Egon Börger (Pisa)

Alternating Bit and Sliding Window File Transfer Protocol

A ground model ASM and its refinement

Dipartimento di Informatica, Università di Pisa, Italy
boerger@di.unipi.it

Requirements for acknowledged file transfer

- **Goal**. Transfer a finite sequence $F(1), \ldots, F(n)$ of files from a sender to a receiver s.t.
  - eventually the receiver has received the entire sequence (say $F = G$)
  - eventually the sender has received an ack of receipt for each file
  - assuming that during the transmission
    - finitely many consecutive messages may get lost (but not changed)
    - messages do not overtake (i.e. arrive in order of sending)
  - with the communication medium kept abstract

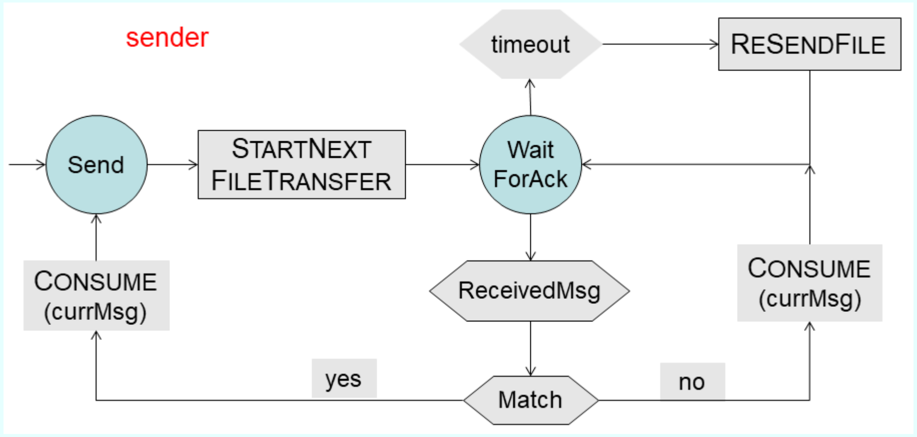
NB. It is not required that the receiver receives an ack that the sender has received an ack for the receipt of the last file.

Algorithmic idea for acknowledged file transfer

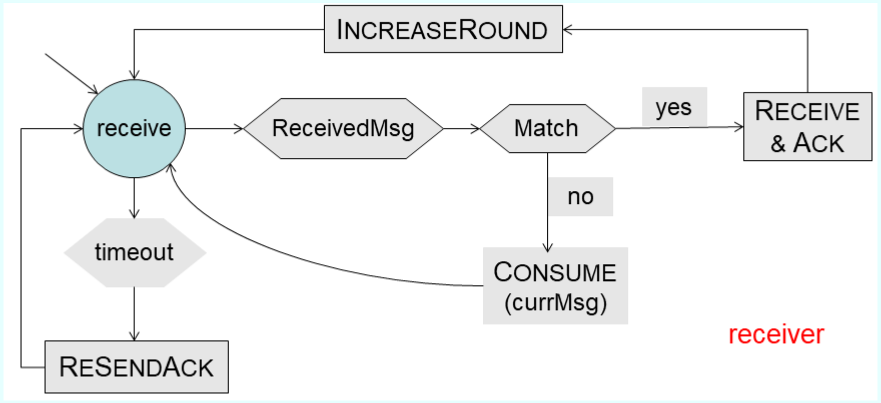
- In rounds $1, \ldots, n$, one per file, the *sender*
  - $\textsc{Send}$s one (the current) file and waits for an acknowledgement
  - while *wait*ing continues to $\textsc{ReSendFile}$ upon *timeout*
    - until an ack of receipt arrives from the *receiver*
      - · 'closing the *currRound*' by stopping resending and enabling to $\textsc{StartNxtFileTransfer}$ if not yet $currRound = n$

  until the last file transfer (in $currRound = n$) is acknowledged.
- When sending file $F(round)$, a *syncBit*, here $round \bmod 2$, is
  - attached to $FileMsgs\ (F(round), round \bmod 2)$
  - extracted and resent by the receiver as *AckMsgs*
  - checked upon *ReceivedMsg* by *sender*/*receiver* for *Match*ing its own *syncBit*, namely $currRound \bmod 2$
    - in case of matching, the *syncBit* is flipped for next $round + 1$

Turning the algorithmic idea into a 2-agent ASM AltBit



AltBit is a communicating ASM with 2 agents, sender and receiver, each coming with its respective program.[1]



[1] Figures © 2016 Springer-Verlag Germany, reused with permission.

