ENSC 835: HIGH-PERFORMANCE NETWORKS

Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Network

Spring 2008

Final Project Report

Instructor:

Dr. Ljiljana Trajković

Student:

Chih-Hao Howard Chang (20007-2192)

howardc@sfu.ca

www.sfu.ca/~howardc/ensc835/mcmi_project.html

School of Engineering Science

Simon Fraser University, Burnaby BC

Canada V5A 1S6

Abstract

The IEEE 802.11 standard defines multiple non-overlapping channels at the physical layer in the 2.4 GHz and 5 GHz spectrums. However, most ad-hoc wireless networks today are configured to operate under a single channel in order to ensure connectivity of all their nodes. Hence, the aggregate bandwidth provided by the radio spectrums is not fully-utilized. In order to meet the high throughput demand, it is essential to use all available spectrums. Several past research proposals have exploited multiple channels and multiple interfaces to increase the network capacity, by having multiple simultaneous transmissions without interference.

In this project, we implement and evaluate the technique of *interface switching* to utilize multiple channels and multiple interfaces using the Network Simulator (ns-2). This includes incorporating the multiple channel and multiple interface supports to the core of ns-2, which are not available in the current version of the simulator. The simulation results demonstrate the effectiveness of the approach in improving the network capacity.

Table of Contents

List of Figures

1. Introduction

In recent years, wireless local area networks (WLAN) have become popular, as they provide us the mobility to move around within a broad coverage area and still be connected with each other. The increasing popularity of WLANs is primarily due to their convenience, cost efficiency, and ease of integration with other networks and network components. More electronic and computer devices sold to consumers nowadays come pre-equipped with a wireless network interface card. This trend, along with reducing wireless hardware costs, has accelerated the use of WLANs, increasing the throughput demand.

IEEE 802.11 is a widely used set of standards for WLANs. Endpoint devices (or hosts) designed for this wireless technology can be operated in two modes. In infrastructure mode, hosts are connected via a base station, also known as an access point, which provides network services such as address assignment and routing. Another mode is ad-hoc, whereas wireless hosts have no such infrastructure with which to connect; the hosts must provide for these network services. Figure 1 illustrates the fundamental differences between these two types of wireless networks.

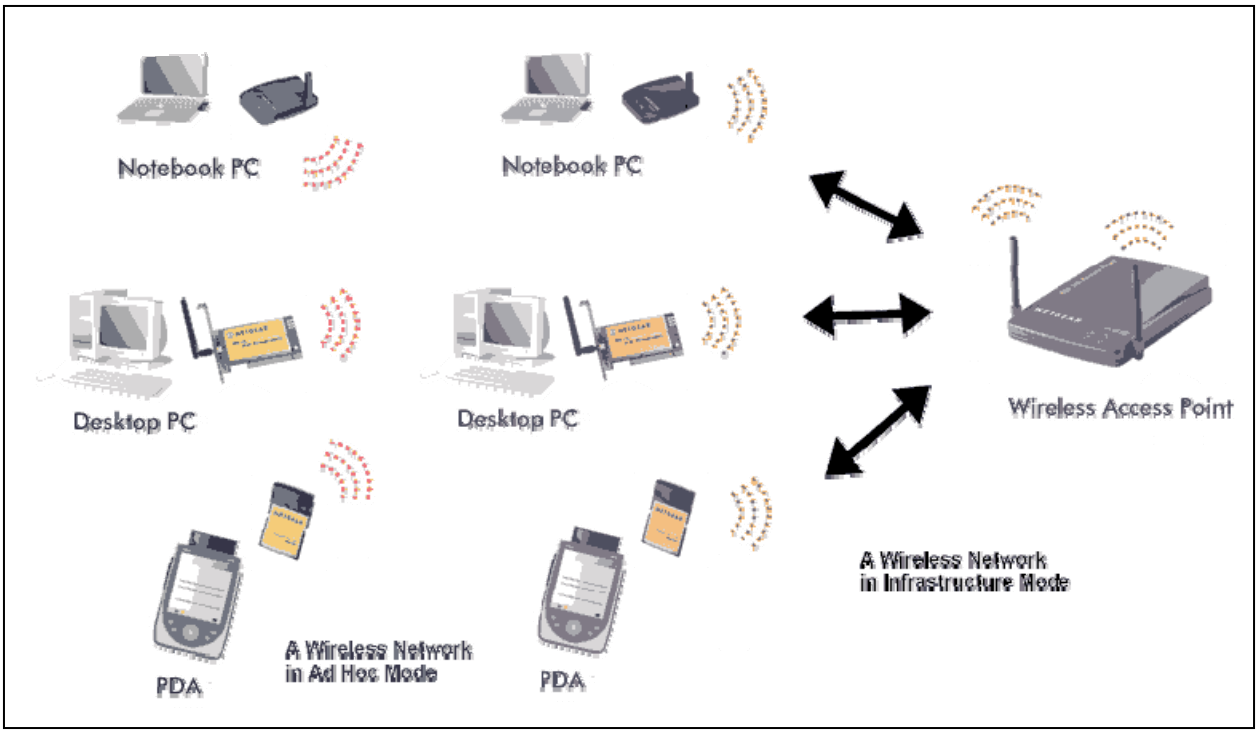

Figure 1: Comparison between wireless networks in ad-hoc and infrastructure modes[1]

[1] From: "Selecting Between Infrastructure and Ad Hoc Wireless Modes," http://kbserver.netgear.com/kb_web_files.

Multiple channels[2] have been utilized in infrastructure-based wireless networks by assigning distinctive channels to adjacent access points. However, typical ad-hoc wireless networks are configured to use only a single channel to ensure connectivity of all their nodes; thus, the aggregate bandwidth provided by the available radio spectrums is not fully-utilized. Such a dilemma has inspired many researchers to propose the exploitation of multiple channels and multiple interfaces to increase the network capacity. Many IEEE 802.11 interfaces[3] can be switched from one channel to another, allowing an interface to access multiple channels.

1.1. Project Scope

In this project, we start with extending the Network Simulator (ns-2) to support multiple channels and multiple interfaces by referring to the guideline written by R. A. Calvo and J. P. Campo in [1]. We then explore the use of multiple channels and multiple interfaces in ad-hoc wireless networks by implementing the interface assignment strategy proposed by P. Kyasanur and N. H. Vaidya in [2] using ns-2. Two key benefits of the approach in this paper are that the number of available interfaces can be less than the number of channels since equipping a node with one interface per channel is expensive, and that it can be implemented over existing 802.11 devices without any hardware modifications. Finally, we simulate a simple multi-channel multi-interface ad-hoc wireless network using the modified ns-2 to demonstrate the effectiveness of *interface switching* and the improvement in network throughput.

[2] A channel is a wireless spectrum with a specified bandwidth.

[3] An interface is a network interface card equipped with a half-duplex radio transceiver.

2. Wireless Ad-hoc Network

2.1. Overview

Ad-hoc networks are infrastructure-less. The interconnected nodes coordinate the transmissions with other nodes and they may have to relay messages among several other nodes in order to reach a destination. In other words, an ad-hoc network is a self-configuring network of wireless links connecting mobile nodes, where these nodes may be routers and/or hosts as shown in Figure 2. The decision of which nodes forward data is made dynamically based on the network connectivity. This mechanism is contrary to wired networks in which routers perform the task of forwarding. Ad-hoc networks are also in contrast to the infrastructure-based wireless networks, which require an access point to manage the connection among other nodes.

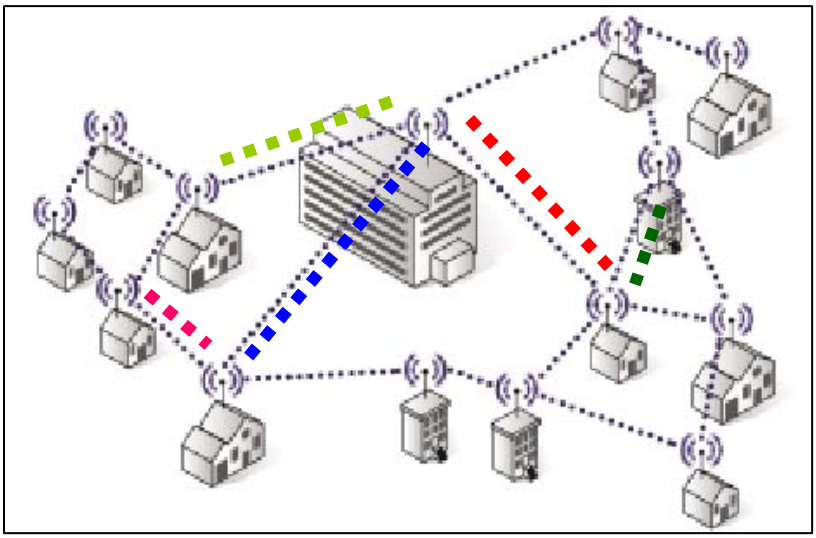

Figure 2: A wireless ad-hoc network

Nodes in a wireless ad-hoc network relies on each other to establish a connection, as there is no central access point which co-ordinates address assignment and routing among the nodes. Thus, a packet can travel from a source to a destination indirectly through some intermediate nodes in the neighbourhood. To maintain a persistent packet forwarding path, a channel is commonly shared with all nodes in the ad-hoc network, due to the fact that two adjacent nodes can only communicate if they are on the same channel. Unfortunately, utilizing one channel among all nodes may eventually lead to traffic congestion as the number of nodes in the network increases.

2.2. Multiple Channels and Multiple Interfaces

IEEE 802.11 offers several non-overlapping channels in the 2.4 GHz and 5 GHz spectrums that are separated in frequency. For example, the 802.11b/g standards operating in the 2.4 GHz bands have 3 non-overlapping channels (1, 6, 11) for use in Canada and the United States. On the other hand, 802.11a operating in 5 GHz offers 12 non-overlapping channels. The large number of available channels in 802.11a is an example in which equipping a node with one interface per channel may not be feasible.

In infrastructure-based wireless networks, multiple channels have already been utilized by assigning different channels to adjacent access points to minimize interference. The reason that most wireless ad-hoc networks today are configured to use a single channel is to avoid the need for node co-ordination. Due to the increasing throughput demand, the idea of exploiting multiple channels is appealing in ad-hoc wireless networks, thus motivating many research communities to develop new algorithms for multi-channel operation.

When multiple wireless channels are available, having more than one interface on a node allows two different nodes communicating in parallel on different channels. An interface has a capability to dynamically switch to different channels over time. Two adjacent nodes can communicate with each other if they have at least one interface on a common channel. In a wireless ad-hoc network consisting of multiple hops, if each node only uses a single interface, packets may be delayed at some hops if their next hops are not on the same channel. The reason is that node A needs to wait after node B's interface has switched to listen to the same channel as node A is using before transmitting a packet to node B, as shown in the figure below.

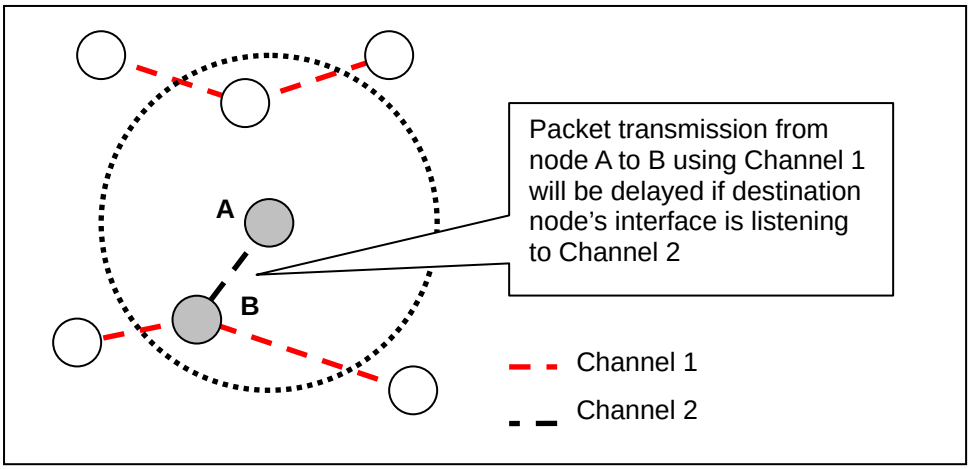

Figure 3: In the presence of only 1 interface, destination node has to listen to the same channel

In other words, the end-to-end delay increases if the traversed hops are on different channels. One might be tempted to solve this problem by assigning each node to be on the same channel. Unfortunately, transmissions on consecutive nodes could interfere with each other, therefore degrading the transmission capacity, as shown in Figure 4.

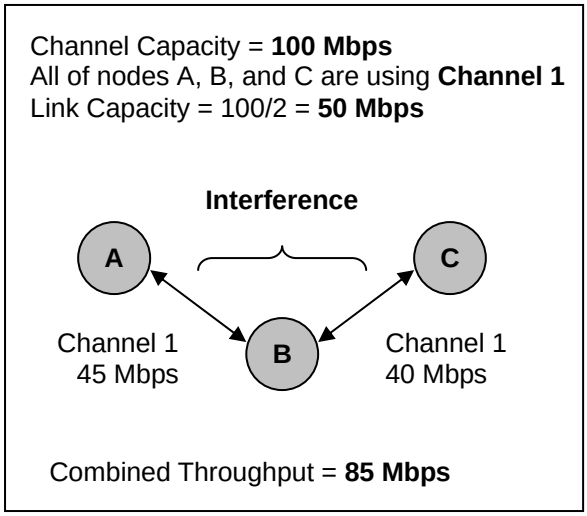

Figure 4: Transmission capacity is degraded when adjacent nodes are on the same channel

Using multiple interfaces, a node is able to transmit and receive data simultaneously. Such an action cannot be carried out when only a single wireless interface is used, as it is half-duplex. When two interfaces and multiple channels are available, we can have one interface transmitting data on one channel while the other interface is receiving data on another channel. The maximum achievable throughput can in turn be nearly doubled as illustrated in the following diagram.

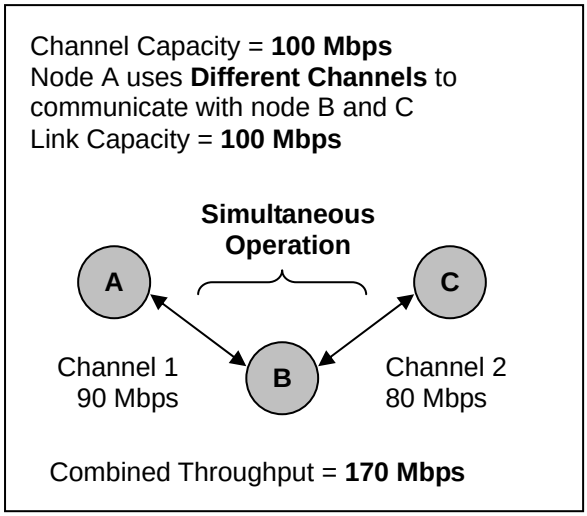

Figure 5: Transmission capacity is doubled when adjacent nodes are on different channels

3. Related Work

Many researchers have proposed the use of multiple channels and multiple interfaces for ad-hoc wireless networks. However, we have selected P. Kyasanur's and N. H. Vaidya's interface assignment scheme in [2] for our implementation, as this approach is more flexible and versatile among others. Below we present a brief comparison between various related work and [2].

For instance, A. Raniwala, K. Gopalan, and T. Chiueh propose the centralized channel assignment and routing solution in [3]. Unlike this solution that is designed for use in static networks where traffic is directed towards specific gateway nodes, the approach in [2] allows any node to communicate with any other. Additionally, the approach in [3] expects stationary nodes and traffic load on every link, since this information is used to assign interfaces and compute routes. Oppositely, the solution in [2] does not need this information and thus is more suitable for ad-hoc networks involving mobile nodes.

Next, S. Wu, C. Lin, Y. Tseng, and J. Sheu propose a MAC layer solution in [4] that requires two interfaces: one for control, therefore assigned to a common channel; the other for data exchange, thus is switched among the remaining channels. Also, R. Maheshwari, H. Gupta, and S. R. Das propose new MAC protocols for multi-channel operation in wireless ad-hoc and mesh networks in [5]. The aforementioned two proposals all require changes to the existing IEEE 802.11 standard. On the contrary, the approach in [2] can be implemented using existing IEEE 802.11 wireless network interface cards.

To address this compatibility issue, P. Bahl, R. Chandra, and J. Dunagan propose Slotted Seeded Channel Hopping (SSCH) in [6], which is a link-layer solution that uses a single interface and can run over unmodified IEEE 802.11. However, SSCH does not support multiple interfaces.

Finally, paper [7] proposes a new routing metric called Weighted Cumulative Expected Transmission Time (WCETT), for multi-channel ad-hoc networks. WCETT ensures channel diverse routes are selected by assuming the number of interfaces per node is equal to the number of channels used by the network. In contrast, the *interface switching* technique proposed in [2] can allow the number of interfaces to be smaller than the number of available channels, while still manages to utilize all the channels.

4. Multi-Channel Multi-Interface Solution

In paper [2], P. Kyasanur and N. H. Vaidya have proposed the following designs for routing and interface assignment in multi-channel multi-interface ad-hoc wireless networks:

1. A multi-interface solution for exploiting multiple channels that can be implemented on existing IEEE 802.11 hardware.
2. An interface assignment strategy using the technique of *interface switching*, that simplifies coordination among nodes while utilizing multiple available channels.
3. A routing protocol, namely the Multiple-Channel Routing (MCR), that selects routes with the highest throughput by accounting for channel diversity and *interface switching* cost.

For the scope of this project, we have limited our implementation in ns-2 to only realize and evaluate the first two goals. Point 1 suggests that existing behaviours of the data-link and physical layers in ns-2 constituting the IEEE 802.11 standard should not be altered. In the next section, we will show how our modifications to ns-2's *MobileNode* to add the support for multiple interfaces preserve the legacy operations of IEEE 802.11 interfaces. We will also incorporate the interface assignment algorithm from point 2; however, since point 3 will not be implemented, point 2 will be added to the existing ad-hoc on-demand distance vector (AODV) routing agent in ns-2 as an extended feature. One shortcoming with AODV is that it selects the shortest-path route (the route that has the least number of hops through other nodes), which may not utilize all the available channels [2].

4.1. Interface Switching

In order to maximize the utilization of all available channels, interfaces have to be switched from one channel to another. A switching protocol is required to assign interfaces to specific channels, ensuring that the neighbour nodes of node X can communicate with itself on-demand. In a sense, all neighbour nodes of node X must at least have knowledge about the channel being used by one channel of X. We describe the proposed algorithm from paper [2] in the following sub-sections.

4.1.1. Fixed and Switchable Interfaces

In a multi-channel multi-interface ad-hoc wireless network, *M* interfaces available at each node are divided into the two groups:

1. **Fixed Interface**: Some *K* of the *M* interfaces at each node are assigned for long intervals of time to some *K* channels. The corresponding channels are regarded as fixed channels. Fixed interfaces are used to receive data and are to be switched based on the number of nodes using a channel.

2. **Switchable Interface**: The remaining *M-K* interfaces are dynamically assigned to any of the remaining *M-K* channels over short time scales based on data traffic. The corresponding channels are designated as switchable channels. A switchable interface enables node X to transmit to node Y in its neighbourhood by switching to the fixed channel used by Y.

The value of *M* and *K* can vary in different nodes; however, for simplicity, let's consider a simple scenario in which *M* = 2 and *K* = 1 for all nodes (one fixed and one switchable) and there are three non-overlapping channels. In the following figure, nodes A, B and C have the fixed interfaces on channels 1, 2, and 3 and the switchable interfaces as shown initially. Suppose that the routing path is A → B → C such that node A wants to send a packet to node C. In the first step, node A switches its switchable interface from 3 to 2 before transmitting the packet to node B since the fixed interface (channel) of node B is 2. Similarly, in the next step, node B switches its switchable interface from 1 to 3 to forward the packet to node C as the fixed interface (channel) of node C is 3. Once the interface has been appropriately set up during a flow initiation, subsequent flow of packets will not need to switch the interfaces for a long time.

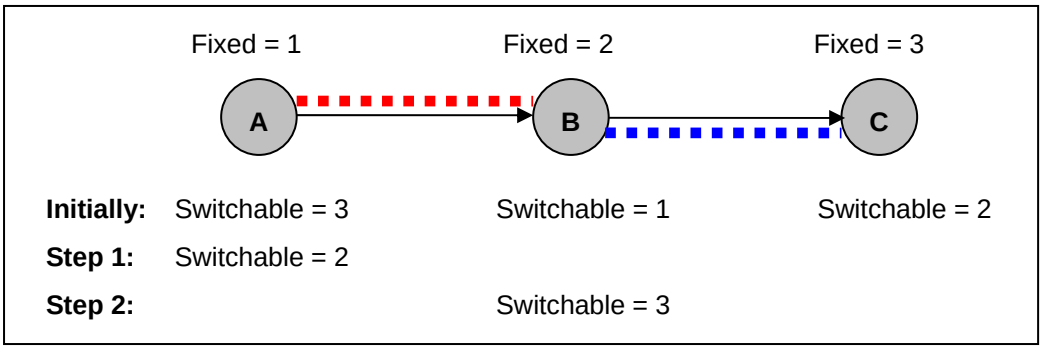

Figure 6: Interface switching with 3 channels and 2 interfaces [2]

In general, when node X has to communicate with node Y over channel *c*, the switchable interface of X is switched to channel *c* for communicating with Y. If the fixed channel being used by X is already *c*, then no switching will be required. In other words, the transmission between two nodes is receiver-based. The sender adapts to the receiver by changing its switchable interface to the receiver's fixed interface, which is the channel used by the receiver.

4.1.2. Fixed Interface Assignment

As illustrated in Figure 6, nodes A, B, and C use channels 1, 2 and 3, respectively; all transmissions directed to A, B, and C will be on channel 1, 2, and 3, respectively. The purposes of the fixed interface assignment are to choose the channel to be assigned to a fixed interface and to inform the neighbour nodes about the channel being used by the fixed interface. First of all, each node maintains two tables:

- *NeighbourTable (NT)*:
  - Contains the fixed channels being used by its neighbours.
- *ChannelUsageList (CUL)*:
  - Keeps the number of nodes using each channel as their fixed channel, but it only tracks the nodes present within its communication range.

Below we present the localized assignment algorithm proposed in [2] along with a visual example of the approach for an ad-hoc wireless network with 3 nodes, each has 3 channels and 2 interfaces (one fixed and switchable). Please note that the node's table we present in each of the following figures is actually an implicit combination of *NeighbourTable* and *ChannelUsageList*.

1. Initially, each node chooses a random channel for its fixed interface.

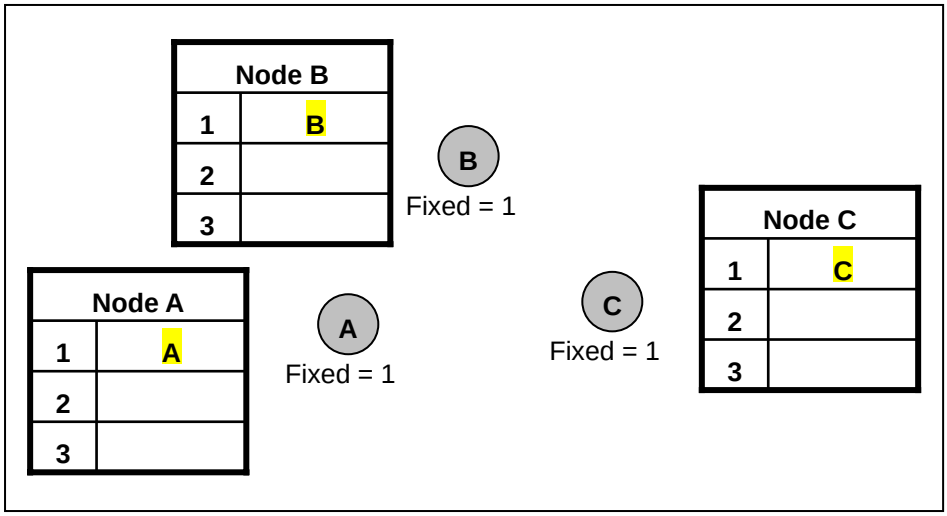

Figure 7: Each node chooses a random channel for its fixed interface

2. Periodically, each node broadcasts a Hello or Route Discovery packet on every channel, which contains the fixed channel being used by the node and its current *NeighbourTable*.
   a. Many routing protocols such as AODV broadcast a Hello or Route Discovery packet to offer connectivity information.

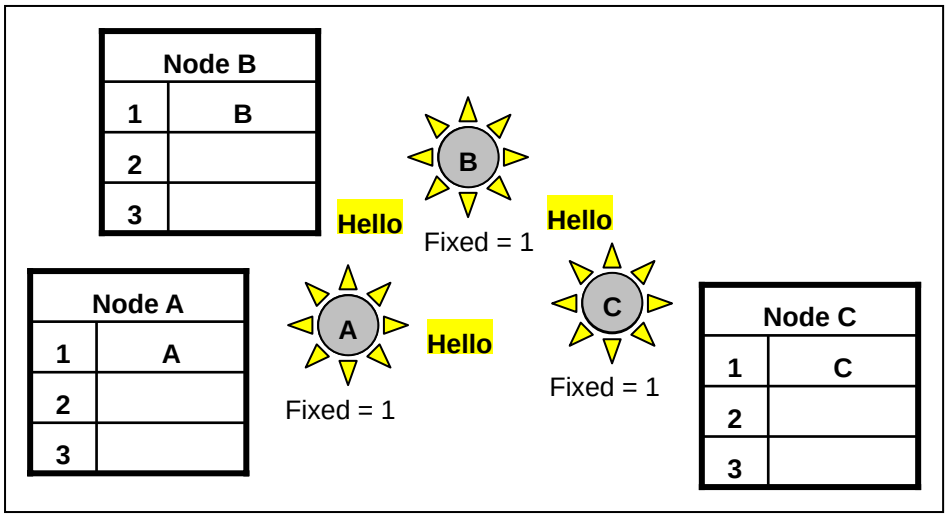

Figure 8: Each node periodically broadcasts on every channel its current fixed channel

3. When a node receives a Hello or Route Discovery packet from a neighbour, it updates its:
    a. *NeighbourTable* with the fixed channel of that neighbour.
    b. *ChannelUsageList* using the *NeighbourTable* of its neighbour so *ChannelUsageList* will contain two-hop usage information.

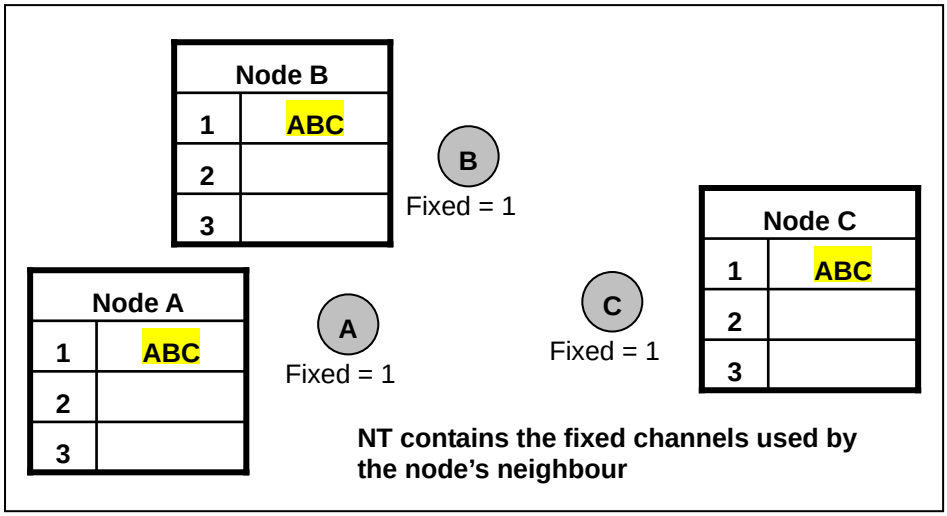

Figure 9: On receiving a Hello packet, a node updates its NeighborTable and ChannelUsageList

4. Each node periodically consults its *ChannelUsageList*. The period chosen is large relatively to packet transmission time.
    a. If the number of other nodes using the same fixed channel as itself is large, then a node with some probability *p* changes its fixed channel to a less used channel.
        i. Then the node transmits a Hello or Route Discovery packet to inform its neighbors of the new fixed channel it is using.

