

# Gestione di Reti

Federico Matteoni

A.A. 2019/20



# Indice

0.1	Introduzione . . . . .	5
0.1.1	Terminologia e concetti fondamentali . . . . .	5
0.1.2	Ethernet . . . . .	6
0.1.3	Pacchetti . . . . .	6
0.1.4	FCAPS . . . . .	7
0.1.5	Interagire con management objects . . . . .	7
0.1.6	Servizi . . . . .	7
0.1.7	Standardizzazione . . . . .	8
0.2	Abstract syntax notation one . . . . .	9
0.2.1	Common problems with packet capture . . . . .	9
0.2.2	Interoperabilità . . . . .	10
<b>1</b>	<b>Gestione di Rete</b>	<b>13</b>
1.1	Nascita . . . . .	13
1.2	Gestione di Rete Internet . . . . .	13
1.3	SNMP . . . . .	13
1.3.1	Trap Directed Polling . . . . .	14
1.3.2	MIB . . . . .	16
1.3.3	Primitive . . . . .	18
1.3.4	SNMPv2 . . . . .	20
1.3.5	SNMPv3 . . . . .	20
1.3.6	Bridge MIB . . . . .	22
<b>2</b>	<b>Rete Cellulare</b>	<b>23</b>
<b>3</b>	<b>RRDtool</b>	<b>25</b>
3.1	RRDtool Database . . . . .	25
<b>4</b>	<b>Cattura</b>	<b>29</b>
4.1	libpcap . . . . .	30
<b>5</b>	<b>Monitoring</b>	<b>31</b>
5.1	Monitoring requirements . . . . .	31
5.2	Vari attori, varie metriche . . . . .	31
5.3	Applicazioni di analisi del traffico . . . . .	31
5.3.1	Richieste . . . . .	31
5.3.2	Ulteriori problemi . . . . .	32
<b>6</b>	<b>Benchmarking per dispositivi interconnessi</b>	<b>33</b>
<b>7</b>	<b>Deep Packet Inspection</b>	<b>35</b>
7.1	Calcolo delle serie temporali . . . . .	37
7.1.1	Timeseries, Forecast, Anomaly Detection . . . . .	38
<b>8</b>	<b>Remote Monitoring</b>	<b>43</b>
8.1	RMON . . . . .	43
8.2	Real-Time Flow Measurement . . . . .	44
8.3	Monitoraggio a flussi . . . . .	44

<b>9 ScaPy</b>	<b>47</b>
<b>10 Flussi</b>	<b>49</b>
10.0.1 NetFlow . . . . .	49
10.0.2 IPFIX . . . . .	52
10.0.3 Flow Aggregation e Filtering . . . . .	52
10.0.4 sFlow . . . . .	53
10.0.5 Radius . . . . .	54
<b>11 Data Structures</b>	<b>55</b>
<b>12 Network Traffic Monitoring su Container</b>	<b>59</b>
12.1 eBPF . . . . .	59

## 0.1 Introduzione

**Perché bisogna studiare la gestione?** La situazione corrente comprende: un **aumento delle risorse strategiche informative**, le reti di computer che da strumento di supporto sono diventate **elemento chiave** delle organizzazioni, l'**aumento esponenziale** dei dispositivi interconnessi e aumento anche **della complessità** e delle funzionalità.

C'è quindi la richiesta di servizi di rete permanenti e di qualità ottimale, oltre alla necessità di ridurre i costi per le infrastrutture di rete di un'azienda.

**Necessità** Gestione di reti eterogenee con l'aiuto dei computer.

### 0.1.1 Terminologia e concetti fondamentali

**Managed Objects** Il controllo, la coordinazione e il monitoraggio delle risorse avviene tramite la manipolazione dei cosiddetti **managed objects**: un MO è una **visione astratta di una risorsa che presenta le proprietà dal punto di vista della gestione**. Sono **rappresentazioni astratte di risorse reali**.

I confini di un MO specificano quali dettagli sono accessibili ai sistemi di monitoraggio e quali sono schermati (**black box**)

Management System  $\leftrightarrow$  Managed Object  $\leftrightarrow$  Real Object

#### Caratteristiche

**Attributi:** descrivono lo stato/condizione dell'MO, possono cambiare quando cambia lo stato dell'oggetto reale e possono essere manipolati attraverso operazioni di management

**Operazioni:** consentono l'accesso all'MO. Operazioni tipiche sono get, set, create e delete, ma il numero e tipo delle operazioni influenzano performance e complessità dell'oggetto

**Comportamento:** determina la semantica e l'interazione con la risorsa reale. Normalmente definito in linguaggio naturale

**Notifiche:** quantità e tipologia dei messaggi, che possono essere generati da situazioni pre-definite da un MO quando avviene una specifica situazione

**Management Information Base** L'unione di tutti i MO contenuti in un sistema forma la MIB del sistema. La **Management Information Base** è la collezione di tutti i management object all'interno del sistema, con i loro attributi.

Una MIB deve essere conosciuta sia da chi la implementa che da chi la gestisce.

**Modularità** Gli MO di un sistema sono solitamente definiti in più MIB. Nelle MIB sono introdotti i moduli per consentire un design modulare: moduli diversi possono essere definiti da team diversi, le funzionalità di gestione possono essere estese e modificate...

#### Paradigma Gestore/Agente Agent

Implementa i MIB delle MO accedendo alle risorse reali

Riceve le richieste da un gestore, le processa e trasmette le risposte appropriate

Smista le notifiche riguardanti cambiamenti di stato importanti nel MIB

Protegge gli MO da accessi non autorizzati usando regole di controllo degli accessi e autenticazione della comunicazione

#### Manager

Esercita il controllo delle funzioni

Avvia operazioni di gestione tramite opportune operazioni del protocollo per la manipolazione degli MO

Riceve messaggi dagli agenti e li inoltra alle applicazioni interessate per la gestione

**Management Protocol** Un protocollo di gestione implementa l'accesso a MO distanti attraverso la codifica di dati di gestione (management data)

**Livello 2** Consente di identificare un device sulla rete. In tutte le reti c'è la necessità di identificare la porta di rete. Ogni dispositivo ha almeno un'interfaccia di rete: loopback, che consente di far comunicare processi di rete sulla stessa macchina. 127.0.0.1 consente di parlare su stessa macchina senza trasmettere sul filo, fondamentalmente un cortocircuito.

`ifconfig` consente di vedere le interfacce di rete disponibili su unix.  
Se si vuole gestire una rete è fondamentale la standardizzazione.

Output `ifconfig`. Parte degli indirizzi, no indirizzo hw su loopback perché il traffico non esce mai (loopback sulla pila OSI è nel livello 3 Network, il MAC address è sul livello 2 Data Link, che non viene toccato da loopback). Indirizzo MAC 6 byte divisi in blocchi dai due punti. I primi 3 identificano il costruttore della scheda di rete. I successivi tre identificano la scheda di rete per il costruttore, che lo setta univocamente. Ciò garantisce univocità. Per primo blocco di tre ho 16M di dispositivi possibili. I MAC address quindi **non sono univoci**, lo sono *probabilmente*. L'univocità è fondamentale sulla stessa rete. Quindi indirizzo hw identifica univocamente device sulla rete locale. divisi in due blocchi, il primo identifica costruttore della scheda di rete.

Qualsiasi dispositivo ha indirizzo hw diverso per ciascuna porta.

## 0.1.2 Ethernet

L'ethernet è un cavo seriale: trasmissione e ricezione a mezzo seriale. **Un filo.**

Quando si mandano dati non si possono mandare tutti insieme, ma bisogna mandarli un po' per volta. Non c'è collisione perché la ricezione e la trasmissione sono su due fili separati.

I pacchetti vengono inviati nel tempo sul filo, vengono distinti fra loro dal **preamble**, un'intestazione. I pacchetti sono inviati con una direzione: preambolo, destinazione, sorgente, tipo dei dati, dati effettivi, padding (per rendere pacchetto di 64 se pacchetto è troppo corto), CRC.

Quindi per spedire un pacchetto necessito degli indirizzi (mittente e destinatario) e dei dati da mandare. Chi sono lo so, è scritto nella mia scheda, ma devo conoscere l'indirizzo destinatario.

Alla connessione del cavo, se ho impostato il DHCP allora manda fuori un pacchetto per la richiesta DHCP, quindi lo switch impara il mio indirizzo. Se ho impostato l'IP statico, manda un pacchetto ARP quindi lo switch lo impara lo stesso.

Il MAC address può essere randomizzato per privacy, e sui dispositivi mobili lo è spesso e volentieri.

Può essere possibile avere più di un utente sulla stessa rete con i soliti indirizzi, apparati avanzati se ne possono accorgere.

## 0.1.3 Pacchetti

Ogni pacchetto è interamente creato dal computer, quindi "*non ci si può fidare*".

Bisogna andare a livello fisico e **autenticare**, un po' come chiedere la carta d'identità. Metter in atto **meccanismi che impediscano di inibire riconoscimento della sorgente**.

802.1x permette di entrare in rete. Se configurato, il device prima di entrare in rete espone delle credenziali (utente, password, protocollo autenticazione...). Da quel momento in poi **allegato ad ogni pacchetto c'è il mio nome**, ma **le informazioni di autenticazione non fanno parte del pacchetto**: i formati dei pacchetti sono stati creati quando non c'era preoccupazione e interesse in fattori di sicurezza delle trasmissioni. Il riconoscimento del device è eseguito in qualche altro modo e **rimane nel device** (Access Point). Ciò **non serve per autenticazione fisica sul cavo** (mezzo unico per un utente, so che sei tu) ma è necessaria per autenticazione su mezzi condivisi (ad esempio, wireless).

Sui router il MAC cambia ad ogni hop (perché l'ethernet è un protocollo di comunicazione punto-punto), l'IP cambia solo se c'è NAT. Le parti dal livello 3 in su, a meno di frammentazione, non cambiano.

La robustezza delle reti dipende dalla ridondanza. Un modo tipico è mettere più strade per spedire il traffico: **load balancing**.

Vale sia per corrente elettrica che per traffico di rete.

### 0.1.4 FCAPS

Aree funzionali **per gestire qualsiasi sistema**, da giochi a sistemi di rete. Non sono mutualmente indipendenti.

**Fault Management:** un fault è un evento che ha molta rilevanza, e l'obiettivo è l'error detection, isolation, repair e log.

Se qualcuno rileva un malfunzionamento (riempito disco, ram, sovraccarico CPU...) lo deve **notificare**.

**Configuration Management:** sapere com'è configurato il sistema. Leggere la configurazione è importante, così che le app si possano basare sulle API comuni e funzionare correttamente. Fondamentale capire la configurazione perché permette di definire l'amministrazione, servizi..., possibile riconoscere anche le adiacenze e *"questo filo qui va su questa porta qua"*, che impatto ho se stacco questo cavo, o si rovina? Informazioni sufficienti per amministrare la rete.

Quindi raccogliere e conservare informazioni sulle configurazioni dei dispositivi, per poter semplificare la loro configurazione, tenere traccia delle modifiche fatte alla configurazione e pianificare espansioni e scalabilità futura.

**Account Management:** l'obiettivo è rilevare il consumo di risorse e statistiche di utilizzo per utente. Questo è utile sia per tenere traccia dell'utilizzo della rete sia per poter tassare gli utenti in base al loro utilizzo della rete. Ma anche in assenza di questa necessità, tenere traccia dell'utilizzo di ogni utente è utile.

**Performance Management:** efficienze e statistiche, performance di sistema sia lato utente sia lato fornitore. Per l'utente è riuscire ad usare la rete, per l'operatore è il giusto compromesso tra investimento sul mezzo e contentezza utente. L'obiettivo è assicurare che le performance di rete rimangano accettabili, solitamente attraverso SNMP e monitoraggio attivo, oppure anche configurando alert quando si verificano problemi.

**Security Management:** controllare l'accesso alla rete, autoproteggendosi perché con le reti odierne posso intasare rete (volente o no) e quindi intasare internet, provocando danni anche ingenti. Verificare, autenticare, crittografare...

Importante anche raccogliere informazioni di sicurezza e analizzarle periodicamente.

### 0.1.5 Interagire con management objects

Primitive: **get, set, create, delete**

Quando faccio richiesta ad un protocollo mi aspetto una risposta: richiesta → risposta

Il contenuto della richieste varia durante il transito aggiungendo determinate informazioni, per esempio: ad un SMS durante il transito si aggiunge numero mittente per poter comunicare a destinatario chi inviava.

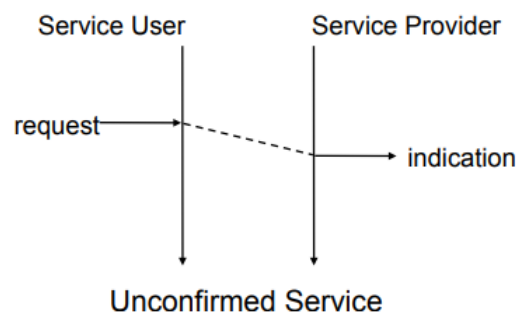
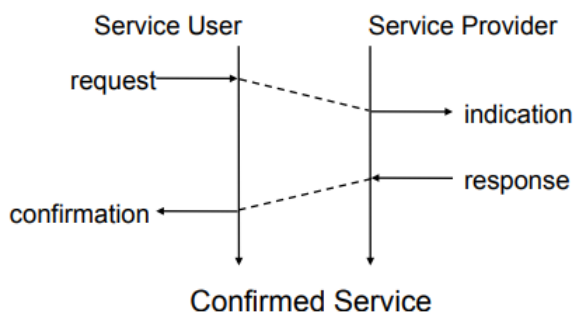
### 0.1.6 Servizi

**Confermati** Faccio richiesta → mi aspetto risposta.

Es: Telegram/Whatsapp

**Non confermati** Faccio richiesta e fine.

Es: SMS



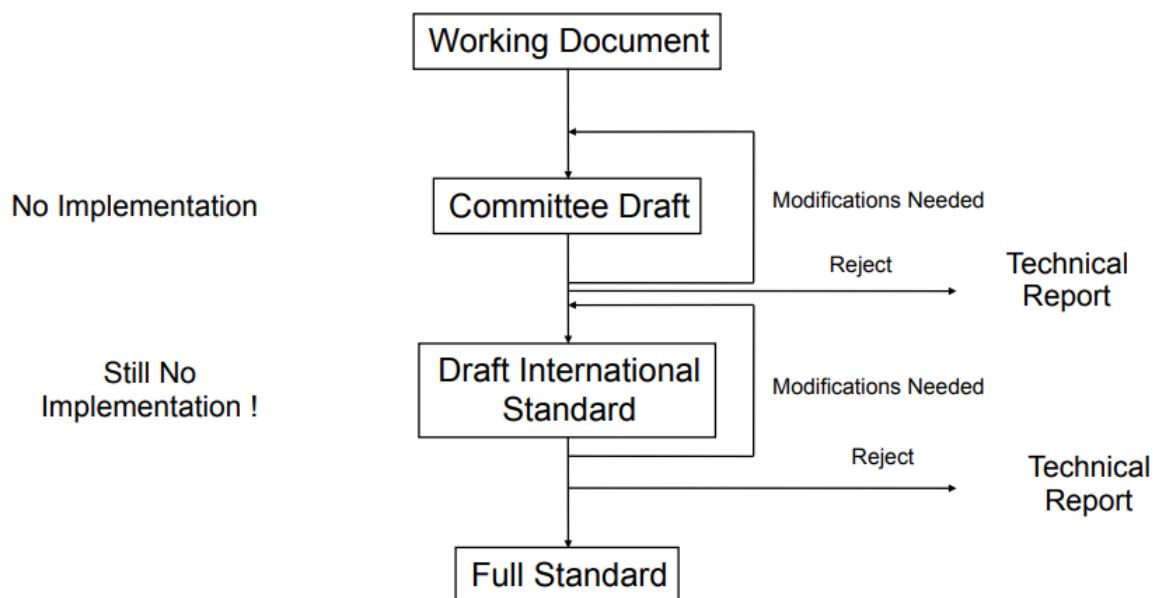
### 0.1.7 Standardizzazione

La grande differenza tra ISO/OSI e Internet è il processo di standardizzazione.

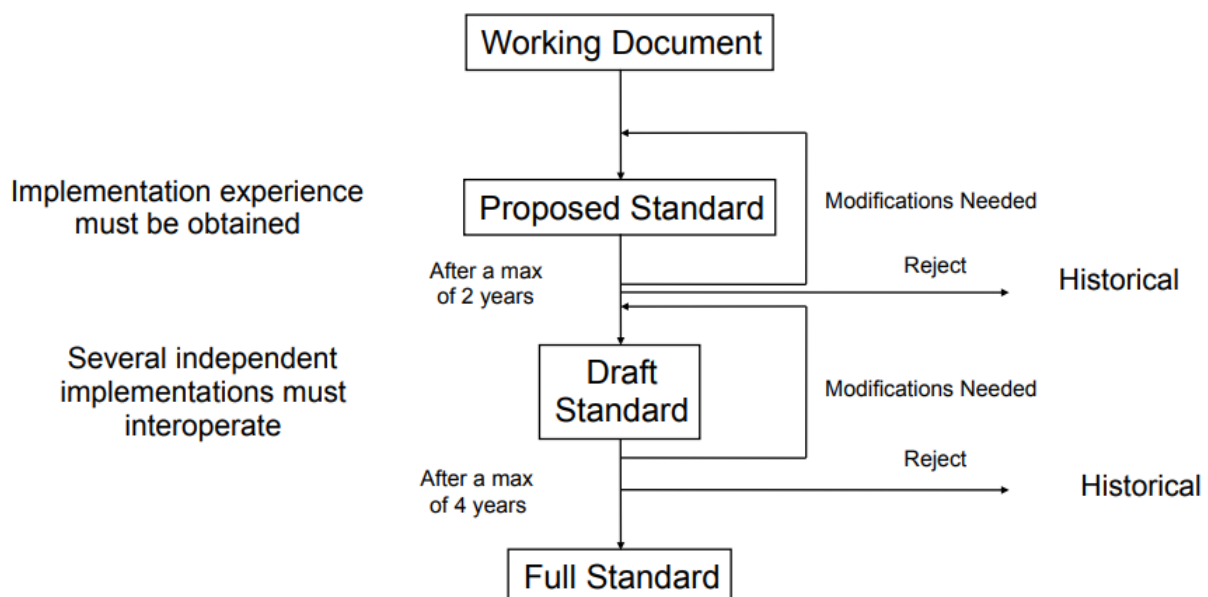
**ISO** Nella standardizzazione ISO tutte le varie aziende si accordano su come fare la rete: si creano gruppi di lavoro che si riuniscono (es. ICANN) e producono un documento di lavoro, poi vari comitati lo discutono (fino a qui **ad alto livello**).

