1 Lezione del 13-10-25

1.1 Rilevamento errori

Il *rilevamento degli errori* (**error detection**) è il meccanismo attraverso il cui verifichiamo se i dati arrivati attraverso il link fisico sono corretti.

Dobbiamo innanzitutto stabilire un codice di **rilevamento** degli errori. Per ogni datagramma trasmettiamo d bit di dati (i bit D) e r bit di *rilevamento errore*, o di **ridondanza** (i bit R). Minuscolo è il numero di bit, maiuscolo è il campo di bit vero e proprio. Sia i bit D che i bit R vengono trasmessi attraverso il link, che ricordiamo è *suscettibile ad errori*.

Per ricavare i bit R si applica una certa funzione $H()$ (sostanzialmente di *hashing*) sui bit D. Assunto che mittente e destinatario usino la stessa funzione $H()$, una volta ricevuti i dati il destinatario potrà applicare $H()$ sui bit D ricevuti e ottenere la sua copia dei bit R da confrontare con quelli ricevuti.

Se i bit R corrispondono non ci sono stati errori (a meno della sfortunata circostanza in cui sia i bit D che R sono stati alterati per risultare erroneamente corretti, si sceglie la funzione $H()$ in modo che questo sia difficilmente vero), altrimenti qualcosa è andato storto nella trasmissione sul link.

I bit R, nel contesto del *framing* visto in 9.5.1, vengono inseriti nel *trailer* del frame.

1.1.1 Algoritmi di rilevamento errori

Vediamo alcune possibili funzioni $H()$ usate per fare rilevamento errori.

- **Parity checking**: posto che si voglia trasferire una parola da d bit: si sceglie un solo bit R di ridondanza, determinato come segue:
  - Se il numero di bit a 1 fra i bit D è pari, si imposta il bit in R a 1;
  - Se il numero di bit a 1 fra i bit D è dispari, si imposta il bit in R a 0.

  In questo caso il bit in R viene detto **bit di parità** (*parity bit*). Questo algoritmo viene detto di tipo **even parity**: l'algoritmo opposto si direbbe **odd parity**.

  Questo algoritmo funziona bene solo nella circostanza in cui il numero di errori sui bit d è dispari (per quanto ci riguarda, 1). Quando si trasferiscono quantità relativamente basse di bit per parola su canali abbastanza affidabili (8, 16, 24 bit su rame o simili) abbiamo che è abbastanza sicuro. Con parole più grandi chiaramente può essere molto più soggetto ad errori.

- **Checksum**: questo sistema, detto della *"somma di controllo"* viene usato in UDP e TCP.

  In questo caso si prende ogni pacchetto e si divide in blocchi da 16 bit. Sommando questi blocchi bit-a-bit in *complemento a uno* si ottiene un campo R su $r = 16$ bit detto *checksum*, che può essere ricalcolato lato destinatario per fare rilevamento degli errori.

  Questo metodo è più sensibile rispetto al bit di parità, e viene usato perlopiù a livello di *trasporto* UDP o TCP (più che a livello *datalink*, dove si fanno altri tipi di controlli meno sensibili).

- **CRC** (*Cyclic Redundancy Check*): questo è il metodo usato più spesso in ambito di *Datalink*.

Supponiamo di avere R bit dati, e un *pattern* detto **generatore** G di $r+1$ bit (dove ricordiamo che R sono i bit di ridondanza e r il loro numero). Mittente e destinatario dovranno entrambi conoscere il generatore.

Si calcolano quindi il valore dell'operazione $< D, R >$ come:

$$< D, R >= D \times 2^r \text{ XOR } R$$

senza ancora sapere quanto vale R. poco chiaro

A questo punto il mittente dovrà scegliere i bit R in modo che $< D, R >$ diviso G abbia resto zero: il destinatario potrà quindi calcolarsi $< D, R >$ dai D ed R ottenuti sul link fisico, e noto G effettuare la divisione. Se il resto è diverso da 0, allora si rileva un errore di trasmissione.

Vediamo come R viene calcolato nella pratica. Vogliamo:

$$D \times 2^r \text{ XOR } R = nG \implies D \times 2^r \text{ XOR } R$$

per cui R dovrà soddisfare:

$$R = \text{mod}\,(D \times 2^r, G)$$

dove *mod* indica il resto della divisione fra i due argomenti (cioè l'operatore modulo).

1.2 Correzione errori

Dopo aver discusso il *rilevamento degli errori*, vediamo come procedere nella **correzione degli errori** in caso se ne rilevino.

La soluzione più semplice sarebbe quella di ritrasmettere i datagrammi sbagliati. Se poi si avesse un algoritmo di rilevamento errori che restituisse esattamente *quali* bit sono errati, basterebbe commutarli per risolvere gli errori.

