Cut-through Routing in Self-stabilization*

Joffroy Beauquier‡ Ajoy K. Datta⋆ Sébastien Tixeuil‡

‡ LRI - CNRS UMR 8623, Université Paris Sud, France
⋆ School of Computer Science, University of Nevada Las Vegas

Abstract

In this paper, we propose three *self-stabilizing* algorithms, all in the *cut-through* model, where the messages must be forwarded to a neighbor before they are completely received. A self-stabilizing algorithm [4, 6], regardless of the initial system configuration, converges in finite time to a set of legitimate configurations.

First, we provide a self-stabilizing communication scheme that is based on the IBM token ring protocol. This protocol is obtained as an output of a transformer that takes as input a self-stabilizing token passing protocol. Our solution is cut-through compliant and exploits the cut-through routing scheme to preserve a low round trip delay.

Then, we present a self-stabilizing census algorithm for unidirectional rings in the cut-through routing scheme. The distributed *census* problem can be informally described as follows: The processors cooperate to reach a global configuration where every processor can determine, within finite time, which processors are and which processors are not present in the network. This protocol takes advantage of cut-through routing by distributing the global census information among all processors in the network. It also preserves the low round trip delay.

Keywords: census, cut-through routing, self-stabilization, token ring, token passing.

1 Introduction

Self-stabilization. In 1974, Dijkstra pioneered the concept of self-stabilization in a distributed network [4]. A distributed system is self-stabilizing if it returns to a *legitimate* state in a finite number of steps regardless of the initial state, and the system remains in a legitimate state until another fault occurs. Thus, a self-stabilizing algorithm tolerates transient processor faults. These transient faults include variable corruption, program counter corruption (which temporarily cause a processor to execute its code from any point), and communication channel corruption.

In the context of computer networks, resuming correct behavior after a fault occurs can be very costly [11]: the whole network may have to be shut down and globally reset in a

*Contact author: Sébastien Tixeuil, LRI - CNRS UMR 8623, Bâtiment 490, Université Paris Sud, FR91405 Orsay cedex, France. Email: tixeuil@lri.fr. Fax: 33 1 69 15 65 86.

good initial state. While this approach is feasible for small networks, it is far from practical in large networks such as the Internet. Self-stabilization provides a way to recover from faults without the cost and inconvenience of a generalized human intervention: after a fault is diagnosed, one simply has to remove, repair, or reinitialize the faulty components, and the system, by itself, will return to a good global state within a relatively short amount of time.

Routing. In the *store and forward* routing technique, no allocation of bandwidth is reserved in advance. Only one message (or packet) is transmitted at one time on any communication link. This message uses the full capacity of the link. The links are shared among different sessions only on a demand basis, not on a static allocation basis. When a message arrives at an intermediate processor on its path to the destination processor, the message must be *fully* received and stored at the intermediate processor. The message then waits in a delay queue if the required output communication link is busy, and finally, when its turn comes, the message in its *entirety* is forwarded to another link. There are several drawbacks of the store-and-forward technique. Each processor in the path from the source to the destination processor needs a local memory of at least the size of the message to store the message. The copying to and from the memory introduces an overhead. The queuing delays in the processors can be significant.

The *cut-through* routing is used in many ring networks (including Token Ring and FDDI). In this routing scheme, a processor can start forwarding any portion of a message to the next processor on the message's path before receiving the message in its entirety. If this message is the only traffic on the path, the total delay incurred by the message is bounded by the transmission time (calculated on the slowest link on the path) plus the propagation delay. So, the total message delay is proportional to the length of the message and to the number of links on the path. Some pieces of the same message may simultaneously be traveling on different links and some other pieces are stored at different processors. As the first bit of the message is transmitted on the links on the message's routing path, the corresponding links are reserved, and the reservation of a link is released when the last bit of the message is transmitted on the link. This approach removes the need of having a local memory of any processor greater than the one required to store a bounded number of bits, and also reduces the message delay to a small (bounded by the buffer size of the processor) value. As with the current processors, the time needed for sending/receiving bits to/from a communication medium is far greater than the time needed to perform the basic computational steps (such as integer calculations, tests, read/write from/to registers, etc.), we can assume that a given process can perform a limited number of steps between the receipt of two pieces of a message.

Related Work. In the context of the communication networks, the paper [9] showed that crash and restart failures may lead the distributed systems into an arbitrary configurations, highlighting the need for algorithms that are resilient to those failures (such as self-stabilizing algorithms). Dijkstra [4] used shared registers to model the communication among neighboring processors. In the recent years, some more realistic models (like message-

passing) have been addressed (see [6] for further references).

Varghese [15] introduced a new scheme, called *counter flushing*, and used this technique to model Dijkstra's k-state token passing algorithm [4]. In [1], Costello and Varghese used [15] to construct a FDDI token ring communication protocol. This token passing technique leads to short stabilization time (compared to the non-stabilizing version of the protocol) but introduces a penalty in terms of time efficiency when the protocol is stabilized.

The topology update problem in the self-stabilizing setting has been well-studied in the literature [2, 5, 7, 10, 12]. But, all these protocols are written for the store-and-forward model. A self-stabilizing census algorithm was presented in [2, 3], but the space required at each processor in this algorithm is quite high—proportional to the size of the network.

Our Contributions. In this paper, we investigate the possibility of cut-through routing in the context of self-stabilization. This work shows the advantages of cut-through routing in several ways: the reduction of the round trip delay between the receipt of two messages by the same processor, and distribution of collective global information across the whole network.

We design a transformer that takes as input a self-stabilizing token passing algorithm and provides as output a self-stabilizing token ring communication protocol. This transformer preserves the cut-through routing property of the original algorithm, and does not add any extra delay at any node. Then, we provide a self-stabilizing token passing scheme. The stabilization time of our scheme may be worse than that of [15], but once stabilized, the overhead due to self-stabilization is reduced by a factor up to 32. So, our approach is suitable in networks where faults exist but are rare.

Finally, we take a high memory consuming task (the census) and provide a self-stabilizing solution using the Cut-through routing scheme. Our algorithm, in addition to being failure resilient, optimizes the transmission of messages (a fraction of the time required in the store-and-forward model is needed). Moreover, the memory used at each processor is $O(\log_2 N)$ bits of memory while the census task typically requires $O(N \log_2 N)$ bits.

Outline of the Paper. The rest of the paper is organized as follows. Section 2 includes some background concepts, including the cut-through routing scheme. The cut-through transformer and our token passing scheme are presented in Section 3. The census algorithm and its correctness proof are given in Section 4. In Section 4.2, we discuss the complexity issues before concluding in Section 5.

2 Preliminaries

System. A *processor* is a sequential deterministic machine that uses a local memory, a local algorithm, and input/output capabilities. Intuitively, a processor executes its local algorithm. This algorithm can modify the state of its local memory and send/receive messages using its communication ports. An *unidirectional communication link* transmits messages from a processor o (*origin*) to a processor d (*destination*. A link connects an output port of o with an input port of d.

Let $\mathcal{S} = (\mathcal{P}, \mathcal{M})$ be a *distributed system*, where $\mathcal{P}$ is a set of processes and $\mathcal{M}$ is a set of communication links. A distributed system is represented by a directed graph with nodes representing processors and directed edges (or arcs) representing communication channels (or links). We consider directed unidirectional ring in this work. We also assume that processors communicate in a clockwise manner (*i.e.,* processor i can send information to processor $i + 1 \bmod n$, where n is the size of the ring. The state of a processor can be reduced to the state of its local memory and the state of a communication link can be reduced to its contents. Then the global system state (or *configuration*) can be defined as the product of the states of all its processes and the contents of all its links.

The set of all configurations of $\mathcal{S}$ is denoted by $\mathcal{C}$. A *computation* e of an algorithm $\mathcal{A}$ is a maximal sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration c_{i+1} is reached from c_i by a single step of at least one process. This sequence is maximal in the sense that it is either infinite, or it reaches a configuration where no further action is possible for any of the processors. In the previous example, c_1 is called the *initial configuration* of e.

Self-Stabilization. In the most general case, the specification of a problem is defined by enumerating computations that solve the problem. A particular configuration c satisfies a specification A if every computation starting from c is in A. A set of configurations $B \subset \mathcal{C}$ is *closed* if for any $b \in B$, any computation of System $\mathcal{S}$ starting from b contains configurations only in B. A set of configurations $B_2 \subset \mathcal{C}$ is an *attractor* for a set of configurations $B_1 \subset \mathcal{C}$ if for any $b \in B_1$, any computation of $\mathcal{S}$ starting from b reaches a configuration in B_2.

Definition 2.1 (Self-stabilization) *A system $\mathcal{S}$ is* self-stabilizing *for a specification A if there exists a non-empty set of configurations $\mathcal{L} \subset \mathcal{C}$ such that the following two conditions are true:*

Closure *Any computation of $\mathcal{S}$ starting from a configuration in $\mathcal{L}$ satisfies $\mathcal{A}$.*

Convergence *$\mathcal{L}$ is an attractor for $\mathcal{C}$.*

The set $\mathcal{L}$ is known as the set of *legitimate configurations*. In order to prove that a system is stabilizing, we carefully choose $\mathcal{L}$ to show the following two properties: (i) Any computation starting from a legitimate configuration satisfies the considered problem (closure property). (ii) Any computation starting from any configuration (possibly not in $\mathcal{L}$) leads to a legitimate configuration (convergence property).

