Self-Stabilizing Wormhole Routing on Ring Networks[*]

Ajoy K. Datta
Department of Computer Science, University of Nevada Las Vegas.

Maria Gradinariu
IRISA, Campus de Beaulieu, France.

Anthony B. Kenitzki
Department of Computer Science, University of Nevada Las Vegas.

Sébastien Tixeuil[†]
LRI-CNRS UMR 8623, Université Paris Sud, France.

Abstract

Wormhole routing is most common 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 hazard) for a given protocol. Self-stabilization guarantees that the network recovers to a correct behavior in finite 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.

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 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 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*), 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 the header flit advances. Thus, the flits of the packet wind from 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 a 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 an 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 large (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 in making wormhole routing fault tolerant. Papers such as [4] add *virtual channels* to the network to handle faults. Virtual channels divide a single physical channel into many, sharing the bandwidth between them. Papers such as [8] use 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 papers are written 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] present 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 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 the self-stabilization is achieved. Concluding remarks can be found in Section 5 while extensive formal 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 $\{1, 2, 3, ...\ n\}$ of processors, and E is the set $\{(1, 2), (2, 3), (3, 4), ...(n, 1)\}$ of channels. An individual Processor p can only receive messages on its incoming (*right*) channel $(\mathrm{predecessor}(p), p)$, and it can only transmit messages on its outgoing (*left*) channel $(p, \mathrm{successor}(p))$.

An action is of the form $< \mathrm{guard} > \longrightarrow < \mathrm{statement} >$. 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 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 — first flit called head, followed by a sequence of data flits, and finally, a flit at the end called tail. The communication channels are FIFO. In our self-stabilizing wormhole routing algorithm, we assume that all messages sent 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:

Liveness: Once the network is in a legitimate state, the network may not deadlock, livelock, or starve.

Reliable Delivery: Once the network is in a legitimate state, messages sent must be properly received.

Convergence: Regardless of 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 the 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 will be 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, then 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, 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 to have every processor filled by a non-header value. All processors believe that they are forwarding a valid message. The solution to this 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 above.

In some instances, corrupted messages may not be detected by our protocol, and hence delivered to the application layer of the destination protocol. It is the responsibility of the application layer to recognize and discard the message in this 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 of a flit as $<$ flit $>$. $<$ variable $>$. We will use the following data structures for the three types of flits: Header Flits (hflit), hflit(mid, ttl, dest), consist of a global unique message identifier (mid), a time to live (ttl), and a destination (dest). Data Flits (dflit), dflit(mid, dat), consist of a message id and a fragment of the actual message payload to be sent. Tail Flits (tflit), tflit(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 transmit, 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 account 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 ($<$ empty $>$).

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 $<$ predecessor$(p), p >$ 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, an arbitrary destination, and then it will send its left neighbor a new correct message starting with a header, numerous 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. [**RECV**] is a function used to read transmitted data from incoming channel. [**SEND**] is a function that transmits data across an 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:

$$\text{Full} \equiv (\text{flit}\#\text{ch}.1.2 + \ldots$$
$$+\text{flit}\#\text{ch}.n-1.n + \text{flit}\#\text{ch}.n.1 = n)$$

$$\text{Empty} \equiv (\text{flit}\#\text{ch}.1.2 + \ldots$$
$$+\text{flit}\#\text{ch}.n-1.n + \text{flit}\#\text{ch}.n.1 = 0)$$

where $\text{flit}\#\text{ch}.p.q$ 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:

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

---

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

Receive actions. The 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). The 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 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.

The action (**R3**) allows a processor to receive tail flits. Tail flits do not require a check against the ftotal variable, but they are handled the same in all other aspects as data flits in (**R2**). The action (**R4**) allows the network to recover from a deadlock. This action does not activate until a sufficient 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 (Figure 1), then 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 ∧ Buffer = hflit(mid, ttl, dest) ⟶
   /* *Can send a header flit.* */
     if Buffer.ttl ≥ maxttl ⟶
       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 ∧ Buffer = dflit(mid, dat) ⟶
   /* *Can send a data flit.* */
     if ftotal ≥ maxlen ⟶
       ftotal := 0; lchannel := 0; SEND tflit(mid); CTS := LOW;
     [] ftotal := ftotal + 1; SEND dflit(mid, dat); CTS := LOW;
     fi
(**S3**) [] Left.CTS = LOW ∧ Buffer = tflit(mid) ⟶
   /* *Can send a tail flit.* */
     ftotal := 0; lchannel := 0; SEND tflit(mid); CTS := LOW;
(**S4**) [] CTS = HIGH ∧ Buffer =< empty >⟶
   /* *Local invalid condition* */
     CTS := LOW;
(**S5**) [] TIMEOUT Buffer :=< empty > ∧lchannel = 0
   ∧Left.CTS = LOW ∧ Empty ⟶
     lchannel := 0; SENDNEWMESSAGE;
fi

