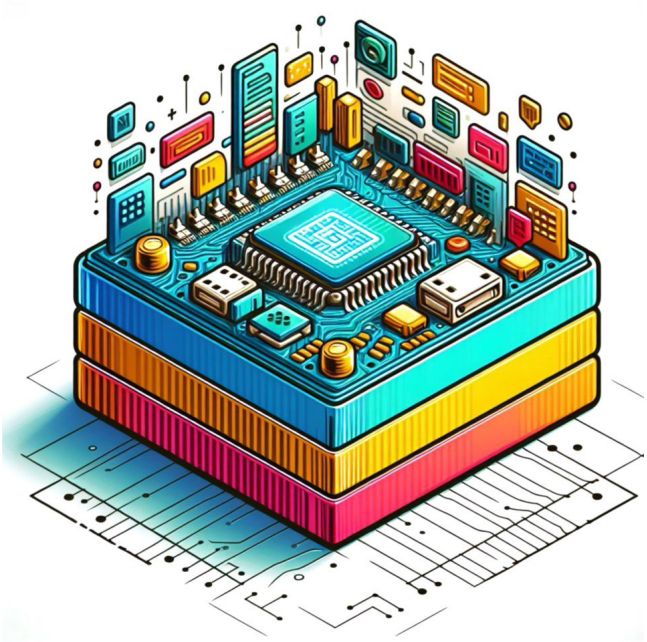
Progetto RTIS: IVSHMEM



Nome Cognome	Matricola
Stefano Marano	M63001428
Vito Romano	M63001504
Paolo Russo	M63001426

Indice

1 Introduzione

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.
Per facilitare la comunicazione tra le VM, verrà utilizzato il protocollo IVSHMEM (Inter-VM Shared Memory), che consente uno scambio di dati efficiente e rapido. Questa tecnologia garantirà 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. Questo assicurerà che il sistema non solo sia 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 approfonditi per valutarne le performance con protocolli di comunicazione TCP e UDP. Questi test hanno permesso di analizzare la latenza, la larghezza di banda e la stabilità delle connessioni di rete tra le VM e con l'esterno.
Inoltre, sono stati valutati gli effetti di vari tipi di stress sul sistema:

- Stress del Carico di Lavoro: Simulazione di carichi di lavoro intensivi per osservare il comportamento delle VM e dell'host sotto condizioni di alta richiesta.
- Stress della Rete: Generazione di traffico di rete elevato per valutare la capacità del sistema di gestire picchi di traffico e mantenere prestazioni accettabili.
- Stress delle Risorse: Sovraccarico delle risorse hardware come CPU e memoria per testare la robustezza dell'isolamento e la gestione delle risorse da parte di Jailhouse.

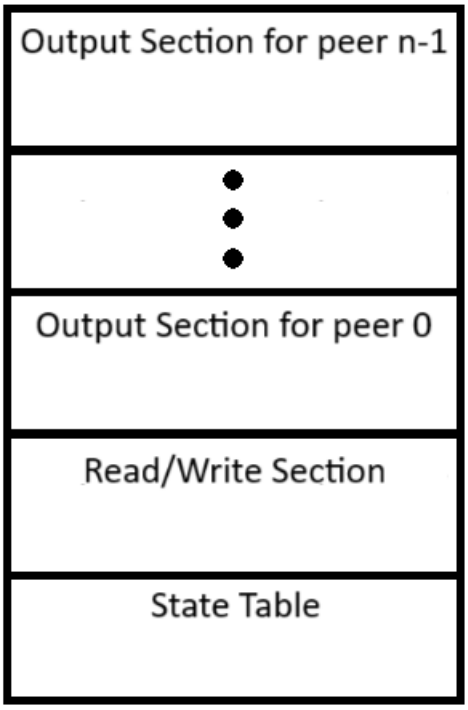
2 | IVSHMEM: Inter-VM Shared Memory Communication

IVSHMEM consente alle macchine virtuali di comunicare tra loro attraverso un meccanismo di memoria condivisa. Ad esempio, gli utenti del segmento industriale possono utilizzare una regione di memoria condivisa per scambiare comandi e risposte tra una VM Windows che riceve input dagli operatori e una VM real-time che esegue attività in tempo reale.
L'obiettivo del modello di dispositivo Inter-VM Shared Memory (IVSHMEM) è quello di definire gli elementi costruttivi minimamente necessari che un hypervisor deve fornire per consentire la comunicazione tra guest e guest. I dettagli dei protocolli di comunicazione devono rimanere opachi per l'hypervisor, in modo che gli ospiti siano liberi di definirli in modo semplice o sofisticato.
A tale scopo, IVSHMEM offre ai suoi utenti le seguenti caratteristiche:

- Interconnessione tra un massimo di 65536 peer
- Regione di memoria condivisa multiuso:
  - Sezione comune di lettura/scrittura;
  - Sezioni di uscita che sono leggibili/scrivibili per un peer e solo leggibili per gli altri;
  - Sezione con gli stati dei peer;
- Segnalazione di eventi tramite interrupt agli end-point;
- Supporto per la gestione del ciclo di vita attraverso lo scambio di valori di stato e la notifica di interrupt sui cambiamenti, supportati da una sezione di memoria condivisa;
- Libera scelta dei protocolli da utilizzare negli strati successivi.
- Comunicazione del tipo di protocollo
- I registri possono essere implementati sia in memory-mapped che tramite I/O, a seconda del supporto della piattaforma e del costo di VM-exit.
- Possibilità di accesso non privilegiato ai registri di I/O o memory-mapped.
- Singola Individuazione e configurazione tramite PCI standard, nessuna complessità aggiuntiva.

Per fornire un collegamento coerente tra i peer, tutte le istanze collegate dei dispositivi IVSHMEM devono essere configurate, create ed eseguite dall'hypervisor in base ai seguenti requisiti:

- Le istanze del dispositivo devono apparire come un dispositivo PCI ai loro utenti;
- La sezione di memoria condivisa in lettura/scrittura deve avere la stessa dimensione per tutti i peer. La dimensione può essere pari a zero;
- Se sono presenti sezioni di uscita della memoria condivisa, deve essercene una riservata per ciascun peer con accesso esclusivo in scrittura. Tutte le sezioni di uscita devono avere la stessa dimensione e devono essere leggibili da tutti i peer;
- La tabella di stato deve avere la stessa dimensione per tutti i peer, deve essere abbastanza grande da contenere i valori di stato di tutti i peer e deve essere di sola lettura per l'utente;
- Le modifiche al registro di stato (scritture esplicite, reset dei peer) devono essere propagate agli altri peer aggiornando la voce corrispondente della tabella di stato ed emettendo un interrupt a tutti gli altri peer se hanno abilitato la ricezione;
- Tutti i peer devono avere a disposizione le stesse funzioni di consegna degli interrupt, cioè MSI-X con lo stesso numero massimo di vettori sulle piattaforme che supportano questo meccanismo, altrimenti INTx con un solo vettore.



