Appunti delle lezioni di Gestione di Sistemi e Reti
2006-2007

Franco Sirovich

7. SNMP: Simple Network Management Protocol

Il protocollo è stato definito nell'RFC1157, emesso nel 1990. E' interessante notare con quanta prontezza, rispetto allo sviluppo degli standard della infrastruttura, è stato compresa la necessità di avere un protocollo di gestione.

7.1. Concetti fondamentali

I concetti fondamentali che illustreremo in questo capitolo sono:

- Le operazioni del protocollo
- Il concetto di community
- Come si identificano le istanze degli object type
- Ordinamento imposto della convenzione per identificare gli oggetti

7.1.0. Operazioni supportate

Il protocollo supporta solo tre operazioni di uso generale:

- `get`: l'operazione ha in realtà due varianti (`get` e `get-next`); in ambedue i casi il manager recupera dall'agent il valore di (uno o) più oggetti scalari;
- `set`: il manager modifica sull'agent il valore di (uno o) più oggetti scalari;
- `trap`: l'agent invia al manager, senza esserne richiesto, il valore di (uno o) più oggetti scalari, assieme ad informazioni sull'occorrenza di un evento rilevato dall'agent.

Quindi le operazioni agiscono su liste di oggetti scalari che possono avere lunghezza diversa. Non sono disponibili operazioni per compiere azione che invece sono importanti,

---

quali:

- Operare su righe o colonne (si può solo operare sugli elementi delle tabelle)
- Aggiungere o togliere righe dalle tabelle
- Modificare la struttura della MIB
- Richiedere all'agent l'esecuzione di azioni

7.1.1. Community e Community Name

Il sistema di gestione di applicazioni distribuite è a sua volta una applicazione distribuita, le cui entità sono i manager e gli agent. Ma è una applicazione distribuita caratterizzata da relazioni uno-a-molti, che rendono complessa la questione dell'autenticazione e del controllo degli accessi. Le interazioni uno-a-molti che possiamo individuare nel sistema di gestione sono le seguenti:

- Un manager legge, scrive e riceve trap da un certo numero di agent: diciamo che il manager gestisce un insieme di agent;
- Diversi manager gestiscono diversi insiemi di agent che non necessariamente sono coincidenti e che possono avere elementi in comune;
- Un agent possiede una MIB locale, e la rende disponibile per la gestione a un certo numero di manager: diciamo che l'agent è al servizio di un insieme di manager;
- Diversi agent offrono la gestione della propria MIB a diversi insiemi di manager, non necessariamente coincidenti e che possono avere elementi in comune.

Dall'analisi di queste interazioni si ricava che vi sono quindi due tipi di relazioni uno-a-molti: quella fra un manager e diversi agent e quella fra un agent e diversi manager. Se ci mettiamo dal punto di vista degli agent, gli agent devono controllare l'uso che i manager possono fare della MIB locale, mentre dal punto di vista dei manager, i manager devono controllare quali agent possono provocare il loro intervento inviando delle trap.

Il controllo che gli agent devono poter esercitare ha tre aspetti:

- *Servizio di autenticazione*: l'agent deve poter limitare l'accesso solo a certi manager ben identificati.
- *Politica di accesso*: l'agent deve poter dare differenti privilegi di accesso a differenti manager autenticati
- *Proxy*: l'agent deve poter dare accesso alle funzionalità di proxy solo a certi manager autenticati, e con restrizioni sui privilegi di accesso.

Questi sono gli aspetti da affrontare per risolvere il problema della sicurezza, che nell'ambito della gestione è molto importante, perché i danni che si possono compiere con una errata (o malevola) gestione sono chiaramente molto gravi. Per mantenersi semplice, SNMPv1 (RFC1157) fornisce agli agent solo un limitato strumento per affrontare il problema della sicurezza: il concetto di *community.*

In SNMP una *community* è una *relazione* fra un agent SNMP e *un insieme di manager* SNMP (NMS) che definisce *autenticazione* (a cui sottomettere i manager), *controllo degli accessi* e *caratteristiche di* proxy (da concedere ai manager una volta autenticati con successo).

Si noti che una community è un concetto locale all'agent, il quale definisce una community per ogni combinazione di (insieme di manager, autenticazione, accesso e

proxy) che desidera far rispettare o che, da un altro punto di vista, si impegna di onorare.

In realtà il concetto di community è molto potente: perché allora lo abbiamo appena presentato come uno strumento “limitato”? Perché (e questa è la critica che si può fare a SNMPv1) questo potente strumento viene però specificato e implementato in modo molto debole, per non rendere costosa la realizzazione degli agent.

Poiché un agent può definire più di una community, ciascuna community è identificata da un *community name*, univoco nell'ambito di ogni singolo agent: agent diversi possono usare lo stesso nome di community con "significato" diverso, perché la community è un concetto locale ad ogni singolo agent.

Vediamo ora come sono specificati i componenti del servizio di sicurezza: autenticazione, controllo dell'accesso e uso del proxy.

7.1.1.1. Servizio di Autenticazione

Quando un manager si presenta ad un agent e richiede la esecuzione di una delle operazioni del protocollo, deve *autenticarsi*: deve cioè dimostrare di essere membro di diritto della community a cui pretende di appartenere. Si noti che il fatto che SNMP usa UDP, rende il problema della autenticazione più difficile rispetto alla situazione di un protocollo che usi TCP, perché l'indirizzo IP da cui proviene la richiesta non è utile alla autenticazione perché può facilmente essere falsificato, come pure è banale re-inviare una PDU “spiata” in precedenza e in questo modo ingannare l'agent.

Rispetto al difficile compito di eseguire una autenticazione “sicura” usando UDP, la semplicità (e la limitazione) di SNMPv1 consiste nel fatto che SNMP specifica che la sola *conoscenza* del *community name* (che è definito in ciascun agent in completa autonomia) è la prova che la sorgente del messaggio è autentica, cioè che il messaggio proviene effettivamente dal manager che pretende di averlo mandato.

Bisogna fare attenzione che il problema non è che (sembra che) manchi una "password", ma il fatto che *i messaggi del protocollo viaggiano in chiaro* sulla rete IP e quindi possono tranquillamente essere analizzati e il community name "svelato". Inoltre, non vi è nessuna protezione contro il replay, cioè il semplice re-invio di una PDU catturata in passato. Anche se venisse scambiata una password, l'autenticazione avrebbe esattamente la stessa capacità di resistere agli attacchi.

Come abbiamo già sottolineato, dato che SNMP usa UDP, è molto costoso proteggere il processo di autenticazione (specificato solo in SNMPv3, anche per questo non accettata largamente sul mercato). In SNMPv1 e SNMPv2 hanno quindi deciso di mantenere l'autenticazione molto semplice, perché non ritenevano di potersi permettere una autenticazione *forte*.

7.1.1.2. Politica di accesso

Usando una community, un agent può limitare l'accesso alla sua MIB ad un particolare insieme di manager. Definendo più community un agent può conferire *privilegi diversi* di accesso a *diversi insiemi di manager*: può quindi implementare *differenti politiche di accesso*.

Ci possono essere due aspetti che definiscono una politica di accesso di una community:

- *SNMP MIB view*: Solo una parte della MIB può essere vista dai manager della community, e gli oggetti visti possono non appartenere allo stesso sottoalbero. In questo modo posso specializzare i manager a gestire solo parti della MIB dell'agent, requisito

importante quando l'agent permette di controllare un sistema costituito da numerose applicazioni distribuite.

- *SNMP access mode*: A parità di MIB view, e necessario poter distinguere diversi insiemi di manager sulla base delle operazioni che essi possono compiere sugli oggetti appartenenti alla MIB view. Dato che le operazioni che si possono invocare con il protocollo sono molto limitate, è sufficiente introdurre due soli access mode: *READ-ONLY* e *READ-WRITE*, da usare ovviamente in alternativa. L'access mode è lo stesso per tutti gli OID nella vista.

La combinazione di MIB view e di access mode è detta *community profile*.

Ma anche nella definizione della MIB vi è definito, mediante il parametro ACCESS della macro OBJECT-TYPE, una moda modalità di accesso all'oggetto definito: ad es. vi sono object type read-only ed object type read-write. Come si combinano allora queste due "restrizioni", quella istituita a tempo di definizione della MIB e quella imposta dall'agent? L'ovvia risposta è che si può solo rendere più severe le restrizioni definite nella MIB. È molto semplice illustrare queste combinazioni in una tabella che descrive la combinazione delle due restrizioni sulla accessibilità di un oggetto:

MIB ACCESS	SNMP Community Access Mode	
	`READ-ONLY`	`READ-WRITE`
`read-only`	disponibile per get e trap	
`read-write`	disponibile per get e trap	disponibile per get, trap e set
`write-only`	disponibile per get e trap, ma il valore ritornato è implementation-specific	disponibile per get, trap e set get, trap e set, ma il valore per le get e le trap è implementation-specific
`not accessible`	non disponibile	

Si noti che anche se la modalità specificata è `write-only`, e la community concede diritti di lettura, una implementazione può permettere l'accesso in `read`, ma fornendo un comportamento che ovviamente è implementation specific. Queste sono le concessioni che si devono fare alle pressioni dei costruttori perché la propria implementazione non risulti non compatibile con gli standard; peccato che uno standard che definisse come legale qualunque comportamento implementation-specific non avrebbe un grande valore per gli utilizzatori.

La combinazione di una community e di un particolare community profile definisce una *SNMP access policy*, in quanto definisce il comportamento permesso a particolari manager. Abbiamo conseguentemente una tassonomia illustrata nella figura seguente:

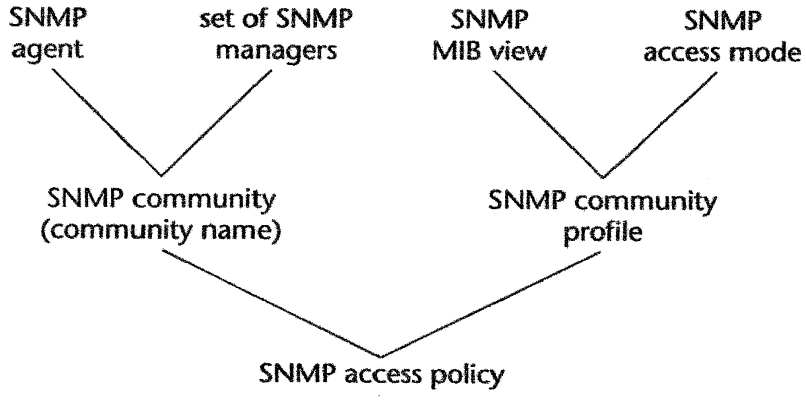


FIGURE 7.1 Administrative concepts

7.1.1.3. Proxy

Un agent SNMP può agire come intermediario verso un agent che non è SNMP, oppure che non si vuole che venga gestito direttamente dai manager, per ridurre le interazioni fra manager e agent. Anche la politica di accesso verso gli apparati proxati (cioè quali manager hanno il diritto di utilizzare la funzione di proxy) è legata alla community, e l'agent deve definirla e farla rispettare.

