SICHERUNG DER ÜBERTRAGUNG

Problemstellung:

Was tun mit verlorenen/verfälschten Frames?

Lernziele:

- Die Teilnehmer sollen die Funktionsweise von ARQ-Protokollen (mit Acknowledgements und Timeouts) erklären können.
- Die Teilnehmer sollen die Funktionsweise von Sliding-Window Protokollen erklären und deren Performance bei verschiedenen Netzwerkverhalten (Bandbreite, RTT, Fehlerrate) abschätzen können.

Inhalt:

- Simplex-Protokolle
- Duplex-Protokoll, Stop-and-Wait
- Sliding-Window-Protokolle

- Problem:
  - Pakete können verloren gehen oder fehlerhaft beim Empfänger ankommen.
  - Ursachen:
    - ◇ Übertragungsfehler auf Leitungen oder Überlastung von Switches/Routern
    - ◇ Überlastung des Empfängers
- Fehlerhafte Pakete können zwar erkannt werden (z.B. mit CRC), allerdings müssen diese dann erneut übertragen werden, da Fehler üblicherweise nur erkannt aber nicht korrigiert werden können.
- Aufgabe der Sicherungsschicht:
  - zuverlässige Zustellung aller abgesendeten Daten
  - Zustellung der abgesendeten Daten in der korrekten Reihenfolge (jede Daten-Einheit genau ein mal)
    - ◇ Design-Frage: Gehört das wirklich hierher?
- Algorithmen der Sicherungsschicht transportieren:
  - Pakete von der Netzwerkschicht als Frames zur Bitübertragungsschicht (Sender-Seite)
  - Frames von der Bitübertragungsschicht als Pakete zur Netzwerkschicht (Empfänger-Seite)
- Im Folgenden werden solche Algorithmen systematisch entwickelt.
  - Annahme: Ein Paket der Netzwerkschicht kann in jeweils einem Frame der Bitübertragungsschicht versendet werden. (keine weitere Fragmentierung)

Grundlegende Datenstrukturen und Routinen

```
#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 f. */
void from_physical_layer(frame *f);

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

/* Start the clock 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);

/* 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
```


Ein uneingeschränktes Simplex-Protokoll

- Annahmen:
  - Übertragung nur in eine Richtung
  - beide Protokollpartner sind immer zum Senden bzw. Empfangen bereit
  - Vernachlässigung der Rechenzeiten für die Abwicklung der Kommunikation
  - keine Übertragungsfehler
  - unbegrenzte Pufferspeicherkapazität
- Folgen:
  - zu übertragender Rahmen besteht nur aus Nutzdaten
  - keine Kontrollinformationen
  - keine Quittungen erforderlich, da keinerlei Fehler auftreten können
  - keine Flußkontrolle notwendig

```
typedef enum {frame_arrival} event_type;

void sender1(void)
{
  frame s;              /* buffer for an outbound frame */
  packet buffer;        /* buffer for an outbound packet */

  while (true) {
        from_network_layer(&buffer);    /* go get something to send */
        s.info = buffer;                /* copy it into s for transmission */
        to_physical_layer(&s);          /* send it on its way */
  }
}

void receiver1(void)
{
  frame r;
  event_type event;     /* filled in by wait, but not used here */

  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 */
  }
}
```

Ein Simplex-Protokoll mit Stop-and-Wait Eigenschaft

- Annahmen:
  - Übertragung nur in eine Richtung
    - ◇ Der Kanal selbst kann jedoch in beide Richtungen übertragen.
  - keine Übertragungsfehler
  - Empfänger hat nur endliche Rechengeschwindigkeit und endlichen Puffer
- Folgen:
  - Sender darf Empfänger nicht "überfluten" $\Longrightarrow$ Flußkontrolle
  - Einführung von Quittungen
- Frage:
  - Auslastung des Kanals? (vgl. "Bandbreite × Latenz")

```
typedef enum {frame_arrival} event_type;
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 */
        from_physical_layer(&s);      /* "remove" frame from phys. layer */
  }
}
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 wake up sender */
  }
}
```


Ein Simplex-Protokoll für einen fehlerbehafteten Kanal

- Annahmen:
  - Übertragung nur in eine Richtung
  - Empfänger hat nur endliche Rechengeschwindigkeit und endlichen Puffer.
  - Empfänger bestätigt erhaltene Frames jeweils mit einer Quittung (ACK).
  - Durch Übertragungsfehler können Frames verloren gehen oder verfälscht werden. (CRC-Error)
    - ◇ Verfälschte Frames werden von der Bitübertragungsschicht erkannt und der Sicherungsschicht gemeldet.
- Folge:
  - Einführung von Timeouts beim Sender
    und ggf. wiederholtes Senden, falls Quittung nicht rechtzeitig eintrifft
  - Schema auch bekannt als:
    - ◇ PAR (Positive Acknowledgement with Retransmission)
    - ◇ ARQ (Automatic Repeat reQuest)
  - Garantiert die zuverlässige Zustellung aller gesendeten Daten.
- Problem:
  - Wird auch die zuverlässige Zustellung in der korrekten Reihenfolge garantiert?
    - ◇ Vergleiche Abb. SIC–1, (c) bzw. (d)