La prima sezione è obbligatoria ed costituita dalla tabella di stato, la sua dimensione è definita dal registro State Table Size e non può essere pari a zero. Questa sezione è di sola lettura per tutti i peer.
La seconda sezione consiste in una memoria condivisa che può essere letta/scritta da tutti i peer. La sua dimensione è definita dal registro R/W Section Size. È ammessa una dimensione pari a zero.
La terza e le successive sezioni sono sezioni di uscita, una per ogni peer. Le loro dimensioni sono tutte identiche. La dimensione di una singola sezione di uscita è definita dal registro Output Section Size. 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.
Tutte le dimensioni devono essere arrotondate a multipli di una pagina mappabile per consentire il controllo dell'accesso in base alle restrizioni della sezione.
Per definire le regioni di memoria di un dispositivo ivshmem-net, è disponibile la macro **JAILHOUSE_SHMEM_NET_REGIONS**(indirizzo_base, id). Utilizza 1 MB di di memoria all'indirizzo di base specificato e assegna l'accesso in base all'ID specificato. Quindi per collegare 2 peer, saranno aggiunte 4 regioni di memoria.
Dopo aver creato le regioni di memoria, è necessario aggiungere anche un dispositivo PCI a ciascuna cella collegata, per far ciò si imposta il tipo di dispositivo su **JAILHOUSE_PCI_TYPE_IVSHMEM**. Inoltre il campo bdf deve specificare uno slot bus-device-function non utilizzato da un controller PCI fisico o virtuale e tutti i peer collegati devono usare lo stesso valore bdf per stabilire il collegamento. Tuttavia, possono utilizzare valori di dominio diversi.

3 | Zephyr

Il sistema operativo Zephyr si basa su un kernel di dimensioni ridotte progettato per l'uso in sistemi embedded e con risorse limitate: da semplici sensori ambientali a sofisticati controller embedded, smart watches e applicazioni wireless IoT. 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.
Nel contesto di Zephyr OS, un sottosistema si riferisce a una parte logicamente distinta del sistema operativo che gestisce funzionalità specifiche o fornisce determinati servizi. Ogni sottosistema è progettato per essere modulare e può essere configurato, personalizzato ed esteso per soddisfare i requisiti di diverse applicazioni embedded.

Zephyr offre un numero elevato e sempre crescente di funzionalità, tra cui: Un'ampia suite di servizi del kernel, tra cui:

- **Servizi multi-threading per thread cooperativi**, basati sulla priorità, non preemptive e preemptive con time-slicing round robin opzionale. Include il supporto dell'API POSIX pthreads compatibile.
- **Algoritmi di programmazione multipli:** Scheduling cooperativo e preemptive, Earliest Deadline First (EDF), Meta IRQ scheduling che implementa il comportamento "interrupt bottom half" o "tasklet".
- **Protezioni della memoria** specifiche per l'architettura:
  - Protezione dallo stack-overflow;
  - il tracciamento dei permessi degli oggetti del kerner e dei driver dei dispositivi;
  - l'isolamento dei thread con protezione della memoria a livello di thread sulle architetture x86, ARC e ARM, sullo spazio utente e sui domini di memoria.
- Per le piattaforme senza MMU/MPU e i dispositivi con vincoli di memoria, supporta la combinazione di codice specifico dell'applicazione con un kernel personalizzato per creare un'immagine monolitica che viene caricata ed eseguita sull'hardware del sistema. Sia il codice dell'applicazione che quello del kernel vengono eseguiti in un unico spazio di indirizzi condiviso.
- **Modello ottimizzato del driver di periferica** Oltre al supporto a Devicetree, fornisce un modello di dispositivo coerente per la configurazione dei driver che fanno parte della piattaforma/sistema e un modello coerente per l'inizializzazione di tutti i driver configurati nel sistema e consente il riutilizzo dei driver tra piattaforme che hanno dispositivi/blocchi IP comuni.
- **Stack di rete nativo** che supporta diversi protocolli, completo e ottimizzato, compreso il supporto con i socket LwM2M e BSD. È previsto anche il supporto di OpenThread (su chipset Nordic), una rete mesh progettata per collegare in modo sicuro e affidabile centinaia di prodotti in tutta la casa.

3.1 | Stack di Rete

Lo stack di rete è stratificato ed è composto dalle seguenti parti:

- **Applicazione di rete.** L'applicazione di rete può utilizzare le librerie di protocollo a livello di applicazione, fornite o accedere direttamente all'API BSD socket, per creare una connessione di rete, inviare o ricevere dati e chiudere una connessione.

  L'applicazione può anche utilizzare l'API di gestione della rete per configurare la rete e impostare i relativi parametri, come le opzioni del collegamento, l'avvio di una scansione (se applicabile), l'ascolto degli eventi di configurazione della rete, ecc.

  L'API dell'interfaccia di rete può essere utilizzata per impostare l'indirizzo IP di un'interfaccia di rete, per disattivare l'interfaccia di rete, ecc.
- **Protocolli di rete.** Protocolli di rete principali come IPv6, IPv4, UDP, TCP, ICMPv4 e ICMPv6 sono supportati, ma anche implementazioni per vari protocolli a livello di applicazione, come ad esempio: CoAP, LWM2M e MQTT.
- **Astrazione dell'interfaccia di rete.** Fornisce funzionalità comuni a tutte le interfacce di rete, come l'impostazione dell'interfaccia di rete, ecc.
- **Tecnologie di rete L2.** Fornisce un'API comune per l'invio e la ricezione di dati da e verso un dispositivo di rete reale.

