$$
$$E[X]_{large} = 79.92\text{MB}, \qquad \sigma_X = 1.55\text{MB}$$

Durante quest'ultima analisi, i risultati ottenuti sono stati molto simili per entrambi gli approcci, ma sostanzialmente differenti rispetto ai test precedenti.
In particolare, Node.js presenta un consumo di memoria leggermnete maggiore, ma in entrambi i casi, tale parametro viene mantenuto pressochè costante per ciascuna tipologia di immagine, non presentando differenze significative.

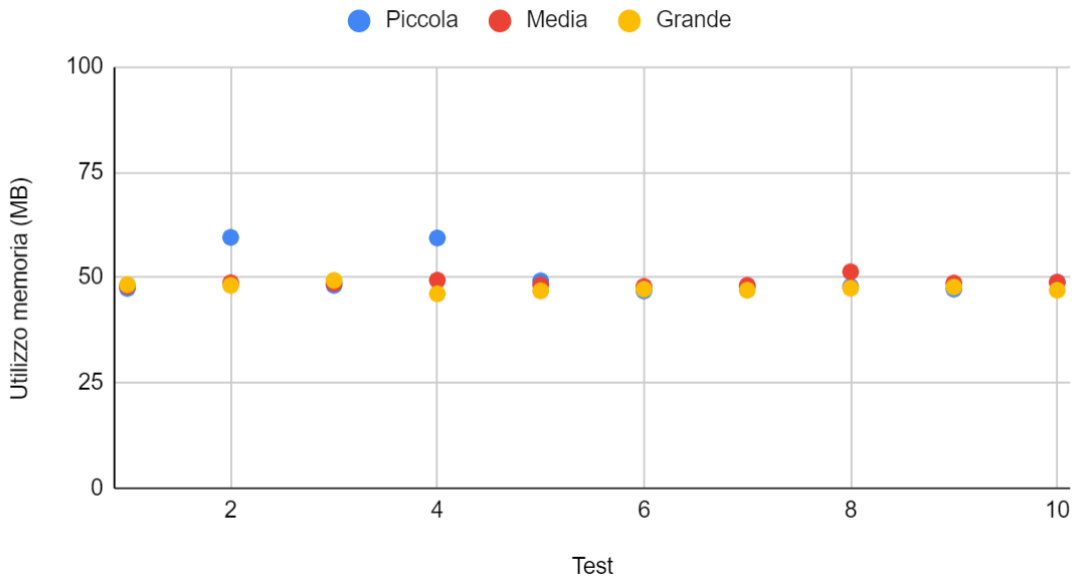

Figura 3.7: Consumo di memoria misurato utilizzando Rust in combinazione con Wasm.

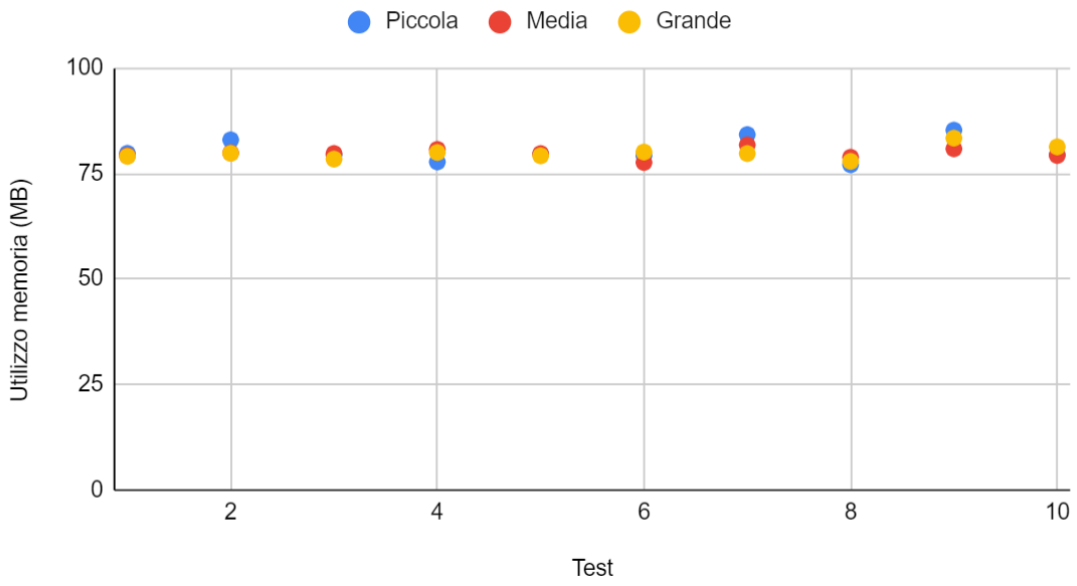

Figura 3.8: Consumo di memoria misurato utilizzando Node.js

3.7 Conclusioni

Per concludere questo lavoro di tesi, vengono presentate le considerazioni derivate dai risultati precedentemente esposti.