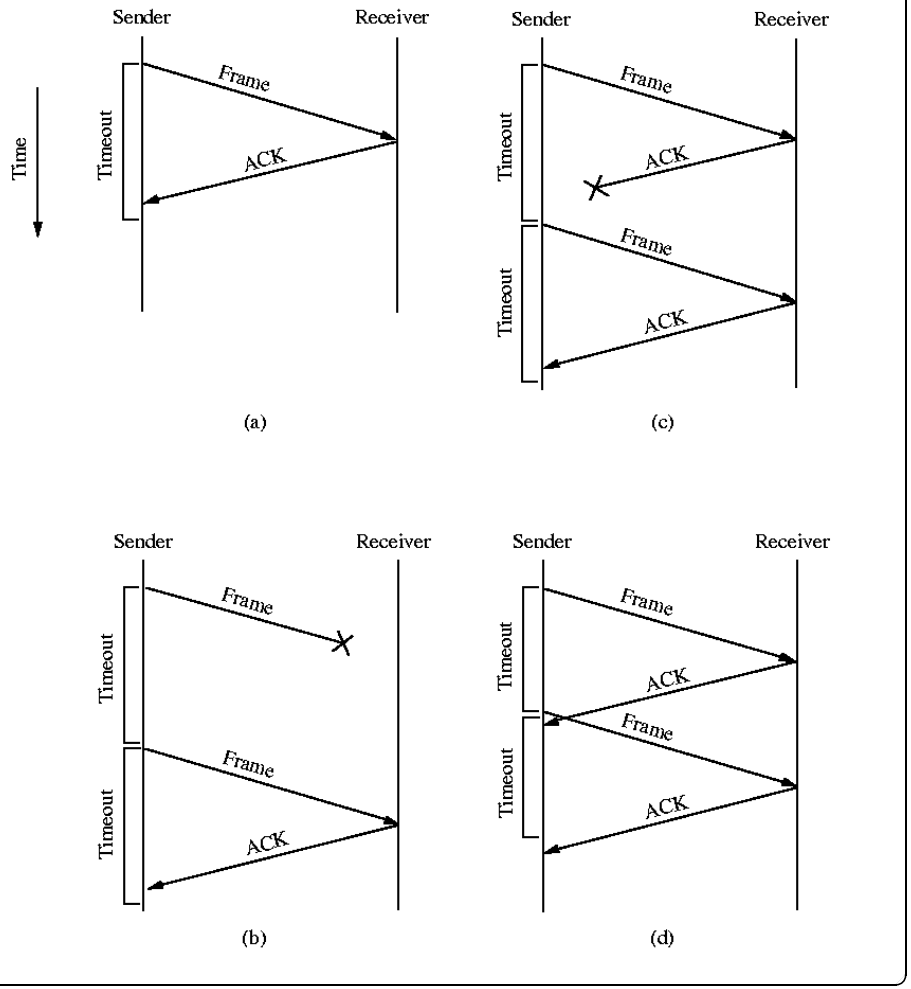

Abb. SIC–1 Positive Acknowledgement with Retransmission

- Lösung:
  - Einführung von Sequenz-Nummern für Frames
    - Bei Stop-and-Wait genügen 2 Sequenz-Nummern
      ("Alternating Bit Protocol")

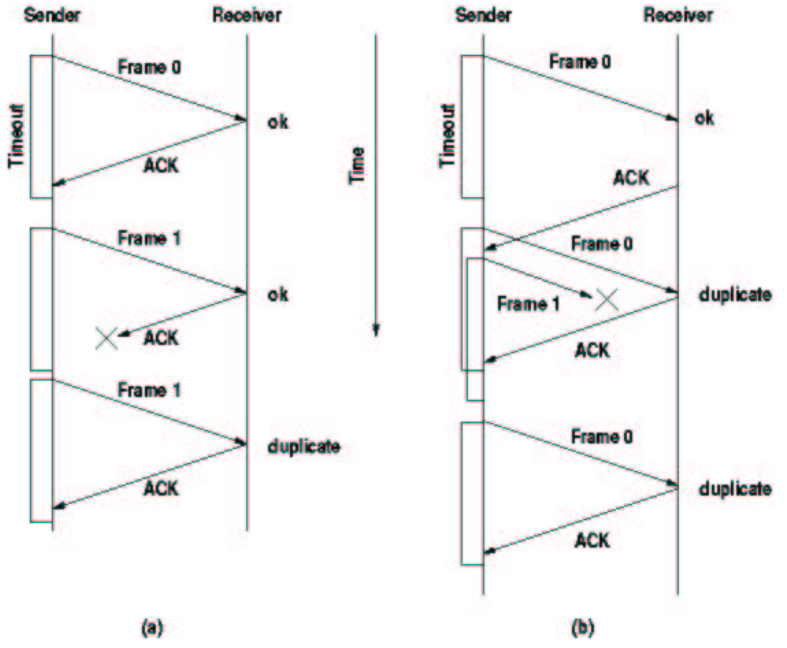

Abb. SIC–2 (a) Quittung geht verloren
(b) Quittung kommt zu spät und nächstes Frame geht verloren