  Queste tecnologie di rete includono Ethernet, IEEE 802.15.4, Bluetooth, CANBUS, ecc. Inoltre alcune di queste tecnologie supportano la compressione dell'intestazione IPv6 (6Lo); Ad esempio, ARP per IPv4 viene eseguito dal componente Ethernet.
- **Driver dei dispositivi di rete.** I driver dei dispositivi di basso livello gestiscono l'invio o la ricezione fisica dei pacchetti di rete.

3.2 | Ivshmem Ethernet

Ivshmem Ethernet utilizza una regione di memoria condivisa per memorizzare i dati che devono essere scambiati tra le VM, tale è accessibile a tutte le VM che partecipano alla comunicazione. Per scambiare dati, una VM scrive i dati nella regione di memoria condivisa e poi notifica (tramite Doorbell) all'altra VM che i dati sono disponibili. L'altra VM quindi legge i dati dalla regione di memoria condivisa.
Per utilizzare ivshmem Ethernet in Zephyr, è necessario abilitare la configurazione CONFIG_IVSHMEM e configurare il driver Ethernet ivshmem. 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 la 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 Zephyr forniscono funzioni per allocare e deallocare la memoria condivisa, scrivere e leggere i dati dalla memoria condivisa e notificare alle altre VM che i dati sono disponibili.

perché si usano i meccanismi di rete? Anche se a livello fisico le VM possono accedere ad una memoria condivisa, la comunicazione tra di esse avviene spesso utilizzando i classici meccanismi di rete per diverse ragioni.
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. Questo aiuta a prevenire accessi non autorizzati, attacchi informatici e la diffusione di malware tra le VM.
I meccanismi di rete forniscono un'interfaccia astratta e standardizzata per la comunicazione, indipendentemente dal tipo di hardware o software sottostante, quindi consente una maggiore flessibilità nella configurazione e nella gestione delle VM, 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. Inoltre, facilita lo sviluppo di applicazioni che possono comunicare con le VM indipendentemente dalla loro implementazione specifica.
I meccanismi di rete sono ottimizzati per il trasferimento di dati efficienti e possono supportare reti con un gran numero di peer, ciò è particolarmente importante per le applicazioni in cloud computing che devono gestire un gran numero di VM contemporaneamente. L'utilizzo di meccanismi di rete consente di distribuire il carico di lavoro su più reti, migliorando le prestazioni e la scalabilità complessive del sistema. Ciò facilita anche l'interoperabilità tra le 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.

Ivshmem DoorBell Zephyr integra il meccanismo di notificazione dei dispositivi ivshmem come parte del suo stack di comunicazione inter-VM utilizzando vari componenti del sistema operativo.
La notifica (Doorbell) del dispositivo ivshmem consente alle macchine virtuali con dispositivi ivshmem abilitati di notificarsi (interrompersi) a vicenda seguendo questo flusso:

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.

La VM di destinazione alla ricezione dell'interrupt si occupa del inoltro all'applicazione, 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.

4 | Implementazione del sistema

Nel capitolo che segue, illustreremo in dettaglio il processo di realizzazione del sistema sfruttando il progetto Runphi. Questo progetto ha rappresentato una base solida su cui costruire un ambiente virtuale funzionale, utilizzando l'emulazione QEMU, il Real-Time Operating System (RTOS) Zephyr e la configurazione di una rete interconnessa tra varie macchine virtuali (VM).
Attraverso queste fasi, il capitolo offrirà una panoramica completa su come integrare e utilizzare questi strumenti per creare un sistema robusto ed efficiente. Dettagli tecnici, configurazioni specifiche e best practice saranno illustrati per fornire una guida pratica e applicabile a chiunque desideri replicare o adattare questo setup per i propri progetti.

4.1 | Interfaccia TAP

Un'interfaccia TAP (Terminal Access Point) è un'interfaccia di rete virtuale che opera a livello del collegamento dati (livello 2 del modello OSI). Viene utilizzata per simulare una scheda di rete fisica, permettendo il trasferimento di pacchetti Ethernet tra il sistema operativo e le applicazioni, quindi riceve pacchetti Ethernet dall'applicazione, li invia attraverso la rete virtuale e viceversa, come farebbe una scheda di rete fisica. 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. Questo permette di trasportare pacchetti di rete attraverso una rete incompatibile o sicura.
I pacchetti di rete sono incapsulati in un altro protocollo. Ad esempio, un pacchetto IP può essere incapsulato in un protocollo GRE (Generic Routing Encapsulation), in questo modo permette di trasportare pacchetti attraverso reti che normalmente non supporterebbero quel tipo di pacchetto. Ad esempio, trasportare pacchetti IPv6 attraverso una rete IPv4.
Per i nostri utilizzi abbiamo utilizzato **QEMU**, per emulare un processore cortex a55 e su di esso introdurre jailhouse. Avevamo bisogno di realizzare un interfaccia equivalente a un collegamento alla rete LAN per realizzare i nostri test.
QEMU può utilizzare le interfacce TAP per fornire connettività di rete alle macchine virtuali. I pacchetti appaiono come provenienti da un'interfaccia di rete reale (spesso Ethernet) all'interno della macchina virtuale e tutto il traffico inviato dalla macchina virtuale attraverso la sua interfaccia di rete appare sull'interfaccia TAP dell'host. Analogamente, i pacchetti ricevuti dall'interfaccia TAP dell'host vengono consegnati alla macchina virtuale, permettendo una comunicazione bidirezionale e con basso overhead.

Supporto del Driver Bridge 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 flessibili e facilmente testabile, inoltre collegando il gateway al bridge è possibile una comunicazione trasparente tra VM e dispositivi fisici sulla stessa LAN e quindi estendere facilmente l'architettura.

4.2 | Routing

Durante la fase di configurazione della comunicazione tra le vari celle Zephyr e il sistema sottostante, abbiamo valutato diverse opzioni per ottimizzare la connessione.

Scelta della Configurazione di Rete: Bridge Doppi L'idea era di implementare un doppio bridge per migliorare la comunicazione e l'integrazione tra i vari componenti del sistema.

- **Bridge Primario** (br0): Questo bridge collega la macchina host Linux con la root cell attraverso l'utilizzo dell'interfaccai TAP e il tunneling
- **Bridge Secondario**: Al bridge viene assegnato un indirizzo IP e il traffico ad esso destinato è consentito, ma nessuna interfaccia reale (ad esempio eth0) è collegata al bridge, 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.

