

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Dipartimento di Informatica · Scienza e Ingegneria · DISI
Corso di Laurea in Ingegneria Informatica

**SVILUPPO DI APPLICAZIONI BASATE SU WEB
ASSEMBLY SYSTEM INTERFACE (WASI)**

Relatore:
Prof. Paolo Bellavista

Presentata da:
Luca Corsetti

Anno Accademico 2022/2023

Indice

Elenco delle figure	5
Introduzione	7
1 Al di là del browser	9
1.1 Un focus su WebAssembly	9
1.2 Cos'è una System Interface?	10
1.3 Il valore aggiunto	12
1.3.1 Portabilità	13
1.3.2 Sicurezza	15
1.4 Stato dell'arte	15
1.4.1 Chi supporta Wasm?	15
1.4.2 A che punto è lo standard	16
1.4.3 Runtime WASI	17
1.4.4 I container Wasm	18
2 WASI	21
2.1 WASI-libc	21
2.1.1 Musl-libc	22
2.1.2 Libpreopen	22
2.2 La Portabilità	22
2.2.1 Un accenno ai compilatori	22
2.2.2 Cranelift	24
2.2.3 LLVM	24
2.2.4 La differenza	24
2.2.5 Linguaggi compilati ed interpretati	25
2.3 La gestione delle risorse	26
2.3.1 Docker+Wasm	27

2.3.2	Uno use case: i microservizi con Wasm	29
2.4	La sicurezza con WASI	30
2.4.1	Il problema	30
2.4.2	Una possibile soluzione	31
2.4.3	La soluzione: i nanoprocessi di WebAssembly	33
2.4.4	Capability Based Model	34
3	Prototipo	39
3.1	Architettura	39
3.1.1	Le funzionalità	39
3.1.2	Le tecnologie backend	40
3.1.3	Le tecnologie frontend	42
3.1.4	La configurazione	43
3.1.5	Interazione	43
3.2	Un confronto	44
3.2.1	Benchmark	45
3.2.2	Il deployment	46
3.3	Vantaggi e Svantaggi	48
	Conclusione	51
	Ringraziamenti	53

Elenco delle figure

1.1	Esempio di una funzione compilata nel modo 'tradizionale', in .wasm e la sua rappresentazione in .wat	10
1.2	System Interface: Dall'API alla sua implementazione	11
1.3	Una System Interface per un sistema operativo concettuale	12
1.4	Portabilità in senso tradizionale	13
1.5	Portabilità con Wasm e WASI	14
1.6	Evoluzione del cloud computing	19
2.1	Una overview sul funzionamento dei compilatori	23
2.2	Dal runtime al linguaggio macchina	25
2.3	Linguaggi compilati e interpretati con Wasm	26
2.4	Esempio di immagine Docker per mysql	27
2.5	Container tradizionali e container Wasm su Docker	28
2.6	Come sono composte le applicazioni moderne	30
2.7	Tempo necessario per risolvere le vulnerabilità, un confronto tra open source e codice proprietario	31
2.8	Isolamento dei processi nei sistemi operativi	32
2.9	Isolamento dei package in sandbox isolate, evidente inefficienza e overhead	33
2.10	Nanoprocessi di Wasm	34
2.11	Il modello capability based in azione	37
3.1	Architettura dell'applicazione	42
3.2	Configurazione di un microservizio	43
3.3	Entry point delle operazioni CRUD al microservizio dei messaggi	44
3.4	10000 richieste in sequenza verso Wasm	45
3.5	10000 richieste in sequenza verso NodeJS	45
3.6	Utilizzo CPU per Wasm	45
3.7	Utilizzo CPU per NodeJS	45

3.8	100 utenti e 1000 richieste in simultanea verso Wasm	46
3.9	100 utenti e 1000 richieste in simultanea verso NodeJS	46
3.10	Utilizzo CPU per Wasm	46
3.11	Utilizzo CPU per NodeJS	46
3.12	Creazione immagine Docker per l'applicazione NodeJS	47
3.13	Docker compose per l'applicazione NodeJS	48

Introduzione

Negli ultimi decenni, la crescente richiesta di servizi e applicazioni online ha portato ad un'esplosione dello sviluppo delle tecnologie web. Questo cambiamento è stato reso possibile dall'aumento dell'ubiquità dei dispositivi connessi a Internet, che rendono l'accesso ai servizi web sempre più immediato, semplice e alla portata di tutti. Di conseguenza, le tecnologie e le applicazioni che garantiscono questi servizi devono essere in grado di rispondere in modo efficiente a volumi di dati e utenti sempre maggiori. In questo scenario, il cloud computing rappresenta un elemento essenziale per la gestione di applicazioni distribuite, ovvero applicazioni suddivise in più componenti spesso eterogenei, sia per composizione che locazione fisica.

Il cloud computing è un modello di erogazione di servizi che consente di gestire l'infrastruttura informatica necessaria per rendere disponibili applicazioni, dati e servizi online in modo rapido, efficiente e flessibile. Grazie al cloud computing, le risorse informatiche come server, storage e software possono essere facilmente scalate per rispondere alle esigenze delle organizzazioni e degli utenti finali, consentendo un accesso sicuro e veloce ai servizi e alle applicazioni da qualsiasi dispositivo connesso a Internet. La rapidità e velocità con cui il cloud computing eroga questi servizi è data dalle tecnologie che lo compongono.

In questo contesto andremo a discutere di una emergente tecnologia, chiamata **WASI (WebAssembly System Interface)**¹ e come questa possa essere considerata la terza ondata del cloud computing[1]. Grazie a WASI, è possibile eseguire applicazioni in un ambiente isolato e sicuro, senza la necessità di dover conoscere il sistema operativo sottostante. È stato progettato per essere altamente portabile, consentendo alle applicazioni di essere eseguite in modo efficiente su qualsiasi piattaforma.

Nasce e si sviluppa sopra ad una tecnologia già esistente: **WebAssembly** (o **Wasm**). Quest'ultima è un'innovativa tecnologia nata con l'obiettivo di migliorare le prestazioni delle applicazioni web **sul browser**. È stata progettata con l'intento di superare le limitazioni poste da Javascript. In particolare, si propone di essere veloce, efficiente e portabile, oltre che retro-compatibile con le tecnologie già esistenti. Va notato che WebAssembly non è pensato per sostituire JavaScript, ma piuttosto per migliorare le aree in cui quest'ultimo presenta alcune lacune: come il rendering 3D, il video editing, giochi in-browser e così via.

WASI eredita tutte queste caratteristiche da Wasm e le utilizza per lo sviluppo di applicazioni **al di fuori** dei browser.

Di seguito andremo ad approfondire WASI ed esporremo come rappresenti una tecnologia estremamente promettente per il futuro nell'ambito del cloud computing. Si

¹<https://wasi.dev/>

partirà affrontando l'argomento da un punto di vista generale, andando a definire le motivazioni storiche che hanno portato alla sua ideazione, il suo funzionamento e lo stato dell'arte della tecnologia. Lo si metterà a confronto con le soluzioni esistenti, in particolare affrontando le sue principali somiglianze e differenze con il modello a container, estremamente popolare al giorno d'oggi e con il suo predecessore, le macchine virtuali. Si studieranno le principali caratteristiche ereditate da Wasm ponendo particolare attenzione alla loro implementazione fuori dal browser da un punto di vista tecnico. Tra queste caratteristiche ritroviamo:

- **efficienza**, dato il suo formato binario di piccole dimensioni, simile al linguaggio macchina
- **portabilità**, grazie alla possibilità di essere eseguito su molteplici sistemi e architetture allo stesso modo mediante applicazioni dette runtime simili, per funzionamento, alla JVM di Java.
- **interoperabilità**, grazie al fatto che non è strettamente legato ad alcun linguaggio di programmazione specifico
- **sicurezza**, in quanto eseguito in ambiente sandbox secondo il modello capability-based

La spiegazione teorica sul funzionamento e sui vantaggi sfocerà in un'implementazione di un piccolo prototipo scritto in linguaggio Rust basato su un'architettura a microservizi REST API in cui si andranno a mettere in pratica i concetti visti in precedenza. Si andrà a valutare il deployment dell'applicazione e ad effettuare benchmark per valutarne i tempi di risposta a livelli di carico differenti per simularne un caso d'uso reale. I risultati ottenuti saranno messi a confronto con un'applicazione che propone le stesse funzionalità ma implementata con una diversa tecnologia.

Capitolo 1

Al di là del browser

1.1 Un focus su WebAssembly

Tradizionalmente, l'unico linguaggio utilizzabile all'interno dei browser era JavaScript, perfetto per la creazione di interfacce utente ma non per operazioni che richiedono una complessità maggiore. WebAssembly è stato progettato per essere un formato di esecuzione più efficiente, veloce e sicuro rispetto a JavaScript, da usare in combinazione con esso[2].

In sintesi, WebAssembly è un formato binario (.wasm) progettato per essere eseguito da una macchina virtuale integrata all'interno dei browser. Grazie alla sua natura binaria, è possibile utilizzare diversi linguaggi di programmazione, come C, C++ e Rust, che supportano questo formato.

Ogni file .wasm contiene un **modulo**, che può essere visto come un'unità di codice autonomo, composto da funzioni, dati e altre risorse. Il modulo viene eseguito all'interno della macchina virtuale del browser in modalità sandboxed, che garantisce l'isolamento e la sicurezza dell'esecuzione.

Ogni rappresentazione binaria possiede anche una duale rappresentazione testuale chiamata **WebAssembly Text Format (.wat)**. Questo formato ha una sintassi simile ai linguaggi Assembly, il che lo rende più leggibile per gli esseri umani rispetto al formato binario. Un file .wat è costituito da una serie di istruzioni che definiscono la struttura del modulo organizzate in sezioni.

Il vantaggio di avere una rappresentazione testuale come il formato .wat è che può aiutare gli sviluppatori a comprendere meglio la struttura e il funzionamento dei moduli, anche se non sono esperti nel linguaggio Assembly o nell'architettura della CPU.

Come noto, i linguaggi Assembly tradizionali sono strettamente legati all'architettura della CPU sottostante. Ciò significa che ogni programma scritto in Assembly è vincolato alla specifica architettura su cui deve essere eseguito. Pertanto, per rendere un'applicazione compatibile ed eseguibile su diverse macchine, è necessario compilarla per le diverse architetture che si vogliono supportare. Nel corso degli anni, gli sviluppatori hanno cercato di risolvere questo problema attraverso diverse soluzioni. Ad esempio, il linguaggio Java ha introdotto il motto "Write once, run anywhere". Wasm può essere considerato come una soluzione simile, ma con un vantaggio fondamentale: mentre Java richiede agli sviluppatori di scrivere codice nel

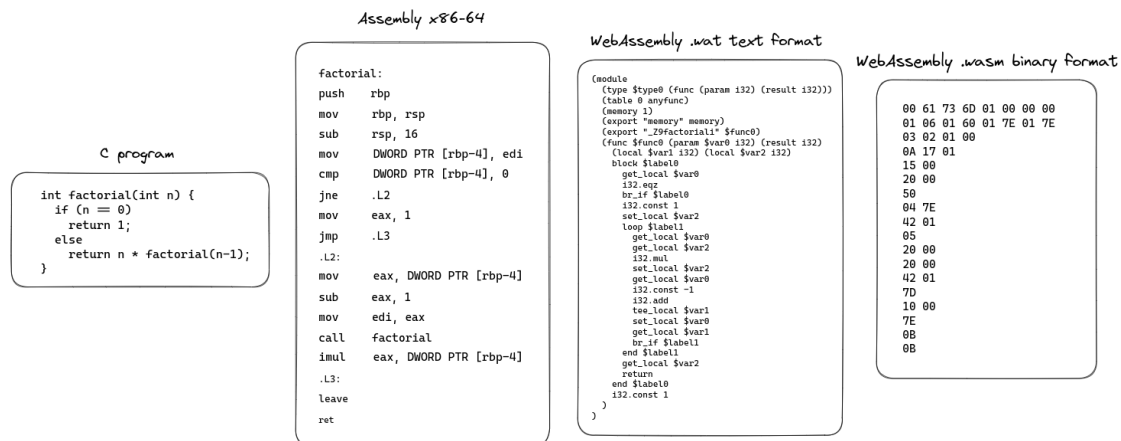


Figura 1.1: Esempio di una funzione compilata nel modo 'tradizionale', in .wasm e la sua rappresentazione in .wat

linguaggio Java o in linguaggi compatibili con la Java Virtual Machine, per poterlo eseguire su qualsiasi piattaforma, Wasm consente di utilizzare qualsiasi linguaggio di programmazione compilabile in formato .wasm e il risultante codice prodotto può essere eseguito su tutti i browser compatibili.

Tuttavia, gli sviluppatori non si sono limitati ad utilizzare Wasm solo all'interno dei browser. Anche per lo sviluppo di applicazioni tradizionali, Wasm sta diventando sempre più popolare. In questo contesto, WASI svolge un ruolo cruciale. Mentre all'interno del browser Wasm non ha bisogno di comunicare con il sistema operativo, poiché il browser funge da intermediario, al di fuori di esso, la situazione è diversa. Qui, l'applicazione deve comunicare con il filesystem, creare connessioni di rete, eseguire codice in parallelo e così via, e farlo in modo sicuro, isolando l'applicazione dal sistema operativo sottostante e garantendo che non possa interferire con altri processi o con la memoria del sistema.

WASI affronta queste sfide fornendo un insieme di interfacce standardizzate, una **system interface**, tra le applicazioni Wasm e l'ambiente di esecuzione sottostante.

1.2 Cos'è una System Interface?

Normalmente, le applicazioni non si interfacciano direttamente con le risorse del sistema, ma attraverso il sistema operativo, il cui nucleo è il kernel, che media l'accesso alle risorse. Ciò è necessario per evitare accessi indiscriminati, che potrebbero causare instabilità e problemi di sicurezza. Per questo motivo, il sistema operativo organizza la protezione in strati a livelli crescenti di privilegi. Ogni programma viene eseguito in "user mode" e se vuole eseguire operazioni privilegiate, deve chiedere al kernel di farlo attraverso le **system call**, che eseguono i controlli necessari prima di permettere l'operazione.

Per semplificare l'accesso alle risorse del sistema, molti linguaggi di programmazione forniscono una libreria standard che definisce un'interfaccia comune, chiamata system interface, indipendente dal sistema operativo sottostante. Ciò significa che gli sviluppatori non devono preoccuparsi dell'implementazione specifica del sistema

operativo, poiché possono utilizzare l'interfaccia fornita dal linguaggio di programmazione. Sarà il compilatore a scegliere l'implementazione corretta dell'interfaccia in base al sistema operativo in cui viene eseguita l'applicazione.

