Transporte:
Control de Errores(S&W)

2018

UNIVERSIDAD
NACIONAL
DE LA PLATA

Contenidos

Contenidos

Contenidos

Contenidos

Contenidos

Control de Errores

- Se requiere un mecanismo de control sobre un canal no confiable.
- Se realiza con ARQ: Automatic Repeat reQuest/Automatic Repeat Query.
- Utiliza confirmaciones para validar que los datos se recibieron OK.
- Estudiar que otros mecanismos se necesitan para agregar confiabilidad.

Notas: Análisis basado en slides "Supplements: Powerpoint Slides Computer Networking: A Top-Down Approach 6th ed. J.F. Kurose and K.W. Ross", aunque no utiliza exactamente las mismas funciones y llamadas.
Los algoritmos están escritos en pseudo-código a la "C" y solo están como referencia, pueden contener errores ya que no fueron compilados ni testeados.

Este texto no explica como funciona TCP, sino es un enfoque simplificado y progresivo para entender los problemas que TCP debe intentar solucionar.

Control de Errores, Implementación

- Para simplificar, se supone solo un emisor y un receptor, solo se transmite en un sentido datos y no se requiere elegir origen ni destino.
- Interfaz/API pública de la capa que ofrece los servicios confiables:

```
rdt_rcv(byte_t * data);  // asociada a la func. (non-block/blo
                         // deliver_data(byte_t *data);

rdt_snd(byte_t *data);   // asociada a la func. (block)
                         // rdt_send(byte_t *data);
```

- Interfaz/API pública de la capa inferior no confiable:

```
int udt_send(packet_t  *pkt); // Non-block

int udt_recv(packet_t  *pkt); // Block
```

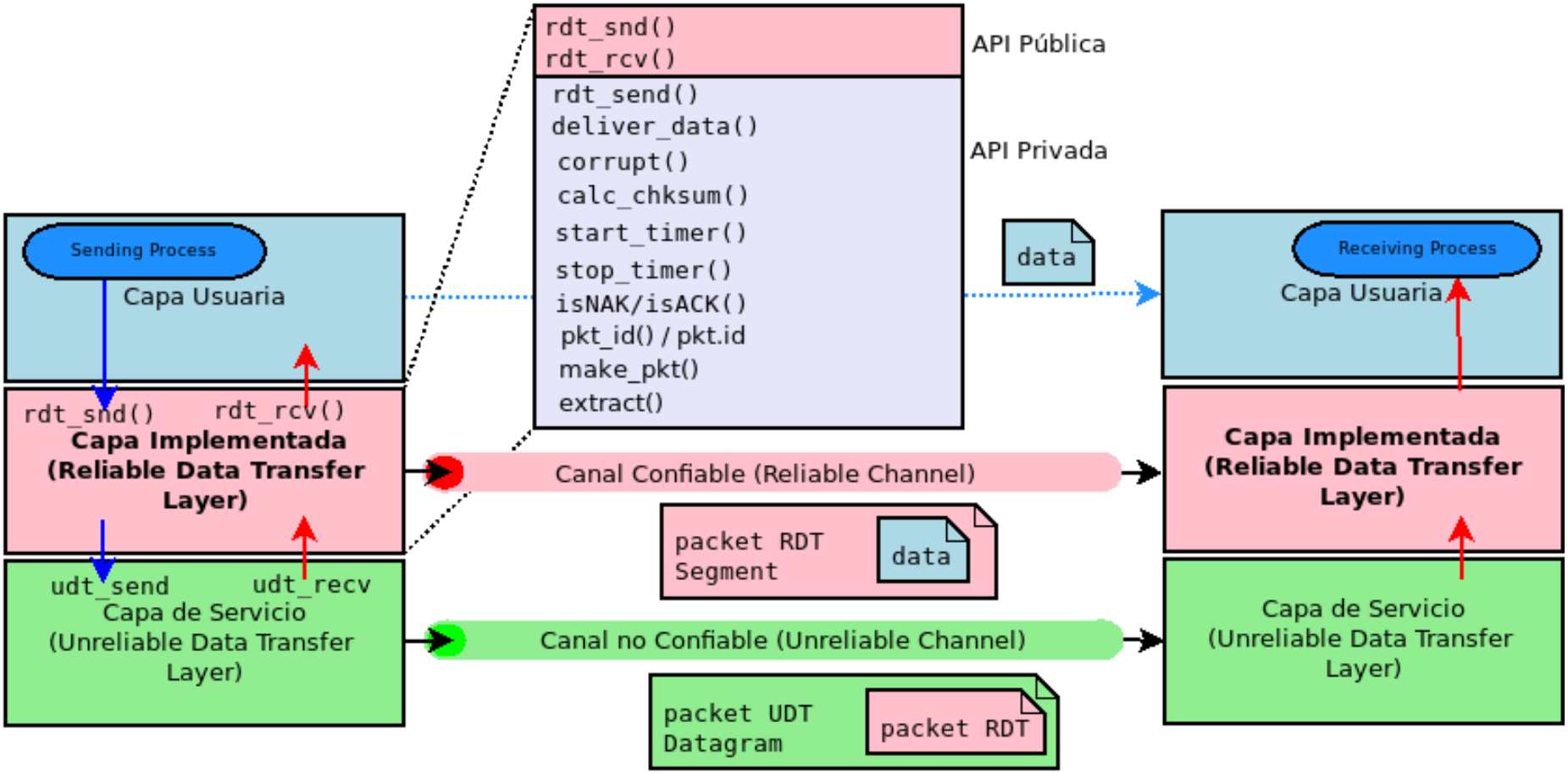
Control de Errores, Procesos

- Se implementa con 2 procesos:
  - **Sender**, para procesar los requerimientos de enviar datos de la capa superior mediante la llamada `rdt_snd()`.
  - **Receiver**, para procesar los requerimientos de recibir datos de la capa superior mediante la llamada `rdt_rcv()`.
- El **Sender** invoca a `rdt_send()` y se bloquea hasta que la capa superior lo llamé mediante `rdt_snd()` dejando datos. En ese momento los datos de la capa superior se copian al buffer/espacio de la capa que implementa el transporte confiable y se desbloquea **Sender**.
- El **Sender** para pasar los datos a la capa inferior y enviarlos invoca a `udt_send()`, se copian los datos y se desbloquea. Asincrónico.

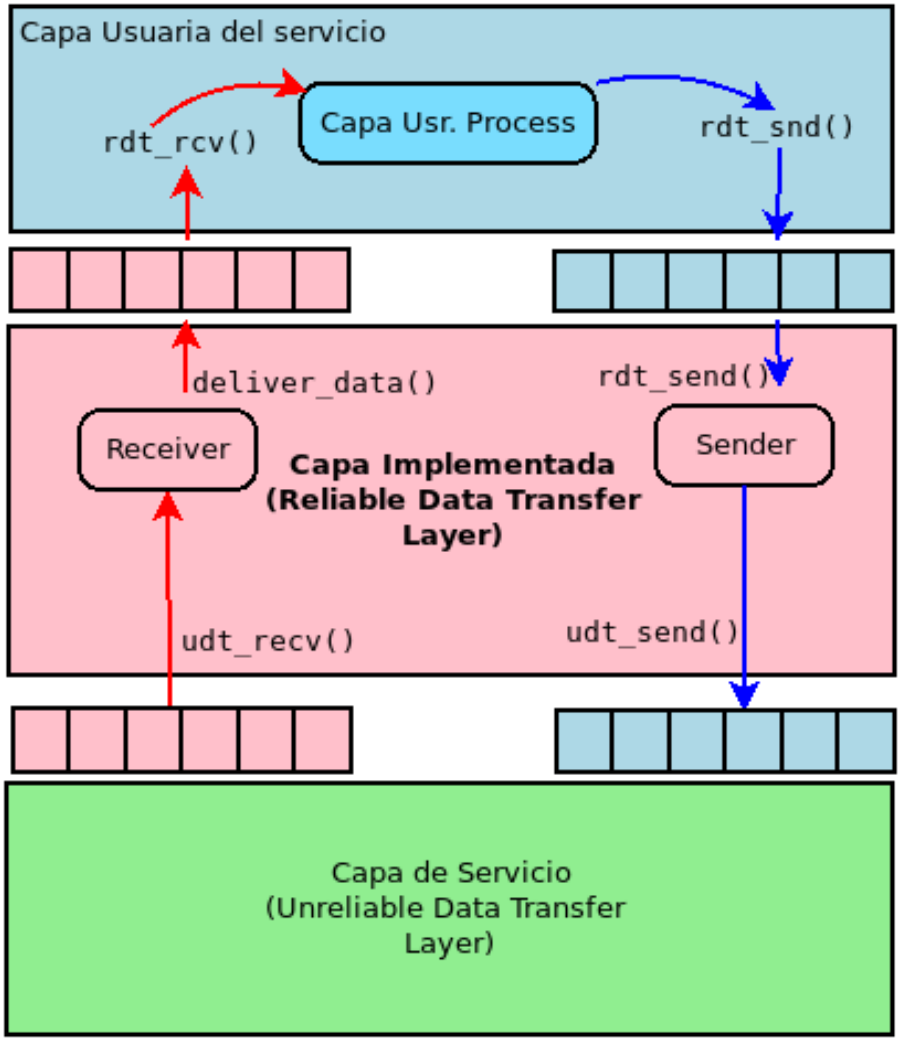
Control de Errores, Procesos (Cont.)

- El **Receiver** invoca a `udt_recv()` y se bloquea hasta que la capa inferior tenga paquetes en el buffer y los pueda entregar.
- El **Receiver** una vez que proceso los datos recibidos los pasa a la capa superior mediante `deliver_data()`. En ese momento se copia al espacio de buffer de la capa superior y el proceso **Receiver** puede seguir su curso sin bloquearse. La capa superior los puede recibe de forma asincrónica mediante `rdt_rcv()` si tiene buffering, sino será bloqueante.
- Se considera que el envío desde el transporte a implementar a al capa inferior no es bloqueante (Buffer infinito).

Control de Errores, API



Control de Errores, API, Comunicación local



Canal Confiable/Errores Posibles

- No se pierden datos.
- No se duplican.
- No se desordenan.
- No se corrompen.
- No existe delay mayor de 1 RTT.

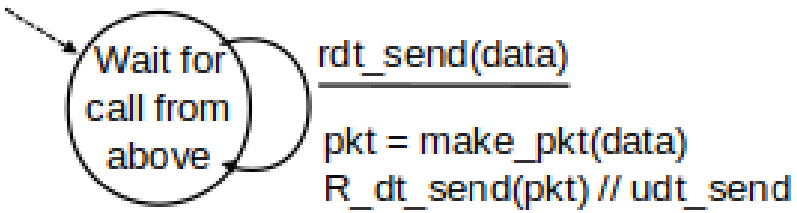
En conclusión, no hay errores, No hay problemas. Para diferenciar que estamos en presencia de un canal confiable las operaciones utilizada sobre la capa de servicio son llamadas:

```
int R_dt_send(packet_t  *pkt); // en lugar de udt_send()
int R_dt_recv(packet_t  *pkt); // en lugar de udt_recv()
```

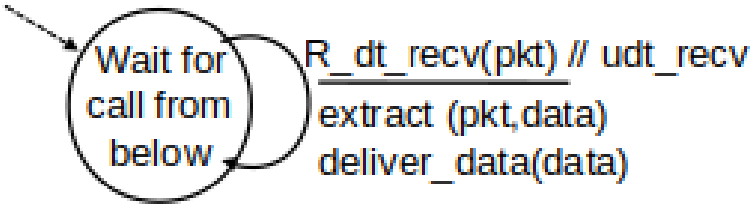
```
void sender1()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result = OK;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {

O0: // Emisor recibe datos de capa superior
    // y arma PDU desde data
    result = rdt_send(data);

    // Arma paquete
    pkt = make_pkt(data)
    //pkt->payload = data;

O1: // Emisor envía datos
    R_dt_send(pkt); // udt_send(pkt);

O2: // Emisor vuelve a 0
    // goto O0 -- esta en el while

   }
}
```

```
void receiver1()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result = OK;

  // Mientras no se termine la com.
  while (result != EOT)
   {

O0: // Receptor recibe datos de capa inf.

    R_dt_recv(pkt); // udt_recv(pkt);

O1: // Receptor entrega datos a
    // capa superior data = pkt->payload
    extract(pkt, data);
    result = deliver_data(data);

O2: // Receptor vuelve a 0
    // goto O0 -- esta en el while

   }
}
```

Algoritmo Base - FSM Canal Confiable



sender



receiver

Análisis de Algoritmo Base

- Un solo loop por cada proceso.
- Se supone que los datos siempre llegan y en orden.
- No hay sincronismo entre emisor y receptor, buffer ilimitado.
- Escenario irreal. Qué sucede si se cambia ?
- Se reemplaza:

`R_dt_send(data);` por `udt_send(data);`
`R_dt_recv(data);` por `udt_recv(data);`

El algoritmo deja de ser seguro, los errores del canal aparecen en la transmisión y la solución no funciona.

