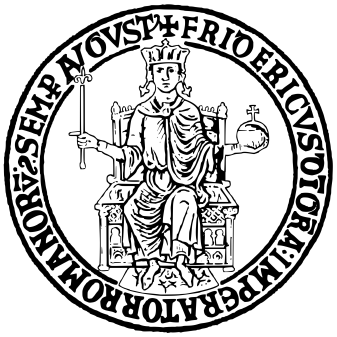
Università degli Studi di Napoli
Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Individuazione e mitigazione di attacchi DoS in un ambiente SDN usando Mininet e switch OpenFlow

SDN Project Work (Firewall)

Network and Cloud Infrastructures

Studente: Iovino Andrea

Matricola: M63001869

Studente: Papale Livio

Matricola: M63001824

Docente: Ventre Giorgio

Anno Accademico 2024/2025

1 Network Setup and Performance

In questo lavoro, partendo da un elaborato dello scorso anno che utilizziamo come base e miglioriamo in alcuni aspetti come richiesto, si utilizza l'emulatore di rete **Mininet**. Ripercorriamo brevemente quanto è stato svolto nell'elaborato dello scorso anno.

Si simulano due scenari con Mininet. Nel primo scenario, viene simulato un traffico regolare con flussi TCP e UDP "buoni", per osservare il comportamento normale della rete, in cui le risorse sono condivise correttamente e non si verifica congestione.

Nel secondo scenario, invece, viene introdotto un attacco UDP di tipo DoS (**Denial of Service**), mentre è ancora presente traffico TCP legittimo. In questo caso si osserva come l'attacco UDP saturi la rete, causando un degrado delle prestazioni anche per il traffico TCP, in particolare in termini di riduzione della banda disponibile (**bandwidth**) e aumento della perdita di pacchetti (**packet loss**). Un attacco Denial of Service (DoS) mira a sovraccaricare un sistema o una rete con un flusso massivo di richieste. Questo eccesso di traffico esaurisce risorse quali CPU, memoria o banda, impedendo agli utenti legittimi di accedere ai servizi. L'obiettivo è rendere il servizio non disponibile o così lento da risultare inutilizzabile.

Lo script Python **topology.py** definisce una topologia di rete virtuale utilizzando Mininet. Viene creata una classe chiamata Environment che, al momento dell'inizializzazione, imposta una rete Mininet specificando che verrà usato un controller remoto (Ryu), che viene avviato con il nome c1. All'interno della rete vengono aggiunti tre host (h1, h2, h3), ognuno con indirizzi IP e MAC specifici e quattro switch (s1, s2, s3, s4). Successivamente vengono creati i collegamenti (link) tra gli host e gli switch, e tra gli switch stessi, assegnando a ciascun collegamento una larghezza di banda (bandwidth) e un ritardo (delay). Alla fine, la rete viene costruita e avviata. Infine, viene avviata l'interfaccia CLI di Mininet per poter interagire manualmente con la rete simulata.

Lo script Python **controller.py** rappresenta un'app sviluppata per il controller SDN Ryu, al fine di gestire in modo intelligente e automatizzato una rete **software-defined** (**SDN**) realizzata tramite Mininet. Il controller, compatibile con **OpenFlow** 1.3 (protocollo di comunicazione che consente al controller SDN di interagire e gestire direttamente il comportamento degli switch di rete), implementa sia un comportamento da switch Ethernet di tipo *learning switch*, sia un sistema di monitoraggio e gestione della

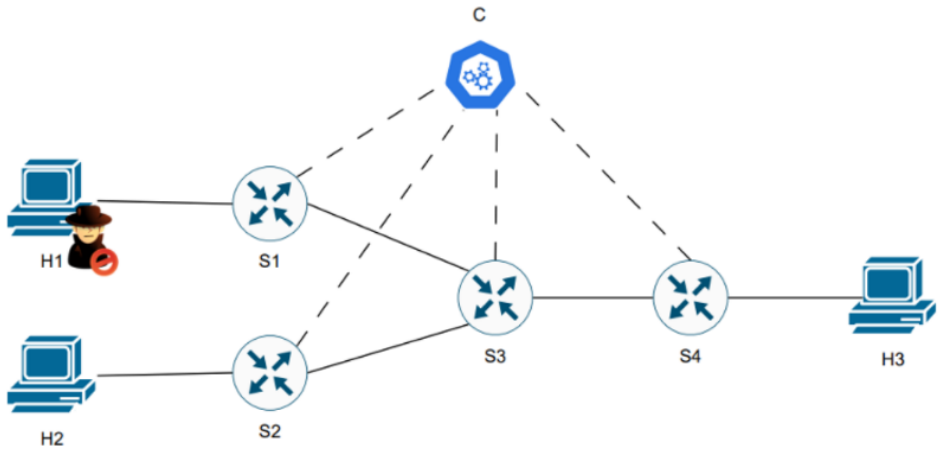


Figura 1: La topologia da cui siamo partiti e che abbiamo utilizzato.

congestione del traffico di rete.

Nella parte di commutazione, il controller riceve i pacchetti dagli switch quando non ci sono ancora regole installate (evento PacketIn), estrae le informazioni dai pacchetti come gli indirizzi MAC sorgente e destinazione, e aggiorna dinamicamente una mappa che associa i MAC alle porte in ingresso (come una **bridging table**). Quando conosce la porta di uscita per un determinato destinatario, il controller genera una *flow rule* per inoltrare i successivi pacchetti simili direttamente nello switch, evitando di dover intervenire ogni volta, riducendo la latenza e migliorando le prestazioni. Se la destinazione non è ancora nota, il pacchetto viene inviato in *broadcast*.

Oltre alla logica di commutazione, il controller implementa una funzionalità di monitoraggio attivo, mediante la quale ogni 10 secondi viene inviata una richiesta statistica (*PortStatsRequest*) a ciascuno switch della rete. Al ricevimento della risposta (*PortStatsReply*), il controller calcola le statistiche di traffico delle porte, come il numero di pacchetti ricevuti e trasmessi, i byte ricevuti e trasmessi al secondo, e il numero di errori. Queste informazioni vengono salvate in una struttura dati interna che consente di confrontare i dati nel tempo e rilevare eventuali anomalie.

Si implementa un **meccanismo di allarme** basato su **soglie**: se la banda in ingresso o uscita su una porta supera un certo valore predefinito (in questo caso 700.000 byte/s), il controller incrementa un contatore associato a quella porta. Se la soglia viene superata

per tre cicli consecutivi (quindi per circa 30 secondi), viene generato un allarme e il controller blocca automaticamente il traffico in ingresso su quella porta tramite una regola di flow che impedisce di inoltrare i pacchetti, agendo quindi in ottica di mitigazione dell'attacco o della congestione. Il blocco viene mantenuto fino a quando i livelli di traffico tornano sotto soglia per almeno due cicli consecutivi (ovvero il contatore scende da 3 a 2, e poi da 2 ad 1), momento in cui il controller rimuove la regola di blocco e ripristina il flusso del traffico. Questo approccio consente di rilevare situazioni di anomalia come attacchi DoS o congestionamenti anomali, intervenendo tempestivamente ed automaticamente. L'intero comportamento è accompagnato da messaggi colorati (rosso o verde) a terminale che evidenziano l'attivazione o la disattivazione degli allarmi, facilitando il monitoraggio visivo da parte dell'operatore di rete.

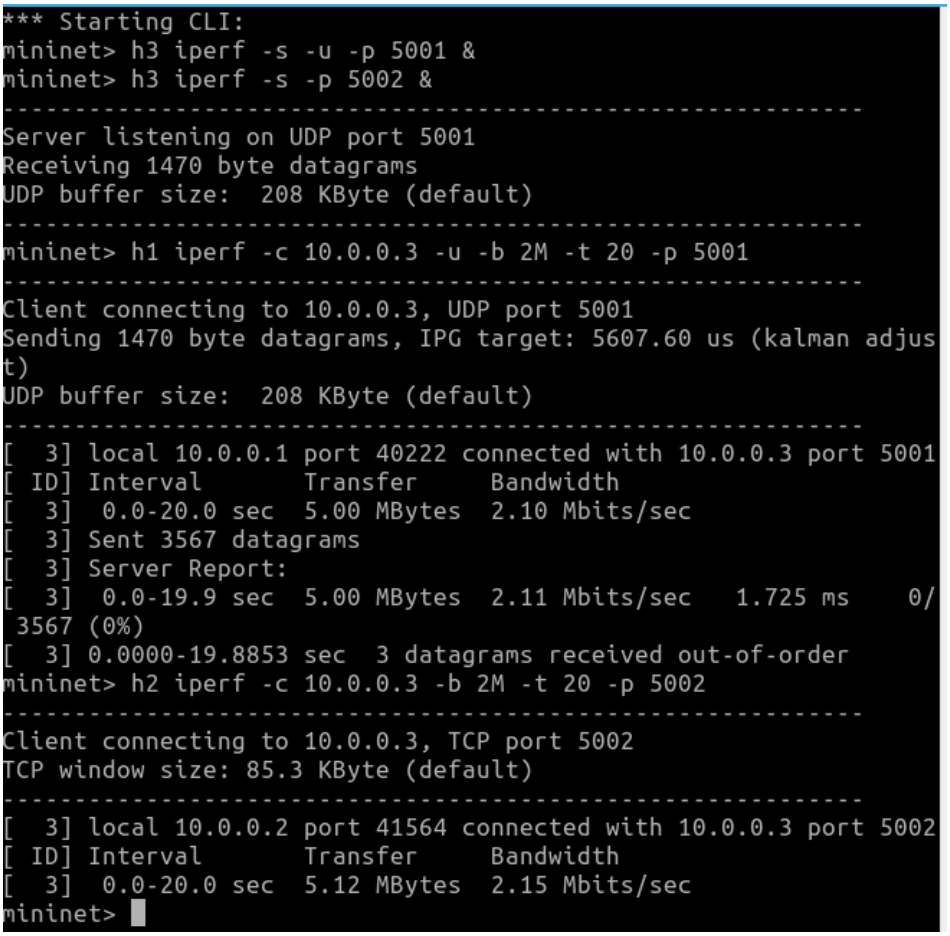


Figura 2: Con i primi due comandi visibili in foto, dopo aver attivato la Mininet e inserito topology.py come topologia di rete, poniamo h3 in ascolto di traffico UDP (-u) sul porto 5001 e di traffico TCP sul porto 5002. Poi, iniziamo l'invio di traffico UDP da h1 ad h3 (10.0.0.3 è l'indirizzo IP di h3), con -b specifichiamo la bandwidth di 2 Megabit/s, con -t il tempo di 20 secondi e con -p il porto su cui inviamo, coerentemente con quello su cui il destinatario si è messo in ascolto, ed infine diamo inizio all'invio di traffico TCP da h2 ad h3 con la stessa bandwidth, stesso tempo e l'altro porto.

