1 Lezione del 15-10-25

1.0.1 RDT 2.1

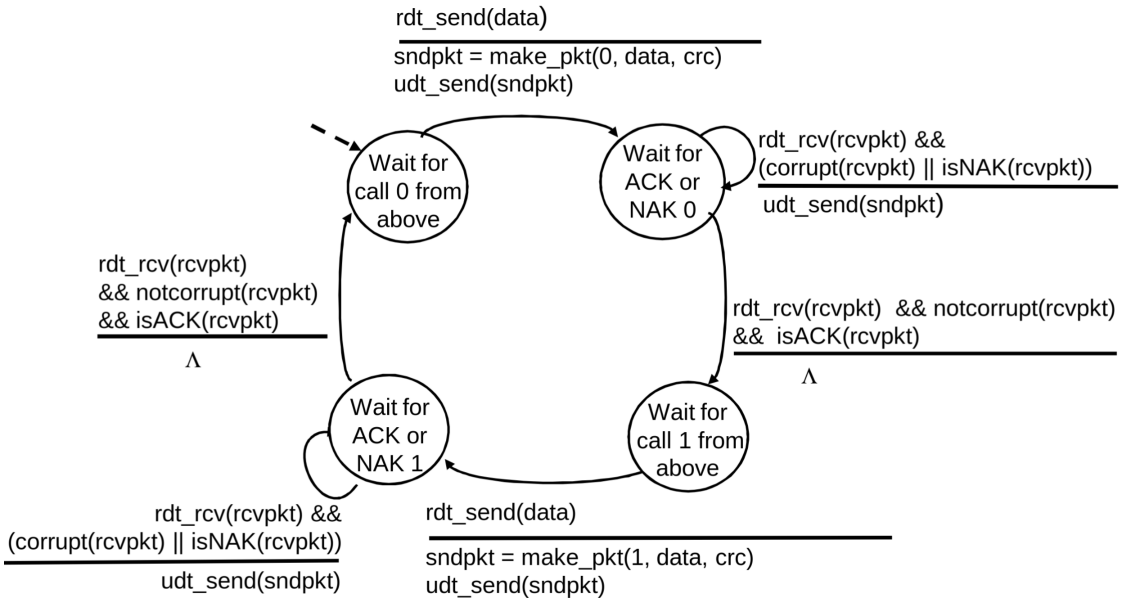
Avevamo introdotto il protocollo di trasferimento fittizio RDT 2.0. Questo era basato su segnali ACK/NAK mandati sul livello datalink dal destinatario al mittente.

Un problema che non avevamo considerato è che questi segnali possono venire corrotti, proprio come i dati veri e propri che volevamo trasmettere. Se il mittente riceve un ACK/NAK corrotto, non può semplicemente ritrasmettere: il destinatario potrebbe ricevere frame duplicati.

Per risolvere questo problema introduciamo fra le *informazioni di controllo* dei frami inviati un cosiddetto **numero di sequenza**. Il mittente allega ad ogni frame un numero di sequenza, e il destinatario implementa un *contatore* del numero di sequenza. I frame con numero di sequenza già visto vengono ignorati.

Abbiamo che per un protocollo *stop-and-wait*, un contatore a 1 bit è più che abbastanza: si inviano due pacchetti alla volta.

In questo caso la macchina a stati del trasmettitore sarà la seguente:

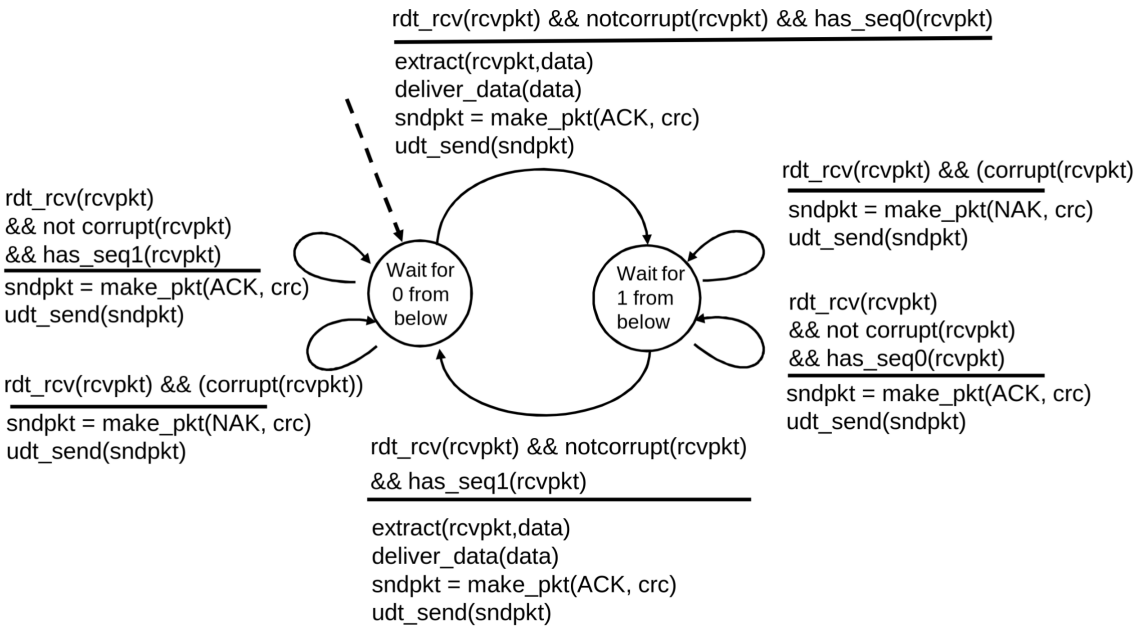


Quello che facciamo è:

- Aspettare la chiamata dall'alto ed inviare il primo pacchetto (con numero di sequenza 0);
- Continuare ad inviare il pacchetto finche non si riceve un ACK non corrotto;
- Aspettare la chiamata dall'alto ed inviare il secondo pacchetto (con numero di sequenza 1);
- Di nuovo, continuare ad inviare il pacchetto finche non si riceve un ACK non corrotto.

Questo andamento si ripete in maniera ciclica.

Il ricevitore potrà a questo punto eseguire il seguente automa:



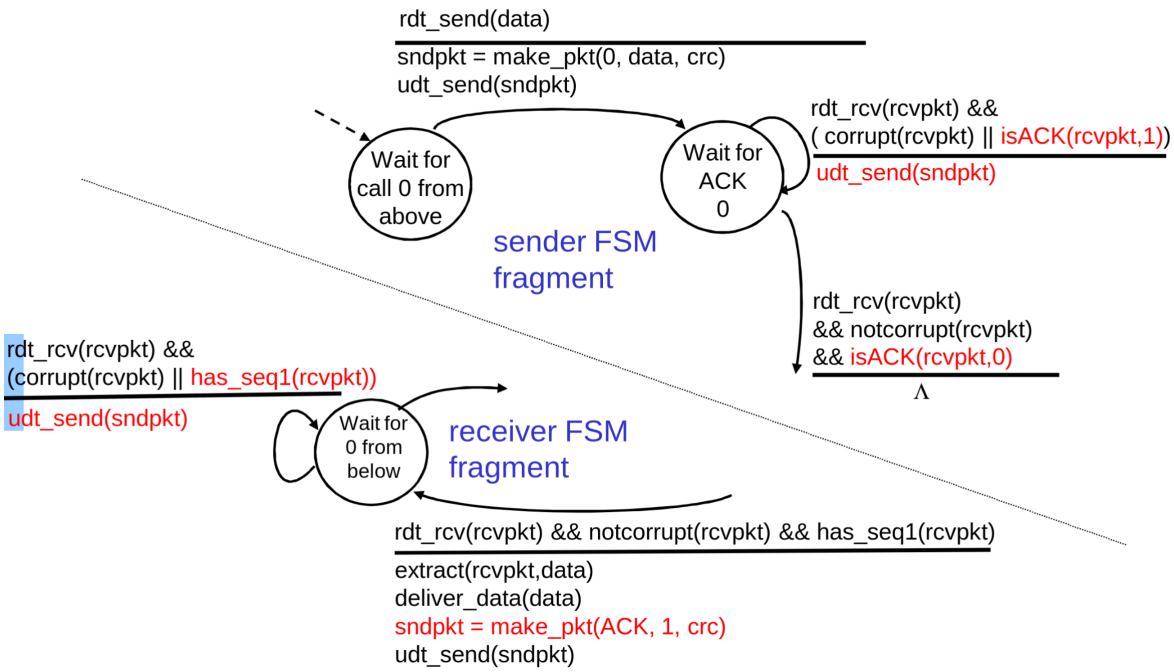
In questo caso il flusso è:

- Si aspetta per il primo pacchetto (dopo la chiamata dall'alto), si invia un pacchetto ACK nel caso questo arrivi integro e con numero di sequenza 0, NAK altrimenti;
- Si aspetta il secondo pacchetto, comportandoci in maniera analoga, ma verificando che il numero di sequenza sia 1.

1.0.2 RDT 2.2

Si può semplificare l'approccio sopra riportato guardando semplicemente agli ACK e ai numeri di sequenza: in questo caso il ricevitore deve solo inviare ACK per gli ultimi pacchetti ottenuti integri. ACK per pacchetti già trasmessi verrano interpretati dal trasmettitore come avevamo interpretato i NAK fino ad ora.

In questo caso le macchine a stati si modificano come segue:



1.1 RDT 3.0

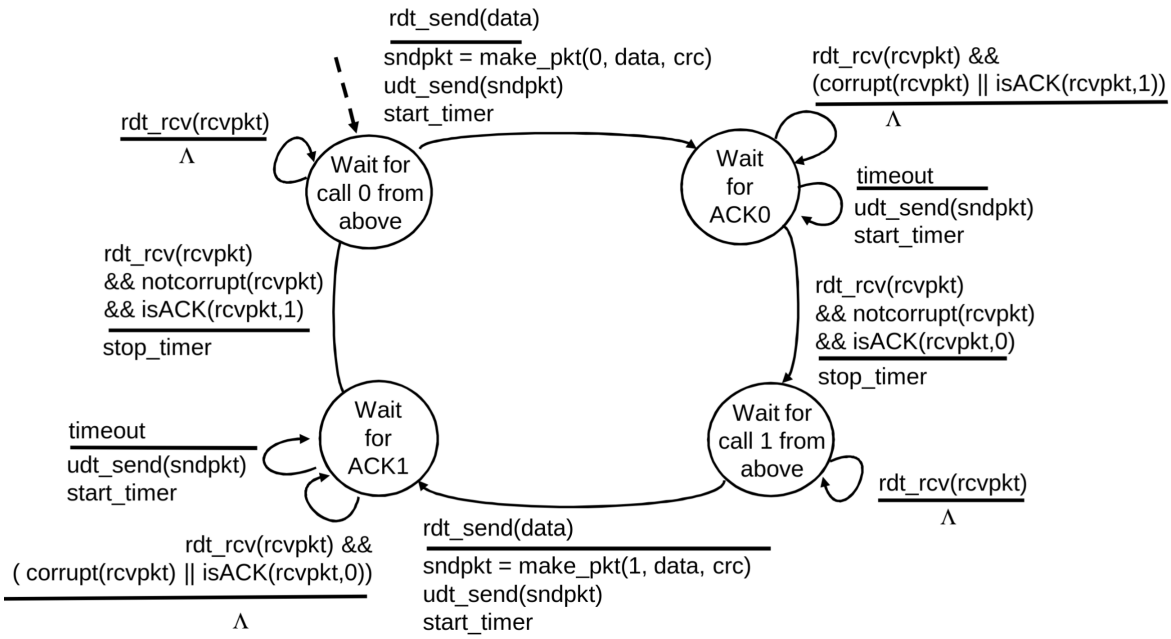
Facciamo una nuova assunzione sui canali: diciamo che il canale sottostante può anche *perdere* (sia data che ACK), mentre prima poteva solo avere errori.

