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 WEBASSEMBLY SYSTEM INTERFACE

Relatore:
Prof. Paolo Bellavista

Presentata da:
Luca Corsetti

Anno Accademico 2022/2023

Indice

Elenco delle figure

Capitolo 1

Introduzione

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 in particolare, sebbene sia ancora in fase di sviluppo e di adozione.

[1] `https://wasi.dev/`

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.

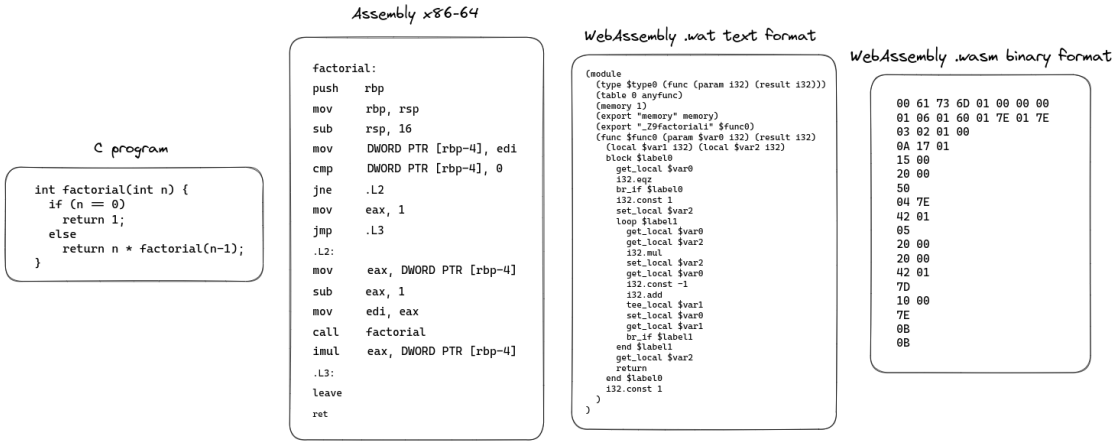


Figura 1.1: Esempio di una funzione compilata nel modo 'tradizionale', in .wasm e la sua rappresentazione in .wat

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 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.

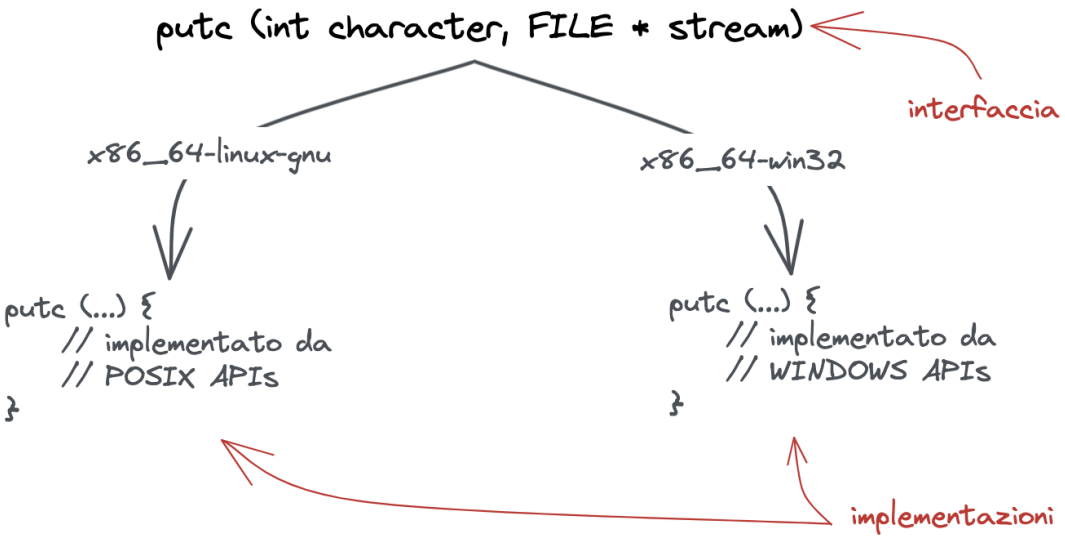


Figura 1.2: System Interface: Dall'API alla sua implementazione

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.