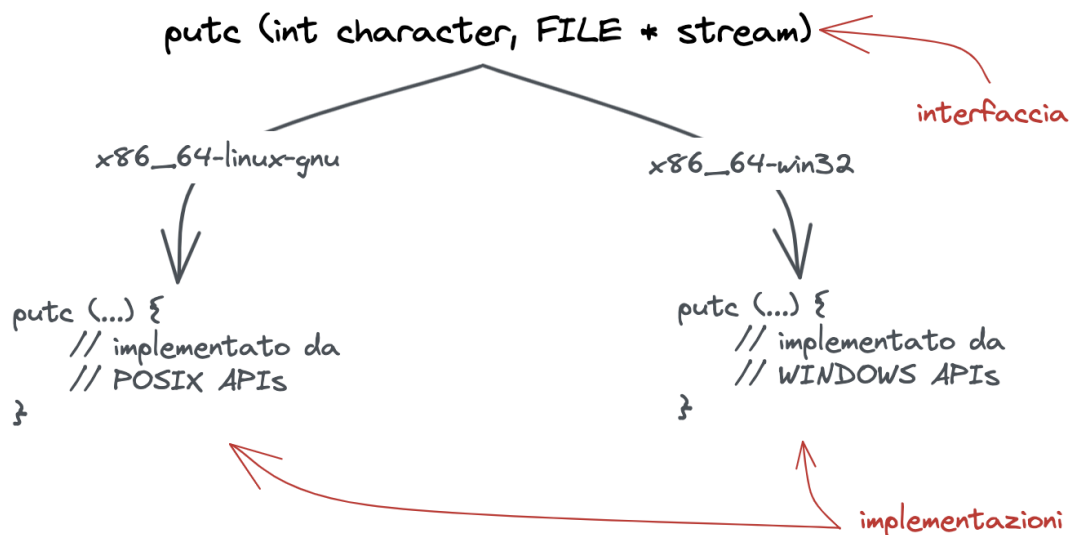


Figura 1.2: System Interface: Dall'API alla sua implementazione

Con WebAssembly, invece, è necessario definire un'interfaccia per un sistema operativo concettuale e un runtime che la implementi, poiché non si conosce a priori il sistema operativo su cui verrà eseguito il modulo `.wasm`.

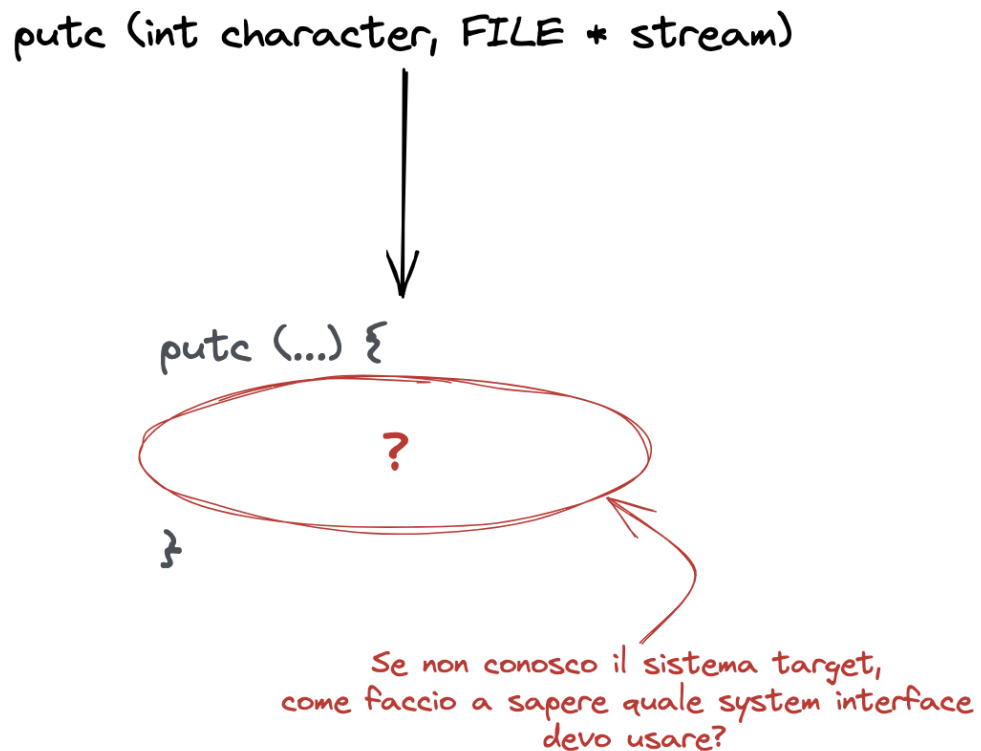


Figura 1.3: Una System Interface per un sistema operativo concettuale

Questa interfaccia deve essere poi implementata dai runtime capaci di eseguire effettivamente i moduli wasm.

1.3 Il valore aggiunto

Cosa differenzia Wasm e WASI da altri linguaggi di programmazione e tecniche di sviluppo? Wasm è stato concepito tenendo a mente le necessità del web, ogni runtime browser deve quindi soddisfare i seguenti requisiti:

- **Sicurezza:** dato che il browser esegue codice proveniente da Internet, è essenziale che il codice eseguito sia controllato e limitato affinché non possa fare ciò che vuole.
- **Dimensioni Ridotte:** poiché la larghezza di banda è una limitazione costante su Internet, il codice scaricato deve essere il più piccolo possibile, occupando al più pochi megabyte di dati.
- **Caricamento ed esecuzioni veloci:** se una pagina web non risponde entro pochissimo tempo (circa 3 secondi¹), gli utenti la abbandonano.

¹<https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics/>

- **Portabilità:** nell'era degli smartphone, tablet, dispositivi IoT, sensori e altre tecnologie eterogenee, è necessario garantire che la stessa applicazione possa essere eseguita su tutti i dispositivi, a prescindere dal sistema operativo e dall'architettura sottostante.

Si può notare come questi requisiti possono sicuramente essere un punto di forza anche al di fuori del browser. In particolare, Wasm e WASI migliorano due aspetti molto importanti già ampiamente affrontati in letteratura: la portabilità e la sicurezza.

1.3.1 Portabilità

I linguaggi tradizionali consentono di eseguire codice su diverse architetture, ma richiedono di essere compilati una volta per ogni architettura di destinazione.

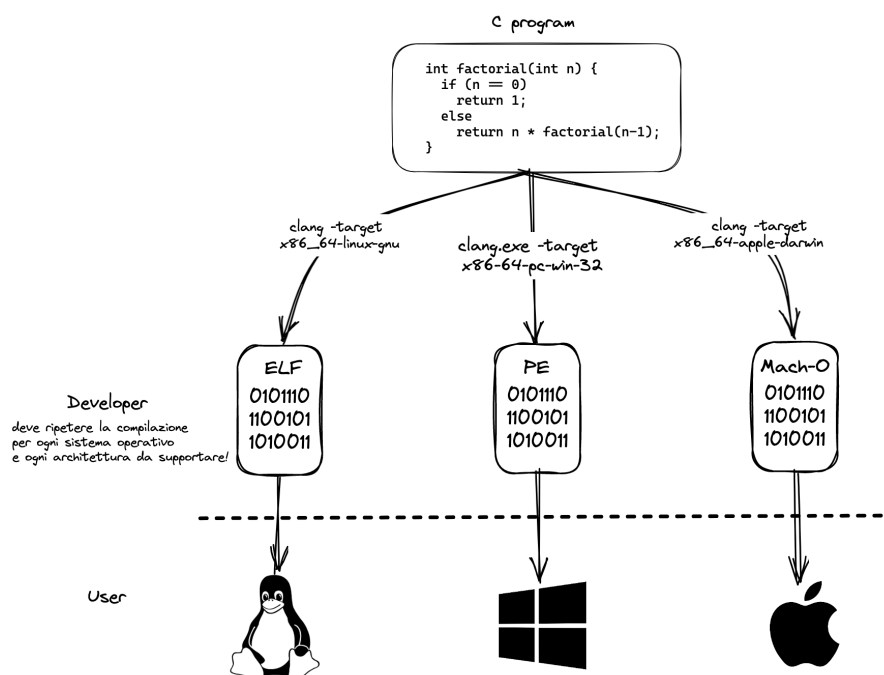


Figura 1.4: Portabilità in senso tradizionale

Con Wasm e WASI, invece, la situazione è completamente diversa. Una volta compilata l'applicazione, è possibile eseguirla su qualsiasi runtime in grado di gestire il codice Wasm. Questo approccio è noto come "Write once, run anywhere".

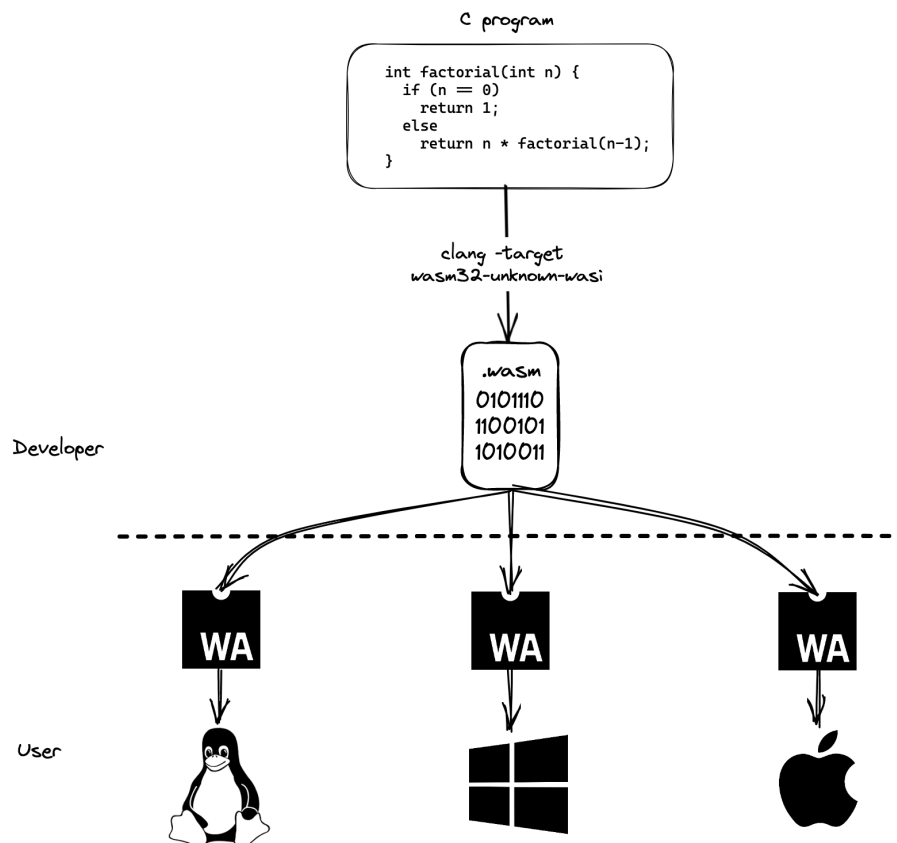


Figura 1.5: Portabilità con Wasm e WASI

Si potrebbe obiettare che l'approccio di WebAssembly non rappresenta nulla di innovativo, dato che linguaggi interpretati come Java hanno già affrontato questa problematica. Tuttavia, bisogna considerare che WebAssembly non è legato a nessun linguaggio di programmazione specifico, ma rappresenta un formato binario generico indipendente. Al contrario, il bytecode di Java (metalinguaggio simile al .wat di Wasm) è strettamente legato al linguaggio stesso e rappresenta quindi un ostacolo alla portabilità del codice. Questo problema riguarda anche altri linguaggi interpretati, prendiamo come altro esempio Python, che utilizza un approccio simile generando il bytecode prima di poter eseguire il codice sorgente nella sua apposita macchina virtuale. Anche in questo caso, il bytecode di Python è strettamente legato al linguaggio stesso, impedendo una piena portabilità del codice.

In secondo luogo Wasm non dipende da una azienda privata, è uno standard del World Wide Web Consortium (W3C)² e quindi non segue una sola linea di pensiero per la sua sua evoluzione.

Infine, c'è da considerare come WASI affronta in maniera diversa dagli altri un'altro aspetto fondamentale: la sicurezza.

²<https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en>

1.3.2 Sicurezza

Abbiamo precedentemente descritto come le applicazioni accedono alle risorse del sistema tramite le system call, richiedendo al kernel le operazioni desiderate. Tuttavia, questo modello non garantisce una protezione completa da eventuali minacce, poiché l'accesso alle risorse viene concesso in base ai permessi dell'utente che sta eseguendo l'applicazione. Questo approccio, nato agli albori dei sistemi operativi, si adattava bene alle esigenze dell'epoca, quando le applicazioni venivano controllate ed installate dagli amministratori di sistema. Tuttavia, con la diffusione di internet, molte applicazioni, la cui origine non può sempre essere verificata con certezza, vengono eseguite sui computer degli utenti finali. Questo comporta un elevato rischio di vulnerabilità e di compromissione della sicurezza del sistema. Il codice Wasm, invece, viene eseguito in un ambiente isolato e controllato separato dal sistema operativo, chiamato "sandbox". In questo ambiente, il codice non ha accesso diretto alle risorse del sistema e non può interagire direttamente con il livello sottostante. Al contrario, deve passare attraverso l'host (il browser o il runtime Wasm) che contiene le funzioni necessarie per farlo. Questo permette all'host di limitare ciò che un programma può fare a priori. In altre parole, un'applicazione non viene eseguita con gli stessi privilegi dell'utente e quindi non può direttamente invocare le system call. Questo modello si basa sul concetto di "capacità" che definisce ciò che un programma Wasm può fare e sono definite prima dell'esecuzione dell'applicazione. Questo modello, noto come "capability-based model", fornisce una maggiore sicurezza e limita il rischio di compromissione del sistema.

1.4 Stato dell'arte

Per poter cogliere appieno la direzione verso cui si stanno muovendo Wasm e WASI, è necessario considerare l'attuale panorama tecnologico.

1.4.1 Chi supporta Wasm?

Dal punto di vista di un linguaggio di programmazione, Wasm non è altro che un altro target di compilazione.

