

Progetto RTIS: IVSHMEM

Realizzato da:

Stefano Marano M63001428

Vito Romano M63001504

Paolo Russo M63001426

Obiettivi del progetto

L'obiettivo di questo progetto è la realizzazione di una rete di macchine virtuali (VM) utilizzando un hypervisor di partizionamento denominato **Jailhouse**.

Questo sistema è concepito per essere altamente scalabile, permettendo l'aggiunta e la gestione di un numero variabile di VM in base alle necessità dell'utente. Ogni VM nella rete sarà completamente visibile e capace di comunicare in modo bidirezionale sia con l'esterno che tra di esse.

La comunicazione tra le VM avverrà utilizzando il protocollo **IVSHMEM** (Inter-VM Shared Memory), che consente uno scambio di dati efficiente e rapido. Questa tecnologia garantisce che le VM possano condividere informazioni in tempo reale, migliorando le performance complessive della rete e riducendo la latenza.

Un altro aspetto fondamentale del progetto è la compatibilità con orchestratori, strumenti essenziali per il controllo e la gestione centralizzata delle VM, in questo modo il sistema non solo è facile da monitorare e controllare, ma anche che possa integrarsi agevolmente con infrastrutture esistenti o future.

Una volta realizzato il sistema, sono stati effettuati test per valutarne le performance con protocolli di comunicazione TCP e UDP. Questi test hanno permesso di analizzare la larghezza di banda e la stabilità delle connessioni di rete tra le VM e l'esterno. Inoltre, sono stati valutati gli effetti di vari tipi di stress sul sistema.

Jailhouse: Un Hypervisor Flessibile

Cos'è Jailhouse?

Jailhouse è stato sviluppato da Siemens come un partitioning hypervisor Linux-based nel 2013.

È semplice e non offre tante funzioni, ciò lo rende leggero e altamente performante.

Non schedula le VM sui core, poiché assegna 1:1 l'hardware alle VM.

Isolamento

Permette di creare partizioni statiche di sistema, chiamate "celle", completamente isolate, garantendo la sicurezza e l'indipendenza delle diverse applicazioni o sistemi operativi in esecuzione sulla stessa piattaforma hardware.

Configurabilità Avanzata

Attraverso file di configurazione, consente di personalizzare in modo granulare le risorse hardware assegnate a ciascuna cella, come CPU, memoria e periferiche. Questa flessibilità lo rende adatto a una vasta gamma di scenari applicativi.

Supporto Real-Time

L'intervento è minimo, minimizza le VM-exit, in questo modo non solo ha buone performance ma anche un'architettura a bassa latenza.



IVSHMEM: Comunicazione Condivisa tra VM

Interconnessione Flessibile

IVSHMEM consente la comunicazione tra un massimo di 65.536 peer, offrendo una regione di memoria condivisa multiuso per lo scambio di dati e la segnalazione di eventi tramite interrupt.

Configurazione Semplice

L'integrazione di IVSHMEM avviene in modo standard attraverso il protocollo PCI, senza richiedere complessità aggiuntiva.



Protocolli Personalizzati

IVSHMEM lascia libertà agli utenti di definire i protocolli di comunicazione, mantenendo l'hypervisor agnostico rispetto ai dettagli di implementazione.

Registri Personalizzati

I registri possono essere implementati sia memory mapped che tramite I/O, secondo il supporto della piattaforma ed il costo di VM-exit.

IVSHMEM: Struttura della memoria condivisa

La prima sezione è obbligatoria

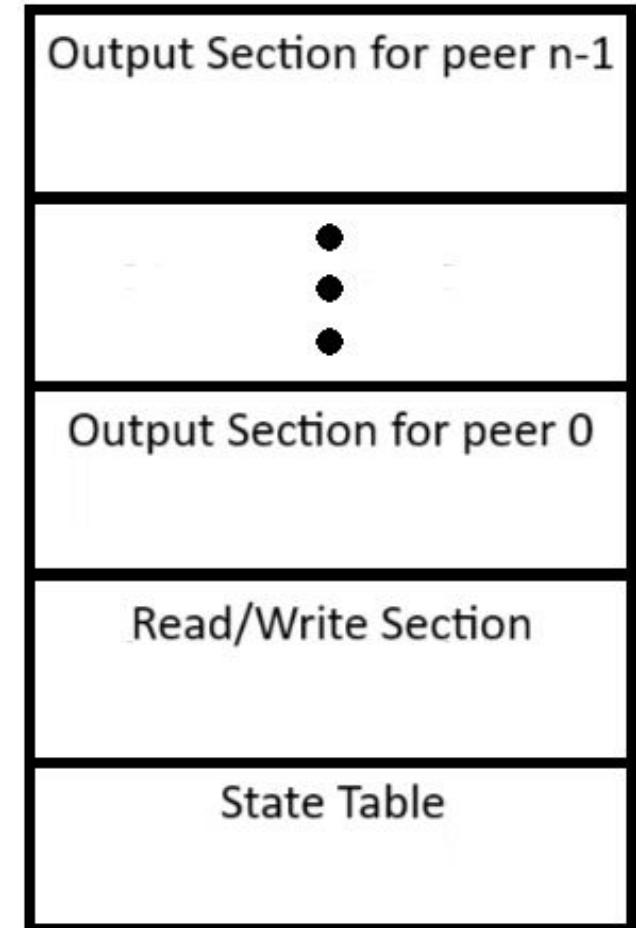
Costituisce la tabella di stato, la sua dimensione è definita dal registro State Table Size, ed è di sola lettura per tutti i peer, permette la gestione del ciclo di vita attraverso lo scambio di valori di stato e la notifica di interrupt sui cambiamenti.

La seconda sezione può essere letta/scritta da tutti i peer.

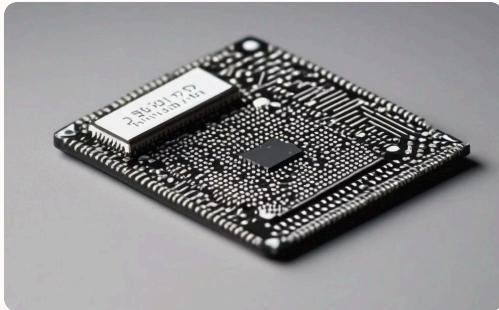
La sua dimensione è definita dal registro R/W Section Size ed è ammessa una dimensione pari a zero.

La terza e le successive sezioni sono sezioni di uscita

Una sezione di uscita è leggibile/scrivibile per il peer corrispondente e di sola lettura per tutti gli altri peer. Ad esempio, solo il peer con ID 3 può scrivere nella quarta sezione di uscita, ma tutti i peer possono leggere da questa sezione.

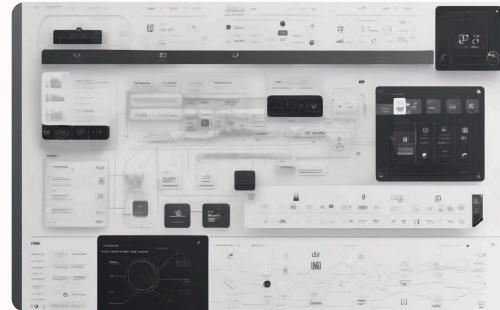


Zephyr: Un RTOS Versatile



Kernel Leggero

Zephyr è un sistema operativo in tempo reale (RTOS) progettato per sistemi embedded con risorse limitate.



