UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Un Passo Verso l'Introspezione del Kernel Linux
A Step Towards Linux Kernel Introspection

Candidato:

Alessandro Marcolini

Relatore:

Chiar.mo Prof. Luca Spalazzi

Correlatore:

Dott. Andrea Claudi

Anno Accademico 2022-2023

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

ALESSANDRO MARCOLINI

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Un Passo Verso l'Introspezione del Kernel Linux
A Step Towards Linux Kernel Introspection

Laurea Magistrale - Ingegneria Informatica e dell'Automazione - Feb 2024

“Given enough eyeballs, all the bugs are shallow.”

– Eric S. Raymond

Sommario

Il kernel Linux, uno dei più ampi progetti open source, costituisce il cuore di milioni di dispositivi, dai server agli smartphone, rappresentando un pilastro fondamentale nelle attuali infrastrutture IT.

Il focus di questa tesi è contribuire allo sviluppo di funzionalità volte a semplificare l'interrogazione del kernel, consentendo di identificare alcuni moduli supportati. Questo obiettivo è stato raggiunto attraverso due approcci distinti: la modifica di uno scheduler di rete per la validazione di un attributo tramite Netlink, un protocollo di comunicazione tra user-space e kernel-space, anziché manualmente, e l'aggiunta del supporto alla codifica di specifici tipi di attributi per la comunicazione con il kernel tramite uno strumento a riga di comando.

Le modifiche proposte, sottoposte a un'attenta revisione da parte della community, hanno ottenuto un discreto successo, venendo integrate con successo nel kernel.

Il lavoro presentato fornisce una solida base per possibili sviluppi nell'ambito dell'introspezione del kernel.

Indice

Sommario ii

1 Introduzione 1

2 Il Sottosistema di Rete del Kernel Linux 4

2.1 Panoramica del sottosistema di rete 4

2.1.1 Lo stack di rete Linux 4

2.1.2 I componenti fondamentali 5

2.1.3 Virtualizzazione di reti 6

2.1.4 Filtro di pacchetti e firewalling 6

2.2 Configurazione della rete: iproute2 6

2.2.1 Configurazioni Ipv4 e IPv6 7

2.2.2 Controllo del traffico 8

3 Netlink 11

3.1 Implementazione 11

3.1.1 Netlink Socket Bus 13

3.1.2 Formato dei Messaggi Netlink 13

3.1.3 Validazione degli attributi 15

3.1.4 Messaggi di Errore in Netlink 17

3.2 Classic Netlink vs Generic Netlink 17

3.2.1 Generic Netlink 18

4 Processi di Revisione del Codice nello Sviluppo del Kernel 19
4.1 Le Fasi dello Sviluppo del Kernel 19
4.2 Ciclo di Vita di una Patch 21

5 Validazione dei Parametri della qdisc TAPRIO mediante Netlink 24
5.1 MQPRIO (Multiqueue Priority Qdisc) 24
5.2 TAPRIO (Time-Aware Priority Scheduler) 25
5.2.1 Parametri di una Qdisc TAPRIO 26
5.2.2 Attributi Netlink di TAPRIO 27
5.3 Primo Approccio ad una Validazione Migliore 29
5.3.1 Considerazioni sul Primo Approccio 31
5.4 Un Migliore Approccio alla Validazione 32

6 Lo Strumento a Riga di Comando ynl 37
6.1 Archittettura di ynl 37
6.2 Le Specifiche Netlink in YAML 38
6.2.1 Proprietà Aggiuntive Richieste da Classic Netlink 38
6.3 Introspezione del Kernel 42
6.4 Stato Attuale di ynl e della Specifica YAML di tc 42
6.5 Modifiche Proposte ad ynl 42
6.6 Review Iniziale alle Modifiche 46
6.7 Aggiornamento della Patch dopo la Review 48

7 Risultati 50
7.1 Test dei Cambiamenti Effettuati su TAPRIO 50
7.2 Esito della Patch Relativa a TAPRIO 51
7.3 Risultati dei Cambiamenti Effettuati su ynl 51
7.4 Esito della Patch Relativa a ynl 53

8 Conclusioni 54

Indice dei Listati

3.1 Apertura di un socket Netlink e invio di un messaggio 12
3.2 Definizione di struct nlmsghdr in <include/linux/netlink.h> 13
3.3 Definizione di nla_policy in <include/net/netlink.h> 16
3.4 Definizione di nlmsgerr in <include/uapi/linux/netlink.h> 17
3.5 Definizione di genlmsghdr in <include/uapi/linux/genetlink.h> . . 18
5.1 Definizione degli attributi Netlink di TAPRIO 28
5.2 Policy di validzione degli attributi Netlink di TAPRIO 28
5.3 Validazione manuale dell'attributo "flags" di TAPRIO 30
5.4 Definizione delle macro relative alle modalità operative di TAPRIO . . 30
5.5 La struct `nla_bitfield32` 31
5.6 La policy `NLA_POLICY_MASK` 32
5.7 Definizione della bitmask da utilizzare per la validazione 33
5.8 Modifica della policy Netlink di validazione 33
5.9 Patch finale per la qdisc TAPRIO - parte 1 34
5.10 Patch finale per la qdisc TAPRIO - parte 2 35
5.11 Patch finale per la qdisc TAPRIO - parte 3 36
6.1 Specifica YAML (parziale) di `tc` - parte 1 40
6.2 Specifica YAML (parziale) di `tc` - parte 2 41
6.3 La funzione `_add_attr` 43
6.4 La funzione `_resolve_selector` 44
6.5 Cambiamenti apportati a `_resolve_selector` 45
6.6 Cambiamenti apportati a `_add_attr` 47

6.7 Modifica della patch per una maggiore leggibilità 48
6.8 La patch finale per ynl 49

Capitolo 1

Introduzione

Con GNU/Linux, o comunemente Linux, ci riferiamo ad una famiglia di sistemi operativi unix-like che usano come kernel, il kernel Linux. Parliamo di sistemi operativi open source molto utilizzati dalle principali aziende nell'ambito dell'informatica come Google, Microsoft, Amazon, IBM, Oracle, RedHat, Canonical, le quali a loro volta, contribuiscono allo sviluppo degli stessi.

Il kernel Linux è continuamente e liberamente sviluppato da collaboratori di tutto il mondo attraverso la relativa community, con lo sviluppo che ogni giorno avviene sfruttando la relativa mailing list. Il codice sorgente di Linux è dunque disponibile a tutti ed è ampiamente personalizzabile, al punto da rendere possibile, in fase di compilazione, l'esclusione di codice non strettamente indispensabile.

In quanto "cuore" di un sistema operativo (nucleo) fornisce tutte le funzioni essenziali per il sistema, come la gestione della memoria primaria, delle risorse hardware del sistema e delle periferiche. Si occupa quindi di gestire il tempo processore, le comunicazioni e la memoria distribuendole ai processi in corso a seconda delle priorità (scheduling) realizzando così il multitasking. Supporta dunque il multitasking ed è multiutente: ciò permette che diversi utenti (con privilegi differenziati) possano eseguire sullo stesso sistema diversi processi software in simultanea.

La flessibilità di questo kernel lo rende adatto a tutte quelle tecnologie embedded emergenti e anche nei centri di calcolo distribuito fino ad essere incorporato negli smartphone (Android si basa sul kernel Linux).

Il kernel Linux ha un'architettura monolitica con un design modulare, che permette l'inserimento e la rimozione di moduli a runtime. I moduli permettono l'estensione delle funzionalità del kernel senza la necessità di riavviare il sistema.

Vista la sua complessità, il kernel è composto di diversi sottosistemi [1], che possono essere suddivisi in:

- sottosistemi core, relativi a driver, gestione della memoria, gestione dell'alimentazione, scheduling, locking
- sottosistemi di rete, che gestiscono le interfacce di rete e il networking
- sottosistemi relativi allo storage
- sottosistemi relativi a dispositivi di interfaccia umana (HID), che gestiscono dispositivi di input e output, audio e GPU
- altri sottosistemi non categorizzati

Questa tesi si concentrerà sul sottosistema di rete. Nel particolare, l'obiettivo è quello di aggiungere al kernel delle funzionalità che permettano l'interrogazione dello stesso, riguardo ai moduli compilati che esso possiede. Questo perché, finora, non è possibile sapere a priori quali moduli siano stati compilati e caricati nel kernel che si sta utilizzando.

La tesi è articolata in tre parti. Nella prima parte, verrà descritto nel particolare il sottosistema di rete e Netlink, il protocollo di comunicazione tra processi, assieme ad una breve descrizione del processo di revisione del codice nel kernel. Nella seconda parte, poi, verrà descritto il flusso di lavoro e le scelte di progettazione che hanno portato alle modifiche applicate al kernel. E infine, nella terza parte, verranno presentati i risultati assieme a dei test di collaudo del codice. Nel particolare:

Il Capitolo 2 offre una panoramica del sottosistema di rete del kernel Linux.

Il Capitolo 3 fornisce una dettagliata descrizione di Netlink, il protocollo ampiamente utilizzato nel contesto di questa tesi.

Il Capitolo 4 accenna alle peculiarità del processo di revisione del codice nel kernel.

Il Capitolo 5 descrive la qdisc TAPRIO e le modifiche apportate ad essa.

Il Capitolo 6 si focalizza sullo strumento ynl, illustrando le sue funzionalità e le modifiche apportate.

Il Capitolo 7 spiega come sono stati valutati i cambiamenti proposti nel kernel.

Il Capitolo 8 conclude il lavoro di tesi riepilogando gli argomenti trattati.

Capitolo 2

Il Sottosistema di Rete del Kernel Linux

Il sottosistema di rete è un componente cruciale del kernel che permette, tra le altre cose, la comunicazione tra dispositivi, la facilitazione del trasferimento dei dati e la gestione delle reti attraverso strumenti avanzati di configurazione.

2.1 Panoramica del sottosistema di rete

2.1.1 Lo stack di rete Linux

Secondo il modello OSI (Open Systems Interconnection), i sette livelli logici di rete sono:

- L1 - fisico: gestisce i segnali elettrici e i dettagli di basso livello.
- L2 - collegamento dati: gestisce il trasferimento dati. Il tipo più comune di collegamento dati è Ethernet. In questo livello, troviamo i driver di rete Ethernet di Linux.
- L3 - rete: è responsabile dell'inoltro dei pacchetti e della traduzione degli indirizzi di rete. I livelli di rete più comuni all'interno del kernel sono IPv4 e IPv6.

- L4 - trasporto: gestisce l'invio dei dati tra nodi. I protocolli più conosciuti che operano a questo livello sono TCP e UDP.
- L5 - sessione: si occupa di instaurare, mantenere e terminare connessioni.
- L6 - presentazione: trasforma i dati forniti dalle applicazione in un formato standardizzato.
- L7 - applicazione: svolge la funzione di interfaccia tra utente e macchina

I livelli che il kernel Linux gestisce sono il livello di collegamento dati, di rete e di trasporto, rispettivamente, L2, L3 e L4 [2].

2.1.2 I componenti fondamentali

Il sottosistema di rete comprende diversi componenti fondamentali che lavorano all'unisono per offrire solide funzionalità di networking [3]. Questi componenti sono:

- dispositivi di rete: Linux supporta svariati dispositivi di rete, tra cui Ethernet, Wi-Fi, Bluetooth e interfacce Virtual Private Network (VPN). Il kernel fornisce i driver necessari per comunicare con questi dispositivi.
- protocolli di rete: essi sono il fondamento principale della trasmissione dei dati attraverso le reti. Alcuni esempi dei protocolli supportati sono Internet Protocol (IP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP) e Internet Control Message Protocol (ICMP).
- interfacce di rete: esse permettono la comunicazione tra differenti livelli di rete. Queste includono l'interfaccia di loopback, ethernet, wireless e interfacce virtuali.

L'essenza dello stack del kernel consiste nel passare i pacchetti in arrivo da L2 (i driver dei dispositivi di rete) a L3 (il livello di rete, di solito IPv4 o IPv6) e quindi a L4 (dove si possono avere socket in ascolto UDP o TCP) se sono destinati alla macchina

locale o di nuovo a L2 per la trasmissione quando i pacchetti devono essere inoltrati. I pacchetti in uscita generati localmente passano da L4 a L3 e quindi a L2 per la trasmissione effettiva da parte del driver del dispositivo di rete.