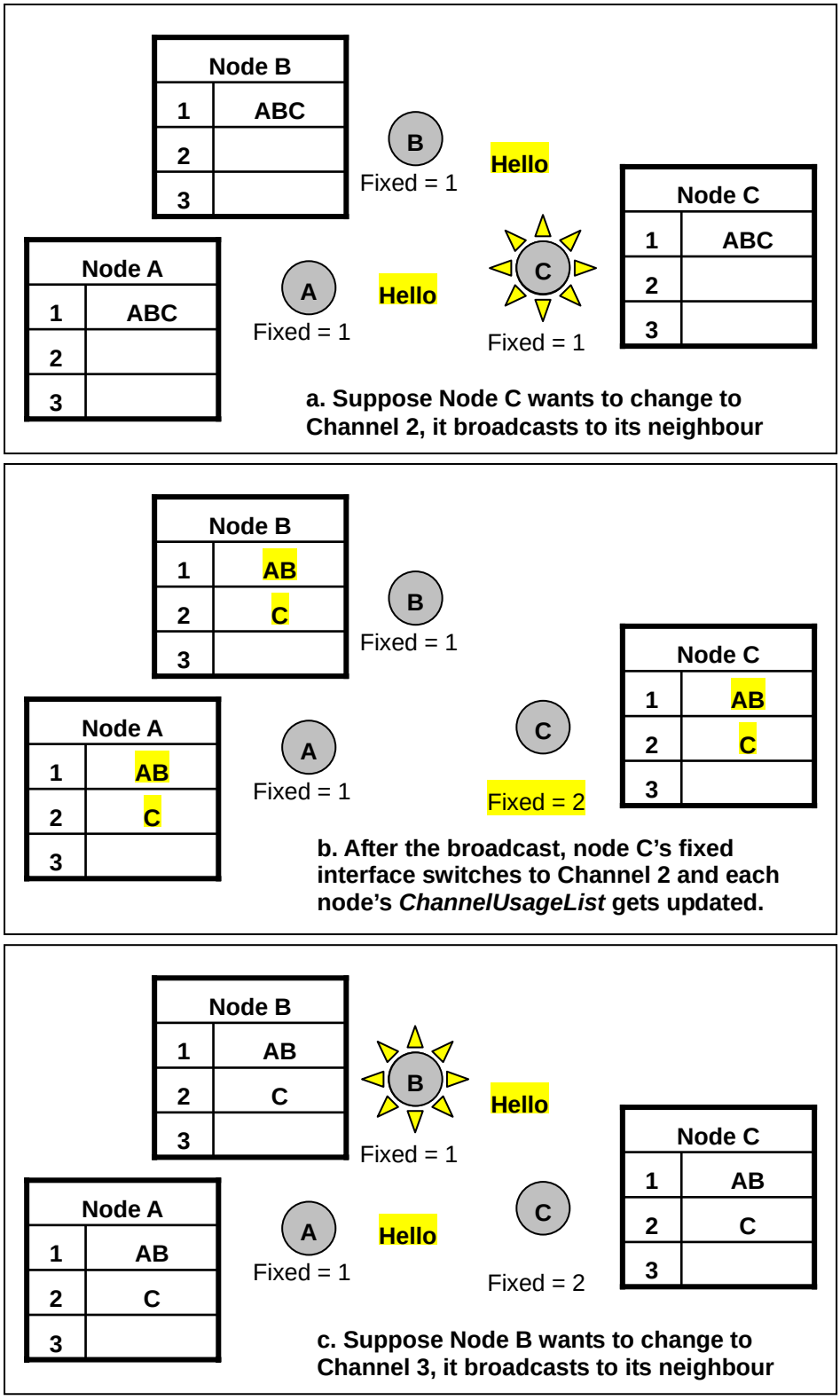

Figure 10: Each node periodically consults its ChannelUsageList

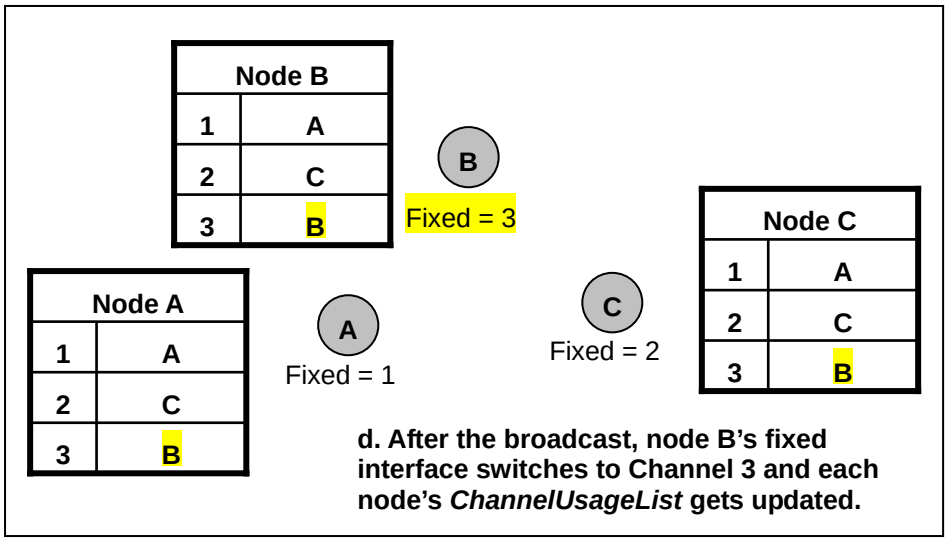

Figure 9: Each node periodically consults its ChannelUsageList (continued)

According to paper [2], the period of consulting the *ChannelUsageList* chosen is large relatively to packet transmission time. Also, the probability, *p*, of switching the fixed interface is set to **0.4**. The enforcement of the switching probability and the decision of switching based on the number of nodes using as channel avoid frequent change of fixed channels. The only scenario that switching will be required is when the network topology changes significantly.

4.1.3. Switchable Interface Assignment

As mentioned earlier, switchable interfaces are used to transmit data from node X whenever the fixed channel of the destination is different from the fixed channel of X. Each node maintains a separate packet queue for each channel as depicted in the figure below:

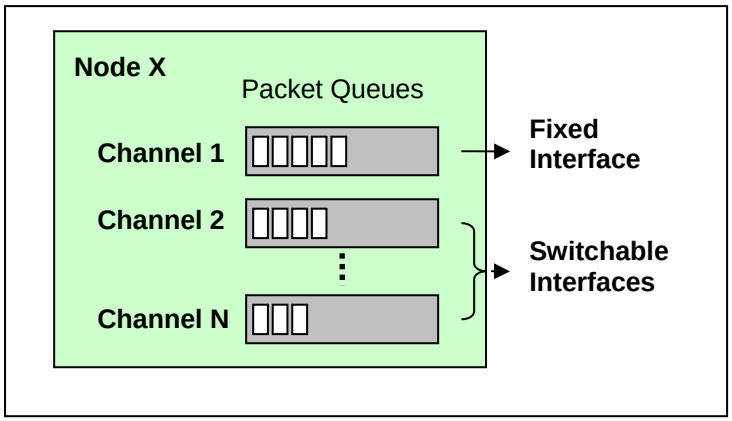

Figure 11: Packet queue maintained by each node for each channel [2]

Below, we present the rules for assigning the switchable interface of a node proposed in [2]:

1. When an unicast packet is received at the ling layer for transmission,
    a. The node looks up the fixed channel of the destination of the packet in the *NeighbourTable*. If the sender has the same fixed channel as the receiver, enqueue the packet to the fixed channel. Otherwise, enqueue to the switchable channel.
    b. For broadcasting the packet, the node copies it to each channel's queue. The packet will be sent out when that channel is scheduled for transmission.
2. The switchable interface changes channels if there are packets queued for another channel.

5. Adding Multi-Channel Multi-Interface Support to ns-2

The latest version (at the time of the project) of the ns-2 simulator is 2.32; it does not support wireless network simulation involving multiple channels and multiple interfaces. As a result, before attempting to evaluate the exploitation of multiple channels and multiple interfaces in wireless ad-hoc networks, we must add this capability to the current framework of ns-2.

5.1. Overview of ns-2

ns-2 is an open-source discrete event network simulator that supports a variety of different network protocols, producing simulation results for both wired and wireless networks. In this project, we choose ns-2 to implement a multi-channel multi-interface ad-hoc wireless network, primarily due to the flexible extensibility of the simulator. ns-2 is developed in C++ but it also provides a simulation interface through an object-oriented tool command language (OTCL).

When performing a network simulation, users first write an OTCL script to describe the relevant network topology, and to define traffic sources and when to start and stop transmitting packets through an event scheduler. ns-2 has an OTCL interpreter that translates the event scheduler and the network component OTCL objects and member functions into their corresponding C++ counterparts. Such a mechanism provides a linkage between the OTCL and C++ realms, making the control functions and the configurable variables interoperable. Next, ns-2's main program simulates the topology with user-specified parameters. Eventually, the simulation is finished; ns-2 generates an output trace file that contains detailed simulation data. The data can be either post-processed to create simulation graphs or input to a graphical simulation tool called Network Animator (nam) for later viewing. The architecture of ns-2 is shown in the figure below.

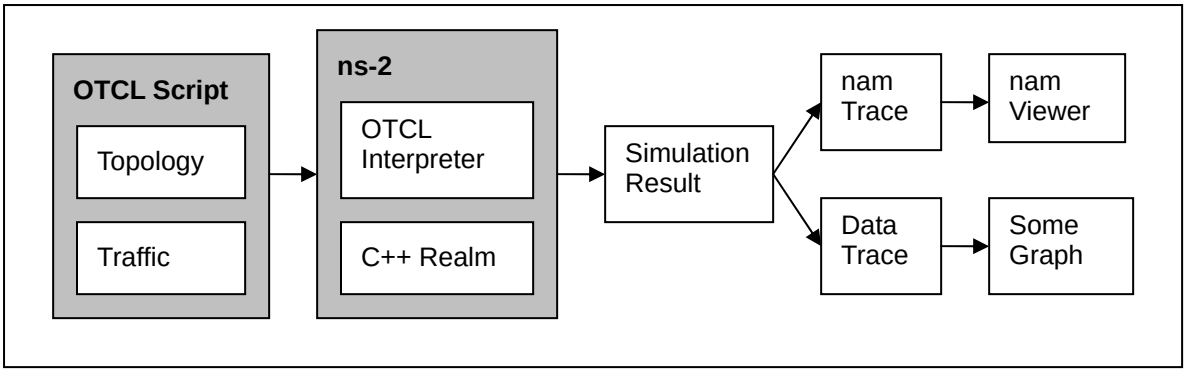

Figure 12: Overall Architecture of ns-2

5.2. Multiple Interface Node Model

The reason that the guideline in [1] is chosen for our implementation is that the multiple interface node model it proposes offer the following advantages which are suitable for the approach in [2]:

- The number of channels is modifiable.
- The number of interfaces per node is variable; it needs not to be the same for all nodes.
- Each node can be connected to different number of channels.
- Legacy operations of ns-2 are still preserved. This backward compatibility is particularly important as we will use the modified simulator to simulate wireless ad-hoc networks without the exploitation of multiple channels and multiple interfaces.

As mentioned by the guideline, extending ns-2 to support multiple interfaces requires modifying its existing *MobileNode* model whose architecture is depicted in Figure 13. *MobileNode* is an extension of the *Node* object in ns-2 with additional functionalities such as mobility, ability to transmit and receive on a channel that allows it to be used in mobile and wireless network simulations. Some main component objects that a *MobileNode* object is consisted of are:

- **Routing Agent**
  - Routes data packets to the next-hop node on the *MobileNode*'s behalf. Currently, ns-2 supports four ad-hoc routing protocols as follows:
    - Ad-hoc On-demand Distance Vector (AODV)
    - Destination-Sequenced Distance-Vector Routing (DSDV)
    - Dynamic Source Routing (DSR)
    - Temporally Ordered Routing Algorithm (TORA)
- **Link Layer**
  - Sets the MAC destination address in the MAC packet header by finding the IP address of the next-hop-node directed by the routing agent and resolving this address into the corresponding MAC address by ARP.
- **Address Resolution Protocol (ARP)**
  - Receives queries from the link layer to resolve the associated hardware address by referring to an ARP table. If the address can be found locally, it is written to the MAC header of the packet; otherwise, an ARP query is broadcasted. Once the MAC address is retrieved, the packet in inserted into the interface queue.
- **Interface Queue**
  - Gives priority to the stored routing protocol packets.
- **Media Access Control (MAC)**
  - Processes data packets received from or to be sent to the link layer. When

sending, it adds the MAC header and transmits the packet onto the channel. Or, it asynchronously receives packets from the classifier of the physical layer.

- **Network Interface**
  - Serves as a hardware interface for *MobileNode* to access the channel.
- **Channel**
  - Simulates the effect of the real wireless channel on the transmitted signal.

Each of the objects emulates a real-life entity in the physical and data-link layers of a wireless network. Together, these components allow simulation of WLANs or multi-hop ad-hoc networks.

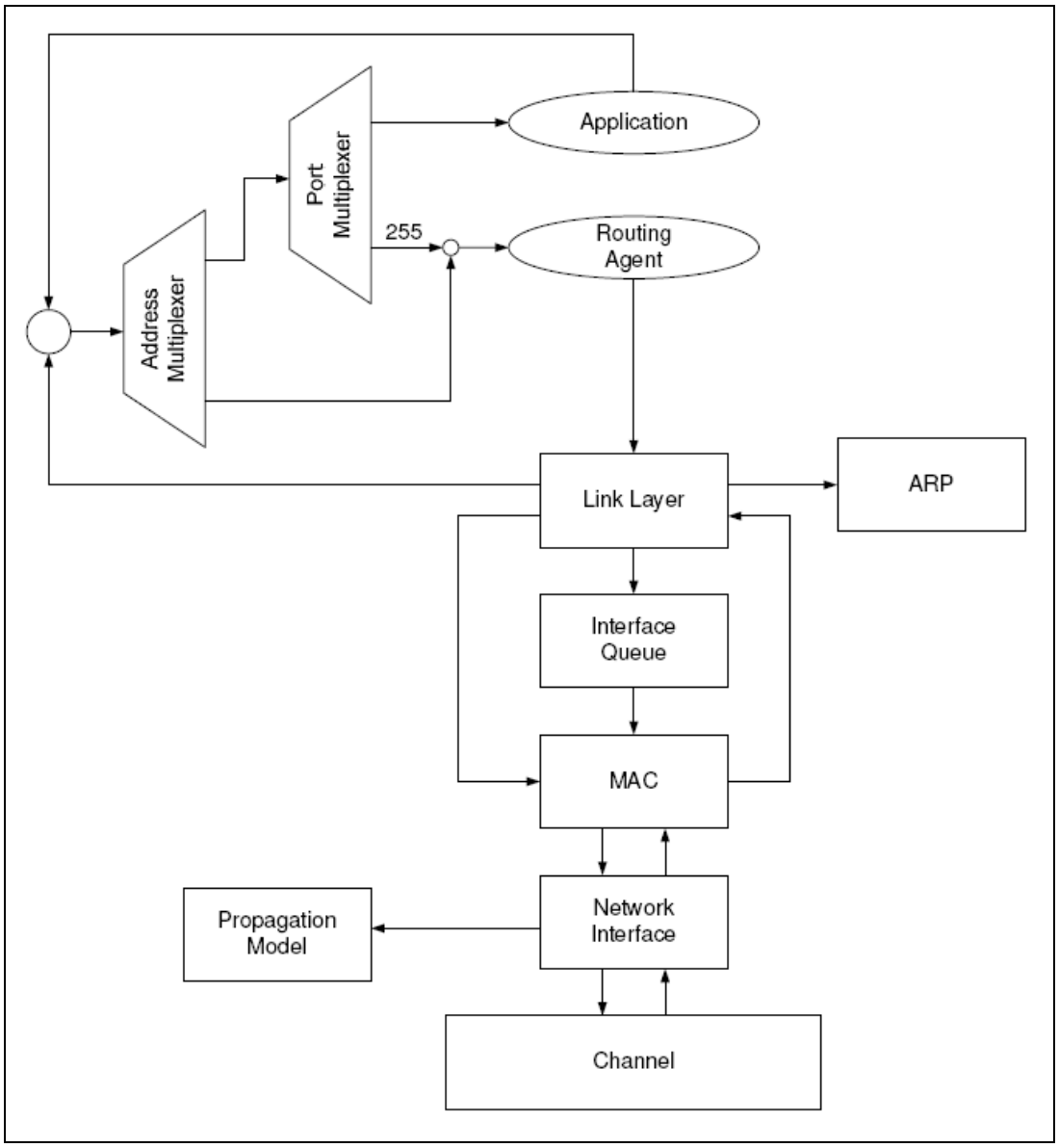

Figure 13: Original MobileNode Architecture in ns-2 [1]

Figure 14 presents a high-level architecture of the modified *MobileNode* object proposed in [1]. Each node can have as many instances of the link layer, ARP, interface queue, MAC, network interface and channel entities as the number of interfaces it has. One can imagine that each instance actually represents a wireless network interface. Thus, this design scheme emulates the fact that our multi-channel multi-interface ad-hoc network implementation will not require any modification to existing IEEE 802.11 hardware.

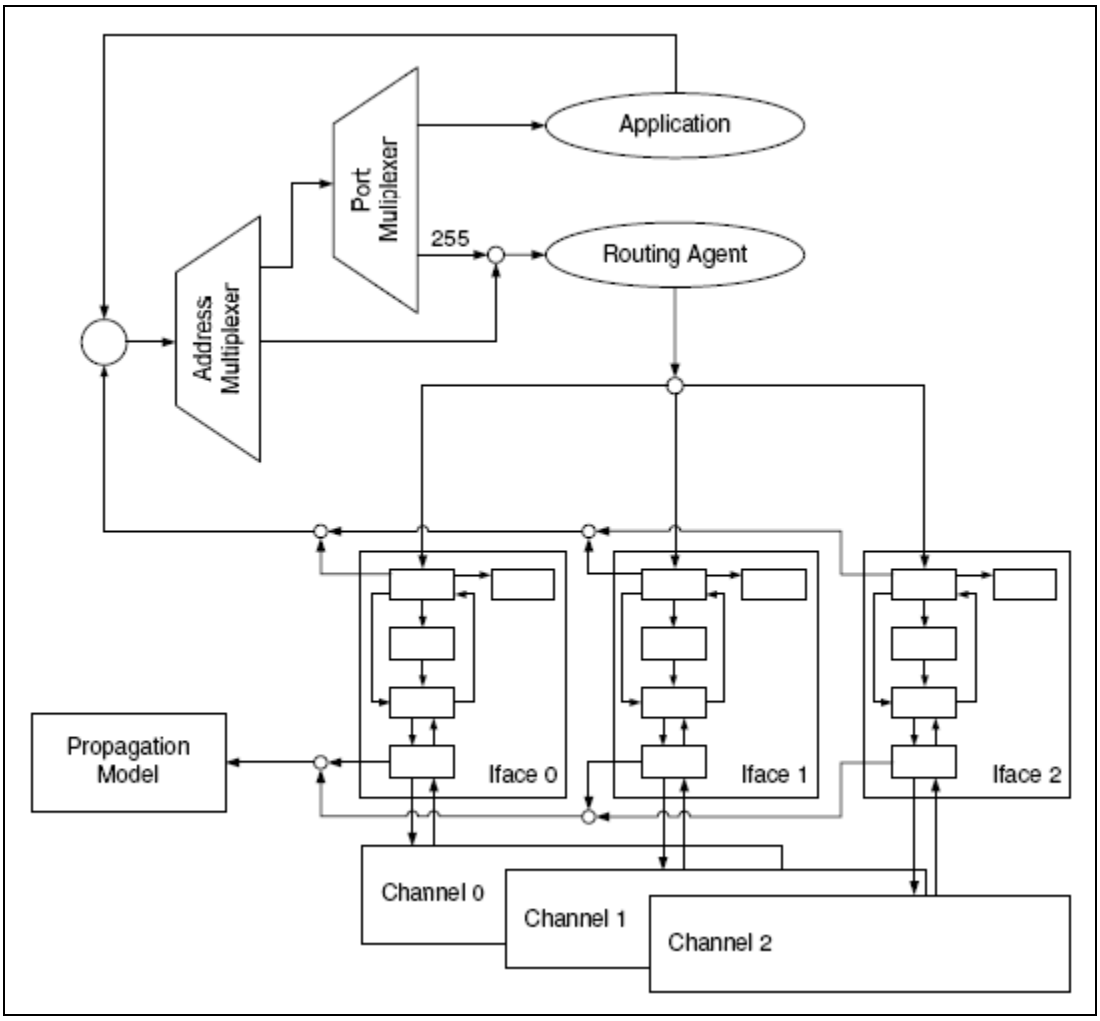

Figure 14: Modified MobileNode architecture, with the support for multiple interfaces [1]

As can be observed, most legacy operations of ns-2 are still preserved. Incoming traffic arrives through the corresponding channel and travel through the different components in ascending order then eventually merges to a single point at the address multiplexer. For outgoing traffic, the determination of selecting which interface to pass the data packets is to be handled by the routing agent. In other words, modifications will be required in implementing the routing agent to add the intelligence of selecting the appropriate interface, as will be discussed in Section 5.4.

5.3. Extending ns-2 with Multi-Interface Support

This section provides a description of the various changes that the guideline in [1] proposes to add the multiple interface support in ns-2. As can be deduced from the ns-2 architecture mentioned in earlier sections, the modifications involves both the OTCL and C++ realms. The changed lines of code in each modified file can be found in the Appendix section, marked with a commented tag, “MCMI”.

5.3.1. Changes on OTCL Code

Four new OTCL procedures as follows are created in **/ns-2.32/tcl/lib/ns-lib.tcl**:

- **`change-numifs {`**`newnumifs`**`}`**
  - Allows users to specify `newnumifs` interfaces per node. This procedure is to be called prior to creating a wireless node in the scenario script. Once called, it will affect all subsequent nodes until another invocation of the procedure is issued.
- **`add-channel {`**`indexch ch`**`}`**
  - Adds an interface (channel) to a node. This procedure takes two arguments: `indexch` is the index of the channel to be added; `ch` references to the channel object previously created.
- **`get-numifs {}`**
  - Retrieves the number of interfaces currently defined.
- **`ifNum {val}`**
  - Adds multiple interfaces as an argument to `node-config`, which is an existing ns-2 command used to configure a *MobileNode* object, by setting a value for the local variable, `numifs_`.

Secondly, two of the existing OTCL procedures in **/ns-2.32/tcl/lib/ns-lib.tcl** are modified:

- **`node-config args {}`**
  - Adds the support for multiple channels.
  - For backward compatibility, initializes `chan` as a single variable if normal operation is used, or as an array if multiple interfaces are defined.
  - Adds the `numifs_` variable as a new member in the argument list, `args`, that is passed to the procedure.

- **`create-wireless-node {}`**
  - Takes in the number of interfaces specified in `numifs_` and iteratively calls

`add-interface` as many times as the number of interfaces that the node has.
    - `add-interface` is an existing ns-2 procedure that adds an interface to a previously created *MobileNode* object.

Next, several modifications are performed on the following existing ns-2 procedures in **/ns-2.32/tcl/lib/ns-mobilenode.tcl**:

- **`add-target {agent port}`**
    - This procedure attaches a routing `agent` to a *MobileNode* object, picks a `port` and binds the agent to the port number.
    - Gets the number of interfaces via calling `get-numifs`; doing so allows the procedure to determine whether multiple interfaces are present. If the number of interfaces is non-zero (implying that the multiple interface extension is used), the procedure iteratively calls the `if-queue` command as many times as the value returned by `get-numifs`.
- **`add-target-rtagent {agent port}`**
    - This procedure, called by `add-target`, adds a target routing agent.
    - It first gets the number of interfaces that a node has from `get-numifs`. If the number of interfaces is non-zero, the procedure associates the routing agent with the corresponding link layer `target` entity as many times as the number of interfaces for both of the sending and receiving targets.
- **`add-interface {}`**
    - This procedure adds an interface to a *MobileNode* object.
    - Originally, it creates one ARP table (for address resolution) per node. We modify it to be one ARP table per interface although having one ARP table per node resembles the real-life case. The reason for such a walk-around is that, if a node is using one interface to communicate with another one, the current design of *MobileNode* in ns-2 will not allow the node to use another interface since the request to the ARP entity will still be serving the previous interface.
- **`init args {}`** and **`reset {}`**
    - Due to the above change of assigning the ARP table, these two procedures are modified to initialize and reset the ARP table of a *MobileNode* object per the number of interfaces defined.

5.3.2. Changes on C++ Code

After adding the multiple interface support in the OTCL realm, relevant changes have to be made in the C++ code for the *MobileNode* object, the channel entity, and the MAC layer model.

/ns-2.32/common/mobilenode.h

- We define a new declaration of the *MobileNode* lists to replace the existing ones.
  - ns-2 controls each instance of the *MobileNode* objects which are associated with a channel by means of a linked-list. Two lists are managed; one references the previous node, `prevX_`, while the other references the next node, `nextX_`.
  - The original format of the list is simply a pointer to a node. In order to support multiple channels, the list is modified to be an array of pointers with the size of the array being the maximum of number of channels:
    - `nextX_[MAX_CHANNELS]`
    - `prevX_[MAX_CHANNELS]`
  - The index for referring to a node is the channel number.
- We remove the inline declaration of the `getLoc()` function. Due to the above changes on the *MobileNode* lists, the original declaration has been found to always return a zero distance, which leads to wrong packet receptions.

/ns-2.32/common/mobilenode.cc

- We add the `getLoc()` method definition which retrieves the location of a node.

/ns-2.32/mac/channel.cc

- Due to the changes on the *MobileNode* lists, we modify accessing each node entry when attaching, removing, and updating a new node to a channel to refer to the corresponding channel number. This number can be accessed by `this->index()`, where `this` is the current instance of the channel class object. In other words, whenever `nextX_` and `prevX_` appear in **channel.cc**, they need to be replaced by:
  - `nextX_[this->index()]`
  - `prevX_[this->index()]`
- In the `affectedNodes()` function, we add a new condition to check which of the interfaces of the destination node is connected to the same channel before transmitting the packet to another interface. The original design of ns-2 does not consider the case with multiple channels; it simply sends the packet to all of the destination interfaces. Accessing the channel of the destination interface is carried out by:

- `rifp->channel()`, where `rifp` is a pointer to the receiving interface. The `channel()` member returns a channel object that has the same type as `this`.