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 possibilmente dovuto all'utilizzo di IVSHMEM, poiché lo stesso bridge riesce a collegarmi attraverso il tunneling al bridge primario. Questo rende impossibile utilizzare il doppio bridge come originariamente previsto.
Alla luce di questi problemi, abbiamo deciso di abbandonare l'idea e ci siamo concentrati sulla creazione di regole di routing per instradare correttamente il traffico da e verso Zephyr.

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. Questo è utile per reindirizzare il traffico a un indirizzo IP interno specifico all'interno di una rete privata.
Socat è uno strumento di rete che può creare connessioni bidirezionali tra due flussi di dati. È molto versatile e può essere utilizzato 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.

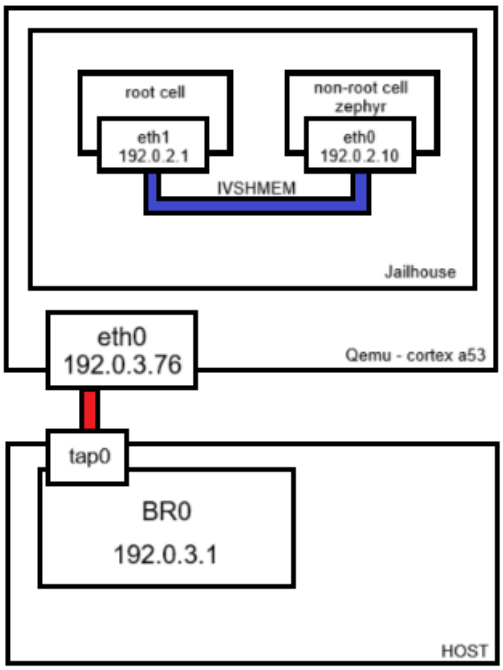


Figura 4.1: Struttura della rete.

4.3 | Configurazione Zephyr

Le prestazioni di rete possono variare in base alla configurazione e all'ottimizzazione dei parametri di sistema. Migliorare le performance di TCP/UDP su Zephyr richiede un'attenta regolazione di diverse impostazioni di configurazione.
In questa introduzione, esploreremo le possibilità di ottimizzazione delle performance di TCP/UDP su Zephyr, delineando i principali parametri di configurazione e le tecniche di tuning che possono essere impiegate per massimizzare l'efficacia delle connessioni di rete in ambienti embedded.
Quando si considera il tempo totale di trasmissione di un singolo frame, non bisogna limitarsi al solo tempo di trasmissione fisica, ma anche al tempo necessario per l'elaborazione dei livelli UDP/IP. Nella configurazione predefinita, Zephyr esegue l'elaborazione dei livelli L4 (trasporto), L3 (rete), L2 (data link) e del driver in un unico thread. Questo approccio cumulativo dei tempi di elaborazione può influenzare negativamente il throughput finale.

Una strategia efficace per aumentare il throughput è abilitare la coda di trasmissione(CONFIG_NET_TC_TX_COUNT). Con questa configurazione, un pacchetto viene messo in coda invece di essere passato direttamente al livello L2, 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. Un ulteriore miglioramento del throughput può essere ottenuto ottimizzando la dimensione del buffer di rete per adattarsi all'MTU (Maximum Transmission Unit) effettivo della rete (CONFIG_NET_BUF_DATA_SIZE). 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.
Di seguito riportiamo i risultati ottenuti su una STM nucleo_h723zg andando opportunamente a fare tuning dei parametri giusti.

Configurazione	**TCP RX/TX Window**	**UDP upload**	**TCP upload**	**UDP download**	**TCP 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	40000	77.9 Mbits/sec	75.0 Mbits/sec	88.12 Mbits/sec	79.56 Mbits/sec

4.4 | Guida all'implementazione

1. **Preparazione dei File Necessari:**
   - Copia i file forniti nei seguenti percorsi all'interno della cartella **runphi**:

     `.../runphi/environment/qemu/jailhouse/install/root/scripts_jailhouse_qemu/`
     `.../runphi/environment/qemu/jailhouse/install/etc/dhcp/dhcpd.conf`

     Nota: Se la directory **/etc/dhcp/** non esiste, creala manualmente.
   - Assicurati che i due file sopra menzionati abbiano i permessi adeguati:
     - Fai clic destro su ciascun file.
     - Vai su "Permessi" e assegna "Lettura e Scrittura" per tutti gli utenti.
     - Spunta l'opzione "Permetti di eseguire come programma".
2. **Avvio di QEMU con lo Script:**
   - Apri un terminale e avvia lo script **start_qemu.sh** con i permessi di amministratore:

     `sudo ./start_qemu.sh`
   - Inserisci il tuo user e password quando richiesto.
3. **Caricamento dei File nel Sistema Ospite (HOST):**
   - In un nuovo terminale, naviga nella cartella **runphi**.
     Utilizza lo script **load_install_dir_to_remote.sh** per caricare i file precedentemente inseriti:

     `cd /path/to/runphi`
     `./scripts/remote/load_install_dir_to_remote.sh`
   - **Esci dalla Macchina QEMU**
4. **Modifica della Configurazione di Rete di QEMU:**
   - Modifica lo script **start_qemu.sh** sostituendo la linea seguente:

```
netdev user,id=mynet0,restrict=off,hostfwd=tcp::5022-:22
```

con questa nuova configurazione:

```
netdev tap,id=mynet0,ifname=tap0,script=no,downscript=no
```

Nota: Fare attenzione alla formattazione!

5. **Installazione dei Pacchetti Necessari:**
   - Apri un nuovo terminale sulla macchina HOST e installa i pacchetti **bridge-utils** e **dnsmasq** con il comando:

     ```
     sudo apt-get install bridge-utils dnsmasq
     ```

   - **Nota:** I passaggi da 1 a 6 sono da eseguire solo una volta.
6. **Configurazione del Bridge di Rete:**
   - Esegui lo script **setup_bridge.sh** sulla macchina host, con i permessi di amministratore:

     ```
     sudo ./setup_bridge.sh
     ```

7. **Riavvio di QEMU:**
   - In un nuovo terminale, avvia nuovamente lo script **start_qemu.sh** e attendi che si carichi. Inserisci user e password quando richiesto:

     ```
     sudo ./start_qemu.sh
     ```

8. **Avvio di Jailhouse su QEMU:**
   - All'interno della macchina virtuale QEMU, naviga nella cartella **scripts_jailhouse_qemu** e lancia lo script **jailhouse_start.sh**:

     ```
     cd /path/to/scripts_jailhouse_qemu
     ./jailhouse_start.sh
     ```

9. **Configurazione della Rete su root cell:**
   - In un nuovo terminale da macchina host, connetti al sistema QEMU usando SSH:

     ```
     ssh root@192.0.3.76
     ```

     Nota: Eventualmente sostituisci **192.0.3.76** con l'indirizzo IP effettivo dell'interfaccia **tap0**
   - Inserisci la password quando richiesto e accedi alla cartella **scripts_jailhouse_qemu**. Esegui lo script **config_net.sh**:

     ```
     cd /path/to/scripts_jailhouse_qemu
     ./config_net.sh
     ```

10. **Avvio del Servizio DHCP su root cell:**
    - Nella finestra della macchina root, esegui lo script **zephyr_dhcp.sh** situato nella cartella **scripts_jailhouse_qemu**:

      ```
      ./zephyr_dhcp.sh
      ```

    - Premi **Invio** per lasciare il server DHCP in esecuzione in background.
11. **Esecuzione dei Test con zperf:**

- Per eseguire i test con zperf, lancia il seguente comando su root cell:

  `socat tcp-listen:5001,reuseaddr tcp:192.0.3.1:5001`

- Assicurati che il sistema sia pronto a ricevere le connessioni in ascolto sulla porta 5001.

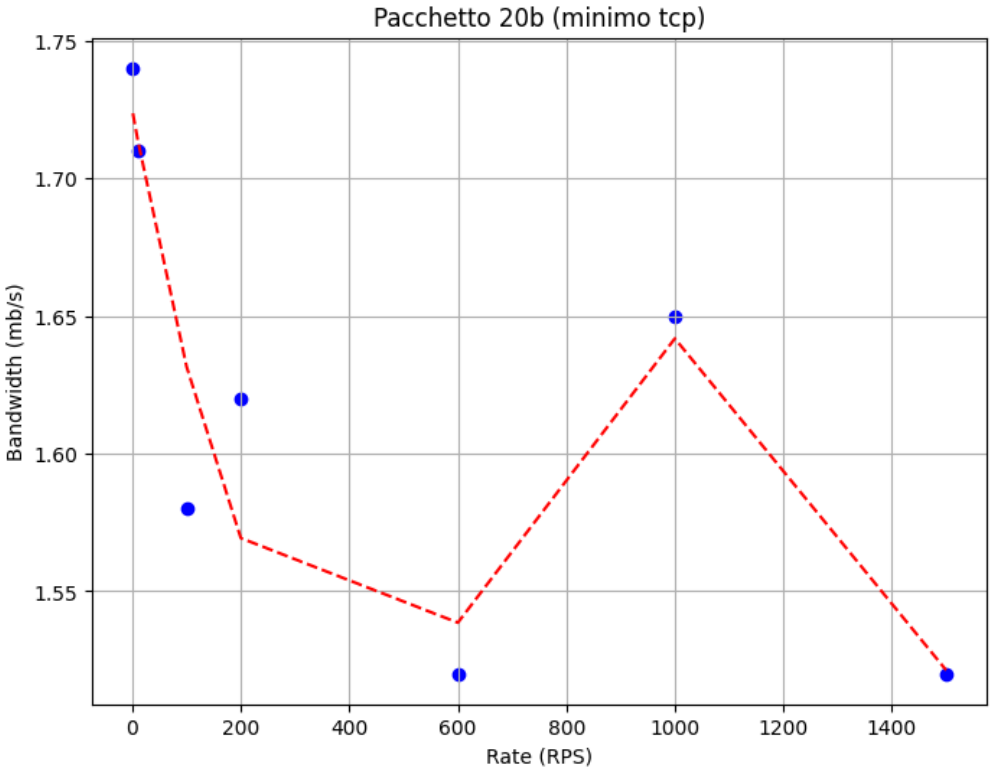
5 | Test e risultati

Questo report documenta una serie di test di capacità condotti per valutare le prestazioni di un sistema in tempo reale nella comunicazione tra macchine virtuali (VM). L'obiettivo principale di questi test è stato analizzare il comportamento del sistema utilizzando sia il protocollo UDP che il protocollo TCP, con particolare attenzione alla larghezza di banda e al tempo di esecuzione peggiore.
Inoltre, sono stati condotti esperimenti di progettazione (DOE, Design of Experiments) per valutare l'impatto dello stress sulla root cell. Questi esperimenti hanno permesso di identificare come diversi livelli di carico influenzino le prestazioni complessive del sistema, contribuendo a delineare le condizioni operative ottimali e a identificare eventuali colli di bottiglia.
Gli esperimenti sono stati svolti utilizzando zperf sulla cella non-root e iperf sull'host. È importante sottolineare che il tuning del kernel Zephyr è stato effettuato in modo molto basico.
Nella configurazione sperimentale, sulla macchina host era attivo un server, permettendo così di valutare l'upload dalla macchina Zephyr. Questa configurazione è stata necessaria a causa di un conflitto che impediva l'attivazione di un server in modalità download sulla macchina Zephyr.

5.1 | Capacity su TCP

Abbiamo effettuato due test di capacità sul protocollo TCP in upload da un dispositivo Zephyr verso una macchina host. L'obiettivo era valutare l'efficienza della trasmissione con pacchetti di dimensioni diverse.

Primo Test abbiamo utilizzato pacchetti di dimensioni molto ridotte. L'obiettivo era verificare come il sistema gestisce un elevato numero di pacchetti di piccole dimensioni e valutare l'overhead introdotto dalla frequente creazione e gestione di molti pacchetti piccoli.
Infatti ci aspettiamo una minor ritardo di ritrasmissione e minor rischio di frammentazione, migliorando l'efficienza del trasferimento, ma anche maggiore overhead poiché ogni pacchetto TCP/IP ha un'intestazione di almeno 40 byte (20 byte per TCP e 20 byte per IP), quindi significa una maggiore percentuale di overhead rispetto ai dati utili. Inoltre c'è un maggiore carico di lavoro: 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 macchina emulata.



Secondo Test abbiamo utilizzato pacchetti di dimensioni standard per le comunicazioni TCP.
In questo caso ci aspettiamo maggiore efficienza e Velocità di trasmissione, poichè con pacchetti grandi, la quantità di overhead (intestazione TCP/IP) per byte di dati utile è ridotta. Questo significa che si trasmette più dati utili per ogni byte di overhead, aumentando l'efficienza complessiva della trasmissione ed 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 passo attraverso la root cell. Abbiamo infatti trovato in generale performance migliori e più stabili.