2.1.3 Virtualizzazione di reti

La virtualizzazione della rete è diventata una parte integrante delle moderne infrastrutture IT. Essa permette di avere una rete basata su software invece che su hardware, creando un livello di astrazione fra l'hardware fisico e le applicazioni e i servizi che lo utilizzano. Può essere utilizzata per suddividere una rete fisica o connettere fra loro varie macchine virtuali.

Linux offre un eccellente supporto alla virtualizzazione della rete. Tecnologie come Virtual LAN (VLAN), Virtual Extensible LAN (VXLAN) e i namespace di rete consentono la creazione di ambienti di rete isolati, permettendo la coesistenza di più reti virtuali all'interno di una singola infrastruttura fisica.

2.1.4 Filtro di pacchetti e firewalling

Il sottosistema di rete include Netfilter, un potente framework di packet-filtering. Netfilter permette agli amministratori di sistema di implementare regole di firewall, effettuare traduzione degli indirizzi di rete (NAT - Network Address Translation) e controllare il flusso di traffico.

2.2 Configurazione della rete: iproute2

Il kernel Linux fornisce un esaustivo insieme di strumenti per configurare la rete. Questi consentono di avere un controllo granulare che va dalla gestione dei dispositivi e delle interfacce di rete, fino al controllo dei pacchetti in entrata e in uscita. Gli amministratori del sistema possono sfruttare questi strumenti per configurare la rete secondo le loro necessità.

La maggior parte degli strumenti atti a configurare la rete sono racchiusi nel progetto iproute2. Iproute è un insieme di utilità per controllare le reti tcp/ip e il traffico di rete in generale. I due strumenti più importanti all'interno di iproute sono *ip* e *tc*. *ip* controlla le configurazioni IPv4 e IPv6, mentre *tc* sta per *traffic control* e permette di avere un controllo maggiore a livello dei pacchetti di rete.

Per la configurazione delle reti, entrambi gli strumenti sopracitati fanno uso di Netlink (vedi Capitolo 3), un protocollo di comunicazione tra processi che verrà descritto in dettaglio più avanti.

Di seguito verrà descritto nel dettaglio cosa è possibile fare attraverso questi due strumenti.

2.2.1 Configurazioni Ipv4 e IPv6

Tramite lo strumento *ip* è possibile svolgere le seguenti operazioni (e non solo):

- visualizzare le informazioni di tutte le interfacce di rete
- gestire le tabelle di routing
- visualizzare tutte le regole in vigore
- aggiungere/rimuovere un indirizzo associato ad un'interfaccia
- creare/eliminare un'interfaccia virtuale
- aggiungere delle rotte di instradamento del traffico
- impostare un dispositivo come gateway predefinito
- gestire i namespace di rete

La lista sopra è stata limitata, in quanto, ai fini della tesi, si è più interessati alla sezione successiva riguardante il controllo del traffico.

2.2.2 Controllo del traffico

È possibile configurare il controllo del traffico attraverso lo strumento *tc*. Il controllo del traffico [4] consiste in:

- *Shaping* (modellare): un traffico modellato, è un traffico la cui velocità di trasmissione è sotto controllo. Non solo, il modellamento viene utilizzato anche per attenuare i picchi di traffico e migliorare il comportamento della rete.
- *Scheduling* (pianificazione): pianificando la trasmissione dei pacchetti è possibile migliorare l'interattività per il traffico che ne ha bisogno, garantendo comunque la larghezza di banda per i trasferimenti di massa.
- *Policing*: è il duale dello shaping, ma per il traffico in arrivo.
- *Dropping* (interrompere): il traffico che supera una determinata larghezza di banda può essere interrotto.

Il controllo del traffico sopra descritto si basa su tre tipi diversi di oggetti, qdiscs, classi e filtri, che sono alla base del funzionamento di *tc* [4].

qdiscs *Qdiscs* è l'abbreviazione di *queueing discipline*, vale a dire una disciplina di accodamento, e sono fondamentali per capire meglio come avviene il controllo del traffico. Ogniqualvolta il kernel invia un pacchetto ad un'interfaccia, esso viene messo in coda nella qdisc configurata per quella interfaccia. Una qdisc, quindi è come se fosse uno scheduler per i pacchetti di rete di una determinata interfaccia di rete. Per ogni interfaccia di rete sono presenti due qdisc: quella di ingresso e quella di uscita. La qdisc di default è una qdisc FIFO (First-In First-Out), che non opera alcun tipo di modifica o filtro ai pacchetti di rete.

classi Alcune qdisc possono contenere classi. In questo caso si parla di *classful qdiscs*. Le classi sono estremamente flessibili e possono contenere a loro volta altre classi o una singola qdisc; il traffico può essere messo in coda in una qualsiasi

delle qdisc interne alle classi. Quando il kernel cerca di estrarre un pacchetto da una classful qdisc, esso può venire da una qualsiasi classe al suo interno. Ad esempio, una qdisc potrebbe dare priorità ad un certo tipo di traffico, estraendo i pacchetti da determinate classi prima delle altre.

filtri I filtri vengono utilizzati dalle classful qdiscs per determinare in quale classe il pacchetto deve essere messo in coda. La classificazione può avvenire per via di un singolo filtro o di un insieme di filtri differenti. Il filtro può essere associato alla qdisc o anche ad una classe specifica.

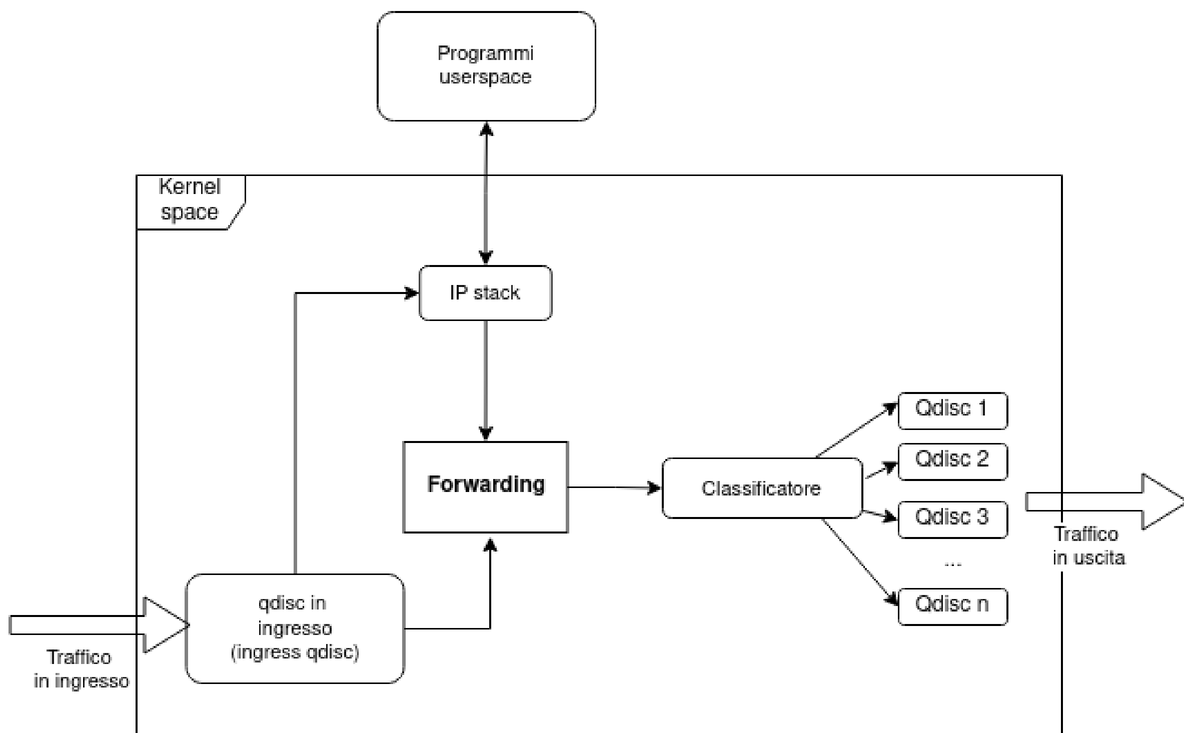

Figura 2.1: Schema di funzionamento del controllo del traffico

Nella figura 2.1 viene riportato lo schema di funzionamento del controllo del traffico nel kernel [5]. Il blocco principale rappresenta il kernel. La freccia a sinistra rappresenta il traffico in entrata nella macchina. Esso viene subito indirizzato alla qdisc di ingresso, la quale potrebbe applicare filtri e/o scartare il pacchetto (in base a delle policy date dai filtri stessi). Se il pacchetto viene lasciato proseguire, esso

potrebbe essere destinato ad un'applicazione locale, nel qual caso entra nello stack IP per essere elaborato e consegnato ad un programma user-space. Il pacchetto può anche essere inoltrato senza passare per un'applicazione locale; in questo caso è destinato all'uscita. Lì viene analizzato e messo in coda in una delle qdisc presenti, in base alla classificazione effettuata. Ora il pacchetto è in attesa che il kernel lo richieda per la trasmissione attraverso l'interfaccia di rete.

Come si è visto, il controllo del traffico nel kernel Linux offre la possibilità di gestire i pacchetti in entrata ed uscita in maniera efficace e personalizzata. Alcuni esempi di ciò che è possibile fare mediante *tc* sono:

- Limitazione della larghezza di banda di un'interfaccia di rete
- Simulazione di perdita o ritardo di pacchetti
- Prioritizzazione di certi tipi di traffico rispetto ad altri
- Gestione delle code di traffico
- Filtrare pacchetti in base a delle regole ed applicare loro azioni diverse

Capitolo 3

Netlink

Netlink è un protocollo usato all'interno dei sistemi operativi Linux per comunicare tra lo spazio del kernel space e lo spazio utente. Viene descritto nella RFC 3549 [6] anche come un protocollo intra-kernel. Netlink viene spesso definito come un rimpiazzo per `ioctl()` [7]. Il metodo `ioctl` presuppone che ogni configurazione diversa abbia un operazione ioctl associata e richiede che la configurazione sia espressa in un certo formato. Tradizionalmente, sono state utilizzate delle strutture dati con un layout fisso per le configurazioni. Quindi se l'aggiunta di nuove configurazione richiede di modificare questa struttura, è necessario aggiungere una nuova operazione ioctl, in quanto, modificando la precedente, si perderebbe la retrocompatibilità. Uno dei principali vantaggi di Netlink rispetto a ioctl è proprio questo: un messaggio Netlink ha un header di lunghezza fissa, seguito dal corpo del messaggio, che è un insieme di attributi rappresentati nel formato TLV (Tipo-Lunghezza-Valore). Questa struttura è spesso utilizzata in molti protocolli (ad esempio il campo "options" dell'header IPv4) per via della sua flessibilità e rende possibile l'aggiunta di funzionalità future [8].

3.1 Implementazione

Netlink poggia le sue fondamenta sull'archittetura dei socket generici BSD (vedi Appendice A), supporta quindi le primitive usuali come `socket()`, `bind()`, `sendmsg()` e

Netlink	ioctl
Comunicazione diretta: il kernel può inviare informazioni direttamente alle applicazioni attraverso Netlink	Polling esplicito: ioctl richiede l'interrogazione esplicita del kernel per ricevere informazioni, un'operazione relativamente costosa
Comunicazione asincrona	Comunicazione sincrona
Supporto di connessioni multicast	Solo connessioni unicast
Alcuni messaggi netlink potrebbero andare persi (ad esempio per memoria esaurita)	Più affidabile per via del processamento immediato

Tabella 3.1: Differenze tra Netlink e ioctl

`recvmsg()`. L'implementeazione del protocollo risiede quasi interamente nella cartella *net/netlink* nel codice sorgente del kernel.

Un esempio di invio di un messaggio Netlink, preso dalla documentazione del kernel [1], è mostrato nel Listato 3.1:

Listato 3.1: Apertura di un socket Netlink e invio di un messaggio

```
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_GENERIC);
/* format the request */
send(fd, &request, sizeof(request));
n = recv(fd, &response, RSP_BUFFER_SIZE);
/* interpret the response */
```

Nella prima riga, avviene l'apertura di un socket Netlink, specificando la famiglia `AF_NETLINK`, il tipo `SOCK_RAW` e il protocollo `NETLINK_GENERIC`. Più avanti verrà mostrata la differenza tra Netlink classico e Netlink generico. Si procede con la formattazione della richiesta, omessa per semplicità, e l'invio della stessa mediante `send()`. Si può quindi ricevere la risposta semplicemente utilizzando `recv()`, come un qualunque socket.

