

Self-Stabilizing Wormhole Routing on Ring Networks

AJOY K. DATTA, MARIA GRADINARIU*, ANTHONY B. KENITZKI
AND SÉBASTIEN TIXEUIL⁺
*School of Computer Science,
University of Nevada Las Vegas
Las Vegas, NV89154, U.S.A.*
^{*}*IRISA, Campus de Beaulieu, France*
⁺*Université Paris Sud
LRI-CNRS UMR 8623, France*

Wormhole routing is most commonly used in parallel architectures in which messages are sent in small fragments, called flits. It is a lightweight and efficient method of routing messages between parallel processors. Self-stabilization is a technique that guarantees tolerance to transient faults (e.g., memory corruption or communication hazards) for a given protocol. Self-stabilization guarantees that the network recovers to a correct behavior in a finite amount of time, without the need for human intervention. Self-stabilization also guarantees the safety property, meaning that once the network is in a legitimate state, it will remain there until another fault occurs.

This paper presents the first self-stabilizing network algorithm in the wormhole routing model, using the unidirectional ring topology. Our solution benefits from wormhole routing by providing high throughput and low latency, and from self-stabilization by ensuring automatic resilience to all possible transient failures.

Keywords: distributed algorithms, fault-tolerance, self-stabilization, wormhole routing, interconnexion networks, virtual channels

1. INTRODUCTION

Self-stabilization. In 1974, Dijkstra pioneered the concept of self-stabilization in a distributed network [5]. 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 if 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 corruptions, program counter corruptions (which temporarily cause a processor to execute its code in any order), and communication channel corruptions.

Routing Protocols. There are many routing protocols for interconnected processor networks. Some of the most popular schemes include store and forward, virtual cut-through, and wormhole routing. In the store and forward protocol, messages are broken into *packets*, and each packet is forwarded in full to each processor along a path. A processor cannot forward a message packet until the entire message packet is received. In 1979, Kermani and Kleinrock proposed an improvement on the store and forward

Received May 15, 2002; accepted July 25, 2002.

Communicated by Biing-Feng Wang, Stephan Olariu and Gen-Huey Chen.

* A preliminary version of the paper was presented at the 2002 International Conference on Parallel and Distributed Systems, Chungli, Taiwan.

routing scheme called virtual cut-through [11]. Virtual cut-through is a protocol similar to store and forward, except that a packet is only stored at a processor if the required outgoing channel is not available. Wormhole routing uses a cut through routing technique with a few differences.

In wormhole routing, message packets are broken into flow control digits (or *flits*), where each flit is only a few bytes in size. All routing and message control information is stored in the first flit (also called the *header flit*). As the header flit moves through the network toward its destination, every processor it passes through will reserve a channel for the content (*data*) flits of the message to pass through. The other flits of the message will, thus, follow the header flit in a pipe-line fashion. When the last (*tail*) flit of the message passes through a processor, the channel reservation for that message is released. If a header flit reaches a processor where there is no available output channel, the other flits in the message packet remain where they are until their header flit advances. Thus, the flits of the packet wind from the current processor containing the header flit, all the way back to the source processor (much like a worm).

A routing protocol needs to be simple and robust [7], and have low latency and high throughput. *Latency* refers to the time that it takes for a packet to travel from the source to its destination. Wormhole routing has extremely low transmission latency since a flit of a message packet does not have to wait for the entire packet to arrive at a processor before it can be transmitted again. The protocol is *simple* in that the packet buffers required at each processor need only be a few flits in size (a few bytes). *High throughput* is achieved through *adaptive routing*, in which a message may take many paths from the source to the destination. A message may make many adaptive turns in order to avoid *congestion*, meaning that if a header flit reaches a processor where an outgoing channel is blocked, it is allowed to move in another direction.

Related Work. Considerable research has been done on making wormhole routing fault-tolerant. Papers such as [4] have added *virtual channels* to the network to handle faults. Virtual channels divide a single physical channel into many channels, sharing the bandwidth between them. Papers such as [8] have used an adaptive turn-based model to avoid faults. If a faulty processor is encountered on the network, a message will choose a path around the failed processor. All of these wormhole routing works are designed to tolerate *fail-stop* faults [12], meaning that one or more processors will cease to function entirely on the network, while the remainder will faithfully execute their programs. Papers such as [1, 2] have presented self-stabilizing network algorithms in a virtual cut through setting, but not in a wormhole routing environment.