Canal con algunos errores, corrompen Datos

- No se pierden datos.
- No se duplican.
- No se desordenan.
- Sí se corrompen datos, se chequean con Checksum/CRC o mecanismo similar, se avisa cuando no se reciben de forma correcta.
- Utilización de confirmaciones ACK/NAK.
- No se corrompen las confirmaciones ACK/NAK.
- No hay delay mayor de 1 RTT.

Algoritmo Retrans (Detección de Errores)

- Se debe cambiar la estructura de la PDU de la capa que hace el control de errores.
- Se agrega campo para checksum/CRC y se agrega un flag para indicar si es paquete de datos o de confirmación.
- Se implementa cálculo de código de detección de errores en `Sender2:01`
- Se implementa chequeo de errores y confirmación `Receive2:01`.
- Se implementa chequeo de feedback `Sender2:03`.
- Si los datos llegan corruptos no se pasan a la capa superior.

Estructura de Datos para Algoritmo Retrans

- Estructura tentativa del paquete/segmento.

```
typedef enum {ack, nak, dat} type_t;

typedef struct pk {
   // Header
   int       len;
   type_t     ack;   // ACK | NAK | DATA
   ck_t       chksum;
   // Data
   byte_t   payload[...];
} packet_t;
```

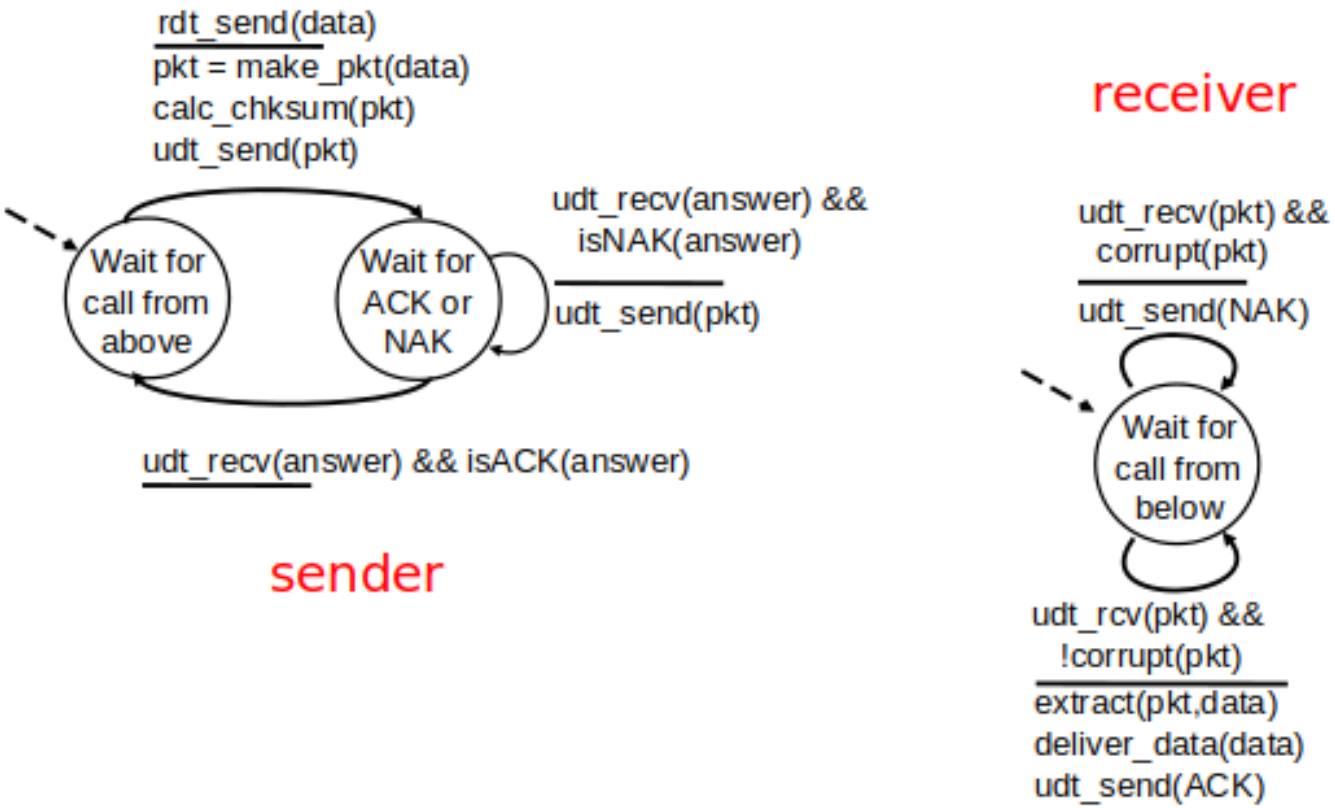
```
void sender2()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result  = OK;
  packet_t  *answer = NULL;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {

O0: // Emisor recibe datos de capa
    // superior y arma PDU desde data
    result = rdt_send(data);

    // Arma paquete
    pkt = make_pkt(data)

O1: //pkt->header = ...
    calc_chksum(pkt);

    // answer == NAK
    while  (isNAK(answer))
    {
...
```

```
void receiver2()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result = OK;
  int       ok     = 0;

  // Mientras no se termine la com.
  while (result != EOT)
   {

      while (!ok)
       {
O0:      // Receptor recibe datos
         udt_recv(pkt);

O1:      // Receptor chequea datos
         // y confirma con ACK/NAK
         if (!corrupt(pkt))
            {
              udt_send(ACK);
              ok = 1;
            }
         else
            {
              udt_send(NAK);
              ok = 0;
            }
       }
...
```

```
...
    while (isNAK(answer))
    {
O2:     // Emisor envía datos:
        udt_send(pkt);

O3:     // Emisor espera respuesta
        // answer ::= ACK | NAK
        udt_recv(answer);

    } // Emisor se queda
      // retransmitendo el mismo dato
      // isNAK(answer)

O4: // Emisor vuelve a 0
    // goto O0 -- esta en el while

  } // isACK(answer)
}
```

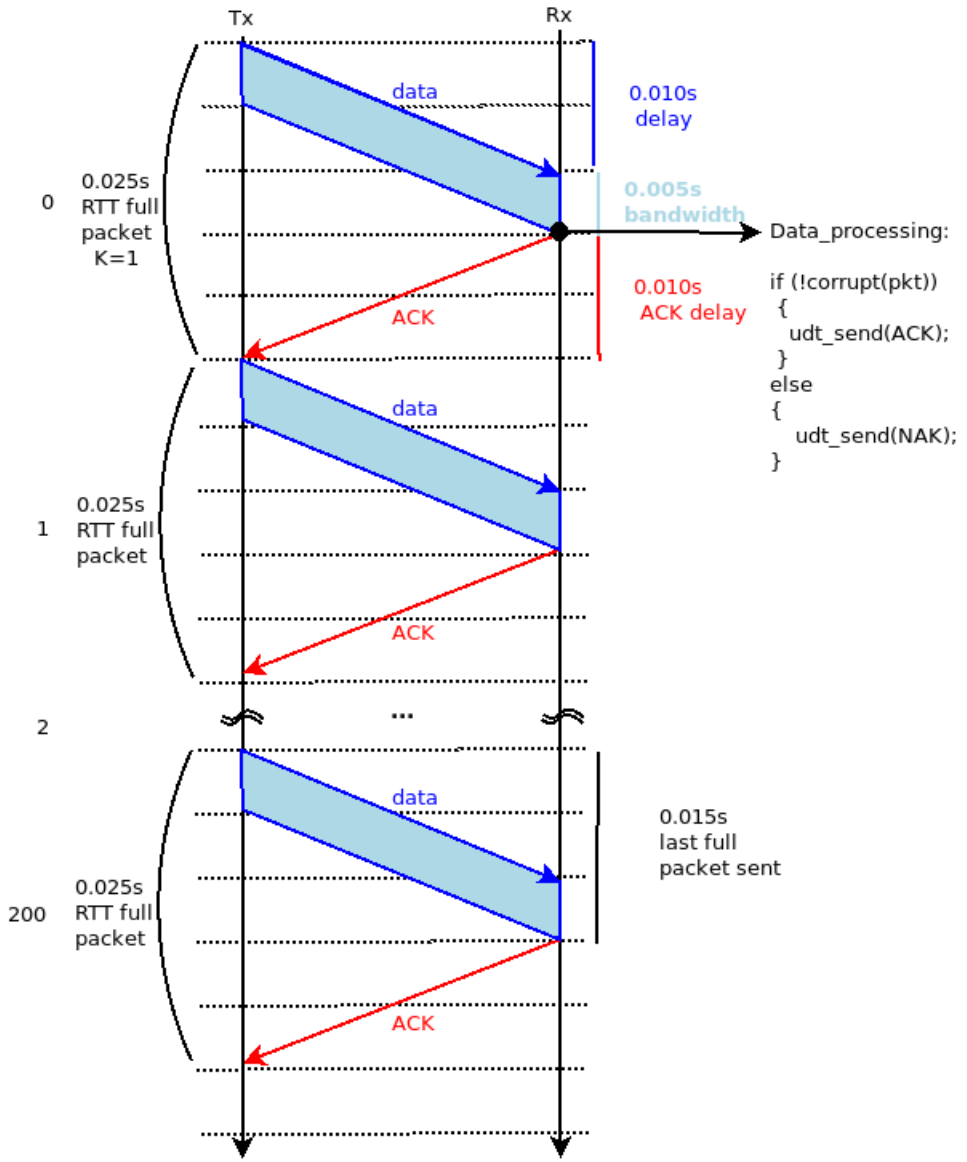
```
...
O2:   // Receptor entrega datos a
      // capa superior data=pkt->payload
      extract(pkt, data);
      result = deliver_data(data);

O3: // Receptor vuelve a 0
    // goto O0 -- esta en el while
  }
}
```