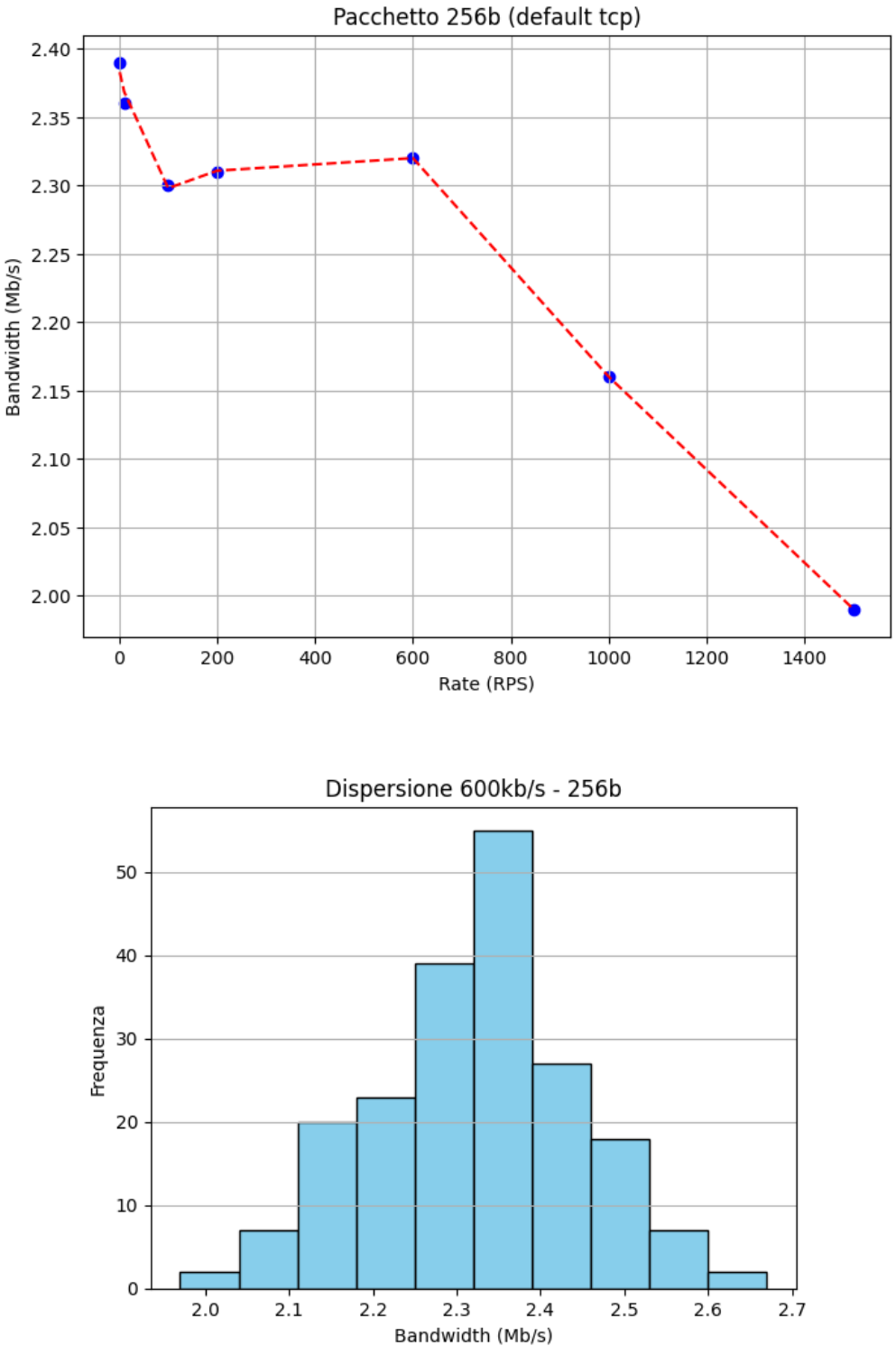
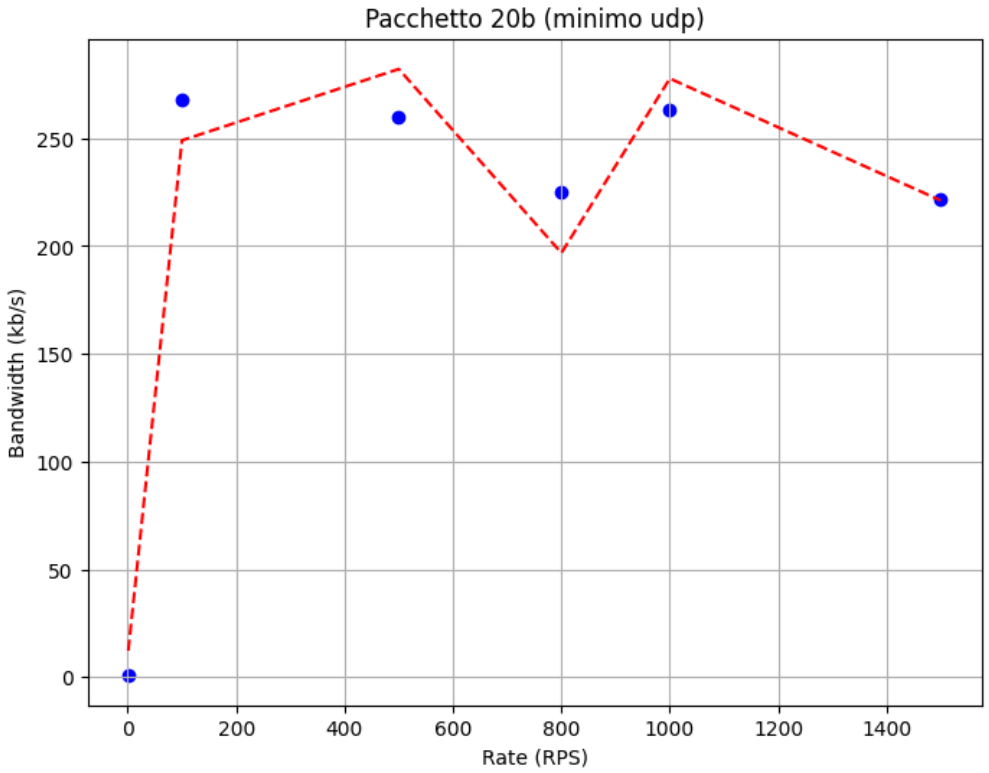