- In Abb. SIC–2(a) wird durch die Sequenz-Nummer der Frames das Duplikat von Frame 1 korrekt erkannt und vom Empfänger verworfen.
- In Abb. SIC–2(b) kommt das ACK für Frame 0 erst nach dem Timeout beim Sender an.
  - Der Sender wiederholt deshalb Frame 0.
  - Das ACK wird vom Sender fälschlicherweise als zur Wiederholung gehörig interpretiert.
  - Der Sender schickt nun deshalb Frame 1 ab.
    (geht verloren)
  - Das ACK zum wiederholten Frame 0 wird fälschlicherweise als zu Frame 1 gehörig interpretiert.
  - Der Sender schickt nun deshalb wieder einen Frame 0 ab.
    (nun mit dem dritten Daten-Paket)
  - Der Empfänger hält dies für eine erneute Wiederholung vom ursprünglichen Frame 0 und verwirft auch diesen Frame.

- Wenn nun das zugehörige ACK korrekt beim Sender ankommt, dann wird dieser erneut einen Frame mit Nummer 1 absenden.
  (nun mit dem vierten Daten-Paket)
- Dieser wird vom Empfänger als Frame mit dem zweiten Daten-Paket akzeptiert. `:-(`
- Dabei sind dann zwei Frames nicht bei der Netzwerkschicht des Empfängers angekommen !!!

- Frage:
  - Was wäre passiert, wenn Frame 1 nicht verloren gegangen wäre?

- Lösung der Probleme:
  - Die ACK Frames enthalten die Sequenz-Nr. des zu bestätigenden Frames.

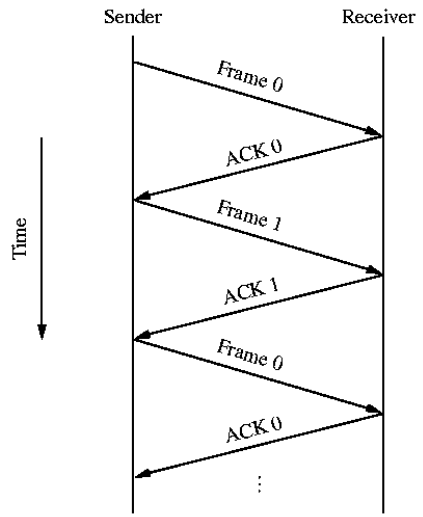

Abb. SIC–3 Alternating-Bit-Protocol mit nummerierten Quittungen

- Protokoll 3:
  - Alternating Bit Protokoll mit Sequenz-Nummern in Quittungen
  - Implementiert als `sender3()` und `receiver3()`

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

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 */
        stop_timer(s.seq);      /* disable now obsoleted timeout event */
        if (event == frame_arrival) {
                from_physical_layer(&s);        /* get the acknowledgement */
                if (s.ack == next_frame_to_send) {
                        from_network_layer(&buffer);    /* get the next one to send */
                        inc(next_frame_to_send);        /* invert 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); /* 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);  /* only the ack field is used */
        }
  }
}
```

Ein Stop-and-Wait Duplex Protokoll

- Annahme
  - auf beiden Seiten liegen ständig zu versendende Pakete vor
- Eigenschaften
  - Sender benötigt genau einen Pufferplatz
  - Sender muß nach Abschicken eines Rahmens auf die Quittung warten
  - Empfänger nimmt Rahmen nur in der sequentiellen Reihenfolge entgegen
  - frame_expected definiert aktuelles Empfangsfenster
  - Acknowledge-Feld eines Rahmens enthält Sequenznummer des letzten empfangenen Rahmens
  - Definition des Sendefensters über next_frame_to_send


```
#define MAX_SEQ 1    /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;

void protocol4 (void)     /* stop-and-wait sliding window */
{
  seq_nr next_frame_to_send;    /* 0 or 1 only */
  seq_nr frame_expected;        /* 0 or 1 only */
  frame r, s;              /* scratch variables */
  packet buffer;           /* current packet being sent */
  event_type event;

  next_frame_to_send = 0;      /* next frame on the outbound stream */
  frame_expected = 0;          /* number of frame arriving frame expected */
  from_network_layer(&buffer); /* fetch a packet from the network layer */
  s.info = buffer;             /* prepare to send the initial frame */
  s.seq = next_frame_to_send;  /* insert sequence number into frame */
  s.ack = 1 - frame_expected;  /* piggybacked ack */
  to_physical_layer(&s);       /* transmit the frame */
  start_timer(s.seq);          /* start the timer running */

  while (true) {
        wait_for_event(&event);    /* could be: frame_arrival, cksum_err, timeout */
        stop_timer(s.seq);         /* stop the now obsolete timer */
        if (event == frame_arrival) { /* a frame has arrived undamaged. */
                from_physical_layer(&r);    /* go get it */

                if (r.seq == frame_expected) {
                      /* Handle inbound frame stream. */
                      to_network_layer(&r.info);  /* pass packet to network layer */
                      inc(frame_expected);    /* invert sequence number expected next */
                }
                if (r.ack == next_frame_to_send) {   /* handle outbound frame stream. */
                      from_network_layer(&buffer); /* fetch new pkt from network layer */
                      inc(next_frame_to_send);     /* invert sender's sequence number */
                }
        }

