Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Lecture 10

Self-stabilizing Data-link

Roadmap

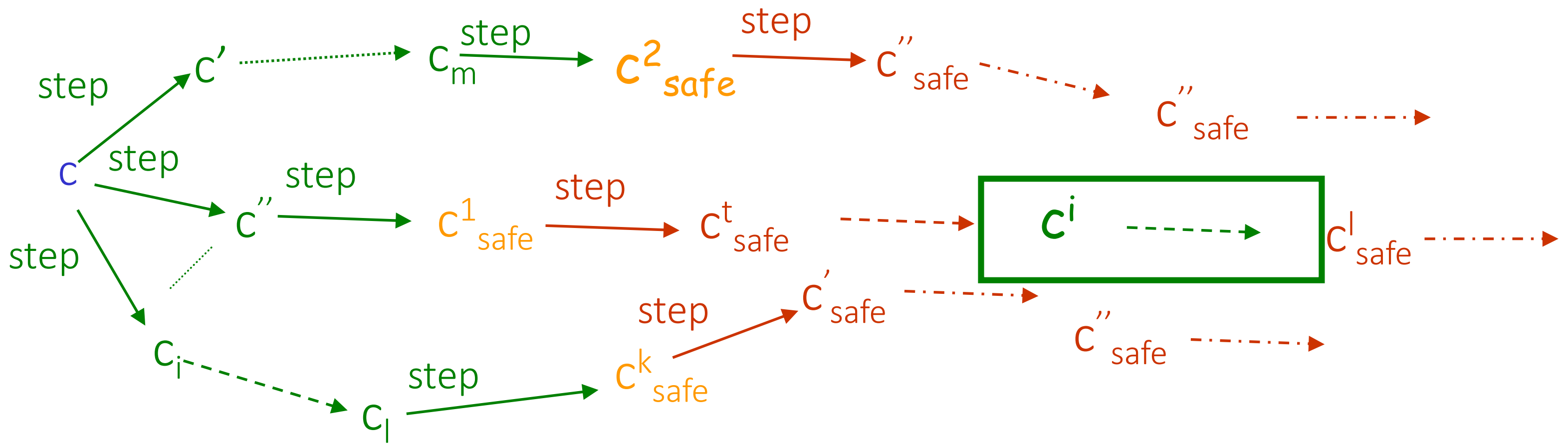
2.10 Pseudo-Self-Stabilization

3.1 Initialization of a Data-Link Algorithm in the Presence of Faults

3.2 Arbitrary Configuration Because of Crashes

4.2 Data-Link Algorithms: Converting Shared Memory to Message Passing

What is Pseudo-Self-Stabilization ?



- The algorithm exhibits a legal behavior; but may deviate from this legal behavior a finite number of times

An Abstract Task

- An abstract task - variables and restrictions on their values
- The token passing abstract task AT for a system of 2 processors; Sender (S) and Receiver (R). S and R have boolean variable $token_S$ and $token_R$
- Given $E = (c_1, a_1, c_2, a_2, \ldots)$ one may consider only the values of $token_S$ and $token_R$ in every configuration c_i to check whether the token-passing task is achieved

Pseudo-self-Stabilization

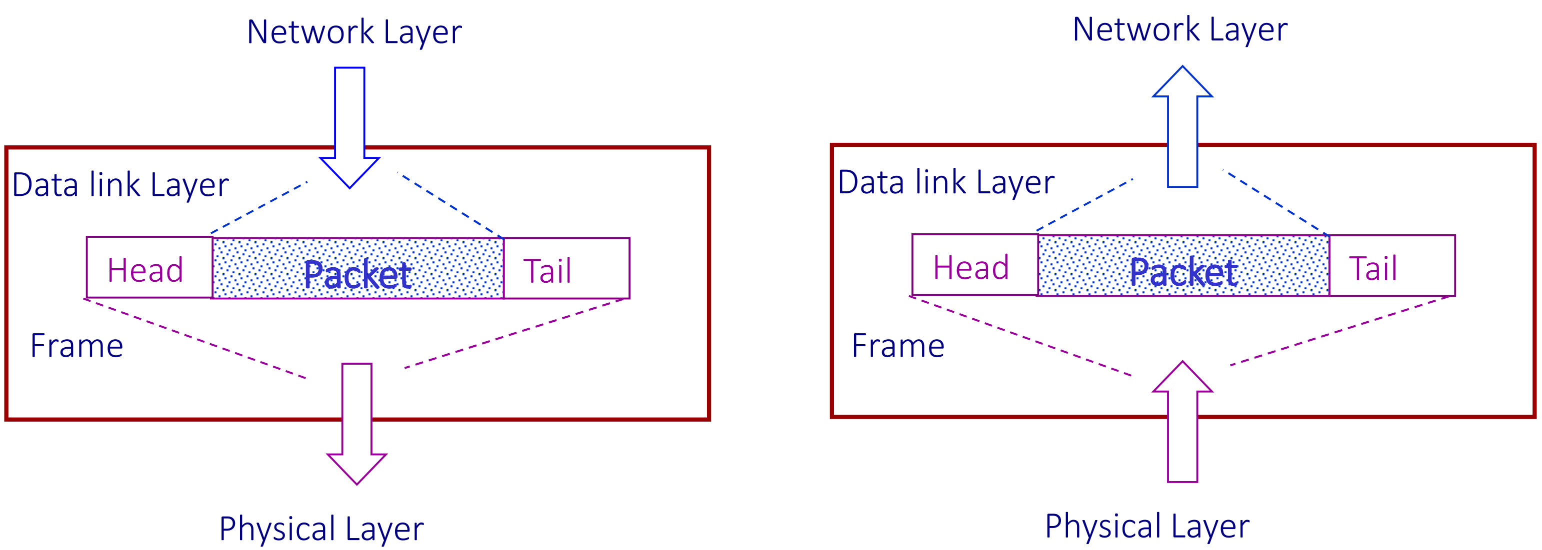
- Denote :
  - $c_i|tkns$ the value of the boolean variables ($token_S$, $token_R$) in c_i
  - $E|tkns$ as ($c_1|tkns$, $c_2|tkns$, $c_3|tkns$, ...)
- We can define AT by $E|tkns$ as follows:
  there is no $c_i|tkns$ for which $token_S=token_R=true$
- It is impossible to define a safe configuration in terms of $c_i|tkns$, since we ignore the state variables of R/S