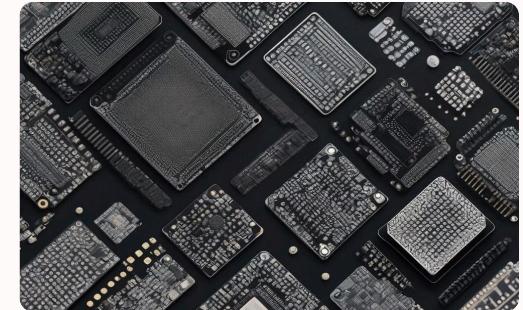
Funzionalità Avanzate

Zephyr offre un ampio set di servizi del kernel, come il multi-threading, la protezione della memoria e un modello ottimizzato per i driver di periferica. Include il supporto dell'API POSIX.



Stack di Rete Nativo

Lo stack di rete di Zephyr è completo e ottimizzato, supportando diversi protocolli come TCP, UDP, IPv4 e IPv6.



Vasto supporto alle board

Il kernel Zephyr supporta diverse architetture, tra cui: ARCv2 (EM e HS) e ARCv3 (HS6X), ARM v6/7/8-M e ARMv7/8-A, Intel x86 (32 e 64 bit), in totale sono supportate più di 600 schede.



Made with Gamma

Zephyr: Stack di rete



Zephyr: IVSHMEM DoorBell

Zephyr integra un meccanismo di notifica dei dispositivi ivshmem come parte del suo stack di comunicazione inter-VM utilizzando vari componenti del sistema operativo.

La notifica consente alle macchine virtuali con dispositivi ivshmem abilitati di notificarsi (interrompersi) a vicenda.

Zephyr include meccanismi per inoltrare l'interrupt all'applicazione corretta, questo può includere la registrazione di un gestore di interrupt specifico per ivshmem che processa l'interrupt e notifica l'applicazione interessata.

Le fasi della comunicazione:

1. **Notification Sender** (VM): La VM invia la notifica alla VM di destinazione scrivendo l'ID del Peer (uguale all'ID della VM di destinazione) e l'indice del vettore nel registro doorbell del dispositivo ivshmem;
2. **Hypervisor**: Quando il registro doorbell è programmato, l'hypervisor cerca la VM di destinazione in base all'ID Peer di destinazione e inietta l'interrupt alla VM di destinazione.
3. **Notification Receiver** (VM): La VM riceve l'interrupt MSI e lo inoltra all'applicazione correlata.

Ivshmem Ethernet su Zephyr

Per utilizzare ivshmem Ethernet in Zephyr è necessario abilitare la configurazione **CONFIG_IVSHMEM** e configurare il driver Ethernet ivshmem. Può essere utilizzato con qualsiasi dispositivo Ethernet supportato da Zephyr.

Zephyr configura il dispositivo ivshmem tramite una serie di passi iniziali che includono: la configurazione dell'hardware in base al sistema di descrizione hardware (**devicetree**), questo include la definizione degli indirizzi di memoria, delle interruzioni e degli altri parametri necessari, successivamente avviene l'inizializzazione del driver, infatti Zephyr include un driver ivshmem che inizializza il dispositivo durante il boot del sistema. Questo driver gestisce le operazioni di lettura e scrittura dal registro doorbell.

Una volta configurato ivshmem Ethernet è possibile utilizzare le API Zephyr per scambiare dati tra le VM. Le API forniscono sia funzioni per allocare e deallocare, sia per scrivere e leggere i dati dalla memoria condivisa e notificare alle altre VM che i dati sono disponibili.



Perchè si usano i meccanismi di rete?



Isolamento e sicurezza

Le VM sono progettate per essere isolate l'una dall'altra, il che significa che non dovrebbero essere in grado di accedere direttamente alla memoria di altre VM, quindi l'utilizzo di meccanismi di rete consente di implementare meccanismi di sicurezza come l'autenticazione, l'autorizzazione e la crittografia per proteggere i dati scambiati tra le VM.



Prestazioni e scalabilità

I meccanismi di rete sono ottimizzati per il trasferimento efficiente dei dati e possono supportare reti con un gran numero di peer (importante per il cloud computing) che devono gestire un gran numero di VM contemporaneamente. Inoltre consentono di distribuire il carico di lavoro su più reti.



Flessibilità

I meccanismi di rete forniscono un'interfaccia astratta e standardizzata per la comunicazione, in quanto è possibile utilizzare diversi tipi di reti (ad esempio, Ethernet, Wi-Fi) e protocolli di rete (ad esempio, TCP/IP) per la comunicazione.



Interoperabilità

Per VM che eseguono sistemi operativi e applicazioni diverse, poiché i meccanismi di rete sono ampiamente utilizzati e supportati da un'ampia gamma di sistemi operativi e applicazioni.

L'utilizzo di meccanismi di rete standard consente di integrare facilmente le VM in reti esistenti e di sfruttare i servizi di rete già disponibili.



Implementazione

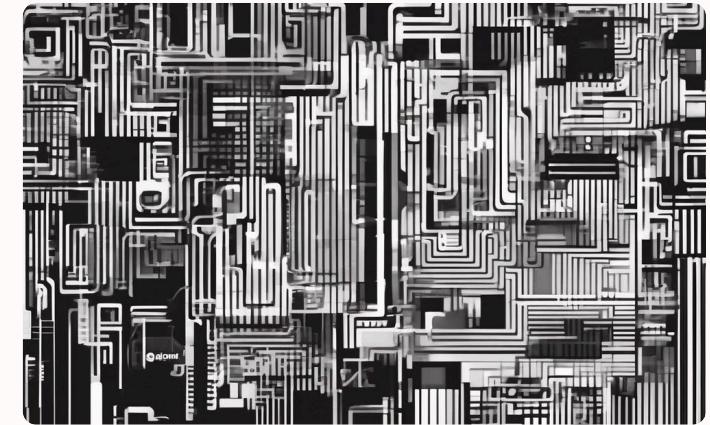
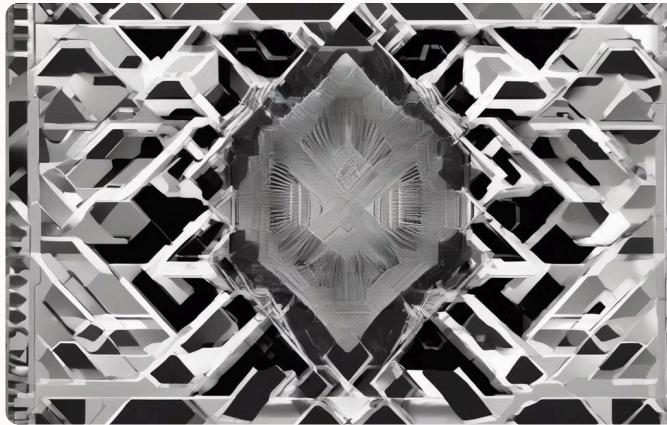
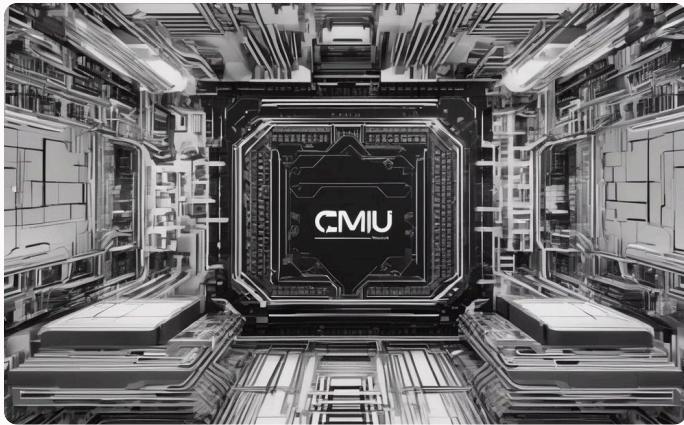


Qemu

QEMU (Quick Emulator) è un software open source che permette l'emulazione e la virtualizzazione di sistemi operativi e processori. Funziona emulando diverse architetture hardware, come x86, ARM, PowerPC e molte altre, consentendo l'esecuzione di sistemi operativi e applicazioni progettati per queste architetture su host che utilizzano un hardware diverso.