Send actions. The 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. The 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. The 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. The 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. The action (**S5**) is a TIMEOUT action that prevents a network deadlock condition. Just like action (**R4**), the action is not activated until enough time has passed such that every message channel on the network should be empty. The network can deadlock if all buffer

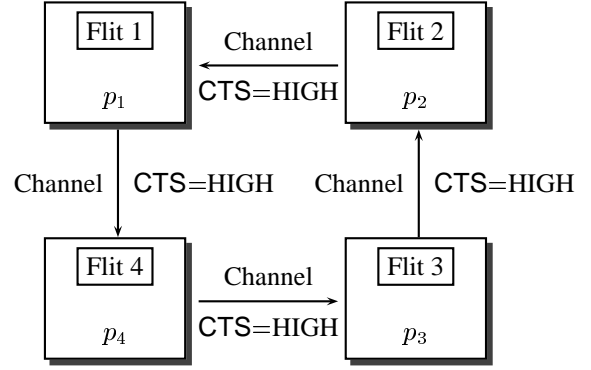flits on the network are full, and no processor has a CTS value of LOW.

Figure 1. Wormhole Routing Deadlock

4. Proof of Correctness

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

$$\mathcal{L}_{WR} \equiv P_1 \wedge P_2 \wedge P_3 \wedge M_1 \wedge M_2 \wedge M_3 \wedge M_4$$

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 < empty > 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 and reliable deliv-

---

[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 yet to be transmitted.

ery and convergence properties. The proofs are omitted due to lack of space.

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

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

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

Theorem 4.1 (Reliable Delivery) *Once the network is in a legitimate state, messages sent are properly received.*

We prove that this algorithm will converge to a legitimate state from any arbitrary initialization in finite 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 show that $\mathcal{L}_{WR}$ is an attractor for *true*. The conjunction of all message predicates is an attractor for the processor predicates. Thus, we prove that the conjunction of all predicates will eventually hold in the system, and the system converges to a legitimate state.

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

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

Theorem 4.2 (Convergence) *Predicate $\mathcal{L}_{WR}$ is an attractor for* true.

5. Conclusions

We 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: *(1.)* 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. *(2.)* It is possible for messages to *deadlock*. Since we have a ring topology, any two messages

introduced onto 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 for any processor to initiate a message upon. 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),* Austin, Texas, IEEE CS Press, pp. 70-77, June 4-5, 1999.

[2] A.M. Costello and G. Varghese. The FDDI MAC meets Self-stabilization. *Fourth Workshop on Self-Stabilizing Systems (WSS'99),* Austin, Texas, IEEE CS Press, pp. 1-9, June 4-5, 1999.

[3] W.J. Dally and C.L. Seitz. Deadlock-free Message Routing in Multiprocessor Interconnection Networks, *IEEE Transactions on Computers*, C-36:547-553, 1987.

[4] W.J. Dally. Virtual Channel Flow Control, *Proc. of the 17th International Symposium on Computer Architecture*, 60–68, May 1990

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

[6] S. Dolev. *Self Stabilization* MIT Press. 2000

[7] C.J. Glass and L.M. Ni. The Turn Model for Adaptive Routing *Proc. 19th Annual Int'l Symposium on Computer Architecture*, May, 1992.

[8] C.J. Glass and L.M. Ni. Fault Tolerant Wormhole Routing in Meshes *Proc. 23rd Annual Int'l Symposium on Fault Tolerant Computing*, June, 1993.

[9] M.G. Gouda and N.J. Multari. Stabilizing Communication Protocols *IEEE Transactions on Computers*, 40(4):448-458, 1991

[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*, 3:267-286, 1979.

[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*, 26:62–76, 1993.