[1] https://docs.kernel.org/userspace-api/netlink/intro.html

3.1.1 Netlink Socket Bus

Netlink supporta fino a 32 bus nello spazio del kernel. In generale, ogni bus è collegato ad un sottosistema del kernel, ma è anche possibile che diversi sottosistemi condividano lo stesso bus. È il caso del bus Netfilter, *nfnetlink* usato in tutti i sottosistemi che si occupano di firewalling, e del bus di networking, *rtnetlink*, usato dai sottosistemi di gestione dei dispositivi di rete, routing, neighbouring e queueing discipline.

3.1.2 Formato dei Messaggi Netlink

I messaggi Netlink sono allineati a 32 bits, e generalmente i dati al loro interno sono espressi secondo l'ordine dei byte dell'host [8]. Come accennato precedentemente, un messaggio Netlink comincia sempre con un header fisso di 16 bytes definito dalla struttura *struct nlmsghdr* (vedi Listato 3.2).

Listato 3.2: Definizione di struct nlmsghdr in `<include/linux/netlink.h>`

```
    struct nlmsghdr
{
        __u32   nlmsg_len;
        __u16   nlmsg_type;
        __u16   nlmsg_flags;
        __u32   nlmsg_seq;
        __u32   nlmsg_pid;
};
```

Questo header contiene i seguenti campi:

- Lunghezza del messaggio (32 bits): dimensione del messaggio in bytes, incluso questo header.
- Tipo del messaggio (16 bits): il tipo del messaggio, che può essere dati o controllo. Un messaggio dati, dipende dall'insieme di azioni che il sottosistema permette; un messaggio di controllo, invece, è comune a tutti i sottosistemi, e ne esistono quattro tipi. I tipi di messaggi di controllo esistenti sono:
  - `NLMLG_NOOP`: "no operation", può essere utilizzato come una sorta di "ping", per verificare che il bus Netlink sia disponibile.

- NLMSG_ERROR: il messaggio contiene un errore.
- NLMSG_DONE: ultimo messaggio di una serie di messaggi.
- NLMSG_OVERRUN: attualmente inutilizzato.

■ Flags dei messaggi (16 bits): esistono diverse flag, o opzioni, come:

- NLM_F_REQUEST: indica che il messaggio Netlink contiene una richiesta. I messaggi inviati dallo user-space al kernel-space devono specificare questa flag.
- NLM_F_CREATE: indica che il messaggio vuole eseguire un comando o aggiungere una configurazione al sottosistema relativo.
- NLM_F_EXCL: comunemente utilizzata insieme a NLM_F_CREATE per attivare un errore se la configurazione che si vuole aggiungere è già esistente nel kernel-space.
- NLM_F_REPLACE: indica che il messaggio vuole rimpiazzare una configurazione esistente nel kernel.
- NLM_F_APPEND: aggiunge in coda una nuova configurazione a quella esistente. Questa flag viene utilizzata per dati ordinati, dove di default il dato viene aggiunto in testa e non in coda.
- NLM_F_DUMP: indica la richiesta di una sincronizzazione con il sottosistema del kernel. Come conseguenza, il kernel restituisce informazioni sul sottosistema.
- NLM_F_MULTI: indica che il messaggio è un messaggio “multi-part”, cioè una serie di messaggi.
- NLM_F_ACK: richiesta di una conferma dell’esecuzione con successo di una richiesta.
- NLM_F_ECHO: utilizzata quando si vuole avere un report di una richiesta inviata.

- Numero di sequenza (32 bits): numero di sequenza del messaggio. Utile quando si usa la flag `NLM_F_ACK`.

- ID di porta (32 bits): contiene un numero identificativo assegnato da Netlink, per identificare diversi canali aperti dallo stesso process user-space. Nel caso di comunicazione iniziata dal kernel, il suo valore è zero.

Una rappresentazione schematica della struttura dell'header è riportata nella Tabella 3.2.

Lunghezza del messaggio (32 bits)	
Tipo (16 bits)	Flags (16 bits)
Numero di sequenza (32 bits)	
Numero di porta (32 bits)	

Tabella 3.2: Header di un messaggio Netlink

Subito dopo l'header, viene il corpo del messaggio che è composto da un insieme di attributi espressi nel formato TLV (Tipo-Lunghezza-Valore). Ogni attributo Netlink è definito dalla struttura `struct nlattr` ed è composto dai seguenti campi:

- Tipo (16 bits): il tipo dell'attributo secondo l'insieme di tipi definiti nel kernel. I due bit più significativi sono usati per codificare se questo è un attributo nidificato (bit 0), cioè contiene un insieme di attributi nel corpo di un solo attributo e se il corpo dell'attributo è rappresentato secondo l'ordine dei byte della rete (bit 1).

- Lunghezza (16 bits): dimensione dell'attributo in byte.

- Valore: campo variabile in dimensione, ma sempre allineato a 32 bits.

Un esempio del corpo di un messaggio Netlink è riportato nella Tabella 3.3.

3.1.3 Validazione degli attributi

Netlink mette a disposizione una serie di policy per validare gli attributi nei messaggi. Una policy netlink è definita come segue:

Tipo (16 bits)	Lunghezza (16 bits)
Valore (32 bits)	
Tipo (16 bits)	Lunghezza (16 bits)
Valore (64 bits)	

Tabella 3.3: Corpo di un messaggio Netlink nel formato TLV

Listato 3.3: Definizione di nla_policy in <include/net/netlink.h>

```c
struct nla_policy {
  u8    type;
  u8    validation_type;
  u16   len;
  union {
    u16 strict_start_type;

    /* private: use NLA_POLICY_*() to set */
    const u32 bitfield32_valid;
    const u32 mask;
    const char *reject_message;
    const struct nla_policy *nested_policy;
    const struct netlink_range_validation *range;
    const struct netlink_range_validation_signed
                    *range_signed;
    struct {
      s16 min, max;
    };
    int (*validate)(const struct nlattr *attr,
        struct netlink_ext_ack *extack);
  };
};
```

Nel dettaglio, i campi che una struttura `nla_policy` ha sono:

- Tipo: il tipo che l'attributo deve avere.
- Tipo di validazione: il tipo di validazione aggiuntiva da effettuare, oltre a quello di verifica del tipo di attributo. Ad esempio, accertarsi che l'attributo sia all'interno di un determinato intervallo o che rispetti una maschera di bit.
- Lunghezza: lunghezza specifica di quel tipo di attributo
- `union \{...\}` : la union è considerata privata e deve essere valorizzata solo mediante le apposite macro definite.

Le policy vengono definite come array di questa struct. L'array deve essere accessibile per tipo di attributo, fino all'identificatore più alto che ci si aspetta. Gli attributi Netlink vengono dichiarati come `enum`, di conseguenza avranno un numero progressivo da 0 a N, dove N è il numero massimo di attributi che quel tipo di sottosistema supporta.

Nel momento in cui il messaggio Netlink inviato è malformato o genera un errore, si otterrà come risposta un messaggio di errore.

3.1.4 Messaggi di Errore in Netlink

Per riportare gli errori allo user-space, Netlink fornisce un tipo di messaggio di errore. Questo è un messaggio Netlink definito dalla struttura `struct nlmsgerr`, visibile nel Listato 3.4. Essa contiene due campi:

- Tipo di errore (32 bits): intero che definisce il tipo di errore. Il significato del codice di errore può essere ricavato dal file di header *<errno.h>*.
- Messaggio Netlink che contiene la richiesta da cui è scaturito l'errore.

Listato 3.4: Definizione di nlmsgerr in `<include/uapi/linux/netlink.h>`

```
struct nlmsgerr {
  int   error;
  struct nlmsghdr msg;
 };
```

3.2 Classic Netlink vs Generic Netlink

L'implementazione iniziale di Netlink, nota come *Classic Netlink* e descritta finora, dipende da un'allocazione statica degli ID dei sottosistemi. Questo permette solo la definizione di un numero limitato di sottosistemi. *Generic Netlink* è stato proposto per risolvere questo problema.

3.2.1 Generic Netlink

In Generic Netlink, ogni protocollo Netlink definisce il suo header di metadati. Il messaggio Netlink comincia sempre con la
`struct nlmsghdr` (vedi Listato 3.2), dove alcuni campi assumono però un significato differente, seguita da una `struct genlmgshdr` (Listato 3.5). In Classic Netlink, il campo `nlmgs_type` specificava l'operazione da svolgere nel sottosistema. In Generic Netlink, invece, dato che c'è bisogno di condensare più protocolli in uno solo, questo campo è riservato all'ID del sottosistema che si sta utilizzando e il campo `cmd` di `genlmsghdr` identifica l'operazione da svolgere. Gli altri campi non sono rilevanti nella costruzione del messaggio, in quanto devono essere impostati a dei valori predefiniti.

Listato 3.5: Definizione di genlmsghdr in `<include/uapi/linux/genetlink.h>`

```
struct genlmsghdr {
      __u8    cmd;
      __u8    version;        /* Irrelevant, set to 1 */
      __u16   reserved;       /* Reserved, set to 0 */
};
```

Capitolo 4

Processi di Revisione del Codice nello Sviluppo del Kernel

Per poter discutere dei processi di revisione del codice nello sviluppo del kernel, è bene prima avere un'idea generale di come avviene il processo di sviluppo.

Esistono diverse categorie in cui possono rientrare le release del kernel, e sono:

Prepatch Chiamate anche "rc", sono versioni del kernel pre-release. Spesso contengono nuove funzionalità che però devono ancora essere testate prima di introdurle in una versione stabile

Mainline È il tree di git dove avviene il vero e proprio sviluppo del kernel

Stable È il rilascio di un kernel mainline.

Longterm Sono release del kernel fornite per estendere la soluzione di un bug anche alle versioni precedenti.

Nella Figura 4.1, viene riportato un diagramma di flusso rappresentante i vari possibili stati.

4.1 Le Fasi dello Sviluppo del Kernel

Gli sviluppatori del kernel seguono un processo di rilascio basato sul tempo. Normalmente viene pubblicata una nuova versione del kernel ogni due o tre mesi.

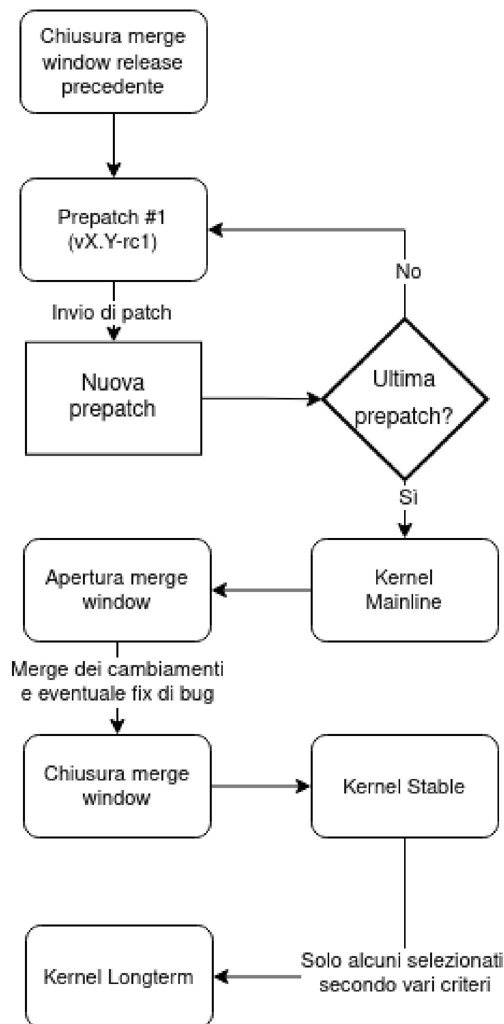

Figura 4.1: Diagramma di flusso degli stadi che attraversa il kernel durante il suo sviluppo

Il ciclo di sviluppo di una nuova versione, inizia con l'apertura della "merge window". In questo periodo, i cambiamenti fatti finora, che vengono ritenuti adeguatamente stabili e che sono stati accettati dalla comunità di sviluppatori, vengono inclusi nel kernel. Questa finestra di tempo in cui unire i cambiamenti già fatti, dura all'incirca due settimane. Alla fine di questo periodo Linus Torvalds chiude la finestra e rilascia la prima versione *rc* del kernel, chiamata *rc1*, che segna la fine della merge window e l'inizio di un nuovo kernel . Seguono da sei a dodici settimane in cui si inviano nel kernel mainline solo le patch che risolvono problemi. Linus rilascia una nuova

versione *rc* del kernel con cadenza più o meno settimanale. Solitamente, questo processo si stoppa tra la sesta e la nona iterazione (rc6/rc9), fino a quando il kernel non è considerato sufficientemente stabile per la release finale. Un esempio è visibile nella Figura 4.2.