Nel nostro caso, per realizzare il nostro progetto abbiamo emulato un dispositivo con processore Cortex-A53.

Qemu: Potente Ambiente di Sviluppo Virtuale



Versatilità

Qemu supporta un'ampia gamma di architetture hardware, consentendo di testare e sviluppare software su diverse piattaforme virtuali.

Accelerazione Hardware

Sfruttando le capacità di virtualizzazione hardware, offre prestazioni ottimali per eseguire macchine virtuali in modo efficiente.

Interfaccia di Rete

Supporta la creazione di reti virtuali, la configurazione di bridge di rete, il NAT e altre configurazioni di rete complesse tra le macchine virtuali e il sistema host.

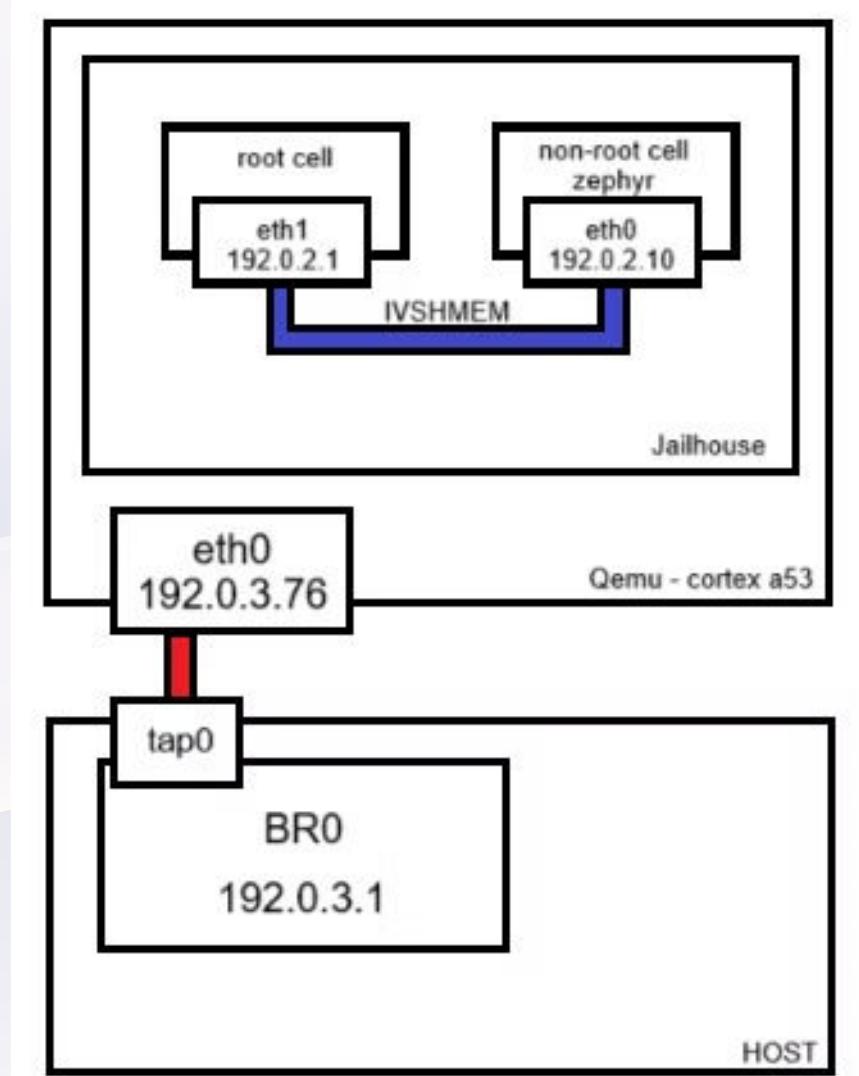
Rete finale

Abbiamo strutturato la nostra piattaforma in due LAN separate:

- **LAN1** - è la rete con l'indirizzo 192.0.3.X, che gestisce le varie istanze Jailhouse, nel nostro caso vi è un'unica istanza ed è emulata usando Qemu
- **LAN2** - è la rete interna a una macchina, riunisce le varie VM ottenute dal partizionamento statico del hardware. Questa rete è identificata dall'indirizzo 192.0.2.X.

È possibile collegare un bridge a un gateway per estendere o segmentare la rete, ma nel nostro caso specifico non era necessario.

Inoltre, poiché utilizziamo una scheda di rete WiFi, non esiste una compatibilità diretta tra il bridge e il protocollo WiFi, poiché i bridge tradizionali lavorano con indirizzi MAC e si aspettano di vedere pacchetti Ethernet standard. I pacchetti Wi-Fi, tuttavia, possono includere informazioni aggiuntive specifiche del protocollo 802.11.



Implementazione del Sistema

Interfaccia TAP

Necessaria per creare un ambiente che simulasse una LAN di varie VM.

Routing Flessibile

Fulcro del progetto andare a realizzare una rete che fosse il più possibile flessibile e scalabile.

Configurazione Zephyr

L'ottimizzazione di parametri può migliorare significativamente le prestazioni di TCP e UDP su Zephyr.

Configurazione Automatica con Script

Tramite degli script bash è facilitata la replicazione del sistema e l'integrazione in altri contesti.



Interfaccia TAP

Un'interfaccia TAP (Terminal Access Point) è un'interfaccia di rete virtuale che opera a livello del collegamento dati, viene utilizzata per simulare una scheda di rete fisica, permettendo il trasferimento di pacchetti Ethernet tra il sistema operativo e le applicazioni. Le interfacce TAP sono spesso utilizzate in contesti di virtualizzazione, VPN e reti overlay.

Il tunneling è una tecnica di rete che permette di incapsulare un protocollo di rete all'interno di un altro protocollo, quindi può trasportare pacchetti di rete attraverso una rete incompatibile o sicura. Ad esempio, trasportare pacchetti IPv6 attraverso una rete IPv4.

In questo modo i dispositivi TAP sono supportati dai driver bridge di Linux, il che significa che possono essere collegati tra loro o con altre interfacce host. Questo setup è utile per creare reti virtuali in cui le macchine virtuali possono comunicare tra loro o con altre macchine fisiche sulla LAN, questo c'ha permesso di creare una rete di emulatori flessibile e facilmente testabile, inoltre è possibile collegare il gateway al bridge per estendere facilmente l'architettura.

Routing Flessibile

Scelta della Configurazione di Rete: Bridge Doppi

Bridge Primario (br0)

Questo bridge collega la macchina host Linux con la root cell, attraverso l'utilizzo dell'interfaccia TAP e il tunneling.

Bridge Secondario

Viene costruito sulla root-cell in modo che quest'ultima possa instradare il traffico delle varie cell, in questo modo le macchine virtuali saranno in grado di parlare tra loro e con il sistema host.

Nessuna interfaccia reale è collegata al bridge primario, le macchine virtuali saranno in grado di parlare tra loro e con il sistema host. Tuttavia, non saranno in grado di parlare con nulla sulla rete esterna, a condizione che non sia stato impostato il mascheramento IP sull'host fisico. Questa configurazione è chiamata **host-only network** da altri software di virtualizzazione come VirtualBox.