/ns-2.32/mac/mac-802_11.cc

- For correct handling of multiple interfaces by the routing agent, we modify the `recv()` method in the MAC 802.11 class to register the correct MAC receiving interface in the MAC header. The hardware address of the interface can be access by:
  - `hdr->iface() = addr()`

5.4. Changes on the AODV Routing Protocol Code

As mentioned in an earlier section, the MCR protocol proposed in [2] is not implemented. For simplicity, we have chosen the exiting AODV routing protocol for multi-hop networks in ns-2. AODV is a reactive routing protocol, defined in RFC 3561, for ad-hoc wireless networks, as it establishes a route to a destination only when required [8]. Although the RFC defines the AODV protocol to be capable of supporting multiple wireless interfaces at each node, the original ns-2 design does not incorporate this capability. Hence, the enhancements consist of two parts:

- Support to make use of the multi-interface model (the modified *MobileNode* by [1]).
- Implementation of the *interface switching* algorithm in [2].

First of all, we must understand how routing is performed in ns-2's AODV agent. As an example, we examine how AODV establish a route from the first to the last node in a chain topology with four nodes. AODV builds routes using a Route Request (RREQ)/Route Reply (RREP) query cycle, as illustrated in the figure below.

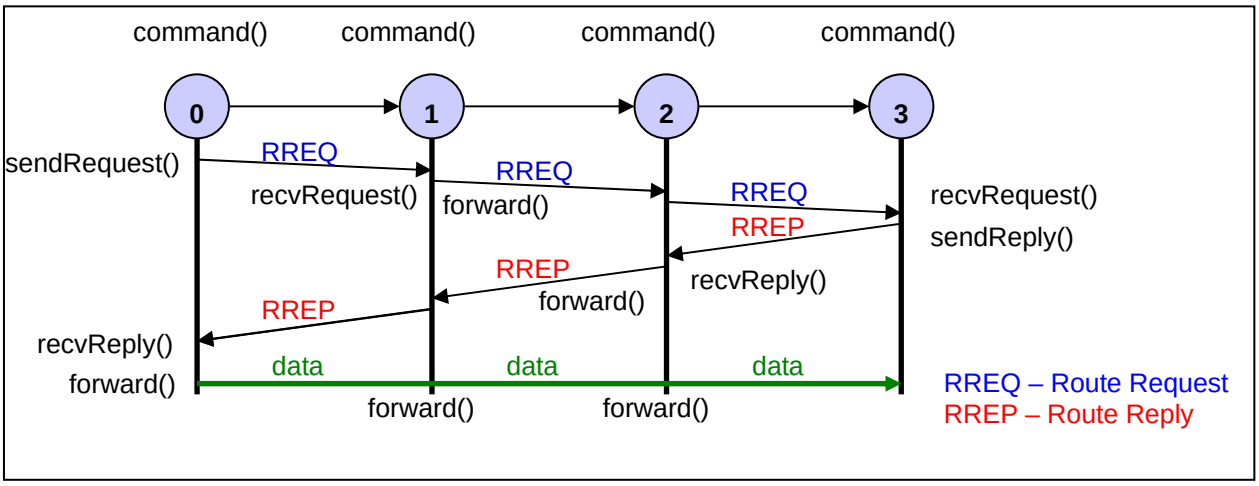

Figure 15: AODV route establishment for a chain topology with 4 nodes

When the nodes in an ad-hoc wireless network are created, function `command()` in ns-2's AODV library is executed by each node to initialize itself in the AODV routing agent. When a source desires a route to a destination node, it broadcasts a RREQ packet across the network by calling `sendRequest()`. Intermediate nodes receiving this packet update their information for the source node and set up a backward pointer to the source node in their routing tables. The packet is also forwarded to the next node via `forward()`. Once a node receiving the RREQ is the desired destination, it unicasts a RREP packet back to the source by calling `sendReply()`. As the RREP propagates back to the source node, the intermediate nodes receiving the packet via `recvReply()` set up a forward pointer to the destination and forward it to the previous node by `forward()`. As soon as the source node receives the RREP, an active route has been established; it may start to forward data packets to the destination by calling `forward()`. The aforementioned steps are known as the Route Discovery state.

Next, the Route Maintenance state (not illustrated in Figure 15 for lack of space) is for updating the route previously established. Occasionally, a node of an active route may offer connectivity information by calling `sendHello()` to broadcast local Hello messages. Whenever a node receives a Hello message from a neighbor via `recvHello()`, the node makes sure that it has an active route to the neighbor, and creates one if necessary. The current node can now begin using this route to forward data packets. Furthermore, when a link breakage is detected by a node, the node will call `sendError()` to notify the source node with a Route Error (RERR) packet. The source node will initialize a new Route Discovery state as mentioned before.

5.4.1. Support to Make Use of the Modified MobileNode

The goal is to utilize the multi-interface model provided by the modified *MobileNode*. The relevant files to be modified are described in the following paragraphs.

/ns-2.32/aodv/aodv.h

- We add a new member to the AODV routing agent class, `nIfaces`, which keeps track of the number of interfaces that the agent is managing.
- We define a constant, `MAX_IF = 12`, for the maximum number of allowable interfaces.
- The routing agent needs to decide which one of the interfaces the outgoing packets should be routed to. Since there are multiple interfaces. The originally used `ifqueue` and `target` pointers need to be modified to two arrays, `ifqueuelist` and `targetlist`. In a sense, these two containers resemble the packet queues maintained by each node, as mentioned in [2].

- `NsObject *targetlist[MAX_IF]`
  - `ifqueuelist` stores the link-layer modules for all the interfaces a particular node has.
- `PriQueue *ifqueuelist[MAX_IF]`
  - `targetlist` keeps the corresponding queue of each individual node.

/ns-2.32/aodv/aodv.cc

- We initialize `nIfaces` to be 0 in the constructor of the AODV agent.
- We modify the `command()` function of the AODV routing agent to make use of the various variables from the following OTCL procedure calls in **/ns-2.32/tcl/lib/ns-mobilenode.tcl**, and to store the corresponding entries in `ifqueuelist` and `targetlist`. Note that all of the OTCL statements highlighted in bold below have four arguments and begin with the routing agent object, `$agent`. Thus, we add a special case in `command()` to deal with the condition in which four parameters are present. The second, the third and the fourth parameters are of our interest; their meanings and respective actions to be carried out in the `command()` function are also described below.
  - **`add-target`**
    - We have previously added the following lines in `add-target`:

```
for {set i 0} {$i < [$self set nifs_]} {incr i} {
      $agent if-queue $i [$self set ifq_($i)]
}
```

    - `if-queue` is a label signifying that the interface queue instance, `ifq_($i)`, should be inserted to index `$i` of `ifqueuelist`.
  - **`add-target-rtagent`**
    - We have also added the following lines in add-target-rtagent:

```
for {set i 0} {$i < [$self set nifs_]} {incr i} {
      set sndT [cmu-trace Send "RTR" $self]
      $agent target $i $sndT
      $sndT target [$self set ll_($i)]
}
for {set i 0} {$i < [$self set nifs_]} {incr i} {
      $agent target $i [$self set ll_($i)]
}
```

    - `target` is a label signifying that the link layer instance, `ll_($i)`, should be inserted to index `$i` of `targetlist`.

Next, we must add some coupling mechanism within the routing agent implementation so that it has a path to which of the interfaces to transmit the packet. In ad-hoc network routing, broadcast transmission is used during the Route Discovery process. In the presence of multiple interfaces, a broadcast packet, RREQ, needs to be sent through all the interfaces that a node has. As a result, we use a for loop to send a copy of the original packet to each interface whenever the AODV routing agent needs to broadcast a packet, which happens in the following methods:

- `sendRequest()`
- `sendError()`
- `sendHello()`
- `forward()`

Please note that, in order to preserve the original behaviours of ns-2, the above changes are only performed when `nIfaces` is non-zero.

Additionally, for routing the packet via a specific interface to a destination, unicast transmission of AODV is exercised. Just knowing the next node is not enough; the routing agent must also consider which output interface to be used to reach the next hop. In the AODV routing table header file, **/ns/2.32/aodv/aodv_rtable.h**, we define a new member, `rt_interface`, which stores the appropriate interface index. `rt_interface` is used to index an entry in `targetlist` within the following functions for the sake of selecting the intended link-layer entity of the interface that the packet needs to be sent to:

- `sendReply()`
- `forward()`

However, the challenge here is how to associate the appropriate output interface with the `rt_interface` index when a new routing table entry is created. This implies that the respective `rt_interface` needs to be updated so as to be able to keep this index. Thanks to the changes in the **/ns-2.32/mac/mac-802_11.cc** file, the address of the incoming interface is one of the members in the common header of the packet; that is, `ch->iface()`. In the following two functions, we have `rt_interface` to hold the index that will be used for creating and/or updating the route table entry:

- `recvRequest()`
- `recvReply()`

Such an index is determined by the following formula:

```
rt_interface = ch->iface()
                      - ((Mac *)ifqueuelist[0]->target())->addr(),
```

where the second term of the subtraction is address of the first interface of the node. If multiple interfaces are not present, we assign `rt_interface` to be -1.

5.4.2. Implementation of Interface Switching

In addition to the various changes in ns-2's AODV that makes use of the multi-interface model from [1], the *interface switching* algorithm from [2] is required to assign interfaces to specific channels, as well as to decide when to switch an interface from one channel to another. We also employ a flag, `MCMI_DEBUG`, for displaying informative debugging messages within each modified function. When building the modified ns-2 with this flag defined, debugging messages will be displayed while a simulation is running; they are useful for validating the functionality of *interface switching* (to be discussed in Section 6.2).

First of all, the following new member variables are created in ns-2's AODV class, **/ns-2.32/aodv/aodv.h**:

- **`MAX_NT_CUL_ENTRIES`**:
  - Maximum number of entries in *NeighbourTable* and *ChannelUsageList*.
- **`neighbour_table[MAX_NT_CUL_ENTRIES]`**:
  - Contains the fixed channels used by the node's neighbours, indexed by the node IP address.
- **`channel_usage_list[MAX_NT_CUL_ENTRIES]`**:
  - Contains the number of nodes using each channel as their fixed channel, indexed by the channel number.
- **`fixed_interface`**, **`switchable_interface`**
  - Fixed and switchable interfaces used by this node.

Next, to facilitate the algorithm of *interface switching*, the RREQ, RREP and Hello packets used by the AODV agent during the Route Discovery and Route Maintenance processes need to contain some additional information about a node, such as its *NeighbourTable*. Hello packets share the same header structure as that of RREP in ns-2's AODV class. The following new variables are added to the `hdr_aodv_request` (for RREQ) and `hdr_aodv_reply` (for RREP and Hello) structures in **/ns-2.32/aodv/aodv_packet.h**:

- **`rq_fixed_channel_used`**/**`rp_fixed_channel_used`**
  - Fixed channel used by this node.
- **`rq_sender_node_ip`**/**`rp_sender_node_ip`**
  - Node IP address of the sender that transmits the RREQ/RREP/Hello packet. It

may not be the originator where the packet is originally from.

- **`*rq_neighbour_table`**/**`*rp_neighbour_table`**
  - A pointer to the *NeighbourTable* of this node.

Finally, the various changes in the relevant functions in ns-2's main AODV class code, **/ns-2.32/aodv/aodv.cc**, are briefly described below. For the completed changes, please refer to the Appendix section. Again, in order to preserve the original behaviours of ns-2, the changes are only performed when `nIfaces` is non-zero.

`command()`

- Initially, the node chooses a random channel for its fixed interface and switchable interface. The following lines of code needs to be performed every time a channel is created by a scenario script and only if the number of interfaces is greater than 1, as the node does not know how many channels it is going to have and there is no need to assignment another interface if the scenario script only intend to simulate an one-interface network. The corresponding lines of code are as follows:
  -
```
fixed_interface = rand() % (temp_+1);
if (nIfaces > 1) {
    do {
        switchable_interface = rand() % (temp_+1);
    } while (switchable_interface == fixed_interface);
}
```
- Next, the node adds the fixed channel used by itself to its *NeighbourTable* and updates the node's *ChannelUsageList* with its fixed channel. The fixed channel is retrieved by accessing the corresponding channel index of the node's current fixed interface.
  -
```
fixed_channel =
((Mac *)ifqueuelist[fixed_interface]->target())->netif()
->channel()->index();
neighbour_table[(int)index] = fixed_channel;
channel_usage_list[fixed_channel]++;
```

`sendRequest()`, `sendReply()`, `sendHello()`

- These function need to add the fixed channel used by this node and its *NeighbourTable* to the outgoing RREQ, RREP, or Hello packet.
  -
```
rq->rq_fixed_channel_used = neighbour_table[(int)index];
rq->rq_sender_node_ip = (int)index;
```

```
rq->rq_neighbour_table = &neighbour_table[0];
```

`recvRequest()`, `recvReply()`, `recvHello()`

- When a node receives a RREQ, RREP, or Hello packet from a neighbour, it updates:
  - The node's *NeighbourTable* with the fixed channel of that neighbour.
    - `neighbour_table[(int)(rq->rq_sender_node_ip)] = rq->rq_fixed_channel_used;`
  - The node's *ChannelUsageList* using the *NeighbourTable* of its neighbour. Doing so ensures the *ChannelUsageList* will contain two-hop channel usage information.

```
for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
   if (i == (int)index) {
      continue;
   }
   if (rq->rq_neighbour_table[i] != -1) {
      channel_usage_list[rq->rq_neighbour_table[i]]++;
   }
}
```

- For `recvReply()` and `recvHello()`, the lines of code are symmetric except for the names of the packet header pointer are `rp` and `rh`, respectively, and the first two letters of the accessed member variables are `rp` in both cases.

`forward()`

- When forwarding a RREQ, RREP or Hello packet received by the node from its neighbour, the function adds the fixed channel used by this node and its *NeighbourTable* to the outgoing packet, as done by the following lines of code:

```
if (rq->rq_type == AODVTYPE_RREQ) {
    rq->rq_fixed_channel_used = neighbour_table[(int)index];
    rq->rq_sender_node_ip = (int)index;
    rq->rq_neighbour_table = &neighbour_table[0];
}
if (rp->rp_type == AODVTYPE_RREP
    || rp->rp_type == AODVTYPE_HELLO) {
    rp->rp_fixed_channel_used = neighbour_table[(int)index];
    rp->rp_sender_node_ip = (int)index;
    rp->rp_neighbour_table = &neighbour_table[0];
}
```

- When forwarding a data packet to a node's neighbour:
  - The function consults the node's *ChannelUsageList* to find the current channel with the largest and the lowest usage.
    - ```
      for (int i = 0; i < nIfaces; i++) {
            if (channel_usage_list[i] == 0) {
                channel_lowest_usage = i;
                continue;
            }
            if (channel_usage_list[i] >=
                channel_usage_list[channel_largest_usage]) {
                channel_largest_usage = i;
            }
            if (channel_usage_list[i] <=
                channel_usage_list[channel_lowest_usage]) {
                channel_lowest_usage = i;
            }
        }
      ```
  - If the node's fixed channel has the largest usage, with a probability of 0.4 (from paper [2]), the node reverses its *ChannelUsageList* about the fixed channel previously used changes its fixed channel to a less used channel. The node then transmits a new Hello packet informing neighbours of its new fixed channel by calling `sendHello()`.
    - ```
      if (channel_usage_list[neighbour_table[(int)index]] ==
            channel_usage_list[channel_largest_usage]) {
          int switch_probability = rand() % 101;

          if (switch_probability <= 40) {
              int fixed_channel =
              ((Mac *)ifqueuelist[fixed_interface]->target())
              ->netif()->channel()->index();
              channel_usage_list[fixed_channel]--;

              fixed_channel = channel_lowest_usage;
              channel_usage_list[fixed_channel]++;
              neighbour_table[(int)index] = fixed_channel;
      ```

```
                for (int i = 0; i < nIfaces; i++) {
                    if (((Mac *)ifqueuelist[i]->target())->
                netif()->channel()->index() == fixed_channel) {
                      fixed_interface = i;
                      break;
                    }
                 }
                 sendHello();
            }
        }
```

- If the usage of the node's fixed channel is ok, the node looks up the fixed channel of the next node in its *NeighbourTable* and assigns this fixed channel to the its switchable interface.
  - ```
    int switchable_channel =
    neighbour_table[(int)(rt->rt_nexthop)];
    for (int i = 0; i < MAX_IF; i++) {
          if ((((Mac *)ifqueuelist[i]->target())->netif()
              ->channel()->index()) == switchable_channel) {
              switchable_interface = i;
              break;
          }
    }
    ```

5.5. Modified ns-2 with Multi-Interface and Interface Switching Capabilities

In summary, the following chart provides a hierarchical view of the various modified files in ns-2.32 as discussed in earlier sub-sections of Section 5.

Interface Switching	Multi-Interface Support		
/ns-2.32	**/ns-2.32**	**/ns-2.32**	**/ns-2.32**
/aodv	**/common**	**/mac**	**/tcl**
/aodv.cc	/mobilenode.cc	/channel.cc	/lib
/aodv	/mobilenode.h	/mac.h	/ns-lib.tcl
/aodv_packet.h		/mac-802_11.cc	/ns-mobilenode.tcl
/aodv_rtable.h			

Figure 16: Modified files ns-2 with multi-interface and interface switching capabilities
    ```

6. Results and Discussions

We have simulated a single-route multi-channel multi-interface ad-hoc wireless network using the modified ns-2 simulator that incorporates the support for multiple channels and multiple interfaces as well as the approach of *interface switching*. The interface assignment algorithm does not require any modifications to IEEE 802.11's data-link and physical layers. We wish to evaluate the effectiveness of the *interface switching* algorithm implemented in ns-2's existing AODV routing agent, as well as the throughput performance between ah-hoc wireless networks with and without the exploitation of multiple channels and multiple interfaces. Multi-path routing is not tested as the MCR protocol proposed by [2] is not implemented.

6.1. Simulation Scenario

To demonstrate the performance of our implemented *interface switching* algorithm in ns-2 and the benefits of using multiple channels and multiple interfaces, ad-hoc wireless networks in simple chain topologies are used for simplicity. The reasons are that only one single route between the source and the destination is involved and that direct communication is only possible between adjacent nodes on the chain, as illustrated in the following block diagram.

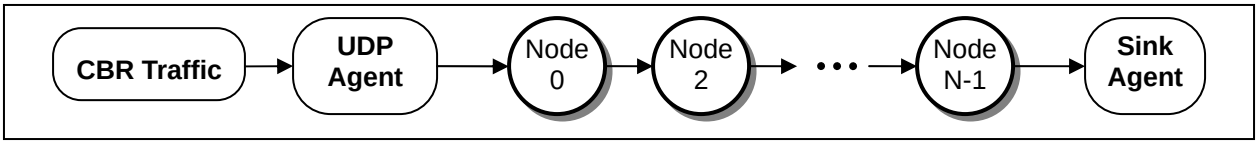

Figure 17: Block diagram of the ns-2 simulation scenario in chain topologies

The following list summarizes the various scenario parameters for our simulations:

- The length of the chain topologies is varied from **2** to **11 nodes**, denoted by *N*.
- The nodes are stationary within a confined area of **1000 m x 1000 m**.
- The number of orthogonal channels is varied from **2** to **4**, denoted by *C*.
- Each node is always equipped with **two interfaces** (**1 fixed**, **1 switchable**); thus, $I = 2$.
- The duration of each simulation is **60 s**; however, the actual simulation time increases as the length of the chain topology as well as the number of channels increase.
- The data rate or the channel capacity is set to **5.4 Mbps**.
- **Constant Bit-Rate (CBR) traffic** is passed from Node 0 to Node *N*-1, as denoted by the ns-2 CBR Traffic application and the Sink Agent in Figure 17.
    - The data packet size is set to **1000 bytes**.

- The CBR data (packetization) rate is set to **1.4 ms**, which is chosen to be large enough to saturate the network according to the following calculation:

$$\frac{(1000bytes)(8bits/byte)}{1.4ms} = 5.7Mbps > 5.4Mbps,$$

  where 5.4 Mbps is our specified channel capacity.
- Data packets are transported using the **User Datagram Protocol (UDP)**, as indicated by the UDP Agent in Figure 17. Therefore, no flow and congestion controls will take place that may impair the throughput performance. As well, the packets can be transmitted as fast as the packetization rate.

The performance of the above multi-channel multi-interface scenarios is compared with the performance of AODV when using a single channel single interface ad-hoc wireless network. The complete ns-2 simulation script, **wireless_chain_mcmi.tcl**, can be found in the Appendix section. The following highlights some key configurations for the case of 4 nodes in the chain topology, 3 channels and 2 interfaces:

```
Mac/802_11 set dataRate_  5.4e6
set val(nn)               4
set val(nc)               3
set val(ni)               2
set pktsize               1000
set pktrate               0.0014

set udp0 [new Agent/UDP]
$ns_ attach-agent $n(0) $udp0

set sink0 [new Agent/Null]
set last_node_id [expr $val(nn)-1]
$ns_ attach-agent $n($last_node_id) $sink0
$ns_ connect $udp0 $sink0