Messages. A message on a communication link is represented as a signal composed of the *signal elements*. A signal element is one of the following types:

Signal element	**Denoted by**
start-of-message	`S`
binary element	`0` or `1`
end-of-message	`E`

We define an abstract enumerated data type, (`SignalElement: S, E, 0, 1`) for signal elements. In the algorithm in Section 4, the messages are abstracted as sequences of

SignalElements containing 0s and 1s delimited by an S and E, and can be expressed using the following regular expression:

$$S(1|0)^*E$$

Buffer. The Buffer is a FIFO structure with the random access capabilities and may contain SignalElements or N, an additional symbol used to indicate any kind of error. It may be accessed using the following functions:

Add(b,s) appends the SignalElement s to the end of Buffer b. If b is full, the last element is simply replaced by s.

Remove(b) removes and returns the SignalElement from the head of Buffer b. If b is empty, Remove has no effect on b and returns N.

Read(b,index) returns the SignalElement at the position index in Buffer b. The parameter index ranges from 0 (head) to Size(b)- 1 (tail). If index is out of range, Read returns N.

Write(b,index,s) overwrites the SignalElement s at the position index in Buffer b. The index ranges from 0 (head) to Size(b) - 1 (tail). If index is out of range, Write does nothing.

InputElement(b) reads a SignalElement s from the incoming communication link. If s is either 0 or 1, it calls the function Add(b,s) and returns true. If the signal element read is neither 0 nor 1, the function returns false.

OutputElement(b) writes a SignalElement to an outgoing communication link. If Buffer b is empty, an N is written to the outgoing channel and false is returned. Otherwise, Read(b,0) is written to the outgoing channel, Remove(b) is called, and OutputElement returns true.

Size(b) returns the current size of the Buffer b, *i.e.*, it returns the number of SignalElements currently in the buffer. Note that invocation of the functions Add, Remove, InputElement, and OutputElement on a buffer may change the size of the buffer, but the functions Read and Write have no effect on the buffer size.

Clear(b) clears the Buffer b by repeatedly calling Remove(b) while Size(b) is greater than 0.

Forward(s) writes the SignalElement s to the outgoing communication link. Note that this is the only function that does not use the buffer.

Communication Layer. Each processor maintains a communication layer (CL) whose role is to inform the processor when a message arrives. The CL continuously scans the input communication link. When an `S` is received, the CL calls the `MessageHandler` function (described below). After the `MessageHandler` returns, the CL sends an `E` to the outgoing communication link.

We assume that the `MessageHandler` function outputs an `S SignalElement` before executing the first call to `OutputElement` or `Forward`, and outputs an `E SignalElement` after executing the last call of `OutputElement`. This hides the `S` and `E SignalElement`s in the `MessageHandler` functions. We also assume that the communication buffer maintained by the processor is cleared before the `MessageHandler` function is called so that the code within the `MessageHandler` function can assume that the Buffer is initially empty. In some algorithms presented in this paper, we reduce the effective size of some messages. This is done by inserting an early `E SignalElement` at the end of the `MessageHandler` function. The remaining of the incoming message is then ignored by the communication layer.

Cut-Through Constraints. A cut-through `MessageHandler` function (*i.e.*, a cut-through algorithm) must satisfy certain characteristics (well-formedness) as defined below. This can be viewed as an extended version of the producer-consumer problem.

Producer-Consumer Constraint: A processor may not send an `N SignalElement`, *i.e.*, `OutputElemer` will be executed only when `Size(b)` is greater than 0.

Rate Constraint: When a processor forwards a message, if there are still some `SignalElement`s to be read, then `input` and `output` actions must be executed alternately. The alternative execution of the input and output actions is to maintain a certain rate of information flow rate at each processor in the asynchronous model. This can be described by the following regular expression:

$$\texttt{input}^{+}(\texttt{output},\texttt{input})^{*}\texttt{output}^{+}$$

In our model, the input actions are the calls to `InputElement`, and the output actions are the calls to `OutputElement` and `Forward`.

Message Re-initialization with Cut-Through Constraints. In some situations, a message must be deleted and replaced by a new one. This is easy to implement in the store-and-forward model, but is rather difficult to implement in the cut-through model because when the new message is created and ready to be forwarded, some initial parts of the message to be deleted may already have been forwarded. In order to cope with this kind of problems, we introduce a technique using a special logical entity, called a `ZeroMarker` (denoted by `Z`). When a processor finds out that a message needs to be destroyed, but is being forwarded, it inserts a `Z` in its buffer. Thus, contrary to the store-and-forward algorithm, the actual message initialization does not occur immediately, but will instead be delayed depending on the buffer size of the processors. Although the local memory at any processor is bounded, it

is possible for the processor to advance Z to the beginning of a message. After visiting every processor, Z advances towards the beginning of the message by one buffer size. This is illustrated using the following example, where we consider processors having 3-bit buffers.

Incoming	Processor Buffer	Outgoing	Comments
$\overrightarrow{0Z0101}$	[]		
$\overrightarrow{0Z0}$	[101]		3 InputElements
$\overrightarrow{0Z0}$	[10]	$\overrightarrow{1}$	OutputElement
$\overrightarrow{0Z}$	[010]	$\overrightarrow{1}$	InputElement
$\overrightarrow{0Z}$	[01]	$\overrightarrow{01}$	OutputElement
$\overrightarrow{0}$	[Z01]	$\overrightarrow{01}$	InputElement
$\overrightarrow{0}$	[Z0Z]	$\overrightarrow{01}$	Marker advances
$\overrightarrow{0}$	[Z0]	$\overrightarrow{Z01}$	OutputElement

2.1 Complexity

This section introduces some time complexity metrics used in the cut-through model.

Definition 2.2 (Transmission Time) *The transmission time is the time needed by a processor i to receive (resp., send) a message from (resp., to) a communication link. The transmission time is given by the following relation :*

$$T_{t_i} = \frac{l}{C}$$

where l is the length of the message in bits and C is the capacity (the number of bits that can be held) of the communication link.

Definition 2.3 (Propagation Time) *The propagation time is the time needed by a bit to traverse a communication link between two processors i and j. The propagation time is given by the following relation :*

$$T_{p_{i\rightarrow j}} = \frac{d}{\tau}$$

where d is the physical length of the link and τ is the celerity (speed) of the information unit.

Definition 2.4 (Buffer Delay) *The delay time is the time needed to completely fill a buffer of a processor i. The delay time is given by the following relation :*

$$T_{d_i} = \frac{b}{C}$$

where b is the length of the buffer in bits, and C is the rate of the communication link.

Definition 2.5 (Communication Time) *The communication time is the time needed for a message to go from a processor i to a processor j is given by the following relation:*

$$T_c = T_{t_i} + T_{p_{i\rightarrow i_1}} + T_{d_{i_1}} + \cdots + T_{p_{i_{k-1}\rightarrow i_k}} + T_{d_{i_k}} + T_{p_{i_k\rightarrow j}}$$

where i_1, i_2,..., i_k are the intermediate processors between i and j.

Definition 2.6 (Round Trip Time) *When a message is circulating in a ring network, the time needed for one complete traversal around the ring is given by the following relation:*

$$T_r = \sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i \leftarrow i+1 \bmod N}} \right)$$

2.2 Specifications

Specification of the Communication problem. We consider a distributed system $\mathcal{S} = (\mathcal{P}, \mathcal{M})$ where every processor p has a unique non-corruptible processor identifier, $\mathtt{Id}_p$. An external user of the system can perform SendMessage and ReceiveMessage functions at any processor in the network. A protocol $\mathcal{SSTR}$ (for Self-stabilizing Token Ring) satisfies the Communication problem if a processor i executes SendMessage(Id_j, m), and a processor j executes ReceiveMessage, then j will eventually receive m sent by i. We also require that messages are transmitted in the order they were sent, with no loss nor duplication. Finally, the protocol should be compatible with cut-through routing. We refer to this specification as $\mathsf{Spec}_{\mathcal{COMM}}$.

Specification of the Census problem. We consider a distributed system $\mathcal{S} = (\mathcal{P}, \mathcal{M})$ where every processor p has a unique non-corruptible processor identifier, $\mathtt{Id}_p$. An external user of the system can perform Exists function at any processor in the network. A protocol $\mathcal{CCT}$ (for Census Cut-through) satisfies the Census problem if at a processor i, Exists(Id_j) returns true if $j \in \mathcal{P}$, and false otherwise. Finally, the protocol should be compatible with cut-through routing. We refer to this specification as $\mathsf{Spec}_{\mathcal{CENS}}$.

3 Self-stabilizing Communications in Cut-through

In this section, we provide a self-stabilizing communication protocol that performs in the cut-through model. This protocol is based on the IBM Token Ring protocol for the communication part, and on the counter flushing mechanism for the self-stabilizing part.

We first present an automatic transformer (denoted by $\mathcal{SSTR}$) that takes as input a self-stabilizing token passing algorithm that is cut-through compliant, and provides as output a self-stabilizing token ring communication protocol that is also cut-through compliant. This solution is presented in Section 3.1 and satisfies Specification $\mathsf{Spec}_{\mathcal{COMM}}$.

Then, we discuss the input of the transformer. The counter-flushing mechanism [15] can be used as a possible input to $\mathcal{SSTR}$. Nevertheless, we provide a new self-stabilizing token passing algorithm (denoted by $\mathcal{ACF}$) that is also cut-through compliant. Compared with the counter-flushing technique [15], Algorithm $\mathcal{ACF}$ has a longer stabilization time. However, once stabilized, its overhead (in terms of network load) is reduced by a factor 32 with respect to the scheme of [15]. The two schemes will be compared in more details in Section 3.2.

3.1 Token Ring Transformer ($\mathcal{SSTR}$)

Our objective in this section is to implement a communication scheme (*i.e.* the SendMessage and ReceiveMessage functions) between any two processors in the ring. We now informally describe the implementation of these two functions using a token passing protocol.

