

# Esercizio Timed Automata 17-18 (versione lunga)

L'esercizio consiste nell'analisi via Timed Automata di due protocolli data link: stop&wait e stop&wait per canali rumorosi, come introdotti nel corso di reti e sul libro di testo del Tanenbaum.

Si chiede di modellare i due protocolli e di verificare delle proprietà, da voi definite, che identifichino il corretto funzionamento dei protocolli stessi. Scrivete le proprietà in TCTL, precisando poi se, e come, tali proprietà possono essere verificate in Uppaal (ricordo che Uppaal ha solo un model checker per un sottoinsieme di CTL in cui le proposizioni atomiche possono fare riferimento al valore dei clock degli automi)

# Stop and wait elementare

**MODELLO A.** Si assume che il canale sia perfetto, e quindi nè il messaggio, nè l'ack possono essere persi. Il tempo di trasmissione sul link è variabile all'interno di un intervallo limitato, con poca variabilità (quindi upper e lower bound del tempo di trasmissione sul link sono molto vicini, diciamo non sopra un decimo del tempo di trasmissione).

Si definiscano e provino le proprietà di corretto funzionamento del protocollo, in particolare si provi qual è il tempo minimo e massimo che intercorre dalla spedizione di un messaggio alla ricezione del suo ack

**MODELLO B.** Si modifichi il modello in modo da prevedere la possibilità che il link perda un messaggio o un ack (la perdita equivale sia alla perdita effettiva che al caso di messaggio corrotto, perchè l'algoritmo si comporta in modo analogo). **Non cambiate il protocollo**, modellate semplicemente un link con perdita o corruzione del messaggio. Ovviamente il modello complessivo avrà dei problemi, che potete evidenziare facendo la verifica delle stesse proprietà del modello A. Si mostri, se possibile, un controesempio (che potete riportare in forma riassunta nella relazione)

# Stop and wait con identificazione messaggi e ack

**MODELLO C.** Mantenendo il comportamento del canale previsto per il modello B, si modelli il mittente e il ricevente descritti dall'algoritmo 3 (utilizzo di messaggi e ack numerati, uso della variabile `next_frame_to_send` e dei timer).

Il tempo di trasmissione sul link continua ad essere variabile all'interno di un intervallo limitato, con poca variabilità, come nel caso precedente.

E' possibile identificare un valore di timer che ci permette di affermare con certezza che se il timer scatta il messaggio è stato perso? Questo valore che impatto ha sull'efficienza del canale trasmissivo (vedi lucidi seguenti)?

L'algoritmo prevede l'utilizzo di due timer, uno per ogni possibile numerazione dei messaggi: sono davvero necessari ambedue?

Si definiscano e provino le proprietà di corretto funzionamento del protocollo, in particolare si provi a stabilire qual è il tempo minimo e massimo che intercorre dalla spedizione di un messaggio alla ricezione del suo ack

# Commenti Generali

Provate a mantenere la modularità propria dell' algebra dei processi e di Uppaal, prevedendo un modello per ricevente, uno per trasmittente e uno per il canale.

Valutate con attenzione se canali e locazioni siano urgenti o meno, o se ci siano locazioni committed. Trattandosi di un protocollo di comunicazione, può aver senso far sì che in ogni locazione in cui il sistema "fà" qualcosa (per esempio preparazione del messaggio, lettura dell' ack, e simili) l'automa trascorra del tempo. Nelle locazioni che invece sono di pura sincronizzazione (per esempio per l' acquisizione del canale) il tempo trascorre per effetto dell' eventuale attesa (per esempio nel caso che il canale sia occupato).

# Commenti generali

SUGGERIMENTI.

Tre processi: mittente M, ricevente R e link L.

Uso tre variabili: frame (contiene solo il numero di sequenza), ack (contiene il numero di sequenza del frame di cui è ack), nxt (nextframetosend).