Tag	Download	Author	Age
v6.8-rc1	net-next-6.8-rc1.tar.gz	Linus Torvalds	11 days
v6.7	net-next-6.7.tar.gz	Linus Torvalds	4 weeks
v6.7-rc8	net-next-6.7-rc8.tar.gz	Linus Torvalds	5 weeks
v6.7-rc7	net-next-6.7-rc7.tar.gz	Linus Torvalds	6 weeks
v6.7-rc6	net-next-6.7-rc6.tar.gz	Linus Torvalds	7 weeks
v6.7-rc5	net-next-6.7-rc5.tar.gz	Linus Torvalds	8 weeks
v6.7-rc4	net-next-6.7-rc4.tar.gz	Linus Torvalds	2 months
v6.7-rc3	net-next-6.7-rc3.tar.gz	Linus Torvalds	2 months
v6.7-rc2	net-next-6.7-rc2.tar.gz	Linus Torvalds	2 months
v6.7-rc1	net-next-6.7-rc1.tar.gz	Linus Torvalds	3 months

Figura 4.2: Rilascio delle versioni rc del kernel 6.7

Una volta che la versione del kernel diventa stabile, la sua manutenzione viene affidata allo “stable team”, che si occuperà del rilascio di aggiornamenti occasionali.

Il processo di revisione del codice del kernel Linux, è molto differente dai moderni processi di revisione che utilizzano strumenti e piattaforme online innovative e user-friendly. Al contrario, esso si basa esclusivamente su git e sull’utilizzo di mailing list: per ogni sottosistema (o parte significativa di esso) esiste una mailing list dedicata a cui inviare le proprie patch. Questo è dovuto sia a ragioni storiche e sia alle esigenze della comunità sviluppate nel tempo[1].

4.2 Ciclo di Vita di una Patch

Le *patch*, sono il mezzo tramite il quale proporre cambiamenti nel kernel. Esse non finiscono direttamente nel kernel mainline, ma seguono un processo di revisione per assicurarsi che soddisfino requisiti di qualità e che implementino cambiamen-

[1] Torvalds a riguardo delle pull request di Github: https://github.com/torvalds/linux/pull/17

ti desiderabili. Questo processo può essere veloce, come molto lento, nel caso di cambiamenti controversi e articolati.

I passi che una patch compie nel suo ciclo di vita sono raffigurati nella Figura 4.3 e vengono descritti nel dettaglio di seguito:

- progettazione: in questo step si delineano i requisiti della patch, nel particolare, si specificano quali problemi la patch va a risolvere e come.

- invio della patch: prima di inviare una patch, ci si deve assicurare di soddisfare dei requisiti minimi in termini di formattazione e qualità del codice. Infatti, è necessario riuscire a compilare senza problemi il kernel dopo l'applicazione della patch; per la formattazione, invece, viene messo a disposizione uno script per verificare il rispetto delle principali linee guida[2], situato in *scripts/checkpatch.pl*.

- review iniziale: in questa fase, la patch dopo essere inviata alla mailing list di competenza, viene revisionata dal resto degli sviluppatori per risolvere eventuali dubbi e problemi

- review finale: prima di essere accettata nel kernel mainline, la review deve essere effettuata da un *mantainer* rilevante del sottosistema in oggetto.

- inclusione nel kernel mainline: una patch accettata, viene poi aggiunta al kernel mainline gestito da Torvalds.

Nelle ultime due fasi, la patch viene integrata assieme ad altri cambiamenti, facendo emergere eventuali problemi o conflitti, e venendo così sottoposta ad ulteriori revisioni da parte di altri membri della comunità.

Questo processo assicura degli standard di qualità molto elevati per il codice del kernel, oltre a garantire una maggiore affidabilità dello stesso, visti i diversi passaggi effettuati prima di essere considerato stabile.

[2] È possibile che il sottosistema abbia delle sue regole specifiche, è quindi consigliabile verificare che la patch le soddisfi prima di inviarla.

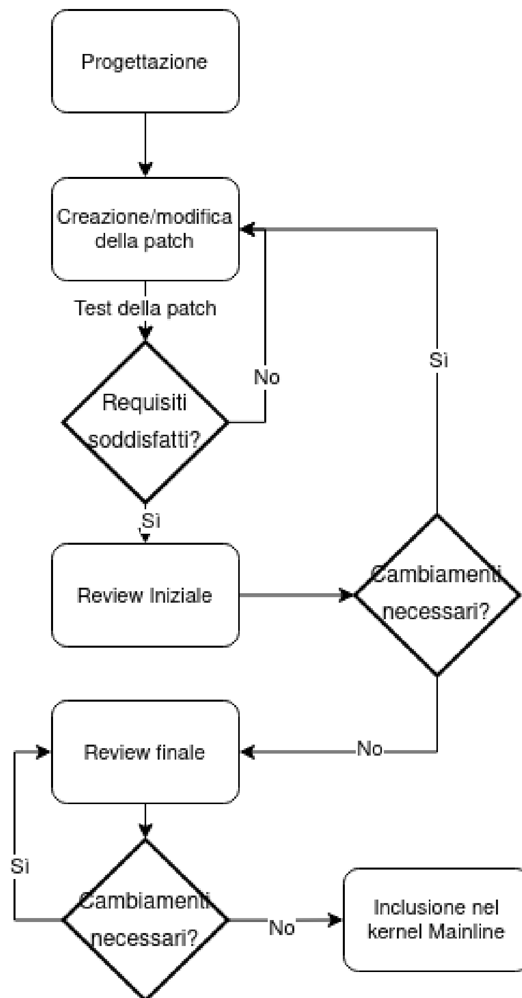

Figura 4.3: Diagramma di flusso degli stadi che attraversa una patch durante il suo sviluppo

Capitolo 5

Validazione dei Parametri della qdisc TAPRIO mediante Netlink

Il primo lavoro di questa tesi, si è svolto sullo scheduler di rete TAPRIO (Time-Aware Priority Scheduler). TAPRIO è una qdisc che implementa una versione semplificata dell'automa di scheduling a stati finiti definito in IEEE 802.1Q-2018.

Dato che la qdisc TAPRIO si basa sulla qdisc MQPRIO (Multiqueue Priority Qdisc), verrà trattata dopo una breve introduzione a quest'ultima.

5.1 MQPRIO (Multiqueue Priority Qdisc)

La qdisc MQPRIO [9] consente di mappare i flussi di traffico su intervalli di code hardware, definendo una priorità. Per impostazione predefinita, la qdisc alloca una qdisc *pfifo* (una coda first in, first out a pacchetti limitati) per ogni coda di trasmissione esposta dal dispositivo di livello inferiore. Altre discipline di accodamento possono essere aggiunte successivamente.

Di seguito viene descritto cosa accade a livello software e hardware, durante la creazione di una qdisc MQPRIO:

1. La qdisc viene assegnata all'interfaccia di rete
2. Viene creato il numero specificato di classi di traffico
3. Viene mappata ogni classe ad una priorità.

4. Ogni classe viene assegnata ad una coda
5. Infine, il driver dell'interfaccia di rete si occupa di prioritizzare correttamente ciascuna coda.

In Figura 5.1, è possibile vedere a livello schematico, le azioni appena descritte.

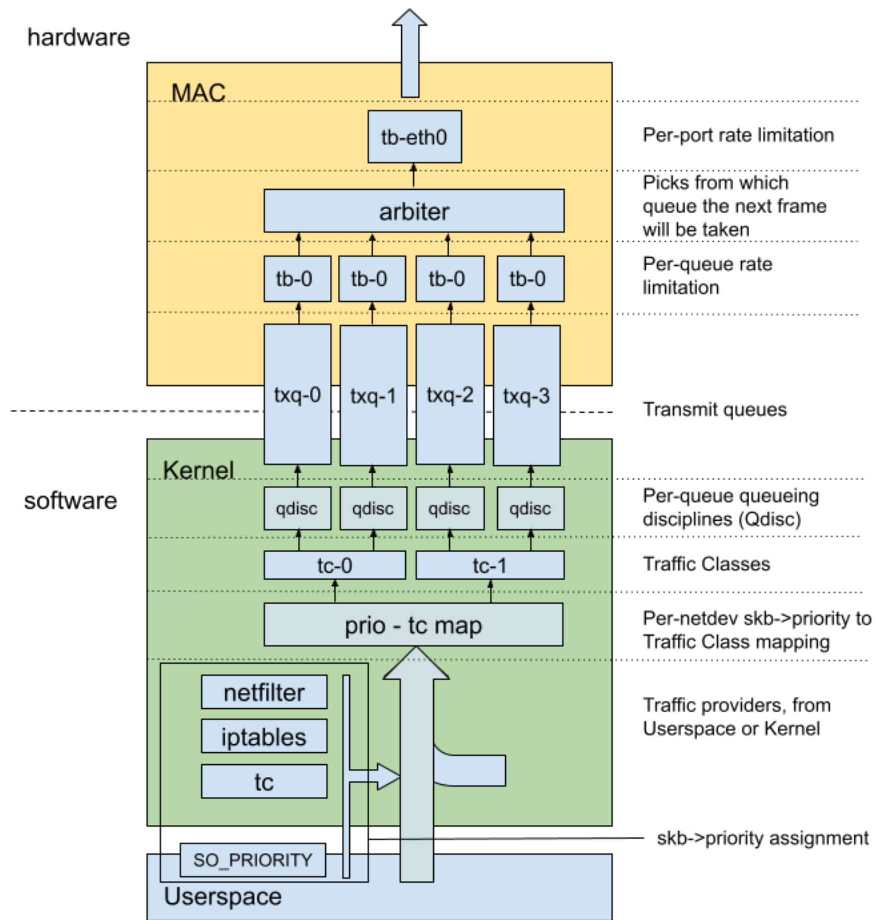

Figura 5.1: La qdisc MQPRIO. Fonte: [10]

5.2 TAPRIO (Time-Aware Priority Scheduler)

L'automa di scheduling a stati finiti che questa qdisc implementa, permette la configurazione di una serie di gate, i quali, permettono (o meno) il traffico in uscita per

un sottoinsieme di classi di traffico. Il modo in cui il traffico viene mappato a code hardware è uguale alla qdisc *mqprio*, descritta precedentemente.

5.2.1 Parametri di una Qdisc TAPRIO

La Tabella 5.1 mostra i parametri che è possibile specificare in fase di creazione di una qdisc TAPRIO[11].