Adding data to $\textsc{AltBit}$ control flow: file (re-)sending action

```
$\textsc{StartNextFileTransfer} =$
    if $currRound < n$ then                                  -- initially $currRound = 0$
        $\textsc{Send}((nextFile, nextSyncBit),$ to $receiver)$
        $\textsc{IncreaseRound}$
where
    $\textsc{IncreaseRound} = (currRound := currRound + 1)$
    $nextFile = F(currRound + 1)$
    $nextSyncBit = currRound + 1 \bmod 2$                    -- flipped $syncBit$
```

```
$\textsc{ReSendFile} =$
    $\textsc{Send}((F(currRound), currRound \bmod 2),$ to $receiver)$
```

Initially, at $sender$, $currRound = 0$, $mode = send$, no msgs around.

From English to ASM: when msgs $Match$

The assumption that msgs do not overtake is reflected by using as (initally empty) mailboxes *MsgQueue*s.

- the receiver's $MsgQueue$ contains *FileMsgs* $(F(i), i \bmod 2)$, where $i \bmod 2$ is the $syncBit$ of the fileMsg
- the sender's $MsgQueue$ contains *AckMsgs*, which are simply a $syncBit \in \{0,1\}$ (so that formally an ackMsg and its $syncBit$ are the same).

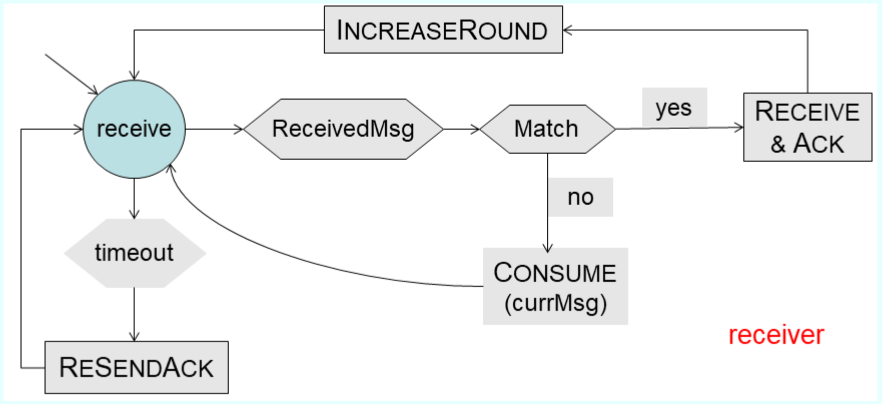
$ReceivedMsg$ **iff** $MsgQueue \neq [\,]$ -- mailbox not empty

$currMsg = head(MsgQueue)$ -- msgs arrive in order of sending

$\textsc{Consume}(msg) = \textsc{Delete}(msg, MsgQueue)$

$Match$ **iff** $syncBit(currMsg) = currRound \bmod 2$

Adding data to AltBit control flow: receiver actions



$\textsc{Receive\&Ack} =$

$\quad \textsc{StoreFile} \quad \textsc{SendAck} \quad \textsc{Consume}(currMsg)$

$\textsc{ReSendAck} =$ -- receiver is round-ahead of sender

$\quad \textsc{Send}(\mathit{flip}(currRound \bmod 2), \textbf{to}\ sender)$ -- previous sync bit

where $\textsc{SendAck} = \textsc{Send}(syncBit(currMsg), \textbf{to}\ sender)$

$\qquad \textsc{StoreFile} = (G(currRound) := \mathit{file}(currMsg))$

Initially, at receiver, $currRound = 1$, $mode = receive$, no msgs around.

Remark on timing

We keep the timing abstract by treating $timeout$ as a monitored location with appropriate assumptions.

