05_03_Stabilization

1. Background

1.1. Faults

computer systems can exhibit **faults:** for example

- memory may be corrupted
- messages may not be (correctly) received

Faults can be permanent or transient

- a processor may restart in some unknown state
- a processor may crash completely

1.2. Dealing with faults

1.2.1. Explicitly

- **detect and correct** errors in data
- **resend** a message when no acknowledgment is received
- use a **consensus protocol**

1.2.2. Implicitly

have algorithms **converge to a correct state** as a side effect of their operation

1.3. Stabilizing algorithm

A **stabilizing algorithm** is to

- deals with transient faults in an implicit way by **converging to the desired behavior**
- can be **started in an arbitrary state** without the need for a global (synchronized) system initialization
- can deal **with system reconfigurations** without the need for a
  complete system reset

Also is called **self-stabilizing algorithms**

Stabilizing algorithms are used for **non-terminating control and coordination tasks**:

1.4. Real-word example

- an orchestra plays outdoors without a conductor
- the pages of the score can be turned by the wind
- the players **try to (re-)synchronize** their playing
- each player **restarts at the lowest-numbered page** he hears being played from around him
- the score should be long enough

2. Pre-Definitions

- in order to describe "functioning correctly", we define **legal and illegal configuration** by **predicate**
- Apredicate on system configurations (a “property”) is called **stable** if once **it holds and no faults occur, it continues to hold.** Stable predicates have also been called **closed predicate**

2.1. Q-stabilize

Let **P and Q** be **two stable predicates** on the configurations of a system **S:**

An algorithm **Q-stabilizes from P in S** if from any configuration in which P holds, within a finite number of steps, S is in a configuation in which Q holds

- it is sometimes said that the algorithm **converges from P to Q**

2.2. Model of communication

two types:

- shared-memory model
- message-passing model

2.2.1. Shared-Memory Model

- in addition to having **local registers**, processors communicate through sets of **non-local registers**
- each register has associated sets of processors that can read and that can write it
- read/write atomicity
- composite atomicity

2.2.2. Message-passing model

- if processor **p is connected by a datalink to processor q**,there is a **message buffer** B_{pq} containing the set of messages **sent by p but not yet received by q**
- if datalink **FIFO**, then that is a **queue**

2.3. Demons

Two types

- central demon
- distributed demon

2.3.1. Cetnral demon

- this demon **picks one** process at a time to make a step.
- Only when this process has **completed** its step, the demon **picks anothe**r process to make a step.

Always we assume the demon is **fair**, that is: in **every infinite execution** of the system, every process is picked **infinitely often**

2.3.2. Dsitributed demon

- The demon picks **a set of processes** to make a step.
- Then, all processes in this set make their step **in a synchronous fashion**
- Only **after all of the processes in the set have completed** their step does the demon pick another set.

Always we assume the demon is **fair**, that is: in **every infinite execution** of the system, every process belongs to the set of activated processes **infinitely often**

3. Stabilizing mutual Exclusion Algorithms

3.1. Dijkstra's stabilizing mutual-exclusion algorithm for a unidirectional ring in the shared-memory model with a central or a distributed demon

Dijkstra's stabilizing mutual-exclusion algorithm for a **unidirectional ring** in the **shared-memory model** with a **central or a distributed demon**

<u>Assumption</u>

- Assume a **unidirectional ring** of N processes
- Assume a **shared-memory model** with **composite atomicity (**reading and writing in one atomic step**)**

<u>Model and Configuration</u>

Legal Configuration

$$(\mathrm{v}_0 = \mathrm{v}_{N-1}) \quad \text{XOR} \quad (\#\{n \mid 0 < n < N, \mathrm{v}_n \neq \mathrm{v}_{n-1}\} = 1)$$

