

Modular Implementation of Temporally Ordered Routing Algorithm

by

NANDESH KUMAR PALANISAMY

B.E., Bharathidasan University, India, 1992

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Master of Engineering
Department of Computer Science
2001

This thesis entitled:
Modular Implementation of Temporally Ordered Routing Algorithm
written by Nandesh Kumar Palanisamy
has been approved for the Department of Computer Science

Associate Professor Dr. Dirk Grunwald

Assistant Professor Dr. Richard Han

Assistant Professor Dr. Timothy Brown

Date _____

The final copy of this thesis has been examined by the signators, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Palanisamy, Nandesh Kumar(M.E., Computer Science)

Modular Implementation of Temporally Ordered Routing Algorithm

Dissertation directed by Assoc. Professor Dirk Grunwald

Traditionally routing algorithms have been implemented in a monolithic manner. This approach doesn't lend itself easily to add new functionality to the implementation or to remove or replace a piece of the functionality which might not be relevant any more. Also in a monolithic implementation it is not very easy to reuse modules that provide common functionality across different protocols.

The present work implements the Temporally Ordered Routing Algorithm in a modular manner using the Click Modular Router. The temporally ordered routing algorithm was decomposed in to components that are easy to understand and simple to implement. The components are implemented as click modules.

Dedication

To my family.

Acknowledgments

Though the work described in this thesis would not have been possible without the help of many I would like to thank specifically a few in particular.

I thank Dr. Dirk Grunwald for proposing this topic and providing me with the information I need to work on this project.

I thank Michael Neufeld for his help in setting up the development environment and in testing and debugging the implementation.

Contents

1	Introduction	1
	Thesis Overview	1
2	Click Modular Router.....	2
	Click Architecture	2
	A sample IP Router configuration.....	3
3	Temporally Ordered Routing Algorithm	7
	Introduction to TORA	7
	The Protocol.....	8
	Height Metric.....	10
	Link Direction Assignment	11
	Creating Routes	11
	Maintaining routes	12
	Case 1(Generate)	13
	Case 2(Propagate).....	14
	Case 3(Reflect).....	14
	Case 4(Detect)	14
	Case 5(Generate)	15
	Erasing Routes	15
4	Modular Implementation of Temporally Ordered Routing Algorithm	17
	Overview	17
	Architecture	19
	ToraLookUpIpAddress	27
	ToraGetNextHopNeighbor	27
	ToraSendQRY	27
	ToraDispatchControlPkt	27
	ToraProcessQRY	28
	ToraProcessUPD	28
	ToraProcessCLR	29
	ToraOpt	29
	ToraSendPending.....	29
	ToraOutput.....	29
	Conclusion and Future work	29

References	31
Appendix A Click configuration file for Tora Router	32
Click configuration file for Tora Router	32
Appendix B Source code for ToraProcessQRY element	40
Source code for ToraProcessQRY element	40
Appendix C Source code for ToraProcessUPD element	48
Source code for ToraProcessUPD element	48
Appendix D Source code for ToraProcessCLR element	59
Source code for ToraProcessCLR element	59
Appendix E Source code for ToraOpt element	67
Source code for ToraOpt element	67

Tables

1. Height metric description.....	11
2. Link direction assignment.....	11
3. Tora control packets and its handling elements.....	27

Figures

1. A Sample IP Router Configuration.....	4
2. Configuration file for the sample IP Router	5
3. Directed acyclic graph of routers defined by the relative height of the routers.....	10
4. Maintaining Routes - Decision Tree	13
5. Class diagram for Tora Routing Table.....	18
6. Input elements - From Network Interface.....	20
7. Common Elements.....	21
8. Input element - From kernel	22
9. Get Next Hop	23
10. Dispatch Control Packet	25
11. Output - Network Interface.....	26

Chapter 1

Introduction

This thesis describes a modular implementation of Temporally Ordered Routing Algorithm (TORA) by extending the Click Modular Router [1]. The click modular router provides a framework for constructing and connecting routing elements to create routers of different configurations. The existing click implementation supports a wide range of routing elements supporting various router configurations. The existing click implementation is extended to implement the Temporally Ordered Routing Algorithm by implementing new click elements specifically developed for TORA and by reusing elements which are already available in the existing click implementation.

1.1 Thesis Overview

Second chapter of this thesis provides an overview of Click Modular router.

Third chapter of this thesis provides an overview of TORA.

Fourth chapter of this thesis describes the TORA implementation and the future work.

Appendix A provides a sample click configuration file for a Tora Router.

Appendix B provides the source code for ToraProcessQRY element.

Appendix C provides the source code for ToraProcessUPD element.

Appendix D provides the source code for ToraProcessCLR element.

Appendix E provides the source code for ToraOpt element.

Chapter 2

Click Modular Router

This chapter describes the design of the Click modular router. In order to understand the design of the TORA implementation as a click modular router, it is important to understand the design of the Click Modular Router and its usage in building various router configurations.

2.1 Click Architecture

Click is a flexible, modular software architecture for building routers. Click routers are built using the basic building blocks of the click routers called *elements*. To build a router configuration, the user connects a collection of elements into a graph. Packets move from one element to the other along the edges of the graph. To extend a router configuration, the user need to simply implement a new element or find an appropriate element if it is already available and connect it to the appropriate elements in the graph. This graphs resembles a flow chart, except that connections represent packet flow, not control flow, and elements are actual objects that may maintain private state. Inside a running router, each element is a C++ object and connections are pointers to elements. The overhead of passing a packet along a connection is a single virtual method invocation.

An element in a click modular router has the following major properties:

- *Element class*: Like objects in an object-oriented program, each element

has a class that determines its behavior.

- *Input and output ports*: Ports are the end points of connections between elements. An element can have any number of input or output ports, which can have different semantic meanings. A normal and an error output, for example.

- *Configuration String*: Some element classes support additional arguments, used to initialize per-element state and fine-tune element behavior.

Click provides two kinds of connections between elements, push and pull. In a push connection, the upstream element hands a packet to the downstream element; in a pull connection, the downstream element asks the upstream element to return a packet. Each kind of packet hand off is implemented as virtual function. Packet arrival usually initiates push processing, which stops when an element discards the packet or stores it for later. Output interfaces initiate pull processing when they are ready to transmit data. Apart from implementing push or pull methods an element can be implemented with agnostic ports, meaning it can work as either push or pull depending on its context in the router. Push outputs must be connected to push inputs and pull output must be connected to pull inputs. Each agnostic port must be used as push or pull exclusively and when connecting two agnostic ports together they both must be used as either push or pull ports.

The next section explains the click modular router configuration and functioning with an example of click based trivial router implementation.

2.2 A sample IP Router configuration

This section shows how a real router configuration - an IP router that discards ARP Query and Responses and forwards only IP packets. The rest of this section describes the IP router in more detail. Figure 1 shows the sample IP router configuration. Figure 2 shows the configuration file for the sample IP router. The rest of this section describes the sample IP router in detail.

Figure 1: A Sample IP Router Configuration

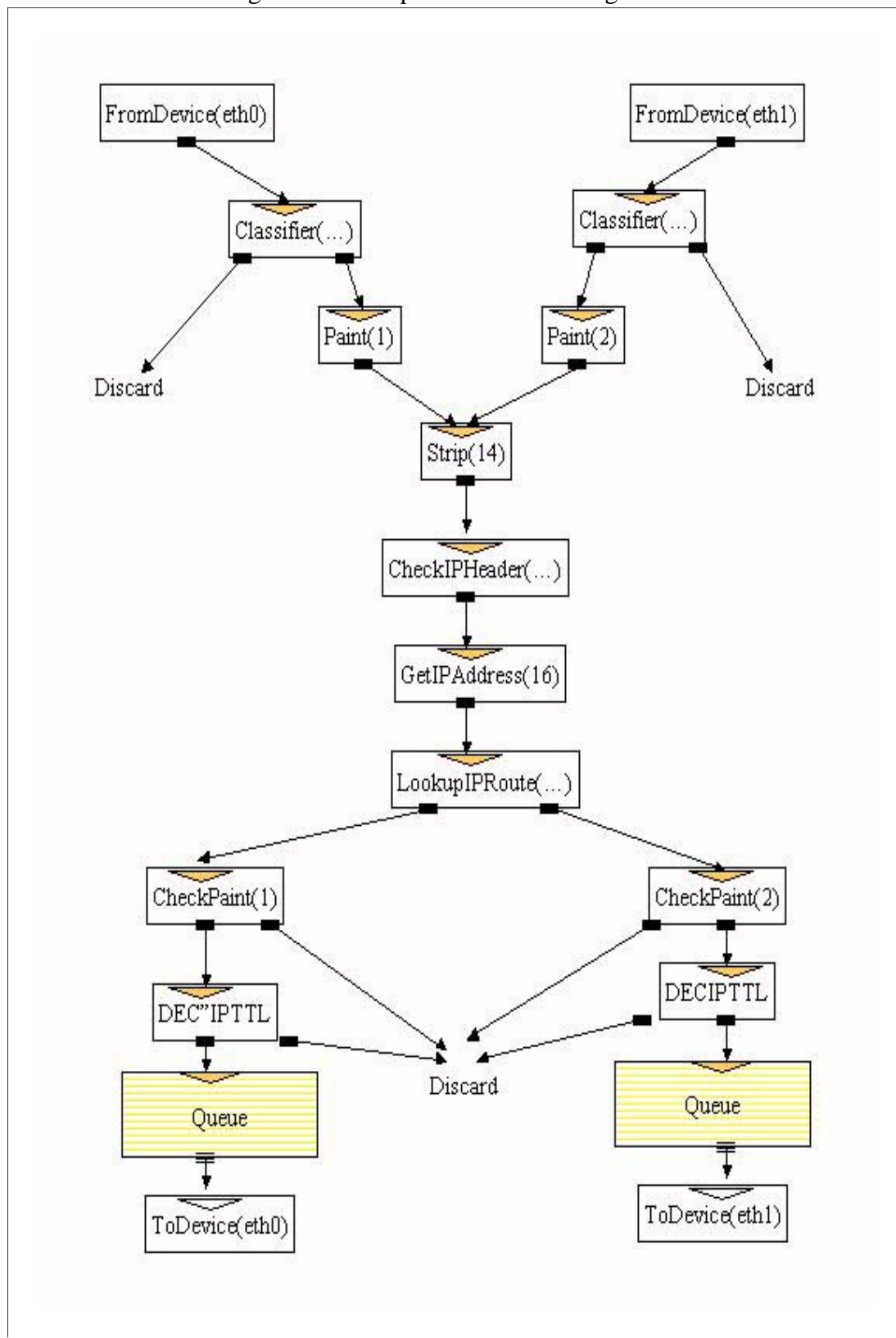


Figure 2: Configuration file for the sample IP Router

```

From Device(eth0) ->
c0::Classifier(12/0806 20/0001, 12/0806 20/0002, 12/0800, -);
out0:: Queue(1024)->ToDevice(eth0);
FromDevice(eth1)->
c1::Classifier(12/0806 20/0001, 12/0806 20/0002, 12/0800, -);
out1::Queue(1024)->ToDevice(eth1);
lkup::LookUpIpRoute;
co[0] -> Discard; co[1] -> Discard; c0[2] -> Discard;
c0[3]->Paint(1)-> Strip(14)->CheckIPHeader()->GetIPAddress(16)->lkup;
c1[0]->Discard; c1[1]->Discard; c1[2]->Discard;
c1[3]->Paint(2)->Strip(14)->CheckIPHeader()->GetIPAddress(16)->lkup;
lkup[0]-> ch0::CheckPaint(1); lkup[1]->ch1::CheckPaint(2);
ch0[0]->dttl0::DecIPTTL; ch0[1]->Discard;
dttl0[0]->out0; dttl0[1]->Discard;
ch1[0]->dttl1::DecIPTTL; ch1[1]->Discard;
dttl1[0]->out1; dttl1[1]->Discard;

```

The sample router upon receiving packets from the interface pushes them on to the classifier. The classifier identifies the packet type and pushes them on to various elements depending on the packet type. In the current configuration ARP queries, ARP responses and Non IP Packets are pushed to the Discard element. Only the IP packets are pushed on to the Paint element. The paint element marks the packets so that it can be verified whether it is being sent back on the same interface it was received. The paint element upon marking the packet pushes them on to the Strip element that removes the ethernet header and then the packet is checked for a valid IP header. Then the source and destination ip addresses are extracted from the packet using GetIPAd-

dress. The packet is then forwarded to the LookUpIPRoute Element which pushes the packet towards either interface 0 or interface 1. Before the packet was put on the queue the packet was checked for the interface on which it was received by the CheckPaint element which discards the packet if it is being sent on the same interface it was received on. After that the DecIPTTL element discards the TTL value in the packet and pushes the packet on to queue associated with that interface if the TTL value is not zero. Once the packet is pushed on to the queue, it waits there till it gets pulled by the ToDevice element which then sends it on the actual interface.