In our scheme, in order to send data (that comes from the application layer), a processor must have a special *privilege*. Also, for the system to operate properly, there must be exactly one such privilege, and the privilege must pass from one processor to the other infinitely often. An self-stabilizing algorithm that solves the token passing problem guarantees that these two properties (unique privilege and fair access to the privilige) are eventually satisfied. So, the input to our transformer is a self-stabilizing token passing algorithm. The correctness of the self-stabilizing token passing algorithm eventually guarantees a single privilege input to the transformer. The "Privilege" in the token passing algorithm corresponds to the "frame" in the transformer. Therefore, the "single privilege" in the token passing algorithm ensures the "single frame" in the transformer.

A frame may be a *token frame* or a *data frame*. The token frame represents the ability of a processor to transmit data issued by its application layer, while the data frame encapsulates the data transmitted by the application layer. The `isToken` bit in the frame (refer to Table 1) signifies the type (token *vs.* data) of the frame. A processor is allowed to send a data frame only if it holds the token. In other words, a processor is able to execute `SendMessage` only if it has the token. A processor can execute `ReceiveMessage` only if it receives a data frame destined for itself. The `hasToken` bit at a processor i is true if i currently holds the token, and false otherwise. If a processor i wants to send data, but does not have the token, then $\texttt{wantToken}_i$ will be true.

The processor (say, i) holding the token starts sending the data frame. The other processors check if they are the destination of the message. If yes, they deliver the message to the upper application layer and forward the frame to the following processor on the ring; otherwise, they simply forward the frame to the following processor on the ring. When the frame comes back to i, i becomes sure that the message has been delivered to the destination processor. So, i now passes the token to the next processor to allow other processors to send their messages.

In some wrong initial configurations, a message with a wrong destination (*i.e.,* a non existing identifier) may be transmitted in the network. To remove this message, we use a distinguished processor, called the *leader*. The leader is simply a processor in the unidirectionnal ring that does not execute the same code as the other processors (that are called *middle* processors). It is well known [6] that no deterministic uniform self-stabilizing token passing algorithm is possible if the ring is of arbitrary size (as we assumed in this paper). This implies that the self-stabilizing token passing algorithm used as an input to our transformer must be using a processor as the leader. So, the same leader can be used to remove the wrong message from the ring.

The leader processor uses the `seenByLeader` field of the message. First time a data frame passes through the leader, it sets the `seenByLeader` field of the message to true. Then, if the message makes one more complete round in the ring, when it gets back to the leader processor (meaning that the message has been neither delivered to its destination nor

removed by its source), the leader processor removes the messages and replaces this wrong data frame by a token frame.

The fields inside the frames and the variables used by the processors are described in Tables 1 and 2, respectively. The $\mathcal{SSTR}$ algorithm is given in Algorithms 3.1 (for the leader processor) and 3.2 (for the middle processors). Algorithm $\mathcal{SSTR}$ is executed at each processor whenever the processor receives the token from the token passing algorithm.

The code for a middle processor (presented as Algorithm 3.2) is almost the same as the code for the leader, except for the part where the leader checks if it has already seen the message (in case it has, it deletes the message).

The `SendMessage` and `ReceiveMessage` functions are implemented using a thread that is distinct from the one of Algorithms 3.1 and 3.2. We assume that each processor has two dedicated buffers for handling user data (those buffers are different from the buffer that is used to communicate with the communication layer). When `SendMessage` is called, the data to be transmitted is stored in the outgoing data buffer. When the processor is able to transmit data (this occurs in Lines 20-22 for both the leader and middle processors), data from the outgoing buffer is sent to the successor processor and the `SendMessage` function returns. When `ReceiveMessage` is called, the first thing checked is if some data is available in the incoming buffer. If yes, the data is returned immediately by the `ReceiveMessage` function. If no, then the processor waits until the data is available (see Lines 36-39 for the leader processor and Lines 32-35 for the middle processors), then copies incoming data into the incoming buffer, and finally, returns from `ReceiveMessage`.

`isToken`	boolean field; true for a token frame, false for a data frame.
`seenByLeader`	boolean; true if the message has already been seen by the *leader*.
`to`	integer field (6-byte long in Token Ring); the destination of the data frame.
`from`	integer field (6-byte long in Token Ring); the initiator of the data frame.
`data`	raw data bits of the upper application layer message.

Table 1: Frame structure for Algorithm $\mathcal{SSTR}$

`hasToken`	boolean; true if the processor owns the token.
`wantToken`	boolean; true if the processor wants to transmit data.

Table 2: Processor variables for Algorithm $\mathcal{SSTR}$

3.1.1 Correctness Proof of Algorithm $\mathcal{SSTR}$

Definition 3.1 (Legitimate State) *The system is in a legitimate state (in a set denoted by $\mathcal{L}_{\mathcal{SSTR}}$) if and only if there is exactly one frame circulating around the ring and exactly one of the following assertions is verified:*

Algorithm 3.1 Algorithm $\mathcal{SSTR}$ (leader processor)

```
01:  { This code is executed when the processor receives the privilege.  }
02:  Local i ← 0 { i is the index in the destination field }
03:  Local to ← 0 { to is the value of the destination read so far }
04:  InputElement( b ) { Receive Message.isToken }
05:  If Read( b, 0 ) = 1 Then { Received a token frame }
06:     If hasToken = true Then { This case may occur only in case of a transient
failure }
07:        hasToken ← false
08:        OutputElement( b )
09:     Else
10:        If wantToken = true Then
11:           { SendMessage invocation is pending }
12:           hasToken ← true
13:           Forward ( 0 ) { Message.isToken denotes a data frame }
14:        Else
15:           hasToken ← false
16:           Forward ( 1 ) { Message.isToken denotes a token frame }
17:        EndIf
18:        Forward ( 0 ) { Message.seenByLeader is false since this is a new
message }
19:     EndIf
20:     If hasToken = true and wantToken = true Then
21:        Transmit Message Data {  SendMessage returns  }
22:     EndIf
23:  Else { Received a data frame }
24:     If hasToken = true Then { My data frame comes back to me }
25:        hasToken ← false
26:        Forward ( 1 ) { Message.isToken denotes a token frame }
27:        Forward ( 0 ) { Message.seenByLeader is false since this is a new
message }
28:     Else
29:        InputElement( b ) { Receive Message.seenByLeader }
30:        If Read( b, Size( b ) - 1 ) = 0 Then { First time this message is seen
}
31:           While InputElement( b ) and i < DestinationSize 32:  { 6 bytes }
33:              i ← i + 1
34:              to ← ( to × 2 ) + Read( b, Size( b ) - 1 )
35:           EndWhile
36:           If to = Id Then
37:              Deliver Message data 38:  { ReceiveMessage returns }
39:           EndIf
40:           Forward ( 0 ) { Message.isToken denotes a data frame }
41:           Forward ( 1 ) { Message.seenByLeader indicates that the leader saw
the message }
42:        Else { Convert the incorrect data frame into a new token frame }
43:           Forward ( 1 ) { Message.isToken a token frame }
44:           Forward ( 0 ) { Message.seenByLeader is false since this is a new
message }
45:        EndIf
46:     EndIf
47:  EndIf
```

Algorithm 3.2 Algorithm $\mathcal{SSTR}$ (middle processor)

```
01:  { This code is executed when the processor receives the privilege.  }
02:  Local i ← 0 { i is the index in the destination field }
03:  Local to ← 0 { to is the value of the destination read so far }
04:  InputElement( b ) { Receive Message.isToken }
05:  If Read( b, 0 ) = 1 Then { Received a token frame }
06:     If hasToken = true Then { This case may occur only in case of a transient
failure }
07:        hasToken ← false
08:        OutputElement( b )
09:     Else { Normal case }
10:        If wantToken = true Then { Processor wants to send data }
11:  { SendMessage invocation is pending }
12:           hasToken ← true
13:           Forward ( 0 ) { Message.isToken denotes a data frame }
14:        Else
15:           hasToken ← false
16:           Forward ( 1 ) { Message.isToken denotes a token frame }
17:        EndIf
18:        Forward ( 0 ) { Message.seenByLeader is false since this is a new
message }
19:     EndIf
20:     If hasToken = true and wantToken = true Then
21:        Transmit Message Data {  SendMessage returns  }
22:     EndIf
23:  Else { Received a data frame }
24:     InputElement( b ) { Receive Message.seenByLeader }
25:     If hasToken = true and Read( b, Size( b ) - 1 ) = true Then { My data
frame comes back to me }
26:        hasToken ← false { Release the token frame }
27:        Forward ( 1 ) { Message.isToken denotes a token frame }
28:        Forward ( 0 ) { Message.seenByLeader is false since this is a new
message }
29:     Else { Check if the data frame is for me }
30:        While InputElement( b ) and i < DestinationSize
31:  { read 6 bytes }
32:           i ← i + 1
33:           to ← ( to × 2 ) + Read( b, Size( b ) - 1 )
34:        EndWhile
35:        If to = id Then { I am the destination }
36:           Deliver Message data 37:  { ReceiveMessage returns }
38:        EndIf
39:        Forward ( 0 ) { Message.isToken denotes a data frame }
40:        Forward ( Read( b, 1 ) ) { Message.seenByLeader is forwarded as is }
41:     EndIf
42:  EndIf
```

1. *The* `isToken` *bit of the frame is true and the* `hasToken` *bit is false at every processor. (The set of these configurations will referred as* $\mathcal{L}_1$*).*

2. *The* `isToken` *bit of the frame is false and the* `hasToken` *bit is true at exactly one processor* i*, and one of the two following conditions is verified:*

   2.a. *The* `seenByLeader` *bit of the frame is false and the frame is at a processor somewhere between* i *and the* leader *(in clockwise order). (The set of these configurations will be denoted as* $\mathcal{L}_{2a}$*).*

   2.b. *The* `seenByLeader` *bit of the frame is true and the frame is at a processor somewhere between the* leader *and* i *(in clockwise order). (The set of these configurations will be denoted as* $\mathcal{L}_{2b}$*).*

Lemma 3.1 (Closure) $\mathcal{L}_{\mathcal{SSTR}}$ *is closed.*

PROOF : We need to consider all three possible types of legitimate configurations ($\mathcal{L}_1$, $\mathcal{L}_{2a}$, and $\mathcal{L}_{2b}$) as defined in Definition 3.1. Assume that the system is in one of the following three sets of legitimate configurations:

$\mathcal{L}_1$. Either no processor has any data to transmit (so, the system stays in Configuration $\mathcal{L}_1$), or the frame arrives at a processor i which has some data to send ($\texttt{wantToken}_i$ is true). In the latter case, Processor i sets `isToken` and `seenByLeader` to false, so the system is now in Configuration $\mathcal{L}_{2a}$.