<u>Preparation (Structure)</u>

Process P_i maintains only one variable v_i, modulo some K

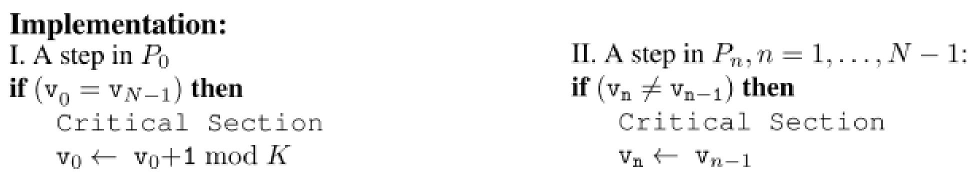
- central demon: K $\geq$ N-1
- distributed demon K $\geq$ N

<u>Process</u>

In a singel step, a process P_i reads v_{i-1} and v_i, compares them and assigns a new value to v_i:

- if i≠0 and $v_i \neq v_{i-1}, v_i \leftarrow v_{i-1}$
- if P_0 finds $v_0 = v_{N-1}, v_0 \leftarrow v_0 + 1$

Implementation



Implementation:

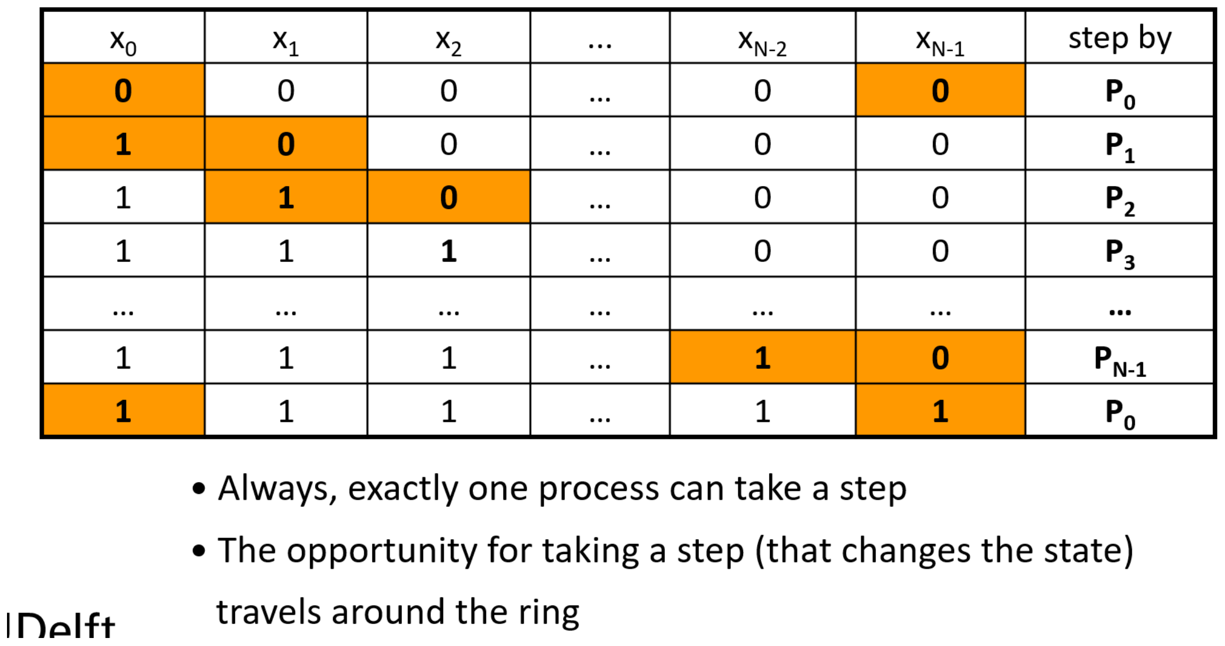
```
I. A step in $P_0$
if ($v_0 = v_{N-1}$) then
    Critical Section
    $v_0 \leftarrow v_0 + 1 \bmod K$
```

```
II. A step in $P_n, n = 1, \ldots, N-1$:
if ($v_n \neq v_{n-1}$) then
    Critical Section
    $v_n \leftarrow v_{n-1}$
```