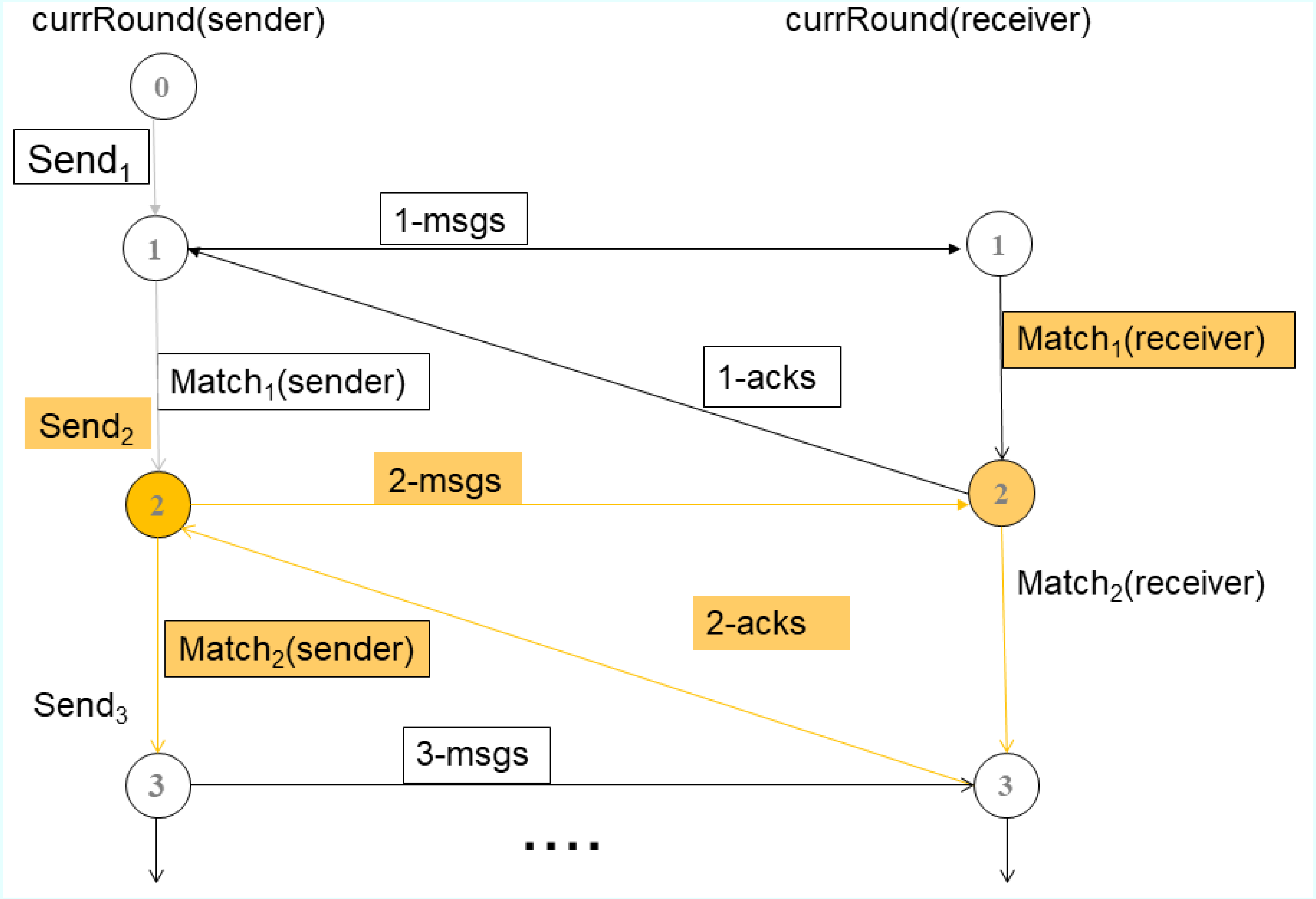
For example, one has to assume that

- $timeout$ at the receiver is initialized in such a way that $\textsc{ReSendAck}$ can happen only after the first $\textsc{Receive\&Ack}$
  - one way to achieve this is to initialize the receiver's $timer$ by ∞

For more details on timing assumptions see AsmBook Ch.6.3.1.

Correctness of AltBit file transfer: eventually $F = G$

Easily proved by induction on $currRound$ phases in AltBit runs:[2]



[2] Figure © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission

Improving AltBit protocol by 'sliding window' technique

Idea: get rid of no-msg-overtaking assumption as follows:

- replace single file transfer rounds by (re)sending, in any order, multiple files $F(i)$ and the corresponding acks
- using as $syncBit$, instead of $i \bmod 2$, directly the indeces i of files $F(i)$
- indeces represented as members of a dynamically updated sliding *window*, one at the sender, one at the receiver
  - an interval $[low, high]$ of indeces of the files whose transfer is still open
    - initially $window$ is empty ($low = 1, high = 0$), at both $sender$ and $receiver$

Characteristics of ‘sliding windows’

- *sender* is refined to perform StartNextFileTransfer
  - where IncreaseRound becomes IncreaseWindow at the window’s right end, namely $high := high + 1$

  only if **not** $FullWindow$
  - where $FullWindow$ **iff** $high - low + 1 = maxWinSize$
- sender must RecordAcks for files that have been received; each *Acknowledgement of low triggers* ReduceWindow, once, at its left end, namely by $low := low + 1$
- the *receiver*’s window ‘follows’ the sender’s window
  - maintaining $high_{receiver} \leq high_{sender}$

  so that upon receiving a file for the first time, if the file index i is larger than the right end $high$ of the *receiver*’s window, then
  - it triggers ‘a round increase’ by SlideWindow at the right end, by setting $high := i$ (and adapting the left end low correspondingly)