This example shows that it is easy to specify complex routing or packet processing behavior using the existing elements. This may also lead to a tendency to design smaller elements so the elements can be reused on other router configurations. But one need to be cautious when designing such an element as click relies on packet flow as an organizational principle. Each element must be self contained and it should be capable of doing all the required processing with the packet it receives rather than relying on some information being passed by other elements.

Chapter 3

Temporally Ordered Routing Algorithm

This chapter describes the Temporally Ordered Routing Algorithm (TORA). Refer to [4] for a complete description of the TORA standard.

3.1 Introduction to TORA

TORA is a distributed routing algorithm. It is developed mainly to solve the problem of routing in mobile wireless network. The mobility of the routers and the variability of other connectivity factors result in a network with potentially rapid and unpredictable changes. Congestion is also a major factor as wireless links mostly have lower bandwidth than the wired links. Existing shortest path algorithms and adaptive shortest path algorithms are not well suited for this environment as these algorithms may not be able to react fast enough to potentially rapid topological changes of a mobile wireless network. This leads to flooding in the network which causes congestion in a low bandwidth network to make matters more worse. Also these algorithms only provide one shortest path route between any source/destination pair which increases the chances for congestion. Though the link-state routing algorithms are capable of providing multi path routing, the communication overhead associated with maintaining full topological knowledge at each node makes them also not suitable for this environment.

There are some existing algorithms such as GafniBertsekas (GB), Lightweight

Mobile Routing (LMR), Destination-Sequenced Distance Vector (DSDV), Dynamic Source Routing (DSR) Protocol and Wireless Routing Protocol (WRP) each has its own drawbacks. Refer to [5] for a discussion on the drawbacks of the above mentioned algorithms.

TORA addresses the above mentioned problems in the mobile wireless network by providing the following minimal routing functionalities:

- Distributed routing - Each router needs to maintain information about only its adjacent routers.
- Loop free routing at each router
- Fast route establishment to enable the usage of routing information before it becomes irrelevant
- Multiple route establishment to alleviate congestion
- Though TORA supports route optimization, it is not an important function as TORA is meant for dynamic fast changing network.
- Minimizes algorithmic reactions/communications overhead to minimize bandwidth utilization. It reacts only when necessary, not for every topological change. Also the scope of failure reaction is local to the router.
- Though it provides two modes of operation namely Reactive mode and Proactive mode, TORA favors Reactive mode to minimize communication overhead as it may not be necessary to maintain routing between all possible source/destination pairs all the time.
- In the event of a network partition, the protocol detects the partition and erases all invalid routes within a finite time.

3.2 The Protocol

TORA is designed to work on top of lower layer mechanisms or protocols that provide the following basic services:

- Link Status sensing and neighbor discovery

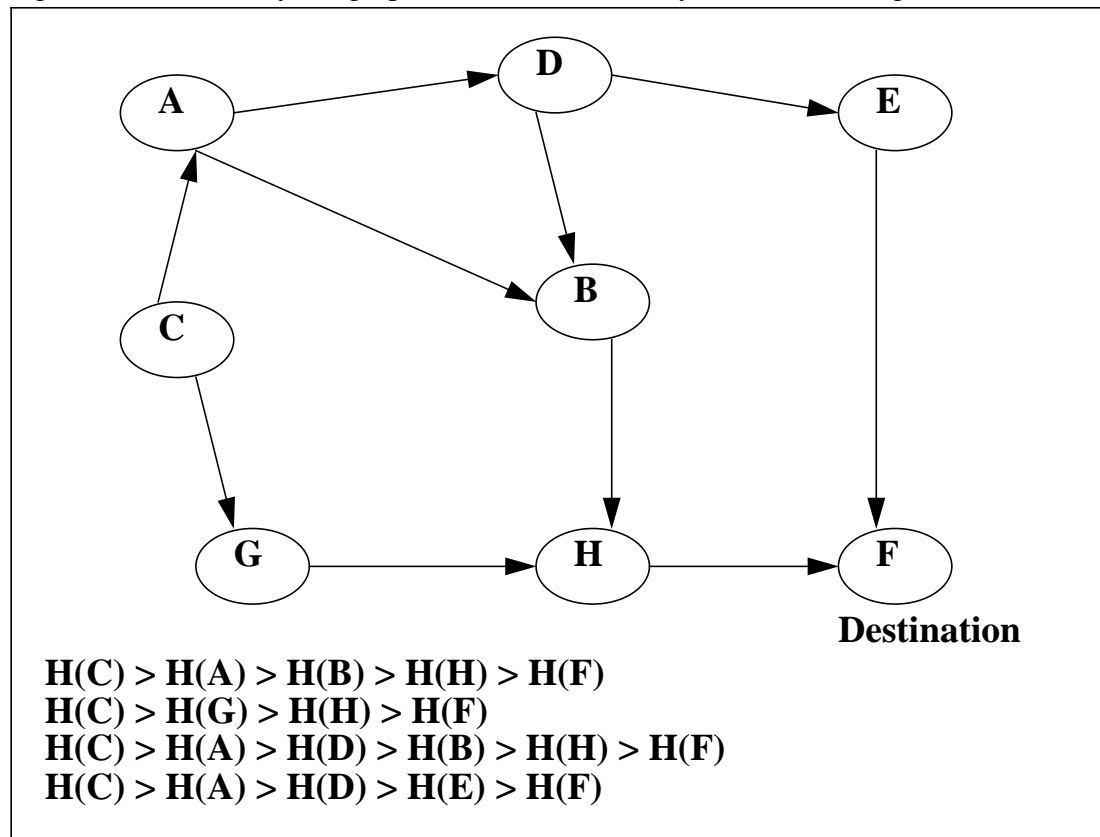
- Reliable in-order control packet delivery
- Link and network layer address resolution and mapping
- Security authentication

Events such as the reception of control messages and changes in connectivity with neighboring routers trigger TORA's algorithmic reactions.

A logically separate version of TORA is run for each "destination" to which routing is required. The following discussion focuses on a single version of TORA running for a given destination. The term "destination" refers to a traditional IP routing destination, which is identified by an IP address and mask. Thus the route to a destination may correspond to the individual address of interface or a group of addresses.

TORA assigns directions to the links between routers to form a routing structure that is used to forward datagrams to the destination. The term "link" refers to a "logical connection" consisting of the logical union of one or more connections between two MANET routers. A router assigns a direction ("Upstream" or "Downstream") to the link with a neighboring router based on the relative values of a metric associated with each router. The metric maintained by a router can conceptually be thought of as the router's "height" (i.e. Links are directed from the higher router to the lower router). Links from a router to any neighboring routers with an unknown or undefined heights are considered undirected. The router can only forward datagrams in the downstream. Together the router heights and link directional assignments form a loop-free, multi path routing structure in which all directed paths lead downstream to the destination. Figure 3 shows a directed acyclic graph defined by the relative heights of the routers. It also shows a multi path routing structure in which all directed links leading downstream to the destination F.

Figure 3: Directed acyclic graph of routers defined by the relative height of the routers



TORA has four basic functions namely

- Creating Routes
- Maintaining Routes
- Erasing Routes
- Optimizing Routes

The fourth functionality of TORA, Optimizing Routes is considered less important and [4] has not defined this functionality of TORA yet. So, the rest of this section only describes the first three functionalities along with description of Height Metric and Link Direction Assignment.

3.2.1 Height Metric

Each node I contains a “Height” - a quintuple $(\tau, \text{oid}, r, \delta, I)$ which is used in assigning directions to links associated with each neighbor based on their height for

each destination. Table 1 describes the elements of the height metric.

Table 1: Height metric description

Attribute	Definition
tau	“Logical time” of a link failure.
oid	Unique ID of the node that defined the reference level.
r	Reflection indicator
delta	Propagation ordering parameter
I	Unique ID of the node

Heights can be ordered lexicographically.

3.2.2 Link Direction Assignment

Link direction assignment is based on height of node I and height of corresponding neighbor J. Table 2 describes the Link direction assignment procedure for various values of height of node I and the height of its neighbor J.

Table 2: Link direction assignment

Height[I] == NULL	Height[J]== NULL	Height[I] > Height[J]	Height[I] < Height[J]	Link Direction
TRUE	X	X	X	UNASSIGNED
FALSE	TRUE	X	X	DOWN
FALSE	FALSE	TRUE	X	DOWN
FALSE	FALSE	FALSE	TRUE	UP

The ordering of non-null heights always forms a directed acyclic graph.

3.2.3 Creating Routes

Creating routes can be initiated on-demand by a source in reactive mode or by a destination in proactive mode. In both cases routers select heights with respect to a given destination and assign directions to the links between neighboring routers.

In the reactive mode, a source initiates route discovery by sending a QRY

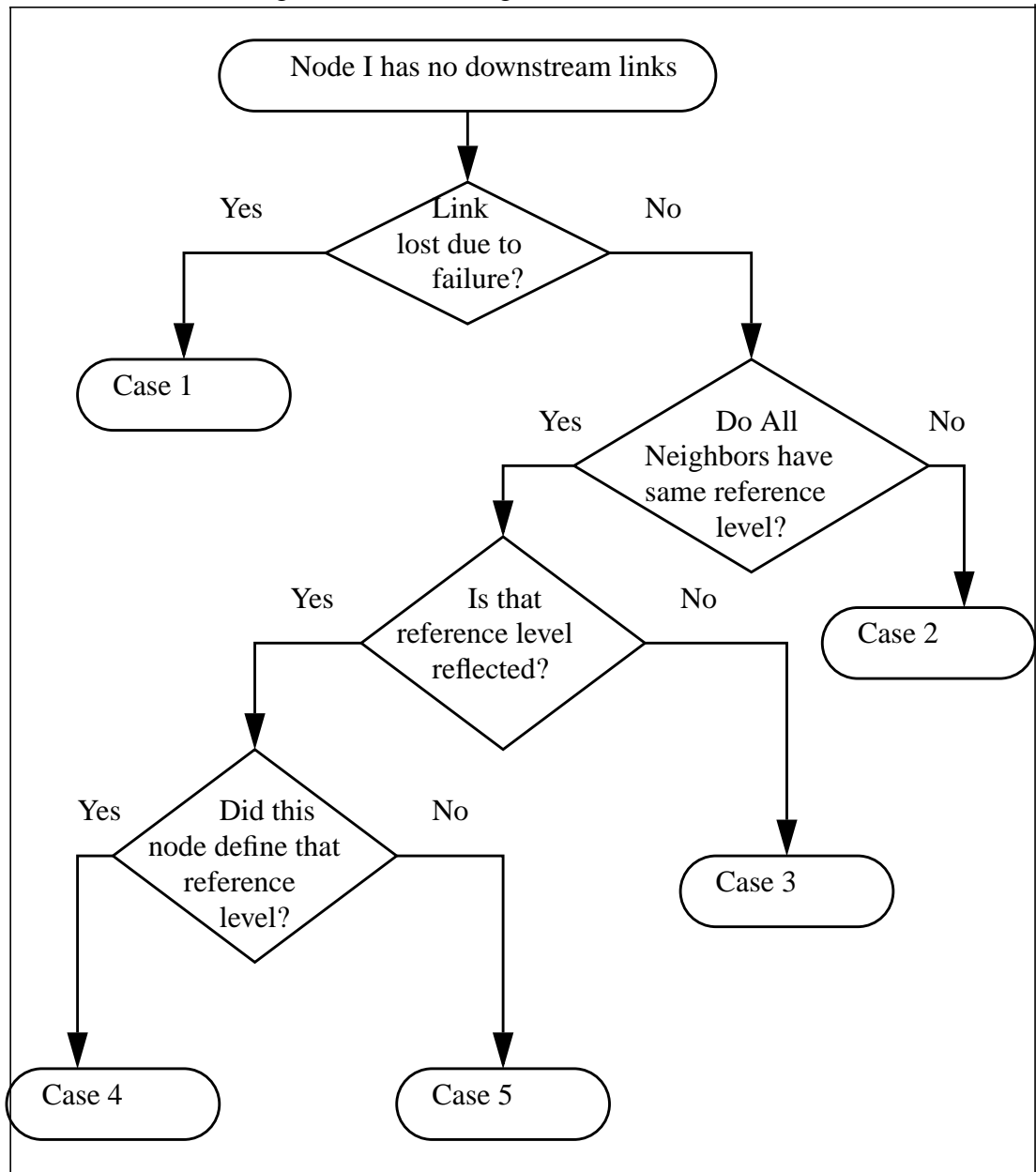
packet to all its neighbors. The QRY packet is propagated until it is received by one or more routers that have a trusted route to the destination. The routers set the Route requested flag for each destination as they forward QRY packets and discard subsequent QRY packets for the same destination. The router that has a trusted route to the destination responds to the QRY by sending an UPD packet to all its neighbors. Routers also maintain the time at which last UPD packet was sent and also the time at which the link became active to avoid redundant replies. Routers with a route requested flag set for a particular destination will send QRY packets for that destination when ever a new link gets activated to ensure route creation. Upon receiving the UPD packet the router sets its height and sends an UPD packet to its neighbors along with the destination information and the height information of the router sending the UPD packet. Thus routers learn of their neighbors height with respect to a given destination and assign link directions to them accordingly.