set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packetSize_ $pktsize
```

Set up the channel capacity, the numbers of nodes, channels, and interfaces, packet size and packetization rate.

Define the UDP Agent and attach it to Node 0

Define the Sink Agent and attach it to Node *N*-1. Finally connect the UDP Agent with the Sink Agent to specify the data flow path

Define the CBR Traffic application, attach it to the UDP Agent, and specify the CBR packet size and data rate

Figure 18: Simulation set-up for chain topology with 4 nodes, 3 channels and 2 interfaces

6.2. Interface Switching Validation

In this section, we validate our implemented *interface switching* algorithm in ns-2's AODV routing agent, seeing if the simulated results comply with the proposed algorithm in [2]. A validation example involving 4 nodes in a chain topology, 3 channels (0, 1, 2), and 2 interfaces (0, 1) is presented as a table in Figure 19. This table is manually constructed by examining the debugging messages generated by a simulation run (with the `MCMI_DEBUG` flag turned on when building the modified ns-2 simulator).

We will employ the following terminologies in the table:

- NT[*N*] is the *NeighbourTable* denoting the fixed channel used by Node *N*.
- CUL[*C*] is the *ChannelUsageList* denoting the number of nodes using channel *C* as their fixed channel.
- FC is the fixed channel currently used by the node for receiving data from its neighbour.
- SC is the switchable channel currently used by the node for sending data to its neighbour.

	Node 0	Node 1	Node 2	Node 3
Initially	NT[0] = 2 CUL[2] = 1 FC = 2 SC = 0	NT[1] = 0 CUL[0] = 1 FC = 0 SC = 2	NT[2] = 2 CUL[2] = 1 FC = 2 SC = 0	NT[3] = 1 CUL[1] = 1 FC = 1 SC = 2
Step 1		NT[0] = 2 NT[1] = 0 CUL[0] = 1 CUL[2] = 1	NT[1] = 0 NT[2] = 2 CUL[0] = 1 CUL[2] = 2	NT[2] = 2 NT[3] = 1 CUL[0] = 1 CUL[1] = 1 CUL[2] = 1
Step 2	NT[0] = 2 NT[1] = 0 CUL[0] = 1 CUL[2] = 2	NT[0] = 2 NT[1] = 0 NT[2] = 2 CUL[0] = 1 CUL[1] = 1 CUL[2] = 2	NT[1] = 0 NT[2] = 2 NT[3] = 1 CUL[0] = 1 CUL[1] = 1 CUL[2] = 2	
Step 3	FC = 2 but CUL[2] is large → FC = 1 NT[0] = 1 NT[1] = 0 CUL[0] = 1 CUL[1] = 1 CUL[2] = 1	NT[0] = 1 NT[1] = 0 NT[2] = 2 CUL[0] = 1 CUL[1] = 2 CUL[2] = 1	NT[1] = 0 NT[2] = 2 NT[3] = 1 CUL[0] = 1 CUL[1] = 1 CUL[2] = 1	NT[2] = 2 NT[3] = 1 CUL[0] = 1 CUL[1] = 1 CUL[2] = 1
Finally	FC = 1 SC = 0	FC = 0 SC = 2	FC = 2 SC = 1	FC = 1 SC = 2

Figure 19: Interface switching in chain topology with 4 nodes, 3 channels and 2 interfaces

Changes in the *NeighbourTable* and *ChannelUsageList* are highlighted in red; final results are highlighted in green. As can be seen in Figure 19, all the nodes choose a random channel for its fixed and switchable interfaces initially; their *NeighbourTable* and *ChannelUsageList* are populated accordingly. These tasks are performed in `command()`.

In Step 1, Node 0 calls `sendRequest()` to transmit a RREQ packet to Node 1 which carries the fixed channel used by Node 0 and its *NeighbourTable*. Upon `recvRequest()`receiving the RREQ, Node 1 updates its *NeighbourTable* with the fixed channel of Node 0 as well as its *ChannelUsageList* using Node 0's *NeighbourTable*. Next, Node 1 calls `forward()` to forward the RREQ packet to Node 2, which contains the fixed channel used by Node 1 and its updated *NeighbourTable*. Node 2 then updates its *NeighbourTable* with the fixed channel of Node 1 as well as its *ChannelUsageList* using Node 1's *NeighbourTable*. The similar RREQ packet exchange and *NeighbourTable* update are performed between Node 2 and Node 3 later as well.

In Step 2, Node 3 calls `sendReply()` to send a RREP to Node 2, which includes Node 3's fixed channel and *NeighbourTable*. In `recvReply()`, Node 2 updates its *NeighbourTable* with the fixed channel of Node 3 and its *ChannelUsageList* using Node 3's *NeighbourTable*. Node 2 then calls `forward()` to forward the RREP packet to Node 1, which contains the updated information. The similar actions are also performed between Node 1 and Node 0.

In Step 3, since the destination (Node 3) and the intermediate hops (Nodes 1, 2) have replied to the source (Node 0), a route has been established. CBR data traffic is now ready to be transported along the chain. However, while executing `forward()`, Node 0 realizes that its fixed interface is on Channel 2, which has a usage of 2 among all the available channels. With 2 being the largest usage currently, Node 0 reverses its *ChannelUsageList* about the fixed channel previously used and changes its fixed channel to Channel 1 since CUL[1] = 0. The randomly determined switching tendency at the time of simulation is 0.14, which is below the switching probability of 0.4; thus, the channel switching occurs. Node 0 then calls `sendHello()` to broadcast a Hello packet to inform its neighbours of its new fixed channel. When Node 0 and Node 1 receives the broadcasted Hello packet via `recvHello()`, they further update their *NeighbourTable* with the fixed channel of Node 0 and their *ChannelUsageList* using the *NeighbourTable* of Node 0.

Finally, we have verified that *interface switching* effectively assigns distinctive fixed channels to each successive node. None of the adjacent channels are interfering each other when data packets are transmitted along the chain. At a result, intermediate nodes can send data to the next node using its switchable interface, while receiving data on its fixed interface.

6.3. Simulation Output

Figure 20 presents a sample ns-2 simulation output for chain topology with 4 nodes, 3 Channels, and 2 interfaces. As can be observed, CBR traffic is flowed from Node 0 to Node 3. The simulation duration is 60 s and the average end-to-end throughput is 2586.36 kbps.

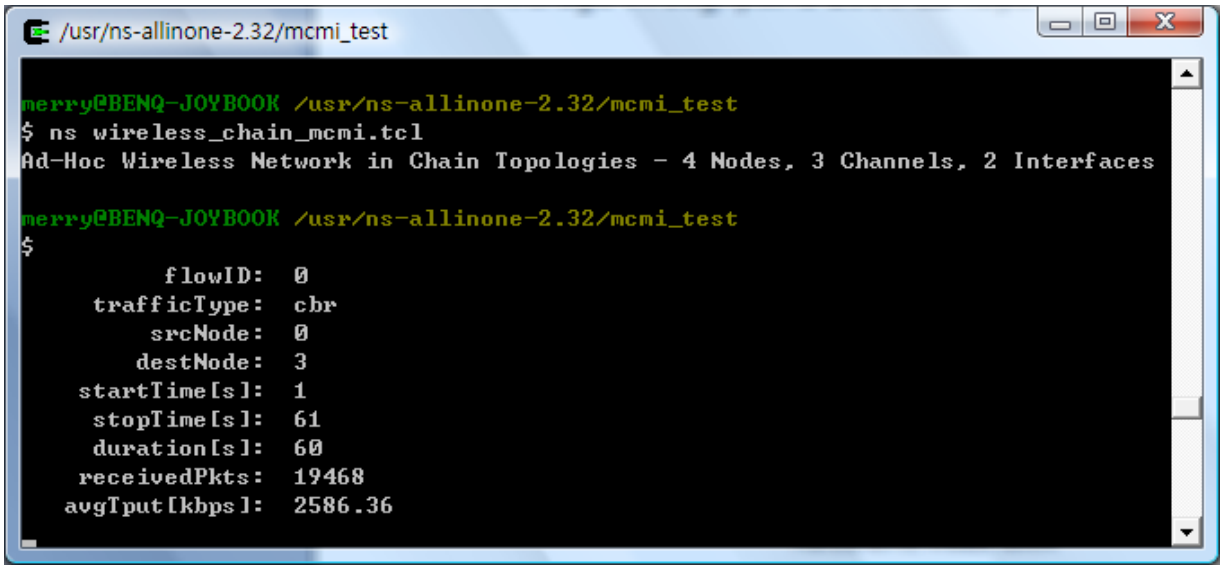

Figure 20: ns-2 output for chain topology with 4 nodes, 3 channels and 2 interfaces

The corresponding network animator (nam) view of the aforementioned simulation scenario is shown in the following figure. Please note that only wireless nodes (MobileNode) can be displayed in nam in the current ns-2. Dumping of traffic data and thus visualization of data packet movements for wireless scenarios is still not supported.

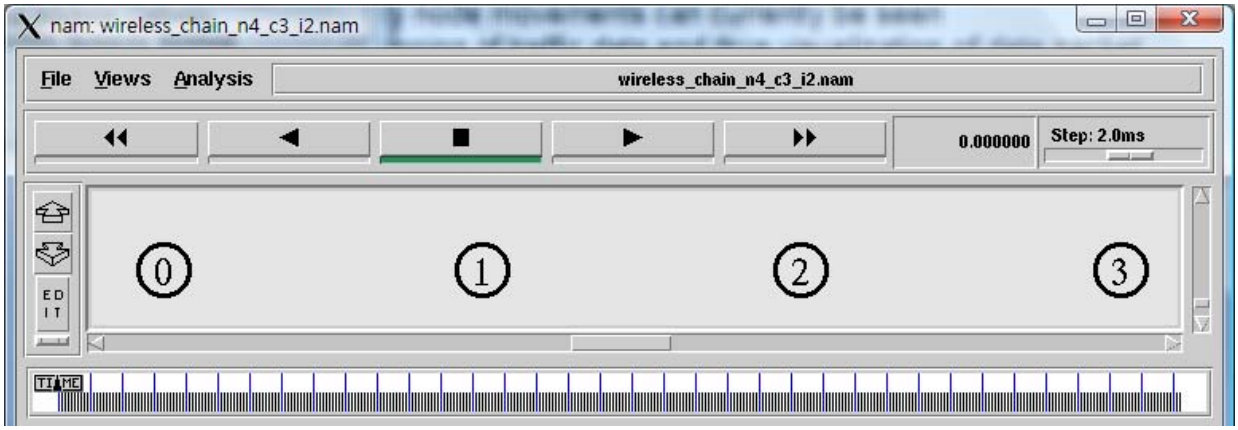

Figure 21: nam output for chain topology with 4 nodes, 3 channels and 2 interfaces

6.4. Throughput Performance

We evaluate the end-to-end throughput of the ad-hoc wireless network in simple chain topologies. The throughput is obtained using an AWK script written by Marco Fiore in **avgStats.awk**, which can be found in the Appendix section. When an ns-2 wireless simulation completes, we let the simulation script to call this AWK script automatically to analyze the wireless traffic trace generated by the simulator. While varying the number of channels from 2 to 4 and equipping each node with two IEEE 802.11 interfaces, we designate each scenario setting as *x*C2I, where *x* is the number of channels.

In each scenario, the length of the chain is varied from 2 to 11 nodes. We compare the results of xC2I with chain topologies in single channel and single interface; that is, 1C1I. Since channel assignments for each node are carried out on a random basis, for the throughput of each scenario, we perform the simulation four times and use the average value of the four obtained throughputs to plot the comparison graph, as illustrated in the following figure.

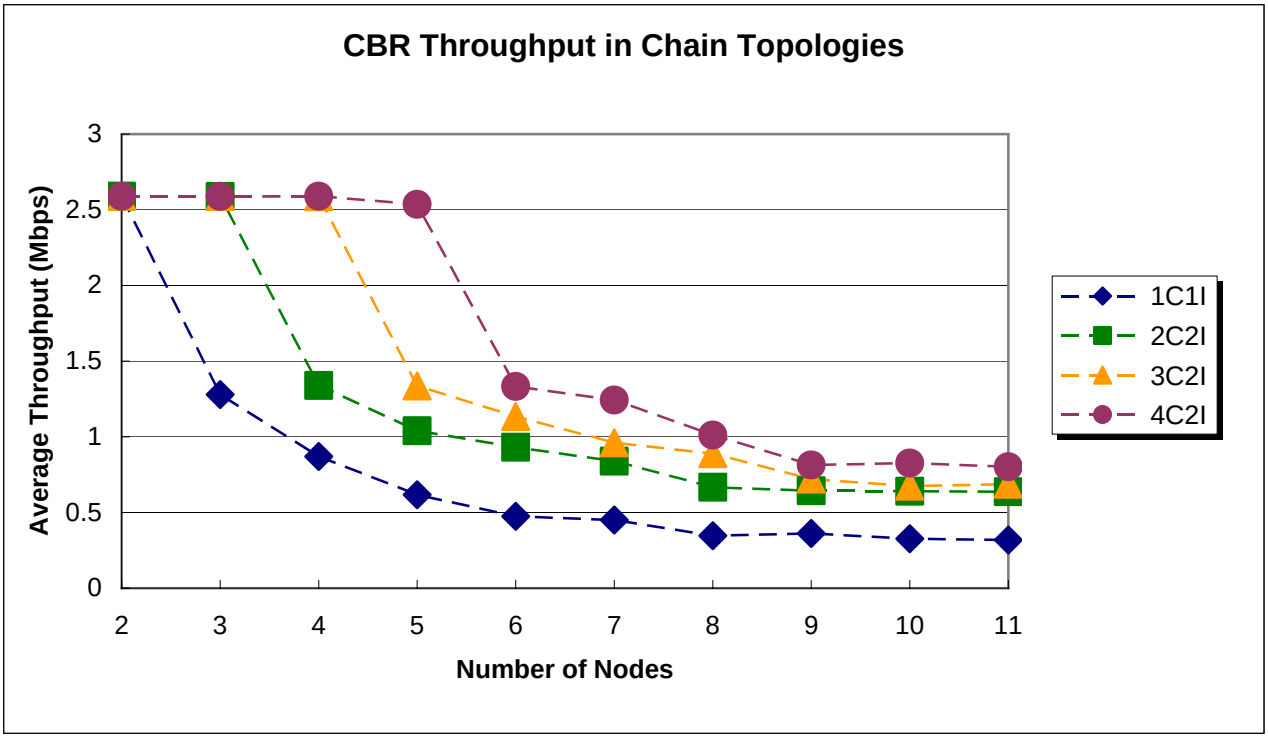

Figure 22: Throughputs of single-channel and multi-channel networks in chain topologies

As can be seen from the graph above, the throughput of the 1C1I scenario degrades as the number of nodes along the chain increases by 1 each time. One reason is that intermediate nodes

cannot send and receive data at the same time, reducing the achievable throughput by about half. Another factor is that the interference within the carrier sense range inhibits transmissions on adjacent nodes; thus, the resultant throughput is also impaired.

By contrast, we observe higher throughputs with multiple channels and multiple interfaces on each node, such as 2C2I, 3C2I and 4C2I. Due to the fact that *interface switching* assigns the fixed channel of successive nodes to different channels, intermediate nodes can send data to the next node using its switchable interface, while receiving data on its fixed interface.

However, the throughput improvements are smaller when the number of nodes is greater than the number of channels + 1. When the chain length goes beyond this threshold, some adjacent nodes will be likely on some common channels; thus, interference will degrade the throughput. Nevertheless, the overall throughputs of these multi-channel multi-interface scenarios are generally still higher than the case of 1C1I.

In addition, more channels are useful with longer chains. Over a chain of two nodes, at most two channels can be utilized (one channel for each node) despite more available channels may be available. Therefore, the performance with three or more channels is similar as the performance of two channels over a two-node chain (though higher than that of 1C1I). By the same idea, at most three channels can be used over a chain of three nodes; the performance with more channels over a three-node chain is also similar. On the other hand, over a chain of 5 nodes, more channels can be utilized over different nodes; therefore, having 4 channels is better than having 3 or less number of channels.

In words, the simulation results have demonstrated that multiple channels and multiple interfaces can noticeably improve the throughput in ad-hoc wireless networks. Furthermore, the number of wireless interfaces can be less than the number of available channels. In our simulation, even two interfaces can utilize multiple channels.

7. Future Work and Improvement

Due to the time constraint of the project, we do not implement the Multi-Channel Routing (MCR) metric proposed by [2] in the ns-2 simulator. Therefore, routing is not tested during our simulation. As pointed out in the paper, merely the *interface switching* solution implemented at the data-link layer may not be sufficient for effectively utilizing multiple channels, as the routing protocol may select routes wherein successive hops interfere with each other.

In this project, we have enhanced the existing implementation of the AODV routing agent in ns-2 to support multiple interfaces. AODV selects the route that has the least number of hops through other nodes [2]. The shortest path does not distinguish between a route that uses many channels and a route that uses fewer channels. The following diagram illustrates the need for a specialized routing protocol. Suppose all nodes in the ad-hoc wireless network in the figure have already chosen their fixed interfaces. AODV would favour routes A-B-C as it is the shortest. However, routes A-D-E-C use different channels between two adjacent nodes. This path allows all links to be active simultaneously; thus, the end-to-end throughput can be potentially higher. On the other hand, both links A-B and B-C use channel 3; only one of the links can be active at a time, implying the resulting throughput is halved.

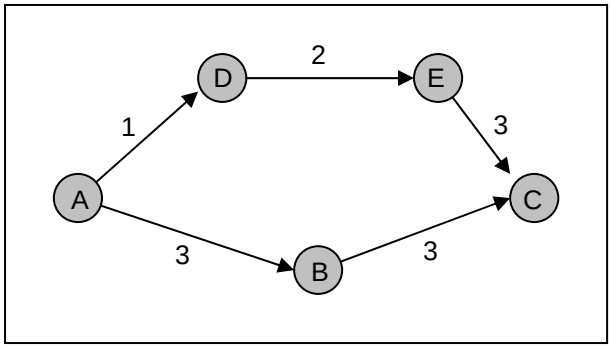

Figure 23: The scenario that illustrates the need for selecting channel deviated routes [2]

As a result, the more ideal implementation of a multi-channel multi-interface ad-hoc wireless network is to incorporate the proposed MCR as the routing agent in ns-2. With this capability, we are able to effectively evaluate the throughput performance in random topologies with multiple traffic flows. However, the underlying work required for implementing the MCR may be beyond the scope of a regular course project.

8. Conclusion

In this project, we have extended the Network Simulator (ns-2.32) to support multiple channels and multiple interfaces by modifying ns-2's *MobileNode* library to support multiple interfaces. By referring to the guideline written by R. A. Calvo and J. P. Campo in [1], the legacy operations of IEEE 802.11 wireless interfaces in ns-2 can be preserved. Not only have we gained valuable exposure to ns-2's wireless node model, we have an opportunity to acquire a good understanding of ns-2's architecture.

We have also studied and explored the use of multiple channels and multiple interfaces in wireless ad-hoc networks. We have learned about the interface assignment strategy proposed by P. Kyasanur and N. H. Vaidya in [2], which uses the notion of fixed and switchable interfaces. This assignment approach allows effective utilization of all the available channels even when the number of interfaces is smaller than the number of channels. Although quite a few amount of times have been spent to study the Ad-hoc On-demand Distance Vector (AODV) routing in ns-2, we have successfully integrated this algorithm in the AODV routing agent in ns-2. With the implementation of the multi-channel multi-interface support in ns-2 as well as the development of the simulation script, we have expanded our horizons in C++ and TCL programming.

Next, using the modified ns-2 with the multi-interface and *interface switching* capabilities, we have simulated a multi-channel multi-interface ad-hoc wireless network in simple chain topologies. By varying the number of available channels and the number of nodes in the chain while keeping each node to have two interfaces (one fixed and one switchable), we have generated various *interface switching* interactions and end-to-end throughputs using the modified ns-2. Simulation results validate the effectiveness of our implemented *interface switching* algorithm in ns-2's AODV routing protocol. More importantly, the results demonstrate that network throughput can significantly improves in ad-hoc wireless networks with multiple channels and multiple interfaces in each wireless node.

Glossary

AODV	Ad-hoc On-demand Distance Vector
ARP	Address Resolution Protocol
CBR	Constant Bit Rate
CUL	ChannelUsageList
DSDV	Destination-Sequenced Distance-Vector Routing
DSR	Dynamic Source Routing
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
MAC	Media Access Control
MCMI	Multi-Channel Multi-Interface
nam	Network Animator
ns-2	Network Simulator 2
NT	NeighbourTable
OTCL	Object TCL
RFC	Request for Comments
RERR	Route Error
RREP	Route Reply
RREQ	Route Request
SSCH	Slotted Seeded Channel Hopping
TCL	Tool Command Language
TORA	Temporally Ordered Routing Algorithm
UDP	User Datagram Protocol
WCETT	Weighted Cumulative Expected Transmission Time

References

[1] R. A. Calvo and J. P. Campo, “Adding Multiple Interface Support in NS-2,” [User Guide], University of Cantabria, Jan. 2007.

[2] P. Kyasanur and N. H. Vaidya, “Routing and Link-layer Protocols for Multi-Channel Multi-Interface Ad Hoc Wireless Networks,” *SIGMOBILE Mobile Computing and Communications Review*, vol. 10, no. 1, pp. 31-43, Jan. 2006.

[3] A. Raniwala, K. Gopalan, and T. Chiueh, “Centralized Channel Assignment and Routing Algorithms for Multi-Channel Wireless Mesh Networks,” *Mobile Computing and Communications Review (MC2R) 2004*, vol. 8, no. 2, pp. 50-65, Apr. 2004.

[4] S. Wu, C. Lin, Y. Tseng, and J. Sheu, “A New Multi-Channel MAC Protocol with On-Demand Channel Assignment for Multi-Hop Mobile Ad Hoc Networks,” *International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), 2000*, Dallas, TX, pp. 232-237, Dec. 2000.

[5] R. Maheshwari, H. Gupta, and S. R. Das, “Multichannel MAC Protocols for Wireless Networks,” *Sensor and Ad Hoc Communications and Networks (SECON) 2006*, Reston, VA, vol. 2, pp. 393-401, Sept. 2006.

[6] P. Bahl, R. Chandra, and J. Dunagan, “SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-Hoc Wireless Networks,” *ACM Annual International Conference on Mobile Computing and Networking (MobiCom) 2004*, Philadelphia, PA, pp. 216-230, Oct. 2004.

[7] R. Draves, J. Padhye, and B. Zill, “Routing in Multi- Radio, Multi-Hop Wireless Mesh Networks,” *ACM Annual International Conference on Mobile Computing and Networking (MobiCom) 2004*, Philadelphia, PA, pp. 114-128, Oct. 2004.

[8] C. Perkins, et. al., “Ad hoc On-Demand Distance Vector (AODV) Routing,” RFC 3561, *Network Working Group*, The Internet Society, Jul. 2003.

[9] K. Fall and K. Varadhan, The ns Manual [Online]. Available: http://www.isi.edu/nsnam/ns.

Appendix

A.1. Code Listing

This section provides the ns-2 source code that we have modified in implementing the multi-channel multi-interface ad-hoc wireless network, as well as the scenario (test) scripts we use to demonstrate the effectiveness of exploiting multiple channels and multiple interfaces. Please note that due to lack of space, only the modified lines of code (marked with a “MCMI” tag) along with the entire function body that they belong to are presented in this section.

Test Script

- wireless_chain_mcmi.tcl
- avgStats.awk

Source Code

/ns-2.32

- **/aodv**
  - /aodv.cc
  - /aodv
  - /aodv_packet.h
  - /aodv_rtable.h
- **/common**
  - /mobilenode.cc
  - /mobilenode.h
- **/mac**
  - /channel.cc
  - /mac.h
  - /mac-802_11.cc
- **/tcl**
  - **/lib**
    - /ns-lib.tcl
    - /ns-mobilenode.tcl

```
#===================================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:     Chih-Hao Howard Chang
#              20007-2192
#              howardc@sfu.ca
#
# Description: Simulation script for multi-Channel multi-Interface Ad-Hoc
#              wireless network in chain topologies
#
# File:        wireless_chain_mcmi.tcl
#===================================================================================


#=====================================
# MAC Layer Setup
#=====================================
Mac/802_11 set dataRate_    5.4e6          ;# rate for data frames in Mbps


#=====================================
# Simulation Parameters Setup
#=====================================
set val(chan)   Channel/WirelessChannel    ;# channel type
set val(prop)   Propagation/TwoRayGround   ;# radio-propagation model
set val(netif)  Phy/WirelessPhy            ;# network interface type
set val(mac)    Mac/802_11                 ;# MAC type
set val(ifq)    Queue/DropTail/PriQueue    ;# interface queue type
set val(ll)     LL                         ;# link layer type
set val(ant)    Antenna/OmniAntenna        ;# antenna model
set val(ifqlen) 50                         ;# max packet in ifq
set val(rp)     AODV                       ;# routing protocol
set val(x)      1000                       ;# X dimension of topography
set val(y)      1000                       ;# Y dimension of topography
set val(stop)   70                         ;# nam stop time

set val(nn)     4                          ;# number of mobilenodes
set val(nc)     3                          ;# number of channels
set val(ni)     2                          ;# number of interfaces, <= number of channels
set pktsize     1000                       ;# packet size in bytes
set pktrate     0.0014                     ;# packet rate in seconds

set filename    wireless_chain_n$val(nn)_c$val(nc)_i$val(ni)
puts "Ad-Hoc Wireless Network in Chain Topologies - $val(nn) Nodes, $val(nc) Channels,
$val(ni) Interfaces"


#=====================================
# Initialization
#=====================================
# Create a ns simulator
set ns_ [new Simulator]

# Setup topography object
set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)
set god_ [create-god [expr $val(nn)*$val(nc)]]

# Open the NS trace file
set tracefd [open $filename.tr w]
$ns_ trace-all $tracefd
$ns_ use-newtrace

# Open the NAM trace file
set namfile [open $filename.nam w]
$ns_ namtrace-all $namfile
$ns_ namtrace-all-wireless $namfile $val(x) $val(y)

# Create wireless channels
for {set i 0} {$i < $val(nc)} {incr i} {
```

```
    set chan($i) [new $val(chan)]
}


#===================================
# Mobile Node Parameter Setup
#===================================
$ns_ node-config -adhocRouting  $val(rp) \
                 -llType        $val(ll) \
                 -macType       $val(mac) \
                 -ifqType       $val(ifq) \
                 -ifqLen        $val(ifqlen) \
                 -antType       $val(ant) \
                 -propType      $val(prop) \
                 -phyType       $val(netif) \
                 -channel       $chan(0) \
                 -topoInstance  $topo \
                 -agentTrace    ON \
                 -routerTrace   ON \
                 -macTrace      OFF \
                 -movementTrace OFF \
                 -ifNum         $val(ni)


#===================================
# Nodes Definition
#===================================
$ns_ change-numifs $val(nc)
for {set i 0} {$i < $val(nc)} {incr i} {
    $ns_ add-channel $i $chan($i)
}

# Create nodes
for {set i 0} {$i < $val(nn)} {incr i} {
    set n($i) [$ns_ node]
    $god_ new_node $n($i)
}

# Set node positions in horizontal chain topology
set nodedist 250
for {set i 0} {$i < $val(nn)} {incr i} {
    $n($i) set X_ [expr $i * $nodedist + 20]
    $n($i) set Y_ 50
    $n($i) set Z_ 0.0
    $ns_ initial_node_pos $n($i) 40
    $n($i) random-motion 0
}


#===================================
# Agents Definition
#===================================
set udp0 [new Agent/UDP]
$ns_ attach-agent $n(0) $udp0

set sink0 [new Agent/Null]
set last_node_id [expr $val(nn)-1]
$ns_ attach-agent $n($last_node_id) $sink0
$ns_ connect $udp0 $sink0


#===================================
# Applications Definition
#===================================
# Setup a CBR Application over UDP connection
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packetSize_ $pktsize
$cbr0 set interval_ $pktrate
$ns_ at 1.0 "$cbr0 start"
$ns_ at 61.0 "$cbr0 stop"
```

```
142
143
144 #=====================================
145 # Simulation Termination
146 #=====================================
147 # Define a finish procedure
148 proc finish {} {
149     global ns_ tracefd filename pktsize last_node_id
150     global namfile
151     $ns_ flush-trace
152     close $tracefd
153     close $namfile
154     exec nam $filename.nam &
155
156     # Call throughput analyzer (AWK scripts written by Marco Fiore, marco.fiore@polito.it)
157     exec awk -f avgStats.awk src=0 dst=$last_node_id flow=0 pkt=$pktsize $filename.tr &
158     exit 0
159 }
160 for {set i 0} {$i < $val(nn) } { incr i } {
161     $ns_ at $val(stop) "\$n($i) reset"
162 }
163 $ns_ at $val(stop) "$ns_ nam-end-wireless $val(stop)"
164 $ns_ at $val(stop) "finish"
165 $ns_ at $val(stop) "puts \"done\" ; $ns_ halt"
166 $ns_ run
167
```

```
#===============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Post-processing script for analyzing the average throughput based
#               on the wireless traffic trace produced by a ns-2 wireless
#               simulation, written by Marco Fiore, marco.fiore@polito.it
#
# File:         avgStats.awk
#===============================================================================


BEGIN {
        recvdSize = 0
        startTime = 1e6
        stopTime = 0
}

{
        # Trace line format: normal
        if ($2 != "-t") {
                event = $1
                time = $2
                if (event == "+" || event == "-") node_id = $3
                if (event == "r" || event == "d") node_id = $4
                flow_id = $8
                pkt_id = $12
                pkt_size = $6
                flow_t = $5
                level = "AGT"
        }
        # Trace line format: new
        if ($2 == "-t") {
                event = $1
                time = $3
                node_id = $5
                flow_id = $39
                pkt_id = $41
                pkt_size = $37
                flow_t = $45
                level = $19
        }

        # Store packets send time
        if (level == "AGT" && flow_id == flow && node_id == src &&
            sendTime[pkt_id] == 0 && (event == "+" || event == "s") && pkt_size >= pkt) {
                if (time < startTime) {
                        startTime = time
                }
                sendTime[pkt_id] = time
                this_flow = flow_t
        }

        # Update total received packets' size and store packets arrival time
        if (level == "AGT" && flow_id == flow && node_id == dst &&
            event == "r" && pkt_size >= pkt) {
                if (time > stopTime) {
                        stopTime = time
                }
                # Rip off the header
                hdr_size = pkt_size % pkt
                pkt_size -= hdr_size
                # Store received packet's size
                recvdSize += pkt_size
                # Store packet's reception time
                recvTime[pkt_id] = time
        }
```

```
 72
 73 }
 74
 75 END {
 76         # Compute average delay
 77         delay = avg_delay = recvdNum = 0
 78         for (i in recvTime) {
 79                 if (sendTime[i] == 0) {
 80                         printf("\nError in delay.awk: receiving a packet that wasn't sent
    %g\n",i)
 81                 }
 82                 delay += recvTime[i] - sendTime[i]
 83                 recvdNum ++
 84         }
 85         if (recvdNum != 0) {
 86                 avg_delay = delay / recvdNum
 87         } else {
 88                 avg_delay = 0
 89         }
 90
 91         # Compute average jitters
 92         jitter1 = jitter2 = jitter3 = jitter4 = jitter5 = 0
 93         prev_time = delay = prev_delay = processed = deviation = 0
 94         prev_delay = -1
 95         for (i=0; processed<recvdNum; i++) {
 96                 if(recvTime[i] != 0) {
 97                         if(prev_time != 0) {
 98                                 delay = recvTime[i] - prev_time
 99                                 e2eDelay = recvTime[i] - sendTime[i]
100                                 if(delay < 0) delay = 0
101                                 if(prev_delay != -1) {
102                                         jitter1 += abs(e2eDelay - prev_e2eDelay)
103                                         jitter2 += abs(delay-prev_delay)
104                                         jitter3 += (abs(e2eDelay-prev_e2eDelay) - jitter3) /
    16
105                                         jitter4 += (abs(delay-prev_delay) - jitter4) / 16
106                                  }
107 #                                deviation += (e2eDelay-avg_delay)*(e2eDelay-avg_delay)
108                                  prev_delay = delay
109                                  prev_e2eDelay = e2eDelay
110                         }
111                         prev_time = recvTime[i]
112                         processed++
113                 }
114         }
115         if (recvdNum != 0) {
116                 jitter1 = jitter1*1000/recvdNum
117                 jitter2 = jitter2*1000/recvdNum
118         }
119 #       if (recvdNum > 1) {
120 #               jitter5 = sqrt(deviation/(recvdNum-1))
121 #       }
122
123         # Output
124         if (recvdNum == 0) {
125                 printf("################################################################################\
    n" \
126                        "#  Warning: no packets were received, simulation may be too short
    #\n" \
127                        "################################################################################\
    n\n")
128         }
129         printf("\n")
130         printf(" %15s:  %g\n", "flowID", flow)
131         printf(" %15s:  %s\n", "trafficType", this_flow)
132         printf(" %15s:  %d\n", "srcNode", src)
133         printf(" %15s:  %d\n", "destNode", dst)
134         printf(" %15s:  %d\n", "startTime[s]", startTime)
135         printf(" %15s:  %d\n", "stopTime[s]", stopTime)
136         printf(" %15s:  %d\n", "duration[s]", stopTime-startTime)
137         printf(" %15s:  %g\n", "receivedPkts", recvdNum)
```

```
138         printf(" %15s:  %g\n", "avgTput[kbps]", (recvdSize/(stopTime-startTime))*(8/1000))
139 }
140
141 function abs(value) {
142         if (value < 0) value = 0-value
143         return value
144 }
145
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified AODV routing agent with the implementation of the
#               multi-interface approach developed by [1] and the interface
#               switching protocol proposed by [2], which support multi-channel
#               multi-interface ad-hoc wireless network simulation
#
# File:         aodv.cc
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#
# [2] P. Kyasanur and N. H. Vaidya, "Routing and Link-layer Protocols for
#     Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#     Computing and Communications Review, vol. 10, no. 1, pp. 31-43, Jan. 2006.
#============================================================================
*/

...