Our Contribution. This paper presents the first self-stabilizing wormhole routing algorithm for the ring topology. We identify the faults that may occur due to transient failures in the wormhole routing setting. Although we only consider ring networks in this work, all of these can also occur in other topologies, such as meshes, hypercubes, *etc.* For example, a local processor fault can cause message flits to be lost or introduced at random, leaving fragmented and corrupted messages in the network. Data flits can flood all of the processor buffer flits on the network. Misrouted header flits can cause the network to deadlock. Our solution handles these problems in a simple and consistent manner, by locally checking for memory corruption and locally resetting the processor state.

Outline. In section 2, we provide the underlying model, system settings, and specification of the problem to be solved. In section 3, our self-stabilizing wormhole routing algorithm is presented, along with informal ideas on how self-stabilization is achieved. Concluding remarks can be found in section 5, while proofs of correctness can be found in section 4.

2. PRELIMINARIES

Our network model is a clockwise unidirectional ring $G = \{V, E\}$, where V is a set $\{1, 2, 3, \dots, n\}$ of processors, and E is the set $\{(1, 2), (2, 3), (3, 4), \dots, (n, 1)\}$ of channels. An individual processor p can only receive messages on its incoming (*right*) channel (predecessor(p), p), and can only transmit messages on its outgoing (*left*) channel (p , successor(p)).

An action is of the form $\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$. A *guard* is a boolean expression over processor variables and an input (such as a message). A *statement* is a sequence of program statements. An action can only be executed at a processor p if the corresponding guard is true. When an action is executed, all the statements in this action are executed atomically. We assume a *weakly fair* asynchronous environment for all processors. By weakly fair, we mean that if a processor has a guarded command that is continuously enabled, then this guard is eventually executed.

Every message circulating in the network consists of a sequence of flits. Messages have three parts – the first flit, called the head, followed by a sequence of data flits, and finally, a flit at the end called the tail. The communication channels are FIFO. In our self-stabilizing wormhole routing algorithm, we assume that all messages originate from a single sender. This assumption is made to prevent deadlock and starvation after the system is stabilized. A self-stabilizing token passing algorithm on rings [6] can be used to maintain a single sender at any time.

Section 5 includes ideas to extend our single sender algorithm to a multiple sender scheme.

Problem Specification Our wormhole routing self-stabilizing algorithm is correct if and only if the following three properties hold:

1. Liveness: Once the network is in a legitimate state, the network may not deadlock, livelock, or starve.
2. Reliable Delivery: Once the network is in a legitimate state, messages sent must be properly received.
3. Convergence: Regardless of the initial state, the network must return to a legitimate state in finite time.

3. WORMHOLE ROUTING

Network faults can corrupt the local variables of any network processor. Thus, message flits and their wormhole routing paths can be spontaneously introduced, lost, or corrupted. There are two kinds of corrupted messages to deal with:

1. Messages that are *structurally* not correct. A transient fault can cause message fragments to be corrupted beyond usefulness, or lost altogether. These messages may not contain a header flit or a tail flit, and are of one of the following types: (a) *Header-less Message Fragments*: This happens when several message flits are in a network without a header. (b) *Header Message Fragments*: A header without a tail moves alone in the network. (c) *Header-less Flooding*: A single message without a header or a tail occupies all the network flits except one and moves throughout the network. (d) *Misrouted Messages*: A message header flit is forwarded onward rather than delivered by the destination processor. It is then possible to deadlock the network.
2. Messages that are *logically* not correct. These messages contain both a header and a tail, but the contents of the message are corrupted from an application point of view or from a routing point of view.