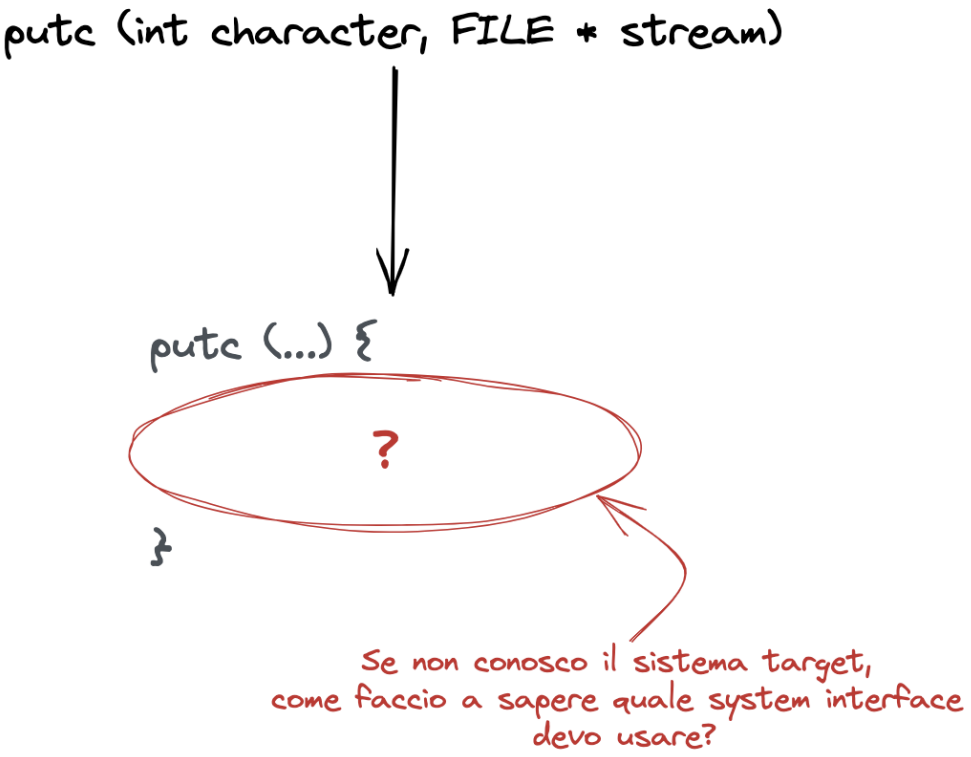


Figura 1.3: Una System Interface per un sistema operativo concettuale

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 secondi[2]), gli utenti la abbandonano.
- **Portabilità**: nell'era degli smartphone, tablet, dispositivi IoT, sensori e altre tecnologie eterogenee, è necessario garantire che la stessa applicazione possa

[2] `https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics/`

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.

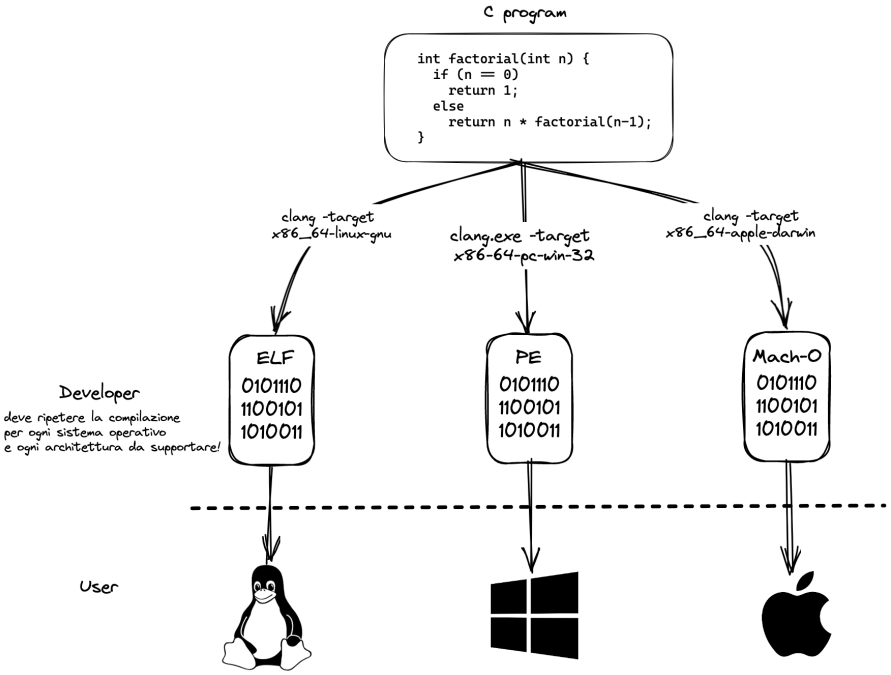


Figura 1.4: Portabilità in senso tradizionale

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".