// MCMI: Toggle MCMI debugging (used only when required)
//#define MCMI_DEBUG

...

int
AODV::command(int argc, const char*const* argv) {
    if(argc == 2) {
        Tcl& tcl = Tcl::instance();

        if(strncasecmp(argv[1], "id", 2) == 0) {
            tcl.resultf("%d", index);
            return TCL_OK;
        }

        if(strncasecmp(argv[1], "start", 2) == 0) {
            btimer.handle((Event*) 0);

#ifndef AODV_LINK_LAYER_DETECTION
            htimer.handle((Event*) 0);
            ntimer.handle((Event*) 0);
#endif // LINK LAYER DETECTION

            rtimer.handle((Event*) 0);
            return TCL_OK;
        }
    }

    else if(argc == 3) {
        if(strcmp(argv[1], "index") == 0) {
            index = atoi(argv[2]);
            return TCL_OK;
        }

        else if(strcmp(argv[1], "log-target") == 0 || strcmp(argv[1], "tracetarget") == 0) {
            logtarget = (Trace*) TclObject::lookup(argv[2]);
            if(logtarget == 0)
                return TCL_ERROR;
            return TCL_OK;
        }
        else if(strcmp(argv[1], "drop-target") == 0) {
            int stat = rqueue.command(argc,argv);
            if (stat != TCL_OK) return stat;
                return Agent::command(argc, argv);
```

```
72          }
73          else if(strcmp(argv[1], "if-queue") == 0) {
74              ifqueue = (PriQueue*) TclObject::lookup(argv[2]);
75
76              if(ifqueue == 0)
77                  return TCL_ERROR;
78              return TCL_OK;
79          }
80          else if (strcmp(argv[1], "port-dmux") == 0) {
81              dmux_ = (PortClassifier *)TclObject::lookup(argv[2]);
82              if (dmux_ == 0) {
83                  fprintf (stderr, "%s: %s lookup of %s failed\n", __FILE__,
84                  argv[1], argv[2]);
85                  return TCL_ERROR;
86              }
87              return TCL_OK;
88          }
89      }
90
91      // MCMI: Populate ifqueuelist and targetlist, take the corresponding values
92      //       while the interfacs are being created in the nodes.
93      else if (argc == 4) {
94          if (strcmp(argv[1], "if-queue") == 0) {
95              int temp_ = atoi(argv[2]);
96              if (temp_ == nIfaces) {
97                  nIfaces++;
98              }
99              ifqueuelist[temp_] = (PriQueue *)TclObject::lookup(argv[3]);
100             if (ifqueuelist[temp_]) {
101                 // MCMI: Initially, choose a random interface index in the range
102                 //       0 to temp_. Have to do it everytime since the node does
103                 //       not know how many channels it is going to have.
104                 fixed_interface = rand() % (temp_+1);
105 #ifdef MCMI_DEBUG
106                 printf("n%d\tcommand - fixed_interface (initial): %d\n", (int)index,
    fixed_interface);
107 #endif
108                 if (nIfaces > 1) {
109                     do {
110                         switchable_interface = rand() % (temp_+1);
111                     } while (switchable_interface == fixed_interface);
112 #ifdef MCMI_DEBUG
113                     printf("n%d\tcommand - switchable_interface (initial): %d\n",
    (int)index, switchable_interface);
114 #endif
115                 }
116
117                 // MCMI: Initialize NeighbourTable and ChannelUsageList
118                 for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
119                     neighbour_table[i] = -1;
120                     channel_usage_list[i] = 0;
121                 }
122
123                 // MCMI: Add the fixed channel used by this node to its NeighbourTable
124                 int fixed_channel = ((Mac
    *)ifqueuelist[fixed_interface]->target())->netif()->channel()->index();
125                 neighbour_table[(int)index] = fixed_channel; // fixed channel being used by
    the node itself
126                 channel_usage_list[fixed_channel]++;
127 #ifdef MCMI_DEBUG
128                 printf("n%d\tcommand - fixed_channel: %d\n", (int)index, fixed_channel);
129                 rt_print_NT_CUL();
130 #endif
131                 return TCL_OK;
132             } else {
133                 return TCL_ERROR;
134             }
135         } // if (strcmp(argv[1], "if-queue") == 0)
136         if (strcmp(argv[1], "target") == 0) {
137             int temp_ = atoi(argv[2]);
138             if (temp_ == nIfaces) {
```

```
                nIfaces++;
            }
            targetlist[temp_] = (NsObject *)TclObject::lookup(argv[3]);
            if (targetlist[temp_]) {
                return TCL_OK;
            } else {
                return TCL_ERROR;
            }
        } // if (strcmp(argv[1], "target") == 0)
    }

    return Agent::command(argc, argv);
}


/*
 * Constructor
 */
AODV::AODV(nsaddr_t id) : Agent(PT_AODV),
           btimer(this), htimer(this), ntimer(this),
           rtimer(this), lrtimer(this), rqueue() {

    index = id;
    seqno = 2;
    bid = 1;

    LIST_INIT(&nbhead);
    LIST_INIT(&bihead);

    logtarget = 0;
    ifqueue = 0;
    nIfaces = 0; // MCMI: Initialize the number of interfaces used by this node
}

...

// MCMI: Print the routing table of a node
void AODV::rt_print(nsaddr_t node_id) {
    FILE *out_file;
    char nowfile[50];
    sprintf(nowfile, "rtable_node%d.txt", (int)node_id);
    out_file = fopen(nowfile, "a");
    aodv_rt_entry *rt;
    fprintf(out_file, "----------------------------------------\n");
    for (rt = rtable.head(); rt; rt = rt->rt_link.le_next) {
        fprintf(out_file, "NODE: %i\tTIME: %.4lf\tDST: %i\tNXTHOP: %i\tHOPS: %i\tSEQ:
        %i\tEXP: %.4lf\tFLAG: n%d\tIF: %d\n",
                (int)node_id, CURRENT_TIME, rt->rt_dst, rt->rt_nexthop,
                rt->rt_hops, rt->rt_seqno, rt->rt_expire, rt->rt_flags,
                (int)rt->rt_interface);
    }
    fclose(out_file);
}


// MCMI: Print NeighbourTable and ChannelUsageList of this node
void AODV::rt_print_NT_CUL() {
    printf("n%d\t\tNT\tCUL\n", (int)index);
    for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
        printf("\tn/c%d\t%d\t%d\n", i, neighbour_table[i], channel_usage_list[i]);
    }
    printf("\n");
}

...

void
AODV::recvRequest(Packet *p) {
    struct hdr_cmn *ch = HDR_CMN(p); // MCMI: Get the common header of this packet
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
```

```
209     aodv_rt_entry *rt;
210
211     /*
212      * Drop if:
213      *      - I'm the source
214      *      - I recently heard this request.
215      */
216     if(rq->rq_src == index) {
217 #ifdef DEBUG
218         fprintf(stderr, "%s: got my own REQUEST\n", __FUNCTION__);
219 #endif // DEBUG
220         Packet::free(p);
221         return;
222     }
223     if (id_lookup(rq->rq_src, rq->rq_bcast_id)) {
224 #ifdef DEBUG
225         fprintf(stderr, "%s: discarding request\n", __FUNCTION__);
226 #endif // DEBUG
227         Packet::free(p);
228         return;
229     }
230
231     /*
232      * Cache the broadcast ID
233      */
234     id_insert(rq->rq_src, rq->rq_bcast_id);
235
236
237     if (nIfaces) {
238         // MCMI: When the node receives a "Hello" (RREQ) packet from a neighbour,
239         //       update its NeighbourTable with the fixed channel of that neighbour.
240         neighbour_table[(int)(rq->rq_sender_node_ip)] = rq->rq_fixed_channel_used;
241
242         // MCMI: Update the node's ChannelUsageList using the NeighbourTable of its
243         //       neighbour. Doing so ensures ChannelUsageList will contain two-hop
244         //       channel usage information.
245         for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
246             if (i == (int)index) {
247                 continue; // MCMI: No need to deal with the fixed channel used
248                           //       by itself again; avoid repetition
249             }
250             if (rq->rq_neighbour_table[i] != -1) {
251                 channel_usage_list[rq->rq_neighbour_table[i]]++;
252             }
253         }
254 #ifdef MCMI_DEBUG
255         printf("n%d\trecvRequest - rq->rq_sender_node_ip: %d\n", (int)index,
        rq->rq_sender_node_ip);
256         printf("n%d\trecvRequest - rq_rq->fixed_channel_used: %d\n", (int)index,
        rq->rq_fixed_channel_used);
257         rt_print_NT_CUL();
258 #endif
259     } // if (nIfaces)
260
261
262     /*
263      * We are either going to forward the REQUEST or generate a
264      * REPLY. Before we do anything, we make sure that the REVERSE
265      * route is in the route table.
266      */
267     aodv_rt_entry *rt0; // rt0 is the reverse route
268
269     rt0 = rtable.rt_lookup(rq->rq_src);
270     if(rt0 == 0) { /* if not in the route table */
271         // create an entry for the reverse route.
272         rt0 = rtable.rt_add(rq->rq_src);
273
274
275         // MCMI: Get the interface index
276         if (nIfaces) {
277 #ifdef MCMI_DEBUG
```

```
278             printf("n%d\trecvRequest - ch->iface() (rt0 = 0): %d\n", (int)index,
    ch->iface());
279             printf("n%d\trecvRequest - ((Mac *)ifqueuelist[0]->target())->addr() (rt0 = 0):
    %d\n", (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
280 #endif
281             rt0->rt_interface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr();
282         } else {
283             rt0->rt_interface = -1;
284         }
285 #ifdef MCMI_DEBUG
286         printf("n%d\trecvRequest - rt0->rt_interface (rt0 = 0): %d\n\n", (int)index,
        rt0->rt_interface);
287 #endif
288     }
289
290     rt0->rt_expire = max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE));
291
292     if ( (rq->rq_src_seqno > rt0->rt_seqno ) ||
293         ((rq->rq_src_seqno == rt0->rt_seqno) &&
294         (rq->rq_hop_count < rt0->rt_hops)) ) {
295         // If we have a fresher seq no. or lesser #hops for the
296         // same seq no., update the rt entry. Else don't bother.
297         rt_update(rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr(),
298                   max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE)) );
299
300
301         // MCMI: Get the interface index
302         if (nIfaces) {
303 #ifdef MCMI_DEBUG
304             printf("n%d\trecvRequest - ch->iface(): %d\n", (int)index, ch->iface());
305             printf("n%d\trecvRequest - ((Mac *)ifqueuelist[0]->target())->addr(): %d\n",
    (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
306 #endif
307             rt0->rt_interface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr();
308         } else {
309             rt0->rt_interface = -1;
310         }
311 #ifdef MCMI_DEBUG
312         printf("n%d\trecvRequest - rt0->rt_interface: %d\n\n", (int)index,
        rt0->rt_interface);
313 #endif
314
315
316         if (rt0->rt_req_timeout > 0.0) {
317             // Reset the soft state and
318             // Set expiry time to CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT
319             // This is because route is used in the forward direction,
320             // but only sources get benefited by this change
321             rt0->rt_req_cnt = 0;
322             rt0->rt_req_timeout = 0.0;
323             rt0->rt_req_last_ttl = rq->rq_hop_count;
324             rt0->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
325         }
326
327         /* Find out whether any buffered packet can benefit from the
328          * reverse route.
329          * May need some change in the following code - Mahesh 09/11/99
330          */
331         assert (rt0->rt_flags == RTF_UP);
332         Packet *buffered_pkt;
333         while ((buffered_pkt = rqueue.deque(rt0->rt_dst))) {
334             if (rt0 && (rt0->rt_flags == RTF_UP)) {
335                 assert(rt0->rt_hops != INFINITY2);
336                 forward(rt0, buffered_pkt, NO_DELAY);
337             }
338         }
339     }
340     // End for putting reverse route in rt table
341
342
343     /*
```

```
344      * We have taken care of the reverse route stuff.
345      * Now see whether we can send a route reply.
346      */
347     rt = rtable.rt_lookup(rq->rq_dst);
348
349     // First check if I am the destination ..
350     if(rq->rq_dst == index) {
351
352 #ifdef DEBUG
353         fprintf(stderr, "%d - %s: destination sending reply\n", index, __FUNCTION__);
354 #endif // DEBUG
355
356         // Just to be safe, I use the max. Somebody may have
357         // incremented the dst seqno.
358         seqno = max(seqno, rq->rq_dst_seqno)+1;
359         if (seqno%2) seqno++;
360
361         destination_ip = (int)index;
362         sendReply(rq->rq_src,           // IP Destination
363                   1,                    // Hop Count
364                   index,                // Dest IP Address
365                   seqno,                // Dest Sequence Num
366                   MY_ROUTE_TIMEOUT,     // Lifetime
367                   rq->rq_timestamp);    // timestamp
368
369         Packet::free(p);
370     }
371
372     // I am not the destination, but I may have a fresh enough route.
373     else if (rt && (rt->rt_hops != INFINITY2) &&
374              (rt->rt_seqno >= rq->rq_dst_seqno) ) {
375
376         //assert (rt->rt_flags == RTF_UP);
377         assert(rq->rq_dst == rt->rt_dst);
378         //assert ((rt->rt_seqno%2) == 0);    // is the seqno even?
379         sendReply(rq->rq_src,
380                   rt->rt_hops + 1,
381                   rq->rq_dst,
382                   rt->rt_seqno,
383                   (u_int32_t) (rt->rt_expire - CURRENT_TIME),
384         //        rt->rt_expire - CURRENT_TIME,
385                   rq->rq_timestamp);
386         // Insert nexthops to RREQ source and RREQ destination in the
387         // precursor lists of destination and source respectively
388         rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
389         rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination
390
391 #ifdef RREQ_GRAT_RREP
392         sendReply(rq->rq_dst,
393                   rq->rq_hop_count,
394                   rq->rq_src,
395                   rq->rq_src_seqno,
396                   (u_int32_t) (rt->rt_expire - CURRENT_TIME),
397         //        rt->rt_expire - CURRENT_TIME,
398                   rq->rq_timestamp);
399 #endif
400
401         // TODO: send grat RREP to dst if G flag set in RREQ using rq->rq_src_seqno,
          rq->rq_hop_counT
402
403         // DONE: Included gratuitous replies to be sent as per IETF aodv draft
          specification. As of now, G flag has not been dynamically used and is always set or
          reset in aodv-packet.h --- Anant Utgikar, 09/16/02.
404
405         Packet::free(p);
406     }
407
408     /*
409     * Can't reply. So forward the  Route Request
410     */
411     else {
```

```
412         ih->saddr() = index;
413         ih->daddr() = IP_BROADCAST;
414         rq->rq_hop_count += 1;
415         // Maximum sequence number seen en route
416         if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
417         forward((aodv_rt_entry*) 0, p, DELAY);
418     }
419 }
420
421
422 void
423 AODV::recvReply(Packet *p) {
424     struct hdr_cmn *ch = HDR_CMN(p); // MCMI: Get the common header of this packet
425     struct hdr_ip *ih = HDR_IP(p);
426     struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
427     aodv_rt_entry *rt;
428     char suppress_reply = 0;
429     double delay = 0.0;
430
431 #ifdef DEBUG
432     fprintf(stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
433 #endif // DEBUG
434
435
436     if (nIfaces) {
437         // MCMI: When the node receives a "Hello" (RREP) packet from a neighbour,
438         //       update its NeighbourTable with the fixed channel of that neighbour
439         neighbour_table[(int)(rp->rp_sender_node_ip)] = rp->rp_fixed_channel_used;
440
441         // MCMI: Update the node's ChannelUsageList using the NeighbourTable of its
442         //       neighbour. Doing so ensures ChannelUsageList will contain two-hop
443         //       channel usage information.
444         for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
445             if (i == (int)index) {
446                 continue; // MCMI: No need to deal with the fixed channel used
447                           //       by itself again; avoid repetition
448             }
449             if (rp->rp_neighbour_table[i] != -1) {
450             //if ((rp->rp_neighbour_table[i] != -1) &&
    !(channel_usage_list[rp->rp_neighbour_table[i]])) {
451                 channel_usage_list[rp->rp_neighbour_table[i]]++;
452             }
453         }
454 #ifdef MCMI_DEBUG
455         printf("n%d\trecvReply - rp->rp_sender_node_ip: %d\n", (int)index,
        rp->rp_sender_node_ip);
456         printf("n%d\trecvReply - rp->rp_fixed_channel_used: %d\n", (int)index,
        rp->rp_fixed_channel_used);
457         rt_print_NT_CUL();
458 #endif
459     } // if (nIfaces)
460
461
462     /*
463      *  Got a reply. So reset the "soft state" maintained for
464      *  route requests in the request table. We don't really have
465      *  have a separate request table. It is just a part of the
466      *  routing table itself.
467      */
468     // Note that rp_dst is the dest of the data packets, not the
469     // the dest of the reply, which is the src of the data packets.
470     rt = rtable.rt_lookup(rp->rp_dst);
471
472     /*
473      *  If I don't have a rt entry to this host... adding
474      */
475     if(rt == 0) {
476     rt = rtable.rt_add(rp->rp_dst);
477
478
479         // MCMI: Get the interface index
```

```
480         if (nIfaces) {
481 #ifdef MCMI_DEBUG
482             printf("n%d\trecvReply - ch->iface() (rt = 0): %d\n", (int)index, ch->iface());
483             printf("n%d\trecvReply - ((Mac *)ifqueuelist[0]->target())->addr() (rt = 0):
    %d\n", (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
484 #endif
485             rt->rt_interface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr();
486         } else {
487             rt->rt_interface = -1;
488         }
489 #ifdef MCMI_DEBUG
490         printf("n%d\trecvReply - rt->rt_interface (rt = 0): %d\n\n", (int)index,
         rt->rt_interface);
491 #endif
492     }
493
494     /*
495      * Add a forward route table entry... here I am following
496      * Perkins-Royer AODV paper almost literally - SRD 5/99
497      */
498
499     if ( (rt->rt_seqno < rp->rp_dst_seqno) ||   // newer route
500          ((rt->rt_seqno == rp->rp_dst_seqno) &&
501          (rt->rt_hops > rp->rp_hop_count)) ) { // shorter or better route
502
503         // Update the rt entry
504         rt_update(rt, rp->rp_dst_seqno, rp->rp_hop_count,
505         rp->rp_src, CURRENT_TIME + rp->rp_lifetime);
506
507
508         // MCMI: Get the interface index
509         if (nIfaces) {
510 #ifdef MCMI_DEBUG
511             printf("n%d\trecvReply - ch->iface(): %d\n", (int)index, ch->iface());
512             printf("n%d\trecvReply - ((Mac *)ifqueuelist[0]->target())->addr(): %d\n",
    (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
513 #endif
514             rt->rt_interface = ch->iface()-((Mac *)ifqueuelist[0]->target())->addr();
515         } else {
516             rt->rt_interface = -1;
517         }
518 #ifdef MCMI_DEBUG
519         printf("n%d\trecvReply - rt->rt_interface: %d\n\n", (int)index, rt->rt_interface);
520 #endif
521
522
523         // reset the soft state
524         rt->rt_req_cnt = 0;
525         rt->rt_req_timeout = 0.0;
526         rt->rt_req_last_ttl = rp->rp_hop_count;
527
528         if (ih->daddr() == index) { // If I am the original source
529             // Update the route discovery latency statistics
530             // rp->rp_timestamp is the time of request origination
531
532             rt->rt_disc_latency[(unsigned char)rt->hist_indx] = (CURRENT_TIME -
    rp->rp_timestamp)
533                                     / (double) rp->rp_hop_count;
534             // increment indx for next time
535             rt->hist_indx = (rt->hist_indx + 1) % MAX_HISTORY;
536         }
537
538         /*
539          * Send all packets queued in the sendbuffer destined for
540          * this destination.
541          * XXX - observe the "second" use of p.
542          */
543         Packet *buf_pkt;
544         while((buf_pkt = rqueue.deque(rt->rt_dst))) {
545             if(rt->rt_hops != INFINITY2) {
546                 assert (rt->rt_flags == RTF_UP);
```

```
547                 // Delay them a little to help ARP. Otherwise ARP
548                 // may drop packets. -SRD 5/23/99
549                 forward(rt, buf_pkt, delay);
550                 delay += ARP_DELAY;
551             }
552         }
553     }
554     else {
555         suppress_reply = 1;
556     }
557
558     /*
559      * If reply is for me, discard it.
560      */
561     if(ih->daddr() == index || suppress_reply) {
562         Packet::free(p);
563     }
564
565     /*
566      * Otherwise, forward the Route Reply.
567      */
568     else {
569         // Find the rt entry
570         aodv_rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
571         // If the rt is up, forward
572         if(rt0 && (rt0->rt_hops != INFINITY2)) {
573             assert (rt0->rt_flags == RTF_UP);
574             rp->rp_hop_count += 1;
575             rp->rp_src = index;
576             forward(rt0, p, NO_DELAY);
577             // Insert the nexthop towards the RREQ source to
578             // the precursor list of the RREQ destination
579             rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
580         }
581         else {
582             // I don't know how to forward .. drop the reply.
583 #ifdef DEBUG
584             fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
585 #endif // DEBUG
586             drop(p, DROP_RTR_NO_ROUTE);
587         }
588     }
589 }
590
591
592 void
593 AODV::recvError(Packet *p) {
594     struct hdr_ip *ih = HDR_IP(p);
595     struct hdr_aodv_error *re = HDR_AODV_ERROR(p);
596     aodv_rt_entry *rt;
597     u_int8_t i;
598     Packet *rerr = Packet::alloc();
599     struct hdr_aodv_error *nre = HDR_AODV_ERROR(rerr);
600
601     nre->DestCount = 0;
602
603     for (i=0; i<re->DestCount; i++) {
604         // For each unreachable destination
605         rt = rtable.rt_lookup(re->unreachable_dst[i]);
606         if ( rt && (rt->rt_hops != INFINITY2) &&
607             (rt->rt_nexthop == ih->saddr()) &&
608             (rt->rt_seqno <= re->unreachable_dst_seqno[i]) ) {
609             assert(rt->rt_flags == RTF_UP);
610             assert((rt->rt_seqno%2) == 0); // is the seqno even?
611 #ifdef DEBUG
612             fprintf(stderr, "%s(%f): n%d\t(n%d\t%u\t%d)\t(n%d\t%u\t%d)\n",
    __FUNCTION__,CURRENT_TIME,
613             index, rt->rt_dst, rt->rt_seqno, rt->rt_nexthop,
614             re->unreachable_dst[i],re->unreachable_dst_seqno[i],
615             ih->saddr());
616 #endif // DEBUG
```

```
617             rt->rt_seqno = re->unreachable_dst_seqno[i];
618             rt_down(rt);
619
620             // Not sure whether this is the right thing to do
621             Packet *pkt;
622
623
624             // MCMI: Get the corresponding interface queue
625             for (int i = 0; i < nIfaces; i++)
626                 if (rt->rt_interface == i) {
627                     ifqueue = ifqueuelist[i];
628                     break;
629                 }
630
631
632             while((pkt = ifqueue->filter(ih->saddr()))) {
633                 drop(pkt, DROP_RTR_MAC_CALLBACK);
634             }
635
636             // if precursor list non-empty add to RERR and delete the precursor list
637             if (!rt->pc_empty()) {
638                 nre->unreachable_dst[nre->DestCount] = rt->rt_dst;
639                 nre->unreachable_dst_seqno[nre->DestCount] = rt->rt_seqno;
640                 nre->DestCount += 1;
641                 rt->pc_delete();
642             }
643         }
644     }
645
646     if (nre->DestCount > 0) {
647         #ifdef DEBUG
648         fprintf(stderr, "%s(%f): n%d\t sending RERR...\n", __FUNCTION__, CURRENT_TIME,
        index);
649         #endif // DEBUG
650         sendError(rerr);
651     }
652     else {
653         Packet::free(rerr);
654     }
655
656     Packet::free(p);
657 }
658
659
660 /*
661    Packet Transmission Routines
662  */
663
664 void
665 AODV::forward(aodv_rt_entry *rt, Packet *p, double delay) {
666     struct hdr_cmn *ch = HDR_CMN(p);
667     struct hdr_ip *ih = HDR_IP(p);
668     struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);  // MCMI: RREQ packet
669     struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);      // MCMI: RREP packet
670
671     if(ih->ttl_ == 0) {
672 #ifdef DEBUG
673         fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
674 #endif // DEBUG
675
676         drop(p, DROP_RTR_TTL);
677         return;
678     }
679
680     if (ch->ptype() != PT_AODV && ch->direction() == hdr_cmn::UP
681         && ((u_int32_t)ih->daddr() == IP_BROADCAST)
682         || (ih->daddr() == here_.addr_)) {
683         dmux_->recv(p,0);
684         return;
685     }
686
```

```
687     if (rt) {
688         assert(rt->rt_flags == RTF_UP);
689         rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
690         ch->next_hop_ = rt->rt_nexthop;
691         ch->addr_type() = NS_AF_INET;
692         ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
693     }
694     else { // if it is a broadcast packet
695         // assert(ch->ptype() == PT_AODV); // maybe a diff pkt type like gaf
696         assert(ih->daddr() == (nsaddr_t) IP_BROADCAST);
697         ch->addr_type() = NS_AF_NONE;
698         ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
699     }
700
701     if (nIfaces) {
702 #ifdef MCMI_DEBUG
703         printf("n%d\tforward - rq_type: 0x%x\n", (int)index, rq->rq_type);
704         printf("n%d\tforward - rp_type: 0x%x\n", (int)index, rp->rp_type);
705 #endif
706         // MCMI: For the case of forwarding a RREQ...
707         if (rq->rq_type == AODVTYPE_RREQ) {
708             // MCMI: Add the fixed channel used by this node to the RREQ packet
709             rq->rq_fixed_channel_used = neighbour_table[(int)index];
710             rq->rq_sender_node_ip = (int)index;
711             rq->rq_neighbour_table = &neighbour_table[0];
712 #ifdef MCMI_DEBUG
713             printf("n%d\tforward - rq_fixed_channel_used: %d\n", (int)index,
    rq->rq_fixed_channel_used);
714             printf("n%d\tforward - rq_sender_node_ip: %d\n", (int)index,
    rq->rq_sender_node_ip);
715 #endif
716         }
717
718         // MCMI: For the case of forwarding a RREP...
719         if (rp->rp_type == AODVTYPE_RREP || rp->rp_type == AODVTYPE_HELLO) {
720             // MCMI: Add the fixed channel used by this node to the RREP packet
721             rp->rp_fixed_channel_used = neighbour_table[(int)index];
722             rp->rp_sender_node_ip = (int)index;
723             rp->rp_neighbour_table = &neighbour_table[0];
724 #ifdef MCMI_DEBUG
725             printf("n%d\tforward - rp_fixed_channel_used: %d\n", (int)index,
    rp->rp_fixed_channel_used);
726             printf("n%d\tforward - rp_sender_node_ip: %d\n", (int)index,
    rp->rp_sender_node_ip);
727 #endif
728         }
729
730         // MCMI: For the case of forwarding data packets...
731         if (rq->rq_type == 0 && rp->rp_type == 0) {
732
733             // MCMI: Consult the node's ChannelUsageList, find the largest usage
734             int channel_largest_usage = 0;
735             int channel_lowest_usage = 0;
736             int found = 0;
737             static int skip = 0;
738             for (int i = 0; i < nIfaces; i++) {
739                 if (channel_usage_list[i] == 0) {
740                     channel_lowest_usage = i;
741                     continue;
742                 }
743                 if (channel_usage_list[i] >= channel_usage_list[channel_largest_usage]) {
744                     channel_largest_usage = i;
745                     found = 1;
746                 }
747                 if (channel_usage_list[i] <= channel_usage_list[channel_lowest_usage]) {
748                     channel_lowest_usage = i;
749                 }
750             }
751 #ifdef MCMI_DEBUG
752             printf("n%d\tforward - channel_largest_usage: %d, #nodes: %d\n", (int)index,
    channel_largest_usage, channel_usage_list[channel_largest_usage]);
```