FSM Canal Errores Algoritmo Retrans, data chksum



Análisis de Algoritmo Retrans

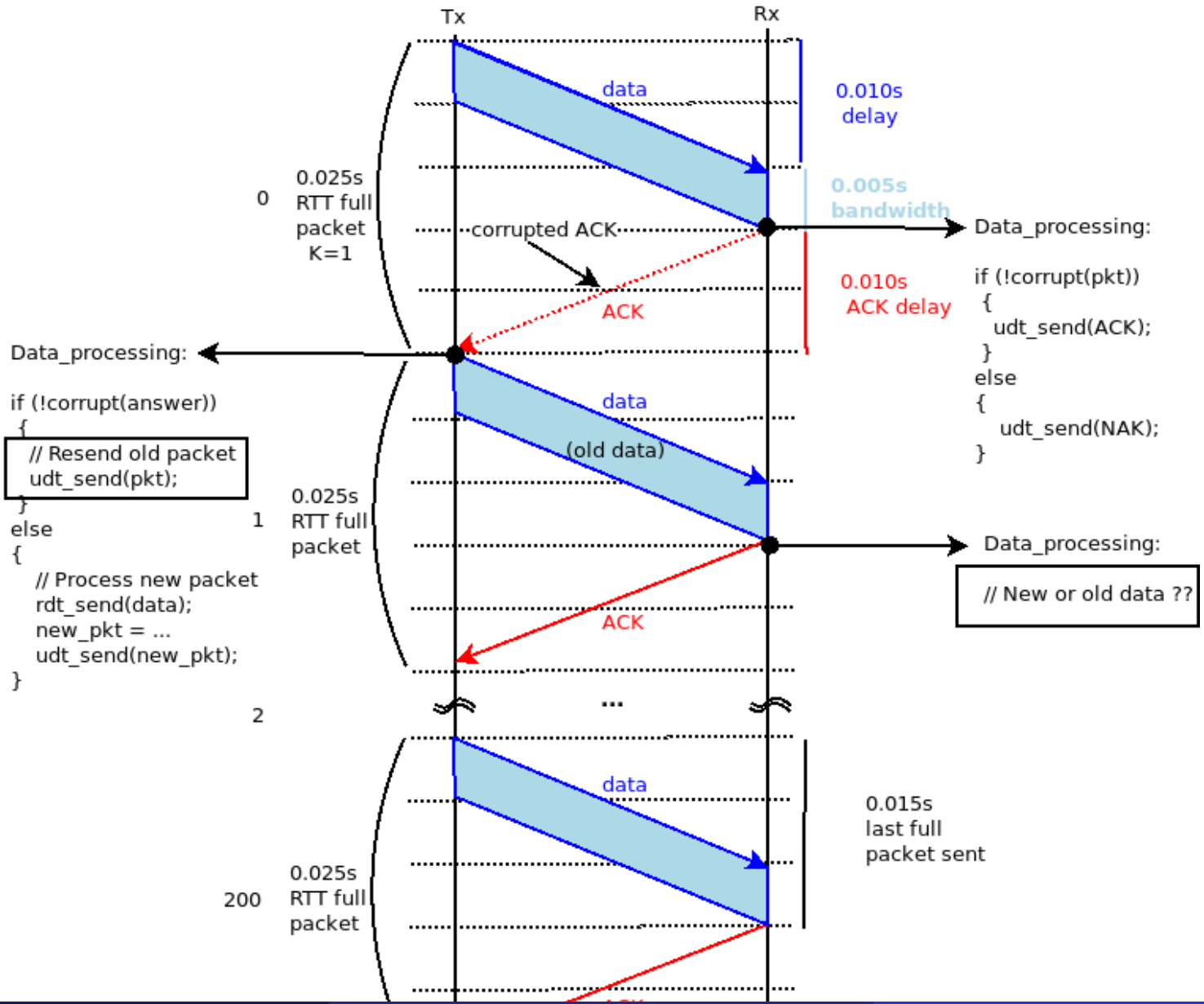
- Ahora emisor y receptor sincronizados, no se requiere buffering entre proceso usuario y RDT. Max=1 dato.
- De a un paquete/segmento por vez.
- Hasta que no se confirma no se pasa al siguiente segmento.
- Para ACK/NAK no requiere calcular el chksum porque se supone seguro.
- Conocido como mecanismo Stop & Wait (S&W), v2.0.



Problemas de ACK/NAK corruptos

- Se pre-suponía entrega de datos NO confiable en cierto grado, se pueden corromper pero siempre llegan. Se suponía además que ACK/NAK llegaban OK.
- Problema, si cambiamos a un modelo aún más real, donde la capa subyacente puede corromper los NAK/ACK.
- Cálculo de chksum ACK. Qué hacer ante un NAK/ACK corrupto ?
- Se debería retransmitir.
  - Si el corrupto fue un NAK, no hay problema, pero
  - Si fue un ACK el receptor recibirá un mensaje duplicado y no podrá distinguirlo (DUP).
- Se debe identificar el mensaje de dato para que el receptor sepa si es un duplicado, ya que no sabe que el ACK se corrompió, se utiliza ID binario: (0,1).

Problemas de ACK/NAK corruptos (gráfico)



Estructura de Datos para Algoritmo S&W con Bit-Counter

- Estructura tentativa del paquete/segmento.

```
typedef enum {ack, nak, dat} type_t;

typedef struct pk {
   // Header
   int       len;
   type_t    ack;   // ACK | NAK | DATA
   int       id;    // seq 0,1
   ck_t      chksum;
   // Data
   byte_t    payload[...];
} packet_t;
```

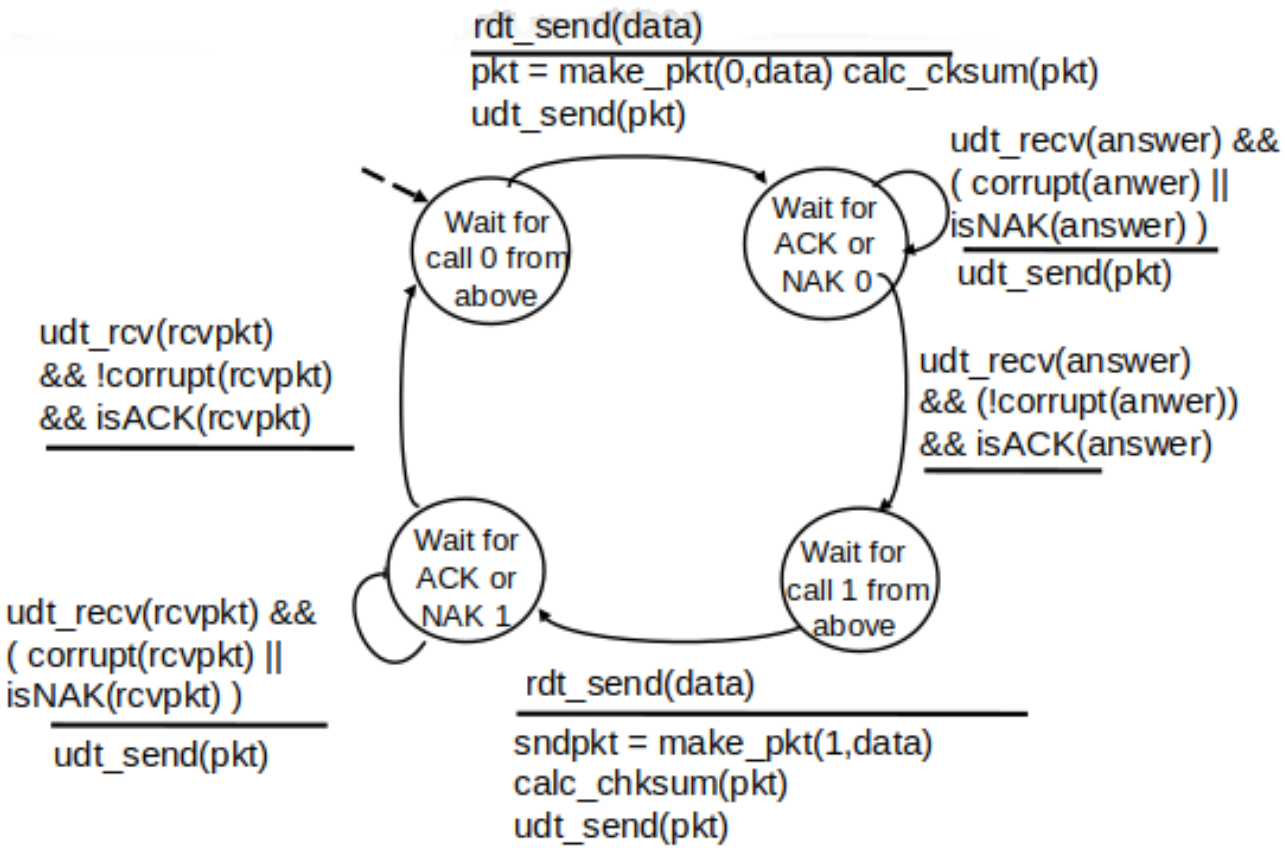
```
void sender2.1()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result  = OK;
  packet_t  *answer = NULL;


  // Mientras no se termine la comunicación
  while (result != EOT)
   {
O0: // Emisor recibe datos de capa
    // superior y arma PDU desde data
    result = rdt_send(data);
    // Arma paquete
    // pkt->header.id = 0
    pkt = make_pkt(0,data); // ID==0
O1: //pkt->header = ...
    calc_chksum(pkt);
    udt_send(pkt);
O2: // recv && (NAK || corrupt)
    while  ((udt_resv(answer)&&(!isACK(answer))
    {
       udt_send(pkt);
    } // isACK ...
```

```
void receiver2.1()
{
  packet_t  *pkt;
  packet_t  *answer;
  byte_t    *data;
  result_t  result  = OK;
  int       ok      = 0;
  int       wait_id = 0;
  // Mientras no se termine la com.
  while (result != EOT)
   {
O0:       // Receptor recibe datos
          udt_recv(pkt);

O1:       // Receptor chequea datos
          // y confirma con ACK/NAK
          if (!corrupt(pkt)) {
             answer = make_pkt(ACK);
          }
          else {
             answer = make_pkt(NAK);
          }
          calc_chksum(answer);
          udt_send(answer);
          ...
```

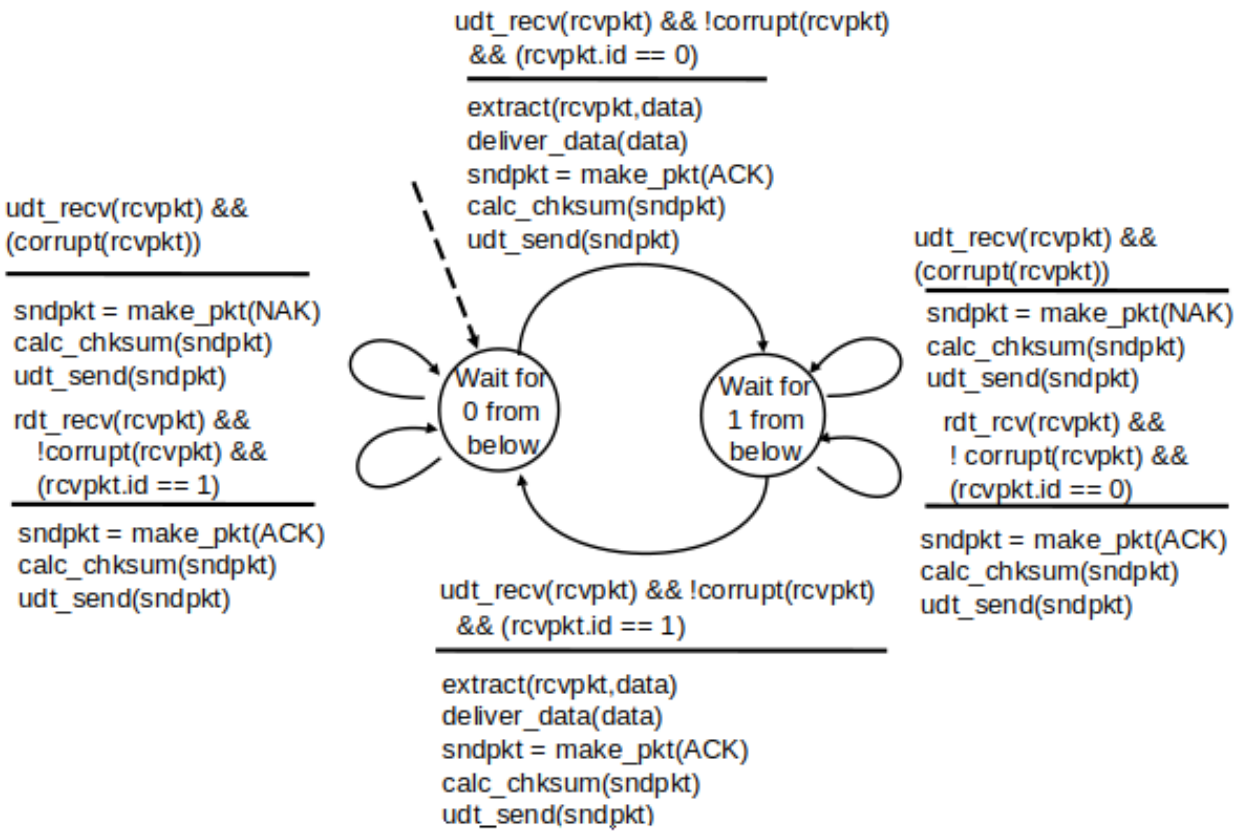
```
   ...
O3: // Emisor recibe nuevos datos de capa
    // superior y arma PDU desde new data
    result = rdt_send(data);

    // Arma paquete con nuevo id
    // pkt->header.id = 1
    pkt = make_pkt(1,data); // ID==1

O4: //pkt->header = ...
    calc_chksum(pkt);
    udt_send(pkt);

O5: // recv && (NAK || corrupt)
    while ((udt_resv(answer)&&(!isACK(answer))
    {
        udt_send(pkt);
    }
O6: // Emisor vuelve a 0
    // goto O0 -- esta en el while
  }
}
```

```
...
O2:   // Receptor entrega datos a
      // capa superior data=pkt->payload
      // Solo si el id es el que esperaba
      if (pkt->id == wait_id)
      {
          extract(pkt, data);
          result = deliver_data(data);
          wait_id = (wait_id + 1) % 2:
      }
O3:  // Receptor vuelve a 0
     // goto O0 -- esta en el while
   }
}
```