Correctness

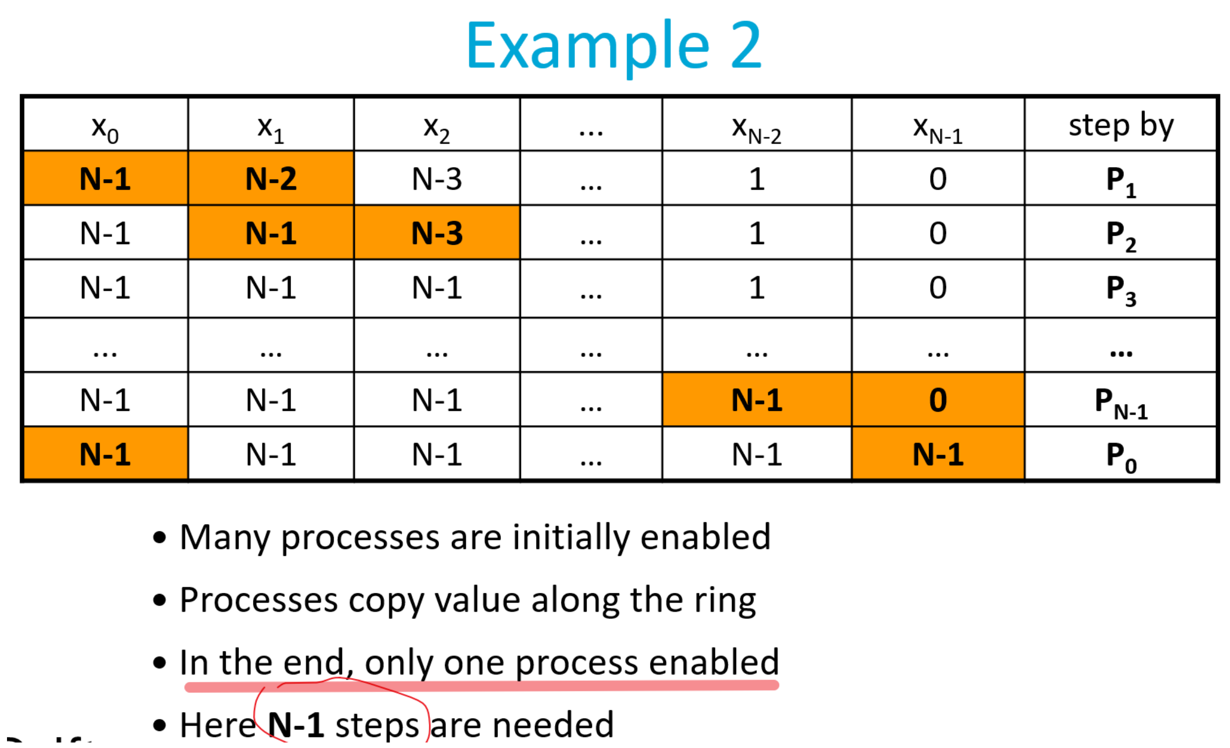
- The purpose of this kind of algorithm is: in finit steps, with disttibuted or central fair demon, the system will come to a state where it is satisfies its stable legal configuration

Examples



x_0	x_1	x_2	…	x_{N-2}	x_{N-1}	step by
0	0	0	…	0	**0**	**P_0**
1	**0**	0	…	0	0	**P_1**
1	**1**	**0**	…	0	0	**P_2**
1	1	**1**	…	0	0	**P_3**
…	…	…	…	…	…	**…**
1	1	1	…	**1**	**0**	**P_{N-1}**
1	1	1	…	1	**1**	**P_0**

- Always, exactly one process can take a step
- The opportunity for taking a step (that changes the state) travels around the ring



