

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Triennale

Titolo

Tesi in Tecnologie Web

Relatore
prof. Paolo Bellavista

Laureando
Davide Crociati

Ottobre 2023

Sommario

Piacere, sommario.

Indice

Elenco delle figure	v
Elenco dei listati	vi
1 Introduzione	1
1.1 Contestualizzazione	1
1.1.1 Trend di evoluzione del web	1
1.1.2 Importanza dell'ottimizzazione	2
1.2 Motivazioni e Obiettivi	3
1.2.1 Tipologia di applicazione	3
1.2.2 Metodologie confrontate	4
1.2.3 Analisi Comparativa delle Tecnologie	5
1.2.4 Valutazione dell'Impatto di Wasm	6
1.2.5 Obiettivi	7
1.3 Struttura della tesi	8
2 Tecnologie utilizzate	9
2.1 WebAssembly	9
2.1.1 Storia e Origini	10
2.1.2 Casi d'uso	11
2.1.3 Concetti chiave	12
2.1.4 Fasi semantiche	15
2.1.5 Correttezza logica	18
2.1.6 Creazione e utilizzo di moduli WebAssembly	19
2.2 WebAssembly System Interface	20

2.2.1	Gli obiettivi	20
2.2.2	I principi di design	21
2.2.3	Wasmtime	23
2.2.4	Rust	24
2.2.5	Esempi	25
2.3	Node.js	29
2.3.1	Funzionalità cardine	29
2.3.2	Casi d'uso	30
2.4	Confronto tra tecnologie	31
2.4.1	Prestazioni	31
2.4.2	Sicurezza	31
2.4.3	Facilità di Sviluppo	32
2.4.4	Scalabilità ed Espandibilità	32
2.5	Conclusioni preliminari	33
3	Capitolo 3	34
3.1	Prototipo	34
	Riferimenti bibliografici	35

Elenco delle figure

1.1	L'evoluzione del Web.	2
1.2	Image Processing	4
1.3	Rust e WebAssembly	5
1.4	Node.js	6
2.1	WebAssembly all'interno del browser	10
2.2	Architettura di WebAssembly	14
2.3	La sintassi astratta di WebAssembly	15
2.4	La semantica di WebAssembly	17
2.5	Architettura event-driven di Node.js	30

Elenco dei listati

2.1	Esempio in Rust	26
2.2	Le API importate di WASI	27

Capitolo 1

Introduzione

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 server. 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.

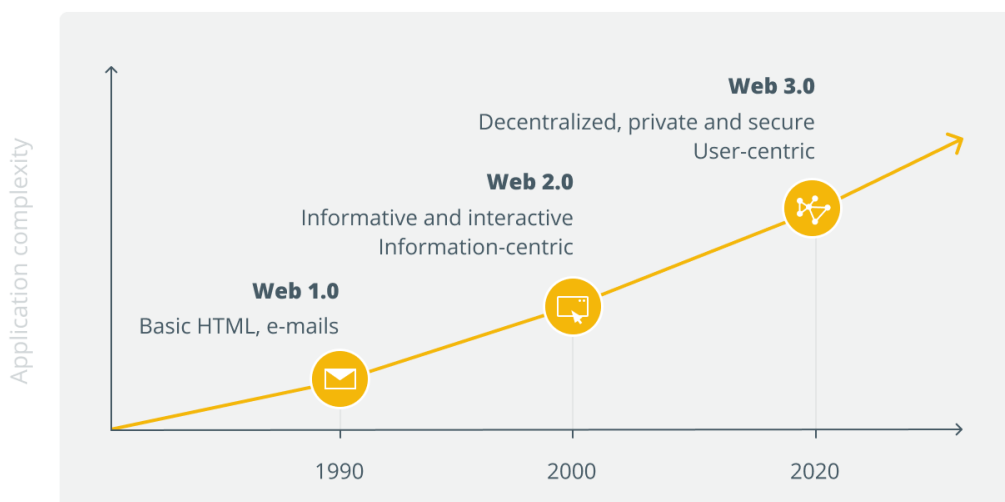


Figura 1.1: L'evoluzione del Web.

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 riguarda 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 e 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.