Pseudo-Self-Stabilization, The Alternating Bit Algorithm

- A data link algorithm used for message transfer over a communication link
- Messages can be lost, since the common communication link is unreliable
- The algorithm uses retransmission of messages to cope with message loss
- frame distinguishes the higher level messages, from the messages that are actually sent (between S and R)

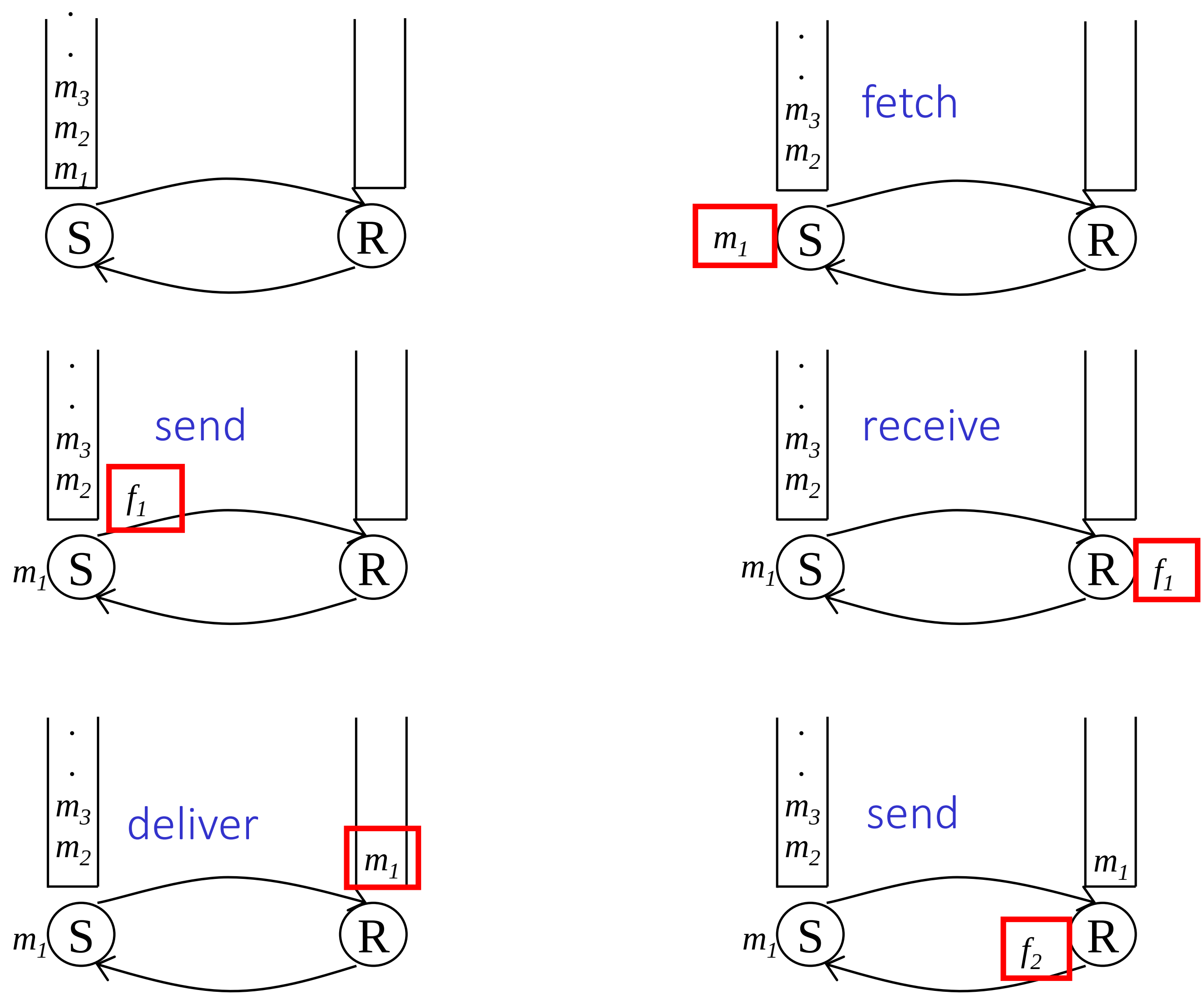
Pseudo-Self-Stabilization, The Data Link Algorithm

- The task of delivering a message is sophisticated, and may cause message corruption or even loss
- The layers involved



Pseudo-Self-Stabilization, The Data Link Algorithm

The flow of a message:



Pseudo-Self-Stabilization, Back to The Alternating Bit Algorithm

- **The abstract task of the algorithm:**
  S has an infinite queue of input messages (im_1,im_2,…) that should be transferred to the receiver in the same order without duplications, reordering or omissions.
  R has an output queue of messages (om_1,om_2,…). The sequence of messages in the output queue should always be the prefix of the sequence of messages in the input queue

The alternating bit algorithm - Sender

```
initialization
begin
    i := 1
    bit_s := 0
    send(⟨bit_s,im_i⟩) (*im_i is fetched*)
end (*end initialization*)
upon a timeout
            send(⟨bit_s,im_i⟩)
upon frame arrival
begin
            receive(FrameBit)
            if FrameBit = bit_s then (*acknowledge arrives*)
               begin
                        bit_s := (bit_s + 1) mod 2
                        i := i + 1
            end
            send(⟨bit_s,im_i⟩) (*im_i is fetched*)
end
```

The alternating bit algorithm - Receiver

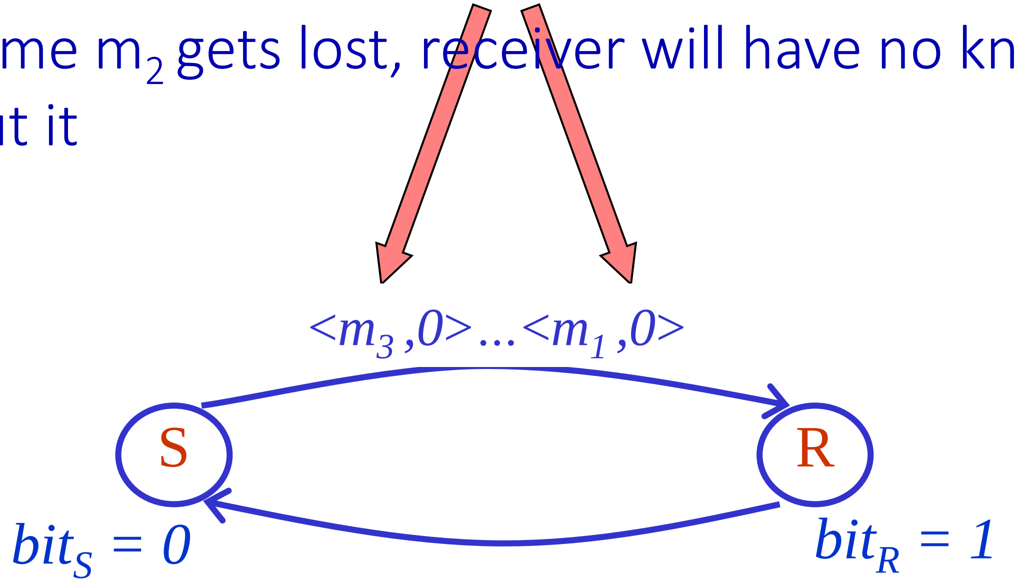
```
initialization
begin
    j := 1
    bit_r := 1
end (*end initialization*)
upon frame arrival
begin
        receive(⟨FrameBit , msg⟩)
        if FrameBit ≠ bit_r then (*a new message arrived*)
            begin
                bit_r := FrameBit
                j := j + 1
                om_j := msg (*om_j is delivered*)
        end
        send(bit_r)
end
```