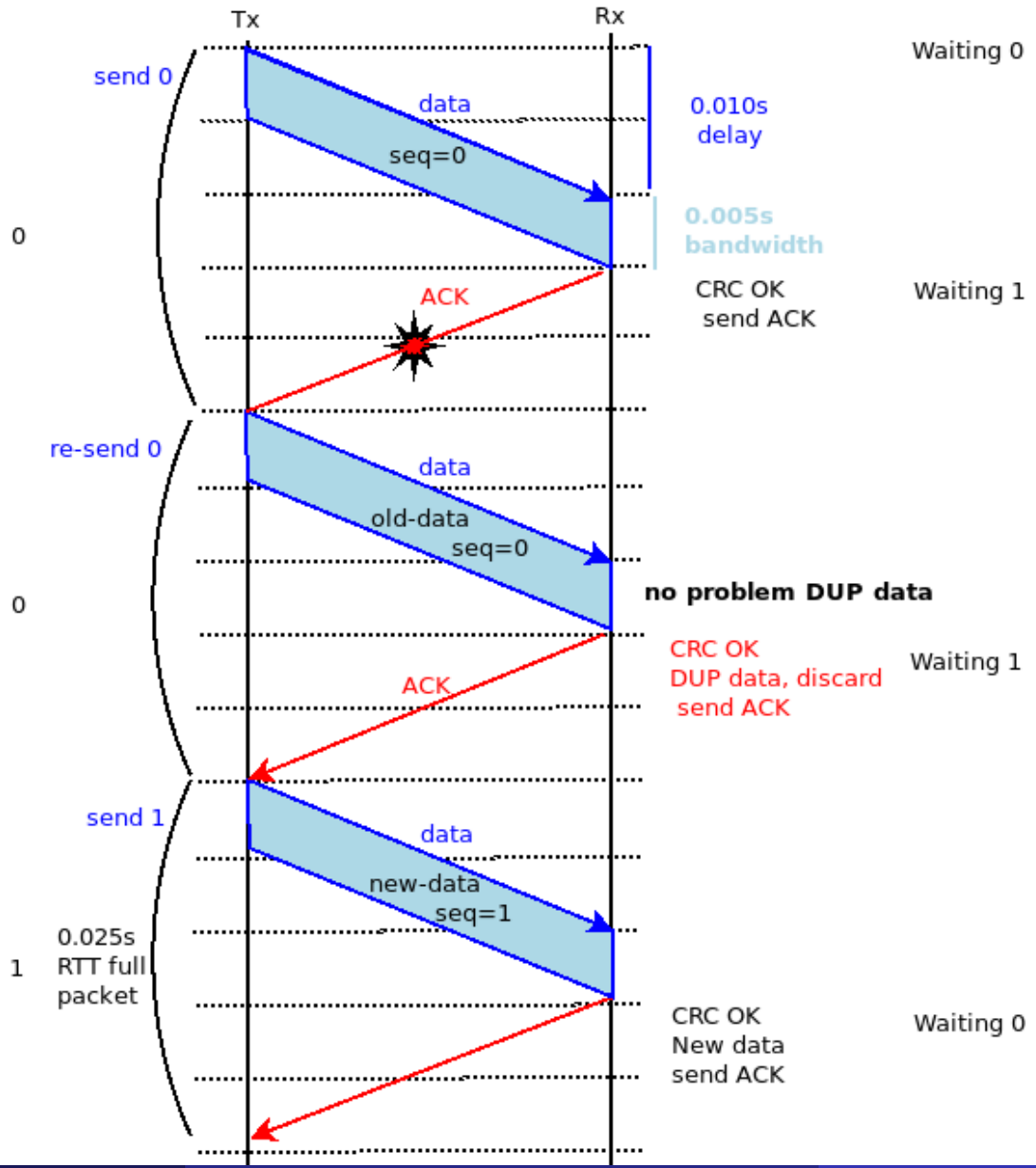
FSM Canal Errores data/ack chksum, Send



FSM Canal Errores data/ack chksum, Recv



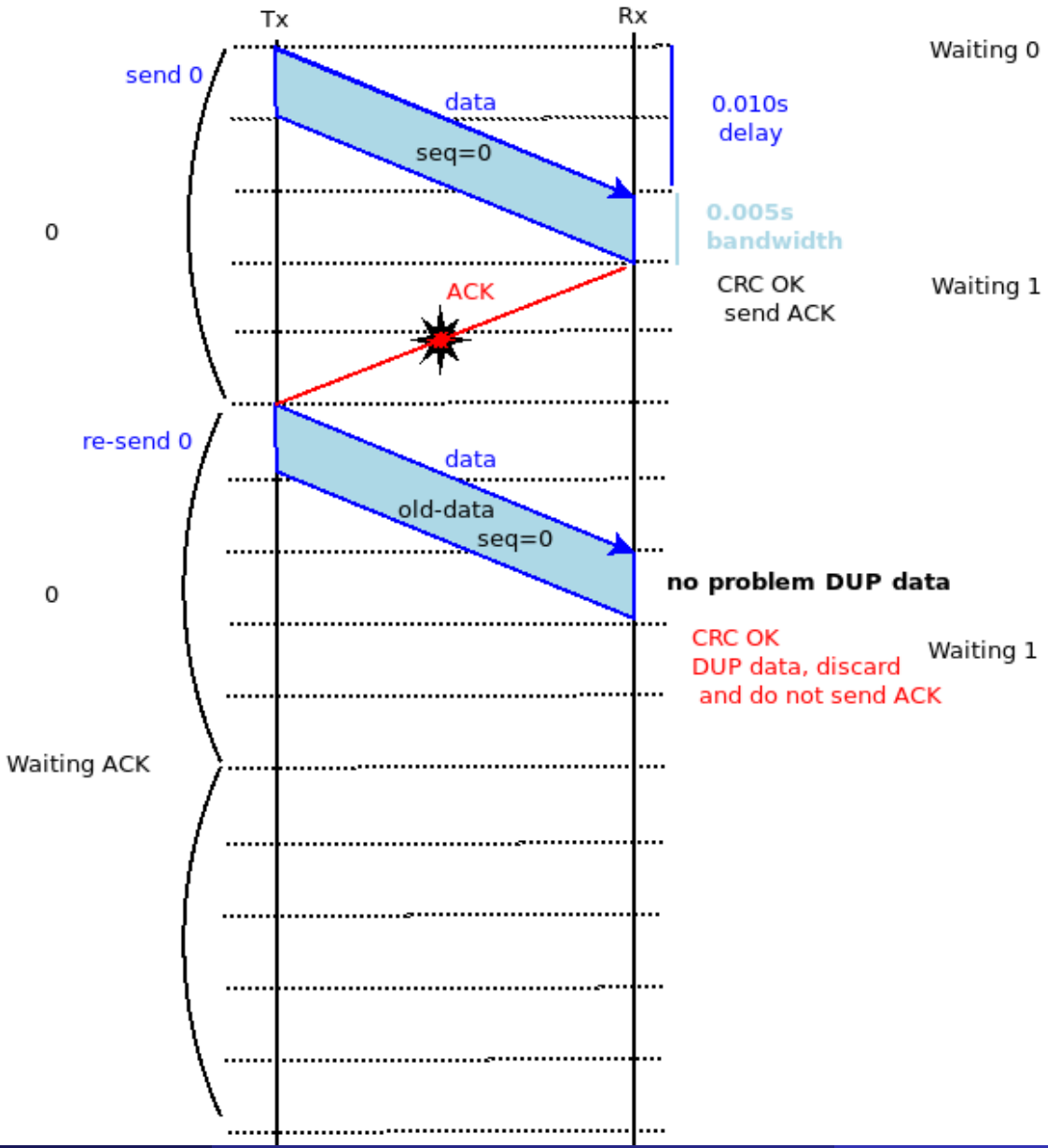
ACK/NAK corruptos con Bit Counter en Data



Análisis de Algoritmo para ACK/NAK corruptos

- Valores binarios (0,1) alcanza, por qué?
- Sender debe chequear si el ACK/NAK esta corrupto.
- Sender se queda en el estado hasta que se asegura que el ACK llegó OK.
- El emisor se puede simplificar con loop y un módulo 2 para los nros. de secuencia y no copiar el código.
- El receptor debe chequear que el paquete recibido no sea duplicado.
- Siempre debe confirmar, no importa si recibe un duplicado. No sabe si se recibió OK en el emisor. Si no confirma, el emisor se queda bloqueado esperando el ACK/NAK.

Problema si no confirma DUP



Canal con pérdida de Datos

- Se presuponía entrega de datos NO confiable en cierto grado, se pueden corromper pero siempre llegan.
- Problema, si cambia a un modelo aún más real, donde la capa subyacente no es confiable en el grado en el que se pueden perder los datos, no solo corromper.
- Para el primer análisis se vuelve a relajar la corrupción de los ACK/NAK. No hay ACK/NAK corruptos.
- Al perderse los datos, con los algoritmos anteriores, se quedarían bloqueados los procesos en `udt_recv()`, uno esperando por el dato y el otro por la confirmación.

Canal con nuevos errores, pérdida de Datos

- Sí se pierden datos, no los ACK/NAK.
- No se duplican.
- No se desordenan.
- Sí se corrompen datos:
  - Se chequean con Checksum/CRC o mecanismo similar.
  - No se pasan a la capa superior si tienen errores.
  - Requiere mecanismo para avisar al emisor del error.
- No se corrompen las confirmaciones, ACK/NAK.
- No hay delay mayor de 1 RTT.

Algoritmo: Stop&Wait (S&W) v2.1.1

- Se agregar timeout en `Sender2.1.1:01` y `Sender2.1.1:04`.
- Se debe indicar donde se detiene el timer.
- Ante un error de checksum/CRC el receptor podría omitir el NAK ?
- Si los ACK/NAK son seguros no se requiere identificar los mensajes de datos, ID (0,1).
- Cómo calcular el timer? En base al RTT.
- Siempre es el mismo el RTT?