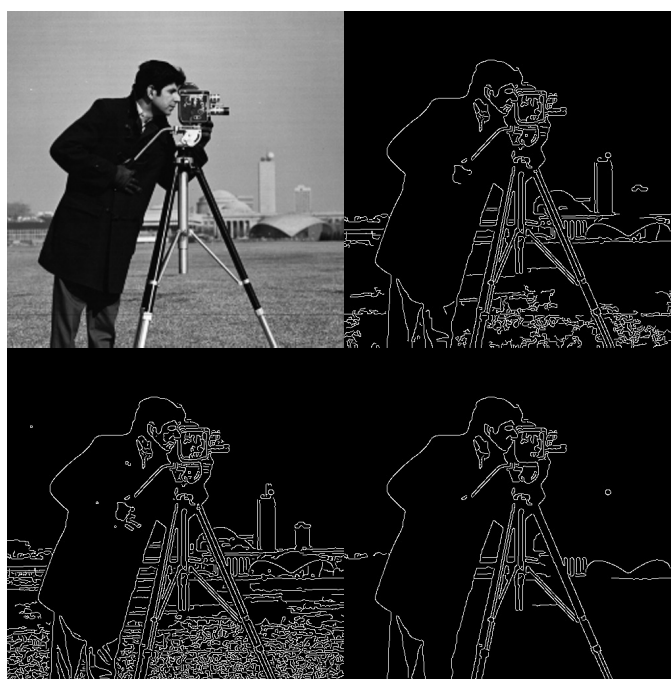


Figura 1.2: Il campo dell'elaborazione digitale di immagini è molto ampio e sarebbe possibile integrare anche operazioni avanzate (edge detection, pattern recognition etc.), che richiederebbero risorse computazionali molto elevate. Per lo scopo di questa tesi ci si limiterà a elaborazioni più semplici, 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** che 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 successivamente durante i test e i benchmark dell'applicazione sviluppata.

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.



Figura 1.3: Rust e WebAssembly

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, per lo scopo di questa tesi si è scelto di confrontare due approcci differenti, ma sempre più utilizzati nelle applicazioni web moderne. In particolare si partirà da WebAssembly e dall'interfaccia di sistema WASI, per finire con un'introduzione anche su 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. Wasm è attualmente sviluppato in maniera open-source, da un *Community Group* del W3C che comprende membri di tutti i browser più utilizzati.

Esso è nato principalmente come integrazione a JavaScript, per consentire l'esecuzione di codice ad una velocità paragonabile a quella del codice nativo, grazie alla dimensione ridotta dei file binari generati e alla loro efficienza.

Inoltre grazie all'esecuzione all'interno di una sandbox è garantita un'ottima sicurezza, anche sotto il punto di vista della memoria.

Grazie al formato testuale (.wat), è possibile 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]

Al giorno d'oggi è in costante crescita, sia il numero di linguaggi che possono avere come *compilation target* WebAssembly (Rust, C, Go, Kotlin, etc), sia il numero di siti che

sfruttano tale tecnologia, grazie anche al fatto che praticamente ogni browser attualmente utilizzato, supporta Wasm.



Figura 2.1: WebAssembly all'interno del browser

2.1.1 Storia e Origini

I predecessori

Nel 2011 Google rilasciò un progetto open source chiamato **Native Client (NaCl)**. L'obiettivo era quello di consentire l'esecuzione di codice nativo nel browser all'interno di una sandbox con privilegi limitati. In particolare si stava cercando di supportare software che richiedevano grande sforzo computazionale (simulazioni, elaborazioni di audio e 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. Non mancavano però diversi problemi. Il codice ottenuto era eseguibile solo nel browser Chrome ed era impossibile l'interazione con JavaScript o con altre API sul 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 possibilità 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. Si voleva quindi trovare un modo per unificare gli enormi vantaggi offerti da ognuno dei due approcci.[6]

La nascita di Wasm

Fu nel Giugno 2015 che Brendan Eich (creatore di JavaScript), insieme ad altri sviluppatori di Mozilla, annunciarono che lo sviluppo di WebAssembly era cominciato.[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 per essere un 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 a quella di parsing da parte dell'interprete JavaScript per i file di asm.js. Nel 2017 venne 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 da 2 f64, oppure da 4 i32 etc.);
- Un tipo riferimento per puntatori a differenti entità;

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

Infine è anche possibile .

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, vengono normalmente gestiti.

Funzioni

Il codice è diviso in funzioni. Ognuna di queste ha una certa sequenza di valori sia come parametri che come tipo di ritorno. In una funzione viene eseguita una serie di istruzioni, possono inoltre essere chiamate 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.