Pseudo-Self-Stabilization, The Alternating Bit Algorithm

- Denote L = $bit_s, q_{s,r}, bit_r, q_{r,s}$, the value of the of this label sequence is in [0*1*] or [1*0*]
  - where $q_{s,r}$ and $q_{r,s}$ are the queue messages in transit on the link from S to R and from R to S respectively
- We say that a single border between the labels of value 0 and the labels of value 1 slides from the sender to the receiver and back to the sender
- Once a safe configuration is reached, there is at most one border in L, where a border is two consecutive but different labels

The Alternating Bit Algorithm, borders sample

- Suppose we have two borders
- If frame m_2 gets lost, receiver will have no knowledge about it



Pseudo-Self-Stabilization, The Alternating Bit Algorithm

- Denote $L(c_i)$ - the sequence L of the configuration c_i
- A loaded configuration c_i is a configuration in which the first and last values in $L(c_i)$ are equal

Pseudo-Self-Stabilization, The Alternating Bit Algorithm

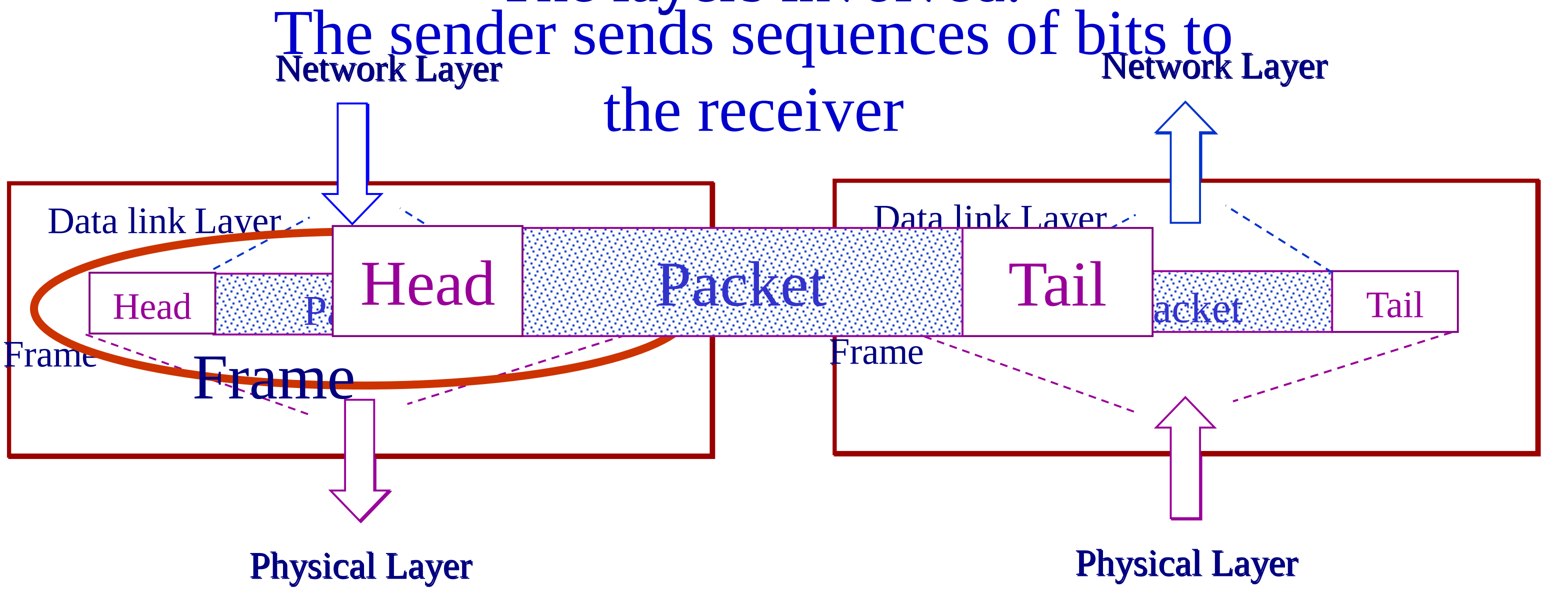
- The algorithm is pseudo self-stabilizing for the data-link task, guaranteeing that the number of messages that are lost during the infinite execution is bounded, and the performance between any such two losses is according to the abstract task of the data-link

Roadmap

The Data Link Algorithm

- The task of delivering a message is sophisticated, and may cause message corruption or even loss

The layers involved:



The alternating-bit algorithm

Is used to cope with possibility of frame corruption or loss

Sender

```
initialization
begin
    i := 1
    bit_s := 0
    send(⟨bit_s,im_i⟩) (*im_i is fetched*)
end (*end initialization*)
upon a timeout
        send(⟨bit_s,im_i⟩)
upon frame arrival
begin
        receive(FrameBit)
        if FrameBit = bit_s then
            begin
                bit_s := (bit_s + 1) mod 2
                i := i + 1
        end
        send(⟨bit_s,im_i⟩) (*im_i is fetched*)
end
```

acknowledgement

Receiver

```
initialization
begin
    j := 1
    bit_r := 1
end (*end initialization*)
upon frame arrival
begin
        receive(⟨FrameBit , msg⟩)
        if FrameBit ≠ bit_r then
            begin
                bit_r := FrameBit
                j := j + 1
                om_j := msg
        end
        send(bit_r)
end
```