In questo caso non serve più un codice a *rilevamento* degli errori, ma un codice a **correzione** degli errori. Per ogni datagramma trasmettiamo d bit di dati (i bit D) e r bit di *correzione errore*, o ancora di **ridondanza** (i bit EDC, da *Error Detection Code*).

Questi sono simmetrici al codice definito per il rilevamento errori: la differenza è che dati D ed EDC, lato destinatario possiamo rilevare esattamente l'errore di trasmissione (se c'è stato).

1.2.1 Algoritmi di correzione errori

Vediamo alcuni (1) modi per calcolare i bit EDC ed effettuare quindi la correzione degli errori.

- **Parity checking bidimensionale**: si sistemano i bit D in una struttura matriciale, e si calcolano i bit di parità per ogni riga e per ogni colonna.

  Ad esempio, si può dire:

$$D = \{0, 1, 0, ...\} = \{d_1, d_2, d_3, ...d_d\} \Rightarrow D_m = \begin{pmatrix} d_{1,1} & d_{1,2} & ... & d_{1,j} \\ d_{2,1} & d_{2,2} & ... & d_{2,j} \\ ... & ... & ... & ... \\ d_{i,1} & d_{i,2} & ... & d_{i,j} \end{pmatrix}$$

Posti i e j tali che $i \cdot j = d$ (numero di bit in D). A questo punto si orla D_m con i bit di parità di *riga* (parity_{row}) e i bit di parità di *colonna* (parity_{row}):

$$\begin{pmatrix} D_m & \text{parity}_{row} \\ \text{parity}_{col} & \end{pmatrix} = \left(\begin{array}{cccc|c} d_{1,1} & d_{1,2} & ... & d_{1,j} & d_{1,j+1} \\ d_{2,1} & d_{2,2} & ... & d_{2,j} & d_{2,j+1} \\ ... & ... & ... & ... & ... \\ d_{i,1} & d_{i,2} & ... & d_{i,j} & d_{i,j+1} \\ \hline d_{i+1,1} & d_{i+1,2} & ... & d_{i+1,j} & \end{array}\right)$$

dove

$$\text{parity}_{row}: \quad d_{r,j+1} = \text{parity}\left(\sum_{c=1}^{j} d_{r,c}\right)$$

$$\text{parity}_{col}: \quad d_{j+1,c} = \text{parity}\left(\sum_{r=1}^{i} d_{r,c}\right)$$

A questo punto un errore su un singolo $d_{r,c}$ verrà rilevato in quanto incrocierà una riga e una colonna (l'intersezione andrà commutata). Due o più errori su righe e colonne disgiunte veranno similmente rilevati, a meno di casi di errori multipli su più righe e colonnne (che potrebbero addirittura dare falsi positivi). Infine, un numero pari di errori sulla stessa riga o sulla stessa colonna ci permettono di rilevare, ma non correggere errori (vedremo 2 colonne sbagliate e una riga giusta, o viceversa, senza poter quindi incrociare).

Chiaramente i bit EDC nel trailer saranno molti di più: per la precisione $i + j$.

1.3 Trasferimento dati

Abbiamo visto alcuni algoritmi di *rilevamento errori*, introdotto l'ipotesi di effettuare un *reinvio* dei dati corrotti nel caso di errori, e visto anche un algoritmo di *rilevamento e correzione errori*.

Vediamo adesso come realizzare un livello superiore a quello *datalink*, riportando indietro l'ipotesi del reinvio dati, cioè il cosiddetto livello *transfer* o **trasferimento**, che considereremo un livello *affidabile*.

Le considerazioni che facciamo adesso saranno quelle che nel modello OSI vengono implementate nel cosiddetto livello **transport**.

L'idea è quella di usare i servizi offerti dal livello datalink per realizzare un'ulteriore astrazione, quella appunto di *trasferimento affidabile*. *"Confezioneremo"* quindi i pacchetti che vogliamo spedire in datagrammi livello datalink provvisti di un opportuno *header*.

1.3.1 Primitive di trasferimento

Per implementare questo tipo di livello *transfer* ci doteremo quindi di alcune primitive a servizio di un'altro *livello superiore*:

- `rdt_send()`: chiamata dal livello superiore (nella macchina *trasmettitore*), implementa l'invio sul canale affidabile (cioè implementa un protocollo **RDT** (*Reliable Data Transfer*));
- `udt_send()`: implementa l'invio sul canale inaffidabile fino al ricevitore;
- `rdt_rcv()`: implementa il ricevimento sul canale affidabile lato ricevitore, cioè ottiene i dati sul canale inaffidabile, e li corregge (se necessario);

- `deliver_data()`: si occupa di inoltrare i dati ottenuti attraverso il protocollo RDT al livello superiore (nella macchina *ricevitore*).

Notiamo che le primitive `udt_send()` e `rdt_recv()` dovranno implementare un qualche tipo di comunicazione bidirezionale (ad esempio se la `rdt_rcv()` vuole chiedere il reinvio di un frame perso).