In the proactive mode the destination initiates route creation by sending OPT packet to all its neighbors which is then processed and forwarded by neighboring routers to its neighboring routers. Apart from identifying the destination the OPT packet contains a sequence number to ensure that each router process and forwards a given OPT packet from a destination only once.

3.2.4 Maintaining routes

Maintaining routes is performed only for routers that have a non NULL height. Routers with NULL height are not used for computations. A node I is said to have no downstream links if $\text{Height}[I] < \text{Height}[K]$ for all non NULL neighbor of K. This will result in one of the five possible reactions depending on the state of the node and the preceding event. Figure 4 shows the route maintenance decision tree. The rest of the section describes the processing involved for each case.

Figure 4: Maintaining Routes - Decision Tree



3.2.4.1 Case 1(Generate)

Node I has no downstream links due to link failure. This node defines a new reference level.

Assuming node I has at least one upstream neighbor, set Height[I] to $(t, I, 0, 0, I)$, where t is the time of the failure. If there are no upstream neighbors for I, then set Height[I] to NULL.

3.2.4.2 Case 2(Propagate)

Node I has no down stream links due to a link reversal following reception of an UPD packet and the ordered sets $(\tau[K], \text{oid}[K], r[K])$ are not equal for all neighbors K. Node I propagates the reference level of its highest neighbor and selects a height that is lower than all neighbors with that reference level.

Set $(\tau[I], \text{oid}[I], r[I])$ to maximum of $(\tau[K], \text{oid}[K], r[K])$ for all neighbors K.

Set $(\delta[I], I)$ to $(\delta[M]-1, I)$ where M is the lowest neighbor with the maximum reference level defined above.

3.2.4.3 Case 3(Reflect)

Node I has no downstream links due to a link reversal following reception of an UPD packet and the ordered sets $(\tau[K], \text{oid}[K], r[K])$ are equal with $r[K] = 0$ for all neighbors K. In essence the same level which has not been reflected has propagated to node I from all of its neighbors. Node I reflects back a higher sub level by setting the value of r to 1.

Set $(\tau[I], \text{oid}[I], r[I], \delta[I], I) = (\tau[K], \text{oid}[K], 1, 0, I)$

3.2.4.4 Case 4(Detect)

Node I has no downstream links due to a link reversal following reception of an UPD packet, the ordered sets $(\tau[K], \text{oid}[K], r[K])$ are equal with $r[K] = 1$ for all neighbors K, and $\text{oid}[K] = I$.

The last reference level defined by node I has been reflected and propagated back as a higher sub-level from all of its neighbors. This corresponds to detection of a partition.

Set Height[I] to NULL and initiate the process of erasing routes as described in Section 3.2.5, "Erasing Routes".

3.2.4.5 Case 5(Generate)

Node I has no downstream links due to a link reversal following reception of an UPD packet. The ordered sets $(\tau[K], \text{oid}[K], r[K])$ are equal with $r[K] = 1$ for all neighbors K, and $\text{oid}[K] \neq I$.

Node I experienced a link failure which did not require a reaction between the time it propagated a reference level and the reflected higher sub-level returned from all neighbors. This is not necessarily an indication of a partition. Node I defines a new reference level.

Set $(\tau[I], \text{oid}[I], r[I], \delta[I], I) = (t, I, 0, 0, I)$, where t is the time of the failure.

3.2.5 Erasing Routes

Following detection of a partition (See Section 3.2.4.4, “Case 4(Detect)”) node I sets its height and the height entry for each neighbor K to NULL unless the destination J is a neighbor, in which case the corresponding height entry is set to Zero. It updates all the entries in its link-status table, and broadcast a CLR packet. The CLR packet consists of the destination Id J, and the reflected reference level of node I, $(\tau[I], \text{oid}[I], 1)$. In actuality the value $r[I] = 1$ need not be included since it is always 1 for a reflected reference level. When a node I receives a CLR packet from a neighbor K it reacts as follows:

- If the reference level in the CLR packet matches the reference level of node I, it sets its height and the height entry for each neighbor node K to NULL (unless destination J is a neighbor, in which case the corresponding height entry is set to Zero), updates all the entries in its link status table and broad casts a CLR packet.

- If the reference level in the CLR packet does not match the reference level of node I, it sets the height entry for each neighbor K (with the same reference level as the CLR packet) to NULL and updates the corresponding link status table entries. Thus the height of each node in the portion of the network that was partitioned

is set to NULL and all invalid routes are erased. If this causes node I to lose its last downstream link, it reacts as in Section 3.2.4.1, “Case 1(Generate)”.

Chapter 4

Modular Implementation of Temporally Ordered Routing Algorithm

This chapter describes a modular implementation of Temporally Ordered Routing Algorithm (TORA) using the click modular router.

4.1 Overview

This section provides an overview of the modular implementation of TORA. This implementation consists of many new elements developed specifically for TORA apart from reusing existing common click elements. The following are the TORA specific elements newly developed:

- ToraRouter
- ToraHeight
- ToraInterface
- ToraNeighborInfo
- ToraNeighbor
- ToraDestination
- ToraSendQRY
- ToraProcessQRY
- ToraProcessUPD
- ToraProcessCLR
- ToraLookUpIpAddress

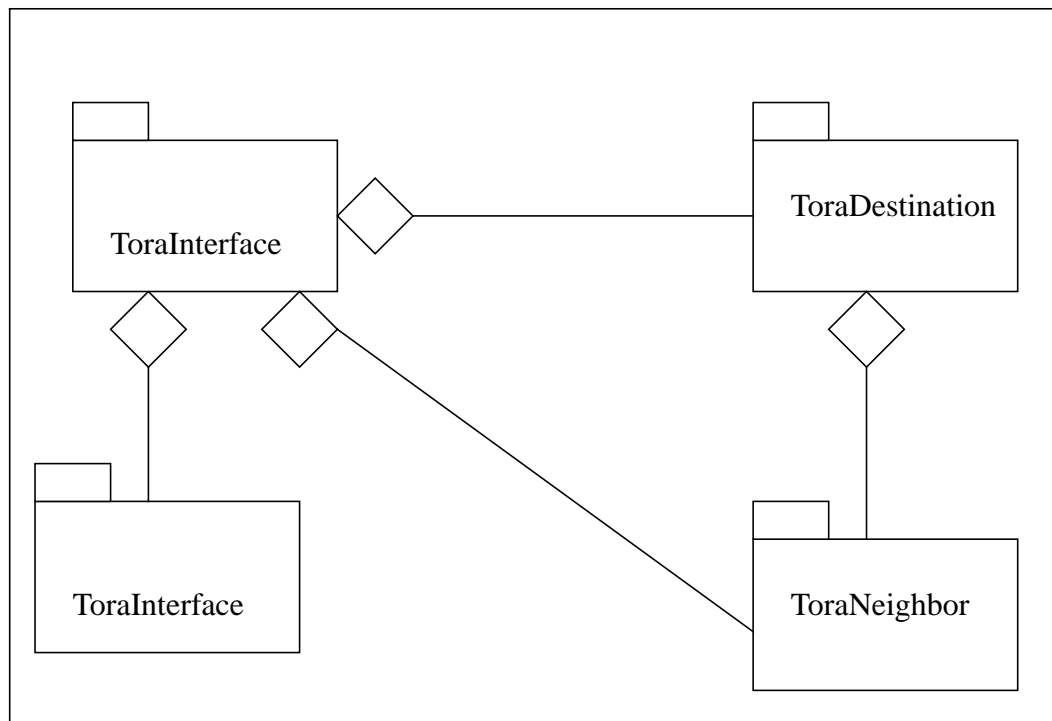
- ToraGetNextHopNeighbor
- ToraDispatchControlPkt
- ToraOpt
- ToraOutput
- ToraSendPending

Out of the above classes, the following classes together implement the routing table for the TORA router.

- ToraRouter
- ToraHeight
- ToraInterface
- ToraNeighbor

Figure 5 shows the class diagram for the Tora Routing Table.

Figure 5: Class diagram for Tora Routing Table



Since the nclick environment does not have a link layer mechanism to dis-

cover neighbors the following classes together implement a neighbor discovery mechanism:

- ToraNeighborInfo
- ToraOpt

The rest of the chapter describes the architecture in detail.

4.2 Architecture

This section describes the architecture of the modular implementation of TORA using the click modular router. TORA routes user data using normal TCP, UDP or ICMP packet types. Input to the TORA router comes from both the ethernet interface and from the Linux kernel. Figure 6 shows the elements involved in handling the input to the Tora Router from the network interface. Packets received from the interface are painted to identify the interface on which they were received. Then the packets are classified into unicast and broadcast packets. The broadcast packets may be Tora Control packets. Then the MAC address is extracted from both types of packets. Then the packets are applied a common transformation as shown in Figure 7. During the common transformation it removes the ethernet header from the packet and then validates the IP header of the packet. It then marks the IP header of the packet and extracts the IP address. After which it sets the ADHOC header annotation. Then for unicast packets it checks to see if the packet is destined for the local node. If so it sends it to the Linux kernel. If it is meant for some other node it then tries to look for the next hop to send the packet to as shown in Figure 9. In case of the broadcast packets it looks at their type to see whether if it is a Tora Control packet. If so it then sends it to the ToraDispatchControlPkt element for further processing as shown in Figure 10. If it is not a Tora Control packet it discards that packet.

Figure 6: Input elements - From Network Interface

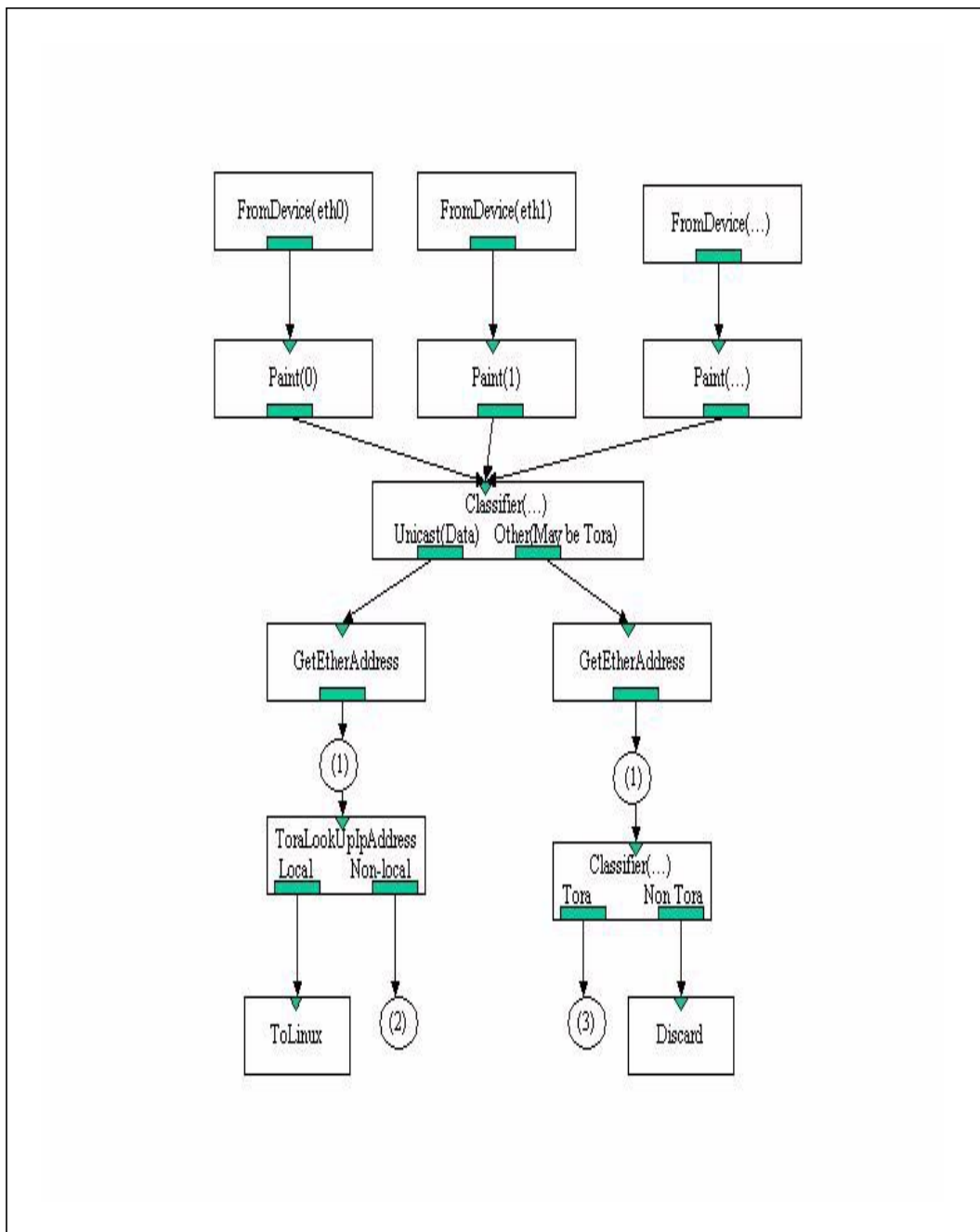


Figure 8 describes how the input received from the kernel is handled by the Tora router. It validates the IP header of the packets it is receiving, extracts the IP address from the packet and classifies the packets into IP and ICMP packets. It discards the ICMP packets and passes on the IP packets to another classifier element to distinguish between Tora and Non Tora packets. If it is a Tora control packet it for-

wards that packet to the ToraDispatchControlPkt element for further processing as shown in Figure 10. If it is not a Tora control packet, it forwards that packet to ToraLookupIpAddress element which then forwards it to the kernel if it is destined for this node and to ToraGetNextHop for further processing as shown in Figure 9 if it is not destined for this node.