- Per qualsiasi linguaggio compilato, l'ostacolo per l'esecuzione in un contesto WebAssembly è rappresentato dalla possibilità o meno di compilare i programmi nel formato binario Wasm.
- Per qualsiasi linguaggio di scripting, l'ostacolo per l'esecuzione in un contesto WebAssembly è la possibilità di compilare l'interprete in WebAssembly.

Fortunatamente, il supporto a WebAssembly sta crescendo rapidamente. Inizialmente, solo C/C++ e Rust erano in grado di generare codice .wasm, ma ora sempre più linguaggi stanno iniziando a supportarlo. Di seguito è riportata una tabella che riassume il supporto dei linguaggi più popolari[3]. È divisa in tre categorie: *Browser* (supporto nativo nei browser) e in *ambiente WASI*.

Linguaggio	Browser	WASI
Javascript	Wip	Wip
Python	Wip	Si
Java	Si	No
PHP	Si	Si
C# e .NET	Si	Si
Ruby	Si	Si
Swift	Si	Si
Typescript	No	No
C/C++	Si	Si
Rust	Si	Si
Go	Si	Si
Kotlin	Si	Wip

Tabella 1.1: Supporto a Wasm

1.4.2 A che punto è lo standard

Nel caso di WASI, ogni nuova interfaccia inserita nello standard, nota come WASI API, segue un rigoroso percorso diviso in sei fasi³ prima di essere standardizzata:

- **Pre-Proposal:** presentazione dell'idea iniziale e valutazione della sua fattibilità e rilevanza dalla comunità degli sviluppatori.
- **Feature Proposal:** presentazione dettagliata di una nuova funzionalità o modifica all'API esistente, valutata dalla comunità degli sviluppatori.
- **Spec Text Available:** scrittura del testo di specifica proposto con la descrizione dettagliata di come la nuova funzionalità o modifica dell'API dovrebbe essere implementata.
- **Implementation Phase:** fase di implementazione, in cui gli sviluppatori creano un'implementazione funzionante della funzionalità o modifica dell'API.
- **Standardize:** la nuova funzionalità o modifica dell'API viene sottoposta a una revisione formale per verificare che l'implementazione soddisfi i requisiti specificati nella proposta e nel testo della specifica proposto.
- **The Feature is Standardized:** la funzionalità o modifica dell'API viene aggiunta alla specifica ufficiale e diventa disponibile per gli sviluppatori.

Attualmente, nessuna delle nuove interfacce aggiunte a WASI ha superato la fase 2⁴ del processo di standardizzazione. Tuttavia, questo non dovrebbe sorprendere poiché gli standard si muovono lentamente per garantire la sicurezza e la stabilità delle tecnologie. Nonostante ciò, c'è la necessità di trovare un equilibrio tra la lentezza dei processi di standardizzazione e la necessità di mantenere il passo con l'evoluzione

³<https://github.com/WebAssembly/meetings/blob/main/process/phases.md>

⁴<https://github.com/WebAssembly/WASI/blob/main/Proposals.md>

della tecnologia. Il rischio che si corre è che il tempo di attesa per la standardizzazione di una nuova interfaccia possa essere troppo lungo per alcune realtà che hanno bisogno di determinate funzionalità nell'immediato. Ciò può portare a una divisione tra le realtà che seguono lo standard e quelle che non lo seguono. Attualmente, ci sono molti runtime che permettono di eseguire codice Wasm al di fuori del browser e molti di questi seguono le interfacce definite dallo standard WASI in modo più o meno stretto. Tuttavia, dove lo standard non è ancora pronto, alcune realtà prendono la loro strada, come nel caso dell'API per le socket che è ancora in fase 1 (Feature Proposal) ma già disponibile in alcuni runtime che ne hanno bisogno. Questo può portare a una frammentazione all'interno dello stesso standard e, di conseguenza, alla creazione di una divisione tra realtà che dovrebbero essere interoperabili. Un esempio di runtime che segue questa filosofia è WasmEdge⁵, che adotta lo standard WASI per le API già mature, ma prende decisioni diverse per quanto riguarda i moduli ancora in fase di discussione. È importante sottolineare questo aspetto in quanto pone un pericolo per la stabilità futura del progetto.

1.4.3 Runtime WASI

Wasmtime

Wasmtime⁶ è il runtime ufficiale di WASI ed è un progetto gestito dalla Bytecode Alliance⁷, l'organizzazione leader nello sviluppo di WebAssembly e WASI che riunisce le più grandi aziende presenti nel panorama informatico, come Amazon, Google, Microsoft, Cisco, Mozilla per citare le più importanti. Grazie a questa posizione privilegiata, Wasmtime segue scrupolosamente lo standard WASI senza discostarsi minimamente da esso. Questo runtime supporta diversi linguaggi di programmazione, tra cui Rust, C/C++, Python, .NET e Go. Per tradurre il codice .wasm in codice nativo per l'architettura sottostante utilizza Cranelift, un compilatore sviluppato dalla stessa Bytecode Alliance.

Wasmer

Wasmer, come Wasmtime, è stato progettato per aderire strettamente allo standard WASI. Una delle caratteristiche più interessanti di Wasmer è l'introduzione di un package manager per i moduli WebAssembly, chiamato WAPM (WebAssembly Package Manager). WAPM è un registro pubblico di pacchetti WebAssembly, simile ai package manager come npm per JavaScript o pip per Python. Consente agli sviluppatori di trovare, installare e distribuire pacchetti WebAssembly in modo semplice e veloce. Wasmer, oltre a Cranelift, supporta anche LLVM, uno dei compilatori più popolari e ampiamente utilizzati in ambito di sviluppo di software.

⁵<https://github.com/WasmEdge/WasmEdge>

⁶<https://wasmtime.dev/>

⁷<https://bytecodealliance.org/>

WasmEdge

WasmEdge⁸ è un runtime che pone il focus sul cloud computing e l'edge computing in particolare e per questo, come anticipato prima, non segue lo standard WASI in modo rigoroso.

È interessante notare che tutti e tre i runtime sono Open Source.

1.4.4 I container Wasm

L'evoluzione del cloud computing ha seguito un percorso di innovazione e miglioramento costante, divisibile in tre grandi fasi che hanno introdotto nuove tecnologie e strumenti per l'esecuzione delle applicazioni in ambiente cloud. Partiamo dalla fase zero, quella in cui ogni applicazione veniva eseguita su una macchina dedicata, con il proprio sistema operativo, risorse hardware e configurazione. Questa soluzione risultava essere molto costosa in termini di manutenzione e gestione, oltre che poco flessibile. La prima fase ha visto quindi l'adozione delle Macchine Virtuali (VM) come strumento principale per l'esecuzione delle applicazioni. Tuttavia, le VM sono considerate pesanti perché sono in tutto e per tutto sistemi operativi virtualizzati su altri sistemi, il che le rende più lente e meno flessibili rispetto ad altre soluzioni. Questo ha portato all'evoluzione successiva, caratterizzata dall'avvento dei container, che rappresentano un'unità standardizzata di software che racchiude tutto il necessario per eseguire un'applicazione in modo isolato e portatile, senza dipendere dal sistema operativo sottostante. I container condividono il Kernel del sistema operativo ospitante, rendendoli più veloci e leggeri delle VM e quindi ideali per l'esecuzione di applicazioni in ambiente cloud. Infine, quella che potrebbe considerarsi la terza fase dell'evoluzione del cloud computing è stata annunciata di recente da Docker, che ha introdotto il supporto ai container wasm[4]. I container wasm sono ancora più leggeri e portabili dei container tradizionali, offrendo una maggiore efficienza, scalabilità e flessibilità nell'esecuzione delle applicazioni in ambiente cloud.

⁸<https://wasmedge.org/>

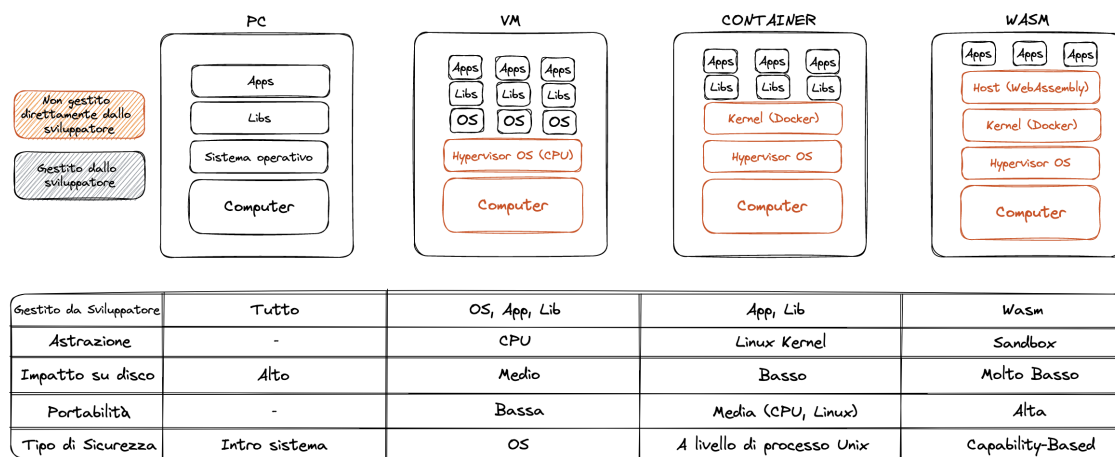


Figura 1.6: Evoluzione del cloud computing

If WASM+WASI existed in 2008, we wouldn't have needed to create Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!

*Solomon Hykes - Former Docker Founder*⁹

Significa che Wasm rimpiazzerà completamente docker? No, ma è un'ulteriore tecnologia che si aggiunge a quelle già esistenti, con i propri punti di forza e di debolezza.

"So will wasm replace Docker?" No, but imagine a future where Docker runs linux containers, windows containers and wasm containers side by side. Over time wasm might become the most popular container type. Docker will love them all equally, and run it all :)

*Solomon Hykes - Former Docker Founder*¹⁰

⁹<https://twitter.com/solomonstre/status/1111004913222324225>

¹⁰<https://twitter.com/solomonstre/status/1111113329647325185>

Capitolo 2

WASI

L'obiettivo principale di WASI è fornire un set di API standard per WebAssembly, indipendenti dall'architettura sottostante. Per raggiungere questo obiettivo è stato necessario creare, in prima istanza, un primo modulo su cui basare tutti gli altri. Questo modulo è stato chiamato WASI-core e ha lo scopo di fornire le funzionalità di base per l'ambiente di esecuzione, come la gestione dei file, delle reti e di essere generico in modo da poter essere utilizzato in qualsiasi ambiente fuori dal browser. Al di fuori del contesto web, WebAssembly ha bisogno di una serie di API per interagire con il sistema operativo sottostante, ovvero una libreria di sistema per Wasm. Questa libreria, chiamata WASI-libc è in grado di fornire un ponte tra i moduli Wasm e le system call del sistema sottostante. È basata sullo standard POSIX[5], in particolare su musl-libc¹ ed utilizza le funzioni di libpreopen².

2.1 WASI-libc

WASI-libc³ è una libreria che definisce un'interfaccia C standard per le system call necessarie a WebAssembly. È il ponte fondamentale che unisce i moduli Wasm con il sottostante sistema operativo. L'obiettivo chiave della libreria è quello di fornire un set di funzioni che si comportino come quelle dello standard POSIX, ma che siano implementate specificamente per WebAssembly tra cui l'allocazione e la manipolazione della memoria, l'I/O dei file, la gestione delle stringhe, le funzioni matematiche e così via. Essendo un'implementazione personalizzata, non include tutte le funzioni standard di POSIX. Ad esempio funzioni come la `fork()` e l'`exec()` sono escluse in quanto complesse da implementare in ambito sandbox e difficili da gestire secondo il modello capability-based. La libreria è basata su due altre librerie già esistenti: la musl-libc e la libpreopen.

¹<http://musl.libc.org/>

²<https://github.com/musec/libpreopen>

³<https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md>

2.1.1 Musl-libc

Musl-libc è una libreria standard del linguaggio C che fornisce una serie di funzioni predefinite per semplificare lo sviluppo di software. La libreria è stata progettata per essere leggera, efficiente e altamente portabile, ed è stata sviluppata con un focus particolare sulla compatibilità POSIX. Si propone come un'alternativa più leggera e veloce rispetto alla GNU C Library (glibc). Tra le funzionalità supportate da musl-libc vi sono la gestione della memoria, le operazioni su stringhe, la gestione dei file, i socket di rete e la gestione dei processi.

2.1.2 Libpreopen

Libpreopen è una libreria che fornisce un meccanismo per caricare in anticipo e intercettare le operazioni sui file in un sistema operativo. Consente alle applicazioni di aprire i file utilizzando un insieme predefinito di regole e percorsi, anziché il percorso effettivo sul filesystem. Libpreopen intercetta le chiamate di sistema relative, come `open()`, `stat()` e `opendir()`, e le traduce in operazioni definite dalle applicazioni. Consente all'applicazione di specificare un insieme di regole che definiscono come i file devono essere accessibili. Ad esempio, un'applicazione potrebbe specificare che tutte le operazioni sui file dovrebbero essere eseguite in una directory o file system montato specifico, o che determinati file dovrebbero essere in sola lettura o scrittura. Su questi concetti si basa il capability-based security di WASI ed è perciò di fondamentale importanza per il suo funzionamento.

2.2 La Portabilità

Nell'introduzione abbiamo discusso del ruolo di WASI come standard che consente di eseguire il codice WebAssembly su sistemi operativi e architetture diverse, in modo efficiente e al di fuori del contesto del browser. Ma quali sono i meccanismi sottostanti che consentono a un binario WebAssembly di essere eseguito su così tante piattaforme diverse? Innanzitutto, è importante comprendere che i processori possono eseguire solo il linguaggio macchina specifico per la propria architettura. In questo contesto, il codice Wasm deve essere tradotto in un linguaggio macchina che la CPU di destinazione sia in grado di comprendere ed eseguire. Questo passaggio è eseguito dai compilatori all'interno dei runtime.

