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/memi_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

Abstract.......................................................................................................................................... 1

List of Figures................................................................................................................................ 4

1. **Introduction** ........................................................................................................................... 5  
   1.1. Project Scope ..................................................................................................................... 6

2. **Wireless Ad-hoc Network** ..................................................................................................... 7  
   2.1. Overview ........................................................................................................................... 7  
   2.2. Multiple Channels and Multiple Interfaces ........................................................................... 8

3. **Related Work** ....................................................................................................................... 10

4. **Multi-Channel Multi-Interface Solution** ............................................................................11  
   4.1. Interface Switching ..........................................................................................................11  
       4.1.1. Fixed and Switchable Interfaces .............................................................................11  
       4.1.2. Fixed Interface Assignment ............................................................................... 13  
       4.1.3. Switchable Interface Assignment ........................................................................ 16

5. **Adding Multi-Channel Multi-Interface Support to ns-2** .................................................... 18  
   5.1. Overview of ns-2 ............................................................................................................... 18  
   5.2. Multiple Interface Node Model ........................................................................................ 19  
   5.3. Extending ns-2 with Multi-Interface Support ................................................................... 22  
       5.3.1. Changes on OTCL Code ....................................................................................... 22  
       5.3.2. Changes on C++ Code ......................................................................................... 24  
   5.4. Changes on the AODV Routing Protocol Code ............................................................... 25  
       5.4.1. Support to Make Use of the Modified MobileNode ............................................. 26  
       5.4.2. Implementation of Interface Switching ................................................................. 29  
   5.5. Modified ns-2 with Multi-Interface and Interface Switching Capabilities .................. 33

6. **Results and Discussions** ...................................................................................................... 34  
   6.1. Simulation Scenario ......................................................................................................... 34  
   6.2. Interface Switching Validation ....................................................................................... 36
List of Figures

Figure 1: Comparison between wireless networks in ad-hoc and infrastructure modes ............ 5
Figure 2: A wireless ad-hoc network ............................................................................................ 7
Figure 3: In the presence of only 1 interface, destination node has to listen to the same channel . 8
Figure 4: Transmission capacity is degraded when adjacent nodes are on the same channel ....... 9
Figure 5: Transmission capacity is doubled when adjacent nodes are on different channels ....... 9
Figure 6: Interface switching with 3 channels and 2 interfaces [2] .............................................. 12
Figure 7: Each node chooses a random channel for its fixed interface ...................................... 13
Figure 8: Each node periodically broadcasts on every channel its current fixed channel ............ 14
Figure 9: On receiving a Hello packet, a node updates its NeighborTable and ChannelUsageList ........................................................................................................................................... 14
Figure 10: Each node periodically consults its ChannelUsageList ............................................... 15
Figure 11: Packet queue maintained by each node for each channel [2] ........................................ 16
Figure 12: Overall Architecture of ns-2 ..................................................................................... 18
Figure 13: Original MobileNode Architecture in ns-2 [1] .............................................................. 20
Figure 14: Modified MobileNode architecture, with the support for multiple interfaces [1] ...... 21
Figure 15: AODV route establishment for a chain topology with 4 nodes ................................. 25
Figure 16: Modified files ns-2 with multi-interface and interface switching capabilities .......... 33
Figure 17: Block diagram of the ns-2 simulation scenario in chain topologies ............................ 34
Figure 18: Simulation set-up for chain topology with 4 nodes, 3 channels and 2 interfaces ...... 35
Figure 19: Interface switching in chain topology with 4 nodes, 3 channels and 2 interfaces ...... 35
Figure 20: ns-2 output for chain topology with 4 nodes, 3 channels and 2 interfaces ............... 36
Figure 21: nam output for chain topology with 4 nodes, 3 channels and 2 interfaces ............... 38
Figure 22: The scenario that illustrates the need for selecting channel deviated routes [2] ....... 41
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

---

Multiple channels 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 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.

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](image-url)
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](image)

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](image)
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 $\rightarrow$ B $\rightarrow$ 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]](image)

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](image)

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.
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.

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.
a. Suppose Node C wants to change to Channel 2, it broadcasts to its neighbour

b. After the broadcast, node C’s fixed interface switches to Channel 2 and each node’s ChannelUsageList gets updated.

c. Suppose Node B wants to change to Channel 3, it broadcasts to its neighbour.

Figure 10: Each node periodically consults its ChannelUsageList.
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:
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](image-url)
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](image)

*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.

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-rtarget {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 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](image-url)
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:

\[ \text{rt\_interface} = \text{ch->iface()} - ((\text{Mac \ *)ifqueuelist[0]->target()})->addr(), \]
where the second term of the subtraction is address of the first interface of the node. If multiple interfaces are not present, we assign \texttt{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 \textit{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, \texttt{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 \textit{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, \\
\texttt{/ns-2.32/aodv/aodv.h}:

- \texttt{MAX\_NT\_CUL\_ENTRIES}:
  - Maximum number of entries in \texttt{NeighbourTable} and \texttt{ChannelUsageList}.
- \texttt{neighbour\_table[MAX\_NT\_CUL\_ENTRIES]}:
  - Contains the fixed channels used by the node's neighbours, indexed by the node IP address.
- \texttt{channel\_usage\_list[MAX\_NT\_CUL\_ENTRIES]}:
  - Contains the number of nodes using each channel as their fixed channel, indexed by the channel number.
- \texttt{fixed\_interface, switchable\_interface}
  - Fixed and switchable interfaces used by this node.

Next, to facilitate the algorithm of \textit{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 \texttt{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 \texttt{hdr\_aodv\_request} (for RREQ) and \texttt{hdr\_aodv\_reply} (for RREP and Hello) structures in \texttt{/ns-2.32/aodv/aodv\_packet.h}:

- \texttt{rq\_fixed\_channel\_used/rp\_fixed\_channel\_used}
  - Fixed channel used by this node.
- \texttt{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;
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:
  o 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;
      }
    }
  o 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.

<table>
<thead>
<tr>
<th>Interface Switching</th>
<th>Multi-Interface Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ns-2.32</td>
<td>/ns-2.32</td>
</tr>
<tr>
<td>/aodv</td>
<td>/aodv</td>
</tr>
<tr>
<td>/aodv.cc</td>
<td>/common</td>
</tr>
<tr>
<td>/aodv</td>
<td>/mobilenode.cc</td>
</tr>
<tr>
<td>/aodv_packet.h</td>
<td>/mac</td>
</tr>
<tr>
<td>/aodv_rtable.h</td>
<td>/channel.cc</td>
</tr>
<tr>
<td></td>
<td>/mac.h</td>
</tr>
<tr>
<td></td>
<td>/mac-802_11.cc</td>
</tr>
<tr>
<td></td>
<td>/ns-2.32</td>
</tr>
<tr>
<td></td>
<td>/tcl</td>
</tr>
<tr>
<td></td>
<td>/lib</td>
</tr>
<tr>
<td></td>
<td>/ns-lib.tcl</td>
</tr>
<tr>
<td></td>
<td>/ns-mobilenode.tcl</td>
</tr>
</tbody>
</table>

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.

![Block diagram of the ns-2 simulation scenario in chain topologies](image)

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:

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

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:

```tcl
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
$ns_attach-agent $n(1) $udp0
$ns_attach-agent $n(2) $udp0
$ns_attach-agent $n(3) $udp0
$ns_connect $udp0 $sink0

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

**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.

![Table showing interface switching validation](image)

**Figure 19: Interface switching in chain topology with 4 nodes, 3 channels and 2 interfaces**
Changes in the \textit{NeighbourTable} and \textit{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 \textit{NeighbourTable} and \textit{ChannelUsageList} are populated accordingly. These tasks are performed in \texttt{command()}. 

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

In Step 2, Node 3 calls \texttt{sendReply()} to send a RREP to Node 2, which includes Node 3’s fixed channel and \textit{NeighbourTable}. In \texttt{recvReply()}, Node 2 updates its \textit{NeighbourTable} with the fixed channel of Node 3 and its \textit{ChannelUsageList} using Node 3’s \textit{NeighbourTable}. Node 2 then calls \texttt{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 \texttt{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 \textit{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 \texttt{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 \texttt{recvHello()}, they further update their \textit{NeighbourTable} with the fixed channel of Node 0 and their \textit{ChannelUsageList} using the \textit{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](image)

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](image)
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 $x$C2I 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.

![CBR Throughput in Chain Topologies](image)

**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.

![Diagram]

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

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


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
#!/usr/bin/env python

# 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
#======================================
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
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"
# Simulation Termination
#--------------------------------------
# Define a finish procedure
proc finish {} {
    global ns_tracefd filename pktsize last_node_id
    global namfile
    $ns_flush-trace
    close $tracefd
    close $namfile
    exec nam $filename.nam &

    # Call throughput analyzer (AWK scripts written by Marco Fiore, marco.fiore@polito.it)
    exec awk -f avgStats.awk src=0 dst=$last_node_id flow=0 pkt=$pktsize $filename.tr &
    exit 0
}
for {set i 0} {$i < $val(nn) } { incr i } {
    $ns_at $val(stop) "\$n($i) reset"
}
$ns_at $val(stop) "$ns_nam-end-wireless $val(stop)"
$ns_at $val(stop) "finish"
$ns_at $val(stop) "puts \"done\"" ; $ns_halt
$ns_run
avgStats.awk

#===============================================================================
# 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
    event = $1
    time = $2
    if (event == "+" || event == "-"这段时间 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
}
END {
    # Compute average delay
    delay = avg_delay = recvdNum = 0
    for (i in recvTime) {
        if (sendTime[i] == 0) {
            printf("\nError in delay.awk: receiving a packet that wasn't sent
%g
", i)
        }
        delay += recvTime[i] - sendTime[i]
        recvdNum ++
    }
    if (recvdNum != 0) {
        avg_delay = delay / recvdNum
    } else {
        avg_delay = 0
    }

    # Compute average jitters
    jitter1 = jitter2 = jitter3 = jitter4 = jitter5 = 0
    prev_time = delay = prev_delay = processed = deviation = 0
    prev_delay = -1
    for (i=0; processed<recvdNum; i++) {
        if(recvTime[i] != 0) {
            if(prev_time != 0) {
                delay = recvTime[i] - prev_time
                e2eDelay = recvTime[i] - sendTime[i]
                if(delay < 0) delay = 0
                if(prev_delay != -1) {
                    jitter1 += abs(e2eDelay - prev_e2eDelay)
                    jitter2 += abs(delay-prev_delay)
                    jitter3 += (abs(e2eDelay-prev_e2eDelay) - jitter3) / 16
                    jitter4 += (abs(delay-prev_delay) - jitter4) / 16
                }
                deviation += (e2eDelay-avg_delay)*(e2eDelay-avg_delay)
                prev_delay = delay
                prev_e2eDelay = e2eDelay
                prev_time = recvTime[i]
                processed++
            }
            prev_time = recvTime[i]
            processed++
        }
    }
    if (recvdNum != 0) {
        jitter1 = jitter1*1000/recvdNum
        jitter2 = jitter2*1000/recvdNum
    }
    # if (recvdNum > 1) {
    #    jitter5 = sqrt(deviation/(recvdNum-1))
    #} 
    # Output
    if (recvdNum == 0) {
        printf("###
            # Warning: no packets were received, simulation may be too short
            #
            ###")
    }
    printf("%g
", flows)
    printf("%s
", "trafficType", this_flow)
    printf("%d
", "srcNode", src)
    printf("%d
", "destNode", dst)
    printf("%s
", "startTime[s]", startTime)
    printf("%s
", "stopTime[s]", stopTime)
    printf("%s
", "duration[s]", stopTime-startTime)
    printf("%g
", "receivedPkts", recvdNum)
printf(" %15s:  %g\n", "avgTput[kbps]", (recvdSize/(stopTime-startTime))*(8/1000))
}

function abs(value) {
    if (value < 0) value = 0-value
    return value
}
```c
/*
#===============================================================================
# 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
#
# University of Cantabria, Jan. 2007 (User Guide).
#
# Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#===============================================================================
*/
...