7.1.2. Identificazione delle istanze

Ogni oggetto è identificato dal un OID univoco nella MIB. La specifica della MIB (in questo caso intendiamo lo schema dei dati di gestione) definisce gli OID degli *object type* le cui istanze popoleranno la MIB degli agent. Che OID avranno le istanze di un certo object type?

SNMP impone che *una istanza di un object type ha un OID che come prefisso ha l'OID dell'object type.* Quindi le istanze sono le foglie del sotto-albero radicato del nodo che identifica l'object type. Questa regola è molto semplice ma anche molto naturale e in fondo corrisponde a principio di raggruppare le istanze di un certo object type usando la struttura gerarchica degli OID.

Il protocollo SNMP prescrive che nelle operazioni di SNMP si può accedere solo a istanze di oggetti e non a interi object type o a tutte le loro istanze. Inoltre vi sono restrizioni per l'accesso a object type la cui sintassi (definizione di tipo ASN.1) è "complessa". Per mantenere il protocollo semplice, si può accedere solo a object type la cui sintassi sia un tipo semplice dell'ASN.1, anche se si vuole mantenere nella definizione della MIB (schema) la definizione di object type la cui sintassi è complessa (tabelle) per dare "leggibilità alla definizione della MIB.

Occorre risolvere il problema originato da cui due esigenze che sono fra di loro in contraddizione:per questa ragione la SMI obbliga a fare le definizione delle MIB in modo particolare, introducendo al di sotto di un object type tabellare (che non può essere istanziato né può essere acceduto, e quindi introdotto solo per leggibilità) un object type il cui scopo è quello di definire la generica riga della tabella (che non può essere acceduto né istanziato, quindi introdotto solo per leggibilità), al di sotto del quale devono essere definiti tanti *object type colonnari* quanti sono i componenti della generica riga della tabella.

A questo punto, abbiamo object type la cui sintassi è una delle sintassi semplice di ASN.1 permesse dalla SMI, e quindi object type che sono istanziabili e accessibili via

protocollo, ma che hanno numerose istanze ciascuna delle quali deve essere identificabile da un OID distinto. Vediamo la regola che permette di associare ad ogni istanza uno specifico OID che la identifica univocamente.

7.1.2.1. Oggetti colonnari

Ci sono molte istanze degli object type che specificano gli elementi di una colonna della tabella. Ricordiamo che l'OID delle istanze è creato *aggiungendo* all'OID dell'object type della colonna una successione di interi non negativi che viene derivata dal *valore dei campi* che agiscono come *indice* delle righe. Ogni campo indice produce una successione di interi non negativi. Se vi sono numerosi campi indice, vi saranno altrettante successioni di interi non negativi che completano/seguono l'OID dell'object type colonnare. Se si conosce la regola di generazione delle successioni di interi non negativi e si conosce la definizione della MIB, si può riconoscere le successioni di interi non negativi all'interno dell'OID dell'oggetto e ricavare il valore dei campi indice qualora questo non fosse noto. La regola che definisce la corrispondenza fra valore dell'indice e sequenza di interi non negativi è stata spiegata precedentemente.

Riprendiamo l'esempio della tabella delle connessioni TCP, che ha ben 4 dei 5 campi che agiscono da INDEX: `tcpConnLocalAddress`, `tcpConnLocalPort`, `tcpConnRemAddress`, `tcpConnRemPort`. Ogni istanza di ciascuna colonna, ad esempio della colonna `tcpConnState`, ma anche della altre 4, ha la forma

```
x.i.(tcpConnLocalAddress).(tcpConnLocalPort).(tcpConnRemAddress)
.(tcpConnRemPort)
```

dove

`x = 1.3.6.12.1.6.13.1` , cioè l'OID di `tcpConnEntry`,

i = indice della colonna nella tabella, ad es. per `tcpConnState` vale 1,

(*name*) = la sequenza di interi non negativi prodotta dal valore del campo *name* su quella riga della tabella. Ricordiamo che le regole che determinano esattamente la sequenza di interi non negativi generata da un valore che serve da INDEX sono:

Valore intero: produce un solo valore; si noti che un INDEX non deve mai essere negativo!

Valore stringa di lunghezza fissata dalla specifica: (quindi sono un sottotipo delle OCTET STRING) ogni ottetto viene interpretato come la rappresentazione in binario di un intero senza segno, e questo intero è aggiunto all'OID; quindi tanti interi quanti sono gli elementi della stringa.

Valore stringa di lunghezza variabile: (quindi OCTET STRING o un sottotipo di lunghezza non fissata dalla specifica) il primo sotto-identificare è dato dalla lunghezza *n* della stringa, seguito poi tanti sotto-identificatori quanti sono gli ottetti della stringa, ottenuti come nel caso precedente di stringhe a lunghezza fissa.

Valore di tipo OBJECT IDENTIFIER: il primo sotto-identificatore è la lunghezza dell'OID, seguito dai sotto-identificatori che costituiscono l'OID.

7.1.2.2. Il problema dei riferimenti ambigui alle righe concettuali delle tabelle

L'RFC1212 che definisce la clausola INDEX dice che i campi indici sono quei campi i cui valori *"distinguono in modo non ambiguo una riga concettuale della tabella dalle altre righe"*.

Se il progetto della tabella (nella specifica della MIB) è mal fatto, cioè i campi indice non sono in realtà in grado di identificare univocamente le righe, non ci sono soluzioni ragionevoli, generalmente accettate e di complessità ragionevole. La conclusione è che occorre fare bene il progetto della MIB (vedi il caso della `ipRouteTable`).

7.1.2.3. La Tabella Concettuale e l'object type Riga

L'object type di una tabella e l'object type della generica riga non hanno una sintassi semplice e quindi devono essere dichiarati `not-accessible`. Non essendo accessibili non devono neanche essere istanziati e quindi non esiste alcun attributo associato alle loro istanze.

Questi object type sono utili esclusivamente come "documentazione" e come supporto alla generazione degli OID delle istanze degli object type colonnari che sono situati al disotto dell'object type della generica riga: gli object type colonnari invece devono avere una sintassi semplice.

Gli object type colonnari possono avere, a run time, un numero di istanze maggiore di 1 e quindi serve una regola per generare gli OID di ciascuna istanza a partire dai valori dei campi INDEX (la regola descritta precedentemente).

7.1.2.4. Object type Scalari

Sono quelli che a run time devono avere *esattamente una sola istanza*, perché non sono parte di una tabella. La regola per generare l'OID della istanza è molto semplice: l'agent aggiunge l'intero 0 (zero) all'OID dell'object type.

7.1.3. Ordinamento lessicografico

Gli OID sono una sequenza di interi non negativi, che riflette la struttura ad albero dello spazio degli OID. È quindi facile definire una nozione di *ordinamento* fra di essi, che essendo basata sulla struttura di successione viene detto *lessicografico.*

La definizione di ordinamento è la seguente:

Date due sequenze di interi non negativi

$(x_1,x_2,...x_n)$

$(y_1,y_2,...y_m)$, con n <=m,

Diciamo che

$(x_1,x_2,...x_n)$ precede $(y_1,y_2,...y_m)$

quando una delle seguenti condizioni è soddisfatta

1. [($x_i = y_i$ per 1<=i<k; 1<=k<=n, m) AND

($x_k < y_k$)]

cioè esiste un valore di k per cui le prime (k-1) etichette sono uguali, la k-esima è diversa e determina l'ordinamento fra i due OID; oppure

2. [($x_i = y_i$, per 1<=i<=n) AND (n < m)]

cioè la stringa più corta è esattamente contenuta come prefisso della più lunga: la più corta precede la più lunga.

Si noti che questa nozione di ordinamento coincide con l'ordine con cui vengono visitate le foglie dell'albero degli OID nell'agent quando la visita viene fatta in pre-order, ma è più generale della nozione basata sulla visita dell'albero, perché permette di decidere un ordine fra due OID anche se non appartengono allo stesso albero!

Su questa nozione di ordinamento è basata la definizione di una operazione del protocollo SNMP che è molto importante: la `get-next`.

7.2. Specifica del Protocollo

7.2.1. Formati SNMPv1

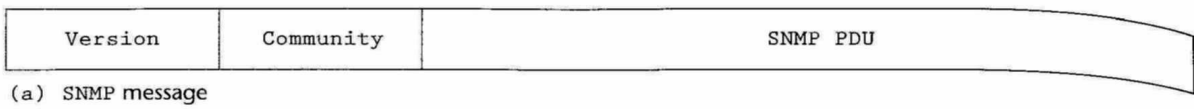
Ogni messaggio scambiato fra agent e manager SNMP è composto da:

- Numero di versione
- Community string
- Una *Protocol Data Unit* (*PDU*) che può essere di 5 tipi diversi

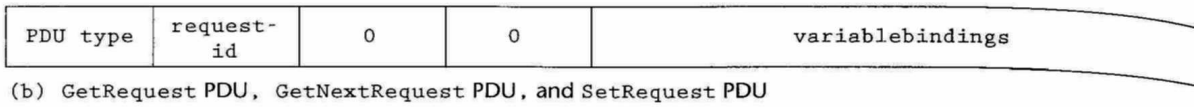
Si noti subito che SNMP usa il termine PDU in modo molto "particolare": in tutto il resto del mondo per PDU si intende l'*intero messaggio* scambiato fra due entità, e non solo una parte del messaggio. Questa anomalia è dovuta alla volontà di trattare in modo molto aperto al futuro la questione dell'autenticazione. Nuovi metodi di autenticazione “forte” ci si attendeva che avrebbero cambiato la struttura di questa terza parte, e si voleva che le due parti iniziali permettessero di distinguere non solo una versione da un'altra ma anche fornissero gli elementi per decodificare (decifrare) la terza parte.

Le PDU si devono distinguere l'una dall'altra; sono cinque, ma sono di solo *tre* formati diversi. L'analisi che segue evidenzierà che in realtà ci sono molti elementi in comune fra i tre formati per cui in realtà due di essi possono essere trattati nello stesso modo, riducendo i formati a solo due.