Given the previous hazards to be taken care of, our algorithm implements the following solutions to these problems:

- **Header-less Message Fragments.** If the header of a message is lost before it reaches its destination, the message is discarded. When a header flit of a message is received in the incoming channel of a processor, the channel is *locked* for that message until the tail of that message is encountered. Whenever a processor receives a non-header message fragment on an incoming channel that is not reserved for that message, the fragment is discarded.
- **Header Message Fragments.** Corruption can cause the network to be flooded with message headers without tails. To correct this, we use a maximum hop counter in the message header. When a processor receives a header, it will know how long the header has been active on the network. A global maximum time can be specified by the application, *e.g.*, an upper bound on the number of nodes in the network, and this bound can be used as the maximum number of hops.
- **Header-less Flooding.** As the network can start in any arbitrary state, it is possible for every processor to be filled by a non-header value. All processors believe that they are forwarding a valid message. The solution to this problem is to have every processor count how many flits have been forwarded in a message. The application layer will specify a maximum message length. Since the header-less message has no end, at least one processor eventually decides to begin discarding the message fragments.
- **Misrouted Messages.** Program counter corruption can cause a processor to simply forward a message rather than deliver it. This can be dealt with in the same manner as header message fragments. As long as the maximum number of hops for a message is set to $|V| - 1$, a message can never be routed again by its originator.
- **Messages that are logically not correct.** It is possible for a header flit to contain a destination that does not exist in the network. Since each header flit has a timeout stamp in the header, the message is eventually dropped. The message will then be a header-less message, which was handled as described above.

In some instances, corrupted messages may not be detected by our protocol and, hence, maybe delivered to the application layer of the destination protocol. It is the responsibility of the application layer to recognize and discard the message in that case.

3.1 Messages and Data Structures

Messages. A message is a sequence of flits of a few bytes long. We refer to a member (variable) of a flit as $\langle \text{flit} \rangle . \langle \text{variable} \rangle$. We will use the following data structures for the three types of flits:

1. Header Flits (hflit), denoted as $\text{hflit}(\text{mid}, \text{ttl}, \text{dest})$, consist of a global unique message identifier (mid), a time to live (ttl), and a destination (dest).
2. Data Flits (dfliit), denoted as $\text{dfliit}(\text{mid}, \text{dat})$, consist of a message id and a fragment of the actual message payload to be sent.
3. Tail Flits (tflit), denoted as $\text{tflit}(\text{mid})$, consist only of a message identifier.

Constants. Three constants are used in the protocol. The maximum time to live in hops (maxttl) and the maximum message length (maxlen) are constant inputs supplied by the application layer. The third constant is the maximum message identifier (maxmid) – the largest allowed by the processor software or register size.

Variables. The left channel lock (lchannel) variable holds the current message identifier to be transmitted, or 0 if the local processor is not routing a message. If a processor p is not routing a message, then p knows that it may deliver received data and tail flits. The total flits received (ftotal) variable is used to record the total flits received for a message. This variable is used to prevent a data flit flood, where one or more data flits can remain in the network forever, moving in a circle. The **Buffer** variable represents the flit buffer of a processor. The **Buffer** variable can only hold a flit value or no value at all ($\langle \text{empty} \rangle$).

Flow Control. Wormhole Routing flow control is guaranteed by a Clear To Send (CTS) wire that connects each processor in a uni-directional link. The CTS wire on a processor p for the link $\langle \text{predecessor}(p), p \rangle$ is set to LOW when p is ready for a new message; it is set to HIGH otherwise. This wire can also be modeled as a read-modify-write shared register between the two processors in the unidirectional link. A processor can read the CTS variable of its successor, but it can only write to its own. Thus the CTS variable will allow only one flit to be in the flit buffer of a processor at any time, and that the processor will not accept another flit into its local buffer until it is empty. Each processor will have a single CTS variable for each incoming link. This variable will simply be called CTS for the ring protocol since every processor only has a single incoming link.

3.2 Helper Functions

The following are the functions called in the main program.

- **SENDNEWMESSAGE** is a function that will activate when the privileged processor p is idle for too long (that is, when p has nothing to forward and has nothing in its flit buffer). The processor will generate a new unique message id and an arbitrary destination, and then it will send its left neighbor a new correct message starting with a header, some data flits, and a tail flit.