2.2.1 Un accenno ai compilatori

Prima di continuare è utile capire come funziona il processo di compilazione in generale[6]. È importante distinguere le due fasi principali della compilazione: l'analisi e la sintesi. La fase di analisi, o front-end, rappresenta la prima fase della compilazione. In questa fase, il compilatore analizza il codice sorgente e crea una rappresentazione interna del programma, chiamata "albero sintattico" o "AST" (Abstract Syntax Tree). L'albero sintattico rappresenta il codice sorgente in modo strutturato, gerarchico e viene utilizzato per generare un linguaggio intermedio chiamato IR

(intermediate Representation). L'IR funge da ponte tra la fase di analisi e la fase di sintesi, o backend. Contiene informazioni sull'organizzazione del programma, come ad esempio le variabili, le funzioni e le istruzioni, ma è ancora abbastanza astratto da poter essere utilizzato per la generazione di codice per diverse architetture. Il frontend, dopo aver ottimizzato l'IR per migliorarne le prestazioni lo passa al backend per la generazione di codice specifico per l'architettura di destinazione. Il backend traduce l'IR in istruzioni specifiche per la CPU e genera il codice eseguibile.

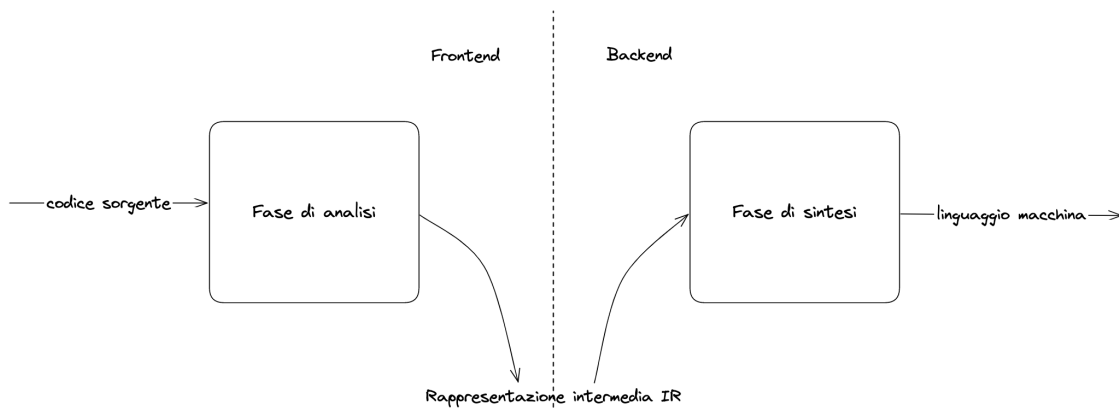


Figura 2.1: Una overview sul funzionamento dei compilatori

In ambito Wasm, troviamo diverse strategie di compilazione, tra cui JIT (Just In Time) e AOT (Ahead Of Time). I compilatori JIT traducono il codice Wasm in linguaggio macchina durante l'esecuzione del programma, appena prima che il codice venga eseguito. Questo significa che il compilatore può ottimizzare il codice in base alle informazioni disponibili solo a tempo di esecuzione. L'approccio JIT può portare a prestazioni migliori rispetto all'AOT, in quanto il compilatore ha maggiori informazioni a disposizione per l'ottimizzazione del codice. Tuttavia, la traduzione JIT richiede una certa quantità di tempo di elaborazione durante l'esecuzione del programma. D'altra parte, i compilatori AOT traducono il codice Wasm in linguaggio macchina prima dell'esecuzione del programma, solitamente al momento dell'installazione o del caricamento del modulo. Questo significa che il codice viene tradotto una sola volta, rendendo l'esecuzione del programma successiva più veloce, poiché il codice già tradotto viene eseguito direttamente. L'approccio AOT può essere particolarmente utile per applicazioni in cui la velocità di avvio è un fattore critico, come ad esempio negli ambienti cloud. Tuttavia, la traduzione AOT potrebbe non essere in grado di ottimizzare il codice in modo dinamico come il compilatore JIT, in quanto non ha informazioni sulle condizioni di esecuzione effettive del programma.

Ci sono vari tipi di compilatori, i più famosi ed usati dai runtime Wasm sono Cranelift e LLVM.

2.2.2 Cranelift

Cranelift⁴ è un compilatore altamente ottimizzante che è stato progettato specificamente per il bytecode di Wasm. Utilizzando l'IR intermedio CLIF (Cranelift IR Format), Cranelift è in grado di rappresentare il codice sorgente Wasm in modo strutturato e applicare una vasta gamma di ottimizzazioni durante la fase di compilazione. CLIF è stato progettato per essere facile da analizzare e trasformare, consentendo a Cranelift di effettuare una serie di trasformazioni, come la riscrittura di codice per migliorare le prestazioni o per renderlo più sicuro. Una volta che il codice Wasm è stato tradotto in CLIF, Cranelift applica una serie di fasi di ottimizzazione dopo le quali, viene convertito in un altro IR intermedio chiamato VCode. Il VCode è un'astrazione del codice macchina ed è specificatamente legato ad esso, il che significa che descrive le istruzioni in modo simile a come il processore le esegue, ma a un livello più alto di astrazione. Può essere generato per diverse architetture, come x86-64, AArch64 (ARM64), RISC-V e IBM z/Architecture. Infine, il VCode viene convertito in codice macchina specifico per l'architettura di destinazione.

L'utilizzo di due IR intermedie, come CLIF e VCode, consente al compilatore, tramite la prima traduzione, di eseguire una vasta gamma di ottimizzazioni sul codice sorgente prima della generazione del codice macchina finale. Mentre la conversione in un secondo IR intermedio, come VCode, permette al compilatore di generare il codice finale specifico per l'architettura di destinazione, indipendentemente dalla complessità dell'IR intermedio utilizzato per l'ottimizzazione del codice.

2.2.3 LLVM

LLVM⁵ è un framework di compilazione modulare, che fornisce un insieme di strumenti e librerie per la compilazione di programmi in diversi linguaggi di programmazione. È molto diffuso ed usato in altri ambiti al di fuori di Wasm. Analizza e trasforma il codice sorgente in LLVM IR, un linguaggio intermedio di basso livello che rappresenta il codice sorgente in modo strutturato e indipendente dall'architettura.

Dopo la fase di analisi, LLVM applica diverse fasi di ottimizzazione per migliorare le prestazioni e la sicurezza del codice generato. Le fasi di ottimizzazione possono essere personalizzate e configurate in modo da ottenere un equilibrio tra le prestazioni e la complessità del codice generato. Ad esempio, è possibile configurare LLVM per eseguire solo alcune ottimizzazioni di base per ridurre il tempo di compilazione, oppure è possibile configurarlo per applicare un set completo di ottimizzazioni per ottenere il massimo delle prestazioni possibili. Una volta completata la fase di ottimizzazione, LLVM genera codice macchina per l'architettura di destinazione.

2.2.4 La differenza

La differenza principale tra Cranelift e LLVM è nel modo in cui gestiscono l'IR intermedio durante il processo di compilazione. Cranelift utilizza due IR intermedi,

⁴<https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md>

⁵<https://llvm.org/>

il CLIF, che è stato progettato specificamente per la compilazione di WebAssembly e il VCode. Invece, LLVM utilizza il suo proprio IR intermedio, che è stato progettato per essere utilizzato con una vasta gamma di linguaggi di programmazione. Inoltre, mentre Cranelift è stato progettato specificamente per WebAssembly e supporta solo alcune architetture, LLVM è stato progettato per supportare diverse architetture di destinazione, tra cui x86, ARM, MIPS, PowerPC, RISC-V e altre. Ciò significa che LLVM può essere utilizzato per compilare codice sorgente in una vasta gamma di linguaggi di programmazione per diverse architetture di destinazione, mentre Cranelift è specificamente progettato per compilare codice WebAssembly in alcune architetture specifiche.

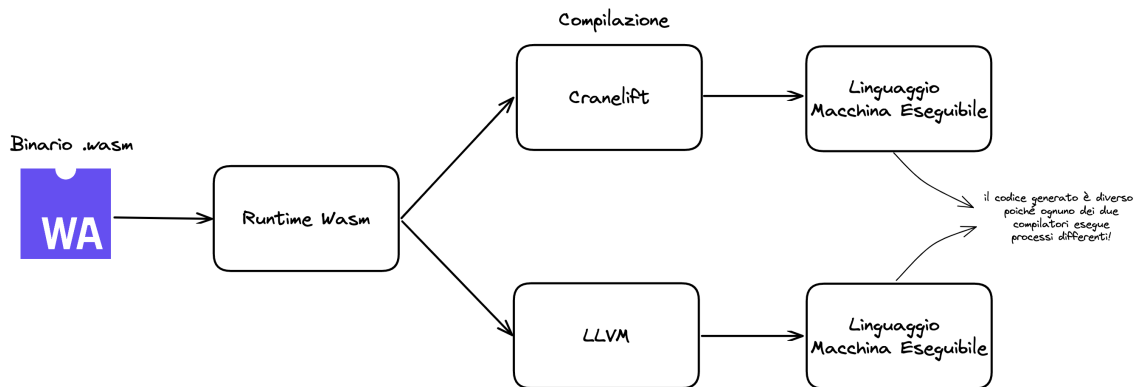


Figura 2.2: Dal runtime al linguaggio macchina

2.2.5 Linguaggi compilati ed interpretati

Quando si parla di linguaggi di programmazione, sia compilati che interpretati, la creazione di moduli .wasm rappresenta una sfida diversa per le due realtà. Nel caso dei linguaggi compilati, l'ostacolo principale consiste nel supporto del compilatore al formato .wasm, che richiede una specifica conoscenza e competenza nel lavorare con questo tipo di architettura. D'altra parte, per i linguaggi interpretati, la sfida principale è rappresentata dall'interprete stesso, che deve essere compilato in .wasm per eseguire il codice nativo. Tuttavia, è interessante notare che molti interpreti dei linguaggi più comuni sono scritti in C/C++, il che rende la compilazione in .wasm più semplice. Pertanto, nella maggior parte dei casi, i linguaggi interpretati possono essere usati per produrre moduli .wasm senza problemi.

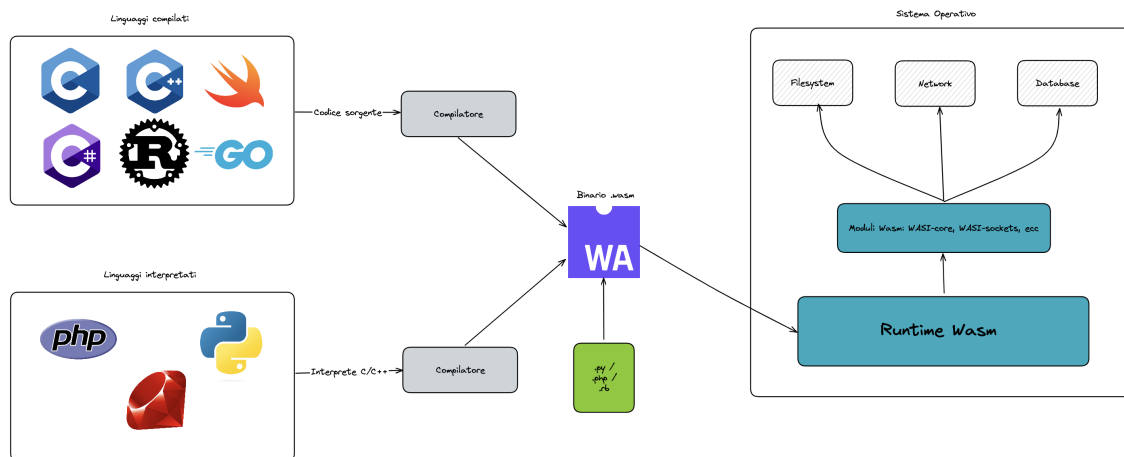


Figura 2.3: Linguaggi compilati e interpretati con Wasm

2.3 La gestione delle risorse

Dopo aver compreso che Wasm è un formato binario di esecuzione compilato da qualsiasi linguaggio di programmazione che lo supporti, e che WASI è uno standard che definisce interfacce per consentire ai moduli Wasm di comunicare in modo sicuro ed isolato dal sistema sottostante, ci concentriamo ora sul motivo per cui vengono definiti container leggeri e considerati la terza ondata del cloud computing.

Il cloud computing ha subito due evoluzioni significative. La prima ha visto il passaggio dalla gestione manuale dei server offrendo servizi alla creazione di macchine virtuali che ha permesso di ridurre i costi e la manutenzione dei servizi, garantendo l'isolamento delle applicazioni. Siccome una VM è essenzialmente un intero sistema operativo con all'interno diversi package, dati e applicazioni, occupa molti GB di memoria e non è adatta per essere scalata in modo veloce in quanto il boot time è nell'ordine dei secondi-minuti.

Poi c'è stato l'avvento delle immagini Docker e dei relativi container. I container sono un'unità software relativamente leggera e portatile che esegue le applicazioni in modo isolato, ma condividendo il kernel del sistema operativo sottostante. Ciò significa che, a differenza delle macchine virtuali, non richiedono l'esecuzione di un intero sistema operativo all'interno di un altro sistema operativo, il che rende la creazione e l'istanziamento molto più veloce rispetto alle VM; funzionalità che ha portato all'ascesa di strumenti di orchestrazione come Kubernetes⁶ che si occupa di gestire i carichi del sistema e di scalare automaticamente le applicazioni per garantire una risposta efficiente alle richieste in arrivo. Inoltre, i container offrono un'ulteriore vantaggio in quanto possono essere facilmente interconnessi tramite Docker Compose, strumento che permette di organizzare i container in gruppi logici e interconnetterli in modo veloce e affidabile, semplificando notevolmente la gestione di applicazioni complesse composte da diversi componenti.

⁶<https://kubernetes.io/docs/home/>

Per fare un esempio concreto, un'immagine Docker è composta da un sistema operativo leggero come Alpine Linux, all'interno del quale viene installata l'applicazione desiderata e le sue dipendenze. Quando si avvia un container, viene effettuato il boot di questo sistema operativo, vengono caricate le dipendenze e infine si esegue l'applicazione.

Si può quindi evincere che:

- Un'immagine docker è un intero filesystem
- Ogni immagine ha un sistema operativo: esistono immagini Linux e immagini Windows, non immagini generiche
- L'immagine docker dipende dall'architettura sottostante; un Mac con processore ARM può solo eseguire immagini compilate per ARM
- Le applicazioni devono tener conto del sistema operativo e dell'architettura dell'immagine Docker su cui verranno eseguite

Una tipica immagine Docker è più leggera di un'intera VM e tende anche ad usare meno risorse rispetto ad essa. Tuttavia possono arrivare facilmente a dimensioni nell'ordine delle centinaia di MB e dei GB.