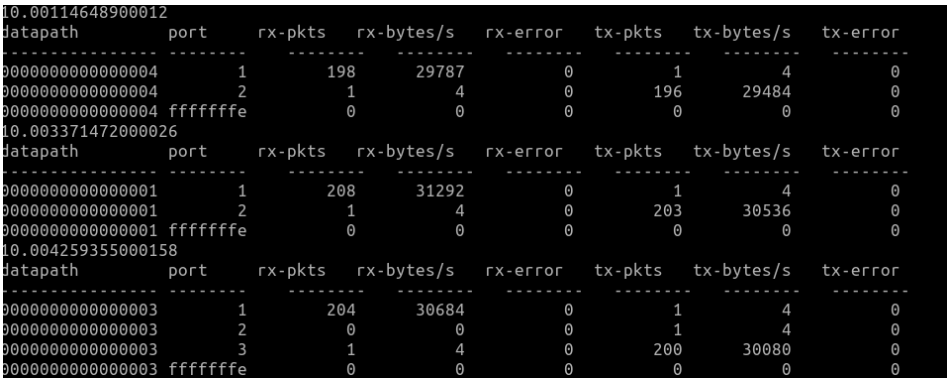


Figura 3: Questo è il terminale ove abbiamo eseguito ryu-manager controller.py, vediamo la ricezione del traffico.

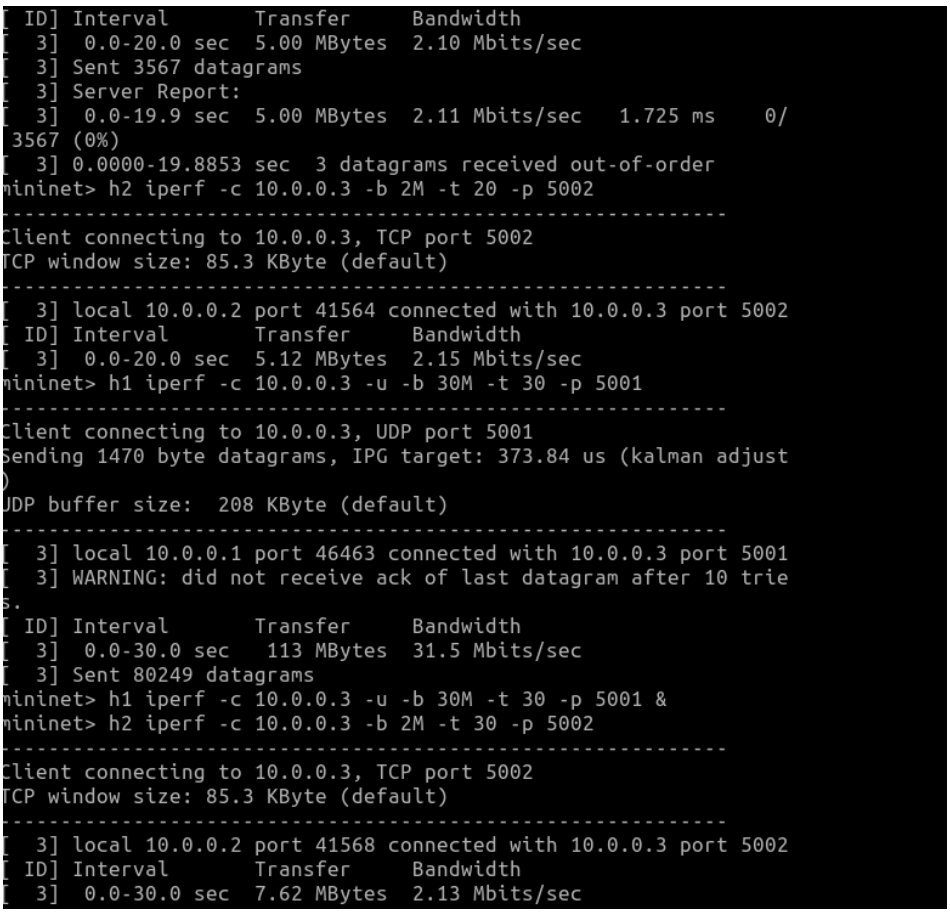


Figura 4: Simulazione dell'attacco DOS da parte di h1 (-b 30M satura la rete), dal punto di vista della Mininet. Vediamo come non viene ricevuto l'ACK finale per il traffico UDP da parte di h3, dato che parte l'allarme a causa del superamento della soglia stabilita in controller.py.

Figura 5: Allarme (in rosso) che indica il superamento della soglia nel caso in cui si verifica l'attacco DOS. Caso con il solo traffico UDP "malevolo".

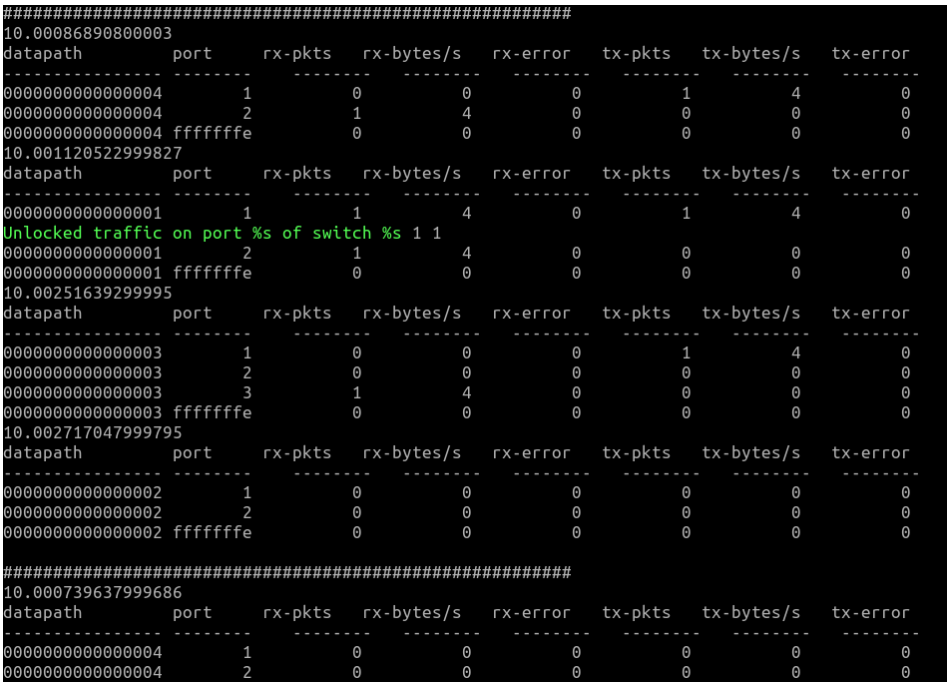


Figura 6: Sblocco dell'inoltro verso h3 dopo il superamento della soglia e il conseguente blocco. Caso con il solo traffico UDP "malevolo".

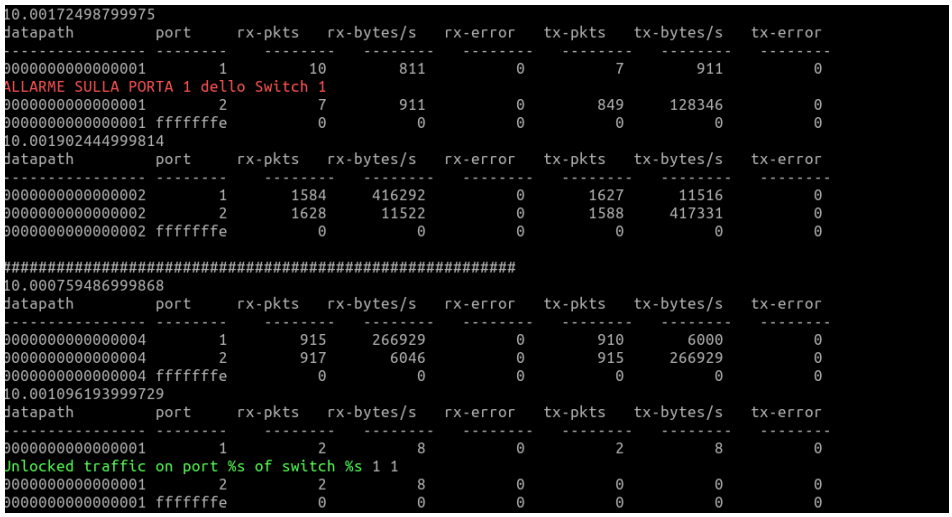


Figura 7: Allarme (in rosso) che indica il superamento della soglia nel caso in cui si verifica l'attacco DOS. Sblocco (in verde) dopo l'avvenimento. Caso con sia traffico TCP "buono" che UDP "malevolo".