1.3.2 Macchine a stati finiti

Per descrivere il protocollo **RDT** useremo il formalismo della *macchina a stati finiti* (**FSM**, *Finite State Machine*).

Una macchina a stati finiti rappresenta un *automa* dotato di **stati** e **transizioni** fra tali stati: dato uno stato ed un evento si può determinare la transizione successiva, e quindi come si evolve il protocollo.

Nelle prossime sezioni definiremo *iterativamente* versioni sempre più accurate di RDT rispetto a un qualche protocollo reale.

1.3.3 RDT 1.0

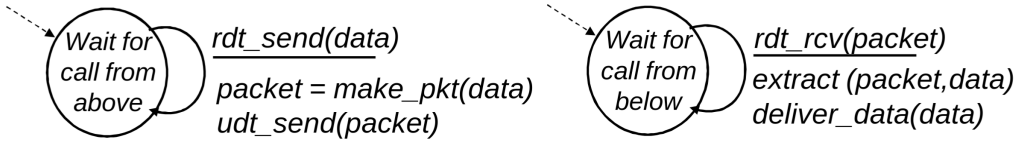
Iniziamo ad iterare sul nostro protocollo *RDT*. Assumiamo come prime ipotesi, largamente semplificate:

- Non ci sono errori di bit;
- Non ci sono perdite di pacchetti. chiarisci chi è frame, chi è datagramma e chi è pacchetto

Prevederemo quindi FSM diverse per *trasmettitore* e *ricevitore*:

- Il trasmettitore invia dati su un canale sottostante;
- Il ricevitore legge i dat dal canale sottostante.

L'FSM in questo caso sarà semplice:



- Il trasmettitore avrà il compito di aspettare la chiamata dall'alto, e una volta arrivata di creare un pacchetto da spedire sulla linea inaffidabile;
- Il ricevitore avrà il compito di aspettare anch'esso la chiamata dall'alto, e una volta ricevuta mettersi in ascolto per il pacchetto spedito dal trasmettitore. Una volta arrivato, dovrà estrarre i dati dal pacchetto e consegnarli al livello superiore.

1.3.4 RDT 2.0

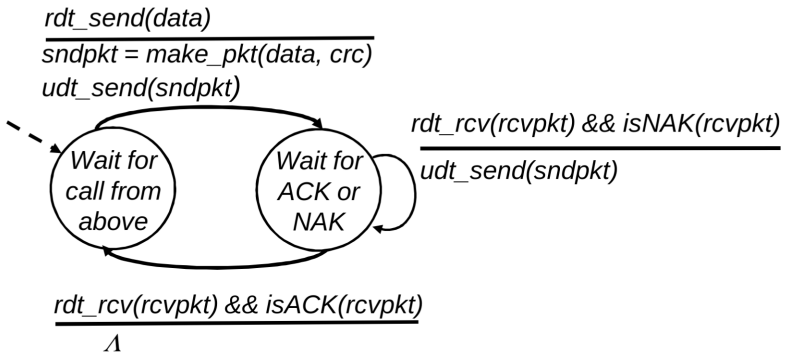
Introduciamo un mezzo di trasmissione inaffidabile che potrebbe avere errori di bit, e presumiamo che tale mezzo implementi un qualche livello sottostante (di tipo datalink datalink maiusc o minusc) che implementa correzione degli errori attraverso CRC o checksum (come visto in 10.1 e 10.2).

Il problema sarà: come riprendersi dagli errori? Abbiamo due modi principali:

- *Acknowledgment* (**ACK**), significa che il ricevitore ha *"capito"*, cioè ha ricevuto il frame correttamente;
- *Negative acknowledgment* (**NAK**), significa che il ricevitor *non ha capito*, cioè non ha ricevuto il frame correttamente.

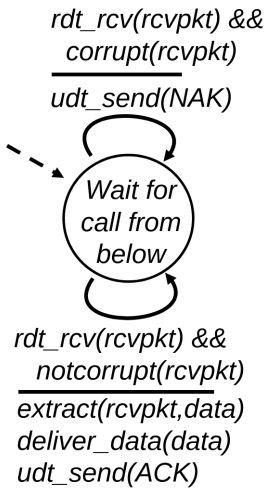
Quando il trasmettitore riceve un **ACK**, sa di poter procedere con il frame successivo, mentre quando riceve un **NAK** sa che deve reinviare il frame corrente.

In questo caso l'FSM del trasmettitore ha il seguente aspetto:



- Il trasmettitore avrà il compito, come prima, di aspettare la chiamata dall'alto e quindi inviare il pacchetto sulla linea inaffidabile;
- A questo punto dovrà aspettare per un ACK o un NAK dal ricevitore:
  - Se riceve un NAK, deve reinviare lo stesso frame;
  - Altrimenti, cioè se riceve un ACK, deve tornare ad aspettare la chiamata dall'alto per il pacchetto successivo.

Lato ricevitore l'FSM sarà invece:



-