// 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);
        }
    }
    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);
    }
    else if(strcmp(argv[1], "if-queue") == 0) {
        ifqueue = (PriQueue*) TclObject::lookup(argv[2]);
        if(ifqueue == 0)
            return TCL_ERROR;
        return TCL_OK;
    }
}
    else if (strcmp(argv[1], "port-dmux") == 0) {
        dmux_ = (PortClassifier *)TclObject::lookup(argv[2]);
        if (dmux_ == 0) {
            fprintf (stderr, "%s: %s lookup of %s failed\n", __FILE__,
                argv[1], argv[2]);
            return TCL_ERROR;
        }
        return TCL_OK;
    }
}
// MCMI: Populate ifqueuelist and targetlist, take the corresponding values
//       while the interfacs are being created in the nodes.
else if (argc == 4) {
    if (strcmp(argv[1], "if-queue") == 0) {
        int temp_ = atoi(argv[2]);
        if (temp_ == nIfaces) {
            nIfaces++;
            ifqueuelist[temp_] = (PriQueue *)TclObject::lookup(argv[3]);
            if (ifqueuelist[temp_]) {
                // MCMI: Initially, choose a random interface index in the range
                //       0 to temp_. Have to do it everytime since the node does
                //       not know how many channels it is going to have.
                fixed_interface = rand() % (temp_+1);
                #ifdef MCMI_DEBUG
                printf("n%d\tcommand - fixed_interface (initial): %d\n", (int)index,
                    fixed_interface);
                #endif
                if (nIfaces > 1) {
                    do {
                        switchable_interface = rand() % (temp_+1);
                    } while (switchable_interface == fixed_interface);
                    #ifdef MCMI_DEBUG
                    printf("n%d\tcommand - switchable_interface (initial): %d\n",
                        (int)index, switchable_interface);
                    #endif
                }
            }
        }
    // MCMI: Initialize NeighbourTable and ChannelUsageList
    for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
        neighbour_table[i] = -1;
        channel_usage_list[i] = 0;
    }
    // MCMI: Add the fixed channel used by this node to its NeighbourTable
    int fixed_channel = ((Mac*)ifqueuelist[fixed_interface]->target())->netif()->channel()->index();
    neighbour_table[(int)index] = fixed_channel; // fixed channel being used by
    channel_usage_list[fixed_channel]++;
    #ifdef MCMI_DEBUG
    printf("n%d\tcommand - fixed_channel: %d\n", (int)index, fixed_channel);
    rt_print_NT_CUL();
    #endif
    return TCL_OK;
    } else {
        return TCL_ERROR;
    }
} // if (strcmp(argv[1], "if-queue") == 0)
if (strcmp(argv[1], "target") == 0) {
    int temp_ = atoi(argv[2]);
    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() {
prompt("n%d\tNT\tCUL\n", (int)index);
for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
    prompt("n\t%", neighbour_table[i], channel_usage_list[i]);
}
prompt("\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);
aodv_t_entry *rt;
/*
 * Drop if:
 *      - I'm the source
 *      - I recently heard this request.
 */
if(rq->rq_src == index) {  
  if (id_lookup(rq->rq_src, rq->rq_bcast_id)) {  
    #ifdef DEBUG
    fprintf(stderr, "%s: discarding request\n", __FUNCTION__);  
    #endif // DEBUG
    Packet::free(p);
    return;
  }  
  #ifdef DEBUG
  fprintf(stderr, "%s: got my own REQUEST\n", __FUNCTION__);  
  #endif // DEBUG
  Packet::free(p);
  return;
  }  
/*
 * Cache the broadcast ID
 */
  id_insert(rq->rq_src, rq->rq_bcast_id);

if (nIfaces) {  
  // MCMI: When the node receives a "Hello" (RREQ) packet from a neighbour,  
  // update its NeighbourTable with the fixed channel of that neighbour.
  neighbour_table[(int)(rq->rq_sender_node_ip)] = rq->rq_fixed_channel_used;

  // MCMI: Update the node's ChannelUsageList using the NeighbourTable of its  
  // neighbour. Doing so ensures ChannelUsageList will contain two-hop  
  // channel usage information.
  for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {  
    if (i == (int)index) {  
      continue; // MCMI: No need to deal with the fixed channel used  
      // by itself again; avoid repetition
    }  
    if (rq->rq_neighbour_table[i] != -1) {  
      channel_usage_list[rq->rq_neighbour_table[i]]++;
    }
  }  
  #ifdef MCMI_DEBUG
  printf("n%d	recvRequest - rq->rq_sender_node_ip: %d\n", (int)index,  
    rq->rq_sender_node_ip);
  printf("n%d	recvRequest - rq_rq->fixed_channel_used: %d\n", (int)index,  
    rq->rq_fixed_channel_used);
  rt_print_NT_CUL();
  #endif
  } // if (nIfaces)

/*
 * We are either going to forward the REQUEST or generate a
 * REPLY. Before we do anything, we make sure that the REVERSE
 * route is in the route table.
 */
  aodv_t_entry *rt0; // rt0 is the reverse route
  rt0 = rtable.rt_lookup(rq->rq_src);
  if(rt0 == 0) { /* if not in the route table */  
    // create an entry for the reverse route.
    rt0 = rtable.rt_add(rq->rq_src);

    // MCMI: Get the interface index
    if (nIfaces) {
      #ifdef MCMI_DEBUG

printf("%d\trecvRequest - ch->iface() (rt0 = 0): %d\n", (int)index, 
        ch->iface());
    printf("%d\trecvRequest - ((Mac *)ifqueuelist[0]->target())->addr() (rt0 = 0): %d
", (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
#endif
    rt0->rt_interface = ch->iface() - ((Mac *)ifqueuelist[0]->target())->addr();
    } else {
        rt0->rt_interface = -1;
    }  
#endif  
    printf("%d	recvRequest - rt0->rt_interface (rt0 = 0): %d\n", (int)index, 
            rt0->rt_interface);
    rt0->rt_expire = max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE));
    if ( (rq->rq_src_seqno > rt0->rt_seqno ) ||
         ((rq->rq_src_seqno == rt0->rt_seqno) &&
          (rq->rq_hop_count < rt0->rt_hops)) ) {
        // If we have a fresher seq no. or lesser #hops for the
        // same seq no., update the rt entry. Else don't bother.
        rt_update(rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr(),
                  max(rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE)) );
    }
    // MCFI: Get the interface index
    if (nIffaces) {
        #ifdef MCFI_DEBUG
            printf("%d\trecvRequest - ch->iface(): %d\n", (int)index, ch->iface());
            printf("%d\trecvRequest - ((Mac *)ifqueuelist[0]->target())->addr(): %d
", (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
        #endif
        rt0->rt_interface = ch->iface() - ((Mac *)ifqueuelist[0]->target())->addr();
    } else {
        rt0->rt_interface = -1;
    }  
    #ifdef MCFI_DEBUG
        printf("%d	recvRequest - rt0->rt_interface: %d\n\n", (int)index,
                rt0->rt_interface);
    #endif
    if (rt0->rt_req_timeout > 0.0) {
        // Reset the soft state and
        // Set expiry time to CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT
        // This is because route is used in the forward direction,
        // but only sources get benefited by this change
        rt0->rt_req_cnt = 0;
        rt0->rt_req_timeout = 0.0;
        rt0->rt_req_last_ttl = rq->rq_hop_count;
        rt0->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
    }
    /* Find out whether any buffered packet can benefit from the
     * reverse route.
     * May need some change in the following code - Mahesh 09/11/99
     */
    assert (rt0->rt_flags == RTF_UP);
    Packet *buffered_pkt;
    while ((buffered_pkt = rqueue.deque(rt0->rt_dst))) {
        if (rt0 && (rt0->rt_flags == RTF_UP)) {
            assert (rt0->rt_hops != INFINITY2);
            forward(rt0, buffered_pkt, NO_DELAY);
        }
    }  
    // End for putting reverse route in rt table
    /*
We have taken care of the reverse route stuff.
Now see whether we can send a route reply.
rt = rtable.rt_lookup(rq->rq_dst);

// First check if I am the destination ..
if(rq->rq_dst == index) {

#ifdef DEBUG
fprintf(stderr, "%d - %s: destination sending reply\n", index, __FUNCTION__); #endif // DEBUG

// Just to be safe, I use the max. Somebody may have incremented the dst seqno.
seqno = max(seqno, rq->rq_dst_seqno)+1;
if (seqno%2) seqno++;

destination_ip = (int)index;
sendReply(rq->rq_src, // IP Destination
          1, // Hop Count
          index, // Dest IP Address
          seqno, // Dest Sequence Num
          MY_ROUTE_TIMEOUT, // Lifetime
          rq->rq_timestamp); // timestamp
Packet::free(p);
}

// I am not the destination, but I may have a fresh enough route.
else if (rt && (rt->rt_hops != INFINITY2) &&
          (rt->rt_seqno >= rq->rq_dst_seqno) ) {

#ifdef DEBUG
fprintf(stderr, "%d - %s: route cache send reply\n", index, __FUNCTION__); #endif // DEBUG

assert (rt->rt_flags == RTF_UP);
assert(rq->rq_dst == rt->rt_dst);
//assert ((rt->rt_seqno%2) == 0); // is the seqno even?
sendReply(rq->rq_src,
          rt->rt_hops + 1,
          rq->rq_dst,
          rt->rt_seqno,
          (u_int32_t) (rt->rt_expire - CURRENT_TIME),
          //rt->rt_expire - CURRENT_TIME,
          // Insert nexthops to RREQ source and RREQ destination in the
          // precursor lists of destination and source respectively
          rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
          rt0->pc_insert(rt->rt_nexthop); // nexthop to RREQ destination
#endif