2 Fix 1 (facoltativo): Soglia dinamica anziché statica

Nel progetto originario, il rilevamento di traffico anomalo era basato su una **soglia statica** fissata a 700 000 byte al secondo. Tuttavia, questa soluzione **non si adattava dinamicamente alle variazioni del traffico legittimo**, esponendo il sistema al rischio di falsi positivi o negativi.

Per superare tali limitazioni, è stata introdotta una **soglia dinamica** che sfrutta tecniche statistiche per l'analisi del traffico. In particolare, il nuovo approccio prevede il mantenimento di una **finestra mobile** (`byte_rate_history`) contenente gli ultimi 6 (`window_size`) valori osservati del tasso di traffico (espresso in byte per secondo) per ciascuna porta di ogni switch. Una volta che la finestra è completamente popolata — e quindi dopo 60 secondi, considerando un intervallo di campionamento di 10 secondi — viene calcolata la soglia dinamica secondo la formula:

$$\text{soglia\_dinamica} = \mu + k \cdot \sigma,$$

dove μ rappresenta la media dei valori presenti nella finestra, σ la relativa deviazione standard, e k è un parametro di tolleranza che nel nostro caso è stato fissato a 2 per garantire maggiore robustezza. Tale formula si utilizza in quanto consente di distinguere un comportamento normale dalla presenza di picchi (infatti, in una distribuzione gaussiana, ad esempio, circa il 95 % dei valori ricade nell'intorno di $\pm\ 2\ \cdot\sigma$ dalla media, ed un valore fuori da tale intervallo è quindi decisamente lontano dalla media, rappresentando un picco). Prima che la finestra raggiunga la dimensione prevista, il sistema continua ad operare con la soglia statica preesistente.

Il funzionamento dell'algoritmo prevede che se il traffico osservato su una porta supera la soglia dinamica per tre cicli consecutivi (ciò è verificato ad ogni ricezione di statistiche, ovvero ogni 10 secondi, e il contatore `alc` deve quindi raggiungere il valore 3), venga attivato un allarme che comporta il blocco temporaneo del traffico su quella porta (in seguito il blocco riguarderà i flussi e non più le porte). Tale blocco è realizzato attraverso l'inserimento di una regola di flusso a priorità elevata, che impedisce il forwarding dei pacchetti in ingresso. Quando il traffico rientra al di sotto della soglia, l'allarme viene progressivamente disattivato (quando il contatore `alc` scende ad 1) e la porta sbloccata, seguendo un meccanismo che evita cambiamenti troppo frequenti di

stato.

Il sistema è stato testato in tre scenari distinti. Nel **primo caso** è stato generato un traffico UDP continuativo legittimo, a bassa intensità (2 Mbps), su una singola porta, ed un analogo traffico TCP legittimo a 2 Mbps su un'altra porta, e la soglia dinamica, correttamente, non è stata superata neanche dopo due minuti. Nel **secondo caso**, è stato generato solo traffico UDP simile al primo caso ma ad alta intensità (30 Mbps) su una singola porta. La soglia dinamica, in questo scenario, è stata correttamente superata dopo 60 secondi, attivando l'allarme e bloccando la porta come previsto. Nel **terzo scenario**, invece, è stato simulato un traffico misto UDP e TCP, distribuito su due porte, di cui quello UDP intenso (30 Mbps, considerabile come un possibile attacco DoS) e quello TCP legittimo (2 Mbps). In questo caso, la soglia è stata superata e il sistema ha attivato il blocco sulla porta 1 dello switch s1 (che riceve il traffico intenso dall'host h1), disattivato dopo circa 20 secondi, dimostrando una buona capacità di distinguere tra traffico anomalo e traffico legittimo. L'introduzione della soglia dinamica ha conferito al sistema una maggiore flessibilità e adattabilità.

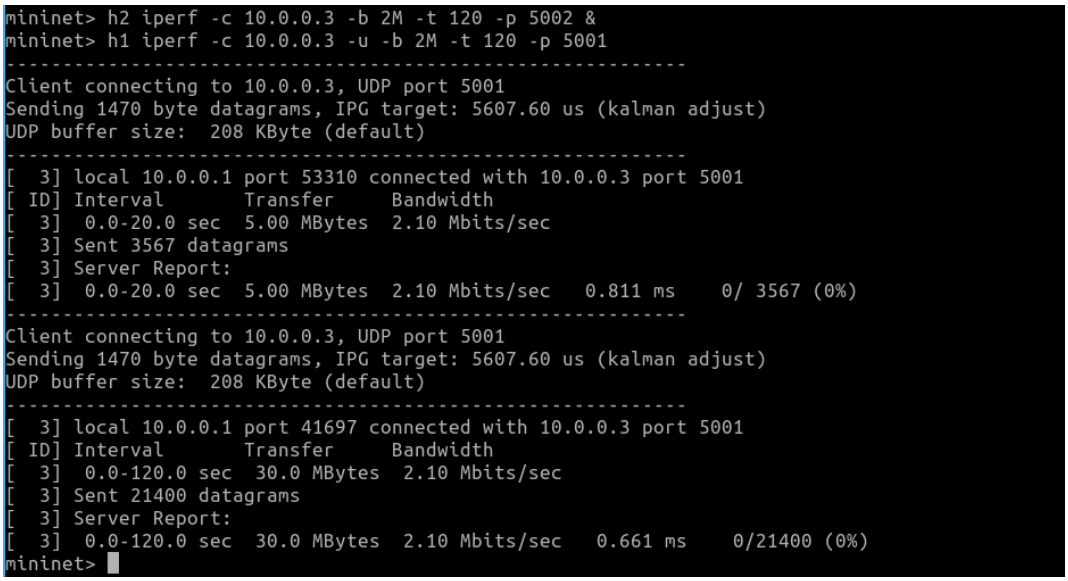


Figura 8: (Primo caso) Caso in cui abbiamo inviato traffico UDP e traffico TCP entrambi "benevoli" (2 Mbps), rispettivamente da h1 e da h2, entrambi verso h3.

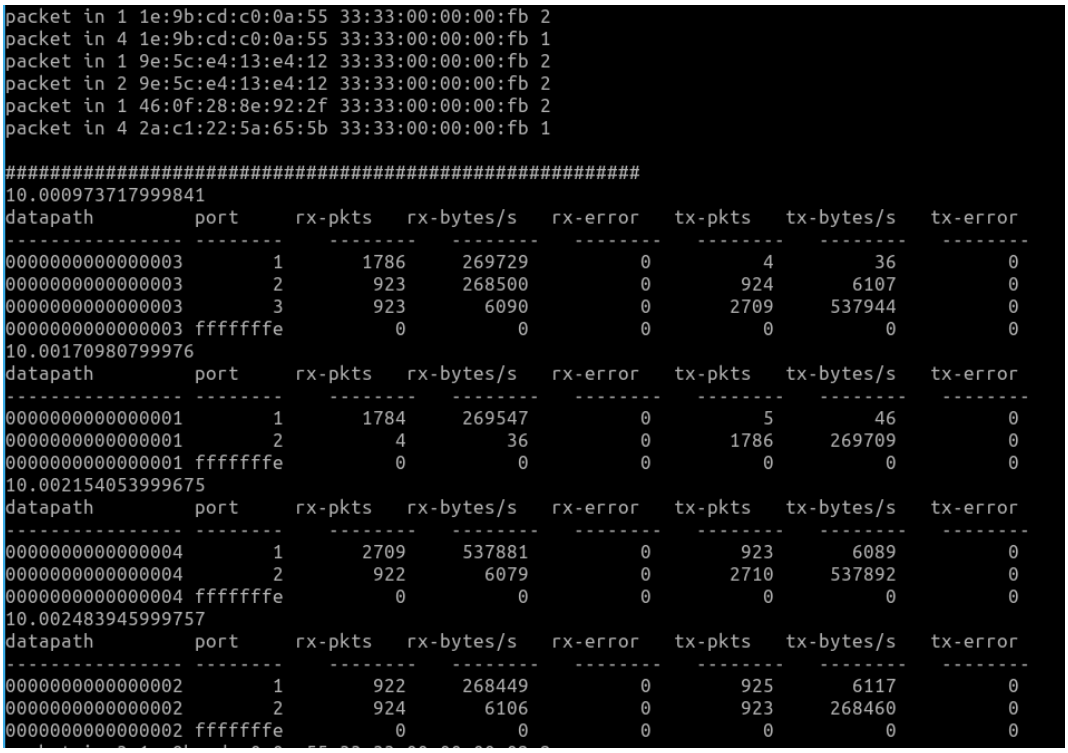


Figura 9: (Primo caso) Caso in cui abbiamo inviato traffico UDP e traffico TCP entrambi "benevoli" (2 Mbps), rispettivamente da h1 e da h2, entrambi verso h3: la soglia non viene superata.