- **DELIVERMSG** is a function that will deliver a message to the application layer, clear out the channel flit buffer, and set the CTS variable of the incoming channel to LOW.
- **DISCARD** is a function that will clear out the channel flit buffer and set the CTS variable of the incoming channel to LOW.
- **RCV** is a function used to read transmitted data from the incoming channel.
- **SEND** is a function that transmits data across the outgoing channel.
- **TIMEOUT** is a function that will wait a sufficiently long time for a network condition to hold [10]. The normal timeout actions we use can be implemented with a local clock at each processor using the approach given in [10]. We make use of the following two predicates in our TIMEOUT actions:

$$\begin{aligned} \text{Full} &\equiv (\text{flit\#ch.1.2} + \dots + \text{flit\#ch.n-1.n} + \text{flit\#ch.n.1} = n), \\ \text{Empty} &\equiv (\text{flit\#ch.1.2} + \dots + \text{flit\#ch.n-1.n} + \text{flit\#ch.n.1} = 0), \end{aligned}$$

where flit\#ch.p.q is the number of messages in transit from Processor p to Processor q on the channel $(p, q) \in E$, and n is the size of the network.

3.3 Algorithm

Algorithm 1 Self-stabilizing Wormhole routing on rings (Main program)

```

inputs  maxttl, maxmid, maxlen
var     lchannel: {0..maxmid},
        ftotal: {0..maxlen+1},
        CTS: {LOW, HIGH}
        Buffer: {⟨empty⟩, hflit, dflit, tflit}
begin
    RECEIVE actions (presented as Algorithm 2)
    [] SEND actions (presented as Algorithm 3)
end

```

The wormhole routing algorithm is presented as Algorithms 1, 2, and 3. The *Receive actions* (Algorithm 2) and *Send actions* (Algorithm 3) are described in detail below:

Receive actions. Action **(R1)** allows a processor to receive header flits. Header flits are first checked to see if they have arrived at the correct destination. When a header flit is delivered, the *lchannel* lock variable is set to 0, the *not routing* status. Header flits that are not delivered are first checked for faults (time to live). Faulty header flits are discarded, and all others are written to the local *Buffer* variable to be routed. Once a flit is written to the *Buffer* variable, the clear to send (CTS) variable is set to HIGH (not ready to receive). Action **(R2)** allows a processor to receive data flits. When a data flit is received, the *lchannel* variable is examined against the message identifier of the data flit. If the *lchannel* variable is set to 0, then the flit is delivered. If the *lchannel* variable is not equal to the message id of the data flit, then the flit is discarded. The flit is only routable if the message id of the data flit is equal to the *lchannel* variable and the total

Algorithm 2 Self-stabilizing Wormhole routing on rings (Receive actions) – Processor i .

```

(R1) RECV hflit (mid, ttl, dest)  $\wedge$  CTS = LOW  $\rightarrow$ 
  /* Receive a header flit. */
  if hflit.ttl  $\leq$  maxttl  $\wedge$  hflit.dest =  $i$   $\rightarrow$ 
    lchannel := 0; DELIVERMSG hflit (mid, ttl, dest);
  [] (D1) hflit.ttl > maxttl  $\rightarrow$ 
    lchannel := 0; DISCARD hflit (mid, ttl, dest);
  [] hflit.ttl  $\leq$  maxttl  $\wedge$  hflit.dest  $\neq$   $i$   $\rightarrow$ 
    Buffer := hflit (mid, ttl, dest); CTS := HIGH;
  fi
(R2) [] RECV dflit (mid, dat)  $\wedge$  CTS = LOW  $\rightarrow$ 
  /* Receive a data flit. */
  if lchannel = 0  $\rightarrow$ 
    DELIVERMSG dflit (mid, dat);
  [] (D2) lchannel = dflit.mid  $\wedge$  ftotal > maxlen  $\rightarrow$ 
    DISCARD dflit (mid, dat);
  [] lchannel = dflit.mid  $\wedge$  ftotal  $\leq$  maxlen  $\rightarrow$ 
    Buffer := dflit (mid, dat); CTS := HIGH;
  [] (D3) lchannel > 0  $\wedge$  lchannel  $\neq$  dflit.mid  $\rightarrow$ 
    DISCARD dflit (mid, dat);
  fi
(R3) [] RECV tflit (mid)  $\wedge$  CTS = LOW  $\rightarrow$ 
  /* Receive a tail flit. */
  if lchannel = 0  $\rightarrow$ 
    DELIVERMSG tflit (mid);
  [] lchannel = tflit.mid  $\rightarrow$ 
    Buffer := tflit (mid); CTS := HIGH;
  [] (D4) lchannel > 0  $\wedge$  lchannel  $\neq$  tflit.mid  $\rightarrow$ 
    DISCARD tflit (mid);
  fi
(R4) [] TIMEOUT CTS = HIGH  $\wedge$  Full  $\rightarrow$ 
  Buffer :=  $\langle$ empty $\rangle$ ; CTS := LOW;
fi