        s.info = buffer;    /* construct outbound frame */
        s.seq = next_frame_to_send;    /* insert sequence number into it */
        s.ack = 1 - frame_expected;    /* seq number of last received frame */
        to_physical_layer(&s);         /* transmit a frame */
        start_timer(s.seq);            /* start the timer running */
  }
}
```

- Abb. SIC–4(a) zeigt ein Beispiel für einen ordnungsgemäßen Protokollablauf.
- Abb. SIC–4(b) zeigt was passiert, wenn beide Partner gleichzeitig den ersten Rahmen senden.

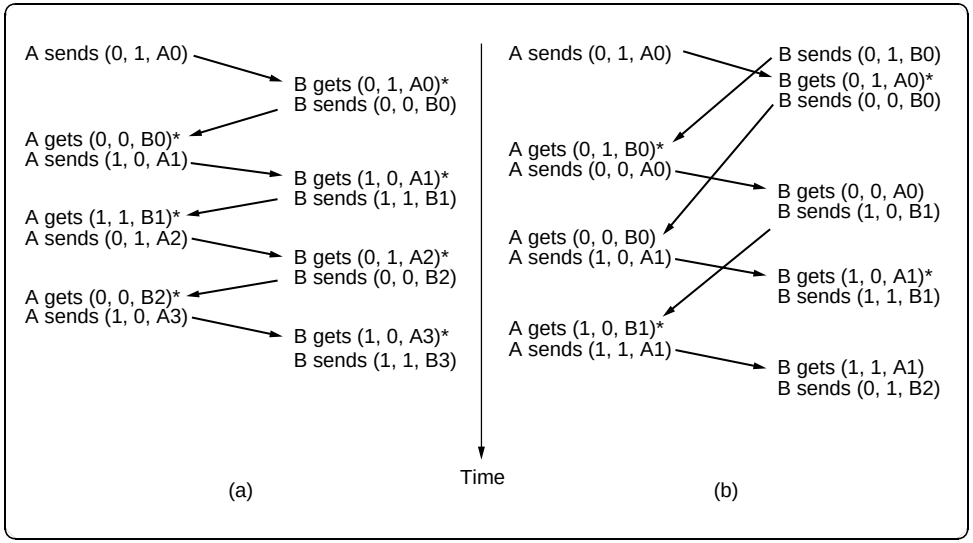

Abb. SIC–4 Szenarien für das Stop-and-Wait Sliding-Window Protokoll
Notation: (seq,ack,packet)
Ein ∗ zeigt an, wann ein Paket von der Netzwerkschicht akzeptiert wird.

- Nur jeder zweite gesendete Rahmen liefert wirklich ein Paket bei der Netzwerkschicht ab.
  - ◇ Sogar dann, wenn keine Übertragungsfehler vorliegen!
- Maßnahme: nur ein Protokollpartner darf die Sequenz
  - ◇ `to_physical_layer(&s);`
  - ◇ `start_timer(s.seq);`

  vor der Endlosschleife ausführen.

  Zerstört aber die Symmetrie des Protokolls.
- Pathologischer Fall ?!?

- Protokollverhalten bei einem vorzeitigen Timer-Ablauf
  - A sendet Rahmen (0,1,A0) an B
  - Timeout-Intervall ist zu kurz ⇒ A wiederholt den Rahmen mehrmals
  - B empfängt ersten Rahmen von A
    - ◇ frame_expected = 1
    - ◇ B sendet einen Rahmen (0,0,B0) an A mit der Quittung für A0
  - alle Duplikate werden von B wegen des falschen seq-Wertes weggeworfen
    - ◇ als Reaktion wiederholt B jeweils seinen gesendeten Rahmen B0
    - ◇ einen neuen Rahmen darf er nicht senden, weil er noch auf die Quittung 0 wartet
  - A erhält den Rahmen B0
    - ◇ A sendet seinen nächsten Rahmen mit der Quittung für B0
    - ◇ alle noch ankommenden Duplikate für B0 wirft A weg

Performance-Analyse: Stop-and Wait

- Beispiel-Link
  - 1.5 Mbps Bandbreite, 45 ms Roundtrip-Time
    - ◇ ⟹ Delay × Bandbreite = 67.5 Kb
  - Annahme: Paket-Größe 1KB
- Durchsatz = Bit per Frame / Time per Frame

  $= 1024 \times 8/0.045 = 182$ Kbps $\approx 12\%$ der Bandbreite
- Zur Verbesserung des Durchsatzes muß in der Größenordnung von Delay × Bandbreite Daten gesendet werden, bevor der Sender auf eine Bestätigung wartet.
- Lösungsansätze
  - längere Nachrichten (wenn denn immer so viele Daten anstehen)
  - mehrere Frames senden, bevor auf die erste Bestätigung gewartet wird

    ⟹ Sliding-Windows

Sliding Window Protokolle

- Eigenschaften der bisher vorgestellten Protokolle
  - Verzögerungszeit einer Übertragung ist wesentlich größer als die Sendezeit für einen Rahmen (= Zeit, die ein Knoten zum Absenden des Rahmens benötigt) ⇒ extrem schlechte Ausnutzung der Leitungsbandbreite
  - Verwendung von Quittungen
    - ◇ die Quittungsübertragung nutzt die Bandbreite noch wesentlich schlechter aus als die Rahmenübertragung
- Idee: Tolerieren mehrerer gleichzeitig nicht quittierter Frames
  - Motto: "keeping the pipe full"

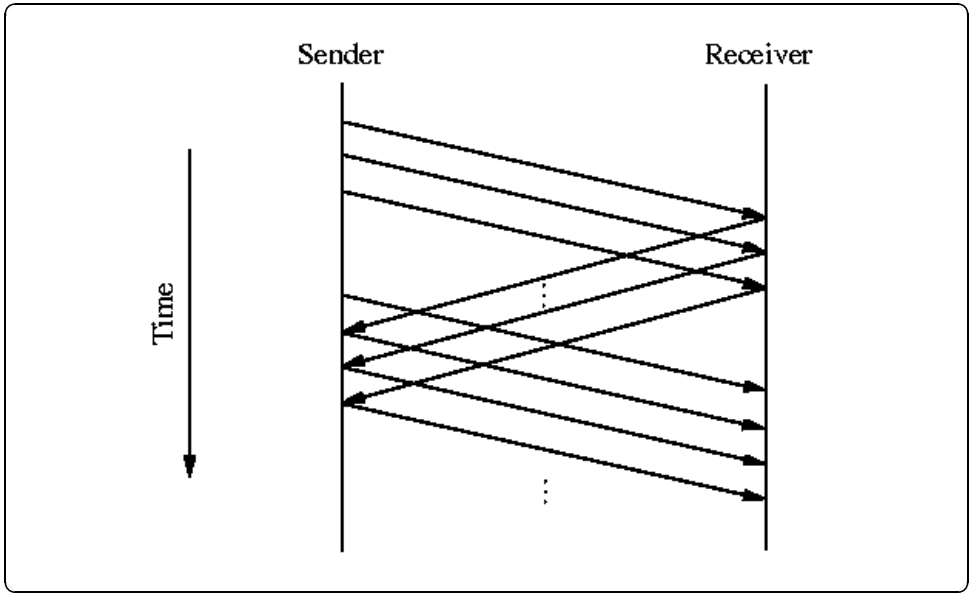

Abb. SIC–5 Überlappung mehrerer Frames beim Sliding-Window Protokoll

- Duplex Protokolle
  - Verwendung einer Leitung für die Datenübertragung in beiden Richtungen