// TODO: send 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.
Packet::free(p);
}
ih->saddr() = index;
ih->daddr() = IP_BROADCAST;
rq->rq_hop_count += 1;
// Maximum sequence number seen en route
if (rt) rq->rq_dst_seqno = max(rt->rt_seqno, rq->rq_dst_seqno);
forward((aodv_rt_entry*) 0, p, DELAY);
}

void
AODV::recvReply(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_reply *rp = HDR_AODV_REPLY(p);
aodv_rt_entry *rt;
char suppress_reply = 0;
double delay = 0.0;
#ifdef DEBUG
fprintf(stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
#endif // DEBUG
if (nIfaces) {
// MCMI: When the node receives a "Hello" (RREP) packet from a neighbour,
// update its NeighbourTable with the fixed channel of that neighbour
neighbour_table[(int)(rp->rp_sender_node_ip)] = rp->rp_fixed_channel_used;
// MCMI: Update the node's ChannelUsageList using the NeighbourTable of its
// neighbour. Doing so ensures ChannelUsageList will contain two-hop
// channel usage information.
for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
  if (i == (int)index) {
    continue; // MCMI: No need to deal with the fixed channel used
    // by itself again; avoid repetition
  }
  if (rp->rp_neighbour_table[i] != -1) {
    if ((rp->rp_neighbour_table[i] != -1) &&
        !(channel_usage_list[rp->rp_neighbour_table[i]])
       ) {
      channel_usage_list[rp->rp_neighbour_table[i]]++;
    }
  }
#ifdef MCMI_DEBUG
printf("n%d	recvReply - rp->rp_sender_node_ip: %d\n", (int)index,
rp->rp_sender_node_ip);
printf("n%d	recvReply - rp->rp_fixed_channel_used: %d\n", (int)index,
rp->rp_fixed_channel_used);
rt_print_NT_CUL();
#endif
} // if (nIfaces)
/*
* Got a reply. So reset the "soft state" maintained for
* route requests in the request table. We don't really have
* have a separate request table. It is just a part of the
* routing table itself.
*/
rt = rtable.rt_lookup(rp->rp_dst);
if(rt == 0) {
  rt = rtable.rt_add(rp->rp_dst);
}
// MCMI: Get the interface index
if (nIfaces) {
  #ifdef MCMI_DEBUG
  printf("n%d\trecvReply - ch->iface() (rt = 0): %d
", (int)index, ch->iface());
  printf("n%d\trecvReply - ((Mac *)ifqueuelist[0]->target())->addr() (rt = 0): %d
", (int)index, ((Mac *)ifqueuelist[0]->target())->addr());
  #endif
  rt->rt_interface = ch->iface() - ((Mac *)ifqueuelist[0]->target())->addr();
} else {
  rt->rt_interface = -1;
}
#ifdef MCMI_DEBUG
printf("n%d\trecvReply - rt->rt_interface (rt = 0): %d

", (int)index, rt->rt_interface);
#endif
/*
 * Add a forward route table entry... here I am following
 * Perkins-Royer AODV paper almost literally - SRD 5/99
 */
if ( (rt->rt_seqno < rp->rp_dst_seqno) ||   // newer route
    (rt->rt_seqno == rp->rp_dst_seqno) &&
    (rt->rt_hops > rp->rp_hop_count)) {   // shorter or better route
  // Update the rt entry
  rt_update(rt, rp->rp_dst_seqno, rp->rp_hop_count,
            rp->rp_src, CURRENT_TIME + rp->rp_lifetime);
  // reset the soft state
  rt->rt_req_cnt = 0;
  rt->rt_req_timeout = 0.0;
  rt->rt_req_last_ttl = rp->rp_hop_count;
  if (ih->daddr() == index) { // If I am the original source
    // Update the route discovery latency statistics
    // rp->rp_timestamp is the time of request origination
    rt->rt_disc_latency[(unsigned char)rt->hist_indx] = (CURRENT_TIME -
    rp->rp_timestamp) / (double) rp->rp_hop_count;
    // increment indx for next time
    rt->hist_indx = (rt->hist_indx + 1) % MAX_HISTORY;
  }
  /* Send all packets queued in the sendbuffer destined for
  * this destination.
  * XXX - observe the "second" use of p.
  */
  Packet *buf_pkt;
  while((buf_pkt = rqueue.deque(rt->rt_dst))) {
    if(rt->rt_hops != INFINITY2) {
      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 else {
554 suppress_reply = 1;
555 }
556 }
557 /*
558 * If reply is for me, discard it.
559 */
560 if(ih->daddr() == index || suppress_reply) {
561 Packet::free(p);
562 }
563 }
564 */
565 /* Otherwise, forward the Route Reply.
566 */
567 else {
568 /* Find the rt entry
569 aodv rt_entry *rt0 = rtable.rt_lookup(ih->daddr());
570 // If the rt is up, forward
571 if(rt0 && (rt0->rt_hops != INFINITY2)) {
572 assert (rt0->rt_flags == RTF_UP);
573 rp->rp_hop_count += 1;
574 rp->rp_src = index;
575 forward(rt0, p, NO_DELAY);
576 // Insert the nexthop towards the RREQ source to
577 // the precursor list of the RREQ destination
578 rt->pc_insert(rt0->rt_nexthop); // nexthop to RREQ source
579 }
580 else {
581 // I don't know how to forward .. drop the reply.
582 #ifdef DEBUG
583 fprintf(stderr, "%s: dropping Route Reply\n", __FUNCTION__);
584 #endif // DEBUG
585 drop(p, DROP_RTR_NO_ROUTE);
586 }
587 }
588 }  
589 void
590 AODV::recvError(Packet *p) {
591 struct hdr_ip *ih = HDR_IP(p);
592 struct hdr_aodv_error *re = HDR_AODV_ERROR(p);
593 aodv rt_entry *rt;
594 u_int8_t i;
595 Packet *rerr = Packet::alloc();
596 struct hdr_aodv_error *nre = HDR_AODV_ERROR(rerr);
597 nre->DestCount = 0;
598 for (i=0; i<re->DestCount; i++) {
599 // For each unreachable destination
600 rt = rtable.rt_lookup(re->unreachable_dst[i]);
601 if ( rt && (rt->rt_hops != INFINITY2) &&
602 (rt->rt_nexthop == ih->saddr()) &&
603 (rt->rt_seqno <= re->unreachable_dst_seqno[i]) ) {
604 assert(rt->rt_flags == RTF_UP);
605 assert((rt->rt_seqno%2) == 0); // is the seqno even?
606 #ifdef DEBUG
607 fprintf(stderr, "%s(%f): n%d: t(n%d\t%u\t%d)\t(n%d\t%u\t%d)\n",
608 __FUNCTION__, CURRENT_TIME,
609 index, rt->rt_dst, rt->rt_seqno, rt->rt_nexthop,
610 re->unreachable_dst[i],re->unreachable_dst_seqno[i],
611 ih->saddr());
612 #endif // DEBUG
613 }
rt->rt_seqno = re->unreachable_dst_seqno[i];
rt_down(rt);

// Not sure whether this is the right thing to do
Packet *pkt;

// MCMI: Get the corresponding interface queue
for (int i = 0; i < nIfaces; i++)
  if (rt->rt_interface == i) {
    ifqueue = ifqueuelist[i];
    break;
  }

while((pkt = ifqueue->filter(ih->saddr()))) {
  drop(pkt, DROP_RTR_MAC_CALLBACK);
}

// if precursor list non-empty add to RERR and delete the precursor list
if (!rt->pc_empty()) {
  nre->unreachable_dst[nre->DestCount] = rt->rt_dst;
  nre->unreachable_dst_seqno[nre->DestCount] = rt->rt_seqno;
  nre->DestCount += 1;
  rt->pc_delete();
}

if (nre->DestCount > 0) {
  #ifdef DEBUG
  fprintf(stderr, "%s(%f): n%d\t sending RERR...\n", __FUNCTION__, CURRENT_TIME,
          index);
  #endif // DEBUG
  sendError(rerr);
}
else {
  Packet::free(rerr);
}

Packet::free(p);

/*
 Packet Transmission Routines
 */

void
AODV::forward(aodv_rt_entry *rt, Packet *p, double delay) {
  struct hdr_cmn *ch = HDR_CMN(p);
  struct hdr_ip *ih = HDR_IP(p);
  struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);  // MCMI: RREQ packet
  struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);      // MCMI: RREP packet

  if(ih->ttl_ == 0) {
    #ifdef DEBUG
    fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
    #endif // DEBUG
    drop(p, DROP_RTR_TTL);
  return;
  }

  if ((ch->ptype() != PT_AODV && ch->direction() == hdr_cmn::UP
      && ((u_int32_t)ih->daddr() == IP_BROADCAST)
      || (ih->daddr() == here_.addr_)) {
    dmux_->recv(p,0);
    return;
  }
if (rt) {
    assert(rt->rt_flags == RTF_UP);
    rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
    ch->next_hop = rt->rt_nexthop;
    ch->addr_type() = NS_AF_INET;
    ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
}
else { // if it is a broadcast packet
    // assert(ch->ptype == PT_AODV); // maybe a diff pkt type like gaf
    assert(ih->daddr() == (nsaddr_t) IP_BROADCAST);
    ch->addr_type() = NS_AF_NONE;
    ch->direction() = hdr_cmn::DOWN;       //important: change the packet's direction
}