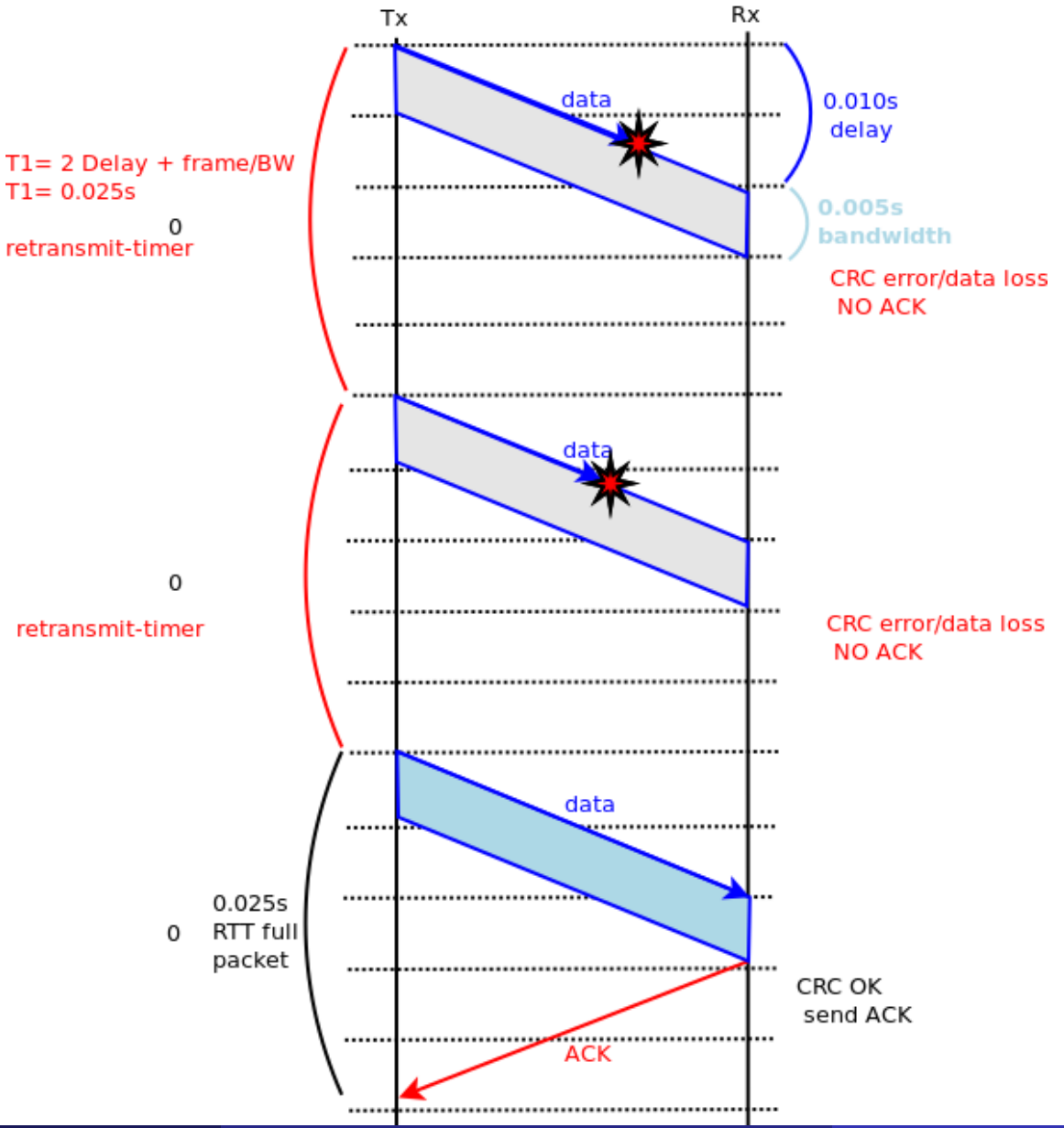
```
void sender2.1.1()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result = OK;
  packet_t  *answer = NULL;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {
O0: // Emisor recibe datos de capa
    // superior y arma PDU en data:
    result = rdt_send(data);

    // Arma paquete
    pkt data = make_pkt(data);
    calc_chksum(pkt);
    // TMOUT | NAK
    while  ( ! isACK(answer) )
    {
O1:     // Antes de enviar genera timer,
        // si habia uno corriendo, reset
        start_timer(RTT+delta);
...
```

```
void receiver2.1.1()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result = OK;
  int       ok     = 0;

  // Mientras no se termine la com.
  while (result != EOT)
   {

     while (!ok)
      {
O0:     // Receptor recibe datos
        // desde capa inferior
        udt_recv(pkt);

O1:     // Receptor chequea datos
        // y confirma con ACK/NAK
        if (!corrupt(pkt)) {
            udt_send(ACK); ok = 1;
          }
        else {
            udt_send(NAK); ok = 0;
          }
      }
...
```

```
...
O2:     // Emisor envía datos:
        udt_send(pkt);

O3:     // Emisor espera respuesta
        // o timeout, valor de salida
        // answer ::= ACK | NAK | TMOUT
        udt_recv(answer);

    } // Emisor se queda
      // retranmitendo el mismo dato

O4: // Se detiene timer
    stop_timer();
   // Emisor vuelve a 0
   // goto O0 -- esta en el while

   }
}
```

```
...
O2:  // Receptor entrega datos
     // a capa superior data=pkt->payload;
     data = extract(pkt, data);
     result = deliver_data(data);

03:  // Receptor vuelve a 0
     // goto 0 -- esta en el while
   }
}
```

Stop & Wait 2.1.1



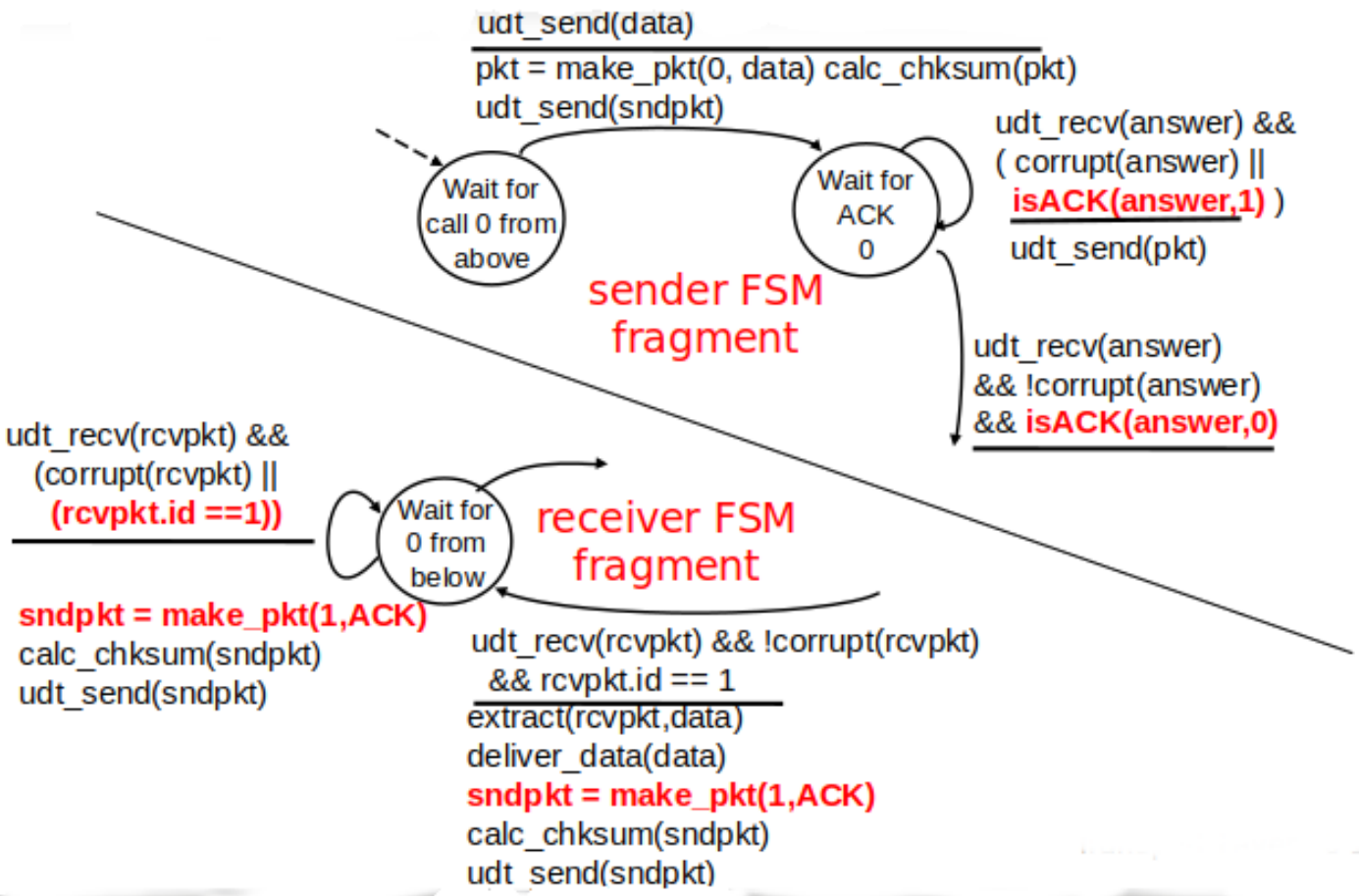
Problemas de S&W v2.1.1

- Estamos suponiendo que tenemos entrega confiable de confirmaciones, ACK/NAK:
  - No se pierden ACK/NAK.
  - No se corrompen ACK/NAK.
- Problema, si cambiamos a un modelo más real, donde la capa subyacente no es confiable tampoco para las confirmaciones, como v2.1.
- Se podrían corromper los ACK/NAK. Habría que retransmitir.
- Al perderse los datos se retransmiten con el timer, si se pierde el ACK/NAK es como que no llego el mensaje y se retransmite por el timer también.
- El receptor puede recibir duplicados y no podrá distinguirlos, necesidad de identificar los mensajes, como en 2.1.

Algoritmo Free NAK

- Tratar de hacer un algoritmo que no necesite NAK.
- Alternativas:
  - Incluir nro. de secuencia explícito en ACK, indicar que dato recibió OK.
    - V2.2: si dato corrupto, se envía el nro. anterior., si dato OK, envío el que se recibió (módulo 2).
    - Se podría utilizar también que dato espera recibir (Confirmación hacia adelante). Si dato corrupto, envía el nro. que espera recibir, si dato OK, envía el siguiente módulo N (S&W, $N = 2$).
  - V2.3: Usar timer en el emisor, si dato corrupto, no se confirma y se espera que el emisor retransmita como si se hubiese perdido (Como en v2.1.1).

FSM Errores Algoritmo Free NAK sin timer v2.2



Estructura de Datos para Algoritmo Free NAK v2.2

- En el ejemplo el id del dato se puede usar para el id del ACK, solo se envían datos en un sentido.
- Si hay envío de datos en los dos sentidos y se confirma en los mismos datos (piggy-backing) se requiere un id para el ACK.

```
typedef enum {ack, nak, dat} type_t;

typedef struct pk {
  // Header
  int      len;
  type_t    ack;     // ACK | NAK | DATA
  int       id;      // seq 0,1
  int       ack_id:  // seq 0,1 Admite piggy-backing
  ck_t      chksum;
  // Data
  byte_t   payload[...];
} packet_t;
```

Algoritmo: Stop&Wait (S&W) v2.3 - Free of NAK

- Con el algoritmo Stop&Wait v2.1.1 si se corrompe el dato podrá, omitir la confirmación y el emisor deberá retransmitir por timeout.
- Con el algoritmo Stop&Wait v2.1.1 si se corrompe el ACK, para el emisor es como si se perdiese el mensaje, dará timeout y retransmitirá.
- **ERROR !!!** el receptor pensará que es un nuevo dato y lo pasará a la capa superior como tal, cuando fue una retransmisión. Aparecen datos duplicados.
- Se debe evitar este problema, por ejemplo identificando con IDs los datos enviados como en v2.2, S&W + Bit Counter (0,1).

Algoritmo: Stop&Wait (S&W) v2.3 - Free of NAK

- Cambiar `Sender2.3:03`

```
void sender2.3()
...
O3: // Emisor espera respuesta
    // answer ::= ACK | TMOUT | ERR(corrupt(answer))
    // || NAK (se suprime)
    udt_recv(answer);
...
```

- El envío de ACK, requiere agregar código para detección de errores como en 2.2.
- Se puede simplificar el Receiver, si se corrompe el dato, directamente se descarta y se espera timeout del emisor.

- Se debe agregar el envío de ID/SEQ para los mensajes de datos.
- En principio no requiere ID/SEQ para ACK si se mantiene restricción de RTT.

```
void sender2.3()
{
  ...
  int       counter = 0;
        ...

  // Arma paquete
  pkt = make_pkt ( counter , data );
  ...

  counter = (counter + 1) % 2;
  ...
```

```
void receiver2.3()
{
  packet_t  *pkt;
  packet_t  *answer;
  byte_t    *data;
  result_t  result  = OK;
  int       ok      = 0;
  int       wait_id = 0;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {
      ok = 0;
      while (!ok)
       {
O0:      // Receptor recibe datos
         udt_recv(pkt);
O1:      // Receptor chequea datos
         // y confirma con ACK,
         // sino espera tmout
         if ((!corrupt(pkt))&&(pkt->id == wait_id))
         {
              answer = make_pkt(ACK);
              calc_chksum(answer);
              udt_send(answer);
              ok = 1;
          } // else discard and wait retrans
       }
       ...
  ...
O2:    // Receptor entrega datos a
       // capa superior data=pkt->payload
       // Solo si el id es el que esperaba
       // Es la única forma se salir
       // del while
       //if (pkt->id == wait_id)
       extract(pkt, data);
       result = deliver_data(data);
       wait_id = (wait_id + 1) % 2:
       ok = 0;
O3:  // Receptor vuelve a 0
     // goto 0 -- esta en el while
   }
}
```