```
753             printf("n%d\tforward - channel_lowest_usage: %d, #nodes: %d\n", (int)index,
    channel_lowest_usage, channel_usage_list[channel_lowest_usage]);
754 #endif
755             if (found && !skip) {
756                 // MCMI: If the node's fixed channel is the one with the largest usage...
757                 if (channel_usage_list[neighbour_table[(int)index]] ==
    channel_usage_list[channel_largest_usage]) {
758                     int switch_probability = rand() % 101; // switching probability
759 #ifdef MCMI_DEBUG
760                     printf("n%d\tforward - switch_probability: %d\n", (int)index,
    switch_probability);
761 #endif
762                     if (switch_probability <= 40) {
763                         // MCMI: Reverse the ChannelUsageList about the fixed channel
    previously used by the node
764                         int fixed_channel = ((Mac
    *)ifqueuelist[fixed_interface]->target())->netif()->channel()->index
    ();
765                         channel_usage_list[fixed_channel]--;
766 
767                         // MCMI: Change the node's fixed channel to a less used channel
768                         fixed_channel = channel_lowest_usage; // new fixed channel
769                         channel_usage_list[fixed_channel]++;
770                         neighbour_table[(int)index] = fixed_channel;
771                         for (int i = 0; i < nIfaces; i++) {
772                             if (((Mac
    *)ifqueuelist[i]->target())->netif()->channel()->index() ==
    fixed_channel) {
773                                 fixed_interface = i;
774                                 break;
775                             }
776                         }
777 #ifdef MCMI_DEBUG
778                         printf("n%d\tforward - \tnew fixed_channel: %d\n", (int)index,
    fixed_channel);
779                         printf("n%d\tforward - \tnew fixed_interface: %d\n", (int)index,
    fixed_interface);
780                         rt_print_NT_CUL();
781                         printf("n%d\tforward - \tready to sendHello\n", (int)index);
782 #endif
783                         // MCMI: Transmit a Hello packet informing neighbours of its new
    fixed channel
784                         sendHello();
785                     } // if (switch_probability <= 40)
786                 } // if (channel_usage_list[neighbour_table[(int)index]] >= 2)
787             } // if (found)
788             skip = 1;
789 
790             // MCMI: Look up the fixed channel of the next node in its
791             //       NeighbourTable and assign it as the switchable channel
792             int switchable_channel = neighbour_table[(int)(rt->rt_nexthop)];
793 #ifdef MCMI_DEBUG
794             printf("n%d\tforward - back to forward\n", (int)index);
795             printf("n%d\tforward - rt->rt_nexthop: %d\n", (int)index,
    (int)(rt->rt_nexthop));
796             printf("n%d\tforward - switchable_channel: %d\n", (int)index,
    switchable_channel);
797 #endif
798             // MCMI: Given the switchable channel, find the corresponding interface
799             for (int i = 0; i < MAX_IF; i++) {
800                 if ((((Mac *)ifqueuelist[i]->target())->netif()->channel()->index()) ==
    switchable_channel) {
801                     switchable_interface = i;
802                     break;
803                 }
804             }
805 #ifdef MCMI_DEBUG
806             printf("n%d\tforward - switchable_interface: %d\n", (int)index,
    switchable_interface);
807 #endif
808         } // if (rq->rq_type == 0 && rp->rp_type == 0)
```

```
809     } // if (nIfaces)
810
811
812     if (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
813         // If it is a broadcast packet
814         assert(rt == 0);
815         /*
816          *  Jitter the sending of broadcast packets by 10ms
817          */
818         if (nIfaces) { // MCMI: Send a broadcast packet
819             for (int i = 0; i < nIfaces; i++) {
820                 Packet *p_copy = p->copy();
821                 Scheduler::instance().schedule(targetlist[i], p_copy, 0.01 *
    Random::uniform());
822 #ifdef MCMI_DEBUG
823                 printf("n%d\tforward - broadcast\n\n", (int)index);
824 #endif
825             }
826             Packet::free((Packet *)p);
827         }
828         else {
829             Scheduler::instance().schedule(target_, p, 0.01 * Random::uniform());
830         }
831     }
832     else { // Not a broadcast packet
833         if(delay > 0.0) {
834             if (nIfaces) {
835                 if (rq->rq_type == 0 && rp->rp_type == 0) {     // data packet
836 #ifdef MCMI_DEBUG
837                     printf("n%d\tforward - switchable_interface (delay > 0): %d\n\n",
    (int)index, switchable_interface);
838 #endif
839                     Scheduler::instance().schedule(targetlist[switchable_interface], p,
    delay);
840                 } else {
841                     // MCMI
842 #ifdef MCMI_DEBUG
843                     printf("n%d\tforward - rt->rt_interface (delay > 0): %d\n\n",
    (int)index, rt->rt_interface);
844 #endif
845                     Scheduler::instance().schedule(targetlist[rt->rt_interface], p, delay);
846                 }
847             } else {
848                 Scheduler::instance().schedule(target_, p, delay);
849             }
850         }
851         else {
852             // Not a broadcast packet, no delay, send immediately
853             if (nIfaces) {  // MCMI: Send a unicast packet
854                 if (rq->rq_type == 0 && rp->rp_type == 0) {     // data packet
855 #ifdef MCMI_DEBUG
856                     printf("n%d\tforward - switchable_interface (else): %d\n\n", (int)index,
    switchable_interface);
857 #endif
858                     Scheduler::instance().schedule(targetlist[switchable_interface], p, 0);
859                 } else {
860 #ifdef MCMI_DEBUG
861                     printf("n%d\tforward - rt->rt_interface (else): %d\n\n", (int)index,
    rt->rt_interface);
862 #endif
863                     Scheduler::instance().schedule(targetlist[rt->rt_interface], p, 0);
864                 }
865             } else {
866                 Scheduler::instance().schedule(target_, p, 0);
867             }
868         }
869     }
870 }
871
872
873 void
```

```
874 AODV::sendRequest(nsaddr_t dst) {
875     // Allocate a RREQ packet
876     Packet *p = Packet::alloc();
877     struct hdr_cmn *ch = HDR_CMN(p);
878     struct hdr_ip *ih = HDR_IP(p);
879     struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
880     aodv_rt_entry *rt = rtable.rt_lookup(dst);
881 
882     assert(rt);
883 
884     /*
885      *  Rate limit sending of Route Requests. We are very conservative
886      *  about sending out route requests.
887      */
888 
889     if (rt->rt_flags == RTF_UP) {
890         assert(rt->rt_hops != INFINITY2);
891         Packet::free((Packet *)p);
892         return;
893     }
894 
895     if (rt->rt_req_timeout > CURRENT_TIME) {
896         Packet::free((Packet *)p);
897         return;
898     }
899 
900     // rt_req_cnt is the no. of times we did network-wide broadcast
901     // RREQ_RETRIES is the maximum number we will allow before
902     // going to a long timeout.
903 
904     if (rt->rt_req_cnt > RREQ_RETRIES) {
905         rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
906         rt->rt_req_cnt = 0;
907         Packet *buf_pkt;
908         while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
909             drop(buf_pkt, DROP_RTR_NO_ROUTE);
910         }
911         Packet::free((Packet *)p);
912         return;
913     }
914 
915 #ifdef DEBUG
916     fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
917                 ++route_request, index, rt->rt_dst);
918 #endif // DEBUG
919 
920     // Determine the TTL to be used this time.
921     // Dynamic TTL evaluation - SRD
922 
923     rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);
924 
925     if (0 == rt->rt_req_last_ttl) {
926         // first time query broadcast
927         ih->ttl_ = TTL_START;
928     }
929     else {
930         // Expanding ring search.
931         if (rt->rt_req_last_ttl < TTL_THRESHOLD)
932             ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
933         else {
934             // network-wide broadcast
935             ih->ttl_ = NETWORK_DIAMETER;
936             rt->rt_req_cnt += 1;
937         }
938     }
939 
940     // remember the TTL used  for the next time
941     rt->rt_req_last_ttl = ih->ttl_;
942 
943     // PerHopTime is the roundtrip time per hop for route requests.
944     // The factor 2.0 is just to be safe .. SRD 5/22/99
```

```
945     // Also note that we are making timeouts to be larger if we have
946     // done network wide broadcast before.
947
948     rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
949     if (rt->rt_req_cnt > 0)
950     rt->rt_req_timeout *= rt->rt_req_cnt;
951     rt->rt_req_timeout += CURRENT_TIME;
952
953     // Don't let the timeout to be too large, however .. SRD 6/8/99
954     if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
955     rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
956     rt->rt_expire = 0;
957
958 #ifdef DEBUG
959     fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
960             ++route_request,
961             index, rt->rt_dst,
962             rt->rt_req_timeout - CURRENT_TIME);
963 #endif  // DEBUG
964
965
966     // Fill out the RREQ packet
967     // ch->uid() = 0;
968     ch->ptype() = PT_AODV;
969     ch->size() = IP_HDR_LEN + rq->size();
970     ch->iface() = -2;
971     ch->error() = 0;
972     ch->addr_type() = NS_AF_NONE;
973     ch->prev_hop_ = index;          // AODV hack
974
975     ih->saddr() = index;
976     ih->daddr() = IP_BROADCAST;
977     ih->sport() = RT_PORT;
978     ih->dport() = RT_PORT;
979
980     // Fill up some more fields.
981     rq->rq_type = AODVTYPE_RREQ;
982     rq->rq_hop_count = 1;
983     rq->rq_bcast_id = bid++;
984     rq->rq_dst = dst;
985     rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
986     rq->rq_src = index;
987     seqno += 2;
988     assert ((seqno%2) == 0);
989     rq->rq_src_seqno = seqno;
990     rq->rq_timestamp = CURRENT_TIME;
991
992
993     // MCMI: Add the fixed channel used by this node to the packet
994     rq->rq_fixed_channel_used = neighbour_table[(int)index];
995     rq->rq_sender_node_ip = (int)index;
996     rq->rq_neighbour_table = &neighbour_table[0];
997 #ifdef MCMI_DEBUG
998     printf("n%d\tsendRequest - rq_fixed_channel_used: %d\n", (int)index,
    rq->rq_fixed_channel_used);
999     printf("n%d\tsendRequest - rq_sender_node_ip: %d\n", (int)index, rq->rq_sender_node_ip);
1000 #endif
1001
1002     // MCMI: Send a broadcast packet
1003     if (nIfaces) {
1004         for (int i = 0; i < nIfaces; i++) {
1005             Packet *p_copy = p->copy();
1006             Scheduler::instance().schedule(targetlist[i], p_copy, 0.);
1007 #ifdef MCMI_DEBUG
1008             printf("n%d\tsendRequest - broadcast\n\n", (int)index);
1009 #endif
1010         }
1011         Packet::free((Packet *)p);
1012     }
1013     else {
1014         Scheduler::instance().schedule(target_, p, 0.);
```

```
1015        }
1016    }
1017
1018
1019    void
1020    AODV::sendReply(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
1021                    u_int32_t rpseq, u_int32_t lifetime, double timestamp) {
1022        Packet *p = Packet::alloc();
1023        struct hdr_cmn *ch = HDR_CMN(p);
1024        struct hdr_ip *ih = HDR_IP(p);
1025        struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
1026        aodv_rt_entry *rt = rtable.rt_lookup(ipdst);
1027
1028    #ifdef DEBUG
1029        fprintf(stderr, "sending Reply from %d at %.2f\n", index,
            Scheduler::instance().clock());
1030    #endif // DEBUG
1031        assert(rt);
1032
1033        rp->rp_type = AODVTYPE_RREP;
1034        //rp->rp_flags = 0x00;
1035        rp->rp_hop_count = hop_count;
1036        rp->rp_dst = rpdst;
1037        rp->rp_dst_seqno = rpseq;
1038        rp->rp_src = index;
1039        rp->rp_lifetime = lifetime;
1040        rp->rp_timestamp = timestamp;
1041
1042        // ch->uid() = 0;
1043        ch->ptype() = PT_AODV;
1044        ch->size() = IP_HDR_LEN + rp->size();
1045        ch->iface() = -2;
1046        ch->error() = 0;
1047        ch->addr_type() = NS_AF_INET;
1048        ch->next_hop_ = rt->rt_nexthop;
1049        ch->prev_hop_ = index;      // AODV hack
1050        ch->direction() = hdr_cmn::DOWN;
1051
1052        ih->saddr() = index;
1053        ih->daddr() = ipdst;
1054        ih->sport() = RT_PORT;
1055        ih->dport() = RT_PORT;
1056        ih->ttl_ = NETWORK_DIAMETER;
1057
1058
1059        // MCMI: Add the fixed channel used by this node to the packet
1060        rp->rp_fixed_channel_used = neighbour_table[(int)index];
1061        rp->rp_sender_node_ip = (int)index;
1062        rp->rp_neighbour_table = &neighbour_table[0];
1063    #ifdef MCMI_DEBUG
1064        printf("n%d\tsendReply - rp_fixed_channel_used: %d\n", (int)index,
            rp->rp_fixed_channel_used);
1065        printf("n%d\tsendReply - rp_sender_node_ip: %d\n", (int)index, rp->rp_sender_node_ip);
1066    #endif
1067
1068        // MCMI: Send a unicast packet
1069        if (nIfaces) {
1070    #ifdef MCMI_DEBUG
1071            printf("n%d\tsendReply - rt->rt_interface (unicast): %d\n\n", (int)index,
                rt->rt_interface);
1072    #endif
1073            Scheduler::instance().schedule(targetlist[rt->rt_interface], p, 0.);
1074        }
1075        else {
1076            Scheduler::instance().schedule(target_, p, 0.);
1077        }
1078    }
1079
1080
1081    void
1082    AODV::sendError(Packet *p, bool jitter) {
```

```
1083      struct hdr_cmn *ch = HDR_CMN(p);
1084      struct hdr_ip *ih = HDR_IP(p);
1085      struct hdr_aodv_error *re = HDR_AODV_ERROR(p);
1086
1087  #ifdef ERROR
1088      fprintf(stderr, "sending Error from %d at %.2f\n", index,
          Scheduler::instance().clock());
1089  #endif // DEBUG
1090
1091      re->re_type = AODVTYPE_RERR;
1092      //re->reserved[0] = 0x00; re->reserved[1] = 0x00;
1093      // DestCount and list of unreachable destinations are already filled
1094
1095      // ch->uid() = 0;
1096      ch->ptype() = PT_AODV;
1097      ch->size() = IP_HDR_LEN + re->size();
1098      ch->iface() = -2;
1099      ch->error() = 0;
1100      ch->addr_type() = NS_AF_NONE;
1101      ch->next_hop_ = 0;
1102      ch->prev_hop_ = index;              // AODV hack
1103      ch->direction() = hdr_cmn::DOWN;    // important: change the packet's direction
1104
1105      ih->saddr() = index;
1106      ih->daddr() = IP_BROADCAST;
1107      ih->sport() = RT_PORT;
1108      ih->dport() = RT_PORT;
1109      ih->ttl_ = 1;
1110
1111      // Do we need any jitter? Yes
1112      if (jitter) {
1113          // MCMI: Send a broadcast packet
1114          if (nIfaces) {
1115              for (int i = 0; i < nIfaces; i++) {
1116                  Packet *p_copy = p->copy();
1117                  Scheduler::instance().schedule(targetlist[i], p_copy,
      0.01*Random::uniform());
1118  #ifdef MCMI_DEBUG
1119                  printf("n%d\tsendError - broadcast (jitter)\n\n", (int)index);
1120  #endif
1121              }
1122              Packet::free((Packet *)p);
1123          }
1124          else {
1125              Scheduler::instance().schedule(target_, p, 0.01*Random::uniform());
1126          }
1127      }
1128      else {
1129          // MCMI: Send a broadcast packet
1130          if (nIfaces) {
1131              for (int i = 0; i < nIfaces; i++) {
1132                  Packet *p_copy = p->copy();
1133                  Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
1134  #ifdef MCMI_DEBUG
1135                  printf("n%d\tsendError - broadcast (else)\n\n", (int)index);
1136  #endif
1137              }
1138              Packet::free((Packet *)p);
1139          }
1140          else {
1141              Scheduler::instance().schedule(target_, p, 0.0);
1142          }
1143      }
1144  }
1145
1146
1147  /*
1148     Neighbor Management Functions
1149   */
1150
1151  void
```

```
1152 AODV::sendHello() {
1153     Packet *p = Packet::alloc();
1154     struct hdr_cmn *ch = HDR_CMN(p);
1155     struct hdr_ip *ih = HDR_IP(p);
1156     struct hdr_aodv_reply *rh = HDR_AODV_REPLY(p);
1157 
1158 #ifdef DEBUG
1159     fprintf(stderr, "sending Hello from %d at %.2f\n", index,
        Scheduler::instance().clock());
1160 #endif // DEBUG
1161 
1162     rh->rp_type = AODVTYPE_HELLO;
1163     //rh->rp_flags = 0x00;
1164     rh->rp_hop_count = 1;
1165     rh->rp_dst = index;
1166     rh->rp_dst_seqno = seqno;
1167     rh->rp_lifetime = (1 + ALLOWED_HELLO_LOSS) * HELLO_INTERVAL;
1168 
1169     // ch->uid() = 0;
1170     ch->ptype() = PT_AODV;
1171     ch->size() = IP_HDR_LEN + rh->size();
1172     ch->iface() = -2;
1173     ch->error() = 0;
1174     ch->addr_type() = NS_AF_NONE;
1175     ch->prev_hop_ = index;      // AODV hack
1176 
1177     ih->saddr() = index;
1178     ih->daddr() = IP_BROADCAST;
1179     ih->sport() = RT_PORT;
1180     ih->dport() = RT_PORT;
1181     ih->ttl_ = 1;
1182 
1183 
1184     // MCMI: Send a broadcast packet
1185     if (nIfaces) {
1186         // MCMI: In order for the node to inform its neighbours of its new fixed
1187         //       channel, add the fixed channel used by this node to the packet
1188         rh->rp_fixed_channel_used = neighbour_table[(int)index];
1189         rh->rp_sender_node_ip = (int)index;
1190         rh->rp_neighbour_table = &neighbour_table[0];
1191 #ifdef MCMI_DEBUG
1192         printf("n%d\tsendHello - rh_fixed_channel_used: %d\n", (int)index,
        rh->rp_fixed_channel_used);
1193         printf("n%d\tsendHello - rh_sender_node_ip: %d\n", (int)index,
        rh->rp_sender_node_ip);
1194 #endif
1195 
1196         // MCMI: Send a broadcast packet
1197         for (int i = 0; i < nIfaces; i++) {
1198             Packet *p_copy = p->copy();
1199             Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
1200 #ifdef MCMI_DEBUG
1201             printf("n%d\tsendHello - broadcast\n\n", (int)index);
1202 #endif
1203         }
1204         Packet::free((Packet *)p);
1205     }
1206     else {
1207         Scheduler::instance().schedule(target_, p, 0.0);
1208     }
1209 }
1210 
1211 
1212 void
1213 AODV::recvHello(Packet *p) {
1214     //struct hdr_ip *ih = HDR_IP(p);
1215     struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
1216     AODV_Neighbor *nb;
1217 
1218     /*
1219      * Whenever a node receives a Hello message from a neighbor, the node
```

```
1220        * SHOULD make sure that it has an active route to the neighbor, and
1221        * create one if necessary.  If a route already exists, then the
1222        * Lifetime for the route should be increased, if necessary, to be at
1223        * least ALLOWED_HELLO_LOSS * HELLO_INTERVAL.
1224        */
1225       nb = nb_lookup(rp->rp_dst);
1226       if(nb == 0) {
1227           nb_insert(rp->rp_dst);
1228       }
1229       else {
1230           nb->nb_expire = CURRENT_TIME + (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
1231       }
1232
1233
1234       if (nIfaces) {
1235           // MCMI: Update the node's ChannelUsageList using the NeighbourTable of
1236           //       its neighbour. Doing so ensures ChannelUsageList will contain
1237           //       two-hop channel usage information.
1238           for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
1239               // MCMI: No need to deal with the fixed channel used by itself again
1240               if ((rp->rp_neighbour_table[i] == -1) || i == (int)index
1241                    || (neighbour_table[i] == rp->rp_neighbour_table[i])) {
1242                   continue; // avoid repetition
1243
1244               // MCMI: If the fixed channel used by a neighbour is different
1245               //       from the incoming update, the ChannelUsageList needs to
1246               //       be reflected appropriately to handle the update
1247               } else if (neighbour_table[i] != rp->rp_neighbour_table[i]) {
1248                   if (channel_usage_list[neighbour_table[i]] > 0) {
1249                       channel_usage_list[neighbour_table[i]]--;
1250                   }
1251                   channel_usage_list[rp->rp_neighbour_table[i]]++;
1252
1253               // MCMI: Normally, just update the ChannelUsageList accordingly
1254               } else {
1255                   channel_usage_list[rp->rp_neighbour_table[i]]++;
1256               }
1257           }
1258           // MCMI: When the node receives a "Hello" packet from a neighbour,
1259           //       update its NeighbourTable with the fixed channel of that neighbour.
1260           neighbour_table[(int)(rp->rp_sender_node_ip)] = rp->rp_fixed_channel_used;
1261
1262   #ifdef MCMI_DEBUG
1263           printf("n%d\trecvHello - rp->rp_sender_node_ip: %d\n", (int)index,
           rp->rp_sender_node_ip);
1264           printf("n%d\trecvHello - rp->rp_fixed_channel_used: %d\n", (int)index,
           rp->rp_fixed_channel_used);
1265           rt_print_NT_CUL();
1266   #endif
1267       } // if (nIfaces)
1268
1269       Packet::free(p);
1270   }
1271
1272   ...
1273
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified AODV routing agent with the implementation of the
#               multi-interface approach developed by [1] and the interface
#               switching protocol proposed by [2], which support multi-channel
#               multi-interface ad-hoc wireless network simulation
#
# File:         aodv.h
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#
# [2] P. Kyasanur and N. H. Vaidya, "Routing and Link-layer Protocols for
#     Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#     Computing and Communications Review, vol. 10, no. 1, pp. 31-43, Jan. 2006.
#============================================================================
*/


...

// MCMI: Define various constants
#define MAX_IF               12 // maximum number of interfaces per node
#define MAX_NT_CUL_ENTRIES   20 // maximum number of entries in NeighbourTable
                                // and ChannelUsageList

...

/*
  The Routing Agent
*/
class AODV: public Agent {

        /*
         * make some friends first
         */
        friend class aodv_rt_entry;
        friend class BroadcastTimer;
        friend class HelloTimer;
        friend class NeighborTimer;
        friend class RouteCacheTimer;
        friend class LocalRepairTimer;

 public:
        AODV(nsaddr_t id);
        void            recv(Packet *p, Handler *);


        // MCMI: Keeps track of the number of interfaces that the agent is managing
        int             nIfaces;


 protected:
        int             command(int, const char *const *);
        int             initialized() { return 1 && target_; }

        /*
         * Route Table Management
         */
        void            rt_resolve(Packet *p);
        void            rt_update(aodv_rt_entry *rt, u_int32_t seqnum,
                                  u_int16_t metric, nsaddr_t nexthop,
                                  double expire_time);
        void            rt_down(aodv_rt_entry *rt);
```

```
72
73
74          // MCMI: Print the routing table of this node
75          void            rt_print(nsaddr_t node_id);
76
77          // MCMI: Print NeighbourTable and ChannelUsageList of this node
78          void            rt_print_NT_CUL();
79
80
81          void            local_rt_repair(aodv_rt_entry *rt, Packet *p);
82    public:
83          void            rt_ll_failed(Packet *p);
84          void            handle_link_failure(nsaddr_t id);
85    protected:
86          void            rt_purge(void);
87
88          void            enque(aodv_rt_entry *rt, Packet *p);
89          Packet*         deque(aodv_rt_entry *rt);
90
91          /*
92           * Neighbor Management
93           */
94          void            nb_insert(nsaddr_t id);
95          AODV_Neighbor*  nb_lookup(nsaddr_t id);
96          void            nb_delete(nsaddr_t id);
97          void            nb_purge(void);
98
99          /*
100          * Broadcast ID Management
101          */
102
103         void            id_insert(nsaddr_t id, u_int32_t bid);
104         bool            id_lookup(nsaddr_t id, u_int32_t bid);
105         void            id_purge(void);
106
107         /*
108          * Packet TX Routines
109          */
110         void            forward(aodv_rt_entry *rt, Packet *p, double delay);
111         void            sendHello(void);
112         void            sendRequest(nsaddr_t dst);
113
114         void            sendReply(nsaddr_t ipdst, u_int32_t hop_count,
115                                   nsaddr_t rpdst, u_int32_t rpseq,
116                                   u_int32_t lifetime, double timestamp);
117         void            sendError(Packet *p, bool jitter = true);
118
119         /*
120          * Packet RX Routines
121          */
122         void            recvAODV(Packet *p);
123         void            recvHello(Packet *p);
124         void            recvRequest(Packet *p);
125         void            recvReply(Packet *p);
126         void            recvError(Packet *p);
127
128         /*
129          * History management
130          */
131
132         double      PerHopTime(aodv_rt_entry *rt);
133
134
135         nsaddr_t        index;                  // IP Address of this node
136         u_int32_t       seqno;                  // Sequence Number
137         int             bid;                    // Broadcast ID
138
139         aodv_rtable     rthead;                 // routing table
140         aodv_ncache     nbhead;                 // Neighbor Cache
141         aodv_bcache     bihead;                 // Broadcast ID Cache
142
```