  - Integration der Quittungen in die Nutzdatenrahmen → Piggybacking
    - ◇ weniger Rahmen sind zu übertragen
    - ◇ weniger Interrupts treten beim Empfänger auf

- Problem beim Piggybacking-Verfahren
  - Wann soll statt des Piggybackings eine explizite Quittung als eigener Kontrollrahmen gesendet werden?
    - ◇ Zeitintervall zu lang gewählt ⇒ Timer-Ablauf auf der Senderseite → Rahmenwiederholung
    - ◇ Zeitintervall zu kurz ⇒ Piggybacking wird de facto nicht mehr benutzt
- Struktur eines Sliding Window Protokolls
  - jeder gesendete Rahmen besitzt Sequenznummer
    - ◇ zyklische Wiederverwendung von Sequenznummern
  - Sender verwaltet Sending Window (*Sendefenster*)
    - ◇ entspricht der Rahmenmenge, die gesendet, aber noch nicht quittiert wurde
    - ◇ neue Nachricht kommt von der Netzwerkschicht ⇒ Obergrenze des Sendefensters um 1 verschieben
    - ◇ Quittung kommt an ⇒ Untergrenze des Fensters wird entsprechend angepaßt
      - ▷ mit einer Quittung können mehrere gesendete Rahmen quittiert werden
    - ◇ spätestens wenn das Sendefenster alle verfügbaren Sequenznummern enthält, wird das Senden von Rahmen eingestellt → Flußkontrolle
      - ▷ in Abhängigkeit des gewählten Protokolls kann dies auch schon früher der Fall sein
  - Empfänger verwaltet Receiving Window (*Empfangsfenster*)
    - ◇ entspricht der Rahmenmenge, die er empfangen darf
    - ◇ Empfang eines Rahmens innerhalb des erlaubten Sequenznummernbereichs
      - ▷ Weitergabe des Rahmens an die Netzwerkschicht
      - ▷ Zurücksenden einer Quittung
      - ▷ Rotation des Empfangsfensters um eine Einheit → Anfangsgröße des Empfangsfensters bleibt erhalten
    - ◇ Empfang eines Rahmens außerhalb des erlaubten Sequenznummernbereichs
      - ▷ Wegwerfen des Rahmens
- Sender und Empfänger verwalten zirkuläre Puffer und zirkulär verwendete Sequenznummern.
- Im Folgenden gehen wir zur Vereinfachung davon aus, daß sowohl Puffer als auch Sequnznummern unendlich sind.
- In realen Implementierungen muß alle Arithmetik modulo der Puffergröße bzw. maximaler Sequenznummer durchgeführt werden.
- Sender-Invariante: $\text{LFS} - \text{LAR} \leq \text{SWS}$
- Um ein Frame zu senden muß LFS inkrementiert werden.
  - Solange die Invariante nicht verletzt wird, kann der Sender Frames aussenden.

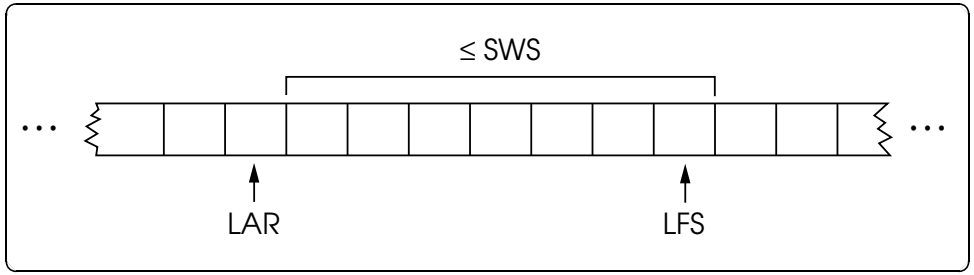

Abb. SIC–6 Sliding Window beim Sender
SWS = Send Window Size
LAR = Last Acknowledgement Received
LFS = Last Frame Sent

- Wenn ein Acknowledgement ankommt, wird LAR entsprechend erhöht, so daß ggf. weitere Frames gesendet werden können.
- Wenn bis zum Timeout für ein bestimmtes Frame kein Acknowledgement ankommt, muß dies erneut gesendet werden.

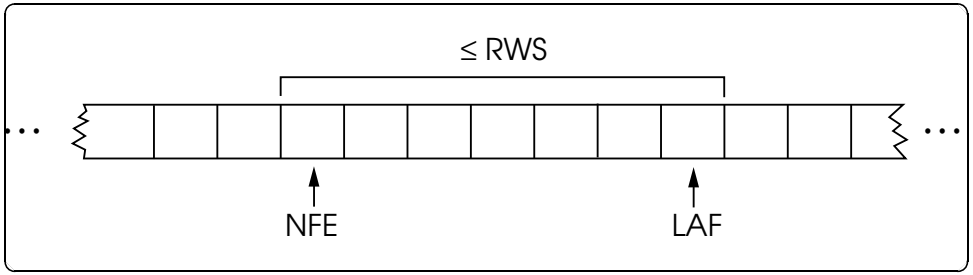

Abb. SIC–7 Sliding Window beim Empfänger
RWS = Receive Window Size
NFE = Next Frame Expected
LAF = Largest Acceptable Frame

- Empfänger-Invariante: LAF − NFE < RWS
- Beim Empfang eines Frames mit SeqNum wird geprüft ob:
  - SeqNum < NFE oder SeqNum > LAF
    $\Longrightarrow$ Frame liegt außerhalb des Fensters und wird ignoriert
  - Wenn NFE ≤ SeqNum ≤ LAF wird das Frame akzeptiert.