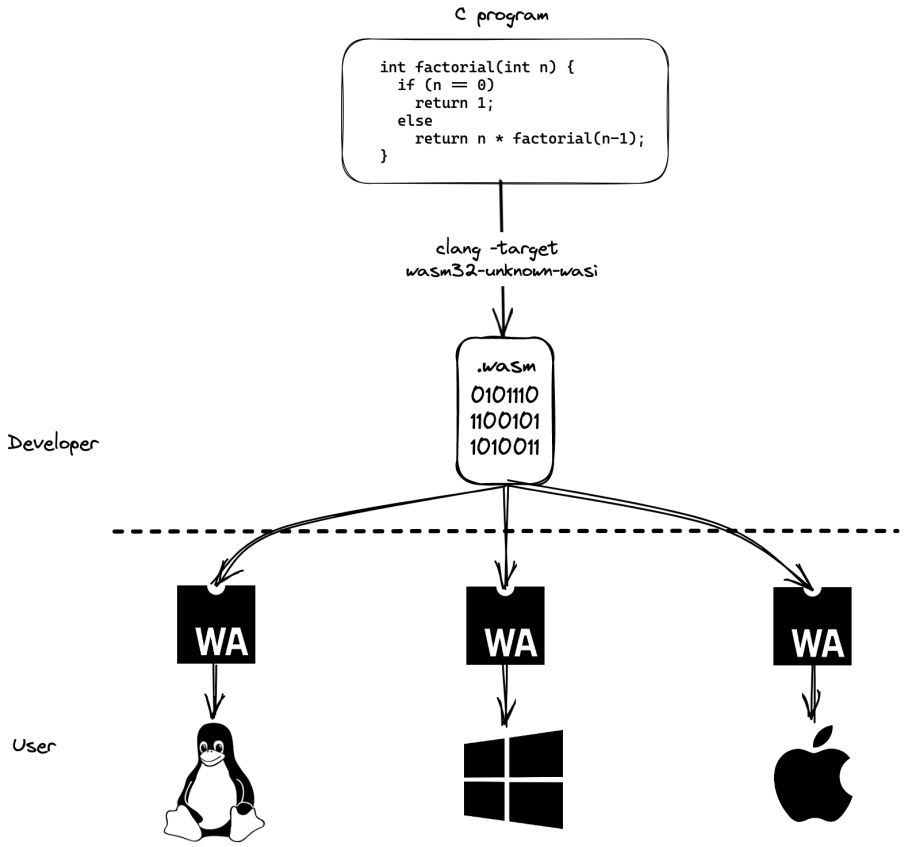


Figura 1.5: Portabilità con Wasm e WASI

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)[3] 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.

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. Tutta-

[3] `https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en`

via, 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	Sì
Java	Sì	No
PHP	Sì	Sì
C# e .NET	Sì	Sì
Ruby	Sì	Sì
Swift	Sì	Sì
Typescript	No	No
C/C++	Sì	Sì
Rust	Sì	Sì
Go	Sì	Sì
Kotlin	Sì	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 fasi[4] 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 2[5] 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 della tecnologia. Il rischio che si corre è che il tempo di attesa per la

[4] `https://github.com/WebAssembly/meetings/blob/main/process/phases.md`
[5] `https://github.com/WebAssembly/WASI/blob/main/Proposals.md`

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 WasmEdge[6], 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

Wasmtime[7] è il runtime ufficiale di WASI ed è un progetto gestito dalla Bytecode Alliance[8], 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.

WasmEdge

WasmEdge[9] 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

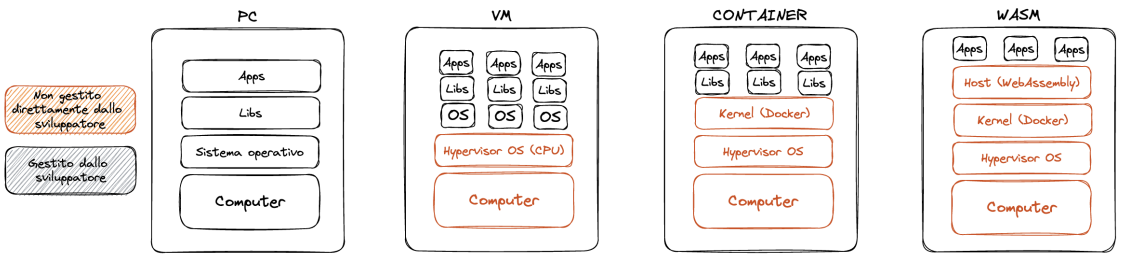
[6] `https://github.com/WasmEdge/WasmEdge`
[7] `https://wasmtime.dev/`
[8] `https://bytecodealliance.org/`
[9] `https://wasmedge.org/`

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.