Figure 7: Common Elements

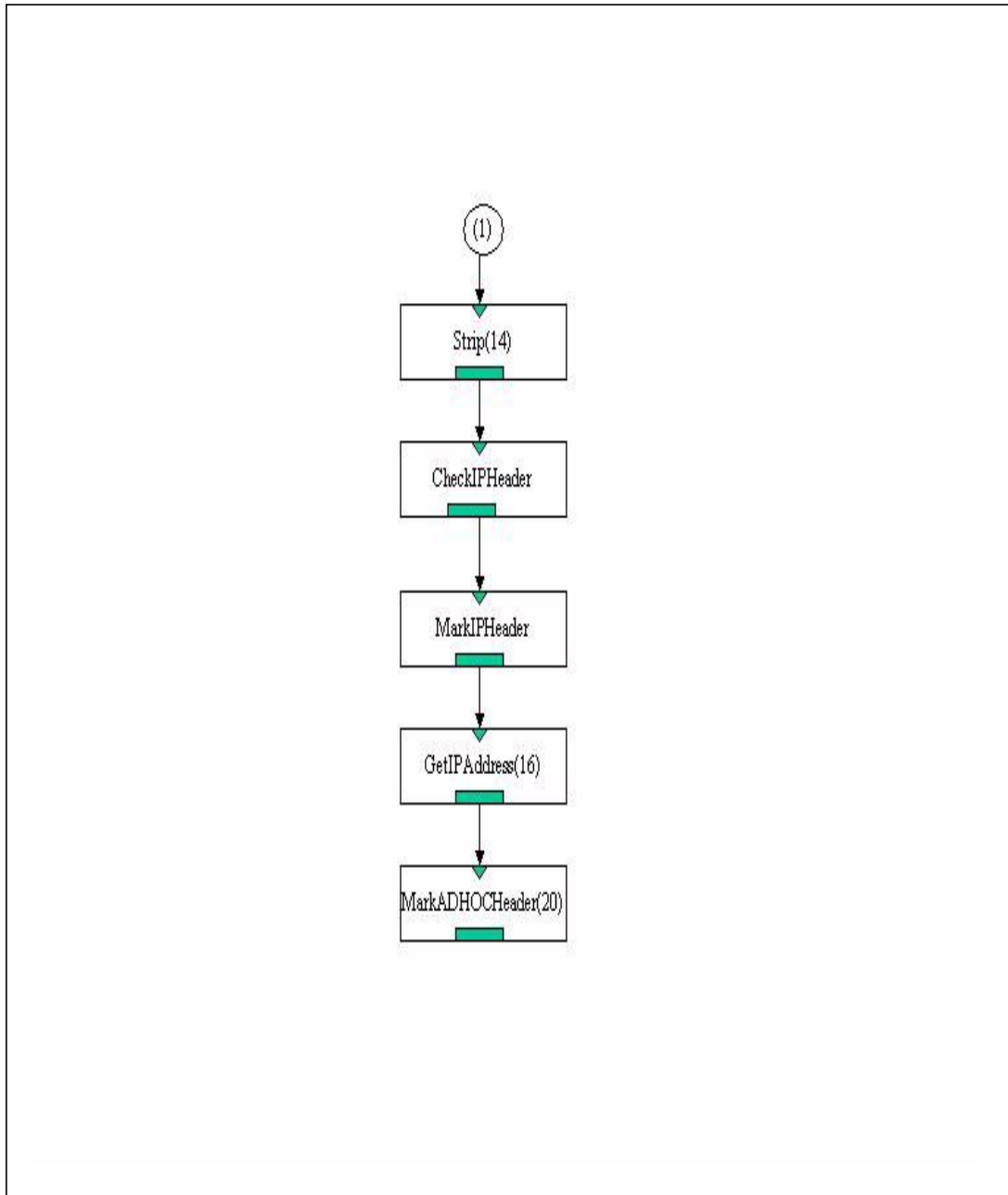
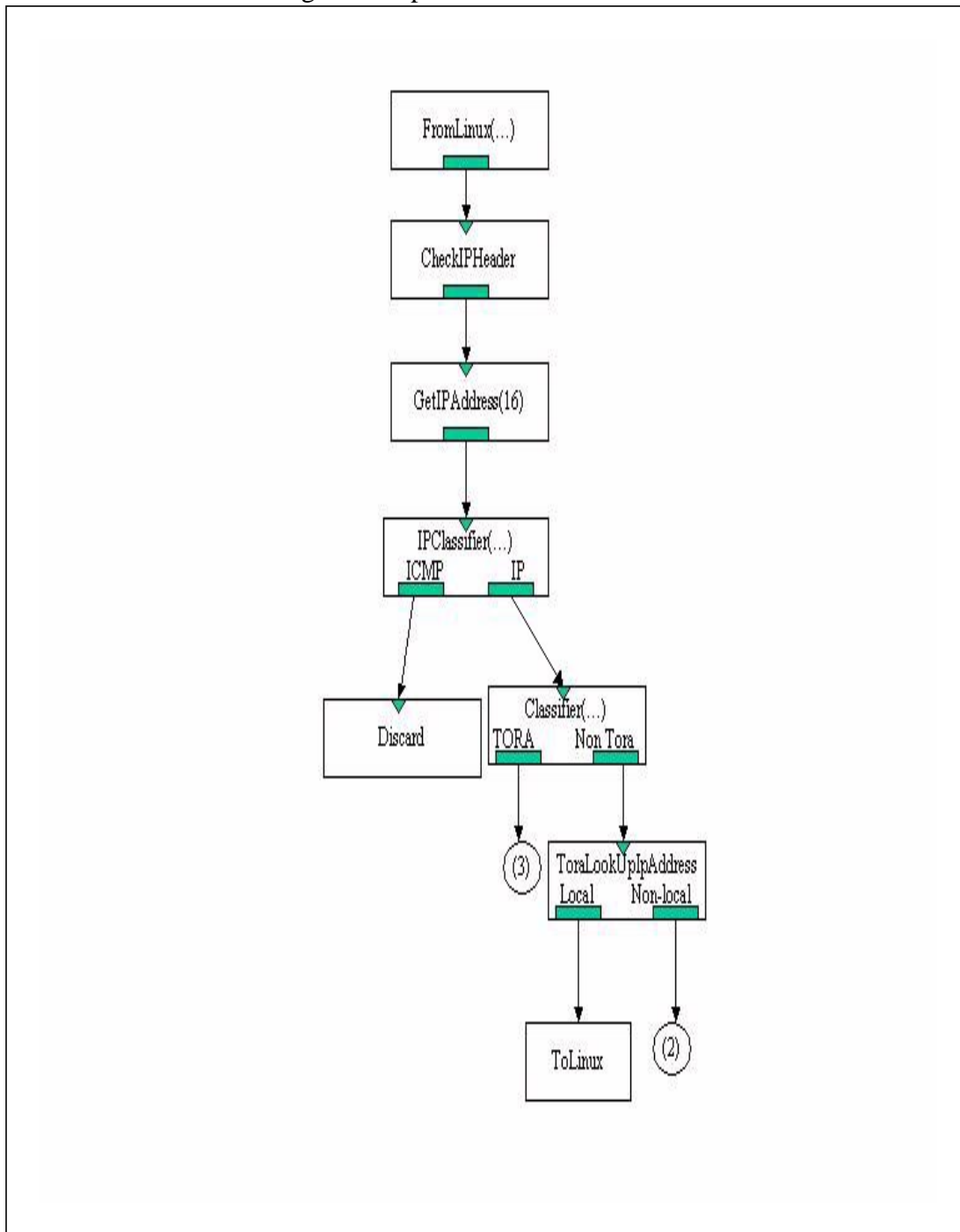


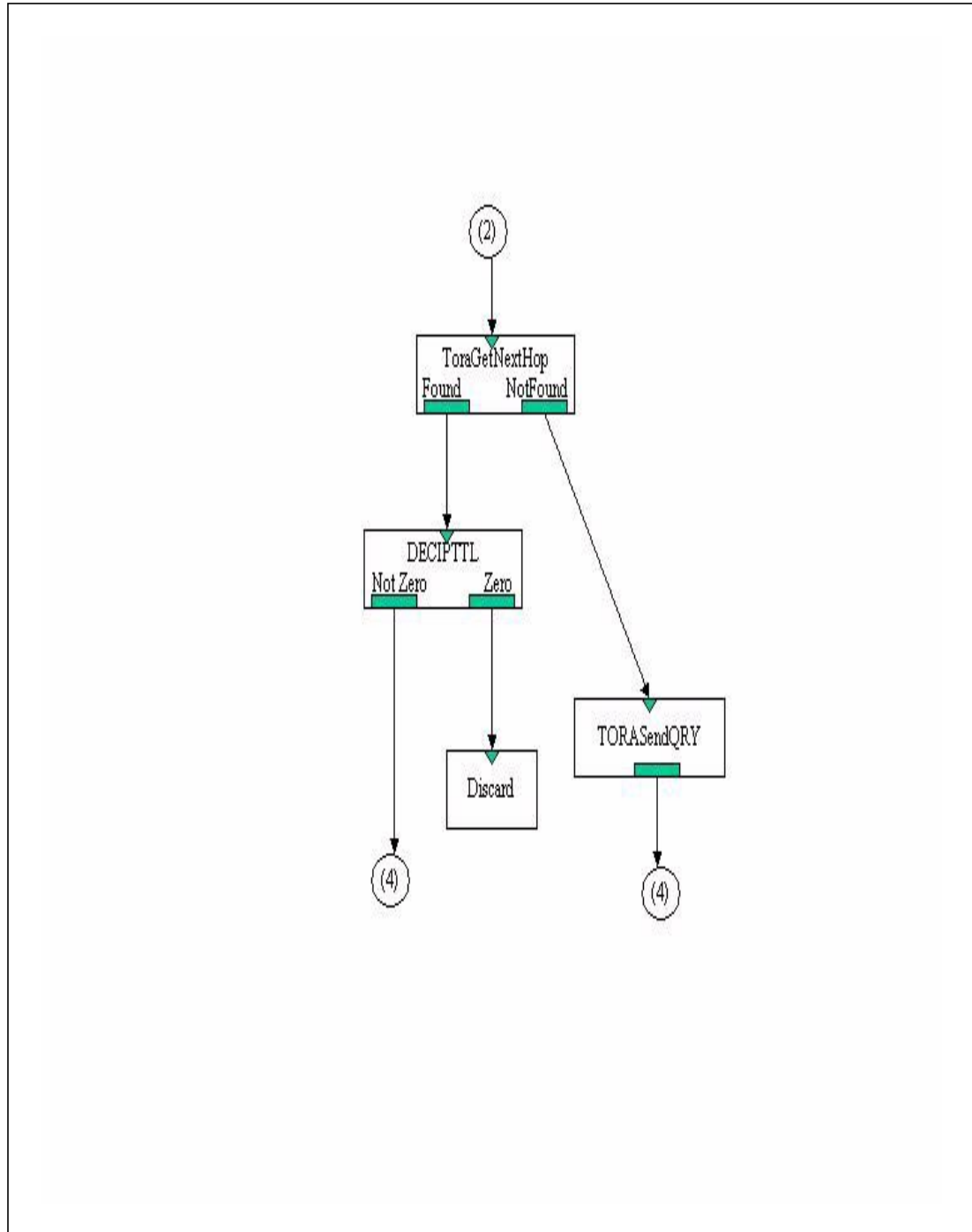
Figure 8: Input element - From kernel



The GetNextHop component as shown in Figure 9, tries to find an active next hop neighbor for this destination. If it finds one, then it decrements the TTL of the packet. If it is not zero it sends to the output element to be sent on the appropriate network interface as shown in Figure 11. If the TTL value becomes zero after decremented then it discards the packet. If it could not find an active next hop neighbor, it

initiates route discovery for this destination by sending a QRY packet through the network interface as shown in Figure 11, if it is not already done. It queues the packet for that destination so that it could be delivered to that destination when a route is found.