Le PDU di `GetRequest`, `GetNextRequest` e `SetRequest` hanno tutte lo stesso formato, e si contraddistinguono (solo) dal `PDUtype`.

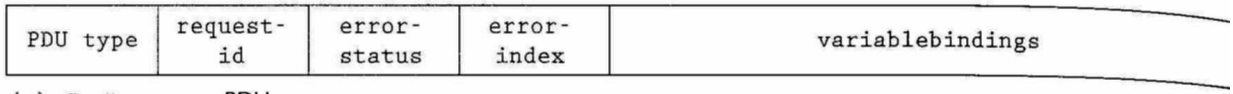


(a) SNMP message



(b) GetRequest PDU, GetNextRequest PDU, and SetRequest PDU

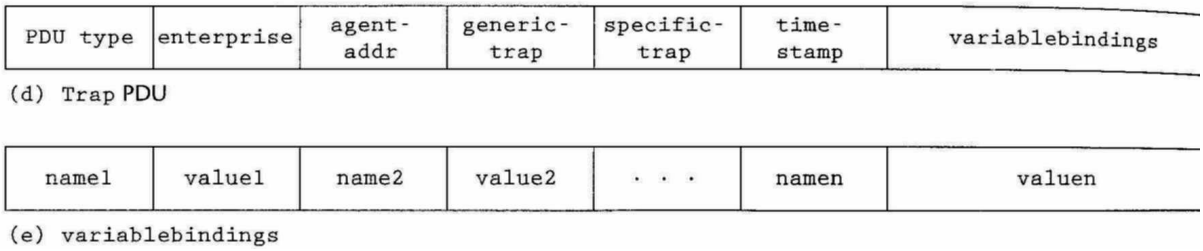
A queste PDU l'agent risponde sempre e solo con un unico formato chiamato `GetResponsePDU`.



(c) GetResponse PDU

Notate che la PDU porta con se non solo i varbind richiesti nel caso di `Get` e `GetNext`, ma anche i varbind dopo l'effettuazione della `Set`, sia che sia andata bene che male.

Infine vi è il formato delle trap e il formato dei campi varbind che è lo stesso in qualunque PDU sia di richiesta che di risposta!



PDU type	enterprise	agent-addr	generic-trap	specific-trap	time-stamp	variablebindings

(d) Trap PDU

name1	value1	name2	value2	. . .	namen	valuen

(e) variablebindings

Vediamo meglio i vari campi

version - la Version 1 è indicata dalla costante intera 0 (zero).

community - molti la chiamano "il nome della community"; è già stata discussa (vedi prima).

PDU type - vedremo che è passato in modo speciale dalle codifiche della PDU: è quella informazione che permette di identificare il tipo della PDU.

request-id - un INTEGER usato per distinguere una invocazione dalle altre; nella risposta deve essere uguale al request-id usato nella richiesta, altrimenti la risposta viene scartata. Se il manager ha sottoposto a timeout la risposta (per evitare di attendere all'infinito una richiesta o una risposta che sia andata persa) il request-id della richiesta “scaduta” viene in genere invalidato e quindi una risposta che arriva in ritardo viene scartata. Conseguentemente, occorre settare il tempo di attesa ad un valore sufficientemente grande.

error-status - un INTEGER che codifica i vari tipi di errore (fra cui `noError (0)` !).

error-index - quando `error-status` è non-zero fornisce addizionale informazione sull'errore.

variablebindings - una lista di varbind, cioè `SEQUENCE OF SEQUENCE (<name>, <value>)`

enterprise - un OID che identifica il tipo di apparato, o di un suo componente, che ha generato la trap; dato che l'apparato è certamente fatto da un vendor, questo OID sta nello spazio `enterprises` degli OID, ed è gestito dai costruttori. L'OID che identifica l'apparato viene anche memorizzato in `sysObjectID`, che è una variabile del gruppo `system`; se la trap è generata da componente di un apparato, allora l'OID che viene messo nel campo `enterprises` è nel sotto-albero radicato nell'OID che si trova in `sysObjectID`.

agent-addr - indirizzo IP dell'agent dell'apparato che ha generato la trap.

generic-trap - un INTEGER che indica il tipo di trap (che abbiamo già visto).

specific-trap - un INTEGER che indica la trap specifica all'interno del tipo generico; ci deve essere anche se il generic non è `enterpriseSpecific`.

time-stamp - è un TimeTicks che indica il momento in cui la trap si è verificata contando

il tempo dall'istante dell'ultima ripartenza fredda dell'agent. La ragione per riportare questo valore è che l'invio della trap può avvenire molto tempo dopo il momento in cui si è verificata, sia perché l'agent può avere altro da fare, sia perché un guasto, ad es. link down, può impedire l'invio della trap.

Le definizioni delle PDU sono contenute in RFC1157 e sono le seguenti. Vediamo di imparare a leggerle per capire cosa dice l'ASN.1 che viene usata, come per le MIB, per la definizione dei messaggi e della parte PDU. Ci servirà per capire poi come vengono davvero codificati i messaggi e quindi capire cosa viene scambiato sulla rete.

```
RFC1157-SNMP DEFINITIONS ::= BEGIN
IMPORTS
        ObjectName, ObjectSyntax, NetworkAddress, IpAddress,
        TimeTicks
FROM RFC1155-SMI;
-- top-level message
Message ::=
        SEQUENCE {
                version INTEGER {version-1(0)},
                                        -- version-1 for this RFC
                community OCTET STRING, -- community name
                                data ANY -- e.g., PDUs if trivial
                                -- authentication is being used

        }
-- protocol data units ...
```

La definizione ci dice che un messaggio SNMP è una sequenza di un *intero*, una *octet string*, e *un qualunque dato* (`ANY`) che verrà però definito in futuro.

L'intero, che viene chiamato `version` nello standard vale la costante chiamata `version-1` che vale `0` per indicare la prima versione dello standard.

Si noti che si specifica che i `data` siano proprio del tipo `PDUs` solo se si userà *autenticazione banale*, e quindi si prevedeva di poter usare metodi di autenticazione più forti. Per essere aperti a future estensioni del meccanismo di autenticazione `data` è stata definita di tipo `ANY`.

Vediamo ora come è definito il tipo `PDUs`.

```
-- protocol data units ...
PDUs ::=CHOICE {
get-request GetRequest-PDU,
get-next-request GetNextRequest-PDU,
get-response GetResponse-PDU,
set-request SetRequest-PDU,
trap Trap-PDU

        }
```

Il tipo `PDUs` può assumere cinque "formati" diversi. Il costrutto `CHOICE` di ASN.1 indica che il tipo `PDUs` può assumere i tipi indicati fra parentesi graffe. Alle alternative è dato un nome per poterlo riferire nello standard (`get-request` è il nome dell'alternativa, `GetRequest-PDU` è il tipo corrispondente). Chi spedisce i data può quindi "scegliere" fra i cinque tipi, ma chi riceve come fa a capire che una certa PDU è di un tipo o dell'altro? È necessario distinguere per poter verificare se il mittente ha eseguito una delle scelte

permesse dalle regole del protocollo e per eseguire le azioni corrispondenti previste dal protocollo.

```
-- PDUs
GetRequest-PDU ::= [0] IMPLICIT PDU
GetNextRequest-PDU ::= [1] IMPLICIT PDU
GetResponse-PDU ::= [2] IMPLICIT PDU
SetRequest-PDU ::= [3] IMPLICIT PDU
```

I primi quattro tipi di PDU sono dello stesso tipo `PDU` (appunto!), con un tag context-specific che permette di distinguere la scelta fatta. I progettisti dello standard hanno voluto fare in modo che dal punto di vista del formato del messaggio i quattro tipi fossero identici, per facilitare la scrittura del codice, ma comunque permettessero al ricevitore di interpretare in modo diverso i campi del messaggio. Se la interpretazione (semantica) del messaggio fosse la stessa, non sarebbe necessario distinguere i tipi!

Un altro esempio di "variabilità/ambiguità" è dato dalla `ObjectSyntax`, cioè dal valore che in un varbind è accoppiato con il nome di managed object.

```
VarBind ::=
SEQUENCE {      name ObjectName,
                value ObjectSyntax }
VarBindList ::= SEQUENCE OF VarBind

-- syntax of objects in the MIB
ObjectSyntax ::=CHOICE {
        simple  SimpleSyntax,
-- note that simple SEQUENCEs are not directly
-- mentioned here to keep things simple (i.e.,
-- prevent mis-use). However, application-wide
-- types which are IMPLICITly encoded simple
-- SEQUENCEs may appear in the following CHOICE
        application-wide        ApplicationSyntax }
SimpleSyntax ::= CHOICE {
                number  INTEGER,
                string  OCTET STRING,
                object  OBJECT IDENTIFIER,
                empty   NULL

                        }
```

In questo caso la `CHOICE` non da problemi: le alternative non sono ambigue perché ogni tipo è diverso (discuteremo in seguito il caso di `ApplicationSyntax`)

Adesso possiamo vedere la definizione di PDU, che può essere la stessa in tutti i casi perché non è necessario introdurre differenze:

```
PDU ::=
SEQUENCE {

   request-id        INTEGER,

   error-status -- sometimes ignored
                INTEGER {

                        noError(0),
```

```
                        tooBig(1),
                        noSuchName(2),
                        badValue(3),
                        readOnly(4),
                        genErr(5)

                        },
    error-index -- sometimes ignored
                INTEGER,

    variable-bindings -- values are          sometimes ignored
            VarBindList

  }
```

Il tipo `PDU` è una `SEQUENCE` e quindi il suo valore dovrebbe normalmente essere preceduto dal TAG di `SEQUENCE`, ma dato che dove viene usato (dentro alla `CHOICE` di `PDUs`) c'è l'`IMPLICIT`, il TAG di `SEQUENCE` salta e viene sostituito da quello context-specific definito nella sintassi.