Se si mettono d'accordo, pubblicano un **Draft International Standard** (senza implementazione). Dopodiché, se il draft non viene accettato creano un **technical report**, altrimenti si produce il **full standard**.

La parte importante è l'**assenza di implementazione fino alla creazione dello standard**.



**Internet** Nella standardizzazione internet è tutto seguito dal comitato IETF, che pubblica aree d'interesse in cui ritiene ci sia possibilità di sviluppo. La sottomissione di un'idea ad un'area d'interesse è libera, oppure si può mandare una mail per idee completamente nuove. Dal working document a draft passa poco e i draft dopo pochi mesi scadono. Dopo massimo due anni o si rifiuta o si fa il draft standard (draft RFC) che o lo si rifiuta o diventa standard in max 4 anni. Necessita di più implementazioni interoperabili.





## 0.2 Abstract syntax notation one

**ASN1** Sintassi per la definizione di strutture dati e formato di messaggi. Ha l'obiettivo di consentire a macchine dalle differenti architetture hardware di scambiare dati, essere language neutral e consentire la negoziazione della codifica di trasmissione.

Come spostare le informazioni? Vari costruttori all'inizio lo facevano "in casa" senza interoperabilità. Col tempo si è reso necessario costruire qualcosa per scambiare le informazioni in maniera interoperabile.

**Endian** Come si ordinano i dati in spedizione, per sapere qual è il più significativo. Bit più significativo a sx è big-endian, ormai poco usato. Altrimenti è little-endian.

Nel monitorare il traffico di rete c'è il problema di come riceverlo. Non sempre siamo nel posto giusto. Se voglio vedere cosa fa altro dispositivo/sottorete a livello di traffico, come faccio? Opzioni: o possiamo mettere la mano sul pc (wireshark) o posso fare finta di essere il pc (chiedendo allo switch, non intrusivamente, di mandare il traffico verso pc pure a me) Prima di iniziare a guardare il traffico, il traffico va ricevuto.

### 0.2.1 Common problems with packet capture

Perché **root**? Perché scavalco ciò che fa un'applicazione, perché vedo tutto il traffico indipendentemente dall'applicazione. Per questo devo essere **root**, o almeno avere le capabilities necessarie per vedere i pacchetti di tutti i processi in esecuzione.

Lo sniffer dovrà essere all'interno di una VM (virtualizzazione del SO con un proprio kernel) o di un container (condivide kernel con l'host), in modo da intercettare il traffico loopback.

**Tecniche di monitoraggio** Spesso e volentieri, nonostante abbia la necessità di vedere il traffico, non posso mettere mano sulla macchina da monitorare: devo avere un S.O. che lo permetta, avere i diritti per installare il software necessario...

La soluzione è monitorare il traffico da fuori, prelevandolo dallo switch. Ci sono diversi metodi, software e hardware. **Metodi software**, riguardanti le porte dello switch:

**port mirror:** comando allo switch di inoltrare tutto il traffico da e verso un host anche su una specifica porta, su cui installo lo sniffer. Può avere più "molteplicità":  
1:1, una porta verso una porta  
1:N, tot porte switch le mandi qua

**VLAN mirror:** simile al port mirror, inoltra tutto il traffico di una VLAN.  
Una VLAN serve a separare i traffici e non mescolare i dati, così che gli host stiano sul solito cavo senza saperlo.

**Traffic filter/mirroring:** filtrare il traffico ad esempio in base alla porta, all'indirizzo IP...

**Problematiche** Uno dei problemi maggiori è dato dall'utilizzo, da parte dello sniffer, della sola parte di ricezione del cavo. Il singolo cavo ha 2 Gbps (1 Gps in upload e 1 Gbps in download), quindi con il port mirror, con cui posso scaricare al massimo a 1 Gbps perché uso solo la parte di ricezione, ho risultati ottimi solo se il traffico sta sotto 1 Gbps. Dovrei avere una scheda di rete da 10 Gbps per reggere comodamente traffico e non perderlo. Necessito quindi di una **scheda di rete più veloce della somma delle due direzioni**.

**Hardware Network Tap** prende le singole direzioni del traffico e ne fa una copia. PC monitor con due schede di rete perché tap divide il filo le direzioni: una prende le direzioni in entrata e una prende le direzioni in uscita. Così scopro anche chi invia cosa. Nel port mirror non sono preservate le direzioni.

## 0.2.2 Interoperabilità

Software di switch e infrastruttura di rete devono essere duraturi, perché infrastruttura di rete si cambia quando c'è veramente necessità. Altrimenti infrastruttura rimane lì.

**Problema** Gestire le cose nel tempo, che rimangano interoperabili negli anni. Siccome informatica va avanti per mode, si sono posti come problema (fatta negli anni '80) dover gestire qualche sistema non attraverso la url come si fa ora, perché è un modo di fare molto volatile che cambia spesso. Allora hanno fatto standard con negoziazione alto livello: oggetto ha attributi ad alto livello, funzionali al suo funzionamento (macchinettà caffè: c'è acqua, quanti caffè fatti, se ha bicchierini...).

Attributi, cosa ti mostro io che sia rilevante per te (non quante viti ha, ma lo stato), operazioni che si possono fare su quegli attributi (accendi, spegni...), comportamento. Software si occupa dell'accensione, non so com'è fatto internamente, io mi limito a chiamare l'operazione. Standardizzazione di funzionamento.

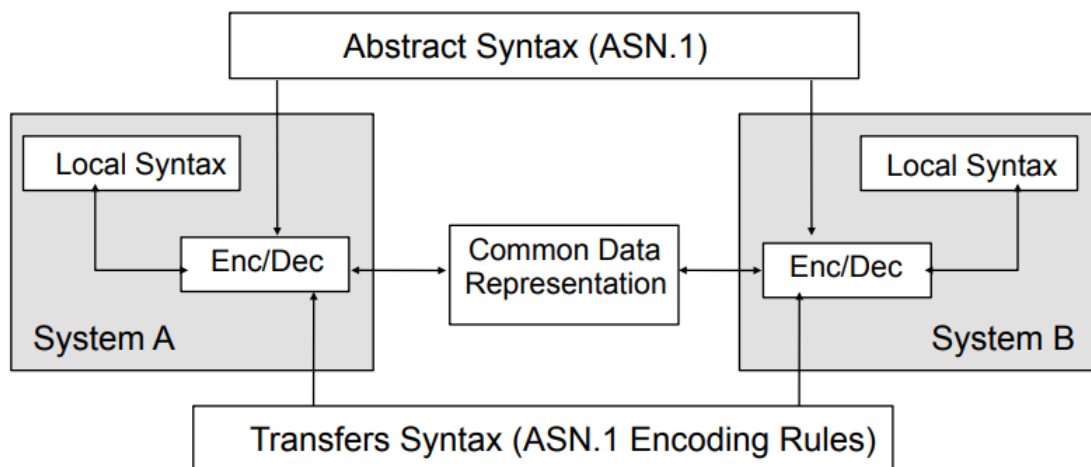
**Manager** Comanda l'operazione, impone politica di gestione.

**Agent** Gira dentro la macchina gestita e fa cosa chiede manager e solo quello.

**Paradigma** Agent gira nella macchina monitorata, e un solo manager raccoglie dati e visualizza.

**Come realizzarlo** Bisogna negoziare rappresentazione dati. Per lavorare col web hanno risolto problema scambio dati convertendo tutto a stringhe. Questo modo di fare non ha grandi problemi, ma è inefficiente. Grande quantità di dati per poche informazioni (`true` scritto invece di un bit). Poco efficiente per tanti dati, chiaro ma non compatto. La URL è breve. Tra due macchine posso scambiarmi dati in maniera binaria, ma bisogna mettersi d'accordo. Se io a 16 bit parlo con una a 64 bit non ci capiamo, quindi bisogna accordarci (Little Endian e Big Endian).

**ASN1** Sintassi astratta, implementata dai linguaggi. Quando iniziano a parlare due macchine negoziano rappresentazione e codifica.



La sintassi locale (**local syntax**) è diversa e tipicamente dipendente dal linguaggio utilizzato: ad esempio una in GO l'altra in C, ma anche per sistema A Arduino Nano e sistema B workstation Windows.

L'ASN1 quindi definisce una sintassi astratta standardizzata. Permette diverse regole di codifica che trasformano la sintassi astratta in un flusso di byte adatto al trasferimento: **BER** (Basic Encoding Rules) definisce il mapping tra sintassi astratta e sintassi di trasferimento.

ASN1 rimane architettura, idea. La sintassi di trasferimento può essere JSON, GO o qualsiasi altra cosa: **l'importante è che le due applicazioni si capiscano**. I **tipi di dato** (datatypes) primitivi dell'ASN1 sono:

BOOLEAN

INTEGER

BIT STRING

OCTET STRING

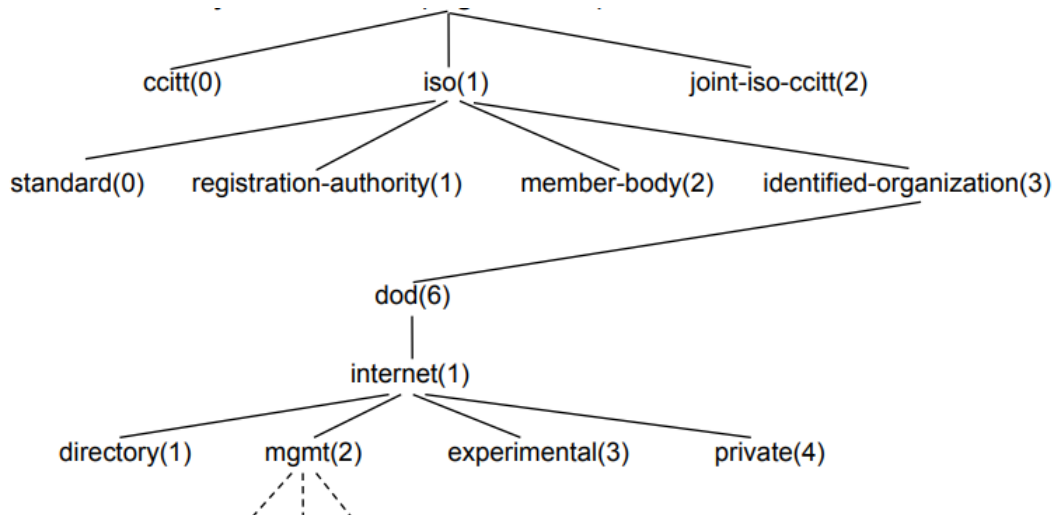
**OBJECT IDENTIFIER**

Quando trasferisco dati da applicazione ad applicazione, devo poter indicare un campo chiave. Questo tipo di dato identifica univocamente l'oggetto che sto trasferendo all'interno dell'albero ISO

...

**ISO Registration Tree** Usato per identificare univocamente definizioni, documenti, oggetti... Ha una struttura gerarchica, simile ai file system gerarchici. Tutti i nodi di un livello sono univocamente identificati da un numero. Il **percorso dalla radice al nodo** fornisce una **sequenza numerica** chiamata **Object Identifier**.

Per esempio, Internet è 1.3.6.1



**Internet** si trova sotto il **Dipartimento della Difesa**, che è una **Organizzazione Identificata** facente parte dell'**ISO**. Noi parleremo della **Gestione** (management) di Internet.

L'**object identifier** quindi, ovvero la sequenza di numeri, risolve il problema dell'identificazione univoca della tipologia dell'oggetto trasmesso sulla rete.

**Tipi Complessi** ASN1 ha anche tipi complessi, come **SEQUENCE OF** che specifica una lista di dati omogenei, o **REAL** che specifica i numeri reali con mantissa ed esponente dagli **INTEGER**.

**Basic Encoding Rules** Regole di codifica, compattano i dati in una stringa di byte da spedire sul filo. Basato su un algoritmo tag/length/value (TLV), dove ogni variabile è identificata da un tag, la lunghezza del valore in byte e il valore di quei byte. Questo permette al ricevente di ricostruire il tipo del messaggio a partire dal flusso di byte ricevuto.



# Capitolo 1

## Gestione di Rete

### 1.1 Nascita

Gestione di rete nacque, storicamente, nel mondo della telefonia. Necessità di codificare numero, connettersi al centralino, riconoscere il numero. . . . Standardizzazione necessaria per poter far telefonate a distanze elevate, attraverso le nazioni.

La necessità era di un protocollo **semplice**.

**Dati** Poi arrivò internet. Prima bisognava instradare la voce, con internet vanno instradati i **dati** (anche voce, VoIP, ma pur sempre dati). Eredità di tutte le tecnologie e teorie dei tempi della telefonia (es. 5G) ma anche innovazione (es. browser web), pur mantenendo il paradigma esistente che era tutto sommato efficiente.

### 1.2 Gestione di Rete Internet

**Cos'è** Sistema di protocolli e tecnologie che permettono di mettere in funzione e controllare un'infrastruttura di rete, per far sì che sia efficiente e che faccia ciò che voglio e che segnali eventuali problemi e comportamenti non previsti.

**Reti Geografiche** Questo discorso si applica a reti geografiche, ampie e complesse, che interconnettono un elevatissimo numero di device. Serve anche per far sì che la connessione/disconnessione di dispositivi non crei problemi, e che un utente della rete non possa creare disservizi e potenzialmente tirarla giù.

**Anni '90** Il problema principale era mantenere bassi i costi, perché doveva essere pervasivo e poter mettere router in ogni casa. Centraline telefoniche, al contrario, non devono stare in ogni casa.

Gli apparati di rete quindi devono costare poco, perciò la gestione di rete non deve costare tanto (nella telefonia costa tanto ed è complicata, quindi "*non facciamo lo stesso errore*": se è semplice anche il costo computazionale è basso, quindi il dispositivo è più economico). Il protocollo, quindi doveva essere **semplice** e **efficiente**.

Altra cosa importante era l'**ubiquità** del protocollo: doveva essere disponibile su tutti i dispositivi, così da poterli gestire tutti.

Inoltre il protocollo doveva essere **estensibile**. Almeno **retrocompatibile**.

### 1.3 SNMP

**Piccoli passi** Il protocollo SNMP è stato progettato di pari passo con il diffondersi di internet, prima a livello universitario e poi industriale. Si è iniziato a sviluppare questo protocollo di gestione dagli albori, perché sin da subito è apparso chiaramente l'importanza che l'infrastruttura stia in piedi.

L'SNMP monitora lo stato della rete, per far sì che la rete risponda alle esigenze. Ci sono più standard, con primi sviluppati nel 1990.

**Semplice** Doveva essere **semplice**, poiché i sistemi erano semplici, poco potenti e a volte nemmeno multitasking. L'SNMP non poteva girare "in hardware", ma **necessita di un computer perché necessita di elaborazione dati** e dello stack IP per comunicare. Negli anni '90 lo stack IP non era necessariamente presente sui computer in commercio.

La parte importante è che sia **semplice** e funzionare sotto l'UDP, che è un protocollo estremamente semplice.

**Separato** La parte dell'SNMP è **separata dalla parte di comunicazione** anche se aiuta l'instradamento. **Non interferisce**, come il cruscotto della macchina (SNMP) col motore (switching).

**Trasparente** Sta fuori dalla comunicazione, ma **deve poterla controllare** e monitorare **senza interferire**. L'idea è che se l'SNMP viene compromesso lo switch continua a funzionare.

**Evoluzione** Nel 1990 viene standardizzata una versione molto semplice: l'**SNMPv1**. Questa versione fu prodotta in fretta, concentrandosi sulle funzionalità base, in modo da poter entrare subito sul mercato che stava per esplodere. Nel 1991 viene pubblicata la **Management Information Base**, ovvero l'insieme degli oggetti manipolati tramite l'SNMP.

Negli anni successivi ci sono varie evoluzioni del protocollo:

**SNMPv1** supportata da tutti i dispositivi sul mercato

**SNMPv2** aggiunge poche funzionalità, ma è molto usata soprattutto perché i contatori ora sono a 64bit

**SNMPv3** aggiunge parecchie funzioni, sacrificando il "simple", quindi non è particolarmente diffuso

**Utile** L'SNMP è quindi utile per il monitoring centralizzato su reti estese: è **importante che questi protocolli siano in funzione in ogni momento**, per riconoscere i problemi in anticipo e avere uno storico della rete per poter fare le verifiche. Anche solo contare il traffico prodotto in termini di byte è un'informazione molto importante.

**Agent** **Apparato di rete**, ad es. nella rete del Fibonacci ci sono diversi agent: access point, computer, stampanti... Sono gli **oggetti da gestire** e possono cambiare nel tempo: possono essere aggiunti/rimossi, ma possono anche cambiare in tipologia

### 1.3.1 Trap Directed Polling

Tutti gli agent rispondono ad un manager (solitamente ridondato) →  $n$  manager per un solo agent.



**Uguaglianza** Con lo stesso protocollo e la stessa funzionalità, devo **poter controllare dispositivi anche molto diversi**: computer, stampanti, badge, schermi...

L'SNMP **non distingue i dispositivi**, ma **prende le info dalla MIB** del singolo agent. Ciò che differenzia i vari sistemi operativi, i dispositivi tra loro ecc. sta tutto nella **MIB** (Management Information Base).

**Polling** Controllo. Il manager non comunica continuamente con gli agent, ma esegue un **polling degli agent**, contattandoli periodicamente per **ricevere le informazioni aggiornate**.

L'SNMP è altamente centralizzato, quindi è compito del manager implementare tutta la funzionalità di monitori e la responsabilità, sicurezza ecc...

**Traps** Una volta configurata, sta alla **periferica avvertire il manager se qualcosa non sta funzionando come previsto**. Non informa il manager ogni volta che succede qualcosa (stampo un foglio, prendo un caffè, mi connetto all'Access Point...) perché il manager verrebbe inondato di informazioni, ma **l'apparato informa il manager se ci sono cose che non funzionano**. Segue la filosofia del "Se non mandi niente va tutto bene".