Uso 4 canali: ML, LR, RL, LM, nel seguente modo (oppure un solo canale che rappresenta il link fisico?????):

Frame da M a L

- Il frame di M viene passato ad L via il canale ML
- L fa trascorrere un certo intervallo di tempo
- Il frame diventa disponibile a R via il canale LR

Frame da L a M

- L'ack da R viene passato ad L via il canale RL
- L fa trascorrere un certo intervallo di tempo
- Il frame diventa disponibile a M via il canale LM

M usa due timer X0 e X1, uno da settare quando parte un frame 0 e l'altro quando parte il frame 1

OVVIAAMENTE, il caso dello stop and wait su canale perfetto non richiede timer e variabili

Nei prossimi lucidi un riassunto dei protocolli a cui fa riferimento il testo dell' esercizio (tratti dal testo di reti degli elaboratori di Tannenbaum)

# Protocollo simplex stop-and-wait

Stop-and-wait utilizza il canale in maniera inefficiente

**Utilizzazione  $U$  del canale:** rapporto tra il tempo in cui il link è utilizzato per trasmettere dati ed il tempo totale

**Tempo di lavoro per spedire/ricevere un frame  $T_p$ :** tempo necessario perchè un calcolatore elabori il frame

**Tempo di propagazione  $T_t$ :** tempo necessario perché il frame arrivi al calcolatore destinatario (legato a  $L/R$  e al tempo di propagazione)

# Protocollo simplex stop-and-wait

Supponiamo di dover trasmettere un frame:

- il tempo totale  $T_{TOT}$  per trasmettere un frame e ricevere l'ACK e' pari a  $T_{TOT} = T_p + T_t + T_p + T_t$
- durante questo intervallo, la rete e' utilizzata per trasmettere il frame solo per  $T_t$  unita' di tempo
- l'utilizzazione  $U$  sara' quindi pari a

$$U = \frac{T_t}{2T_p + 2T_t}$$

```

#define MAX_PKT 1024                /* determines packet size in bytes */

typedef enum {false, true} boolean; /* boolean type */
typedef unsigned int seq_nr;        /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet; /* packet definition */
typedef enum {data, ack, nak} frame_kind; /* frame_kind definition */

typedef struct {                    /* frames are transported in this layer */
    frame_kind kind;                /* what kind of a frame is it? */
    seq_nr seq;                     /* sequence number */
    seq_nr ack;                     /* acknowledgement number */
    packet info;                    /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

```



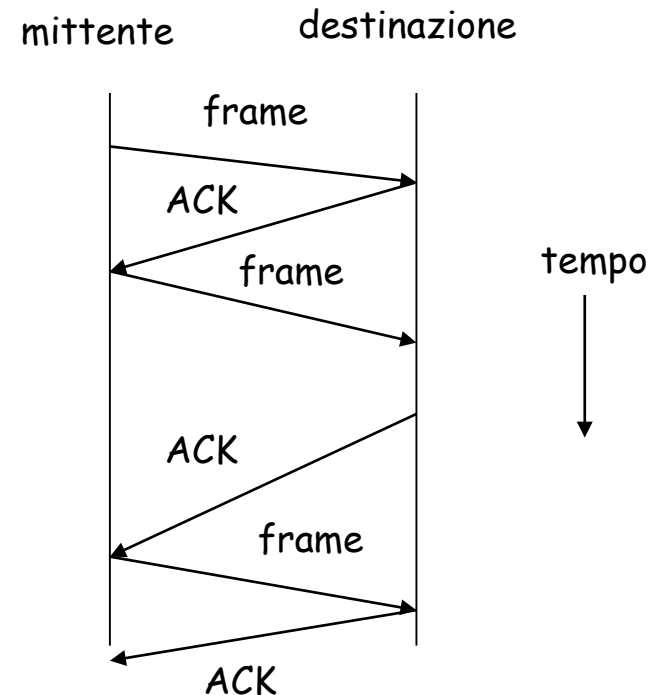
# Protocollo simplex stop-and-wait capacità di bufferizzazione finita

Si suppone che il destinatario **abbia un buffer di capacità finita e pari a uno**, e quindi si ha la necessità di controllare la velocità con la quale il mittente invia i suoi frame. Nel protocollo stop and wait il mittente aspetta un acknowledgment esplicito dal destinatario prima di trasmettere il frame successivo

Vantaggi: non richiede ipotesi sui tempi di lavoro del ricevente

Svantaggi:

- la comunicazione fisica è bidirezionale (si raddoppia il numero di frame inviati)
- Si accoppiano i tempi di mittente e destinatario