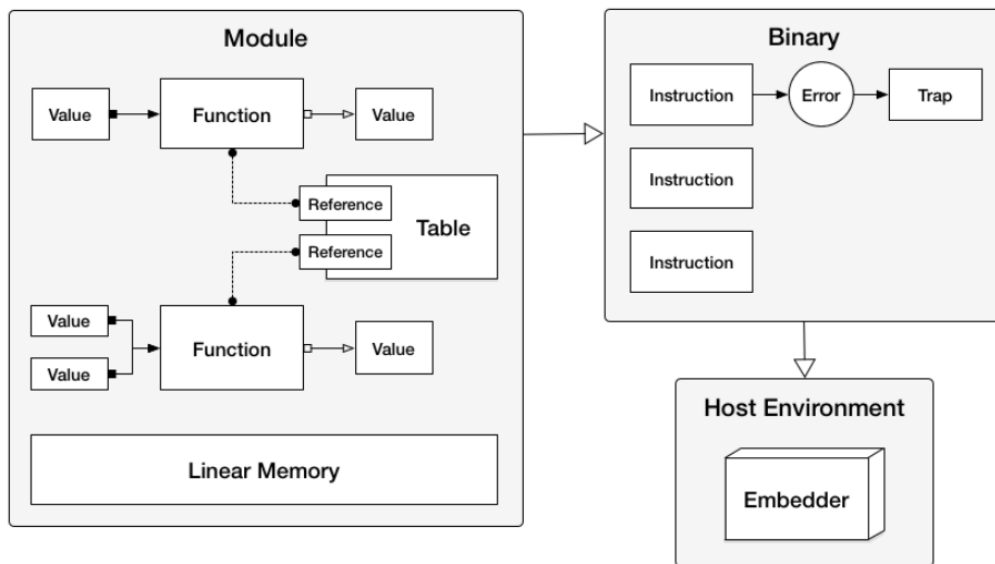


Figura 2.2: Architettura di WebAssembly

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.

(value types) $t ::= i32 \mid i64 \mid f32 \mid f64$	(instructions) $e ::= \text{unreachable} \mid \text{nop} \mid \text{drop} \mid \text{select} \mid$
(packed types) $tp ::= i8 \mid i16 \mid i32$	$\text{block } tf \ e^* \text{ end} \mid \text{loop } tf \ e^* \text{ end} \mid \text{if } tf \ e^* \text{ else } e^* \text{ end} \mid$
(function types) $tf ::= t^* \rightarrow t^*$	$\text{br } i \mid \text{br_if } i \mid \text{br_table } i^+ \mid \text{return} \mid \text{call } i \mid \text{call_indirect } tf \mid$
(global types) $tg ::= \text{mut}^? t$	$\text{get_local } i \mid \text{set_local } i \mid \text{tee_local } i \mid \text{get_global } i \mid$
	$\text{set_global } i \mid t.\text{load } (tp_sx)^? \ a \ o \mid t.\text{store } tp^i \ a \ o \mid$
	$\text{current_memory} \mid \text{grow_memory} \mid t.\text{const } c \mid$
	$t.\text{unop}_t \mid t.\text{binop}_t \mid t.\text{testop}_t \mid t.\text{relop}_t \mid t.\text{cvtop } t.\text{sx}^?$
$\text{unop}_{iN} ::= \text{clz} \mid \text{ctz} \mid \text{popcnt}$	(functions) $f ::= ex^* \text{ func } tf \ \text{local } t^* \ e^* \mid ex^* \text{ func } tf \ im$
$\text{unop}_{tN} ::= \text{neg} \mid \text{abs} \mid \text{ceil} \mid \text{floor} \mid \text{trunc} \mid \text{nearest} \mid \text{sqrt}$	(globals) $glob ::= ex^* \text{ global } tg \ e^* \mid ex^* \text{ global } tg \ im$
$\text{binop}_{iN} ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div_sx} \mid \text{rem_sx} \mid$	(tables) $tab ::= ex^* \text{ table } n \ i^* \mid ex^* \text{ table } n \ im$
$\text{and} \mid \text{or} \mid \text{xor} \mid \text{shl} \mid \text{shr_sx} \mid \text{rotr} \mid \text{rotr}$	(memories) $mem ::= ex^* \text{ memory } n \mid ex^* \text{ memory } n \ im$
$\text{binop}_{tN} ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div} \mid \text{min} \mid \text{max} \mid \text{copysign}$	(imports) $im ::= \text{import } "name" \ "name"$
$\text{testop}_{iN} ::= \text{eqz}$	(exports) $ex ::= \text{export } "name"$
$\text{relop}_{iN} ::= \text{eq} \mid \text{ne} \mid \text{lt_sx} \mid \text{gt_sx} \mid \text{le_sx} \mid \text{ge_sx}$	(modules) $m ::= \text{module } f^* \ glob^* \ tab^? \ mem^?$
$\text{relop}_{tN} ::= \text{eq} \mid \text{ne} \mid \text{lt} \mid \text{gt} \mid \text{le} \mid \text{ge}$	
$\text{cvtop} ::= \text{convert} \mid \text{reinterpret}$	
$sx ::= s \mid u$	