Parametro	Descrizione
num_tc	Numero di classi di traffico da usare (massimo sedici)
map	Assegnazione delle priorità per ogni classe di traffico
queues	Mappatura delle classi di traffico alle code hardware
base-time	Definisce il l'istante d'inizio della schedulazione
clockid	Specifica l'orologio che il timer interno di ogni qdisc deve utilizzare
sched-entry	Segue il formato <comando> <gatemask> <intervallo>. Il comando è uno solo ed è "S" che sta a significare "SetGateStates"; <gate mask> è una maschera di bit, dove ogni bit corrisponde ad una classe di traffico e stabilisce se la classe di traffico è attiva o meno; <intervallo> è la durata di tempo in nanosecondi, che specifica fino a quando lo stato specificato va mantenuto. È possibile specificare più parametri di questo tipo. La Figura 5.2 schematizza questo comportamento
flags	Maschera di bit che specifica differenti modalità operative. I valori possibili sono mutuamente esclusivi e sono `0x1` e `0x2`, per indicare rispettivamente le modalità, *txtime-assist* e *full-offload*
txtime-delay	È specifico della modalità *full-offload*. Definisce il tempo massimo che un pacchetto può impiegare per raggiungere l'interfaccia di rete dalla qdisc.
max-sdu	Specifica la dimensione massima dei pacchetti L2 per ogni classe di traffico

Tabella 5.1: Parametri della Qdisc TAPRIO

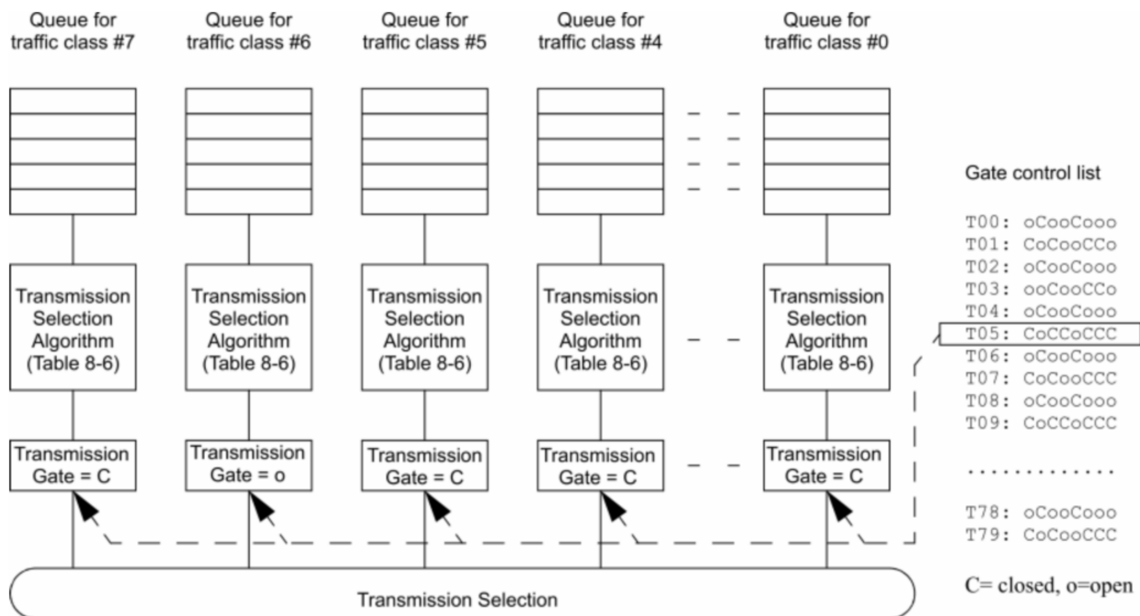

Figura 5.2: Selezione delle classi di traffico mediante gates. Fonte:[12]

5.2.2 Attributi Netlink di TAPRIO

Come già detto in precedenza, per aggiungere o modificare delle configurazioni di rete nel kernel, spesso si interagisce mediante Netlink, e la qdisc TAPRIO non è un'eccezione. La definizione degli attributi Netlink relativi a TAPRIO, è disponibile nel file <*include/uapi/linux/pkt_sched.h*>, visibile nel Listato 5.1.

Mentre la policy di validazione degli stessi è visibile nel file *net/sched/sch_taprio.c* e nel Listato 5.2.

Si vuole porre l'attenzione sull'attributo `TCA_TAPRIO_ATTR_FLAGS` e sulla sua policy di validazione. Dai listati è possibile vedere che la policy di validazione dell'attributo "flags" specifica solo il tipo che l'attributo deve avere e non i valori che esso può assumere. Ma come specificato dal manuale [11] questo attributo può assumere solo tre valori distinti:

- `0x0`: nessuna modalità operativa particolare specificata
- `0x1`: modalità operativa "txtime-assist"

Listato 5.1: Definizione degli attributi Netlink di TAPRIO

```
enum {
  TCA_TAPRIO_ATTR_UNSPEC,
  TCA_TAPRIO_ATTR_PRIOMAP, /* struct tc_mqprio_qopt */
  TCA_TAPRIO_ATTR_SCHED_ENTRY_LIST, /* nested of entry */
  TCA_TAPRIO_ATTR_SCHED_BASE_TIME, /* s64 */
  TCA_TAPRIO_ATTR_SCHED_SINGLE_ENTRY, /* single entry */
  TCA_TAPRIO_ATTR_SCHED_CLOCKID, /* s32 */
  TCA_TAPRIO_PAD,
  TCA_TAPRIO_ATTR_ADMIN_SCHED, /* The admin sched, only used in dump */
  TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME, /* s64 */
  TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME_EXTENSION, /* s64 */
  TCA_TAPRIO_ATTR_FLAGS, /* u32 */
  TCA_TAPRIO_ATTR_TXTIME_DELAY, /* u32 */
  TCA_TAPRIO_ATTR_TC_ENTRY, /* nest */
  __TCA_TAPRIO_ATTR_MAX,
};

#define TCA_TAPRIO_ATTR_MAX (__TCA_TAPRIO_ATTR_MAX - 1)
```

Listato 5.2: Policy di validzione degli attributi Netlink di TAPRIO

```
static const struct nla_policy taprio_policy[TCA_TAPRIO_ATTR_MAX + 1] = {
[TCA_TAPRIO_ATTR_PRIOMAP]       = {
  .len = sizeof(struct tc_mqprio_qopt)
},
[TCA_TAPRIO_ATTR_SCHED_ENTRY_LIST]           = { .type = NLA_NESTED },
[TCA_TAPRIO_ATTR_SCHED_BASE_TIME]            = { .type = NLA_S64 },
[TCA_TAPRIO_ATTR_SCHED_SINGLE_ENTRY]         = { .type = NLA_NESTED },
[TCA_TAPRIO_ATTR_SCHED_CLOCKID]              = { .type = NLA_S32 },
[TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME]           =
  NLA_POLICY_FULL_RANGE_SIGNED(NLA_S64, &taprio_cycle_time_range),
[TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME_EXTENSION] = { .type = NLA_S64 },
[TCA_TAPRIO_ATTR_FLAGS]                      = { .type = NLA_U32 },
[TCA_TAPRIO_ATTR_TXTIME_DELAY]         = { .type = NLA_U32 },
[TCA_TAPRIO_ATTR_TC_ENTRY]         = { .type = NLA_NESTED },
};
```

- `0x2`: modalità operativa "full-offload"

Quindi la policy manca di questo controllo. Infatti, sempre nel file *net/sched/-sch_taprio.c* è possibile notare come questo attributo sia validato mediante controllo "manuale". Le parti rilevanti sono visibili nel Listato 5.3

La funzione `taprio_flags_valid()` si occupa di verificare la validità dell'attributo, controllando per prima cosa che esso assuma solo i valori possibili e, in secondo luogo, che non siano state specificate entrambe le modalità operative contemporaneamente. La funzione `taprio_new_flags()`, invece, è responsabile di estrarre il valore dall'attributo Netlink, mediante `nla_get_u32()`, verificare che non si cambi modalità operativa mentre la qdisc è in esecuzione e, infine, chiamare la funzione precedentemente descritta.

Per avere una migliore visione d'insieme, il Listato 5.4 mostra come sono definite le due macro, `TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST` e `TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD`.

La macro `_BITUL(n)` definisce una maschera di bit che seleziona solo l'n-esimo bit.

5.3 Primo Approccio ad una Validazione Migliore

Una prima soluzione che si è esplorata per validare l'attributo "flags" mediante Netlink è stata la seguente: cambiare il tipo di `TCA_TAPRIO_ATTR_FLAGS` (vedi Listato 5.1) da `nla_u32` a `nla_bitfield32`. Il tipo `nla_bitfield32` (vedi Listato 5.5) è una `struct` con due campi:

- valore: il valore che l'attributo possiede.
- selettore: la maschera di bit valida per quell'attributo.

L'utilizzo di questo tipo di attributo, assieme all'utilizzo della policy Netlink associata (`NLA_POLICY_BITFIELD32`), assicurerebbe una validazione dell'attributo fatta al

Listato 5.3: Validazione manuale dell'attributo "flags" di TAPRIO

```
static bool taprio_flags_valid(u32 flags)
{
  /* Make sure no other flag bits are set. */
  if (flags & ~(TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST |
          TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD))
    return false;
  /* txtime-assist and full offload are mutually exclusive */
  if ((flags & TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST) &&
      (flags & TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD))
    return false;
  return true;
}

[...]

/* The semantics of the 'flags' argument in relation to 'change()'
 * requests, are interpreted following two rules (which are applied in
 * this order): (1) an omitted 'flags' argument is interpreted as
 * zero; (2) the 'flags' of a "running" taprio instance cannot be
 * changed.
 */
static int taprio_new_flags(const struct nlattr *attr, u32 old,
          struct netlink_ext_ack *extack)
{
  u32 new = 0;

  if (attr)
    new = nla_get_u32(attr);

  if (old != TAPRIO_FLAGS_INVALID && old != new) {
    NL_SET_ERR_MSG_MOD(extack,
            "Changing 'flags' of a running schedule is not supported");
    return -EOPNOTSUPP;
  }

  if (!taprio_flags_valid(new)) {
    NL_SET_ERR_MSG_MOD(extack,
            "Specified 'flags' are not valid");
    return -EINVAL;
  }

  return new;
}
```

Listato 5.4: Definizione delle macro relative alle modalità operative di TAPRIO

```
#define TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST  _BITUL(0)
#define TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD _BITUL(1)
```

Listato 5.5: La struct `nla_bitfield32`

```
struct nla_bitfield32 {
  __u32 value;
  __u32 selector;
};
```

livello di Netlink, invece di un controllo manuale effettuato dal kernel. Il problema è che questa soluzione ha un difetto: va a modificare le user-space api e questo non è possibile, in quanto può intaccare la retrocompatibilità con le versioni precedenti del kernel. Infatti, una delle tante prerogative dello sviluppo del kernel è quella di garantire sempre la retrocompatibilità.

Ciò che invece è possibile fare, è aggiungere un nuovo attributo Netlink con tipo `nla_bitfield32` e utilizzarlo in concomitanza con il precedente attributo, avendo cura di controllare che venga utilizzato uno solo dei due alla volta. Con questa soluzione, le nuove versioni del kernel utilizzerebbero il nuovo attributo, mentre, per le versioni precedenti non si avrebbe alcun cambiamento nella modalità operativa.

5.3.1 Considerazioni sul Primo Approccio

Di fatto, questa prima soluzione delineata, non è di certo la migliore possibile. Di seguito alcune considerazioni:

- Utilizza due attributi Netlink per descriverne uno: è una ridondanza non molto utile e comporta un utilizzo maggiore (sebbene minimo) di memoria.
- La validazione viene svolta parzialmente tramite Netlink: se viene utilizzato il vecchio attributo, la validazione verrà effettuata manualmente, con il metodo pre-esistente.
- Richiede un controllo più complesso lato kernel: si devono gestire due attributi che esprimono lo stesso parametro e che richiedono validazioni e trattamenti differenti.

Listato 5.6: La policy `NLA_POLICY_MASK`

```
#define NLA_POLICY_MASK(tp, _mask) {        \
  .type = NLA_ENSURE_UINT_TYPE(tp),    \
  .validation_type = NLA_VALIDATE_MASK,    \
  .mask = _mask,          \
}
```

- Può creare confusione per l'utente finale

Fatte queste considerazioni, si è deciso di esplorare un approccio differente.

5.4 Un Migliore Approccio alla Validazione

Partendo dalle considerazioni sull'approccio precedente, si è capito che l'aggiunta di un nuovo attributo non è una strada percorribile. È quindi necessario trovare una soluzione utilizzando l'attributo esistente così com'è, senza porre alcuna modifica.

Analizzando le varie policy messe a disposizione da Netlink, è interessante approfondire `NLA_POLICY_MASK` (Listato 5.6).