Refining sender actions for SLIDINGWINDOW

STARTNEXTFILETRANSFER guarded by **if not** $\mathit{FullWindow}$:

```
STARTNEXTFILETRANSFER =
  if $high < n$ then                              -- refining $currRound$ by $high$
    $high := high + 1$               -- INCREASEWINDOW at the right end
                                                  -- refining INCREASEROUND
    SEND(($F(high+1), high+1$), to $receiver$)
                                       -- refining $nextFile$, $nextSyncBit$
```

Resending only at the left window end:

RESENDFILE = SEND((*F(low),low*), **to** $receiver$)

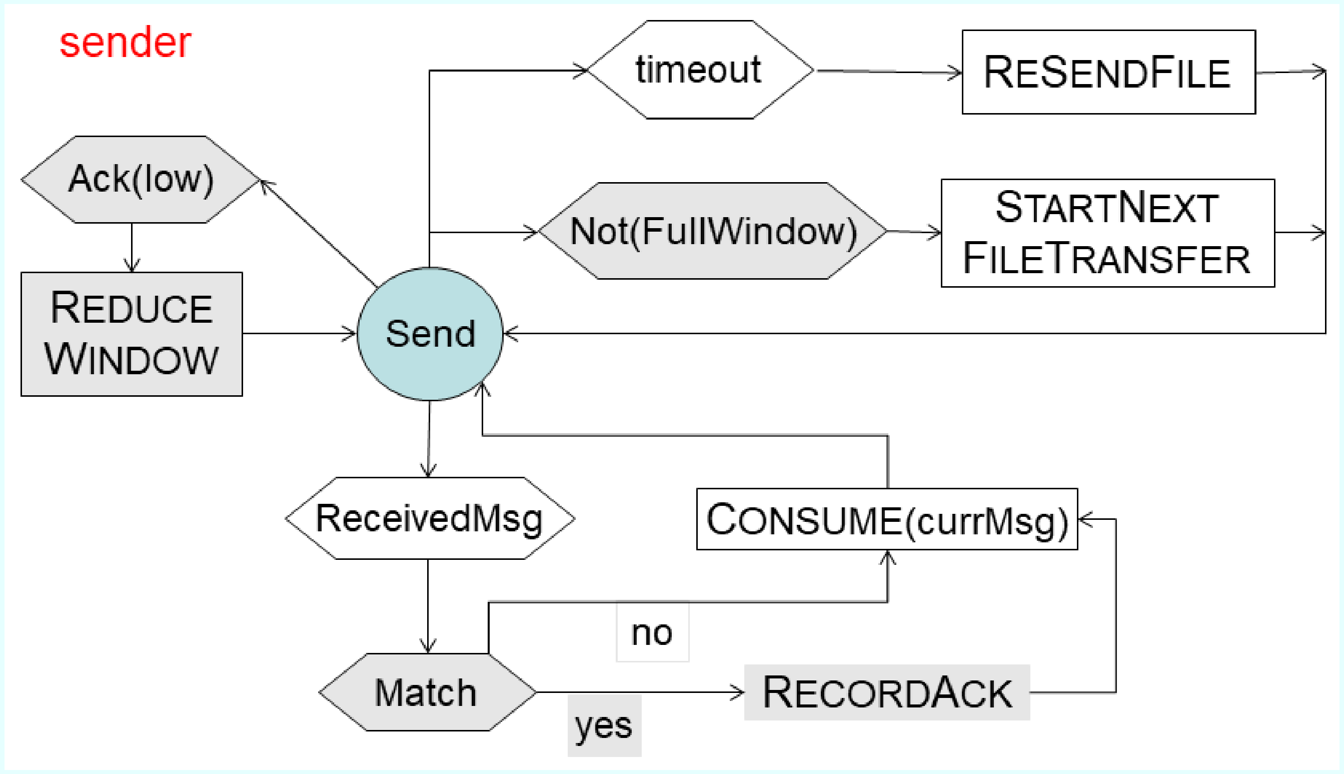
Refining sender's 'closing a round' for SlidingWindow