Il quinto tipo di PDU, quello delle trap, ha invece una struttura diversa dai precedenti:

```
Trap-PDU ::= [4] IMPLICIT SEQUENCE {
        enterprise -- type of object generating trap, see
                   -- sysObjectID in reference [5]
                OBJECT IDENTIFIER,
        agent-addr -- address of object generating trap
                NetworkAddress,
        generic-trap -- generic trap type
                INTEGER {
                        coldStart(0),
                        warmStart(1),
                        linkDown(2),
                        linkUp(3),
                        authenticationFailure(4),
                        egpNeighborLoss(5),
                        enterpriseSpecific(6)

                         },
        specific-trap -- specific code, present even if
                      -- generic-trap is not enterpriseSpecific
                INTEGER,
        time-stamp -- time elapsed between the last
                   -- (re)initialization of the network entity and
                   -- the generation of the trap
                TimeTicks,
        variable-bindings -- "interesting" information
                VarBindList

               }
```

Anche `Trap-PDU` deve avere un TAG, e in realtà essendo la quinta alternativa della `CHOICE` e le altre quattro avendo un TAG context-specific potrebbe mantenere il suo TAG della `SEQUENCE` senza generare ambiguità. Ma questo modo di procedere "minimalista" è

stato ritenuto poco elegante e quindi anche alla `Trap-PDU` è stato assegnato un TAG context-specific e il TAG della `SEQUENCE` è stato fatto "saltare" usando la direttiva `IMPLICIT`:

Vediamo ora come sono definiti i variablebindings che compaiono in tutte le PDU:

```
-- variable bindings
VarBind ::= SEQUENCE {  name ObjectName,
                        value ObjectSyntax

                        }
VarBindList ::= SEQUENCE OF VarBind
-- names of objects in the MIB
ObjectName ::= OBJECT IDENTIFIER
-- syntax of objects in the MIB
ObjectSyntax ::=
```

CHOICE {

simple SimpleSyntax,

```
      -- note that simple SEQUENCEs are not directly
      -- mentioned here to keep things simple (i.e.,
      -- prevent mis-use). However, application-wide
      -- types which are IMPLICITly encoded simple
      -- SEQUENCEs may appear in the following CHOICE
```

application-wide ApplicationSyntax

}

```
SimpleSyntax ::=
```

CHOICE {

number INTEGER,

string OCTET STRING,

object OBJECT IDENTIFIER,

empty NULL

}

```
ApplicationSyntax ::=
```

CHOICE {

address NetworkAddress,

counter Counter,

gauge Gauge,

ticks TimeTicks,

arbitrary Opaque

-- other application-wide types, as they are

-- defined, will be added here

}
NetworkAddress ::=
CHOICE {

internet IpAddress

}
IpAddress ::=

[APPLICATION 0] -- in network-byte order

IMPLICIT OCTET STRING (SIZE (4))
Counter ::=

[APPLICATION 1]

IMPLICIT INTEGER (0..4294967295)

...

Come si può notare, i valori che nei messaggi di protocollo compaiono associati ai "nomi", cioè agli OID, possono essere solo di `SimpleSyntax` oppure di `ApplicationSyntax`. Le `SimpleSyntax` sono solo quattro tipi semplici dell'ASN.1, mentre le `ApplicationSyntax` sono enumerate in una `CHOICE` che il protocollo specifica che potrà essere ampliata in futuro, ma che per ora è limitata a quattro casi. Il testo di commento specifica infine che nulla vieta che nelle `ApplicationSyntax` ci potranno essere delle `SEQUENCE`; per ora non ci sono per tenere le cose semplici.

Dato che due `CHOICE` sono annidate, tutte le alternative devono essere distinguibili e quindi le `ApplicationSyntax` devono avere dei TAG definiti dalle specifiche. In questo caso, il "contesto" è l'intero protocollo applicativo SNMP e quindi la classe di TAG adottata è la application-wide.

Nei paragrafi che seguono sono sviluppate alcune osservazioni sul comportamento delle entità di protocollo.

7.2.1.1. Trattamento di un messaggio da spedire

Il comportamento di una entità di protocollo che vuole inviare un messaggio SNMP, sia essa un agent o un manager deve eseguire i seguenti passi:

1. Il *componente applicativo* costruisce una PDU (nel senso RFC1157, cioè la parte che segue il numero di versione e la community).
2. La PDU viene passata al *servizio di autenticazione*, assieme a tutte informazioni che, in un qualunque processo di autenticazione, potrebbero determinare il modo con cui l'entità SNMP si autentica con l'entità paritaria. In questo modo si è tentato di essere aperti a successive evoluzioni del protocollo, e per questa ragione la parte PDU del messaggio SNMP viene definita di tipo `ANY`. Le informazioni che si possono passare al servizio di autenticazione sono:
   - Numero di versione di SNMP
   - Indirizzo di trasporto sorgente e destinazione
   - Community che si vuole usare per la spedizione

3. Il servizio di autenticazione esegue le eventuali necessarie trasformazioni della PDU (quali inclusione del codice di autenticazione e/o la cifratura) e restituisce il risultato; quindi il risultato potrebbe non essere uguale all'input! E' per questa ragione che nella definizione del messaggio di protocollo SNMP si dice che il terzo campo è di tipo `ANY` e che nel caso di autenticazione banale il tipo è di tipo `PDU`. Come si può vedere si è fatto un grande sforzo per avere uno schema molto generale.

L'entità di protocollo mittente costruisce il messaggio, cioè un nuovo oggetto ASN.1, che consiste nel numero di versione, il nome della community e il risultato dello step 2. L'oggetto ASN.1 viene codificato usando le BER e spedito usando il servizio di trasporto.

7.2.1.2. Trattamento di un messaggio ricevuto

Il primo compito di una entità ricevente consiste nell'eseguire un check sintattico di base; se questo test non ha successo il messaggio deve essere scartato. Si deve poi verificare il numero di versione, e ancora scartare il messaggio se il numero di versione non corrisponde ad una versione supportata dall'entità ricevente.

A questo punto occorre invocare il servizio di verifica dell'autenticazione in entrata, fornendo come parametri il numero di versione, la community, la parte PDU del messaggio, l'indirizzo di trasporto sorgente (reso disponibile da UDP) e di destinazione (reso disponibile da UDP). Se la verifica dell'autenticazione fallisce, il servizio di autenticazione informa SNMP, che scarta il messaggio e manda una trap di autenticazione fallita.

Se la verifica di autenticazione ha successo, il servizio di autenticazione restituisce alla entità SNMP una PDU in formato ASN.1. E' importante notare che l'autenticazione potrebbe decodificare/trasformare la PDU ricevuta, se non è stata adottata l'autenticazione banale!

L'entità di protocollo SNMP fa un controllo sintattico della PDU restituita dall'autenticazione, e la scarta se il controllo fallisce. Altrimenti usando la community e la politica di controllo degli accessi esegue l'operazione richiesta e restituisce i risultati.

7.2.1.3. Variable Bindings

Raggruppare in una singola richiesta operazioni e risposte su numerose foglie aiuta SNMP a essere efficiente, perché riduce l'overhead di protocollo. Nella operazione `GetRequest` io conosco solo gli OID su cui voglio operare e non il loro valore. Ma per semplificare l'analisi dei pacchetti il tipo degli argomenti è sempre `variable-bindings` e quindi nella coppia `VarBind` dovrò mettere `null` come valore, che è un valore di uno dei tipi ammissibili, come si ricava dall'esame della definizione ASN.1 che riportiamo qui sotto per vostra comodità.

```
-- syntax of objects in the MIB
ObjectSyntax ::=CHOICE {
        simple  SimpleSyntax,
                -- note that simple SEQUENCEs are not directly
                -- mentioned here to keep things simple (i.e.,
                -- prevent mis-use). However, application-wide
                -- types which are IMPLICITly encoded simple
                -- SEQUENCEs may appear in the following CHOICE
        application-wide        ApplicationSyntax
```

```
                        }
SimpleSyntax ::=
        CHOICE {
                number  INTEGER,
                string  OCTET STRING,
                object  OBJECT IDENTIFIER,
                empty   NULL

                }
```

Nel progetto di SNMP è stato fatto un notevole sforzo per ridurre il numero di strutture dati che una implementazione deve trattare, in modo da facilitare la riduzione del codice e aumentare la sua compattezze e ridurre il suo costo. Questa attenzione ha portato a scegliere le stesse strutture di `variable-bindings` nelle richieste e nelle risposte, e alla unificazione dei formati delle PDU, che è stata ulteriormente estesa in SNMPv2.

7.2.2. GetRequest PDU

Nella richiesta i valori associati agli OID sono settati a `null`, perché appunto vengono richiesti in quanto non noti. L'agent risponde con una `GetResponse` PDU, che contiene lo stesso `request-id` della `GetRequest` a cui risponde. Il manager scarterà risposte che portino un `request-id` che lui non si aspetti.

Quando la risposta dell'agent tarda ad arrivare, il manager cancella l'attesa della risposta (e invalida il suo `request-id`) e ripete la richiesta (per un certo numero di volte). Questo comportamento è motivato dall'esigenza di correlare la risposta alla domanda anche "nel tempo", per poter associare allo stato dei managed object riportato dalla risposta un certo istante temporale in cui questi valori erano in effetto. Se la nuova domanda ripetuta riportasse lo stesso `request-id` della domanda precedente, non sapremmo a quale delle domande la risposta fa riferimento. La inattesa conseguenza di tale comportamento è però che se il manager viene tarato ad aspettare *troppo poco* le risposte dell'agent non riesce ad reperire nessun valore dall'agent!

Il campo `request-id` serve solo perché l'agent deve rispondere riportando lo stesso `request-id`. Il manager può usare i `request-id` come vuole, anche senza farli crescere monotonicamente.

In SNMPv1 la `GetRequest` è atomica, cioè l'agent risponde con tutti i variable binding richiesti oppure con nessuno dei variable binding richiesti; in SNMPv2 ci sarà una certa modifica di comportamento (non appariscente) che illustreremo. Questo comportamento viene spesso descritto con il termine *best effort*, anche se non è propriamente corretto in quanto nel caso di fallimento parziale la risposta non porta alcuna informazione. Questa specifica rende più semplice da implementare il comportamento dell'agent, ma è molto "spiacevole" per il manager, il quale ha il compito di eseguire tutte richieste perfettamente corrette se vuole avere una risposta significativa.

SNMP definisce poche condizioni di errore per la `GetRequest`, che sono le seguenti:

- L'OID richiesto non esiste nella MIB, oppure il tipo non è semplice: allora il campo `error-status` della `GetResponse` vale `noSuchName` e `error-index` indica la posizione del (primo) variable binding che ha causato errore. Si noti che non vengono segnalati eventuali altri variable binding che provocano errore.
- Le risorse necessarie a generare la risposta eccedono limiti locali dell'agent: allora `error-status` vale `tooBig`. Le risorse "mancanti" possono essere la memoria

necessaria a contenere la risposta, oppure il tempo di CPU. Mentre è chiara la ragione per cui un agent può abortire la generazione della risposta a causa della mancanza di memoria, meno chiara può essere la mancanza di tempo: sembrerebbe che in fondo l'agent può sempre "metterci un po' di più" nel generare la risposta, e quindi perché abortire la risposta? La ragione risiede nel fatto che una risposta che giunge troppo tardi è inutile perché il manager avrà cancellato la domanda ed eventualmente ripetuto la domanda ma invalidato il vecchio `request-id`.