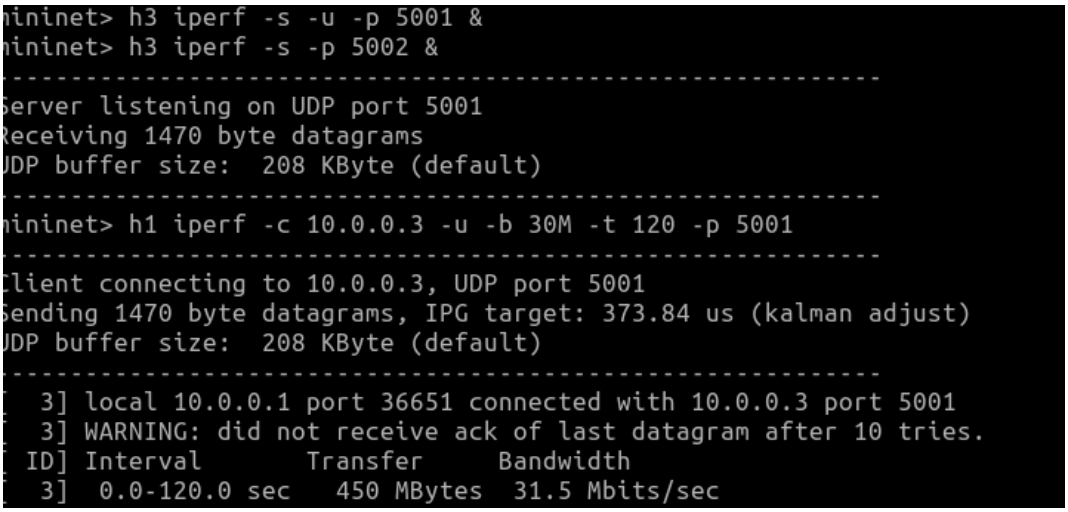


Figura 10: (Secondo caso) Caso in cui abbiamo inviato traffico UDP "malevolo" (30 Mbps) da h1 ad h3: svariati ACK non sono stati ricevuti da h1 (la soglia è stata superata).

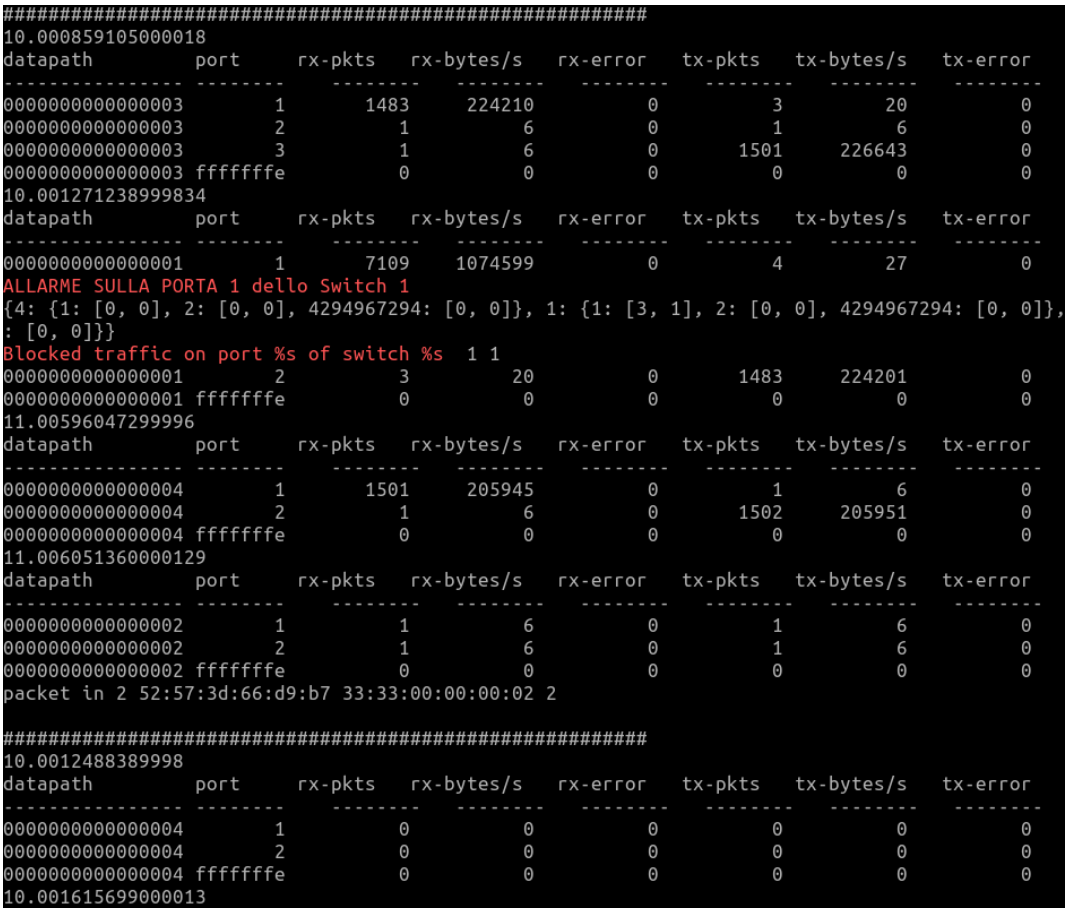


Figura 11: (Secondo caso) Caso in cui abbiamo inviato traffico UDP ”malevolo” (30 Mbps) da h1 ad h3: la soglia viene superata (il blocco si verifica sullo switch s1).

3 Fix 2 (facoltativo): Inflexible Blocking/Unblocking Policy

L’implementazione del nuovo meccanismo di blocco/sblocco si basa sull’introduzione di **due soglie distinte**. Nel modulo di policy (`_policy_engine`), dopo aver calcolato il byte-rate istantaneo e la soglia di blocco (statica o adattiva), il codice definisce una seconda soglia di rilascio pari al 70 % di quella di blocco. In pratica, subito dopo la determinazione di `thr` (il valore soglia usato per innescare l’allarme), viene calcolato:

$$\texttt{release\_thr} = 0.7 \times \texttt{thr}.$$

Questa operazione garantisce che il sistema attenda una riduzione significativa del traffico — al di sotto del 70 % — prima di rimuovere la regola di drop.

Nel ciclo di valutazione del port-rate, la logica di decisione è stata modificata in modo da incrementare il contatore `alc` (*alarm counter*) non appena il rate eccede `thr`, e da decrementare il contatore (che resta però sempre non negativo, quindi non può scendere sotto lo 0) quando il rate scende al di sotto di `release_thr`. Ciò si integra poi con la logica del blocco solo nel caso in cui il contatore raggiunga 3 e non sia ancora avvenuto il blocco e dello sblocco solo nel caso in cui il contatore scenda ad 1 e non sia ancora avvenuto lo sblocco, già discussa nel paragrafo sul *Fix 1*. Tale isteresi evita oscillazioni troppo rapide: un picco transitorio non scatena immediatamente il blocco, e analogamente il ritorno sotto soglia non riapre la porta fino al raggiungimento del livello di sicurezza prefissato. In questo modo il comportamento risulta più stabile, evitando di bloccare legittimi utilizzatori per tempi eccessivi o di consentire il ritorno degli attaccanti non appena la pressione nominale diminuisce leggermente.

```
if len(hist) >= self.window_size: #Quando ho almeno 6 elementi allora calcolo la soglia dinamica
    rates = list(hist)
    m       = statistics.mean(rates)
    sdev    = statistics.stdev(rates)
    dyn_thr = m + 2 * sdev
    thr     = min(self.threshold, dyn_thr) #Poniamo la thr statica come un tetto, in modo che se
    release_thr = 0.7 * thr
else: #senno uso quella statica
    thr     = self.threshold
    release_thr = 0.7 * thr #Soglia per l'unlocking più bassa
    dyn_thr = None
```

Figura 12: In questa sezione del codice è visibile la soglia di rilascio.

4 Fix 3: Over blocking

Nel progetto iniziale, la strategia di mitigazione in risposta a traffico eccessivo si basava sul blocco indiscriminato di tutte le comunicazioni in ingresso su una porta di uno switch. Questa logica, attivata dal superamento di una soglia di throughput, comportava però un'evidente criticità: non solo il traffico malevolo veniva bloccato, ma anche il traffico lecito che transitava sulla stessa porta veniva interrotto. Tale approccio generava dunque una forma di *overblocking*, in cui venivano penalizzati utenti legittimi e servizi innocui.

Per affrontare questo problema, è stata introdotta una modifica fondamentale nel sistema: la capacità di applicare la politica di blocco a livello di singolo flusso (flow-level). In pratica, una volta che viene rilevato un traffico anomalo su una porta, non viene più bloccata tutta la porta, ma viene attivata una richiesta delle statistiche di flusso (`OFPFlowStatsRequest`) filtrate per la porta incriminata. Qui si calcola il byte rate dei flussi interessati (applicando un match L4, più specifico), successivamente si seleziona il flusso con il byte rate più elevato e si procede con il blocco selettivo di questi specifici flussi attraverso l'inserimento di regole di `flow mod` con priorità più alta e senza azioni associate (drop implicito).

```
alc, active = self.alarm_switch_port[dpid][port_no]
if total_rate > thr:
    alc = min(3, alc + 1)
elif total_rate < release_thr: #Aumenta il contatore (alc) se la soglia thr viene superata, altrimenti lo decrementa se è inferiore a release_thr, se è tra i due non fa nulla
    alc = max(0, alc - 1)

# quando alc sale a 3 e non eri già in allarme - LOCK
if alc == 3 and not active:
    active = True
    self.logger.info(RED + f"ALLARME RILEVATO: port {port_no} switch {dpid}" + RESET)
    self.policy_q.put(('lock',   {'dpid': dpid, 'in_port': port_no}))

# quando alc scende a 0 e l'allarme era attivo - UNLOCK
elif alc == 1 and active:
    active = False
    self.logger.info(GREEN + f"ALLARME RIENTRATO: port {port_no} switch {dpid}" + RESET)
    self.policy_q.put(('unlock', {'dpid': dpid, 'in_port': port_no}))

# salva stato per la prossima iterazione
self.alarm_switch_port[dpid][port_no] = [alc, active]
```

Figura 13: Chiamata dei metodi lock e unlock

In questo modo, l'intervento del controller si concentra solo sui flussi realmente sospetti, riducendo al minimo l'impatto sul traffico legittimo. Inoltre, la selezione basata su `byte_count` garantisce che vengano bloccati solo i flussi maggiormente responsabili della congestione. Questa granularità più fine nella risposta agli attacchi migliora sensibilmente l'efficacia della mitigazione, evitando blocchi ingiustificati e mantenendo attiva la connettività per i flussi benigni.