- sender performs ReduceWindow, at the left end, namely by increasing low when an ack msg for the receipt of the file with index low has been $Received$.
- Matching ack msgs can arrive from the receiver in any order and with $syncBit > low$. Therefore a RecordAck component is needed.

```
ReduceWindow =                                  -- guarded by $Ack(low)$
  $low := low + 1$
  $Ack(low) := false$                           -- shift exactly once per acknowledged file
RecordAck =                                     -- called only in case of $Match$
  if $syncBit(currMsg) > low$                   -- must later ReduceWindow
    then $Ack(syncBit(currMsg)) := 1$
    else ReduceWindow                           -- ack received for $F(low)$
```

Initially $Ack(i) = false$ for each i

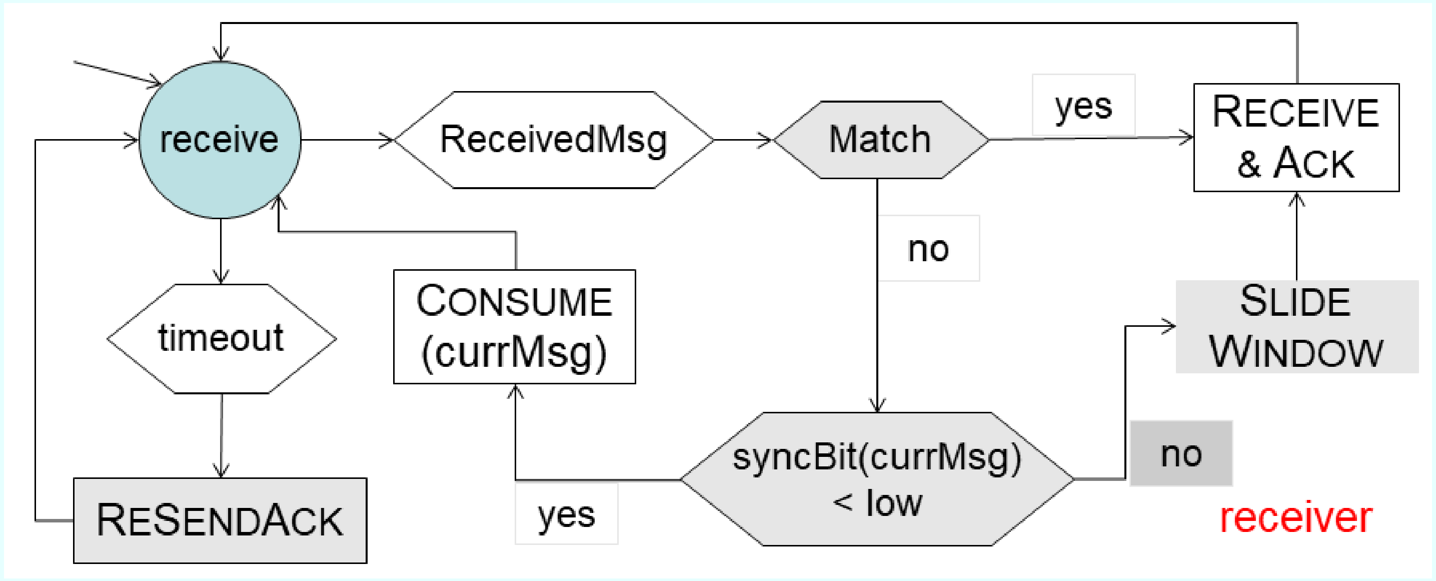
Program of refined SLIDINGWINDOW sender actions



$Match$ **iff** $low \leq syncBit(currMsg) \leq high$ -- $syncBit \in window$[3]

[3] Figure © 2016 Springer-Verlag, reused with permission.