Figura 2.3: La sintassi astratta di WebAssembly

Ad esempio viene controllato l'ordine delle istruzioni nel corpo di una funzione assicurandosi che lo stack sia utilizzato nel modo corretto. Oppure se nelle specifiche esaminiamo un'operazione binaria tra tipi numerici ($i32.add$, $f64.xor$ etc.), notiamo che essa è del tipo $t.binop$ e dato un contesto C essa è valida solamente con tipo $[t \ t] \rightarrow [t]$.

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

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

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'istanziamento). Terminata l'istanziamento, 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 $\text{binop}_t(c_1, c_2)$;
- Viene eseguita l'operazione di **push** del valore $t.\text{const } c$ nello stack.
- Altrimenti:
 - Trap.

In notazione formale:

$$\begin{aligned}
 (t.\text{const } c_1) (t.\text{const } c_2) t.\text{binop} &\hookrightarrow (t.\text{const } c) && (\text{if } c \in \text{binop}_t(c_1, c_2)) \\
 (t.\text{const } c_1) (t.\text{const } c_2) t.\text{binop} &\hookrightarrow \text{trap} && (\text{if } \text{binop}_t(c_1, c_2) = \{\})
 \end{aligned}$$

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'istanziamento, che l'invocazione di funzioni, sono operazioni che avvengono all'interno dell'ambiente di esecuzione dell'host.

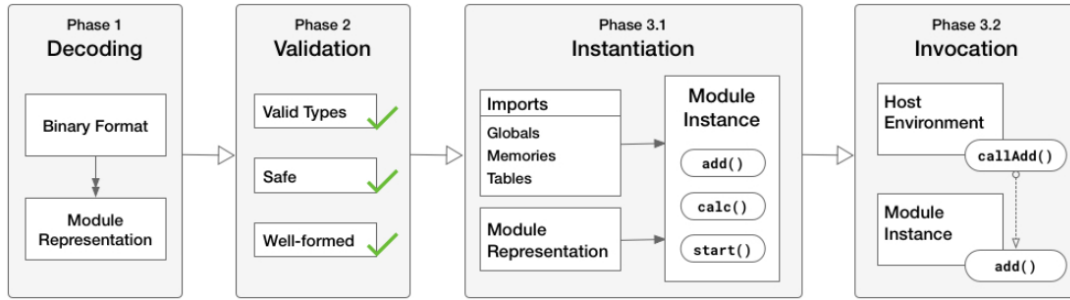


Figura 2.4: La semantica di WebAssembly

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 sicurezza rispetto ai tipi, 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 nessun'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 fuori dalla funzione dove è dichiarata e nessun componente di un modulo può essere acceduto fuori dallo stesso a meno che quest'ultimo 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 dato la definizioni di configurazione valida è stato possibile enunciare due teoremi e derivare un corollario che riassumendo dicono 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 derivante dall'istanziazione o dall'invocazione di un modulo valido, può avere 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 il 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 soprattutto 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 API 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 non è limitato dal bisogno di essere compatibile con JavaScript ed ha integrato 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 progettate direttamente per programmi WebAssembly. Esse devono essere portabili (utilizzabili su diversi sistemi), modulari (composte da più componenti riutilizzabili), indipendenti dal runtime sottostante e devono mantenere le caratteristiche di sicurezza garantite da WebAssembly (sandbox) grazie ad una progettazione basata su **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 in modo 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;

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 illustrati di seguito:[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), contenuti in tabelle esterne che rappresentano 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 normalmente 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 possibile 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 questo 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 possibili per un embedding di questo tipo, sono diverse (Wasmtime, Wasmer), ma non così numerose. Tra queste si è optato per Wasmtime, un progetto sviluppato della Bytecode Alliance.