Questo approccio può essere ulteriormente potenziato in futuro con l'introduzione di meccanismi di *whitelisting* (per escludere a priori alcuni flussi considerati sicuri) e *blocklisting* (per mantenere bloccati temporaneamente flussi identificati come malevoli), ma già nella sua forma attuale costituisce un passo decisivo verso una mitigazione più intelligente e selettiva.

```
if action == 'lock':
    # request flow stats for in_port=port_no
    match = parser.OFPMatch(in_port=port_no)
    req = parser.OFPFlowStatsRequest(
        datapath   = dp,
        table_id   = ofp.OFPTT_ALL,
        out_port   = ofp.OFPP_ANY,
        out_group  = ofp.OFPG_ANY,
        match      = match
    )

    self.pending_block[dpid].add(port_no)
    dp.send_msg(req)

    self.logger.debug(
        RED + f"Requested flow stats for lock on switch {dpid} port {port_no}" + RESET
    )
```

Figura 14: Metodo lock

```python
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_block_handler(self, ev):
    dp = ev.msg.datapath
    dpid = dp.id
    parser = dp.ofproto_parser
    ofp = dp.ofproto

    #Se non è in allarme, esco
    if dpid not in self.pending_block:
        return

    #Se non ho flussi esco
    flows = ev.msg.body
    if not flows:
        return

    now = time.time()
    #Prendo solo i flussi che matchano per la porta su cui ho avuto l'allarme
    filtered_flows = [f for f in flows if f.match.get("in_port") in self.pending_block[dpid]]

    #Calcolo rate per flusso (delta byte / delta time)
    rates = []
    for f in filtered_flows:
        key = (dpid,
            f.match.get("in_port"),
            f.match.get('eth_src'),
            f.match.get('eth_dst'),
            f.match.get('ipv4_src'),
            f.match.get('ipv4_dst'),
            f.match.get('ip_proto'),
            f.match.get('tcp_src'),
            f.match.get('tcp_dst'),
            f.match.get('udp_src'),
            f.match.get('udp_dst'))

        prev_byte, prev_time = self.prev_flow_stats.get(key, (0, now))
        delta_bytes = f.byte_count - prev_byte
        delta_time = now - prev_time if now > prev_time else 1
        rate = delta_bytes / delta_time
```

Figura 15: Snippet dell'handler delle richieste di statistiche

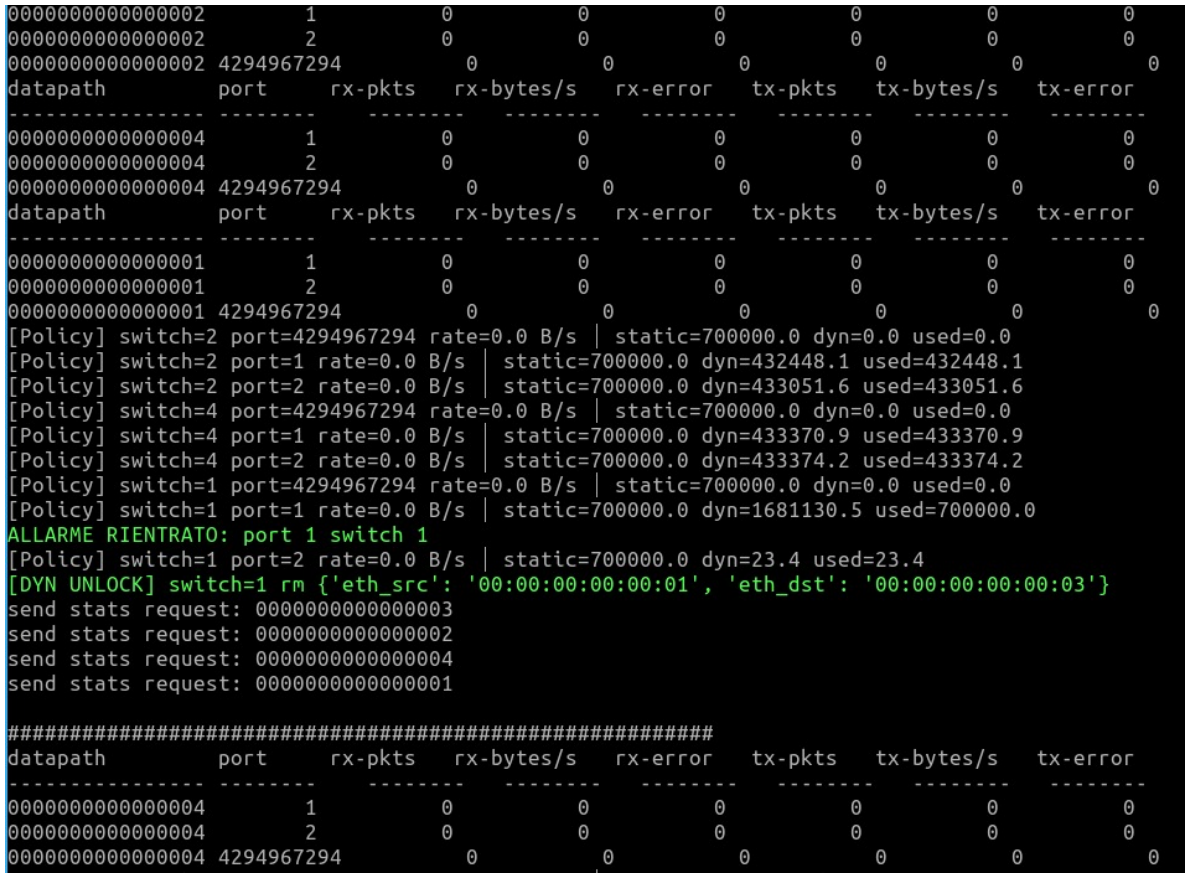


Figura 16: Esempio di blocco di un flusso da h1 ad h3, bloccato sul porto 1 dello switch s1.

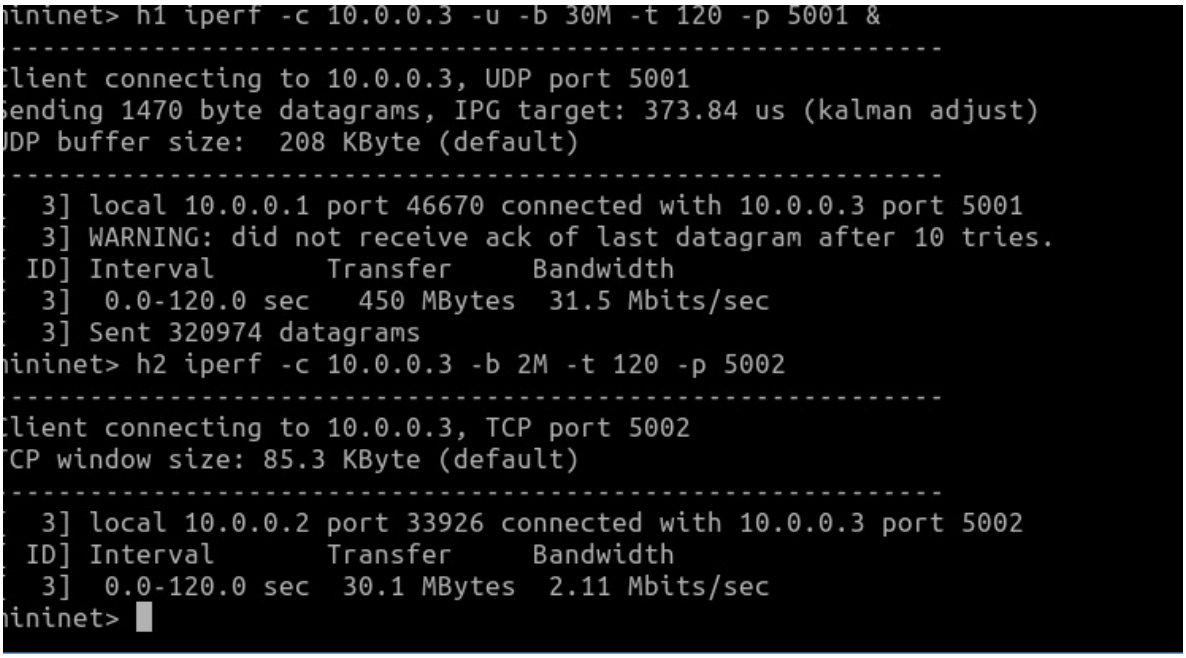


Figura 17: Esempio in cui inviamo un flusso intenso da h1 ad h3 ed un flusso ”normale” da h2 ad h3.