Figura 5.1: Distribuzione dei valori di Bandwidth per TCP

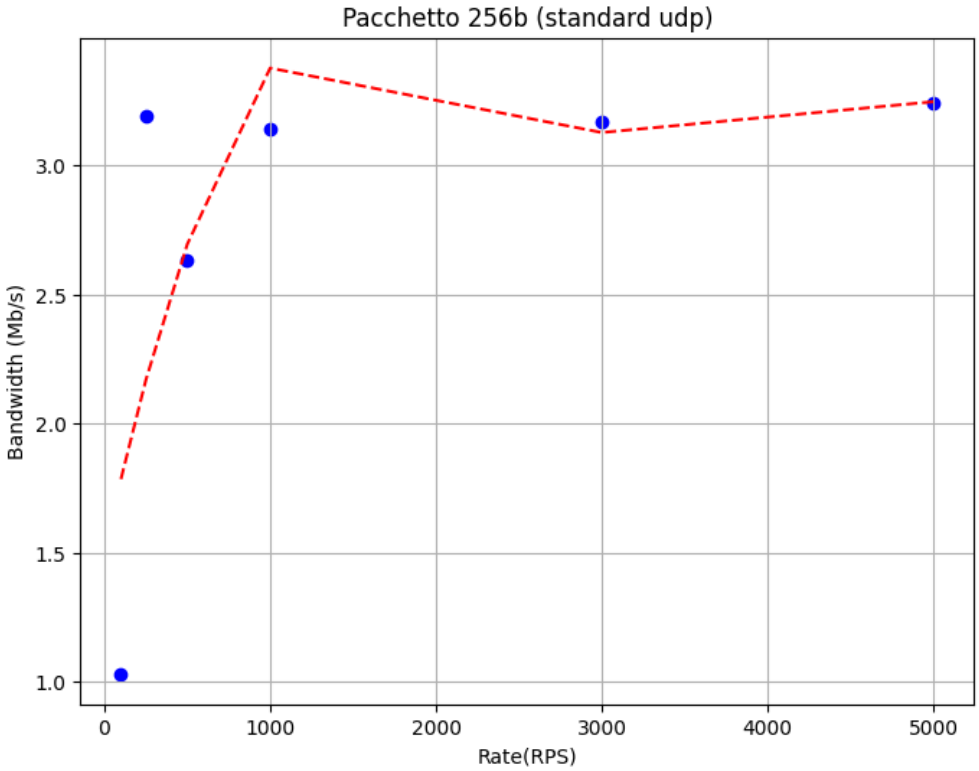
5.2 | Capacity su UDP

Abbiamo effettuato tre test di capacità sul protocollo UDP in upload da un dispositivo Zephyr verso una macchina host.
L'efficienza di un protocollo di comunicazione come l'UDP varia notevolmente in base alla dimensione dei pacchetti utilizzati. In generale, ci si aspetta che pacchetti più grandi siano più efficienti in termini di utilizzo della larghezza di banda, ma con maggiori rischi di perdita di dati e ritrasmissioni in ambienti di rete non ottimali.

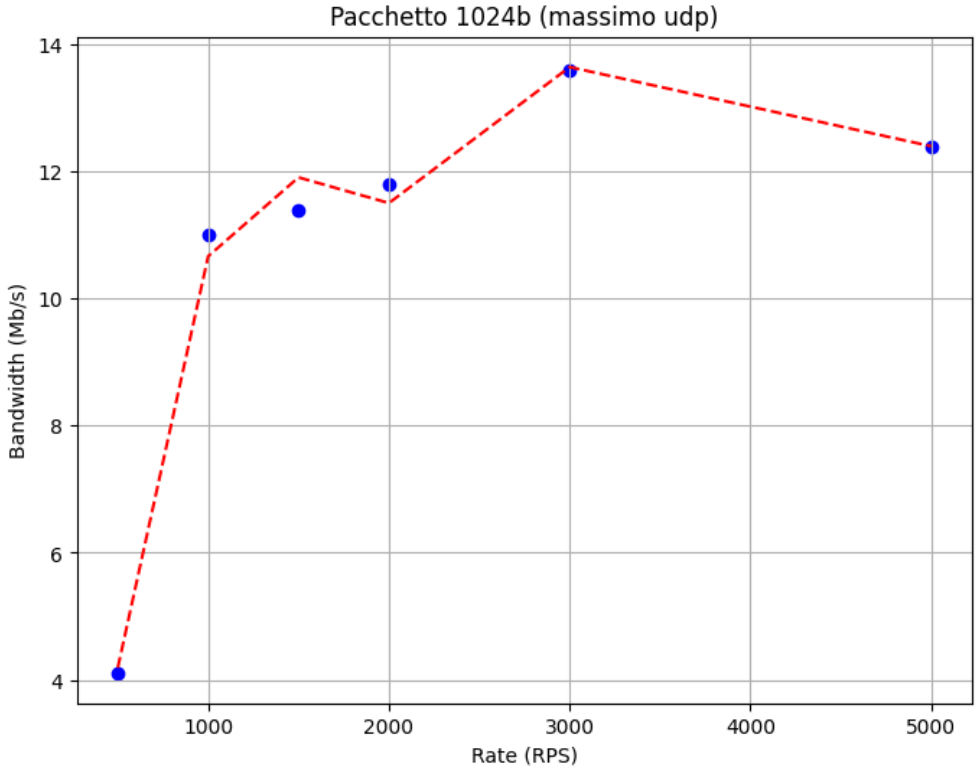
Il primo test mira a valutare come il protocollo UDP gestisce un grande numero di pacchetti piccoli. Si tratta di una situazione comune in applicazioni che richiedono frequenti aggiornamenti di stato o invio di dati molto frammentati.
I risultati attesi sono un alto overhead dovuto alla maggiore proporzione di intestazioni rispetto ai dati effettivi, potenzialmente causando una riduzione dell'efficienza complessiva, ed effettivamente abbiamo avuto una bandwidth che si attesta intoerno ai 250kb/s.