La policy, è una macro che si occupa di valorizzare alcuni dei campi privati della policy Netlink, come visto in 3.1.3. Nel particolare, impone che:

- il tipo dell'attributo sia un `uint` (unsigned integer)
- il tipo di validazione sia un controllo su una maschera di bit
- la maschera di bit da utilizzare per la validazione sia `_mask`

Quindi, è possibile utilizzare questa policy per validare l'attributo flags, visto che si applica al caso d'uso in questione. Prima di tutto è necessario definire una maschera di bit che rappresenti i possibili valori che l'attributo può assumere. È possibile fare ciò con la seguente macro:

Così facendo, `TAPRIO_SUPPORTED_FLAGS` sarà una maschera di bit pari a `00000011`, cioè permetterà solo valori compresi tra `0x0` e `0x3`. Avendo definito la maschera di bit, viene modificata la policy di conseguenza:

Listato 5.7: Definizione della bitmask da utilizzare per la validazione

```
#define TAPRIO_SUPPORTED_FLAGS \
  (TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST | TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD
     )
```

Listato 5.8: Modifica della policy Netlink di validazione

```
- [TCA_TAPRIO_ATTR_FLAGS]                         = { .type = NLA_U32 },
+ [TCA_TAPRIO_ATTR_FLAGS]                         =
+   NLA_POLICY_MASK(NLA_U32, TAPRIO_SUPPORTED_FLAGS),
```

Rimane da verificare la mutua esclusività: infatti, il controllo sulla maschera di bit permette anche valori pari a `0x3`, ma ciò non è possibile in quanto si può utilizzare una sola modalità operativa alla volta. Dopo alcune valutazioni, però, si è deciso di continuare a verificare la mutua esclusività manualmente. Questo perché non esiste una policy su misura per questo obiettivo, e vista la particolarità, non ha senso implementarne una ad hoc. Nel Listati 5.9-5.11 è visibile il *diff* completo della patch.

Listato 5.9: Patch finale per la qdisc TAPRIO - parte 1

```
diff --git a/net/sched/sch_taprio.c b/net/sched/sch_taprio.c
index 31a8252bd09c91..827cb683efbff9 100644
--- a/net/sched/sch_taprio.c
+++ b/net/sched/sch_taprio.c
@@ -40,6 +40,8 @@ static struct static_key_false
    taprio_have_working_mqprio;

 #define TXTIME_ASSIST_IS_ENABLED(flags) ((flags) &
     TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST)
 #define FULL_OFFLOAD_IS_ENABLED(flags) ((flags) &
     TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD)
+#define TAPRIO_SUPPORTED_FLAGS \
+ (TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST | TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD
    )
 #define TAPRIO_FLAGS_INVALID U32_MAX

 struct sched_entry {
@@ -408,19 +410,6 @@ static bool is_valid_interval(struct sk_buff *skb,
    struct Qdisc *sch)
  return entry;
 }

-static bool taprio_flags_valid(u32 flags)
-{
- /* Make sure no other flag bits are set. */
- if (flags & ~(TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST |
-        TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD))
-   return false;
- /* txtime-assist and full offload are mutually exclusive */
- if ((flags & TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST) &&
-     (flags & TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD))
-   return false;
- return true;
-}
-
 /* This returns the tstamp value set by TCP in terms of the set clock.
     */
 static ktime_t get_tcp_tstamp(struct taprio_sched *q, struct sk_buff *
     skb)
 {
@@ -1031,7 +1020,8 @@ static const struct nla_policy taprio_policy[
    TCA_TAPRIO_ATTR_MAX + 1] = {
  [TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME]          =
    NLA_POLICY_FULL_RANGE_SIGNED(NLA_S64, &taprio_cycle_time_range),
  [TCA_TAPRIO_ATTR_SCHED_CYCLE_TIME_EXTENSION] = { .type = NLA_S64 },
- [TCA_TAPRIO_ATTR_FLAGS]                      = { .type = NLA_U32 },
+ [TCA_TAPRIO_ATTR_FLAGS]                      =
+   NLA_POLICY_MASK(NLA_U32, TAPRIO_SUPPORTED_FLAGS),
  [TCA_TAPRIO_ATTR_TXTIME_DELAY]         = { .type = NLA_U32 },
  [TCA_TAPRIO_ATTR_TC_ENTRY]        = { .type = NLA_NESTED },
 };
```

Listato 5.10: Patch finale per la qdisc TAPRIO - parte 2

```
@@ -1815,33 +1805,6 @@ static int taprio_mqprio_cmp(const struct
    net_device *dev,
  return 0;
 }

-/* The semantics of the 'flags' argument in relation to 'change()'
- * requests, are interpreted following two rules (which are applied in
- * this order): (1) an omitted 'flags' argument is interpreted as
- * zero; (2) the 'flags' of a "running" taprio instance cannot be
- * changed.
- */
-static int taprio_new_flags(const struct nlattr *attr, u32 old,
-         struct netlink_ext_ack *extack)
-{
- u32 new = 0;
-
- if (attr)
-   new = nla_get_u32(attr);
-
- if (old != TAPRIO_FLAGS_INVALID && old != new) {
-   NL_SET_ERR_MSG_MOD(extack, "Changing 'flags' of a running schedule is
    not supported");
-   return -EOPNOTSUPP;
- }
-
- if (!taprio_flags_valid(new)) {
-   NL_SET_ERR_MSG_MOD(extack, "Specified 'flags' are not valid");
-   return -EINVAL;
- }
-
- return new;
-}
-
 static int taprio_change(struct Qdisc *sch, struct nlattr *opt,
       struct netlink_ext_ack *extack)
 {
@@ -1852,6 +1815,7 @@ static int taprio_change(struct Qdisc *sch, struct
    nlattr *opt,
  struct net_device *dev = qdisc_dev(sch);
  struct tc_mqprio_qopt *mqprio = NULL;
  unsigned long flags;
+ u32 taprio_flags;
  ktime_t start;
  int i, err;
```

Listato 5.11: Patch finale per la qdisc TAPRIO - parte 3

```
@@ -1863,12 +1827,28 @@ static int taprio_change(struct Qdisc *sch,
    struct nlattr *opt,
  if (tb[TCA_TAPRIO_ATTR_PRIOMAP])
    mqprio = nla_data(tb[TCA_TAPRIO_ATTR_PRIOMAP]);

- err = taprio_new_flags(tb[TCA_TAPRIO_ATTR_FLAGS],
-                 q->flags, extack);
- if (err < 0)
-   return err;
+ /* The semantics of the 'flags' argument in relation to 'change()'
+  * requests, are interpreted following two rules (which are applied in
+  * this order): (1) an omitted 'flags' argument is interpreted as
+  * zero; (2) the 'flags' of a "running" taprio instance cannot be
+  * changed.
+  */
+ taprio_flags = tb[TCA_TAPRIO_ATTR_FLAGS] ? nla_get_u32(tb[
    TCA_TAPRIO_ATTR_FLAGS]) : 0;

- q->flags = err;
+ /* txtime-assist and full offload are mutually exclusive */
+ if ((taprio_flags & TCA_TAPRIO_ATTR_FLAG_TXTIME_ASSIST) &&
+     (taprio_flags & TCA_TAPRIO_ATTR_FLAG_FULL_OFFLOAD)) {
+   NL_SET_ERR_MSG_ATTR(extack, tb[TCA_TAPRIO_ATTR_FLAGS],
+           "TXTIME_ASSIST and FULL_OFFLOAD are mutually exclusive");
+   return -EINVAL;
+ }
+
+ if (q->flags != TAPRIO_FLAGS_INVALID && q->flags != taprio_flags) {
+   NL_SET_ERR_MSG_MOD(extack,
+          "Changing 'flags' of a running schedule is not supported");
+   return -EOPNOTSUPP;
+ }
+ q->flags = taprio_flags;

  err = taprio_parse_mqprio_opt(dev, mqprio, extack, q->flags);
  if (err < 0)
```

Capitolo 6

Lo Strumento a Riga di Comando ynl

Citando Jakub Kicinski, maintainer del sottosistema di networking e ideatore di ynl:

> Netlink seems simple and reasonable to those who understand it. It appears cumbersome and arcane to those who don't.

è chiaro che Netlink, pur essendo un potente mezzo, può apparire complesso e arcano ad alcuni. Per questo, lui stesso ha proposto [13] delle specifiche del protocollo Netlink scritte in YAML, per creare, inviare e ricevere messaggi Netlink mediante un'utilità a riga di comando: ynl.

6.1 Archittettura di ynl

Ynl è capace di svolgere diverse funzioni, come comunicare con il kernel attraverso Netlink o generare codice per il kernel da delle specifiche. Tutto questo è reso possibile dalle seguenti componenti:

- Un insieme di librerie, scritte sia in C che in Python, che definiscono le caratteristiche del protocollo Netlink e permettono la codifica e decodifica dei messaggi.
- `cli.py`: uno script Python che si occupa di gestire l'input ed interagire con le libreria sopracitate.

- Un insieme di specifiche, scritte in YAML, ognuna relativa ad una famiglia Netlink diversa.

Tutti i file relativi ad ynl sono visibili nella cartella `tools/net/ynl`, mentre le specifiche YAML sono in `Documentation/netlink/specs`.

6.2 Le Specifiche Netlink in YAML

Le specifiche in YAML, relative alle famiglie Netlink, sono caratterizzate dalle seguenti sezioni [14]:

- *globals*: attributi definiti al livello principale del file. Specificano informazioni relativi alla famiglia Netlink utilizzata e al tipo di Netlink (classic o generic).
- *definitions*: definiscono una serie di tipi e costanti rilevanti. Per ogni voce all'interno di questa sezione viene definito un nome, un tipo e un valore.
- *attribute-sets*: specifica tutti gli attributi Netlink per quella famiglia. Tutte le famiglie hanno almeno un insieme di attributi (molto spesso più di uno). Per ogni insieme di attributo viene specificato il nome e una lista di attributi contenuti al suo interno, specificandone nome e tipo.
- *operations*: definiscono le operazioni che si possono eseguire. Ci sono tre possibili tipi di operazioni: operazioni, notifiche ed eventi. Le operazioni definiscono il tipo di interazione più comune con il kernel, quella del tipo richiesta-risposta. Le notifiche e gli eventi, invece, fanno entrambi riferimento ai messaggi asincroni inviati dal kernel su connessioni multi-cast.

Ai fini della tesi, verrà posta l'attenzione sulla versione "classica" di Netlink.

6.2.1 Proprietà Aggiuntive Richieste da Classic Netlink

Le famiglie che si appoggiano sulla versione classica di Netlink, la cui specifica YAML si trova in `Documentation/netlink/netlink-raw.yaml`, richiedono alcune proprietà

addizionali [15]. Nella sezione *globals*, viene aggiunto un campo chiamato `protonum` che specifica l'ID del protocollo Netlink da utilizzare, ad esempio. Un'altra peculiarità che non si trova nel tipo Generic è la sezione `sub-message`. I `sub-message` sono utili, in quanto, in diversi protocolli Netlink Classic si fa uso di attributi annidati come livello di astrazione per informazioni specfiche dei moduli sottostanti.

```
[LIVELLO PRINCIPALE DELLA SPECIFICA]
  [ATTRIBUTO GENERICO 1]
  [ATTRIBUTO GENERICO 2]
  [ATTRIBUTO GENERICO 3]
  [ATTRIBUTO GENERICO - contenitore]
    [ATTRIBUTO SPECIFICO MODULO 1]
    [ATTRIBUTO SPECIFICO MODULO 2]
```

Nel particolare, si ha un attributo che fa da contenitore, e che contiene informazioni specifiche di un modulo o di un altro in base a un selettore. Il selettore, fa riferimento al valore di un altro attributo. Nella specifica YAML, ogni `sub-message` viene espresso come una voce all'interno dell'omonima sezione, e contiene una serie di formati costituiti da un valore e un set di attributi. Un esempio pratico, lo si trova nella specifica YAML di `tc` (Listati 6.1-6.2), al momento ancora in fase di sviluppo.

Come si può notare, all'interno dell'insieme di attributi `tc-attrs`, è presente un attributo `options` di tipo `sub-message` e con selettore "kind". Per capire quale attributo specifico andrà inserito, è sufficiente andare a cercare nella sezione dei sottomessaggi, il formato dell'attributo `tc-options-msg`, che abbia un valore pari al selettore specificato. Ad esempio, nel caso in cui l'input preveda nell'attributo `tc-attrs` un valore di "kind" pari a "bpf", allora l'attributo `options` conterrà un oggetto il cui insieme di attributi sarà `tc-bpf-ttrs`.