```
143         /*
144          * Timers
145          */
146         BroadcastTimer  btimer;
147         HelloTimer      htimer;
148         NeighborTimer   ntimer;
149         RouteCacheTimer rtimer;
150         LocalRepairTimer lrtimer;
151
152         /*
153          * Routing Table
154          */
155         aodv_rtable     rtable;
156         /*
157          *  A "drop-front" queue used by the routing layer to buffer
158          *  packets to which it does not have a route.
159          */
160         aodv_rqueue     rqueue;
161
162         /*
163          * A mechanism for logging the contents of the routing
164          * table.
165          */
166         Trace           *logtarget;
167
168         /*
169          * A pointer to the network interface queue that sits
170          * between the "classifier" and the "link layer".
171          */
172         PriQueue        *ifqueue;
173
174
175         // MCMI: The routing agent needs to decide which one of the interfaces
176         //       the outgoing packets should be routed to. Since there are now
177         //       multiple interfaces. The originally used ifqueue and target
178         //       pointers need to be modified as follows:
179
180         // MCMI: Store the LL modules for all the interfaces a node has
181         NsObject        *targetlist[MAX_IF];
182
183         // MCMI: Keep the corresponding queues of all the interfaces
184         PriQueue        *ifqueuelist[MAX_IF];
185
186         // MCMI: Contain the fixed channels used by the node's neighbours
187         int neighbour_table[MAX_NT_CUL_ENTRIES]; // index by node IP
188
189         // MCMI: Contain the number of nodes using each channel as their fixed channel
190         int channel_usage_list[MAX_NT_CUL_ENTRIES]; // index by channel ID
191
192         // MCMI: Fixed and switchable interfaces used by this node
193         int fixed_interface;
194         int switchable_interface;
195
196
197         /*
198          * Logging stuff
199          */
200         void            log_link_del(nsaddr_t dst);
201         void            log_link_broke(Packet *p);
202         void            log_link_kept(nsaddr_t dst);
203
204         /* for passing packets up to agents */
205         PortClassifier *dmux_;
206
207 };
208
209 #endif /* __aodv_h__ */
210
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:     Chih-Hao Howard Chang
#              20007-2192
#              howardc@sfu.ca
#
# Description: Modified AODV routing agent with the implementation of the
#              multi-interface approach developed by [1] and the interface
#              switching protocol proposed by [2], which support multi-channel
#              multi-interface ad-hoc wireless network simulation
#
# File:        aodv_packet.h
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#
# [2] P. Kyasanur and N. H. Vaidya, "Routing and Link-layer Protocols for
#     Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#     Computing and Communications Review, vol. 10, no. 1, pp. 31-43, Jan. 2006.
#============================================================================
*/


...

struct hdr_aodv_request {
        u_int8_t        rq_type;    // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rq_hop_count;   // Hop Count
        u_int32_t       rq_bcast_id;    // Broadcast ID

        nsaddr_t        rq_dst;         // Destination IP Address
        u_int32_t       rq_dst_seqno;   // Destination Sequence Number
        nsaddr_t        rq_src;         // Source IP Address
        u_int32_t       rq_src_seqno;   // Source Sequence Number

        double          rq_timestamp;   // when REQUEST sent;
                    // used to compute route discovery latency


        // MCMI
        int             rq_fixed_channel_used;  // fixed channel used by this node
        nsaddr_t        rq_sender_node_ip;      // node IP of the RREQ sender (not
        originator)
        int             *rq_neighbour_table;    // NeighbourTable of this node


  // This define turns on gratuitous replies- see aodv.cc for implementation contributed by
  // Anant Utgikar, 09/16/02.
  //#define RREQ_GRAT_RREP  0x80

  inline int size() {
  int sz = 0;
  /*
    sz = sizeof(u_int8_t)       // rq_type
         + 2*sizeof(u_int8_t)   // reserved
         + sizeof(u_int8_t)     // rq_hop_count
         + sizeof(double)       // rq_timestamp
         + sizeof(u_int32_t)    // rq_bcast_id
         + sizeof(nsaddr_t)     // rq_dst
         + sizeof(u_int32_t)    // rq_dst_seqno
         + sizeof(nsaddr_t)     // rq_src
         + sizeof(u_int32_t);   // rq_src_seqno
  */
    sz = 7*sizeof(u_int32_t);
    assert (sz >= 0);
    return sz;
  }
```

```
};

struct hdr_aodv_reply {
        u_int8_t        rp_type;        // Packet Type
        u_int8_t        reserved[2];
        u_int8_t        rp_hop_count;           // Hop Count
        nsaddr_t        rp_dst;                 // Destination IP Address
        u_int32_t       rp_dst_seqno;           // Destination Sequence Number
        nsaddr_t        rp_src;                 // Source IP Address
        double          rp_lifetime;            // Lifetime

        double          rp_timestamp;           // when corresponding REQ sent;
                        // used to compute route discovery latency


        // MCMI
        int             rp_fixed_channel_used;  // fixed channel used by this node
        nsaddr_t        rp_sender_node_ip;      // node IP of the RREP sender (not
        originator)
        int             *rp_neighbour_table;    // NeighbourTable of this node


  inline int size() {
  int sz = 0;
  /*
    sz = sizeof(u_int8_t)       // rp_type
         + 2*sizeof(u_int8_t)   // rp_flags + reserved
         + sizeof(u_int8_t)     // rp_hop_count
         + sizeof(double)       // rp_timestamp
         + sizeof(nsaddr_t)     // rp_dst
         + sizeof(u_int32_t)    // rp_dst_seqno
         + sizeof(nsaddr_t)     // rp_src
         + sizeof(u_int32_t);   // rp_lifetime
  */
    sz = 6*sizeof(u_int32_t);
    assert (sz >= 0);
    return sz;
  }

};

...
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:     Chih-Hao Howard Chang
#              20007-2192
#              howardc@sfu.ca
#
# Description: Modified AODV routing agent with the implementation of the
#              multi-interface approach developed by [1] and the interface
#              switching protocol proposed by [2], which support multi-channel
#              multi-interface ad-hoc wireless network simulation
#
# File:        aodv_rtable.h
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#
# [2] P. Kyasanur and N. H. Vaidya, "Routing and Link-layer Protocols for
#     Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#     Computing and Communications Review, vol. 10, no. 1, pp. 31-43, Jan. 2006.
#============================================================================
*/


...

/*
  Route Table Entry
*/

class aodv_rt_entry {
    friend class aodv_rtable;
    friend class AODV;
    friend class LocalRepairTimer;
public:
    aodv_rt_entry();
    ~aodv_rt_entry();

    void            nb_insert(nsaddr_t id);
    AODV_Neighbor*  nb_lookup(nsaddr_t id);

    void            pc_insert(nsaddr_t id);
    AODV_Precursor* pc_lookup(nsaddr_t id);
    void            pc_delete(nsaddr_t id);
    void            pc_delete(void);
    bool            pc_empty(void);

    double          rt_req_timeout;         // when I can send another req
    u_int8_t        rt_req_cnt;             // number of route requests

 protected:
    LIST_ENTRY(aodv_rt_entry) rt_link;

    nsaddr_t        rt_dst;
    u_int32_t       rt_seqno;
    /* u_int8_t     rt_interface; */
    u_int16_t       rt_hops;            // hop count
    int             rt_last_hop_count;  // last valid hop count
    nsaddr_t        rt_nexthop;         // next hop IP address


    // MCMI: For routing the packet via a specific interface to a destination,
    //       unicast transmission of AODV is exercised. Just knowing the next
    //       hop is not enough; the routing agent must also consider which output
    //       interface to be used to reach the next hop.
    int             rt_interface; // interface index for outgoing route packets


    /* list of precursors */
```

```
72      aodv_precursors rt_pclist;
73      double          rt_expire;          // when entry expires
74      u_int8_t        rt_flags;
75
76  #define RTF_DOWN 0
77  #define RTF_UP 1
78  #define RTF_IN_REPAIR 2
79
80      /*
81       *  Must receive 4 errors within 3 seconds in order to mark
82       *  the route down.
83      u_int8_t        rt_errors;      // error count
84      double          rt_error_time;
85  #define MAX_RT_ERROR            4       // errors
86  #define MAX_RT_ERROR_TIME       3       // seconds
87      */
88
89  #define MAX_HISTORY 3
90      double          rt_disc_latency[MAX_HISTORY];
91      char            hist_indx;
92      int             rt_req_last_ttl;        // last ttl value used
93      // last few route discovery latencies
94      // double       rt_length [MAX_HISTORY];
95      // last few route lengths
96
97      /*
98       * a list of neighbors that are using this route.
99       */
100     aodv_ncache          rt_nblist;
101 };
102
103 ...
104
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified MobileNode class with the implementation of the
#               multi-interface approach developed by [1]
#
# File:         mobilenode.cc
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#============================================================================
*/


...