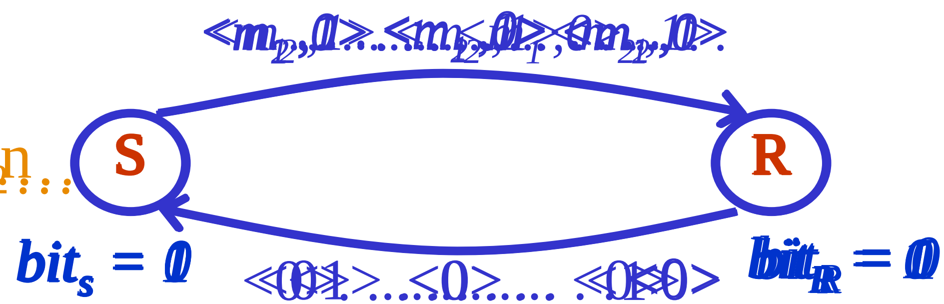
Send acknowledgement

Every message from the sender is repeatedly sent in a frame to the receiver until acknowledges arrives

The alternating-bit algorithm – run sample

S received ack.

R received m_1 again



Once the sender receives an acknowledgment <1>, no frame with sequence number 0 exists in the system

There Is No Data-link Algorithm that can Tolerate Crashes

- It is usually assumed that a crash causes the sender/receiver to reach an initial state
- No initialization procedure exists such that we can guarantee that every message fetched by the sender, following the last crash, will arrive at its destination
- The next Execution will demonstrate this point. Denote:
  - $Crash_R$ – receiver crash
  - $Crash_S$ – sender crash
- $Crash_X$ causes X to perform an initialization procedure

The Pumping Technique

Reference Execution (RE) = $Crash_S$, $Crash_R$, $send_S(f_{s1})$, $receive_R(f_{s1})$, $send_R(f_{r1})$, $receive_S(f_{r1})$, $send_S(f_{s2})$, … , $receive_S(f_{rk})$

The idea : repeatedly crash the sender and the receiver and to replay parts of the RE in order to construct a new execution E'

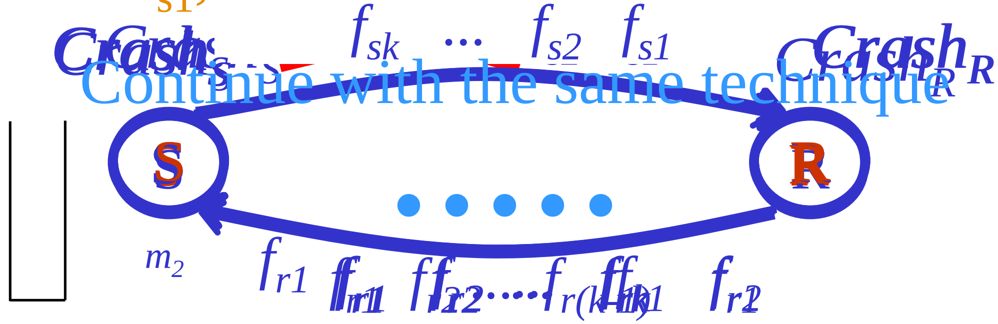
S sends f_{s1}, receives f_{r1}, sends f_{s2}, receives f_{r2}, … , receives

If these k frames are lost, no information about the message exists in the system

Suppose $Crash_S$ and $Crash_R$ occurred

R receives f_{s1}, … and sends f_{r2}

Continue with the same technique



Conclusion !

- It is possible to show that there is no guarantee that the k^{th} message will be received
- We want to require that eventually every message fetched by the sender reaches the receiver, thus requiring a Self-Stabilizing Data-Link Algorithm

Roadmap

2.10 Pseudo-Self-Stabilization

3.1 Initialization of a Data-Link Algorithm in the Presence of Faults

3.2 Arbitrary Configuration Because of Crashes

4.2 Data-Link Algorithms: Converting Shared Memory to Message Passing

Arbitrary configuration because of crashes

- A combination of crashes and frame losses can bring a system to any arbitrary states of processors and an arbitrary configuration

Any Configuration Can be Reached by a Sequence of Crashes

- The pumping technique is used to reach any arbitrary configuration starting with the reference execution

  Reference Execution (RE) = $Crash_S$, $Crash_R$, $send_S(f_{s1})$, $receive_R(f_{s1})$, $send_R(f_{r1})$, $receive_S(f_{r1})$, $send_S(f_{s2})$, … , $receive_S(f_{rk})$

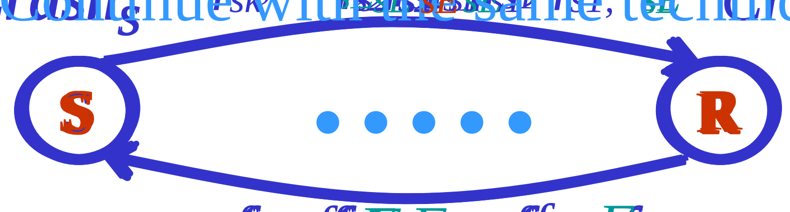
- The technique is used to accumulate a long sequence of frames

Reaching an Arbitrary Configuration

- Our first goal – creating an execution in which RE appears i times in a row $(RE)^i$

Denote : F_{rE} (F_{sE}) – the sequence of frames sent by the receiver (sender) in RE

F^i_{rE} (F^i_{sE}) = the sequence $F_{r(s)E}\ F_{r(s)E}\ \ldots\ F_{r(s)E}$ (i times)



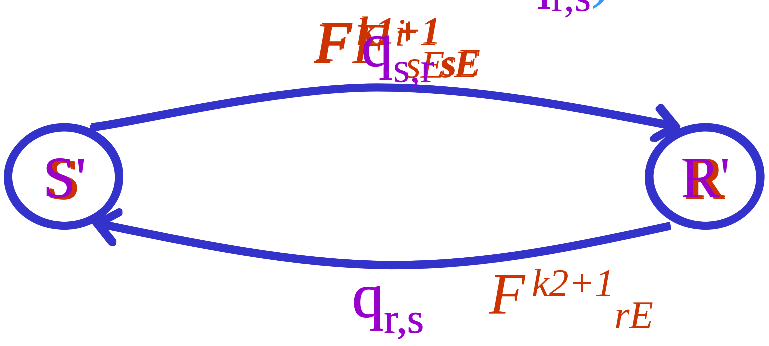
Reaching an Arbitrary Configuration

- Our second goal – achieving c_a (an arbitrary configuration)
- Denote k_1 (k_2)- the number of frames in $q_{s,r}$ ($q_{r,s}$) in c_a
- $i = k_1+k_2+2$