Routing Flessibile

Durante i test di configurazione, abbiamo scoperto che il secondo bridge presenta un problema significativo: quando proviamo a istanziare un collegamento tra il bridge e una cella Jailhouse, si verifica una perdita completa della connessione. Ciò è probabilmente dovuto all'utilizzo di IVSHMEM, poiché lo stesso bridge riesce a collegarsi attraverso il tunneling al bridge primario.

Scelta della Configurazione di Rete: DNAT

Implementare una rete che utilizza il DNAT (Destination Network Address Translation) e il port forwarding con lo strumento Socat è una soluzione potente per gestire la comunicazione tra varie macchine in una rete.

Il DNAT è una forma di Network Address Translation (NAT) che viene utilizzata per modificare l'indirizzo IP di destinazione di un pacchetto IP in transito. Invece Socat è uno strumento di rete che può creare connessioni bidirezionali tra due flussi di dati, risulta essere molto versatile e utile per vari scopi, inclusi tunneling e port forwarding.

Questa soluzione è utile in scenari in cui si ha la necessità di esporre servizi interni sulla rete pubblica senza compromettere la sicurezza interna, ma per renderla dinamica e scalabile c'è bisogno di maggior lavoro rispetto a una soluzione con bridge.

Configurazione Zephyr

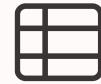


Coda di trasmissione

Nella configurazione predefinita, Zephyr esegue l'elaborazione dei livelli L4 (trasporto), L3 (rete), L2 (data link) e del driver in un unico thread. Ciò può influenzare negativamente il throughput finale.

Una strategia efficace per aumentare il throughput è abilitare la coda di trasmissione. Con questa configurazione, un pacchetto viene messo in coda, consentendo l'elaborazione del livello L2 e del driver in un thread separato.

Questo parallelismo permette al thread responsabile dell'elaborazione dei livelli L4/L3 di continuare a processare i frame successivi anche quando il driver è occupato nella trasmissione, migliorando così il throughput complessivo.



Buffer di rete

Un miglioramento del throughput può essere ottenuto ottimizzando la dimensione del buffer di rete per adattarsi all'MTU (Maximum Transmission Unit) effettivo della rete.

Con un buffer configurato per l'MTU, l'elaborazione ai livelli L3/L4 diventa più efficiente, poiché il pacchetto è contenuto in un singolo buffer anziché in una catena di buffer, riducendo il tempo di elaborazione.

Inoltre, questa configurazione aumenta la dimensione della finestra TCP TX/RX predefinita, migliorando il throughput TCP sia in trasmissione che in ricezione.

Configurazione Zephyr

Di seguito riportiamo i risultati ottenuti su una STM nucleo h723zg andando opportunamente a fare tuning dei parametri.

Configurazione	TCP RX/TX	UDP	TCP	UDP	TCP
	Window	upload	upload	download	download
Default	1194	51.2 Mbits/sec	670 Kbits/sec	88.12 Mbits/sec	7.71 Mbits/sec
CONFIG NET TC TX COUNT = 1	1194	73.6 Mbits/sec	670 Kbits/sec	88.03 Mbits/sec	7.73 Mbits/sec
CONFIG NET TC TX COUNT = 1 CONFIG NET BUF DATA SIZE = 1500	14000	78.3 Mbits/sec	69.8 Mbits/sec	88.01 Mbits/sec	75.03 Mbits/sec
CONFIG NET TC TX COUNT = 1 CONFIG NET BUF DATA SIZE = 1500 CONFIG NET PKT RX/TX COUNT = 80 CONFIG NET BUF RX/TX COUNT = 80	4000	77.9 Mbits/sec	75.0 Mbits/sec	88.12 Mbits/sec	79.56 Mbits/sec

Configurazione Automatica con Script

Script 1 - setup_bridge

1. Crea il bridge `br0`.
2. Crea la `tap0` e assegna un ID.
3. Effettua il **binding** tra il bridge e la tap.
4. Configura l'indirizzo IP del bridge.
5. Imposta un server DHCP per i peer del bridge.

Da questo momento, è possibile creare una rete di macchine virtuali (VM) basate su QEMU che possono comunicare tra loro tramite l'host.

Configurazione Automatica con Script

Script 2 - Config_net

Da utilizzare su una root cell con Jailhouse in esecuzione, permette di creare un interfaccia di rete per comunicare con una non-root cell e introduce le regole di routing per rendere raggiungibile la cella. Inoltre se necessario introduce il servizio DHCP per la negoziazione dell'indirizzo IP.

Da questo momento, le celle create da Jailhouse sono visibili all'esterno ed è possibile comunicare in maniera bidirezionale con esse e comandarle utilizzando un orchestratore.

Configurazione Automatica con Script

Script 2 - Regole di routing

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

```
iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