# Protocollo simplex stop-and-wait – capacità di bufferizzazione finita

Ipotesi per il corretto funzionamento:

- il canale è perfetto
- network sempre pronto (a dare e ricevere)
- capacità del ricevente finita (necessita di flow-control)

# Protocollo simplex stop-and- wait

```
typedef enum {frame_arrival} event_type;  
#include "protocol.h"
```

```
void sender2(void)  
{  
    frame s;                /* buffer for an outbound frame */  
    packet buffer;         /* buffer for an outbound packet */  
    event_type event;      /* frame_arrival is the only possibility */  
  
    while (true) {  
        from_network_layer(&buffer); /* go get something to send */  
        s.info = buffer;           /* copy it into s for transmission */  
        to_physical_layer(&s);     /* bye-bye little frame */  
        wait_for_event(&event);    /* do not proceed until given the go ahead */  
    }  
}
```

```
void receiver2(void)  
{  
    frame r, s;            /* buffers for frames */  
    event_type event;     /* frame_arrival is the only possibility */  
    while (true) {  
        wait_for_event(&event); /* only possibility is frame_arrival */  
        from_physical_layer(&r); /* go get the inbound frame */  
        to_network_layer(&r.info); /* pass the data to the network layer */  
        to_physical_layer(&s);    /* send a dummy frame to awaken sender */  
    }  
}
```

```
void sender2(void)
{
    frame s;
    packet buffer;
    event_type event;

    while (true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
        wait_for_event(&event);
    }
}
```

```
void receiver2(void)
{
    frame r, s;
    event_type event;
    while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
        to_physical_layer(&s);
    }
}
```

# Protocolli data link elementari – altre funzioni di controllo

```
/* Start the clock running and enable the timeout event. */  
void start_timer(seq_nr k);
```

```
/* Stop the clock and disable the timeout event. */  
void stop_timer(seq_nr k);
```

```
/* Start an auxiliary timer and enable the ack_timeout event. */  
void start_ack_timer(void);
```

```
/* Stop the auxiliary timer and disable the ack_timeout event. */  
void stop_ack_timer(void);
```

```
/* Allow the network layer to cause a network_layer_ready event. */  
void enable_network_layer(void);
```

```
/* Forbid the network layer from causing a network_layer_ready event. */  
void disable_network_layer(void);
```

```
/* Macro inc is expanded in-line: Increment k circularly. */  
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

# Stop-and-wait per canali rumorosi

Supponiamo ora che il canale possa generare errori di trasmissione a causa della presenza di rumore

Prima soluzione: estendiamo il protocollo 2 con un timer

- avviato quando si manda un frame
- se il frame e' danneggiato (o non arriva affatto), il destinatario non risponde
- se arriva un ack, il timer viene fermato
- se il timer scatta, la sorgente reinvia il frame

Problema: e se si perde un acknowledgment?

- il mittente reinvierra' di nuovo il frame, che il destinatario non sara' in grado di riconoscere come duplicato

# Stop-and-wait per canali rumorosi

Per riconoscere i duplicati, e' sufficiente assegnare a ciascun frame un numero di sequenza che lo distingua dagli altri

Nel caso di stop-and-wait l'unica ambiguita' possibile e' tra un frame ed il suo immediato successore

- se si invia il frame  $m+2$ , il frame  $m$  e' stato correttamente riscontrato, altrimenti  $m+1$  (e quindi  $m+2$ ) non sarebbero mai stati trasmessi
- bastano quindi due soli numeri di sequenza (e quindi un solo bit nell'header dei frame)


Protocolli di questo tipo sono detti **Positive Acknowledgement with Retranmission** (PAR) o anche **Automatic Repeat Request** (ARQ)

# Stop-and-wait per canali rumorosi