Example 2

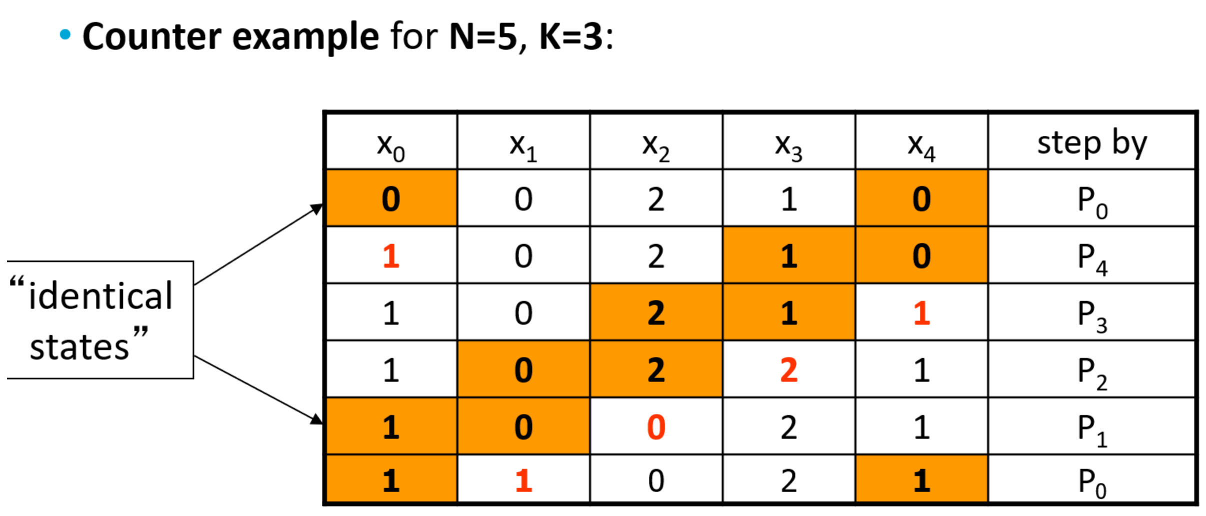
x_0	x_1	x_2	…	x_{N-2}	x_{N-1}	step by
N-1	**N-2**	N-3	…	1	0	**P_1**
N-1	**N-1**	**N-3**	…	1	0	**P_2**
N-1	N-1	N-1	…	1	0	**P_3**
…	…	…	…	…	…	**…**
N-1	N-1	N-1	…	**N-1**	**0**	**P_{N-1}**
N-1	N-1	N-1	…	N-1	**N-1**	**P_0**

- Many processes are initially enabled
- Processes copy value along the ring
- In the end, only one process enabled
- Here **N-1** steps are needed

In example 2, the maximum number of steps before only one process is enabled is : (from $X_n - 1$ back propogate)

<u>CounterExamples</u>

For central demon

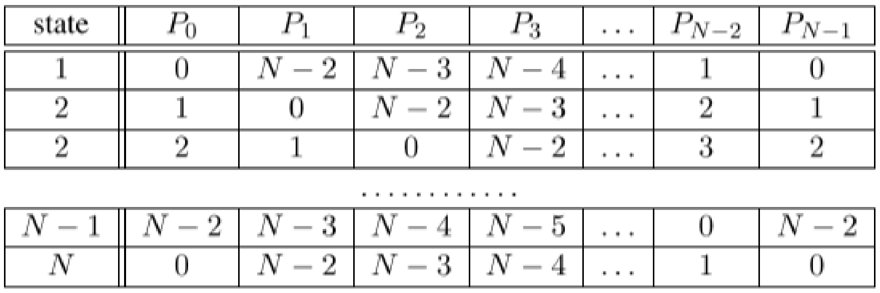


- **Counter example** for **N=5**, **K=3**:

"identical states"

x_0	x_1	x_2	x_3	x_4	step by
0	0	2	1	**0**	P_0
1	0	2	**1**	**0**	P_4
1	0	**2**	**1**	1	P_3
1	**0**	**2**	2	1	P_2
1	**0**	0	2	1	P_1
1	1	0	2	**1**	P_0

For distributed demon



state	P_0	P_1	P_2	P_3	...	P_{N-2}	P_{N-1}
1	0	$N-2$	$N-3$	$N-4$	...	1	0
2	1	0	$N-2$	$N-3$	...	2	1
2	2	1	0	$N-2$	...	3	2
....							
$N-1$	$N-2$	$N-3$	$N-4$	$N-5$	...	0	$N-2$
N	0	$N-2$	$N-3$	$N-4$	...	1	0

Table 5.2: Successive states in the system in Example 5.25.