```
iptables -A FORWARD -i eth0 -o eth1 -j ACCEPT
```

Test

Capacity test

I test sono stati condotti utilizzando il tool **iperf**, fissando una durata di 200 secondi e valutando diverse dimensioni del pacchetto, la connessione è in **upload** da non-root cell verso un server in ascolto sulla macchina host. Inoltre il kernel zephyr è stato impostato con un tuning minimo e semplice, quindi sono da intendersi come valutazioni preliminari delle possibili performance del sistema.



TCP

I test di capacità TCP hanno evidenziato una maggiore efficienza e stabilità utilizzando pacchetti di dimensioni standard, ma in generale una certa instabilità.



UDP

I test di capacità UDP hanno dimostrato che l'utilizzo di pacchetti di dimensioni maggiori garantisce una larghezza di banda più elevata, in generale è stato possibile spingere di più le performance del sistema.

Capacity Test: TCP a 20b

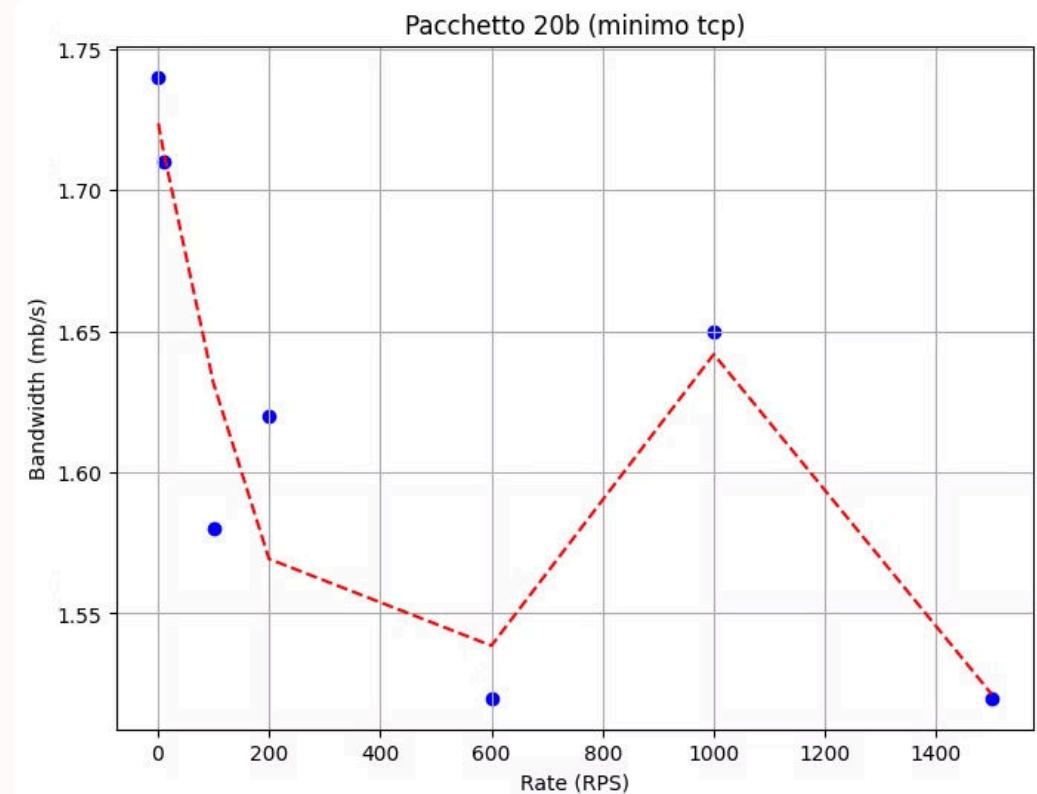
L'obiettivo

Verificare come il sistema gestisce un elevato numero di pacchetti e valutare l'overhead introdotto dalla frequente creazione e gestione.

Ci aspettiamo una minor ritardo di ritrasmissione e minor rischio di frammentazione, migliorando l'efficienza del trasferimento, anche se si trasportano pochi dati utili.

Risultato

Più pacchetti totali significano più lavoro per router, switch e la CPU del dispositivo finale, il che può influenzare le prestazioni. Abbiamo riscontrato dell'overhead significativo con una buona parte della larghezza di banda occupata dal protocollo di gestione, ma anche forti variazioni da un valore di rate a un altro e ciò è imputabile al fatto che ci trovavamo su una VM.



Capacity Test: TCP a 256b

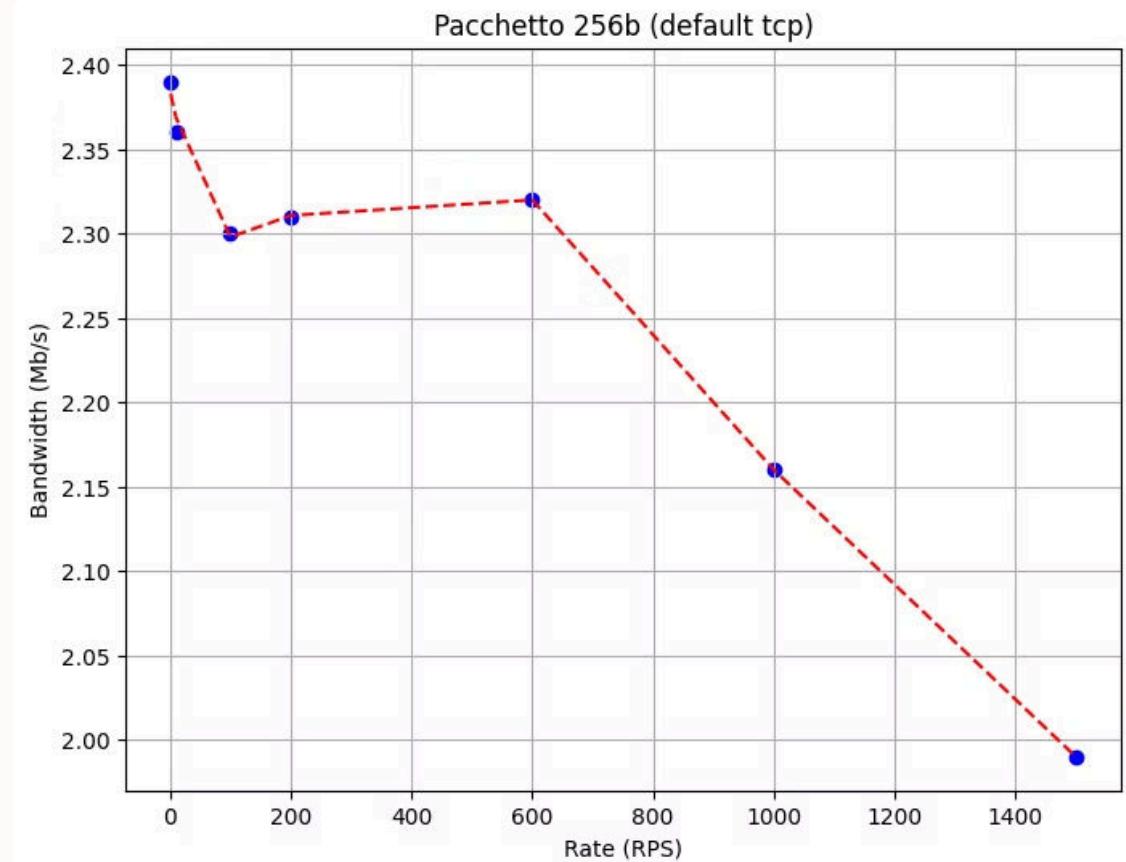
L'obiettivo

In questo caso ci aspettiamo maggiore efficienza e velocità di trasmissione, poiché con pacchetti grandi, la quantità di overhead (intestazione TCP/IP) per byte di dati utile è ridotta.

Inoltre, essendo meno pacchetti totali si riduce il carico di lavoro su router e switch, cosa importante in questo contesto poiché abbiamo che tutto il traffico passa attraverso la root cell.

Risultato

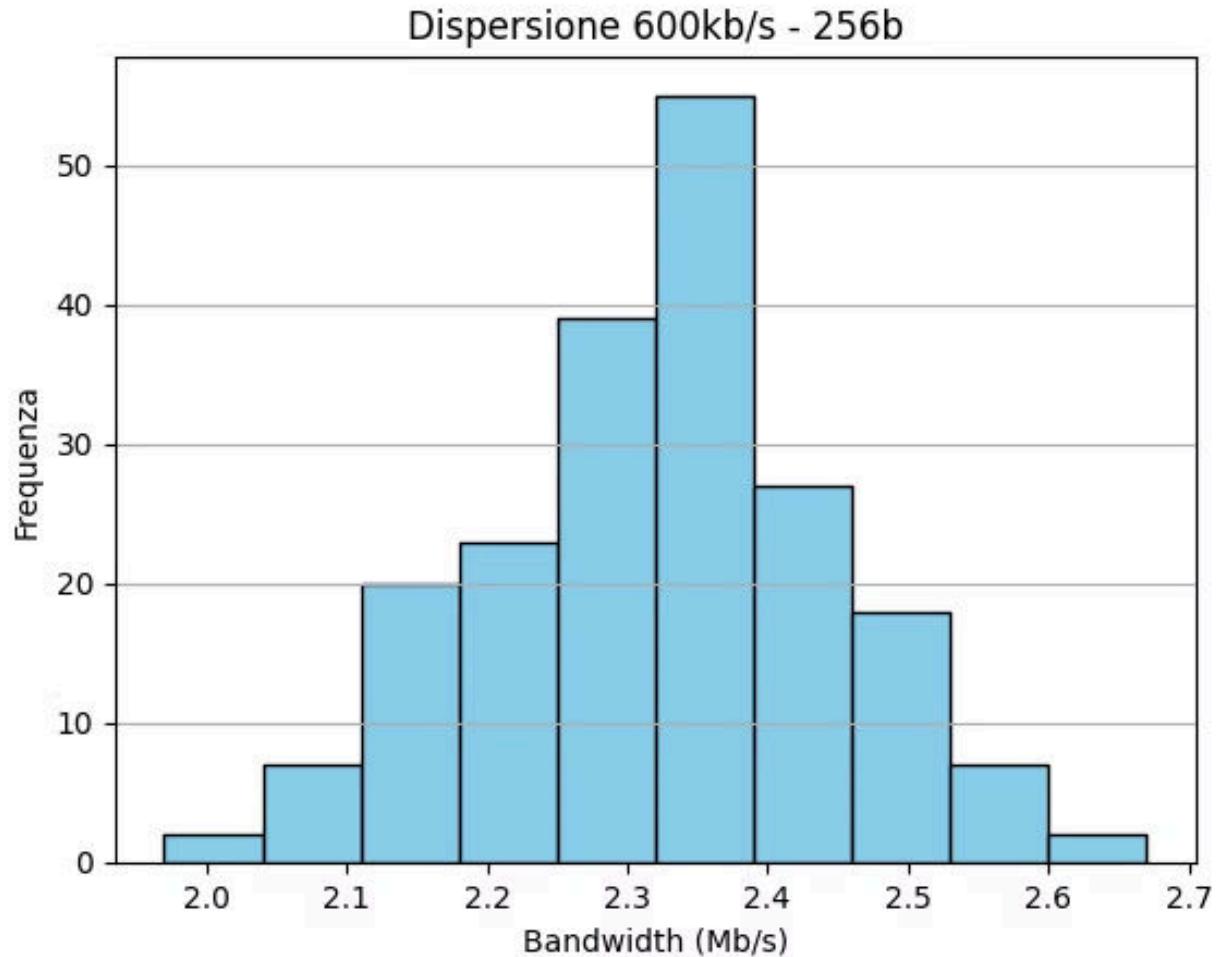
Abbiamo infatti trovato in generale performance migliori e più stabili.