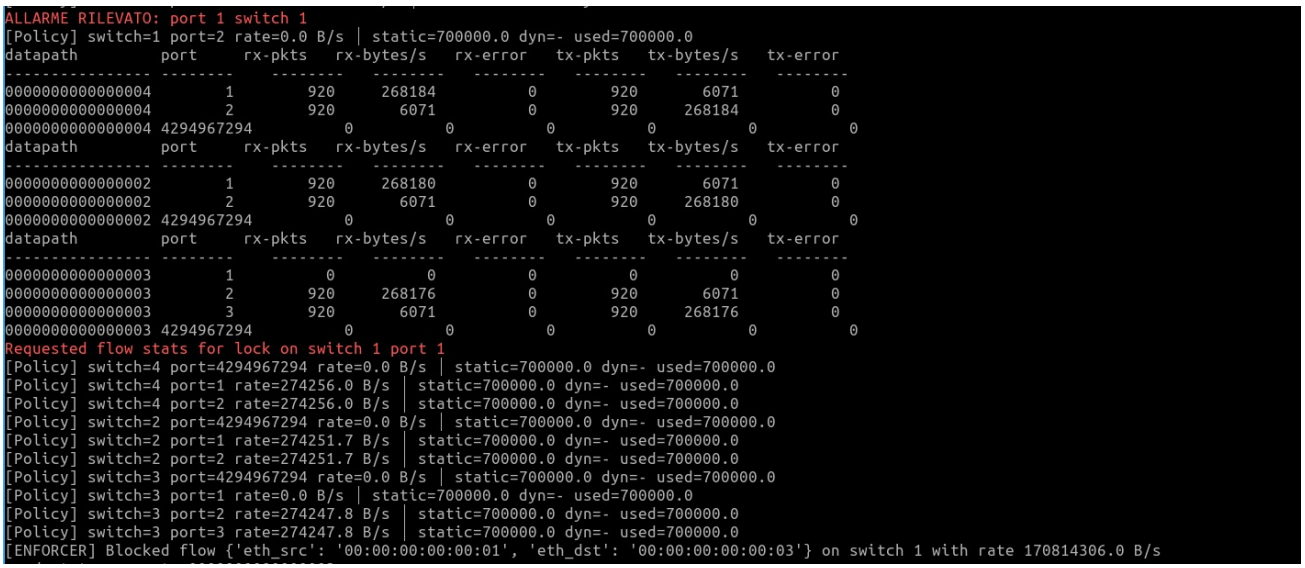


Figura 18: Richiesta di blocco sul porto 1 dello switch 1.

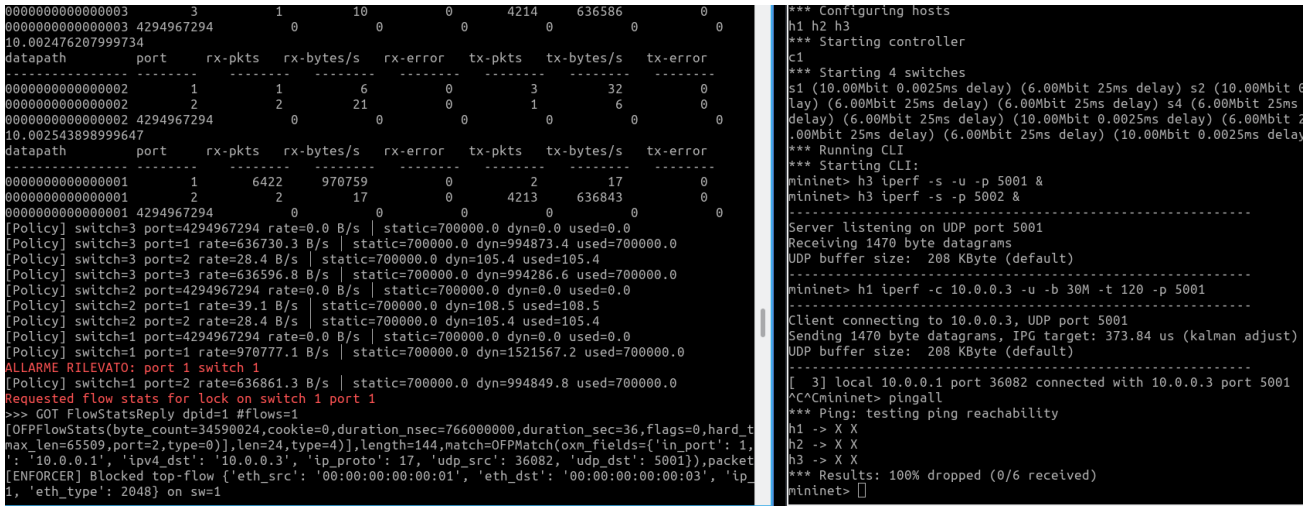
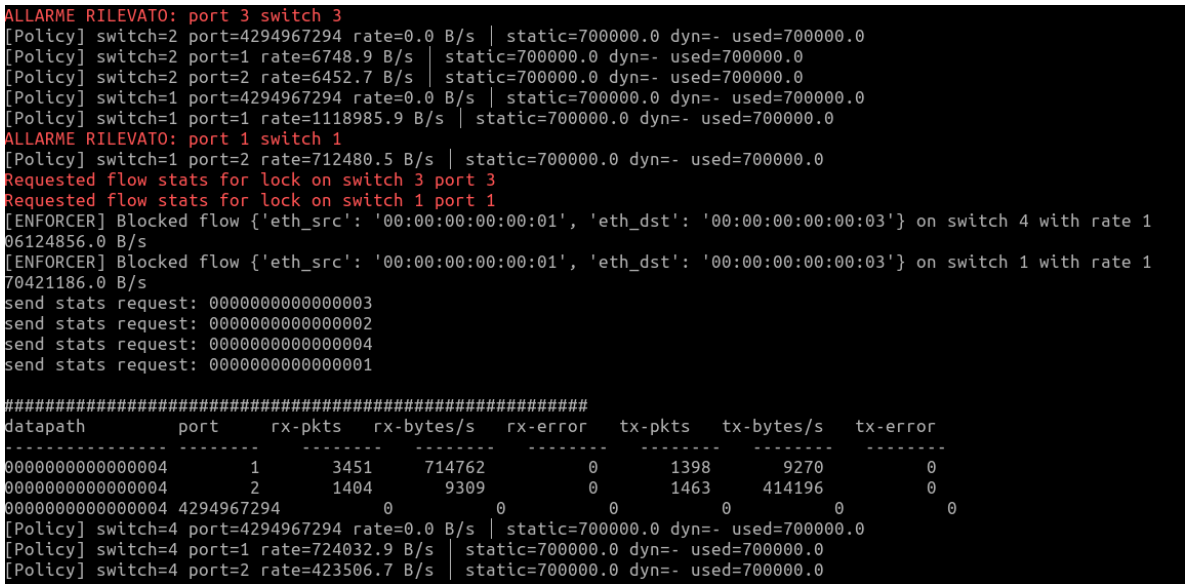


Figura 19: Esempio di blocco a livello flusso.



5 Fix 4: Modularizzazione

L'intervento al fine di svolgere questo fix ha, come richiesto, introdotto una netta separazione tra **monitoraggio**, **decisione delle politiche** e **applicazione delle regole**, trasformando il controller in tre moduli concorrenti e disaccoppiati. Il primo modulo, racchiuso nei metodi `_monitor` e `_port_stats_reply_handler`, invia periodicamente richieste di statistiche alle porte degli switch e raccoglie i dati grezzi. Questi dati vengono impacchettati in tuple `(dpid, elapsed, port_stats)` e accodati nella struttura thread-safe `stats_q`.

```
def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        self.logger.info('\n############################################################')
        hub.sleep(timeInterval) #Richiede le statistiche ogni timeInterval = 10 sec
```

Il secondo modulo (decisione delle politiche), implementato in `_policy_engine`, estrae da `stats_q` le informazioni di traffico e calcola per ogni porta il byte-rate istantaneo. Una volta popolata una finestra mobile di lunghezza prefissata, il modulo stima una soglia adattiva basata su media e deviazione standard, quindi decide se emettere comandi di `lock` o di `unlock`. Tali comandi, codificati come triplette `('lock'/'unlock', dpid, port_no)`, vengono inseriti in un'altra coda thread-safe chiamata `policy_q`.