Questo può non essere sufficiente, per esempio in caso di problema di rete la comunicazione può non andare a buon fine. In tal caso il polling può risolvere questa cosa, anche se ciò significa apprendere il verificarsi del problema in maniera non tempestiva.

**SMI** La struttura delle informazioni di gestione (**Structure of Management Informative**, la seconda versione) si basa su un sottoinsieme dei datatype ASN.1 più altri sottotipi:

**Integer32** interi con il segno

**Unsigned32** interi senza segno

**Gauge32** per misure pronte comprese tra min e max

**Counter32** e **Counter64** per i contatori: per conoscere la misura effettiva devo fare la differenza tra il valore assunto in due istanti separati.

**IpAddress** per IPv4

**TimeTicks** per i centesimi di secondo passati

**Opaque** che è come il **void**, non consigliato

Le variabili, inoltre, possono essere

**Scalar**: esistono una sola volta per agent. Es: il nome di un host.

**Concettuali**: esistono in una tabella concettuale, con valori che cambiano nel tempo.

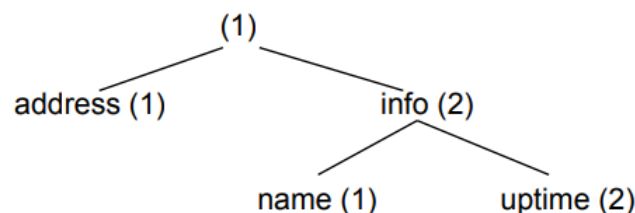
**Read** o **write**, lette o scritte. Non esistono incrementi o reset a valori iniziali, questo per semplicità di protocollo

**MIB** Le MIB di SMIV2 sono **definite tramite speciali macro ASN.1**

Use Case



Le variabili sono definite nell'ISO Registration Tree. Una foglia dell'albero rappresenta un managed object.



**Instance Identifier** Ogni managed object è identificato dall'object identifier. Ma tale oggetto ha istanze con determinati valori che cambiano nel tempo: tale istanza è identificata univocamente da un **Instance Identifier**. La singola istanza di un managed object è **univocamente identificata concatenando l'Instance Identifier all'Object Identifier**.

Gli scalari hanno una sola istanza, e l'instance identifier è .0. Negli altri casi (non identifier) si parte da .1.

L'instance identifier è il valore della colonna indice.



In questo caso, per prendere il valore della terza istanza di mask:  $1.3.1.2.9 \rightarrow 5$ .

Altri esempi:

$1.3.1.1.1 \rightarrow 1$

$1.3.1.1.4 \rightarrow 4$

$1.3.1.2.1 \rightarrow 2$

$1.3.1.2.4 \rightarrow 7$

$1.3.1.3.1 \rightarrow 2$

$1.3.1.2.7 \rightarrow \emptyset$

### 1.3.2 MIB

Come si definisce un MIB

```
nome DEFINITIONS ::= BEGIN
```

```
IMPORT MODULE-IDENTITY, OBJECT-TYPE, enterprises, IpAddress, TimeTicks FROM SNMPv2-SMI;
...
```

```
END
```

I nomi servono solo per gli umani, perché sulla rete passano gli object identifier.

**Module Identity** Non aggiunge informazione ma indica la versione, questo perché negli anni le reti cambiano ma l'infrastruttura rimane fissa almeno in parte. I device a lungo termine (ad esempio lettori di badge) devono convivere con i nuovi: eterogeneità nelle velocità di elaborazione, protocolli...

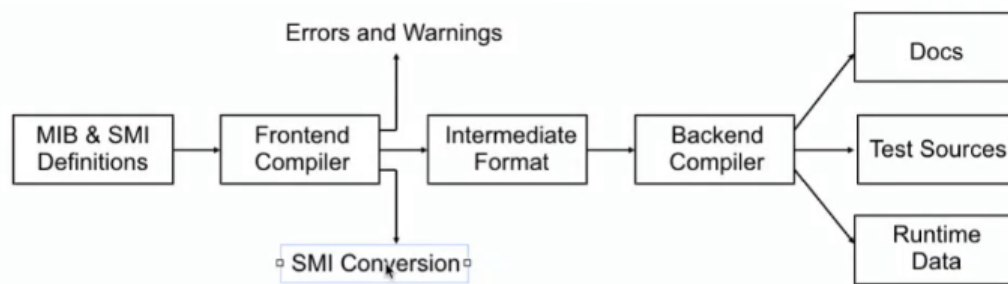
Non posso togliere oggetto, perché magari versioni precedenti lo usano. Se voglio eliminarlo posso indicarne lo **STATUS**: obsolete, current o deprecated.

**Object Type** nome OBJECT-TYPE, SYNTAX, UNITS...

**Notification Type** Invio anche oggetti interessati alle trap

linkDown per segnalare se ha perso il link, linkUp per collegamento preso





**Utilità dei MIB** Compilatore non è detto che generi linguaggio macchina ma trasforma semplicemente. I **frontend compilers** SNMP prendono i MIB e producono warning/errori e conversioni di formato SMI

Quando è nato l'SNMP sono stati fatti due standard: SNMP e MIB iniziale (MIB-II, RFC 1213) dove vengono descritti gli object più importanti per la gestione di internet.

Vogliono controllare stack TCP/IP per vedere cosa non funziona e poi monitorare porta di rete. I device espongono dei contatori per ogni interfaccia tramite SNMP: contare i byte e i pacchetti è **la base del monitoraggio**, ma nel tempo vanno superati perché i contatori su byte sono poco utili.

Obiettivi MIB-II: informazioni di base sugli errori, pochi e semplici control objects (porte su/giù, errori, pacchetti e byte ingresso/uscita), cerca il più possibile di evitare info ridondanti, **non deve assolutamente interferire con le operazioni**, devono essere separati e non rallentare o inficiare in nessuna maniera prestazioni dei dispositivi.

Si implementa in hardware, **eth-tool**. Non si mette a contare i pacchetti, ma è direttamente l'hardware a rispondere con le informazioni. Se chiedo numero pacchetti in un minuto, lui delega all'hardware di contare, quindi il carico non cambia in base al traffico di rete.

Sono circa 170 oggetti, negli anni alcune def sono divenute troppo semplici e presuppone IPv4 (IPv6 gestito in MIB separati).

schema dei MIB-II (1.3.6.1.2)  
Albero serve a identificare oggetti.

**Composizione MIB-II** Diviso in gruppi: sistema, interfacce, arptable, componenti stack IP. Anche una parte relativa all'agent SNMP: quante query, quanti errori...

La parte trasmissione è aperta, tanti sottoalberi: com'è connesso un computer dipende dalla tecnologia, quindi è estensibile a seconda delle nuove tecnologie e cambia nel tempo. Negli anni viene modificato ma è tutto sommato integro sin dalla nascita. Ci sono sovrapposizioni fra i MIB, però.

Per info più specifiche ci sono i MIB successivi. Essi guardano la parte di trasmissione a vari livelli, quindi qualche informazione si può ripetere.

tabella

Le statistiche sulla stessa riga sono a livello diverso, repeater a livello 1. Fanno la stessa cosa ma livelli diversi.

Per navigare le tabelle si necessita di un meccanismo efficiente rispetto a fare tante richieste. Questo perché sistemi modulari, per vedere se valore/elemento esiste.

**Ordine lessicografico** walk: lettura consecutiva di oggetti in MIB.

Leggo oggetti e metto ordinati per numero in ordine lessicografico. Incremento OID e se ho oggetti nel sottoalbero lo visito, da sinistra verso destra. Ricorsivamente, ogni nodo che incontro se ha figli lo visito. Con questo ordinamento si perde la struttura della tabella: **SNMP lavora solo su questo array ordinato**.

**Trap** Unica operazione iniziata dall'agent, che manda direttamente al manager informazioni. Maniera asincrona per informare manager di qualche avvenimento, tipicamente un cambio di stato.

L'**administrative** è il ciclo di vita, l'**operational** è il ciclo operativo. **ifAdminStatus** su **linkDown** equivale proprio a scollegare fisicamente la scheda di rete. **ifOperStatus** su **linkDown** è lo stato operativo, e significa che la connessione è caduta.

**Estensione dei MIB** Un MIB si può estendere con comandi arbitrari. Ad esempio **extend test1 /bin/echo "Hello, World!"** eseguirà lo script indicato ogni volta che si esplorerà **test1** all'interno del MIB **NET-SNMP-EXTEND-MIB**. Lo script è eseguibile da chiunque abbia accesso alla community a cui è assegnato, quindi bisogna stare attenti alla sicurezza.

### 1.3.3 Primitive

Due tipi di valori, scalari e tabellari.

**Get** Richiesta diretta e precisa di un object identifier

Può essere usato per leggere una o più variabili. Basato su UDP, quindi dimensione massima pacchetto: nella risposta bisogna stare attenti alle info. Possibile errore (**errorStatus**): **tooBig**, non entra la risposta nel pacchetto UDP. Altri errori: **noSuchName** (istanza non esiste o non è foglia, esempio chiedere bicchierini ad una stampante), **genErr** (qualsiasi altro errore: comunità sbagliata, agent sovraccarico...).

**errorIndex** indica quale delle variabili ha avuto problema. Es **noSuchName@1** indica che la prima variabile non esiste. Posso fare get con più richieste, quindi errore può essere su altra variabile (es **noSuchName@3** cioè non esiste il terzo object identifier richiesto ma indica il primo indice fallito, quindi le altre due precedenti esistono ma non vengono ritornati i valori).

**GetNext** Richiesta che ritorna l'object identifier successivo a quello chiesto.

Non legge istanze identifier richiesto ma ritorna il prossimo istanze identifier rispetto all'ordine lessicografico. usata per fare discovery delle strutture e leggere le tabelle.

Nella getNext, nosuchname significa che è finito il MIB (non esiste il successivo). Molto implementation depended, alcuni dopo l'ultimo ritornano il primo invece che noSuchName.

**GetBulk** Un po' l'unione di una Get e una GetNext.

**non-repeaters**: di tutti gli OID specificati, indica quanti di essi sono da Get e non da GetNext (i primi **non-repeaters**)

**max-repetitions**: indica il numero di OID dove bisogna fare GetNext (cioè **max-repetitions** GetNext sui rimanenti)

**Set** Scrivere un valore

Equivalente in sostanza alla get ma scrivo. Atomica: più valori sono scritti contemporaneamente, quindi o vanno bene tutti o non scrivo niente.

Errore **badValue** quando valore scritto è del tipo sbagliato, fuori dal range o comunque non accettabile. Ma anche errore quando oggetto è **read-only**.

**noError** quando tutto ok, **noSuchName** quando istanze identifier non esiste. Anche qua stesso discorso della get con l'indice.

**Trap** Trap dall'agent

**Unico messaggio non richiesto che va dall'agent verso il manager.**

Può succedere trap storm, esempio dopo perdita alimentazione tutti dispositivi segnalano riavvio apparati tutti insieme. Se tolgo corrente non posso fisicamente mandare trap, per questo succede quando riavvio e torna su, non quando va giù tutto insieme.

Non ci si può fidare totalmente della trap, bisogna fare polling.

**ColdStart/WarmStart** inviate quando avvio un agent (cold da freddo, spento, caso tipico, invece warm è da riavvio) **LinkDown/LinkUp**

**AuthenticationFailure** quando faccio richiesta posso non avere autorizzazioni, trap segnala al manager della situazione (tipicamente se provo tante password una dietro l'altra).

**EnterpriseSpecific**, vedi valore enterprise e specific nel formato del pacchetto. ad esempio fine carta di una stampante. **Specific=1** per dire che l'interpretazione è a seconda del mib.

#### Formato pacchetti

**Community**: una sorta di password, vedo se utente che richiede è abilitato a certa operazione. PDU payload data unity

Get getNext e set ricevono getresponse tutte e tre.

Importante sapere IP host che manda trap (potrebbe vederlo dal pacchetto UDP), perché l'ip nell'udp può essere mascherato o cambiato ad esempio dal nat

**sysDescr(1)** breve descrizione del sistema (stringa, **arbitraria**)

**sysObjectID(2)** identifica sia modello che costruttore, per riconoscerlo e andare nel relativo MIB o, ad esempio, mettere l'icona giusta.

**sysUptime(3)** da quanto è attivo l'**agent** (non è per forza identico a quello di sistema). Serve anche per poter leggere correttamente le variabili **gauge** per esempio, per fare la differenza per sapere quanto è cambiato il valore

nell'ultimo minuto (ad esempio). Ma non posso aspettarmi che il nuovo valore sia maggiore (es: è saltata la corrente, oppure se il contatore ha fatto wrap (fine dei bit e ripartito da 0)) Quindi devo distinguere riavvio agent da wrap, non posso fare semplice differenza perché potrei tirare fuori numero sbagliato (es negativo, ma valore **unsigned** fa diventare grandissimo). Per sapere se si è riavviato agent o se contatore ha fatto wrap uso **sysUpTime**.

**sysContact(4)** chi contattare.

**sysName(5)**

**sysLocation(6)**

**sysServices(7)** dà info sui servizi gestiti dal sistema. Stringa di bit che identifica a quale livello ISO/OSI si trova il sistema

### **Gruppo interfaces** Informazioni sulle interfacce di rete

**ifIndex** da indice interfaccia di rete. Non sono consecutivi, se aggiungo o rimuovo porte numerazione non cambia.

**ifDescr** che descrive cosa fa l'interfaccia

**ifType** il tipo di interfaccia, eth, wireless...

**ifMTU** la Maximu Transmission Unit. La frammentazione (livello IP) la fa chi riceve un pacchetto deve spedirlo su una linea con MTU più piccolo

**ifSpeed** la velocità attuale dell'interfaccia

**ifPhysiAddress** indirizzo fisico, il MAC

**ifAdminStatus** indica se l'interfaccia è fisicamente presente o meno

**ifOperStatus** indica se l'interfaccia è operativa (accesa, up...)

**ifLastChange** valore del **SysUptime** dell'ultima volta che l'interfaccia di rete ha cambiato stato

**ifInOctets** numero ottetti in ingresso alla porta

**ifInUcastPkts** pacchetti unicast

**ifInNUcastPkts** pacchetti non unicast

**ifInDiscards** pacchetti scartati

**ifInErrors** pacchetti in errore

**ifInUnknownProtos**

**ifOutOctets** numero ottetti in uscita alla porta

**ifOutUcastPkts** pacchetti unicast

**ifOutNUcastPkts** pacchetti non unicast

**ifOutDiscards** pacchetti scartati

**ifOutErrors** pacchetti in errore

**ifOutQlen** numero di pacchetti in coda in uscita. Dovrebbe essere 0 o quasi, altrimenti interfaccia non riesce a spedire i dati al giusto ritmo

**ifSpecific**

**enp0s1** Perché si chiama così? Dà informazione su dove si trova la scheda

**Gruppo arp** Informazioni sulla risoluzione dei nomi. Riempita automaticamente dallo switch: su quali interfacce ha scoperto quali indirizzi fisici. Non è una corrispondenza delle porte, è dell'ARP.

### 1.3.4 SNMPv2

Usato snmpv2 invece che v1 perché aveva problemi strutturali: il primo sono i contatori a 32bit che sono molto limitati per la tecnologia odierna (10-100Gbit il contatore si "riempie" molto velocemente). Inoltre nella v2 c'è tutto quello scartato in v1 per questioni di tempo (necesario fare protocollo in fretta)

**Primitive di SNMPv2** Sono tutto sommato le stesse con **GetBulk** e **Inform**

**GetBulk** per richieste più efficienti per massimizzare numero richieste in un pacchetto. Metà strada tra **Get** e **GetNext**. Oltre OID due parametri:

**non-repeaters**: di tutti gli OID specificati, **non-repeaters** indicano quanti di quelli sono quelle variabili a cui applicare una **Get** secca e non in sequenza. Dei restanti, devo fare **max-repetitions GetNext**

**max-repetitions** quante **GetNext** applicare

Se chiedo poche cose non risparmio molto rispetto ad una **GetNext**, se chiedo troppe cose (**max-repetitions** troppo alto) devo stare attento che entri tutto in un pacchetto.

**Inform** è una sorta di **Trap** informata perché riceve una **response**. Altra differenza è che la **Inform** la può mandare sia un Manager (che può informare un Manager di livello superiore, per esempio) che un Agent

**Formato** Hanno la stessa PDU ma con modifiche all'interno, però sostanzialmente è uguale

**Eccezioni** A differenza degli errori di SNMPv1, le eccezioni sono più esplicative. Ad esempio **noSuchName** spaccettata in **noSuchObject**, **noSuchInstance** o **endOfMibView**. Sono un **raffinamento dei codici di errore di SNMPv1** e un **miglioramento delle primitive**.

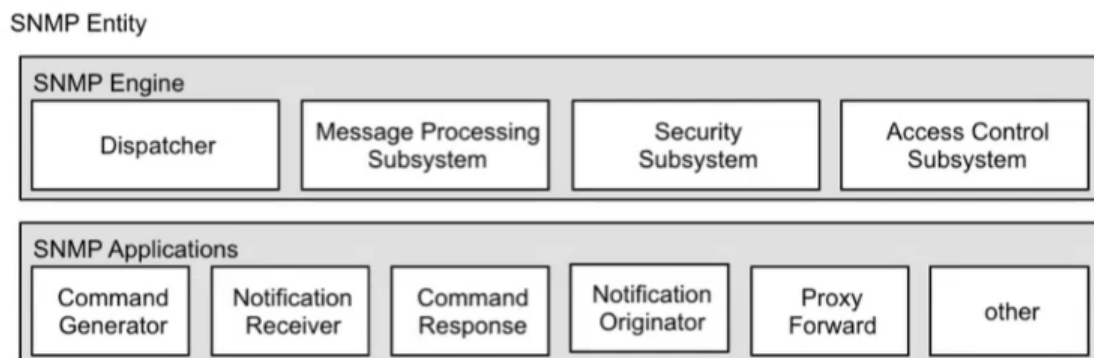
**Differenze principali** Più velocità con la **Bulk**, contatori a 64 bit, errori molto più dettagliati. Ma **non risolve granché in termini di sicurezza**.

### 1.3.5 SNMPv3

**Obiettivi** Bisogna risolvere problemi di sicurezza soprattutto per la **Set**. Inoltre bisogna rendere SNMP un'architettura a lungo termine, adatta a supportare implementazioni sia complesse che semplici (**scalabilità**). Tutto questo **rimanendo il più semplici possibile**.