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. [14]

2.2.4 Rust

Come è già stato menzionato, si è scelto Rust come linguaggio di programmazione dove integrare l'esecuzione di moduli WebAssembly. 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:** WebAssembly consente la compilazione di codice in moduli WebAssembly, consentendo l'esecuzione di codice Rust 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, sviluppo di giochi e 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** Permessso di invocare `fd_read` e `sock_recv`
- `fd_write`: **bool** Permessso di invocare `fd_write` e `sock_send`
- `path_create_directory`: **bool** Permessso di invocare `path_create_directory`
- `sock_accept`: **bool** Permessso di invocare `sock_accept`

Si procede 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
```



```
4     fn main() {
5         let greeting = "Hello, world!";
6         let outfile = "hello.txt";
7         let mut output_file = fs::File::create(outfile)
8             .expect(&format!("Error creating {}", outfile));
9
10        output_file.write_all(greeting.as_bytes())
11            .expect(&format!("Error writing: {}", outfile));
12
13        let mut input_file = fs::File::open(outfile)
14            .expect(&format!("Error opening {}", outfile));
15
16        let mut input = String::new();
17        input_file.read_to_string(&mut input)
18            .expect(&format!("Error reading: {} ", outfile));
19
20        println!("Read in from file: {}", input);
21    }
```

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 `.wasm` all'interno della directory **target/wasm32-wasi/release**.

Se convertiamo tale file nell'equivalente testuale `.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 hello.txt: Custom {
  kind: Uncategorized, error: "failed to find a pre-opened
  file descriptor through which \"hello.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 hello.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: Hello, world!
```

2.3 Node.js

Node.js è un runtime environment che consente l'esecuzione di codice JavaScript server-side, fornendo 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 casi d'uso.

- **Architettura Event-Driven:** viene usato un modello di I/O asincrono, non bloccante che gestisce in maniera efficiente richieste di I/O concorrenti;
- **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. Diventa possibile 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;

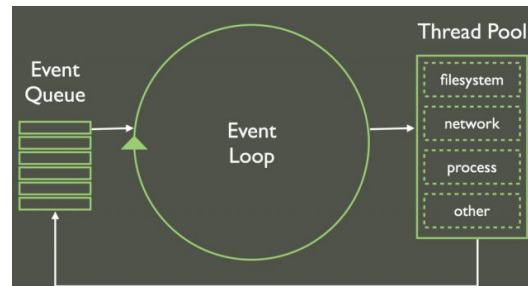


Figura 2.5: Architettura event-driven di Node.js

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. Utilizzando 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 rimarrebbero 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** per WASM, dell'esecuzione all'interno di una sandbox. 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 spiegati precedentemente attirano molto ma per metterli in pratica potrebbe essere necessario uno sforzo iniziale dei programmatori non indifferente. La forte tipizzazione di Rust in combinazione col suo compilatore (famoso per la sua rigidità), potrebbero far faticare non poco chi si approccia per la prima volta a Rust. Questi due elementi contribuiscono ad una curva di apprendimento già impegnativa a causa delle funzionalità uniche di Rust.

Risulta molto diverso invece un primo approccio a Node.js. L'utilizzo di un linguaggio famoso e ampiamente usato come JavaScript e del package manager NPM possono accelerare fortemente lo sviluppo.

Può quindi cambiare molto le tempistiche di sviluppo la scelta di un certo approccio, ma con Rust potremmo essere ricompensati con un codice più robusto e affidabile nel tempo. La scelta deve essere guidata dai requisiti di progetto e non dalla voglia di sbrigarsi.

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 WebAssembly si possono sfruttare a pieno anche processori multi-core. Inoltre è anche consentito lo sviluppo di microservizi e la loro combinazione con Wasm/WASI.

Node.js riesce a scalare orizzontalmente grazie a tecniche di clustering e load balancing. Ciò però è efficace soprattutto per task I/O bound. A causa del thread singolo utilizzato da Node.js, la scalabilità per task CPU-intensive è molto limitata. Si potrebbe scalare in verticale un singolo server, aumentando le risorse disponibili, ma sono ovviamente presenti limiti pratici.

2.5 Conclusioni preliminari

Capitolo 3

Capitolo 3

3.1 Prototipo

Ringraziamenti

Grazie a Grazia, Graziella e la sorella.

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>.