if (nIfaces) {
    #ifdef MCMI_DEBUG
    printf("%d\tforward - rq_type: 0x%x\n", (int)index, rq->rq_type);
    printf("%d\tforward - rp_type: 0x%x\n", (int)index, rp->rp_type);
    #endif
    // MCMI: For the case of forwarding a RREQ...
    if (rq->rq_type == AODVTYPE_RREQ) {
        // MCMI: Add the fixed channel used by this node to the RREQ packet
        rq->rq_fixed_channel_used = neighbour_table[(int)index];
        rq->rq_sender_node_ip = (int)index;
        rq->rq_neighbour_table = &neighbour_table[0];
        #ifdef MCMI_DEBUG
        printf("%d\tforward - rq_fixed_channel_used: %d\n", (int)index,
               rq->rq_fixed_channel_used);
        printf("%d\tforward - rq_sender_node_ip: %d\n", (int)index,
               rq->rq_sender_node_ip);
        #endif
    }
    // MCMI: For the case of forwarding a RREP...
    if (rp->rp_type == AODVTYPE_RREP || rp->rp_type == AODVTYPE_HELLO) {
        // MCMI: Add the fixed channel used by this node to the RREP packet
        rp->rp_fixed_channel_used = neighbour_table[(int)index];
        rp->rp_sender_node_ip = (int)index;
        rp->rp_neighbour_table = &neighbour_table[0];
        #ifdef MCMI_DEBUG
        printf("%d\tforward - rp_fixed_channel_used: %d\n", (int)index,
               rp->rp_fixed_channel_used);
        printf("%d\tforward - rp_sender_node_ip: %d\n", (int)index,
               rp->rp_sender_node_ip);
        #endif
    }
    // MCMI: For the case of forwarding data packets...
    if (rq->rq_type == 0 && rp->rp_type == 0) {
        // MCMI: consult the node's ChannelUsageList, find the largest usage
        int channel_largest_usage = 0;
        int channel_lowest_usage = 0;
        int found = 0;
        static int skip = 0;
        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;
                found = 1;
            }
            if (channel_usage_list[i] <= channel_usage_list[channel_lowest_usage]) {
                channel_lowest_usage = i;
            }
        }
        #ifdef MCMI_DEBUG
        printf("%d\tforward - channel_largest_usage: %d, #nodes: %d\n", (int)index,
                channel_largest_usage, channel_usage_list[channel_largest_usage]);
        #endif
    }
printf("n%d\tforward - channel_lowest_usage: %d, #nodes: %d\n", (int)index, channel_lowest_usage, channel_usage_list[channel_lowest_usage]);
#endif
if (found && !skip) {
    // MCMI: If the node's fixed channel is the one with the largest usage...
    if (channel_usage_list[neighbour_table[(int)index]] ==
        channel_usage_list[channel_largest_usage]) {
        int switch_probability = rand() % 101; // switching probability
        #ifdef MCMI_DEBUG
        printf("n%d\tforward - switch_probability: %d\n", (int)index, switch_probability);
        #endif
        if (switch_probability <= 40) {
            // MCMI: Reverse the ChannelUsageList about the fixed channel
            int fixed_channel = ((Mac*)ifqueuelist[fixed_interface]->target())->netif()->channel()->index();
            channel_usage_list[fixed_channel]--;
            // MCMI: Change the node's fixed channel to a less used channel
            fixed_channel = channel_lowest_usage; // new fixed channel
            channel_usage_list[fixed_channel]++;
            for (int i = 0; i < nIfaces; i++) {
                neighbour_table[(int)index] = fixed_channel;
                if (((Mac*)ifqueuelist[i]->target())->netif()->channel()->index() == fixed_channel) {
                    fixed_interface = i;
                    break;
                }
            }
            #ifdef MCMI_DEBUG
            printf("n%d\tforward - \tnew fixed_channel: %d\n", (int)index, fixed_channel);
            printf("n%d\tforward - \tnew fixed_interface: %d\n", (int)index, fixed_interface);
            rt_print_NT_CUL();
            printf("n%d\tforward - \tready to sendHello\n", (int)index);
            #endif
            // MCMI: Transmit a Hello packet informing neighbours of its new fixed channel
            sendHello();
            if (switch_probability <= 40)
                if (channel_usage_list[neighbour_table[(int)index]] >= 1)
                    skip = 1;
    } // if (found)
    skip = 1;
    // MCMI: Look up the fixed channel of the next node in its NeighbourTable and assign it as the switchable channel
    int switchable_channel = neighbour_table[(int)(rt->rt_nexthop)];
    #ifdef MCMI_DEBUG
    printf("n%d\tforward - back to forward\n", (int)index);
    printf("n%d\tforward - rt->rt_nexthop: %d\n", (int)index, (int)(rt->rt_nexthop));
    printf("n%d\tforward - switchable_channel: %d\n", (int)index, switchable_channel);
    #endif
    // MCMI: Given the switchable channel, find the corresponding interface
    switchable_channel
    for (int i = 0; i < MAX_IF; i++) {
        if (((Mac*)ifqueuelist[i]->target())->netif()->channel()->index() ==
            switchable_channel)
            switchable_interface = i;
        break;
    }
    #ifdef MCMI_DEBUG
    printf("n%d\tforward - switchable_interface: %d\n", (int)index, switchable_interface);
    #endif
} // if (rq->rq_type == 0 && rp->rp_type == 0)
```cpp
void aodv.cc

if (ih->daddr() == (nsaddr_t) IP_BROADCAST) {
    // If it is a broadcast packet
    assert(rt == 0);
    /*
     * Jitter the sending of broadcast packets by 10ms
    */
    if (nIfaces) { // MCI: Send a broadcast packet
        for (int i = 0; i < nIfaces; i++) {
            Packet *p_copy = p->copy();
            Scheduler::instance().schedule(targetlist[i], p_copy, 0.01 *
            Random::uniform());
        }
    } else {
        Scheduler::instance().schedule(target_, p, 0.01 * Random::uniform());
    }
} else { // Not a broadcast packet
    if(delay > 0.0) {
        if (nIfaces) {
            if (rq->rq_type == 0 && rp->rp_type == 0) { // data packet
                #ifdef MCI_DEBUG
                printf("n%d\tforward - switchable_interface (delay > 0): %d\n\n", (int)index, switchable_interface);
                #endif
                Scheduler::instance().schedule(targetlist[switchable_interface], p, delay);
            } else {
                #ifdef MCI_DEBUG
                printf("n%d\tforward - rt->rt_interface (delay > 0): %d\n\n", (int)index, rt->rt_interface);
                #endif
                Scheduler::instance().schedule(targetlist[rt->rt_interface], p, delay);
            }
        } else {
            Scheduler::instance().schedule(target_, p, delay);
        }
    } else { // Not a broadcast packet, no delay, send immediately
        if (nIfaces) { // MCI: Send a unicast packet
            if (rq->rq_type == 0 && rp->rp_type == 0) { // data packet
                #ifdef MCI_DEBUG
                printf("n%d\tforward - switchable_interface (else): %d\n\n", (int)index, switchable_interface);
                #endif
                Scheduler::instance().schedule(targetlist[switchable_interface], p, 0);
            } else {
                #ifdef MCI_DEBUG
                printf("n%d\tforward - rt->rt_interface (else): %d\n\n", (int)index, rt->rt_interface);
                #endif
                Scheduler::instance().schedule(targetlist[rt->rt_interface], p, 0);
            }
        } else {
            Scheduler::instance().schedule(target_, p, 0);
        }
    }
}
```
AODV::sendRequest(nsaddr_t dst) {
    // Allocate a RREQ packet
    Packet *p = Packet::alloc();
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_aodv_request *rq = HDR_AODV_REQUEST(p);
    aodv_rt_entry *rt = rtable.rt_lookup(dst);
    assert(rt);

    /*
     * Rate limit sending of Route Requests. We are very conservative
     * about sending out route requests.
     */
    if (rt->rt_flags == RTF_UP) {
        assert(rt->rt_hops != INFINITY2);
        Packet::free((Packet *)p);
        return;
    }

    if (rt->rt_req_timeout > CURRENT_TIME) {
        Packet::free((Packet *)p);
        return;
    }

    // rt_req_cnt is the no. of times we did network-wide broadcast
    // RREQ_RETRIES is the maximum number we will allow before
    // going to a long timeout.
    if (rt->rt_req_cnt > RREQ_RETRIES) {
        rt->rt_req_cnt = 0;
        Packet *buf_pkt;
        while ((buf_pkt = rqueue.deque(rt->rt_dst))) {
            drop(buf_pkt, DROP_RTR_NO_ROUTE);
        }
        Packet::free((Packet *)p);
        return;
    }

    #ifdef DEBUG
    fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
            ++route_request, index, rt->rt_dst);
    #endif // DEBUG

    // Determine the TTL to be used this time.
    // Dynamic TTL evaluation - SRD
    rt->rt_req_last_ttl = max(rt->rt_req_last_ttl,rt->rt_last_hop_count);
    if (0 == rt->rt_req_last_ttl) {
        // first time query broadcast
        ih->ttl_ = TTL_START;
    }
    else {
        // Expanding ring search.
        if (rt->rt_req_last_ttl < TTL_THRESHOLD)
            ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
        else {
            // network-wide broadcast
            ih->ttl_ = NETWORK_DIAMETER;
            rt->rt_req_cnt += 1;
        }
    }

    // remember the TTL used for the next time
    rt->rt_req_last_ttl = ih->ttl_;

    // PerHopTime is the roundtrip time per hop for route requests.
    // The factor 2.0 is just to be safe .. SRD 5/22/99
}
rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime(rt);
if (rt->rt_req_cnt > 0)
  rt->rt_req_timeout *= rt->rt_req_cnt;
rt->rt_req_timeout += CURRENT_TIME;

// Don't let the timeout to be too large, however .. SRD 6/8/99
if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
  rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
rt->rt_expire = 0;
#ifdef DEBUG
  fprintf(stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
          ++route_request,
          index, rt->rt_dst,
          rt->rt_req_timeout - CURRENT_TIME);
#endif  // DEBUG

// Fill out the RREQ packet
ch->uid() = 0;
ch->ptype() = PT_AODV;
ch->size() = IP_HDR_LEN + rq->size();
ch->iface() = -2;
ch->error() = 0;
ch->addr_type() = NS_AF_NONE;
ch->prev_hop_ = index;          // AODV hack
ih->saddr() = index;
ih->daddr() = IP_BROADCAST;
ih->sport() = RT_PORT;
ih->dport() = RT_PORT;
// Fill up some more fields.
rq->rq_type = AODVTYPE_RREQ;
rq->rq_hop_count = 1;
rq->rq_bcast_id = bid++;
rq->rq_dst = dst;
rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
rq->rq_src = index;
seqno += 2;
assert ((seqno%2) == 0);
rq->rq_src_seqno = seqno;
rq->rq_timestamp = CURRENT_TIME;

// MCMI: Add the fixed channel used by this node to the packet
rq->rq_fixed_channel_used = neighbour_table[(int)index];
rq->rq_sender_node_ip = (int)index;
rq->rq_neighbour_table = &neighbour_table[0];
#ifdef MCMI_DEBUG
  printf("\n%d\tsendRequest - rq_fixed_channel_used: %d\n", (int)index,
             rq->rq_fixed_channel_used);
  printf("\n%d\tsendRequest - rq_sender_node_ip: %d\n", (int)index, rq->rq_sender_node_ip);
#endif