Il terzo modulo (applicazione delle politiche), racchiuso in `_flow_enforcer`, consuma comandi da `policy_q` e, a seconda dell'azione richiesta, invia `OFPFlowStatsRequest` per identificare i flussi da bloccare o `OFPFlowMod` per cancellare le regole esistenti. Il montaggio dei tre thread avviene nella routine di inizializzazione, dove ciascuno viene avviato con `hub.spawn(...)`, garantendo cicli di esecuzione indipendenti e sincronizzazione esclusivamente tramite le due code.

```python
def _flow_enforcer(self):
    while True:
        try:
            action, match_dict = self.policy_q.get(timeout=1)
            port_no = match_dict.get('in_port')
            dpid = match_dict.pop("dpid", None)

            if dpid is None or port_no is None:
                self.logger.warning(f"Invalid enforcer request, missing dpid or in_port: {match_dict}")
                continue

        except QueueEmpty:
            continue

        dp = self.datapaths.get(dpid)
        if not dp:
            continue  # switch down

        ofp, parser = dp.ofproto, dp.ofproto_parser

        if dpid not in self.pending_block:
            self.pending_block[dpid] = set()

        if dpid not in self.dyn_blocks:
            self.dyn_blocks[dpid] = []

        if action == 'lock':
            # request flow stats for in_port=port_no
            match = parser.OFPMatch(in_port=port_no)
            req = parser.OFPFlowStatsRequest(
                datapath   = dp,
                table_id   = ofp.OFPTT_ALL,
                out_port   = ofp.OFPP_ANY,
                out_group  = ofp.OFPG_ANY,
                match      = match
            )

            self.pending_block[dpid].add(port_no)
```

```
elif action == 'unlock':
    # recupera tutti i match_fields bloccati per questo dpid
    for match_fields in self.dyn_blocks.get(dpid, []):
        # aggiungi prereq come per l'add (eth_type/ip_proto) se serve
        try:
            match = parser.OFPMatch(**match_fields)
        except:
            continue
        fm = parser.OFPFlowMod(
            datapath   = dp,
            table_id   = 0,
            match      = match,
            command    = ofp.OFPFC_DELETE,
            out_port   = ofp.OFPP_ANY,
            out_group  = ofp.OFPG_ANY
        )
        dp.send_msg(fm)
        self.logger.info(
            GREEN + f"[DYN UNLOCK] switch={dpid} rm {match_fields}" + RESET
        )

    # pulisci la lista dei blocchi dinamici
    self.dyn_blocks[dpid].clear()

    self.pending_block[dpid].clear()
```

Questa architettura modulare riduce significativamente l'accoppiamento tra le componenti, aumenta la coesione interna di ciascun modulo e semplifica l'estensibilità futura: per aggiungere nuove logiche di analisi o enforcement basta intervenire sul solo thread di interesse, mantenendo invariati gli altri. Anche il debug e la gestione dei log risultano più semplici, poiché ogni flusso di esecuzione produce messaggi con fini chiari e indipendenti.

6 Fix 5: Controller-Centric Blocking Decisions

Controller e applicazioni esterne ora dialogano attraverso una struttura dati condivisa, `external_blocklist`, protetta da un lock. Inizialmente questo dizionario mappa ciascun `dpid` (ovvero il datapath ID) a una lista di match-field definiti da amministratori o altri moduli, anziché far decidere esclusivamente al thread interno di policy. La funzione `_watch_blocklist_file` scandisce periodicamente un file JSON esterno (`blocklist.json`), carica gli ingressi aggiornati e, sotto protezione del `blocklist_lock`, rimpiazza l'intero contenuto di `external_blocklist`. In questo modo ogni update addizionale — sia generato da un tool CLI dedicato sia da un'app remota — viene importato dinamicamente nel controller, senza riavvii né modifiche al codice sorgente.

All'interno del thread di policy (`_policy_engine`), subito dopo la logica di soglia automatica, si introduce un blocco di sincronizzazione:

```
with self.blocklist_lock:
    all_dpids = set(self.external_blocklist.keys())
                | set(self._installed_ext.keys())
    ...
```

Qui si calcola l'insieme degli switch per cui esistono regole esterne o già installate, garantendo che non vengano mai ignorate rimozioni o aggiunte consecutive. Per ogni `dpid_ext` il codice confronta la lista appena letta (`mlist`) con quella già applicata (`installed_list`), ricavando due insiemi: `new` contenente i match da aggiungere, e `removed` per quelli da eliminare. Ciascun elemento `match_dict` in `new` o `removed` viene quindi arricchito con la chiave `"dpid"` e accodato in `policy_q` come azione `'lock_ext'` o `'unlock_ext'`. Infine `_installed_ext[dpid_ext]` viene aggiornato in copia per riflettere lo stato corrente, evitando duplicati nei cicli successivi.

Nel modulo di enforcement (`_flow_enforcer`), i casi `'lock_ext'` e `'unlock_ext'` vengono gestiti in sequenza. Alla ricezione di `'lock_ext', match_dict`, si aggiungono automaticamente i prerequisiti di livello Ethernet/IP/TCP/UDP se il dizionario contiene campi di layer superiore:

```
if any(k in match_dict for k in ('ipv4_src',...)):
    match_dict.setdefault('eth_type',0x0800)
...
```

Questo passaggio costruisce uno `OFPMatch` valido e specifico, evitando eccezioni. Quindi si invia un `OFPFlowMod` con priorità elevata (30) e istruzioni vuote (drop), determinando l'effettivo blocco all'ingresso del flusso. Il comando di sblocco (`'unlock_ext'`)

segue analoga procedura: si ricostruisce `OFPMatch` dallo stesso dizionario e si invia un `OFPFC_DELETE`, rimuovendo la regola di drop.

Questa architettura introduce piena estendibilità, poiché in tal modo agenti esterni non necessitano di conoscere la logica interna di monitoraggio o soglia; bastano semplici comandi di inserimento/rimozione in un JSON perché il controller recepisca e applichi le policy. Il `blocklist_lock` e l'uso coordinato di `_installed_ext` garantiscono coerenza tra ciò che l'admin chiede e quanto effettivamente installato sullo switch, mentre la separazione in code (`policy_q`) mantiene disaccoppiata la parte di decisione dall'effettivo invio dei FlowMod.

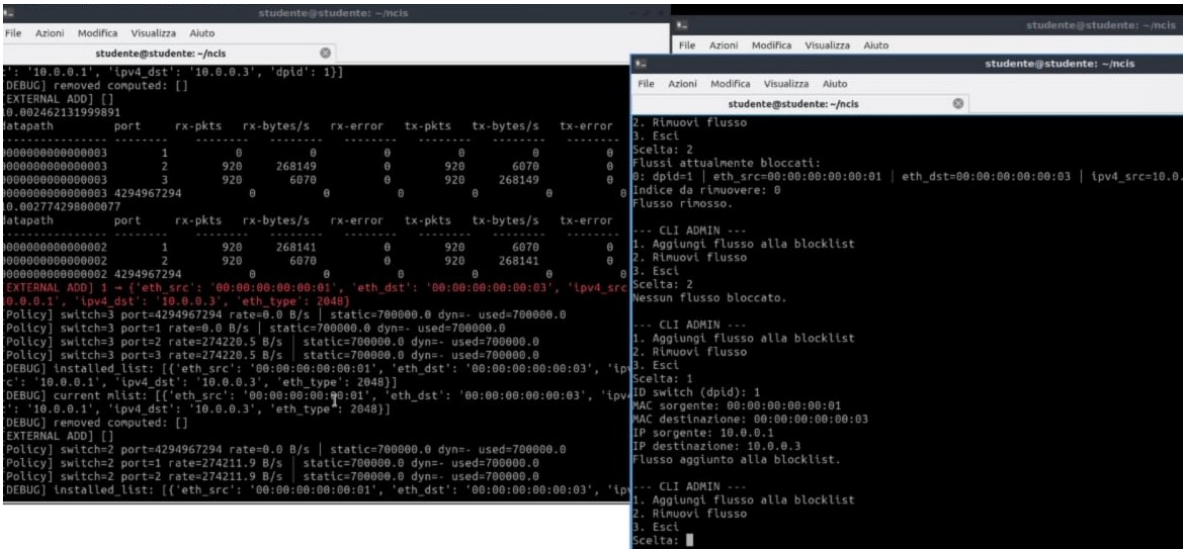


Figura 20: Blocco manuale da parte dell'utente del flusso da h1 ad h3 e visualizzazione (nelle print di debug) della lista dei blocchi esistenti.

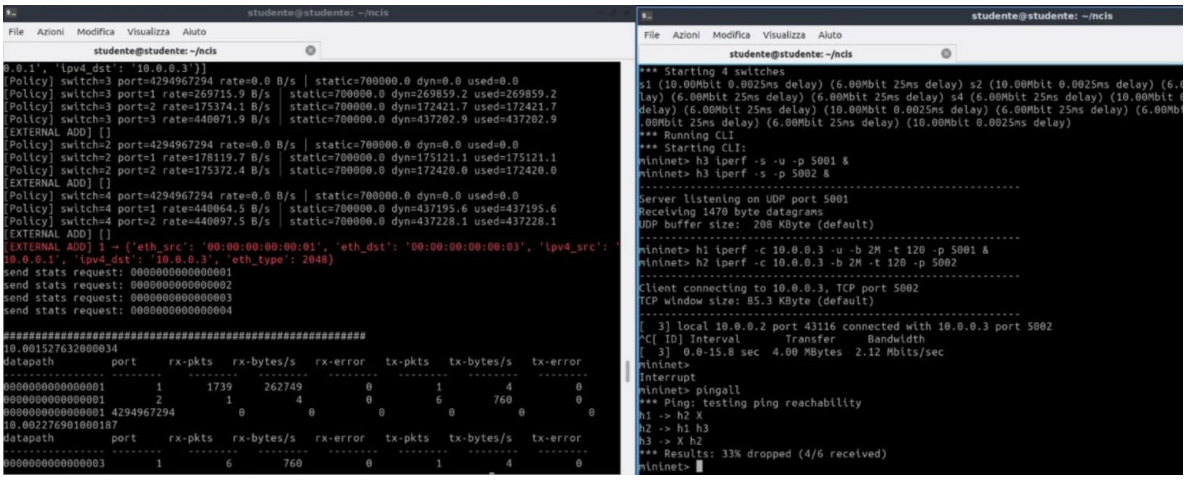


Figura 21: In questa immagine è visibile l'esecuzione del comando di ping, per controllare la raggiungibilità tra tutte le possibili coppie di host, che h1 non può raggiungere h3 nel momento in cui si impone da utente (e quindi viene inserita la regola nel JSON) il blocco del traffico da h1 ad h3.