- Altri tipi di errore: allora `error-status` vale `genErr` e `error-index` indica la posizione del (primo) variable binding che ha causato errore.

Come si può apprezzare i codici di errore sono veramente pochi, e questo semplifica il trattamento delle risposte.

SNMP permette di recuperare solo il valore di nodi foglia dell'albero dell'agent, perché solo questi sono istanze di object type e hanno sintassi semplice. Non si possono recuperare con un solo variable biding intere righe oppure intere tabelle. Il manager può recuperare una intera riga o una intera tabella solo indicando nella `GetRequest` tutti gli OID della riga o del tabella (e quindi deve conoscerli in anticipo per poter usare la `GetRequest`). Il manager deve anche fare attenzione a non superare i limiti dell'agent, perché in tal caso l'agent risponde con l'errore `tooBig` e nessun variable binding! Questa è una (grave) limitazione di SNMPv1!

Ma rimane il problema che abbiamo citato da tempo: *come si fa ad accedere ad un elemento di una tabella se non conosco il valore dei campi indice di quella riga*? E abbiamo visto che molte definizioni di MIB sembrano ignorare questo problema definendo tabelle in cui i valori degli indici sono ignoti agli operatori oppure tutti i campi sono indici! La soluzione offerta da SNMP fa udo di un'altra operazione, perché con la `GetResponse` il problema non ha soluzione.

7.2.3. GetNextRequest PDU

Questa operazione ha gli stessi parametri della `GetRequest`, ma la risposta non contiene il variable binding per ognuno degli OID della richiesta, ma per ogni OID della richiesta viene restituito il variable binding dell'*OID della prima foglia*, nell'albero degli OID dell'agent, *successivo* all'OID richiesto. Per questo è importante la definizione dell'ordinamento lessicografico precedentemente presentato. E' anche importante che la definizione di ordinamento si basi sul concetto di ordinamento degli OID invece che sul concetto di visita dell'albero degli OID dell'agent, perché in tale modo posso richiedere anche il next di un OID che non esiste nell'albero dell'agent.

Ovviamente, la `GetNextRequest` di una foglia mi restituisce il valore della foglia successiva. Se non conosco il valori dei campi indice posso usare la `GetNextRequest` per "scoprire" la prima foglia del sotto-albero di un object type colonnare. un buon esempio è fornito dal caso della tabella delle connessioni TCP. In questa tabella vi è un solo campo che non è indice e cioè tcpConnStatus; gli altri 4 campi, che riportano indirizzi IP e porta, locali e remoti, costituiscono l'indice della tabella. Se io conosco il descrittore di una connessione, per esempio perché su di un log viene tracciato, posso benissimo usare la `GetRequest` per esaminare il suo stato, perché posso "costruire l'OID del managed object che riporta il suo stato. Ma se invece voglio scoprire tutte le connessioni presenti in una entità TCP e il loro stato non posso "indovinare" tutti i descrittori e nemmeno provare tutti i possibili perché il numero è troppo grande. Con la `GetNextRequest` invece ho un modo di risolvere il problema.

Eseguo una prima `GetNextRequest` usando l'OID del object type `tcpConnStatus`. Nel varbind restituito da `GetNextRequest` trovo il valore dei campi indice codificato all'interno dell'OID, e il valore dello stato della connessione nel valore del varbind. E così ho scoperto la prima connessione nella tabella. Iterando poi le richieste, usando sempre come OID nella richiesta l'ultimo OID restituito scandisco tutta la colonna e mi fermo quando l'OID restituito indica un elemento della colonna successiva della tabella. Per scandire una intera tabella non serve quindi scandire le colonne dei campi che sono indice. Nel caso che tutti i campi siano indice della tabella è sufficiente scandire una sola colonna per ricavare i valori di tutti i campi!

Questa "decodifica" del valore dei campi indice "immersi" nell'OID di una istanza di un tipo colonnare richiede che quando ho nell'OID uno "spezzone" di lunghezza variabile questo sia preceduto da un intero che esprima la lunghezza dello "spezzone".

7.2.4. SetRequest PDU

La Set`Request` PDU ha lo stesso formato delle altre request (e il valore del tag permette di distinguerle, ovviamente), ma questa volta il campo valore dei varbind è significativo. La `SetRequest` modifica il valore di uno o più managed object. In caso di successo nella esecuzione dell'operazione, la risposta ha il solito formato `GetResponse`, con lo stesso `request-id` della richiesta, e gli stessi varbind della richiesta.

L'operazione è atomica: basta che fallisca un assegnamento perché nessun assegnamento venga eseguito, viene restituito un codice di errore e non viene restituito alcun varbind (lista vuota). Come nel caso della `GetResponse`, il valore del campo `error-index` della risposta riporta la posizione del varbind, nella lista di varbind, che ha causato il fallimento dell'operazione. Vi possono naturalmente anche essere altri varbind "non corretti", ma l'agent si arresta la primo varbind che provoca errore.

Oltre agli errori della `GetRequest` (`noSuchName, tooBig`, `genErr`), un possibile errore è `badValue` che viene ritornato quando (almeno) un varbind è inconsistente (tipo, lunghezza o valore attuale).

7.2.4.1. Aggiornare una tabella

Lo standard RFC 1157 non dice nulla di particolare sull'uso della `SetRequest` nel caso di istanze di object type colonnari. Se l'istanza da modificare non contiene un valore che è anche indice della tabella non ci sono problemi di interpretazione: ma occorre usare i valori "giusti" degli indici nell'OID del varbind della richiesta.

Modificare invece una istanza che contiene un valore che è anche indice della tabella da invece problemi di interpretazione, prima che di esecuzione. Infatti, se uso il vecchio valore come indice allora il nuovo oggetto avrebbe un OID errato. Se uso invece il nuovo valore nell'OID allora l'oggetto da modificare non esiste! In altre parole, il dubbio è:

- Si vuole aggiungere una nuova riga che ha un indice diverso da quello esistente? oppure
- Si vuole cancellare la riga precedente e aggiungerne una nuova?

Cambiando punto di vista, domandiamoci come si può operare per eseguire due operazioni sicuramente "sensate": *aggiungere una riga*, e *rimuovere di una riga.*

Aggiunta di una riga ad una tabella

Per indicare questa sua volontà il manager deve inviare una operazione di `SetRequest`

che assegni ad ogni componente della nuova riga il suo valore, assegnando *coerentemente*

- i valori indici ai campi indici, e
- i valori desiderati ai campi che non sono indice

Si noti che vi sono numerosi vincoli di integrità fra i valori assegnati e i valori degli OID usati nell'assegnamento, perché tutti gli OID usati devono contenere già i valori degli indici che stiamo costruendo mediante questo assegnamento.

Il compito dell'agent non è facile perché tutti questi oggetti non esistono ancora nella sua MIB, e quindi, dopo aver capito che si desidera aggiungere la riga, deve verificare che gli assegnamenti siano coerenti, e infine interagire con la risorsa perché esegua l'operazione che astrattamente corrisponde alla creazione di una nuova riga.

Un buon esempio è fornito dal caso della `ipRouteTable`: il manager deve invocare qualcosa tipo

```
SetRequest(
        (ipRouteDest.11.3.3.12=11.3.3.12),
        (ipRouteMetric.11.3.3.12=9),
        (ipRouteNextHop.11.3.3.12=91.0.0.5)

                )
```

per aggiungere una nuova riga con destinazione 11.3.3.12, metrica 9 e next hop 91.0.0.5. Infatti il campo destinazione di una route è indice della tabella.

Poiché l'operazione di creazione di una riga richiesta all'agent è complessa, l'RFC1212 indica che l'agent può comportarsi in tre modi diversi:

- Rifiutare l'operazione con un `error-status` di `noSuchName`. Quindi non è obbligatorio implementare questa complessa operazione, che evidentemente molti costruttori non volevano sobbarcarsi.
- Accettare di compiere l'operazione ma rilevando poi un errore di integrità negli assegnamenti richiesti rispondere con un `error-status` di `badValue.`
- Accettare l'operazione e, verificata la sua correttezza, eseguirla creando la nuova riga.

Cosa succede se il manager crea solo gli oggetti i cui valori costituiscono indice della nuova riga quindi cerca di creare una *riga incompleta*; ad esempio mediante

```
SetRequest(
        (ipRouteDest.11.3.3.12=11.3.3.12))
```

Anche in questo caso SNMP non è troppo prescrittivo; l'agent può

1. Aggiungere una nuova riga fornendo dei valori di default per i campi non specificati dal manager
2. Rifiutare l'operazione perché non sono forniti i valori di tutti i campi della riga

7.2.4.2. Cancellazione di una riga

SNMP non prevede una operazione specifica; in una architettura che imposta la gestione esclusivamente in termini di *operazioni su dati di gestione*, occorre usare ancora la SetRequest per indicare che la riga della tabella è *logicamente cancellata*. I dati presenti in una tabella sono però decisi dalla specifica della MIB e quindi è necessario che nella specifica della MIB sia previsto che il valore di un opportuno campo della tabella abbia appunto il significato di invalidare la corrispondente riga.

Nel caso della `ipRouteTable`, è il campo `ipRouteType` che ha questo ruolo; ad esempio

```
SetRequest(
        (ipRouteType.11.3.3.12=invalid))
```

L'effetto è dunque quello della cancellazione logica; l'agent può decidere se effettuare anche la cancellazione fisica (la riga scompare dalla tabella) oppure solo quella logica.

È importante comprendere che è nella specifica della MIB che deve essere previsto questo significato/effetto! È nel momento della specifica che si prescrive che l'agent debba implementare operazioni di cancellazione sulla tabella. Peccato che SNMPv1 sia poi molto permissivo e accetti che una agent non implementi completamente questo aspetto della MIB e ciononostante si possa vantare di essere "conforme" allo standard!

7.2.4.3. Eseguire una azione

SNMP non prevede un comando specifico per richiedere all'agent la esecuzione di una particolare azione, nuovamente a causa del suo orientamento all'accesso/modifica di dati di gestione. Si può usare la `SetRequest` su particolari oggetti per chiedere all'agent di eseguire una azione.

Un buon esempio è dato dall'oggetto `ifAdminStatus`, la cui specifica richiede che l'agent metta off-line la interfaccia quando il manager lo setta al valore `down(2)`, e la riporti on-line quando viene settato a `up(1)`.