Nella realtà non è stato così, e la diffusione di SNMPv3 è ancora molto limitata con la maggior parte delle reti che continuano ad usare v1 e v2.

**Architettura** Tanti componenti, rimovibili e intercambiabili. Vuoti se una certa implementazione non la supporta, rimovibili a seconda di cambiamenti tecnologici o di implementazione.



Supporta vari tipi di protocollo per lo scambio di messaggi e per il sistema di sicurezza (comunità, utente, "altro")

## Caratteristiche più importanti

### 1. Sicurezza

Il messaggio è autentico? Data Integrity e Authentication

Tramite una funzione hash, con chiave simmetrica faccio l'hash dei dati e calcolo un MAC (Message Authentication Code) es MD5. Pacchetto diventa User (key), MAC e Data.

Il ricevente ricalcola l'hash e se MAC calcolato e ricevuto sono uguali allora apposto.

Il problema è la **chiave simmetrica**.

### 2. Protezione dalla ripetizione di vecchi messaggi

Il ricevente conosce l'orario del messaggio ricevuto, se il messaggio ricevuto è nell'intervallo di validità e *più giovane* dell'ultimo messaggio valido, allora è processato e l'orologio viene aggiornato. Quindi, ad inizio comunicazione gli orologi vengono sincronizzati.

### 3. Protezione dallo sniffing: si crittano i dati



**Messaggi** Possedendo, tra le altre cose, il supporto all'encryption, SNMPv3 ha un formato dei messaggi completamente diverso dalle versioni precedenti. Può essere un problema, in particolare per chi scrive i device: la complessità per implementarlo è sicuramente diversa.

### SNMPv3Message:

msgVersion	msgGlobalData	msgSecurityParameter	msgData (scopedPDU)
------------	---------------	----------------------	---------------------

### MsgGlobalData:

msgID	msgMaxSize	msgFlags	msgSecurityModel
-------	------------	----------	------------------

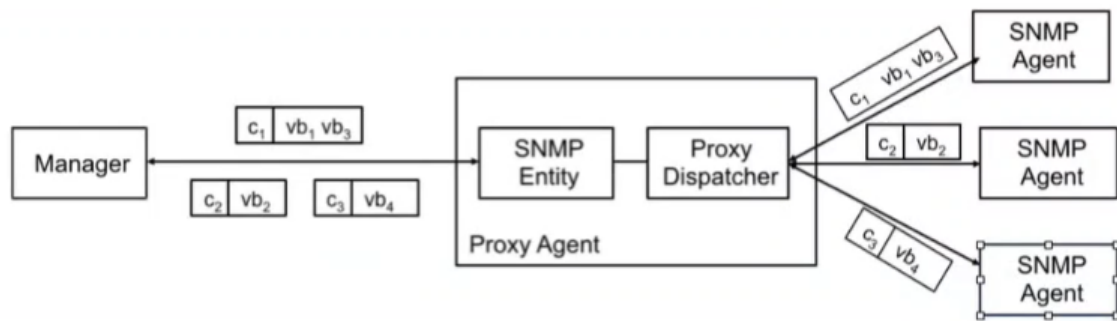
### UsmSecurityParameter:

msgEngineID	msgEngineBoots	msgEngineTime	msgUserName	msgAuthParams	msgPrivParams
-------------	----------------	---------------	-------------	---------------	---------------

### ScopedPDU:

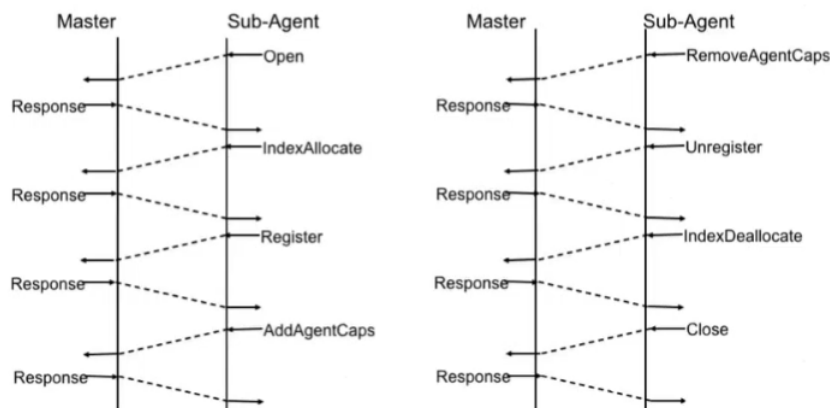
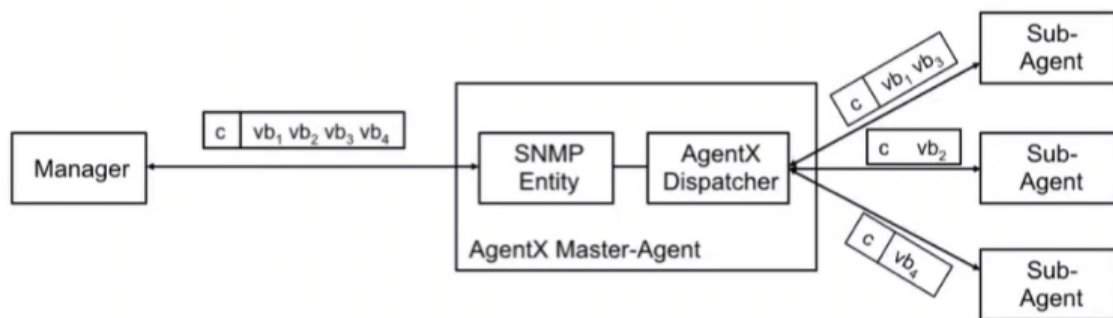
contextEngineID	contextName	SNMPv2 PDU (as defined in RFC 1905)
-----------------	-------------	-------------------------------------

**Proxy Agents** Permettono ai manager di accedere ad altri agent non accessibili direttamente (es dietro firewall) o accessibili mediante protocolli non IP (es IPX)



Le applicazioni di management devono, solitamente, selezionare le opportune stringhe di comunità o contesto, così da abilitare il proxy a raggiungere gli agenti (non c'è trasparenza). I proxy sono importanti per l'implementazione dei firewall o per conversioni tra diverse versioni di SNMP.

**Agent estensibili** Il master agent è un agent vuoto, senza variabili all'interno. Il subagent è l'agent stesso, che implementa il MIB.



SubAgent fa la open, a cui il Master risponde con la response.

SubAgent poi comunica, con la IndexAllocate, quali OID implementa. Il Master vedrà se nessun altro ha richiesto gli stessi OID, in caso risponde al SubAgent, che risponderà registrandosi e in caso con altre capabilities.

**Quindi ogni subagent specifica al master cosa implementa.**

ASN1 finisce a SNMP entity, nella parte successiva c'è il protocollo AgentX, molto più semplice su TCP.

### 1.3.6 Bridge MIB

Bridge è mettere insieme due interfacce di rete

Risponde alla domanda: come fare per iniziare a vedere la topologia della rete e, quindi, i dispositivi collegati ad un host?

Attraverso bridge-mib per la topologia di rete. Importante anche per inquadrare che impatto può avere un'allarme sul sistema. Così so sia quali sono i nodi principali ma anche l'impatto di qualche problema: se cade una macchina che collega tante altre macchine, cadono tutte, mentre se cade un nodo marginale il problema è più contenuto.

## Capitolo 2

# Rete Cellulare

Quando telefono si connette a rete mobile il gateway lo autentica.

GTPC realizza autenticazione, il telefono si annuncia sulla rete (register) e disannuncia quando esce, oppure update quando cambia cella. L'annuncio comprende il MEI (identificativo univoco del telefono), IMSI (international mobile subscriber identity) e altre info, ad esempio sul profilo tariffario (velocità max ad es). Sulla rete passano anche info su nodo di rete e identificativo della cella. Le informazioni sono ricavate e comunicate dal provider e dal GTPC, non dal telefono. Ad esempio, il numero di cellulare è dato dal protocollo, non è memorizzato sulla SIM.

Create session crea la connessione e la stabilisce.

Protocollo GTP inoltre negozia i **tunnel**: permettono che cambiando cella cambiano i parametri della connessione ma la comunicazione rimane. Pacchetti incapsulati in tunnel GTP, io continuo a vedere il mio ip verso l'esterno e percepisco come se fosse rete fissa.

Traffico è incapsulato dentro il pacchetto GTP.

**Analisi traffico radio** Per analizzare il traffico radio delle reti mobili (ad esempio, tramite Wireshark) su smartphone si può implementare una "VPN senza VPN": cioè i pacchetti vengono catturati dal sistema operativo per mandarli su una VPN che in realtà è la rete mobile stessa.





## Capitolo 3

# RRDtool

### 3.1 RRDtool Database

I db relaz hanno tabelle connesse con relazioni...con inserimento, cancellazione e aggiornamento. Questo non è abbastanza. Inoltre su uso questo modo la tabella cresce all'infinito, durante il monitoraggio. Poi DB relaz non efficienti nel fare aggregazione nel tempo dei valori. Inoltre, i dati più vecchi diventano via via meno importanti.

**Scalabilità** Per calcolare la scalabilità, si può fare un semplice programma che esegue la computazione necessaria (ad esempio popolare il database RRD con  $n$  dati). Prima e dopo la computazione si fa una `getTimeOfDay()` e si fa la differenza tra i due tempi. Si calcola così quanto tempo ci impiega. Se per esempio il programma è troppo lento a gestire  $n$  dati/host/agent, si possono fare  $k$  thread e ognuno gestisce  $n/k$  elementi.

**RRDtool** Padre di tanti DB a sede temporale. RR=Round Robin. RRD è un **file**, in un formato particolare:

**Static Header** che indica cosa contiene, ultimo timestamp di aggiunta e ultimo dato aggiunto...

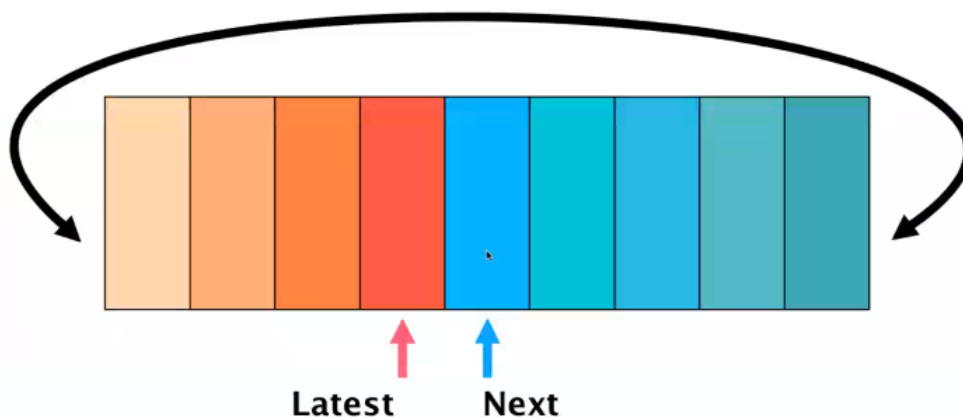
**Live Header**

**Round Robin Archive** specifica che tipo di dato mantenere per quanto tempo e con che risoluzione misurare

**Round Robin Archive...**

...altri RRA

**RRD** Array circolare con numero fisso di slot di storage.



**Data Source** Qualsiasi cosa con numeri, file di log, contatori SNMP...

**Sconosciuto** Come gestire UNKNOWN? Unknown non è 0, è contagioso ( $1 + \text{unknown} = \text{unknown}$ ). RRDtool gestisce gli unknown, configurabile quanto unknown ignorare (default: 50%) e come gestirli

**Archivi multipli** Tieni i dati pronti alle giuste risoluzioni  
Media su 5 minuti per un giorno. Massimo su 1h per un mese

**Anomaly Detection** Per accorgersi di problemi, soglie: colorare grafici in base a soglie (sopra 60 giallo, sopra 85 rosso ad esempio). Però sono statici.

Altrimenti in base ai valori storici: vedo qual'è la media nel passato e se mi discosto tanto...

Una cosa interessante che fa RRDtool. La **derivata** da indicazione di quanto mi discosto dall'osservazione precedente: 0 è costante, numeri più alti indicano grandi variazioni dall'osservazione precedente. Quindi la prima domanda è quanto tempo osservare. Anche dalla natura della metrica: una derivata violenta può significare problemi al sensore, in altre metriche può essere normale (trasmissione, attività processore...).

Inoltre la derivata su altre misure possono anche essere molto ampie senza che sia allarmante: download/upload, ad esempio. Dipende molto da cosa misuro: derivata su più intervalli o solo sulla precedente misura?

Però il color-coding: verde ok, giallo mh e rosso male male, è ottimo. Keep it simple, sepicativa e semplice è meglio di troppi valori disaggregati.

Nella pratica, l'**anomaly detection** consiste nel fare una istantanea del sistema monitorato in due momenti diversi e vedere quanto differiscono i due momenti, interpretando le differenze.

**Holt-Winters** Media mobile stagionale. Parametri:

$\alpha$

$\beta$

Periodo di analisi

All'interno dell'array di valori si crea le bande. tutti valori letti dalla scheda di rete

ifindex: identifica l'interfaccia

ifdescr: descrizione interfaccia (ad es: eth0 ma anche descrizioni significative)

iftype: tipo interfaccia (ethernet, ...)

ifmtu: maximum transfer unit, indicata per poter evitare cattive configurazioni nell'mtu. MTU deve essere la stessa perché se uno trasmette a  $> \text{mtu}$  di uno dei due non comunicano più

ifspeed:

ifphysaddress: indirizzo fisico della porta di rete, mac address della porta dello switch

ifadminstatus: se scheda è fisicamente presente sullo switch

ifoperstatus: per una scheda presente, se è operativa oppure no

iflastchanged: sysuptime dell'ultima volta che l'interfaccia è cambiata di stato

ifinotets: otteti in

ifinucastpkts: unicast in

ifinnucastpkts: not-unicast in

ifindiscards: discarded in

ifinerrors: errori in

ifinunknownprotos: sconosciuti in

ifoutoctets: otteti out

ifoutucastpkts: unicast out

ifoutnucastpkts: not-unicast out

ifoutdiscards: discarded out, dipende dall'implementazione della scheda

ifouterrors: errori out, dipende dall'implementazione della scheda

ifoutqlen: stessa info del comando **netstat** e simili, info sulle code in uscita

ifspecific:

router frammenta il pacchetto quando mtu della linea ricevente è più piccolo. ma può essere che mittente mandi pacchetti frammentati quando qualcuno chiede a lui di mandare pacchetto troppo lungo rispetto mtu.

Frammentazione è concetto dell'IPv4

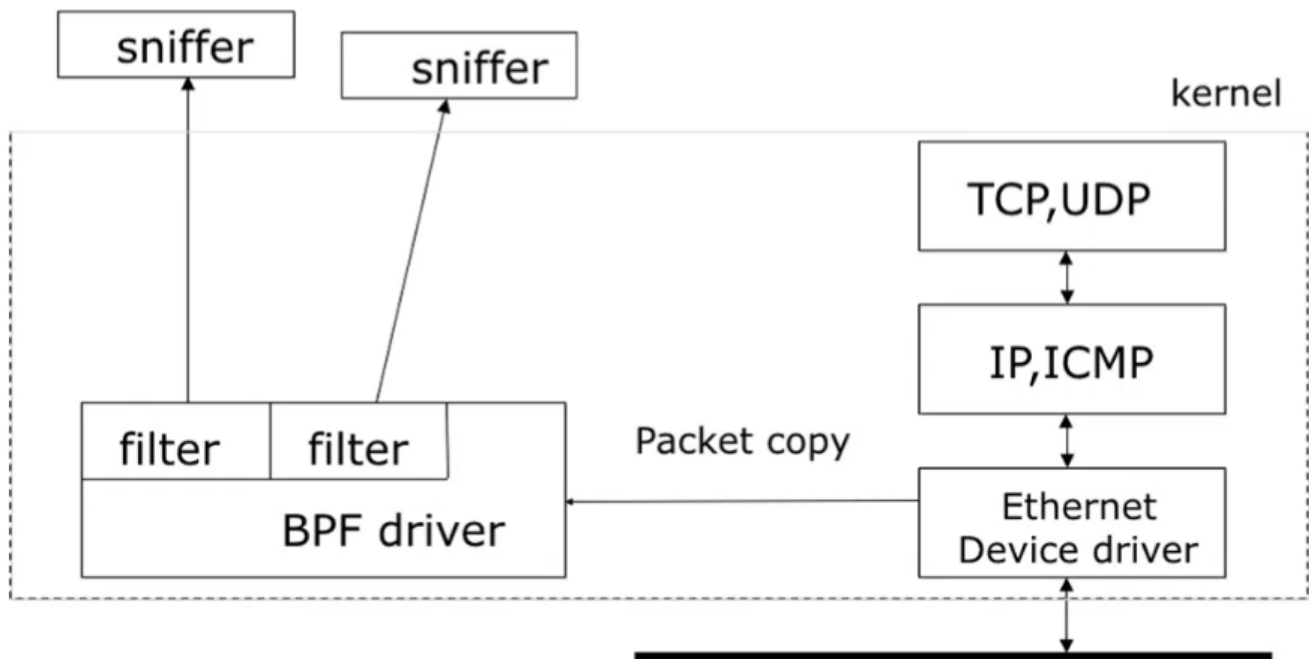
TCP frammentazione non c'è ma non perché non si possa fare, ma perché un pacchetto TCP non può essere frammentato (è indicazione di errore che qualcosa nel TCP non ha funzionato). Path MTU discovery (PMTUD), pacchetti sempre più grandi finché non si trova dimensione massima con l'**obiettivo di evitare la frammentazione**. Perché meccanismi TCP permettono di farlo. UDP pacchetti vivono di vita propria quindi può succedere frammentazione.

Pacchetti non unicast sono "problematici" perché riempino le porte dello switch.



## Capitolo 4

# Cattura



Quando cattura traffico intervengo sul traffico e me lo copio (**shallow copy**) verso l'applicazione che cattura ciò che sarebbe destinato ad altri. Quindi ethernet copia pacchetto al BPF driver che, tramite filtri, manda copie agli sniffer. Gli sniffer necessiterebbero di diritti su

**Capabilities** Quando uno è SU ha diritti eccessivi rispetto a ciò che deve fare, quindi le **capabilities** sono le possibilità di spezzare i diritti di amministrazione in categorie: non c'è bisogno di essere root che fa veramente veramente tutto perché se ne può abusare. Quindi il concetto di root è spaccettato in varie possibilità: configurazione del MAC address, Net Admin...