<input type="checkbox"/>	mysql 2846b2a84d49 	8.0	In use	530.66 MB	▶	⋮	
<input type="checkbox"/>	mysql/mysql-server 423da140c8c0 	8.0	Unused	507.39 MB	▶	⋮	

Figura 2.4: Esempio di immagine Docker per mysql

WebAssembly invece viene definito come un container leggero poiché migliora il concetto dei container. Una delle ragioni che lo rende così leggero è il fatto che, a differenza delle immagini Docker, un modulo Wasm non richiede di essere inserito all'interno di un mini sistema operativo. Inoltre, grazie al suo formato neutrale, con l'ausilio di WASI, può essere eseguito su qualsiasi macchina supportata dal runtime utilizzato, che si occupa di gestire l'interazione, l'isolamento e il caricamento di tutte le risorse necessarie per il modulo.

2.3.1 Docker+Wasm

Di recente, Docker ha introdotto il supporto ai container Wasm[4], costituendo così un notevole passo avanti per l'adozione di Wasm nel mondo cloud. Grazie alla sua diffusione, Docker rende più agevole la distribuzione dei container. Tuttavia, i container Wasm su Docker operano in maniera leggermente differente rispetto ai container tradizionali.

In genere, su Docker, l'avvio e l'esecuzione dei container avviene tramite Containd, un runtime di container open source che funge da interfaccia tra il demone Docker e i container. Quando viene eseguito un comando Docker come "docker

run", il demone Docker comunica con Containerd per avviare un nuovo container. Containerd si occupa di gestire l'avvio e l'esecuzione dei container, compresa la creazione del filesystem, l'assegnazione di namespace di sistema isolati e l'aggiunta di eventuali opzioni di rete. Quando il container viene avviato, Containerd lo esegue in una sandbox, proteggendo il sistema host da eventuali problemi che potrebbero verificarsi all'interno del container. Per eseguirlo viene utilizzato Runc, un runtime di container che implementa lo standard Open Container Initiative (OCI). Questo viene eseguito all'interno di un "container shim", un piccolo processo che funge da ponte tra Runc e Containerd. Il container shim svolge il compito di trasferire i comandi tra Containerd e Runc, gestendo inoltre eventuali segnalazioni di errori tra i due processi. Questa divisione dei ruoli è necessaria per garantire che il container venga eseguito in modo isolato dal resto del sistema.

La principale differenza tra un container tradizionale e uno Wasm risiede nella loro architettura. Infatti, mentre un container tradizionale necessita di un sistema operativo leggero e dipende dal kernel dell'host per funzionare, uno Wasm è basato su un'architettura sandboxed e isolata gestita da un runtime. Ciò significa che i container Wasm non richiedono un sistema operativo gestito all'interno del container, ma sono semplicemente composti da un modulo .wasm, il che li rende più leggeri. Per eseguire un container Wasm su Docker, è necessario utilizzare una versione modificata del containerd-shim chiamata containerd-wasm-shim che è versione appositamente progettata per gestire l'architettura sandbox di Wasm e serve da ponte tra Docker e WasmEdge, che è il runtime Wasm utilizzato per eseguire l'applicazione al posto di Runc.

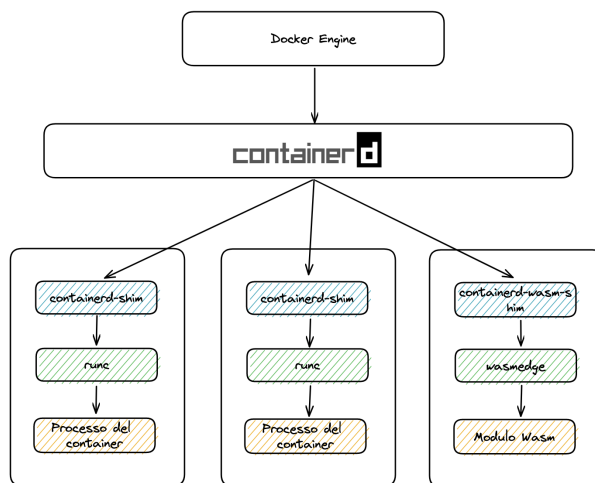


Figura 2.5: Container tradizionali e container Wasm su Docker

Wasm al posto di Docker?

Viene da chiedersi a questo punto se Wasm prenderà il sopravvento su Docker[7]. In generale possiamo dire di no, entrambi hanno i loro punti di forza e debolezza. I container sono utili in quei contesti in cui la portabilità è importante ma si vuole anche mantenere un forte controllo sul sistema mentre Wasm permette lo sviluppo di applicazioni sicure di default con pochissimo overhead aggiuntivo. Si possono

intendere come tecnologie complementari, che in alcuni contesti possono coesistere come vedremo nel terzo capitolo del documento.

2.3.2 Uno use case: i microservizi con Wasm

Il problema

Consideriamo ora Wasm in uno scenario reale: quello dei microservizi[8] riprendendo la spiegazione fornita riguardante la sicurezza in WASI. Nella costruzione di un'applicazione è ormai la norma utilizzare codice scaricato da internet attraverso i package manager. Questo può causare non pochi problemi di sicurezza, soprattutto se si tiene conto del fatto che ogni pacchetto richiede a sua volta altre dipendenze per funzionare. Prendiamo come esempio NodeJS, dove per creare un'applicazione ad architettura a microservizi, potremmo utilizzare il framework ExpressJS per aggiungere le funzionalità necessarie, come il routing, il middleware, la gestione dei file, l'integrazione dei database, e così via. Tuttavia, questo significherebbe anche installare tutte le dipendenze utilizzate da ExpressJS, che potrebbero essere circa un centinaio. Prima di iniziare a scrivere una sola riga di codice avremmo già importato migliaia di altre righe di codice che potrebbero presentare vulnerabilità. Poi, per eseguire il nostro servizio, avremmo bisogno di un server HTTP che andrebbe eseguito per ogni microservizio e che avrebbe bisogno di rimanere sempre in ascolto di nuove connessioni. Nel caso di un servizio particolarmente utilizzato, avremmo anche bisogno di duplicare le istanze del microservizio per gestire il traffico. Tutto questo comporterebbe un aumento dei costi, poiché più pesante è il nostro codice, maggiori le risorse necessarie in termini di storage e utilizzo della CPU. Inoltre, la maggior parte del tempo, a meno di un servizio super trafficato, staremo probabilmente pagando per un servizio in esecuzione, anche se in idle, replicato.

L'adozione di questa metodologia comporta l'acquisizione di un debito tecnico prima ancora di iniziare a scrivere una sola riga di codice. Infatti, installando le dipendenze che gestiscono il core della nostra applicazione, come il server HTTP, l'implementazione TLS e l'interazione con il sistema sottostante, ci troviamo a dover gestire la sicurezza dell'intero sistema ospitante, inclusi i permessi corretti nel sistema. Inoltre, se dovesse emergere una vulnerabilità di sicurezza in una di queste dipendenze, saremo costretti a risolverla da soli, poiché un fix potrebbe non essere disponibile in tempi brevi. Fortunatamente, nel corso degli anni sono state sviluppate soluzioni come le PaaS o le IaaS, in grado di gestire e alleviare gli sviluppatori dal fardello di questa gestione low-level.

Tuttavia, queste soluzioni sono spesso dipendenti dai provider cloud di turno, i quali effettuano scelte in base alle proprie necessità. Ad esempio, un servizio PaaS come AWS Elastic Beanstalk può differire dall'omonimo Google App Engine, entrambi servono scopi simili ma con modalità differenti.

La soluzione

Wasm offre una soluzione in grado di eliminare la necessità di interagire con il sistema ospitante, il web server HTTP, il livello TLS e così via. Queste operazioni sono

comuni a tutti i web framework, indipendentemente dal linguaggio di programmazione utilizzato, che sia questo Javascript, Rust o Java. Grazie a Wasm, possiamo concentrarci sulle funzionalità della nostra applicazione, senza dover gestire l'infrastruttura sottostante e la sua comunicazione con essa. Anche se le dipendenze introdotte nell'applicazione sono ancora a carico nostro, sarebbe il runtime sottostante a gestire gran parte del lavoro in quanto ci garantirebbe l'isolamento e la sicurezza dell'esecuzione. Inoltre, la nostra applicazione con Wasm può scalare veramente a zero, poiché un solo server HTTP gestito dal runtime sarebbe in grado di redirigere le richieste al servizio adatto con tempi di risposta rapidi in quanto i nostri microservizi non sono altro che minuscoli binari .wasm. Combiniamo queste caratteristiche al fatto che Wasm ci permette di usare il linguaggio di programmazione che vogliamo e abbiamo una tecnologia estremamente versatile per quanto riguarda lo sviluppo in cloud.

Il terzo capitolo metterà un focus proprio su questi concetti appena introdotti per dimostrare l'effettivo valore di Wasm in ambiente cloud.

2.4 La sicurezza con WASI

2.4.1 Il problema

Nell'ambito dello sviluppo software, l'utilizzo di librerie esterne è diventato una pratica comune per facilitare la creazione di applicazioni complesse. Queste librerie open source, disponibili sotto forma di pacchetti software, sono facilmente accessibili tramite l'uso di appositi strumenti chiamati package manager. I package manager sono strumenti di gestione del software che consentono di gestire, salvare e distribuire le librerie di codice necessarie alla creazione di un'applicazione. Inoltre, consentono di installare automaticamente tutte le dipendenze richieste per far funzionare un software, semplificando notevolmente il processo di integrazione di nuove funzionalità all'interno di un'applicazione. Con l'utilizzo di un package manager, gli sviluppatori possono concentrarsi maggiormente sullo sviluppo di funzionalità specifiche, piuttosto che sulla gestione delle dipendenze. In questo modo, è possibile ridurre il

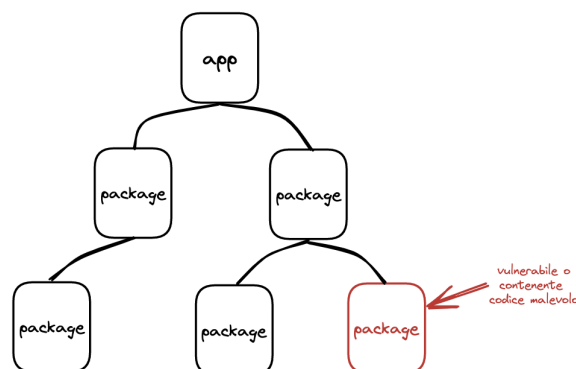


Figura 2.6: Come sono composte le applicazioni moderne

lavoro necessario per la creazione di applicazioni complesse, migliorare l'efficienza

del processo di sviluppo e aumentare la qualità del software prodotto. D'altra parte, questo modello di sviluppo espone le applicazioni ad una grossa problematica, cosa succederebbe se in uno di questi pacchetti software ci fosse inserito codice malevolo o codice vulnerabile? Purtroppo non è uno scenario remoto:

The average application development project has 49 vulnerabilities and 80 direct dependencies (open source code called by a project);
Snyk 2022 State of Open Source Security Report[9]

Per peggiorare le cose, non tutte queste vulnerabilità vengono risolte, si stima che solo il 59% dei pacchetti software risolva le vulnerabilità trovate e in media ci impieghi circa 110 giorni⁷.

Time to fix for proprietary projects vs. open source projects

Project type:

■ Proprietary ■ Open source



Figura 2.7: Tempo necessario per risolvere le vulnerabilità, un confronto tra open source e codice proprietario

2.4.2 Una possibile soluzione

Esistono diverse strategie che gli sviluppatori possono adottare per proteggere il loro software e gli utenti finali. Ad esempio, si potrebbe utilizzare uno scanner per

⁷<https://snyk.io/reports/open-source-security>

controllare le applicazioni e le dipendenze, tuttavia questa soluzione non è sempre efficace, poiché gli scanner sono facilmente aggirabili. Un'altra opzione potrebbe essere quella di registrarsi ad un servizio che notifichi gli sviluppatori delle vulnerabilità trovate nel codice, ma anche questa soluzione presenta delle limitazioni, in quanto potrebbe non individuare tutte le vulnerabilità presenti nel software. Infine, un'altra strategia possibile sarebbe quella di revisionare il codice ogni volta che si fa un update delle dipendenze, ma questa soluzione potrebbe funzionare solo per progetti piccoli con poche dipendenze, mentre per progetti più grandi e complessi potrebbe risultare impraticabile. Queste soluzioni esistenti cercano di individuare le vulnerabilità del software, ma non offrono una vera e propria prevenzione. La soluzione ideale sarebbe quella di prevenire tali vulnerabilità alla radice, ma è più facile a dirsi che a farsi. E se avessimo un modo per limitare le applicazioni e i loro package in modo che siano confinati dentro ad una sorta di contenitore chiuso che non permetta di disturbare le altre applicazioni del sistema e quindi di causare danni? Possiamo farlo attraverso molti modi, molti dei quali sono già stati affrontati in passato.

Partiamo nel capire come due applicazioni non si interferiscono durante la loro esecuzione. È una soluzione che esiste fin dai primi sistemi operativi. La soluzione adottata è affidare al sistema operativo il compito di proteggere e controllare l'esecuzione delle applicazioni tramite l'utilizzo dei "processi". Quando una nuova applicazione viene avviata, il sistema operativo crea un nuovo processo con un'area di memoria dedicata, che non può accedere alle aree riservate degli altri. Se ha bisogno di comunicare con altri, deve prima richiedere il permesso e farlo tramite le pipe.

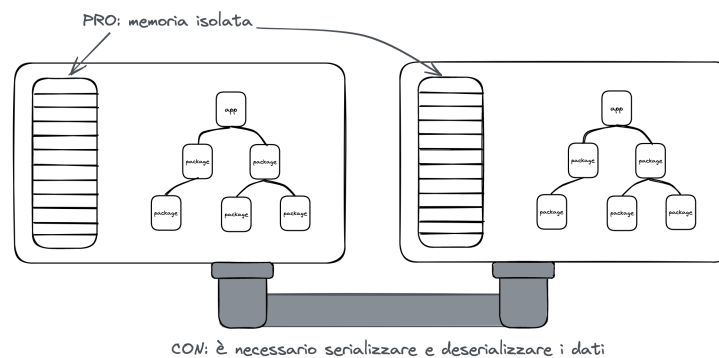


Figura 2.8: Isolamento dei processi nei sistemi operativi

Questo sistema risolve il problema della condivisione della memoria a runtime ed evita che un eventuale applicazione manipoli la memoria di altri, ma non garantisce, ad esempio, che non possa accedere al filesystem per effettuare qualche operazione non prevista.