Remarks

- If the number of processes in the ring is **not a prime**, then **a special process** is needed
- shared memory with read/write atomicity (in one atomic communication step only reading or writing)

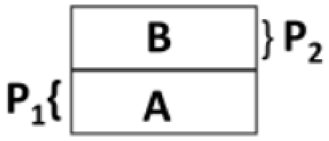
4. Fair Protocal Composition

4.1. Idea

split up a stabilizing algorithm into two or more simpler stabilizing algorithms ("levels")

4.2. Usage

the possibility of algorithm conversions between different system models, we divide it into layers



4.3. Example

a stabilizing datalink protocol in an asynchronous message-passing system:

5. Stabilizing Datalink Algorithms

5.1. Background

- In order to derive **stabilizing algorithms for message-passing systems**
- From Section 4, we can divide the process into **two steps**
  - design a stabilizing algorithm **for message passing along a datalink**
  - use ashared-memory stabilizing algorithm for the original problem

5.2. A stabilizing stop-and-wait datalink algorithm

Basic Ideas

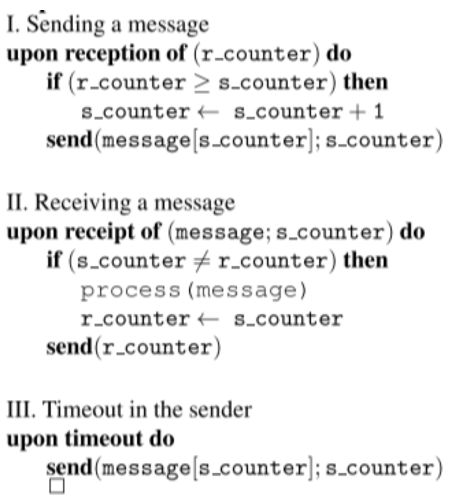
for every message the sender **waits for an acknowledgment** before sending the next message

Preparation (Structure)

- Sender and receiver maintain **counters: s_counter, r_counter**
- Message type:
  - From Sender: [message[s_counter]; s_counter]
  - From Receiver: only r_counter

Implementation & understanding

- By using the counter, a crashed receiver can catch up with the newest state of sender.
- A crashed sender may update its counter even though it lose the infromation of the latest message it has sent
- Although some fault may not be revised, the timeout mechanism do help



```
I. Sending a message
upon reception of (r_counter) do
    if (r_counter $\geq$ s_counter) then
        s_counter $\leftarrow$ s_counter + 1
    send(message[s_counter]; s_counter)

II. Receiving a message
upon receipt of (message; s_counter) do
    if (s_counter $\neq$ r_counter) then
        process(message)
        r_counter $\leftarrow$ s_counter
    send(r_counter)

III. Timeout in the sender
upon timeout do
    send(message[s_counter]; s_counter)
    □
```

Correctness illustration

- system is in **a legal state** if **all counters** (in sender, in receiver, and in messages) **are equal**

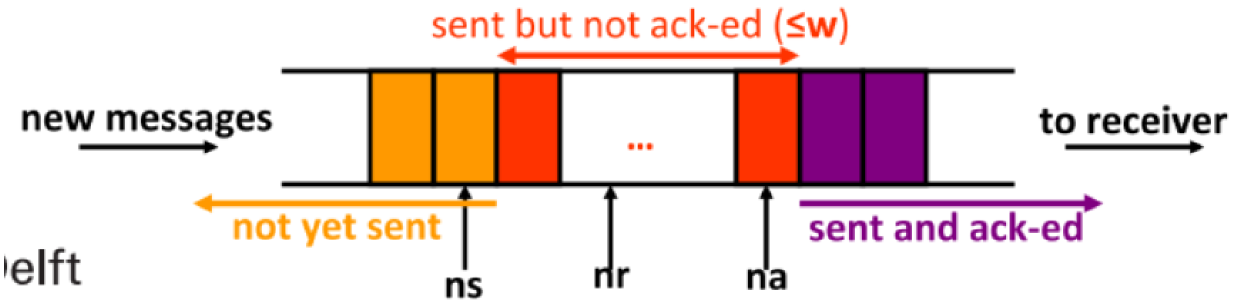
- the sender will always **introduce new (higher) values** for its counter into the system
- then, at some later time, **all counters will be equal**
- resembles missing-label proof of Dijkstra's algorithm