Un'app quindi dovrebbe nascere con concetto di capabilities e avere diritti minimi necessari per fare ciò che vuole/deve fare.

Una **capability** è implementata come una struttura dati privilegiata, con una sezione che specifica i diritti di accesso ed una sezione che identifica univocamente l'oggetto.

**Filtro** Devo indicare l'interfaccia a cui sono interessato catturare pacchetti (eventualmente tutte) e in caso che tipo di pacchetti ricevere. Così il BPF driver **non** invia all'applicazioni pacchetti non interessanti, perché anche scartare qualcosa richiede tempo.

BPF e sottosistema è localizzato nel kernel. Solamente la parte del pacchetto è inoltrata, il resto (filtro e copia dei pacchetti) è nel kernel.

Nel mondo linux, la cattura dei pacchetti equivale ad aprire un socket.

## 4.1 libpcap

**pcap\_open\_live** Parametri: **deviceName** scheda di rete, **maxCaptureLen** lunghezza massima del pacchetto (sia per efficienza, sia per privacy se non ho bisogno dell'intero pacchetto), **setPromiscuousMode** per abilitare la modalità promiscua (cattura tutto il traffico sulla scheda), **pktDelay** non c'erano i thread quindi hanno pensato di mettere un tempo massimo per cui aspettare un pacchetto e in caso ritornare senza valore, **errorBuffer** sempre per ragioni storiche se la **pcap\_open\_live** fallisce il buffer contiene la ragione per cui è fallita

**live**: leggo traffico di rete in tempo reale. C'è la possibilità di aprire file **pcap** per leggere traffico salvato su file. In realtime bisogna controllare il tempo del pacchetto con il mio tempo. L'orario contenuto nel pacchetto è quello della sua ricezione, quindi nell'rrd è bene registrare i dati all'orario del pacchetto, non a quello di sistema

**pcap\_compile** Compila l'espressione del filtro

**pcap\_setfilter** Trasferisce nel kernel l'espressione compilata

**pcap\_next** Prende il prossimo pacchetto catturato

**Offset** Offset del pacchetto all'interno di Wireshark sarebbe il byte (esempio offset 12 → byte 12)

**BPF** Un esempio di filtro BPF compilato (TCP, in questo caso)

```
(000) ldh      [12]
(001) jeq      #0x86dd      jt 2 jf 7
(002) ldb      [20]
(003) jeq      #0x6         jt 10 jf 4
(004) jeq      #0x2c        jt 5 jf 11
(005) ldb      [54]
(006) jeq      #0x6         jt 10 jf 11
(007) jeq      #0x800       jt 8 jf 11
(008) ldb      [23]
(009) jeq      #0x6         jt 10 jf 11
(010) ret      #262144
(011) ret      #0
```

Di seguito un esempio UDP

```
(000) ldh      [12]
(001) jeq      #0x86dd      jt 2 jf 7
(002) ldb      [20]
(003) jeq      #0x11        jt 10 jf 4
(004) jeq      #0x2c        jt 5 jf 11
(005) ldb      [54]
(006) jeq      #0x11        jt 10 jf 11
(007) jeq      #0x800       jt 8 jf 11
(008) ldb      [23]
(009) jeq      #0x11        jt 10 jf 11
(010) ret      #262144
(011) ret      #0
```

**jeq** Jump if equal

**jt, jf** Jump if true, jump if false

**ldh** Load header

Inoltre, su Linux sniffare i pacchetti equivale ad aprire un socket.

# Capitolo 5

## Monitoring

### 5.1 Monitoring requirements

**Garantire la disponibilità dei servizi di rete** Bisogna mantenere l'attività del servizio anche in caso di variazioni tecnologiche o di carico. Inoltre bisogna impostare risposte automatiche o semiautomatiche in caso di anomalie: ad esempio modifiche alla configurazione in tempo reale, o attivazione di componenti ridondanti.

**Reazioni dinamiche a fronte di cambiamenti sulla rete o nell'ambiente** Applicare e far rispettare politiche, adattando il sistema (banda, memoria, processi...) in base ai cambiamenti o al carico.

**Controllo della rete** Collezione e compressione delle informazioni rilevanti sulla rete, mantenimento dei database di configurazione della rete, centralizzazione del controllo sulle periferiche (central management console, dove possibile) e integrazione delle procedure di manutenzione su ambienti eterogenei.

**Migliorare le condizioni di lavoro degli amministratori di sistema/rete** Standardizzando e migliorando gli strumenti, identificando e implementando la graduale automazione delle funzioni di manutenzione, integrando gli strumenti nelle esistenti sequenze di operazione.

**Progresso attraverso la standardizzazione** Transizione delle esistenti soluzioni (spesso proprietarie) verso ambienti standardizzati.

### 5.2 Vari attori, varie metriche

**Utenti diversi** Utenti diversi forniscono e richiedono servizi diversi (ad esempio, utenti finali vs service provider)

**Utenti finali** Monitoraggio delle performance delle applicazioni, verificare se le attuali condizioni di rete consentono di accedere al servizio, verificare se condizioni povere di rete sono attribuibili ad un attacco o se sono normali...

**Service provider** Monitorare i servizi e l'attività di rete, monitorare le violazioni dei contratti, rilevazione dei problemi di rete, riprogettazione della rete e dei servizi in base al feedback utente e risultati del monitoraggio, produrre previsioni per pianificare il futuro carico di rete quindi implementando estensioni prima che sia troppo tardi...

### 5.3 Applicazioni di analisi del traffico

#### 5.3.1 Richieste

**Cosa:** misurare volume e rateo per applicazione, host e conversazione

**Perché:** identifica la crescita e occorrenze anomale

**Cosa:** raggruppare il traffico in gruppi logici (compagnia, classi di utenti...), geografici (regione...), subnet...

**Perché:** associa il traffico con entità di business e trend di crescita per gruppo (dati aggregati non sono molto utili qua, bisogna fare analisi a livello utente)

**Cosa:** filtri ed eccezioni in base al traffico di rete

**Perché:** filtri associabili a notifiche nel caso si verificano occorrenze anomale sulla rete

**Cosa:** time-periods personalizzabili, per supportare il reporting nei giorni lavorativi

**Perché:** analizzare i dati in base al calendario aiuta a identificare i problemi (es: DHCP che finisce gli indirizzi ogni lunedì tra le 9 e le 10 del mattino, ma il problema sparisce nel resto della settimana)

### 5.3.2 Ulteriori problemi

**I dispositivi di rete hanno capacità di misurazione molto limitate** Prima di tutto deve smistare i pacchetti! Limitato a pochi protocolli, misure aggregate (ad esempio per interfaccia), reti ad alta velocità rendono il lavoro ancora più difficile.

**Costante sviluppo di nuovi servizi e applicazioni**, nella maggior parte dei casi non progettati per essere monitorati. La maggior parte del traffico in internet, inoltre, è composto da traffico progettato per essere difficile da rilevare e monitorare. Inoltre, i servizi internet moderni provengono da traffico mobile (quindi non legato ad IP e luogo), criptato e basato su porte TCP/UDP dinamiche (quindi niente fingerprinting, cioè corrispondenza 1:1 servizio:porta)

**Ulteriori problemi** Gli utenti richiedono le misure dei servizi, ma i dispositivi di rete forniscono misurazioni semplici e aggregate. Non si può sempre installare i dispositivi dove si vuole (cablaggio, privacy...) e nuovi protocolli ogni mese, mentre protocollo di monitoraggio molto statici e lenti ad evolversi.



## Capitolo 6

# Benchmarking per dispositivi interconnessi

**RFC 1944** Definisce come eseguire misure del traffico di rete:

- Architettura per il test (dove piazzare il sistema testato)
- Dimensione dei pacchetti usati per le misure
- Indirizzi IP da assegnare al SUT (**System Under Test**)
- Protocolli IP da usare per il test (es: UDP vs TCP)
- Uso di picchi di traffico durante la misura (picchi vs traffico costante)

Definisce e specifica come:

- Verificare e valutare i risultati dei test
- Misurare metriche comuni definite nella **RFC 1242** come throughput, latenza, frame loss
- Gestire "test modifiers" come
  - Traffico di Broadcast
  - Durata del test

### Metriche

**Disponibilità** La disponibilità è espressa come la **percentuale del tempo che un servizio è disponibile**. È la prima misura di affidabilità di un sistema. Si basa sull'affidabilità del componente di rete individuale.

$$\%disponibilità = \frac{MTBF}{MTBF + MTTR}$$

**MTBF** = mean time between failures

**MTTR** = mean time to repair following a failure

**Tempo di risposta** Quanto tempo ci mette il sistema a reagire ad una richiesta

**Accuratezza** Quanto posso essere accurato nel fare la misura?

**Throughput**

**Utilisation**

**Latency e jitter**

**ETSI** Definisce metriche che nel mondo internet non sono state definite.



## Capitolo 7

# Deep Packet Inspection

**Traffic classification** Traffic classification importante per capire cosa viaggia sulla rete. SNMP permette vedere se unicast o meno, ma non dice niente del traffico: buono, conosciuto...

Per classificare traffico quattro metodi:

1. Basato sulle porte TCP/UDP

Più o meno sempre fatta, specialmente nei primi anni di internet. Identificati da protocollo e porta nel range delle well known ports. Facile da evitare (porte dinamiche) quindi inaffidabile ( $TCP/80 \neq HTTP$ )

2. Flag del pacchetto (DSCP)

3. classificazione statistica

Di moda per un certo periodo, utilizzo del machine learning per classificare pacchetti. Si pensava che ML costasse meno rispetto analisi profonda del pacchetto. Però con ML errore statistico che rischiava, ad esempio, di bloccare traffico che non andava bloccato.

4. Deep Packet Inspection

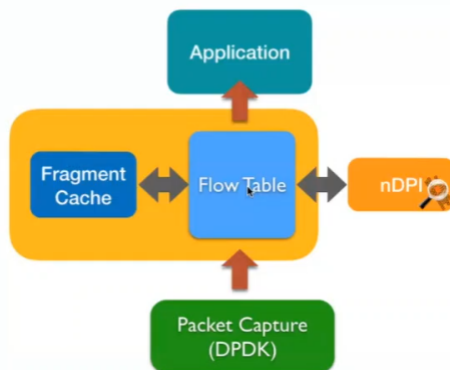
Analisi del pacchetto internamente. Selective metadata extraction (HTTP URL o User-Agent) necessario per fare monitoring accurato. Lo fa il DPI toolkit senza replicarlo sulle applicazioni di monitoring.

**Protocolli in nDPI** Identificati da `major.minor`

Major è protocollo di rete, minor è protocollo.

Oggi molti protocolli basati su HTTP e TLS. nDPI supporta riconoscimento protocolli basato su stringhe: DNS query name, HTTP campi host/server, SSL/QUIC SNI

I protocolli sono tantissimi, ma nDPI consente di raggrupparli in categorie, che possono includere migliaia di protocolli ed essere (ri)caricate dinamicamente.



**Come usare** Applicazione cattura il pacchetto e mantiene il flusso di stato, nDPI si aspetta di ricevere il pacchetto. Ogni disettore è codificato in un differente `.c`, ognuno classifica il singolo protocollo, per modularità ed estensibilità.

**Flow Lifecycle** La flowtable spesso esiste già. I flussi sono tenuti ordinati da qualcuno con una quintupletta+1 (VLAN, protocollo, IP/Porta, src/dst)

Applicazione va nella flowtable e cerca pacchetto precedente del flusso. Se esiste, prende l'informazione già memorizzata (cioè fin dove era rimasto nDPI su quel flusso), sennò si crea il nuovo flusso.

Se dopo un tot di iterazioni non riconosce ancora il protocollo lo cataloga come sconosciuto.

Il **riconoscimento** avviene **solo all'inizio del flusso**. Flussi mantenuti in hash sulla quintupletta.

Costo computazionale è sia numero pacchetti che numero di flussi

**Traffic Classification Lifecycle** Basato sul tipo di traffico, dissectors applicati sequenzialmente iniziando con quello che più probabilmente matcherà (es HTTP dissector se traffico su TCP/80)

Ogni flusso mantiene lo stato per dissector non-matching per saltarli in futuro.

Analisi dura fino a match o dopo troppi tentativi (8 pacchetti solitamente)

Flusso è bidirezionale e corrisponde alla quintupletta IP+prot+porta+src+dest(+vlan)...

Riconoscimento avviene solamente ad inizio del flusso.

nDPI funziona con i pacchetti IP.

**Latenza applicativa di rete** Calcolata utilizzando il TCP durante l'handshaking, monitorando con un probe in un punto della rete ma non si può dire se ho messo più tempo in andata che al ritorno.

Con sottrazioni trovo latenza rete e applicativa.



**Throughput** Metrica misura quantità dati inviati su un link in un certo istante.

Quant'è il tempo? Tipicamente al secondo ma la durata del monitoring dev'essere maggiore, si guarda al secondo ma la durata dell'esperimento dev'essere maggiore. Throughput dice indicazione idea di quanto è la banda disponibile.

Misura quantità di dati inviata in istante di tempo, **conservativa della banda disponibile** cioè misurando questo ho idea di quello che faccio effettivamente passare ma non della banda reale perché parte della banda la sto usando. Quindi **il throughput non è la banda**. Il throughput tipicamente cambia, la banda no. Inoltre è una misura orientata all'applicazione.

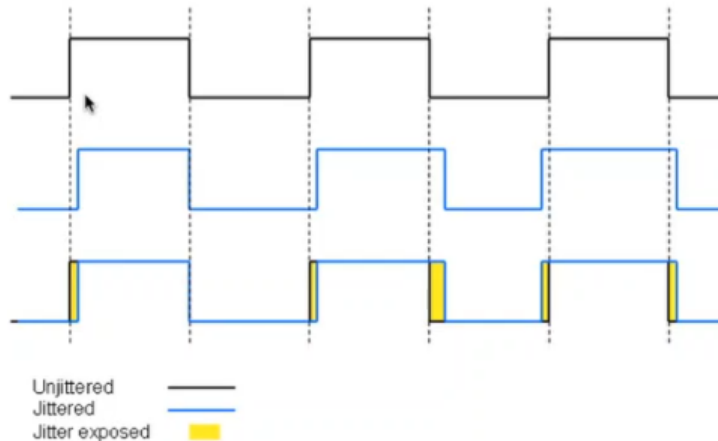
**Goodput** Nei pacchetti c'è tanto overhead: ethernet, header IP, UDP... fino al payload che è la **parte interessante**. Con il **goodput** si vede quanti dati effettivamente si spostano. Invece di misurare i byte inviati sul filo, vado a vedere i byte inviati di contenuto. Quando scarico un file di 1Mb ho mandato sul filo molto più di un 1Mb: header di ogni pacchetto, ACK per la connessione, RDY e FIN...

Goodput mostra quanti dei dati transitati sono effettivamente usati per la connessione.

**Latenza** Tempo che serve ad un pacchetto per andare **da sorgente a destinazione, monodirezionale**.

RTT (round trip time) è tempo di andata e ritorno. La latenza non è RTT/2, ma la somma della latenza src2dst e latenza dst2src è l'RTT.

**Jitter** Come cambia la latenza nel tempo: varianza dell'intrapacket delay su link monodirezionale. Finché jitter è inferiore alla durata temporale del buffer lato ricezione, la comunicazione sarà ideale. Se io tengo buffer grande (1-2s) la voce la sento bene risolvendo il jitter, ma ho aumentata la latenza. Questo perché i ritardi introdotti dalla rete non sono costanti.



**Bandwidth** La banda è quella misura che stima più o meno quello che dovrebbe passare sul cavo. Viene calcolata:

TC intervallo di misura, tipicamente 1s

BC burst committed, n max bit/s che il network accetta di trasferire nel TC

CIR committed information rate, banda garantita BC/TC

BE burst excess, parte che trasporta in eccesso se ha disponibilità proverà a trasportarne di più

MaxR maximum data rate:  $(BC + BE)/TC = ((BC + BE)/BC) * CIR$

**Misure End-To-End** Nelle misure devo mettere in conto il tempo della mia applicazione per elaborare ed anche il ritardo di rete.

Performance di rete  $\neq$  Performance app.

Bisogna fare attenzione rispetto alle misure per link, ad esempio perché traffico fa routing diverso andata-ritorno quindi  $rtt/2 = \text{latenza}$  è in generale sbagliato. misure endtoend sono più vicine all'utente, si rapporta al server remoto non alla misura link link. Quella link link serve a noi per fare troubleshooting.

EndToEnd e LinkLink sono in relazione, endtoend dà se c'è problema che si scorpa sul link. EndtoEnd dà informazione dal punto di vista utente, link dà info sul singolo link utile per troubleshooting.

**Approcci di monitoraggio** Sono multipli. fin'ora è una misura di tipo passivo: si cattura traffico, senza influenzare niente.

Nella misura attiva inietto del traffico e vedo la reazione della rete, ad es. un ping.

Ad es traceroute calcola latenza e da dove passa un certo tipo di traffico. Usa un trucco sul TTL: manda pacchetti che tornano indietro a causa del TTL basso.

**Inline** Inline significa quando monitoriamo se tramite snmp connesso allo switch con lo stesso filo che snmp sta monitorando: **inline**.

Offline invece è monitoring tramite altra rete: reti sdoppiate, una per monitorare e una per i servizi. Vincolata dalla struttura della rete, a volte non possibile.

Quindi approcci attivi/passivi, inline/offline.

## 7.1 Calcolo delle serie temporali

Posso fare una misura mettendo una soglia assoluta, ma posso anche impostare queste soglie a seconda del comportamento della rete.

La media è la misura media, la deviazione standard è una misura "standard" di cosa è normale e di cosa è molto grande ( $> \text{media} + \text{devstd}$ ) o molto piccolo ( $< \text{media} - \text{devstd}$ )

**Percentile** Quando si prende una misurazione, es. quantità della banda, nel caso delle rete il 95esimo percentile è interessante. **Percentile** è valore sotto il quale ricade una certa percentuale di osservazioni.

Esempio: l'80esimo percentile è l'età in cui cade l'80% della popolazione.