Figure 9: Get Next Hop



The DispatchControlPacket component as shown in Figure 10, handles the Tora Control packets. It decrements the TTL value of the TORA control packets. If it is zero then it discards them. If it is not zero, then the packet is forwarded to ToraDispatchControlPkt element for further processing. The ToraDispatchControlPkt forwards the packet to ToraProcessQRY, ToraProcessUPD, ToraProcessCLR and ToraOpt elements for QRY, UPD, CLR and OPT packets respectively. It discards other types of packets. These elements in turn process the QRY/UPD/CLR packets as described in Section 3.2, “The Protocol”. The OPT packets alone are handled a bit differently. The OPT packets are not really the TORA opt packets. Though these packets advertise the destination like the TORA OPT packets this is not used to implement proactive mode as suggested by [4]. Instead this is a variant of Tora OPT packets which does not have the height information. The primary purpose of this packet is to enable neighbor discovery as there is no link layer mechanism which provides such a capability present in click modular router. A similar approach has been implemented in the click AODV implementation as well. Refer to [3] for further information. The OPT packets are sent by ToraOpt element at preconfigured intervals to enable neighbor discovery. The ToraDispatchControlPkt element dispatches the OPT packet to ToraOpt element which then updates the neighboring list associated with that node and also each of the destination entries. The ToraDispatchControlPkt element discards packets which does not belong to any of the above types. The ToraDispatchControlPkt element also forwards the UPD and OPT packets to ToraSendPending element after forwarding them to ToraProcessUPD and ToraOpt elements respectively to enable sending pending packets to those destinations, if a next hop neighbor has become active for that destination due to the receipt of the UPD/OPT packet.

Figure 11 describes how the packets that need to be sent over the network interface are handled. The packets are first forwarded to IPFragmenter which discards invalid MTUs after fragmenting them and forwarding the valid ones to ToraOutput

element. The ToraOutput element adds the ethernet header and forwards the packets to queues associated with all the interfaces for broadcast packets and to the queue associated with a specific interface for unicast packets. The interface then pulls the packet from its queue and sends it over the network.

Figure 10: Dispatch Control Packet

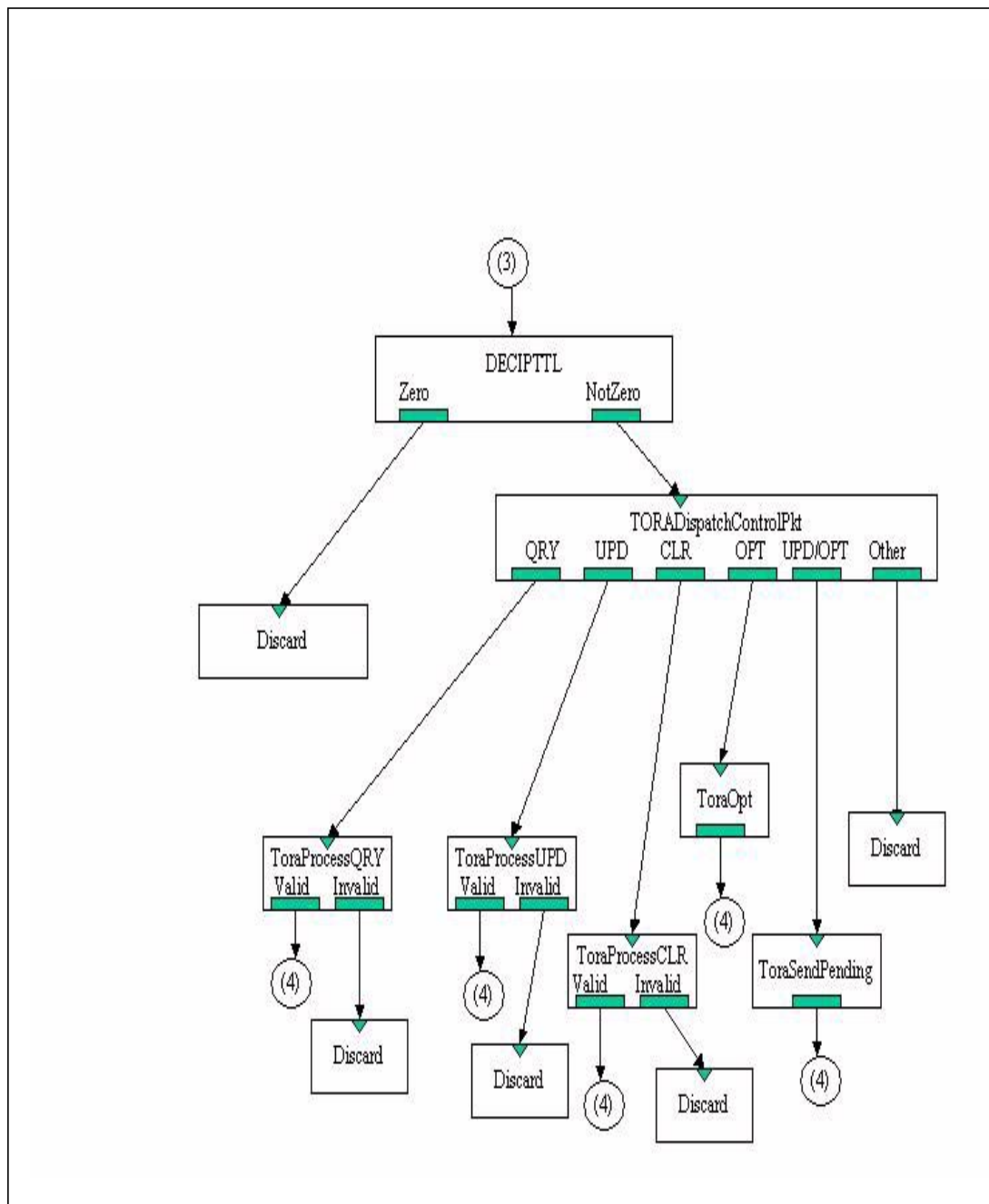
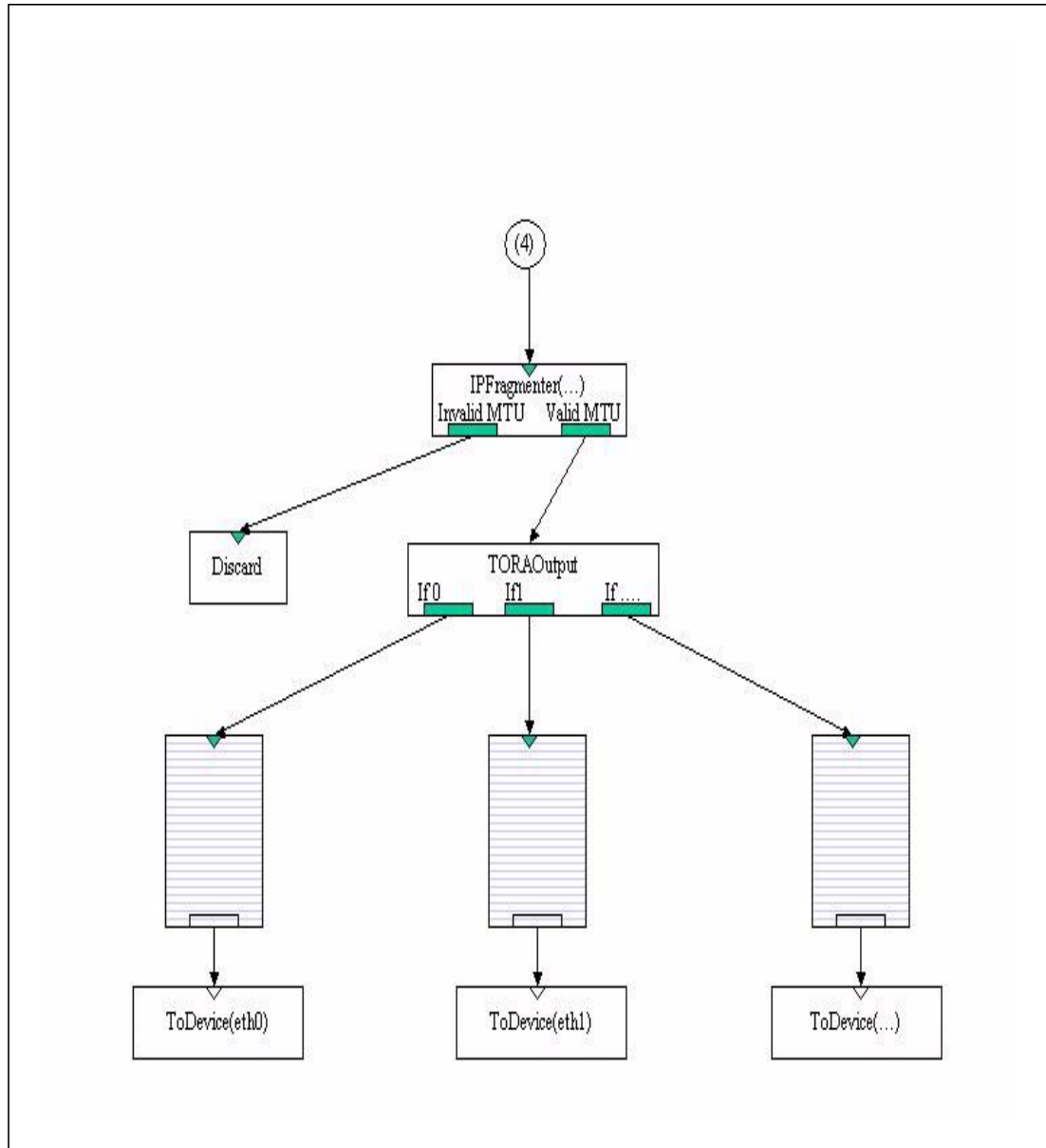


Figure 11: Output - Network Interface



Refer to Appendix A, “Click configuration file for Tora Router” for a Tora router click configuration file.

The rest of the section describes various elements developed specifically for the modular implementation of TORA. For a description of the common reused elements see [2].

4.2.1 ToraLookUpIpAddress

This element delivers the packet to Linux kernel, if the packet is destined for the one of the IP addresses of this system. If it is not destined for any of the IP addresses that belongs to this system, then it passes the packet on to ToraGetNextHopNeighbor for further processing.

4.2.2 ToraGetNextHopNeighbor

This element passes the packet to the ToraOutput element to be sent to the next hop if it finds a valid next hop neighbor for this destination. If it does not find any valid next hop neighbor, then it initiates route discovery for that destination and queues the packet.

4.2.3 ToraSendQRY

This element creates a QRY packet for a particular destination and passes it on to the ToraOutput element to be broadcast on all the interfaces of this router.

4.2.4 ToraDispatchControlPkt

This element looks at the Tora Packet type and passes it to the appropriate element for further processing. See Table 3 for information on various Tora Control Packet types and their handling element names. After completing the packet processing it forwards the packet to ToraSendPending element, if it is an OPT or UPD packet for to send any pending packets meant for that destination, if there is a new active neighbor found for that destination due to processing the last received OPT or UPD packet.

Table 3: Tora control packets and its handling elements

Packet Type	Handling Element Name
QRY	ToraProcessQRY

Table 3: Tora control packets and its handling elements

Packet Type	Handling Element Name
UPD	ToraProcessUPD
CLR	ToraProcessCLR
OPT	ToraOpt

4.2.5 ToraProcessQRY

This element processes the TORA QRY packet as described in section 4.7.5 of [4]. This element discards the packet if it has already received a QRY packet for the same destination. If it has not received a QRY packet before for this destination, then it checks the reference level of its height. If it is 0 and the activation time for the neighbor from which the packet was received is later than the update time for this destination, then it sets the update time to current time and sends an UPD packet to all its neighbors. If the reference level is not 0, and one of the neighbors have a height with reference level 0 for this destination then it updates the height of this destination with the height of the neighbor with the minimum height and reference level 0. It also increments the value of delta and sets the id field of the destination's height to the unique id of this node. It also updates the update time of the destination to current time and sends an UPD packet to all its neighbors. If the reference level of the height of the destination is not 0, and there is no such neighbor with a height with 0 reference level, then it sets the route requested flag and sends a QRY packet to all its neighbors if there are more than 1 active neighbor. Refer to section 4.7.5 of [4] for the QRY packet processing procedure and Appendix B, "Source code for ToraProcessQRY element" for the source code for the ToraProcessQRY element implementation.

4.2.6 ToraProcessUPD

This element processes the TORA UPD packet as described in section 4.7.6 of [4]. Refer to section 4.7.6 of [4] for the UPD packet processing procedure and Appen-

dix C, “Source code for ToraProcessUPD element” for the source code for the ToraProcessUPD element implementation.

4.2.7 ToraProcessCLR

This element processes the TORA CLR packet as described in section 4.7.7 of [4]. Refer to section 4.7.7 of [4] for the CLR packet processing procedure and Appendix D, “Source code for ToraProcessCLR element” for the source code for the ToraProcessCLR element implementation.

4.2.8 ToraOpt

This element implements a pseudo neighbor discovery mechanism to overcome the absence of such a mechanism in the nsclick link layer. This element sends a modified version of ToraOpt packet announcing the IP address of the router on each of its interfaces at preconfigured time intervals. This element also updates the neighbor information of TORA router itself and all the destination entries whenever it discovers a new neighbor or it loses track of a neighbor. Refer to Appendix E, “Source code for ToraOpt element” for the source code for the ToraOpt element implementation.