Le VM e i container sono stati sviluppati in origine proprio per questo motivo, garantiscono che un'applicazione in esecuzione in una VM o container non possa accedere al filesystem di altri degli altri in esecuzione sulla stessa macchina. Questo però non risolve del tutto un'altro problema, l'applicazione in un container o in una VM è comunque in grado di accedere al proprio sistema e causare danni al suo interno. Se volessimo proprio vietare anche determinate azioni all'interno di uno stesso sistema? Con il modello sandbox, potremmo farlo isolando le applicazioni,

rimuovendo l'accesso alle API e alle system call su questo. Potremmo perciò arrivare alla conclusione che per garantire la sicurezza e l'isolamento di ogni package sia necessario isolarlo all'interno della propria sandbox. Ciò però risulterebbe inefficiente in quanto porterebbe presto ad un esaurimento delle risorse del sistema ed introdurrebbe un overhead nella comunicazione tra le componenti dell'applicazione.

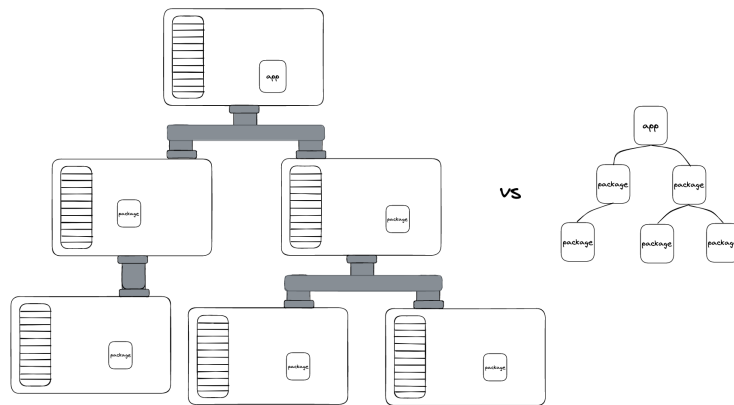


Figura 2.9: Isolamento dei package in sandbox isolate, evidente inefficienza e overhead

Come facciamo quindi a garantire l'isolamento di ogni package senza esaurire le risorse del sistema?

2.4.3 La soluzione: i nanoprocessi di WebAssembly

WebAssembly offre un'efficace forma di isolamento che previene al codice di operare a proprio piacimento, i nanoprocessi[10]. I nanoprocessi sono strutture simili ai processi standard di Unix ma che risultano essere più leggeri e di dimensioni ridotte. Risiedono all'interno di un ambiente di esecuzione chiamato sandbox, che rappresenta un processo contenitore isolato dall'esterno. La memoria a cui ogni nanoprocesso ha accesso è uno slice della memoria della sandbox e per accedere ai dati di un altro modulo deve avere l'autorizzazione esplicita per farlo. Questa autorizzazione viene passata in modo gerarchico, dall'alto verso il basso a partire dalla sandbox che a sua volta ha ricevuto i permessi all'avvio. In questo modo, la sandbox coordina l'accesso ai dati e garantisce che ogni nanoprocesso operi in modo indipendente e sicuro, senza che eventuali package vulnerabili possano causare danni all'esterno. L'uso di nanoprocessi consente di ottimizzare le prestazioni dell'applicazione, evitando costose chiamate di sistema e semplificando la comunicazione tra i vari moduli Wasm. Tutte le caratteristiche di WebAssembly descritte finora rendono questa tecnologia una soluzione sicura per l'esecuzione di codice al di fuori del browser. Per esempio, se del codice malevolo tentasse di accedere a un file su cui la sandbox non ha i permessi, il modulo WebAssembly verrebbe interrotto sollevando un'eccezione e il processo terminerebbe con un errore. Inoltre, anche se la sandbox avesse i permessi per accedere a un determinato file, questi permessi potrebbero non essere associati al modulo contenente il codice malevolo e anche in questo caso il processo verrebbe interrotto. In relazione al codice vulnerabile, sarebbe estremamente difficile

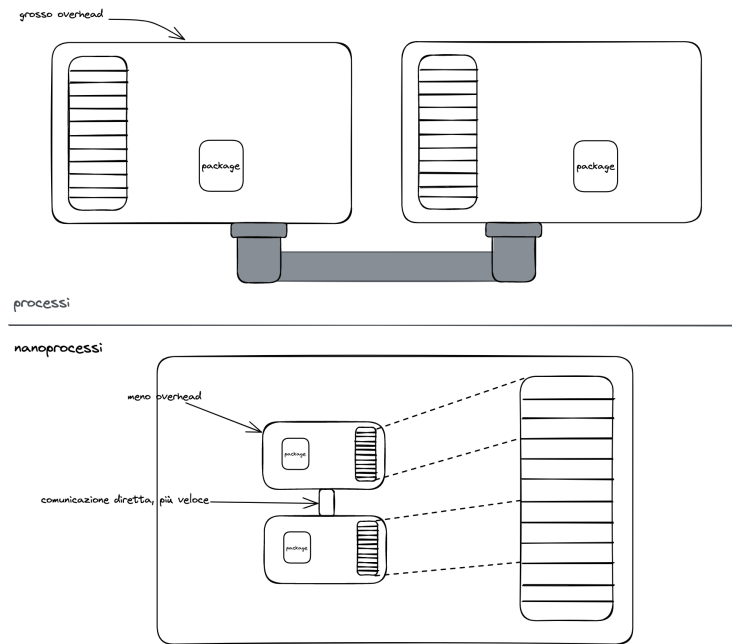


Figura 2.10: Nanoprogrammi di Wasm

per un attaccante trovare un modulo che abbia sia l'autorizzazione a utilizzare una determinata system call sia l'autorizzazione su un determinato file nel filesystem.

2.4.4 Capability Based Model

Il modello basato sulle capacità è un approccio alla sicurezza informatica che mira a garantire la protezione dei sistemi mediante la concessione di autorizzazioni specifiche, chiamate "capacità", ai processi del sistema. Questo modello è stato concepito per la prima volta negli anni sessanta[11] e si basa su tecniche di sviluppo che mirano ad assegnare privilegi minimi[12] ai processi per consentire loro di eseguire solo le operazioni necessarie per svolgere il loro compito specifico. È un'alternativa alle ACL (Access Control List)[13] che sono un metodo di controllo degli accessi basato su liste di controllo degli accessi, dove ogni oggetto (come file, cartelle, stampanti, ecc.) ha una lista di controllo degli accessi associata che contiene una serie di voci per specificare quali utenti o gruppi hanno il permesso di accedere all'oggetto e quali tipi di accesso sono consentiti (come lettura, scrittura, esecuzione, ecc.). In un modello di sicurezza basato su capability, ogni processo ha un insieme di capacità specifiche che gli consentono di accedere alle risorse del sistema. Una capacità non è altro che una chiave di autorità, associata ad un particolare oggetto. Possiamo pensarla come un indice che punta ad una tupla[14] (x, r) , dove x è l'oggetto ed r è l'insieme dei diritti d'accesso su di esso. Ad esempio, se la tupla $(\text{file1}, \{\text{leggi}, \text{scrivi}\})$ è associata ad un processo tramite una reference mantenuta da questo, tale processo può leggere e scrivere sul file1, ma non può eseguire altre azioni su di esso. Quando un processo deve accedere all'oggetto, può farlo senza richiedere l'autorizzazione al sistema operativo, in quanto il possesso della capability (la reference) rappresenta già l'autorizzazione per farlo. In un modello basato sulle capabilities, il sistema operativo non è responsabile del controllo delle autorizzazioni per i processi,

ma piuttosto della gestione della lista di capabilities disponibili e della prevenzione di eventuali modifiche non autorizzate.

Un esempio

Supponiamo che un processo abbia in memoria il seguente path:

```
/etc/shadow
```

Questo rappresenta il nostro oggetto `x` della tupla e supponiamo poi di aver associato all'oggetto anche il diritto di lettura `r`:

```
O_RDWR
```

Così da avere la tupla `(/etc/shadow, O_RDWR)`. Questa non rappresenta ancora la capability per il processo di accedere all'oggetto poiché non si può verificare l'autorità su di esso. Ipotizziamo ora di avere anche il file descriptor per tale file.

```
int fd = open("/etc/shadow", O_RDWR);
```

La variabile `fd` contiene l'indice di un descrittore di file presente nella tabella dei descrittori di file del processo e ciò rappresenta la capability dell'oggetto a cui si sta accedendo. La presenza di questo descrittore nella tabella dei descrittori è sufficiente per garantire che il processo abbia accesso legittimo all'oggetto.

WASI adotta il modello di Nuxi CloudABI⁸, che unisce i concetti di POSIX e del modello capability based. In questo modello, ogni risorsa del sistema, come file, directory e socket, è identificata da un file descriptor UNIX-like, che rappresenta la capability su un certo oggetto del sistema. Ciò significa che un processo non può accedere a tale oggetto se non è in possesso della capability adatta. Per esempio, WASI non permette l'accesso alla system call `open()` che restituisce un file descriptor, ma al contrario, utilizza la `openat` system call, che richiede un file descriptor precedentemente aperto, ovvero la capability associata. Essenzialmente quindi, un modulo Wasm potrà interagire, ad esempio con il filesystem, solo se la sandbox gli fornirà in precedenza le capabilities necessarie su di una certa directory all'avvio. Questi file descriptor saranno aperti in anticipo dalla sandbox tramite la `Libpreopen` e passati ai moduli Wasm quando necessario.

Il modello in azione: un esempio

Per comprendere meglio questo modello, possiamo analizzare un'applicazione⁹ scritta in Rust¹⁰ che si occupa di copiare dei file del filesystem da una posizione ad un'altra. Per eseguire il modulo Wasm generato dalla compilazione del codice sorgente, utilizzeremo il runtime `Wasmtime`. Compiliamo il programma tramite `cargo`, il gestore di pacchetti predefinito di Rust.

⁸<https://github.com/NuxiNL/cloudabi>

⁹<https://github.com/ilcors-dev/bachelor-thesis/blob/main/prototypes/1.file/src/main.rs>

¹⁰<https://www.rust-lang.org/>

```
cargo build --target wasm32-wasi
```

E proviamo ad eseguirlo senza garantire le capability necessarie tramite.

```
wasmtime target/wasm32-wasi/debug/file.wasm tocopy.txt /tmp/out.txt
error opening input tocopy.txt: failed to find a pre-opened file
descriptor through which "tocopy.txt" could be opened
```

Cosa è andato storto? Analizzando l'errore, si evince che il programma sta tentando di aprire un file chiamato 'tocopy.txt', ma non dispone del file descriptor corrispondente necessario. In altre parole, Wasmtime non ha fornito le capability adatte per consentire all'applicazione di eseguire la copia del file.

Proviamo invece ora a fornire le giuste capability in questo modo:

```
wasmtime --dir=. --dir=/tmp target/wasm32-wasi/debug/file.wasm tocopy.
txt /tmp/out.txt
done!
```

Analizziamo cosa è successo. L'opzione `--dir` istruisce Wasmtime di pre-aprire la cartella corrente `'.'` e la cartella di destinazione `'/tmp'` rendendole entrambi disponibili al programma. Quando l'applicazione chiama la funzione `File::open` sta effettivamente chiamando la system call `open()` passando come parametro il path del file. In questo frangente interviene la WASI libc, che in modo trasparente, traduce il path in un path relativo a quello già pre-aperto in precedenza da Wasmtime utilizzando la libreria `Libpreopen`.

In questo modo WASI è in grado di adottare il modello capability based e portarlo a livello di system call senza che il programma in sé debba essere modificato in qualche modo particolare.

È importante anche notare che, di default, la sandbox non espone la variabile d'ambiente `$PWD` all'applicazione WebAssembly, la directory corrente va pertanto passata attraverso un path relativo. Infatti:


```
wasmtime --dir=.$PWD --dir=/tmp target/wasm32-wasi/debug/file.wasm
tocopy.txt /tmp/out.txt
error opening input tocopy.txt: failed to find a pre-opened file
descriptor through which "tocopy.txt" could be opened
```

Parlando di directory corrente con `'.'`, cosa succede con `'..'`, ovvero i path traversal? Funzionano? Sappiamo che in applicazioni tradizionali la sanificazione dell'input è a carico dello sviluppatore, mentre con WASI?

```
wasmtime --dir=. --dir=/tmp target/wasm32-wasi/debug/file.wasm tocopy.
txt /tmp/../etc/passwd
error opening output /tmp/../etc/passwd: failed to find a pre-opened
file descriptor through which "/tmp/../etc/passwd" could be opened
```

Come possiamo notare l'applicazione viene bloccata prima di raggiungere il sistema operativo, infatti l'errore non riguarda gli eventuali permessi necessari per accedere ad `/etc/passwd`, bensì parla della mancanza di capacità necessaria per effettuare l'operazione. Le applicazioni WASI non hanno possibilità di accedere a file al di fuori delle directory esplicitamente garantite dalla sandbox!

Ciò si traduce in applicazioni molto più sicure di default grazie alla riduzione della superficie di attacco disponibile.



```
1 use std::env;
2 use std::fs;
3 use std::io::{Read, Write};
4
5 fn process(input_fname: &str, output_fname: &str) → Result<(), String> {
6     // apertura dei file, stiamo interagendo con il filesystem
7     // è necessario che l'host dia le capability necessarie al programma e se non lo fa
8     // il metodo map_err() di rust causerà un errore a runtime facendo terminare il programma
9     let mut input = fs::File::open(input_fname)
10         .map_err(|err| format!("error opening input {}: {}", input_fname, err))?;
11
12     let mut contents = Vec::new();
13
14     input
15         .read_to_end(&mut contents)
16         .map_err(|err| format!("read error: {}", err))?;
17
18     let mut output = fs::File::create(output_fname)
19         .map_err(|err| format!("error opening output {}: {}", output_fname, err))?;
20
21     output
22         .write_all(&contents)
23         .map_err(|err| format!("write error: {}", err))
24 }
25
26 fn main() {
27     let args: Vec<String> = env::args().collect();
28     let program = args[0].clone();
29
30     if args.len() < 3 {
31         // interagisce con lo stderr tramite la system call fd_write
32         eprintln!("usage: {} <from> <to>", program);
33         return;
34     }
35
36     if let Err(err) = process(&args[1], &args[2]) {
37         eprintln!("{}", err);
38         return;
39     }
40
41     println!("done!");
42 }
43
```

Figura 2.11: Il modello capability based in azione

Capitolo 3

Prototipo

Andiamo ora a mettere in pratica ciò che è stato detto su WASI sviluppando un semplice prototipo di una chat¹ sviluppata secondo l'architettura a microservizi.