Listato 6.1: Specifica YAML (parziale) di `tc` - parte 1

```yaml
# SPDX-License-Identifier: ((GPL-2.0 WITH Linux-syscall-note) OR BSD-3-
    Clause)

name: tc
protocol: netlink-raw
protonum: 0

doc: Netlink raw family for tc qdisc, chain, class and filter
    configuration over rtnetlink.

definitions:
  -
    name: tcmsg
    type: struct
    members:
      -
        name: family
        type: u8
      -
        name: pad
        type: pad
        len: 3
      -
        name: ifindex
        type: s32
      -
        name: handle
        type: u32
      -
        name: parent
        type: u32
      -
        name: info
        type: u32
[...]
attribute-sets:
  -
    name: tc-attrs
    attributes:
      -
        name: kind
        type: string
      -
        name: options
        type: sub-message
        sub-message: tc-options-msg
        selector: kind
[...]
```

Listato 6.2: Specifica YAML (parziale) di `tc` - parte 2

```yaml
sub-messages:
  -
    name: tc-options-msg
    formats:
      -
        value: basic
        attribute-set: tc-basic-attrs
      -
        value: bpf
        attribute-set: tc-bpf-attrs
[...]
operations:
  enum-model: directional
  list:
    -
      name: newqdisc
      doc: Create new tc qdisc.
      attribute-set: tc-attrs
      fixed-header: tcmsg
      do:
        request:
          value: 36
          attributes: &create-params
            - kind
            - options
            - rate
            - chain
            - ingress-block
            - egress-block
[...]
```

6.3 Introspezione del Kernel

Le potenzialità dello strumento ynl non finiscono qui. Avendo la possibilità di comunicare col kernel mediante Netlink, ynl è un ottimo candidato a diventare il primo strumento con capacità di introspezione del kernel. Questo perchè, finora, non è possibile sapere a priori se un modulo del kernel è compilato o meno. Mediante ynl e la specifica di `tc`, potrebbe essere possibile raggiungere questo obiettivo. Un esempio: se si vuole sapere se il kernel supporta la qdisc TAPRIO, si potrebbe inviare una richiesta Netlink di creazione di una qdisc TAPRIO e poi vedere il messaggio di risposta dal kernel. Esaminando un eventuale messaggio di errore, si può dedurre se il modulo corrispondente alla qdisc (in questo caso `sch_taprio`) è compilato o meno. Il secondo lavoro di questa tesi vuole fare un passo in avanti rispetto a dove si è ora.

6.4 Stato Attuale di ynl e della Specifica YAML di tc

Recentemente, Donald Hunter ha introdotto [16] il tipo `sub-message` assieme alla specifica YAML per `tc`. Come da lui sottolineato, questi cambiamenti mancano del supporto alla codifica degli attributi di tipo `sub-message` e alla risoluzione dei loro selettori quando essi si trovano ad un livello di nidificazione.

6.5 Modifiche Proposte ad ynl

Vista la mancanza del supporto alla codifica dei messaggi, si è deciso di procedere all'implementazione di questa funzionalità. Nei Listati 6.3-6.4 viene riportato il codice Python rilevante per questa funzionalità.

La funzione `_add_attr()` si occupa di codificare gli attributi in messaggi Netlink binari. Come si vede, manca il supporto al tipo `sub-message`. La funzione

Listato 6.3: La funzione `_add_attr`

```
def _add_attr(self, space, name, value):
    try:
        attr = self.attr_sets[space][name]
    except KeyError:
        raise Exception(f"Space '{space}' has no attribute '{name}'")
    nl_type = attr.value
    if attr["type"] == 'nest':
        nl_type |= Netlink.NLA_F_NESTED
        attr_payload = b''
        for subname, subvalue in value.items():
            attr_payload += self._add_attr(attr['nested-attributes'],
                subname, subvalue)
    elif attr["type"] == 'flag':
        attr_payload = b''
    elif attr["type"] == 'string':
        attr_payload = str(value).encode('ascii') + b'\x00'
    elif attr["type"] == 'binary':
        if isinstance(value, bytes):
            attr_payload = value
        elif isinstance(value, str):
            attr_payload = bytes.fromhex(value)
        else:
            raise Exception(f'Unknown type for binary attribute, value: {
                value}')
    elif attr.is_auto_scalar:
        scalar = int(value)
        real_type = attr["type"][0] + ('32' if scalar.bit_length() <= 32
            else '64')
        format = NlAttr.get_format(real_type, attr.byte_order)
        attr_payload = format.pack(int(value))
    elif attr['type'] in NlAttr.type_formats:
        format = NlAttr.get_format(attr['type'], attr.byte_order)
        attr_payload = format.pack(int(value))
    elif attr['type'] in "bitfield32":
        attr_payload = struct.pack("II", int(value["value"]), int(value["
            selector"]))
    else:
        raise Exception(f'Unknown type at {space} {name} {value} {attr["
            type"]}')

    pad = b'\x00' * ((4 - len(attr_payload) % 4) % 4)
    return struct.pack('HH', len(attr_payload) + 4, nl_type) +
        attr_payload + pad
```

Listato 6.4: La funzione `_resolve_selector`

```
def _resolve_selector(self, attr_spec, vals):
    sub_msg = attr_spec.sub_message
    if sub_msg not in self.sub_msgs:
        raise Exception(f"No sub-message spec named {sub_msg} for {
            attr_spec.name}")
    sub_msg_spec = self.sub_msgs[sub_msg]

    selector = attr_spec.selector
    if selector not in vals:
        raise Exception(f"There is no value for {selector} to resolve '{
            attr_spec.name}'")
    value = vals[selector]
    if value not in sub_msg_spec.formats:
        raise Exception(f"No message format for '{value}' in sub-message
            spec '{sub_msg}'")

    spec = sub_msg_spec.formats[value]
    return spec
```

`_resolve_selector()` invece, è responsabile di trovare il valore corretto indicato dal selettore del sotto-messaggio.

Per prima cosa, si è proceduto con l'implementare una ricerca del selettore anche su livelli diversi rispetto a quello dell'attributo. Per fare ciò, si è fatto uso di due funzioni ausiliarie dichiarate nello scope della funzione `_resolve_selector()`: `_find_attr_path()` e `_find_selector()`. `_find_attr_path()` è una funzione ricorsiva che ha il compito di restituire il percorso di "chiavi" che portano al livello dell'attributo. Questo perché l'input dell'utente, che viene passato al programma ynl, è espresso in formato JSON, quindi l'oggetto risultante, rappresentato in questa funzione dalla variabile `vals`, è una struttura dati `dict` (una struttura dati del tipo chiave-valore) in Python. Per percorso di chiavi si intende quindi la successione di chiavi da specificare per poter accedere allo stesso livello dell'attributo. Una volta trovata la lista di chiavi essa è utilizzata per ripercorrere la struttura dati dal livello più interno (lo stesso dell'attributo) a quello più esterno, in cerca del selettore relativo all'attributo. Così facendo, si ha la certezza di trovare il selettore più vicino all'attributo `sub-message` specificato. Entrambe le modifiche sono visibili nel Listato 6.5.

Si è quindi aggiunto il controllo sul tipo dell'attributo nella funzione `_add_attr()`

Listato 6.5: Cambiamenti apportati a `_resolve_selector`

```
-         if selector not in vals:
+
+         def _find_attr_path(attr, vals, path=None):
+             if path is None:
+                 path = []
+             if isinstance(vals, dict):
+                 if attr in vals:
+                     return path
+                 for k, v in vals.items():
+                     result = _find_attr_path(attr, v, path + [k])
+                     if result is not None:
+                         return result
+             elif isinstance(vals, list):
+                 for idx, v in enumerate(vals):
+                     result = _find_attr_path(attr, v, path + [idx])
+                     if result is not None:
+                         return result
+             return None
+
+         def _find_selector_val(sel, vals, path):
+             while path != []:
+                 v = vals.copy()
+                 for step in path:
+                     v = v[step]
+                 if sel in v:
+                     return v[sel]
+                 path.pop()
+             return vals[sel] if sel in vals else None
+
+         attr_path = _find_attr_path(attr_spec.name, vals)
+         value = _find_selector_val(selector, vals, attr_path)
+
+         if value is None:
              raise Exception(f"There is no value for {selector} to
                 resolve '{attr_spec.name}'")
-         value = vals[selector]
+
          if value not in sub_msg_spec.formats:
              raise Exception(f"No message format for '{value}' in sub-
                 message spec '{sub_msg}'")
```

per includere la codifica degli attributi `sub-message`.

Dopo aver effettuato dei test, però, si è scoperto che un altro tipo di attributi mancava del supporto alla codifica: gli attributi `multi-attr`. Gli attributi `multi-attr` sono un tipo particolare di attributi, in quanto possono essere ripetuti più volte nell'input (basti pensare al parametro `sched-entry` di TAPRIO, vedi Tabella 5.1). Si è quindi proceduto ad aggiungere anche questa funzionalità all'interno della medesima patch. Le modifiche sono visibili al Listato 6.6.

Di seguito vengono analizzati i tre pezzi di codice modificati e aggiunti:

1. Aggiunta del parametro `vals`, per poter poi passare l'input dell'utente a `_resolve_selector()`.
2. Gestione degli attributi `multi-attr`, verificando se il valore che si sta codificando attualmente è una lista.
3. Gestione degli attributi `sub-message`.

La patch inviata, oltre a questi cambiamenti, includeva anche altri cambiamenti minori, come la correzione della documentazione di una funzione e l'aggiunta di un attributo `multi-attr` mancante nella specifica di `tc`.

6.6 Review Iniziale alle Modifiche

Dopo aver inviato la patch, si sono susseguite una serie di review da parte della comunità del kernel. La prima è stata da parte di Breno Leitao, il quale ha scritto:

> This is a bit hard to read.
>
> Is it possible to make it a bit easier to read?

a riguardo della gestione degli attributi `multi-attr`. Nonostante una versione più semplificata da leggere sembri ridonante, è più facile da capire e da manutenere, per cui è stata proposta la seguente versione:

Listato 6.6: Cambiamenti apportati a `_add_attr`

```
@@ -449,7 +449,7 @@ class YnlFamily(SpecFamily):
         self.sock.setsockopt(Netlink.SOL_NETLINK, Netlink.
             NETLINK_ADD_MEMBERSHIP,
                              mcast_id)

-    def _add_attr(self, space, name, value):
+    def _add_attr(self, space, name, value, vals):
         try:
             attr = self.attr_sets[space][name]
         except KeyError:
@@ -458,8 +458,13 @@ class YnlFamily(SpecFamily):
         if attr["type"] == 'nest':
             nl_type |= Netlink.NLA_F_NESTED
             attr_payload = b''
-            for subname, subvalue in value.items():
-                attr_payload += self._add_attr(attr['nested-attributes
   '], subname, subvalue)
+            # Check if it's a list of values (i.e. it contains multi-
   attr elements)
+            for subname, subvalue in (
+                ((k, v) for item in value for k, v in item.items())
+                if isinstance(value, list)
+                else value.items()
+            ):
+                attr_payload += self._add_attr(attr['nested-attributes
   '], subname, subvalue, vals)
         elif attr["type"] == 'flag':
             attr_payload = b''
         elif attr["type"] == 'string':
@@ -481,6 +486,12 @@ class YnlFamily(SpecFamily):
             attr_payload = format.pack(int(value))
         elif attr['type'] in "bitfield32":
             attr_payload = struct.pack("II", int(value["value"]), int(
                 value["selector"]))
+        elif attr['type'] == "sub-message":
+            spec = self._resolve_selector(attr, vals)
+            attr_spec = spec["attribute-set"]
+            attr_payload = b''
+            for subname, subvalue in value.items():
+                attr_payload += self._add_attr(attr_spec, subname,
   subvalue, vals)
         else:
             raise Exception(f'Unknown type at {space} {name} {value} {
                 attr["type"]}')
```

Listato 6.7: Modifica della patch per una maggiore leggibilità

```
-         for subname, subvalue in (
-             ((k, v) for item in value for k, v in item.items())
-             if isinstance(value, list)
-             else value.items()
-         ):
-             attr_payload += self._add_attr(attr['nested-attributes'],
    subname, subvalue, vals)
+         if isinstance(value, list):
+             for item in value:
+                 for subname, subvalue in item.items():
+                     attr_payload += self._add_attr(attr['nested-
    attributes'], subname, subvalue, vals)
+         else:
+             for subname, subvalue in value.items():
+                 attr_payload += self._add_attr(attr['nested-attributes
    '], subname, subvalue, vals)
```

Invece, Donald Hunter ha risposto alla patch dicendo:

> I have a longer patchset that covers this plus some refactoring for nested struct definitions and a lot of addtions to the tc spec. Do you mind if I post it and we review to see if there is anything from your patchset that is missing from mine?

La patch, quindi, non è stata accettata, in attesa della pubblicazione dei cambiamenti da parte di Donald Hunter.