Program of refined SlidingWindow receiver actions



$\text{ReSendAck} = \text{Send}(low, \textbf{to } sender)$

$\text{SlideWindow} =$

$\quad \textbf{let } i = syncBit(currMsg)$

$\qquad high := i$

$\qquad low := max\{1, i - (maxWinSize - 1)\}$

The other components (in white) are taken unchanged from AltBit

Some properties in states of SLIDINGWINDOW runs

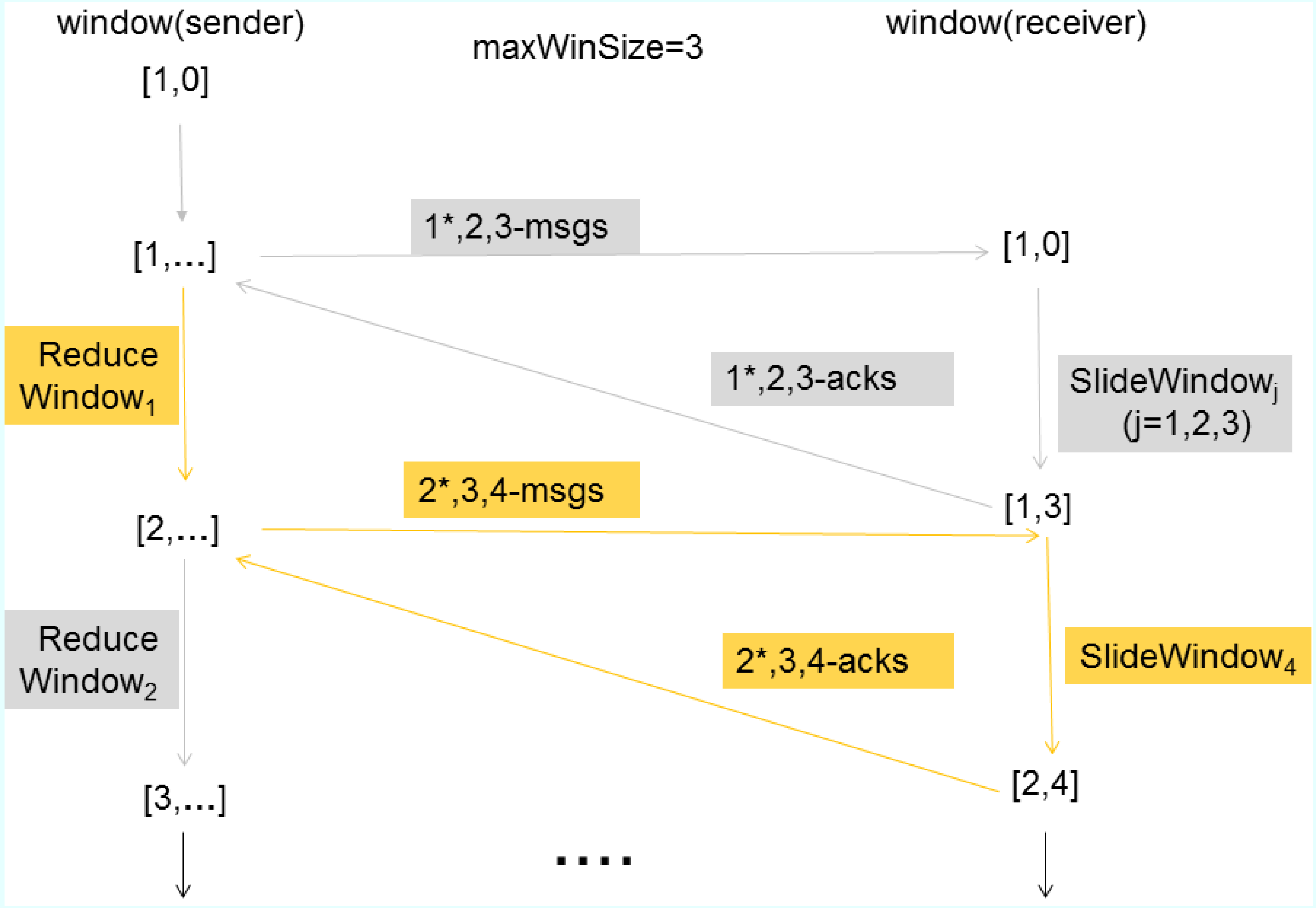
- call *files whose transfer is still open* (in a state) those files $F(i)$ which (in this state) have been sent but
  - either the sender has not yet received an Acknowledgement for $F(i)$
  - or $Ack(i) = true$ but
    - either $low < i$
    - or $low = i$ and the REDUCEWINDOW rule has not yet been executed
- indices i of files whose transfer is still open are in the sender's window, i.e. $low_{sender} \leq i \leq high_{sender}$
- $high_{receiver} \leq high_{sender}$ holds in every state
  - since $high_{receiver}$ assumes only values assumed before by $high_{sender}$

Correctness of SlideWindow

- If in a state of a SlidingWindow run, a file msg $(F(i), i)$ is $Received$ which does not $Match$ and has $i \geq low$, then:
  - $F(i)$ is a file whose transfer is still open
  - thus $high_{receiver} < i \leq high_{sender}$ and $high_{receiver}$ is updated to i and an ack msg for the receipt of $F(i)$ is sent
  - at most $maxWinSize - 1$ elements of the receiver's window are index of files whose transfer is still open
    - because there can be at most $maxWinSize$ files whose transfer is still open: once the sender has a $FullWindow$, no not-yet-sent file is sent before the sender performs a ReduceWindow step
- Therefore, low can be updated to $max\{1, i - (maxWinSize - 1)\}$
  - because files with index $j < max\{1, i - (maxWinSize - 1)\}$ are files whose transfer is not any more open