```
#define MAX_SEQ 1          /* must be 1 for protocol 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
    seq_nr next_frame_to_send;    /* seq number of next outgoing frame */
    frame s;                      /* scratch variable */
    packet buffer;                /* buffer for an outbound packet */
    event_type event;

    next_frame_to_send = 0;       /* initialize outbound sequence numbers */
    from_network_layer(&buffer);  /* fetch first packet */
    while (true) {
        s.info = buffer;          /* construct a frame for transmission */
        s.seq = next_frame_to_send; /* insert sequence number in frame */
        to_physical_layer(&s);    /* send it on its way */
        start_timer(s.seq);       /* if answer takes too long, time out */
        wait_for_event(&event);   /* frame_arrival, cksum_err, timeout */
        if (event == frame_arrival) {
            from_physical_layer(&s); /* get the acknowledgement */
            if (s.ack == next_frame_to_send) {
                stop_timer(s.ack); /* turn the timer off */
                from_network_layer(&buffer); /* get the next one to send */
                inc(next_frame_to_send); /* invert next_frame_to_send */
            }
        }
    }
}
```





# Stop-and-wait per canali rumorosi

```
void receiver3(void)
{
    seq_nr frame_expected;
    frame r, s;
    event_type event;

    frame_expected = 0;
    while (true) {
        wait_for_event(&event);      /* possibilities: frame_arrival, cksum_err */
        if (event == frame_arrival) { /* a valid frame has arrived. */
            from_physical_layer(&r); /* go get the newly arrived frame */
            if (r.seq == frame_expected) { /* this is what we have been waiting for. */
                to_network_layer(&r.info); /* pass the data to the network layer */
                inc(frame_expected); /* next time expect the other sequence nr */
            }
            s.ack = 1 - frame_expected; /* tell which frame is being acked */
            to_physical_layer(&s); /* send acknowledgement */
        }
    }
}
```

```

void receiver3(void)
{
    seq_nr frame_expected;
    frame r, s;
    event_type event;

    frame_expected = 0;
    while (true) {
        wait_for_event(&event); /* pc
        if (event == frame_arrival) { /* a
            from_physical_layer(&r); /* gc
            if (r.seq == frame_expected) { /*
                to_network_layer(&r.info);
                inc(frame_expected); /* ne
            }
            s.ack = 1 - frame_expected; /*
            to_physical_layer(&s); /* se
        }
    }
}

```

```

#define MAX_SEQ 1 /* must l
typedef enum {frame_arrival, cksum_err, time
#include "protocol.h"

```

```

void sender3(void)
{
    seq_nr next_frame_to_send; /* seq nr
    frame s; /* scratc
    packet buffer; /* buffer
    event_type event;

    next_frame_to_send = 0; /* initiali
    from_network_layer(&buffer); /* fetch l
    while (true) {
        s.info = buffer; /* constri
        s.seq = next_frame_to_send; /* insert
        to_physical_layer(&s); /* send i
        start_timer(s.seq); /* if ans
        wait_for_event(&event); /* frame
        if (event == frame_arrival) {
            from_physical_layer(&s); /* get th
            if (s.ack == next_frame_to_send) {
                stop_timer(s.ack); /* turn tt
                from_network_layer(&buffer);
                inc(next_frame_to_send); /*
            }
        }
    }
}

```

```

void receiver3(void)
{
    seq_nr frame_expected;
    frame r, s;
    event_type event;

    frame_expected = 0;
    while (true) {
        wait_for_event(&event); /* pc
        if (event == frame_arrival) { /* a
            from_physical_layer(&r); /* gc
            if (r.seq == frame_expected) { /*
                to_network_layer(&r.info);
                inc(frame_expected); /* ne
            }
            s.ack = 1 - frame_expected; /*
            to_physical_layer(&s); /* se
        }
    }
}

```

```

#define MAX_SEQ 1 /* must l
typedef enum {frame_arrival, cksum_err, time
#include "protocol.h"

```

```

void sender3(void)
{
    seq_nr next_frame_to_send; /* seq nr
    frame s; /* scratc
    packet buffer; /* buffer
    event_type event;

    next_frame_to_send = 0; /* initiali
    from_network_layer(&buffer); /* fetch l
    while (true) {
        s.info = buffer; /* constri
        s.seq = next_frame_to_send; /* insert
        to_physical_layer(&s); /* send i
        start_timer(s.seq); /* if ans
        wait_for_event(&event); /* frame
        if (event == frame_arrival) {
            from_physical_layer(&s); /* get th
            if (s.ack == next_frame_to_send) {
                stop_timer(s.ack); /* turn tt
                from_network_layer(&buffer);
                inc(next_frame_to_send); /*
            }
        }
    }
}

```