// MCMI: Retrieve the location of a node
void
MobileNode::getLoc(double *x, double *y, double *z) {
    update_position();
    *x = X_;
    *y = Y_;
    *z = Z_;
}

```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified MobileNode class with the implementation of the
#               multi-interface approach developed by [1]
#
# File:         mobilenode.h
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#============================================================================
*/


...

#define MAX_CHANNELS        12 // MCMI: Maximum number of channels per node

...

class MobileNode : public Node
{
    friend class PositionHandler;
public:
    MobileNode();
    virtual int command(int argc, const char*const* argv);

    double  distance(MobileNode*);
    double  propdelay(MobileNode*);
    void    start(void);
    void getLoc(double *x, double *y, double *z);   // MCMI

    // MCMI: Remove the inline declaration of the getLoc() function. Due to the
    //       above changes on the MobileNode lists, the original declaration has been found to always
    //       return a zero distance, which leads to wrong packet receptions.
//  inline void getLoc(double *x, double *y, double *z) {
//      update_position();  *x = X_; *y = Y_; *z = Z_;
//  }
        inline void getVelo(double *dx, double *dy, double *dz) {
        *dx = dX_ * speed_; *dy = dY_ * speed_; *dz = 0.0;
    }
    inline MobileNode* nextnode() { return link_.le_next; }
    inline int base_stn() { return base_stn_;}
    inline void set_base_stn(int addr) { base_stn_ = addr; }

    void dump(void);

    inline MobileNode*& next() { return next_; }
    inline double X() { return X_; }
    inline double Y() { return Y_; }
    inline double Z() { return Z_; }
    inline double speed() { return speed_; }
    inline double dX() { return dX_; }
    inline double dY() { return dY_; }
    inline double dZ() { return dZ_; }
    inline double destX() { return destX_; }
    inline double destY() { return destY_; }
    inline double radius() { return radius_; }
    inline double getUpdateTime() { return position_update_time_; }
    //inline double last_routingtime() { return last_rt_time_;}

    void update_position();
    void log_energy(int);
    //void logrttime(double);
```

```
71      virtual void idle_energy_patch(float, float);
72
73      /* For list-keeper */
74  //  MobileNode* nextX_;
75  //  MobileNode* prevX_;
76      // MCMI: We define a new declaration of the MobileNode lists to replace the
77      //       existing ones. ns-2 controls each instance of the MobileNode objects
78      //       which are associated with a channel by means of a linked-list. Two
79      //       lists are managed; one references the previous node, prevX_, while
80      //       the other references the next node, nextX_. The original format of
81      //       the list is simply a pointer to a node. In order to support multiple
82      //       channels, the list is modified to be an array of pointers with the
83      //       size of the array being the maximum of number of channels:
84      MobileNode* nextX_[MAX_CHANNELS];
85      MobileNode* prevX_[MAX_CHANNELS];
86
87  protected:
88      /*
89       * Last time the position of this node was updated.
90       */
91      double position_update_time_;
92          double position_update_interval_;
93
94      /*
95           *  The following indicate the (x,y,z) position of the node on
96           *  the "terrain" of the simulation.
97           */
98      double X_;
99      double Y_;
100     double Z_;
101     double speed_;  // meters per second
102
103     /*
104          *  The following is a unit vector that specifies the
105          *  direction of the mobile node.  It is used to update
106          *  position
107          */
108     double dX_;
109     double dY_;
110     double dZ_;
111
112         /* where are we going? */
113     double destX_;
114     double destY_;
115
116     /*
117      * for gridkeeper use only
118      */
119     MobileNode* next_;
120     double          radius_;
121
122     // Used to generate position updates
123     PositionHandler pos_handle_;
124     Event pos_intr_;
125
126     void    log_movement();
127     void    random_direction();
128     void    random_speed();
129         void    random_destination();
130         int set_destination(double x, double y, double speed);
131
132 private:
133     inline int initialized() {
134         return (T_ && log_target_ &&
135             X_ >= T_->lowerX() && X_ <= T_->upperX() &&
136             Y_ >= T_->lowerY() && Y_ <= T_->upperY());
137     }
138     void        random_position();
139     void        bound_position();
140     int     random_motion_; // is mobile
141
```

```
142     /*
143      * A global list of mobile nodes
144      */
145     LIST_ENTRY(MobileNode) link_;
146
147
148     /*
149      * The topography over which the mobile node moves.
150      */
151     Topography *T_;
152     /*
153      * Trace Target
154      */
155     Trace* log_target_;
156         /*
157      * base_stn for mobilenodes communicating with wired nodes
158           */
159     int base_stn_;
160
161
162     //int last_rt_time_;
163 };
164
165 #endif // ns_mobilenode_h
166
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified Channel class with the implementation of the
#               multi-interface approach developed by [1]
#
# File:         channel.cc
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#============================================================================
*/


...

// MCMI: Due to the changes on the MobileNode lists, we modify accessing each
//       node entry when attaching, removing, and updating a new node to a channel
//       to refer to the corresponding channel number. This number can be
//       accessed by this->index(), where this is the current instance of the
//       channel class object. In other words, whenever nextX_ and prevX_ appear
//       in channel.cc, they need to be replaced by:
//           nextX_[this->index()]
//           prevX_[this->index()]

...

void
WirelessChannel::sendUp(Packet* p, Phy *tifp)
{
    Scheduler &s = Scheduler::instance();
    Phy *rifp = ifhead_.lh_first;
    Node *tnode = tifp->node();
    Node *rnode = 0;
    Packet *newp;
    double propdelay = 0.0;
    struct hdr_cmn *hdr = HDR_CMN(p);

    /* list-based improvement */
    if(highestAntennaZ_ == -1) {
        calcHighestAntennaZ(tifp);
#ifdef DEBUG
        fprintf(stdout, "channel.cc:sendUp - Calc highestAntennaZ_ and distCST_\n");
        fprintf(stdout, "highestAntennaZ_ = %0.1f,  distCST_ = %0.1f\n", highestAntennaZ_,
        distCST_);
#endif
    }

     hdr->direction() = hdr_cmn::UP;

     // still keep grid-keeper around ??
     if (GridKeeper::instance()) {
        int i;
        GridKeeper* gk = GridKeeper::instance();
        int size = gk->size_;

        MobileNode **outlist = new MobileNode *[size];

            int out_index = gk->get_neighbors((MobileNode*)tnode,
                                    outlist);
        for (i=0; i < out_index; i ++) {

          newp = p->copy();
          rnode = outlist[i];
          propdelay = get_pdelay(tnode, rnode);

          rifp = (rnode->ifhead()).lh_first;
          for(; rifp; rifp = rifp->nextnode()){
              if (rifp->channel() == this){
                 s.schedule(rifp, newp, propdelay);
```

```
76                  break;
77              }
78          }
79      }
80      delete [] outlist;
81
82   } else { // use list-based improvement
83
84       MobileNode *mtnode = (MobileNode *) tnode;
85       MobileNode **affectedNodes;// **aN;
86       int numAffectedNodes = -1, i;
87
88       if(!sorted_){
89           sortLists();
90       }
91
92       affectedNodes = getAffectedNodes(mtnode, distCST_ + /* safety */ 5,
         &numAffectedNodes);
93       for (i=0; i < numAffectedNodes; i++) {
94           rnode = affectedNodes[i];
95
96           if(rnode == tnode)
97               continue;
98
99           newp = p->copy();
100
101          propdelay = get_pdelay(tnode, rnode);
102
103          rifp = (rnode->ifhead()).lh_first;
104          for(; rifp; rifp = rifp->nextnode()){
105             // MCMI: Checks which of the interfaces of the destination node
106             //       is connected to the same channel
107             if (rifp->channel() == this) {
108                 s.schedule(rifp, newp, propdelay);
109             }
110         }
111     }
112     delete [] affectedNodes;
113 }
114 Packet::free(p);
115 }
116
117
118 void
119 WirelessChannel::addNodeToList(MobileNode *mn)
120 {
121     MobileNode *tmp;
122
123     // create list of mobilenodes for this channel
124     if (xListHead_ == NULL) {
125 #ifdef DEBUG
126         fprintf(stderr, "INITIALIZE THE LIST xListHead\n");
127 #endif
128         xListHead_ = mn;
129         xListHead_->nextX_[this->index()] = NULL;
130         xListHead_->prevX_[this->index()] = NULL;
131     } else {
132         for (tmp = xListHead_; tmp->nextX_[this->index()] != NULL;
         tmp=tmp->nextX_[this->index()]);
133         tmp->nextX_[this->index()] = mn;
134         mn->prevX_[this->index()] = tmp;
135         mn->nextX_[this->index()] = NULL;
136     }
137     numNodes_++;
138 }
139
140 void
141 WirelessChannel::removeNodeFromList(MobileNode *mn) {
142
143     MobileNode *tmp;
144     // Find node in list
145     for (tmp = xListHead_; tmp->nextX_[this->index()] != NULL;
     tmp=tmp->nextX_[this->index()]) {
146         if (tmp == mn) {
147             if (tmp == xListHead_) {
148                 xListHead_ = tmp->nextX_[this->index()];
```

```
149                 if (tmp->nextX_[this->index()] != NULL)
150                     tmp->nextX_[this->index()]->prevX_[this->index()] = NULL;
151             } else if (tmp->nextX_[this->index()] == NULL)
152                 tmp->prevX_[this->index()]->nextX_[this->index()] = NULL;
153             else {
154                 tmp->prevX_[this->index()]->nextX_[this->index()] =
    tmp->nextX_[this->index()];
155                 tmp->nextX_[this->index()]->prevX_[this->index()] =
    tmp->prevX_[this->index()];
156             }
157             numNodes_--;
158             return;
159         }
160     }
161     fprintf(stderr, "Channel: node not found in list\n");
162 }
163
164 void
165 WirelessChannel::sortLists(void) {
166     bool flag = true;
167     MobileNode *m, *q;
168     sorted_ = true;
169
170 #ifdef DEBUG
171     fprintf(stderr, "SORTING LISTS ...");
172 #endif
173
174     /* Buble sort algorithm */
175     // SORT x-list
176     while(flag) {
177         flag = false;
178         m = xListHead_;
179         while (m != NULL){
180             if(m->nextX_[this->index()] != NULL)
181                 if ( m->X() > m->nextX_[this->index()]->X() ){
182                     flag = true;
183                     //delete_after m;
184                     q = m->nextX_[this->index()];
185                     m->nextX_[this->index()] = q->nextX_[this->index()];
186                     if (q->nextX_[this->index()] != NULL)
187                         q->nextX_[this->index()]->prevX_[this->index()] = m;
188
189                     //insert_before m;
190                     q->nextX_[this->index()] = m;
191                     q->prevX_[this->index()] = m->prevX_[this->index()];
192                     m->prevX_[this->index()] = q;
193                     if (q->prevX_[this->index()] != NULL)
194                         q->prevX_[this->index()]->nextX_[this->index()] = q;
195
196                     // adjust Head of List
197                     if(m == xListHead_)
198                         xListHead_ = m->prevX_[this->index()];
199                 }
200             m = m -> nextX_[this->index()];
201         }
202     }
203
204 #ifdef DEBUG
205     fprintf(stderr, "DONE!\n");
206 #endif
207 }
208
209 void
210 WirelessChannel::updateNodesList(class MobileNode *mn, double oldX) {
211
212     MobileNode* tmp;
213     double X = mn->X();
214     bool skipX=false;
215
216     if(!sorted_) {
217         sortLists();
218         return;
219     }
220
221     /* xListHead cannot be NULL here (they are created during creation of mobilenode) */
222
```

```
223     /***  DELETE ***/
224     // deleting mn from x-list
225     if(mn->nextX_[this->index()] != NULL) {
226         if(mn->prevX_[this->index()] != NULL){
227             if((mn->nextX_[this->index()]->X() >= X) && (mn->prevX_[this->index()]->X() <=
    X)) skipX = true; // the node doesn't change its position in the list
228             else{
229                 mn->nextX_[this->index()]->prevX_[this->index()] =
    mn->prevX_[this->index()];
230                 mn->prevX_[this->index()]->nextX_[this->index()] =
    mn->nextX_[this->index()];
231             }
232         }
233 
234         else{
235             if(mn->nextX_[this->index()]->X() >= X) skipX = true; // skip updating the
    first element
236             else{
237                 mn->nextX_[this->index()]->prevX_[this->index()] = NULL;
238                 xListHead_ = mn->nextX_[this->index()];
239             }
240         }
241     }
242 
243     else if(mn->prevX_[this->index()] !=NULL){
244         if(mn->prevX_[this->index()]->X() <= X) skipX = true; // skip updating the last
        element
245         else mn->prevX_[this->index()]->nextX_[this->index()] = NULL;
246     }
247 
248     if ((mn->prevX_[this->index()] == NULL) && (mn->nextX_[this->index()] == NULL)) skipX
    = true; //skip updating if only one element in list
249 
250     /*** INSERT ***/
251     //inserting mn in x-list
252     if(!skipX){
253         if(X > oldX){
254             for(tmp = mn; tmp->nextX_[this->index()] != NULL &&
    tmp->nextX_[this->index()]->X() < X; tmp = tmp->nextX_[this->index()]);
255             //fprintf(stdout,"Scanning the element addr %d X=%0.f, next addr %d X=%0.f\n",
    tmp, tmp->X(), tmp->nextX_[this->index()], tmp->nextX_[this->index()]->X());
256             if(tmp->nextX_[this->index()] == NULL) {
257                 //fprintf(stdout, "tmp->nextX_[this->index()] is NULL\n");
258                 tmp->nextX_[this->index()] = mn;
259                 mn->prevX_[this->index()] = tmp;
260                 mn->nextX_[this->index()] = NULL;
261             }
262             else{
263                 //fprintf(stdout, "tmp->nextX_[this->index()] is not NULL,
    tmp->nextX_[this->index()]->X()=%0.f\n", tmp->nextX_[this->index()]->X());
264                 mn->prevX_[this->index()] =
    tmp->nextX_[this->index()]->prevX_[this->index()];
265                 mn->nextX_[this->index()] = tmp->nextX_[this->index()];
266                 tmp->nextX_[this->index()]->prevX_[this->index()] = mn;
267                 tmp->nextX_[this->index()] = mn;
268             }
269         }
270         else{
271             for(tmp = mn; tmp->prevX_[this->index()] != NULL &&
    tmp->prevX_[this->index()]->X() > X; tmp = tmp->prevX_[this->index()]);
272             //fprintf(stdout,"Scanning the element addr %d X=%0.f, prev addr %d
    X=%0.f\n", tmp, tmp->X(), tmp->prevX_[this->index()],
    tmp->prevX_[this->index()]->X());
273             if(tmp->prevX_[this->index()] == NULL) {
274                 //fprintf(stdout, "tmp->prevX_[this->index()] is NULL\n");
275                 tmp->prevX_[this->index()] = mn;
276                 mn->nextX_[this->index()] = tmp;
277                 mn->prevX_[this->index()] = NULL;
278                 xListHead_ = mn;
279             }
280             else{
281                 //fprintf(stdout, "tmp->prevX_[this->index()] is not NULL,
    tmp->prevX_[this->index()]->X()=%0.f\n", tmp->prevX_[this->index()]->X());
282                 mn->nextX_[this->index()] =
    tmp->prevX_[this->index()]->nextX_[this->index()];
283                 mn->prevX_[this->index()] = tmp->prevX_[this->index()];
```

```
284                 tmp->prevX_[this->index()]->nextX_[this->index()] = mn;
285                 tmp->prevX_[this->index()] = mn;
286             }
287         }
288     }
289 }
290
291
292 MobileNode **
293 WirelessChannel::getAffectedNodes(MobileNode *mn, double radius,
294                   int *numAffectedNodes)
295 {
296     double xmin, xmax, ymin, ymax;
297     int n = 0;
298     MobileNode *tmp, **list, **tmpList;
299
300     if (xListHead_ == NULL) {
301         *numAffectedNodes=-1;
302         fprintf(stderr, "xListHead_ is NULL when trying to send!!!\n");
303         return NULL;
304     }
305
306     xmin = mn->X() - radius;
307     xmax = mn->X() + radius;
308     ymin = mn->Y() - radius;
309     ymax = mn->Y() + radius;
310
311     // First allocate as much as possibly needed
312     tmpList = new MobileNode*[numNodes_];
313
314     for(tmp = xListHead_; tmp != NULL; tmp = tmp->nextX_[this->index()]) tmpList[n++] =
    tmp;
315     for(int i = 0; i < n; ++i)
316         if(tmpList[i]->speed()!=0.0 && (Scheduler::instance().clock() -
317                         tmpList[i]->getUpdateTime()) > XLIST_POSITION_UPDATE_INTERVAL )
318             tmpList[i]->update_position();
319     n=0;
320
321     for(tmp = mn; tmp != NULL && tmp->X() >= xmin; tmp=tmp->prevX_[this->index()])
322         if(tmp->Y() >= ymin && tmp->Y() <= ymax){
323             tmpList[n++] = tmp;
324         }
325     for(tmp = mn->nextX_[this->index()]; tmp != NULL && tmp->X() <= xmax;
    tmp=tmp->nextX_[this->index()]){
326         if(tmp->Y() >= ymin && tmp->Y() <= ymax){
327             tmpList[n++] = tmp;
328         }
329     }
330
331     list = new MobileNode*[n];
332     memcpy(list, tmpList, n * sizeof(MobileNode *));
333     delete [] tmpList;
334
335     *numAffectedNodes = n;
336     return list;
337 }
338
339
340
341 /* Only to be used with mobile nodes (WirelessPhy).
342  * NS-2 at its current state support only a flat (non 3D) movement of nodes,
343  * so we assume antenna heights do not change for the dureation of
344  * a simulation.
345  * Another assumption - all nodes have the same wireless interface, so that
346  * the maximum distance, corresponding to CST (at max transmission power
347  * level) stays the same for all nodes.
348  */
349 void
350 WirelessChannel::calcHighestAntennaZ(Phy *tifp)
351 {
352        double highestZ = 0;
353        Phy *n;
354
355        for(n = ifhead_.lh_first; n; n = n->nextchnl()) {
356                    if(((WirelessPhy *)n)->getAntennaZ() > highestZ)
357                                highestZ = ((WirelessPhy *)n)->getAntennaZ();
```

```
358         }
359
360         highestAntennaZ_ = highestZ;
361
362         WirelessPhy *wifp = (WirelessPhy *)tifp;
363         distCST_ = wifp->getDist(wifp->getCSThresh(), wifp->getPt(), 1.0, 1.0,
364                  highestZ , highestZ, wifp->getL(),
365                  wifp->getLambda());
366 }
367
368
369 double
370 WirelessChannel::get_pdelay(Node* tnode, Node* rnode)
371 {
372     // Scheduler    &s = Scheduler::instance();
373     MobileNode* tmnode = (MobileNode*)tnode;
374     MobileNode* rmnode = (MobileNode*)rnode;
375     double propdelay = 0;
376
377     propdelay = tmnode->propdelay(rmnode);
378
379     assert(propdelay >= 0.0);
380     if (propdelay == 0.0) {
381         /* if the propdelay is 0 b/c two nodes are on top of
382            each other, move them slightly apart -dam 7/28/98 */
383         propdelay = 2 * DBL_EPSILON;
384         //printf ("propdelay 0: %d->%d at %f\n",
385         //  tmnode->address(), rmnode->address(), s.clock());
386     }
387     return propdelay;
388 }
389
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified Mac class to support the implementation of the
#               interface protocol proposed by [2]
#
# File:         mac.h
#
# [2] P. Kyasanur and N. H. Vaidya, "Routing and Interface Assignment in
#     Multi-Channel Multi-Interface Wireless Networks," Wireless Communications
#     and Networking Conference (WCNC) 2005, vol. 4, pp. 2051-2056, Mar. 2005.
#============================================================================
*/


...

/* ======================================================================
   MAC data structure
   ======================================================================*/

class Mac : public BiConnector {
public:
    Mac();
    virtual void recv(Packet* p, Handler* h);
    virtual void sendDown(Packet* p);
    virtual void sendUp(Packet *p);

    virtual void resume(Packet* p = 0);
    virtual void installTap(Tap *t) { tap_ = t; }

    inline double txtime(int bytes) {
        return (8. * bytes / bandwidth_);
    }
    inline double txtime(Packet* p) {
        return 8. * (MAC_HDR_LEN + \
                  (HDR_CMN(p))->size()) / bandwidth_;
    }
    inline double bandwidth() const { return bandwidth_; }

    inline int addr() { return index_; }
    inline MacState state() { return state_; }
    inline MacState state(int m) { return state_ = (MacState) m; }

    // MCMI: So the network interface that this MAC is connecting to can be
    //       accessed externally. Before, it was a protected member variable.
    inline Phy *netif() { return netif_; }

    //mac methods to set dst, src and hdt_type in pkt hdrs.
    // note: -1 is the broadcast mac addr.
    virtual inline int hdr_dst(char* hdr, int dst = -2) {
        struct hdr_mac *dh = (struct hdr_mac*) hdr;
        if(dst > -2)
            dh->macDA_ = dst;
        return dh->macDA();
    }
    virtual inline int hdr_src(char* hdr, int src = -2) {
        struct hdr_mac *dh = (struct hdr_mac*) hdr;
        if(src > -2)
            dh->macSA_ = src;
        return dh->macSA();
    }
    virtual inline int hdr_type(char *hdr, u_int16_t type = 0) {
        struct hdr_mac *dh = (struct hdr_mac*) hdr;
        if (type)
            dh->hdr_type_ = type;
        return dh->hdr_type();
    }

private:
```

```
77          void mac_log(Packet *p) {
78                  logtarget_->recv(p, (Handler*) 0);
79          }
80          NsObject*       logtarget_;
81
82  protected:
83      int command(int argc, const char*const* argv);
84      virtual int initialized() {
85          return (netif_ && uptarget_ && downtarget_);
86      }
87      int index_;     // MAC address
88      double bandwidth_;      // channel bitrate
89      double delay_;      // MAC overhead
90      int abstract_;         //   MAC support for abstract LAN
91
92      Phy *netif_;            // network interface
93          Tap *tap_;              // tap agent
94      LL *ll_;                // LL this MAC is connected to
95      Channel *channel_;  // channel this MAC is connected to
96
97      Handler* callback_; // callback for end-of-transmission
98      MacHandlerResume hRes_; // resume handler
99      MacHandlerSend hSend_;  // handle delay send due to busy channel
100     Event intr_;
101
102     /*
103      * Internal MAC State
104      */
105     MacState state_;    // MAC's current state
106     Packet *pktRx_;
107     Packet *pktTx_;
108 };
109
110 #endif
111
```

```
/*
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified Mac802_11 class with the implementation of the
#               multi-interface approach developed by [1]
#
# File:         mac-802_11.cc
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#============================================================================
*/


...

void
Mac802_11::recv(Packet *p, Handler *h)
{
    struct hdr_cmn *hdr = HDR_CMN(p);
    /*
     * Sanity Check
     */
    assert(initialized());

    /*
     *  Handle outgoing packets.
     */
    if(hdr->direction() == hdr_cmn::DOWN) {
                send(p, h);
                return;
        }
    /*
     *  Handle incoming packets.
     *
     *  We just received the 1st bit of a packet on the network
     *  interface.
     *
     */

    /*
     *  If the interface is currently in transmit mode, then
     *  it probably won't even see this packet.  However, the
     *  "air" around me is BUSY so I need to let the packet
     *  proceed.  Just set the error flag in the common header
     *  to that the packet gets thrown away.
     */
    if(tx_active_ && hdr->error() == 0) {
        hdr->error() = 1;
    }

    // MCMI: For correct handling of multiple interfaces by the routing agent,
    //       register the correct MAC receiving interface - which a message was
    //       received through
    hdr->iface() = addr();

    if(rx_state_ == MAC_IDLE) {
        setRxState(MAC_RECV);
        pktRx_ = p;
        /*
         * Schedule the reception of this packet, in
         * txtime seconds.
         */
        mhRecv_.start(txtime(p));
    } else {
        /*
         *  If the power of the incoming packet is smaller than the
         *  power of the packet currently being received by at least
                *  the capture threshold, then we ignore the new packet.
         */
```

```
77          if(pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= p->txinfo_.CPThresh) {
78              capture(p);
79          } else {
80              collision(p);
81          }
82      }
83  }
84
85  ...
86
```

```
#============================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:      Chih-Hao Howard Chang
#               20007-2192
#               howardc@sfu.ca
#
# Description: Modified ns-lib TCL script with the implementation of the
#               multi-interface approach developed by [1]
#
# File:         ns-lib.tcl
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#============================================================================


...

# XXX This should be moved into the node initialization procedure instead
# of standing here in ns-lib.tcl.
Simulator instproc create-wireless-node args {
        $self instvar routingAgent_ wiredRouting_ propInstance_ llType_ \
        macType_ ifqType_ ifqlen_ phyType_ chan antType_ \
        energyModel_ initialEnergy_ txPower_ rxPower_ \
        idlePower_ sleepPower_ sleepTime_ transitionPower_ transitionTime_ \
        topoInstance_ level1_ level2_ inerrProc_ outerrProc_ FECProc_ \
        numifs_

    Simulator set IMEPFlag_ OFF

        # create node instance
        set node [eval $self create-node-instance $args]

        # basestation address setting
        if { [info exist wiredRouting_] && $wiredRouting_ == "ON" } {
        $node base-station [AddrParams addr2id [$node node-addr]]
        }
        switch -exact $routingAgent_ {
        DSDV {
            set ragent [$self create-dsdv-agent $node]
        }
        DSR {
            $self at 0.0 "$node start-dsr"
        }
        AODV {
            set ragent [$self create-aodv-agent $node]
        }
        TORA {
            Simulator set IMEPFlag_ ON
            set ragent [$self create-tora-agent $node]
        }
        DIFFUSION/RATE {
            eval $node addr $args
            set ragent [$self create-diffusion-rate-agent $node]
        }
        DIFFUSION/PROB {
            eval $node addr $args
            set ragent [$self create-diffusion-probability-agent $node]
        }
        Directed_Diffusion {
            eval $node addr $args
            set ragent [$self create-core-diffusion-rtg-agent $node]
        }
        FLOODING {
            eval $node addr $args
            set ragent [$self create-flooding-agent $node]
        }
        OMNIMCAST {
            eval $node addr $args
            set ragent [$self create-omnimcast-agent $node]
        }
        DumbAgent {
            set ragent [$self create-dumb-agent $node]
        }
```

```
77          ManualRtg {
78              set ragent [$self create-manual-rtg-agent $node]
79          }
80          # Manual Routing
81          MANUAL {
82              set ragent [$self create-manual-routing-agent $node]
83          }
84          default {
85              eval $node addr $args
86              puts "Wrong node routing agent!"
87              exit
88          }
89      }
90
91      # errProc_ and FECProc_ are an option unlike other
92          # parameters for node interface
93      if ![info exists inerrProc_] {
94          set inerrProc_ ""
95      }
96      if ![info exists outerrProc_] {
97          set outerrProc_ ""
98      }
99      if ![info exists FECProc_] {
100         set FECProc_ ""
101     }
102
103     # MCMI: Add main node interface
104     if {[info exists numifs_]} {
105         for {set i 0} {$i < $numifs_} {incr i} {
106             # Add one interface per channel
107             #puts "chan $i: $chan($i)"
108             $node add-interface $chan($i) $propInstance_ $llType_ $macType_ \
109                     $ifqType_ $ifqlen_ $phyType_ $antType_ $topoInstance_ \
110                     $inerrProc_ $outerrProc_ $FECProc_
111         }
112     } else {
113         $node add-interface $chan $propInstance_ $llType_ $macType_ \
114             $ifqType_ $ifqlen_ $phyType_ $antType_ $topoInstance_ \
115             $inerrProc_ $outerrProc_ $FECProc_
116     }
117
118     # Attach agent
119     if {$routingAgent_ != "DSR"} {
120         $node attach $ragent [Node set rtagent_port_]
121     }
122     if {$routingAgent_ == "DIFFUSION/RATE" ||
123             $routingAgent_ == "DIFFUSION/PROB" ||
124             $routingAgent_ == "FLOODING" ||
125             $routingAgent_ == "OMNIMCAST" ||
126         $routingAgent_ == "Directed_Diffusion" } {
127         $ragent port-dmux [$node demux]
128         $node instvar ll_
129         $ragent add-ll $ll_(0)
130     }
131     if { $routingAgent_ == "DumbAgent" } {
132         $ragent port-dmux [$node demux]
133     }
134
135
136     # Bind routing agent and mip agent if existing basestation
137     # address setting
138         if { [info exists wiredRouting_] && $wiredRouting_ == "ON" } {
139         if { $routingAgent_ != "DSR" } {
140             $node mip-call $ragent
141         }
142     }
143     #
144         # This Trace Target is used to log changes in direction
145         # and velocity for the mobile node.
146         #
147     set tracefd [$self get-ns-traceall]
148         if {$tracefd != "" } {
149         $node nodetrace $tracefd
150         $node agenttrace $tracefd
151     }
152     set namtracefd [$self get-nam-traceall]
```

```
153     if {$namtracefd != "" } {
154         $node namattach $namtracefd
155     }
156     if [info exists energyModel_] {
157         if  [info exists level1_] {
158             set l1 $level1_
159         } else {
160             set l1 0.5
161         }
162         if  [info exists level2_] {
163             set l2 $level2_
164         } else {
165             set l2 0.2
166         }
167         $node addenergymodel [new $energyModel_ $node \
168                 $initialEnergy_ $l1 $l2]
169         }
170         if [info exists txPower_] {
171         $node setPt $txPower_
172         }
173         if [info exists rxPower_] {
174         $node setPr $rxPower_
175         }
176         if [info exists idlePower_] {
177         $node setPidle $idlePower_
178         }
179 #
180     if [info exists sleepPower_] {
181         $node setPsleep $sleepPower_
182         }
183     if [info exists sleepTime_] {
184         $node setTSleep $sleepTime_
185         }
186     if [info exists transitionPower_] {
187         $node setPtransition $transitionPower_
188         }
189     if [info exists transitionTime_] {
190         $node setTtransition $transitionTime_
191         }
192 #
193     $node topography $topoInstance_
194
195     return $node
196 }
197
198 ...
199
200 # MCMI: Procedure to change the number of interfaces
201 Simulator instproc change-numifs {newnumifs} {
202     $self instvar numifs_
203     set numifs_ $newnumifs
204 }
205
206 # MCMI: Procedure to add an interface on a node
207 Simulator instproc add-channel {indexch ch} {
208     $self instvar chan
209     set chan($indexch) $ch
210 }
211
212 # MCMI: Procedure to get the number of interfaces
213 Simulator instproc get-numifs { } {
214     $self instvar numifs_
215     if [info exists numifs_] {
216         return $numifs_
217     } else {
218         return ""
219     }
220 }
221
222 # Procedure to add multiple interfaces as an argument to node-config label
223 Simulator instproc ifNum {val} {
224     $self set numifs_ $val
225 }
226
```

```
#===================================================================================
# ENSC 835: High-Performance Networks
# Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
#
# Student:     Chih-Hao Howard Chang
#              20007-2192
#              howardc@sfu.ca
#
# Description: Modified ns-mobilenode TCL library script with the implementation
#              of the multi-interface approach developed by [1]
#
# File:        ns-lib.tcl
#
# [1] R. A. Calvo and J. P. Campo, "Adding Multiple Interface Support in NS-2,"
#     University of Cantabria, Jan. 2007 (User Guide).
#===================================================================================


...

Node/MobileNode instproc reset {} {
    $self instvar arptable_ nifs_ netif_ mac_ ifq_ ll_ imep_
    for {set i 0} {$i < $nifs_} {incr i} {
        $netif_($i) reset
        $mac_($i) reset
        $ll_($i) reset
        $ifq_($i) reset
        if { [info exists opt(imep)] && $opt(imep) == "ON" } {
            $imep_($i) reset
        }
        # MCMI: Reset the ARP table of a MobileNode object per the number of
        #       interfaces defined.
        if {$arptable_($i) != ""} {
            $arptable_($i) reset
        }
    }
}

#
# Attach an agent to a node.  Pick a port and
# bind the agent to the port number.
# if portnumber is 255, default target is set to the routing agent
#
Node/MobileNode instproc add-target { agent port } {
    $self instvar dmux_ imep_ toraDebug_

    set ns [Simulator instance]
    set newapi [$ns imep-support]

    $agent set sport_ $port

    # MCMI: Get the number of interfaces from the simulator object
    set numIfsSimulator [$ns get-numifs]

    # special processing for TORA/IMEP node
    set toraonly [string first "TORA" [$agent info class]]
    if {$toraonly != -1 } {
        $agent if-queue [$self set ifq_(0)]  ;# ifq between LL and MAC
        #
        # XXX: The routing protocol and the IMEP agents needs handles
        # to each other.
        #
        $agent imep-agent [$self set imep_(0)]
        [$self set imep_(0)] rtagent $agent
    }

    # Special processing for AODV
    set aodvonly [string first "AODV" [$agent info class]]
    if {$aodvonly != -1 } {
        $agent if-queue [$self set ifq_(0)]   ;# ifq between LL and MAC
    }

    #<zheng: add>
    # Special processing for ZBR
    #set zbronly [string first "ZBR" [$agent info class]]
    #if {$zbronly != -1 } {
```

```
77      #   $agent if-queue [$self set ifq_(0)]   ;# ifq between LL and MAC
78      #}
79      #</zheng: add>
80
81      if { $port == [Node set rtagent_port_] } {
82          # MCMI: Special processing when multiple interfaces are supported
83          if {$numIfsSimulator != ""} {
84              for {set i 0} {$i < [$self set nifs_]} {incr i} {
85                  $agent if-queue $i [$self set ifq_($i)]
86              }
87          }
88
89          # Ad hoc routing agent setup needs special handling
90          $self add-target-rtagent $agent $port
91          return
92      }
93
94      # Attaching a normal agent
95      set namfp [$ns get-nam-traceall]
96      if { [Simulator set AgentTrace_] == "ON" } {
97          #
98          # Send Target
99          #
100         if {$newapi != ""} {
101             set sndT [$self mobility-trace Send "AGT"]
102         } else {
103             set sndT [cmu-trace Send AGT $self]
104         }
105         if { $namfp != "" } {
106             $sndT namattach $namfp
107         }
108         $sndT target [$self entry]
109         $agent target $sndT
110         #
111         # Recv Target
112         #
113         if {$newapi != ""} {
114             set rcvT [$self mobility-trace Recv "AGT"]
115         } else {
116             set rcvT [cmu-trace Recv AGT $self]
117         }
118         if { $namfp != "" } {
119             $rcvT namattach $namfp
120         }
121         $rcvT target $agent
122         $dmux_ install $port $rcvT
123     } else {
124         #
125         # Send Target
126         #
127         $agent target [$self entry]
128         #
129         # Recv Target
130         #
131         $dmux_ install $port $agent
132     }
133 }
134
135 Node/MobileNode instproc add-target-rtagent { agent port } {
136     $self instvar imep_ toraDebug_
137
138     set ns [Simulator instance]
139     set newapi [$ns imep-support]
140     set namfp [$ns get-nam-traceall]
141
142     set dmux_ [$self demux]
143     set classifier_ [$self entry]
144
145     # MCMI: Whether multiple interfaces exist in the simulation
146     set numIfsSimulator [$ns get-numifs]
147
148     # let the routing agent know about the port dmux
149     $agent port-dmux $dmux_
150
151     if { [Simulator set RouterTrace_] == "ON" } {
152         #
```

```
153         # Send Target
154         #
155         if {$newapi != ""} {
156             set sndT [$self mobility-trace Send "RTR"]
157         } else {
158             set sndT [cmu-trace Send "RTR" $self]
159         }
160         if { $namfp != "" } {
161             $sndT namattach $namfp
162         }
163         if { $newapi == "ON" } {
164             $agent target $imep_(0)
165             $imep_(0) sendtarget $sndT
166             # second tracer to see the actual
167             # types of tora packets before imep packs them
168             if { [info exists toraDebug_] && $toraDebug_ == "ON"} {
169                 set sndT2 [$self mobility-trace Send "TRP"]
170                 $sndT2 target $imep_(0)
171                 $agent target $sndT2
172             }
173             # MCMI
174             $sndT target [$self set ll_(0)]
175         } else {  ;#  no IMEP
176             # MCMI: If the number of interfaces is non-zero, the procedure
177             #       associates the routing agent with the corresponding link
178             #       layer target entity as many times as the number of
179             #       interfaces for the receiving target.
180             if {$numIfsSimulator != ""} {
181                 for {set i 0} {$i < [$self set nifs_]} {incr i} {
182                     set sndT [cmu-trace Send "RTR" $self]
183                     $agent target $i $sndT
184                     $sndT target [$self set ll_($i)]
185                 }
186             } else {
187                 $agent target $sndT
188                 $sndT target [$self set ll_(0)]
189             }
190         }
191 
192         #
193         # Recv Target
194         #
195         if {$newapi != ""} {
196             set rcvT [$self mobility-trace Recv "RTR"]
197         } else {
198             set rcvT [cmu-trace Recv "RTR" $self]
199         }
200         if { $namfp != "" } {
201             $rcvT namattach $namfp
202         }
203         if {$newapi == "ON" } {
204             [$self set ll_(0)] up-target $imep_(0)
205             $classifier_ defaulttarget $agent
206             # need a second tracer to see the actual
207             # types of tora packets after imep unpacks them
208             # no need to support any hier node
209             if {[info exists toraDebug_] && $toraDebug_ == "ON" } {
210                 set rcvT2 [$self mobility-trace Recv "TRP"]
211                 $rcvT2 target $agent
212                 $classifier_ defaulttarget $rcvT2
213             }
214         } else {
215             $rcvT target $agent
216             $classifier_ defaulttarget $rcvT
217             $dmux_ install $port $rcvT
218         }
219     } else {
220         #
221         # Send Target
222         #
223         # if tora is used
224         if { $newapi == "ON" } {
225             $agent target $imep_(0)
226             # second tracer to see the actual
227             # types of tora packets before imep packs them
228             if { [info exists toraDebug_] && $toraDebug_ == "ON"} {
```

```
229                 set sndT2 [$self mobility-trace Send "TRP"]
230                 $sndT2 target $imep_(0)
231                 $agent target $sndT2
232             }
233             $imep_(0) sendtarget [$self set ll_(0)]
234
235         } else {  ;#  no IMEP
236             # MCMI: If the number of interfaces is non-zero, the procedure
237             #       associates the routing agent with the corresponding link
238             #       layer target entity as many times as the number of
239             #       interfaces for the sending target.
240             if {$numIfsSimulator != ""} {
241                 for {set i 0} {$i < [$self set nifs_]} {incr i} {
242                     $agent target $i [$self set ll_($i)]
243                 }
244             } else {
245                 $agent target [$self set ll_(0)]
246             }
247         }
248
249         #
250         # Recv Target
251         #
252         if {$newapi == "ON" } {
253             [$self set ll_(0)] up-target $imep_(0)
254             $classifier_ defaulttarget $agent
255             # need a second tracer to see the actual
256             # types of tora packets after imep unpacks them
257             # no need to support any hier node
258             if {[info exists toraDebug_] && $toraDebug_ == "ON" } {
259                 set rcvT2 [$self mobility-trace Recv "TRP"]
260                 $rcvT2 target $agent
261                 [$self set classifier_] defaulttarget $rcvT2
262             }
263         } else {
264             $classifier_ defaulttarget $agent
265             $dmux_ install $port $agent
266         }
267     }
268 }
269
270 #
271 # The following setups up link layer, mac layer, network interface
272 # and physical layer structures for the mobile node.
273 #
274 Node/MobileNode instproc add-interface { channel pmodel lltype mactype qtype qlen iftype
    anttype topo inerrproc outerrproc fecproc } {
275     $self instvar arptable_ nifs_ netif_ mac_ ifq_ ll_ imep_ inerr_ outerr_ fec_
276
277     set ns [Simulator instance]
278     set imepflag [$ns imep-support]
279     set t $nifs_
280     incr nifs_
281
282     set netif_($t)  [new $iftype]       ;# interface
283     set mac_($t)    [new $mactype]      ;# mac layer
284     set ifq_($t)    [new $qtype]        ;# interface queue
285     set ll_($t) [new $lltype]       ;# link layer
286         set ant_($t)    [new $anttype]
287
288     $ns mac-type $mactype
289     set inerr_($t) ""
290     if {$inerrproc != ""} {
291         set inerr_($t) [$inerrproc]
292     }
293     set outerr_($t) ""
294     if {$outerrproc != ""} {
295         set outerr_($t) [$outerrproc]
296     }
297     set fec_($t) ""
298     if {$fecproc != ""} {
299         set fec_($t) [$fecproc]
300     }
301
302     set namfp [$ns get-nam-traceall]
303     if {$imepflag == "ON" } {
```

```
304         # IMEP layer
305         set imep_($t) [new Agent/IMEP [$self id]]
306         set imep $imep_($t)
307         set drpT [$self mobility-trace Drop "RTR"]
308         if { $namfp != "" } {
309             $drpT namattach $namfp
310         }
311         $imep drop-target $drpT
312         $ns at 0.[$self id] "$imep_($t) start"   ;# start beacon timer
313     }
314
315     #
316     # Local Variables
317     #
318     set nullAgent_ [$ns set nullAgent_]
319     set netif $netif_($t)
320     set mac $mac_($t)
321     set ifq $ifq_($t)
322     set ll $ll_($t)
323
324     set inerr $inerr_($t)
325     set outerr $outerr_($t)
326     set fec $fec_($t)
327
328     # MCMI: Create one ARP table per interface. Originally, it creates one ARP
329     #       table (for address resolution) per node. The reason for such a
330     #       walk-around is that, if a node is using one interface to communicate
331     #       with another one, the current design of MobileNode in ns-2 will not
332     #       allow the node to use another interface since the request to the ARP
333     #       entity will still be serving the previous interface.
334     set arptable_($t) [new ARPTable $self $mac]
335     set arptable $arptable_($t)
336
337     # FOR backward compatibility sake, hack only
338     if {$imepflag != ""} {
339         set drpT [$self mobility-trace Drop "IFQ"]
340     } else {
341         set drpT [cmu-trace Drop "IFQ" $self]
342     }
343     $arptable drop-target $drpT
344     if { $namfp != "" } {
345         $drpT namattach $namfp
346     }
347
348     #
349     # Link Layer
350     #
351     $ll arptable $arptable; # MCMI
352     $ll mac $mac
353     $ll down-target $ifq
354
355     if {$imepflag == "ON" } {
356         $imep recvtarget [$self entry]
357         $imep sendtarget $ll
358         $ll up-target $imep
359         } else {
360         $ll up-target [$self entry]
361     }
362
363     #
364     # Interface Queue
365     #
366     $ifq target $mac
367     $ifq set limit_ $qlen
368     if {$imepflag != ""} {
369         set drpT [$self mobility-trace Drop "IFQ"]
370     } else {
371         set drpT [cmu-trace Drop "IFQ" $self]
372         }
373     $ifq drop-target $drpT
374     if { $namfp != "" } {
375         $drpT namattach $namfp
376     }
377     if {[$ifq info class] == "Queue/XCP"} {
378         $mac set bandwidth_ [$ll set bandwidth_]
379         $mac set delay_ [$ll set delay_]
```

```
380         $ifq set-link-capacity [$mac set bandwidth_]
381         $ifq queue-limit $qlen
382         $ifq link $ll
383         $ifq reset
384
385     }
386
387     #
388     # Mac Layer
389     #
390
391     $mac netif $netif
392     $mac up-target $ll
393
394     if {$outerr == "" && $fec == ""} {
395         $mac down-target $netif
396     } elseif {$outerr != "" && $fec == ""} {
397         $mac down-target $outerr
398         $outerr target $netif
399     } elseif {$outerr == "" && $fec != ""} {
400         $mac down-target $fec
401         $fec down-target $netif
402     } else {
403         $mac down-target $fec
404         $fec down-target $outerr
405         $err target $netif
406     }
407
408     set god_ [God instance]
409         if {$mactype == "Mac/802_11"} {
410         $mac nodes [$god_ num_nodes]
411     }
412     #
413     # Network Interface
414     #
415     #if {$fec == ""} {
416         #       $netif up-target $mac
417     #} else {
418         #       $netif up-target $fec
419     #   $fec up-target $mac
420     #}
421
422     $netif channel $channel
423     if {$inerr == "" && $fec == ""} {
424         $netif up-target $mac
425     } elseif {$inerr != "" && $fec == ""} {
426         $netif up-target $inerr
427         $inerr target $mac
428     } elseif {$err == "" && $fec != ""} {
429         $netif up-target $fec
430         $fec up-target $mac
431     } else {
432         $netif up-target $inerr
433         $inerr target $fec
434         $fec up-target $mac
435     }
436
437     $netif propagation $pmodel  ;# Propagation Model
438     $netif node $self       ;# Bind node <---> interface
439     $netif antenna $ant_($t)
440     #
441     # Physical Channel
442     #
443     $channel addif $netif
444
445         # List-based improvement
446     # For nodes talking to multiple channels this should
447     # be called multiple times for each channel
448     $channel add-node $self
449
450     # let topo keep handle of channel
451     $topo channel $channel
452     # ============================================================
453
454     if { [Simulator set MacTrace_] == "ON" } {
455         #
```

```
456         # Trace RTS/CTS/ACK Packets
457         #
458         if {$imepflag != ""} {
459             set rcvT [$self mobility-trace Recv "MAC"]
460         } else {
461             set rcvT [cmu-trace Recv "MAC" $self]
462         }
463         $mac log-target $rcvT
464         if { $namfp != "" } {
465             $rcvT namattach $namfp
466         }
467         #
468         # Trace Sent Packets
469         #
470         if {$imepflag != ""} {
471             set sndT [$self mobility-trace Send "MAC"]
472         } else {
473             set sndT [cmu-trace Send "MAC" $self]
474         }
475         $sndT target [$mac down-target]
476         $mac down-target $sndT
477         if { $namfp != "" } {
478             $sndT namattach $namfp
479         }
480         #
481         # Trace Received Packets
482         #
483         if {$imepflag != ""} {
484             set rcvT [$self mobility-trace Recv "MAC"]
485         } else {
486             set rcvT [cmu-trace Recv "MAC" $self]
487         }
488         $rcvT target [$mac up-target]
489         $mac up-target $rcvT
490         if { $namfp != "" } {
491             $rcvT namattach $namfp
492         }
493         #
494         # Trace Dropped Packets
495         #
496         if {$imepflag != ""} {
497             set drpT [$self mobility-trace Drop "MAC"]
498         } else {
499             set drpT [cmu-trace Drop "MAC" $self]
500         }
501         $mac drop-target $drpT
502         if { $namfp != "" } {
503             $drpT namattach $namfp
504         }
505     } else {
506         $mac log-target [$ns set nullAgent_]
507         $mac drop-target [$ns set nullAgent_]
508     }
509
510 # change wrt Mike's code
511        if { [Simulator set EotTrace_] == "ON" } {
512                #
513                # Also trace end of transmission time for packets
514                #
515
516                if {$imepflag != ""} {
517                        set eotT [$self mobility-trace EOT "MAC"]
518                } else {
519                        set eoT [cmu-trace EOT "MAC" $self]
520                }
521                $mac eot-target $eotT
522        }
523
524     # ============================================================
525
526     $self addif $netif
527 }
528
529 ...
530
```