// MCMI: Send a broadcast packet
if (nIfaces) {
  for (int i = 0; i < nIfaces; i++) {
    Packet *p_copy = p->copy();
    Scheduler::instance().schedule(targetlist[i], p_copy, 0.);
    #ifdef MCMI_DEBUG
      printf("\n%d\tsendRequest - broadcast\n\n", (int)index);
    #endif
    Packet::free((Packet *)p);
  }
} else {
  Scheduler::instance().schedule(target_, p, 0.);
}
void AODV::sendReply(nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
    u_int32_t rpseq, u_int32_t lifetime, double timestamp) {
    Packet *p = Packet::alloc();
    struct hdr_cmn *ch = HDR_CMN(p);
    struct hdr_ip *ih = HDR_IP(p);
    struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
    aodv_rt_entry *rt = rtable.rt_lookup(ipdst);
    #ifdef DEBUG
    fprintf(stderr, "sending Reply from %d at %.2f\n", index,
        Scheduler::instance().clock());
    #endif // DEBUG
    assert(rt);
    rp->rp_type = AODVTYPE_RREP;
    //rp->rp_flags = 0x00;
    rp->rp_hop_count = hop_count;
    rp->rp_dst = rpdst;
    rp->rp_dst_seqno = rpseq;
    rp->rp_src = index;
    rp->rp_lifetime = lifetime;
    rp->rp_timestamp = timestamp;
    // ch->uid() = 0;
    ch->ptype() = PT_AODV;
    ch->size() = IP_HDR_LEN + rp->size();
    ch->iface() = -2;
    ch->error() = 0;
    ch->addr_type() = NS_AF_INET;
    ch->next_hop_ = rt->rt_nexthop;
    ch->prev_hop_ = index;      // AODV hack
    ch->direction() = hdr_cmn::DOWN;
    ih->saddr() = index;
    ih->daddr() = ipdst;
    ih->sport() = RT_PORT;
    ih->dport() = RT_PORT;
    ih->ttl_ = NETWORK_DIAMETER;
    // MCMI: Add the fixed channel used by this node to the packet
    rp->rp_fixed_channel_used = neighbour_table[(int)index];
    rp->rp_sender_node_ip = (int)index;
    #ifdef MCMI_DEBUG
    printf("n%d\tsendReply - rp_fixed_channel_used: %d\n", (int)index,
        rp->rp_fixed_channel_used);
    printf("n%d\tsendReply - rp_sender_node_ip: %d\n", (int)index, rp->rp_sender_node_ip);
    #endif
    // MCMI: Send a unicast packet
    if (nIfaces) {
        #ifdef MCMI_DEBUG
        printf("n%d\tsendReply - rt->rt_interface (unicast): %d\n/n", (int)index,
            rt->rt_interface);
        #endif
        Scheduler::instance().schedule(targetlist[rt->rt_interface], p, 0.);
    } else {
        Scheduler::instance().schedule(target_, p, 0.);
    }
}

void AODV::sendError(Packet *p, bool jitter) {
struct hdr_cmn *ch = HDR_CMN(p);
struct hdr_ip *ih = HDR_IP(p);
struct hdr_aodv_error *re = HDR_AODV_ERROR(p);

#ifdef ERROR
    fprintf(stderr, "sending Error from %d at %.2f\n", index,
            Scheduler::instance().clock());
#endif // DEBUG

re->re_type = AODVTYPE_RERR;
//re->Reserved[0] = 0x00; re->reserved[1] = 0x00;
// DestCount and list of unreachable destinations are already filled
// ch->uid() = 0;
ch->ptype() = PT_AODV;
ch->size() = IP_HDR_LEN + re->size();
ch->iface() = -2;
ch->error() = 0;
ch->addr_type() = NS_AF_NONE;
ch->next_hop_ = 0;
ch->prev_hop_ = index;  // AODV hack
ch->direction() = hdr_cmn::DOWN;  // important: change the packet's direction

ih->saddr() = index;
ih->daddr() = IP_BROADCAST;
ih->sport() = RT_PORT;
ih->dport() = RT_PORT;
ih->ttl_ = 1;

// Do we need any jitter? Yes
if (jitter) {
    // MCM: Send a broadcast packet
    if (nIfaces) {
        for (int i = 0; i < nIfaces; i++) {
            Packet *p_copy = p->copy();
            Scheduler::instance().schedule(targetlist[i], p_copy,
                0.01*Random::uniform());
        }
    } #ifdef MCMI_DEBUG
    printf("%d\tsendError - broadcast (jitter)\n\n", (int)index);
    #endif
}
else {
    Scheduler::instance().schedule(target_, p, 0.01*Random::uniform());
}
else {
    // MCM: Send a broadcast packet
    if (nIfaces) {
        for (int i = 0; i < nIfaces; i++) {
            Packet *p_copy = p->copy();
            Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
        }
    } #ifdef MCMI_DEBUG
    printf("%d\tsendError - broadcast (else)\n\n", (int)index);
    #endif
}
else {
    Packet::free((Packet *)p);
}
else {
    Scheduler::instance().schedule(target_, p, 0.0);
}

void
/*
 Neighbor Management Functions
 */
AODV::sendHello() {
   Packet *p = Packet::alloc();
   struct hdr_cmn *ch = HDR_CMN(p);
   struct hdr_ip *ih = HDR_IP(p);
   struct hdr_aodv_reply *rh = HDR_AODV_REPLY(p);

   #ifdef DEBUG
   fprintf(stderr, "sending Hello from %d at %.2f\n", index,
            Scheduler::instance().clock());
   #endif // DEBUG

   rh->rp_type = AODVTYPE_HELLO;
   // rh->rp_flags = 0x00;
   rh->rp_hop_count = 1;
   rh->rp_dst = index;
   rh->rp_dst_seqno = seqno;
   rh->rp_lifetime = (1 + ALLOWED_HELLO_LOSS) * HELLO_INTERVAL;
   // ch->uid() = 0;
   ch->ptype() = PT_AODV;
   ch->size() = IP_HDR_LEN + rh->size();
   ch->iface() = -2;
   ch->error() = 0;
   ch->addr_type() = NS_AF_NONE;
   ch->prev_hop_ = index;      // AODV hack

   ih->saddr() = index;
   ih->daddr() = IP_BROADCAST;
   ih->sport() = RT_PORT;
   ih->dport() = RT_PORT;
   ih->ttl_ = 1;

   // ch->uid() = 0;
   if (nIfaces) {
      // MCMI: Send a broadcast packet
      for (int i = 0; i < nIfaces; i++) {
         Packet *p_copy = p->copy();
         Scheduler::instance().schedule(targetlist[i], p_copy, 0.0);
      }
      Packet::free((Packet *)p);
   } else {
      Scheduler::instance().schedule(target_, p, 0.0);
   }
}

void
AODV::recvHello(Packet *p) {
   //struct hdr_ip *ih = HDR_IP(p);
   struct hdr_aodv_reply *rp = HDR_AODV_REPLY(p);
   AODV_Neighbor *nb;

   /* Whenever a node receives a Hello message from a neighbor, the node
* SHOULD make sure that it has an active route to the neighbor, and
* create one if necessary. If a route already exists, then the
* Lifetime for the route should be increased, if necessary, to be at
* least ALLOWED_HELLO_LOSS * HELLO_INTERVAL.
*/

nb = nb_lookup(rp->rp_dst);
if(nb == 0) {
  nb_insert(rp->rp_dst);
} else {
  nb->nb_expire = CURRENT_TIME + (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
}

if (nIfaces) {
  // MCMI: Update the node's ChannelUsageList using the NeighbourTable of
  // its neighbour. Doing so ensures ChannelUsageList will contain
  // two-hop channel usage information.
  for (int i = 0; i < MAX_NT_CUL_ENTRIES; i++) {
    // MCMI: No need to deal with the fixed channel used by itself again
    if ((rp->rp_neighbour_table[i] == -1) || i == (int)index
        || (neighbour_table[i] == rp->rp_neighbour_table[i])) {
      continue; // avoid repetition
    } else if (neighbour_table[i] != rp->rp_neighbour_table[i]) {
      if (channel_usage_list[neighbour_table[i]] > 0) {
        channel_usage_list[neighbour_table[i]]--;
      } else {
        channel_usage_list[rp->rp_neighbour_table[i]]++;
      }
    } else {
      channel_usage_list[rp->rp_neighbour_table[i]]++;
    }
  }
  // MCMI: When the node receives a "Hello" packet from a neighbour,
  // update its NeighbourTable with the fixed channel of that neighbour.
  neighbour_table[(int)(rp->rp_sender_node_ip)] = rp->rp_fixed_channel_used;
}

#define MCMI_DEBUG

printf("%d\nrecvHello - rp->rp_sender_node_ip: %d\n", (int)index,
rp->rp_sender_node_ip);
printf("%d\nrecvHello - rp->rp_fixed_channel_used: %d\n", (int)index,
rp->rp_fixed_channel_used);
rt_print_NT_CUL();
#endif

} // if (nIfaces)
Packet::free(p);
}
....

/*
# 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
#
# University of Cantabria, Jan. 2007 (User Guide).
#
# Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#
*/

// MCM: 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 *);

    // MCM: 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    /* MCMC: Print the routing table of this node */
74    void            rt_print(nsaddr_t node_id);
75
76    /* MCMC: Print NeighbourTable and ChannelUsageList of this node */
77    void            rt_print_NT_CUL();
78
79    void            local rt_repair(aodv_rt_entry *rt, Packet *p);
80
81    public:
82    void            rt_ll_failed(Packet *p);
83    void            handle_link_failure(nsaddr_t id);
84
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    void            id_insert(nsaddr_t id, u_int32_t bid);
103    bool           id_lookup(nsaddr_t id, u_int32_t bid);
104    void            id_purge(void);
105
106    /*
107     * Packet TX Routines
108     */
109    void            forward(aodv_rt_entry *rt, Packet *p, double delay);
110    void            sendHello(void);
111    void            sendRequest(nsaddr_t dst);
112    void            sendReply(nsaddr_t ipdst, u_int32_t hop_count,
113                                nsaddr_t rpdst, u_int32_t rpeq,
114                                u_int32_t lifetime, double timestamp);
115    void            sendError(Packet *p, bool jitter = true);
116
117    /*
118     * Packet RX Routines
119     */
120    void            recvAODV(Packet *p);
121    void            recvHello(Packet *p);
122    void            recvRequest(Packet *p);
123    void            recvReply(Packet *p);
124    void            recvError(Packet *p);
125
126    /*
127     * History management
128     */
129    double          PerHopTime(aodv_rt_entry *rt);
130
131    nsaddr_t        index;                  // IP Address of this node
132    u_int32_t       seqno;                  // Sequence Number
133    int             bid;                    // Broadcast ID
134    aodv_rtable     rthead;                 // routing table
135    aodv_ncache     nbhead;                 // Neighbor Cache
136    aodv_bcach     bihead;                 // Broadcast ID Cache
/* Timers */
BroadcastTimer btimer;
HelloTimer htimer;
NeighborTimer ntimer;
RouteCacheTimer rtimer;
LocalRepairTimer lrtimer;

/* Routing Table */
aodv_rtable  rtable;
/*
A "drop-front" queue used by the routing layer to buffer
packets to which it does not have a route.
*/
aodv_rqueue  rqueue;

/* A mechanism for logging the contents of the routing
table. */
Trace       *logtarget;

/* A pointer to the network interface queue that sits
between the "classifier" and the "link layer". */
PriQueue    *ifqueue;

// MCMI: The routing agent needs to decide which one of the interfaces
// the outgoing packets should be routed to. Since there are now
// multiple interfaces. The originally used ifqueue and target
// pointers need to be modified as follows:
// MCMI: Store the LL modules for all the interfaces a node has
NsObject    *targetlist[MAX_IF];

// MCMI: Keep the corresponding queues of all the interfaces
PriQueue    *ifqueue list[MAX_IF];