$\mathcal{L}_{2a}$. When the frame arrives at the leader, the message was initiated either by the leader (in this case, the leader first delivers the message, then passes the token by setting `isToken` to true, and the system goes back to Configuration $\mathcal{L}_1$) or by another processor (in this case, the leader sets the `seenByLeader` bit of the frame, and the system goes to Configuration $\mathcal{L}_{2b}$).

$\mathcal{L}_{2b}$. The frame is received by its initiator which sets `hasToken` bit to false before passing the token to the next processor. The system then goes back to Configuration $\mathcal{L}_1$.

□

Lemma 3.2 (Convergence) *Starting from any initial configuration, Algorithm* $\mathcal{SSTR}$ *eventually reaches a legitimate state.*

PROOF : A single frame eventually circulates around the ring due to the self-stabilizing property of the token passing algorithm provided as input to the $\mathcal{SSTR}$ algorithm. Let t_0 be the time at which the frame leaves the leader processor.

1. Assume that it is a *data frame*.

1.a. Assume that `seenByLeader` is true. Let j be the first processor in clockwise direction from the frame that has `hasToken` bit set to true. Then, j assumes that it is the source of the data frame, and eventually initiates a token frame that has `seenByLeader` bit set to false. Let i the first processor in clockwise order from j with `hasToken` false and `wantToken` true. i takes the token, sets its `hasToken` to true and initiates a data frame with `seenByLeader` false. As the data frame is being forwarded around the ring, all processors between i and the leader set their `hasToken` bit to false. We now consider two cases corresponding to the leader's `hasToken` bit value.

1.a.α. Assume that the leader's `hasToken` bit is true. Upon receiving the data frame, the leader removes the data frame and sends a new token frame clearing its `hasToken` bit. The system now is in $\mathcal{L}_{2b}$.

1.a.β. Assume that the leader's `hasToken` bit is false. The system is already in $\mathcal{L}_{2b}$.

1.b. Assume that `seenByLeader` is false. Then no middle processor will be able to convert the data frame into a token frame, and they all will set their `hasToken` bit to false.

1.b.α. Assume that the leader's `hasToken` bit is true. Upon receiving the data frame, the leader removes the data frame and sends a new token frame clearing its `hasToken` bit. The system now is in $\mathcal{L}_{2b}$.

1.b.β. Assume that leader's `hasToken` bit is false. The leader now sets `seenByLeader` to true and forwards the frame. Then when the frame comes back to the leader after one traversal around the ring, we arrive at Case 1.a.β.

2. Assume that it is a *token frame*. Let i be the first processor which has `hasToken` false and `wantToken` true. Every processor between the leader and i sets its `hasToken` to false. i then initiates a data frame with `seenByLeader` set to false. As the data frame is being forwarded around the ring, all processors between i and the leader set their `hasToken` bit to false.

2.a. Assume that the leader's `hasToken` bit is true. We reach Case 1.a.α.

2.b. Assume that leader's `hasToken` bit is false. We reach Case 1.a.β.

□

Theorem 3.1 *The $\mathcal{SSTR}$ protocol is self-stabilizing.*

PROOF : Follows from Lemmas 3.1 and 3.2. □

Theorem 3.2 *Algorithm $\mathcal{SSTR}$ satisfies cut-through constraints.*

PROOF : First, note that `OutputElement` is called only when `InputElement` returns a `SignalElement` different from `N`. So, the producer consumer constraint is satisfied.

Then we prove the rate constraint of the algorithm is also satisfied. The leader's input and output actions match the following pattern:

$$\texttt{input}\left(\texttt{output}|\texttt{output}^2|\texttt{input}^{\texttt{DestinationSize}+1},\texttt{output}^2\right)$$

Any middle processor's input and output actions match the following pattern:

$$\texttt{input}\left(\texttt{output}|\texttt{input},\texttt{output}^2|\texttt{input}^{\texttt{DestinationSize}+1},\texttt{output}^2\right)$$

Both patterns are subsets of

$$\texttt{input}^+(\texttt{output},\texttt{input})^*\texttt{output}^+$$

So, Algorithm $\mathcal{SSTR}$ satisfies the cut-through constraints. □

3.1.2 Complexity of Algorithm $\mathcal{SSTR}$

Space Complexity. Every processor uses two boolean variables. So, the space requirement is two bits per processor. Each message has two address variables (six bytes each in Token Ring) and two boolean variables. Assuming the number of processors in the ring to be n, the address size is $O(\log_2 n)$ bits. Therefore, the message size is $O(\log_2 n)$ bits (excluding the data size of the application layer).

Time Complexity.

Definition 3.2 (Maximum Waiting Time) *The maximum waiting time, T_w is defined as the maximum time a processor may need to wait between sending two consecutive data frames.*

Proposition 3.1 (Token Ring Waiting Time) *After stabilization, the maximum waiting time of Algorithm $\mathcal{SSTR}$ is given by the following relation:*

$$T_w = n \times (T_{t_d} + T_r) + T_r$$

where T_{t_d} is the time needed to transmit a data frame.

PROOF : Let t_0 be the time when Processor i starts transmitting a data frame. The transmission takes T_{t_d}, and the last bit of the data frame returns to i in T_r. A new token frame is then initiated by i to Processor $i+1 \bmod n$, and converted to a new data frame by Processor $i+1 \bmod n$. This takes

$$T_{p_{i \leftarrow i+1 \bmod N}} + T_{d_{i+1 \bmod N}}$$

time before a new processor can send its data frame. There are n processors in the network, and by definition of T_r, the result of the proposition follows. □

Proposition 3.2 (Self-stabilization Overhead) *The overhead in time incurred due to the addition of the self-stabilizing property in the algorithm is*

$$T_{o_{ss}} = N \times T_{t_1}$$

where T_{t_1} is the transmission time needed to transmit one bit.

PROOF : Only one extra bit (`seenByLeader` bit) is added to every message. This bit does not cause any extra delay while forwarding messages, because the buffer needs to be only one bit size. However, it causes extra time to transmit the bit. The result follows from Proposition 3.1. □

The following result follows from Proposition 3.1:

Proposition 3.3 (Token Passing Overhead) *The extra time spent due to the underlying self-stabilizing token passing protocol is*

$$T_{o_{me}} = N \times (T_{t_{me}} + T_{r_{me}}) + T_{d_{me}}$$

where $T_{t_{me}}$ is the time to transmit the token passing bits, and $T_{d_{me}}$ is the time delay (corresponding to the buffer delay) needed to forward the token passing bits.

3.2 Alternate Counter Flushing ($\mathcal{ACF}$) Algorithm

Let us briefly recall the counter flushing scheme used to model Dijkstra's algorithm (as presented in [15]). In this scheme, every processor maintains a counter. The leader (or distinguished) processor of the ring sends a message with a *new* counter value. The leader saves this counter value in a local counter variable. In a good system state, the value of the counter would be something different than any value existing in the ring currently *i.e.,* different than the value of the counter field of any message in transit or the counter value of any other processor. A non-leader (that we call middle) processor i upon receiving the message simply forwards it to the next processor (in clockwise direction). In addition, if the counter in the message is new, the middle processor enters the critical section before forwarding the message. After a complete traversal of the message with the new counter value, the leader receives the message with a counter value equal to its own counter value. Then the leader changes the counter value to a new value and repeats the above process. In summary, the single privilege is passed around the ring. The leader is privileged if it receives a message with a counter value same as its own counter variable, whereas the middle processors are privileged if the counter value in the message is different than their own counter value.

In [15], the counter is not only used to circulate the privilege, but also to stabilize the token passing algorithm. If the counter size is bigger than the maximum number of messages that can be in all the links of the network (this number depends on the link capacity) plus the number of different counter values at the processors (every processor holds exactly one counter value), eventually all the initial corrupted messages will be flushed out of the network.

From the above description, we observe that the counter is used in [15] to achieve two goals — to stabilize the network and to circulate the privilege in the network. Our main idea is to suggest an alternate scheme, called *Alternate Counter Flushing* ($\mathcal{ACF}$) scheme where messages with the counter are used only to stabilize the network. However, to pass the privilege around the ring, we send a message with no counter. We call the messages with (resp., without) counter *counter_message*(resp., *light_message*).

The motivation behind this revised counter flushing scheme is the following: We expect the frequency of transient faults in a network to be moderate or low. This implies that the *light_message*s will be used much more often than the *counter_message*s. So, effectively,

we save the overall message size, thereby saving the number of bits transmitted. This is specially important in the cut-through model. After stabilization, our algorithm has less round-trip delay as compared to the counter-flushing scheme based algorithm [15]. However, the stabilization time of $\mathcal{ACF}$ can be higher than that of [15]. As mentioned earlier, the system is expected not be stabilized too often.