Capacity Test: TCP a 256b

Siamo andati anche a costruire un istogramma dei valori di bandwidth, prendendo 200 campioni.

Abbiamo verificato che è una distribuzione gaussiana, il che indica un collegamento relativamente stabile, con variazioni casuali, ma limitate intorno alla media.



Capacity Test: UDP a 20b

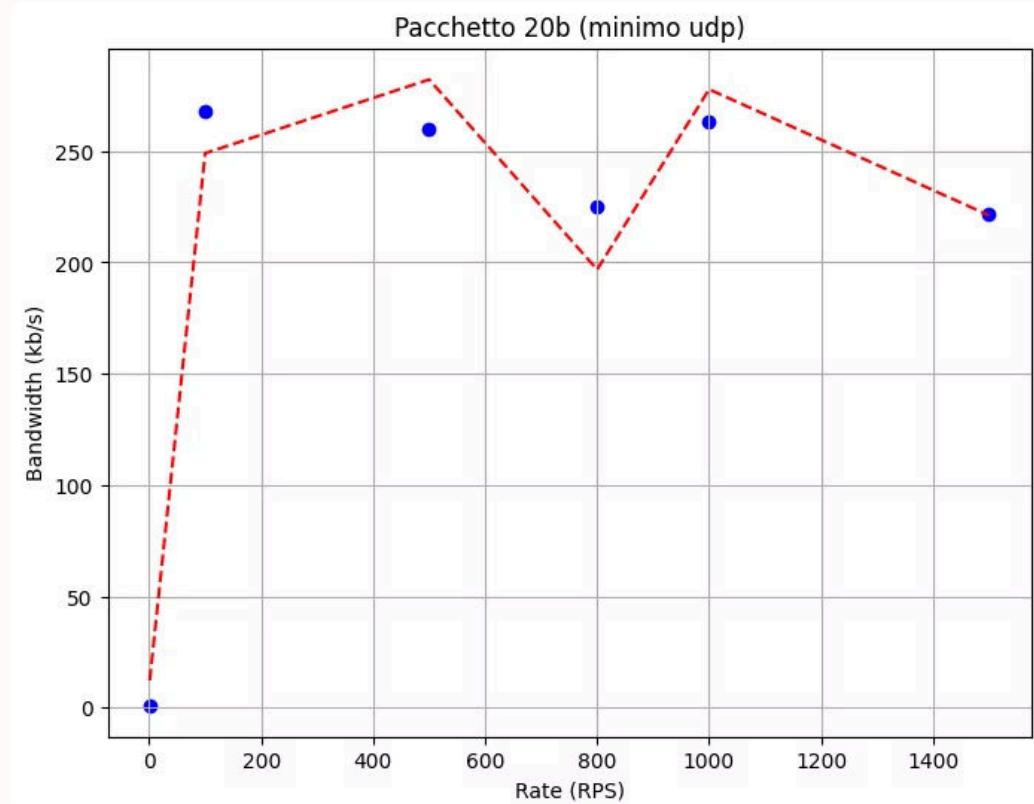
L'obiettivo

Valutare come il protocollo UDP gestisce un grande numero di pacchetti.

Si tratta di una situazione comune in applicazioni che richiedono frequenti aggiornamenti di stato o invio di dati molto frammentati.

Risultato

I risultati attesi sono un alto overhead dovuto alla maggiore proporzione di intestazioni rispetto ai dati effettivi, potenzialmente causando una riduzione dell'efficienza complessiva, ed effettivamente abbiamo avuto una bandwidth che si attesta intorno ai 250kb/s.



Capacity Test: UDP a 256b

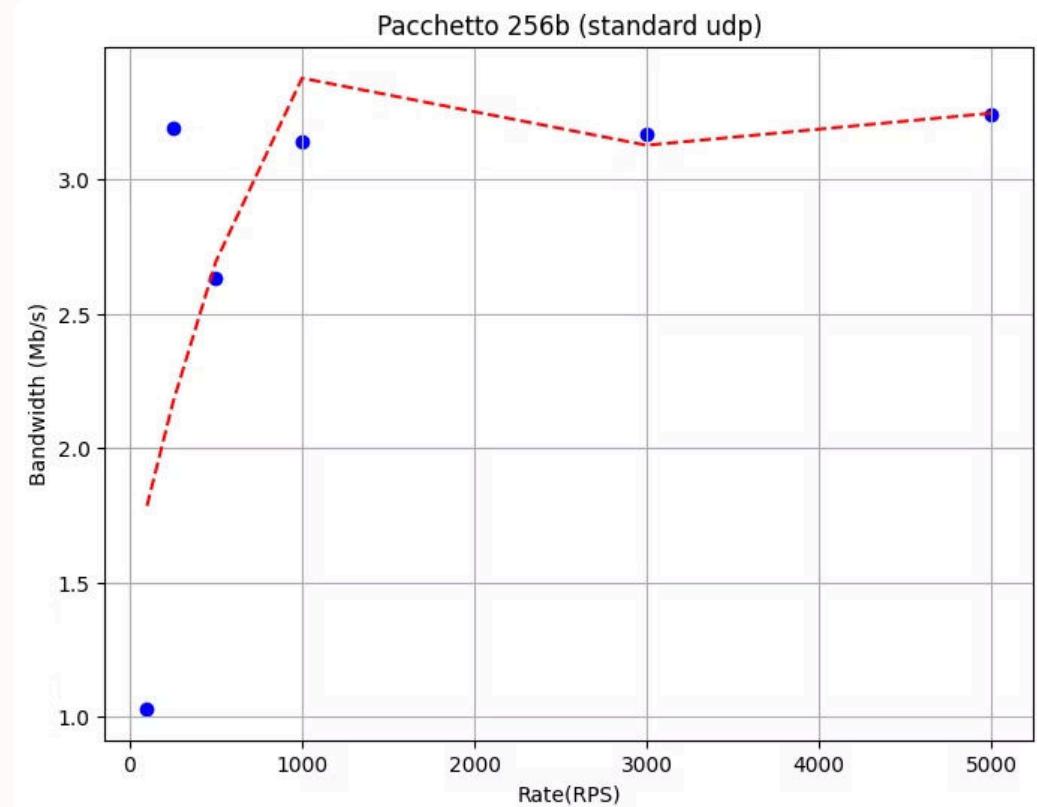
L'obiettivo

Valutare l'efficienza del protocollo UDP con pacchetti di dimensioni comunemente utilizzate in varie applicazioni.

Risultato

Abbiamo ottenuto un aumento significativo della bandwidth, che si attesta a più di 3mb/s e risulta stabile il suo andamento incrementando il Rate.

Interessante notare che si ha una loss sempre compresa tra il 10-12% per qualsiasi valore di rate.



Capacity Test: UDP a 1024b

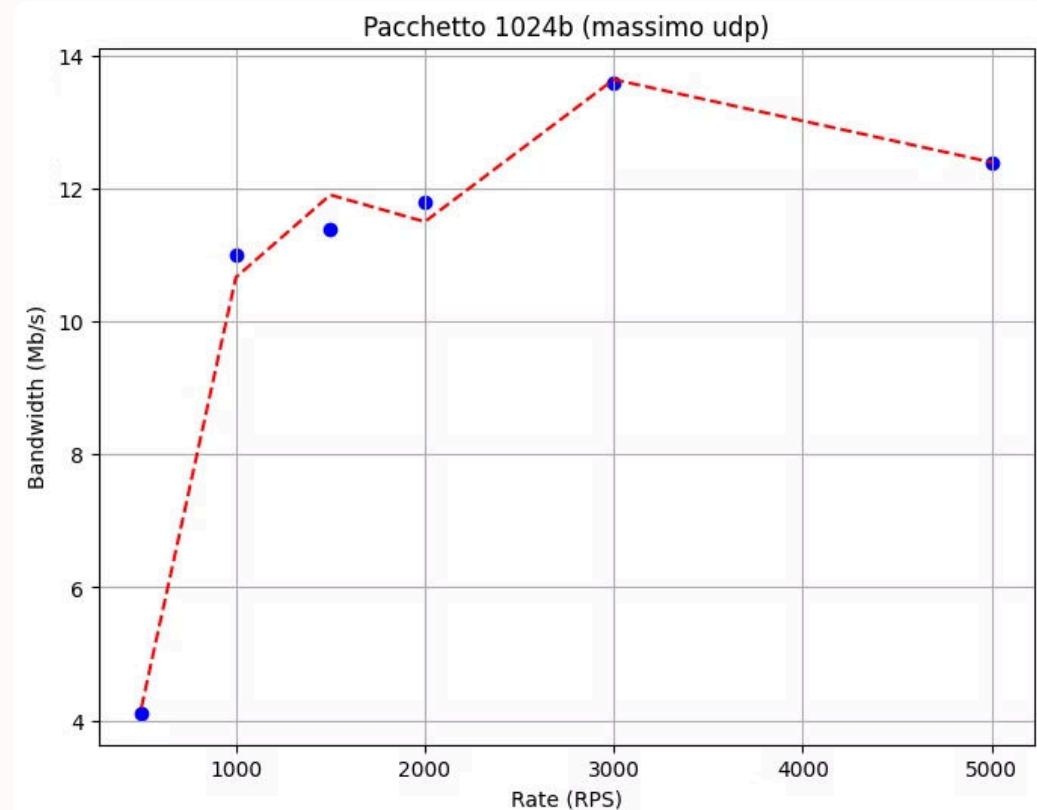
L'obiettivo

Spingere il più possibile il collegamento utilizzando il protocollo alla massima dimensione consentita.

Risultato

Ci aspettavamo la massima efficienza di utilizzo della banda, riduzione dell'overhead, ma possibile aumento della latenza e potenziale perdita di pacchetti in reti congestionate.

Ciò è avvenuto, ma la loss è rimasta agli stessi valori ottenuti dal test precedente, indicando una rete molto stabile e robusta.



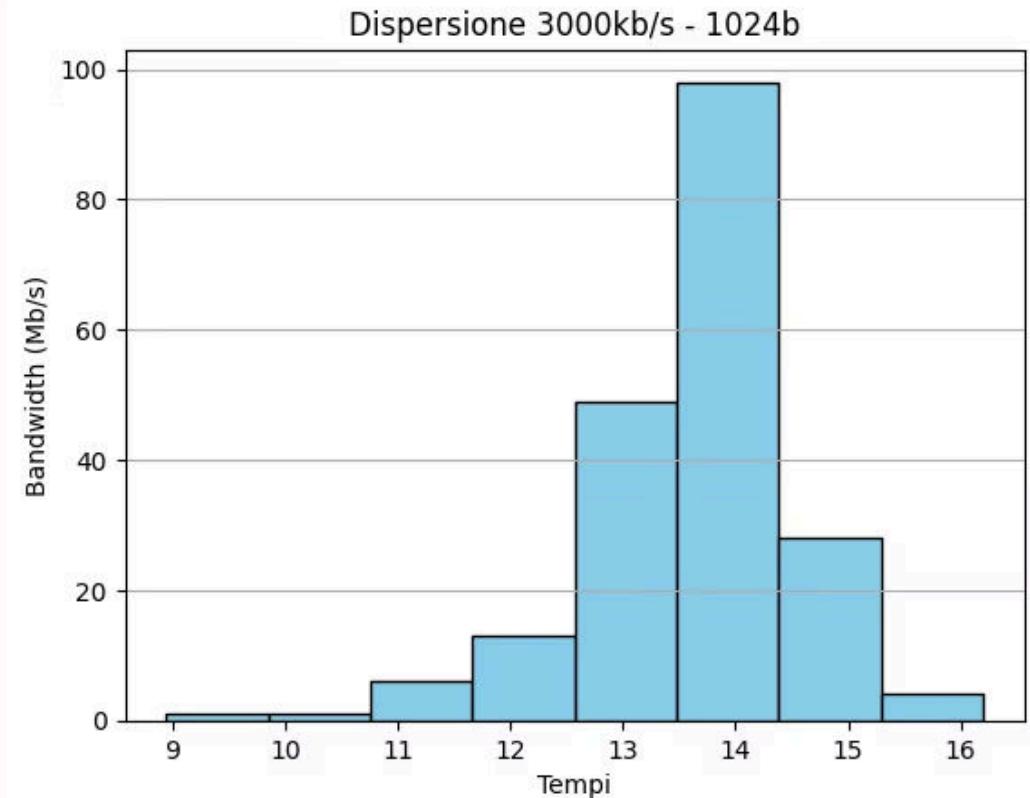
Capacity Test: UDP

Siamo andati anche a costruire un istogramma dei valori di bandwidth, prendendo 200 campioni.

Andando a studiare i qqplot abbiamo verificato che è una distribuzione gaussiana, quindi anche per UDP possiamo trarre le stesse conclusioni di TCP sulla stabilità della rete.

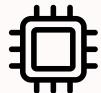