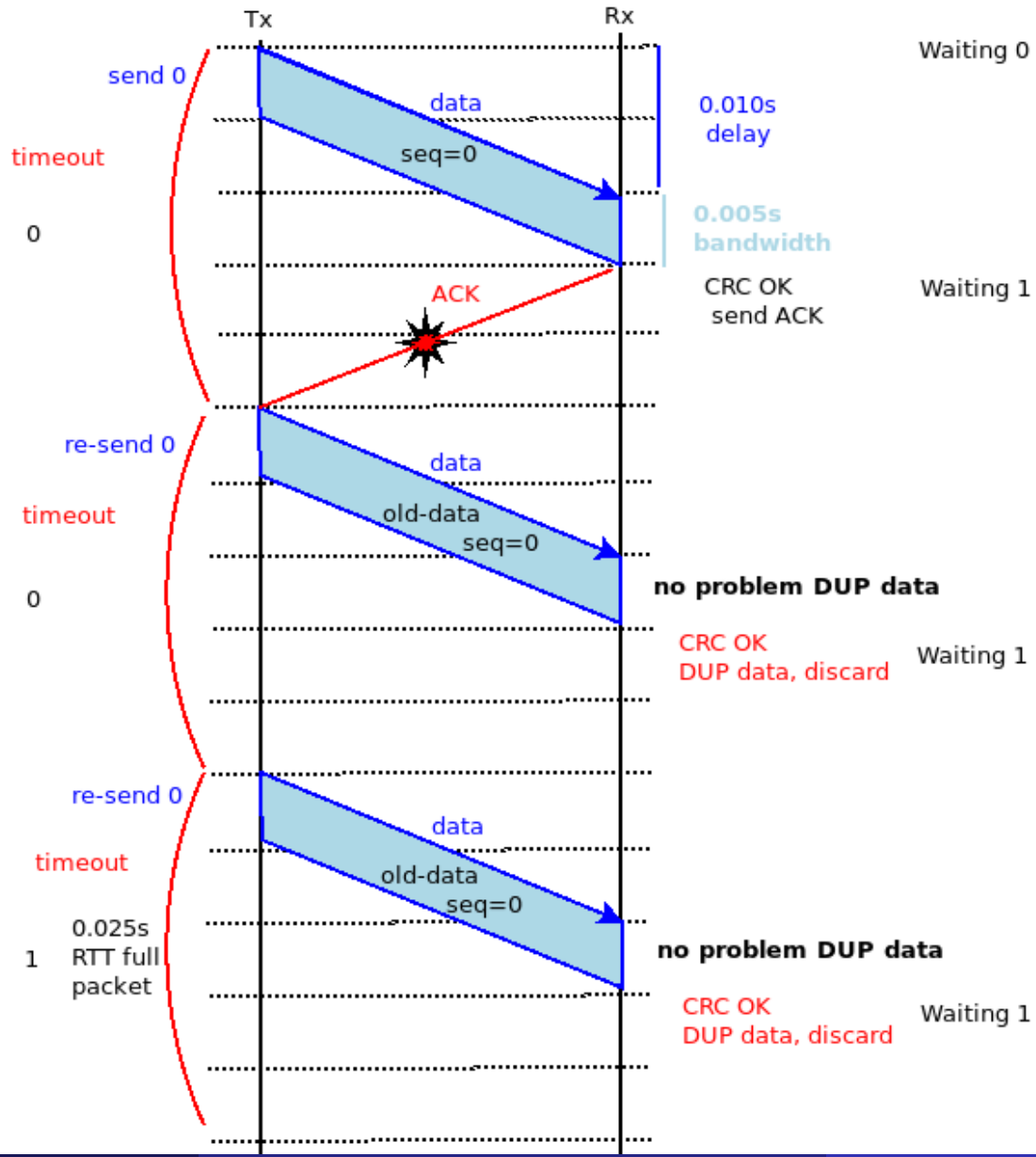
S&W + Bit Counter, Canal con más errores

- El algoritmo v2.3 S&W + Bit Counter parece ser adecuado para el siguiente entorno:
  - Sí se pierden datos, y Sí se pierden los ACK/NAK.
  - No se duplican.
  - No se desordenan.
  - Sí se corrompen datos, se chequean con Checksum/CRC o mecanismo similar, se avisa cuando no se reciben de forma correcta.
  - Sí se corrompen las confirmaciones, ACK/NAK.
  - No hay delay mayor de 1 RTT.

Problemas de S&W + Bit Counter

- **ERROR !!!**, el receptor no esta confirmando los duplicados.
- Dará timeout en el emisor y volverá a enviar el mismo.
- Si al receptor le llego bien y confirmo, pero la confirmación se corrompió o perdió, el emisor se queda enviando siempre el mismo, ya que el receptor espera el nuevo dato.
- El Receptor debe confirmar los DUPs.

Problema Stop & Wait 2.3 (No ACK DUP)

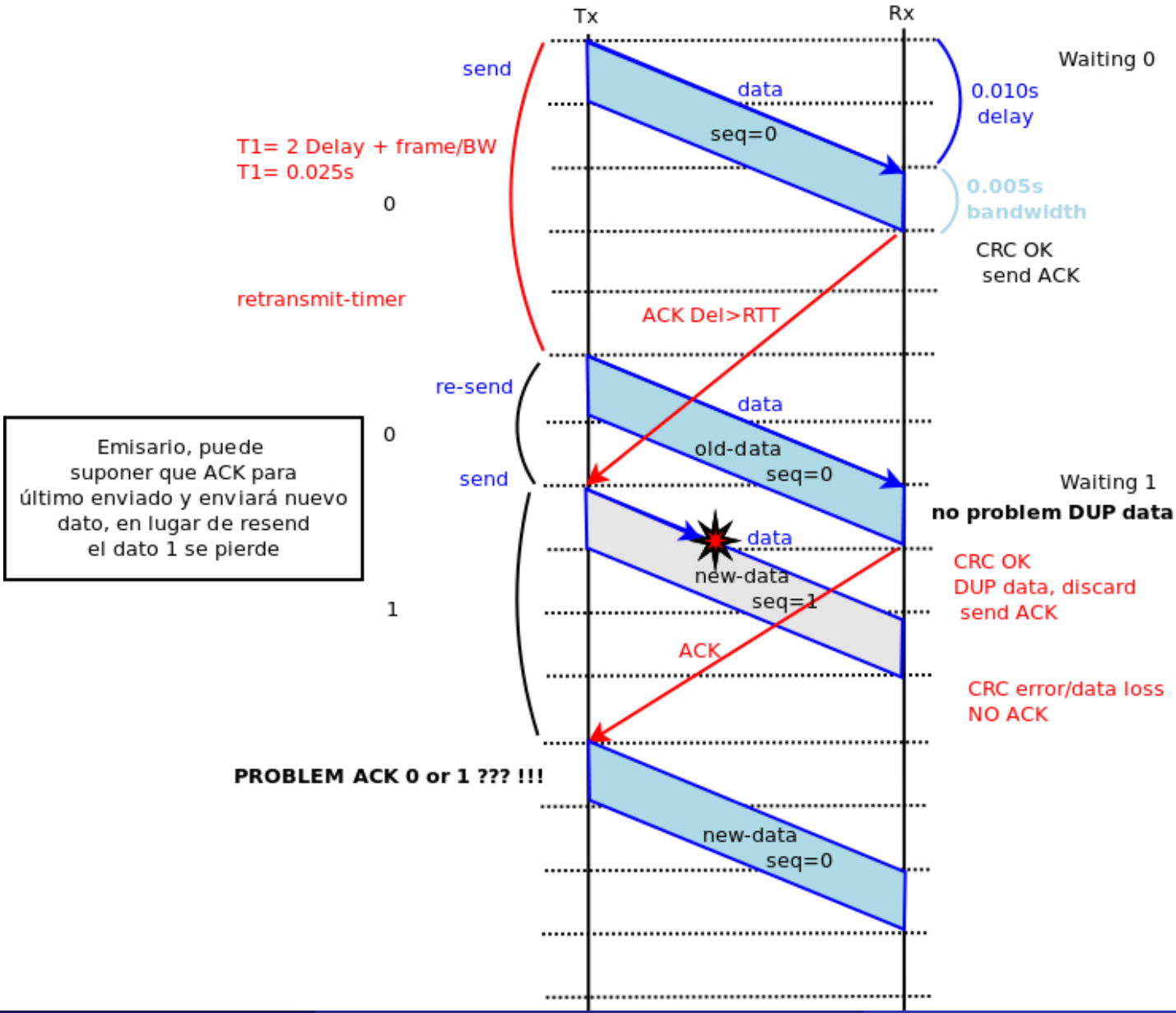


```
void receiver2.4()
{
...
O1:       // Receptor chequea datos
          // y confirma con ACK, incluso DUP
          // pero !corrupt, sino espera tmout
          if (!corrupt(pkt))
          {
               answer = make_pkt(ACK);
               calc_chksum(answer);
               udt_send(answer);
               if (pkt->id == wait_id)) ok = 1;
               else ok = 0; // (pkt->id != wait_id)
          }
          else
          {
              // discard and wait retrans
          }
...
```

Problemas de S&W + Bit Counter, v2.4

- No se considera que los Delayed ACK con valores mayores a 1 RTT.
- Los datos siempre van ordenados.
- No hay problema, de datos repetido porque tienen nro. de secuencia.
- Si hay problema de confundir un ACK con otro, ya que no tienen nro. de secuencia.
- El transmisor cree que siempre se confirma el corriente, pero puede suceder:
  1. Ocurre un timeout, re-envía e inmediatamente le llega el ACK retardado.
  2. Al recibir este envía segmento nuevo.
  3. Segmento nuevo se pierde.
  4. Recibe el ACK duplicado del segmento que retransmitió.
  5. Cree que es del nuevo.
- **ERROR!!!!** con $ACKDelay > 1RTT$.

Problema Stop & Wait 2.4 (No ACK ID)



Canal con Errores y 1 RTT < ACK Delay

- Sí se pierden datos, y Sí se pierden los ACK/NAK.
- No se duplican.
- No se desordenan.
- Sí se corrompen datos, se chequean con Checksum/CRC o mecanismo similar, se avisa cuando no se reciben de forma correcta.
- Sí se corrompen las confirmaciones, ACK/NAK.
- Sí $2RTT < MAXDelay < 3RTT$, con 2 nros. de secuencia alcanza 0..1.

Estructura de Datos para Algoritmo S&W v3.0

```
typedef struct pk {
   // Header
   int         len;
   bool        ack;      // 1,ACK | 0,DAT
   int         id;       // seq 0,1
   int         ack_id:   // seq 0,1 Admite piggy-backing
   ck_t        chksum;
   // Data
   byte_t   payload[...];
} packet_t;
```

Algoritmo: Stop&Wait + Bit Counter Data/ACK

- Los ACK también necesitan nro. de secuencia como primer algoritmo Free NAK: v2.2.
- Funciona con $ACKDelay > 2RTT$, pero ...
- Deja de funciona en casos con $ACKDelay > 3RTT$.

Algoritmo: S&W + Bit Counter Data/ACK: Sender

```
void sender3()
{
  packet_t  *pkt;
  byte_t    *data;
  result_t  result  =  OK;
  packet_t  *answer = NULL;
  int       counter = 1;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {

O0: // Emisor recibe datos de capa
    // superior y arma PDU en data:
    result = rdt_send(data);

O1: // Arma paquete
    counter = (counter + 1) % 2;
    pkt = make_packet(counter, data)
    calc_chksum(pkt);
...
```