Perché importante? Se devo parlare ad esempio della banda, posso calcolare economicamente parlando per vedere se quella banda è utilizzata, o per vedere se la banda che metto a disposizione è utilizzata. Quando arriva il momento di fare un upgrade, ad esempio? Oppure anche usabile per far pagare delle penalty.

Il 95esimo percentile di una rete è la soglia sotto cui sono stato per il 95% del tempo.

Calcolo: data una serie di valori, si definisce il percentile (es. 80), si ordina la serie dal basso all'alto, calcolo indice che corrisponde al percentile ( $\text{len}(\text{serie}) * (\text{percentile}/100)$ ), prendo l'indice intero (es  $\text{int}(\text{index} + 0.5)$ )

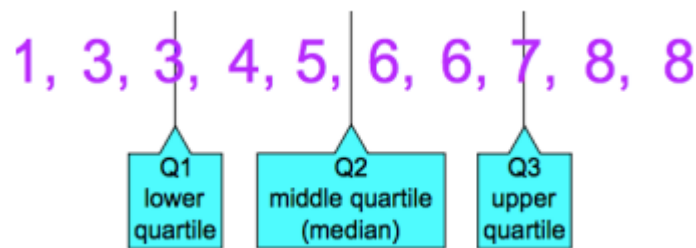
Percentile = serie\_ordinata[indiceintero-1]

**Quartile** Definiti così:

$Q_1$  è il 25esimo percentile

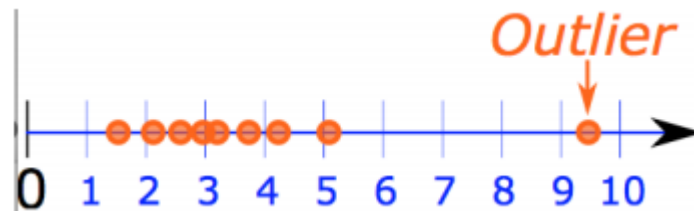
$Q_2$  è il 50esimo percentile

$Q_3$  è il 75esimo percentile



**Outlier** Ciò che devia un sacco dallo standard.

Punti che discostano tantissimo rispetto all'average. Si usa l'IQR (InterQuartile Range):  $Q_3 - Q_1$



In statistica, un valore è un outlier quando cado fuori da

Lower Fence:  $Q_1 - 1.5 \times IQR$

Upper Fence:  $Q_3 + 1.5 \times IQR$

Si usano per trovare valori anomali, come errori di misura o sperimentali, o distribuzioni "heavy tailed", cioè che vanno a zero molto lentamente poiché una o più valori molto grandi hanno effetto sulla statistica.

**Calcolo del jitter**  $\text{jitter} = \text{sum}(|x(i) - x(i-1)|) / (n-1)$

Se il jitter è nell'ordine di grandezza del valore allora la situazione è drammatica. Jitter basso: valore costante. Jitter alto: valore cambia spesso nel tempo.

### 7.1.1 Timeseries, Forecast, Anomaly Detection

Negli RRD c'erano dei valori  $\alpha$ ,  $\beta$ ,  $\gamma$  che avevamo detto di lasciare quelli di default. Sono serie temporali su cui vogliamo fare delle previsioni, che ci forniscono le anomalie. Quand'è che una cosa è anormale? Quando ho il concetto di normalità. Ad esempio, una soglia. Quindi come trovare le condizioni tali in cui il sistema che sto guardando si comporta stranamente?

**Definizioni:**

**Serie:** sequenza ordinata di numeri, non sorted ma con indici

**Ordine:** indice di un numero in una serie

**Serie temporale:** serie di punti ordinati temporalmente

**Osservazione:** valore numerico osservato (nella realtà) in un momento specifico

**Previsione:** stima di un valore atteso (che non sappiamo ancora) in un momento specifico (*alle 12 quanto traffico farò?*)

**Anomalia** quando la **previsione** scosta dall'**osservazione**

**Forecast error:** differenza positiva/negativa nell'osservazione rispetto alla previsione. Solitamente l'errore è riportato come quadrato per avere sempre numero positivo.

**SSE:** la somma degli errori quadrati (**sum of squared errors**) di una serie  $\sum((\text{osservazione}_i - \text{previsione}_i)^2)$   
Idealmente deve essere 0, o comunque basso

Dato  $y$ ,  $\bar{y}_{x+1}$  è la previsione di  $y$  al tempo  $x + 1$

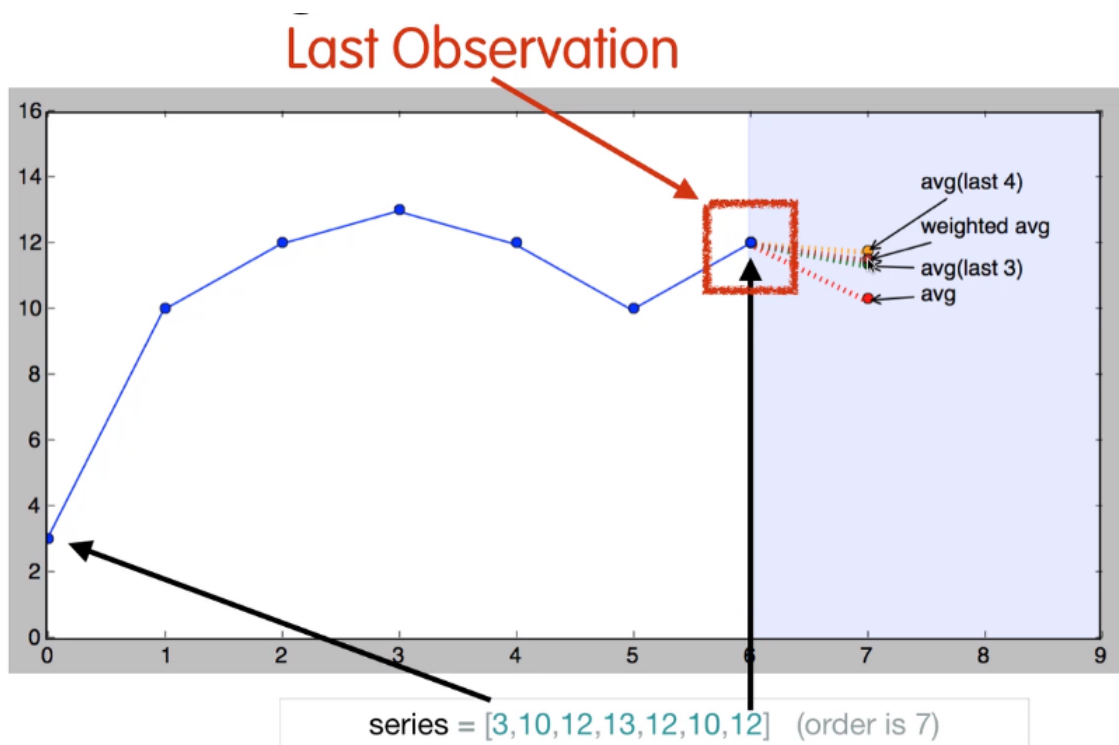
**Simple Average:** media di tutti i punti della serie

**Moving Average:** media degli ultimi  $n$  punti

**Weighted Moving Average:** come la media mobile ma con i valori pesati (tipicamente pesando di più i valori più recenti)

**Anomalia** quando la **predizione** v'è in conflitto con l'**osservazione**.

**Predirre il futuro**



**Single Exponential Smoothing**  $\bar{y}_x = \alpha \cdot y_x + (1 - \alpha) \cdot \bar{y}_{x-1}$

$\alpha$  è un fattore di **ammorbidimento** o **memory decay rate**, più alto è più tengo presente gli ultimi valori e meno tengo presente i vecchi.

$\bar{y}_x$  è chiamato livello  $l_x$

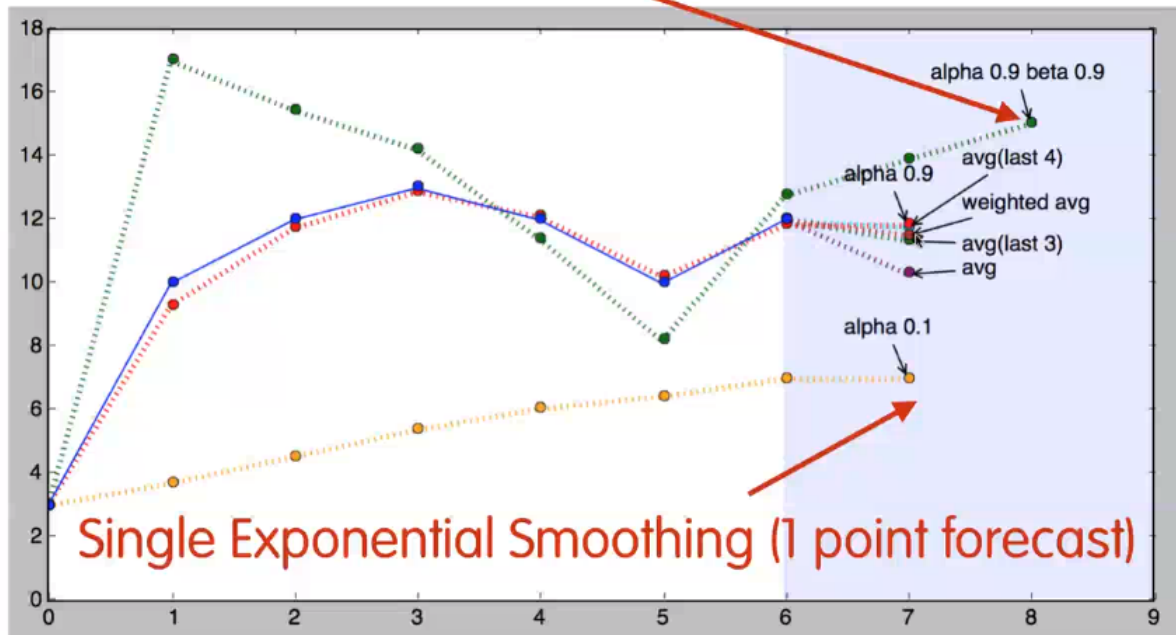
**Double Exponential Smoothing** Trend  $b$  su una serie temporale, fra due punti è  $y_x - y_{x-1}$   $b_x = \beta \cdot (l_x - l_{x-1}) \dots$  Con  $\beta$  non prendo solo la media ma tengo conto anche del trend, così da avere previsioni più accurate. Quindi alla previsione si aggiunge  $b_x$

$$\text{Livello } l_x = \alpha \cdot y_x + (1 - \alpha) \cdot (l_{x-1} + b_{x-1})$$

$$\text{Trend } b_x = \beta \cdot (l_x - l_{x-1}) + (1 - \beta) \cdot b_{x-1}$$

$$\text{Previsione } \bar{y}_{x+1} = l_x + b_x$$

## Double Exponential Smoothing (2 points forecast)



## Holt-Winters Method Anche detto Triple Exponential Smoothing

Nel double exponential smoothing manca il concetto di **stagione**: quando una serie è ripetitiva a intervalli regolari si dice **stagionale**.

Lunghezza della stagione: numero dei data point in una stagione.

Componente stagionale: deviazione da level+trend del double exponential che si ripete all'interno della stagione.

$\gamma$  fornisce il concetto di stagionalità. Serve il modulo della durata della stagione, perché è in loop. Inoltre con H-W posso **predire un numero arbitrario di punti**. valori più grandi danno previsione meno conservativa.  $\alpha, \beta, \gamma$  si assegnano tramite il **fitting**.

$$\text{Livello } l_x = \alpha \cdot y_x + (1 - \alpha) \cdot (l_{x-1} + b_{x-1})$$

$$\text{Trend } b_x = \beta \cdot (l_x - l_{x-1}) + (1 - \beta) \cdot b_{x-1}$$

$$\text{Stagionale } s_x = \gamma \cdot (y_x - l_x) + (1 - \gamma) \cdot s_{x-L}$$

$$\text{Previsione } \bar{y}_{x+m} = l_x + m \cdot b_x + s_{x-L+1+((m-1) \bmod L)}$$

Per prevedere un numero arbitrario  $m$  di punti.

$\alpha, \beta$  e  $\gamma$  sono i parametri di adattamento per l'algoritmo, con  $0 < \alpha, \beta, \gamma < 1$ . Valori più grandi rendono l'algoritmo in grado di adattarsi più rapidamente e fanno in modo che le previsioni riflettano di più i cambiamenti recenti nella time serie. Valori più piccoli, invece, rendono l'algoritmo più lento all'adattamento, rendendo più importanti i valori passati nell'history.

I valori per  $\alpha, \beta$  e  $\gamma$  possono essere determinati provando a determinare la più piccola SSE (Sum of Squared Errors) in un processo iterativo chiamato fitting (simile al ML).



**Deviazione in H-W** La deviazione può essere identificata quando l'osservazione cade al di fuori delle fasce di confidenza minima e massima per un dato punto.

$$d_t = \gamma \cdot |y_t - \bar{y}_t| + (1 - \gamma) \cdot d_{t-m}$$

Fasce di confidenza con  $\delta$  tra 2 e 3:

$$\text{Superiore: } \bar{y}_t - \delta \cdot d_{t-m}$$

$$\text{Inferiore: } \bar{y}_t + \delta \cdot d_{t-m}$$



## Capitolo 8

# Remote Monitoring

### Analisi rete remota rispetto dove siamo noi.

Importante perché tutta la parte interessata dalle informazioni sta alla periferia di internet (**edge computing**), inoltre è anche "forzato" dalle parti wireless che stanno diventando predominanti. Dal core ci passa sì traffico ma è sempre minore. Il core diventa meno importante perché comunicaz in perferia.

Diventa necessario . . . Inote tipi traffici che nascono e muoiono in periferia senza passare dal core.

Il problema è andare a spillare il traffico remoto, vedere il traffico che fluisce in questa rete. Nn sempre banale perché magari necessario modificare la rete. Quindi attenzione alla topologia, e attenzione ai tap di rete usati perché hanno limitazioni importanti.

Anche le funzionalità di monitoraggio degli apparati di rete comuni sono limitate, spesso il massimo che posso vedere sono i byte totali, ip assegnati. . .

Spesso i costi per sistemi di monitoring sono importanti.

Misura in remoto tramite snmp nasce subito dopo mibv2. con snmp misuro quanto ma non posso fare misure più complicate, e non posso assegnare soglie e in base mandare allarme ma misurare in continuazione e fare i nostri calcolo. MIB RMON (MIB SNMP), risolve problema monitoraggio remoto. Consente di gestire disservizi. Inserisce un minimo di logica per generare allarmi.

## 8.1 RMON

Cosa fa? Primo passaggio da monitoraggio snmp a monitoraggio a pacchetti.

Vedo traffico di rete remoto e metto una sonda remota.

**RMON è un MIB SNMP.**

**RMON vs SNMP** SNMP controlla e configura un probe, solitamente con GUI. SNMP esegue la **richiesta periodica** (polling). L'azione è eseguita dal manager.

RMON sposta un po' della logica nell'apparato, che raccoglie da sé stat e impostare allarmi direttamente sull'apparato.  
**Gruppi RMON**

Statistics: vedo info che SNMP non dà precisamente, es pacchetti droppati

History: dati i contatori (es. quante volte una porta cambia stato) consente di non andare sullo switch e leggere e scrivermeli ma chiedere la storia recente del contatore. Memoria piccola, 3-5-10 misure, poche. RMON legge un po' nel passato, SNMP legge sempre e solo in real-time

Alarm:

Host: a livello di host quali traffici per un certo host, es. byte in out per certo host.

HostTopN: per i top N host quanto traffico fanno, quale porta più occupata. Se c'è connesso un PC è il colpevole, ma se ce ne sono N non mi è chiaro. Con HostTopN analizzo traffico della porta e trovo N computer che fanno più traffico. . .

Matrix: mostra comunicazioni tra indirizzi

Filters: per filtrare il traffico di rete

Packet capture: per catturare un pacchetto una volta che è passato attraverso il canale

Events: per generare eventi sotto certe condizioni

**Statistiche ethernet RMON** Informazioni di basso livello

Pacchetti: unicast, multicast e broadcast.

Eventi di drop: pacchetti droppati per sovraccarico linea/porta

Frammenti: quanti frammenti sono passati. Ci sono attacchi basati sui frammenti.

Jabbers: pacchetti ricevuti più lunghi di 1518 ottetti e con errori di allineamento

Oversized packets: pacchetti più lunghi di 1518 ottetti ma altrimenti ben formati

C'è anche una formula per capire quanto è usata una porta di rete

$$usage = \frac{100 \cdot [(packets \cdot 160) + (octets \cdot 8)]}{portspeed \cdot timeinsec}$$

**Allarmi** Impostare soglie su contatori di RMON, soglie di due tipi: rising e falling. Quando troppo alta (rising) o bassa (falling) posso effettuare operazioni.

Es troppo traffico se utilizzo sup a 80% manda allarme.

**Case Study** ... RMON va a mappare ogni quanto misuro dati, valori di soglia

**NBAR** Statistiche di traffico stile RMON

## 8.2 Real-Time Flow Measurement

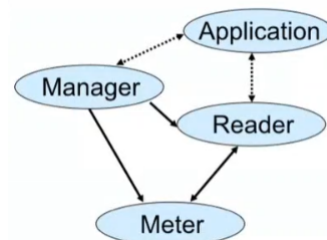
Cambio di architettura per andare oltre l'architettura SNMP

Meter: colui che misura, il contatore. V'è messo dove passa il traffico

Reader: parla col meter e tira fuori i suoi dati. Deve essere "vicino" al meter, ma non è dentro

Manager: gestisce configurazione di meter e reader e che chiede i dati. Implementato in SNMP

Applicazione: fa i grafici, configura il manager...



## 8.3 Monitoraggio a flussi

Nel caso di SNMP agent (dove passa traffico) e manager (query all'agent). Nel caso di un problema, agent avverte manager tramite trap (cambio di stato, o superamento soglia in caso di RMON).

Agent snmp non è libero di monitorare tutto, si limita al MIB.

Nei flussi prendo traffico di rete, lo metto insieme in base a caratt (es protocollo, stesso ip:porta sorg, stesso ip:porta dest = quintupletta)

Non configuro il flusso. Nel caso di NetFlow la device stessa dove passa il traffico di rete (es router) prende e spara verso collezionatore (manager) le informazioni da monitorare. In questo caso quindi i flussi sono prodotti ed emessi verso colui che deve riceverli.

No concetto allarme né abilità per agents di fare alcunché: tutta la logica è nel collezionatore.