5.3. A non-stabilizing sliding-window datalink algorithm

Basic Ideas (Process)

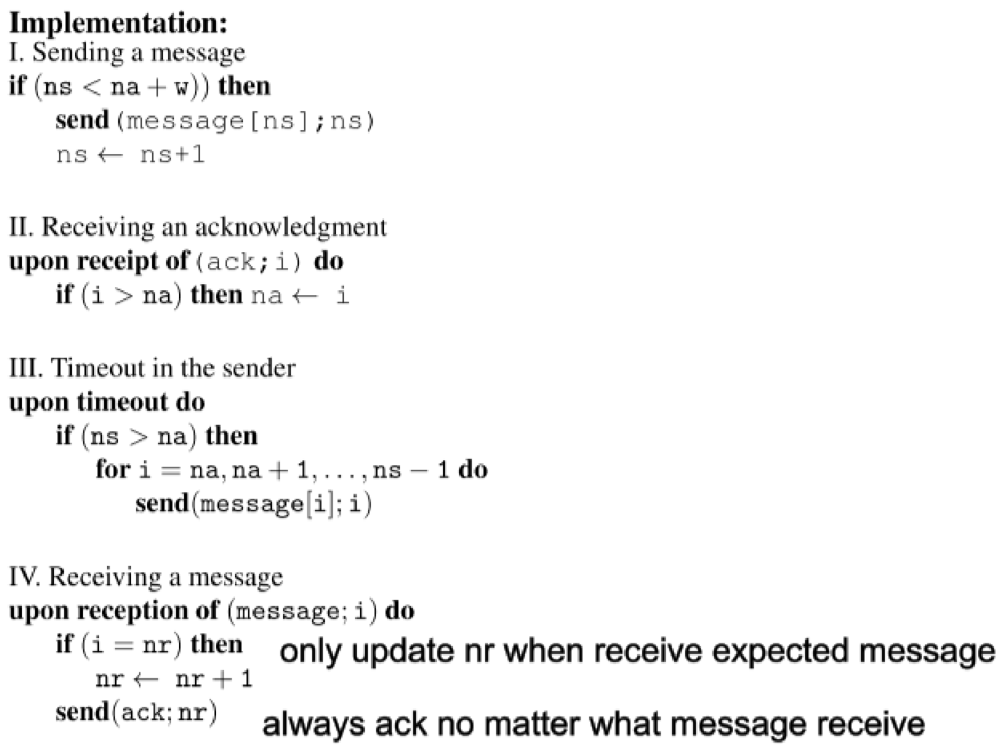
- When R receives a message with number **n**, it:
    1. first sets **nr** to n+1
    2. then sends a message **(ack;nr)**, (ack,i) **acknowledges the receipt** of messages 1,2,…,i-1
- Upon reception of (ack,i), S sets **na**=i
- When timed-out, resend the messages in window

Preparation (Structure)



- Variables in **Sender S**:
    - **w**: the window size
    - **ns**: the number of the next message to send
    - **na**: the number of the first non-acknowledged message
- Variables in the **Receiver R:**
    - **nr**: the number of the next message to be received
- Message:
    - Message sent by the sender carry the message number
    - acknowledge message (ack, nr)

Implementation



Implementation:

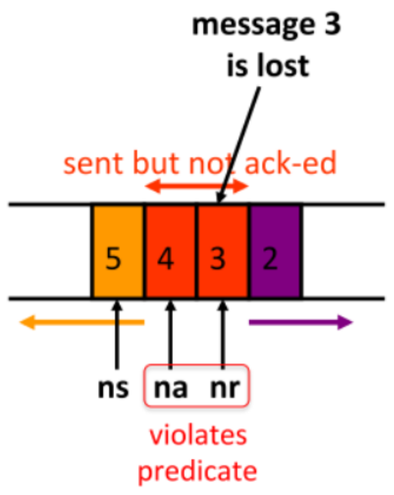
```
I. Sending a message
if $(ns < na + w)$ then
    send (message[ns];ns)
    ns ← ns+1

II. Receiving an acknowledgment
upon receipt of (ack;i) do
    if $(i > na)$ then na ← i

III. Timeout in the sender
upon timeout do
    if $(ns > na)$ then
        for $i = na, na + 1, \ldots, ns - 1$ do
            send(message[i]; i)

IV. Receiving a message
upon reception of (message; i) do
    if $(i = nr)$ then      only update nr when receive expected message
        $nr \leftarrow nr + 1$
    send(ack; nr)       always ack no matter what message receive
```