```

flits received ftotal variable does not exceed the maxlen constant. Routable data flits are written to the Buffer variable and the CTS variable is set to HIGH . Action **(R3)** allows a processor to receive tail flits. Tail flits do not require a check against the ftotal variable, but they are otherwise handled in the same way as data flits in **(R2)**. Action **(R4)** allows the network to recover from a deadlock. This action does not activate until a sufficient amount of time has passed such that no message flit may be on any channel in the network. Since Processor p is unable to receive a new flit for an extremely long time, and is not clear to receive new flits (Fig. 1), p sets Buffer to nothing, and the CTS variable to LOW (ready to receive a new flit).

Algorithm 3 Self-stabilizing Wormhole routing on rings (Send actions) – Processor i .

(S1) if Left.CTS = LOW \wedge Buffer = hflit (mid, ttl, dest) \rightarrow
 /* Can send a header flit. */
 if Buffer.ttl \geq maxttl \rightarrow
 ftotal := 0; lchannel := 0; SEND tflit (mid); CTS := LOW;
 [] lchannel := Buffer.mid; Buffer.ttl := Buffer.ttl + 1;
 ftotal := 1; SEND hflit (mid, ttl, dest); CTS := LOW;
 fi

(S2) [] Left.CTS = LOW \wedge Buffer = dflit (mid, dat) \rightarrow
 /* Can send a data flit. */
 if ftotal \geq maxlen \rightarrow
 ftotal := 0; lchannel := 0; SEND tflit (mid); CTS := LOW;
 [] ftotal := ftotal + 1; SEND dflit (mid, dat); CTS := LOW;
 fi

(S3) [] Left.CTS = LOW \wedge Buffer = tflit (mid) \rightarrow
 /* Can send a tail flit. */
 ftotal := 0; lchannel := 0; SEND tflit (mid); CTS := LOW;

(S4) [] CTS = HIGH \wedge Buffer = \langle empty \rangle \rightarrow
 /* Local invalid condition */
 CTS := LOW;

(S5) [] TIMEOUT Buffer := \langle empty \rangle \wedge lchannel = 0
 \wedge Left.CTS = LOW \wedge Empty \rightarrow
 lchannel := 0; SENDNEWMESSAGE;

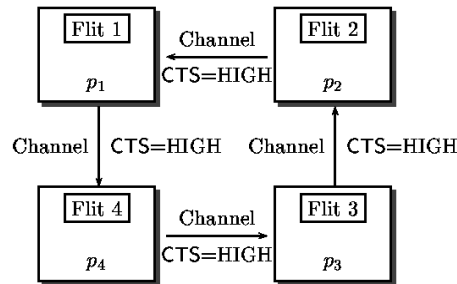


Fig. 1. Wormhole routing deadlock.

Send actions. Action (S1) allows a processor to route a header flit. A processor will lock its outgoing channel, initialize its **ftotal** variable to 1, transmit the flit, and set its CTS to LOW. Action (S2) allows a processor to route a data flit. A processor will transmit the flit, increment the **ftotal** variable, and set its CTS to LOW. Action (S3) allows a processor to route a tail flit. A processor will transmit the flit, set the **lchannel** variable to 0, and set its CTS to LOW. Action (S4) prevents a local fault condition in which the CTS variable is set to HIGH, and the Buffer variable is empty. A processor will merely reset its CTS variable back to LOW. Action (S5) is a TIMEOUT action that

prevents a network deadlock condition. Just like Action **(R4)**, **(S5)** is not activated until enough time has passed such that every channel in the network is empty. The network can deadlock if all the buffer flits on the network are full, and if no processor has a CTS value of LOW.