4.2.9 ToraSendPending

This element checks for a valid next hop neighbor for the destination associated with the TORA packet it was passed on and forwards any pending packets for that destination if it finds a valid next hop neighbor.

4.2.10 ToraOutput

This element adds the ethernet header and sends out the packet on a specific interface if it is a unicast packet and sends it on all the interfaces if it is a broadcast packet.

4.3 Conclusion and Future work

Overall the implementation activity under the click environment turned out to

be implementing elements which look at the contents of the packet and apply a set of transformations to them. It was easy to implement and due to the availability of lot of existing reusable common elements implemented for other protocols such as classifier, IPClassifier, Fragmenter etc., it also reduced the development effort required in implementing the protocol.

Though performance was not a criteria to be measured, the Tora Router implemented using click modules provided a comparable performance with the NS implementation of Tora under similar testing conditions. Considering the overheads involved in implementing a separate neighbor discovery mechanism using a customized Tora OPT packets due to the unavailability of a link layer mechanism which provides such a feature the performance of the Tora Router under nsclick provided a comparable throughput over a similar Tora implementation under NS. Using the same connection pattern and the scenario file, the Tora implementation under nsclick was able to send 1029 packets when compared to 1239 packets by the Tora implementation under NS. Accounting for the overheads associated with sending customized OPT packets every 1000 ms to implement a neighbor discovery mechanism under the nsclick environment this performance is comparable to that of the performance of the Tora implementation under NS.

Since the lack of a link layer mechanism is a big drawback of the click environment, and implementing such a mechanism for every protocol is an additional work that could be avoided, apart from adding performance overheads the future work should involve implementing a generic link layer mechanism for click environment.

References

- [1] Robert Morris, Eddie Kohler, John Jannotti & M.Frans Kasshoek. (1999).The Click Modular Router. 17th ACM Symposium on Operating Systems Principles(SOSP' 99).
- [2] Eddie Kohler, The Click Modular Router. Ph. D Thesis. Massachusetts Institute of Technology, February 2001.
- [3] Audun Tornquist, Michael Neufeld & Dirk Grunwald. The Design of a Modular Implementation of AdHoc Routing Protocols. INFOCOMM-2002 SUBMISSION.
- [4] V.Park, S.Corson, Flarion Technologies Inc, 20 July 2001, Temporally-Ordered Routing Algorithm (TORA) Version 1, Functional Specification. IETF MANET Working Group, INTERNET-DRAFT, draft-ietf-manet-tora-spec-04.txt
- [5] Vincent D. Park, M. Scott Corson, A Highly Adaptive Distributed Algorithm for Mobile Networks, Proceedings of INFOCOM'97.

Appendix A

Click configuration file for Tora Router

1.1 Click configuration file for Tora Router

This appendix provides a sample click configuration file for the Tora Router.

```
// tora-simrouter.click
// TORA router
// Run with
// click conf/tora-simrouter.click

// Address aliases
AddressInfo(me0 SIMNET:eth0);

// Tora Router
ToraRouter(me0);

// Packet classifier 0
// 0: local ip ethernet me0
// 1: ip ethernet bcast
torap_cl0 :: Classifier(!0/FFFFFFFFFFFFFF,-);
```

```
// Packet classifier 1
// 0: Tora Packet
// 1: others = Data packets . Look for route
torap_cl1 :: Classifier(09/F0,-);

// IPPacket Classifier
// ICMP
// IP
tora_ip_cl :: IPClassifier(icmp type dst_unreachable,-);

// Packet classifier 3
// 0: Tora Packet
// 1: others = Data packets . Look for route
torap_cl2 :: Classifier(09/F0,-);

// Lookup 1
// Local
// Non local
rt_torat :: ToraLookUpIpAddress;

// Tora Dispatch Control Packet
// QRY
// UPD
// CLR
// OPT
// Dispatch Pending
// Others - discard
```

```
rt_tora_dcp :: ToraDispatchControlPkt;
```

```
// TORA GetNextHop 0
```

```
// found
```

```
// Not found
```

```
rt_nh0 :: ToraGetNextHop;
```

```
// TORA Send QRY
```

```
tora_snd_qry :: ToraSendQRY;
```

```
// DecIPTTL 0
```

```
dec_ttl0 :: DecIPTTL;
```

```
// DecIPTTL 1
```

```
dec_ttl1 :: DecIPTTL;
```

```
//TORA Process received QRY packet
```

```
tora_pqry :: ToraProcessQRY;
```

```
// TORA Process received UPD packet
```

```
tora_pupd :: ToraProcessUPD;
```

```
// TORA Process received CLR packet
```

```
tora_pclr :: ToraProcessCLR;
```

```
// OPT
```

```
tora_opt :: ToraOpt(1000);
```

```
//Send pending packets
tora_snd_pending :: ToraSendPending;

// Output
// 0: interface 0
// 1: interface 1
// ...
tora_out :: ToraOutput;

//IP Fragmenter
// 0: Frgamented packets
// 1: error
tora_ipf :: IPFragmenter(2000);

// Input 0, Ethertap
ktapin :: FromSimDevice(tap0,4096);
ktapout :: ToSimDevice(tap0,IP);
out0 :: ToSimDump(tokernel,2000,IP) -> ktapout;

// Input 1
in1 :: FromSimDevice(eth0, 4096);
//out1 :: ToSimDevice(eth0,IP);

// from input 0
ktapin -> CheckIPHeader -> GetIPAddress(16) -> ToSimDump(fromker-
```

```
nel,2000,IP) -> tora_ip_cl;
```

```
// from input 1
```

```
//in1 -> Paint(0) -> torap_cl0;
```

```
in1->HostEtherFilter(me0)->ToSimDump(in_eth0)->Paint(0)->torap_cl0;
```

```
// from classifier 0
```

```
// Not broadcast, Data packet
```

```
torap_cl0[0] -> GetEtherAddress -> Strip(14) -> CheckIPHeader -> MarkIP-Header -> GetIPAddress(16) -> MarkAdhocHeader(20) -> rt_torat;
```

```
// Broadcast, may be tora
```

```
torap_cl0[1] -> GetEtherAddress -> Strip(14) -> CheckIPHeader -> MarkIP-Header -> GetIPAddress(16) -> MarkAdhocHeader(20) -> torap_cl1;
```

```
// from classifier 1
```

```
// Tora
```

```
torap_cl1[0] -> dec_ttl1;
```

```
// Non tora broadcast
```

```
torap_cl1[1] -> Discard;
```

```
// from classifier ip
```

```
// ICMP
```

```
tora_ip_cl[0] -> Discard;
```

```
// IP
```

```
tora_ip_cl[1] -> torap_cl2;
```

```
// from classifier 2
```

```
// Tora
torap_cl2[0] -> dec_ttl1;

//Non Tora
torap_cl2[1] -> rt_torat;

//from dec_ttl1

//Lookup
dec_ttl1[0] -> rt_tora_dcp;

//Discard
dec_ttl1[1] -> Discard;

// from rt_torat

// Local destination
rt_torat[0] -> out0;

// Not local. Find the next hop
rt_torat[1] -> rt_nh0;

// from rt_nh0

// Next Hop found
rt_nh0[0] -> dec_ttl0;

// Next Hop not found
rt_nh0[1] -> tora_snd_qry;

// from dec_ttl0

// Send it to the next hop
dec_ttl0[0] -> tora_ipf;

// Discard
```

```
dec_ttl0[1] -> Discard;

// from rt_tora_dcp
rt_tora_dcp[0] -> tora_pqry;
rt_tora_dcp[1] -> tora_pupd;
rt_tora_dcp[2] -> tora_pclr;
rt_tora_dcp[3] -> tora_opt;
rt_tora_dcp[4] -> tora_snd_pending->toro_ipf;
rt_tora_dcp[5] -> Discard;

// from torap_qry
tora_pqry[0] -> tora_ipf;
tora_pqry[1] -> Discard;

// from torap_upd
tora_pupd[0] -> tora_ipf;
tora_pupd[1] -> Discard;

// from torap_clr
tora_pclr[0] -> tora_ipf;
tora_pclr[1] -> Discard;

//from tora_opt
tora_opt[0] -> tora_ipf;

//from tora_snd_qry
tora_snd_qry -> tora_ipf;
```

```
//from tora_ipf
```

```
tora_ipf[0] -> tora_out;
```

```
tora_ipf[1] -> Discard;
```

```
//from tora_out
```

```
tora_out[0] -> Queue ->ToSimDump(out_eth0)->ToSimDevice(eth0,IP);
```

Appendix B

Source code for ToraProcessQRY element

2.1 Source code for ToraProcessQRY element

This appendix provides the source code for ToraProcessQRY element.

```
// ToraProcessQRY.cc: implementation of the ToraProcessQRY class.
// Implements Tora QRY packet processing as defined in section 4.7.5 of
// Temporally-Ordered Routing Algorithm (TORA) Version 1
//     Functional Specification
// Nandesh Kumar Palanisamy
////////////////////////////////////
#include <click/config.h>
#include <click/package.hh>
#include <click/element.hh>
#include <click/etheraddress.hh>
#include <click/ipaddress.hh>
#include <click/timer.hh>
#include <click/hashmap.hh>
#include <click/click_adhoc.hh>
#include <click/click_ether.h>
#include <click/confparse.hh>
```

```
#include <click/bitvector.hh>
#include <click/error.hh>
#include <click/glue.hh>
#include <click/click_adhoc.hh>
#include <click/click_ip.h>
#include <click/router.hh>
#include <sys/queue.h>
#include <click/packet_anno.hh>
```

```
#include "Tora.hh"
#include "ToraNeighbor.hh"
#include "ToraDestination.hh"
#include "ToraRouter.hh"
```

```
#include "ToraProcessQRY.hh"
```

```
#ifdef CLICK_SIM
using namespace CLICKSIM;
#endif
```

```
////////////////////////////////////
```

```
// Construction/Destruction
```

```
////////////////////////////////////
```

```
ToraProcessQRY::ToraProcessQRY()
{
    MOD_INC_USE_COUNT;
```

```
    add_input();
    add_output();
    add_output();
}

ToraProcessQRY::~ToraProcessQRY()
{
    MOD_DEC_USE_COUNT;
}

const char * ToraProcessQRY::class_name() const
{
    return "ToraProcessQRY";
}

ToraProcessQRY *
ToraProcessQRY::clone() const
{
    return new ToraProcessQRY;
}

int ToraProcessQRY::initialize(ErrorHandler * errh)
{
    const Vector<Element *> &ev = router()->elements();
    _toraRtr = NULL;

    for (int i = 0; i < ev.size(); i++)
```

```

    if (ToraRouter *e = (ToraRouter *)ev[i]->cast("ToraRouter"))
        _toraRtr = e;

    if (!_toraRtr)
        return errh->error("could not find ToraRouter");

    return 0;
}

Packet * ToraProcessQRY::simple_action(Packet * p_in)
{
    WritablePacket *p = static_cast<WritablePacket *>(p_in);
    struct tora_qry *tq = reinterpret_cast<struct tora_qry *>(p->data()
+ADHOC_HEADER_ANNO(p));
    ToraNeighbor *tn;
    ToraInterface *ifc = NULL;
    struct timeval cur_time;

    // QRY packet regarding destination j received from neighbor k
    click_ip *ip = reinterpret_cast<click_ip *>(p->data());
    unsigned from = ip->ip_src.s_addr;
    ToraInterface *ifc1 = _toraRtr->acquire_interface(from);
    if (ifc1 != NULL) {
        // received a QRY packet sent by itself.
        CLICKCHATTER("ToraProcessQRY:%x received a QRY packet sent by
itself\n",from);
    }
}

```

```

    output(1).push(p);
    return 0;
}

ifc = _toraRtr->acquire_interface_by_index(PAINT_ANNO(p_in));
assert(ifc);

int src = ifc->getIPAddress();

unsigned dst_ip = tq->tora_cmn_hdr.tora_dst;

CLICKCHATTER("ToraProcessQRY:My addr:%x, Src:%x,
Dst:%x\n",src,from,dst_ip);

if (dst_ip <= 0 || dst_ip >IP_BROADCAST) {
    CLICKCHATTER("ToraProcessQRY:Invalid destination address
%x\n",dst_ip);

    output(1).push(p);
    return 0;
}

ToraDestination *info = _toraRtr->acquire_destination(dst_ip);
if (info == NULL) {
    CLICKCHATTER("ToraProcessQRY: Adding new destination
%x\n",dst_ip);

    info = _toraRtr->addDestination(dst_ip);
}

if (info->isRouteRequested() == TORA_RT_REQ ) {
    // Route requested already. Discard

    CLICKCHATTER("ToraProcessQRY:Route for %x requested
already\n",dst_ip);

    output(1).push(p);
    return 0;
}