7.2.4.4. Il caso curioso dell'errore readOnly

Una attenta lettura della specifica in ASN.1 delle PDU di SNMP mostra che uno degli error-status che possono essere restituiti è readOnly(4). Viene da pensare che debba essere restituito quando si cerca di modificare un oggetto che è `read-only`, ma non è così.

Nella specifica del comportamento della `Set`, l'RFC1157 dice che quando l'oggetto non è disponibile per scrittura nella MIB view della community si deve restituire noSuchName e come error-index la posizione del varbind nella lista dei varbind. Non vi sono altri riferimenti al codice di errore `readOnly`, che quindi non deve essere restituito in alcun caso: questa "inconsistenza" è probabilmente dovuta ad un errore di "scrittura" dell'RFC!

A parte la curiosità, il problema vero è che se l'OID su cui si vuole operare non esiste nella MIB, anche in questo caso verrebbe restituito l'errore `noSuchName`, che quindi è ambiguo.

Per distinguere i due casi il manager deve fare una `Get` sull'OID in questione: se la `Get` ha successo allora è un problema di mancanza di diritti di accesso, se la `Get` fallisce ancora con `noSuchName` allora l'OID non esiste.

7.2.5. Trap PDU

Questa PDU viene inviata dall'entità SNMP quando l'agent vuole fornire al manager la notifica asincrona di un evento significativo assieme al valore di alcuni oggetti significativi. I campi sono:

- *tipo di PDU*
- *enterprises*: è un OID che serve per identificare il sottosistema che ha generato la trap (lo stesso valore che esiste in `sysObjectID`)

- *agent-addr*: l'indirizzo IP dell'agent che genera la trap
- *generic-trap*: uno dei tipi di trap predefiniti in SNMPv1
- *specific-trap*: un codice che indica più specificamente la natura della trap
- *time-stamp*: il tempo passato (in centesimi di secondo) dalla ultima (re)inizializzazione dell'agent al momento in cui la trap è stata generata; perché il manager non lo può ricavare dal momento in cui ha ricevuto la trap? Si capirà fra poco ...
- *variablebindings*: contengono informazione addizionale relativa alla trap (spesso almeno in parte implementation-specific)

Il campo `generic-trap` ha solo 7 varianti:

- *cold-start(0)*: l'entità SNMP inviante si sta reinizializzando, tipicamente a causa di un grave errore o guasto, e lo stato dell'agent è alterato (resettato)
- *warm-start(1)*: reinizializzazione a caldo senza perdita di stato
- *linkDown(2)*: segnala il fallimento di una interfaccia di rete; il primo elemento dei `variablebindings` è costituito dal nome e dal valore della istanza di `ifIndex` della interfaccia in causa; quindi identifica la riga della tabella che descrive la interfaccia
- *linkUp(3)*: segnala che una interfaccia di rete è tornata in funzione; il primo elemento dei `variablebindings` è costituito dal nome e dal valore della istanza di `ifIndex` relativa alla interfaccia in causa; quindi identifica la riga della tabella che descrive la interfaccia
- *authenticationFailure(4)*: segnala al manager che l'entità SNMP ha ricevuto una richiesta per la quale è fallita l'autenticazione
- *egpNeighborLoss(5)*: è un errore del protocollo di calcolo delle tabelle di routing
- *enterpriseSpecific(6)*: segnala un evento non standard, specifico del vendor; il valore di `specific-trap` indica il tipo di trap che è avvenuto; i `variablebindings` sono in questo caso sempre vendor-specific

A differenza degli altri comandi SNMP la `Trap` non richiede nessuna risposta da parte del manager (e non è possibile dare risposta). La ragione di questa strana scelta è certamente nel fatto (storico) che la `Trap` era stata pensata per situazioni in cui non si ha comunque tempo per aspettare la risposta e ripetere l'invio in caso contrario. Ma in questo modo la `Trap` è vista come uno strumento su cui non si può contare per fare fault management.

Come si deve allora organizzare la gestione usando SNMP? Per rispondere alla domanda, e per capire a fondo la ragione del progetto di una operazione di Trap organizzata in questo modo molto strano, dobbiamo ragionare sul tipo di transport che si deve/può usare.

7.3. Supporto a livello di Transport

SNMP richiede un livello di trasporto, perché è un protocollo applicativo, ma non vuole fare nessuna ipotesi sulla affidabilità o meno del trasporto, che sia connection-less o connection-oriented, all'interno di Internet o al di fuori (volevano progettare un protocollo di gestione valido per suite di protocolli di rete!). Quindi di fatto è progettato contando solo sul livello di servizio più povero: il servizio connection-less. Come vedremo alla fine di questa discussione, nei progettisti c'era però anche la convinzione che fosse "cosa buona e

giusta" utilizzare un servizio connection-less per effettuare la gestione.

7.3.1. Servizio di Trasporto connection-less

La maggior parte delle implementazioni di SNMP usa la infrastruttura TCP/IP e usano il trasporto UDP, per il quale lo standard specifica le regole di uso (mentre non specifica come usare TCP e quindi non è standard usare TCP).

Lo standard prevede che l'agent ascolti sulla porta 161 per le operazioni di `Get`, `GetNext` e `Set`, e il manager ascolti sulla porta 162 per le `Trap`. Le due diverse porte permettono di far coesistere tranquillamente su una macchina sia un manager che un agent, e permette di dare priorità alle `Trap` nel caso che si usi la porta 161 sia per inviare le richieste di operazione di manager che per ricevere le risposte a tali operazioni (questo permetterebbe di dare maggiore priorità alle operazione di gestione rispetto alle operazioni di altri protocolli applicativi).

Può anche essere usato il servizio equivalente di ISO-OSI (CLTS) e in tal caso l'equivalente delle porte, il *Transport Selector*, deve essere "`snmp`" sull'agent e "`snmp-trap`" sul manager.

Poiché UDP non garantisce l'affidabilità, è l'applicazione che lo usa (manager o agent) che si deve preoccupare della affidabilità. Lo standard non prescrive molto di più e si affida al buon senso degli implementatori. Se manca risposta a una richiesta di operazione, il manager può ripetere la domanda alcune volte assumendo che sia stata persa la domanda o la risposta.

Il `request-id` permette di risolvere il problema delle risposte duplicate, anche se alcune implementazioni generano diversi `request-id` per domande ripetute e invalidano i `request-id` per cui è scaduto il timeout, così finendo in un vicolo cieco se il timeout è fissato troppo basso. La ragione di questo comportamento apparentemente strano è che il manager è interessato a collocare "nel tempo" la risposta che gli giunge dall'agent, anche se con i margini di errore nell'apprezzamento del tempo causati dal ritardo di rete e dal ritardo introdotto dall'agent nel rispondere. Ma se il manager prendesse per buona una risposta che in realtà è stata inviata a fronte di una domanda effettuata molto tempo prima, non avrebbe controllo sul margine di errore. Quindi cambiare `request-id` ad ogni ripetizione di richiesta e invalidare i `request-id` scaduti permette di stimare l'errore che si commette nel valutare l'istante in cui i valori restituiti dall'agent erano effettivamente validi sul dispositivo controllato.

Nel caso di mancata risposta ad una `Set` si dovrebbe verificare con una `Get` se l'azione è stata eseguita o meno (vi possono essere operazioni che non sono idempotenti o che provocano effetti laterali).

L'agent non ha modo di verificare che la `Trap` inviata sia stata ricevuta o meno: il manager deve pollare con frequenza *opportuna* l'agent per verificare di non aver perso una trap. Nelle MIB vi sono spesso oggetti il cui scopo è appunto quello di permettere al manager di capire se ha perso alcune `Trap`, e contare le trap eventualmente andate perse.

7.3.2. Servizio di Trasporto connection-oriented

SNMP è stato pensato per essere usato su un trasporto connection-less perché questo aggiunge robustezza al sistema di gestione. Se SNMP contasse sulla esistenza della connessione, la mancanza di tale connessione avrebbe effetti disastrosi sul funzionamento di SNMP.

Ma vi sono ulteriori ragioni per preferire l'uso di un trasporto connection-less, che hanno indotto i progettisti di SNMP a scegliere una architettura di gestione che non ipotizza il servizio "forte" offerto da un trasporto connection-oriented.

Mantenere una connessione aperta richiede molte risorse, e quindi non contare su una connessione aggiunge robustezza. Infatti, una connessione:

- Consuma risorse (buffer e descrittori di connessione) sia sugli agent che sui server;
- Consuma banda per l'apertura della connessione e per la sua chiusura;
- Consuma banda per gli ack.

Nei sistemi di gestione lo scambio dati fra agent e manager è normalmente periodico anche se non "continuo", come vedremo meglio nel seguito. Quindi non è necessario che le connessioni siano tenute sempre aperte: in questo modo però si consuma banda per aprirle e chiuderle, e si introducono ritardi. Ma d'altra tenere le connessioni sempre aperte comporta un grande consumo di risorse sul manager (buffer e descrittori di connessione).

Si potrebbe pensare che tenere le connessioni sempre aperte permette di rilevare la caduta della rete IP fra manager e agent, il che sarebbe effettivamente un grande vantaggio per un sistema di gestione che deve rilevare prontamente l'insorgenza dei guasti. Ma questa osservazione è in realtà "falsa" nel senso che TCP rileva la caduta della rete IP solo se vi è traffico fra le due entità. Se tutti gli ack sono stati ricevuti, TCP si accorge della caduta della rete IP solo quando cerca di inviare altri dati e non riceve i relativi ack. In uno schema di polling periodico verrebbe rilevato al prossimo poll, quindi esattamente con lo stesso ritardo medio che si ha usando UDP, ma senza pagare il prezzo di consumo di risorse che TCP comporta.

Questa ultima considerazione è probabilmente quella che ha indotto i progettisti di SNMP a scegliere di usare UDP, ed è la considerazione che dobbiamo fare per evitare di percepire l'uso di UDP come una debolezza di SNMP!

Per queste ragioni, non c'è nessuna specifica per l'uso di SNMP su TCP. Invece, i progettisti di SNMP hanno adottato una atteggiamento veramente "opportunistico quando hanno preso in considerazione la possibilità di proporre l'uso di SNMP in suite di protocolli diversi da Internet, ad esempio in ISO-OSI. In questa suite di protocolli, è definito un livello di rete che è connection-oriented, X.25, che era molto diffuso al momento della specifica di SNMPv1. Per poter proporre SNMP al disopra di X.25, RFC1283 spiega come usare SNMP sopra al servizio orientato alla connessione di ISO-OSI!