Correctness of SLIDINGWINDOWS runs

Easily proved by induction on phases in SLIDINGWINDOWS runs:[4]



[4] Figure © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission

Optimizing $\textsc{SlidingWindow}$ by a component refinement

- A file msg $(F(i), i)$ which $Match$es may be Received multiple times – namely by resending it until one of the acks is received by the sender.
- Each time msg $(F(i), i)$ is Received, $F(i)$ is stored as part of $\textsc{Receive\&Ack}$.
- To avoid this repetition of file copying, it suffices to refine the $\textsc{Receive\&Ack}$ component $\textsc{StoreFile}$ as follows:

```
$\textsc{StoreFile} =$
  **if** $G(syncBit(currMsg)) = \mathbf{undef}$
    **then** $G(syncBit(currMsg)) := file(currMsg)$
```

Two alternative protocol models

Among numerous other protocol models, expressed in alternative specification languages, we mention two examples, which are representative for a large variety of approaches.

- A complete formalization using the B method, focussed on a *machine-verification of protocol properties* of interest, including the protocol correctness, can be found in:
  - J.-R. Abrial and L. Mussat: Specification and design of a transmission protocol by successive refinements using B.
    - In: Mathematical Methods in Program Development (eds. M. Broy and B. Schieder), Springer 1996

B models are formulae of first order logic enriched by set theory concepts. They use the same abstract notion of state as the ASM method. Defining a model and mechanical property verification are intimately linked, but separated in the ASM method.

Petri net models for AltBit and SlidingWindows

- Petri nets use a *graphical specification language* that is more complex than CSDs, the asynchronous Control State Diagrams used above
  - CSDs are a generalization of Finite State Machine (FSM) flowcharts
    - by permitting to use data and operations of arbitrary level of abstraction, directly, without encoding
      - avoiding the token straitjacket of Petri nets
  - coming with a simple definition of their semantics, being a natural visualization of control state ASMs all of which rules are of the form

```
**if** $mode = i$ **and** $Cond$ **then**
    $M$
    $mode := j$
```

We invite the reader to directly compare the AltBit and SlidingWindows ASMs with their equivalent Petri net definition below, which is copied from the book 'Elements of Distributed Algorithms' (by W. Reisig), Springer-Verlag, 1998.

Petri net definition for AltBit

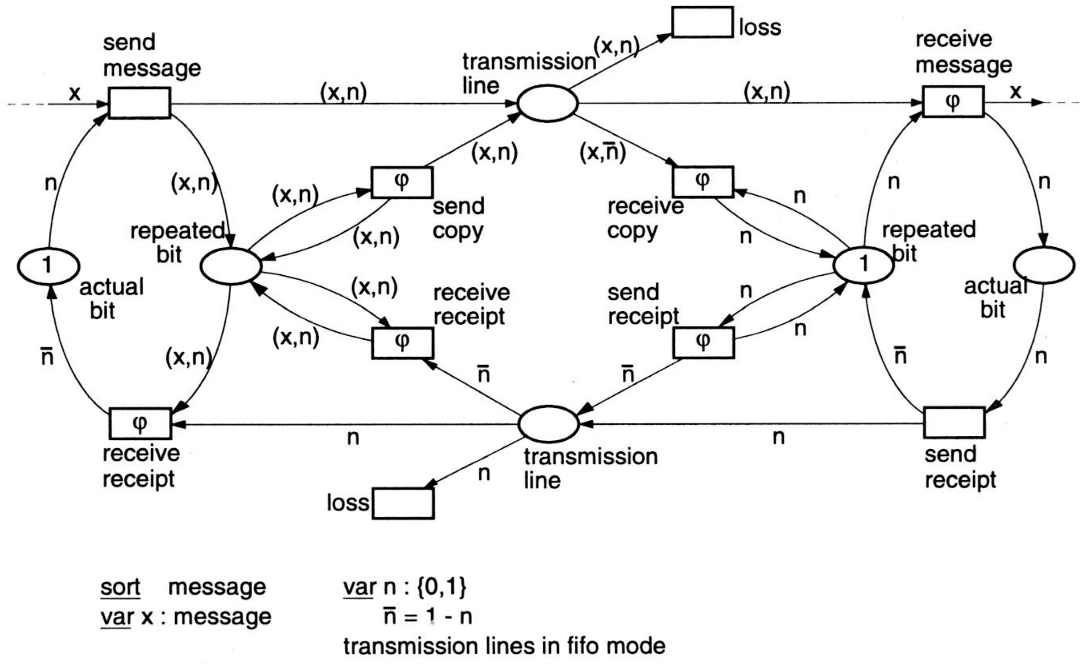


Figure 27.7. The alternating bit protocol

This spec defines a global model of both sender and receiver actions.[5]