In Algorithm $\mathcal{ACF}$, the value of the first field distinguishes the two types of messages (*light_message* and *counter_message*). It is a single bit boolean field (`hasCounter`) — true represents a *counter_message* message. The leader sends a *counter_message* followed by `MaxLight` *light_message*s. (Later, we will discuss the bound for `MaxLight` to achieve stabilization.) A down counter (`nbLight`) is used to send `MaxLight` *light_message*s. If the system is in a good configuration, it is obvious that if the leader sends a *counter_message* (resp., *light_message*), it should receive a *counter_message* (resp., *light_message*) next time because the message would come back to the leader after traversing the whole ring. If the system is not stabilized yet, the leader may receive some unexpected type of messages. In that case, the leader simply drops those messages. A boolean variable (`lightExpected`) is used by the leader to implement this. We assume that the leader has a buffer that is sufficient to hold a counter plus one additional bit.

The data structure of the leader is shown in Tables 3 and 4. The message format is described in Table 5. Algorithm $\mathcal{ACF}$ is shown in Algorithms 3.4 (for the leader processor) and 3.3 (for the middle processors).

`MaxC`	integer constant; the number of different counter values.
`CounterSize`	integer constant; the size (in bits) of `MaxC`
`MaxLight`	integer constant; the number of light messages sent by the *leader* between two successive messages with counter.

Table 3: Leader processor constants for $\mathcal{ACF}$ protocol

`nbLight`	integer variable; the number of light messages to be sent by the *leader* before sending the next counter message.
`lightExpected`	boolean variable; true if the leader is expecting a light message.
`counter`	integer variable; the value ranges between 0 and `MaxC` $-$ 1.

Table 4: Leader processor variables for $\mathcal{ACF}$ protocol

`hasCounter`	boolean field; true if the message is a counter message.
`counter`	integer field; contains the counter of the leader. Note that this field exists only if `hasCounter` is `true`.

Table 5: Message structure for $\mathcal{ACF}$ protocol

We now give an informal explanation of the execution of our algorithm. Periodically, the leader node sends a counter message. The counter in the message and at the leader increases

Algorithm 3.3 The MessageHandler $\mathcal{ACF}$ function at middle processor

```
01:  Local i ← 0 { i is the index in the counter }
02:  Local mcount ← 0 { mcount is the value of the counter read so far }
03:  { Buffer b has been cleared at the begining of MessageHandler }
04:  If InputElement( b ) { Receive Message.hasCounter }    Then
05:     If Read( b, 0 ) = 1 Then { counter_message received }
06:        OutputElement( b ) { Forward a counter_message }
07:        While InputElement( b ) and i < CounterSize
08:           { Receive and forward next bit of Message.counter }
09:           i ← i + 1
10:           Read( b, 0 )
11:           OutputElement( b )
12:        EndWhile
13:     Else { Privileged, enter critical section }
14:        OutputElement( b ) { Forward a light_message }
15:     EndIf
16:  EndIf
```

every time a fresh counter message is sent. Middle processors simply formard the counter messages to their successors (in clockwise direction). When the counter message comes back to the leader and has the same value as its own, the leader sends a light message. When a middle processor receives a light message, it enter its critical section, exits the critical section, and then forwards the light message to its successor. When the light message comes back to the leader, the same process (for light messages) is repeated `MaxLight` times before the leader starts sending a new counter message. The stabilization comes from the fact that eventually, a counter message will be issued that has a new counter value in the whole network such that all remaining messages (that were present due to some transient failures) are flushed from the system. Note that during the stabilization process, most of the time is spent sending light messages that have less overhead than the counter messages.

3.2.1 Correctness Proof of Algorithm $\mathcal{ACF}$

Definition 3.3 (Privilege) *A processor has a privilege if and only if one of the following conditions is satisfied:*

1. *the processor is a middle processor and has just received a* light_message(i.e. *its program counter corresponds to line number `13` in the presented code),*
2. *the processor is the leader and has just received an* expected light_message(i.e. *its program counter corresponds to line number `21` in the presented code).*

Definition 3.4 (Potential privilege) *A processor owns a potential privilege if receiving the next frame makes the processor privileged.*

Algorithm 3.4 The MessageHandler function at the leader processor

```
01:  Local i ← 0 { i is the index in the counter }
02:  Local mcount ← 0 { mcount is the value of the counter read so far }
03:  { Buffer b has been cleared at the begining of MessageHandler }
04:  If InputElement( b ) { Receive Message.hasCounter }   Then
05:     If Read( b, 0 ) = 1 Then { counter_message received }
06:        If lightExpected = false Then { A counter_message was expected }
07:           While InputElement( b ) and i < CounterSize
08:              { Receive next bit of Message.counter }
09:              i ← i + 1
10:              mcount ← ( mcount × 2 ) + Read( b, 0 )
11:           EndWhile
12:           If count = mcount Then { Switch to light_message }
13:              count ← count + 1 mod MaxC
14:              nbLight ← nbLight - 1 mod MaxC
15:              OutputElement( b )
16:              lightExpected ← true
17:           EndIf
18:        EndIf
19:     EndIf
20:  Else { light_message received }
21:     If lightExpected = true Then { Privileged, enter critical section }
22:        If nbLight = 0 { Switch to counter_message }
23:           nbLight ← MaxLight
24:           Forward( 1 ) { As Message.hasCounter }
25:           Forward( count ){ As Message.counter }
26:           lightExpected ← false
27:        Else { Continue to send light_message }
28:           nbLight ← nbLight - 1 mod MaxC
29:           Forward( 0 ){ As Message.hasCounter }
30:           lightExpected ← true
31:        EndIf
32:     EndIf
33:  EndIf
```

Definition 3.5 (Legitimate state) *The system is in a legitimate state (denoted by the set $\mathcal{L}_{\mathcal{ACF}}$) if and only if the following conditions are satisfied:*

1. *There is a single frame in the network.*
2. *There is a single processor with a privilege or a potential privilege.*

Assumption 3.1 *Following the result of Gouda and Multari [8], we assume that the leader may initiate a timer to send a new frame when the network is frame-free.*

This implies the following property:

Property 3.1 (Liveness) *Starting from any configuration, in every computation, every processor sends a frame infinitely often.*

We define a particular sequence of configurations (called *cycle*) to characterize legitimate configurations.

Definition 3.6 (Cycle) *A cycle is a minimal sequence of configurations such that the first and last configurations are equal.*

Property 3.2 *Starting from a configuration where the leader has just sent one frame, the network cannot become frame-free.*

PROOF : Let t_0 be the time when the *leader* sends a new frame. Then the leader expects to receive this frame back within $t_0 + T_r$. However, before this frame arrives at the leader, some unexpected messages may be received by the leader. By the algorithm, the leader drops all those unexpected frames. Finally, upon receiving the expected message, the leader send a new frame, and the system is back to the starting configuration. □

Property 3.3 *All possible changes to the number of frames in the network are defined by the following properties:*

1. *The number of frames in the network decreases if and only if one of the following is true:*
   - *(a) the* leader *receives a* counter_message *when expecting a* light_message*.*
   - *(b) the* leader *receives a* light_message *when expecting a* counter_message*.*
   - *(c) the* leader *receives a* counter_message *when expecting a* counter_message *but the counter message does not match its local counter.*
2. *The number of frames remains unchanged if and only if the next frame to be received by the leader is expected by the leader.*
3. *The number of frames in the network will increase if and only if the network becomes frame-free.*

PROOF : We prove successively each of these properties:

1. In this case, the leader drops the frame.
2. In this case, the leader forwards the frame or sends a new frame; otherwise, the frame is dropped.
3. The proof follows from Assumption 3.1.

□

Property 3.4 *Any execution has exactly one cycle.*

PROOF :

Existence: The number of configurations is finite (bounded variables, bounded FIFO channels, and bounded size messages). Therefore, there exists a configuration which appears more than once in every computation. Moreover, since the algorithm is deterministic, any infinite computation has a cycle.

Uniqueness: This follows from the fact that the algorithm is deterministic.

□

Property 3.5 *A cycle can never become frame-free.*

PROOF : Follows from Assumption 3.1 and Property 3.2. □

Property 3.6 *Any cycle satisfies the following two properties:*

1. *The number of frames in the cycle remains unchanged.*
2. *The leader receives only the expected frames.*

PROOF : Follows from Property 3.3. □

We have so far established that there is exactly one cycle in the network. Moreover, we also know that this unique cycle maintains a *fixed* number of frames. So, in order to prove that the leader receives only expected frames, all we have to show now is that the cycle eventually will contain a single frame.

Lemma 3.3 *The only possible cycle in any execution contains a single frame if and only if* $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ *is greater than the number of messages initially present in the network.*

PROOF : In order to prove this proposition, we proceed in two steps:

1. We need to prove that if $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ is less than or equal to the number of frames initially present in the network, there exists a cycle which contains $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ frames. To prove that, it is sufficient to consider the following example, where the *leader* variables are shown in Table 6, and $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ frames to be received by the *leader* given in Table 7.

   With such an initial configuration, after $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ frames are received and sent by the *leader*, the system goes back to the initial configuration.

nbLight	MaxLight
lightExpected	false
counter	0

Table 6: Leader variables

Number of frames	Type	Counter (if *counter_message*)
1	Counter	0
MaxLight	Light	
1	Counter	1
MaxLight	Light	
⋮	⋮	⋮
MaxLight	Light	
1	Counter	MaxC − 1
MaxLight	Light	
1	Counter	0

Table 7: Frames forming a cycle

2. We want to prove that if $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ is greater than the number of frames initially present in the network, only cycles with exactly one frame can exist.

   Assume that there is more than one frame in the cycle. By Property 3.6, the leader only receives expected frames. The leader eventually sends a *counter_message* with a unique counter in the network since there are less than $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ frames. If this unique *counter_message* is not the only frame in the network, other frames are dropped by the leader, which is impossible by Property 3.6.