4. PROOF OF CORRECTNESS

The network is considered to be in a *legitimate state* if all the messages in the channels satisfy some *message predicates* (defined below), and if the processors satisfy some *processor predicates* (defined below). Formally, the legitimacy predicate, L_{WR} , is defined as follows:

$$L_{WR} \equiv P_1 \wedge P_2 \wedge P_3 \wedge M_1 \wedge M_2 \wedge M_3 \wedge M_4,$$

where P 's and M 's represent the processor predicates and message predicates, respectively.

Processor Predicates. No processor should have a CTS value of HIGH if it has an empty Buffer (Predicate P_1). Each processor p should have an lchannel value equal to zero (not forwarding) or equal to the message identifier of the last header flit received by p (Predicate P_2). At least one processor should have a Buffer variable equal to $\langle \text{empty} \rangle$ and a CTS variable equal to LOW (Predicate P_3).

Message Predicates. A message should be well-structured (Predicate M_1)¹ The number of data flits in the message should be less than maxlen (Predicate M_2). The time to live variable in a header flit should never exceed maxttl (Predicate M_3). The message identifiers of the header, data, and tail flits remain the same and equal throughout the life of a message (predicate M_4).

In the following three sections, we prove the correctness of the algorithm by proving the liveness, reliable delivery and convergence properties.

Lemma 4.1 (Deadlock) Starting from a legitimate configuration, the network does not deadlock.

Proof: Deadlocks occur when processors are waiting on resources that are never freed. It was proven in [3] that a routing algorithm in a direct network is deadlock-free if there is no cycle in the channel dependence graph. Wormhole routing in a unidirectional ring contains no cycles in the channel dependence graph since the sender cannot route its own message twice. \square

Lemma 4.2 (Starvation) Starting from a legitimate configuration, the network does not starve.

¹ A message is constructed with a header flit, one or more data flits, and a tail flit. Many messages may not have all of their flits on the network at one time. A header flit and multiple data flits may have been legitimately delivered to the destination while a tail flit remains on the network. A header flit may be on the network, while data flits and the tail flit may be waiting to be transmitted.

Proof: Starvation occurs whenever a processor needs to send a message but is too busy routing messages for other processors. This cannot happen since we assume that there is only a single sender in the network. Therefore, the action (S4) is eventually activated on the sender processor. \square

Lemma 4.3 (Livelock) Starting from a legitimate configuration, the network does not livelock.

Proof: The initial configuration being legitimate, Predicates M_1 through M_4 hold. Then, Actions (S1), (S2), and (S3) preserve those predicates, while Actions (S4) and (S5) remain disabled. \square

4.1 Reliable Delivery

Lemma 4.4 Starting from a legitimate configuration, every flit sent to a processor is eventually received.

Proof: After a processor p executes a SEND function on a left channel, the successor of p (i.e., $\text{successor}(p)$) will have a RECV action $r \in \{(\mathbf{R1}), (\mathbf{R2}), (\mathbf{R3})\}$ enabled. We assume a weakly fair scheduler, so the RECV action r on $\text{successor}(p)$ will be activated after a finite amount of time. \square

Lemma 4.5 Starting from a legitimate configuration, every flit received at a processor is eventually delivered or written to the local Buffer variable.

Proof: The RECV actions that can process a newly received flit are (R1), (R2), and (R3). There are only three possible outcomes from the statements within those actions: DISCARD, DELIVER, or Buffer write. Since we start from a legitimate configuration, the conditions (D1), (D2), (D3), and (D4) are disabled, so no DISCARD may occur. \square

Lemma 4.6 Starting from a legitimate configuration, every processor having an incoming channel Buffer variable containing a flit eventually sends (forwards) this flit.

Proof: Assume that a Processor p has a flit in its Buffer variable, and that the flit is never sent. By Actions (S1), (S2), and (S3), $\text{successor}(p)$ must have a CTS value of HIGH. Since the network is in a legitimate state, P_3 holds. Therefore, at least one processor has a CTS value equal to LOW.