```

```

    }
    ToraHeight *ih = info->getHeight();
    click_gettimeofday(&cur_time);
    if (ih->getRefLevel() == 0) {
        // II, A
        tn = info->findNeighbor(from);
        ToraNeighborInfo *tninfo= _toraRtr->findNeighborInfo(from);
        if (tninfo) {
            CLICKCHATTER("ToraProcessQRY:%x exists in neighbor info\n",from);
        }
        if (tn && (tn->getActivationTime()->tv_sec > info->getTimeUpdated()-
>tv_sec ||
            (tn->getActivationTime()->tv_sec == info->getTimeUpdated()->tv_sec &&
tn->getActivationTime()->tv_usec > info->getTimeUpdated()->tv_usec))) {
            CLICKCHATTER("ToraProcessQRY:II,A,1:Route found for %x. Sending
UPD\n",dst_ip);
            // II, A, 1
            info->setTimeUpdated(&cur_time);
            // Send UPD
            WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
            output(0).push(q);
        }
        else {
            // II, A, 2
            if (tn == 0) {
                CLICKCHATTER("ToraProcessQRY:II,A,2:%x:Neighbor:%x: not
found.Do nothing\n",dst_ip,from);
            }
        }
    }
}

```

```

    }
    else {
        CLICKCHATTER("ToraProcessQRY:II,A,2:%x:Do nothing\n",dst_ip);
    }
}
}
else {
    // II, B
    tn = info->findMinHeightNeighbor(0);
    if (tn) {
        ToraHeight *tnh = tn->getNeighborHeight();
        // II, B, 1
        info->updateHeight(tnh->getTau(),
            tnh->getOid(),
            tnh->getRefLevel(),
            tnh->getDelta()+1,
            src);
        info->setTimeUpdated(&cur_time);
        // Send UPD
        WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
        CLICKCHATTER("ToraProcessQRY:II,B,1:Route found for %x. Sending
UPD\n",dst_ip);
        output(0).push(q);
    }
    else {
        // II, B, 2
        info->setRouteRequested(TORA_RT_REQ);
    }
}
}

```

```

if (info->getNumActiveNgbrs() > 1) {
    // II, B, 2, a
    // Send QRY
    WritablePacket *q = _toraRtr->createQRYPacket(src,dst_ip);
    CLICKCHATTER("ToraProcessQRY:II,B,2,a:Route not found for %x.
Sending QRY\n",dst_ip);
    output(0).push(q);
}
else {
    // II, B, 2, b
    CLICKCHATTER("ToraProcessQRY:II,B,2,b:Route not found for
%x.Discarding QRY\n",dst_ip);
    output(1).push(p);
}
}
}
if (info)
    _toraRtr->release_destination(info);
return 0;
}

EXPORT_ELEMENT(ToraProcessQRY)

```

Appendix C

Source code for ToraProcessUPD element

3.1 Source code for ToraProcessUPD element

This appendix provides the source code for ToraProcessUPD element.

```
// ToraProcessUPD.cc: implementation of the ToraProcessUPD class.
// Implements Tora UPD packet processing as defined in section 4.7.6 of
// Temporally-Ordered Routing Algorithm (TORA) Version 1
//           Functional Specification
// Nandesh Kumar Palanisamy
////////////////////////////////////
#include <click/config.h>
#include <click/package.hh>
#include <click/element.hh>
#include <click/etheraddress.hh>
#include <click/ipaddress.hh>
#include <click/timer.hh>
#include <click/hashmap.hh>
#include <click/click_adhoc.hh>
#include <click/click_ether.h>
#include <click/confparse.hh>
```

```
#include <click/bitvector.hh>
#include <click/error.hh>
#include <click/glue.hh>
#include <click/click_adhoc.hh>
#include <click/click_ip.h>
#include <click/router.hh>
#include <click/packet_anno.hh>
#include <sys/queue.h>
```

```
#include "Tora.hh"
#include "ToraNeighbor.hh"
#include "ToraDestination.hh"
#include "ToraRouter.hh"
```

```
#include "ToraProcessUPD.hh"
```

```
#ifdef CLICK_SIM
using namespace CLICKSIM;
#endif
```

```
////////////////////////////////////
```

```
// Construction/Destruction
```

```
////////////////////////////////////
```

```
ToraProcessUPD::ToraProcessUPD()
{
    MOD_INC_USE_COUNT;
```

```
add_input();
add_output(); // Tora Output
add_output(); // Discard
}

ToraProcessUPD::~~ToraProcessUPD()
{
    MOD_DEC_USE_COUNT;
}

const char * ToraProcessUPD::class_name() const
{
    return "ToraProcessUPD";
}

ToraProcessUPD *
ToraProcessUPD::clone() const
{
    return new ToraProcessUPD;
}

int ToraProcessUPD::initialize(ErrorHandler * errh)
{
    const Vector<Element *> &ev = router()->elements();
    _toraRtr = NULL;

    for (int i = 0; i < ev.size(); i++)
```

```

    if (ToraRouter *e = (ToraRouter *)ev[i]->cast("ToraRouter"))
        _toraRtr = e;

    if (!_toraRtr)
        return errh->error("could not find ToraRouter");

    return 0;
}

Packet * ToraProcessUPD::simple_action(Packet * p_in)
{
    WritablePacket *p = static_cast<WritablePacket *>(p_in);
    struct tora_upd *tupd = reinterpret_cast<struct tora_upd *>(p->data()
+ADHOC_HEADER_ANNO(p));
    ToraNeighbor *tn;
    ToraInterface *ifc;
    struct timeval cur_time;
    int src;
    unsigned char ethernet[6];
    // copy ethernet address
    for (int i =ADHOC_ETHERNET_ANNO;i<ADHOC_ETHERNET_ANNO
+ 6; i++)
        ethernet[i-ADHOC_ETHERNET_ANNO] = p->user_anno_c(i);
    EtherAddress eth_address(ethernet);

    click_ip *ip = reinterpret_cast<click_ip *>(p->data());
    // Processing the UPDATE packet received from neighbor k regarding desti-

```

nation j.

```

unsigned dst_ip = tupd->tora_cmn_hdr.tora_dst;
unsigned from = ip->ip_src.s_addr;
ToraInterface *ifc1 = _toraRtr->acquire_interface(from);
if (ifc1) {
    CLICKCHATTER("ToraProcessUPD: %x Received an UPD packet sent by
itself.\n",from);
    return p_in;
}
ifc = _toraRtr->acquire_interface_by_index(PAINT_ANNO(p_in));
assert(ifc);
src = ifc->getIPAddress();
CLICKCHATTER("ToraProcessUPD:my
addr:%x,from:%x:dst:%x\n",src,from,dst_ip);
ToraDestination *info = _toraRtr->acquire_destination(dst_ip);
if (info == NULL) {
    CLICKCHATTER("ToraProcessUPD:Adding destination %x\n",dst_ip);
    info = _toraRtr->addDestination(dst_ip);
}

if (ntohl(tupd->tu_mode_seq) > (long)info->getModeSeq()) {
    info->setModeSeq(ntohl(tupd->tu_mode_seq));
}

ToraNeighborInfo *tninfo= _toraRtr->findNeighborInfo(from);
if (tninfo) {
    CLICKCHATTER("ToraProcessUPD:%x exists in neighbor info\n",from);
}

```

```

else {
    CLICKCHATTER("ToraProcessUPD:%x Does not exist in neighbor
info\n",from);
    tinfo = _toraRtr->addNeighborInfo(from , eth_address ,
PAINT_ANNO(p_in));
    }
    tn = info->findNeighbor(from);
    // Update height and link status for neighbor[j][k]
    if (tn == 0) {
        CLICKCHATTER("ToraProcessUPD: Packet is not from a known neigh-
bor\n");
        return 0;
    }
    info->updateNeighborHeight(tn,tupd);
    click_gettimeofday(&cur_time);
    ToraHeight *tnh = tn->getNeighborHeight();

    if ((info->isRouteRequested() == TORA_RT_REQ) &&
tnh->getRefLevel() == 0) {
        // I RT_REQ[j] && HT_NEIGH[j][k].r == 0

        // Set HT[J] = HT_NEIGH[j][k], Increment HEIGHT.delta
        info->updateHeight(tnh->getTau(),
            tnh->getOid(),
            tnh->getRefLevel(),
            tnh->getDelta()+1,
            src);
    }
}

```

```

// Unset routeRequested and timeupdated
info->setRouteRequested(TORA_RT_NOT_REQ);
info->setTimeUpdated(&cur_time);

// Send UPD
CLICKCHATTER("ToraProcessUPD:I:\n");
WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
output(0).push(q);
}
else if (info->getNumDownNgbrs() == 0) {
// II number of down stream neighbors == 0
ToraHeight *ih = info->getHeight();
if (info->getNumUpNgbrs() == 0) {
// II, A Number of upstream neighbors = 0
if (ih->isNull()) {
// II, A, 1, a HT[j] = NULL
CLICKCHATTER("ToraProcessUPD:II,A,1,a:\n");
if (info)
_toraRtr->release_destination(info);
return 0;
}
else {
// II, A, 1, b HT[j] != NULL
ih->setNull();
info->setTimeUpdated(&cur_time);
// Send UPD

```

```

WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
CLICKCHATTER("ToraProcessUPD:II,A,1,b:\n");
output(0).push(q);
}
}
else {
// Number of Downstream neighbors != 0
if (info->neighborCheckSameRef()) {
// II, A, 2, a
ToraNeighbor *tn;
if ((tn=info->findMinHeightNeighbor(0)) {
ToraHeight *tnh = tn->getNeighborHeight();
// II, A, 2, a, i
info->updateHeight(tnh->getTau(),
tnh->getOid(),
1,
0,
src);
info->setTimeUpdated(&cur_time);

//Send UPD
WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
CLICKCHATTER("ToraProcessUPD:II,A,2,a,i:\n");
output(0).push(q);
}
else {
// II, A, 2, a, ii

```

```

ToraNeighbor *tn;
if (tn = info->neighborHeightSameOid(src)) {
    // II, A, 2, a, ii, x
    struct timeval *temp_tau = ih->getTau();
    int temp_oid = ih->getOid();
    ih->setNull();
    info->setNumDownNgbrs(0);
    info->setNumUpNgbrs(0);

    for (tn = info->_ngbrList.lh_first; tn; tn->_link.le_next) {
        ToraHeight *tnh = tn->getNeighborHeight();
        if (tn->getIndex() == info->getIpAddress()) {
            tnh->setZero();
            tn->setLinkStatus(TORA_LINK_DOWN);
        }
        else {
            tnh->setNull();
            tn->setLinkStatus(TORA_LINK_UP);
        }
    }
    CLICKCHATTER("ToraProcessUPD:II,A,2,a,ii,x:\n");
    // Send CLR
    WritablePacket *q = _toraRtr->createCLRPacket(src, dst_ip,
temp_tau,temp_oid);
    output(0).push(q);
}
else {

```

```

// II, A, 2, a, ii, y
info->updateHeight(&cur_time,
    src,
    0,
    0,
    src);

info->setRouteRequested(TORA_RT_NOT_REQ);
info->setTimeUpdated(&cur_time);

// Send UPD
CLICKCHATTER("ToraProcessUPD:II,A,2,a,ii,y:\n");
WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);
output(0).push(q);
}
}
}
else {
// II, A, 2, b
ToraNeighbor *n = info->findMaxHeightNeighbor();
assert(n);
ToraHeight *nh = n->getNeighborHeight();
ToraNeighbor *m = info->findMinNonnullHeightNeighbor(nh);
assert(m);
ToraHeight *mh = m->getNeighborHeight();
info->updateHeight(mh->getTau(),
    mh->getOid(),
    mh->getRefLevel(),
    mh->getDelta()-1,

```

```
        src);

    info->setTimeUpdated(&cur_time);

    // Send UPD

    WritablePacket *q = _toraRtr->createUPDPacket(src,dst_ip);

    CLICKCHATTER("ToraProcessUPD:II,A,2,b:\n");

    output(0).push(q);

    }

    }

    }

else {

    // II, B

    CLICKCHATTER("ToraProcessUPD:II,B:\n");

    //output(1).push(p_in);

    }

if (info)

    _toraRtr->release_destination(info);

return 0;

}

EXPORT_ELEMENT(ToraProcessUPD)
```