7.4. Il gruppo SNMP di MIB-II

Se abbiamo un sistema di gestione, dobbiamo anche poter gestire il sistema di gestione stesso, altrimenti proprio il nostro strumento più importante è fuori controllo! La MIB-II ha appunto lo scopo di permettere una gestione, molto limitata, della implementazione di SNMP su un agent o su un manager. Quindi non ha object type specifici per il sistema gestito, ma ha solo object type necessari per gestire la implementazione di SNMP stesso. Naturalmente, è obbligatorio implementare MIB-II su ogni agent, indipendentemente, in aggiunta, e in fondo prima ancora di implementare la MIB specifica del sistema gestito.

Per ridurre il costo degli agent, la MIB-II è stata progettata con stringenti requisiti di minimalità. Tutti gli object type sono `read-only`, tranne `snmpAuthenTraps` che abilita o disabilita l'invio di trap di autenticazione ai manager. Quindi è una MIB orientata al *monitoring* piuttosto che al *control*.

7.5. Aspetti pratici

7.5.1. Differenze nelle implementazioni

Vi sono numerosi problemi pratici che emergono dalle differenze nelle varie implementazioni che non rispettano gli standard. Sono da considerare con attenzione (leggere sul libro di testo) perché sono interessanti osservazioni, anche se sono abbastanza vecchie (cioè basate su vecchie implementazioni).

7.5.2. Oggetti non supportati

Questo è un aspetto molto spinoso, perché uno dei modi "preferiti" per ridurre il costo della implementazione di una MIB è quello di non implementare la *semantica* di alcuni oggetti, ovviamente quando implementare tale semantica è costoso o a volte impossibile perché il sistema gestito non è in grado di fare ciò che l'agent vorrebbe chiedergli. Ad esempio, se un sistema non implementa un contatore presente nella MIB oppure non fornisce una interfaccia programmativa per accedere a tale informazione o per ricostruirla, l'agent non potrà implementare correttamente la semantica del contatore.

L'RFC richiede che se si implementa un gruppo di una MIB occorre implementarlo completamente; alternativamente si può omettere la implementazione di tutto il gruppo, senza nemmeno rendere accessibili i managed object del gruppo. In questo modo, il manager deve confrontarsi con un minor numero di "variabilità" nei sistemi gestiti che ha di fronte. E' soprattutto molto importante non fare comparire nella MIB dell'agent oggetti che non sono completamente implementati nella loro semantica.

Sfortunatamente, molti venditori, vuoi per ignoranza vuoi per ingannare l'acquirente, invece rendono disponibile l'oggetto ma senza semantica. Pensate a cosa succede se l'oggetto è un contatore di errori, che vale sempre zero: induce in errore l'amministratore che sta cercando di diagnosticare un guasto!

7.5.3. Scelta del software del manager

Anche in questo caso sono molto interessanti le considerazioni che vengono fatte dal libro di testo, in particolare riguardo ai requisiti che il software del manager dovrebbe soddisfare.

7.5.4. Frequenza di Polling

Questo è un argomento molto importante, anzi, addirittura essenziale visto che abbiamo motivato la necessità di ricorrere al polling per realizzare un sistema di gestione robusto (e se un sistema di gestione non è robusto, non è un buon sistema di gestione!). Dato che le trap non sono affidabili, occorre che comunque il manager contatti periodicamente (polling) gli agent che gli sono affidati, e verifichi i valori della MIB che possono indicare situazioni problematiche.

Come raccolgo queste informazioni che possono indicare situazioni problematiche? Queste informazioni sono raccolte usando `GetRequest` e `GetNextRequest`, e sono comunque storicizzate nel DB del manager. Che politica usare per questo polling? La scelta è difficile, per la politica da adottare dipende da molti fattori quali:

- Il *ritardo* con cui ci vogliamo accorgere dei problemi nel caso che la trap venga persa
- La *dimensione* della rete in termini

    - di *geografia/topologia*
    - di *numero di agent*
    - di *quantità di informazioni* da raccogliere per ciascun agent
- La *capacità* della rete di assorbire traffico
- La *potenza* della stazione di gestione

Tutti questi fattori influenzano la frequenza di polling che posso/devo mettere in atto e le informazioni di gestione che posso/devo raccogliere, ma fra di loro interferiscono in modo conflittuale e quindi il buon amministratore deve trovare il giusto compromesso (che può anche non esistere e in questo caso sono dolori!).

Facciamo un esempio semplice, supponendo per iniziare che il manager non abbia problemi di potenza di calcolo, e che possa fare polling di un solo agent alla volta. Quando avremo completato questo primo esame della situazione vedremo che le ipotesi fatte, soprattutto quella del polling di un agent alla volta, non sono restrittive perché i veri problemi derivano in realtà da altri fattori.

Indichiamo con:

Delta = tempo medio per completare il poll di un agent;

T = l'intervallo di polling desiderato;

allora il numero massimo di agent controllabili è

$$N \leq T / Delta$$

Il valore *N* è in fondo un nostro "desiderio": abbiamo un certo numero di agent nel nostro sistema da gestire e li possiamo contare. Vorremmo usare un certo periodo di polling *T*, ma vediamo che non siamo liberi di usare il valore che vogliamo, e che abbiamo un limite inferiore per tale periodo che dipende da *N* (oppure un limite superiore a *N* dato *T*) e da *Delta*: da cosa dipende *Delta*?

ll tempo medio *Delta* necessario per completare il poll dipende da un gran numero di fattori (non possiamo fissarlo noi!):

Tempo di generazione della richiesta sul manager: per preparare la PDU da spedire il manager impiegherà un certo tempo di calcolo e di I/O per reperire le informazioni di configurazione che dettano le informazioni da raccogliere; quanto maggiori sono le informazioni da raccogliere, tanto maggiore sarà questo tempo; tanto più potente è la CPU e il sistema di I/O del manager tanto più breve sarà questo tempo.

- *Ritardo di rete fra manager e agent*: questo tempo dipende dalla topologia della rete, e dalle tecnologie utilizzate nelle sottoreti; quanti più router vi sono fra manager ed agent, tanto maggiore sarà questo tempo; diverse tecnologie utilizzate nella sottorete possono variare il ritardo di trasmissione (se usiamo un link satellitare...).
- *Tempo di elaborazione della richiesta*: l'agent deve verificare la richiesta e preparare la risposta interagendo con il sistema gestito; tante maggiori saranno le informazioni richieste tanto maggiore sarà questo tempo; tanto più potente sarà la CPU dell'agent e del sistema gestito e tanto minore sarà questo tempo.
- *Tempo di generazione della risposta sull'agent*: una volta ottenute le informazioni, l'agent deve costruire la PDU di risposta, e anche in questo caso valgono le considerazioni fatte riguardo al tempo di elaborazione della domanda, solo che in questo caso è solo la potenza dell'agent che entra in gioco.

- *Ritardo di rete fra agent e manager*: valgono le stesse considerazioni fatte riguardo al ritardo di rete fra manager e agent, osservando solo che non è etto che il cammino seguito dalla risposta sia lo stesso che viene seguito dalla domanda, anche se molto preferibile che sia lo stesso!
- *Tempo di elaborazione della risposta sul manager*: valgono le stesse considerazioni fatte riguardo al tempo di generazione della richiesta sul manager; si osservi però che il manager dovrà eseguire una qualche "logica applicativa" sui valori ottenuti, certamente per archiviarli, ma in genere anche per verificare che non vi siano condizioni di errore o di pericolo.
- *Numero di oggetti da reperire sull'agent*: questo è un fattore importante perché determina, in misura minore o maggiore tutti i tempi precedentemente illustrati: sicuramente più informazione richiedo agli agent e maggiori sono tutti i tempi! E notate che se aumentando il numero di oggetti di cui chiedo il valore supero la dimensione massima di un pacchetto, tutti i tempi di ritardo di rete almeno raddoppiano.

Proviamo a fare una valutazione numerica, in un esempio reale, che si trova anche in letteratura: abbiamo una sola LAN e un periodo di poll di 15 minuti, con un tempo di elaborazione di 50 msec e un ritardo di rete di 1 msec, si ha che Delta è 0,202 sec (quattro elaborazioni e due ritardi). Quanti agent riusciamo al massimo a pollare? Vediamo:

$$N \leq (15 \times 60) / (50x4+2x1)/1000=900/0,202 \approx 4500$$

Questo nell'ipotesi di una semplice LAN, ma se la rete è costituita da hub, switch, router e tratte geografiche, si può arrivare benissimo a ritardi di mezzo secondo, e allora il numero massimo di agent cambia:

$$N \leq (15 * 60) / (4 * 0,05 + 2 * 0,5) \approx 750$$

E nel caso di link geografici la MTU può essere minore che nel caso di una LAN e quindi servono più pacchetti per leggere tutta la MIB interessata. Il problema sorge perché SNMP funziona meglio se non si usa la frammentazione perché la frammentazione provoca un aumento del tasso di perdita dei datagrammi UDP (basta perdere un frammento perché sia perduto l'intero datagramma). Se quindi dobbiamo usare una rete in cui la MTU minima è di 128B avremo una probabilità di perdita 8 volte superiore a quella che abbiamo se la MTU minima è 1024B. Per evitare la frammentazione il povero amministratore si deve industriare per spezzare le sue richieste in 8 pacchetti. Vediamo che effetto ha ciò (il che è equivalente a valutare l'effetto di richiedere 8 volte più informazione.

Ho 8 coppie di ritardi di rete da "pagare" e quindi le macchine gestibili in 15 minuti diventano:

$$N \leq (15 * 60) / (4 * 0,05 + 8*2 * 0,5) = 900/8,2 \approx 109$$

Vedete che il numero di dispositivi gestibili diminuisce moltissimo! E 15 minuti non è poi una grande prontezza di intervento, se si perdono per avere la segnalazione del guasto, nel caso peggiore. Occorre riflettere bene sull'effetto del tempo di polling sulla "qualità" del sistema di gestione. Se il periodo di polling è 15 minuti, il tempo massimo di rilevazione dell'errore, nel caso peggiore, è ovviamente 15 minuti. E questo nel caso facile che la lettura mi permetta di identificare in tempo nullo l'elemento che è guasto. Se invece occorre una analisi più approfondita per fare la diagnosi, il tempo per individuare il guasto è ancora maggiore. Solo dopo che ho fatto la diagnosi posso iniziare a risolvere il guasto e quindi il tempo di risoluzione è certamente superiore al tempo di diagnosi.