Gestito da Sviluppatore	Tutto	OS, App, Lib	App, Lib	Wasm
Astrazione	-	CPU	Linux Kernel	Sandbox
Impatto su disco	Alto	Medio	Basso	Molto Basso
Portabilità	-	Bassa	Media (CPU, Linux)	Alta
Tipo di Sicurezza	Intro sistema	OS	A livello di processo Unix	Capability-Based

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*[10]

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*[11]

[10] `https://twitter.com/solomonstre/status/1111004913222324225`
[11] `https://twitter.com/solomonstre/status/1111113329647325185`

Capitolo 2

WASI, nel dettaglio

L'obiettivo principale di WASI è fornire un set di API standard per WebAssembly, indipendenti dal motore di esecuzione 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 ed in particolare su musl-libc[1] ed utilizza le funzioni di libpreopen[2].

2.1 WASI-libc

WASI-libc[3] 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.

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

[1] `http://musl.libc.org/`
[2] `https://github.com/musec/libpreopen`
[3] `https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md`

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 file system. Libpreopen intercetta le chiamate di sistema relative ai file, come open(), stat() e opendir(), e le traduce in operazioni definite dalle applicazioni. Ciò può essere utile in situazioni in cui un'applicazione deve accedere a file che si trovano in una directory o un archivio specifico, o quando un'applicazione deve isolare le sue operazioni sui file. Libpreopen 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 sicurezza con WASI

2.2.1 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 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 2019 State of Open Source Security Report*[4]

Per peggiorare le cose, non tutte queste vulnerabilità vengono risolte, si stima che solo il 59% dei pacchetti software risolva le vulnerabilità trovate.

[4] `https://snyk.io/reports/open-source-security/`

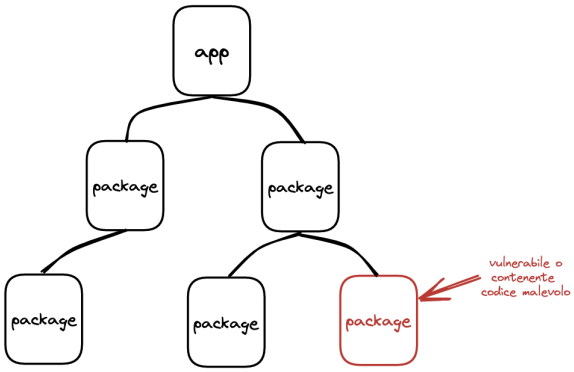


Figura 2.1: Come sono composte le applicazioni moderne

2.2.2 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 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. Le soluzioni esistenti cercano di individuare le vulnerabilità del software, ma non offrono una vera e propria prevenzione. La soluzione ideale sarebbe quella di prevenire le vulnerabilità alla radice. Per arrivare alla soluzione, dobbiamo tornare un po' indietro nel tempo, in quanto è un problema già affrontato in passato, solo con un altro tipo di granularità.

Il problema di far coesistere due applicazioni in esecuzione senza interferenze reciproche 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 processi. Se il processo ha bisogno di comunicare con altri, deve prima richiedere il permesso e farlo tramite le "pipe". Questo sistema risolve il problema della condivisione della memoria a runtime, ma non garantisce che l'applicazione, ad esempio, non acceda al filesystem per effettuare qualche operazione non prevista.
Le VM e i container sono stati sviluppati in origine proprio per questo motivo, garantiscono che qualcosa in esecuzione in una VM o container non possa accedere al filesystem di altri, ma non nel proprio. Con il modello "sandbox", invece, potremmo ulteriormente isolare le applicazioni rimuovendo l'accesso alle API e alle system call del sistema prima che l'applicazione venga eseguita. Guardando alle tre soluzioni potremmo arrivare alla conclusione che per garantire la sicurezza e l'isolamento di ogni package sia necessario isolarlo all'interno della propria sandbox, con i giusti

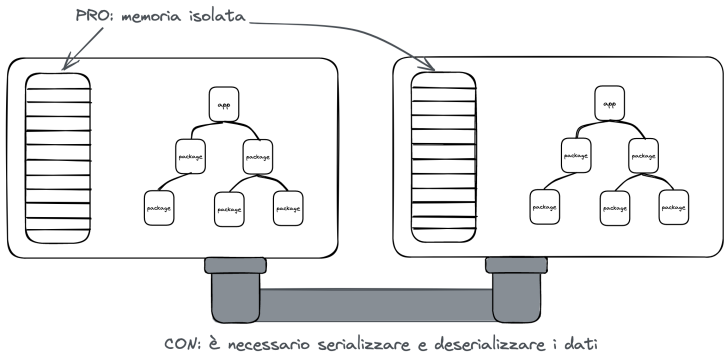


Figura 2.2: Isolamento dei processi nei sistemi operativi

permessi. Ciò però porterebbe presto ad un esaurimento delle risorse del sistema ed introdurrebbe un overhead nella comunicazione tra i package dell'applicazione.