<u>Correctness</u>

This Algorithm is not stable

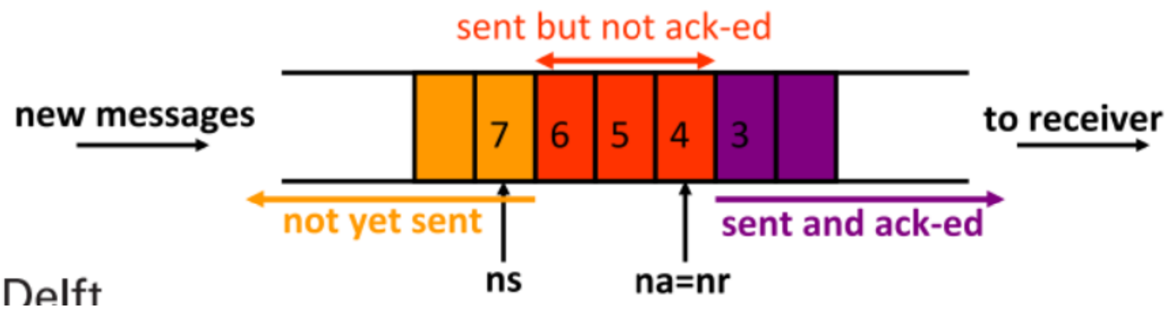
- First the stable predicated P:
  - $((na \leq nr)$ and $(nr \leq ns)$ and $(ns \leq na + w))$
  - for each $(message; i)$ in channel SR $(i < ns)$
  - for each $(ack; i)$ in channel RS $(i \leq nr)$
- Explanation of P:
  - no message acknowledged that has not yet been received; no message received that has not been sent; not more than w outstanding messages
  - no message on channel SR that has not been sent
  - no message acknowledged that has not been received
- A problem that can occur with this algorithm is when the sender thinks that the receiver has received more messages than it in reality has (**na>nr**). It can happen for example the memory crashed and the na is wrongly updated, an example is as follows

- This protocol does **not stabilize**
- Assume **w=1**
- Consider the state
  - **nr=3** (**R** is still waiting for message 3)
  - **na=4** (**S** waits for the ack for message 4)
  - **ns=5** (**S** wants to send message 5)
  - both channels empty



Examples 1

- Let **w=3**
- Let
  - **ns=7**
  - **na=nr=4**
- Sender has to wait because window is full: **ns=na+w**
- Channel SR contains messages 4,5,6