Strumentazione del probe è fatta offline.

Probe sono posizionati dove fluisce il traffico, tipicamente nel router.

**Cosa si misura?** Dove viene scambiato il traffico organizzato per gruppi autonomi, IP...

A livello applicativo quanto traffico in base a porta o protocollo

Quali servizi, livelli di traffico... anche rintracciare virus di rete fino agli host.

Con i flussi non si riesce a misurare traffico nonIP, informazioni livello 2 (stato delle interfacce), traffico filtrato (firewall), statistiche per-link (livello di utilizzo, congestione, perdita...), statistiche per applicazioni (es latenza, risposte positive/negative, errori di protocollo).

**Flusso** : insieme di pacchetti accomunati da una stessa quintupletta. Flussi sia unidirezionali che bidirezionali. Calcolati dal probe (era: agent) che guarda traffico che passa attraverso e lo misura. Flusso creato quando vedo il primo pacchetto del flusso, probe sta fermo.

Massima durata flusso regardless lo stato della connessione: termino flusso o quando dura troppo o quando non c'è traffico per un po' di tempo.

Contenuti del flusso: peers (src e dst), contatori (packets, bytes, tempo), info routing (sistema autonomo, netmask, interfaces).

Possono essere uni e bidirezionali. Due flussi unidirezionali opposti sono equiv a uno bidirezionale. Flussi bid possono contenere altre info come rtt o comportamento tcp.

#### **Problemi vari**

Overhead vs accuratezza

- + misure + dati

- + aggregazione flussi - granularità

- overhead su router, switc e endhosts

Security vs data sharing

- flussi devono andare ai collezionatori su cammini protetti

- privacy va rispettata

- misure di traffico devono essere mantenute protette per non rivelare informazioni di rete importanti a terze parti



# Capitolo 9

## ScaPy

Manipolazione di pacchetti in maniera semplice. Con Pcap si legge e invia, ma gestisce i pacchetti raw e non permette di manipolarli, funge da specie di socket. Wireshark è molto utile ma è un tool a sé state, ma se serve fare cose in batch non è molto utile. Wireshark può essere collegato tramite API a PyShark, ma è sola lettura. ScaPy permette sia di inviare che di ricevere.

**Modulo** ScaPy è un modulo Python.  
Cosa ci possiamo fare?

### Manipolazione dei pacchetti

Ha un operatore particolare, / che permette di creare un pacchetto, esempio: `packet = IP() / TCP()` che crea un pacchetto IP e sopra ci mette una parte TCP. Senza i parametri lo crea con opzioni di default (es IPv4) e non bisogna preoccuparci granché dei campi non riempiti, possiamo spedirlo (anche se non va da nessuna parte perché non ha indirizzo di destinazione).

Con `ls(IP, verbose=True)` stampa i vari campi, con tipo e valore. Più o meno è la stessa informazione vista con Wireshark e Tshark.

Volendo posso stampare un campo voluto, ad esempio con `print(p[IP].src)`.  
`summary()` stampa tutti i campi del pacchetto.

### Interagire con la rete

Permette di inviare e ricevere i pacchetti, oltre a crearli. La funzione per inviare traffico è `srl()`, ad esempio `r = srl(IP(dst="8.8.8.8") / UDP() / DNS(qd=DNSQR()))`.

`srl()` riceve anche la risposta, che nell'esempio sarà memorizzata in `r`, che posso leggere ad esempio con `r[DNS].an`.

`srp()` invece consente anche di specificare la parte ethernet. Spedisce una lista di frame e ritorna due variabili

`r`, lista di query e risposte

`u`, lista di pacchetti non risposti

### Pcap

Possiamo leggere e scrivere un file pcap, con `wrpcap("file.pcap", r)` e `rdpcap("file.pcap")...`

Con `command()` su un pacchetto pcap, ritorna il comando da fare con ScaPy per ricreare un pacchetto di pcap.

### Sniffing

Con il comando `sniff()`, applicando filtri, funzioni...

### ARP

Con `arping("192.168.1.0/24")` per mandare richieste ARP sulla rete specificata. La differenza con ICMP è che il ping classico può essere ignorato. Con ICMP quando c'è risposta la macchina è attiva, quando non c'è non si può sapere.

### AnsweringMachine

Permette di gestire le risposte, un esempio è avviarla in monitoring su un'interfaccia.





# Capitolo 10

## Flussi

**Flussi unidirezionali con IP, Porta e protocollo come key** Avrò più flussi perché distinguo anche i protocolli usati e le porte, quindi ho maggiore informazione sul traffico.

### 10.0.1 NetFlow

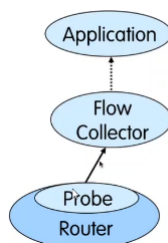
Architettura a flussi, prima coniata da Cisco. Funziona diversamente da ciò che si è visto fin'ora. Abbiamo un router con un probe (sonda di rete, traduce pacchetti in flussi). La funzionalità di misura è secondaria (il router deve andare bene), ma catalogare i flussi richiede memoria. Router calcola i valori con il probe ma non ha capacità di memorizzazione: manda verso Flow Collector (un pc con un software apposito). Il collezionatore riceve flussi e fa qualcosa: es. manda dati a interfaccia web.

C'è anche problema se si monitora rete abbastanza grande. Si possono mettere più collezionatori che collaborano fra loro, ognuno attaccato ad un endpoint (router verso l'esterno) della rete.

**Spazio** Lo spazio richiesto dipende dal traffico. Qualche valore solito:

67320 ottetti/flusso, 92 pacchetti/flusso

Router occupato: 367GB traffico al giorno, 548000000 pacchetti al giorno, cioè 5900000 flussi al giorno



**NetFlow** Flussi unidirezionali fino a v8, bidirezionali dal v9. La più comune è v5, l'ultima è v9.

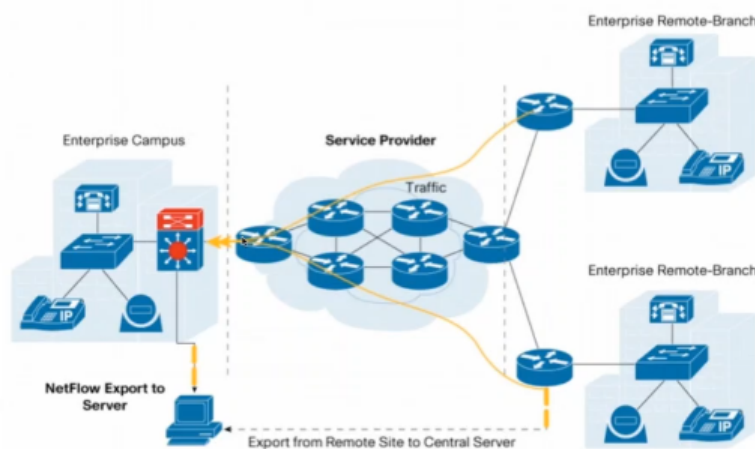
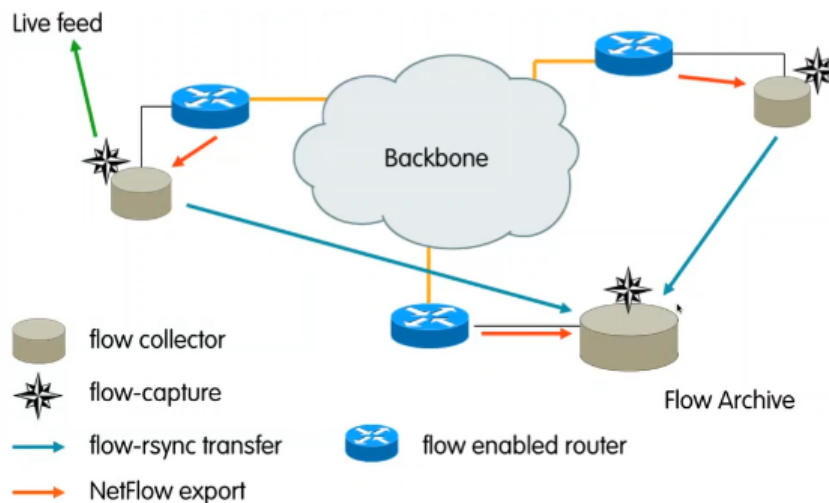
Analisi solo del traffico inbound e IP (non su tutte le piattaforme). IPv4 unicast e multicast. IPv6 solo su v9. La v9 è aperta, cioè integrabile con moduli personalizzati.

Protocollo aperto definito da Cisco e supportato su IOS e CatIOS e su altre piattaforme Cisco.

Ogni versione ha un formato di pacchetti: fino alla 8 era chiuso, dalla v9 è dinamico e aperto a estensioni.

Numeri di sequenza: v1 non ha numeri di sequenza, fino alla 8 numeri di sequenza per flusso e dalla 9 numeri di sequenza per pacchetto (non flusso)

Qualche versione è specifica per alcune piattaforme di Cisco.



**Usare i flussi** Cosa c'è dentro un flusso? Questa domanda risponde anche a cosa ci faccio con essi. Nella parte di chiave: la quintupletta

Protocollo

IP src/dst

Porta src/dst

Altre info

QoS: Flag TCP, tipo protocollo e tipo di servizio

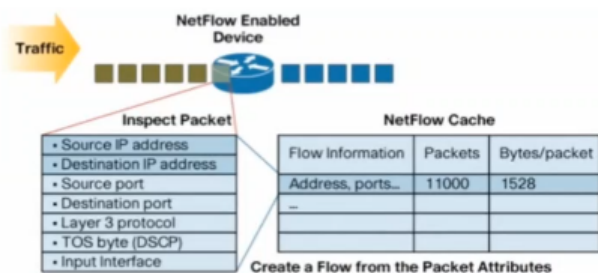
Porta: indice di porta in e out

Info di routing: next hop, da/verso quale AS...

Contatori: di byte e di pacchetti

Tempo: sysuptime start/end

Usage	<ul style="list-style-type: none"> <li>Packet Count</li> <li>Byte Count</li> </ul>	<ul style="list-style-type: none"> <li>Source IP Address</li> <li>Dest IP Address</li> </ul>	From/To
Time of Day	<ul style="list-style-type: none"> <li>Start SysUptime</li> <li>End SysUptime</li> </ul>	<ul style="list-style-type: none"> <li>Source TCP/UDP Port</li> <li>Dest TCP/UDP Port</li> </ul>	Application
Port Utilization	<ul style="list-style-type: none"> <li>Input ifIndex</li> <li>Output ifIndex</li> </ul>	<ul style="list-style-type: none"> <li>Next Hop Address</li> <li>Source AS Number</li> <li>Dest AS Number</li> <li>Source Prefix Mask</li> <li>Dest Prefix Mask</li> </ul>	Routing and Peering
QoS	<ul style="list-style-type: none"> <li>Type of Service</li> <li>TCP Flags</li> <li>Protocol</li> </ul>		



**Nascita e morte di un flusso** Siamo a livello 3. Il traffico deve entrare e uscire dal router, quindi interf ingress e uscita: il traffico è esaminato, cioè passato al probe. Posso mettere nel pc di casa un software per sniffing, ma nel router i pacchetti passano attraverso di lui. Il probe prende dati e li raggruppa, li gestisce. Si tiene una cache dei flussi attivi. Appena c'è pacchetto non conosciuto la popola. Al suo interno ha sistema ispezione pacchetti. In cache c'è una sorta di hashtable, come chiave ho ip porte..., come valore ho *almeno* i byte e i pacchetti.

Terminano quando:

La comunicazione di rete termina (es: FIN)

Dura troppo (default 30m)

Non attivo da troppo (default 15s)

Cache piena e necessario purge (è limitata, non è raro non poter gestire tutti i flussi)

Perché taglio quando il flusso dura troppo? Sennò manderei i dati tutti alla fine, con picchi di traffico, e sballerebbe le misura.

**Pacchetti NetFlow** Header comune rispetto alle varie versioni. A seconda della versione c'è un campo che specifica il numero di record  $N$ .  $N$  è determinato dalla dimensione massima del payload UDP (circa 1480 byte),  $N = 30$  per v5. Nessuna frammentazione.

Header contiene:

**version** es: 5

**count** (numero di record nel payload)

**sysuptime** del router (da quanto è acceso)

**unix\_secs** (epoca del flusso, secondi da 0000 UTC 1970)

**unix\_nsecs**

**flow\_sequence** (numero di flussi inviati, per sapere se collezionatore si è perso qualcosa non sa cosa ma sa di aver perso qualcosa)

**engine\_type** per caratterizzare tramite info su chi ha generato i dati

**engine\_id** numero di slot per l'engine

Flusso contiene:

**srcaddr** ipv4

**dstaddr** ipv4

**nexthop** (per sapere come viene routato traffico dentro il pc:

**input** indice di interfaccia snmp

**output** indice di interfaccia snmp

**dPkts** numero pacchetti

**dOctets** numero ottetti

**First** sysuptime dell'inizio del flusso

**Last** sysuptime dell'ultimo pacchetto

**srcport** TCP/UDP porta origine

**dstport** TCP/UDP porta destinazione

**pad1** padding

**tcp\_flags** or cumulativo dei flag tcp

**prot** protocollo ip (6=TCP, 17=UDP...)

**tos** type-of-service

**src\_as** as origine

**dst\_as** as destinazione

**src\_mask** bitmask source route

**dst\_mask** bitmask dest route

**pad2** padding

**Pacchetto NetFlow** In SNMP dati letti tramite polling (eccettuate le trap). Cambi di stato SNMP sono pochi, agent SNMP non fa attesa attiva ma è S.O. stesso a comunicare a SNMP quando cambia qualcosa. Es: Inotify ([wikipedia.org/wiki/Inotify](http://wikipedia.org/wiki/Inotify), notifica cambiamenti a file/cartelle) o netlink ([wikipedia.org/wiki/Netlink](http://wikipedia.org/wiki/Netlink), notifica cambiamenti su rete e interfacce) su Linux.

In NetFlow però siamo obbligati ad analizzare attivamente ogni pacchetto che passa. Inoltre buffering limitato, quindi pacchetti vanno gestiti subito. Inoltre la quantità di flussi da esportare dipende molto dal tipo di traffico. Può esserci un solo flusso su 10 gigabit, ma posso avere decine di flussi su 7 megabit.

Il traffico conteggiato da NetFlow è inferiore da quello che verrebbe contato da SNMP. SNMP ne cattura di più in caso ci sia traffico multicast/broadcast. Inoltre SNMP è livello 2 (comprende il payload del livello frame) ma NetFlow è livello 3, non conta il traffico di livello frame. Ad esempio ARP non lo vedo in NetFlow perché è di livello 2.

**NetFlow conta solo il traffico che passa attraverso il router, cioè che viene instradato.**

**Perché serve NetFlow v9?** La v5 ha solo IPv4, poche informazioni (pacchetti e byte). Ogni volta che si deve aggiungere qualcosa bisogna rifare il formato, aggiornare software...

In v9 risolvono problema della flessibilità: flussi mandati con un template che indica i campi. Inoltre supporta il livello 2, traffico VLAN e altri tipi di incapsulamento come MPLS, oltre che IPv6.

**Template** Mandato prima dei flussi, così da specificare al collezionatore in che modo interpretare i dati ricevuti. Specifica una maniera di mappare le info relative al flusso. Vengono mandati periodicamente.

**Sampling** Per ridurre il carico sulla macchina introduco il sampling: non analizzo ogni pacchetto, ma ad esempio uno su 10. Questo introduce un errore, sia sul conteggio dei byte che sulle informazioni trovate. Sampling da usare solamente in situazioni particolari.

Si parla di **packet sampling**, ma si può fare anche **flow sampling** (esempio: esportarne uno su 10). Questo per ridurre il traffico sul collezionatore.

**Principi del v9** Modello push probe verso collezionatore come in v5

Manda template regolarmente (ogni tot flussi, ogni tot secondi)

Indipendente dal protocollo sottostante (UDP/TCP)

Può andare sia template che record in una export

Può mischiare le informazioni sui flussi nel singolo record.

Posso inserire nuovi campi oltre counting di pacchetti e byte: contenuto, velocità...

Si aggiorna *ogni tanto*, all'esportazione del flusso (tipicamente scadenza timeout)

Salta 24Byte perché parte da livello 3.

**Message Broker** Prende messaggi e li smista a seconda di caratteristiche a chi si è sottoscritto, volendo anche riformandoli in formati diversi.

**Configurazione Cisco IOS** Configurato su ogni interfaccia, specificando versione e IP del collezionatore. Specifico sampling rate, durata massima e aggregazione ( Si abilitano flussi basandoli sull'interfaccia.

**Configurazione Juniper JunOS** Si appoggia al firewall, quando pacchetti attraversano router (eventualmente non filtrante). Router divisi in data/switching plane (hw) e control plane (sw). Sorta di loopback interno per analisi traffico, per portarlo da data a control plane. Juniper usa questa interfaccia per portare pacchetti in parte di controllo, limitata a 7000 pacchetti al secondo per evitare sovraccarico. Nello spazio utente c'è il probe che calcola i flussi ed esporta.

## 10.0.2 IPFIX

**NetFlow generico: IPFIX** A inizio 2000 perché NetFlow è di Cisco

**IPFIX** definito dalla IETF.

Basato su NetFlow v9, abilità di definire nuovi campi del flusso attraverso un formato standard (OID, Object Identifier). Dopo aver definito il "PEN?" specifico OID univoci per me e in generale.

Protocollo di trasporto (SCTP, Stream Control Transport Protocol) orientato allo stream, connection-oriented, vincoli rilassati così da non perdere eventualmente troppo tempo a riempire il pacchetto. Opzionalmente supporta TCP/UDP. Attualmente: bozza di specifica del protocollo.

**Fondamentalmente**, IPFIX = NetFlow v9 su SCTP con qualche differenza extra.

## 10.0.3 Flow Aggregation e Filtering

**Aggregazione flussi** I flussi raw ok esporto e via, ma sono limitanti. Spesso dobbiamo rispondere a domande diverse: quanto traffico certo protocollo? Quando verso Google?

Una volta che ho i flussi dentro il router posso esportarli raw o aggregarli: accorpate fra loro flussi a seconda di criteri stabiliti (es: IP, maschio le altre colonne come se non ci fossero ad esempio con valore fittizio 0 e sommando eventuali contatori). **Risparmio spazio e trasmissione.** Perdo informazione, perché non posso separare successivamente i flussi ormai aggregati.

