Traffic Stats: QoS Adattivo con nftables, TC-Cake e TimescaleDB

Rete congestionata, latenza alle stelle e router che fanno onco

Avete presente quando sulla vostra rete locale qualcuno avvia un backup sul cloud, o mette a scaricare un file pesante, e improvvisamente la vostra chiamata VoIP gracchia, i DNS rispondono dopo secondi e il ping sui giochi online schizza a 600ms? Ebbene, quel fenomeno fastidioso si chiama **bufferbloat**, ed è dovuto al fatto che i router commerciali accumulano pacchetti nei loro buffer interni fino a scoppiare, invece di scartarli o smistarli con intelligenza.

La cura standard per questo male è applicare delle code di QoS (Quality of Service) per limitare la banda. Ma se avete una connessione instabile, come una FWA o una SIM 4G/5G dove la velocità reale cambia continuamente in base al meteo e al carico della cella, una configurazione statica non serve a nulla.

Per risolvere questa problematica è stato implementato un sistema di **QoS Adattivo** che misura con continuità la latenza della connessione e modula dinamicamente le classi di priorità e i limiti di banda del firewall.

Di seguito viene illustrata la configurazione di basso livello di nftables, TC-Cake e del database temporale TimescaleDB, ottimizzata per minimizzare l'I/O storage sul server.

Il repository con il codice sorgente e gli script di automazione è disponibile su:
https://git.dreadful.work/errno0x0d/traffic-stats (https://git.dreadful.work/errno0x0d/traffic-stats)

L'Architettura

La baracca è divisa su due nodi per evitare di appesantire il firewall con database o interfacce grafiche:

1. **Il Firewall**: Si occupa del lavoro sporco. Rileva i contatori dei singoli IP della rete locale via nftables, misura l'RTT (latenza) verso internet, adatta le code di shaping e invia i log al database.
2. **Il Server centralizzato (`antares`)**: Ospita il database TimescaleDB (PostgreSQL con estensioni per serie storiche) in un container Docker ed espone una dashboard web Flask con grafici in tempo reale.

Le Interfacce di rete (e come non perdersi tra i nomi di sistema)

Se andate a leggere i file di configurazione, vi troverete davanti a una sbrodolata di nomi fisici come `enp1s0`, `enp9s0` e così via. Vediamo di fare ordine spiegando cosa rappresenta ogni interfaccia del nostro sistema:

- **`wan` (nel nostro caso `enp1s0`)**: È l'interfaccia fisica del firewall collegata direttamente alla rete esterna (internet) tramite il modem dell'ISP. È qui che applichiamo lo shaping in upload.
- **`lan_trust` (nel nostro caso `enp9s0`)**: È la nostra rete locale principale e fidata. È da qui che provengono i dispositivi di cui vogliamo monitorare il consumo di banda.
- **`fe` (`enp10s0`)** e **`be` (`enp8s0`)**: Interfacce fisiche per le DMZ rispettivamente di Frontend e Backend.
- **`wg0` / `wt0`**: Interfacce virtuali dedicate alle VPN (WireGuard/ NetBird).
- **`ifb0` (Intermediate Functional Block)**: Questa è un'interfaccia virtuale fondamentale. Linux, per sua natura, non permette di fare lo shaping (ovvero limitare la banda) sul traffico che entra (ingress) in modo efficiente. Per aggirare questo limite, creiamo `ifb0`, ci reindirizziamo dentro tutto il traffico che entra da `wan` (`enp1s0`), e poi applichiamo le regole di shaping del download su `ifb0`.