□

Lemma 3.4 *The set of the configurations in a cycle with a single frame (denoted by O_1) is the set of legitimate configurations.*

PROOF : We want to prove that each legitimate configuration belongs to a cycle with a single frame. By Definition 3.5, in any legitimate configuration, there is a single frame in the network. Since the number of frames remains the same and the algorithm is deterministic, a cycle is performed.

We need to prove that each configuration in a cycle with a single frame is a legitimate configuration. By Property 3.6, this single frame must be the frame expected by the leader. When the leader receives this frame, the leader is privileged, and it passes the privilege on to the next processor. In turn, this processor is privileged, and passes its privilege to the next processor. Since the network is finite and the algorithm is deterministic, a cycle is eventually formed and all configurations in the cycle are legitimate. □

Closure and convergence follow from Lemmas 3.3 and 3.4.

Lemma 3.5 (Closure) *The set of legitimate states of Algorithm $\mathcal{ACF}$ is closed.*

Lemma 3.6 (Convergence) *If $(\texttt{MaxC} \times (\texttt{MaxLight} + 1) + 1)$ is greater than the number of messages initially present in the network,* $\mathsf{true} \triangleright \mathcal{L}_{\mathcal{ACF}}$.

Theorem 3.3 *Algorithm $\mathcal{ACF}$ is self-stabilizing.*

PROOF : Follows from Lemmas 3.5 and 3.6. □

Theorem 3.4 *Algorithm $\mathcal{ACF}$ satisfies cut-through constraints.*

PROOF : First, note that `OutputElement` is called only when `InputElement` returns a `SignalElement` different from `N`. So, the producer consumer constraint is satisfied.

Then we prove the rate constraint of the algorithm is also satisfied. The leader's input and output actions match the following pattern:

$$\texttt{input}\left(\texttt{input}^{\texttt{CounterSize}}, \texttt{output} | \texttt{output}^{\texttt{CounterSize+1}} | \texttt{output}\right)$$

Any middle processor's input and output actions match the following pattern:

$$\texttt{input}\left(\texttt{output}\,(\texttt{input}, \texttt{output})^{\texttt{CounterSize}}\,|\texttt{output}\right)$$

Both patterns are subsets of

$$\texttt{input}^{+}(\texttt{output}, \texttt{input})^{*}\texttt{output}^{+}$$

So, Algorithm $\mathcal{ACF}$ satisfies the cut-through constraints. □

3.2.2 Complexity of Algorithm $\mathcal{ACF}$

Space Complexity. The leader node has a counter of size `MaxC`, a counter of size `MaxLight`, and a boolean variable. So, the middle processors have 0 bits and the leader processor has $O(\log_2(\texttt{MaxC} \times \texttt{MaxLight}))$ bits.

Time Complexity. The self-stabilizing token passing overhead is given by:

$$T_{o_{me}} = N \times (T_{t_{me}} + T_{d_{me}}) + T_{d_{me}}$$

In [15], $T_{t_{me}} = T_{t_{32}}$ since a 32 bit counter is chosen, and $T_{d_{me}} = T_{d_{32}}$ since the extra delay occurs only at the leader processor.

Proposition 3.4 *In the $\mathcal{ACF}$ algorithm, the average time needed to transmit the token passing bits is:*

$$T_{t_{me}} = \frac{T_{t_{32}} + \texttt{MaxLight} \times T_{t_1}}{\texttt{MaxLight} + 1}$$

PROOF : A *counter_message* is sent only every `MaxLight` *light_message*s. □

Proposition 3.5 *In the $\mathcal{ACF}$ algorithm, the average time needed to forward the token passing bits is:*

$$T_{d_{me}} = \frac{T_{d_{32}} + \texttt{MaxLight} \times T_{d_1}}{\texttt{MaxLight} + 1}$$

PROOF : The extra delay due to the *counter_message* occurs at the *leader* once every `MaxLight` *light_message*s. □

Proposition 3.6 *In Algorithm $\mathcal{ACF}$, the average token passing overhead is:*

$$T_{o_{me}} = (2 \times N + 1) \times \left(\frac{T_{t_{32}} + \mathit{MaxLight} \times T_{t_1}}{\mathit{MaxLight} + 1} \right)$$

PROOF :

$$T_{o_{me}} = N \times \left(\frac{T_{t_{32}} + \texttt{MaxLight} \times T_{t_1}}{\texttt{MaxLight} + 1} + \frac{T_{d_{32}} + \texttt{MaxLight} \times T_{d_1}}{\texttt{MaxLight} + 1} \right) + \frac{T_{d_{32}} + \texttt{MaxLight} \times T_{d_1}}{\texttt{MaxLight} + 1}$$

We assume that $T_{t_{32}} = T_{d_{32}}$ and $T_{t_1} = T_{d_1}$. So the result can be simplified as:

$$T_{o_{me}} = (2 \times N + 1) \times \left(\frac{T_{t_{32}} + \texttt{MaxLight} \times T_{t_1}}{\texttt{MaxLight} + 1} \right)$$

□

Proposition 3.7 *In algorithm [15], the average token passing overhead is:*

$$T_{o_{me}} = (2 \times N + 1) \times (T_{t_{32}})$$

PROOF :

$$T_{o_{me}} = N \times (T_{t_{32}} + T_{d_{32}}) + T_{d_{32}}$$

We assume that $T_{t_{32}} = T_{d_{32}}$. So, the result can be simplified as:

$$T_{o_{me}} = (2 \times N + 1) \times (T_{t_{32}})$$

□

By proposition 3.6, with our approach, the average overhead due to self-stabilization can be as low as

$$(2 \times N + 1) \times (T_{t_1})$$

whereas the overhead of Algorithm $\mathcal{CF}$ is at least

$$(2 \times N + 1) \times (T_{t_{32}})$$

Thus, we improve the overhead due to self-stabilization by a factor of 32. However, not sending a counter with every frame slows down the stabilization time by a factor of `MaxLight`. Therefore, the $\mathcal{ACF}$ solution is well suited for networks where corruptions do exist but are rare.

4 A Self-stabilizing Census Algorithm in the Cut-through Model

In this section, we propose a self-stabilizing census algorithm (called Algorithm $\mathcal{CCT}$) in a ring network using the cut-through routing scheme. This algorithm satisfies Specification $\mathsf{Spec}_{\mathcal{CCT}}$ assuming that the Exists function is implemented on top of our algorithm.

We now describe the basic idea of the algorithm. Every processor forwards every message it receives forever. These messages ideally should contain the identifiers of all the processors in the network. When a processor receives a message that does not contain its own identifier, the processor adds its identifier at the end of the message. In this process, every message eventually contains at least the identifiers existing in the network. However, the identifiers which do not exist in the network need to be removed. This task is implemented using one `Nb` field for every identifier in the message. `Nb` field corresponding to an existing identifier i in a message m holds the number of processors visited by m since it was reset to 0 by i. So, the maximum value of any `Nb` should be $N-1$. However, due to wrong initialization, a message m may contain an invalid identifier j. After one complete round of m in the ring, `Nb` of j will become at least N. At that point, the next processor visited by m will detect the fault and will start the re-initialization of m using the `ZeroMarker` technique discussed in Section 2, thereby removing the invalid identifier j from m. When a processor receives a message that starts with a `ZeroMarker`, it issues a new message containing only its identifier. So, this new message contains only valid identifier, and eventually gets filled up with all valid identifiers in the network.

We now give further details about the implementation of Algorithm $\mathcal{CCT}$ followed by its proof of correctness.

Message Forwarding. In this section, we describe the message forwarding scheme under the cut-through constraints. A complete record contains `Id` and `Nb` fields. We will indicate the size of a complete record by `RecSize`. Message forwarding is implemented using two functions: `Receive` and `Send`.

The main purpose of `Receive` is to read the records from the communication link and write them in the buffer. This function uses a boolean variable `first` whose value indicates whether or not this is the first call to the `Receive` function. If it is the first call, `SignalElement`s are read from the incoming link and written in `b` provided `b` is not full and the the incoming bit is not a `ZeroMarker`. If a record can be read from `b`, `true` is returned. Otherwise, `false` is returned. If this is not the first invocation of `Receive`, this function checks if there is a record in `b` to be read. If there is any such record, it implies that `Send` function must have been called earlier, which wrote the record in the buffer. In this case, `Receive` returns `true`. The `Receive` function can be coded as described in Algorithm 4.1.

`Send` forwards a record and also reads the following record in the buffer if any record is in the incoming channel. The `Send` function implements the cut-through constraints. The `Send` function can be coded as described in Algorithm 4.2.

Algorithm 4.1 `Receive` Function of Algorithm $\mathcal{CCT}$

```
01:  If first = true Then { first call to Receive }
02:     Local i ← 0 { i is the index in the current record }
03:     first ← false ; Clear( b )
04:     While i < RecSize { The record is not completely read yet. }
05:        If InputElement( b ) Then
06:           If Read(b,Size(b)-1) = Z Then { Last read element is a ZeroMarker }
07:              Clear(b) { void buffer }
08:              Return false { no record is available }
09:           Else { Input a SignalElement into buffer }
10:              i ← i +1 { Skip to next element }
11:           EndIf
12:        Else { SignalElement is not 0, not 1, not Z }
13:           Clear( b )
14:           Return false
15:        EndIf
16:     EndWhile { Loop only if received 0 or 1 }
17:     Return true { Buffer contains a new record }
18:  Else { Called before }
19:     Return Size(b) ≥ RecSize { Record was received by Send function }
20:  EndIf
```

Algorithm. The census algorithm has three main tasks. We describe these tasks below for processor i.

Checking the Completeness If the message is not a complete message, i.e., some fields in some records seem to be missing, processor i uses a `ZeroMarker` to destroy this message and initiate a new one. Also, if a processor notices that its identifier is not included in the message (using the local variable `amIHere`), it appends a record to the message after forwarding the rest of the message.