Permette di trovare forme tipiche per rispondere a domande tipo: porte più usate, chi parla con chi...

Una volta aggregati i dati, se fatta bene, risparmio molto spazio nella cache.

Se l'aggregazione la fa il probe (**probe aggregation**), si migliora l'utilizzo della memoria da parte del probe NetFlow ma il collector non potrà risalire ai singoli flussi. Se l'aggregazione la fa il collector (**collector aggregation**) allora svolgerà più lavoro, ma potrà risalire ai singoli flussi.

**Filtraggio flussi** Scartare flussi basandosi su criteri: durata, src/dest, porte. . .

Magari esportare solo unidirezionali. Diverso da aggregazione, possono coesistere e solitamente filtraggio applicato prima dell'aggregazione.

**Esempi di intrusione** Flussi con pacchetti o ottetti eccessivi conteggio

Src con tante destinazioni (host scanning) o con tante porte dest su stesso host (port scanning)

#### 10.0.4 sFlow

NetFlow nasce per router e traffico instradato, ossia nelle reti geografiche e non per le reti LAN. Si potrebbe mettere una sonda per analizzare ogni device, ma si avrebbe una cospicua quantità di traffico da analizzare, oltre che ad un aumentato costo dei dispositivi di rete, la scalabilità. . .

NetFlow è nato soprattutto per la parte geografica, non per la LAN. Se voglio sapere quello che succede in rete, il traffico non passa dal router (non vedrò con NetFlow ciò che passa in casa), ma si può applicare anche alle LAN.

sFlow segue questi principi:

Non vuole essere veloce quanto la rete monitorata.

Anche se riuscisse a rilevare tutto il traffico, avrebbe problemi a gestire i flussi generati.

Se ho 8 porte gigabit, ognuna ha 16 gigabit di traffico (8 gigabit andata e 8 gigabit ritorno) e diventa difficile gestirlo. Devo **deduplicare**: dividerlo a metà per le due direzioni.

Effettua il **sampling** sui pacchetti, sapendo che più pacchetti vede più i suoi report saranno precisi.

Il sampling è effettuato su tutte le porte, in modo da avere la stessa proporzione, ma con intervallo casuale, così da non rischiare di avere un traffico periodico e analizzare solo quello.

La **perdita** dei pacchetti è **controllata**, cioè si sa perché quei pacchetti vengono persi.

**Architettura** Si basa su un **probe** che campiona il traffico: i pacchetti campionati sono mandati all'**sFlow collector** in formato sFlow, periodicamente il probe manda al collezionatore le statistiche SNMP MIB-II all'interno dei pacchetti sFlow.

I pacchetti sono usati per scalare il traffico. Il sampling è fatto per porta, per avere visibilità su ognuna.

La differenza è che **il probe manda il pacchetto ogni  $n$  con l'header sFlow e lo manda al collezionatore, senza analizzarlo**. Ci permette di andare veloci.

RFC 3176, definisce formato pacchetti (UDP, non SNMP) e un MIB SNMP per accedere i dati. Architettura simile a NetFlow: probe manda **pacchetti** al collezionatore.

probe fondamentalmente sniffer che cattura 1 ogni  $x$  pacchetti (tipicamente  $x = 400$  e incapsula con info sFlow).

$$\text{Errore statistico} \leq 196 \cdot \sqrt{\frac{1}{\text{numero dei sample}}}$$

Scalabile (aumento il rateo di sampling).

**sFlow  $\neq$  NetFlow** Flussi sFlow e flussi NetFlow **non hanno niente in comune**. In sFlow un flusso è un pacchetto, in NetFlow un flusso è una quintupla con contatori.

Oltre a vedere il pacchetto può mostrare anche info sul contesto agendo sullo switch.

AS source e destination relative ad ip src/dst in netflow

in sflow ho as del router che lo invia e as che lo deve ricevere

Con sflow vedo parte del traffico ma con **più informazioni estratte dal contesto**

Si riesce a sapere, sugli AP, anche user e pwd. Info riguardo al traffico come gira nella rete, grazie al contesto.

Solitamente NetFlow su router e sFlow sugli switch.

**Agent** Implementato in hw. Pacchetto ogni  $x$ , arricchito con info di contesto.

sFlow sugli switch, RMON su switch compatibili L2/L3, NetFlow sui router  $\rightarrow$  soluzioni di analisi del traffico: sorveglianza continua sulla rete. . .

**Ambiente**

sFlow switch, ma anche su router  
NetFlow router

**Velocità**

sFlow Multigigabit  
NetFlow  $\leq 1$  Gigabit

**Campionamento**

sFlow sempre  
NetFlow ogni tanto campiona

**Monitoraggio**

sFlow monitoraggio statistico  
NetFlow monitoraggio accurato (senza packet loss)

### 10.0.5 Radius

**Remote Authentication Dial In User Service**, protocollo comune. **Lato operatore**

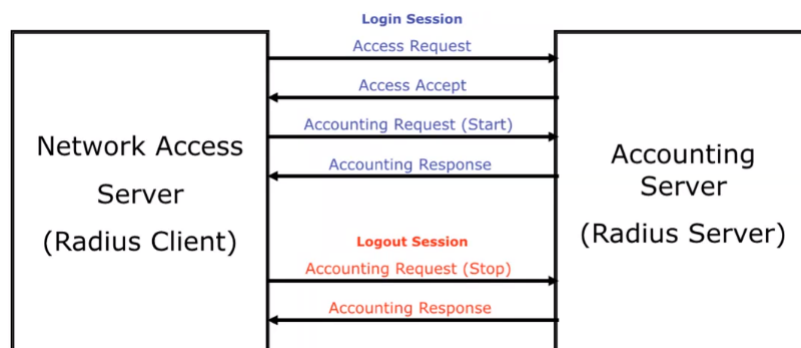
Importante per:

Protocollo più usato per implementare l'autenticazione sui dispositivi di rete

Usato per le attività di billing (calcolo pagamenti) su reti cablate (ADSL, Modem...)

Consente di gestire connessioni per durata o volume

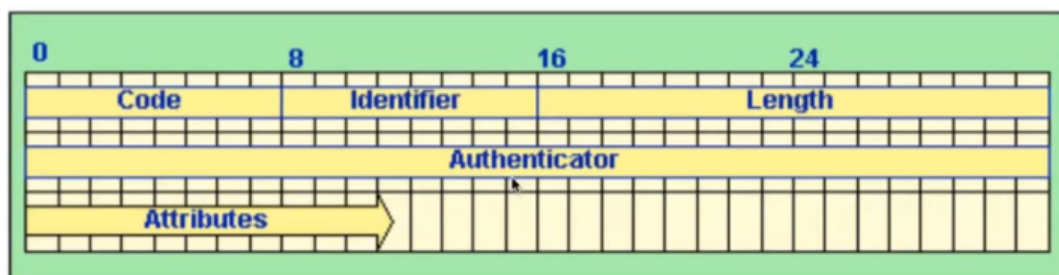
Supportato da tutti i dispositivi di rete (esclusi dispositivi a basso livello)



Il **client** è nella **centrale dell'operatore**. Client comunica con il server che mantiene info sugli utenti e risponderà sì/no per accettare l'autenticazione.

Dopo accesso c'è l'accounting request start: da ora navighi e calcolo informazioni per fare "il conto". A fine si manda richiesta di stop e in base a quello si calcola costo. Quindi c'è comunicazione costante col server RADIUS, in modo da poter interrompere comunicazione se superate eventuali soglie (e non scoprire il superamento solo al logout).

Si può solamente fare billing su tempo e durata.

**Pacchetti**

**Code:** byte contenente il comando/risposta RADIUS

**Identifier:** byte che identifica il comando/risposta RADIUS

**Length:** lunghezza del pacchetto

**Authenticator:** balore usato per autenticare la risposta del server RADIUS

**Attributer:** attributi del comando/risposta

# Capitolo 11

## Data Structures

**Bitmap** Array di bit. Lunghezza arbitraria e aggiunta/rimozione in  $O(1)$  con operazioni su bit semplici. Memoria in base ai bit usati

```
#define bitmap64_t(name, n) u_int64_t name[n / 64]
#define bitmap64_ptr_t u_int64_t *
#define bitmap64_reset(b) memset(b, 0, sizeof(b))
#define bitmap64_set_all(b) memset(b, 0xFF, sizeof(b))
#define bitmap64_clone(b1, b2) memcpy(b1, b2, sizeof(b1))
#define bitmap64_set_bit(b, i) b[i >> 6] |= ((u_int64_t) 1 << (i & 0x3F))
#define bitmap64_clear_bit(b, i) b[i >> 6] &= ~((u_int64_t) 1 << (i & 0x3F))
#define bitmap64_isset_bit(b, i) !!(b[i >> 6] & ((u_int64_t) 1 << (i & 0x3F)))
#define bitmap64_or(b1, b2) for (size_t __i = 0; __i < (sizeof(b1)/8); __i++) b1[__i] |= b2[__i]
```

Example:

```
bitmap64_t(tot_tcp_flags_combinations, 256); /* Define the variable (256 bit) */
bitmap64_reset(tot_tcp_flags_combinations); /* Reset the variable */
bitmap64_set(tot_tcp_flags_combinations, 67); /* Set a bit */
```

**Compressed Bitmaps** Una bitmap è un array di bit solitamente di lunghezza predefinita. Utile per specifici domini, possono essere sparse. Es. [0,12,23,500,510,522,10000] maggioranza a 0 e solo 7 bit a 1. **Inefficiente**.

Si possono comprimere contando il numero di ripetizioni es. 1(1), 11(0), 1(1), 10(0), 1(1), 476(0)... che significa 1 bit a 1, 11 bit a 0...

Molte librerie come WAH, EWAH, COMPAX... la maggioranza non permettono di settare un bit arbitrario "indietro", ma solo in avanti. Se si settano bit solo in avanti si può costruire la bitmap a runtime con memoria minima.

Vantaggio: una volta compressa si possono fare AND, OR, NOT senza fare unroll (cioè decomprimere). Una delle più popolari è [roaringbitmap.org](http://roaringbitmap.org)

### Compressed Bitmap Indexes

Row Id	Value	Col. bit0	Col. bit1	Col. bit2	Col. bit3
0	1	1	0	0	0
1	4	0	0	1	0
2	6	0	1	1	0
3	15	1	1	1	1
4	28	0	1	1	1
5	1	1	0	0	0
6	3	1	1	0	0

Sono utili per memorizzare dati flussi perché molto efficienti. Nell'esempio si ragiona per colonne, prendo uno dei valori e setto nelle colonne il valore corrispondente nei bit. Creo l'indice bit a bit, e ogni colonna rappresenta il bit  $i$ -esimo.  $n$  colonne, una per bit, se vedo ogni colonna indipendente la posso vedere come una bitmap compressa.

Dammi tutte le righe che hanno valore 1:  $C_0 = 1$  AND  $C_1 = 0$  AND  $C_2 = 0$  AND  $C_3 = 0$

Estremamente veloci, molto più delle bitree su dimensioni notevoli. Altra proprietà: per trovare indirizzi nelle netmask

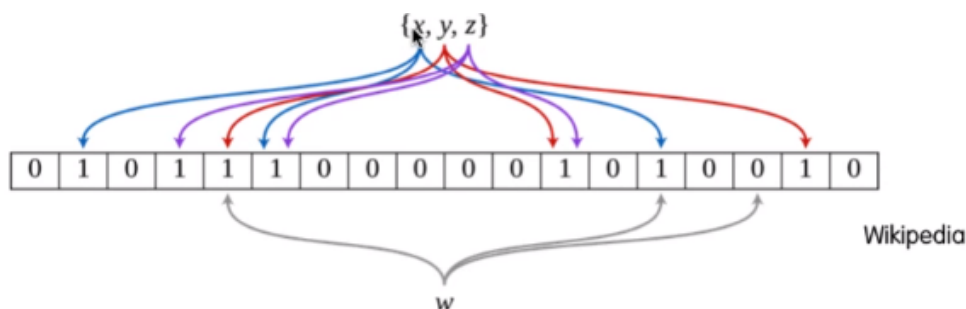
(tutte colonne che fanno parte della maschera settati come voglio), oppure mettere in AND due righe per prendere quelle che si intersecano su valori che voglio.

**Bloom Filters** I bloom filters sono strutture dati probabilistiche che rispondono alla domanda: elemento appartiene a set? Risponde con una certa probabilità.

Si può rispondere a questa domanda con altre strutture dati, come hash o bitmap molto lunghe, al prezzo di un costo molto maggiore perché in quei casi si analizzano dati raw.

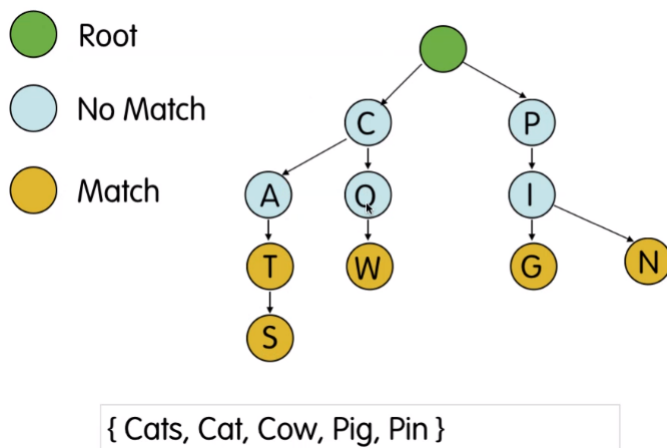
Sono array di bit di lunghezza  $m$ , settando i bit usando almeno due funzioni hash indipendenti e uniformemente distribuite  $h_1()$  e  $h_2()$ . Se voglio aggiungere  $\alpha$ , faccio  $h_1(\alpha)$  e setto i bit che vengono fuori, poi  $h_2(\alpha)$  e setto i bit. Per vedere se  $\beta$  è presente, faccio le due hash e vedo se i due bit sono stati settati, allora **certa probabilità** che  $\beta$  appartiene, se uno o nessuno dei due **non** è settato allora sicuramente non appartiene.

La probabilità non è uno svantaggio, perché permette di fare ricerche molto veloci. Essendo probabilistica **non** posso avere la certezza.



Con i **counting bloom filters** rimpiazzo il vettore di bit con un intero per sapere quanti bit sono settati o quante volte è stato settato un bit. Più memoria, ma si possono rimuovere elementi (con i bloom non si può). Buoni hash sono murmur, nfv e md5.

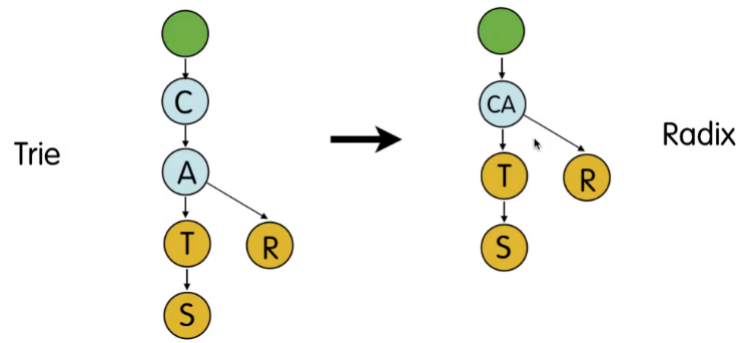
**Trie** Pronunciato try, (pun su *retrieval* e *tree*). Albero non basato su comparazioni, ogni nodo ha etichetta.



Nodi aggiunti/rimossi/cercati. Possibilità di cercare stringhe che iniziano per prefissi, e generare stringa in ordine di dizionario (se link in ordine alfabetico) Performance in  $O(w)$  dove  $w$  è lunghezza dell'albero.

**Radix tree** Come trie ma nodi collassati





**Patricia Tree** Numeri invece di stringhe, ogni nodo contiene una parte di indirizzi IP.  
 Struttura dati di riferimento per subnetting match su IPv4/IPv6.  
 codice su [github.com/ntop/nDPI/blob/dev/src/lib/third\\_party/src/ndpi\\_patricia.c](https://github.com/ntop/nDPI/blob/dev/src/lib/third_party/src/ndpi_patricia.c)

**Entropia** Entropia misura usata per misurare come i dati sono distribuiti su un certo range. Più alta entropia più dati sono sparpagliati.

Formula Esempio per usarla: grado di quanto sono distribuiti i dati

Esempio di uso sul TLS: entropia dei byte prima e dopo crittografia cambia ma è sempre compresa fra delle soglie per dati omogenei. Tramite entropia si può capire il tipo di dato.



## Capitolo 12

# Network Traffic Monitoring su Container

Traffic monitoring: garantire disponibilità funzionamento a lungo termine. A inizio 2000 fondamentalmente solo SNMP. Misure volumetriche, con RMON poco di più: sempre aggregate. Successivamente sFlow (2001) per risolvere problema visibilità: sempre contatori ma con informazioni di contesto.

Si dice che *i pacchetti non mentono mai*, info utili per capire: problemi di traffico, traffico non conforme a politiche. . . Sempre pacchetti e flussi. Non sono sbagliati i pacchetti, ma il problema è che bisogna adattarsi ai pacchetti. Paradigma utile per monitorare in maniera passiva, ma la crittografia ad esempio crea problemi. Non è **naturale per le persone**.

**System Introspection** Senza catturare traffico di rete chiedo al sistema. Ma ho comunque bisogno di info di contesto.

Adesso monitoraggio **unificato**: sempre parte classica (NetFlow, SNMP, IPFIX) ma con anche strumenti moderni per ambienti in container.

### 12.1 eBPF

macchina virtuale semplice eseguita nel kernel linux  
codice eseguito nel kernel space

**Kprobe** Kernel Probe. Permette di collegare ad un qualsiasi indirizzo di memoria d'istruzione del kernel un custom hook, cioè codice utente eseguito quando viene eseguita quell'istruzione, con visibilità completa di quella funzione kernel.



eBPF verifier: rigetta tutto ciò che non reputa adatto ad essere eseguito in kernel space. serie di condizioni per verificare che sia adatto: no loop indefiniti, ricorsione limitata, limitazione in spazio, no variabili globali, no allocazione dinamica, numero massimo istruzioni.

**Container** Istanze utente isolate, ognuna possibilmente su un processo diverso. Solitamente sfruttano feature del kernel: cgroup (limiti a risorse usabili per singoli processi), namespace (gruppi a cui iscriversi per avere vista limitata del sistema)