Appendix D

Source code for ToraProcessCLR element

4.1 Source code for ToraProcessCLR element

This appendix provides the source code for ToraProcessCLR element.

```
// ToraProcessCLR.cc: implementation of the ToraProcessCLR class.
```

```
// Process Tora CLR packets as defined in section 4.7.7 of
```

```
// Temporally-Ordered Routing Algorithm (TORA) Version 1
```

```
//           Functional Specification
```

```
// Nandesh Kumar Palanisamy
```

```
////////////////////////////////////
```

```
#include <click/config.h>
```

```
#include <click/package.hh>
```

```
#include <click/element.hh>
```

```
#include <click/etheraddress.hh>
```

```
#include <click/ipaddress.hh>
```

```
#include <click/timer.hh>
```

```
#include <click/hashmap.hh>
```

```
#include <click/click_adhoc.hh>
```

```
#include <click/click_ether.h>
```

```
#include <click/confparse.hh>
```

```
#include <click/bitvector.hh>
#include <click/error.hh>
#include <click/glue.hh>
#include <click/click_adhoc.hh>
#include <click/click_ip.h>
#include <click/router.hh>
#include <sys/queue.h>
#include <click/packet_anno.hh>
```

```
#include "Tora.hh"
#include "ToraDestination.hh"
#include "ToraNeighbor.hh"
#include "ToraRouter.hh"
```

```
#include "ToraProcessCLR.hh"
```

```
#ifdef CLICK_SIM
using namespace CLICKSIM;
#endif
```

```
////////////////////////////////////
```

```
// Construction/Destruction
```

```
////////////////////////////////////
```

```
ToraProcessCLR::ToraProcessCLR()
{
    MOD_INC_USE_COUNT;
```

```
    add_input();
    add_output(); // Tora Output
    add_output(); // Discard
}

ToraProcessCLR::~ToraProcessCLR()
{
    MOD_DEC_USE_COUNT;
}

const char * ToraProcessCLR::class_name() const
{
    return "ToraProcessCLR";
}

ToraProcessCLR *
ToraProcessCLR::clone() const
{
    return new ToraProcessCLR;
}

int ToraProcessCLR::initialize(ErrorHandler * errh)
{
    const Vector<Element *> &ev = router()->elements();
    _toraRtr = NULL;
}
```

```

for (int i = 0; i < ev.size(); i++)
    if (ToraRouter *e = (ToraRouter *)ev[i]->cast("ToraRouter"))
        _toraRtr = e;

if (!_toraRtr)
    return errh->error("could not find ToraRouter");

return 0;
}

Packet * ToraProcessCLR::simple_action(Packet * p_in)
{
    WritablePacket *p = static_cast<WritablePacket *>(p_in);
    struct tora_clr *th = reinterpret_cast<struct tora_clr *>(p->data()
+ADHOC_HEADER_ANNO(p));
    ToraNeighbor *tn;
    ToraInterface *ifc = NULL;
    struct timeval cur_time;

    ToraDestination *info = _toraRtr->acquire_destination(th-
>tora_cmn_hdr.tora_dst );

    if (info == NULL) {
        output(1).push(p);
        return 0;
    }
}

```

```

// CLR packet regarding destination j received from neighbor k
click_ip *ip = reinterpret_cast<click_ip *>(p->data());
ifc = _toraRtr->acquire_interface_by_index(PAINT_ANNO(p_in));
assert(ifc);

int src = ifc->getIPAddress();
ToraHeight *ih = info->getHeight();
click_gettimeofday(&cur_time);
if (ih->getTau()->tv_sec == th->tc_tau.tv_sec &&
    ih->getTau()->tv_usec == th->tc_tau.tv_usec &&
    ih->getOid() == th->tc_oid &&
    ih->getRefLevel() == 1) {
// I
struct timeval *temp_tau = ih->getTau();
int temp_oid = ih->getOid();

ih->setNull();
info->setNumUpNgbrs(0);
info->setNumDownNgbrs(0);

for(tn= info->_ngbrList.lh_first; tn; tn = tn->_link.le_next) {
    ToraHeight *tnh = tn->getNeighborHeight();
    ToraInterface *ifc = _toraRtr->acquire_interface_by_index(tn->getIn-
dex());
    if (src == info->getIpAddress()) {
        tnh->setZero();
        tn->setLinkStatus(TORA_LINK_DOWN);
    }
}

```

```

else {
    tnh->setNull();
    tn->setLinkStatus(TORA_LINK_UP);
}
}
if (info->getNumActiveNgbrs() > 1) {
    // I, A
    // Send CLR
    WritablePacket *q = _toraRtr->createCLRPacket(src,th-
>tora_cmn_hdr.tora_dst , temp_tau, temp_oid);
    output(0).push(q);
}
else {
    // II, B
}
}
else {
    // II

tn = info->findNeighbor(ip->ip_src.s_addr);
if (tn == 0) {
    output(1).push(p);
    // Non neighbor.
    _toraRtr->release_destination(info);
    return 0;
}
ToraHeight *tnh = tn->getNeighborHeight();

```

```

tnh->setNull();
tn->setLinkStatus(TORA_LINK_UNDIRECTED);
for(tn= info->_nbrList.lh_first; tn; tn = tn->_link.le_next) {
    if (ih->getTau()->tv_sec == th->tc_tau.tv_sec &&
        ih->getTau()->tv_usec == th->tc_tau.tv_usec &&
        ih->getOid() == th->tc_oid &&
        ih->getRefLevel() == 1) {
        ToraHeight *tnh = tn->getNeighborHeight();
        tnh->setNull();
        tn->setLinkStatus(TORA_LINK_UNDIRECTED);
    }
}
if (info->getNumDownNbrs() == 0) {
    // II, A
    if (info->getNumUpNbrs() == 0) {
        // II, A, 1
        if (ih->isNull()) {
            // II, A, 1, a
        }
    }
    else {
        ih->setNull();
        info->setTimeUpdated(&cur_time);
        // Send UPD
        WritablePacket *q = _toraRtr->createUPDPacket(src,th-
>tor_a_cmn_hdr.tora_dst);
        output(0).push(q);
    }
}

```

```
    }  
    else {  
        info->updateHeight(&cur_time,  
                           src,  
                           0,  
                           0,  
                           src);  
  
        info->setRouteRequested(TORA_RT_NOT_REQ);  
        info->setTimeUpdated(&cur_time);  
  
        // Send UPD  
        WritablePacket *q = _toraRtr->createUPDPacket(src,th-  
>tora_cmn_hdr.tora_dst);  
        output(0).push(q);  
    }  
}  
else {  
    // II, B  
}  
}  
if (info)  
    _toraRtr->release_destination(info);  
return p;  
}
```

EXPORT_ELEMENT(ToraProcessCLR)

Appendix E

Source code for ToraOpt element

5.1 Source code for ToraOpt element

This appendix provides the source code for ToraOpt element which is implemented using a modified Tora OPT packet format. This element implements a neighbor discovery mechanism to address the problem of missing link layer mechanism providing such a capability.

```
// ToraOpt.cc - Broadcasts Opt messages every INTERVAL m.s
// Updates neighbor information when an OPT packet is received from a
// neighbor.
// Nandesh Kumar Palanisamy
////////////////////////////////////

#include <click/config.h>
#include <click/package.hh>
#include <click/element.hh>
#include <click/etheraddress.hh>
#include <click/ipaddress.hh>
#include <click/timer.hh>
#include <click/hashmap.hh>
```

```
#include <click/click_adhoc.hh>
#include <click/click_ether.h>
#include <click/confparse.hh>
#include <click/bitvector.hh>
#include <click/error.hh>
#include <click/glue.hh>
#include <click/click_adhoc.hh>
#include <click/click_ip.h>
#include <click/router.hh>
#include <click/packet_anno.hh>
#include <sys/queue.h>

#include "Tora.hh"
#include "ToraRouter.hh"
#include "ToraDestination.hh"
#include "ToraNeighborInfo.hh"
#include "ToraOpt.hh"

#ifdef CLICK_SIM
using namespace CLICKSIM;
#endif

ToraOpt::ToraOpt()
: _expire_timer(expire_hook, this)
{
    MOD_INC_USE_COUNT;
}
```

```
    add_output();
}

ToraOpt::~ToraOpt()
{
    MOD_DEC_USE_COUNT;
    uninitialized();
}

void
ToraOpt::notify_ninputs(int n)
{
    set_ninputs(n);
}

ToraOpt *
ToraOpt::clone() const
{
    return new ToraOpt;
}

int
ToraOpt::configure(const Vector<String> &conf, ErrorHandler *errh)
{
    _opt_interval_jiffies = 1000;

    // use address as source for rerr's
```

```

if (conf.size() > 1 || conf.size() < 0)
    return errh->error("expected INTERVAL");
else {
    if (cp_va_parse(conf, this, errh,
        cpUnsigned, "opt interval ms", &_opt_interval_jiffies, 0) < 0)
        return -1;
}

_opt_interval_jiffies = (CLICK_HZ * _opt_interval_jiffies) / 1000;

return 0;
}

int
ToraOpt::initialize(ErrorHandler *errh)
{
    _expire_timer.initialize(this);
    _expire_timer.schedule_after_ms(_opt_interval_jiffies);

    const Vector<Element *> &ev = router()->elements();
    _toraRtr = NULL;

    for (int i = 0; i < ev.size(); i++)
        if (ToraRouter *e = (ToraRouter *)ev[i]->cast("ToraRouter")) {
            _toraRtr = e;
            break;
        }
}

```

```
if (!_toraRtr)
    return errh->error("could not find routing table");

return 0;
}

void
ToraOpt::uninitialize()
{
    _expire_timer.unschedule();
}

void
ToraOpt::expire_hook(Timer *, void *thunk)
{
    ToraOpt *opt = (ToraOpt*)thunk;

    opt->expireOpt();
}

void
ToraOpt::expireOpt()
{
    sendOpt();

    _toraRtr->updateNeighborInfo (_opt_interval_jiffies,
TORA_ALLOWED_OPT_LOSS);
```

```

    _expire_timer.schedule_after_ms(_opt_interval_jiffies);
}

```

```

void

```

```

ToraOpt::sendOpt()

```

```

{
    ToraInterface *ifc = _toraRtr->_interface.lh_first;
    _toraRtr->_table_lock->acquire();

    for ( ; ifc; ifc = ifc->_iflink.le_next) {
        // make opt packet
        WritablePacket *p = _toraRtr->createOPTPacket(ifc);
        // send to output 0
        p->set_dst_ip_anno(IP_BROADCAST);
        output(0).push(p);
    }
    _toraRtr->_table_lock->release();
}

```

```

Packet *

```

```

ToraOpt::simple_action(Packet *p_in)

```

```

{
    struct timeval tval;
    ToraNeighborInfo *tninfo = NULL;
    WritablePacket *p = static_cast<WritablePacket *>(p_in);
    ToraDestination *destination = _toraRtr->_destinationList.lh_first;

```

```

    struct tora_modified_opt *tmo = reinterpret_cast<struct tora_modified_opt
*>(p->data() + ADHOC_HEADER_ANNO(p));

    click_ip *ip = reinterpret_cast<click_ip *>(p->data());

    unsigned from = ip->ip_src.s_addr;

    unsigned char ethernet[6];

    // copy ethernet address

    for (int i = ADHOC_ETHERNET_ANNO; i < ADHOC_ETHERNET_ANNO
+ 6; i++)

        ethernet[i-ADHOC_ETHERNET_ANNO] = p->user_anno_c(i);

    EtherAddress eth_address(ethernet);

    ToraInterface *ifc1 = _toraRtr->acquire_interface(from);

    if (ifc1 != NULL) {

        // received an OPT packet sent by itself.

        CLICKCHATTER("ToraOpt:%x received an OPT packet sent by
itself\n", from);

        return 0;

    }

    // copy ethernet address

    unsigned dst_ip = tmo->tora_cmh_hdr.tora_dst;

    tinfo = _toraRtr->findNeighborInfo(dst_ip);

    ToraInterface *ifc = _toraRtr-
>acquire_interface_by_index(PAINT_ANNO(p_in));

    assert(ifc);

    int src = ifc->getIPAddress();

    IPAddress srcip(src);

    IPAddress fromip(from);

```

```

IPAddress dstip(dst_ip);
if (dst_ip <= 0 || dst_ip > IP_BROADCAST) {
    CLICKCHATTER("ToraOpt:My Adrs:%s: Received an invalid OPT
packet\n",srcip.s().cc());
    return 0;
}
CLICKCHATTER("ToraOpt:My
Adrs:%s,From:%s,Dst:%s\n",srcip.s().cc(),fromip.s().cc(),dstip.s().cc());
if (tinfo == 0) {
    tinfo=_toraRtr->addNeighborInfo(dst_ip ,eth_address ,
PAINT_ANNNO(p_in));
    for ( ; destination; destination = destination->_dlink.le_next) {
        WritablePacket *q;
        if (destination->isRouteRequested() == TORA_RT_REQ) {
            CLICKCHATTER("ToraOpt:My Adrs:%s:Sending QRY\n",
srcip.s().cc());
            q = _toraRtr->createQRYPacket(src,destination->getIpAddress());
        }
        else {
            CLICKCHATTER("ToraOpt:My Adrs:%s:Sending UPD\n",
srcip.s().cc());
            q = _toraRtr->createUPDPacket(src,destination->getIpAddress());
        }
        output(0).push(q);
    }
    click_gettimeofday(&tval);
    tinfo->setLastOptReceiptTime(&tval);

```

```
    }  
    else {  
        click_gettimeofday(&tval);  
        tinfo->setLastOptReceiptTime(&tval);  
    }  
    return 0;  
}
```

```
EXPORT_ELEMENT(ToraOpt)
```