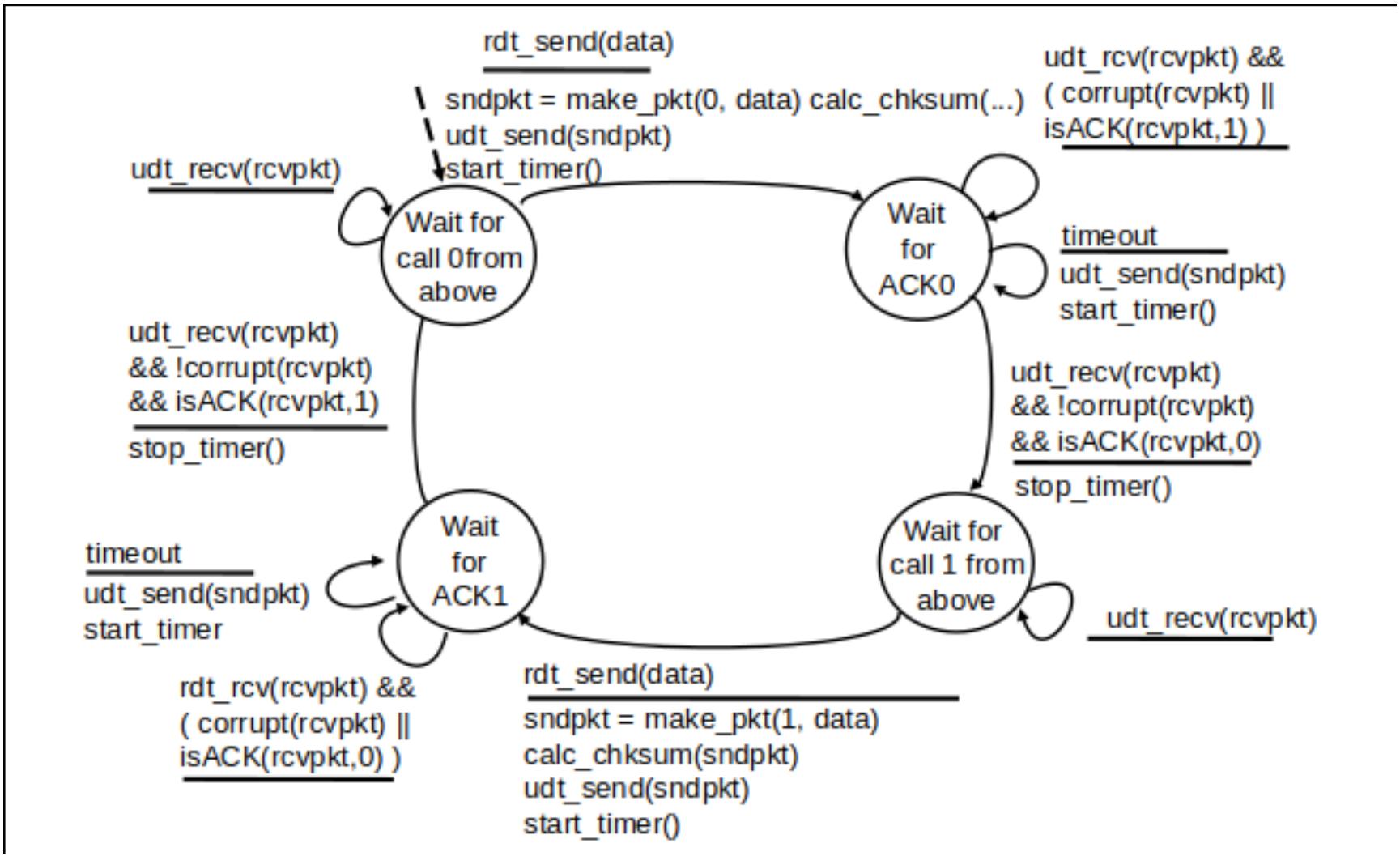
```
...
    // TMOUT | implicit NAK | ERR
    while ( (corrupt(answer))||
    (isACK(answer) &&
    (answer->ack_id !=counter)) )
    {

O2:     // Antes de enviar genera timer,
        // si había uno corriendo, reset
        start_timer(RTT+delta);

        // Emisor envía datos:
        udt_send(pkt);

O3:     // Emisor espera respuesta
        //answer ::= ACK|TMOUT|ERR
        udt_recv(answer);

    } // Emisor se queda
      // retransmitiendo el mismo dato

O4: // Se detiene timer
    stop_timer();
    // Emisor vuelve a 0
    // goto O0 -- esta en el while
  }
}
```

Algoritmo: S&W + Bit Counter Data/ACK: Receiver

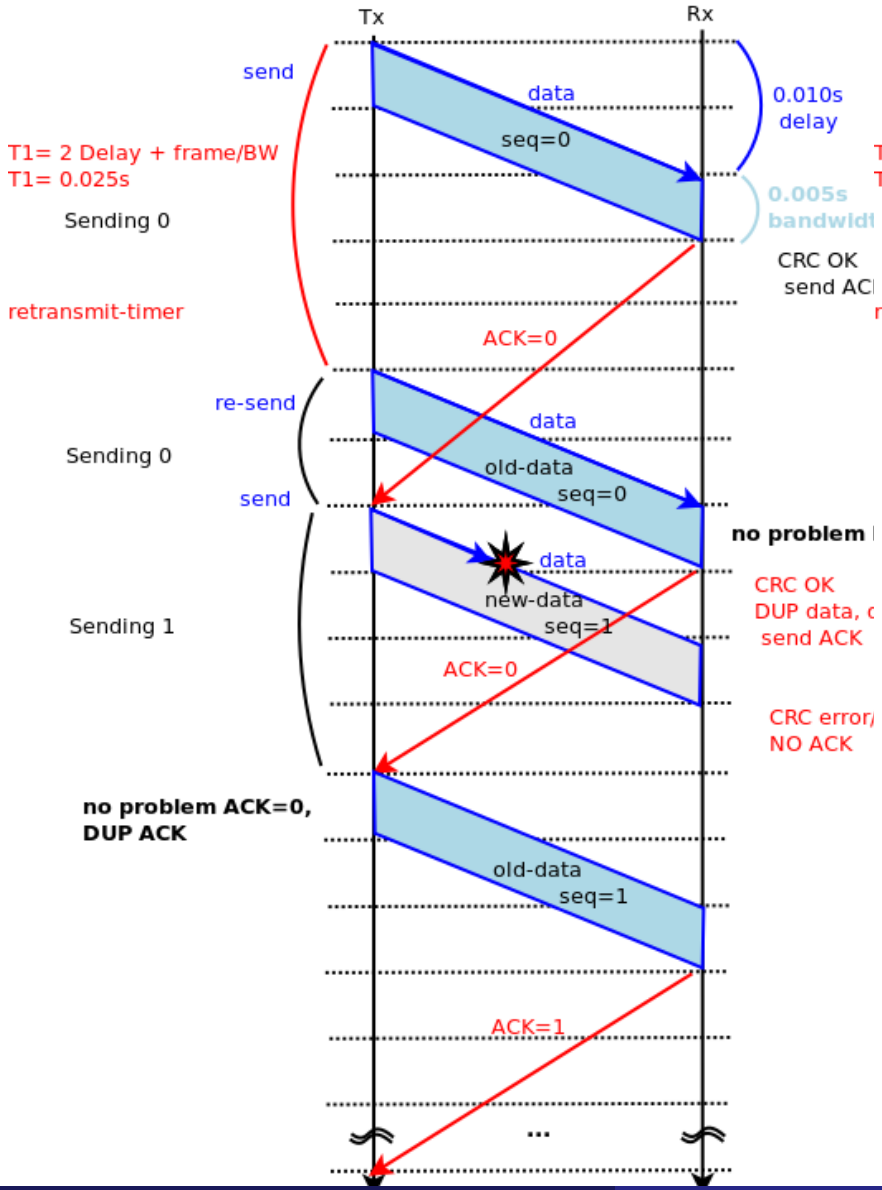
```
void receiver3()
{
  packet_t  *pkt;
  packet_t  *ack;
  byte_t    *data;
  result_t  result  = OK;
  int       ok      = 0;
  int       wait_id = 0;

  // Mientras no se termine la comunicación
  while (result != EOT)
   {
      ok = 0;
      while (!ok)
       {
O0:      // Receptor recibe datos
         udt_recv(pkt);
O1:      // Receptor chequea datos
         // y confirma con ACK,
         // sino espera tmout
         if (!corrupt(pkt))
           {
             // Arma ACK
             //ack->ack     = 1; //true
             //ack->ack_id  = pkt->id;
             ack = make_pkt(pkt->id, ACK);
           }
...
```

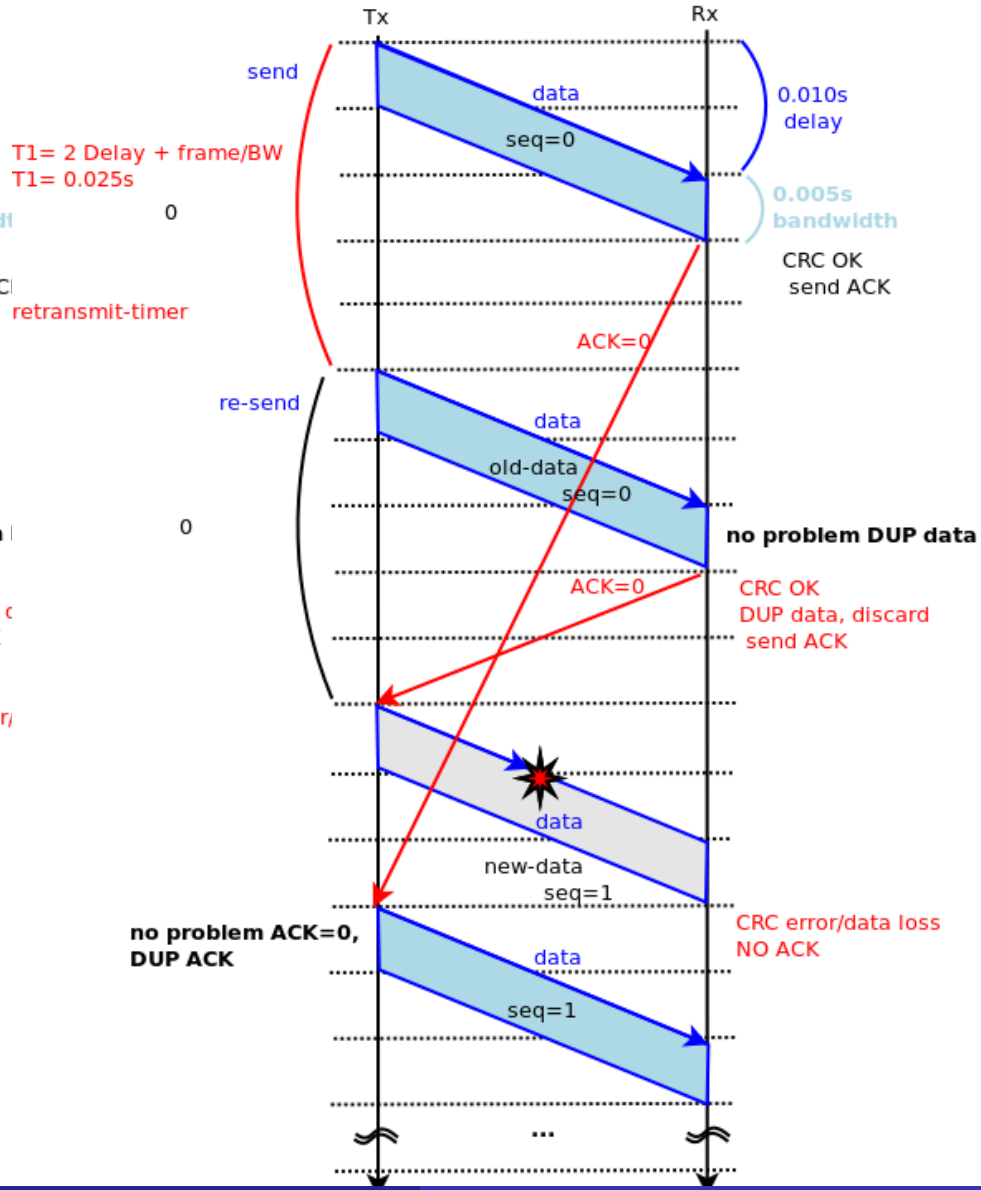
```
...
           if (pkt->id == wait_id)
              {
                ok = 1;
                wait_seq = (wait_seq+1)%2;
              } // else discard
            //else discard, ERR data,ok=0
       } // while (!ok)

O2:    // Receptor entrega datos
       // a capa superior
       // data=pkt->payload
       data = extract(pkt);
       result = deliver_data(data);
O3:    // Receptor vuelve a 0
       // goto 0 -- esta en el while
   } // (result != EOT)
}
```