3.1 Architettura

Il prototipo può essere scomposto in tre layer distinti.

- Backend: rappresenta il cuore del prototipo, ed è strutturato secondo un'architettura a microservizi. Ogni microservizio è una REST API, è rappresentato da un modulo Wasm e si occupa di una funzionalità specifica dell'applicazione
- Frontend: rappresenta l'interfaccia grafica resa disponibile agli utenti. Si noti che non è strettamente necessaria in quanto grazie all'utilizzo della metodologia REST API è possibile interagire con l'applicazione tramite terminale o qualsiasi altro strumento che supporti le richieste HTTP.
- Persistenza: rappresenta il layer che gestisce e salva i dati dell'applicazione.

3.1.1 Le funzionalità

Le funzionalità dell'applicazione, implementate dal layer backend, sono le seguenti:

- Gestione delle chat: si occupa della gestione delle diverse chat presenti, permette la creazione e l'eliminazione di esse.
- Gestione dei messaggi: si occupa delle operazioni CRUD (create, read, update, delete) dei messaggi nelle chat oltre che della loro sincronizzazione in realtime per i clienti.
- Gestione delle sessioni: si occupa della gestione delle sessioni utente, ovvero gli identificativi di ogni cliente connesso. È estremamente importante in quanto ogni entità nell'applicazione si lega ad essa.

¹<https://github.com/ilcors-dev/bachelor-thesis/tree/main/poc>

- Gestione degli utenti connessi: è un servizio a fini statistici, in quanto raccoglie il numero degli utenti attualmente online e ne permette la lettura.
- Gestione del frontend: si occupa della fruizione dell'interfaccia grafica ai clienti dell'applicazione. Può essere visto come il filesystem dell'applicazione.

3.1.2 Le tecnologie backend

Rust

Rust è stato scelto in quanto linguaggio altamente performante e maturo nell'ambiente WebAssembly. È altamente efficiente grazie alla sua natura di basso livello, alla forte tipizzazione e al sistema di gestione della memoria. La forte tipizzazione garantisce una bassa probabilità di errori a runtime se comparato ad altri linguaggi di più alto livello. Si noti che avremmo potuto scegliere qualsiasi altro linguaggio che supporta WebAssembly per le ragioni di portabilità introdotte in precedenza.

Wasmtime

È il runtime utilizzato per eseguire i moduli Wasm e rappresenta il cuore dell'applicazione. Lo abbiamo già menzionato quando si è parlato della sicurezza con WASI. È stato scelto per via della sua conformità allo standard che lo rende una scelta affidabile e robusta per l'esecuzione dei nostri microservizi.

Spin Framework

È un framework basato su Wasmtime che integra in esso le basilari funzionalità necessarie agli scopi del prototipo come l'interazione con le richieste HTTP, il conseguente routing e la gestione della comunicazione con i database. È uno dei primi framework presenti nel panorama WASI e per questo motivo è stato scelto per facilitare la realizzazione del prototipo. Oltre a Rust supporta C/C++, Javascript, Python e Go.

Abbiamo in precedenza visto che WASI non espone ancora in modo stabile alcuna API per il layer networking², per questo motivo il framework utilizza un'implementazione custom per inizializzare un demone in ascolto alle richieste HTTP. Questo demone è gestito da WAGI (WebAssembly Gateway Interface)³ un'implementazione delle interfacce CGI[15] per WebAssembly, in grado anche di gestire le richieste in modalità multi-thread. Si noti che se anche fosse possibile utilizzare le API standard di WASI per creare il demone, non lo si potrebbe creare multi-thread, in quanto anche tale API è ancora in fase di discussione⁴.

²<https://github.com/WebAssembly/wasi-sockets>

³<https://github.com/deislabs/wagi>

⁴<https://github.com/WebAssembly/wasi-threads>

WAGI

Prima di introdurre WAGI, è utile capire cosa sono le interfacce CGI. Le CGI sono una specifica che permette ai web server di eseguire programmi a riga di comando ovvero programmi non inizialmente pensati per essere eseguiti in un ambiente web, e di restituire il risultato al client.

WAGI implementa questa specifica per WebAssembly, esponendo un web server⁵ HTTP in grado di caricare dinamicamente i moduli Wasm attraverso Wasmtime⁶. Esegue i moduli mappando le richieste HTTP nel seguente modo:

- Header = variabili d'ambiente
- I parametri della query = argomenti del programma
- Il payload = standard input del programma

e la conseguente risposta dei moduli Wasm viene, come per le CGI, scritta sullo standard output e inviata ai client.

Il valore aggiunto di WAGI è che risulta a sua volta un modulo Wasm eseguito da Wasmtime e che seppur utilizzi un'implementazione custom per la gestione delle richieste HTTP e dei thread, segue comunque lo standard WASI in quanto:

- Non può accedere ai file del sistema host senza la specifica autorizzazione (capability)
- Non può accedere alle variabili del sistema host se non quelle esplicitate da esso

ed inoltre

- Non può effettuare connessioni di rete esterne
- Non può eseguire altre applicazioni che non siano moduli Wasm

I moduli Wasm caricati da WAGI vengono chiamati WAGIs e sono parte della stessa sandbox di Wasmtime. I WAGIs vengono pre-caricati e mantenuti in memoria per tutta la durata dell'esecuzione dell'applicazione.

Mysql e Redis

Mysql è un database relazionale che si occupa di salvare in modo persistente le chat, i messaggi e le sessioni utente. Redis è un datastore che mantiene in memoria dati utili all'applicazione: nel nostro caso il numero di utenti online.

⁵https://github.com/deislabs/wagi/blob/main/src/wagi_server.rs

⁶https://github.com/deislabs/wagi/blob/main/src/wasm_runner.rs

Docker

Docker viene utilizzato per la creazione e gestione dei container che ospitano il livello della persistenza dei dati.

3.1.3 Le tecnologie frontend

Non entreremo nel dettaglio delle tecnologie frontend in quanto al di fuori dello scopo della tesi, ma possiamo fornire una breve descrizione per comprenderle meglio.

- ReactJS⁷: un framework Javascript ampiamente utilizzato che consente di creare applicazioni web interattive e dinamiche tramite l'uso di componenti - unità di codice riutilizzabili che possono essere combinati per creare interfacce grafiche complesse.
- TanStackQuery⁸: una libreria Javascript che facilita la gestione dello stato dell'applicazione e delle chiamate asincrone ai microservizi.
- Typescript⁹: un linguaggio di programmazione che estende Javascript aggiungendo la tipizzazione statica. Ciò permette di scrivere codice più pulito e di prevenire errori a runtime.
- Vite¹⁰: uno strumento per lo sviluppo locale, la traduzione del codice Typescript in Javascript e il bundling delle risorse statiche di un'applicazione web.

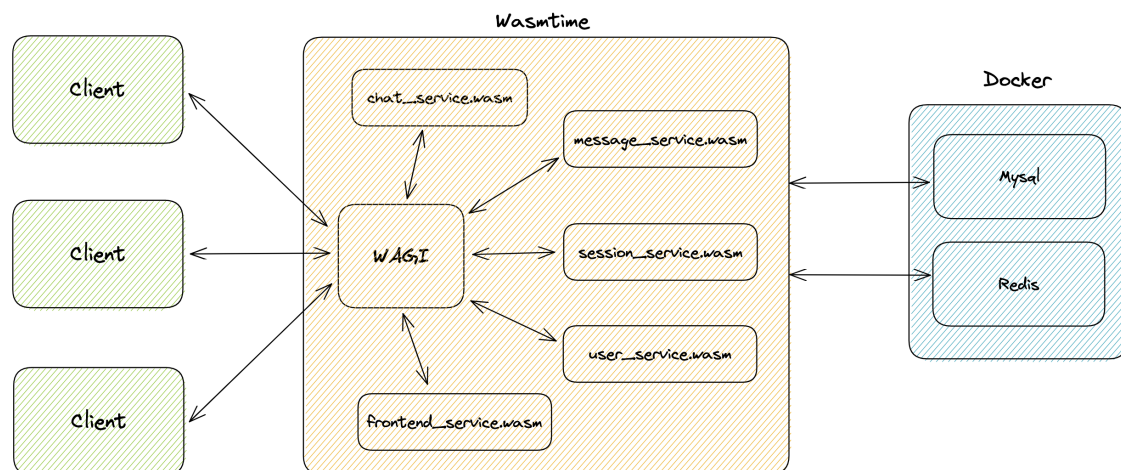


Figura 3.1: Architettura dell'applicazione

⁷<https://github.com/facebook/react>

⁸<https://github.com/TanStack/query>

⁹<https://github.com/microsoft/TypeScript>

¹⁰<https://github.com/vitejs/vite>

3.1.4 La configurazione

La configurazione dell'applicazione avviene modificando il file `spin.toml`, qui sono presenti varie direttive come la configurazione delle variabili d'ambiente, del database, di redis e del webserver sottostante. Inoltre qui si configurano i microservizi presenti nell'applicazione impostando il loro path nella gerarchia delle cartelle, il comando per eseguire il build e l'url corrispondente ad ognuno di essi. È interessante notare che questo step di mapping tra url-microservizio-path è necessario poiché il runtime non potrebbe accedere ai vari moduli senza avere le capabilities necessarie.



Figura 3.2: Configurazione di un microservizio

Si noti la flag `--release` nel comando `'cargo build'`, questa è una direttiva per il compilatore Rust che gli indica di ottimizzare il codice durante la compilazione.

3.1.5 Interazione

Parliamo ora di come avviene l'interazione tra le componenti dell'applicazione.

L'applicazione viene avviata tramite il comando `spin build -up` che esegue il build delle componenti definite nel file di configurazione `spin.toml` e avvia il web server WAGI in ascolto sulla porta 3000. All'arrivo di una richiesta WAGI, carica il modulo corrispondente nella sandbox all'url richiesto, se lo trova, e lo avvia con i parametri necessari. Questo step è trasparente allo sviluppatore in quanto la gestione del web server sottostante è gestita automaticamente.

Il modulo specifica l'entry point attraverso la direttiva `#[http_component]`, esegue il parsing della richiesta e ne elabora la risposta.

La prima interazione tra client e server è necessaria per generare il token di sessione che identifica il cliente che verrà usato per tutte le richieste successive. Il token viene generato dal `session_service` e viene salvato sul database. Una volta ottenuto, il client può effettuare le richieste all'applicazione.

A screenshot of a code editor window with a light gray background and a dark blue border. The editor has three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Rust and is numbered from 1 to 15 on the left side. The code defines an HTTP component with a function `message_service` that takes a `Request` and returns a `Result<Response>`. It uses `Config::get()` to retrieve configuration and a `match` statement to handle different API requests like `BadRequest`, `MethodNotAllowed`, `Create`, `ReadById`, `GetLatest`, `Update`, `Delete`, and a default `not_found` case.

```
1 #[http_component]
2 fn message_service(req: Request) → Result<Response> {
3     let cfg = Config::get();
4
5     match api_from_request(req) {
6         Api::BadRequest ⇒ bad_request(),
7         Api::MethodNotAllowed ⇒ method_not_allowed(),
8         Api::Create(model) ⇒ handle_create(&cfg.db_url, model),
9         Api::ReadById(id) ⇒ handle_read_by_id(&cfg.db_url, id),
10        Api::GetLatest(chat_id) ⇒ handle_get_latest(&cfg.db_url, chat_id),
11        Api::Update(model) ⇒ handle_update(&cfg.db_url, model),
12        Api::Delete(id) ⇒ handle_delete_by_id(&cfg.db_url, id),
13        _ ⇒ not_found(),
14    }
15 }
```

Figura 3.3: Entry point delle operazioni CRUD al microservizio dei messaggi

Una nota sulla sincronizzazione realtime

La sincronizzazione dei dati in tempo reale tra client e server è effettuata tramite un'interazione a polling. Il motivo di questa scelta è dato dal fatto che al momento della scrittura del documento WASI non supporta la gestione delle WebSocket.

3.2 Un confronto

Confrontiamo il microservizio che si occupa della gestione dei messaggi con l'analogo scritto con NodeJS e Express per vedere le principali differenze. Il codice sorgente è disponibile su GitHub¹¹.

Intanto si deve notare come il microservizio scritto in NodeJS debba esporre un web server per poter essere raggiunto dall'esterno mentre il microservizio in Wasm è esposto direttamente da WAGI.

In secondo luogo c'è da notare la gestione di Rust, che risulta sicuramente più complessa, ed è dovuta dal fatto che il linguaggio è più esigente e richiede un approccio più metodico, mentre l'implementazione in NodeJS è più semplice e veloce da scrivere ma più prona agli errori runtime.

¹¹<https://github.com/ilcors-dev/bachelor-thesis/tree/main/poc/nodejs-alternative>

3.2.1 Benchmark

Andiamo ad analizzare le prestazioni dei due microservizi usando uno strumento chiamato JMeter¹². Il benchmark è stato effettuato su un computer con le seguenti caratteristiche: CPU Apple M1 8 core e 16 GB di RAM. I test mirano ad analizzare le prestazioni del microservizio in Wasm e del microservizio in NodeJS in termini di tempo di risposta e di numero di richieste al secondo. Sono stati effettuati vari benchmark secondo carichi diversi, per i servizi di inserimento dei messaggi. Al ripetersi di ogni test il database è stato svuotato.



Figura 3.4: 10000 richieste in sequenza verso Wasm

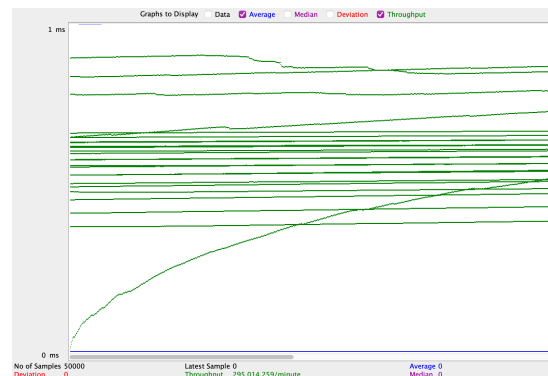


Figura 3.5: 10000 richieste in sequenza verso NodeJS



Figura 3.6: Utilizzo CPU per Wasm

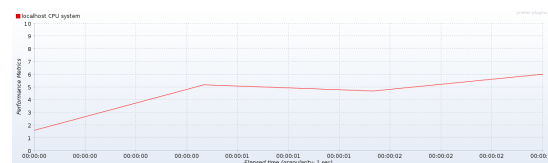


Figura 3.7: Utilizzo CPU per NodeJS

Il primo test effettuato mira a valutare le prestazioni dei due servizi sotto un carico di 10000 richieste effettuate in modo sequenziale. Il test è stato ripetuto 5 volte, per un totale di 50000 richieste. Si può notare come Wasm non riesca a soddisfare al carico di richieste come NodeJS. Nel primo caso i tempi di risposta risultano essere tra i 4ms e i 6ms mentre nel secondo sono tutti sotto al millisecondo. Vediamo ora come si comportano i due nel gestire richieste in parallelo ovvero simulando l'utilizzo del servizio da parte di più utenti nello stesso momento.