// MCMI: Contain the fixed channels used by the node's neighbours
int neighbour_table[MAX_NT_CUL_ENTRIES]; // index by node IP

// MCMI: Contain the number of nodes using each channel as their fixed channel
int channel_usage_list[MAX_NT_CUL_ENTRIES]; // index by channel ID

// MCMI: Fixed and switchable interfaces used by this node
int fixed_interface;
int switchable_interface;

/* Logging stuff */
void            log_link_del(nsaddr_t dst);
void            log_link_broke(Packet *p);
void            log_link_kept(nsaddr_t dst);
/* for passing packets up to agents */
PortClassifier *dmux_;

};
#endif /* __aodv_h__ */
aodv_packet.h

/*
#===============================================================================
# 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
#
# University of Cantabria, Jan. 2007 (User Guide).
#
# Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
#===============================================================================
*/

...
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

    // MCM
    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
# 2007-2192
# 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
#
# University of Cantabria, Jan. 2007 (User Guide).
#
# Multi-Channel Multi-Interface Ad Hoc Wireless Networks," SIGMOBILE Mobile
*/

...
#define RTF_DOWN 0
#define RTF_UP 1
#define RTF_IN_REPAIR 2

/*
 *  Must receive 4 errors within 3 seconds in order to mark
 *  the route down.
 */
#define MAX_RT_ERROR 4
#define MAX_RT_ERROR_TIME 3

#define MAX_HISTORY 3

aodv_rtable.h
mobilenode.cc

1 /*
2 #===============================================================================
3 # ENSC 835: High-Performance Networks
4 # Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
5 #
6 # Student:     Chih-Hao Howard Chang
7 # 20007-2192
8 # howardc@sfu.ca
9 #
10 # Description: Modified MobileNode class with the implementation of the
11 # multi-interface approach developed by [1]
12 #
13 # File:        mobilenode.cc
14 #
16 #     University of Cantabria, Jan. 2007 (User Guide).
17 #===============================================================================
18 */
19
20 ...
21
22 // MCMI: Retrieve the location of a node
23 void
24 MobileNode::getLoc(double *x, double *y, double *z) {
25     update_position();
26     *x = X_;
27     *y = Y_;
28     *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
#
# 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) {
  //   *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_; }
  void update_position();
  void log_energy(int);
   //void logrttime(double);
virtual void idle_energy_patch(float, float);

/* For list-keeper */
// MobileNode* nextX;
// MobileNode* prevX;
// MCM!: 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:
MobileNode* nextX_[MAX_CHANNELS];
MobileNode* prevX_[MAX_CHANNELS];

protected:
/*
 * Last time the position of this node was updated.
 */
double position_update_time;
double position_update_interval;

/*
 * The following indicate the (x,y,z) position of the node on
 * the "terrain" of the simulation.
 */
double X_;
double Y_;
double Z_;
double speed_; // meters per second

/*
 * The following is a unit vector that specifies the
 * direction of the mobile node. It is used to update
 * position
 */
double dX_;
double dY_;
double dZ_;

/* where are we going? */
double destX_;
double destY_;

/*
 * for gridkeeper use only
 */
MobileNode* next_;
double radius_;  

// Used to generate position updates
PositionHandler pos_handler;
Event pos_intr;

void log_movement();
void random_direction();
void random_speed();
void random_destination();
int set_destination(double x, double y, double speed);

private:
inline int initialized() {
    return (T_ && log_target_ &&
            X_ >= T_->lowerX() && X_ <= T_->upperX() &&
            Y_ >= T_->lowerY() && Y_ <= T_->upperY());
}

void random_position();
void bound_position();
int random_motion_; // is mobile
/*
 * A global list of mobile nodes
 */
LIST_ENTRY(MobileNode) link_; 

/*
 * The topography over which the mobile node moves.
 */
Topography *T_;

/*
 * Trace Target
 */
Trace* log_target_; 

/*
 * base_stn for mobilenodes communicating with wired nodes
 */
int base_stn_; 

//int last_rt_time_; 
}; 

#endif // ns_mobilenode_h 

# endif // ns_mobilenode_h
chiall.cc

/*
 # ENSC 835: High-Performance Networks
 # Implementation of a Multi-Channel Multi-Interface Ad-Hoc Wireless Networks
 #
 # Student:  Chih-Hao Howard Chang
 #              2007-2192
 #              howardc@sfu.ca
 #
 # Description: Modified Channel class with the implementation of the
 #              multi-interface approach developed by [1]
 #
 # File:   channel.cc
 #
 #    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);
            }
        }
    }
    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);
            }
        }
    }
}
break;
}
} else { // use list-based improvement
  MobileNode *mtnode = (MobileNode *) tnode;
  MobileNode **affectedNodes; // **aN;
  int numAffectedNodes = -1, i;
  if(!sorted_)
    sortLists();
  }
  affectedNodes = getAffectedNodes(mtnode, distCST_ + /* safety */ 5,
  &numAffectedNodes);
  for (i=0; i < numAffectedNodes; i++) {
    rnode = affectedNodes[i];
    if(rnode == tnode)
      continue;
    newp = p->copy();
    propdelay = get_pdelay(tnode, rnode);
    rifp = (rnode->ifhead()).lh_first;
    for(; rifp; rifp = rifp->nextnode()){
      // MCMI: Checks which of the interfaces of the destination node
      // is connected to the same channel
      if (rifp->channel() == this) {
        s.schedule(rifp, newp, propdelay);
      }
    }
    delete [] affectedNodes;
  }
  Packet::free(p);
}

WirelessChannel::addNodeToList(MobileNode *mn) {
  MobileNode *tmp;
  // create list of mobilenodes for this channel
  if (xListHead_ == NULL) {
    #ifdef DEBUG
    fprintf(stderr, "INITIALIZE THE LIST xListHead\n");
    #endif
    xListHead_ = mn;
    xListHead_->nextX_[this->index()] = NULL;
    xListHead_->prevX_[this->index()] = NULL;
  } else {
    for (tmp = xListHead_; tmp->nextX_[this->index()] != NULL;
      tmp->nextX_[this->index()] = tmp;
    mn->nextX_[this->index()] = NULL;
  }
  numNodes_++;
}

void WirelessChannel::removeNodeFromList(MobileNode *mn) {
  MobileNode *tmp;
  // Find node in list
  for (tmp = xListHead_; tmp->nextX_[this->index()] != NULL;
    tmp->nextX_[this->index()] = tmp;
  mn->nextX_[this->index()] = NULL;
  if (tmp == mn) {
    xListHead_ = tmp->nextX_[this->index()];
  }
if (tmp->nextX_[this->index()] != NULL)
    tmp->nextX_[this->index()]->prevX_[this->index()] = NULL;
else if (tmp->nextX_[this->index()] == NULL)
    tmp->prevX_[this->index()]->nextX_[this->index()] = NULL;
else {
    tmp->prevX_[this->index()]->nextX_[this->index()] =
        tmp->nextX_[this->index()]->nextX_[this->index()];
    tmp->nextX_[this->index()]->prevX_[this->index()] =
        tmp->prevX_[this->index()];
    numNodes_--;
    return;
}
fprintf(stderr, "Channel: node not found in list\n");
void WirelessChannel::sortLists(void) {
    bool flag = true;
    MobileNode *m, *q;
    sorted_ = true;
    #ifdef DEBUG
    fprintf(stderr, "SORTING LISTS ...");
    #endif
    /* Buble sort algorithm */
    // SORT x-list
    while(flag) {
        m = xListHead_;
        while (m != NULL) {
            if (m->nextX_[this->index()] != NULL)
                if (m->X() > m->nextX_[this->index()]->X()) {
                    flag = true;
                    //delete_after m;
                    q = m->nextX_[this->index()] = m;
                    q->nextX_[this->index()] = q->nextX_[this->index()];
                    if (q->nextX_[this->index()] == NULL)
                        q->nextX_[this->index()]->prevX_[this->index()] = m;
                    ////insert_before m;
                    q->nextX_[this->index()] = m->nextX_[this->index()] =
                        q->nextX_[this->index()]->prevX_[this->index()];
                    //// adjust Head of List
                    if (m == xListHead_)
                        xListHead_ = m->prevX_[this->index()];
                    m = m->nextX_[this->index()];
                }
        }
    }
    #ifdef DEBUG
    fprintf(stderr, "DONE!\n");
    #endif
}
void WirelessChannel::updateNodesList(class MobileNode *mn, double oldX) {
    MobileNode* tmp;
    double X = mn->X();
    bool skipX=false;
    if(!sorted_) {
        sortLists();
        return;
    }
    /* xListHead cannot be NULL here (they are created during creation of mobilnode) */
/*** DELETE ***/
// deleting mn from x-list
if((mn->nextX_[this->index()] != NULL) { 
  if(mn->prevX_[this->index()] != NULL){
    if((mn->nextX_[this->index()] == X) && (mn->prevX_[this->index()] == X)) skipX = true; // the node doesn’t change its position in the list
    else{
      mn->nextX_[this->index()] = prevX_[this->index()];
      mn->prevX_[this->index()] = nextX_[this->index()];
      mn->nextX_[this->index()] = prevX_[this->index()];
    }
  }
  else if(mn->nextX_[this->index()] != NULL){
    if((mn->nextX_[this->index()] == X) >= X) skipX = true; // skip updating the first element
    else{
      if(mn->nextX_[this->index()] == NULL){
        if(mn->prevX_[this->index()] == NULL) skipX = true; //skip updating if only one element in list
      }
    }
  }
else if(mn->prevX_[this->index()] != NULL){
  if((mn->prevX_[this->index()] == X) <= X) skipX = true; // skip updating the last element
  else mn->prevX_[this->index()] = NULL;
}
if ((mn->prevX_[this->index()] == NULL) && (mn->nextX_[this->index()] == NULL)) skipX = true; // skip updating if only one element in list