Algoritmo S&W v3 - FSM - Send

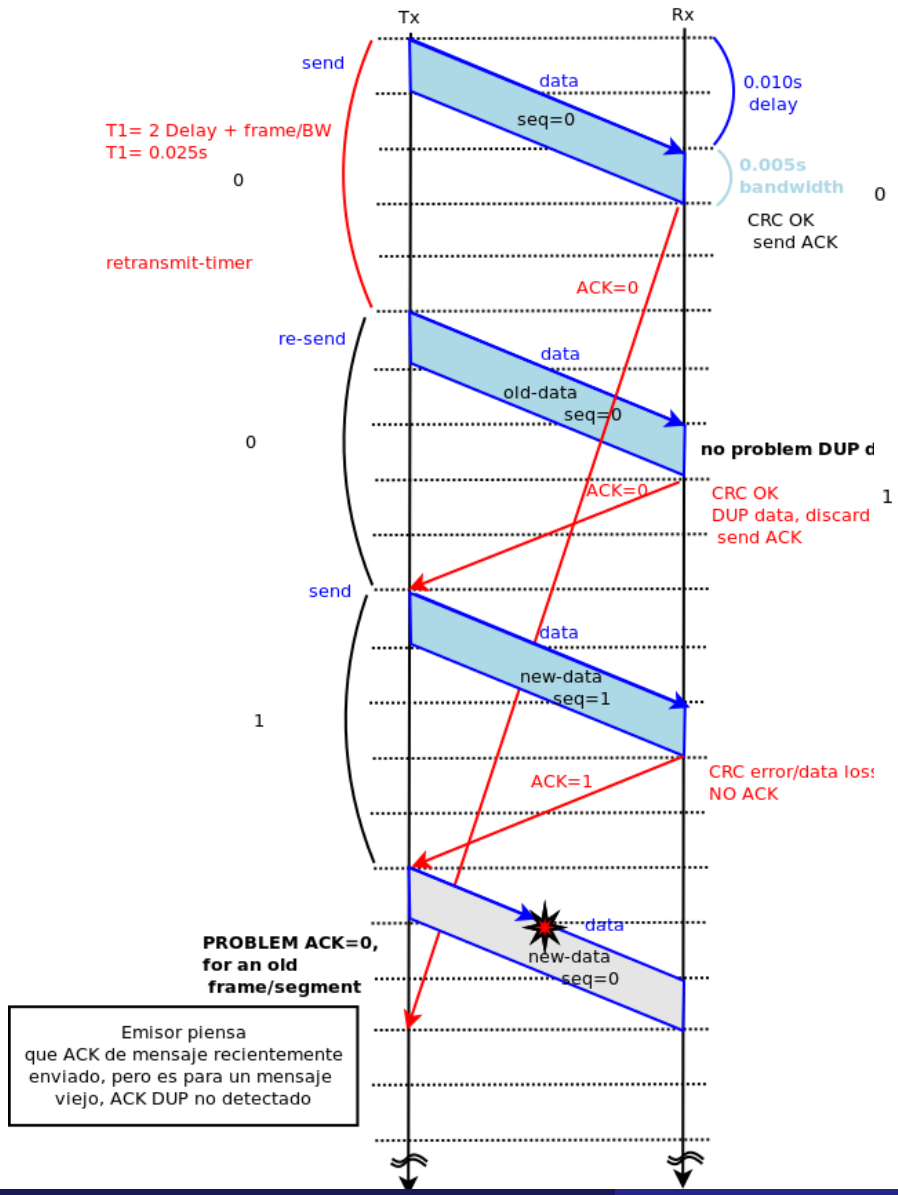


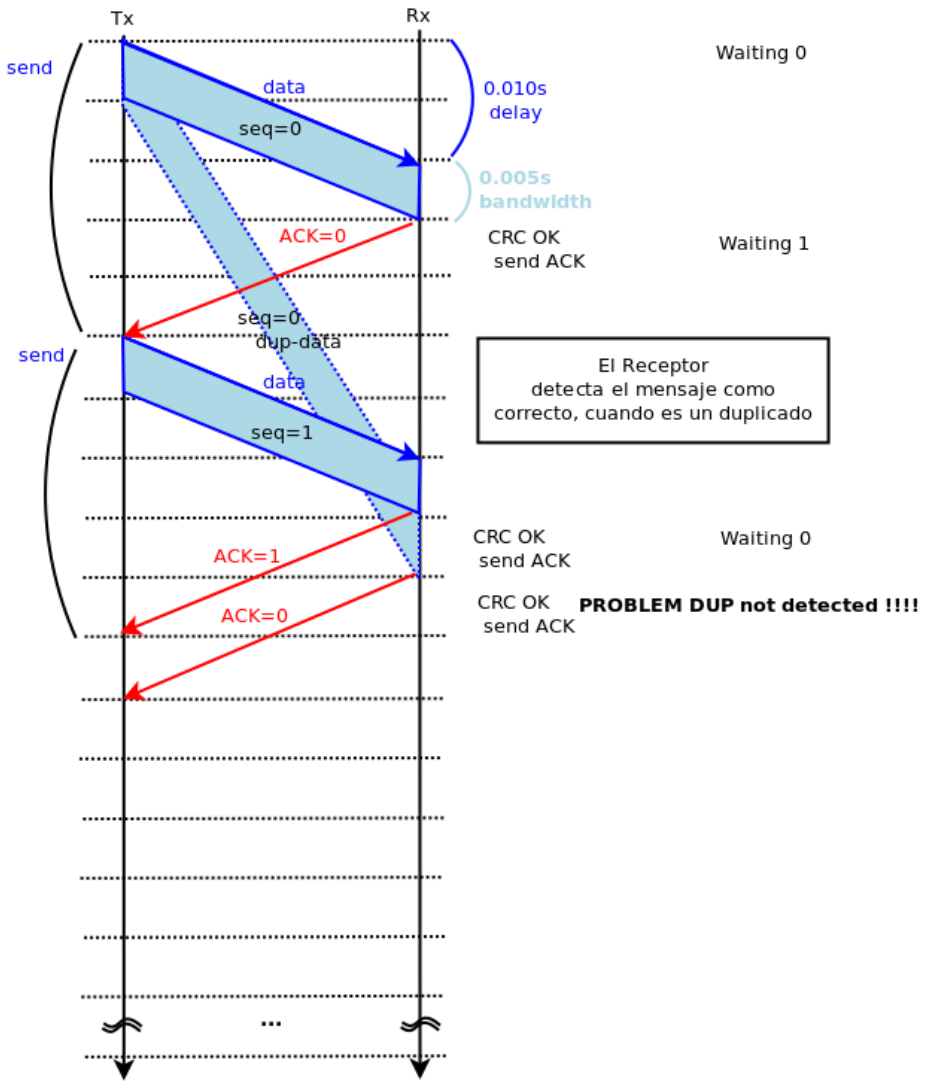
S&W v3 , $ACKDelay > 1RTT$, $ACKDelay > 2RTT$





Problemas S&W v3 , *ACKDelay* > *3RTT* , *DUPmsg*





Canal con Mensajes fuera de Orden y DUPs

- Sí se pierden datos, y Sí se pierden los ACK/NAK.
- Sí se duplican.
- Sí se desordenan.
- Sí se corrompen datos, se chequean con Checksum/CRC o mecanismo similar, se avisa cuando no se reciben de forma correcta.
- Sí se corrompen las confirmaciones, ACK/NAK.
- Sí hay delay mayores: $ACKDelay > N \times RTT$.

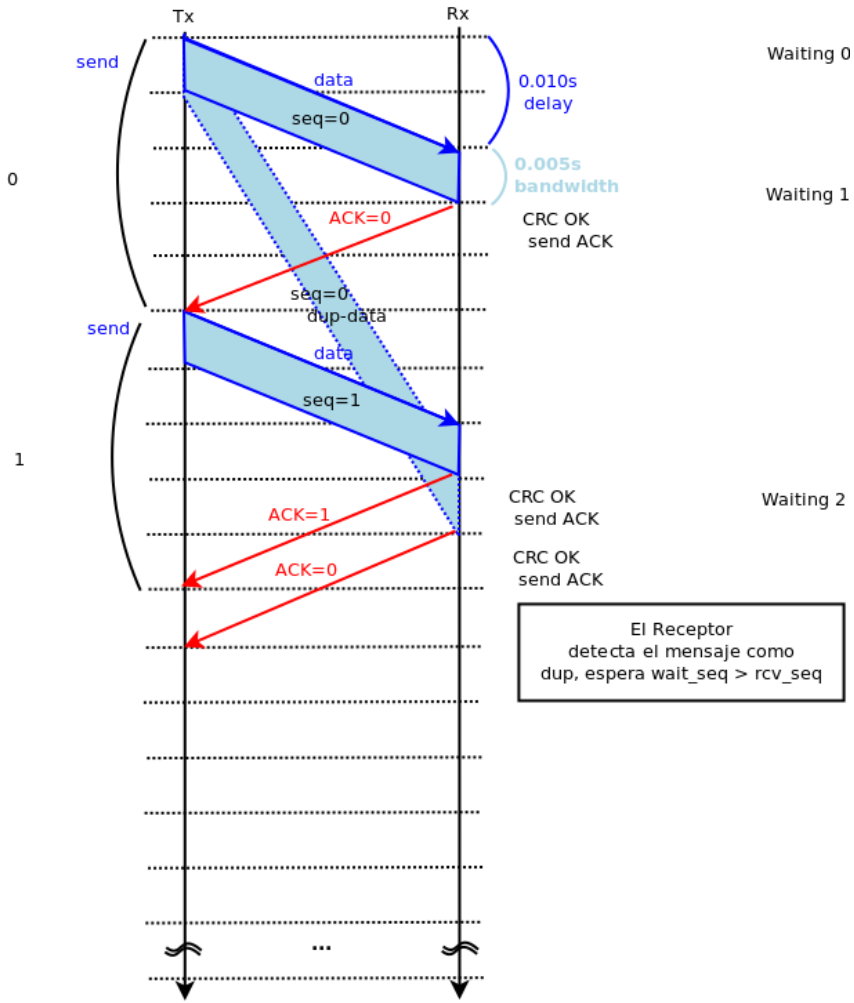
S&W + Seq Num Data/ACK > 1

- Si se considera que los datos pueden llegar desordenados.
- Además se considera los duplicados.
- Se debe aumentar los nros. de secuencia de ráfagas a valores mayores a los Delays de acuerdo a los RTT.
- Si se considera que los Delayed ACK, con valores mayores a varios RTT.
  - $2RTT < MAXDelay < 3RTT$, con 2 nros. de secuencia alcanza 0..1.
  - $N \times RTT < MAXDelay < (N+1) \times RTT$, entonces con N nros. alcanza: $0..N-1$.

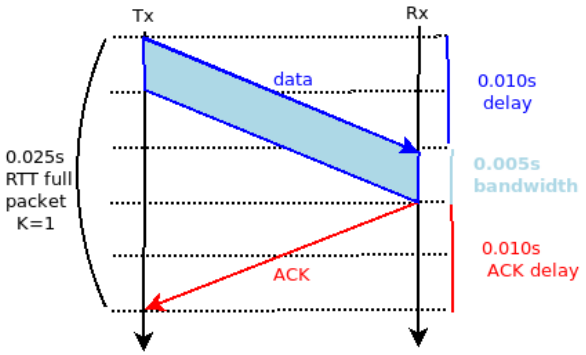
Estructura de Datos para Algoritmo S&W mejorado

```
typedef struct pk {
   // Header
   int         len;
   bool        ack;     // 1,ACK | 0,DAT
   int         id;      // seq 0,1,...N-1 (mod N)
   int         ack_id:  // seq 0,1,...N-1 piggy-backing
   ck_t        chksum;
   // Data
   byte_t   payload[...];
} packet_t;
```

S&W v4, Nros. de secuencias mayores



Análisis de Rendimiento



$$S = MaxSgmt_{bytes} = 1500B$$

$$RTT = Latencia_{seg} = 0.020s = 0.010s + 0.010s$$

$$L = DelayTransf_{seg} = 0.005s$$

$$R = BW_{bps} = \frac{S \times 8bits}{L} = \frac{1500 \times 8}{0.005} = 2400000bps = 2.4Mbps$$

$$U = Utiliz = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = 0.005/(0.020 + 0.005) = 0.2(20\%)$$

Se obtiene: $2.4Mbps \times 0.2 = 0.4Mbps = 400Kbps$

Análisis de Rendimiento (Cont.)

- Si RTT aumenta y BW aumenta se hace peor.
- Por ejemplo enlace de 1 Gbps y 50ms de latencia ida y vuelta $BDP = 1Gbps \times 50ms$.

$$L = 1Gbps$$
$$RTT = 0.050s$$
$$\frac{L}{R} = \frac{1500 \times 8}{(1 \times 1000^3)} = 0.0000120s$$
$$U = \frac{0.0000120}{(0.050 + 0.0000120)} = 0.00024(0.0024\%)$$

Se obtiene: $1Gbps \times 0.00024 = 240Kbps$

Conclusiones

- S&W base soluciona pérdida de datos, con restricciones casi ideales sobre la red.
- S&W no soluciona ACK perdidos o dañados, ni ACK delay.
- S&W+bit counter en datos soluciona pérdida de datos y de ACK.
- S&W+bit counter en datos no soluciona $ACKDelay > 1RTT$.
- S&W+bit counter datos/ACK soluciona los problemas anteriores.

Conclusiones (Cont.)

- S&W+bit counter datos/ACK no soluciona:
  - Datos/ACK delayed múltiples RTT ($> 3RTT$).
  - Datos fuera de orden.
  - Duplicados (DUPs).
- S&W+nros de secuencias $0..N$ datos/ACK soluciona la mayoría de los problemas en la red.
- Otros problemas:
  - Sequence Number Wrap Arround PAWS (puede usarse time-stamping)
  - El sistema es ineficiente, envía un dato por vez, ventana de transmisión/recepción: $K = 1$, $W = 1$.

Referencias

[K-R] Computer Networking International Edition, 6e. James F. Kurose & Keith W. Ross. ISBN: 9780273768968.