We deploy the same with R, reaching the arbitrary configuration c_a

Send the RE using the first F_{rE} until it reaches its desired state (loosing the frames sent by it and the leftovers of F^i_{sE} F^{k2}_{rE} that are not in $q_{r,s}$)

Using the previous technique we accumulate



Crash-Resilient Data-Link Algorithm,With a Bound on the Number of Frames in Transit

- Crashes are not considered severe type of faults (Byzantine are more severe - chapter 6)
- The algorithm uses the initialization procedure, following the crashes of S and R
- bound – the maximal number of frames that can be in transit

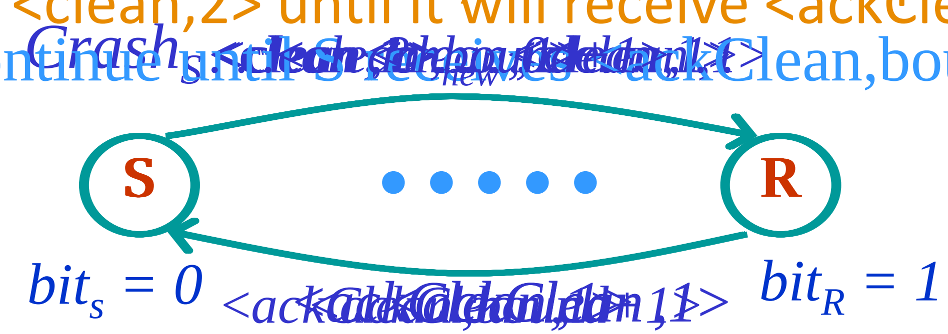
When the sender receives the first <ackClean,bound+1> it can be sure that the only label in transit is bound+1, and can initialize the alternating bit algorithm (similarly R can initialize as well)

S received <ackClean,1>, then sends repeatedly <clean,2> until it will receive <ackClean,2>

S ,in after-crash state, invokes a clean procedure

S crashes

Continue until S receives <ackClean,bound+1>



Crash-Resilient Data-Link Algorithm – R crashes

R received msg and assigned *FrameBit* to bit_R it then delivers msg to the output queue – The Problem : extra copy of msg in the output queue

R crashes

Crash-Resilient Data-Link Algorithm – R crashes

Can we guarantee at most one delivery, and exactly-once delivery after the last crash?

- *bit_R* initialization should assure that a message fetched after the crash will be delivered
- A solution:
  - S sends each message in a frame with label 0, until Ack. arrives and then sends the same message with label 1 until an Ack. arrives
  - R delivers a message only with label 1 that arrives immediately after label 0

Roadmap

2.10 Pseudo-Self-Stabilization

3.1 Initialization of a Data-Link Algorithm in the Presence of Faults

3.2 Arbitrary Configuration Because of Crashes

4.2 Data-Link Algorithms: Converting Shared Memory to Message Passing

Converting Shared Memory to Message Passing

Designing a self-stabilization algorithm for asynchronous message-passing systems is more subtle than the same task in shared memory systems

Main difficulty :

The messages stored in the communication links

- No bound on message delivery time
- No bound on number of messages that can be in link

There are infinitely many initial configurations from which the system must stabilize

Converting Shared Memory to Message Passing

- Our main goal is designing of compiler:

- First goal in designing of such a compiler is a self-stabilizing data-link algorithm

Definition of Self-Stabilizing Data-Link Algorithm

- Data-Link Algorithm : Messages fetched by sender from network layer should be delivered by receiver to network layer without duplications, omissions or reordering
- One of the implementations of data-link task is the token-passing algorithm
- Token-passing task is a set of executions TP
- The legal execution of TP is the sequence of configurations in which :
  - No more than one processor holds the token
  - Both the sender and receiver hold the token in infinitely many configurations

Unbounded solution of TP task

Sender:

```
upon timeout
        send (counter )
upon message arrival
begin
        receive (MsgCounter)
        if MsgCounter ≥ counter then
        begin
                counter := MsgCounter + 1
                send (counter)
        end
        else send (counter)
end
```

A *timeout mechanism* is used to ensure that the system will not enter to *communication-deadlock configuration*

Each message has integer label called *MsgCounter*

Sender (and Receiver) maintains an unbounded local variable called *counter*

Unbounded solution of TP task

Receiver:

upon message arrival
begin
    receive (*MsgCounter*)
    if *MsgCounter* $\neq$ *counter* **then**   *Token* arrives
        counter := *MsgCounter*
    send (*counter*)   *Token* released
end

In safe configuration of TP and the algorithm - *counter* values of all messages and values of the *counters* of sender and receiver, have the same value (lemma 4.1)

The algorithm is self-stabilizing

For every possible configuration *c* , every fair execution that starts in *c* reaches a safe configuration with relation to *TP* (Theorem 4.1)

Question : Whether the unbounded counter and label can be eliminated from the algorithm ?

Answer : NO

Lower Bound on the System Memory

- The memory of the system in configuration *c* is the number of bits for encoding state of sender, receiver, and messages in transit.

- *Weak-Exclusion* task (*WE*): In every legal execution E, there exists a combination of steps, a step for each processor, so that these steps are never executed concurrently.

- We will prove that there is no bound on system memory for *WE* task .