In questo caso potremo sfruttare checksum, numeri di sequenza, ACK e ritrasmissioni, ma non basterà completamente a risolvere il problema.

Un primo approccio potrebbe essere, lato tramsettitore, di aspettare un tempo ragionevole per l'arrivo dell'ACK da parte del ricevitore. *"Ragionevole"* può avere molti significati: assumiamo che il tempo sia tarato sul round-trip-time del link corrente. Dopo che questo tempo passa, quindi, si procede col reinvio del pacchetto:

- Se il pacchetto non era arrivato, il problema è risolto;
- Se il pacchetto era già arrivato e si era perso l'ACK, il ricevitore riceverà pacchetti duplicati: i numeri di sequenza risolvevano già questo prolema. Chiaramente, il ricevitore dovrà specificare il numero di sequenza del pacchetto per cui sta facendo ACK.

La macchina a stati del trasmettitore con questo approccio è:



Ciò che facciamo è analogo a RDT 2.1, con la differenza che:

- Quando inviamo il pacchetto, facciamo partire un timer;
- Se il timer fa timeout durante la fase di attesa per l'ACK, si reinvia il pacchetto al numero di sequenza corrente e si fa ripartire il timer.

Notiamo come si adotta un'approccio *lazy* ai pacchetti ACK corrotti o per pacchetti con numero di sequenza sbagliato (quello che interpretavamo come NAK): in questo caso si ignorano e ci si affida al timeout (che prima o poi arriverà) per effettuare il reinvio.

Lato ricevitore, nulla cambia rispetto a RDT 2.2.

1.1.1 Prestazioni di RDT 3.0

Facciamo alcune note sulle **prestazioni** che riusciamo ad ottenere.

Sia U_{sender} l'**utilizzazione tramsettitore**, cioè la frazione di tempo sul RTT usata dal trasmettitore in fase di invio effettivo del pacchetto.

Se assumiamo di avere un link da 1 Gbps, 15 ms di ritardo di propagazione e di dover trasmettere un pacchetto da 8000 bit. Il tempo per trasmettere un pacchetto sul canale sarà allora circa:

$$D_{\text{trans}} = \frac{L}{R} = \frac{8000}{1000} = 8\,\text{ms}$$

In condizioni ideali, quindi, avremo che dovremo inviare un pacchetto e aspettare un ACK (spedito in tempo trascurabile): questi sono due viaggi sul link (quindi un RTT dato dal due volte il ritardo di propagazione) e il tempo di trasmissione del pacchetto vero e proprio. Si ha quindi:

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.0008}{30 + 0.0008} = 0.00027$$

Cioè l'utilizzazione è terribile: del tempo che impieghiamo a trasmettere un pacchetto, meno dell'0.001% è effettivamente usato per trasmettere i dati che ci interessano! Il problema è chiaramente che passiamo molto tempo ad *aspettare*, quando potremmo avvantaggiarci con l'invio dei pacchetti successivi.

1.2 Pipelining

Il **pipelining** è un meccanismo attraverso il quale il trasmettitore può inviare più pacchetti in massa, senza aver ancora ricevuto l'ACK. Lo poniamo come alternativa allo *stop-and-wait* visto finora.