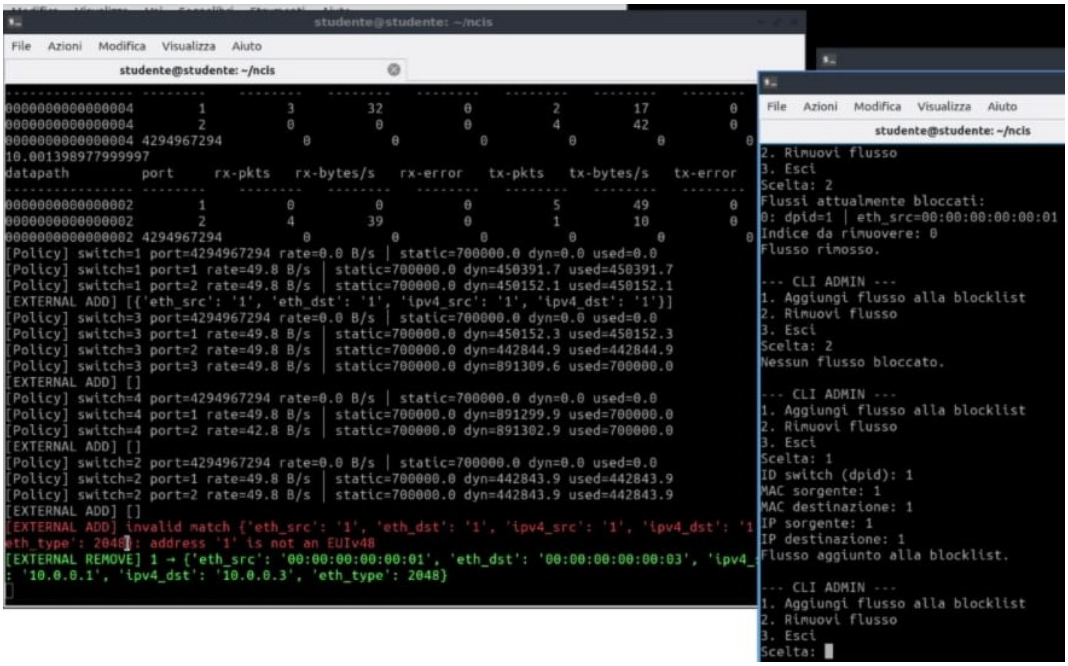


Figura 22: Sblocco di un flusso in seguito alla rimozione manuale, da parte dell'utente della regola relativa al blocco di quel flusso dal file JSON interagendo con l'interfaccia di `cli_admin.py` (file che abbiamo creato per testare questo fix), (c'è anche, subito sopra, il tentativo di blocco di un flusso che aveva parametri scorretti (il che viene correttamente segnalato).

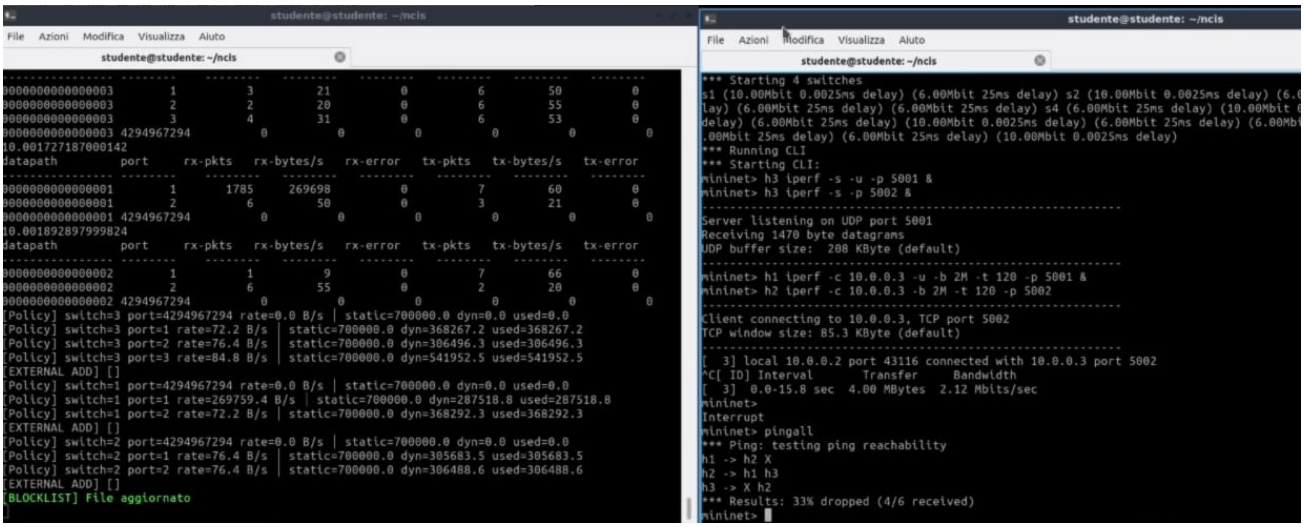


Figura 23: Aggiornamento del file JSON contenente i blocchi da esterno attualmente presenti.

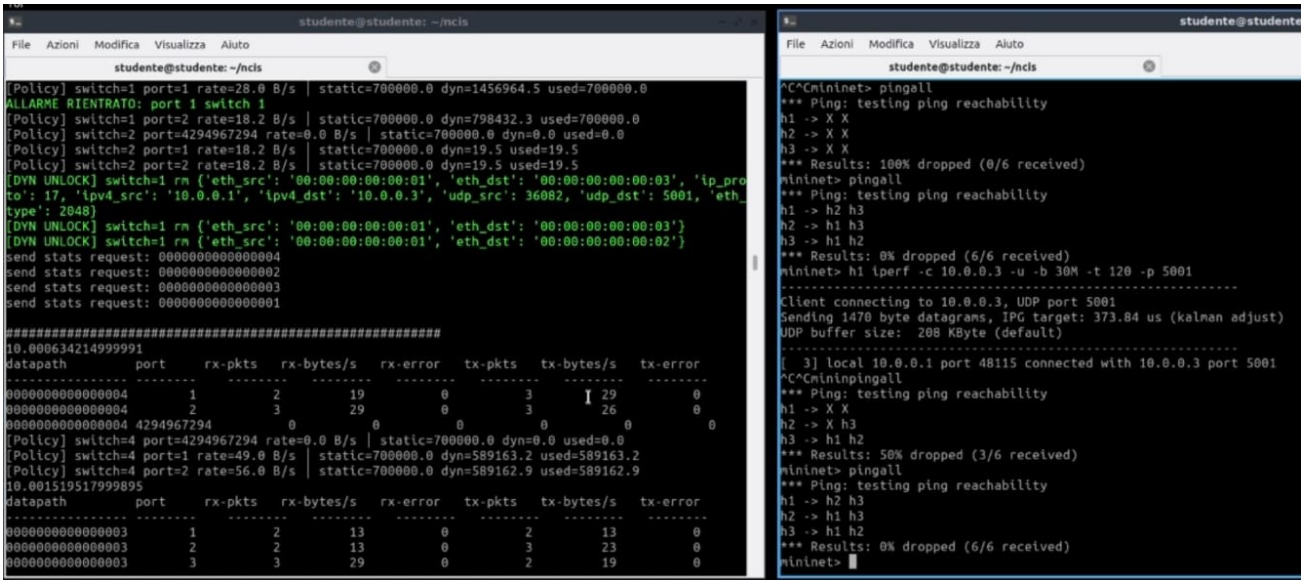


Figura 24: Corretto sblocco e (come visibile dal ping) conseguente ripristino della raggiungibilità tra tutti gli host.

```python
def _watch_blocklist_file(self):
    last_mtime = 0
    while True:
        try:
            if os.path.exists("blocklist.json"):
                mtime = os.path.getmtime("blocklist.json")
                if mtime != last_mtime:
                    with open("blocklist.json", "r") as f: #apro il json in lettura
                        data = json.load(f) #salvo i dati
                        with self.blocklist_lock:
                            self.external_blocklist.clear()
                            for item in data.get("blocked_flows", []):
                                dpid = item["dpid"]
                                match_fields = item["match"]
                                self.external_blocklist.setdefault(dpid, []).append(match_fields)
                            self.logger.info(GREEN + f"[BLOCKLIST] File aggiornato" + RESET )
                    last_mtime = mtime

        except Exception as e:
            self.logger.error(RED + f"[BLOCKLIST] Errore: {e}" + RESET)
        hub.sleep(3)  # controlla ogni 3 secondi
```

```python
#Qui è per i bloccaggi esterni
with self.blocklist_lock:
    # Prendi l'unione di tutti gli switch noti in blocklist o già installati
    all_dpids = set(self.external_blocklist.keys()) | set(self._installed_ext.keys())
    #questo per fare in modo che se non ho più nulla in blocking list comunque mi esegue l'external remove
    for dpid_ext in all_dpids:
        dp_ext = self.datapaths.get(dpid_ext)
        if not dp_ext:
            continue

        mlist = self.external_blocklist.get(dpid_ext, [])
        installed_list = self._installed_ext.get(dpid_ext, [])

        def dict_in_list(d, lst):
            for item in lst:
                if item == d:
                    return True
            return False

        new = [m for m in mlist if not dict_in_list(m, installed_list)]
        removed = [m for m in installed_list if not dict_in_list(m, mlist)]

        #self.logger.info(f"[EXTERNAL ADD] {new}")

        for match_dict in new:
            match_dict["dpid"] = dpid_ext
            self.policy_q.put(('lock_ext', match_dict))

        for match_dict in removed:
            match_dict["dpid"] = dpid_ext
            self.policy_q.put(('unlock_ext', match_dict))

        # Aggiorna lo stato installato
        self._installed_ext[dpid_ext] = mlist.copy()
```