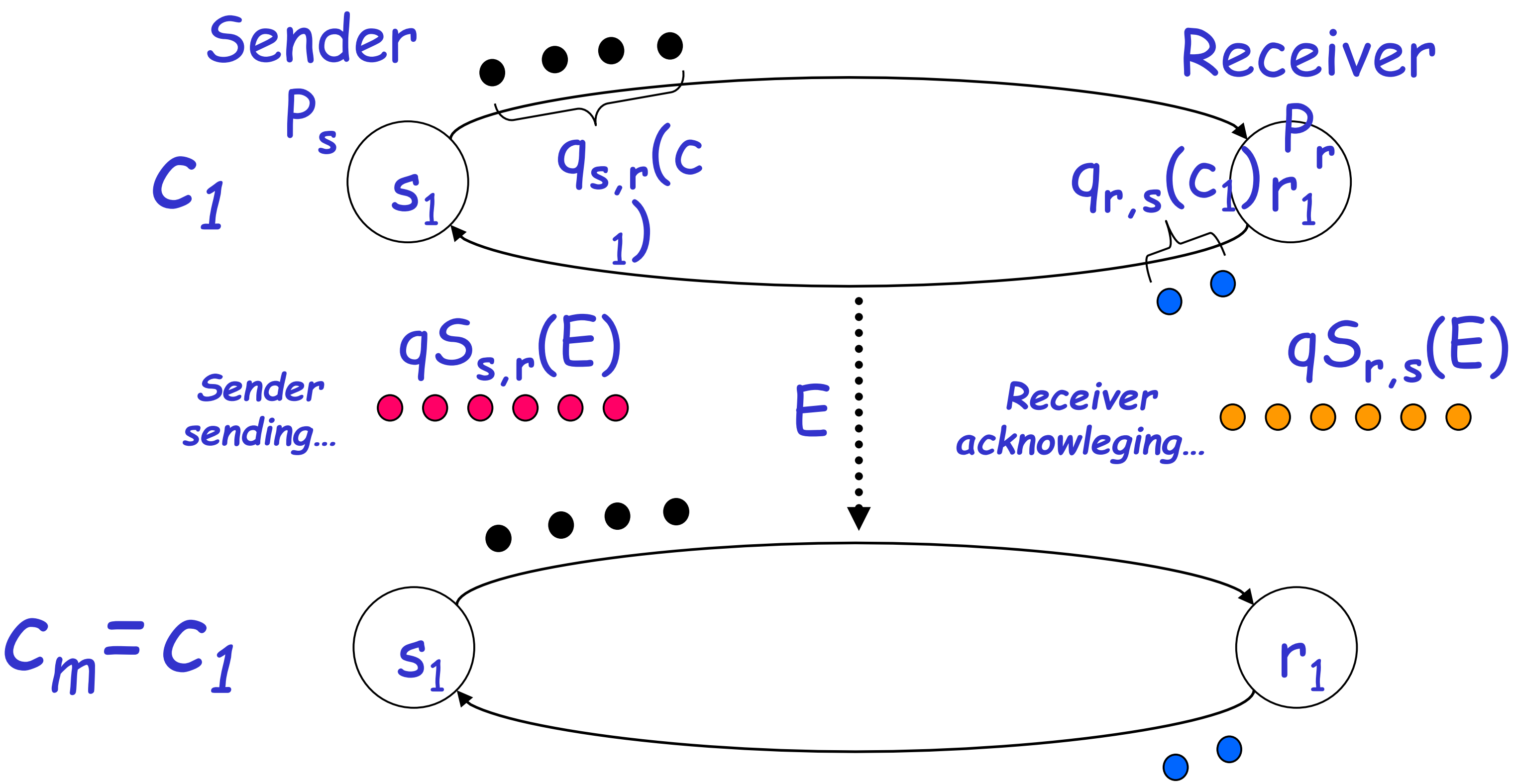
Lower Bound on the System Memory

Theorem: For any self-stabilizing message driven protocol for *WE* task and for any execution E' in *WE* all the configurations are distinct.

Hence for any $t > 0$, the size of at least one of the first t configurations in E is at least $\log_2(t)$

Proof of Theorem

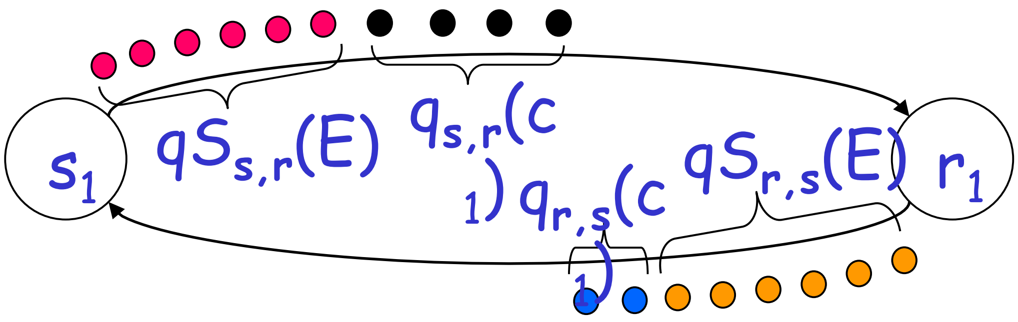
Any execution E' in which not all the configurations are distinct has circular sub-execution $E = (c_1, a_2, \ldots, c_m)$ where $(c_1 = c_m)$