6.7 Aggiornamento della Patch dopo la Review

Una volta pubblicata la patch da parte di Donald Hunter [1], ed aver discusso sulla soluzione migliore da intraprendere, erano presenti dei cambiamenti precedentemente pubblicati non affrontati da lui. Si è quindi deciso di inviare una patch aggiuntiva, basandosi sugli ultimi sviluppi. La patch finale è visibile nel Listato 6.8.

Quest'ultima modifica tiene conto sia degli sviluppi effettuati da Donald Hunter, sia del controllare che l'attributo sia effettivamente del tipo `multi-attr` prima

[1] https://lore.kernel.org/all/20240129223458.52046-1-donald.hunter@gmail.com/

Listato 6.8: La patch finale per ynl

```
diff --git a/tools/net/ynl/lib/ynl.py b/tools/net/ynl/lib/ynl.py
index 0f4193cc2e3b..3f5a7d5388a9 100644
--- a/tools/net/ynl/lib/ynl.py
+++ b/tools/net/ynl/lib/ynl.py
@@ -444,6 +444,13 @@ class YnlFamily(SpecFamily):
         except KeyError:
             raise Exception(f"Space '{space}' has no attribute '{name
                 }'")
         nl_type = attr.value
+
+        if attr.is_multi and isinstance(value, list):
+            attr_payload = b''
+            for subvalue in value:
+                attr_payload += self._add_attr(space, name, subvalue,
    search_attrs)
+            return attr_payload
+
         if attr["type"] == 'nest':
             nl_type |= Netlink.NLA_F_NESTED
             attr_payload = b''
```

di processarlo. Questo controllo era stato erroneamente associato agli attributi di tipo `nest`, ma poi, come evidenziato da Jakub Kicinski in una delle sue revisioni, è stato cambiato e inserito come controllo a sé stante.

Capitolo 7

Risultati

7.1 Test dei Cambiamenti Effettuati su TAPRIO

Prima di inviare la patch contenenti le modifiche effettuate alla qdisc TAPRIO, si sono svolti alcuni test, per verificare che i cambiamenti introdotti non modifichino il comportamento normale. Si è quindi utilizzata la suite di testing specifica per `tc`, chiamata `tdc.py`, presente in `/tools/testing/selftests/tc-testing`.

`tdc.py` è uno script scritto in Python per fare unit testing. È possibile specificare una categoria per restringere i test ad esempio ad una sola qdisc. Questo è ciò che è stato fatto per eseguire solo i test rilevanti per la qdisc TAPRIO.

Il comando per eseguire i test è il seguente (dopo essersi posizionati nella cartella dove si trova lo script): `./tdc.py -c taprio`. Di seguito si riportano i risultati dei test, tutti con esito positivo:

```
All test results:


1..11
ok 1 ba39 - Add taprio Qdisc to multi-queue device (8 queues)
ok 2 9462 - Add taprio Qdisc with multiple sched-entry
ok 3 8d92 - Add taprio Qdisc with txtime-delay
ok 4 d092 - Delete taprio Qdisc with valid handle
ok 5 8471 - Show taprio class
```

```
ok 6 0a85 - Add taprio Qdisc to single-queue device
ok 7 3e1e - Add taprio Qdisc with an invalid cycle-time
ok 8 39b4 - Reject grafting taprio as child qdisc of software taprio
ok 9 e8a1 - Reject grafting taprio as child qdisc of offloaded taprio
ok 10 a7bf - Graft cbs as child of software taprio
ok 11 6a83 - Graft cbs as child of offloaded taprio
```

7.2 Esito della Patch Relativa a TAPRIO

Dopo aver ricevuto due review a riguardo di alcuni dettagli implementativi, la patch è stata accettata con successo [1].

7.3 Risultati dei Cambiamenti Effettuati su ynl

L'aggiunta delle due funzionalità ad ynl, ha permesso di fare un ulteriore passo in avanti nell'introspezione del kernel, rendendo possibile l'aggiunta di qdisc mediante questo strumento. Di seguito due esempi con due qdisc differenti:

- Aggiungere una qdisc taprio:

```
# ./tools/net/ynl/cli.py --spec Documentation/netlink/specs/tc.yaml \
  --do newqdisc --create --json '{
"family":1, "ifindex":4, "handle":65536, "parent":4294967295, "info":0,
 "kind":"taprio",
 "stab":{
     "base": {
       "cell-log": 0,
       "size-log": 0,
       "cell-align": 0,
```

[1]https://git.kernel.org/netdev/net-next/c/0efc7e541fd5

```
        "overhead": 31,
        "linklayer": 0,
        "mpu": 0,
        "mtu": 0,
        "tsize": 0
      }
   },
   "options":{
      "priomap": {
          "num-tc": 3,
          "prio-tc-map": "01010101010101010101010101010101",
          "hw": 0,
          "count":
"0100010002000000000000000000000000000000000000000000000000000000",
          "offset":
"0100020003000000000000000000000000000000000000000000000000000000"
      },
      "sched-clockid":11,
      "sched-entry-list": {"entry": [
          {"index":0, "cmd":0, "gate-mask":1, "interval":300000},
          {"index":1, "cmd":0, "gate-mask":2, "interval":300000},
          {"index":2, "cmd":0, "gate-mask":4, "interval":400000} ]
      },
      "sched-base-time":1528743495910289987, "flags": 1
   }
}'
```

- Aggiungere una qdisc ets

```
# ./tools/net/ynl/cli.py --spec Documentation/netlink/specs/tc.yaml \
```

```
--do newqdisc --create --json '{
 "family":1, "ifindex":4, "handle":65536, "parent":4294967295,
 "kind":"ets",
 "options":{
     "nbands":6,
     "nstrict":3,
     "quanta":{
         "quanta-band": [3500, 3000, 2500]
     },
     "priomap":{
         "priomap-band":[0, 1, 1, 1, 2, 3, 4, 5]
     }
 }
}'
```

Queste modifiche aprono la strada ad ulteriori sviluppi riguardo questa tematica, rendendo possibile la creazione di uno strumento in grado di interrogare il kernel in maniera esaustiva.

7.4 Esito della Patch Relativa a ynl

L'ultima versione della patch ha ricevuto review positive da un kernel developer e da un maintainer del sottosistema di networking, e, al momento della scrittura, è in attesa di essere inclusa ufficialmente nel kernel.

Capitolo 8

Conclusioni

In conclusione, questa tesi ha prodotto risultati concreti con l'inclusione di due patch nel kernel Linux. L'obiettivo principale era quello di approfondire la conoscenza di alcuni dei funzionamenti interni del kernel, per poi applicarla allo sviluppo di una soluzione per l'introspezione dello stesso.

L'esperienza con un progetto open source di tale portata, unita all'interazione con gli sviluppatori e i maintainer del kernel, ha contribuito al perfezionamento delle proposte di modifica, risultando in un lavoro di maggiore qualità conforme agli standard richiesti.

Questo lavoro rappresenta un progresso significativo verso la possibilità di interrogare il kernel, ma non costituisce un punto di arrivo; piuttosto, è un punto di partenza. Basandosi su questo lavoro, si potranno sviluppare ulteriori soluzioni per raggiungere, finalmente, una completa introspezione del kernel.

Appendice A

Richiamo: Socket BSD

I socket BSD, conosciuti anche con il nome di socket Berkeley, sono delle API (Application Programming Interface) per socket internet. Permettono la comunicazione tra una o più macchine Essi mettono a disposizione diverse funzioni di API. Le essenziali sono mostrate nella Tabella A.1 [17].

Funzione	Descrizione
`accept`	Accetta una richiesta di connessione per un socket in stato di ascolto
`bind`	Assegna un indirizzo locale ad un socket
`closesocket`	Chiude un socket esistente e rilascia il socket descriptor
`connect`	Stabilisce una connessione tra due socket
`listen`	Mette il socket in modalità di ascolto
`recv`	Riceve dati in arrivo al socket
`send`	Invia dati su un socket
`socket`	Crea un socket di comunicazione

Tabella A.1: Funzioni essenziali dei socket BSD

I socket BSD si basano sul modello client/server, che utilizza a sua volta un protocollo tra TCP e UDP per stabilire una connessione tra due macchine.

Nel dettaglio, il server crea un socket, utilizza `bind()` per assegnare una porta al socket e successivamente si mette in ascolto con `listen()`. Questo permette al socket di ricevere richieste di connessione. Dopodiché, viene chiamata la funzione `accept()`, che blocca il socket fino a quando non viene ricevuta una richiesta di connessione. Una volta stabilita la connessione, `accept()` crea un nuovo socket, che verrà utilizzato

per ricevere dati fino a quando la connessione non viene chiusa da parte del client. Il client crea un `socket()` e utilizza `connect()` per stabilire una connessione TCP. Poi, viene utilizzata la funzìone `send()` per inviare i dati al server. Notare che in questo caso `bind()` non viene mai utilizzata, perché la porta verrà scelta in maniera casuale. Per terminare la comunicazione, il client utlizza infine `closesocket()`.

Bibliografia e sitografia

[1] The kernel development community. *Kernel subsystem documentation*. 2023. URL: `https://docs.kernel.org/subsystem-apis.html` (visitato il 12/01/2024).

[2] Rami Rosen. *Linux Kernel Networking: Implementation and Theory*. 1st. USA: Apress, 2013. ISBN: 143026196X.

[3] Matt Conran. *Linux Networking Subsystem*. 2016. URL: `https://network-insight.net/2016/04/07/linux-networking-subsystem/` (visitato il 15/01/2024).

[4] Bert Hubert. *tc(8) — Linux manual page*. 2001. URL: `https://man7.org/linux/man-pages/man8/tc.8.html` (visitato il 13/01/2024).

[5] Bert Hubert et al. *Linux Advanced Routing and Traffic Control HOWTO*. 2012. URL: `https://lartc.org/howto/` (visitato il 13/01/2024).

[6] Andi Kleen et al. *Linux Netlink as an IP Services Protocol*. RFC 3549. Lug. 2003. DOI: `10.17487/RFC3549`. URL: `https://www.rfc-editor.org/info/rfc3549`.

[7] Gautam Bhanage. *Linux: Comparison of Netlink vs ioctl*. 2020. URL: `https://www.bhanage.com/2020/11/linux-comparison-of-netlink-vs-ioctl.html` (visitato il 30/01/2024).

[8] Pablo Ayuso, Rafael Gasca e Laurent Lefèvre. «Communicating between the kernel and user-space in Linux using Netlink sockets». In: *Softw., Pract. Exper.* 40 (ago. 2010), pp. 797–810. DOI: 10.1002/spe.981.

[9] John Fastabend. *tc-mqprio(8) — Linux manual page*. 2013. URL: https://man7.org/linux/man-pages/man8/mq-taprio.8.html (visitato il 01/02/2024).

[10] Maxime Chevallier. *Multi-queue improvements in Linux kernel Ethernet driver mvneta*. 2022. URL: Multi-queue%20improvements%20in%20Linux%20kernel%20Ethernet%20driver%20mvneta (visitato il 30/01/2024).

[11] Vinicius C. Gomes. *tc-taprio(8) — Linux manual page*. 2018. URL: https://man7.org/linux/man-pages/man8/tc-taprio.8.html (visitato il 01/02/2024).

[12] «IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks». In: *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (2018), pp. 1–1993. DOI: 10.1109/IEEESTD.2018.8403927.

[13] Jakub Kicinski. *YAML netlink protocol descriptions (RFC)*. 2022. URL: https://lore.kernel.org/netdev/YvjqLDnWUONvnv3E@shredder/t/ (visitato il 02/02/2024).

[14] The kernel development community. *Netlink protocol specification in (YAML)*. 2023. URL: https://docs.kernel.org/userspace-api/netlink/specs.html (visitato il 02/02/2024).

[15] The kernel development community. *Netlink specification support for raw Netlink families*. 2023. URL: https://docs.kernel.org/userspace-api/netlink/netlink-raw.html (visitato il 02/02/2024).

[16] Donald Hunter. *Add 'sub-message' support to ynl*. 2023. URL: https://lore.kernel.org/all/20231215093720.18774-1-donald.hunter@gmail.com/ (visitato il 01/02/2024).

[17] Arm Limited. *BSD Socket functions and communication flow*. 2022. URL: https://www.keil.com/pack/doc/mw/Network/html/group__using__network__sockets__bsd__func.html (visitato il 18/01/2024).

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

© 2024 Alessandro Marcolini