Il setup del Firewall: TC-Cake e nftables

1. Tagging DSCP con nftables

Prima di limitare la banda, dobbiamo dire a Linux quali pacchetti sono importanti e quali possono aspettare. Lo facciamo impostando i contrassegni DSCP (DiffServ) nel firewall:

```
table ip filter {
    chain forward {
        type filter hook forward priority filter; policy accept;

        # Priorità assoluta al VoIP (il nostro PBX locale)
        ip saddr 192.168.152.3 ip dscp set ef
        ip daddr 192.168.152.3 ip dscp set ef

        # Traffico interattivo e DNS prioritario
        tcp dport 22 ip dscp set cs2
        udp dport 53 ip dscp set cs6

        # Traffico di basso profilo (download di bulk o IP non prioritari)
        ip saddr { 192.168.152.60, 192.168.152.61 } ip dscp set cs1
    }
}
```

2. Le regole di TC (Traffic Control)

Il demone in Python `net-admin` esegue questi comandi al boot per inizializzare le code di shaping basate sull'algoritmo **Cake**:

```
# Attiva l'interfaccia IFB per il download
ip link set dev ifb0 up

# Pulisce eventuali vecchie configurazioni sulla WAN fisica
tc qdisc del dev enp1s0 root 2>/dev/null
tc qdisc del dev enp1s0 ingress 2>/dev/null

# Configura Cake in Upload sulla WAN
tc qdisc add dev enp1s0 root cake bandwidth 45Mbit besteffort triple-ack ack-
filter nat rtt 100ms

# Specchia il traffico in ingresso dalla WAN su ifb0
tc qdisc add dev enp1s0 handle ffff: ingress
tc filter add dev enp1s0 parent ffff: protocol all u32 match u32 0 0 action
```

```
mirred egress redirect dev ifb0

# Configura Cake in Download su ifb0
tc qdisc del dev ifb0 root 2>/dev/null
tc qdisc add dev ifb0 root cake bandwidth 220Mbit besteffort triple-ack wash
nat rtt 100ms
```

Il demone in esecuzione calcola costantemente la latenza: se rileva congestione, aggiorna i limiti di banda (`bandwidth XXXMbit`) su `enp1s0` e `ifb0` riducendoli fino a trovare un punto di stabilità.

Come non friggere la scheda SD o il disco del server

Se avete installato il database su una schedina SD o su un SSD di poco conto, scrivere lo storico del traffico di ogni singolo IP locale ogni 10 secondi significa mandare il disco all'altro mondo in pochi mesi a causa dell'usura da riscrittura.

Per salvare la baracca abbiamo implementato due modifiche critiche.

Ottimizzazione 1: synchronous_commit = off

Nel `docker-compose.yml` di TimescaleDB su `antares`, abbiamo disattivato la scrittura sincrona dei commit sul disco fisico:

```
services:
  timescaledb:
    image: timescale/timescaledb:latest-pg16
    container_name: traffic_timescaledb
    command: ["postgres", "-c", "synchronous_commit=off"]
```

Con `synchronous_commit=off`, PostgreSQL conferma la scrittura al client non appena i dati sono nella memoria RAM, riversandoli sul disco in blocchi cumulativi ogni pochi secondi. Se salta la corrente si perdono gli ultimi istanti di log (irrilevante per delle statistiche di traffico), ma l'usura dell'SSD crolla drasticamente.

Ottimizzazione 2: Aggregazione a mezzanotte e retention di 15 giorni

Mantenere dati millimetrici registrati ogni 10 secondi per mesi non ha senso. I log grezzi in tabella `device_logs` vengono tenuti solo per 15 giorni. Ogni notte a mezzanotte, uno script Python (`daily_aggregate.py`) aggrega i byte totali consumati da ciascun IP durante la giornata e scrive il riassunto nella tabella `device_daily_stats`.

La pulizia dei vecchi log grezzi su TimescaleDB non viene fatta tramite una lenta `DELETE` (che bloccherebbe la tabella e farebbe salire l'I/O del disco a mille), ma sfruttando l'eliminazione diretta delle partizioni temporali (i chunks) tramite TimescaleDB:

```
SELECT drop_chunks('device_logs', INTERVAL '15 days');
```

In questo modo la tabella dei log rimane snella e le query della dashboard Flask rispondono in meno di 50ms anche su hardware molto limitato.

Fluxer Icon

Canale di discussione e supporto

Canale di discussione e supporto. Per confrontarti su configurazioni o altro entra su fluxer.dreadful.work (https://fluxer.dreadful.work/invite/dreadful).