Il secondo test valuta l'efficienza del protocollo UDP con pacchetti di dimensioni comunemente utilizzate in varie applicazioni. Le dimensioni standard rappresentano un equilibrio tra overhead di intestazione e volume di dati.
Abbiamo ottenuto un aumento significativo della bandwidth, che si attesta a più di 3mb/s e risulta pure stabile il suo andamento incrementando il Rate avendo una loss sempre compresa tra il 10-12%.



Il terzo test 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.



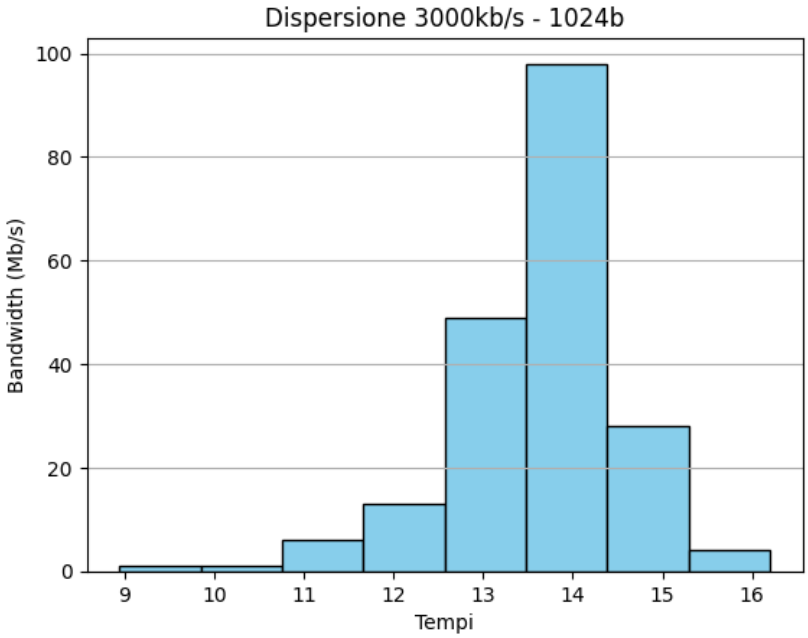
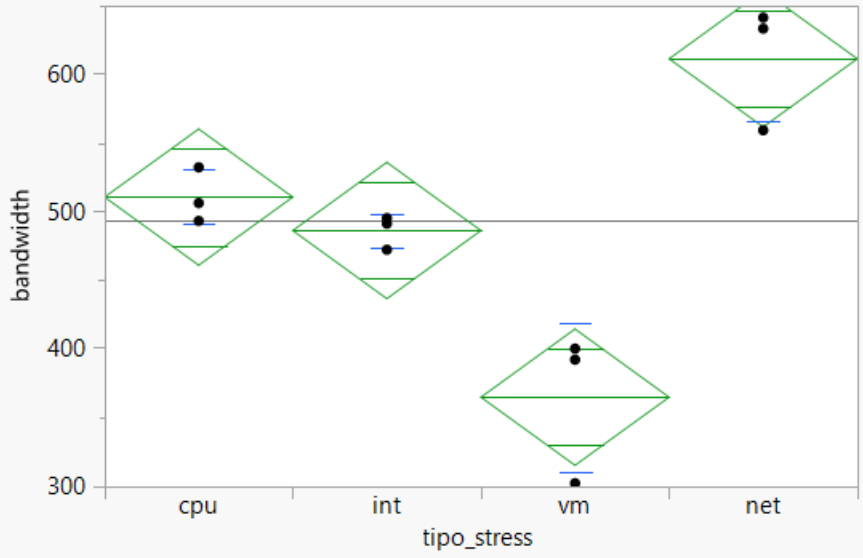


Figura 5.2: Distribuzione dei valori di Bandwidth per UDP

5.3 | Doe

L'obiettivo di questo Design of Experiments (DOE) è analizzare quantitativamente come ciascuno di questi stress individualmente influisce sulle prestazioni di una connessione TCP o UDP.
Per il protocollo TCP, il sistema è stato impostato su un rate di 400 con pacchetti da 256 byte. Le performance sono calate notevolmente, passando dai circa 2.30 Mb/s del test di capacità a valori compresi tra 300 e 600 Kb/s, a seconda del livello di stress applicato.
Per quanto riguarda il protocollo UDP, abbiamo testato con un rate di 3000 e pacchetti da 1024 byte. A differenza del caso precedente, dove tutti i tipi di stress avevano un impatto significativo, in questo scenario l'impatto è generalmente minore. Sebbene alcuni tipi di stress abbiano ridotto notevolmente la larghezza di banda, i valori restano comunque nell'ordine dei Mb/s.



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

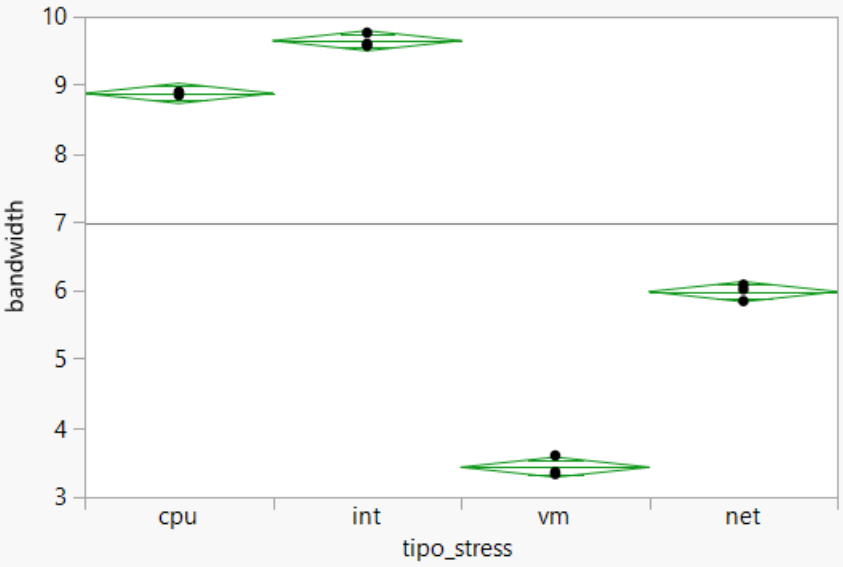
Analisi della varianza

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	660,60

Figura 5.3: Doe per TCP



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	6,1328

Figura 5.4: Doe per UDP