  - Kumulatives Acknowledgement:
    - ACK für Frame n bedeutet auch ACK für alles Frames $< n$.
  - Beim Empfang von Frame n wird ein ACK für die höchste Sequenznummer geschickt, bis zu der alle vorigen Frames bereits korrekt empfangen wurden.
    Ausnahme: Frame n erhöht diese Zahl nicht, weil noch ein anderes Frame $< n$ fehlt.
  - Beispiel: bislang sind alle Frames bis einschließlich Nummer 5 empfangen und bestätigt worden. Wenn jetzt die Frames 7 und 8 empfangen werden, werden diese nur im Empfangsfenster gespeichert, aber nicht bestätigt. Wenn danach das fehlende Frame 6 ankommt, wird ein ACK für Frame 8 geschickt.
  - Immer wenn Frames bestätigt werden, werden sie auch an die Netzwerk-Schicht zugestellt und NFE und LAF entsprechend verschoben.

Ein Sliding-Window Protokoll mit Go-Back-n Strategie

- Fehlerbehandlung
  - Verwendung der Go Back n Strategie (RWS=1)
    - ◇ Empfänger wirft alle Rahmen nach einem verlorengegangenen oder verfälschten Rahmen weg
    - ◇ Empfänger sendet keine Quittung mehr
    - ◇ Timer-Ablauf beim Sender für den ältesten Rahmen des Sendefensters
      ⇒ Neuübertragung (aller) Rahmen aus dem Sendefenster

    Ergebnis
    - ◇ auf einer qualitativ schlechten Leitung kann ein großer Teil der Bandbreite für Rahmenwiederholungen verlorengehen
  - Verwendung des Selective Repeat Verfahrens (RWS > 1)
    - ◇ Empfänger speichert alle einem verlorengegangenen oder verfälschten Rahmen folgenden korrekten Rahmen zwischen
    - ◇ Sender wiederholt nur verlorengegangene oder verfälschte Rahmen
- Eigenschaften des Go Back n Protokolls
  - übergeordnete Schicht muß nicht ständig sendebereit sein
    - ◇ Erzeugung eines network_layer_ready Events, um den Wunsch einer Paketübertragung anzuzeigen
  - Verwendung der Prozeduren
    - ◇ enable_network_layer und
    - ◇ disable_network_layer,

    um die network_layer_ready Events zu erlauben oder zu verbieten
    → explizite Einbeziehung der übergeordneten Schicht in die Flußkontrolle
  - Sequenznummernraum: 0 bis max_seq
  - Größe des Sendefensters: max_seq (und nicht max_seq + 1)
    - ◇ Beispiel: max_seq = 7
      - ▷ Sender sendet Rahmen 0 bis 7
      - ▷ Quittung für Rahmen 7 kommt beim Sender an
      - ▷ Sender sendet erneut acht Rahmen
      - ▷ eine weitere Quittung für Rahmen 7 kommt beim Sender an

      Frage:
      - ▷ Sind alle acht Rahmen korrekt angekommen oder alle acht verlorengegangen?
      - ▷ beachte: auch im letzteren Fall schickt der Empfänger die Quittung 7, da er immer die Quittung für den zuletzt korrekt empfangenen Rahmen wiederholt
  - der Sender benötigt max_seq Pufferplätze, um notfalls alle Rahmen des Sendefensters wiederholen zu können
  - die Quittung n quittiert automatisch alle Rahmen des Sendefensters mit Sequenznummern ≤ n
    - ◇ damit werden verlorengegangene Quittungen implizit wiederholt
    - ◇ Sender kann entsprechende Pufferplätze freigeben
  - Blockierung des Senders, wenn die Gegenseite ihrerseits keine Rahmen mehr sendet → Piggybacking-Mechanismus versagt
    - ◇ Zusätzlicher Timeout mit separaten ACK Framess wird benötigt.