/*** INSERT ***/
//inserting mn in x-list
if(!skipX){
  if(X > oldX){
    for(tmp = mn; tmp->nextX_[this->index()] != NULL && tmp->nextX_[this->index()] < X; tmp = tmp->nextX_[this->index()]);
    if(tmp->nextX_[this->index()] == NULL) {
      tmp->nextX_[this->index()] = mn;
      mn->prevX_[this->index()] = tmp;
      mn->nextX_[this->index()] = NULL;
    } else{
      / fpsrinf(stdout, "Scanning the element addr %d X=%f, next addr %d X=%f\n",
      tmp->nextX_[this->index()]->X(), tmp->nextX_[this->index()]->X());
      if(tmp->nextX_[this->index()] == NULL) {
        if(tmp->nextX_[this->index()] != NULL) {
          if(tmp->nextX_[this->index()] == X) skipX = true; // skip updating if only one element in list
          else{
            if(tmp->nextX_[this->index()] == NULL){
              if(tmp->nextX_[this->index()] == NULL) skipX = true; //skip updating if only one element in list
            }
          }
        }
      }
    }
  }
else if(tmp->prevX_[this->index()] != NULL){
  if(tmp->prevX_[this->index()] > X) skipX = true; // skip updating the last element
  else{
    if(tmp->prevX_[this->index()] == NULL){
      if(tmp->prevX_[this->index()] == NULL) skipX = true; //skip updating if only one element in list
    }
  }
}
else{
  for(tmp = mn; tmp->prevX_[this->index()] != NULL && tmp->prevX_[this->index()] > X; tmp = tmp->prevX_[this->index()]);
  if(tmp->prevX_[this->index()] == NULL) {
    tmp->prevX_[this->index()] = mn;
    mn->nextX_[this->index()] = tmp;
    mn->prevX_[this->index()] = NULL;
    xListHead_ = mn;
  } else{
    / fpsrinf(stdout, "Scanning the element addr %d X=%f, prev addr %d X=%f\n",
    tmp->prevX_[this->index()]->X(), tmp->prevX_[this->index()]->X());
    if(tmp->prevX_[this->index()] == NULL) {
      if(tmp->prevX_[this->index()] != NULL) {
        if(tmp->prevX_[this->index()] == X) skipX = true; // skip updating if only one element in list
        else{
          if(tmp->prevX_[this->index()] == NULL){
            if(tmp->prevX_[this->index()] == NULL) skipX = true; //skip updating if only one element in list
          }
        }
      }
    }
  }
}
}
WirelessChannel::getAffectedNodes(MobileNode *mn, double radius, int *numAffectedNodes)
{
    double xmin, xmax, ymin, ymax;
    int n = 0;
    MobileNode *tmp, **list, **tmpList;
    if (xListHead_ == NULL) {
        *numAffectedNodes = -1;
        fprintf(stderr, "xListHead_ is NULL when trying to send!!!\n");
        return NULL;
    }
    xmin = mn->X() - radius;
    xmax = mn->X() + radius;
    ymin = mn->Y() - radius;
    ymax = mn->Y() + radius;

    // First allocate as much as possibly needed
    tmpList = new MobileNode*[numNodes_];
    for (tmp = xListHead_; tmp != NULL; tmp = tmp->nextX_[this->index()])
        tmpList[n++] = tmp;
    n = 0;

    for (tmp = mn; tmp != NULL && tmp->X() >= xmin; tmp = tmp->prevX_[this->index()])
        if (tmp->Y() >= ymin && tmp->Y() <= ymax) {
            tmpList[n++] = tmp;
        }
    list = new MobileNode*[n];
    memcpy(list, tmpList, n * sizeof(MobileNode *));
    delete [] tmpList;
    *numAffectedNodes = n;
    return list;
}

/* Only to be used with mobile nodes (WirelessPhy).
* NS-2 at its current state support only a flat (non 3D) movement of nodes,
* so we assume antenna heights do not change for the duration of
* a simulation.
* Another assumption - all nodes have the same wireless interface, so that
* the maximum distance, corresponding to CST (at max transmission power
* level) stays the same for all nodes.
*/

WirelessChannel::calcHighestAntennaZ(Phy *tifp)
{
    double highestZ = 0;
    Phy *n;

    for (n = ifhead_.lh_first; n; n = n->nextchnl())
        if (((WirelessPhy *)n)->getAntennaZ() > highestZ)
            highestZ = ((WirelessPhy *)n)->getAntennaZ();
```cpp
highestAntennaZ_ = highestZ;

WirelessPhy *wifp = (WirelessPhy *)tifp;
distCST_ = wifp->getDist(wifp->getCSThresh(), wifp->getPt(), 1.0, 1.0,
                          highestZ, highestZ, wifp->getL(),
                          wifp->getLambda());
}

double
WirelessChannel::get_pdelay(Node* tnode, Node* rnode)
{
    // Scheduler &s = Scheduler::instance();
    MobileNode* tmnode = (MobileNode*)tnode;
    MobileNode* rmnode = (MobileNode*)rnode;
    double propdelay = 0;

    propdelay = tmnode->propdelay(rmnode);

    assert(propdelay >= 0.0);
    if (propdelay == 0.0) {
        /* if the propdelay is 0 b/c two nodes are on top of
         * each other, move them slightly apart -dam 7/28/98 */
        propdelay = 2 * DBL_EPSILON;
        //printf("propdelay 0: %d->%d at %f\n",
                tmnode->address(), rmnode->address(), s.clock());
    }

    return propdelay;
}
```
/*
# 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
#===============================================================================
*/
...

/*
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_; }
    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:
...
void mac_log(Packet *p) {
    logtarget_->recv(p, (Handler*) 0);
    NsObject* logtarget_;
}

protected:
int command(int argc, const char* const* argv);
virtual int initialized() {
    return (netif_ && uptarget_ && downtarget_);
}
int index_; // MAC address
double bandwidth_; // channel bitrate
double delay_; // MAC overhead
int abstract_; // MAC support for abstract LAN
Phy *netif_; // network interface
Tap *tap_; // tap agent
LL *ll_; // LL this MAC is connected to
Channel *channel_; // channel this MAC is connected to
Handler* callback_; // callback for end-of-transmission
MacHandlerResume hRes_; // resume handler
MacHandlerSend hSend_; // handle delay send due to busy channel
Event intr_;

/*
 * Internal MAC State
*/
MacState state_; // MAC's current state
Packet *pktRx_;
Packet *pktTx_;
/*
 * 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
 * 
 * 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.
     */
if(pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= p->txinfo_.CPThresh) {
  capture(p);
} else {
  collision(p);
}
...
# 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
#
# 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 [eval $self create-dsdv-agent $node]
        }
        DSR {
            $self at 0.0 "$node start-dsr"
        }
        AODV {
            set ragent [eval $self create-aodv-agent $node]
        }
        TORA {
            Simulator set IMEPFlag_ ON
            set ragent [eval $self create-tora-agent $node]
        }
        DIFFUSION/RATE {
            eval $node addr $args
            set ragent [eval $self create-diffusion-rate-agent $node]
        }
        DIFFUSION/PROB {
            eval $node addr $args
            set ragent [eval $self create-diffusion-probability-agent $node]
        }
        Directed_Diffusion {
            eval $node addr $args
            set ragent [eval $self create-core-diffusion-rtg-agent $node]
        }
        FLOODING {
            eval $node addr $args
            set ragent [eval $self create-flooding-agent $node]
        }
        OMNIMCAST {
            eval $node addr $args
            set ragent [eval $self create-omnimcast-agent $node]
        }
        DumbAgent {
            set ragent [eval $self create-dumb-agent $node]
        }
    }
}
ManualRtg {
    set ragent [$self create-manual-rtg-agent $node]
}

# Manual Routing
MANUAL {
    set ragent [$self create-manual-routing-agent $node]
}

default {
    eval $node addr $args
    puts "Wrong node routing agent!"
    exit
}

}

# errProc_ and FECProc_ are an option unlike other
# parameters for node interface
if ![info exists inerrProc_] {
    set inerrProc_ ""
}
if ![info exists outerrProc_] {
    set outerrProc_ ""
}
if ![info exists FECProc_] {
    set FECProc_ ""
}

# MCMI: Add main node interface
if { [info exists numifs_] } {
    for {set i 0} {$i < $numifs_} {incr i} {
        # Add one interface per channel
        #puts "chan $i: chan($i)"
        $node add-interface $chan($i) $propInstance_ $llType_ $macType_ \$ifqType_ $ifqlen_ $phyType_ $antType_ $topoInstance_ \$inerrProc_ $outerrProc_ $FECProc_ 
    }
} else {
    $node add-interface $chan $propInstance_ $llType_ $macType_ \$ifqType_ $ifqlen_ $phyType_ $antType_ $topoInstance_ \$inerrProc_ $outerrProc_ $FECProc_ 
}

# Attach agent
if { [$routingAgent_ != "DSR"] } {
    $node attach $ragent [Node set rtagent_port_]
}
if { [$routingAgent_ == "DIFFUSION/RATE" ] || $routingAgent_ == "DIFFUSION/PROB" ] || [$routingAgent_ == "FLOODING" ] || [$routingAgent_ == "OMNICAST" ] || [$routingAgent_ == "Directed_Diffusion" ] } {
    $ragent port-dmux [$node demux]
    $node instvar ll
    $ragent add-l1 $ll(_0)
}
if { [$routingAgent_ == "DumbAgent" ] } {
    $ragent port-dmux [$node demux]
}

# Bind routing agent and mip agent if existing basestation
# address setting
if { [info exists wiredRouting_] && $wiredRouting_ == "ON" } {
    if { [$routingAgent_ != "DSR"] } {
        $node mip-call $ragent
    }
}

# This Trace Target is used to log changes in direction
# and velocity for the mobile node.
set tracefd [$self get-ns-traceall]
if { [tracefd != ""] } {
    $node nodetrace $tracefd
    $node agenttrace $tracefd
}
set namtracefd [$self get-nam-traceall]
if {$namtracefd != ""} {
    $node namattatch $namtracefd
}

if [info exists energyModel_] {
    if [info exists level1_] {
        set 11 $level1_
    } else {
        set 11 0.5
    }
    if [info exists level2_] {
        set 12 $level2_
    } else {
        set 12 0.2
    }
    $node addenergymodel [new $energyModel_ $node \ $initialEnergy_ $11 $12]
}

if [info exists txPower_] {
    $node setPt $txPower_
}

if [info exists rxPower_] {
    $node setPr $rxPower_
}

if [info exists idlePower_] {
    $node setPidle $idlePower_
}

# if [info exists sleepPower_] {
#    $node setPsleep $sleepPower_
#}

if [info exists sleepTime_] {
    $node setTsleep $sleepTime_
}

if [info exists transitionPower_] {
    $node setPtransition $transitionPower_
}

if [info exists transitionTime_] {
    $node setTtransition $transitionTime_
}

# $node topography $topoInstance_
return $node

...
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: Modified ns-mobilenode TCL library script with the implementation
# of the multi-interface approach developed by [1]
#
# File: ns-lib.tcl
#
# 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
}
}
}

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_()] ;# ifq between LL and MAC
# XXX: The routing protocol and the IMEP agents needs handles
# to each other.
$agent imep-agent [$self set imep_()]
($self set imep_()) rtagent $agent
}

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

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

...
if { $port == [Node set rtagent_port_] } {
    # MCMI: Special processing when multiple interfaces are supported
    if {$numIfsSimulator != ""} {
        for {set i 0} {$i < [self set nifs_]} {incr i} {
            $agent if-queue $i [self set ifq_($i)]
        }
    }
    # Ad hoc routing agent setup needs special handling
    $self add-target-rtagent $agent $port
    return
}