[5] Figure © 1998 Springer-Verlag Berlin Heidelberg, reprinted with permission

Petri net definition for SLIDINGWINDOWS

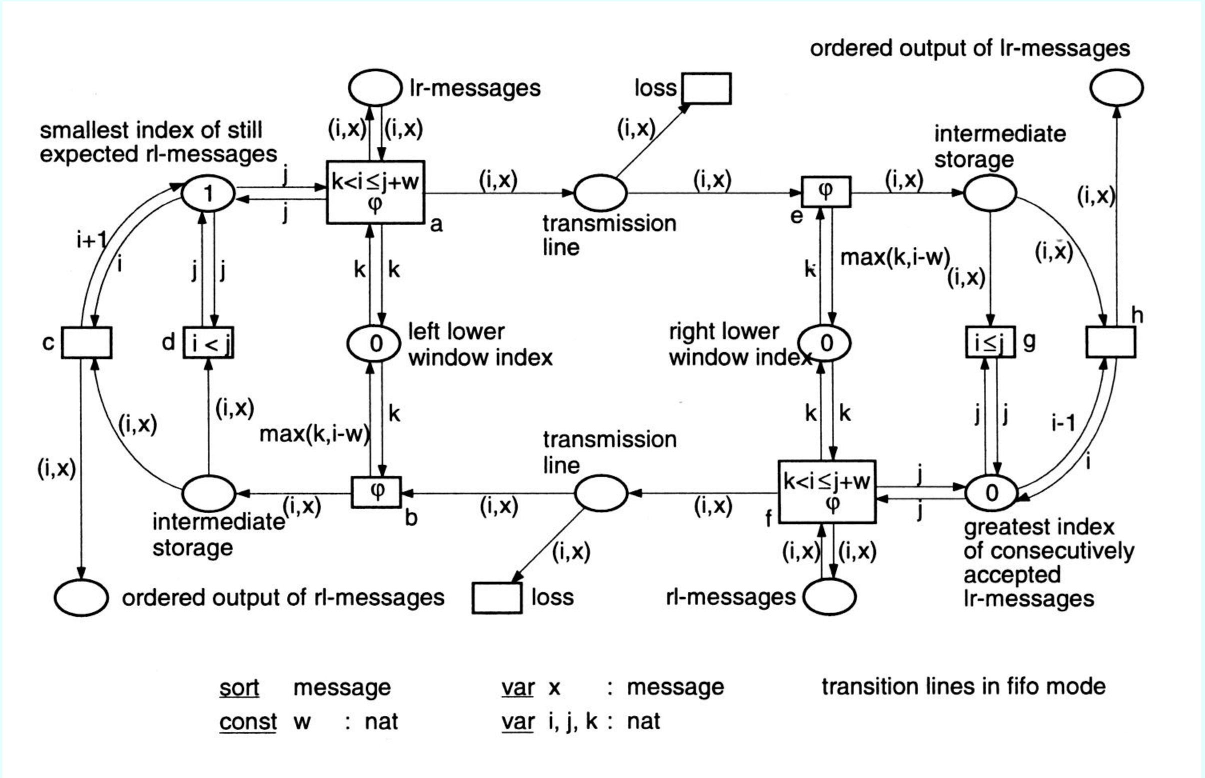


Figure 28.1. Balanced sliding window protocol with unbounded indices

Further direct comparisons of Petri net[6] and equivalent ASM models can be found in E. Börger: Modeling distributed algorithms by Abstract State Machines compared to Petri Nets. Springer LNCS 9675, 3-34 (2016)

[6] Figure © 1998 Springer-Verlag Berlin Heidelberg, reprinted with permission

Related ASM models for the protocol

- J. Huggins: Kermit specification and verification. In: Specification and Validation methods (ed. E. Börger), OUP 1995, 247-293.
  - This paper specifies and analyses the full Kermit protocol, reusing variations of AltBit and SlidingWindow ASMs.
  - A Kermit ASM template, from which AltBit and SlidingWindows can be instantiated, can be found in Ch. 6.3.1. of: E. Börger and R. Stärk: Abstract State Machines. Springer 2003.
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018 http://modelingbook.informatik.uni-ulm.de
  - Fig.4.8 in this book generalizes the sender component of AltBit.

Copyright Notice