Riprendendo i conti della scorsa sezione, prendiamo ad esempio di inviare 3 pacchetti invece di 1 per ciclo di ACK. In questo caso si avrà:

$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{0.0024}{30 + 0.0008} = 0.00081$$

che non è molto ma è già di più.

Chiaramente questo comporta delle complicazioni lato ricevitore: dovremmo prevedere la bufferizzazione dei pacchetti in entrata, e ingrandire il contatore del numero di sequenza (2 valori non basteranno più).

Trascurando quanto avviene lato ricevitore, una domanda interessante è *quanti* pacchetti possiamo anticipare prima dell'ACK successivo: dovremmo riempire la finestra del RTT con più tramissioni (ciascuna occupante tempo $\frac{L}{R}$), cioè inviare un numero di pacchetti al massimo pari a:

$$N = \frac{RTT}{L/R}$$

Approcci di questo tipo vengono detti a **sliding window** (*"finestra scorrevole"*), in quanto prendono buffer scorrevoli sul blocco di pacchetti da inviare.

1.2.1 Recupero errori in pipelining

Abbiamo visto che, se vogliamo implementare un protocollo in pipelining, dobbiamo adoperare alcune soluzioni tecniche:

- Buffering al trasmettitore dei pacchetti da inviare;
- Buffering al ricevitore dei pacchetti da ricevere, senza nessuna assicurazione che questi vengano ricevuti in ordine (serve il numero di sequenza);
- Un contatore per il numero di sequenza più grande al ricevitore.

Le specifiche di queste soluzioni vengono definite dal protocollo per il recupero dagli errori che adottiamo. Ne vederemo i 2 principali, che sono **go-back-N** e **selective-repeat**.

1.2.2 Go-back-N

In questo caso il trasmettitore mette fino a N pacchetti senza ACK in pipeline. Al ricevitore viene permesso solo di inviare ACK cumulativi: non effettua ACK se perde uno degli N pacchetti. Il trasmettitore mantiene quindi un timeout per gli ultimi N pacchetti inviati: se il timer scade reinvia tutti gli N pacchetti.

Lato implementativo si ha quindi che il trasmettitore spedisce fino a N pacchetti consecutivi sulla pipeline. Il ricevitore invia da parte sua ACK consecutivi al pacchetto con numero di sequenza più alto ricevuto fino a quel momento (se bufferizzare o

meno i pacchetti che non combaciano col numero di sequenza aspettato è una scelta implementativa).

Quando il primo pacchetto della sequenza spedita viene ricevuto, il trasmettitore sposta in avanti la sua finestra e spedisce il successivo, spostandosi così lungo il blocco di pacchetti.

Se un pacchetto viene perso, il trasmettitore aspetta senza spostare la finestra: prima o poi scatterà il timeout del pacchetto perso e verrà reinviato. Lato ricevitore, possiamo aspettarci che questo passerà lo stesso intervallo temporale a inviare ACK sempre sull'ultimo pacchetto valido (con numero di sequenza consecutivo) ricevuto, e che il trasmettitore abbia ignorato tale ACK in quanto ascolta solo l'ACK più grande ricevuto.

1.2.3 Selective repeat

In questo caso il trasmettitore invia N pacchetti senza ACK in pipeline (come sopra). La differenza è che il ricevitore invia ACK per ogni pacchetto: il trasmettitore deve quindi mantenere un timer per ogni pacchetto inviato, e nel caso questo timer faccia timeout inviare solo il pacchetto corrispondente.

Questo approccio è chiaramente più efficiente: si ritrasmettono solo i pacchetti persi, a costo di un'implementazione più costosa sia lato trasmettitore (dobbiamo mantenere più timer) che ricevitore (dobbiamo inviare ACK separati per ogni pacchetto).

Il caso in cui approcci come il go-back-N vengono comunque usati è quello di dispositivi *constraining* (come quelli che si usano nell'IoT): questi hanno prestazioni meno soddisfacenti e quindi impongono di usare protocolli meno efficienti.