In primo luogo, è emerso che l'approccio basato su Rust e WebAssembly rappresenta una **scelta eccellente** in situazioni caratterizzate da operazioni ad **alta intensità computazionale**. Tuttavia, si rende necessaria un'analisi approfondita per determinare le funzionalità dell'applicazione che potrebbero effettivamente beneficiare di un'implementazione attraverso WebAssembly.

L'integrazione di codice Wasm all'interno di un programma Rust si è rivelata tutt'altro che immediata, comportando inoltre un **overhead** non trascurabile per un'applicazione che deve gestire un elevato volume di richieste.

D'altro canto Node.js offre un approccio di sviluppo più lineare e accessibile, specialmente per coloro che provengono dalle tecnologie web e posseggono una conoscenza consolidata del linguaggio JavaScript.

Tuttavia, l'adozione di Node.js potrebbe introdurre ulteriori sfide in quanto, per sua natura, risulta più indicato per applicazioni che fanno un ampio utilizzo del file system e in seguito alla debole tipizzazione potrebbe introdurre un numero di controlli molto elevato rispetto a Rust.

Durante lo svolgimento dei test è stato possibile notare, che Node.js introduce tempi di latenza peggiori rispetto a Wasm già con immagini relativamente piccole per gli standard odierni (5 Megapixel, 4,6 MegaByte). Inoltre è stata evidenziata una scarsa capacità nello sfruttare appieno la CPU, con un utilizzo che è rimasto praticamente costante durante tutti i test.

In conclusione è possibile affermare che l'utilizzo di WebAssembly server side può rappresentare una **scelta ottimale** per applicazioni che devono gestire richieste comportanti un elevato numero di calcoli da parte della CPU. Tuttavia, è importante evitare un uso eccessivo di questa tecnologia per richieste troppo semplici, al fine di prevenire un overhead non necessario e che potrebbe essere evitato anche utilizzando un linguaggio interpretato come JavaScript.

Ringraziamenti

Grazie a tutti.

Bibliografia

[1] *Node.js*. URL: https://nodejs.org/.

[2] *WebAssembly*. URL: https://webassembly.org.

[3] *WebAssembly System Interface*. URL: https://wasi.dev/.

[4] *Rust and WebAssembly*. URL: https://www.rust-lang.org/what/wasm.

[5] *WebAssembly Design*. URL: https://github.com/WebAssembly/design.

[6] Brian Sletten. *WebAssembly: The Definitive Guide*. O'Reilly Media, Inc, 2021. URL: www.oreilly.com/library/view/webassembly-the-definitive/9781492089834/.

[7] Brendan Eich. «From ASM.JS to WebAssembly». In: (2015). URL: https://brendaneich.com/2015/06/from-asm-js-to-webassembly.

[8] Andreas Rossberg, cur. *WebAssembly Specification*. 2023. URL: https://webassembly.github.io/spec/core/#webassembly-specification.

[9] *Soundness of WebAssembly*. URL: https://webassembly.github.io/spec/core/appendix/properties.html.

[10] *Soundness of WebAssembly, theorems*. URL: https://webassembly.github.io/spec/core/appendix/properties.html#theorems.

[11] *WebAssembly System Interface, High Level Goals*. URL: https://github.com/WebAssembly/WASI/#wasi-high-level-goals.

[12] *WebAssembly System Interface, Design Principles*. URL: https://github.com/WebAssembly/WASI/#wasi-design-principles.

[13] *WebAssembly System Interface, Capabilities.* URL: https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-capabilities.md.

[14] *Wasmtime Documentation.* URL: https://docs.wasmtime.dev/.

[15] *Rust language.* URL: https://www.rust-lang.org/.

[16] *WASI API Documentation.* URL: https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md.

[17] *Rust image crate.* URL: https://docs.rs/image/latest/image/.

[18] *Express.js.* URL: https://expressjs.com/.