`Nb` Correction We want to ensure that no record for a non-existent processor lasts forever in the network. Once all processors have added their records in the message, each processor i checks how many processors have been visited by this message since i last zeroed i.`Nb`. The value of i`.Nb` should satisfy the following two conditions: (i) It is equal to the maximum `Nb` in the message and (ii) It is equal to the number of processors recorded in the message minus one. (This number is calculated using a local variable `NbProcessorV`.) If any of the above conditions is *false*, it implies that the message contains the records of some non-existent processors. In that case, i uses a `ZeroMarker` to destroy this message and initiate a new one. When the new message is initiated by a processor, in one round, the message will contain all identifiers.

Implementing Exists function This aspect is not given in detail here, but the implementation is straightforward. One can either store one complete message at each processor

Algorithm 4.2 `Send` Function of Algorithm $\mathcal{CCT}$

```
01:  Local i ← 0{ i is the index in the current record }
02:  Local shouldOutput ← false { true if SignalElements are waiting to be
received }
03:  If first = false Then { Receive has been called at least once }
04:     If InputElement( b ) Then { Some elements are waiting }
05:        shouldOutput ← true
06:        If Read(b,Size(b)-1) = Z Then { Last read element is a ZeroMarker }
07:           Write(b,0,Z) { Advance ZeroMarker by RecSize }
08:        EndIf
09:     EndIf
10:  EndIf
11:  While i < RecSize { Process the current record }
12:     OutputElement( b )
13:     If InputElement( b ) Then
14:        If Read(b,Size(b)-1) = Z Then 15:  { Last read element is a ZeroMarker
}
16:           Write(b,0,Z) { Advance ZeroMarker by RecSize }
17:        EndIf
18:     EndIf
19:     i ← i +1 { Skip to the next element }
20:  EndWhile
21:  If shouldOutput = true Then
22:     OutputElement( b )
23:  EndIf
```

(losing many of the benefits of the Cut-though scheme) and have Exists function calculate from this stored message, or wait for a message to be passed through the processor and check if the `Id` passed to Exists function is present. We use the latter approach for the complexity results.

Algorithm 4.3 MessageHandler of Algorithm $\mathcal{CCT}$

```
01: Local amIHere ← false; currentSize ← 0
02: Local NbProcessorsV ← 0; first ← true
03: While Receive(p) { Update local variables }
04:    { If Receive returns true, then there is at least one complete record }
05:    NbProcessorsV ← NbProcessorsV +1
06:    If p.Id = i.Id Then
07:       If amIHere = true Then
08: { Send a ZeroMarker if already present }
09:          Forward(ZeroMarker)
10:       Else
11:          amIHere ← true; currentSize ← p.Nb
12:          Forward([p.Id,0]) { Reset Nb field }
13:       EndIf
14:    Else { Forward the same record with updated Nb field }
15:       Forward([p.Id,p.Nb+1])
16:    EndIf
17: EndWhile
18: If amIHere = false Then { Forward the original record }
19:    Forward([i.Id,0])
20: Else
21:    If currentSize ≠ NbProcessorsV −1 { Nb correction }
22:       Forward(ZeroMarker)
23:    EndIf
24: EndIf
```

4.1 Proof of Census Algorithm $\mathcal{CCT}$

We first prove the liveness of the algorithm. Messages may be discarded in the cut-through routing scheme. We must ensure that eventually, there is at least one message circulating in the ring. To prove this, we first show that the system can always hold at least one legitimate message and then, show that even after the initial bad messages are detected and destroyed, the system still will have at least one message. Then, we show that when badly initialized messages are discovered, new empty messages are issued. However, this task may not be completed atomically in the Cut-Through model. We prove that eventually, every message carrying the `ZeroMarker` is destroyed and a new good message is initiated. Also, we show

that our algorithm satisfies the producer-consumer and rate constraints, as stated in Section 2. Next, we prove that in finite time, any message on the ring will contain a correct record for every processor in the ring.

In the following, $m(i)$ denotes the record corresponding to processor i in message m.

Definition 4.1 (Legitimate Message) *A record in a message m is legitimate if its fields satisfy the following conditions:* (i) *The `Id` field is equal to the `Id` of a processor $i \in \mathcal{P}$ in the network.* (ii) *The `Nb` field is equal to the number of processors visited by m after visiting processor i, i.e., the distance from i. A message m is legitimate if m has exactly N different legitimate records.*

Note that a legitimate message contains a legitimate record for every processor in the network. Thus, it contains N values of `Id`.

Definition 4.2 (Legitimate Configuration) *The set of legitimate configurations $\mathcal{L}_{\mathcal{CCT}}$ of Algorithm $\mathcal{CCT}$ is such that in each configuration $c \in \mathcal{L}_S$, the following conditions hold:* (i) *There is at least one message in the ring.* (ii) *All messages in the ring are legitimate messages.*

Lemma 4.1 *The network can hold at least one legitimate message.*

PROOF : A legitimate message is N records long, plus `S` and `E SignalElement`s. So, the size of a legitimate message is $ms = N \times$`RecSize`$+2$. Each processor has a `Buffer` whose size is greater than a record size. Each communication link can hold at least 1 `SignalElement`. Thus, the network can hold $ns = N \times$ (`RecSize`$+2$) `SignalElements`. For $N \geq 2$, $ns > ms$. □

Definition 4.3 (Message Destruction) *A message is destroyed in execution e if and only if e contains a sequence of configurations from the set $\mathcal{C}$: $c_1, c_2, \ldots, c_{k-1}, c_k$, such that c_1 contains n `S SignalElements` and c_k contains $n-1$ `S SignalElements`.*

Remark 4.1 *A message can be destroyed only in a `Receive` or `Send` function. If a processor receives a new message, m_2 when it is executing the `Send` function, i.e., forwarding a record of the previous message m_1, the new message m_2 is destroyed. If a processor receives a message m_2 when it is executing the `Receive` function on another message m_1, the new message m_2 is destroyed.*

Remark 4.2 *If a processor receives an incomplete message, the last record of this message is not stored in the `Buffer`. If a processor receives a message containing a `ZeroMarker` in the first `RecSize` bits, then the message is considered empty, i.e., the message is destroyed. If a processor receives a message containing a `ZeroMarker`, but not in the first `RecSize` bits, then the `ZeroMarker` is forwarded and advanced by `RecSize` positions.*

Lemma 4.2 *Eventually, a message containing a `ZeroMarker` is destroyed or initialized.*

PROOF : From Remark 4.1, a message containing a `Zeromarker` may be destroyed. From Remark 4.2, if the message is not destroyed, the `ZeroMarker` advances by `RecSize` positions every time it visits processor. Eventually, the `ZeroMarker` will advance enough to be in the first `RecSize` bits of the message. Then a new message is initialized. □

Theorem 4.1 (Cut-through) *The $\mathcal{CCT}$ algorithm satisfies the cut-through constraints.*

PROOF : We first prove that the Producer-Consumer constraint is satisfied. Before the function `Send` is called, `Buffer b` is filled with `RecSize` bits. In the function `Send`, the following two situations may occur: (i) If the first call to `InputElement` succeeds (i.e., returns 0 or 1), `Send` calls `OutputElement` total `RecSize`+1 times. (ii) Otherwise, `Send` calls `OutputElement` total `RecSize` times. In both situations, `Buffer` holds enough `SignalElement`s that `OutputElement` never returns `false`. Thus, no `N SignalElement` is sent to the channel.

Then we prove that **Rate** constraint is satisfied. The calls to the functions `Receive` and `Send` follow the following rational expression: $(\texttt{Receive}, \texttt{Send})^*\texttt{Send}^*$. The first `Receive` call can be reduced to $\texttt{input}^+$. Subsequent `Receive` calls can be reduced to an `empty` expression. Calls to `Send` can be reduced to:

$$(\texttt{input}, \texttt{output})^{\texttt{RecSize}}(\texttt{input}, \texttt{output})?$$

Thus the overall expression becomes:

$$\texttt{input}^+((\texttt{input}, \texttt{output})^{\texttt{RecSize}}(\texttt{input}, \texttt{output})?)^+$$

which is a subset of the rate constraint. □

Theorem 4.2 (Closure) *The set of legitimate configurations $\mathcal{L}_{\mathcal{CCT}}$ is closed.*

PROOF : First, we prove that every legitimate message remains legitimate. Let m be a legitimate message arriving at processor k. Since m is legitimate, every `Nb` field in m is equal to the distance to the corresponding processor. So, `Nb` of processor k is equal to $N-1$ and is the maximum among all `Nb` fields in m. Processor k sets k.`Nb` to 0 and increments other `Nb` fields by one. Thus, the message remains legitimate.

Then, we prove that starting from a legitimate configuration, no message can be lost. In a legitimate configuration, all messages are legitimate and contain N different records (one for each processor). Thus, there are no `ZeroMarker`s in the messages, and no processor may add its `Id` to any of these messages. Then, by Remark 4.1, no messages can be destroyed. □

Definition 4.4 (Attractor) *A set of configurations $\mathcal{B}'$ is an attractor for a set of configurations $\mathcal{B}$ of Algorithm $\mathcal{CCT}$, if for any configuration in $\mathcal{B}$ and any computation of $\mathcal{CCT}$, a configuration of $\mathcal{B}'$is reached. This relation is denoted is $\mathcal{B}' \triangleleft \mathcal{B}$.*

We now define $\mathcal{C}_1$ as the set of configurations such that there is at least one message in the network, and every message contains only those `Id`s which correspond to the `Id`s of some existing processors in the network. Moreover, any `Id` appears in a message only once. It is obvious that $\mathcal{L}_{\mathcal{CCT}} \subset \mathcal{C}_1 \subset \mathcal{C}$. So, to prove the convergence property of Algorithm $\mathcal{CCT}$, our obligation is show that $\mathcal{L}_{\mathcal{CCT}} \triangleleft \mathcal{C}_1 \triangleleft \mathcal{C}$.