```
/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender may transmit up
   to MAX_SEQ frames without waiting for an ack. In addition, unlike the previous
   protocols, the network layer is not assumed to have a new packet all the time.
   Instead, the network layer causes a network_layer_ready event when there is a
   packet to send. */

#define MAX_SEQ 7    /* should be 2^n - 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Return true if (a <=b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return(true);
    else
        return(false);
}
```

```
static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Construct and send a data frame. */
  frame s;    /* scratch variable */

  s.info = buffer[frame_nr];    /* insert packet into frame */
  s.seq = frame_nr;             /* insert sequence number into frame */
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);    /* piggyback ack */
  to_physical_layer(&s);        /* transmit the frame */
  start_timer(frame_nr);        /* start the timer running */
}

void protocol5(void)
{
  seq_nr next_frame_to_send;    /* MAX_SEQ > 1; used for outbound stream */
  seq_nr ack_expected;          /* oldest frame as yet unacknowledged */
  seq_nr frame_expected;        /* next frame expected on inbound stream */
  frame r;                      /* scratch variable */
  packet buffer[MAX_SEQ+1];     /* buffers for the outbound stream */
  seq_nr nbuffered;             /* # output buffers currently in use */
  seq_nr i;                     /* used to index into the buffer array */
  event_type event;

  enable_network_layer();       /* allow network_layer_ready events */
  ack_expected = 0;             /* next ack expected inbound */
  next_frame_to_send = 0;       /* next frame going out */
  frame_expected = 0;           /* number of frame expected inbound */
  nbuffered = 0;                /* initially no packets are buffered */
```



```
  while (true) {                /* protocol5(), continued */
    wait_for_event(&event);     /* four possibilities: see event_type above */

    switch(event) {
      case network_layer_ready:    /* the network layer has a packet to send */
           /* Accept, save, and transmit a new frame. */
           from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
           nbuffered = nbuffered + 1;    /* expand the sender's window */
           send_data(next_frame_to_send, frame_expected, buffer); /* transmit frame */
           inc(next_frame_to_send);    /* advance sender's upper window edge */
           break;

      case frame_arrival:    /* a data or control frame has arrived */
           from_physical_layer(&r);    /* get incoming frame from physical layer */

           if (r.seq == frame_expected) {
              /* Frames are accepted only in order. */
              to_network_layer(&r.info);    /* pass packet to network layer */
              inc(frame_expected);    /* advance lower edge of receiver's window */
           }

           /* Ack n implies n - 1, n - 2, etc.  Check for this. */
           while (between(ack_expected, r.ack, next_frame_to_send)) {
              /* Handle piggybacked ack. */
              nbuffered = nbuffered - 1;    /* one frame fewer buffered */
              stop_timer(ack_expected);    /* frame arrived intact; stop timer */
              inc(ack_expected);    /* contract sender's window */
           }
           break;

      case cksum_err: ;    /* just ignore bad frames */
           break;