Un sistema di gestione viene valutato in primo luogo sul *tempo massimo* che impiega, nel caso peggiore, per *risolvere il guasto.* Il periodo di polling quindi deve essere

sensibilmente minore del tempo massimo che assegnano all'amministratore del sistema, per risolvere il guasto. In genere, questo tempo massimo di risoluzione viene "imposto" dall'esterno, cioè costituisce un importante requisito richiesto al sistema di gestione. Da questo tempo massimo di risoluzione, dobbiamo calcolarci il tempo massimo di rilevazione e quindi il periodo di polling massimo. Se il numero di agent che dobbiamo controllare è inferiore al numero che teoricamente potremmo controllare in questo tempo massimo, tenendo conto di *Delta* allora possiamo pensare di avere risolto il problema.

Sfortunatamente non siamo ancora sicuri di avere risolto il problema, perché occorre anche considerare quanta banda di rete consumiamo per il polling, cioè quanto traffico generiamo per fare polling, soprattutto nelle risposte degli agent che sono quelle che portano più byte perché ci sono i valori degli OID. Vediamo di fare un po' di conti.

Se ipotizziamo sempre 1 KB per poll di agent, abbiamo nel caso della LAN

$$4500*1KB = 4,5MB$$

ogni 15*60=900 secondi. Quindi aggiungiamo un traffico di 5 KB/sec. Su una LAN è sopportabile, ma se vi sono dei link geografici, questo traffico aggiuntivo può diventare inaccettabile a seconda della velocità del link. Supponiamo di avere un link a 128 Kbit/sec, che corrispondono a 16kB/sec: il traffico del polling costituirebbe circa il 30% della capacità totale del link. In linea generale, non è buona pratica usare per la gestione più del 10% della banda disponibile (non ce lo permetterebbero) e quindi nell'esempio avremmo a disposizione solo 1,6 KB/sec.

Se quindi consideriamo i limiti ammessi sulla banda da usare, possiamo da questi limiti ricavare il numero massimo di agent che possiamo gestire in 15 minuti. Per esempio, quanti agent possiamo gestire attraverso questo link geografico? Calcoliamo il numero massimo di byte che possiamo trasferire usando una banda di 1,6kB/sec: 1,6*900=1440 kB totali, quindi circa 1440 agent (abbiamo bisogno di 1kB per agent) invece dei 4500 su una LAN.

Alternativamente, possiamo partire dalla banda a disposizione e dal numero di agent da pollare per calcolare il periodo di poll, ottenendo così il migliore sistema di gestione realizzabile utilizzando al massimo la banda che ci mettono a disposizione. Se abbiamo 720 agent possiamo ridurre il periodo di poll a 7,5 minuti, e così via.

Questi conti sulla banda usata mostrano che l'ipotesi di fare un poll alla volta non è così strana: potrei anche fare i poll in parallelo, per usare bene la CPU del manager, ma in questo modo genererei un traffico che la rete non potrebbe sopportare o che non mi permetterebbero di generare.

Perché i limiti sulla banda usata? Tenete conto che i link geografici sono affittati e quindi costano un tanto al mese: il 10% di questo costo viene attribuito al sistema di gestione, mentre le LAN sono considerate un costo di investimento, ed è molto raro che ne venga attribuito il costo di ammortamento ai vari utilizzatori. Conseguenza: sui link geografici potranno richiedere che la quota di banda utilizzata dalla gestione sia ancora inferiore. Servirebbe quindi poter gestire localmente i dispositivi e poi fare parlare i manager per una gestione di livello superiore.

Come dobbiamo operare quando ci troviamo di fronte a vincoli contraddittori, tipicamente dovuti a tempi di risoluzione molto brevi, numero di agent molto alti e poca banda a disposizione? Vi sono due linee di intervento possibili.

Il primo intervento consiste nell'analizzare con attenzione quali sono le risorse su cui vi chiedono di avere quel tempo massimo di risoluzione dei problemi. Dalla individuazione di

queste risorse, possiamo ricavare le risorse che sono necessarie al loro corretto funzionamento: difficilmente gli utilizzatori del sistema informativo saranno in grado di dire che un certo router è un sistema critico, ma si limiteranno a dire che è critica la applicazione o il servizio da cui dipende il "business" dell'organizzazione. Sono gli amministratori che devono capire quali sono le risorse necessarie al funzionamento della applicazione o del servizio "business critical". Le risorse critiche sono per fortuna molto meno di quelle che sono gestibili e quindi questo analisi porta ad una riduzione del numero di agent da pollare.

Se questa riduzione non è ancora sufficiente, potete cercare di avere una più fine identificazione dei tempi di risoluzione dei guasti, in modo da avere tempi diversi per applicazioni/servizi con diverso livello di criticità per l'organizzazione, che andranno quindi pollati con periodi diversi.

Se ancora non riusciamo ad avere una soluzione alle richieste di tempi, e banda disponibile, dobbiamo analizzare la quantità di informazione da richiedere ad ogni agent (o se preferite questa è la prima analisi da compiere): quali informazioni sono necessarie per individuare la insorgenza di una guasto? Chiaramente è meglio limitare il poll alle informazioni veramente necessarie in modo da diminuire la dimensione delle risposte e quindi la esigenza di banda di comunicazione. Notate che questa analisi va fatta dispositivo per dispositivo e guasto per guasto, ma certamente permette sia di ridurre la banda richiesta, sia di prepararsi una procedura di analisi della segnalazione e di individuazione del guasto, perché ogni informazione richiesta nel poll è legata ad un guasto che essa permette di individuare.

Cosa possiamo fare se nonostante tutto questo lavoro ancora i tempi di risoluzione sono incompatibili con la banda messa a disposizione? L'unica risposta possibile è che avete a disposizione tutti gli elementi per spiegare perché i requisiti sono irrealizzabili e quindi vanno modificati: è facilissimo chiedere tempi di risoluzione tendenti a zero e rendere disponibili risorse tendenti a zero per ottenerli. Ma non potete pretendere che chi non è un esperto sappia riconoscere questo problema senza fatica: dovete spiegare e dimostrare le vostre esigenze!

Abbiamo a questo punto completato le considerazioni che si possono fare sul progetto di un sistema di gestione volto a *ottimizzare la qualità del sistema di gestione espressa in termini di tempo massimo di risoluzione dei guasti.* Dato che ridurre il tempo di risoluzione dei guasti è costoso, occorre organizzare il sistema di gestione in modo che fornisca il tempo massimo di risoluzione dei guasti che è necessario: né di più, né di meno!

Ma la qualità di un sistema di gestione non è valutata solo in termine di tempo massimo di risoluzione, ma anche in termini di *tempo medio di risoluzione dei guasti.* Quindi non ci interessa solo il tempo massimo di rilevazione del guasto ma anche il tempo medio di rilevazione.

Il tempo medio di rilevazione basato sulla attività di polling è pari a metà del periodo di polling, supponendo che il processo stocastico di generazione dei guasti sia poissoniano.

Possiamo fare meglio? Vediamo quale è il ruolo delle trap.

Il ruolo delle trap

Sono utili o tanto vale disabilitarle tanto devo fare lo stesso polling? Se mi preoccupassi solo del consumo di banda di comunicazione e del tempo massimo di rilevazione del guasto, disabiliterei le trap perché in ogni consumano banda e dato che non sono garantite arrivare non migliorano il tempo massimo di rilevazione del guasto.

Le trap non sono garantite arrivare però arrivano con un certa probabilità *tr* con *tr< 1*. Quando una trap arriva a destinazione, possiamo assumere che il tempo di individuazione della condizione di guasto sia nullo. Questo non cambia il tempo di individuazione dell'errore nel caso pessimo che si verifica quando

- La trap viene persa, e
- Il guasto si verifica esattamente dopo che è stato effettuato il polling che lo avrebbe "scoperto".

Proviamo però a calcolare il tempo medio (in secondi) di individuazione dell'errore:

$$Tm = 0*tr + 15*60*(1-tr)/2$$

Se non avessimo le trap il tempo media sarebbe

$$Tm = 15*60/2$$

Se tr è ragionevolmente vicino a 1 (cioè lo stato della rete IP è ragionevole, altrimenti il problema è la rete IP invece che il guasto in questione!) ottengo un tempo medio (molto) minore! Il tempo medio di risoluzione è il secondo importante parametro della qualità della gestione, e quindi le trap sono molto importanti, anche se non è garantita la loro consegna al manager.

A questo punto dovrebbe essere chiara la filosofia di progetto di SNMP: non importa se le trap è garantite che siano consegnate, perché il loro ruolo è quello di abbassare il valore medio (statistico) del tempo di rilevazione e non di garantire tempo di rilevazione nullo. Quando ci si occupa di *guasti*, che avvengono in modo *non deterministico*, non possiamo comunque garantire la consegna delle trap, anche se usassimo un trasporto connection-oriented e se le trap fossero riscontrate. Allora è meglio usare un trasporto che consuma meno risorse come è quello connection-less.

7.5.5. Le limitazioni di SNMPv1

SNMPv1 presenta molte limitazioni che a questo punto dovrebbero essere evidenti (se non non sembrano evidenti al lettore, è bene che rilegga e studi con attenzione il materiale presentato nei capitoli precedenti). Ecco le principali limitazioni:

- Non è adatto per reti veramente grandi,
    - perché nel fare polling genera molto traffico, e
    - perché serve un pacchetto grande di richiesta per avere un pacchetto grande di risposta (quindi tanta banda per le risposte ma anche tanta banda per domande).
- Non è adatto per reperire grandi quantità di dati come ad esempio una tabella di routing, perché deve fare una domanda per ogni elemento di una colonna; quindi per scandire una tabella devo fare tante domande quante sono le righe della tabella (nell'ipotesi che gli elementi della riga a cui sono interessato non producano un pacchetto di dimensioni eccessive).
- Le trap non sono riscontrate né sono affidate ad un meccanismo di trasporto affidabile. L'uso di TCP non è definito nello standard.
- SNMPv1 fornisce solo una autenticazione banale; quindi è adatto solo per il monitoring.
- Non supporta direttamente "comandi imperativi", l'unico modo per ordinare all'agent di compiere una azione è di specificare nella MIB l'azione come effetto laterale di una `Set` su una variabile; comunque non è possibile ottenere direttamente dei risultati

dall'esecuzione dell'azione (occorre che vengano resi disponibili come valori di managed object da leggere dopo che la operazione è completata).

- Il modello della MIB di SNMP è troppo semplice e non supporta veramente la distinzione fra object class e object value.
- SNMPv1 non supporta comunicazioni manager-to-manager, che invece sarebbe necessaria per trattare in modo gerarchico sistemi distribuiti di grandi dimensioni.

Alcune di queste limitazioni sono indirizzate da SNMPv2 (ad es. con una nuova operazione chiamata `GetBulk`).