Predicate P_1 guarantees that every processor with a CTS value of HIGH has a flit in its Buffer variable. Let q be the first processor *upstream* to p , such that $q.\text{CTS} = \text{LOW}$. The predecessor of q , ($\text{predecessor}(q)$), must have a flit in its Buffer; thus, one of $\text{predecessor}(q)$'s SEND actions must be true. Eventually (S1), (S2), or (S3) is activated on $\text{predecessor}(q)$. When one of these actions is executed, the flit in $\text{predecessor}(q)$.Buffer is sent to q . Then, $\text{predecessor}(q)$.Buffer is set to $\langle \text{empty} \rangle$, and $\text{predecessor}(q)$.CTS is set to LOW. By induction, this process repeats until $\text{successor}(p)$.CTS = LOW and one of the SEND actions in p is eventually activated. Therefore, any flit that was written in p .Buffer is eventually sent to $\text{successor}(p)$. \square

Theorem 4.1 (Reliable Delivery) After the network enters in a legitimate state, messages sent are properly received.

Proof: The theorem follows from Lemmas 4.4, 4.5, and 4.6. \square

4.2 Convergence

We will prove that this algorithm will converge to a legitimate state from any arbitrary initialization in a finite amount of time. This is done following the *convergence stair* method [9]. In this method, the system converges to fulfill a number of predicates A_1, A_2, \dots, A_k , such that for $1 \leq i < k$, A_{i+1} is a *refinement* of A_i [6]. A predicate A_{i+1} *refines* A_i iff A_i holds when A_{i+1} holds; A_i is called an *attractor*. Using the convergence stair method, we can show that L_{WR} is an attractor for *true*. The conjunction of all message predicates is an attractor for the processor predicates. Thus, we can prove that the conjunction of all predicates will eventually hold in the system, and that the system will converge to a legitimate state.

Processor Predicates. First, we will prove that starting from an arbitrary configuration, all the processor legitimacy state predicates will be satisfied in a finite amount of time.

Lemma 4.7 Predicate $P_1 \wedge P_2 \wedge P_3$ is an attractor for *true*.

Proof: We show that each predicate eventually holds:

(P_1) is guaranteed by **(S4)**, which will eventually be executed on some processor p .

(P_2) follows from **(R1)**. The `lchannel` variable is set to the message identifier of the received header flit if it is forwarded, or to 0 if the header flit is delivered.

(P_3) follows from **(R4)**. Deadlocks are resolved by *packet preemption* as discussed in [13]. After some time, the `timeout` action **(R4)** is activated on a nonempty set W of processors, the `Buffer` variables of one or more deadlocked processors are discarded from the network, and the `CTS` variables of those processors are set to `LOW`.

By the code of the algorithm, Predicates (P_1), (P_2), and (P_3) are closed. \square

Message Predicates. Next, we will prove that starting from an arbitrary configuration, all the message legitimacy state predicates will be satisfied in a finite amount of time.

Lemma 4.8 Predicate $M_1 \wedge M_2 \wedge M_3 \wedge M_4$ is an attractor for $P_1 \wedge P_2 \wedge P_3$.

Proof: We need to prove that all faulty messages will be removed from the network. Each flit type is handled individually.

Faulty Header Flits Action **(R1)** guarantees that these flits will eventually either be delivered, or that **(D1)** will remove them when their maximum number of hops has expired.

Faulty Data Flits Assume that a data flit d with message id i can remain in the network forever. We can also assume that every processor on the network thinks that it is forwarding message i ; otherwise, a processor with a different `lchannel` variable will execute **(D3)** and drop the flit. Every time the data flit d is received by a processor p , p will increment its local `ftotal` variable, thus correctly recording the total number of flits forwarded for message i so far. However, **(R2)** guarantees that this can happen at most `maxlen` times before the message is discarded by guard **(D2)**.

Faulty Tail Flits These can never pass through a processor twice. Before forwarding a tail to a neighbor, a processor will reset its `lchannel` and `ftotal` variables in **(S1)**, **(S2)**, or **(S3)**. Thus, eventually, a tail flit will reach a processor where it is delivered **(R3)** or discarded **(D4)**.

By the code of the algorithm, predicates (M_1) , (M_2) , (M_3) , and (M_4) are closed. \square

Theorem 4.2 (Convergence) Predicate L_{WR} is an attractor for *true*.

Proof: The theorem follows from Lemmas 4.7 and 4.8, and [9]. \square

Remark 4.1 Following the arguments presented in the proof of Lemma 4.8, the complexity of the wormhole routing protocol on rings is $O(n)$, where n is the ring size.