¹²<https://jmeter.apache.org/>

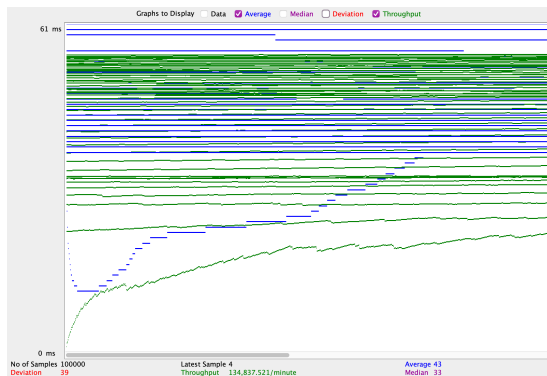


Figura 3.8: 100 utenti e 1000 richieste in simultanea verso Wasm

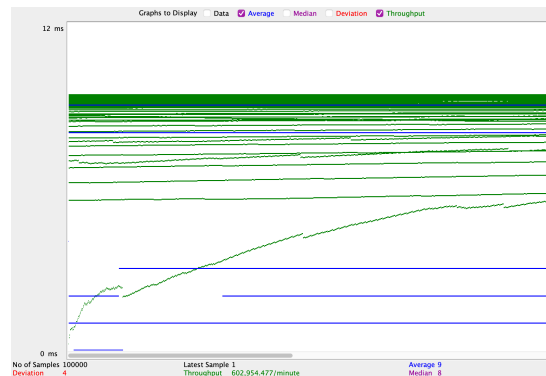


Figura 3.9: 100 utenti e 1000 richieste in simultanea verso NodeJS



Figura 3.10: Utilizzo CPU per Wasm

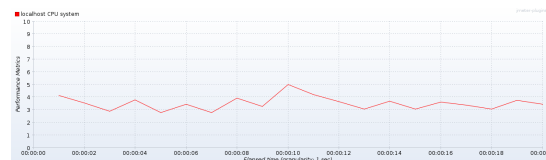


Figura 3.11: Utilizzo CPU per NodeJS

Anche in questo caso Wasm non regge il confronto con NodeJS. Il primo ha tempi di risposta che arrivano fino ai 60ms mentre il secondo non supera i 12ms.

3.2.2 Il deployment

Valutiamo ora invece la gestione del deployment delle due soluzioni. In entrambi i casi si è scelto di creare due immagini dell'applicazione che seguano lo standard OCI. OCI è un formato comune per le immagini container che definisce uno standard per la creazione, la distribuzione e l'esecuzione di essi. Per creare l'immagine dell'applicazione NodeJS è stato utilizzato il seguente Dockerfile:

Il Dockerfile ci permette di creare l'immagine dell'applicazione tramite il comando.

```
docker buildx build . -t ghcr.io/ilcors-dev/chat-nodejs:latest
```

e di pubblicarla su GitHub Container Registry (GHCR), un gestore di immagini container, tramite

```
docker push ghcr.io/ilcors-dev/chat-nodejs:latest
```

Per l'immagine dell'applicazione WASI invece è stato utilizzato il tool integrato nel Framework Spin che ci permette, tramite

```
spin registry push ghcr.io/ilcors-dev/wasi-chat-poc:latest
```



Figura 3.12: Creazione immagine Docker per l'applicazione NodeJS

di creare l'immagine e di pubblicarla su GHCR. Entrambe le immagini sono reperibili al seguente link: <https://github.com/ilcors-dev?tab=packages>

Possiamo analizzare le differenze tra le due soluzioni in termini di dimensioni su disco. Si può notare, attraverso l'immagine manifest reperibile direttamente dall'interfaccia utente del sito, come quella di NodeJS sia estremamente più pesante di quella di Wasm nonostante includa le stesse funzionalità. La prima occupa uno spazio di 65MB¹³ circa, mentre la seconda appena 2,4MB¹⁴

Questo è dovuto al fatto che l'applicazione Wasm è già pronta per essere eseguita in un runtime Wasm e non necessita di altre dipendenze, mentre l'applicazione NodeJS deve, oltre ad includere tutte le dipendenze necessarie, essere eseguita su NodeJS all'interno di una versione leggera di Linux, chiamata Alpine Linux.

Avendo entrambe le immagini disponibili su GHCR è possibile effettuare il deploy-

¹³<https://github.com/users/ilcors-dev/packages/container/chat-nodejs/76069897?tag=latest>

¹⁴<https://github.com/users/ilcors-dev/packages/container/wasi-chat-poc/76056566?tag=latest>

ment delle due applicazioni in locale.

Per l'applicazione NodeJS è necessario configurare sia il container per l'applicazione che il container per la persistenza dei dati tramite il file docker-compose.yml:



```
1 services:
2   db:
3     image: mysql:8.0
4     ports:
5       - '3306:3306'
6     environment:
7       MYSQL_USER: bachelor
8       MYSQL_PASSWORD: bachelor
9       MYSQL_ROOT_PASSWORD: bachelor
10      MYSQL_DATABASE: wasi-chat
11      # test if mysql is ready every 30 seconds
12    healthcheck:
13      test: [ "CMD", "mysqladmin", "ping", "-h", "localhost" ]
14      interval: 30s
15      timeout: 10s
16      retries: 3
17   chat:
18     image: ghcr.io/ilcors-dev/chat-nodejs:latest
19     ports:
20       - '3000:3000'
21     restart: always
22     # since the chat service depends on the db service, we need to wait for the db service to be ready
23     depends_on:
24       - db
25
```

Figura 3.13: Docker compose per l'applicazione NodeJS

e per eseguirla è sufficiente il comando

`docker-compose up`

Anche per l'applicazione Wasm è necessario usare Docker Compose, ma solo per avviare il container per la persistenza. Non se ne riporta l'esempio in quanto è identico a quello precedentemente a meno della configurazione dell'applicazione NodeJS. Per eseguire l'applicazione Wasm è sufficiente il comando

```
spin up -f ghcr.io/ilcors-dev/wasi-chat-poc:latest
```

che si occupa di eseguire un pull dell'immagine e di eseguirla in locale.

3.3 Vantaggi e Svantaggi

Nella soluzione proposta per l'applicazione, l'utilizzo di Wasm ha dimostrato di avere sia vantaggi che svantaggi. Da un lato, ha permesso una maggiore facilità di deployment dell'applicazione grazie alla sua portabilità e alla possibilità di eseguire il codice su diverse piattaforme. D'altra parte, tuttavia, ha avuto un impatto negativo sulle prestazioni dell'applicazione.

Questa nota negativa è dovuta principalmente allo stato ancora acerbo di WASI, il quale non è ancora altrettanto maturo come quello della controparte NodeJS preso in esame. Tuttavia, con il passare del tempo e l'evoluzione della tecnologia, è probabile che WASI diventi sempre più efficiente e performante.

Inoltre, è importante notare che ogni microservizio dell'applicazione NodeJS richiede la sua modalità di deployment, che può essere un'operazione complicata e richiedere molto tempo. Invece, l'applicazione Wasm offre un vantaggio significativo in questo senso, poiché Wasmtime e WAGI si occupano dell'esposizione del server HTTP e includono già tutti i microservizi necessari per l'esecuzione dell'applicazione.

Infine, è importante evidenziare come l'utilizzo combinato di Docker e WASI rappresenti una soluzione ideale per lo sviluppo di applicazioni complesse. In particolare, Docker offre una piattaforma completa per la costruzione dei container e l'esecuzione delle tecnologie complementari necessarie all'applicazione, rappresentando la spina dorsale del sistema. Mentre, WASI offre la portabilità e la sicurezza del codice Wasm, consentendo di eseguire moduli altamente ridotti in una sandbox isolata e garantendo la compatibilità dell'applicazione su diverse piattaforme. Insieme, queste tecnologie consentono di semplificare il processo di deployment dell'applicazione, garantire la scalabilità del sistema e ridurre i tempi di sviluppo.

Conclusione

In un mondo sempre più connesso in cui le applicazioni distribuite sono in costante crescita, la sicurezza, la portabilità e l'efficienza sono fattori cruciali. WASI (WebAssembly System Interface) si presenta come uno standard emergente del W3C per eseguire applicazioni WebAssembly in ambiente distribuito che potrebbe diventare un tassello importante nell'industria informatica, tanto che è definito da alcuni come la terza ondata del cloud computing. Nel corso dell'elaborato, è stata condotta un'analisi delle sue caratteristiche, dei suoi vantaggi e delle problematiche riscontrate.

Nel primo capitolo del presente lavoro si è collocato WASI all'interno dell'ecosistema delle tecnologie web, evidenziando il suo stretto legame con WebAssembly. A tal proposito, si è evidenziata la sua somiglianza al linguaggio Java, poiché entrambi condividono il concetto di WORA (Write Once Run Anywhere), e si sono messi in risalto i tratti distintivi che li differenziano. In seguito, si è approfondita l'analisi tecnica di WASI, al fine di comprendere il suo effettivo funzionamento low-level, se ne è evidenziata la forte predisposizione alla sicurezza basata sul modello capability-based e si è analizzata la possibile collocazione della tecnologia in ambiente distribuito, focalizzandosi in particolare sulla sua natura di container leggero, sulle differenze con i container classici e le Virtual Machine. L'analisi tecnica condotta ha permesso di sviluppare un Proof of Concept che ha dimostrato l'effettiva realizzabilità di applicazioni WebAssembly in ambiente distribuito, nonostante queste siano state originariamente pensate per essere eseguite soltanto all'interno del browser.

Il Proof of Concept è stato sviluppato utilizzando Spin, uno dei primi framework presenti nell'ecosistema, facendo uso di WAGI, un'implementazione delle CGI per WebAssembly e strutturando l'applicazione secondo il modello a microservizi. Se ne è dimostrata la complementarità con Docker e il processo di deployment tramite container OCI. Successivamente, durante la fase di valutazione, si è confrontato il progetto con un'applicazione simile sviluppata in NodeJS. Questo confronto ha evidenziato la natura ancora acerba di WASI rispetto alle soluzioni esistenti in termini di prestazioni ma ne ha evidenziato la grande differenza per quanto riguarda la portabilità e la facilità di deployment. Nonostante i risultati del PoC non siano stati completamente soddisfacenti, la valutazione di WASI rimane positiva in quanto la tecnologia presenta un grande potenziale che potrebbe essere migliorato ulteriormente con lo sviluppo futuro. È importante sottolineare che WASI è ancora in fase di sviluppo, molte funzionalità cruciali come il networking, il supporto ai thread e così via sono ancora in fase di discussione e, in quanto standard, sta impiegando del tempo per essere sviluppato. Il fatto di essere uno standard W3C ne garantisce la

stabilità a lungo termine ma sta obbligando alcune realtà esistenti a implementare soluzioni custom per conseguire i risultati desiderati nel breve termine.

In conclusione, la sicurezza garantita dal modello capability based, la portabilità grazie alla natura basata su WebAssembly e l'efficienza grazie all'uso dei container leggeri sono i principali vantaggi di WASI. Questi fattori, insieme alla possibilità di eseguire codice in modo isolato e sicuro all'interno di una sandbox, possono rendere WASI una valida opportunità futura per lo sviluppo di applicazioni in ambiente distribuito.

Ringraziamenti

Ringrazio il professore relatore del mio lavoro, prof. Paolo Bellavista, per la disponibilità e la professionalità dimostrate.

Ringrazio anche i miei genitori, mio fratello Christian, il mio compagno di studi Leonardo e tutti coloro che hanno creduto in me durante questo percorso. A loro è dedicata questa tesi.

Bibliografia

- [1] Fermyon Technologies Inc. *Hello World*. 8 Feb. 2022. URL: <https://www.fermyon.com/blog/2022-02-08-hello-world>.
- [2] Mozilla. «WebAssembly Concepts». In: (). URL: <https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts>.
- [3] Inc. Stack Exchange. «Stack Overflow Most Popular Technologies Survey 2022». In: (2022). URL: <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages>.
- [4] Michael Irwin. «WebAssembly Concepts». In: (24 ott. 2022). URL: <https://www.docker.com/blog/docker-wasm-technical-preview/>.
- [5] «IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7». In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.
- [6] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.
- [7] Tyler Charboneau. *Why Containers and WebAssembly Work Well Together*. 1 Lug. 2022. URL: <https://www.docker.com/blog/why-containers-and-webassembly-work-well-together/>.
- [8] Fermyon Technologies Inc. *Rethinking Microservices*. 28 Feb. 2023. URL: <https://www.fermyon.com/blog/rethinking-microservices>.
- [9] Snyk Ltd. *State of Open Source Security 2022*. 2022. URL: <https://snyk.io/reports/open-source-security/>.
- [10] Lin Clark. *Building a secure by default, composable future for WebAssembly*. 12 Nov. 2019. URL: <https://bytecodealliance.org/articles/announcing-the-bytecode-alliance>.
- [11] Jack B. Dennis e Earl C. Van Horn. «Early Capability Architectures». In: (1966). URL: <https://homes.cs.washington.edu/~levy/capabook/Chapter3.pdf>.
- [12] Jerome H. Saltzer e Michael D. Schroeder. «The protection of information in computer systems». In: *Proceedings of the IEEE* 63 (1975), pp. 1278–1308.
- [13] Pierangela Samarati e Sabrina Capitani de Vimercati. «Access control: Policies, models, and mechanisms». In: *Foundations of Security Analysis and Design: Tutorial Lectures 1*. Springer. 2001, pp. 137–196.

- [14] Professor Fred B. Schneider. «Capability-based Access Control Mechanisms». In: (). URL: <https://www.cs.cornell.edu/courses/cs513/2000SP/L08.html>.
- [15] D. Robinson e K. Coar. *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875. <http://www.rfc-editor.org/rfc/rfc3875.txt>. RFC Editor, 2004. URL: <http://www.rfc-editor.org/rfc/rfc3875.txt>.