I test UDP hanno fornito risultati altrettanto positivi rispetto ai test TCP. La rete ha dimostrato di gestire in modo efficiente il traffico UDP, garantendo elevate performance in termini di throughput e latenza.

Anche in presenza di picchi improvvisi di traffico UDP, il sistema è stato in grado di mantenere livelli di servizio soddisfacenti, senza incorrere in cali di prestazioni o congestioni della rete.



Design of Experiments (DOE)

Questo studio si propone di investigare come diverse condizioni di stress influenzano le prestazioni delle root cell, abbiamo utilizzato il tool **Stress-ng**, imponendo su tutti i core della cella test casuali di una determinata classe per un tempo prefissato d'osservazione.



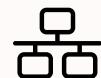
Carico CPU

In questo caso abbiamo imposto carico proveniente da 60 test differenti, distribuiti equamente tra i core della root cell.



Carico VM

Quando la memoria è sotto pressione, il kernel inizia a scrivere pagine in swap. Il sistema VM è fortemente sollecitato anche tramite page fault.



Carico di Rete

In questo caso abbiamo introdotto un aumento del traffico di rete attraverso comunicazione client/server in ricezione e invio.



Carico Interrupt

In questo caso, avviene l'esecuzione di un timer ad alta frequenza per generare un grande carico di interrupt.

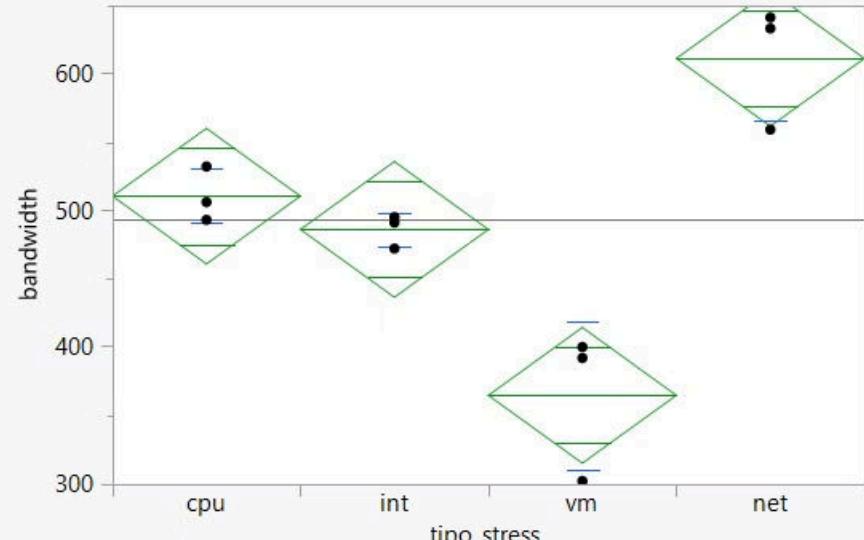


DOE - TCP

I test sono stati effettuati con un pacchetto da 256b e un rate di 400k, quindi secondo i test precedenti ci troviamo in una situazione in cui il sistema lavora bene.

La bandwidth è diminuita significativamente, passando da circa 2.30 Mb/s a circa 500 Kb/s.

Questo decremento rappresenta una riduzione sostanziale dell'efficienza della rete, indicando potenziali problemi di congestione, perdita di pacchetti o altre problematiche che richiedono ulteriori indagini per identificare e mitigare le cause di questo deterioramento delle prestazioni.



ANOVA a una variabile

Riepilogo della stima

R-quadro	0,892564
R-quadro corretto	0,852276
Scarto quadratico medio	37,2514
Media della risposta	493
Osservazioni (o somme pesate)	12

Analisi della varianza

Origine	DF	Somma dei quadrati	Media quadratica	Rapporto F	Prob > F
tipo_stress	3	92228,67	30742,9	22,1544	0,0003*