      case timeout:    /* trouble; retransmit all outstanding frames */
           next_frame_to_send = ack_expected;    /* start retransmitting here */
           for (i = 1; i <= nbuffered; i++) {
              send_data(next_frame_to_send, frame_expected, buffer);  /* resend 1 frame */
              inc(next_frame_to_send);    /* prepare to send the next one */
           }
    }
    if (nbuffered < MAX_SEQ)       enable_network_layer();
    else                           disable_network_layer();
  }
}
```

Selective Repeat

- Go-Back-n verbessert der Durchsatz im Vergleich zu Stop-and-Wait, solange Übertragungsfehler selten sind.
- Bei häufigen Fehlern bietet sich an, das Empfangsfenster zu vergrößern.
  - Go-Back-n $\Longrightarrow$ Selective-Repeat

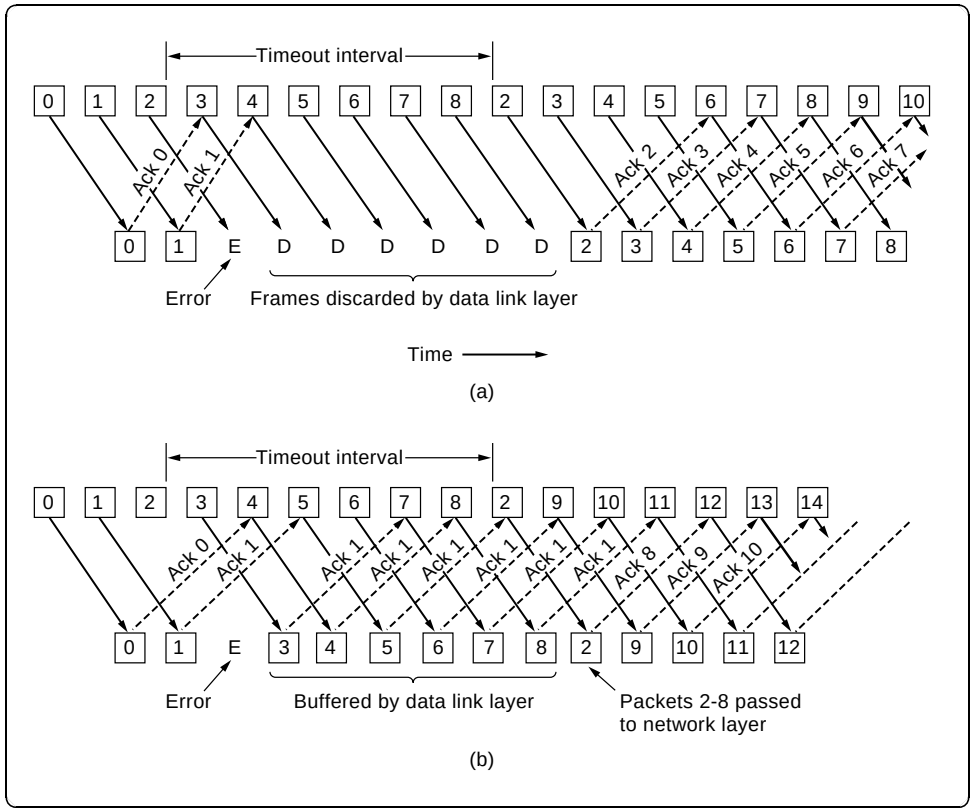

Abb. SIC–8 (a) Auswirkung eines Fehlers mit Go-Back-n
(b) Auswirkung eines Fehlers mit Selective-Repeat

Endliche Sequenznummern und Sendefenstergröße

- Sequenznummern werden in endlichen Header-Einträgen übertragen. Somit werden irgendwann Sequenznummern erneut benötigt.
- Beispiel: 3-bit Sequenznummer erlaubt $0 \ldots 7$.
- Problem: Wie kann der Empfänger die Frames $0 \ldots 7$ der ersten Runde von den Frames $0 \ldots 7$ der zweiten Runde unterscheiden?
- Dies hängt direkt mit der Größe der Sende- und Empfangsfenster zusammen.
- Reicht SWS $\leq$ MaxSeqNum -1 immer aus?
  - Bei RWS $= 1$ schon, bei SWS $=$ RWS nicht
- Beispiel: MaxSeqNum $= 8$, SWS $=$ RWS $= 7$
  - Frames $0 \ldots 6$ werden gesendet und die ACK's gehen verloren.
  - Der Empfänger erwartet 7,$0 \ldots 5$.
  - Der Sender hat ein Timeout und sendet $0 \ldots 6$.
  - Davon interpretiert derEmpfänger $0 \ldots 5$ als Frames der zweiten Runde.

    $\rightarrow$ Fehler!
- Bei SWS $=$ RWS gilt: SWS $\leq$ (MaxSeqNum $+1$)$/2$
  - Intuitiv alterniert das Protokoll zwischen zwei Hälften der Sequenznummern, allerdings nicht schlagartig sondern graduell.
- In der Praxis bedeutet dies daß der Datentyp für die Sequenznummern die maximal mögliche Größe von SWS und RWS begrenzt.

Frame Order and Flow Control

- Das Sliding Windows Protokoll erfüllt implizit drei Funktionen:
  - Sicherung der Wiederholung verlorener Frames.
  - Zustellung der Frames beim Empfänger in der korrekten Reihenfolge.
  - Flußkontrolle, der Empfänger bremst den Sender.
- Unter dem Aspekt "Separation of Concerns" ist es fraglich ob eine solche Kombination sinnvoll ist.
  - Z.B. wird vorausgesetzt daß eine einzige Verbindung immer genügend Frames senden will, um wirklich die verfügbare Kapazität auszunutzen.
  - Kapazitätsauslastung ist problematisch bei verlorenen Frames, wenn nachfolgende Frames den Empfangspuffer blockieren. (frame order)
- Alternativ könnten auch mehrere logische Kanäle über eine Leitung gemultiplext werden.
  - Aufgabe der Reihenfolge zwischen den Kanälen
  - Dadurch blockiert ein verlorener Frame kaum noch das Sendefenster.
  - Implementiert im Arpanet als "Concurrent Logical Channels"