# Attaching a normal agent
set namfp [ns get-nam-traceall]
if { [Simulator set AgentTrace_] == "ON" } {
    # Send Target
    if { $newapi != "" } {
        set sndT [self mobility-trace Send "AGT"]
    } else {
        set sndT [cmu-trace Send AGT $self]
    }
    if { $namfp != "" } {
        $sndT namattach $namfp
    }
    $sndT target [$self entry]
    $agent target $sndT
    # Recv Target
    if { $newapi != "" } {
        set rcvT [self mobility-trace Recv "AGT"]
    } else {
        set rcvT [cmu-trace Recv AGT $self]
    }
    if { $namfp != "" } {
        $rcvT namattach $namfp
    }
    $rcvT target $agent
    $dmux_ install $port $rcvT
} else {
    # Send Target
    # $agent target [$self entry]
    # Recv Target
    # $dmux_ install $port $agent
}

Node/MobileNode instproc add-target-rtagent { agent port } {
    $self instvar imep_ toraDebug_
    set ns [Simulator instance]
    set newapi [$ns imep-support]
    set namfp [$ns get-nam-traceall]
    set dmux_ [$self demux]
    set classifier_ [$self entry]
    # MCMI: Whether multiple interfaces exist in the simulation
    set numIfsSimulator [$ns get-numifs]
    # let the routing agent know about the port dmux
    $agent port-dmux $dmux_
    if { [Simulator set RouterTrace_] == "ON" } {
        #
# Send Target

if {$newapi != ""} {
set sndT [$self mobility-trace Send "RTR"]
} else {
set sndT [cmu-trace Send "RTR" $self]
}

if { $namfp != "" } {
$sndT namattach $namfp
}

if { $newapi == "ON" } {
$agent target $imep_(0)
$imep_(0) sendtarget $sndT

# second tracer to see the actual types of tora packets before imep packs them
if { ![info exists toraDebug_] && $toraDebug_ == "ON"} {
    set sndT2 [$self mobility-trace Send "TRP"]
    $sndT2 target $imep_(0)
    $agent target $sndT2
}

# MCMI
$sndT target [$self set ll_(0)]
} else {  ;#  no IMEP
    # MCMI: 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 the receiving target.
if {$numIfsSimulator != ""} {
    for {set i 0} {$i < [$self set nifs_]} {incr i} {
        set sndT [cmu-trace Send "RTR" $self]
        $agent target $i $sndT
        $sndT target [$self set ll_($i)]
    }
} else {
    $agent target $sndT
    $sndT target [$self set ll_(0)]
}
}

# Recv Target

if {$newapi != ""} {
set rcvT [$self mobility-trace Recv "RTR"]
} else {
set rcvT [cmu-trace Recv "RTR" $self]
}

if { $namfp != "" } {
$rcvT namattach $namfp
}

if { $newapi == "ON" } {
    [info exists toraDebug_] && $toraDebug_ == "ON"
    $classifer_ defaulttarget $agent
    $rcvT target $agent
} else {
    $rcvT target $rcvT
    $classifier_ defaulttarget $rcvT
    $dmux_ install $port $rcvT
}

} else {
    # Send Target
    # if tora is used
if { $newapi == "ON" } {
    $agent target $imep_(0)
    # second tracer to see the actual types of tora packets before imep packs them
    if { ![info exists toraDebug_] && $toraDebug_ == "ON"} {
}
set sndT2 [\$self mobility-trace Send "TRP"]
$sndT2 target $imep_(0)
$agent target $sndT2

$imep_(0) sendtarget [\$self set ll_(0)]

} else { ;# no IMEP

  ;# MCMI: 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 the sending target.
  if {$numIfsSimulator != ""} {
    for {set i 0} {$i < [\$self set nifs_]} {incr i} {
      $agent target $i [\$self set ll_($i)]
    }
  } else {
    $agent target [\$self set ll_(0)]
  }
}

# Recv Target
#
#
if {$newapi == "ON" } {
  [\$self set ll_(0)] up-target $imep_(0)
$classifier_ defaulttarget $agent
  ;# need a second tracer to see the actual
  ;# types of tora packets after imep unpacks them
  ;# no need to support any hier node
  if {[info exists toraDebug_] && $toraDebug_ == "ON" } {
    set rcvT2 [\$self mobility-trace Recv "TRP"]
$rcvT2 target $agent
  [\$self set classifier_] defaulttarget $rcvT2
  }
} else {
  $classifier_ defaulttarget $agent
$dmux_ install $port $agent
}

# The following setups up link layer, mac layer, network interface
# and physical layer structures for the mobile node.
#
node/MobileNode instproc add-interface { channel pmodel lltype mactype qtype qlen iftype
  anttype topo inerrproc outerrproc fecproc } {
  \$self instvar arptable_ nifs_ netif_ mac_ ifq_ ll_ imep_ inerr_ outerr_ fec_

  set ns [Simulator instance]
  set imepflag [\$ns imep-support]
  set t $nifs_
  incr nifs_

  set netif_(\$t) [new $iftype] ;# interface
  set mac_(\$t) [new $mactype] ;# mac layer
  set ifq_(\$t) [new $qtype] ;# interface queue
  set ll_($t) [new $lltype] ;# link layer
  set ant_($t) [new $anttype]

  set namfp [\$ns get-nam-traceall]
  if {$imepflag == "ON" } {

    $ns mac-type $mactype
    set inerr_(\$t) ""
    if {$inerrproc != ""} {
      set inerr_(\$t) [\$inerrproc]
    }
    set outerr_(\$t) ""
    if {$outerrproc != ""} {
      set outerr_(\$t) [\$outerrproc]
    }
    set fec_($t) ""
    if {$fecproc != ""} {
      set fec_($t) [\$fecproc]
    }
    set namfp [$ns get-nam-traceall]
    if {$imepflag == "ON" } {
# IMEP layer
set imep_($t) [new Agent/IMEP [$self id]]
set imep $imep_($t)
set drpT [Self mobility-trace Drop "RTR"]
if { $namfp != "" } {
    $drpT namattach $namfp
}
$imep drop-target $drpT
$ns at 0.[$self id] "$imep_($t) start" ;# start beacon timer

# Local Variables
set nullAgent_ [$ns set nullAgent_]
set netif $netif_($t)
set mac $mac_($t)
set ifq $ifq_($t)
set ll $ll_($t)
set inerr $inerr_($t)
set outerr $outerr_($t)
set fec $fec_($t)

set inerr $inerr_($t)
set outerr $outerr_($t)
set fec $fec_($t)

# MCMII: Create one ARP table per interface. Originally, it creates one ARP
# table (for address resolution) per node. 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.
set arptable_($t) [new ARPTable $self $mac]
set arptable $arptable_($t)

# FOR backward compatibility sake, hack only
if {$imepflag != ""} {
    set drpT [Self mobility-trace Drop "IFQ"]
} else {
    set drpT [cmu-trace Drop "IFQ" $self]
}
$arptable drop-target $drpT
if { $namfp != "" } {
    $drpT namattach $namfp
}

# Link Layer
$ll arptable $arptable; # MCMII
$ll mac $mac
$ll down-target $ifq
if {$imepflag == "ON" } {
    $imep recvtarget [$self entry]
    $imep sendtarget $ll
    $ll up-target $imep
} else {
    $ll up-target [$self entry]
}

# Interface Queue
# $ifq target $mac
$ifq set limit_ $qlen
if {$imepflag != ""} {
    set drpT [Self mobility-trace Drop "IFQ"]
} else {
    set drpT [cmu-trace Drop "IFQ" $self]
}
$ifq drop-target $drpT
if { $namfp != "" } {
    $drpT namattach $namfp
}
if {[$ifq info class] == "Queue/XCP"} {
    $mac set bandwidth_ [$ll set bandwidth_]
    $mac set delay_ [$ll set delay_]
```tcl
$ifq set-link-capacity [$mac set bandwidth_]
$ifq queue-limit $qlen
$ifq link $ll
$ifq reset

}

# Mac Layer

$mac netif $netif
$mac up-target $ll

if {$outerr == "" && $fec == ""} {
$mac down-target $netif
} elseif {$outerr != "" && $fec == ""} {
$mac down-target $outerr
$outerr target $netif
} elseif {$outerr == "" && $fec != ""} {
$fec down-target $netif
} else {
$fec down-target $outerr
$err target $netif
}

set god_ [God instance]
if {$mactype == "Mac/802_11"} {
	$mac nodes [$god_ num_nodes]
}

# Network Interface

# if {$fec == ""} {
# $netif up-target $mac
#} else {
# $netif up-target $fec
# $fec up-target $mac
#}

$netif channel $channel
if {$inerr == "" && $fec == ""} {
$netif up-target $mac
} elseif {$inerr != "" && $fec == ""} {
$netif up-target $inerr
$inerr target $mac
} elseif {$err == "" && $fec != ""} {
$fec up-target $netif
} else {
$fec up-target $inerr
$inerr target $fec
$fec up-target $mac
}

$netif propagation $pmodel ;# Propagation Model
$netif node $self ;# Bind node <-> interface
$netif antenna $ant_($t)

# Physical Channel

# $channel addif $netif

# List-based improvement
# For nodes talking to multiple channels this should
# be called multiple times for each channel
$channel add-node $self

# let topo keep handle of channel
$topo channel $channel

if { [Simulator set MacTrace_] == "ON" } {
	#
```
# Trace RTS/CTS/ACK Packets

# if {$imepflag != ""} {
#     set rcvT [self mobility-trace Recv "MAC"]
# } else {
#     set rcvT [cmu-trace Recv "MAC" $self]
# }
$mac log-target $rcvT
if { $namfp != "" } {
$rcvT namattach $namfp
}

# Trace Sent Packets

# if {$imepflag != ""} {
#     set sndT [self mobility-trace Send "MAC"]
# } else {
#     set sndT [cmu-trace Send "MAC" $self]
# }
$sndT target [$mac down-target]
$mac down-target $sndT
if { $namfp != "" } {
$sndT namattach $namfp
}

# Trace Received Packets

# if {$imepflag != ""} {
#     set rcvT [self mobility-trace Recv "MAC"]
# } else {
#     set rcvT [cmu-trace Recv "MAC" $self]
# }
$rcvT target [$mac up-target]
$mac up-target $rcvT
if { $namfp != "" } {
$rcvT namattach $namfp
}

# Trace Dropped Packets

# if {$imepflag != ""} {
#     set drpT [self mobility-trace Drop "MAC"]
# } else {
#     set drpT [cmu-trace Drop "MAC" $self]
# }
$mac drop-target $drpT
if { $namfp != "" } {
$drpT namattach $namfp
}

} else {
$mac log-target [$ns set nullAgent_]
$mac drop-target [$ns set nullAgent_]
}

# change wrt Mike's code

if { [$Simulator set EotTrace_] == "ON" } {
    # Also trace end of transmission time for packets
    # if {$imepflag != ""} {
    #     set eotT [self mobility-trace EOT "MAC"]
    # } else {
    #     set eotT [cmu-trace EOT "MAC" $self]
    # }
    $mac eot-target $eotT
    
    # ==============================================================
    $self addif $netif
    ...
}