5. CONCLUSIONS

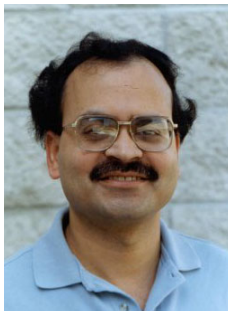
We have presented the first self-stabilizing algorithm in the context of wormhole routing. Our algorithm can be used to transmit messages between nodes so that they can benefit from the high throughput and low latency of wormhole routing. Our solution is for ring networks, where messages are initiated by a single sender.

We can extend our protocol for multiple senders. The complications which may arise due to the introduction of multiple senders include the following: It is possible to *starve* a processor. A processor that needs to send a message can be prevented from doing so by other processors in a unidirectional ring. It is possible for messages to *deadlock*. Since we have a ring topology, any two messages introduced into the network by different processors can acquire resources in a circular-dependent manner.

Both of these problems can be avoided by adding more available channels upon which any processor can initiate a message. A simple solution presented in [4] is to add multiple *virtual channels* to the network for each physical channel. Virtual channels are logical channels which may share the same physical wire, but each virtual channel contains its own flit buffer, control program (including local variables), and data path. The flit buffers can be represented as an array of n flit buffers, along with an array of n `lchannel` lock variables. A flit sent from `flit buffer(i)` over the physical channel will be written to `flit buffer(i)` at the destination processor. If one virtual channel is allowed per sender processor, then we can make the same self-stabilizing guarantees as that of a single processor and a single channel.

REFERENCES

1. J. Beauquier, A. K. Datta, and S. Tixeuil, "Self-stabilizing census with cut-through constraint," *Fourth Workshop on Self-Stabilizing Systems (WSS '99)*, IEEE CS Press, 1999, pp. 70-77.
2. A. M. Costello and G. Varghese, "The FDDI MAC meets self-stabilization," *Fourth Workshop on Self-Stabilizing Systems (WSS '99)*, IEEE CS Press, 1999, pp. 1-9.
3. W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, Vol. C-36, 1987, pp. 547-553.
4. W. J. Dally, "Virtual channel flow control," in *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 60-68.
5. E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communications of the Association of the Computing Machinery*, Vol. 17, 1974, pp. 643-644.
6. S. Dolev, *Self Stabilization*, MIT Press, 2000.
7. C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proceedings of 19th Annual International Symposium on Computer Architecture*, 1992, pp. 278-287.
8. C. J. Glass and L. M. Ni, "Fault tolerant wormhole routing in meshes," in *Proceedings of 23rd Annual International Symposium on Fault Tolerant Computing*, 1993, pp. 240-249.
9. M. G. Gouda and N. J. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers*, Vol. 40, 1991, pp. 448-458.
10. M. G. Gouda, *Elements of Network Protocol Design*, John Wiley and Sons Inc., 1998.
11. P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, Vol. 3, 1979, pp. 267-286.
12. N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
13. L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, Vol. 26, 1993, pp. 62-76.



Ajoy K. Datta is a professor of computer science at the University of Nevada Las Vegas. His primary area of research interest is distributed computing. He works on the fault-tolerance and self-stabilization properties of distributed systems.



Maria Gradinariu received her MS and Ph.D. in Computer Science from the Al. I. Cuza University, Iasi, Romania in 1997 and the University of Paris Sud, France in 2000, respectively. She is currently an assistant professor of Computer Science at the University of Rennes, France. Her research interests include deterministic and probabilistic distributed protocols, mobile systems, peer-to-peer networks, and ad-hoc networks.



Sébastien Tixeuil received the Magistère d'Informatique Appliquée from the University Pierre and Marie Curie (France) in 1995, and his M.Sc and Ph.D. in Computer Science from the University of Paris-Sud XI (France) in 1995 and 2000, respectively. In 2000, he joined the faculty at the University of Paris-Sud XI. His research interests include self-stabilizing and fault-tolerant distributed computing.

Anthony B. Kenitzki is currently employed in the private sector. He received his MS in Computer Science from the University of Nevada Las Vegas in 2002. His research interests are distributed databases and fault-tolerant network computing.