5.4. A stabilizing sliding-window datalink algorithm

Basic Idea

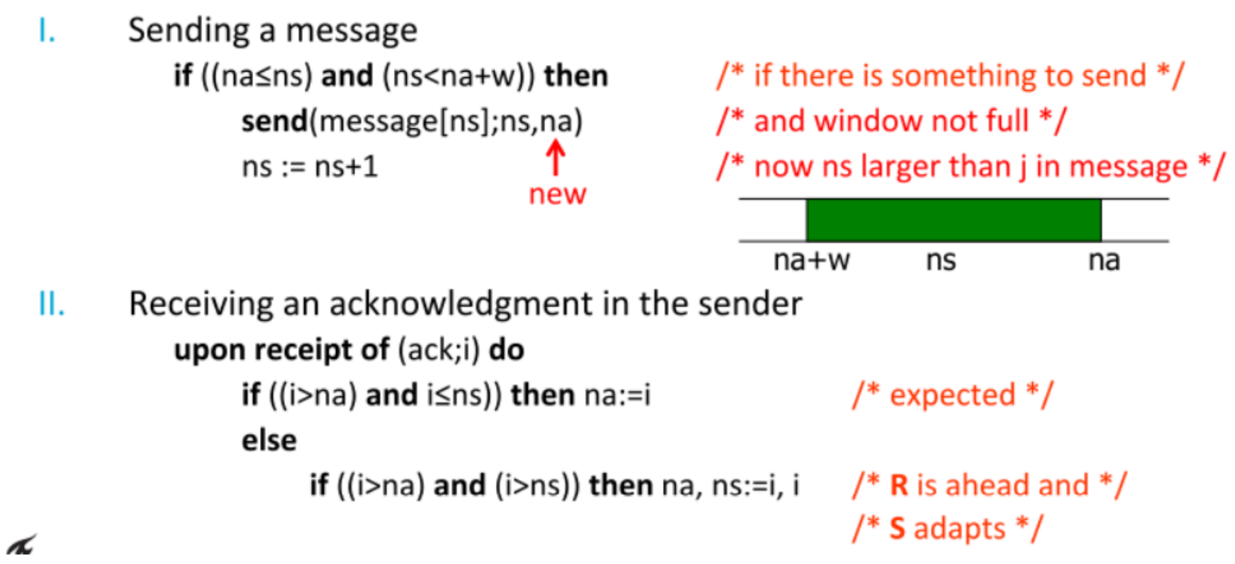
- S catches up with R, when R is ahead of S:
  - when S receives (ack,i) with **i>ns**, it sets **na** and **ns** to i (jump ahead)

- R catches up with S, when S is ahead of R:
    - when R receives message[i,j] with **j>nr**, it sets **nr** to j (so R jumps ahead, and misses messages)

Preparation (Structure)

- Variables in **Sender S**:
    - **w**: the window size
    - **ns**: the number of the next message to send
    - **na**: the number of the first non-acknowledged message
- Variables in the **Receiver R:**
    - **nr**: the number of the next message to be received
- Message:
    - message sent by sender carry two sequence numbers i and j
        - i as before
        - j is the value of **na** at the time of sending the message (with j, **R gets to know what S thinks R has received**)
    - acknowledge message (ack,i)

Implementation



I. Sending a message

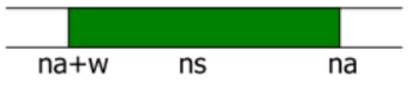
```
if ((na≤ns) and (ns<na+w)) then        /* if there is something to send */
    send(message[ns];ns,na)            /* and window not full */
    ns := ns+1                         /* now ns larger than j in message */
                  ↑
                  new
```

II. Receiving an acknowledgment in the sender

```
upon receipt of (ack;i) do
    if ((i>na) and i≤ns)) then na:=i           /* expected */
    else
        if ((i>na) and (i>ns)) then na, ns:=i, i    /* R is ahead and */
                                                    /* S adapts */
```

III. Timeout in the sender

```
upon timeout do
    if (na≠ns) then                  /* otherwise empty window */
        if (na>ns) then ns:=na        /* wrong situation: reset (window empty) */
        if (ns>na+w) then na:=ns      /* wrong situation: reset (window empty)*/
        for i=na,na+1,...,ns-1 do     /* resend unacknowledged messages */
            send(message[i];i,na)     /* all have j < ns */
```



IV. Receiving a message

```
upon reception of (message;i,j) do
    if ((i=nr) and (j≤nr)) then/* expected message and not */
        nr:=nr+1                 /* too much acked */
    else
        if (j>nr) then nr:=j     /* adapt to S and jump ahead */
    send(ack,nr)                 /* send an ack anyway */
```

Correctness

- the predict P
    - $((na \leq nr)$ and $(nr \leq ns)$ and $(ns \leq na + w))$
    - for each $(message; i)$ in channel SR ($(i < ns)$ and $(j \leq nr)$ and $(j < ns))$
    - for each $(ack; i)$ in channel RS $(i \leq nr)$

Drawbacks

- Cannot Deal with windows size w corrupted
- a processor has to keep track of the identifier of the machine and of the port (or process) number of destination process, and we do not has part to deal with their corruption

5.5. Remarks

One can show that stabilizing datalink protocols:

- do not terminate
- need unbounded counters
- need timeout actions