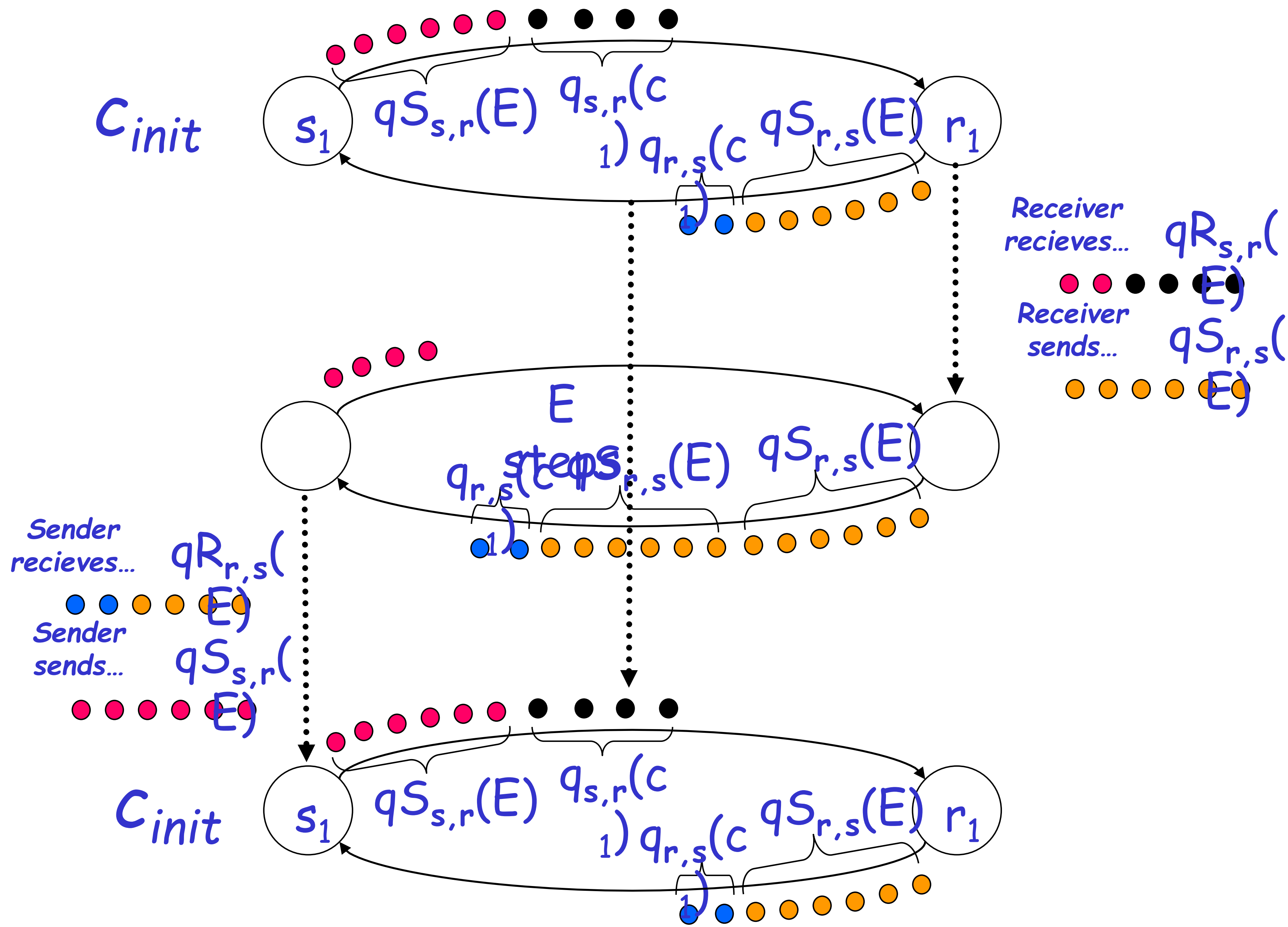
Proof - Building CE and c_{init}

- Let E – be circular sub-execution
- S_i - sequence of steps of P_i
- *CE* – set of circular executions
- Each execution in *CE* – merge of S_i's , while keeping their internal order
- We obtain initial configuration of E_c in CE from c_1 of E, and sequence of messages sent during E

c_{init}

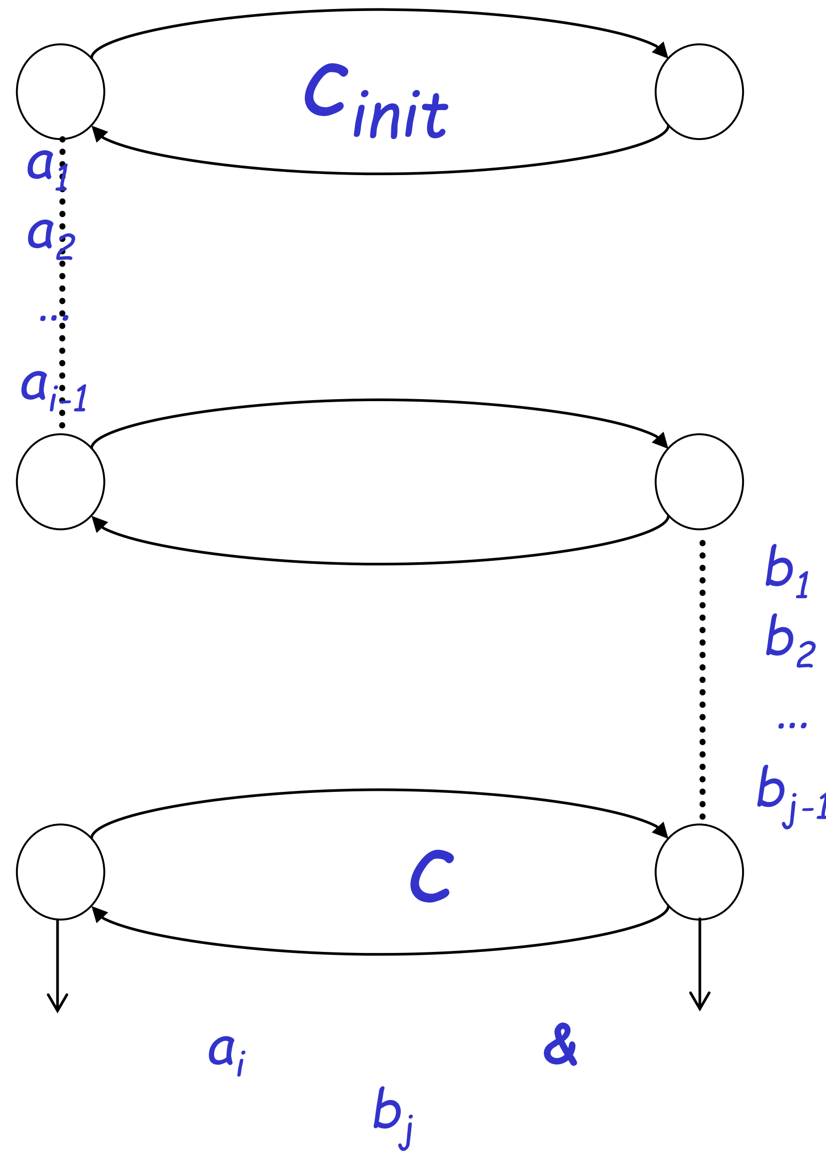


Proof - Possible execution in *CE*



Proof – cont...

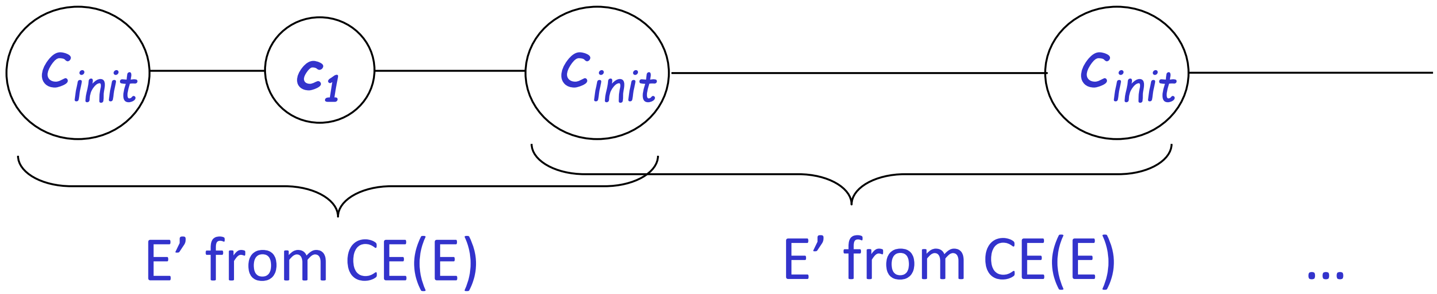
- Sender steps during E is $S_{sender} = \{a_1, a_2 \ldots a_m\}$
- Receiver steps during E is $S_{receiver} = \{ b_1, b_2 \ldots b_k \}$
- For any pair $<a_i, b_j>$ there exists $E'' \in CE$ in which there is configuration *c*, such that a_i and b_j is applicable in *c* → *c* is not safe configuration