Errore	8	11101,33	1387,7		
C. totale	11	103330,00			

Medie per ANOVA a una variabile

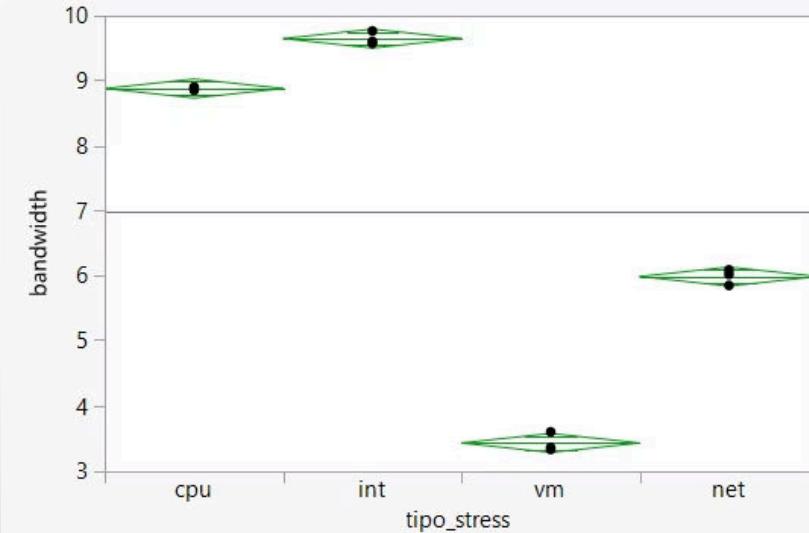
Livello	Numero	Media	Errore std	Inferiore al 95%	Superiore al 95%
cpu	3	510,333	21,507	460,7379	559,93
int	3	486,000	21,507	436,4045	535,60
vm	3	364,667	21,507	315,0712	414,26
net	3	611,000	21,507	561,4045	

DOE - UDP

I test sono stati effettuati con un pacchetto da 1024b e un rate di 2500k, dove otteneva le performance migliori.

In condizioni normali, la bandwidth era di circa 13 Mb/s, sotto stress il sistema ha gestito bene il carico sulla CPU e le interruzioni di sistema, invece è riuscito a gestire discretamente la presenza di traffico di rete attestandosi tra i 6/7 Mb/s. Tuttavia, ha mostrato difficoltà significative nel gestire lo stress sulla memoria virtuale.

Questi risultati evidenziano la robustezza del protocollo UDP nelle condizioni più comuni.



ANOVA a una variabile

Riepilogo della stima

R-quadro	0,998678
R-quadro corretto	0,998183
Scarto quadratico medio	0,109772
Media della risposta	6,984167
Osservazioni (o somme pesate)	12

Analisi della varianza

Origine	DF	Somma dei quadrati	Media quadratica	Rapporto F	Prob > F
tipo_stress	3	72,839492	24,2798	2014,924	<,0001*
Errore	8	0,096400	0,0121		
C. totale	11	72,935892			

Medie per ANOVA a una variabile

Livello	Numero	Media	Errore std	Inferiore al 95%	Superiore al 95%
cpu	3	8,87667	0,06338	8,7305	9,0228
int	3	9,64333	0,06338	9,4972	9,7895
vm	3	3,43000	0,06338	3,2839	3,5761
net	3	5,98667	0,06338	5,8405	6,1334

Conclusioni

Scalabilità	Il sistema basato su Jailhouse e Zephyr si è dimostrato altamente scalabile, consentendo l'aggiunta e la gestione di un numero variabile di VM.
Comunicazione Efficiente	L'utilizzo di IVSHMEM ha permesso una comunicazione rapida e a bassa latenza tra le VM, migliorando le prestazioni complessive.
Integrazione Flessibile	L'adozione di standard di rete ha facilitato l'integrazione del sistema con infrastrutture esistenti e l'adattamento ai vari tool che sono stati utilizzati.
Ulteriore studio	I test di stress hanno evidenziato la necessità di effettuare ulteriori studi sull'attività del sistema, ma anche di dover introdurre un'attenta gestione delle risorse.

Grazie per l'attenzione