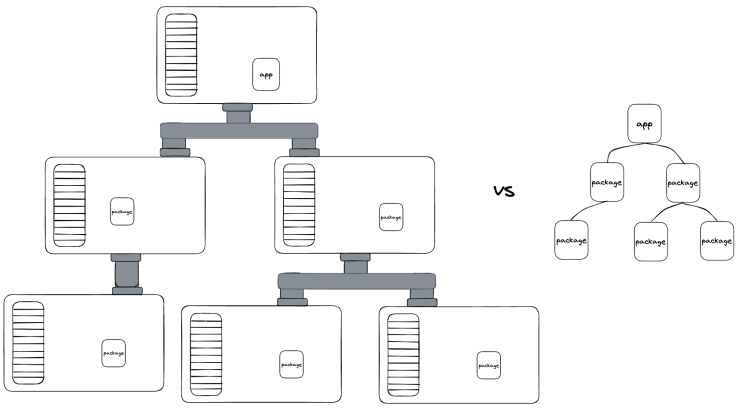


Figura 2.3: Isolamento dei package in sandbox isolate, evidente inefficienza e overhead

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

2.2.3 La soluzione: nanoprocessi

WebAssembly offre un'efficace forma di isolamento che previene al codice non sicuro di fare ciò che vuole, i nanoprocessi. 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 padre isolato dall'esterno. La memoria a cui ogni nanoprocesso ha accesso è uno slice della memoria del processo padre, ovvero della sandbox. Per accedere ai dati di un altro modulo, un modulo deve avere l'autorizzazione esplicita per farlo e questa autorizzazione viene passata in modo gerarchico, dall'alto verso il basso. In questo modo, la sandbox coordina l'accesso ai dati e garantisce che ogni nanoprocesso operi in modo indipendente e sicuro, senza che vadino ad interferire con gli altri. 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. Si noti che il concetto di nanoprocesso in WebAssembly è basato sui

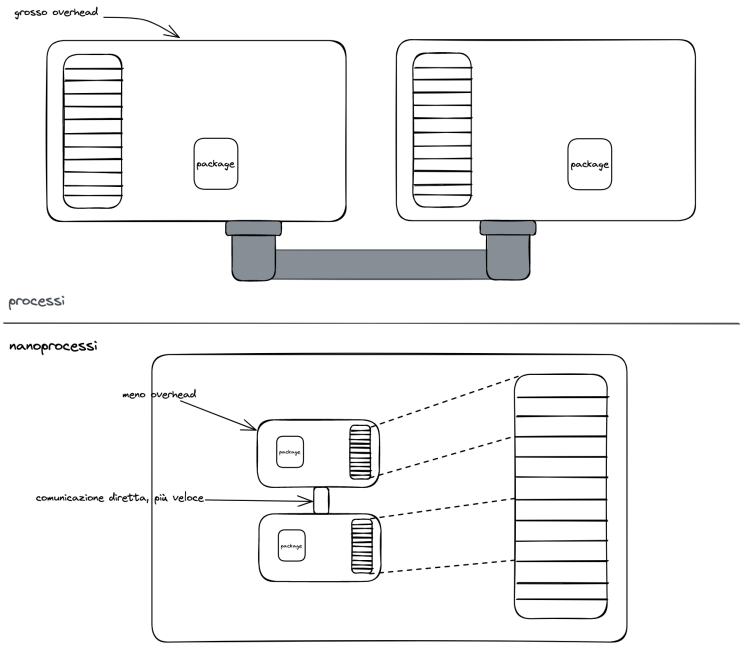


Figura 2.4: Nanoprocessi di Wasm

processi Unix, ma implementato attraverso un pattern specifico di WebAssembly.
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 finirebbe 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. In relazione al codice vulnerabile, sarebbe estremamente difficile 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.
Si può notare come questa filosofia di WebAssembly aderisca al principio di least authority (POLA) e al capability based model.

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/.