Proof – cont…

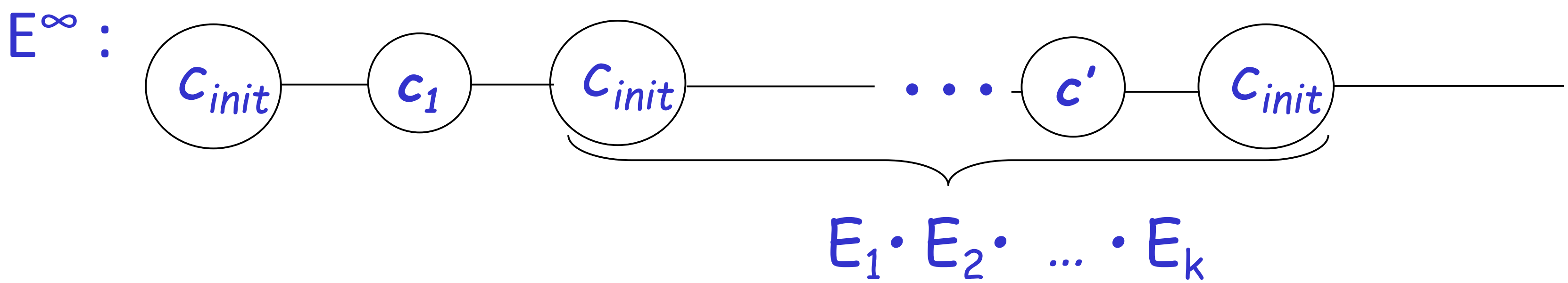
If self-stabilizing algorithm *AL* for *WE* task have circular sub-execution E → exists infinite fair execution E^{∞} of *AL* , none of whose configurations is safe for *WE*

E':



Proof – cont…

- Let assume in contradiction that c_1 is safe
- Then let's extend E' to E^∞ by $E_1, E_2, \ldots, E_k$ - executions from CE(E)



- For each pair $< a_i, b_j >$ there is c' in E^∞ so that both a_i, b_j applicable in c' → c' is not safe
- **The proof is complete!**

Bounded-Link Solution

- Let *cap* be the bound on number of the messages in transit

- The algorithm is the same as presented before with *counter* incremented modulo *cap+1*

```
counter := (MsgCounter + 1)mod(cap+1)
```

- Sender must eventually introduce a counter value that not existing in any message in transit

Randomized Solution

- The algorithm is the same as original one with *counter* chosen randomly

```
label := ChooseLabel(MsgLabel)
```

- At least three labels should be used

- The sender repeatedly send a message with particular label *L* until the a message with the same label *L* arrives

- The sender chooses randomly the next label L’ from the remaining labels so that $L' \neq L$

Self-Stabilizing Simulation of Shared Memory

- The heart of the simulation is a self-stabilizing implementation of the read and write operations
- The simulation implements these operations by using a self-stabilizing, token passing algorithm
- The algorithm run on the two links connecting any pair of neighbors
- In each link the processor with the larger *ID* acts as the sender while the other as the receiver (Remind: all processors have distinct *ID*s)

Self-Stabilizing Simulation of Shared Memory - cont…

- Every time P_i receives a token from P_j. P_i write the current value of R_{ij} in the value of the token

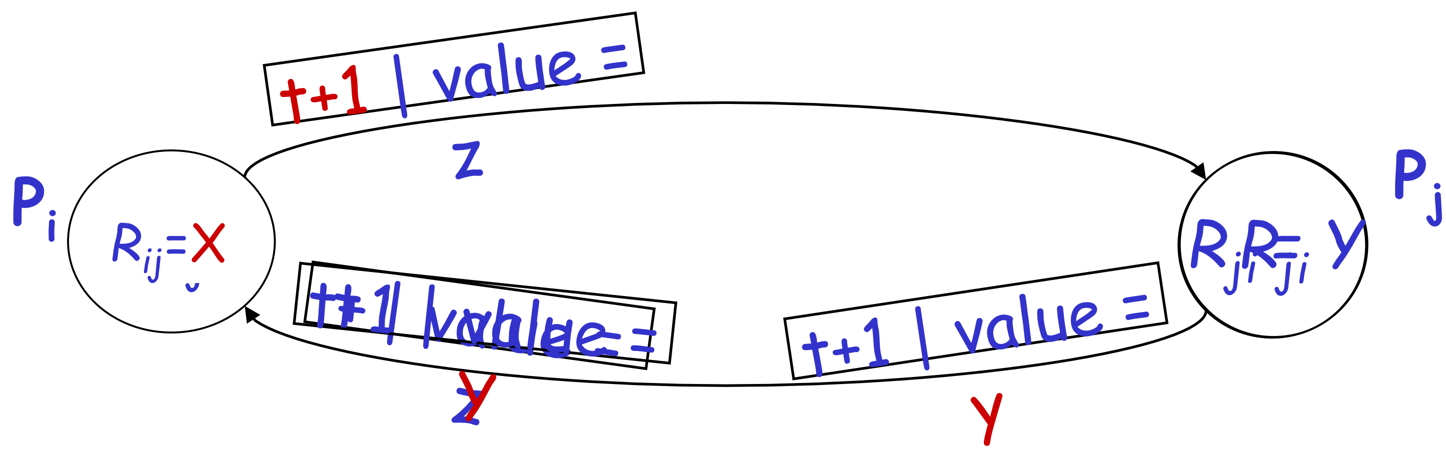
- Write operation of P_i into r_{ij} is implemented by locally writing into R_{ij}

- Read operation of P_i from r_{ji} is implemented by:
  1. P_i receives the token from P_j
  2. P_i receives the token from $P_{j.}$ Return the value attached to this token

Self-Stabilizing Simulation of Shared Memory - Run

Write Operation: P_i write x to r_{ij}

Read Operation: P_i read from value y from r_{ji}



4. P_{ij} receive token from P_{ji} and read the value of the token

2. P_{ij} receive token from P_j and write R_{ij} to value of the token

3. P_{ji} send token from P_j