Lemma 4.3 $\mathcal{C}_1 \triangleleft \mathcal{C}$

PROOF : First we ensure that the content of each message that is not destroyed eventually matches the specification of $\mathcal{C}_1$. There are two cases to consider:

1. Assume that there exists no message in the network. Then one timer exists to send a new message. As this new message visits the processors, the message will grow up to $N \times \texttt{RecSize} + 2$, and thus, can stay in the network. By hypothesis, this message will not be destroyed and we reach a configuration $c \in \mathcal{C}_1$.

2. Assume that there exists at least one message m in the network. We need to consider two situations:

2.a. m contains `Id`s that do not correspond to any processor identifiers. Then by Lemma 4.2, this message is either destroyed or initialized. If the message is destroyed, the same reasoning as in Case 1 applies. Otherwise, Case 2.b is true.

2.b. m contains all valid `Id`s. The duplicate identifiers are removed by forwarding a `ZeroMarker`. This will cause a processor to initialize a new message. Then eventually no message contains duplicate or bad `Id`s. If at least one message is not destroyed, then we reach a configuration $c \in \mathcal{C}_1$.

Then we prove that starting from a configuration $c \in \mathcal{C}_1$, the system will never reach a configuration where there is no message in the network. Assume the contrary, *i.e.* there exists such a computation, e. Then a configuration with a single message m exists in e. This message can be eliminated only if a processor i receives the `S SignalElement` of m in one of the following two situations:

A. i is `Send`ing m. From Lemma 4.1, there is enough space to hold a legitimate message. This situation cannot occur since no message in $\mathcal{C}_1$ can be larger than a legitimate message.

B. i is appending a record to m. This situation may only occur if another record (the record of i) cannot be appended to m. From Lemma 4.1, there is enough space to insert a new record for i. □

Lemma 4.4 $\mathcal{L}_{\mathcal{CCT}} \triangleleft \mathcal{C}_1$.

PROOF : First, we prove that a message with non-legitimate records may not remain forever. Assume that a message m has no identifiers for non-existing processors. But, $m(n).\texttt{Nb} \neq \texttt{NbProcessors}(m) - 1$. This message will eventually be received by processor n. Processor n then will set $m(i).\texttt{Nb}$ to 0. Thus, the record of processor n in the message m becomes legitimate. All other records in m will eventually become legitimate in a similar manner.

Then we prove that a non-legitimate message may not remain forever. Let m be a message with $n < N$ different legitimate records and with no record for processor a. This message m will eventually reach processor a. Then a adds its own legitimate record increasing n by 1. If n was $N-1$ before the record of a was added, then m becomes a legitimate message. Otherwise, m reaches another processor and the same process continues until the records of all processors in the network are added to m. □

Theorem 4.3 *The $\mathcal{CCT}$ algorithm is self-stabilizing and satisfies the Cut-through constraint.*

PROOF : Follows from Theorems 4.1 and 4.2, and Lemmas 4.3 and 4.4. □

4.2 Complexity

Space Complexity. The processor space complexity improves by a factor of N in the cut-through routing over the store-and-forward model. This result is particularly interesting for large networks, where a classical (*i.e.* store and forward) approach would lead to tremendous consumption of memory. More importantly, the cut-through approach leads to a solution that requires less memory than the task itself.

Lemma 4.5 (Message Space Complexity) *A legitimate message is $MS = O(N\log_2 N)$ bits long.*

PROOF : Every legitimate messages has N records. Each record is composed of `Id`, which varies from 0 to $N-1$ ($\log_2 N$ bits), and `Nb`, which varies from 0 to $N-1$ ($\log_2 N$ bits). Thus,

$$\begin{aligned} MS &= N(\log_2 N + \log_2 N) \\ &\approx 2N\log_2 N \\ &= O(N\log_2 N) \end{aligned}$$

□

Lemma 4.6 (Processor Space Complexity) *Every processor needs $NS_{CT} = O(\log_2 N)$ bits.*

PROOF : Every processor needs to store the record being processed, which contains two fields: `Id` ($= \log_2 N$ bits) and `Nb` ($= \log_2 N$ bits). Each processor also holds the following variables: (i) Two $\log_2 N$-bit variables to hold the cut-through related variables (these are *local* variables). (ii) One 1-bit variable to hold the local variable (this is also a *local* variable). Thus, we have:

$$\begin{aligned} NS_{CT} &\approx 2\log_2 N \\ &= O(\log_2 N) \end{aligned}$$

□

Time Complexity. As we are dealing with the unidirectional ring networks, we can compute the time needed to complete one round in the ring. The *Round Time*, T_r is the time needed for the head of the message to complete one round.

$$T_r = \sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i\leftarrow i+1 \bmod N}}\right)$$

In the following, we assume that the propagation time is the same for all communication links and is equal to T_1.

Lemma 4.7 ($\mathcal{CCT}$ Round Time) *The round time in the $\mathcal{CCT}$ algorithm is $O(N\log_2 N)$.*

PROOF :

$$\begin{aligned} T_{r_{CT}} &= \textstyle\sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i\leftarrow i+1 \bmod N}}\right) \\ &\approx \textstyle\sum_{i=0}^{N-1} \left(\frac{\log_2 N}{C} + T_1\right) \\ &\approx N\left(\frac{\log_2 N}{C} + T_1\right) \\ &\approx \frac{N\log_2 N}{C} + N \\ &= O(N\log_2 N) \end{aligned}$$

□

Lemma 4.8 (Stabilization Time) *The stabilization time for the $\mathcal{CCT}$ algorithm is $O(N^2 \log_2 N)$.*

PROOF : If there is no message initially in the network, we can assume that a timer will generate a message after time $T_{r_{CT}}$. Assume that initially, there is an incorrect message in the network. At least one processor will initialize a correct message within $T_{r_{CT}}$ time. But, in our algorithm using the markers, it still takes another $T_{r_{CT}}$ time to forward the marker around the ring. After this message is destroyed and a new message is initiated, it will take one more round, $T_{r_{CT}}$ time, before the message becomes legitimate. In the worst case, the time is:

$$T_{Stabilizing_{CT}} = 3T_{r_{CT}} = O(N \log_2 N)$$

□

5 Conclusions

We presented a general approach to designing a self-stabilizing token ring algorithm in the cut-through model by designing a special transformer. This transformer takes as input a cut-through self-stabilizing token passing protocol and produces as output a token ring algorithm in the same model. Our transformation uses only one extra bit per frame and does not add any extra delay at any processor.

The suitability of the input token passing algorithm in the cut-through setting is extremely important to the performance of the token ring algorithm. With this goal in mind, we exploited the key advantages of the cut-through setting in designing an alternating counter flushing [15] technique. After stabilization, the proposed algorithm lowers the overhead due to self-stabilization by a factor of 32 compared to that of [15].

We presented a self-stabilizing algorithm to achieve a distributed census over a unidirectional ring. As in the token passing algorithm, we again took advantage of the cut-through model in designing an efficient solution to a memory-consuming task (census, where processors typically require $O(N \times \log_2 N)$ bits of memory). Our solution uses only a fraction of this memory ($O(\log_2 N)$ bits at each processor). The census problem solution can be used to design a number of important tasks, *e.g.*, size (number of processors), leader election (highest identifier).

Although there exist several self-stabilizing algorithms that provide a virtual ring in a general directed network ([13]), the proposed algorithms may not be easily adaptable to different routing problems. When processors that do not have as many output channels as input channels receive more than one messages simultaneously, deadlock may occur. In Eulerian directed graphs, any processor has as many input channels as output channels, so the deadlock situation cannot occur. Thus, by composing any of our algorithms with the self-stabilizing virtual circuit construction of [14] (that preserves the cut-through routing property and runs in Eulerian networks), we obtain solutions to the same tasks in Eulerian networks.

Acknowledgements We are grateful to the anonymous reviewers whose valuable comments helped improve the presentation. The first and third authors were supported in part

by the FRAGILE project of the ACI "Sécurité et Informatique".

References

[1] A. Costello and G. Varghese, The FDDI MAC meets Self-stabilization. *Proceedings of Fourth Workshop on Self-stabilizing Systems*. pp. 1-9, Austin Texas, 1999.

[2] S. Delaët and S. Tixeuil. Un algorithme auto-stabilisant en dépit de communications non fiables. *Technique et Science Informatiques*, 17(5):613–634, 1998.

[3] S. Delaët and S. Tixeuil. Tolerating Transient and Intermittent Failures. *Journal of Parallel and Distributed Computing*, Vol.62, No.5, May 2002.

[4] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[5] S. Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42:122–127, 1997.

[6] S. Dolev. Self-stabilization. *The MIT Press*, 2000.

[7] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, 1995.

[8] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[9] M. Jayaram and G. Varghese. Crash failures and drive protocols to arbitrary states. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 247–256, 1996.

[10] T. Masuzawa. A fault-tolerant and self-stabilizing protocol for the topology problem. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 1.1–1.15, 1995.

[11] R. Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Longman, 2000.

[12] J. Spinelli and R. Gallager. Event driven topology broadcast without sequence numbers. *IEEE Transactions on Communications*, 37:468–474, 1989.

[13] M. Tchuente. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Informatique Theoretique*, 15:47–66, 1981.

[14] S. Tixeuil. On a space-optimal distributed traversal algorithm. In *Proceedings of the Fifth Workshop on Self-stabilizing Systems, Liboa, Portugal*, pp. 216–228, 2001.

[15] G. Varghese, Self-stabilisation by counter flushing. *In Proc. 14th ACM PODC Symp.*, November 1994.